La Région
Auvergne-Rhône-Alpes

vPIM : virtualized Processing-In-Memory

Jiaxuan Chen[1], **Dufy Teguia**[2], Oana Balmau[1], Stella Bitchebe[1], Alain Tchana[2]

ABUMPIMP Minisymposium, 2024/08/26

1. McGill University 2. Université Grenoble Alpes

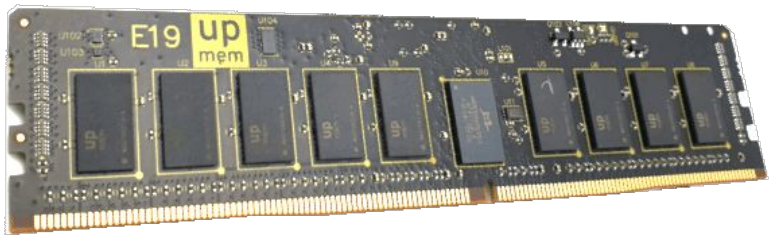
Goal

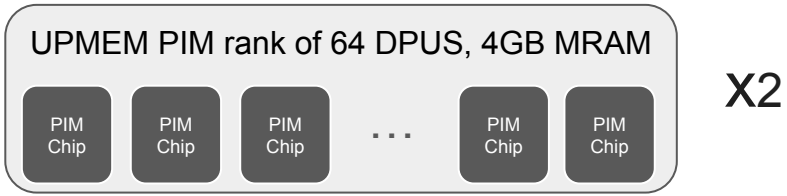
- PIM is a solution to solve the data movement problem in applications
- Commercial PIM hardware should be available in the cloud
- Virtualized devices can be shared by multiple Linux virtual machines

Terminology

Native/Virtualized environment: The environment is/is not virtualized

DIMM (Dual In-line Memory Module): Is a physical memory module with either one or 2 sides. UPMEM PIM DIMM has two sides, called ranks.





VMM: A process in which the virtual machine is being run

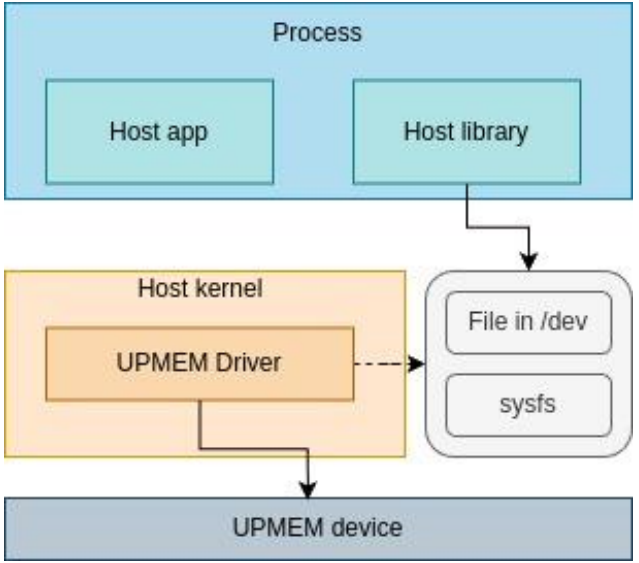
KVM: Linux Kernel Module for virtualization.

Native Application Execution Example

```
DPU_FOREACH(set, dpu, dpu_count) {
    dpu_prepare_xfer(dpu, &array[dpu_count * each_size]);
}
dpu_push_xfer(set, DPU_XFER_TO_DPU, DPU_MRAM_HEAP_POINTER_NAME, 0,
                  each_size * sizeof(uint32_t), DPU_XFER_DEFAULT);
```

Native Application Execution Example

```
DPU_FOREACH(set, dpu, dpu_count) {
    dpu_prepare_xfer(dpu, &array[dpu_count * each_size]);
}
dpu_push_xfer(set, DPU_XFER_TO_DPU, DPU_MRAM_HEAP_POINTER_NAME, 0,
                each_size * sizeof(uint32_t), DPU_XFER_DEFAULT);
```

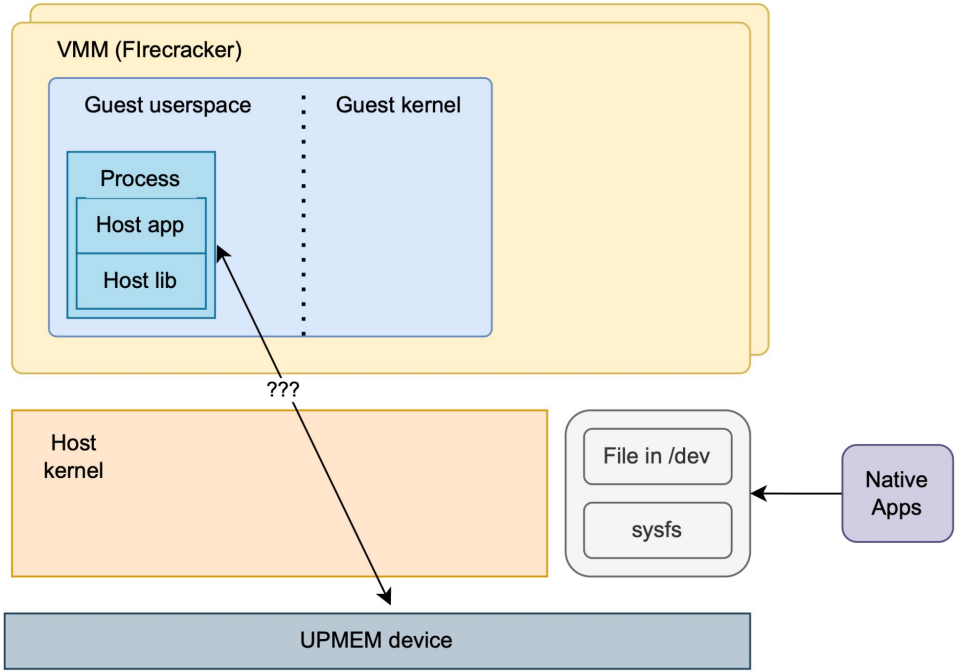


PIM Virtualization Challenges

How to **expose** the device to the VM?

How to let the VM **communicate** with the real hardware?

What **optimizations** can we do to the virtualized model?

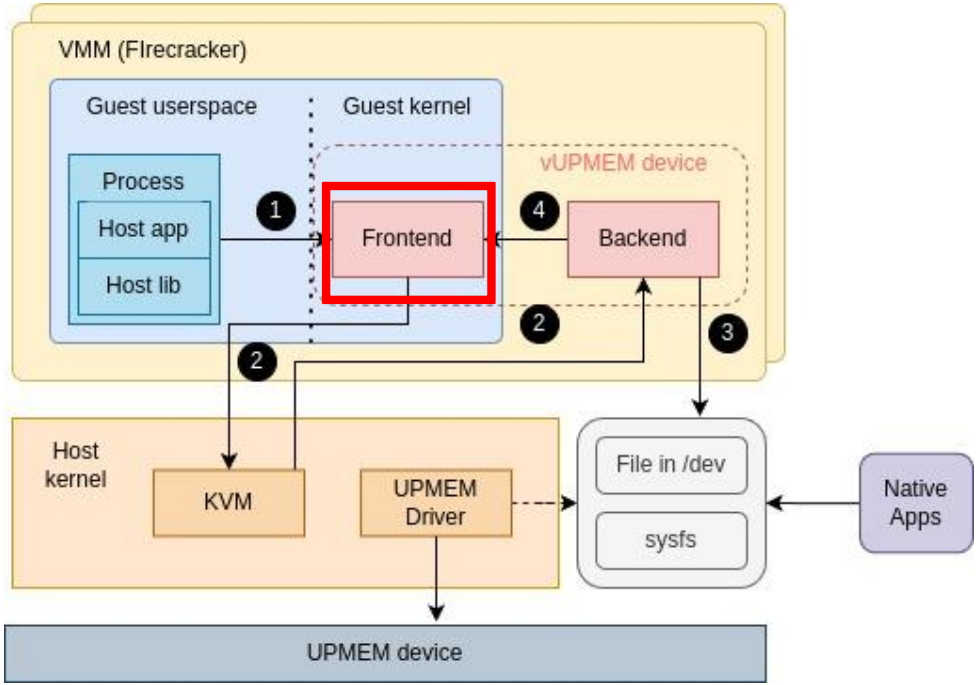


vPIM Architecture

Frontend Driver

- A kernel module in the guest OS
- Accepts requests from guest SDK
- Transfer requests through KVM

```
DPU_FOREACH(set, dpu, dpu_count) {
    dpu_prepare_xfer(dpu, &array[dpu_count * each_size]);
}
dpu_push_xfer(set, DPU_XFER_TO_DPU, DPU_MRAM_HEAP_POINTER_NAME, 0,
                each_size * sizeof(uint32_t), DPU_XFER_DEFAULT);
```

