vPIM: Processing-in-Memory Virtualization

Dufy Teguia, Jiaxuan Chen, Oana Balmau, Stella Bitchebe, Alain Tchana

vPIM: Processing-in-Memory Virtualization

Dufy Teguia*
dufy.teguia@orange.com
University of Grenoble Alpes
France

Jiaxuan Chen*
jiaxuan.chen2@mail.mcgill.ca
McGill University
Canada

Stella Bitchebe
stella.bitchebe@mcgill.ca
McGill University
Canada

Oana Balmau
oana.balmau@mcgill.ca
McGill University
Canada

Alain Tchana
alain.tchana@grenoble-inp.fr
Grenoble INP
France

ABSTRACT

Data movement is the leading cause of performance degradation and energy consumption in modern data centers. Processing in-memory (PIM) is an architecture that addresses data movement by bringing computation inside the memory chips. This paper is the first to study the virtualization of PIM devices by designing and implementing `vPIM`, an open-source UPMEM-based virtualization system for the cloud. Our `vPIM` design considers four requirements: Compatibility such that no hardware and no hypervisor changes are needed; Multiplexing and isolation for a higher utilization ratio; Utilizability and transparency such that applications written for PIM can be efficiently run out-of-the-box, leading to rapid adoption; Minimalization of virtualization performance overhead.

We prototype `vPIM` in Firecracker, expanding the `virtio` standard. Our experimental evaluation uses 16 applications provided by PrIM, a recent PIM benchmark suite. The virtualization overhead is between 1.01× and 2.07× for untouched PrIM applications. To keep overhead low, `vPIM` introduces several optimizations: zero-copy from guest OS to Firecracker, efficient virtio queues management, efficient Guest Physical Address to Host Virtual Address translation, parallel processing on multiple ranks, automatic data batching and pre-fetching, and the reimplementation of some specific functionalities in C instead of Rust. We hope this work will lay the foundation for future research on PIM for cloud computing.

CCS CONCEPTS

• **Software and its engineering → Virtual machines**; • **Hardware → Dynamic memory**.

KEYWORDS

VirtIO, Processing In Memory, Data transfer, DRAM Processing Unit, Virtualization

ACM Reference Format:
Dufy Teguia*, Jiaxuan Chen*, Stella Bitchebe, Oana Balmau, and Alain Tchana. 2024. vPIM: Processing-in-Memory Virtualization. In *25th International Middleware Conference (MIDDLEWARE '24), December 2–6, 2024, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3652892.3700782

MIDDLEWARE '24, December 2–6, 2024, Hong Kong, Hong Kong

1 INTRODUCTION

Modern applications such as health monitoring, augmented/virtual reality, transportation, and recommender systems, require rapid processing of large amounts of data [9, 13, 23]. Data movement is the main cause of performance degradation and energy consumption in modern machines [19, 34]. Near-data processing [14, 29, 46, 54] is an alternative to address the data movement problem in the traditional compute-centric model. The main idea is placing compute inside memory chips or in the memory controllers, virtually eliminating the data movement between memory and the main compute units (e.g., CPU, GPU, and other ASICs). The Processing In-Memory (PIM) architecture follows the near-data processing principles. While PIM is not a new idea [11, 18, 21, 37, 48, 57], emerging commercial devices promise to make PIM practical and easy to deploy. UPMEM [5] is the first commercial PIM device that commodity servers can use out-of-the-box.

This progress in hardware has recently spurred the attention of the research community and the industry. The characterization of PIM DPUs and their comparison with CPUs and GPUs have been extensively researched by prior work [25, 27, 31, 39, 42]. In this paper, in contrast, we focus on PIM virtualization. Virtualization is one of the key building blocks in cloud computing, as it provides flexible resource consumption and efficient resource utilization through resource sharing. To facilitate large-scale adoption of PIM, its virtualization is crucial [32].

Currently, users looking to leverage PIM devices must reserve an entire server with a fixed number of devices. However, many applications do not require the full capacity of such a server, leading to underutilization of these PIM resources. Virtualization addresses this inefficiency by allowing applications to dynamically allocate the exact number of PIM devices they need, optimizing resource usage and enabling more flexible and cost-effective deployment.

This paper studies the virtualization of PIM devices for the first time by designing and implementing `vPIM`, an open-source UPMEM virtualization system for the cloud. We consider the following requirements when designing `vPIM`. (R_1) Compatibility: `vPIM` requires no hardware and no hypervisor changes to allow its rapid adoption by cloud providers. (R_2) Multiplexing: `vPIM` allows UPMEM utilization by several VMs in an isolated manner for a higher utilization ratio. (R_3) Utilization and Transparency: `vPIM` is application transparent, allowing cloud users to deploy and use from inside the

*The authors have made equivalent contributions to the work

VM effortless and does not require application changes. (R_4) Performance: vPIM minimizes virtualization overhead. To satisfy $R_1 - R_3$, vPIM follows the para-virtualization approach by expanding the virtio standard [45]. We prototype vPIM in Firecracker [12], an open-source virtual machine monitor (VMM) powered by AWS.

