OPENFABRICS ALLIANCE

Workload driven MR registration

ANNUAL INTERNATIONAL OPENFABRICS SOFTWARE DEVELOPERS' WORKSHOP

Parav Pandit

Emulex Corporation

Overview

- Block storage access using RDMA is increasingly becoming popular in data centers using iSER, SRP and Ceph.
- NFS, SMB Direct and other distributed file systems based access equally deployed in similar environments.
- Distributed storage is equally getting attention where compute and storage node being in single system.
- To get reasonable IO performance using RDMA transport, its necessary to have certain number of RDMA Memory regions per RDMA QP.
- New nodes in a cluster gets added for compute or storage needs. This increases number of connections and so it demands higher number of MRs as connections grow.

Resource consumption per session

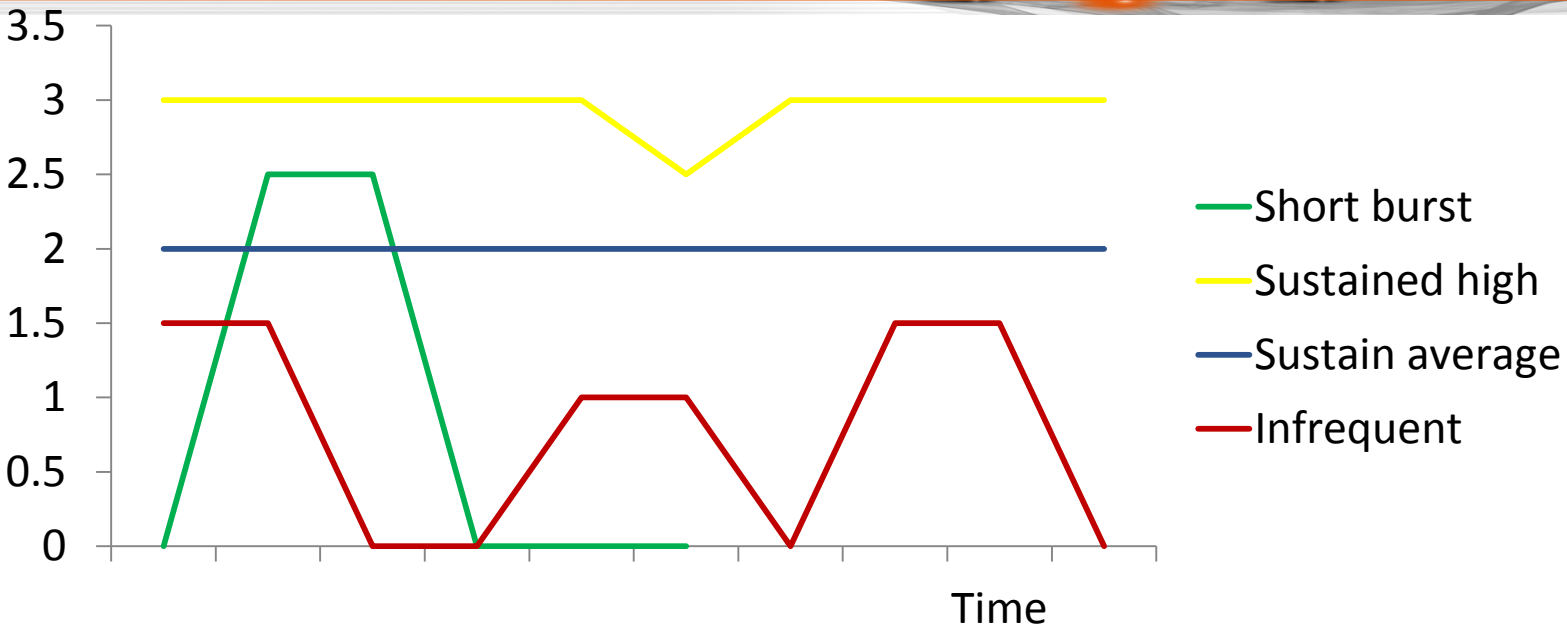
RDMA Protocol	Number of FRMR regions per connection	Default number of QPs per session
SRP initiator	128	1 to more
iSER initiator	113	1
NFS-RDMA client	2178	1

- As the storage become more distributed, with current ULPs number of MRs grows linearly with every connection, while system IOPs continues to remain same reaching the HCA's link speed limit or Message limit.

