



Checkpoint/Restore for RDMA: Saving and Restoring Live Queue Pairs with CRIU

Raphael Norwitz

Linux Plumbers Conference, October 2026



Disclaimer: Not an Official Product Roadmap

This is an extremely complex problem and multiple approaches are being pursued. Nothing discussed here should be taken as related to any NVIDIA roadmap. We are a Linux kernel team pursuing extensions to Linux APIs to migrate QPs.

Current Status - Working Solution with GPU Memory

CRIU + CUDA Checkpoint + RDMA Checkpoint/Restore Support



Software Functionality

RDMA Software Stack: User contexts along with PD, MR, CQ and RC QP user objects can be checkpointed and restored.

End-to-End Validation:

Demonstrated checkpointing and restoring running RDMA Perf test (i.e. `ib_write_bw`) cases with traffic resuming successfully on restore.



Hardware Enablement

NIC hardware: Implemented on NVIDIA ConnectX-7 VFs with stock firmware. A RXE reference is also implemented.

CUDA Checkpoint Integration:

Checkpointed and restored jobs using GPU memory via DMABUF at line rate (up to 400G).

CPU Support: Validated on x86 and ARM.



Current Focus

Moving Upstream: Smoothing over rough edges to make our code suitable for upstream submission.

Realistic Workloads: We would like to validate on realistic large training and inference jobs.

Agenda

1. Motivation
2. Background
3. Implementation Overview
4. Stable IOVA Management
5. Potential Approaches and Solutions

Checkpointing Use Cases in Machine Learning

Key workloads: Large LLM Training and Inference Jobs

- Workload management as part of daily operations
 - Enable SW and FW updates and HW repairs across nodes and racks
 - Relocate jobs to consolidate free capacity
- Hardware failure recovery
 - Restore a saved checkpoint on healthy hosts and resume workload
- Workload switching
 - Suspend and resume jobs as GPU capacity is reassigned
 - ex. Inference during the day (demand high), train at night (demand lower)
- Workload-managed checkpointing is already common across industry
 - Challenging for some operators who run workloads they do not control
- Preference for checkpoint-restore to be transparent to the workload
 - Value add: Restore from golden process image to save on cold start time

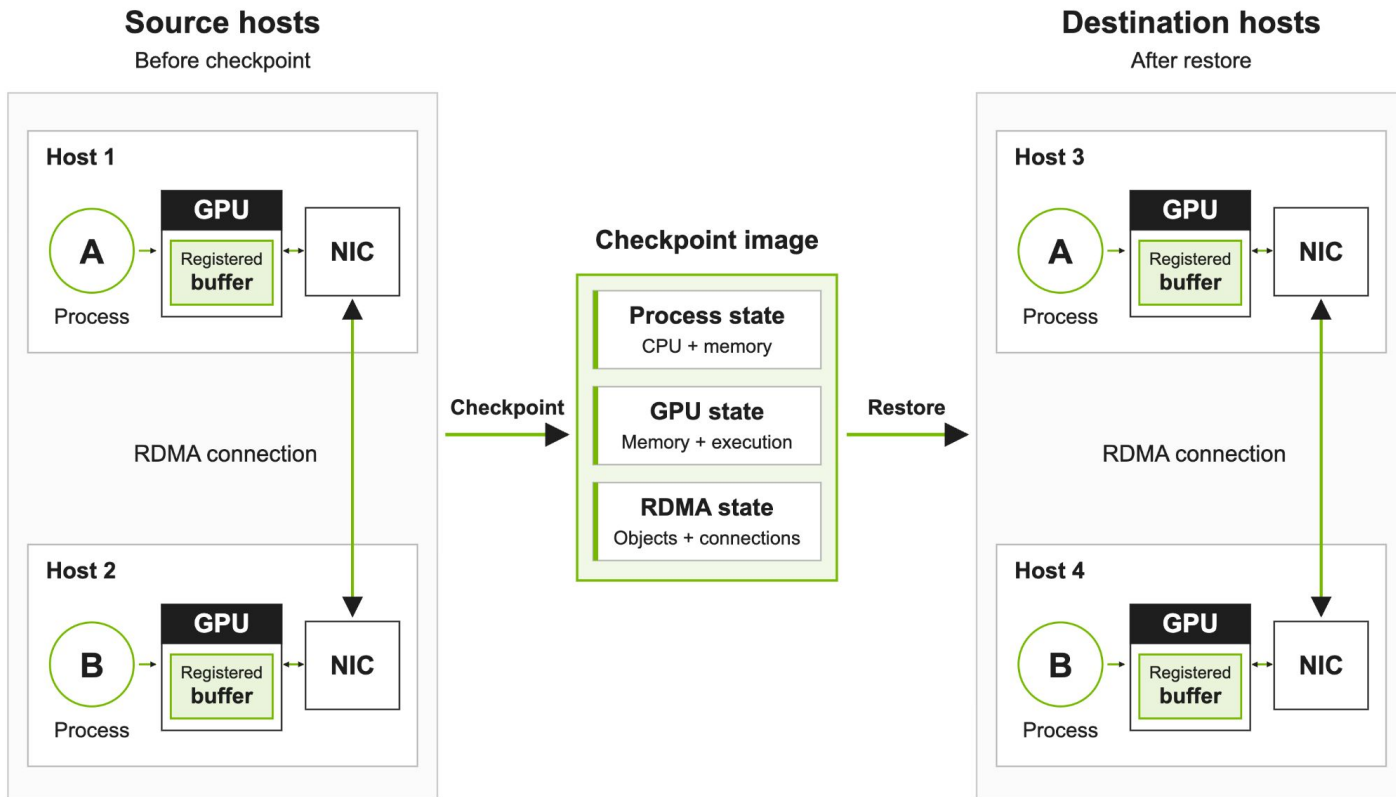
CRIU

CRIU is promising for infra-managed checkpointing but RDMA is a blocker

- Virtual Machines: well established migration and snapshot restore support
- ML infra providers typically run workloads bare metal or in containers
 - Reluctant to adopt VMs
- CRIU: Common Linux process (container) checkpoint-restore solution
- GPU Checkpoint/Restore is already supported on NVIDIA and AMD
- Challenge: Large jobs use RDMA for Scale-out Networking
 - No way to checkpoint/restore RDMA state with CRIU
 - App can shut down RDMA connections but that breaks workload transparency

Migration Objective

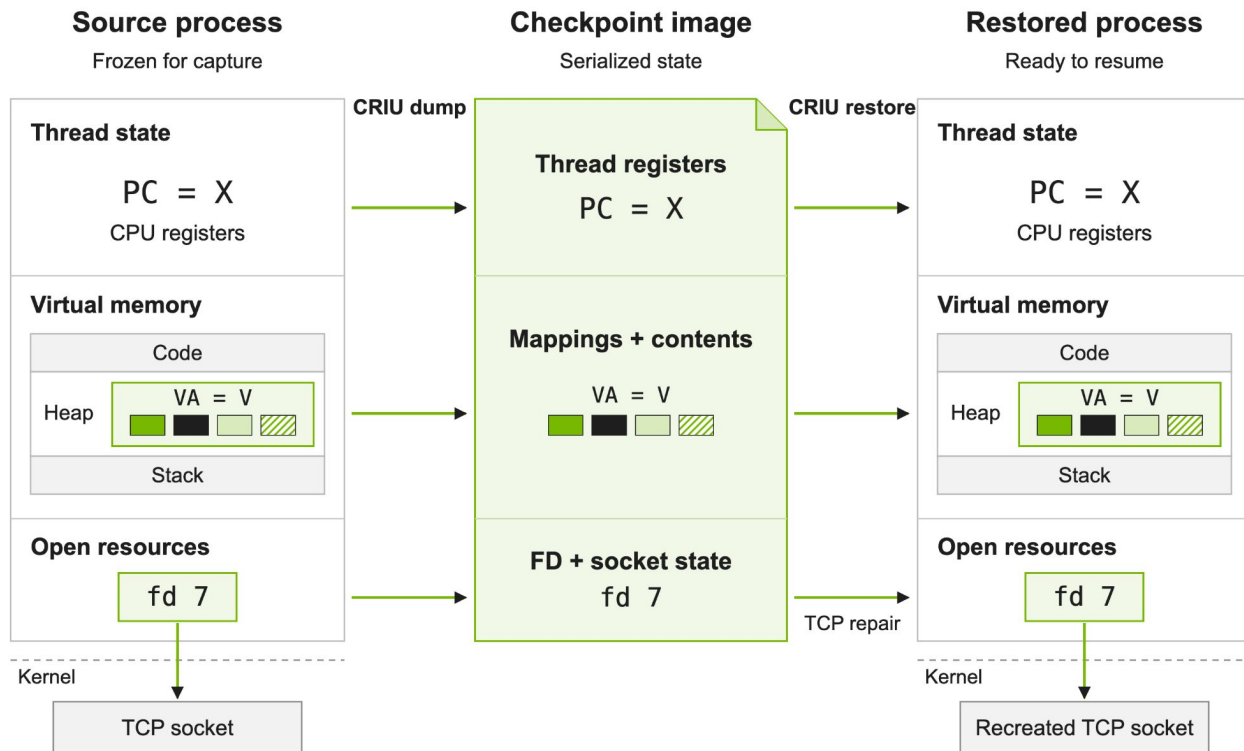
Demonstrate Checkpoint and Restore on Realistic At-Scale LLM Workloads with CRIU