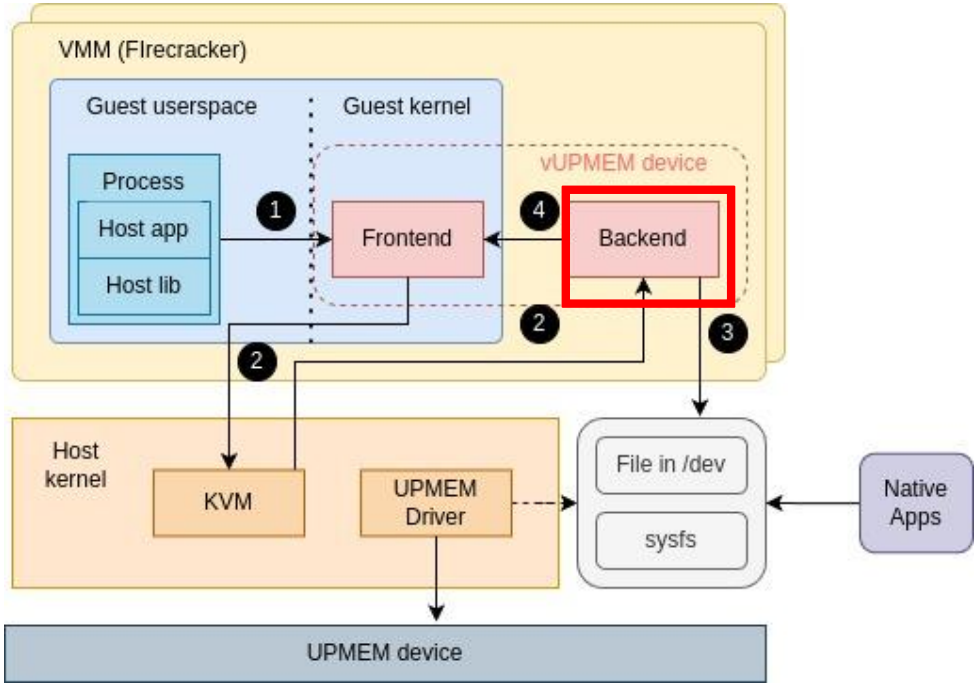


vPIM Architecture

Backend Device

- A module in the VMM (Firecracker)
- Accepts requests from frontend driver
- Perform operation via physical device

```
DPU_FOREACH(set, dpu, dpu_count) {
    dpu_prepare_xfer(dpu, &array[dpu_count * each_size]);
}
dpu_push_xfer(set, DPU_XFER_TO_DPU, DPU_MRAM_HEAP_POINTER_NAME, 0,
                each_size * sizeof(uint32_t), DPU_XFER_DEFAULT);
```



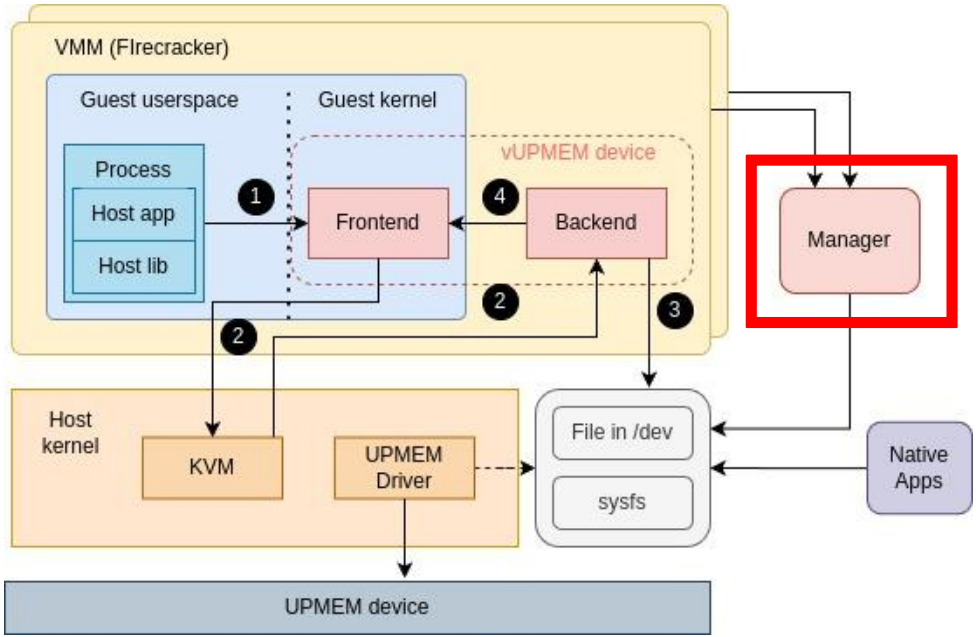
vPIM Architecture: Rank Manager

Device Manager

- A host program that allocates available physical ranks to virtual machines.

Characteristics

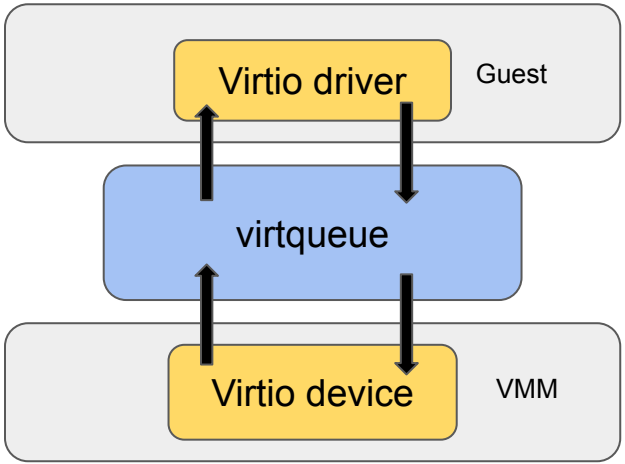
- Ensure ranks are allocated in a controlled manner for VMs or other applications on host.
- Ensures isolation between VMs.



Frontend to Backend: Virtio Transport layer

Virtqueue is a queue which:

- Is part of guest memory, registered in the KVM, accessible by the VMM.
- Saves the addresses of the guest buffers**.**
- The VMM can access these buffers by **address translation**.



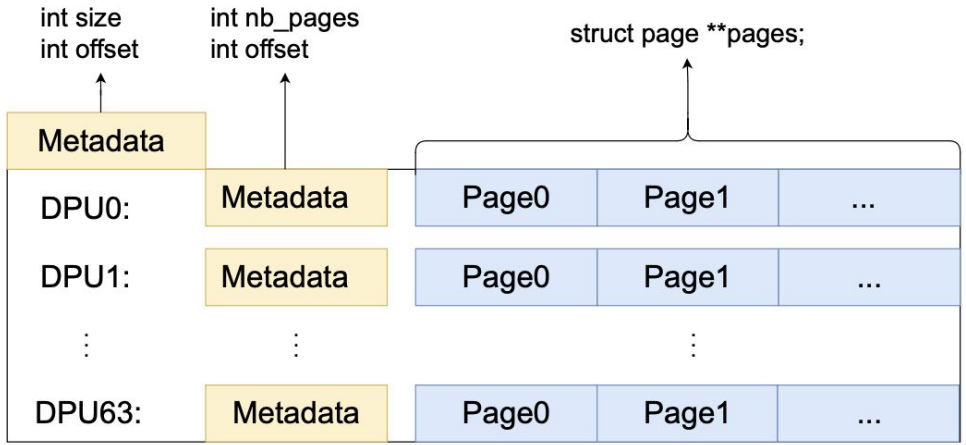
Frontend to Backend: Rank Operations

Here we use **rank reading/writing** as an example to explain request transfer in the frontend.

Data transfer to the MRAM of DPU is formatted as **a matrix of pages.**

```
struct dpu_transfer_mram {
    struct xfer_page *xferp[64];
    uint32_t offset_in_mram;
    uint32_t size;
};

struct xfer_page {
    struct page **pages;
    unsigned long nb_pages;
    int off_first_page;
};
```



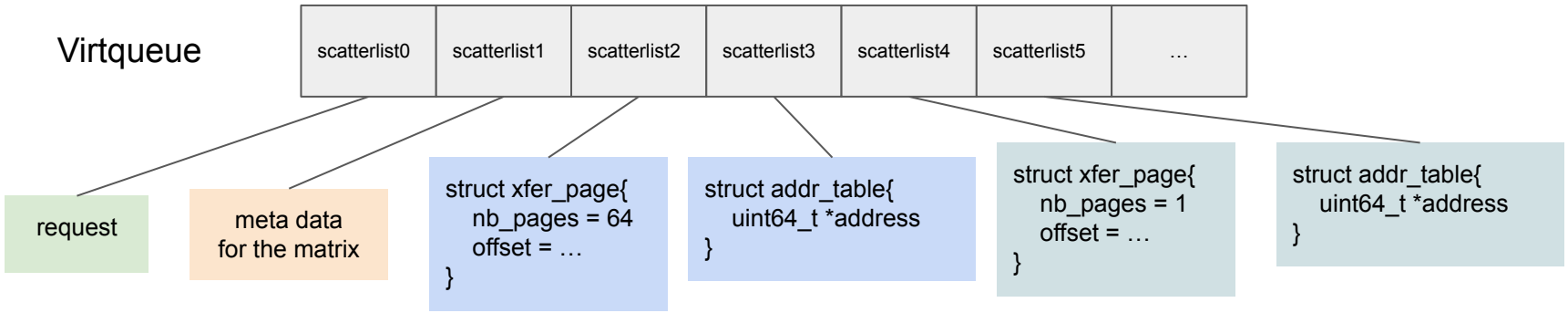
One xfer_page can contain: (mram_size)/(page_size) = 16384 pages

