NRS Test Plan for Lustre version 2.4

Version	Date	Comment	Author
1.0	01/22/2013	Initial revision	Nikitas Angelinas <nikitas_angelinas@xyratex.com>
1.1	01/25/2013	Updated to mirror lprocfs interface changes	Nikitas Angelinas <nikitas_angelinas@xyratex.com>
1.2	01/29/2013	Updated to mirror a minor lprocfs interface change (both regular and hp requests are handled by default)	Nikitas Angelinas <nikitas_angelinas@xyratex.com>

Table of Contents

Terminology

Acronym/Term	Meaning	Description
NRS	Network Request Scheduler	A new PTLRPC layer subcomponent
FIFO	First-In, First-Out	The default NRS policy; it is a logical wrapper around current, non-NRS enabled Lustre code, and so schedules RPCs in exactly the same manner, i.e. in fifo order as they arrive from LNet
CRR-N	Client Round Robin over NIDs	An NRS policy that schedules RPCs in a Round Robin manner over client NIDs
ORR	Object-Based Round Robin	An NRS policy that schedules RPCs in a Round Robin manner over backend-fs objects
TRR	Target-based Round Robin	An NRS policy that schedules RPCs in a Round Robin manner over OSTs
vanilla code	Mainline Lustre code	The non-NRS Lustre source tree
SSF	Single-Shared-File	I/O access type pattern where more than one process/thread is making I/O requests to the same file
FPP	File-Per-Process	I/O access type pattern where more than one process/thread are each making I/O requests to a separate file

Foreword

The purpose of this series of tests is to measure the data and metadata performance of an NRS-enabled build, compared to a vanilla, non-NRS enabled build. It is obviously desirable that the two source tree snapshots differ only in the application of the NRS patches, such that the possibility of any other differences in the trees affecting the

results obtained during the tests is eliminated. The following tests make use of different NRS policies, potentially incorporating a number of runs for each test case, using varying values for the policies' parameters.

Software requirements

The client nodes used for posing an I/O or metadata load on the cluster should be equipped with IOR and mdtest, and satisfy any other requirements that exist for the operation of the aforementioned tools, such as OpenMPI availability and password-less ssh or rsh, etc. It may be useful to be able to shorten any effort that may be required for installing either the vanilla or the NRS-enabled build, as there may be a need to switch back and forth between these at some point, e.g. to re-do a series of tests using different parameters etc, so e.g. generating rpms for the builds may be useful.

It is not considered strictly necessary to make use of NRS-enabled builds on the Lustre client nodes, as **NRS code does not alter client code at all**, although this may be desirable anyway. It may also be best to install the NRS-enabled build on both the MDS and OSS nodes, irrespective on whether one is carrying out a predominantly data or metatada throughput test.

Hardware requirements

Ideally, these tests would be carried out in an HPC site-level cluster, or be substituted with runs of HPC jobs, however if this is not possible in a timely manner, a small test cluster, consisting of a single MDS, and a pair of failover OSS nodes from any vendor should suffice for this series of tests (although a larger test-cluster could also be used).

The important requirement that the test setup should fulfil, is that the set of client nodes used for the tests should be capable of saturating the cluster, and allow for any NRS policies that may have the potential to yield improved results, to do so during the tests. A good strategy may be to over-compensate. e.g. if a pair of particular OSS nodes is known to regularly yield its highest I/O throughput values using a given set of 16 client nodes, perhaps add 8 more (or more) clients to make sure the tests will be able to saturate the system in any case.

Some of the tests that involve the use of policies that may be able to yield increased performance in some cases, could benefit from having a larger number of clients (perhaps 16+, and ideally 32+) for generating the I/O load. If this is not possible, such an environment may be able to be emulated by e.g. increasing the max_rpcs_in_flight parameter value.

It may be best if the test runs take place on the same (rather than just on an identically spec'd) test cluster, if possible.

Tunables affecting performance

It may be best if a set of values that are known to yield good performance results is used for some test environment parameters. These include setting the max_rpcs_in_flight tunable to a favorable value, disabling checksumming on all clients, setting lod.*.qos_threshold_rr=100, and adjusting disk scheduler tunables to known good values for particular hardware, along with perhaps other settings. The important goal again is to carry out tests with consistent settings for such values, but also ones that enable the system to perform at its best.