Preserve application state and RDMA connections across hosts

CRIU Overview

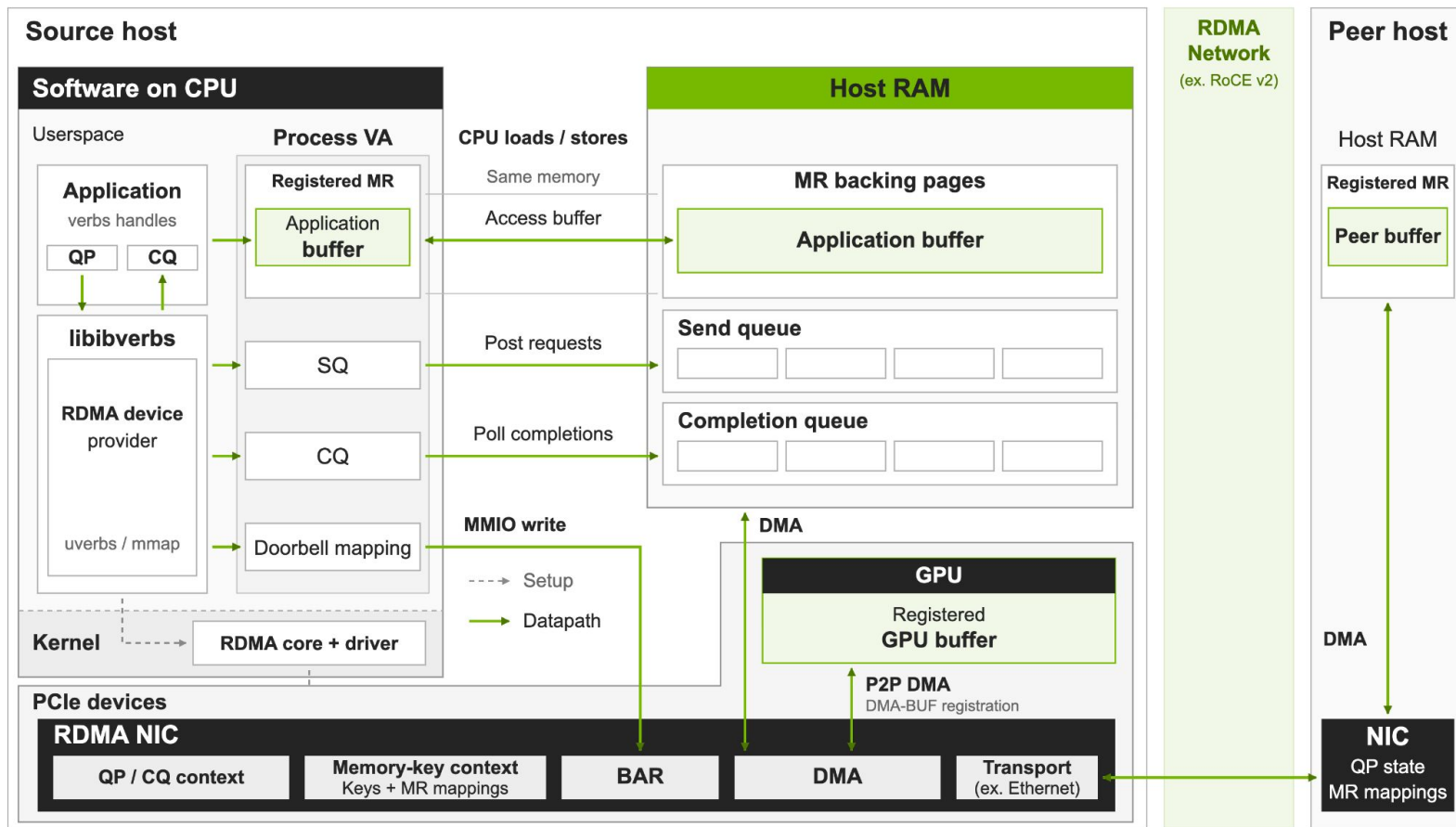
Example of How CRIU Checkpoints and Restores Processes



ptrace / procfs / resource-specific APIs

RDMA Overview

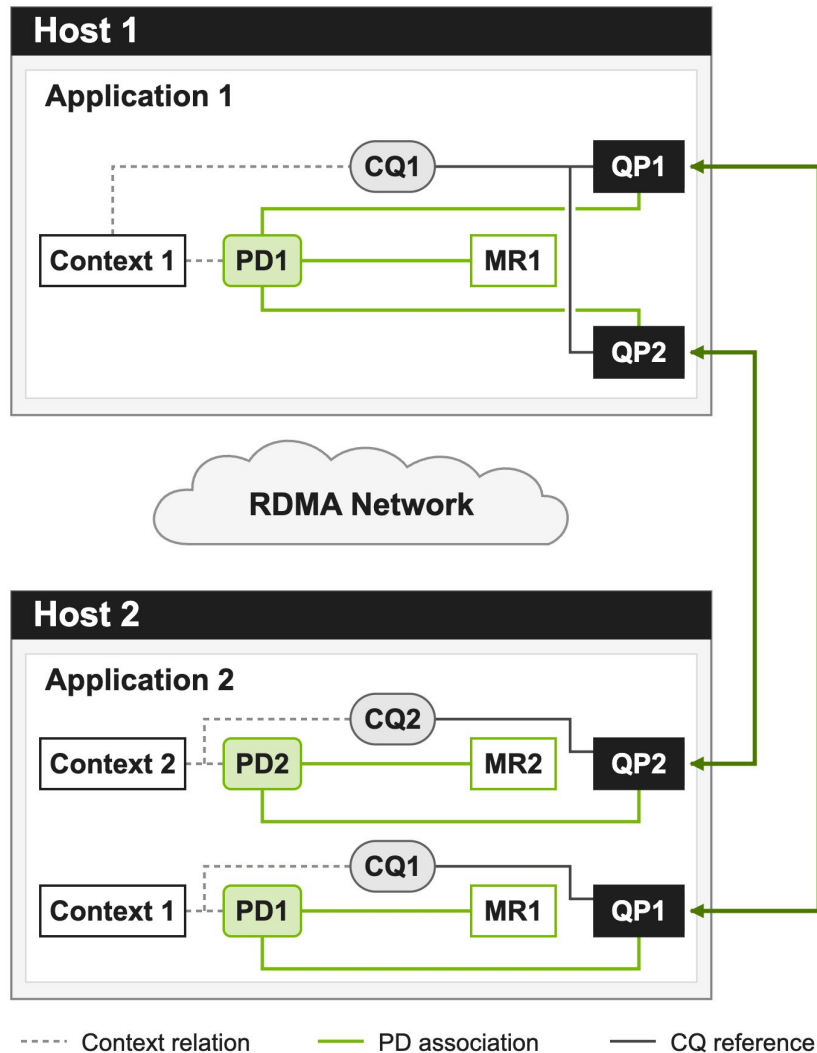
High bandwidth, low latency, kernel bypass networking