To minimize performance overhead compared to native execution on PIM devices (R_4), vPIM incorporates several optimization techniques. At the data level, vPIM leverages zero-copy from guest OS to Firecraker, efficient Guest Physical Address to Host Virtual Address translation, and automatic data batching and pre-fetching to amortize the overhead posed by small data transfers to/from the virtualized PIM device. At the thread management level, we introduce efficient virtio queue management, as well as efficient handling of parallel operations. Finally, we perform additional optimizations related to the utilization of Firecraker (e.g., recasting Rust to C for some portions of code).

One of the main goals of this work is to understand the performance implications posed by PIM virtualization. Our experimental evaluation uses two applications (Checksum [6] and Index Search [7] on a subset of the Wikipedia) provided by UPMEM and PrIM [31]. The latter is a recent PIM benchmark suite containing 16 realistic workloads from different domains, including dense/sparse linear algebra, databases, data analytics, graph processing, neural networks, bioinformatics, and image processing. First, all the applications run on vPIM without errors and with no modifications required to the applications or the hardware, thus confirming the effectiveness of vPIM. Second, vPIM's overhead can be as low as 1.01× compared to the native UPMEM execution. However, a straightforward virtualization of the PIM device can lead to overhead as high as 53.1× in specific workloads. For context, Firecracker's native overhead for 4KB-IO read operations is about 26× [12]. Note that prior studies [25, 27, 31, 39, 42] focused on in-PIM processing, neglecting data transfer cost. Our work fills this gap. To avoid rewriting PIM applications, we observed that it is crucial to minimize guest-hypervisor-VMM transitions. By applying adaptive batching for writes and data prefetching optimizations for reads, we reduce the overhead by 45×, compared to a more naive approach to virtualization. In particular, we found that the primary source of the virtualization overhead is the number of read/write calls and *not* the amount of transferred data. Each data transfer performed by the guest OS traps in the hypervisor (KVM module in Linux), which forwards the trap to the VMM (Firecracker). These transitions cause a performance bottleneck compared to the native UPMEM execution.

Even though our goal is to allow PIM application execution out-of-the-box in virtualized environments, our experience implementing vPIM led to two guiding principles for developing new PIM applications running on virtualized devices. First, when programming applications for vPIM, a fundamental principle should be to minimize data transfers. Second, the choice between serial and parallel transfer impacts the overall performance in virtualized environments. Like all pioneering work, we recognize that vPIM can be perfected and discuss current limitations. We hope this work will lay the foundation for future research on PIM for cloud computing. Our prototype will be open-sourced at anonymous-link upon publication.

In summary, this paper makes the following contributions:

(1) A new specification for PIM virtualization, based on the virtio standard. We instantiate this specification for UPMEM, based on Firecracker.
(2) The design and implementation of vPIM, the first open-source system for virtualizing UPMEM.
(3) An extensive experimental evaluation of vPIM in a wide range of applications provided by UPMEM [6, 7] and PrIM [31].
(4) Key lessons about UPMEM virtualization and its implications on application programming.

The rest of the paper is organized as follows. Section 2 provides details on PIM and UPMEM. Section 3 shows the design of vPIM. Section 4 describes the main performance optimizations in vPIM. Section 5 presents our experimental evaluation. Section 6 presents the related work, and Section 7 concludes the paper. The Appendix describes the PIM virtio specification we plan to discuss with the OASIS VIRTIO Technical Committee.

2 BACKGROUND ON UPMEM PIM

This section introduces UPMEM PIM (UPMEM for short). We use the latter for three main reasons. (1) UPMEM is the most recent PIM technology that can be employed on off-the-shelf machines. Other commercial PIM devices [3, 35] need special memory technologies. (2) UPMEM is provided with a software stack that simplifies the development and deployment of PIM applications. (3) UPMEM is the most popular PIM technology used in recent research [16, 24, 27, 31, 32, 39, 41, 42].

Hardware Architecture. Fig. 1 presents a typical architecture of a machine equipped with UPMEM. The main components are standard CPUs (*host CPUs*), standard DRAM main memory (*host DRAM*), and UPMEM PIMs. The latter are traditional DDR4-2400 DIMM modules and thus can be plugged into standard memory channels. An UPMEM PIM device is called a *rank* and includes 8 memory chips, hereafter called a *PIM chip*. The operations are sent to chips by writing to their control interfaces (CI). A PIM DIMM has 2 ranks, each with 64 DRAM Processing Units (DPUs). There are 8 DPUs per PIM chip.

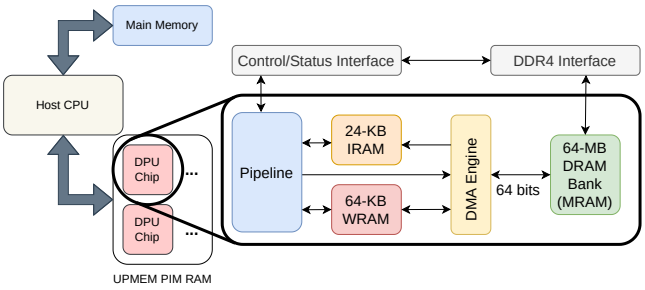


Figure 1: Typical machine architecture with UPMEM.

