Lustre On Demand
Evolution of Data Tiering on Storage System

Shuichi Ihara (sihara@whamcloud.com)
Rahul Deshmukh (rdeshmukh@whamcloud.com)
Agenda

► Introduction
► LOD (Lustre On Demand) as Tiering solution
  • Overview
  • Use case
► Implementation and usage
► Current Status and Future work
► Conclusions
Evolution of data tiering on storage system

► First Generation – primary filesystem to archive
  • E.g. HDD based filesystem to tape or even on cloud
  • HSM provides managing of data residence and transparent data access

► Second Generation – another tier in front of primary filesystem
  • Burst Buffer
  • Local filesystem or other filesystem on flash device

► Simplification, Automation and Transparency are important
  • Users complain about complexity, but administrator and users wants IO acceleration on flash devices
  • /tmp or /dev/shm are easy to use for users, but user need to manage data residence by themselves

Introduce LOD (Lustre On Demand), a tiering approach for Lustre, provides temporary filesystem per job and automated synchronization with primary filesystem job scheduler.
Various Lustre caching options

- **Hardware cache**
  - Memory, SSD/NVMe, write back cache on storage array, cache on drives
  - Lustre relies on hardware capability transparently

- **Ladvise (Lustre fadvise)**
  - Giving hints to files and prefetch data on OSS memory or SSD with DSS
  - This is similar idea of fadvise(), but through Lustre client and Lustre server side caching

- **PCC (Persistent Client Cache)**
  - Leverages local SSD/NVMe on client and keep single namespace
  - Support write/read caches with HSM (and group lock) features for consistency

- **Lustre Write Back Cache**
  - Data and metadata into RAM on client as cache
  - Avoid of LDLM and network latency
LOD (Lustre On Demand)

- Provide Dynamic Lustre filesystem on compute nodes using local SSD/NVMe
  - Temporary fast Lustre filesystem across the compute nodes
  - LOD creates Lustre on computes nodes dynamically

- Integration with job scheduler
  - User turn LOD on/off per job at job submission
  - Currently integrated into SLURM’s Burst Buffer option, but other job scheduler also could work

- Transparent and automated stage-in/out
  - User can define file/directory list on stage-in/out to LOD at Job submission
  - LOD automatically sync/migrate data from persistent Lustre to created temporary Lustre filesystem

- Lots of flexibility and extendibility
  - Configurable MDT/OST configuration for advanced users
LOD architecture and design

- **Compute Nodes (Lustre Client)**
  - LOD-A for Job A
  - LOD-B for Job B

- **Triggers Creating LOD**

- **Automated Stage-in/out**
  - UserA
  - UserB

- **Persistent Lustre Filesystem**
Use cases(1): Accelerated random I/O operations

Avoiding Random I/O.
Prefetching/syncing files with Sequential I/O

Small Random I/O to LOD
Use cases (2): Reduce loads on PFS

Lustre clients mount both LOD and Persistent Lustre

Temporary Clustered filesystem for Job

Reading files from Persistent Lustre directory

Avoid unnecessary I/O to Persistent Lustre Filesystem
Use cases(3): Isolated namespace per JOB or clients

Client can have multiple LOD mount points

Allocate SSD devices and create LOD per JOB

Stage-in/out policy can be per LOD

Persistent Lustre Filesystem
LOD is a framework that runs on top of job scheduler
  • Job scheduler have storage extension/plugin to manage storage resources on compute nodes
  • Prolog/epilog also works, but storage extensions/plugin can allow tighter integration and flexibility

Selected SLURM for demonstration of LOD
  • Open source and one of the major job scheduler for HPC
  • “Burst Buffer” plugin is available
  • LOD framework can be integrated on BB plugin

No more patched kernel for Lustre server
  • Thanks for patchless server support
SLURM : Introduction

► What is SLURM ? (Simple Linux Utility for Resource Management)
  • Allocates resource (compute nodes) for user to execute the job
    o Exclusive and/or non-exclusive access
    o For some duration of time
  • Provides facility to start, execute and monitor jobs on allocated nodes
  • Manages queue of pending jobs in case of contention for resource

► How to use it ?
  • Provides sets of command to start, execute and monitor jobs
    e.g. sbatch, salloc, sinfo, etc.
  • Mainly jobs are run in
    o Batch mode (e.g. sbatch test.sh)
    o Interactive mode (e.g. salloc command)

test.sh
#l/bin/bash
#SBATCH -p debug
#SBATCH -N 1
#SBATCH -t 00:05:00
srun a.out
SLURM : Burst Buffer (BB) Infrastructure

- SLURM Burst Buffer is a pluggable architecture which facilitates usage of high-speed storage resource

- Burst Buffer infrastructure has following workflow:
  - Allocates burst buffer resource
  - Staging in required file(s) into it
  - Schedule compute node(s) for the job execution using these resources
  - Stage out file(s) if needed after completion of job

- Currently there are two plugins available:
  - Generic: Not implemented
  - Cray
BB plugin integration details