RDMA Object Graphs

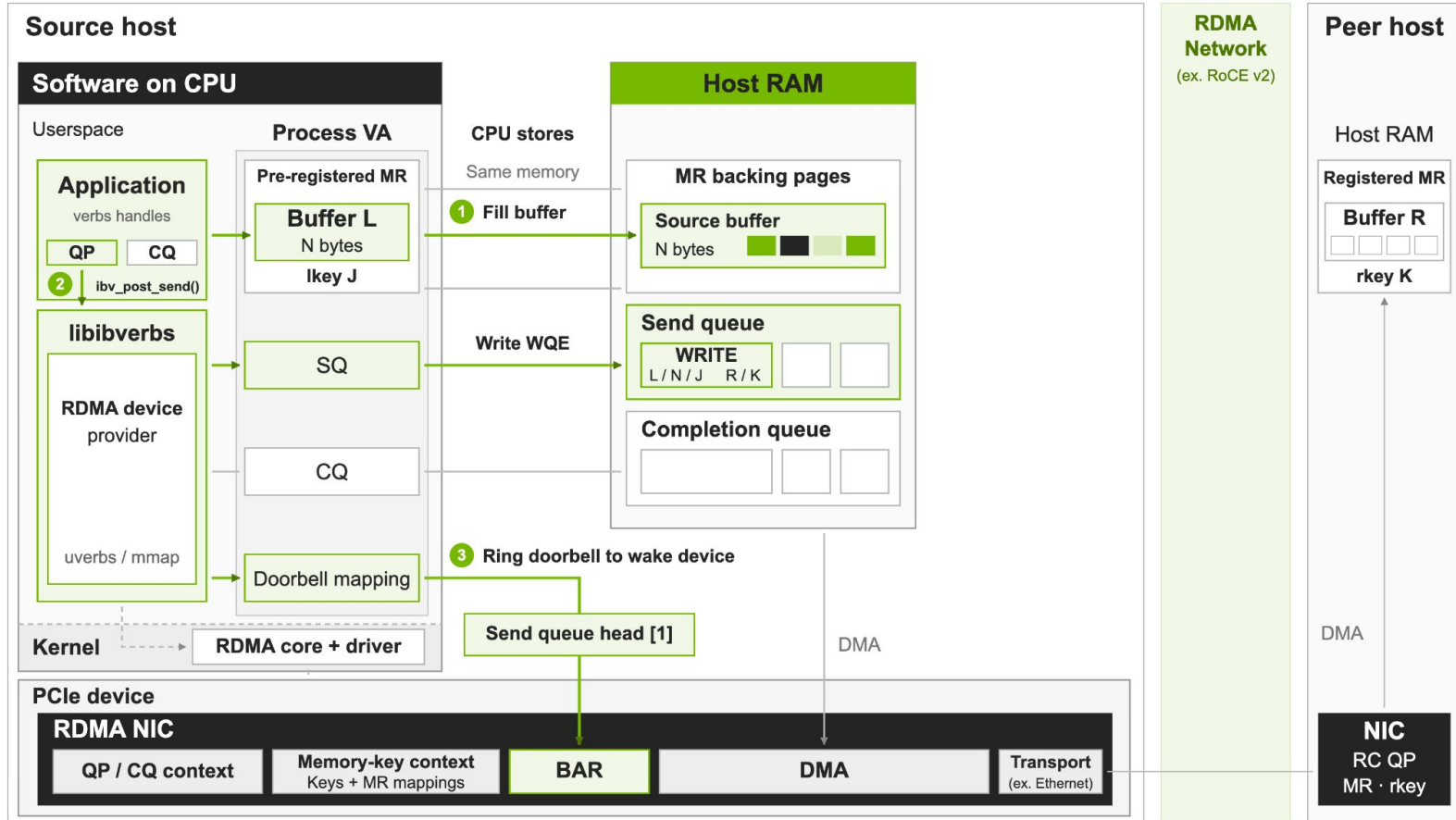
Software object graph used for data transfer

- User Context: Command interface
 - UVERBs ioctl(s)
 - Create user objects and set configurations
- User Objects: Data transfer primitives
 - Memory Regions (MRs)
 - Queue Pairs (QPs)
 - Completion Queues (CQs)
 - Protection Domains (PDs)
- Data transfer:
 - SEND/RECEIVE: Requests
 - READ/WRITE: Bulk data transfer
- Directed Acyclic Graph Mappings:
 - Context: CQs and PDs
 - PD: MRs and QPs
 - CQ: QPs



