OFVWG:
Erasure Coding RDMA
Offload

Sagi Grimberg
Problem Statement

• Modern storage arrays are usually distributed in a clustered environment.

• Problem: Disks and/or nodes inevitably tend to fail.
  – How can we survive failures and keep our data intact?
RAID 1 (Replication)

- Instead of storing the data once, we will store more copies of the data on another disk/node.
- If a disk/node fail, we are able to still recover the data.
- If we want to survive X failures, we need to replicate X instances of the data.
RAID 1 pros/cons

• Pros:
  – Simple to do
  – No need for extra computation
  – No need for reconstruct logic

• Cons:
  – Requires a high storage space for redundancy
  – Inefficient wire utilization
RAID 5 (single parity block)

- We divide our data into X blocks and calculate a single parity block and store it as well.

- If any of the drives fail we can reconstruct the original data back from the parity block.
RAID 5 pros/cons

• Pros:
  – Efficient storage utilization (small storage space for redundancy)
  – Efficient wire utilization

• Cons:
  – Requires computation to generate the parity block
  – Requires computation to reconstruct the original data
  – Need multi-level RAID to survive more than a single failure.
RAID 6 (dual parity block)

• We divide our data into X blocks and calculate two parity block and store them as well.

• If any two drives/nodes fail we can reconstruct the original data back from the parity blocks.
RAID 6 pros/cons

• Pros:
  – Efficient storage utilization (small storage space for redundancy)
  – Efficient wire utilization

• Cons:
  – Requires computation to generate two parity blocks
  – Requires computation to reconstruct the original data
  – Need multi-level RAID to survive more than two failures.
Erasure coding (generalize RAID)

- There are different types of erasure codes (Reed-Solomon, Cauchy and other MDS codes).

- The mathematical approach is to use higher rank polynomials over Galois finite fields $GF(2^w)$ in order to use minimum storage for $K$ number of disk/node failures.

- Codes can be systematic (raw data is stored) or non-systematic (data projections are stored).
Erasure coding (generalize RAID)

• Erasure codes allows us to survive M failures for any K data blocks where: $K + M \leq 2^w$

• For example if we use $GF(2^4)$ and we want to survive 4 disk failures we can protect 12 data blocks.
  – This means we only spend 33.3% of storage to store redundancy metadata.
Erasure coding Illustration

Reed Solomon Systematic Matrix Encoding Process

\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 \\
g_0 & g_1 & g_2 & g_3 & g_4 \\
g_5 & g_6 & g_7 & g_8 & g_9 \\
\end{bmatrix}
\begin{bmatrix}
d_0 \\
d_1 \\
d_2 \\
d_3 \\
d_4 \\
\end{bmatrix}
= 
\begin{bmatrix}
d_0 \\
d_1 \\
d_2 \\
d_3 \\
d_4 \\
P_1 \\
P_2 \\
P_3 \\
\end{bmatrix}
\]
Erasure coding Decode Illustration

Reed Solomon Systematic Matrix Decoding Process

\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 \\
\end{bmatrix}
\times
\begin{bmatrix}
d_0 \\
d_1 \\
d_2 \\
d_3 \\
d_4 \\
\end{bmatrix}
=
\begin{bmatrix}
x_0 \\
x_1 \\
x_2 \\
x_3 \\
p_4 \\
p_1 \\
p_2 \\
\end{bmatrix}
\]

Generator Matrix
Erasure coding Decode Illustration

1. \[
\begin{bmatrix}
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 \\
g_0 & g_1 & g_2 & g_3 & g_4
\end{bmatrix}
\] \rightarrow \text{Invert} \rightarrow \begin{bmatrix} G' \end{bmatrix}

2. \[
\begin{bmatrix} G' \end{bmatrix} \begin{bmatrix} G \end{bmatrix} = \begin{bmatrix} G' \end{bmatrix}
\]
\[
\begin{bmatrix}
\begin{bmatrix} d_0 \\
d_1 \\
d_2 \\
d_3 \\
d_4
\end{bmatrix}
\end{bmatrix} = \begin{bmatrix} G' \\
\begin{bmatrix} d_1 \\
d_2 \\
d_3 \\
d_4 \\
p_1 \\
p_2
\end{bmatrix}
\end{bmatrix} \text{ Remaining}
\]

Here’s the missing data!
Erasure coding pros/cons

• Pros:
  – *Very* Efficient storage utilization (small storage space for redundancy)
  – *Very* Efficient wire utilization
  – User can choose his configuration (K,M) – no need for multi-level RAID.

• Cons:
  – Large computation overhead needed to generate the redundancy metadata blocks
  – Large computation overhead needed to reconstruct the original data
RDMA Erasure coding offload

- Erasure codes calculations is CPU intensive.

- Next generation HCAs can offer a calculation engine.