Frontend to Backend: Matrix Serialization

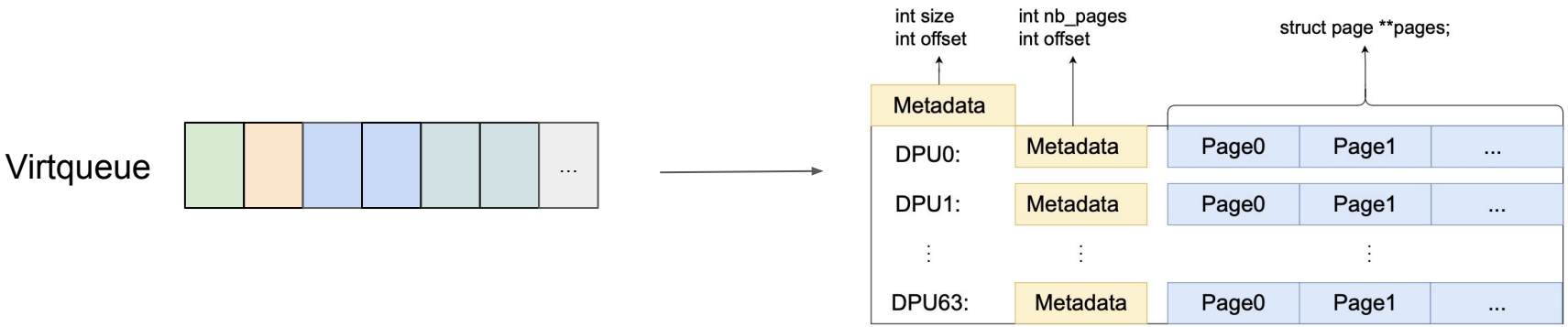
To **avoid direct memory copy:**

Linux pages are converted to physical addresses in the VM (**Guest Physical Addresses**).

VMM can access the memory region of these addresses by address translation.



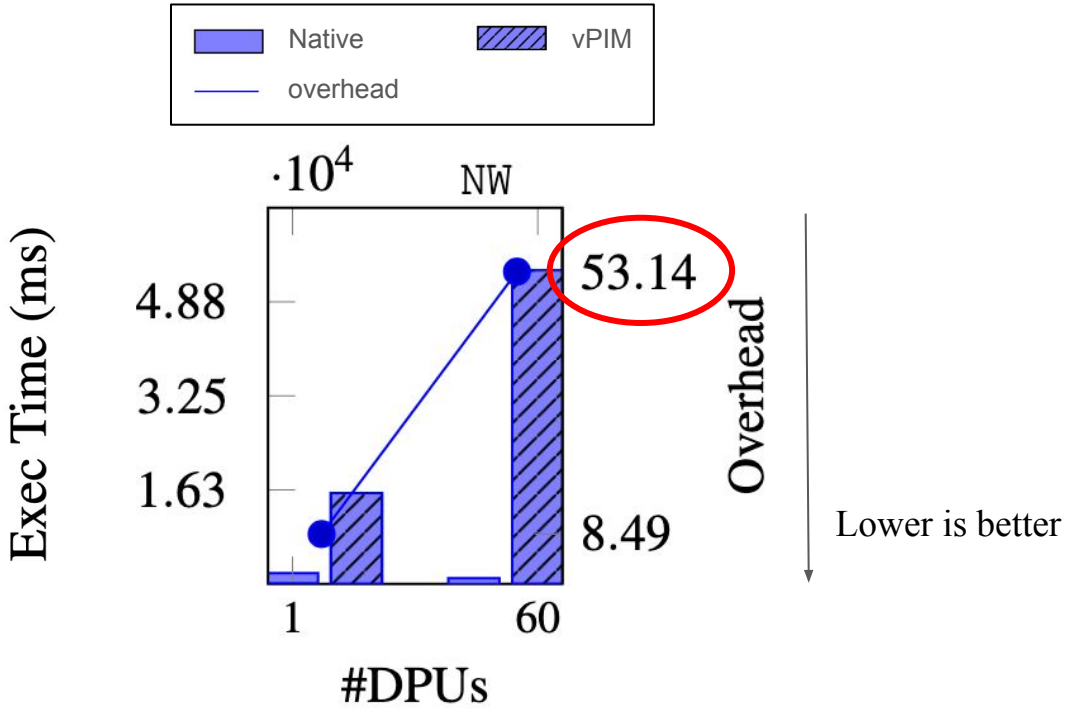
Backend Operates Host Device



vUPMEM Characteristics:

- Device attributes (mram size, serial number…) read from sysfs
- Physical rank device memory mapping
- Operations that are performed as a request (or not) of the Guest VM

Problem: Small-size Data Transfers



Problem: Small-size Data Transfers

→ Each requests requires a **VMEXIT**, no matter data transfer size.

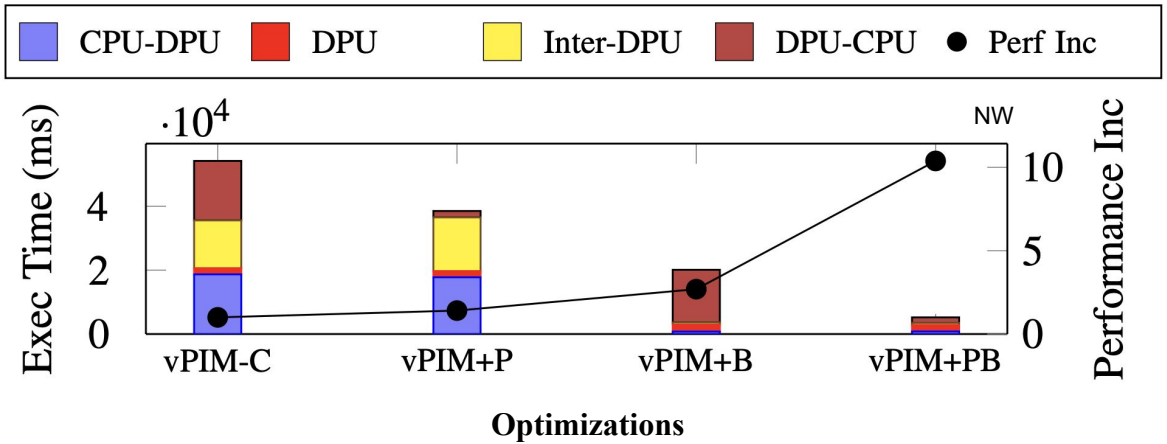
→ In small-size data transfers, **communication dominates** the execution time.

Frequent small-size data transfer results in **repetitive VMEXIT,**

causing a **significant performance bottleneck**.

Optimizations: Small-size Data Transfers

- **Prefetch Cache** proactively caches larger segments when the requested data is too small.
- **Request Batching** buffers small-size write-to-rank requests, which are then collectively flushed to the backend.



Evaluation

Benchmark: We evaluate our system using the PrIM Benchmarks, using the strong scaling configuration

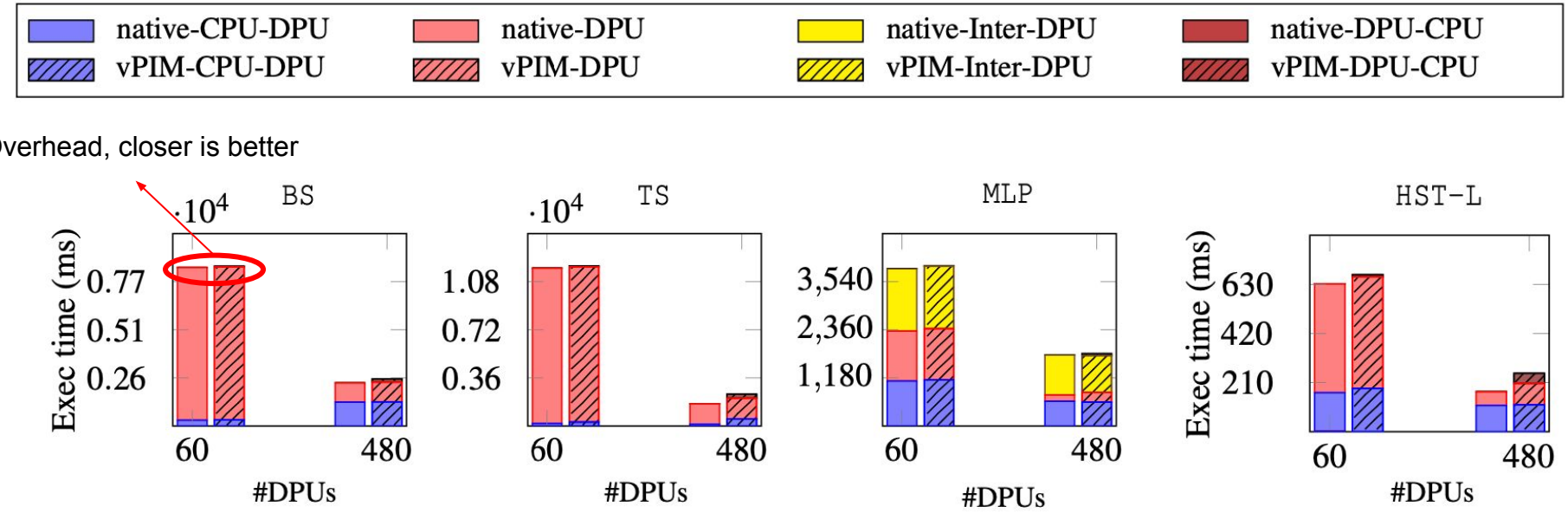
Metric: **Execution time** and virtualization **overhead** compared to the native.