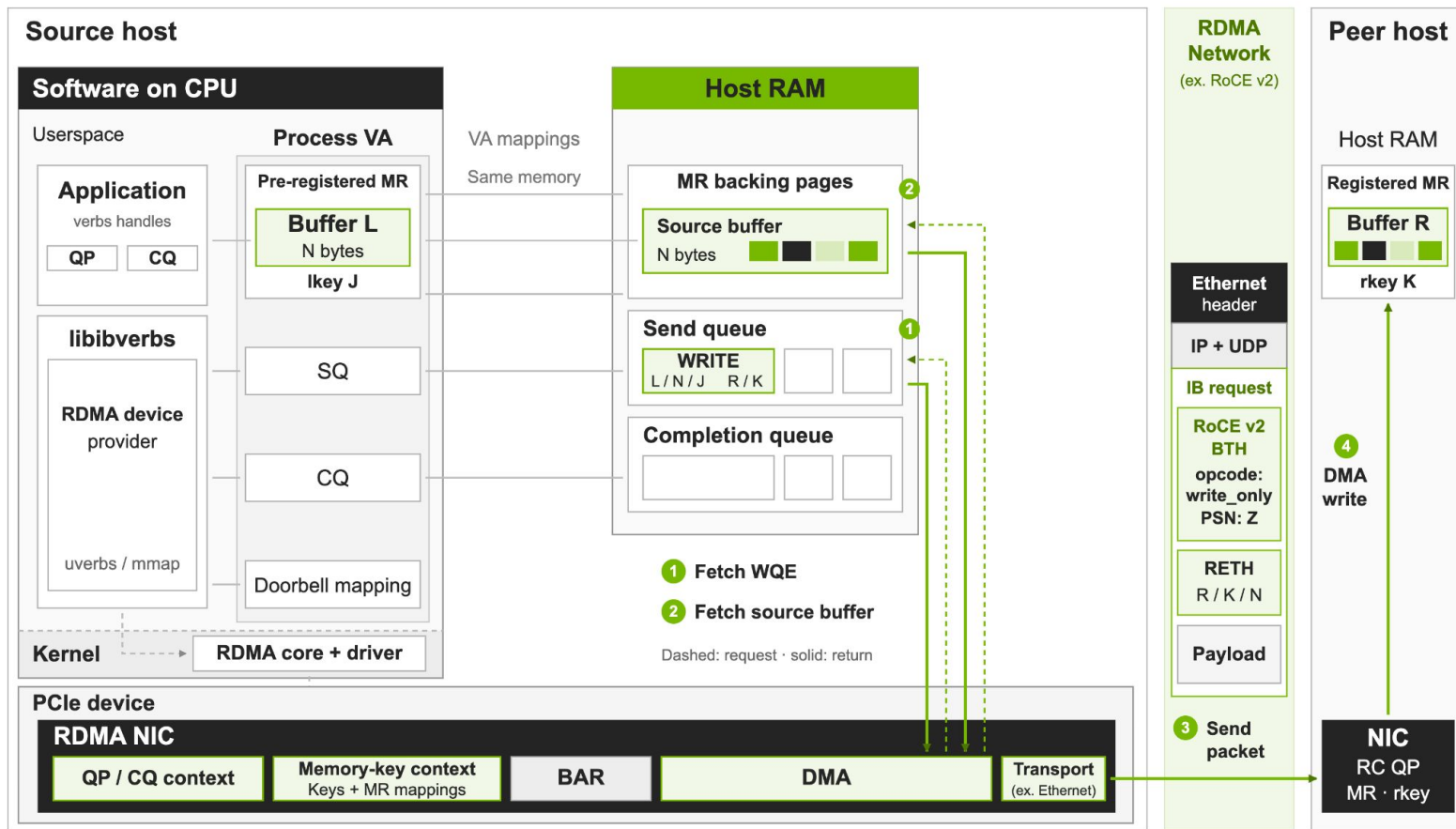
Example: RDMA WRITE Request

Step 1: Dispatch the request



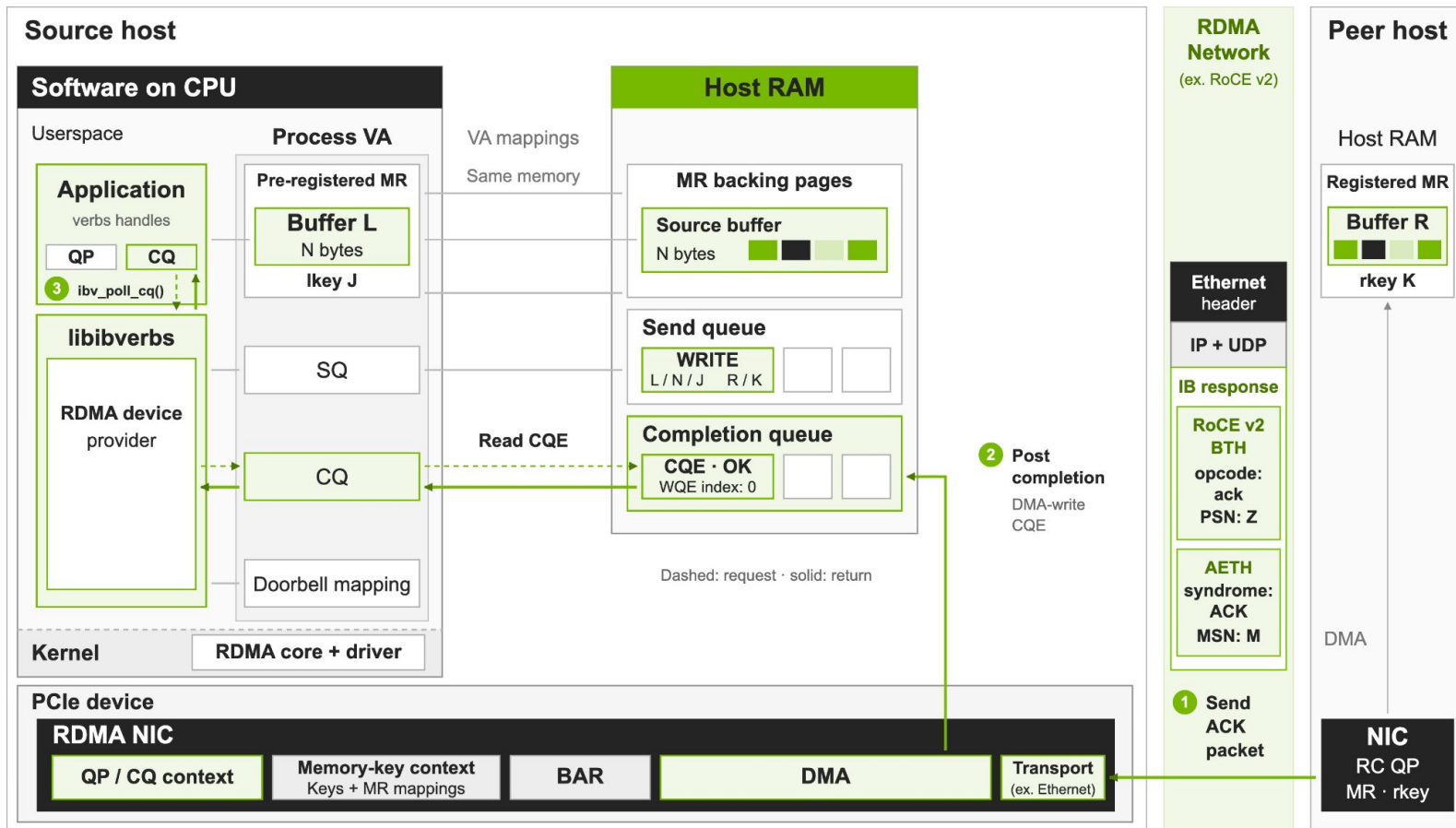
Example: RDMA WRITE Request

Step 2: NIC wakes up and services the request



Example: RDMA WRITE Request

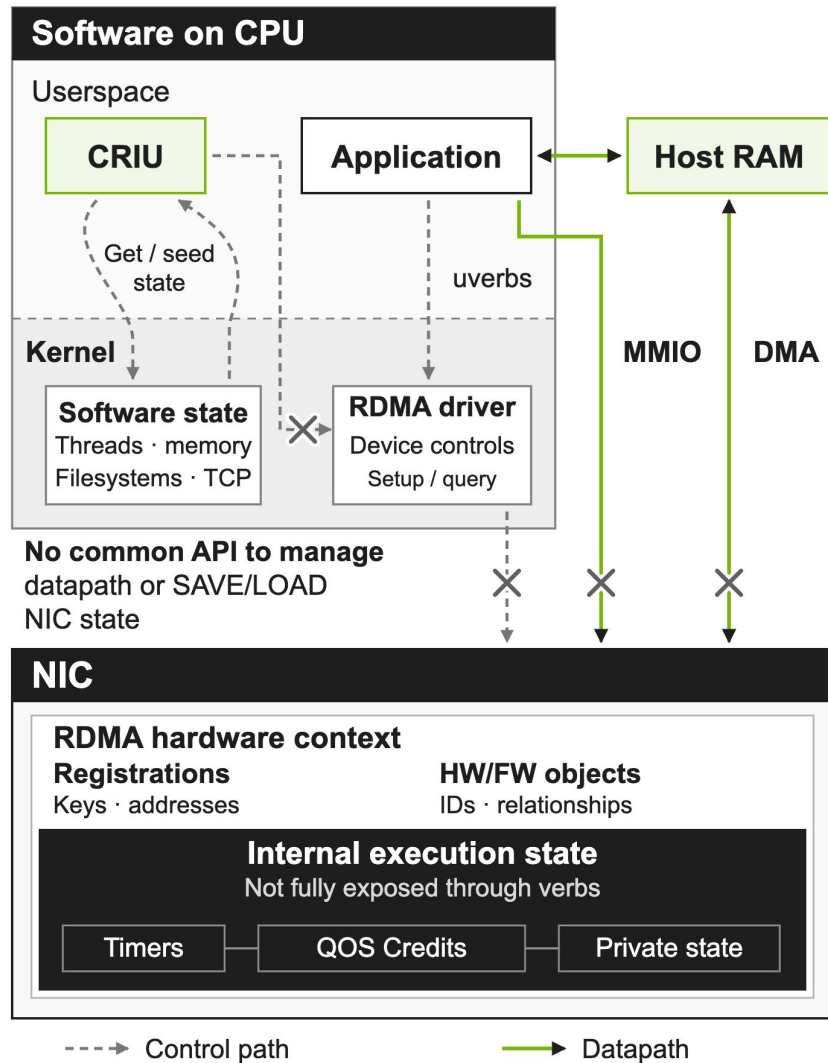
Step 3: Completion comes back



Handling Direct NIC HW Access

CRIU typically relies on kernel hooks, RDMA bypasses the kernel.

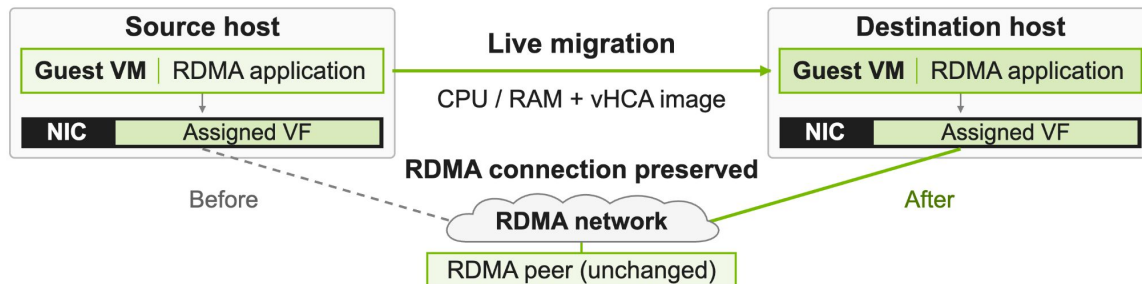
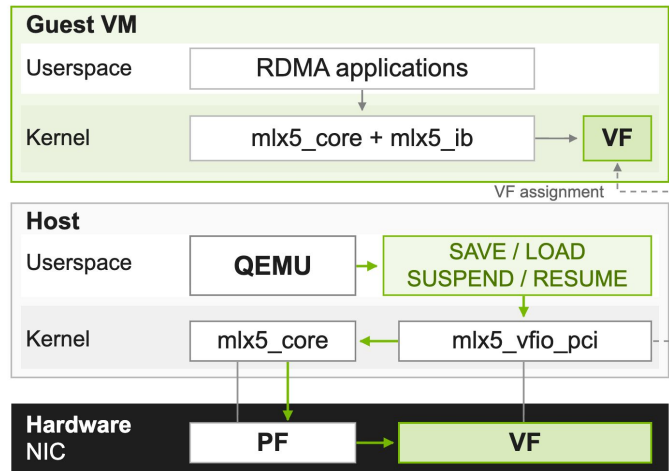
- CRIU relies on kernel interfaces
 - ptrace: CPU threads
 - procfs: access memory
 - filesystem: regular system files
 - Kernel TCP stack
- RDMA datapath bypasses kernel
 - Safely quiesce traffic?
 - Retrieve NIC-internal state?
 - Seed it back?
- Software-only academic solutions
 - Virtualize RDMA IDs ([MigRDMA](#))
 - Modify wire protocol ([MigrOS](#))
 - Both hurt performance
- **Precedent for RDMA connection save/restore?**



SRIOV Virtual Machine Live Migration

NVIDIA Connect-X and Bluefield devices have supported RDMA migration state since 2018

