

|  |  |
| --- | --- |
| **Date:**  **17 March 2016** | **M3.1 Legion on the Exascale FastForward I/O Stack**  **Extreme Scale Storage and I/O RND** |

**Government Purpose Rights**

Prime Contract No.: DE-AC52-07NA27344

LLNL Subcontract No.: B613306

Subcontractor Name: Intel Federal LLC, on behalf of itself, its parent and AffiliatesSubcontractor Address: 4100 Monument Corner Dr, Ste 540, Fairfax, VA 22030

The Government’s rights to use, modify, reproduce, release, perform, display, or disclose this technical data are restricted by the above agreement.

**Limited Rights**

Prime Contract No.: DE-AC52-07NA27344

LLNL Subcontract No.: B613306

Subcontractor Name: Intel Federal LLC, on behalf of itself, its parent and AffiliatesSubcontractor Address: 4100 Monument Corner Dr, Ste 540, Fairfax, VA 22030

The Government’s rights to use, modify, reproduce, release, perform, display, or disclose this technical data are restricted by the above agreement.

**NOTICES**

**Acknowledgment:** This material is based upon work supported by Lawrence Livermore National Laboratory subcontract B613306.

**USG Disclaimer:** This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

**Intel Disclaimer:** Intel makes available this document and the information contained herein in furtherance of extreme-scale storage and I/O research and development. None of the information contained therein is, or should be construed, as advice. While Intel makes every effort to present accurate and reliable information, Intel does not guarantee the accuracy, completeness, efficacy, or timeliness of such information. Use of such information is voluntary, and reliance on it should only be undertaken after an independent review by qualified experts.

Access to this document is with the understanding that Intel is not engaged in rendering advice or other professional services. Information in this document may be changed or updated without notice by Intel.

This document contains copyright information, the terms of which must be observed and followed.

Reference herein to any specific commercial product, process or service does not constitute or imply endorsement, recommendation, or favoring by Intel or the US Government.

Intel makes no representations whatsoever about this document or the information contained herein.

IN NO EVENT SHALL INTEL BE LIABLE TO ANY PARTY FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES FOR ANY USE OF THIS DOCUMENT, INCLUDING, WITHOUT LIMITATION, ANY LOST PROFITS, BUSINESS INTERRUPTION, OR OTHERWISE, EVEN IF INTEL IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Legion on the Exascale FastForward I/O Stack

# Introduction

Legion is a high-level data-centric and task-based application runtime which is intended for use with HPC and distributed systems. HDF5 is a scientific data format and library used to store data in an organized, self-describing, and high-performance manner. Legion is currently capable of attaching to HDF5 files using the low-level *Realm* runtime on which Legion is built. Realm maps HDF5 files (or groups) to memory objects within the runtime using a DMA mechanism to collect updates to individual datasets in an HDF5 file.

The FastForward I/O stack is a high performance parallel file system meant to enable apps to scale to next generation systems while providing an interface for versioning and snapshotting. The interface layer for the current version of FastForward is called IOD, while the interface for the version in development is called DAOS-M.

HDF5 has been implemented on top of IOD, with many extensions to the HDF5 interface to expose transactions, read contexts, and other new FastForward I/O features not covered in this document. We plan to implement HDF5 on top of DAOS-M when it is available. This document describes attempts to port Legion to HDF5 on top of IOD, and discusses plans for implementation on HDF5/DAOS-M.

# Virtual Datasets

Virtual dataset support is a new feature that was recently added to HDF5. A virtual dataset (VDS) does not store any data directly, rather, it stores data in a set of source datasets. Source datasets are created like any other source dataset, and the virtual dataset is created with a set of mappings between regions in the virtual dataset and regions in the source dataset. I/O requests to the VDS are then translated into a set of I/O requests to the source datasets whose mappings overlap with the requested region in the VDS.

Virtual datasets map closely to Legion's data model, so we decided to add VDS support to Legion as part of this work. This meant we needed to add support for VDS to the IOD implementation of HDF5. This ended up taking a significant amount of work, and we had to impose the limitation that source datasets must reside in the same container as the virtual dataset, due to restrictions with IOD that are described below.