Important steps to add new plugin into BB infrastructure are:

► Unique name assigned to plugin and same will be used in slurm.conf
  • E.g. for generic plugin it is like:
    ```c
    const char plugin_type[] = "burst_buffer/generic";
    ```
  • And mention into configuration as:
    ```
    BurstBufferType=burst_buffer/generic
    ```

► Workflow API to be implemented

► Current LOD implementation using generic plugin

```c
bb_p_get_system_size
bb_p_load_state
bb_p_get_status
bb_p_state_pack
bb_p_reconfig
bb_p_job_validate
bb_p_job_validate2
bb_p_job_set_tres_cnt
bb_p_job_get_est_start
bb_p_job_try_stage_in
bb_p_job_test_stage_in
bb_p_job_begin
bb_p_job_revoke_alloc
bb_p_job_start_stage_out
bb_p_job_test_post_run
bb_p_job_test_stage_out
bb_p_job_cancel
bb_p_xlate_bb_2_tres_str
```
Job execution workflow

- LOD Framework and SLURM integration
  - LOD Framework perform underlying management functions e.g. creation of Lustre on demand based on user input
  - Used SLURM generic BB plugin skeleton and integrated LOD framework into it for ease of use
  - How this works
    - User will submit the job, e.g. Job A
    - It will allocate compute node, e.g. C2
    - The files which required for job will be stage_in first
    - Once files stage_in then it will start execution of job
    - After completion of job, only required file(s) to be stage_out (sync) to main PFS
How to use LOD : Example

# Running it in batch mode using slurm command “sbatch <script_file>”

$ sbatch example_job.sh

```bash
#!/bin/bash
#LOD setup_lod type=scratch capacity=10GB
#LOD stage_in source=/mnt/pfs/large_file destination=$LOD_MNT/ type=file
#LOD stage_out source=$LOD_MNT/output destination=/mnt/pfs/sample_task/ type=file
# srun sample_task.sh
```
# LOD : setup_lod

► Information:
This will setup the LOD instance as per job requirement through SLURM

► Parameters

- type = scratch or persistent
- capacity = <number>[MB|GB|TB|PB]
- lod_config = <path to config file> This is optional parameter.
#LOD : setup_lod

► Information :
This will setup the LOD instance as per job requirement through SLURM

► Parameters

- type = scratch or persistent
- capacity = <number>[MB|GB|TB|PB]
- lod_config = <path to config file> This is optional parameter.
#LOD : stage_in

**Information:**
This will fetch file(s)/directory(ies) needed for the job before it starts through SLURM

**Parameters:**
- source : path of source file/directory
- destination : path of destination file/directory
- type : file or directory

```bash
#!/bin/bash
#LOD setup_lod type=scratch capacity=10GB
#LOD stage_in source=/mnt/pfs/large_file destination=$LOD_MNT/ type=file
#LOD stage_out source=$LOD_MNT/output destination=/mnt/pfs/sample_task/ type=file
srun sample_task.sh
```
#LOD : stage_out

► Information :
This will sync file(s)/directory(ies) needed for the job after completion of job

► Parameters :
• source : path of source file/directory
• destination : path of destination file/directory
• type : file or directory

```bash
#!/bin/bash
#LOD setup_lod type=scratch capacity=10GB
#LOD stage_in source=/mnt/pfs/large_file destination=$LOD_MNT/ type=file
#LOD stage_out source=$LOD_MNT/output destination=/mnt/pfs/sample_task/ type=file
srun sample_task.sh
```
srun sample_task.sh

- Information:
  - Ready to execute the job as the required instance is ready with input files needed.
  - After completion of the job all the files except files mention under stage_out will be removed

```bash
#!/bin/bash
#LOD setup_lod type=scratch capacity=10GB
#LOD stage_in source=/mnt/pfs/large_file destination=$LOD_MNT/ type=file
#LOD stage_out source=$LOD_MNT/output destination=/mnt/pfs/sample_task/ type=file
srun sample_task.sh
```
Current Status and future Plan

► Current status:
  • LOD framework: Basic infrastructure ready with limited testing
  • Integration with SLURM: Used generic BB plugin for integration

► stage_in/stage_out:
  Automate data movement in Lustre server side use case where files are transparently move into and from flash storage tier, using FLR as per job requirement. e.g.
  #LOD stage_in source=hdd:/mnt/pfs/large_file destination=flash: type=flr_file
  #LOD stage_out source=flash:large_file destination=hdd type=flr_file

► To improve on LOD creation part by providing profile based creation
  e.g. data intensive, metadata intensive, balanced, default

► Options to tune LOD as per job requirement for small files, I/O size, I/O patterns
Conclusions

► Introduced Lustre On Demand as a new Tiering option on Lustre
  This not only allows new use cases on Lustre and but also accelerate I/O Performance

► Lustre On Demand is a framework and integrated as an extension of the Burst Buffer plugin on the SLURM job scheduler as a prototype implementation

► Will continue to extend LOD framework and look at integration with another job scheduler
Thank you!