Some tests that examine the potential of some NRS policies to yield increased performance values may benefit from e.g. increasing the max_rpcs_in_flight and associated values. This is noted on each of the tests where this applies.

Test cases

1. FIFO performance regression tests

The FIFO policy serves RPCs in **exactly** the same manner as current vanilla, non-NRS Lustre code, i.e. it will serve RPCs in the same order as they arrive on a PTLRPC service from LNet. These two tests aim to measure whether the use of the NRS core framework, along with the default, FIFO NRS policy, have introduced any unwanted performance regressions in data and metadata throughput.

a. Data throughput regression test

Using the NRS FIFO policy (i.e. there is no need to perform any setup or configuration operations, just load the Lustre modules and bring up the Lustre targets on the test cluster equipped with the NRS build, and that will then be using the FIFO policy on all PTLRPC services), measure the system's data throughput performance using IOR, and compare that to the performance of the same system on identical IOR runs when equipped with the vanilla, non-NRS build.

At least two separate IOR runs can be carried out for this test, one testing FPP, and one testing SSF performance. The exact command-line to be used can be altered as one sees fit, however the following parameters are suggested as a good starting point:

IOR FPP write test:

pdsh -w <client hostnames> “sync; echo 3 > /proc/sys/vm/drop_caches”; mpirun -hosfile <hosts file> -np <# of processes> IOR -a POSIX -F -i 1 -t 1M -b 16G -w -e -k -C -g -o /mnt/lustre/IOR.test

Please note the “-i 1” parameter, this will cause IOR to execute just one iteration. Ideally, the test can be carried out for a number of distinct invocations (that are later averaged out manually), and by clearing kernel caches on all clients as shown in the example command line above, in order to exclude any caching effects from affecting the obtained results. This is ultimately at the discretion of the person(s) carrying out the tests, and they can opt for using a single invocation which will carry all the iterations one after the other, however it has been the author's experience that IOR results may be subject to caching effects if this method is used.

IOR FPP read test:

pdsh -w <client hostnames> “sync; echo 3 > /proc/sys/vm/drop_caches”; mpirun -hosfile <hosts file> -np <# of processes> IOR -a POSIX -F -i 1 -t 1M -b 16G -r -e -k -C -g -o /mnt/lustre/IOR.test

Here we are reading the file written in the previous test, again making sure to clear kernel caches before starting the test.

IOR SSF write test:

pdsh -w <client hostnames> “sync; echo 3 > /proc/sys/vm/drop_caches”; mpirun -hosfile <hosts file> -np <# of processes> IOR -a POSIX -i 1 -t 1M -b 16G -w -e -k -C -g -o /mnt/lustre/stripped_dir/IOR.test

Here we are performing an SSF write test; as the target directory name implies, this test is probably better performed on a stripped directory, i.e. using:

lfs setstripe -c -1 <dir_name>

IOR SSF read test:

pdsh -w <client hostnames> “sync; echo 3 > /proc/sys/vm/drop_caches”; mpirun -hosfile <hosts file> -np <# of processes> IOR -a POSIX -i 1 -t 1M -b 16G -r -e -k -C -g -o /mnt/lustre/stripped_dir/IOR.test

Relevant test parameters:

It may be best to keep the “-np” parameter value of mpirun at a number equal to the number of physical clients (assuming one mount point per client; if this is not the case, please multiply by the number of mount points per client). The FIFO policy has no tunables that affect its performance, however one could experiment with obtaining results for different transfer and block sizes (“-t” and “-b” parameters to IOR).

b. Metadata throughput regression test

Using the NRS FIFO policy, measure the system's metadata throughput performance using mdtest, and compare that to the performance of the same system on identical mdtest runs when equipped with the vanilla, non-NRS build.

mdtest run:

The exact mdtest parameters to be used are at the discretion of the person(s) that will carry out the tests, however a good starting point may be to use something similar to:

mdtest -u -d /mnt/lustre/mdtest{1-128} -n 32768 -i 3

Relevant test parameters:

As mentioned in the previous paragraph, all parameters to mdtest could be altered, as long as the requirement for having the MDS saturated during the tests runs is satisfied. The “-i” parameter to mdtest should probably be kept to '3' at a minimum in order to obtain trustworthy results.