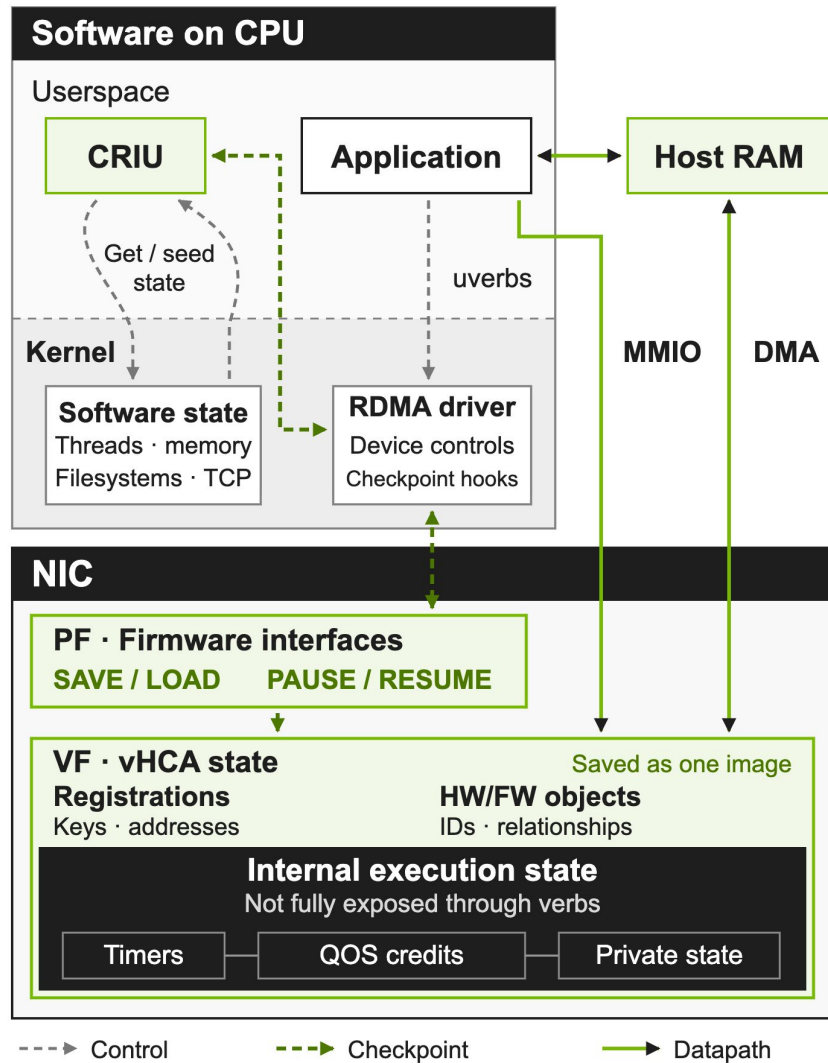
- **Live Migration with mlx5 VFs (VFIO)**
 - Supported on ConnectX-7 and onwards
 - Application RDMA connections kept intact
- **Modified VFIO Configuration**
 - VF is bound to mlx5_core (along with PF on host)
 - On host, VFs are bound to mlx5_vfio_pci driver variant
 - QEMU orchestrates VF migration path
- **Fundamental Primitive: vHCA APIs on PF**
 - Encapsulate device state in vHCA image
 - Suspend and resume the datapath



Key Primitives: vHCA FW APIs

Reusing VF Migration FW API for CRIU

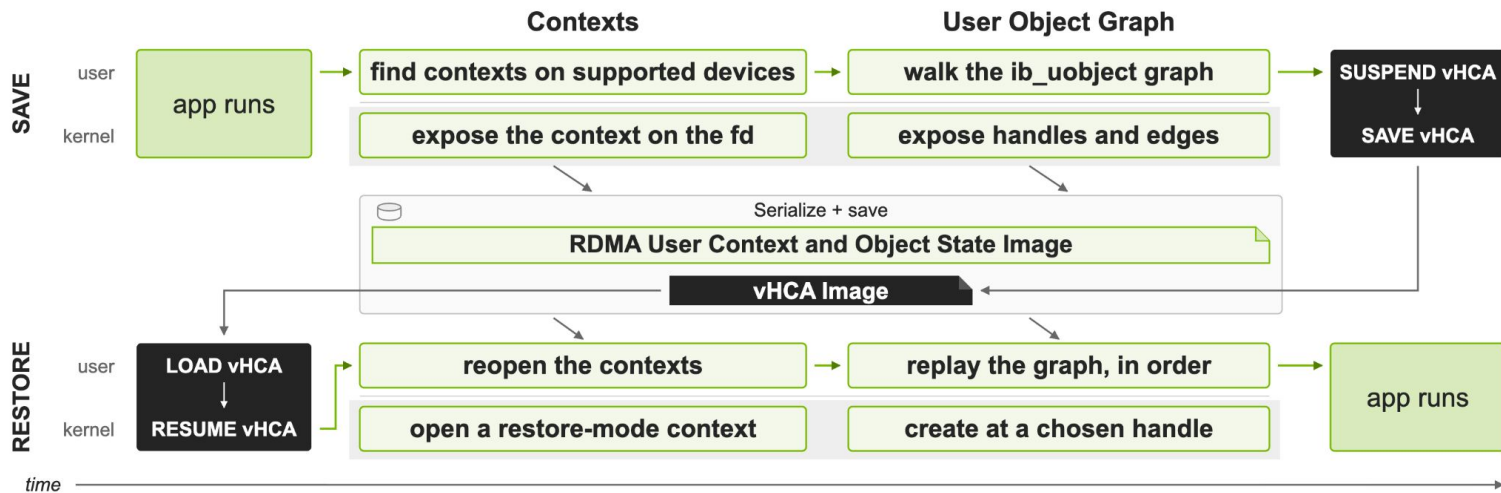
- vHCA: Opaque blob with full VF state
 - API interface on the PF
 - Take in VF index (vf_id)
- State management interfaces
 - Save: SAVE_VHCA(vf_id)
 - Restore: LOAD_VHCA(vf_id, blob)
- Datapath control mechanism
 - SUSPEND_VHCA / RESUME_VHCA
- **How can CRIU consume these APIs?**



Our Approach: RDMA Checkpoint/Restore with the Kernel Stack

CRIU exercises a symmetric workflow for save and restore modulo vHCA handling

- Fundamental approach: Handle contexts and then object graphs
 - Query and re-seed kernel state and FW bindings
 - Serialize CRIU interpretable image
 - VHCA handling happens after save and before restore
- Device agnostic: Same fundamental flow for mlx5 VFs and RXE
 - Core flow implemented in common code
 - Separate CRIU plugins with callbacks for device-specific state



CRIU RDMA Save Flow

Required state is mostly exposed by existing APIs

- Discover RDMA contexts
 - Find UVERBs fds in process FD walk
 - Find context number from fd_info
- Capture RDMA object graph:
 - Enumerate with RDMA Netlink
 - Contains most HW agnostic state
 - Filter with fd_info context number
 - Record relations and dependencies
- Query device-specific object state
 - FW and MMIO bindings
- Serialize and persist rdma_uobj.img
- SUSPEND_VHCA and SAVE_VHCA
 - vHCA persisted as separate file

1

Find the uverbs context

/proc/PID/fd

3 file 4 socket

7 uverbs0

fdinfo: ctxn = 42

Device + context ID

2

RDMA Netlink: records to object graph

Matching context

PD 1 ctxn 42

CQ 2 ctxn 42

MR 3 PD 1

QP 4 PD 1, CQ 2

Reconstruct DAG

PD 1

CQ 2

MR 3

QP 4

3

Query device state through object hooks

Object record

Plugin query

FW IDs + attributes

4

SUSPEND_VHCA

5

Persist object graph

rdma-uobj.img

DAG + object attributes

6

SAVE_VHCA and final image writes

VF state

vHCA image
Per VF

Plugin image
Context metadata

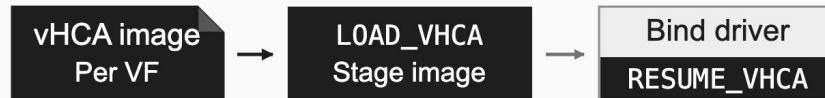
CRIU RDMA Restore Flow

Primary addition: `RESTORE_* UVERBs` family

- Bootstrap vHCA
 - `LOAD_VHCA`, `RESUME_VHCA`
 - bind driver
- Restore process memory
 - includes DMA mapped buffers
- Restore RDMA contexts:
 - open uverbs with `RESTORED` flag set
 - Seed context FW and device bindings
- Restore RDMA object graph:
 - new UVERBS `RESTORE` family
 - Graph replayed in dependency order
 - i.e. CQs, PDs, MRs, QPs
 - HW agnostic restore paths
 - Call into per-`ib_device` callbacks
- Resume Application Threads