Requirements

- Number of connections should scale close to number of QPs instead of having static limit based on number of MRs per session.
- Should able to cater to dynamic workload pattern of hosted VMs to bare metal HA nodes, bare metal systems using Linux containers.
- Should be able to continue to interact to new nodes being added - hosting storage data.
- Block or file system IO data path should avoid stall or task switching at IO transport layer to avoid number of switches between transport layer and block storage layer.
- Memory registration and deregistration usually involves HCA. It should be able to overcome this latency which is relatively higher compared to other RDMA operations.

Dynamic IO workload



- Sudden burst of IOs at start or at later stage
- HA storage pair backend performing IOs to keep two storage nodes in sync
- Sustained IOs for fair amount of time between few client server pair
- VMs consolidated from single server performing relatively lower average IOs per VM in hosted environment.
  - Connections idle from few milliseconds to seconds between client and server

Current solutions

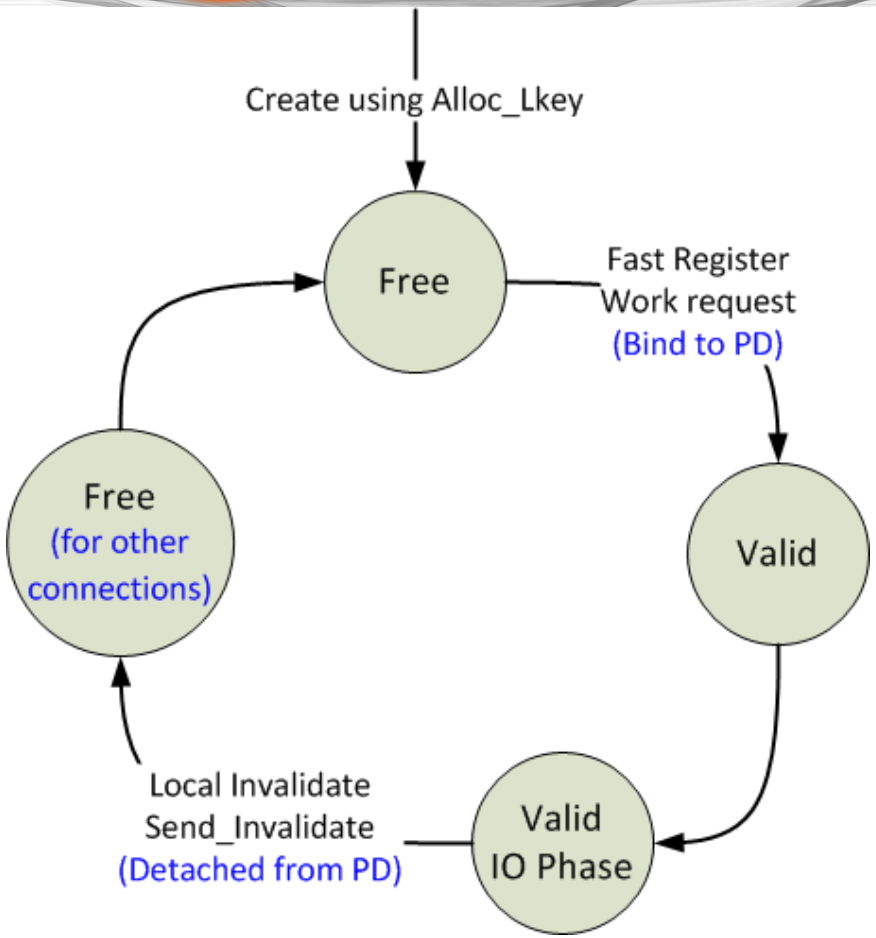
- Statically allocate constant number of MRs during connection setup time. Allocate sufficient amount to meet the IOPs.
  - This continue to keep wider disparity between number of connections and MRs
- Allocate small number of MRs say A at start time, continue to allocate more based on workload via slow allocation process and deploy them via FRMR.
  - Burst IO beyond A number of IOs, will have to stall until MR is allocated from slow/potentially blocking path
  - Need to release those unused MRs to deploy for other connections on different/same PD.