- These HCAs can also offer a coherent calculation and networking solutions.
Programming model - SW

INPUT:
Data blocks

Coding matrix

Parity blocks

Compute redundancy:
Data blocks

Coding matrix

Parity blocks

Send to blocks OSDs:
Data blocks

Parity blocks

Post QP OSD

Post QP OSD

Post QP OSD

Post QP OSD

Post QP OSD
Programming model - Synchronous

1. Compute redundancy:
   Call sync HW EC calculation (Blocking Until HW finish calculation)

2. Send to blocks OSDs:
   Data blocks
   Post
   Post
   Post
   Post
   Post
   QP
   QP
   QP
   QP
   QP

3. Parity blocks
   Post
   Post
   QP
   QP
   QP
   QP
   OSD
   OSD
Programming model - Asynchronous

1. Call async HW EC calculation
2. Send to blocks OSDs: Data blocks
3. Async event notification: HW EC Completed EC calc
4. Parity blocks

HCA
Coding matrix
Parity blocks

Post QP QP QP QP
Post QP QP QP
Post QP

OSD

OSD

OSD

OSD

OSD
Programming model – Full striping

Call async HW EC calculation and send to peers

Send to blocks OSDs:

Data blocks

Parity blocks

Coding matrix
API – Erasure coding context

- EC context verbs representation

```c
/**
 * struct ibv_exp_ec_calc - erasure coding engine context
 *
 * @pd: protection domain
 */
struct ibv exp ec calc {
    struct ibv pd *pd;
};
```

- Allocation/Deallocation API

```c
/**
 * ibv_exp_alloc_ec_calc() - allocate an erasure coding
 * calculation offload context
 *
 * @pd: user allocated protection domain
 *
 * @attrs: initialization attributes
 *
 * Returns handle handle to the EC calculation APIs
 */
struct ibv_exp_ec_calc *
ibv_exp_alloc_ec_calc(struct ibv_pd *pd,
                      struct ibv_exp_ec_calc_init_attr *attrs);

/**
 * ibv_exp_dealloc_ec_calc() - free an erasure coding
 * calculation offload context
 *
 * @ec_calc: ec context
 */
void ibv_exp_dealloc_ec_calc(struct ibv_exp_ec_calc *calc);
```
API – EC init attributes

```c
/**
 * struct ibv_exp_ec_calc_init_attr - erasure coding engine
 * initialization attributes
 *
 * @comp_mask: compatibility bitmask
 * @max_inflight_calcs: maximum inflight calculations
 * @k: number of data blocks
 * @m: number of code blocks
 * @w: Galois field symbol size - GF(2^w)
 * @max_data_sge: maximum data sg elements to be used for encode/decode
 * @max_code_sge: maximum code sg elements to be used for encode/decode
 * @block_size: data/code block size
 * @encode_matrix: buffer contains the encoding matrix
 * @affinity_hint: affinity hint for asynchronous calcs completion steering.
 * @polling: polling mode (if set no completions will be generated by events).
 */

struct ibv_exp_ec_calc_init_attr {
    uint32_t comp_mask;
    uint32_t max_inflight_calcs;
    int k;
    int m;
    int w;
    int max_data_sge;
    int max_code_sge;
    uint8_t encode_matrix;
    int affinity_hint;
    int polling;
};
```
API – EC memory layout

```c
/**
 * struct ibv_exp_ec_mem - erasure coding memory layout context
 *
 * @data_blocks: array of data sg elements
 * @num_data_sge: number of data sg elements
 * @code_blocks: array of code sg elements
 * @num_code_sge: number of code sg elements
 * @block_size: logical block size
 */

struct ibv_exp_ec_mem {
    struct ibv_sge *data_blocks;
    int num_data_sge;
    struct ibv_sge *code_blocks;
    int num_code_sge;
    int block_size;
};
```
/**
 * ibv_exp_ec_encode_sync() - synchronous encode of given data blocks
 * and place in code_blocks
 * @ec_calc: erasure coding calculation engine
 * @ec_mem: erasure coding memory layout
 *
 * Restrictions:
 * - ec_calc is an initialized erasure coding calc engine structure
 * - ec_mem.data_blocks sg array must describe the data memory
 *   layout, the total length of the sg elements must satisfy
 *   k * ec_mem.block_size.
 * - ec_mem.num_data_sg must not exceed the calc max_data_sge
 * - ec_mem.code_blocks sg array must describe the code memory
 *   layout, the total length of the sg elements must satisfy
 *   m * ec_mem.block_size.
 * - ec_mem.num_code_sg must not exceed the calc max_code_sge
 *
 * Returns 0 on success, non-zero on failure.
 *
 */

int ibv_exp_ec_encode_sync(struct ibv_exp_ec_calc *calc,
                           struct ibv_exp_ec_mem *ec_mem)
/**
* struct ibv_exp_ec_comp - completion context of EC calculation
* 
* @done:    function handle of the EC calculation completion
* @status:  status of the EC calculation
* 
* The consumer is expected to embed this structure in his calculation context
* so that the user context would be acquired back using offsetof()
*/
struct ibv_exp_ec_comp {
    void (*done)(struct ibv_exp_ec_comp *comp);
    enum ibv_exp_ec_status status;
};