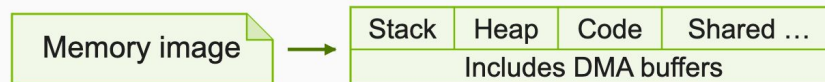
1

Bootstrap vHCA



2

Restore process memory



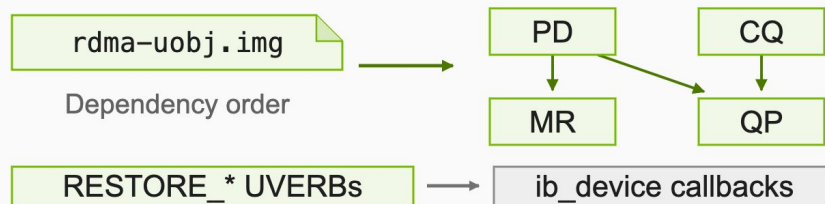
3

Restore contexts



4

Restore object graph



5

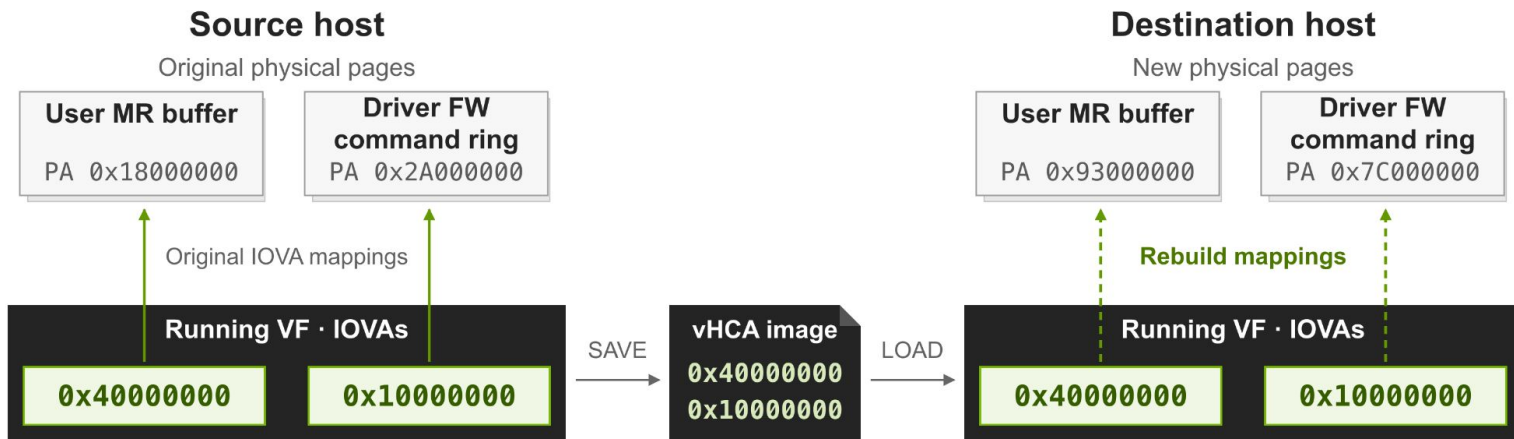
Resume CPUs



Key Challenge: Stable DMA Addressing

Addresses embedded in the VHCA image need to stay consistent

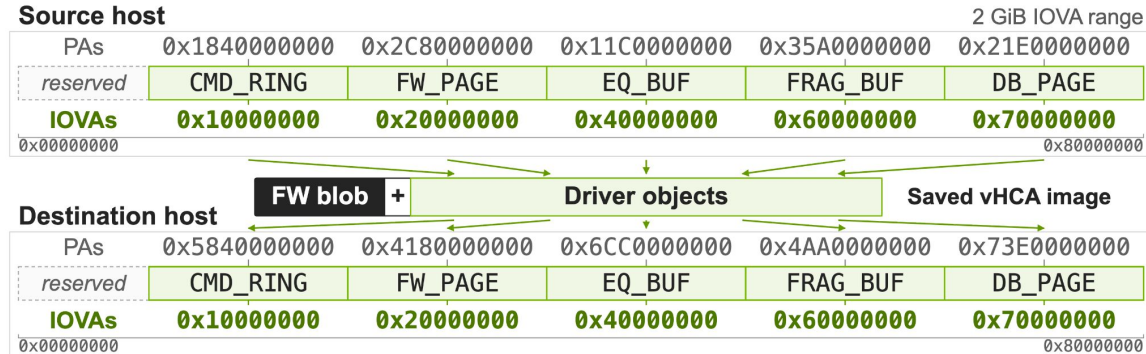
- Devices need direct access to RDMA user objects and driver internal structures
- DMA mappings are part of the device state
 - They come embedded in the vHCA image
 - Removing them from the vHCA is impractical
- Restore must re-create identical mappings
 - Our approach: Give Driver and RDMA stack control of an IOVA range
 - Implement an allocator over managed range



Restorable Driver Objects

Let's assume we have a fixed range of IOVAs carved out - how can we use it?

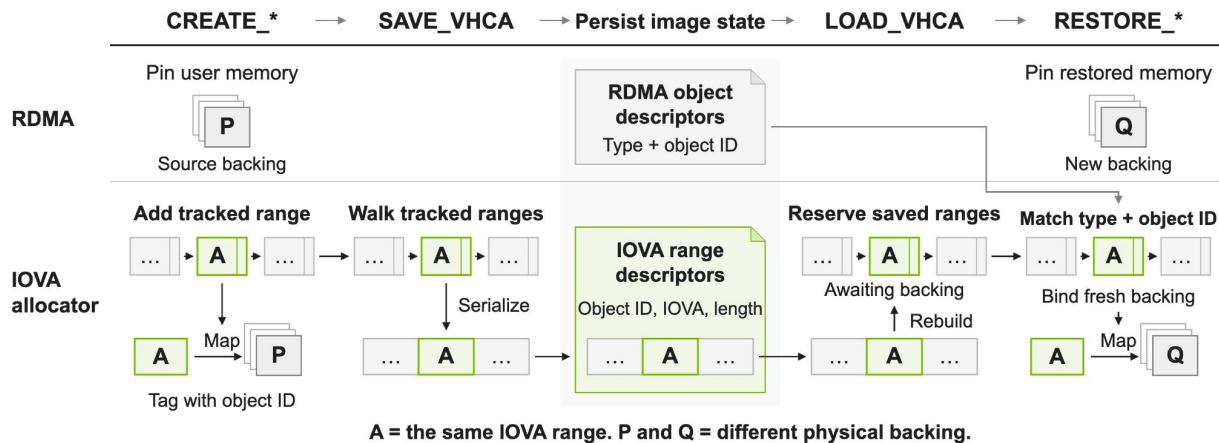
- State required to bootstrap the device (FW cmd rings, FW owned pages..etc.)
 - Setup in driver probe()
- Tag all known allocations with fixed IOVAs slots
- Allocations in fresh device probe(): Allocate fresh pages, return slot IOVA
- SAVE_VHCA: Serialize slot contents alongside FW vHCA Image
- LOAD_VHCA: Replay allocated state and recreate mappings
- Allocations in restored device probe(): Return already mapped IOVAs and PAs



Restorable RDMA User Objects

Let's assume we have a fixed range of IOVAs carved out - how can we use it?

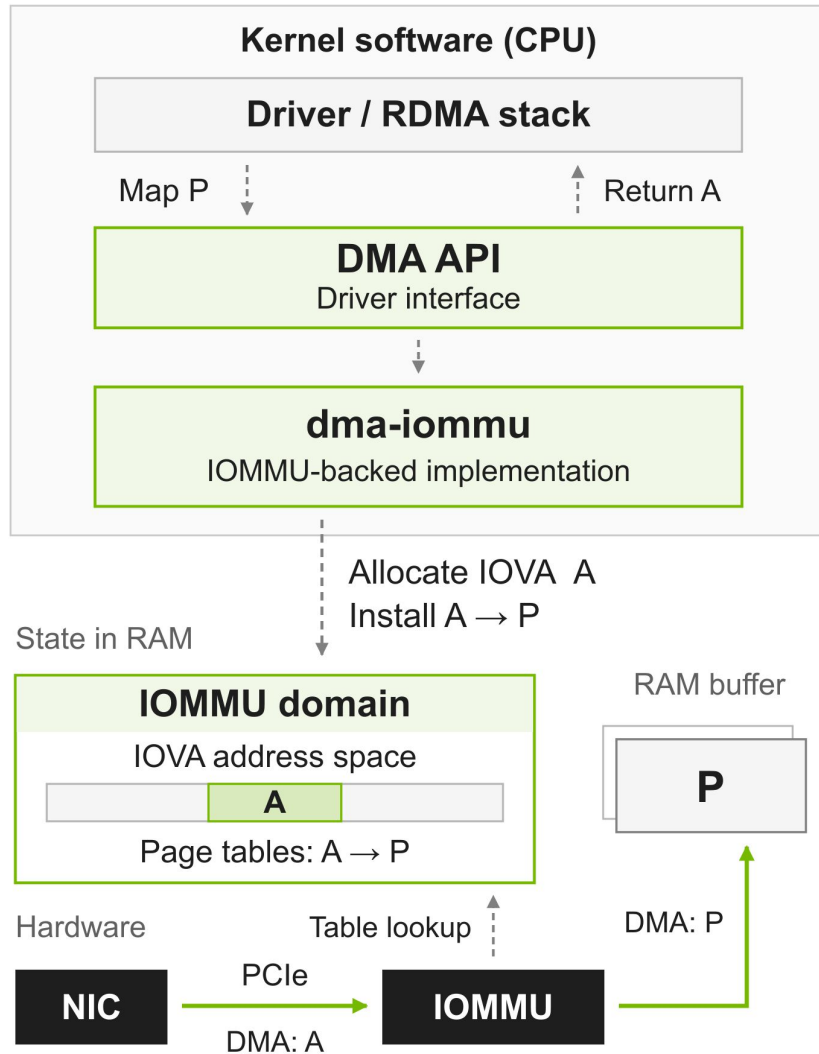
- CRIU saves and restores content like ordinary process memory
- Proposed approach for uobject IOVA stability:
 - CREATE uobject: Call into an allocator to get IOVA and create mapping
 - SAVE_VHCA: Serialize mappings inside VHCA image (separate image?)
 - LOAD_VHCA: Re-Seed mappings
 - RESTORE uobject: Pass in new pages and match against existing records
- How do we match on RESTORE? Currently: uobject type + FW ID. Maybe IOVA?