Current solutions

- Number of MRs allocated is not function of logical or physical link speed.
- Number of MRs allocated is not function of end storage type having varied IO response time (SSD, rotating disks, hybrid, RAM backed)
- Fails the connection on failure to allocate sufficient MRs which might be available at later stage

Extending the solution – Mobile MR

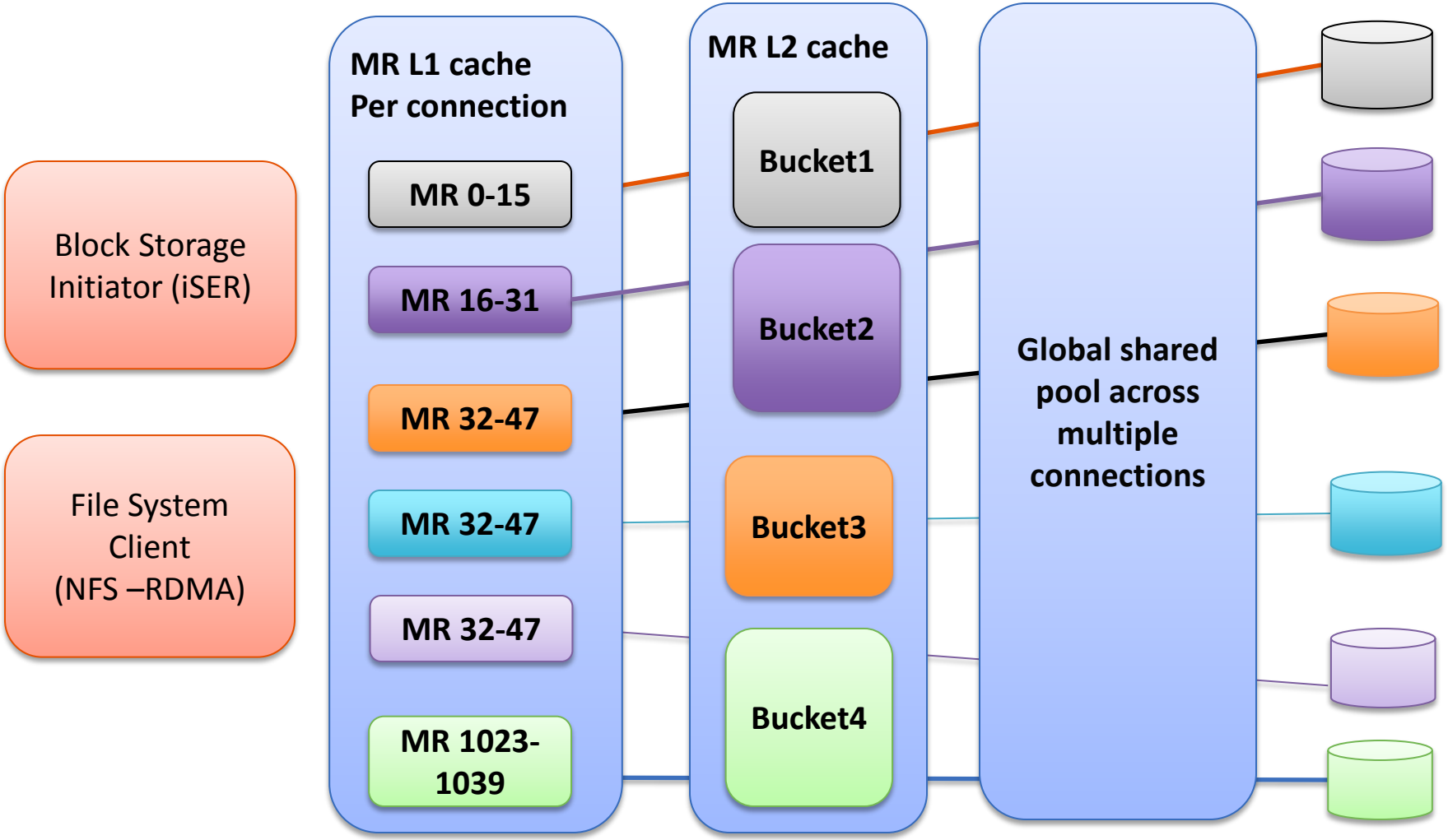
- Allow RDMA ULP applications to register MRs per adapter instead of per PD
- Attach MR to QP (and so to PD) during FRMR Work request processing time
- Local invalidate or send with invalidate operations detach the MR from the PD