# Details on FastForward I/O

In IOD and DAOS-M, the analog for the traditional file is called a “container”, which consists of a collection of key-value store and byte array objects. In IOD, all processes are required to open a container collectively, whereas in DAOS-M, this requirement will be relaxed to allow containers to be opened independently. When an application wants to write to a container, it must first start a transaction (through the HDF5 API). To do this, the application supplies the version number for the transaction it is starting. This version number is global and must be consistent across processes, so the application must coordinate distribution of version numbers to compute ranks, or ensure that processes share a transaction. When finished writing, the application commits the transaction. When an application wants to read data from a container, it similarly provides a version number and requests a read context on the container. This version number uses the same sequence of values as those for transactions, and so committing transactions determines which versions are visible for a read context. Any committed transactions with a version lower or equal to the version of the read context will be visible, while transactions above that value aren't. Finally, IOD does not allow reading from a read context when any lower transactions have not been committed. DAOS-M, however, will not have this limitation.

# Implementation on IOD

## General Approach

In order to port Legion onto the FastForward stack, we decided to extend the existing Legion+HDF5 implementation to handle both normal HDF5 and FastForward HDF5. The primary interface for this is a new function, *attach\_hdf5\_ff*, which mirrors the existing *attach\_hdf5* function, but instead instructs Legion to use the FastForward interface and thereby create the file as an IOD container. Subsequent requests to the HDF5 layer check if it is a FastForward file, and uses the new HDF5/FastForward API calls if so.

In the initial implementation, all processes opened the same transaction and read context, with the transaction version number being one higher than the read context version. In order to be able to read back data that was written, all processes were required to collectively call another function, *commit\_update\_hdf5\_ff*, which committed the transaction, opened it as a read context, and started a new transaction with a version number one greater than the previous.

## Resolved Issues

One issue that came up during testing involved the threadsafety of HDF5/FastForward when compiled with the threadsafe option. Threadsafety in HDF5 is implemented with a simple global lock protecting the entire library. Since HDF5/FastForward can be run with both the client and server process in a single process (co-resident mode), and the server can make HDF5 calls when the client needs to wait for a response, it must drop the threadsafe lock to allow the server to proceed and avoid deadlock. This causes concurrency issues when multiple client threads are making HDF5 calls, as other threads can enter the library while one is waiting, and the HDF5 library is not designed to allow internal concurrency. It is therefore necessary to implement a secondary lock at the Legion level to prevent concurrent access to HDF5. While Legion actually already attempted to do this for the standard HDF5 implementation, it uses read-write locks and only requests a read lock for some operations, allowing multiple “read” operations into the HDF5 library at the same time, which is not supported.

## Unresolved Issues

The original implementation with IOD used a global transaction model and required a collective management of container versions at the application level to increment the version numbers so data written could be read back. This caused problems with the Legion port because we could not find an easy way to have Legion execute the call to increment the versions exactly once in each process.

The next implementation relied on assigning each task a unique index within the index space, using this index to generate a unique transaction version for each task. It was assumed that each task would make the same number of write calls. When reading, the task would acquire a read context for the highest version number that had been committed, as calculated using the number of tasks and the highest task index. Unfortunately, we could not find a way to generate these task indices.

The final implementation relies on the fact that each subregion in Legion is uniquely assigned to a single process, with all I/O for that subregion going through that process. It partitioned the version number space, with each process keeping an independent stream of version numbers within the partition so the versions did not need to be coordinated between processes. It was believed that IOD could read from any version that had been committed, but it turned out that IOD and the older version of DAOS require that all previous versions be committed. Consequently, this implementation could not work since processes with rank greater than 0 would never be able to acquire read contexts unless the other processes used their entire version number partition.

Another issue involves the implementation of VDS on HDF5/FastForward. IOD's requirement that all container opens be collective causes problems when attempting to access a source dataset located in another IOD container. The client would have to either broadcast a message to all other clients to participate in the collective open, or all clients would have to always keep all of the containers for source datasets open. Either option would hurt performance significantly. Since the container open limitation is planned to be addressed with DAOS-M, we decided on, for now, requiring source datasets to reside in the same container as the virtual dataset. This requires “fooling” Legion into thinking it's opening a file when it's really opening a group, and resulted in making adjustments to the file set-up routines in the Legion examples. We will remove these workarounds in the next version.

# Plan with the upcoming DAOS-M

The new DAOS stack should improve our ability to implement Legion on top of the FastForward stack due to several planned features. We will first quickly present the DAOS storage and transaction model and then study how Legion could support this new storage paradigm.