Device Addressing Background

Let's get into some lower level detail

- DMA addresses: IOVAs or PAs
 - Impossible to reserve PAs in Linux
 - IOMMU must be enabled
- IOMMU enforces device addressing
 - domains: IOVA address spaces
 - Default domain for IOMMU group
- Default DMA API:
 - IOVAs in default domain
 - API controls placement
- **How can we carve out a fixed IOVA arena?**



Proposed approach: IOVA-Based DMA API Extension

All we need is a slight tweak on what the DMA API already provides

- Existing Reservation API: bool dma_iova_try_alloc()
 - Input: DMA Device, size and alignment; Output: IOVA address + size
 - Carves range out of default IOMMU domain
- Mapping APIs:
 - dma_iova_link() and dma_iova_unlink() to map/unmap, dma_iova_sync() to persist mappings
- Precedents:
 - ex. On-Demand Paging: Keep pages linked to consistent IOVAs as they are faulted in and out
 - Also used by mlx5_vfio_pci, Block-MQ and DMABUF
- Problem: We need a fixed IOVA as an input rather than an output
 - New API: dma_iova_alloc_fixed()
 - Reuse link/unlink/sync; add exact-base reservation

Existing API

dma_iova_try_alloc()

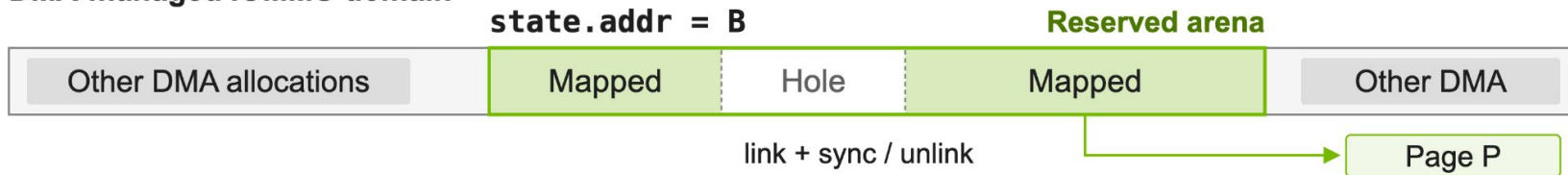
Allocator chooses B

New API

dma_iova_alloc_fixed()

Caller supplies saved B

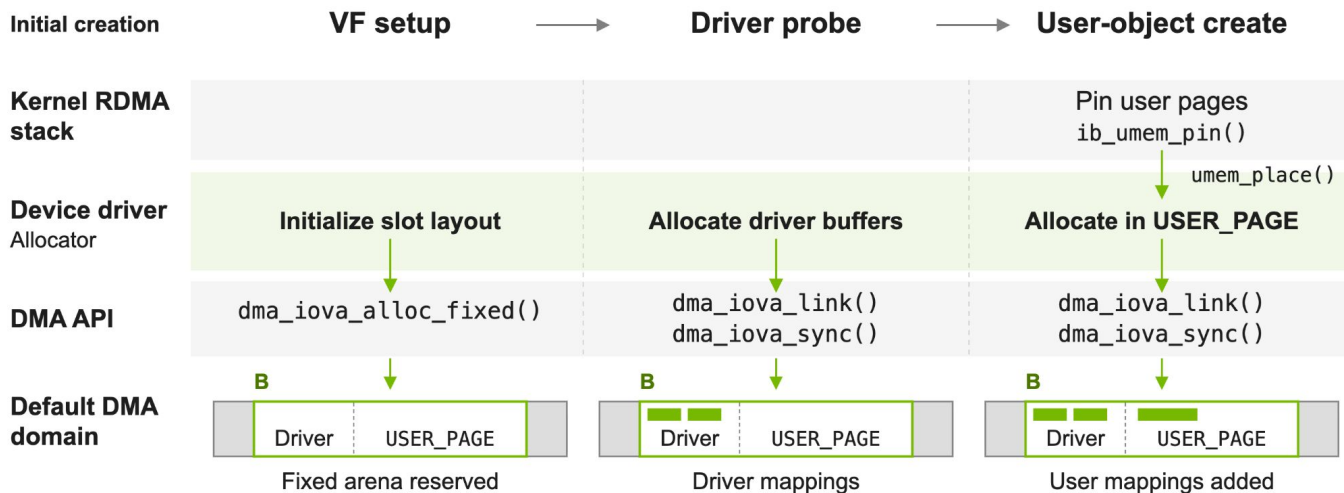
DMA-managed IOMMU domain



Current IOVA Allocator Implementation

With the DMA API extension we have the allocator working inside the DMA API's managed IOMMU domain

- Implements and uses `dma_iova_alloc_fixed()` API
- Allocator implemented inside device driver
 - User objects managed alongside kernel objects with their own slot
- New flag and callbacks added to struct `ib_device`
 - Direct `ib_umem` path calls into Device Driver allocator callbacks rather than default DMA API
- Questions: What should the carveout be? Should it be hardcoded? Chosen by userspace?



Potential Enhancement: PASID

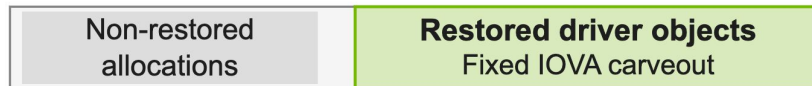
Can we simplify our design by having multiple address spaces?

- Give kernel RDMA stack it's own PASID
 - Driver creates PASID for its internal objects?
- Advantages:
 - No need to define an arena
 - Cleaner ownership split: RDMA stack / Driver
 - Scalability: Device can address larger range
- Disadvantages:
 - PASIDs must be managed across hosts
 - Requires HW support
- Shared Virtual Addressing (SVA):
 - HW uses process VA as DMA address
 - Maybe drop user object allocator?

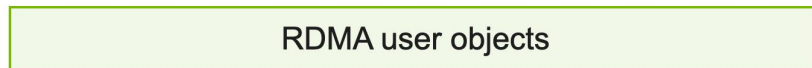
One NIC, multiple address spaces

A. Driver carveout + RDMA PASID

DMA-managed default (no PASID)



RDMA PASID (R)

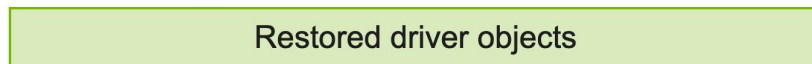


B. Separate driver and RDMA PASIDs

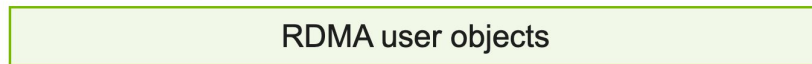
Default domain (no PASID)



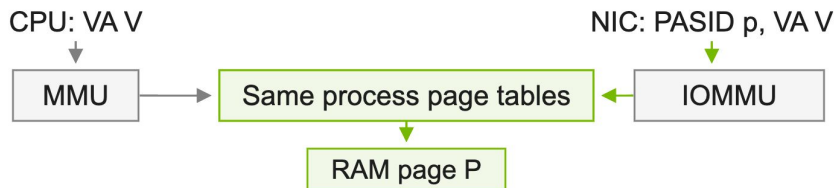
Driver PASID (D)



RDMA PASID (R)