Extending the solution – Mobile MR

- Trusted kernel space driver ensures that one ULP doesn't register memory region of other ULP.
- Works from every VM guest OS kernel as well in SR-IOV.
- PD cannot be destroyed anyway until QP is destroyed holding FRMR or other work requests belonging to the MR.
- Reuse the same MR used in past quickly.
  - to likely find it active in the HCA's cache for repurposing for new IO.
- Requires efficient scheme for sharing dynamically in workload driven way.

Pooling of MRs



Mobile MR caching scheme

- Global pool serves the request for all the connections
  - Non blocking APIs to service the IOs at transport layer
- Pooling layer monitors and can request unused entries from L1 cache using registered callback API and internally from L2 cache
- Pool holds entries per device per ULP

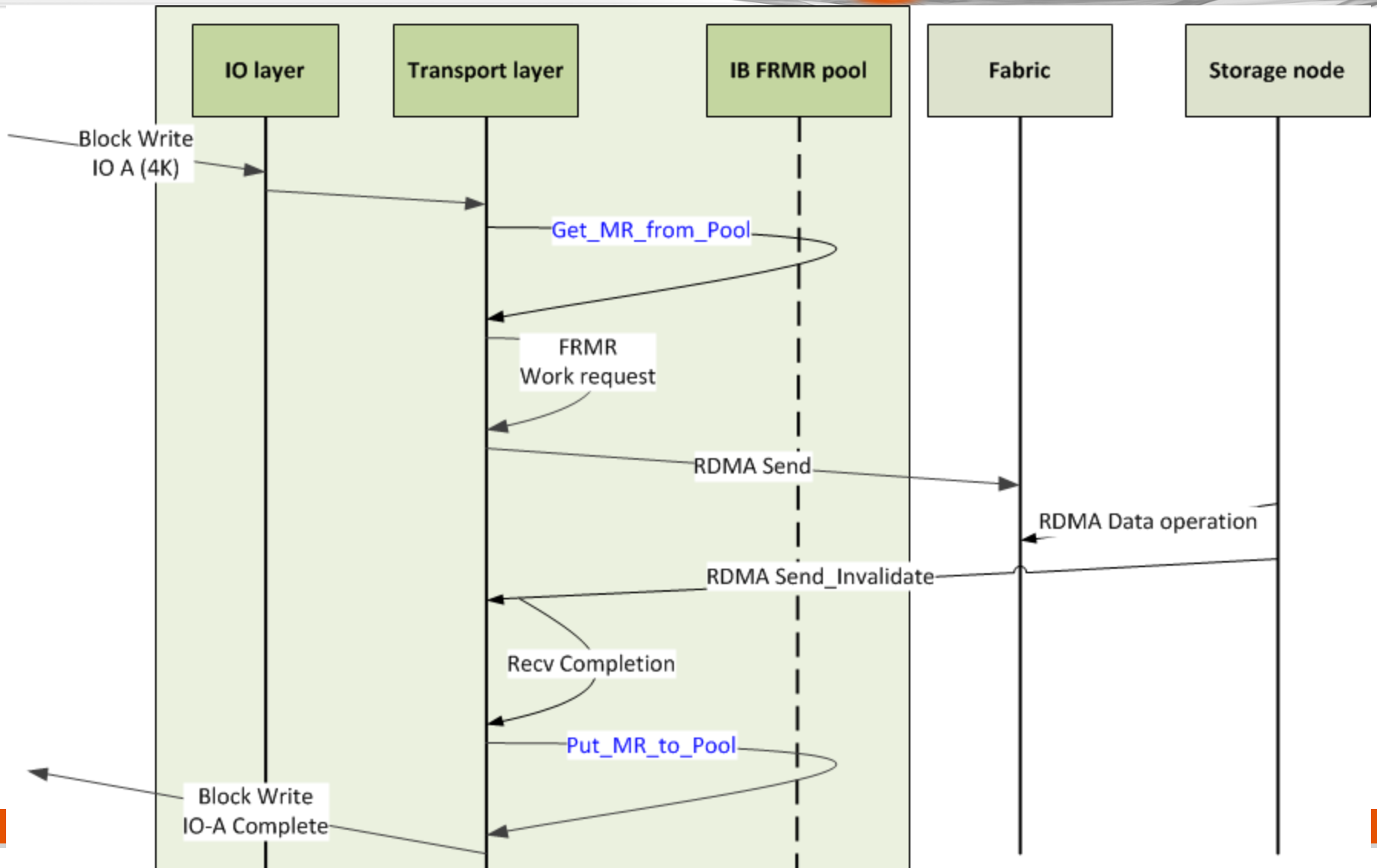
Mobile MR caching scheme

- Two level of cached entries
  - Static number of entries in each cache
  - Connection specific cache - L1 cache
    1. Small number of entries in L1 cache
    2. Enjoys user defined locking scheme
    3. Allows always servicing of minimum set of outstanding IOs
  - Shared among multiple connections – L2 cache bucket
    1. 4 to 8 buckets per pool
    2. Least used bucket is assigned to new connection
    3. Distributes connections to buckets based on connection count sharing bucket
    4. Avoid contention by hundreds of client into the global pool