Config: 1 rank (60 DPUs) and 8 ranks (480 DPUs)

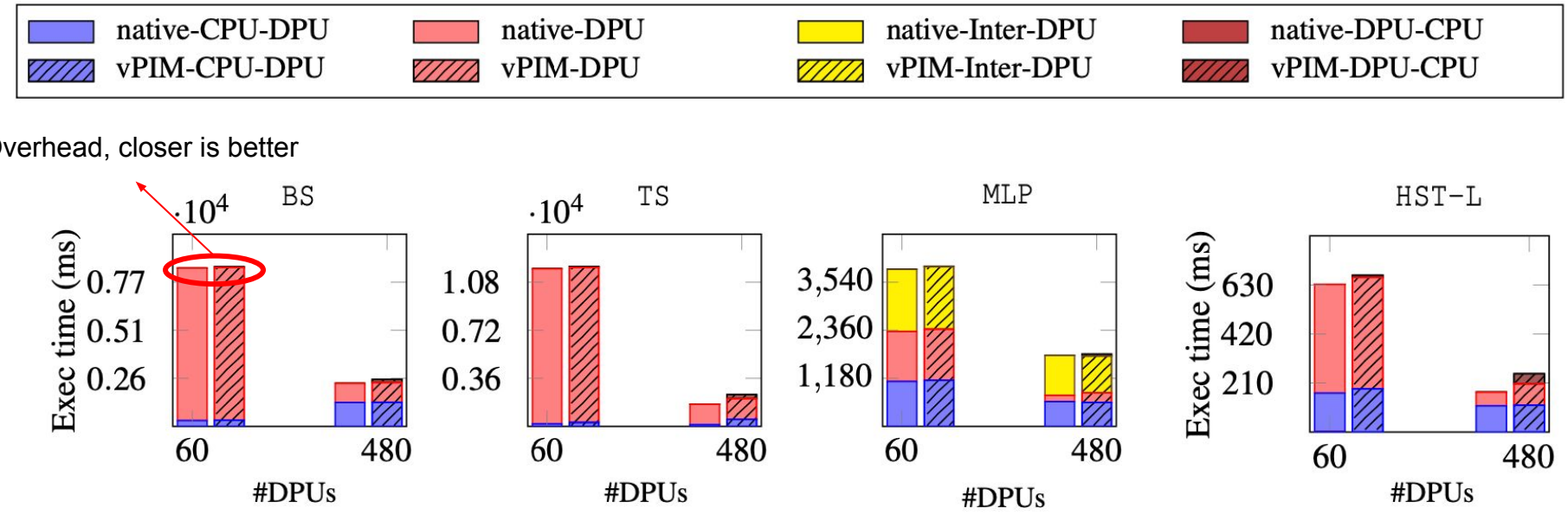
Table 1. Selected PrIM Benchmark Applications

BS	Binary Search
TS	Time Series Analysis
MLP	Multilayer Preceptron
HST-L	Image Histogram
TRNS	Matrix Transposition
NW	Needleman-Wunsch

Native vs vPIM Execution

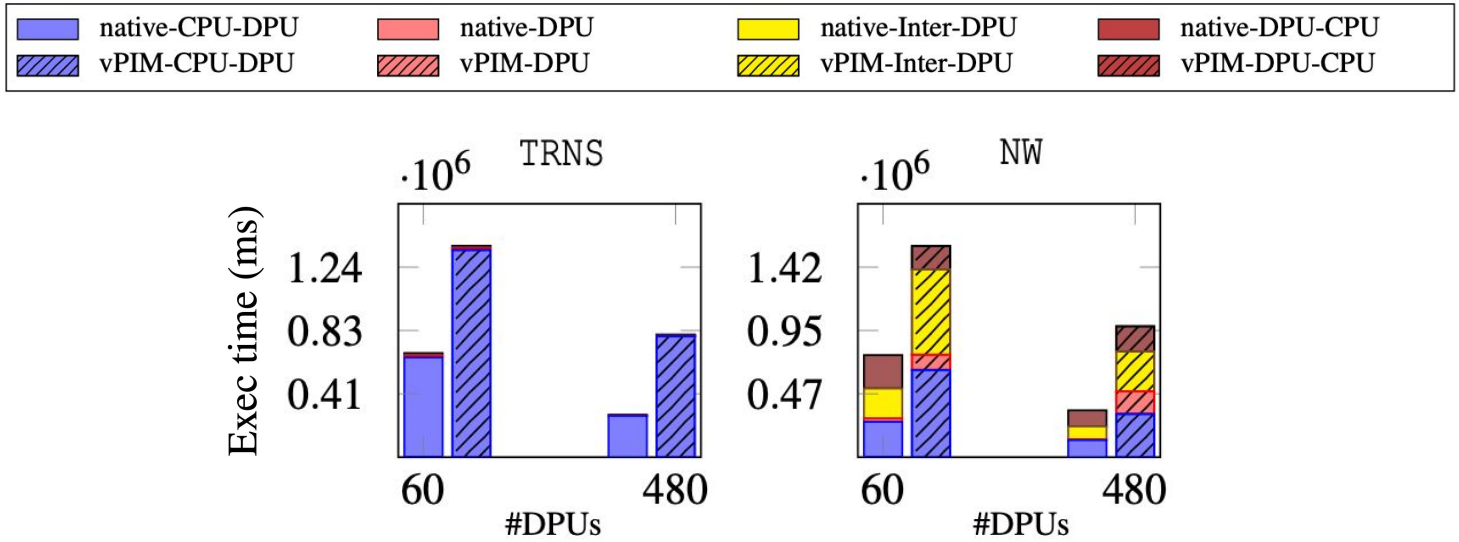


Native vs vPIM Execution



The overhead of vPIM is up to 1.15 in suitable applications

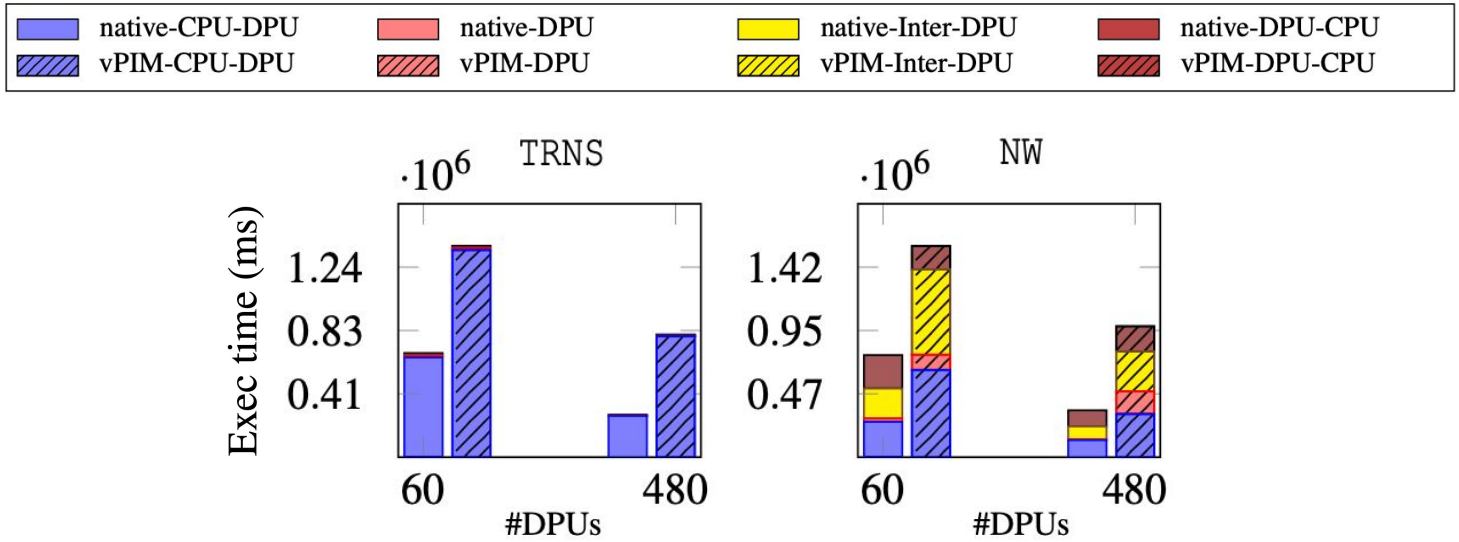
Native vs vPIM Execution



NW: 65000 160-byte transfer operations.

TRNS: 980000 512-byte transfer operations.

Native vs vPIM Execution



NW: 65000 160-byte transfer operations.

TRNS: 980000 512-byte transfer operations.

Rewritten NW Applicaton is only 1.2x slower than the native

Takeaways

- vPIM is a solution for **UPMEM PIM virtualization**. Applications can **run unmodified**.
- Overhead is low for apps that have **dominating DPU execution time** and **do not have frequent small-size data transfers**.
- For the other scenarios, vPIM aggregates data transfers with **batching and prefetching**.