## DAOS Storage Model



The figure above represents the fundamental abstractions of the DAOS storage model. A pool can host multiple transactional object stores called DAOS containers. Each container is a private object address space, which can be atomically modified and snapshotted independently of the other containers sharing the same pool. DAOS objects in a container are identified by a unique object address and are effectively key-value stores with two-level key (distribution and attribute keys) and supporting partial or atomic values.

A pool is only accessible to authenticated and authorized applications. Security is enforced when connecting to the pool. The pool stores a persistent list of containers which includes the container UUIDs and the name associated with each container. Upon successful connection to the pool, a pool handle is granted to the job. There is no restriction on the number of jobs that can connect to the pool. The workflow scheduler is responsible for reporting job failures to DAOS. DAOS will then revoke the pool handle granted to the failed job and reclaim all resources (e.g. container handles) associated with this pool handle.

## DAOS Transaction Model

A container can be opened for write concurrently by multiple processes. On successful open, a container handle is granted to the opener which can then share this handle with peer processes of the same job. All processes submitting I/O operations with the same container handle effectively participate in the same transaction scope. This means that all updates submitted with the same container handle are committed or rolled back as one.

A transaction state is associated with a container handle. It is composed of the following variables:

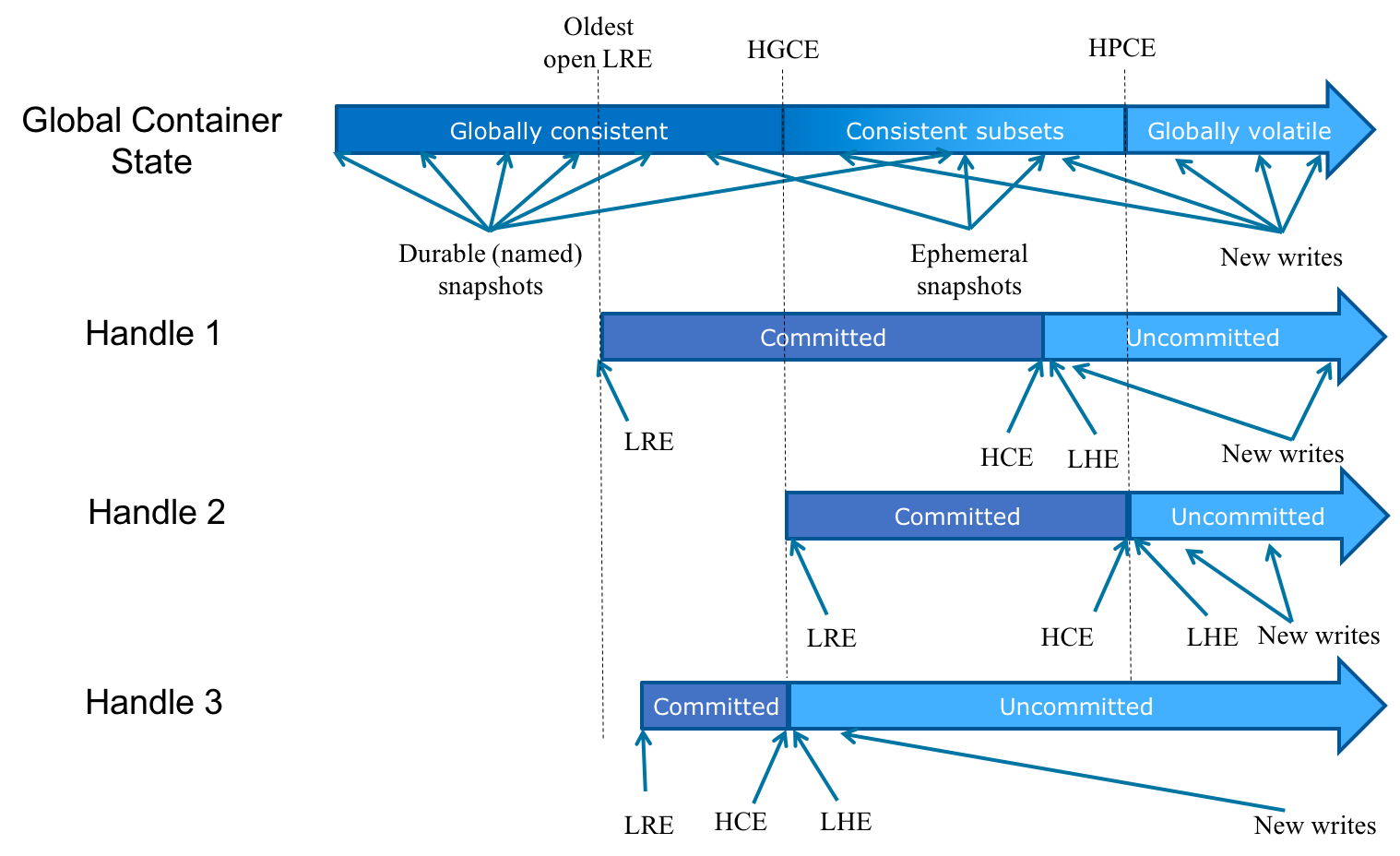
* *LRE*: lowest epoch referenced by this handle. Any epoch >= LRE can be read. The LRE is moved forward with the slip operation.
* *LHE*: lowest epoch held by this handle. The handle is expected to submit I/O operations for and to commit all epochs >= LHE. The LHE is increased automatically on commit to HCE+1 and can be obtained/released at any time with the hold operation.
* *HCE*: high epoch committed by this handle. Any changes submitted by this open handle with an epoch <= HCE are guaranteed to be durable. On the other hand, any updates submitted with an epoch > HCE are automatically rolled back on failure of the container handle. The HCE is increased on successful commit.