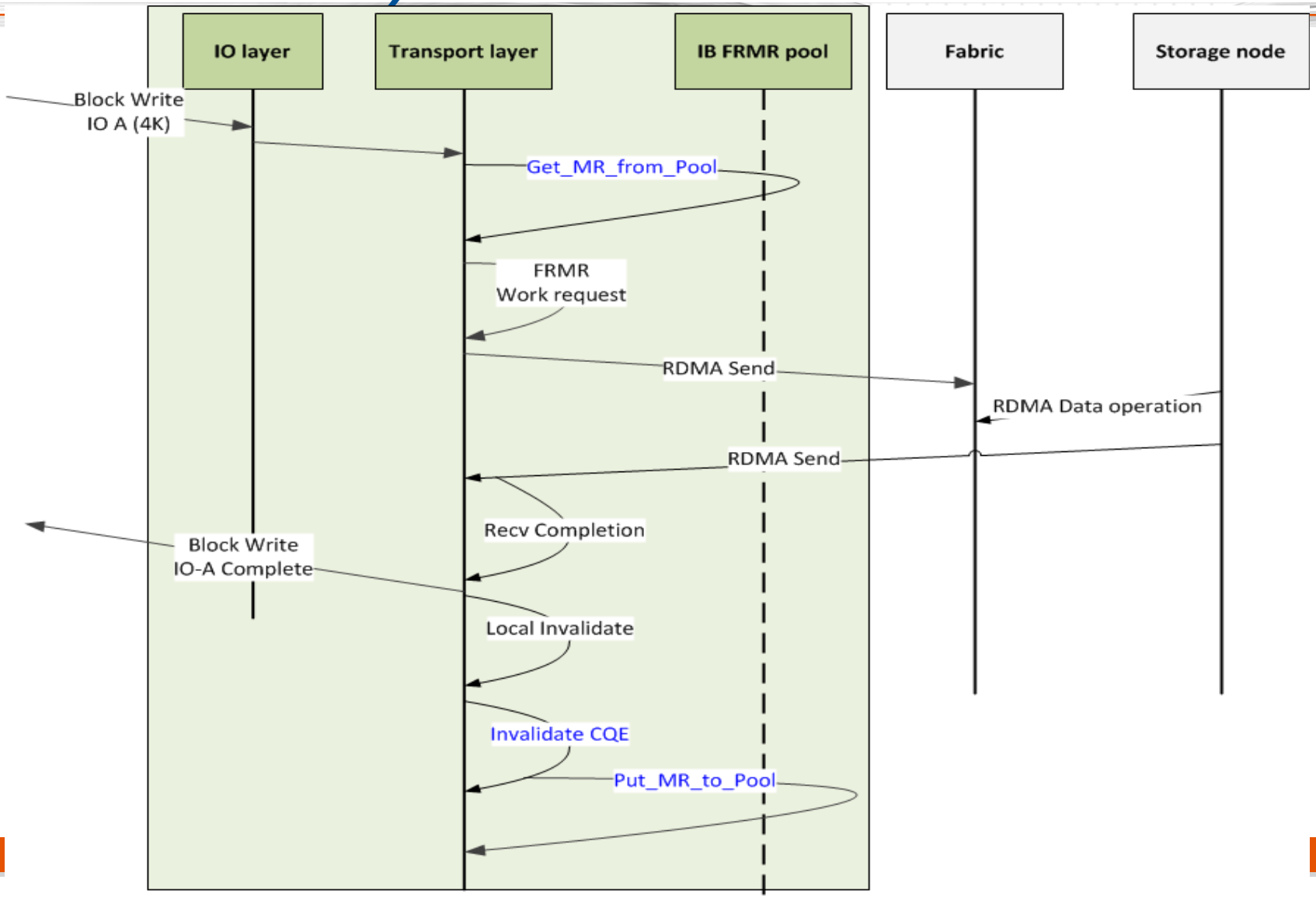
Cache handling

- Cache handling is internal to pool layer
- When MRs of the client are unavailable from connection (L1) cached entries, it fetches from L2 cache bucket.
- When L2 cache bucket is empty, it fetches the entry from the global pool
- Entries added back to L1 connection cache if taken from connection cache
- Added to L2 cache buckets if bucket has space or replenished to global pool for other connections

Pool based MR flow (Send with Invalidate)



Pool based MR data flow (Local Invalidate)



Cache handling

- Entries to global pool are added on receiving of local_invalidate completion to avoid reuse by other connections
- Connection cached entries can be added without local_invalidate completion
- Entries to global pool added on receiving send_with_invalidate completion
- More entries in global pool are created when free entries drop below the minimum threshold
- Entries in global pool are monitored every 1msec, where if there are unused than its freed in small blocks to use by other ULPs
- Entries from L2 cache buckets are put back to global pool if remain unused for certain time

Mobile MR APIs

- Pool creation and deletion blocking APIs:
  - *struct ib_frmr_pool *ib_create_frmr_pool(*
    *struct ib_device *dev, struct ib_pd *pd,*
    *struct ib_frmr_pool_params *params);*
  - *void ib_destroy_frmr_pool(struct ib_frmr_pool *pool);*

Optional

- Pool client consumer registration APIs:
  - *struct ib_frmr_pool_user **
    ib_create_frmr_pool_user(