Thank you!

Contact the authors for more information !

brice.teguia-wakam@univ-grenoble-alpes.fr
jiaxuan.chen2@mail.mcgill.ca

Funded by the PAI2021 "Fault tolerance for Disaggregated Rack-Scale Computing", and the Natural Sciences and Engineering Research Council of Canada

McGill DISCS Lab

UGA LIG Lab

APPENDIX

Evaluation: PrIM Benchmarks

16 applications are evaluated in total.

60 DPU config:

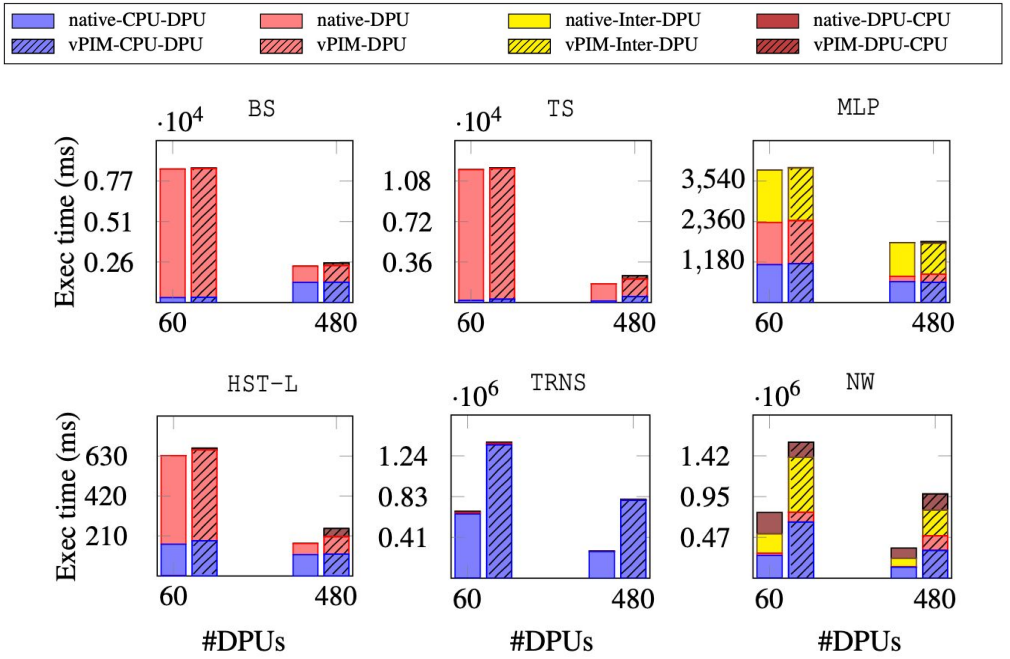
Overhead: 1.01x (BS) to 2.07x (NW),

averagely 1.24x

480 DPU config:

Overhead: 1.02x (MLP) to 2.89 (TRNS),

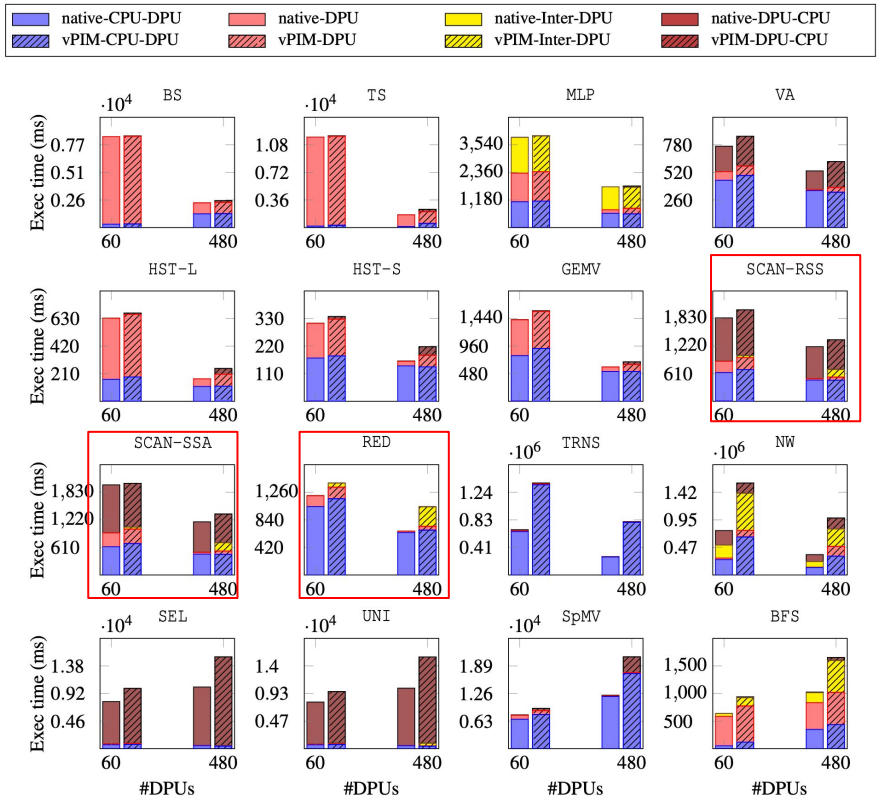
averagely 1.54x



Evaluation: PrIM Benchmarks

Observation 2: Significant overhead in the Inter-DPU step of SCAN-RSS, SCAN-SSA and RED.

This is a scenario showing the **drawback of the Prefetch Cache**.



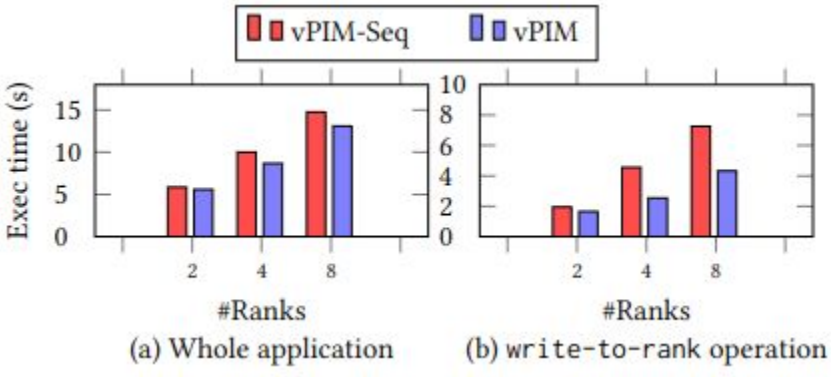
Optimizations: Rank operations overhead

Problem 1:

→ Firecracker is written in Rust. Which is slower than C for rank operations

Problem 2:

→ The Firecracker event manager handles requests one by one



Frontend: forward from guest to the backend

Frontend Driver

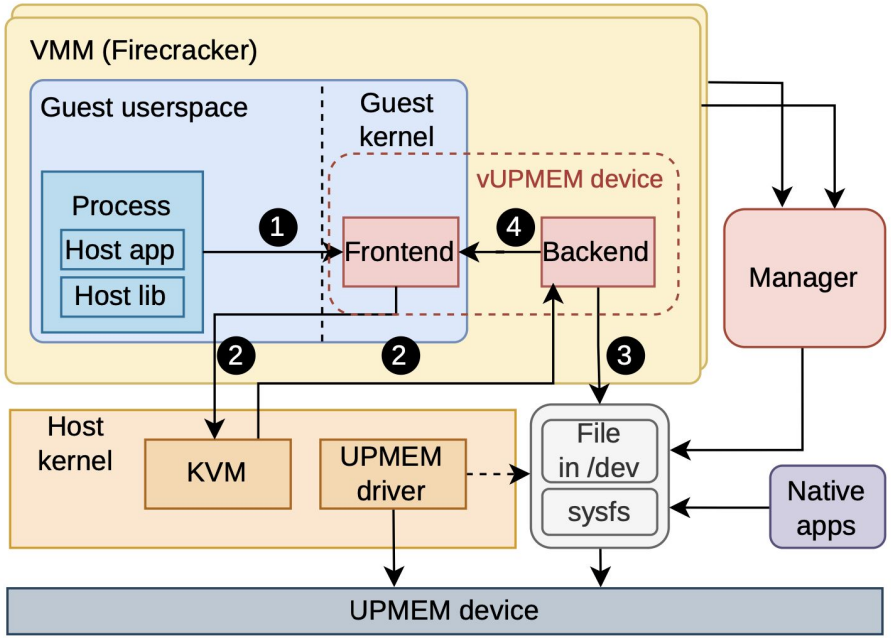
- A kernel module in the guest OS
- Accepts requests from guest SDK
- Transfer requests to backend driver

① SDK ⟷ Frontend Driver:

The Frontend Driver exposes the device to the guest userspace and receive request from the SDK

② Virtio:

The Frontend sends requests to the backend by generating an event in the KVM following the **virtio** specification.



