Optimized Memory Access

Table of contents

Memory Region Re-registration

Memory Region Re-registration allows the user to change attributes of the memory region. The user may change the PD, access flags or the address and length of the memory region. Memory

region supports contagious pages allocation. Consequently, it de-registers memory region followed by register memory region. Where possible, resources are reused instead of de-allocated and reallocated.

Example:

```
int ibv_rereg_mr(struct ibv_mr *mr, int flags, struct ibv_pd *pd,
void *addr, size_t length, uint64_t access, struct
ibv_rereg_mr_attr *attr);
```

@mr:	The memory region to modify.
@flags:	A bit-mask used to indicate which of the following properties of the memory region are being modified. Flags should be one of: IBV_REREG_MR_CHANGE_TRANSLATION /* Change translation (location and length) */ IBV_REREG_MR_CHANGE_PD/* Change protection domain*/ IBV_REREG_MR_CHANGE_ACCESS/* Change access flags*/
@pd:	If IBV_REREG_MR_CHANGE_PD is set in flags, this field specifies the new protection domain to associated with the memory region, otherwise, this parameter is ignored.
@addr:	If IBV_REREG_MR_CHANGE_TRANSLATION is set in flags, this field specifies the start of the virtual address to use in the new translation, otherwise, this parameter is ignored.
@length:	If IBV_REREG_MR_CHANGE_TRANSLATION is set in flags, this field specifies the length of the virtual address to use in the new translation, otherwise, this parameter is ignored.
@access:	If IBV_REREG_MR_CHANGE_ACCESS is set in flags, this field specifies the new memory access rights, otherwise, this parameter is ignored. Could be one of the following: IBV_ACCESS_LOCAL_WRITE IBV_ACCESS_REMOTE_WRITE

	IBV_ACCESS_REMOTE_READ IBV_ACCESS_ALLOCATE_MR /* Let the library allocate the memory for * the user, tries to get contiguous pages */
@attr:	Future extensions

ibv_rereg_mr returns 0 on success, or the value of an errno on failure (which indicates the error reason). In case of an error, the MR is in undefined state. The user needs to call ibv_dereg_mr in order to release it.

Please note that if the MR (Memory Region) is created as a Shared MR and a translation is requested, after the call, the MR is no longer a shared MR. Moreover, Re-registration of MRs that uses NVIDIA PeerDirect™ technology are not supported.

Memory Window

Memory Window allows the application to have a more flexible control over remote access to its memory. It is available only on physical functions/native machines The two types of Memory Windows supported are: type 1 and type 2B.

Memory Windows are intended for situations where the application wants to:

- Grant and revoke remote access rights to a registered region in a dynamic fashion with less of a performance penalty
- Grant different remote access rights to different remote agents and/or grant those rights over different ranges within registered region

For further information, please refer to the InfiniBand specification document.

> **Note**
>
> Memory Windows API cannot co-work with peer memory clients (PeerDirect).

Query Capabilities

Memory Windows are available if and only the hardware supports it. To verify whether Memory Windows are available, run `ibv_query_device`.

For example:

```
struct ibv_device_attr device_attr = {.comp_mask =
IBV_DEVICE_ATTR_RESERVED - 1};
ibv_query_device(context, & device_attr);
if (device_attr.exp_device_cap_flags & IBV_DEVICE_MEM_WINDOW ||
                  device_attr.exp_device_cap_flags &
IBV_DEVICE_MW_TYPE_2B) {
/* Memory window is supported */
```

Memory Window Allocation

Allocating memory window is done by calling the `ibv_alloc_mw` verb.

```
type_mw = IBV_MW_TYPE_2/ IBV_MW_TYPE_1
mw = ibv_alloc_mw(pd, type_mw);
```

Binding Memory Windows

After being allocated, memory window should be bound to a registered memory region. Memory Region should have been registered using the IBV_ACCESS_MW_BIND access flag.

For further information on how to bind memory windows, please see rdma-core man page.

Invalidating Memory Window

Before rebinding Memory Window type 2, it must be invalidated using `ibv_post_send` - see here.

Deallocating Memory Window

Deallocating memory window is done using the ibv_dealloc_mw verb.

```
ibv_dealloc_mw(mw);
```

User-Mode Memory Registration (UMR)

User-mode Memory Registration (UMR) is a fast registration mode which uses send queue. The UMR support enables the usage of RDMA operations and scatters the data at the remote side through the definition of appropriate memory keys on the remote side.

UMR enables the user to:

- Create indirect memory keys from previously registered memory regions, including creation of KLM's from previous KLM's. There are not data alignment or length restrictions associated with the memory regions used to define the new KLM's.
- Create memory regions, which support the definition of regular non-contiguous memory regions.

On-Demand-Paging (ODP)

On-Demand-Paging (ODP) is a technique to alleviate much of the shortcomings of memory registration. Applications no longer need to pin down the underlying physical pages of the address space, and track the validity of the mappings. Rather, the HCA requests the latest translations from the OS when pages are not present, and the OS invalidates translations which are no longer valid due to either non-present pages or mapping changes. ODP does not support contiguous pages.

ODP can be further divided into 2 subclasses: Explicit and Implicit ODP.

- Explicit ODP

  In Explicit ODP, applications still register memory buffers for communication, but this operation is used to define access control for IO rather than pin-down the

pages. ODP Memory Region (MR) does not need to have valid mappings at registration time.

- Implicit ODP

  In Implicit ODP, applications are provided with a special memory key that represents their complete address space. This all IO accesses referencing this key (subject to the access rights associated with the key) does not need to register any virtual address range.

Query Capabilities

On-Demand Paging is available if both the hardware and the kernel support it. To verify whether ODP is supported, run `ibv_query_device.`

For further information, please refer to the `ibv_query_device` manual page.

Registering ODP Explicit and Implicit MR

ODP Explicit MR is registered after allocating the necessary resources (e.g. PD, buffer), while ODP implicit MR registration provides an implicit lkey that represents the complete address space.

For further information, please refer to the `ibv_reg_mr` manual page.

De-registering ODP MR

ODP MR is deregistered the same way a regular MR is deregistered:

```
ibv_dereg_mr(mr);
```

Advice MR Verb

The driver can pre-fetch a given range of pages and map them for access from the HCA. The advice MR verb is applicable for ODP MRs only.

For further information, please refer to the `ibv_advise_mr` manual page.

ODP Statistics

To aid in debugging and performance measurements and tuning, ODP support includes an extensive set of statistics.

For further information, please refer to rdma-statistics manual page.

Inline-Receive

The HCA may write received data to the Receive CQE. Inline-Receive saves PCIe Read transaction since the HCA does not need to read the scatter list. Therefore, it improves performance in case of short receive-messages.

On poll CQ, the driver copies the received data from CQE to the user's buffers.

Inline-Receive is enabled by default and is transparent to the user application. To disable it globally, set MLX5_SCATTER_TO_CQE environment variable to the value of 0. Otherwise, disable it on a specific QP using mlx5dv_create_qp() with MLX5DV_QP_CREATE_DISABLE_SCATTER_TO_CQE.

For further information, please refer to the manual page of mlx5dv_create_qp().