2. Performance testing using other policies

It is uncertain at this point whether the remaining NRS policies besides FIFO (i.e. CRR-N, ORR, and TRR) are capable of yielding increased performance in some usage scenarios, and which usage scenarios these may be, so the ideal proving ground for these policies may be their use under RPC loads imposed in real-life scenarios, when handling data generated by different cluster jobs.

It may be possible to obatin an indication of the performance of these policies using a simple test cluster, although it would be much preferential if they could be deployed and tested at an HPC site. The two cases are first treated separately below, however the guidelines on how to enable and alter the operating parameters of each of the policies apply equally to both cases, unless otherwise stated.

a. Using a test cluster

In the case where a test cluster is to be used for tesing these policies, then performing the same tests as in sections 1a and 1b of this document should suffice.

IOR test runs of section 1a should be repeated using all policies, CRR-N, ORR and TRR, and the results compared with the vanilla baseline results. mdtest test runs of section 1b should also be repeated using all policies, although CRR-N may be the most important one to use in this case, as ORR and TRR are only used on OSS nodes, and may only influence metadata throughput results if the metadata operations also involve the sending of RPCs to the OSS nodes (e.g. delete operations).

For information on how to enable and tune these policies, please see section 2c of the present document below.

b. Testing in an HPC site

In the case of being able to test the performance of these policies in an HPC site, any regular workload that ideally has been known to yield specific performance metrics could be used. When testing the CRR-N policy, we would be interested to see particularly whether job completion times are shortened, while when using the ORR and TRR policies we would be predominantly interested in witnessing performance gains in data throughput numbers, assuming there is a method to capture these.

c. Using NRS policies

i. Enabling/Disabling policies

This section applies to both of the cases in sections 2a and 2b above, unless otherwise stated. Its purpose is to guide the person(s) carrying out the tests in enabling/disabling and altering tunable parameters of the CRR-N, ORR and TRR policies (the FIFO policy has no tunable parameters).

FIFO policy instances are **always enabled** irrespective of whether any of these other policies (CRR-N, ORR, and TRR) is also enabled, however enabling any of these policies (only one can be enabled at any given time) allows them to take priority in handling incoming RPCs at PTLRPC services, with the aim of hopefully yielding increased performance in some cases, or performing other functions. RPCs that these policies fail to handle for any reason, are routed to the respective FIFO policy instances.

There is one instance of each policy on each PTLRPC service for regular priority RPCs, and optionally (if the PTLRPC service can handle some RPC types as high priority ones) one for high-priority RPCs[1]. In order to enable both the regular and high priority instances of a policy on a particular PTLRPC service, one can issue the following command:

lctl set_param x.x.x.nrs_policies=<policy name>

For example:

to enable the CRR-N policy on all PTLRPC services for both regular and high priority requests, please issue:

*lctl set_param *.*.*.nrs_policies=crrn*

(this should be performed on a per-server basis, so would need to be performed on both the MDS, and each of the OSS nodes),

to enable the ORR policy on the ost_io service for both regular and high priority requests, please issue:

lctl set_param ost.OSS.ost_io.nrs_policies=orr

In order to enable either only the regular or only the high-priority instance of a policy on an a PTLRPC service, one can supply an optional “reg” or “hp” token to lctl set_param commands.

For example:

*lctl set_param *.*.*.nrs_policies=”crrn reg”* would enable the CRR-N policy for only regular priority requests on all PTLRPC services,

and

1 Actually, there is one policy instance per PTLRPC service partition, not per PTLRPC service, but this is hidden from the user in their dealings with lprocfs, in order to present a simpler interface.

lctl set_param ost.OSS.ost_io.nrs_policies="trr hp"

would enable the TRR policy only for high-priority requests on the ost_io PTLRPC service.

It is unclear at this point whether testing for the any of the policies should involve enabling the policies only on regular or also on high-priority requests, so there may be a need to carry out test runs using both of these settings, however it may be preferential to first attempt to perform the tests using the default action of enabling a policy on both regular and high priority requests.