Backend: Execute the request on real hardware

Backend

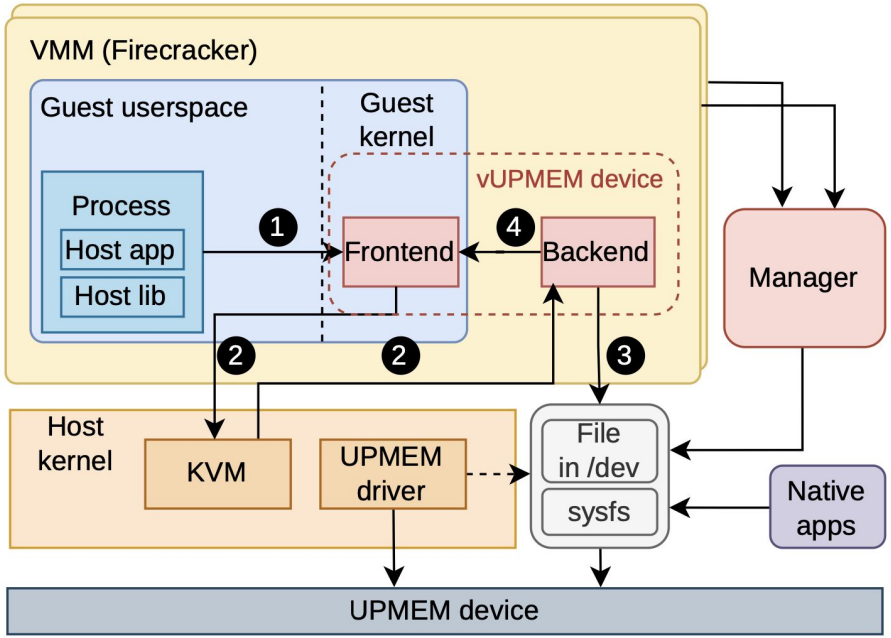
- A module in the VMM (Firecracker)
- Accepts requests from frontend driver
- Perform operation via physical device

③ Backend ←→Host Device:

The Backend controls the UPMEM hardware using direct memory access.

④ Backend writes Guest memory:

The Backend writes the results of the requests to the memory of the Frontend via the physical address sends by the Frontend.

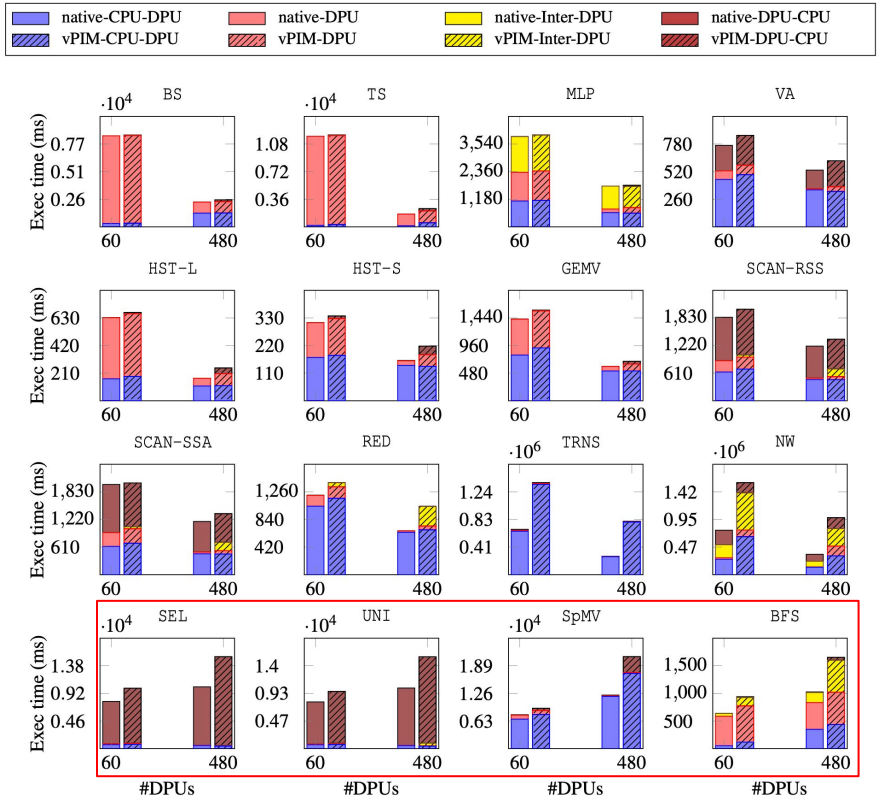


Evaluation: PrIM Benchmarks

Observation 2: For 4 in 16 applications, execution time increases with more DPUs both in vPIM and native.

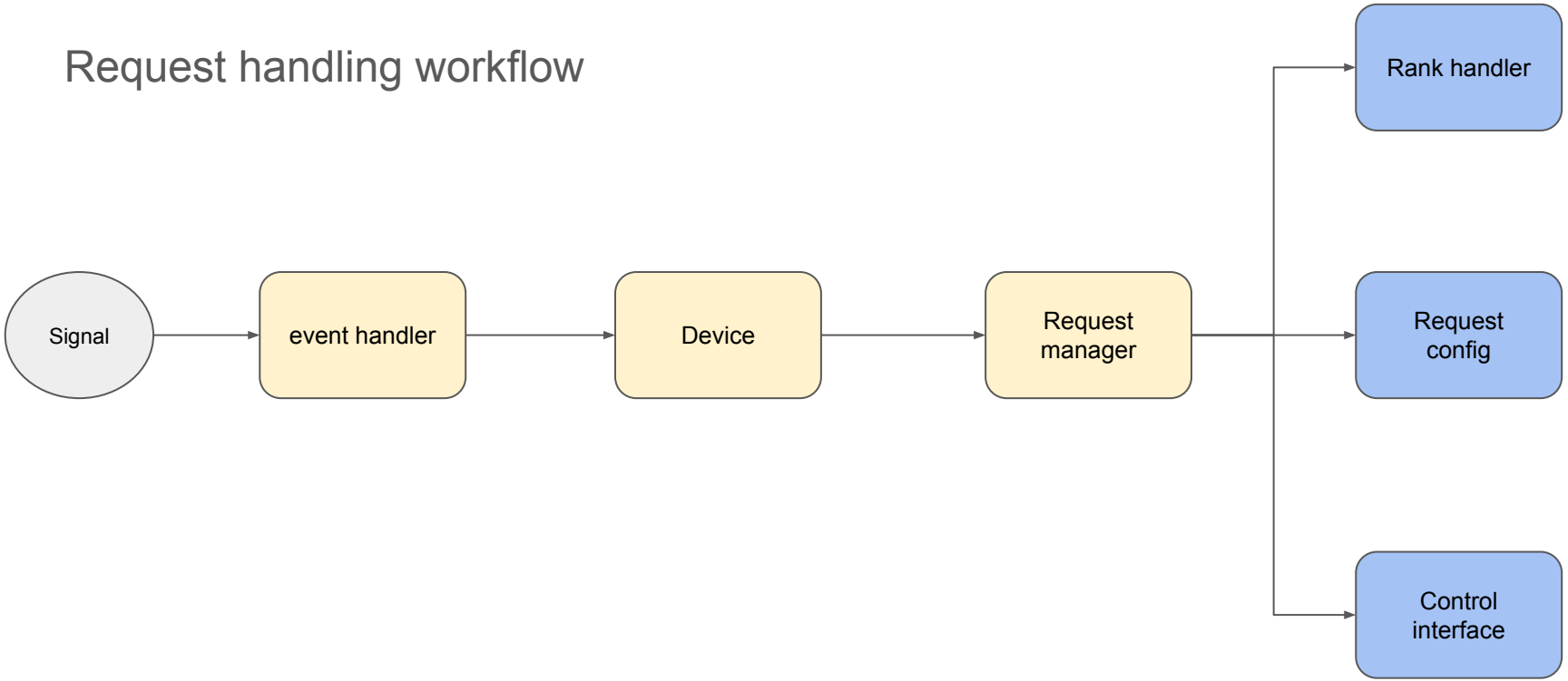
These applications use **serial data transfer**, which leads to increase data transfer time.

In addition, serial data transfer cannot fully benefit from the optimization methods: request batching, prefetch cache and multithreading handling, resulting in higher overhead.