Furthermore, the following global container states may be queried at any time:

* HPCE: the highest partially committed epoch, equal to max({HCE}h)
* HGCE: the highest globally committed epoch, equal to min({min({LHE}h) - 1, HPCE). This epoch is guaranteed to have immutable data.

The commit, slip and hold epoch operations can be performed by any processes sharing the handle, but are not collective operations. In other words, a single member is expected to execute the operation on behalf of the whole group. Epoch commit makes all updates submitted by the handle, in all epochs up to the commit epoch, durable. Epoch hold is guaranteed to return a LHE strictly higher than the HPCE.

The figure below represents the container with three active handles.

****

## Legion Use Case

Let’s now see how Legion could leverage the DAOS epoch model. In this proposal, we are making two assumptions:

* The Legion scheduler always serializes the execution of conflicting tasks (i.e. overlapping updates). In other words, no overwrite can happen from two tasks running at the same time.
* The commit operation in DAOS has a very low latency and is several orders of magnitude shorter than the latency of the flush operation. This will be measured and verify with the DAOS prototype.

When the legion framework starts, it should connect to the DAOS pool and distribute the pool handle to all the Legion top-level tasks. The pool handle must then be passed to any new sub-tasks since it is required to open a container.

Each legion task can then individually open the container. This operation grants a private open handle to the task and also returns the current value of HPCE. A read-only task could then read with an epoch number equal to the HPCE and eventually refresh its knowledge of the HPCE though a DAOS query whenever required (e.g. notification from other processes).

To modify the container, a task must obtain an epoch hold. This operation returns a LHE which is guaranteed to be higher than the current HPCE (typically HPCE+1). The task can then read from the LHE and submit new updates against this epoch or the next ones. When the task is done, it should flush and commit its updates and then close its container handle. This guarantees that any newly spawn tasks can see the changes made by all tasks that have already completed. Indeed, any new container handle should return an HPCE greater than or equal to the HCE of the already completed tasks.

When two tasks want to communicate through the storage system, the producer task must first flush and commit its changes. Once notified, the consumer can query the current HPCE (or the legion framework can pass the epoch number from the writer to the reader) and read from this epoch. If the consumer wants to submit new changes resulting from the data generated by the producer, it must then use an epoch number higher than the HPCE.

If one task fails, the Legion scheduler should close the open handle on behalf of the failed task. DAOS will roll-back any uncommitted updates submitted by this task and will report completion of the close operation. Once the close has completed, the Legion scheduler can restart the failed task.

If the whole Legion job fails, the workflow scheduler informs DAOS and revokes the pool handle. This automatically closes all container handles allocated to the Legion tasks and rolls back any uncommitted updates.

# In Conclusion

Implementation of Legion on top of IOD proved to be difficult due primarily to a conflict between Legion’s highly decentralized task execution model and IOD’s strict global version number scheme. We could not find a reasonable way to either generate a globally consistent set of version numbers for each task, or have all tasks coordinate operations on a sequence of shared version numbers. We have outlined enhancements to DAOS-M that should allow us to implement Legion on top of the Exascale FastForward I/O stack by adding flexibility in transaction number sequencing and allowing independent container opening. These improvements should also aid implementation of other highly decoupled parallel applications on top of FastForward.