Retrieving the state of policy instances on a PTLRPC service is performed by reading the contents of the nrs_policies file for the respective PTLRPC service, by using:

lctl get_param x.x.x.nrs_policies

For example:

to read the state of policy instances on all PTLRPC services, please issue:

*lctl get_param *.*.*.nrs_policies*

to read the state of policy instances on the ost_io service, please issue:

lctl get_param ost.OSS.ost_io.nrs_policies

Disabling a policy occurs either automatically, when another policy is enabled (not immediately, but once requests for the old policy are all served), or is disabled when the name "fifo" is used as the policy name in an attempt to set the policy.

For example:

*lctl set_param *.*.*.nrs_policies=fifo*

will disable any policies other than FIFO for both regular and high-priority requests on all PTLRPC services, thus resetting the NRS framework to its default state.

ii. Altering policy tunable parameters

Each policy instance on each PTLRPC service, maintains its own set of tunable parameters. Altering and obtaining the values of policy tunable parameters also involves interaction with lprocfs, in but in a slightly different manner to the one shown for enabling and disabling policies. One can set either the regular and/or high-priority policy instance tunable value individually if the value name is specified, or both together in a single invocation if only the tunable value is given.

For example:

to set the CRR-N Round Robin quantum value (this parameter instructs CRR-N of the maximum number of requests originating from the same client NID that it can batch-schedule at a time) for regular requests only, on all PTLRPC services to 32, please issue:

*lctl set_param *.*.*.nrs_crrn_quantum="regular_quantum_size: 32"*

to set the types of supported requests for the TRR policy instance on the ost_io service for high-priority requests only, to OST_READ and OST_WRITE RPCs, please issue:

lctl set_param ost.OSS.ost_io.nrs_trr_supported="high_priority_supported_requests: reads_and_writes"

to set the ORR quantum for both regular and high-priority RPCs for the ost_io service to 128, please issue:

lctl set_param ost.OSS.ost_io.nrs_orr_quantum=128

All other tunable parameters of all policies can be set in the same manner; their names and possible and default values are shown in the tables in sections 2[iv-vi] of this document, below.

Reading parameter values happens in the same way as for policies,

For example:

*lctl get_param *.*.*.nrs_crrn_quantum*

will obtain CRR-N quantum values for CRR-N policy instances for both regular and high-priority RPCs for all PTLRPC services.

In cases where a policy is disabled, reading and writing of these tunables is not allowed, and their values will reset each time a policy instance is stopped and restarted.

iii. Service partition numbers

An important parameter in the performance of these policies, is expected to be the number of CPU partitions used in each PTLRPC service, at least for ORR and TRR policies. Since these policies aim to batch-schedule RPCs in order to improve performance, and there is in reality (although this is hidden from the user) one policy instance per service partition, it may be preferential to keep the number of service partitions on OSS nodes low, perhaps best at 1, and at a maximum of 2. Having a larger number of service partitions may cause these policies to underperform, as different instances of a policy will all be competing for service from the storage array (or other storage element), but the offset-ordered RPC streams they each produce end up mixing up with each other, thus causing non-ideal RPC streams to reach the disks.

Hopefully it will be possible to use a small number of service partitions (maybe even 1) on OSS nodes without loosing much or any performance at all, as large SMP scaling improvements have been reported thus far (to the author's knowledge) on MDS nodes. In any case, this parameter may be one that will need to be varied and used for repeated test runs of the same test cases. The easiest way to alter it is by changing the libcfs module parameter “cpu_npartitions” in a modprobe configuration file.

For example:

edit /etc/modprobe.d/lustre.conf to contain

options libcfs cpu_npartitions=1

at Lustre module load time, in order to start a given node with a single service partition.

iv. CRR-N policy

The CRR-N policy can be enabled on all types of PTLRPC services. It may be useful to carry out different performance test runs using different values for its tunable parameter. Imposing a high number of RPCs (in the region of thousands pending at any given point) may be beneficial to this policy's performance.

policy tunables:

Name	Description	Valid values	Default value
nrs_crrn_quantum	The Round Robin quantum size, i.e. the maximum number of RPCs from the same NID that a CRR-N policy instance can batch-schedule at a time.	1-65535	8

v. ORR policy

The ORR policy can be enabled only on ost_io PTLRPC services on OSS nodes. It may be useful to carry out different performance test runs using different values for all of its tunable parameters.