    *struct ib_frmr_pool *pool, int num_mr);*
  - *void ib_destroy_frmr_pool_user(struct ib_frmr_pool *pool, struct ib_frmr_pool_user *user);*

Mobile MR APIs

- Run time non blocking APIs:
  - *struct ib_frmr_desc *ib_frmr_pool_get_mr(struct ib_frmr_pool_user *user);*
  - *void ib_frmr_pool_put_mr(struct ib_frmr_pool_user *user, struct ib_frmr_desc *desc);*

- Dynamically adding MR to pool:
  - *int ib_frmr_add_mr(struct ib_frmr_pool *pool, struct ib_mr *mr);*
  - *int ib_frmr_remove_mr(struct ib_frmr_pool *pool, struct ib_mr *mr);*

Example comparison

- Existing model (iSER)
  - 2048 Memory regions
    - 113 MRs per connection
    - Total Block storage devices = 18 devices
- Mobile MR based scheme
  - 2048 Memory regions
    - 32 MRs per connection
    - 256 MRs in pool
    - Total block storage devices = 54 devices
- Scaling by factor of 3 using same number of MR.

Future extensions

- Possibilities and WIP
  - User space extension:
    - For re-registering memory without repining overheads for different connections?
    - Kernel bypass to bind same memory without re-pin to different QP via data path QP
    - Ensuring check for a given user context during MR registration/ Deregistration.
  - Can file system be made distributed by just distributed storage using just smart logical volumes? Consistency? Or it can be still non shared but distributed?
  - Performance tuning
  - Fairness among connections

Thank You

OPENFABRICS ALLIANCE

11TH ANNUAL INTERNATIONAL OPENFABRICS SOFTWARE DEVELOPERS' WORKSHOP