/**
* enum ibv_exp_ec_status - EC calculation status
* 
* @IBV_EXP_EC_CALC_SUCCESS: EC calc operation succeeded
* @IBV_EXP_EC_CALC_FAIL: EC calc operation failed
*/
enum ibv_exp_ec_status {
    IBV_EXP_EC_CALC_SUCCESS,
    IBV_EXP_EC_CALC_FAIL,
};
/**
 * ibv_exp_ec_encode_async() - asynchronous encode of given data blocks
 *     and place in code_blocks
 * @ec_calc:     erasure coding calculation engine
 * @ec_mem:      erasure coding memory layout
 * @ec_comp:     EC calculation completion context
 *
 * Restrictions:
 * - ec_calc is an initialized erasure coding calc engine structure
 * - ec_mem.data_blocks sg array must describe the data memory
 * - layout, the total length of the sg elements must satisfy
 *   k * ec_mem.block_size.
 * - ec_mem.num_data_sg must not exceed the calc max_data_sge
 * - ec_mem.code_blocks sg array must describe the code memory
 * - layout, the total length of the sg elements must satisfy
 *   m * ec_mem.block_size.
 * - ec_mem.num_code_sg must not exceed the calc max_code_sge
 *
 * Notes:
 * The ec_calc will perform the erasure coding calc operation,
 * once it completes, it will call ec_comp->done() handle.
 * The caller will take it from there.
 */

int ibv_exp_ec_encode_async(struct ibv_exp_ec_calc *calc,
                            struct ibv_exp_ec_mem *ec_mem,
                            struct ibv_exp_ec_comp *ec_comp);
API – Verbs stripe object

• In order to perform the full striping operation via a single API call we need to provide our striping layout (who gets what)

```c
/**
 * struct ibv_exp_ec_stripe - erasure coding stripe descriptor
 *
 * @qp: queue-pair connected to the relevant peer
 * @wr: send work request - can either be a RDMA wr or a SEND
 */

struct ibv_exp_ec_stripe {
    ibv_qp     *qp;
    ibv_send_wr *wr;
};
```
/**
 * ibv_exp_ec_encode_send() - encode a given set of data blocks
 * and place and send the data and code blocks to the wire with the qps array.
 * @ec_calc: erasure coding calculation engine
 * @ec_mem: erasure coding memory layout context
 * @data_stripes: array of stripe handles, each represents a data block channel
 * @code_stripes: array of qp handles, each represents a code block channel
 *
 * Restrictions:
 * - ec_calc is an initialized erasure coding calc engine structure
 * - ec_mem.data_blocks sg array must describe the data memory
 *   layout, the total length of the sg elements must satisfy
 *   k * ec_mem.block_size.
 * - ec_mem.num_data_sg must not exceed the calc max_data_sge
 * - ec_mem.code_blocks sg array must describe the code memory
 *   layout, the total length of the sg elements must satisfy
 *   m * ec_mem.block_size.
 * - ec_mem.num_code_sg must not exceed the calc max_code_sge
 *
 * Returns 0 on success, or non-zero on failure with a corresponding
 * errno.
 */
int ibv_exp_ec_encode_send(struct ibv_exp_ec_calc *ec_calc,
                           struct ibv_exp_ec_mem *ec_mem,
                           struct ibv_ec_stripe *data_stripes,
                           struct ibv_ec_stripe *code_stripes);
*/
/**
 * ibv_exp_ec_decode_sync() - decode a given set of data blocks
 * and code_blocks and place into output recovery blocks
 * @ec_calc: erasure coding calculation engine
 * @ec_mem: erasure coding memory layout
 * @erasures: bitmap of which blocks were erased and needs to be recovered
 * @decode_matrix: registered buffer of the decode matrix
 *
 * Restrictions:
 * - ec_calc is an initialized erasure coding calc engine structure
 * - ec_mem.data_blocks sg array must describe the data memory
 *   layout, the total length of the sg elements must satisfy
 *   k * ec_mem.block_size.
 * - ec_mem.num_data_sg must not exceed the calc max_data_sge
 * - ec_mem.code_blocks sg array must describe the code memory
 *   layout, the total length of the sg elements must satisfy
 *   number of missing blocks * ec_mem.block_size.
 * - ec_mem.num_code_sg must not exceed the calc max_code_sge
 * - erasures bitmask consists of the survived and erased blocks.
 *   The first k LS bits stand for the k data blocks followed by
 *   m bits that stand for the code blocks. All the other bits are
 *   ignored.
 *
 * Returns 0 on success, or non-zero on failure with a corresponding
 * errno.
 */

int ibv_exp_ec_decode_sync(struct ibv_exp_ec_calc *calc,
                            struct ibv_exp_ec_mem *ec_mem,
                            uint32_t erasures,
                            uint8_t *decode_matrix);
API – Asynchronous Decode

• Pretty much the same idea

```c
int ibv_exp_ec_decode_async(struct ibv_exp_ec_calc *calc,
                             struct ibv_exp_ec_mem *ec_mem,
                             uint32_t erasures,
                             uint8_t *decode_matrix,
                             struct ibv_exp_ec_comp *ec_comp);
```
Thank You