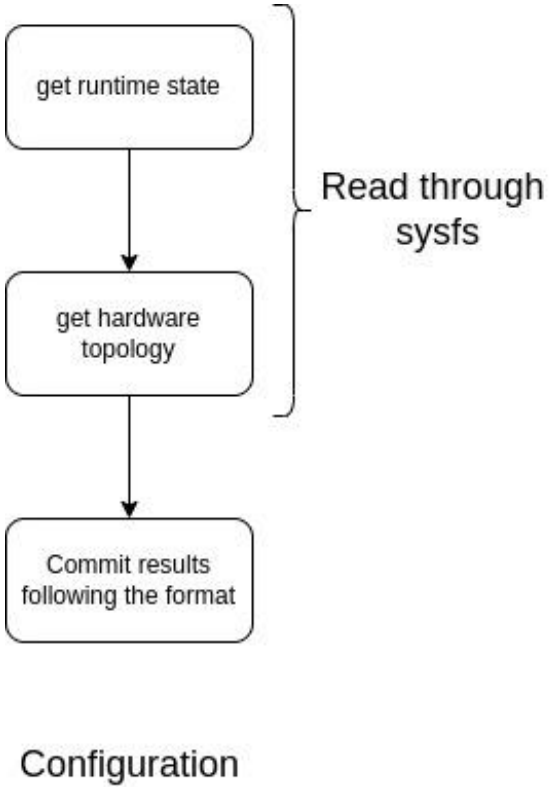


Firecracker Handlers: Handle Request

Request handling workflow



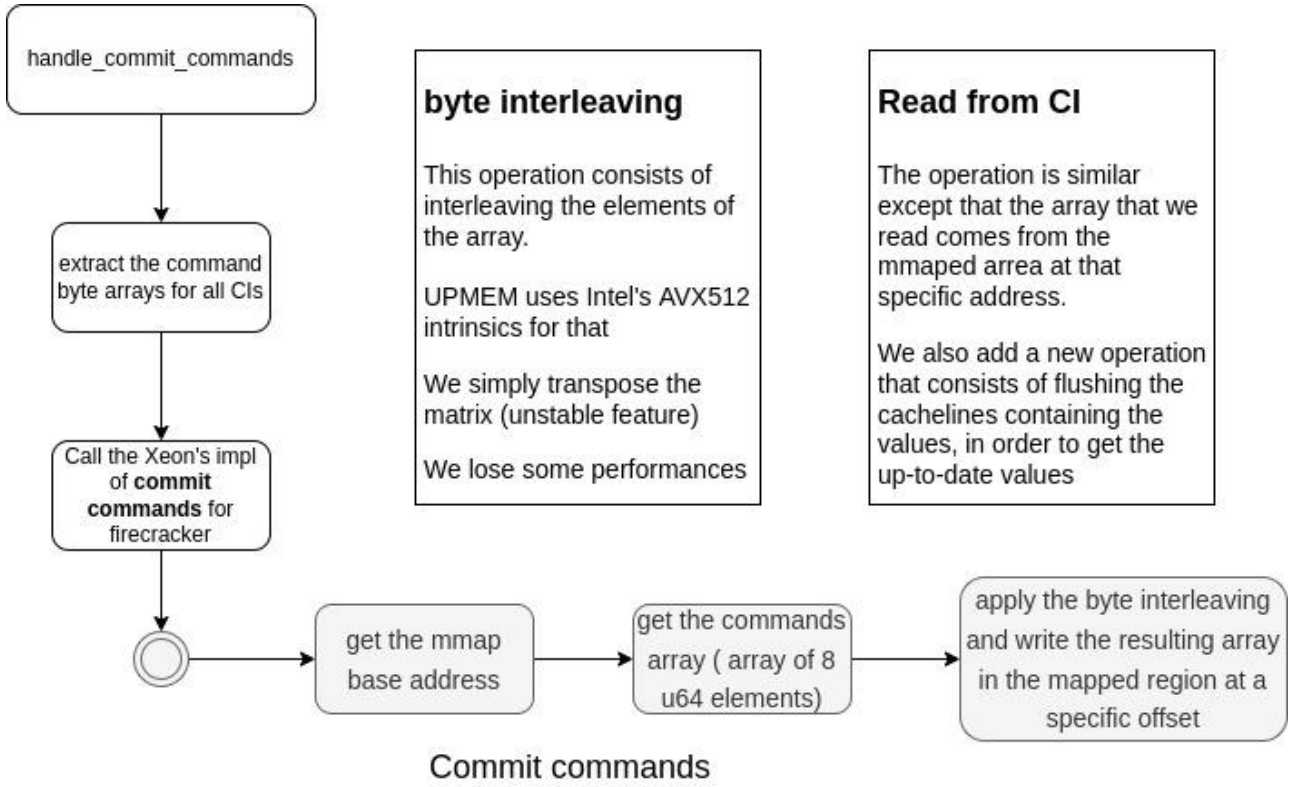
Firecracker Handlers: More on Backend - Request Config



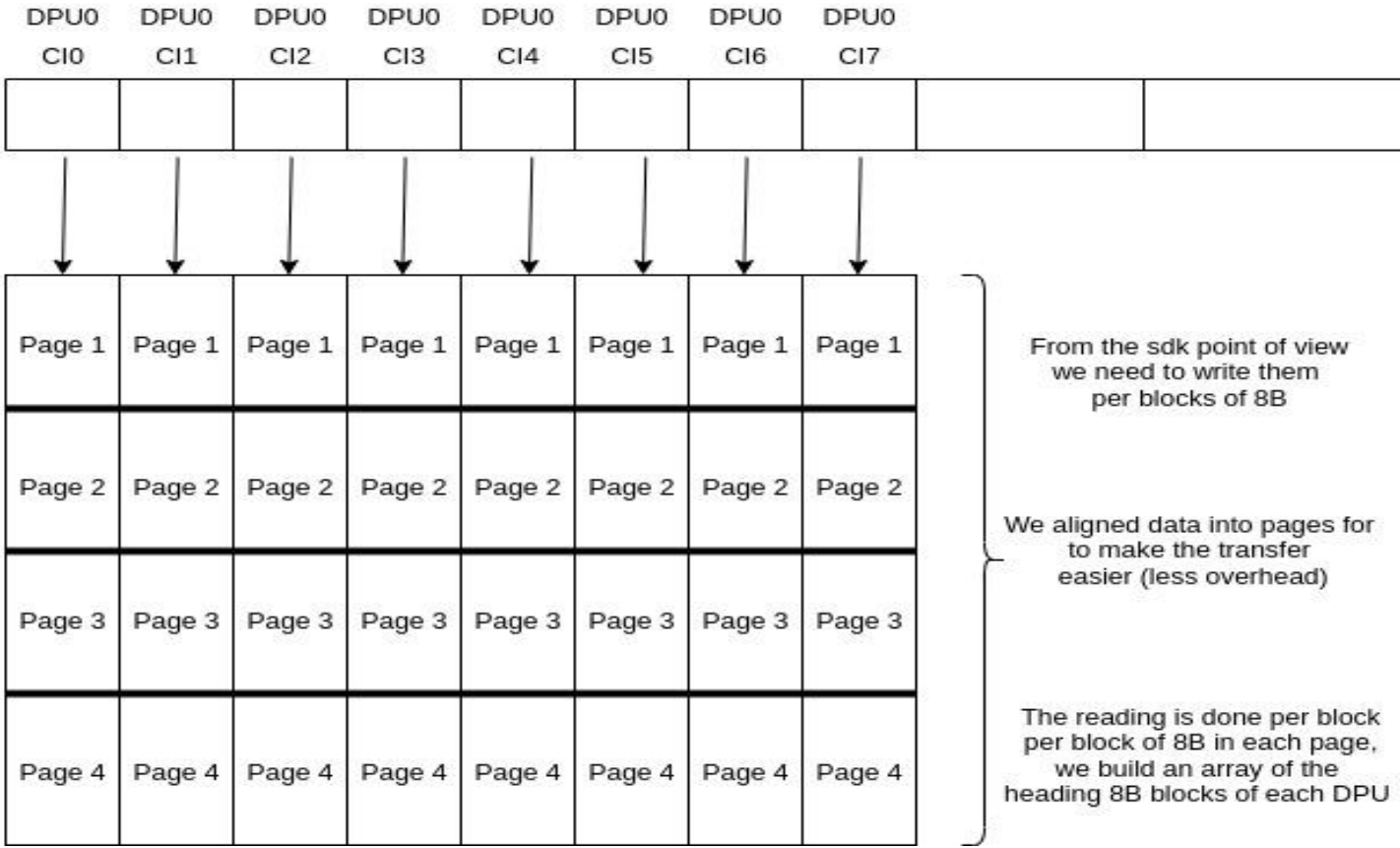
- The Backend here just sends the information that has been gathered during the device initialization (from sysfs)
- These are sent to configure the driver and expose the same informations to the guest machine for a seamless usage

Firecracker Handlers: More on Backend - CIs

- We are using SIMD AVX2 (not AVX512)
- We use non temporal stores to bypass the cache when writing data
- We make a volatile read on CI data



Firecracker Handlers: More on Backend - transfer matrix



Firecracker Handlers: More on Backend - Write to rank

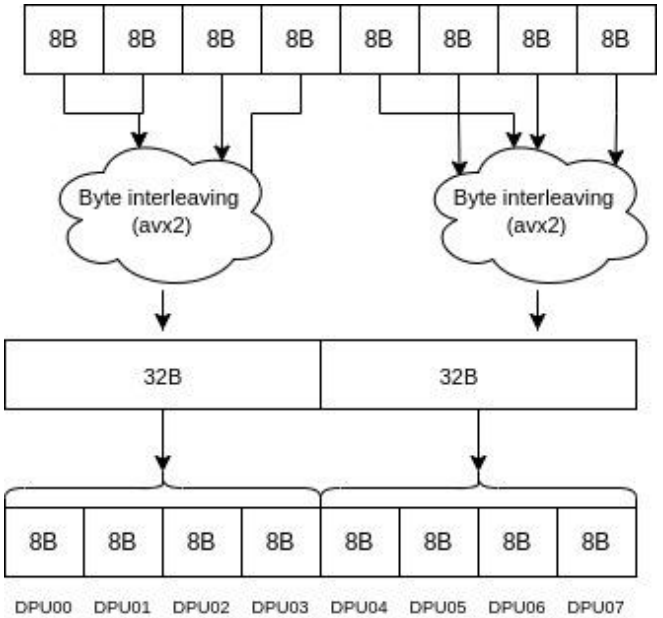
The write to rank operation consists of getting a transfer matrix from the Guest main memory and then write then down in the rank MRAM.
This is done using 8 DPUs (1 DPU per CI) of the same index (from 0 to 7) per loop. In each loop we do :