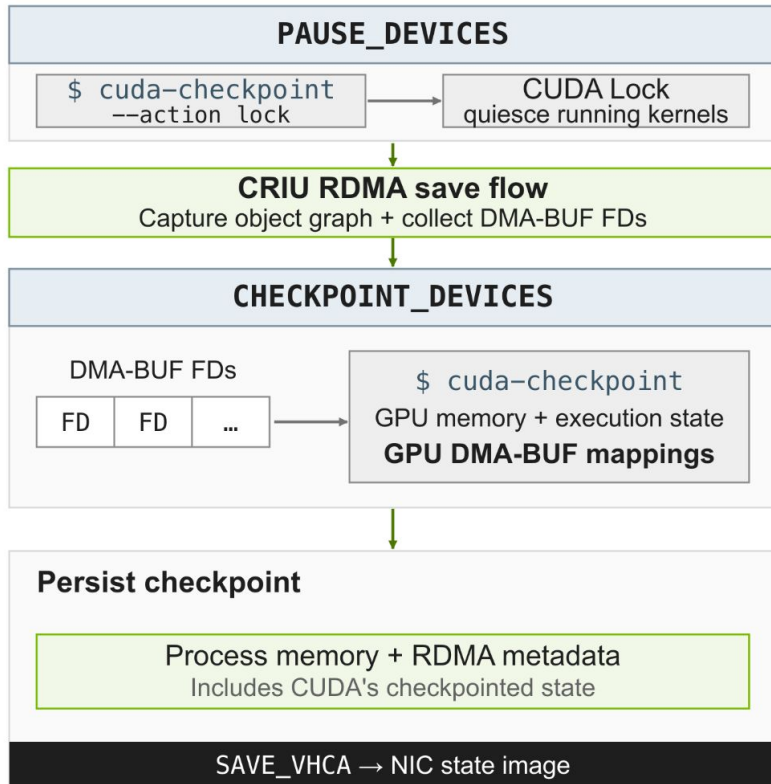
SVA: reuse process page tables



RDMA <> CUDA Checkpoint <> GPU DMABUF Integration

How we restore GPU-backed RDMA MRs

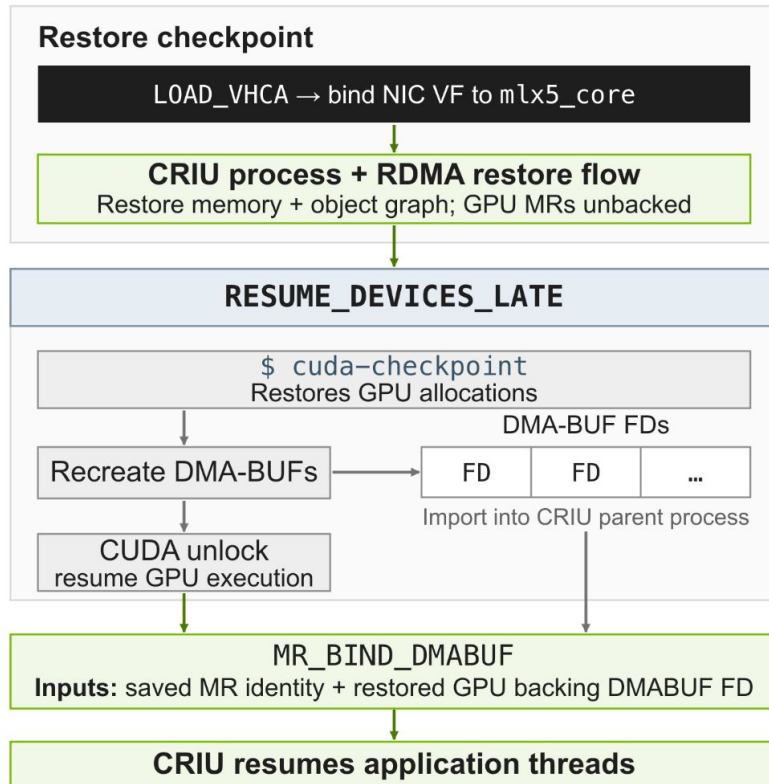
Checkpoint



□ CRIU / RDMA

□ CRIU plugin hooks

Restore



□ GPU operations

■ vHCA firmware

UAPI

/dev/mlx5_vfmig/<pf_bdf> character device, plus the new RESTORE UVERBs family

VIA	CALL	DEVICE OPERATION	PRECEDENT	EFFECT
PROVISION both hosts, VF unattached · orchestrator				
■ chardev	SET_TRACKED	— host only	new · mlx5	enable per-VF allocation tracking for checkpoint/restore
chardev	ENABLE_MIGRATABLE	SET_HCA_CAP · GENERAL_DEVICE2	mlx5_core devlink	migratable=1 — the firmware gate
■ chardev	SET_VF_UUID	— host only	new · mlx5	stamp a 16-byte identity, the orchestrator's choice
DUMP source, VF live · CRIU				
chardev	GET_VHCA_ID	QUERY_HCA_CAP · other_function=1	mlx5_vfio_pci	read the VF's vhca_id
■ chardev	QUERY_VF	QUERY_HCA_CAP · other_function=1	new · mlx5	read VF identity, restore and tracking status
uverbs	QUERY_MR	— reads kernel state	modified	add user_addr and access_flags
■ uverbs	VFMIG_QUERY_PD/CQ/QP	kernel state / optional QUERY_QP	new · mlx5	read per-object hardware identifiers and state
■ uverbs	VFMIG_QUERY_UCONTEXT VFMIG_QUERY_DYN_UARS	— reads kernel state	new · mlx5	static UAR state, or dynamic-UAR objects, according to context mode
uverbs	MR_EXPORT_DMABUF_FD	— host only	new · core	return a new fd for the MR's existing DMA-BUF
uverbs	MR_UNBIND_DMABUF	UMR: clear translations, disable mkey	new · core	detach backing memory, preserve MR keys
chardev	SUSPEND_VHCA	SUSPEND_VHCA	mlx5_vfio_pci	quiesce initiator and/or responder, selected by flags
chardev	SAVE_VHCA_STATE	query image size + SAVE_VHCA_STATE	mlx5_vfio_pci	return a read-only fd for the migration image
STAGE / BRING-UP destination · orchestrator and driver probe				
■ chardev	QUERY_VF	QUERY_HCA_CAP · other_function=1	new · mlx5	scan the PFs for the matching uuid
chardev	LOAD_VHCA_STATE	LOAD_VHCA_STATE — deferred to probe	mlx5_vfio_pci	stage the migration image through a write-only fd
■ chardev	MARK_RESTORED	— host only	new · mlx5	preserve restored device state during the next probe
chardev	RESUME_VHCA	RESUME_VHCA	mlx5_vfio_pci	resume both directions during probe in this CRIU path
RESTORE destination · CRIU				
uverbs	GET_CONTEXT + VFMIG_RESTORE flag	normal context setup skip fresh ALLOC_UAR	new flag	create the context in restore mode; re-use saved UAR identifiers
■ uverbs	VFMIG_RESTORE_UCONTEXT VFMIG_RESTORE_DYN_UARS	— none	new · mlx5	restore static UAR state, or dynamic-UAR objects and mmap bookkeeping
uverbs	RESTORE_PD	CREATE_MKEY + DESTROY_MKEY	new · core	adopt the saved PDN; validate with a temporary mkey
uverbs	RESTORE_MR/CQ/QP RESTORE_MR + DMABUF flag	adopt restored FW objects adopt mkey without backing memory	new · core	rebuild kernel objects and host-memory DMA mappings DMA-BUF flag selects an MR awaiting late bind
uverbs	MR_BIND_DMABUF	UMR: install translations, enable mkey	new · core	attach recreated DMA-BUF after CUDA restore, keep keys
chardev = /dev/mlx5_vfmig/<PF BDF>, per VF and PF-scoped				uverbs = the process's uverbs fd, per ucontext and per object

Acknowledgements

Key Contributions & Project Support



GPU & CUDA Checkpoint

Nikolay Aleksandrov (NVIDIA)

Grigori Inozemtsev (NVIDIA)

Substantial core contributions on GPU state checkpointing and CUDA support.



RXE Enhancements

Lucas Karpinski (NVIDIA)

Software RDMA transport driver enhancements and testing support.



Project Guidance

David Ahern (NVIDIA)

Overall strategic project guidance and architecture oversight.



AI-Enabled Effort

Development and execution were largely AI enabled.



GitHub Repositories

Linux:

<https://github.com/raphael-s-norwitz/linux/tree/criu-rdma-lpc-2026>

CRIU:

<https://github.com/raphael-s-norwitz/criu/tree/criu-rdma-lpc-2026>



Get in Touch

rnorwitz@nvidia.com

Reach out with questions or feedback!

Let's Discuss

Deterministic IOVA Allocator Design

01

DMA API Extension

Particularly interested in feedback and validation on the proposed DMA API extension design.

02

PASID Extension

Evaluating the relative importance, priority, and implementation timeline of the PASID extension.

03

Open Floor

Addressing any other fundamental concerns or architectural questions about the presented material.

The background features a series of overlapping, wavy green bands that create a sense of depth and movement. A solid green vertical bar is positioned on the far left side of the frame.

Appendix

Current Project Limitations

As far as we can tell none of these are be fundamental

Transport coverage: RC workloads validated; UD support incomplete/unvalidated; UC/RD unsupported

RDMA Connection Manager: no checkpoint/restore support

Event notifications: incomplete async-event FD state preservation and completion-event FD support

Additional objects: no Shared Receive Queue (SRQ) or Address Handle (AH) restore

On-Demand Paging (ODP): currently unsupported

Data Direct: currently unsupported

Three Ownership Models

Three possible ways to support RDMA SAVE/RESTORE with a VHCA