The policy should also benefit from being loaded by a high number of RPCs (in the region of thousands pending at any given point), so if the tests are executed on a small test cluster (e.g 16 clients), it may be best to increase the max_rpcs_in_flight value, maybe even to its maximum allowed setting of 256 for OSCs in client nodes and 512 for OSCs in MDS nodes, along with making any associated changes to tunables that may help increase the RPC count arriving at OSS nodes.

policy tunables:

Name	Description	Valid values	Default value
nrs_orr_quantum	The Round Robin quantum size, i.e. the maximum number of RPCs that target the same backend-filesystem object, that an ORR policy instance can batch-schedule at a time.	1-65535	256
nrs_orr_offset_type	Whether an ORR policy orders RPCs within each batch based on ascending logical file offsets, or physical disk offsets. Irrespective of the value of this tunable, only logical offsets can, and are used for ordering OST_WRITE RPCs.	physical, logical	physical
nrs_orr_supported	Whether an ORR policy instance will reorder OST_READ RPCs, OST_WRITE RPCs, or both type of RPCs.	reads, writes, reads_and_ writes	reads

vi. TRR policy

The TRR policy can be enabled only on ost_io PTLRPC services on OSS nodes. It may be useful to carry out different performance test runs using different values for all of its tunable parameters.

The policy should also benefit from being loaded by a high number of RPCs (in the region of thousands pending at any given point), so if the tests are executed on a small test cluster (e.g 16 clients), it may be best to increase the max_rpcs_in_flight value, maybe even to its maximum allowed setting of 256 for OSCs in client nodes and 512 for OSCs in MDS nodes, along with making any associated changes to tunables that may help increase the RPC count arriving at OSS nodes.

policy tunables:

Name	Description	Valid values	Default value
nrs_trr_quantum	The Round Robin quantum size, i.e. the maximum number of RPCs that target the same OST, that a TRR policy instance can batch-schedule at a time.	1-65535	256
nrs_trr_offset_type	Whether a TRR policy orders RPCs within each batch based on ascending logical file offsets, or physical disk offsets. Irrespective of the value of this tunable, only logical offsets can, and are used for ordering OST_WRITE RPCs.	physical, logical	physical
nrs_trr_supported	Whether a TRR policy instance will reorder OST_READ RPCs, OST_WRITE RPCs, or both type of RPCs.	reads, writes, reads_and_ writes	reads

d. Additional test cases

It may be possible that either of the CRR-N, ORR or TRR policies may prove to be beneficial in some some use cases, or may hinder performance in other cases. Moreover,

there may be use cases for these policies which have not been foreseen at this moment, so the person(s) carrying out the tests are encouraged to be inventive in generating additional test cases.

For example, it was found in earlier testing performed using the ORR and TRR policies, that by using either the ORR or TRR policy (better results where obtained using ORR) in order to reorder the initial write stream of RPCs (i.e. for this, set nrs_orr_supported=writes, and perhaps keep nrs_orr_quantum=256), it was then possible to obtain improved read performance on the written data using even the FIFO policy. In other words, it may be possible to obtain increased read performance, by ordering the RPCs on the initial write operation. For some unknown reason, the write operation's performance took a large hit when compared to using just the FIFO policy for the write operation. In any case, this test case definitely deserves to be carried out if that is possible; a pair of IOR FPP and SSF runs as performed in sections 1a and 1b should suffice for such a test.

The person(s) carrying out these tests may also find it useful to employ other tools in either regression testing or performance testing; an example of this could be iozone operated in “cluster” mode.

Contact details

For any required clarification, observed omissions, or anything else related to this document and the NRS feature, please contact the author of this document:
Nikitas Angelinas <nikitas_angelinas@xyratex.com> , and also the co-author of the NRS feature, Liang Zhen <liang@whamcloud.com> .