- We set the current page to be written in the rank
- We perform a write for each 8 bytes-blocks per CI
- Before committing the write, we perform a byte interleaving (avx2) to fit memory requirements
- We do this until we reach the number of pages and the amount of data

NB : We use non temporal stores to bypass the cache

Firecracker Handlers: More on Backend - Write to rank

- The write to rank operation consists of getting a transfer matrix from the Guest main memory and then write then down in the rank MRAM.
- The figure presents how data for 8 CIs (8 bytes each) are written



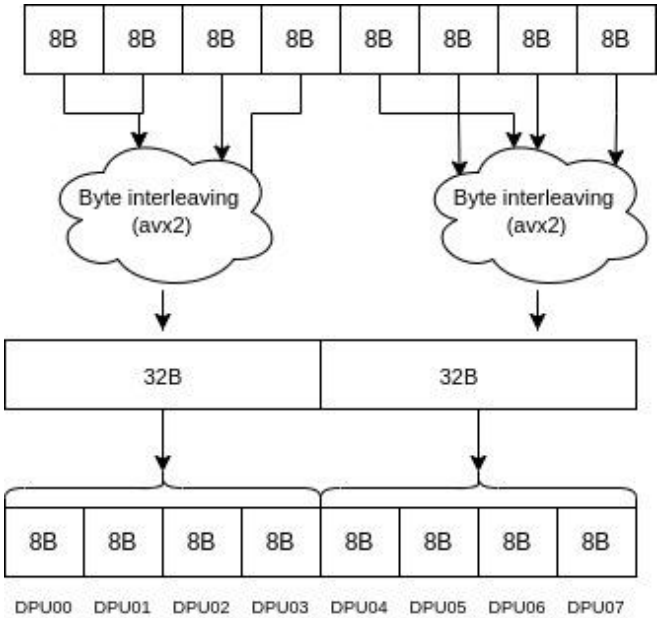
NB: DPU0x means DPU of index 0 in the xth CI

NB : The goal behind byte interleaving is to be able to write one byte at 8 different places in one single operation (avx512) or two operations (avx2)

Firecracker Handlers: More on Backend - Read from rank

The read from rank operations follows the same pattern except that we read data per 8 bytes-blocks

The figure presents how data for 8 CIs (8 bytes each) read



NB: DPU0x means DPU of index 0 in the xth CI

② Virtio Transport layer: CI Operations

Control Interface:

Control interface (CI) is a array of uint64_t of size 8

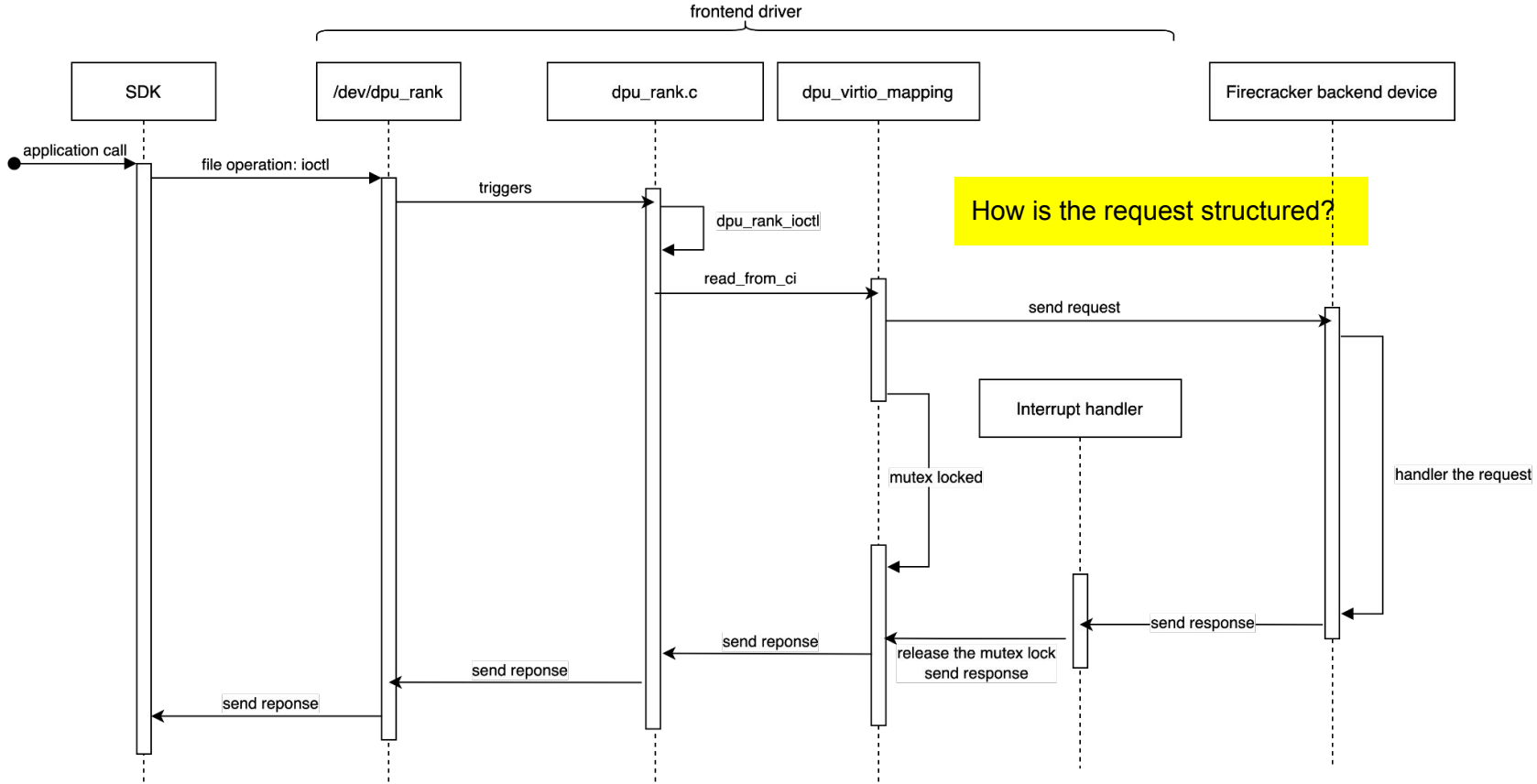
There are two operations can be called by the UPMEM SDK

write_to_cis: write command/program to the dpu

read_from_cis: retrieve the current CIs content from the hardware

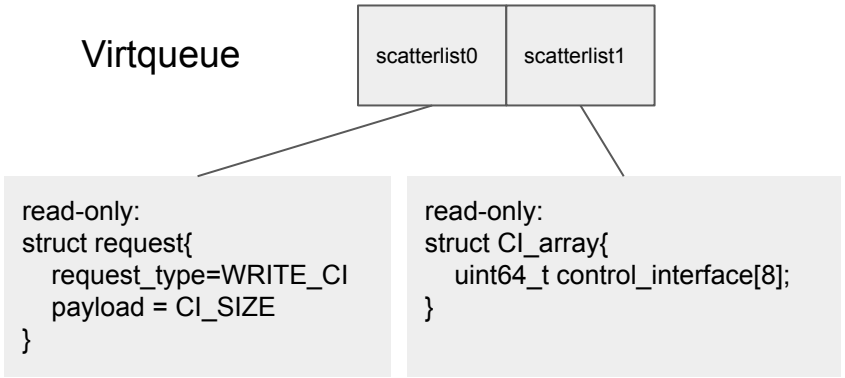
The Frontend Driver forwards these operations to the backend for further processing.

② Virtio Transport layer: CI Operations

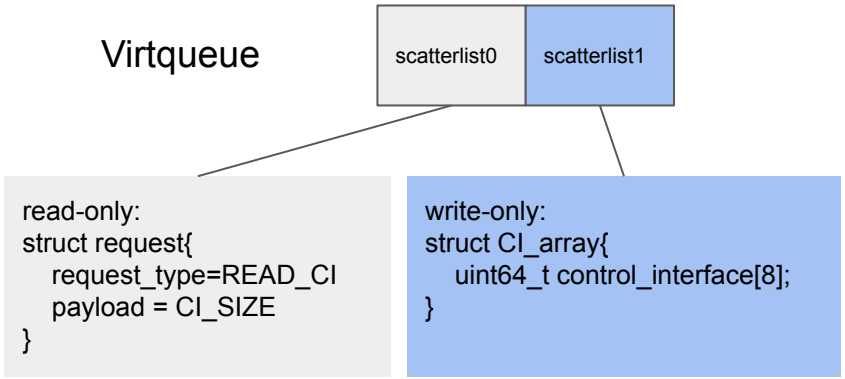


② Virtio Transport layer: CI Operations

Write to CI:



Read from CI:



Rank Manager

Workflow

- A VM request a rank available
- Manager checks either a free rank* to attach to the running VM
- The manager detects when a rank is freed and resets its content.
- The currently implemented algorithm is round robin

Next step

- Ranks cannot be shared at the DPU granularity for the moment.
- There is another ongoing work that aims to improve application colocation within the same rank.