DPU Architecture. DPUs have a specific Instruction Set Architecture (ISA) and a frequency reaching 400 MHz. A given DPU

has access to a 64MB DRAM memory bank called Main Memory (MRAM); 24KB instruction memory (IRAM), where the DPU program shall be loaded; and 64KB of working memory (WRAM). The host CPU can access MRAM banks to transfer input data from host memory to MRAM (CPU→DPU) and transfer results back from MRAM to host memory (DPU→CPU). The DPU can run up to 24 threads called tasklets. These share the different memory regions described above. In order to fully utilize the DPU's performance, the program should run at least 11 tasklets. This is because of a hardware constraints that states that for a given thread, 11 cycles should separate 2 consecutive instructions.

```c
#define DPU_BINARY "./bin/dpu_code"

uint32_t count_zero(uint32_t *array, int array_size){
    struct dpu_set_t set, dpu;
    uint32_t zero_count = 0, dpu_zero_count;
    size_t dpu_count, each_size = array_size / NR_DPUS;
    dpu_alloc(NR_DPUS, NULL, &set); //allocate DPU
    dpu_load(set, DPU_BINARY, NULL); //load DPU program
    DPU_FOREACH(set, dpu, dpu_count) {dpu_prepare_xfer(dpu, &each_size);}
    dpu_push_xfer(set, DPU_XFER_TO_DPU, "partition_size", 0,
        sizeof(uint32_t), DPU_XFER_DEFAULT);
    DPU_FOREACH(set, dpu, dpu_count) {
        dpu_prepare_xfer(dpu, &array[dpu_count * each_size]);
    } //transfer parameter
    dpu_push_xfer(set, DPU_XFER_TO_DPU, DPU_MRAM_HEAP_POINTER_NAME, 0,
        each_size * sizeof(uint32_t), DPU_XFER_DEFAULT); //transfer data
    dpu_launch(set, DPU_SYNCHRONOUS); //launch DPU program
    DPU_FOREACH(set, dpu) {
        DPU_ASSERT(dpu_copy_from(dpu, "zero_count", 0,
                    &dpu_zero_count, sizeof(dpu_zero_count)));
        zero_count += dpu_zero_count;
    } //copy result to CPU
    dpu_free(set);
    return zero_count;
}
```

(a) Host-side code.

```c
#define NUM_TASKLET 16
__host uint32_t zero_count = 0;
__host uint32_t partition_size;
BARRIER_INIT(my_barrier, NUM_TASKLET);

int main() {
    if(me()==0) mem_reset(); barrier_wait(&my_barrier);
    int size_per_tasklet = partition_size/NUM_TASKLET;
    int *partition = (int *) mem_alloc(size_per_tasklet * sizeof(int));
    int *mram = (int *) DPU_MRAM_HEAP_POINTER +
                (me() * size_per_tasklet) * sizeof(int);
    mram_read((const __mram_ptr void*) mram, partition,
                size_per_tasklet * sizeof(int));
    for (int i = 0; i < size_per_tasklet; i++)
        {if(partition[i]==0) zero_count++;}
    return 0;
}
```

(b) DPU-side code.

Figure 2: An example of UPMEM program that counts the total number of zeros in an array.

Programming Model. UPMEM follows the single-program multiple-data (SPMD) programming paradigm. The same code is executed on all threads (tasklets) of all allocated DPUs while each tasklet processes different part of the data. In practice, to use the UPMEM PIM, the developer must split the application logic into the host-side program and the DPU-side program. Fig. 2 gives an example UPMEM program that counts the total number of zeros in an array.

The host-side program (Fig. 2.a) runs on host CPUs. It orchestrates the dataflow and the execution of DPU programs. The typical workflow of a host program is as follows: (1) allocate the desired number of DPUs (line 7) and offload the DPU-side program binary to the hardware (line 8), (2) partition and distribute the input datasets across the DPUs (CPU→DPU, line 12-16), (3) launch the DPU-side program (line 17), (4) after the DPU program terminates, it retrieves the results from the memory bank of the DPUs (DPU→CPU, line 18-22), and (5) free the DPUs (line 23). Similarly to GPUs, UPMEM PIM follows the offload model where the input data has to flow through the host CPU to the UPMEM PIM hardware before computation.

The DPU program (Fig. 2.b) runs inside the UPMEM chips. It manages the workload for each tasklet in one DPU (line 7-16), and defines computation to process the partition of the data loaded to the DPU (line 8-11). Note that the DPU processing pipeline only has access to the 64-KB WRAM while data loaded from the host CPU are stored in the MRAM banks. The DPU program has to explicitly manage the data load/store between WRAM and MRAM to support the computation of the tasks (line 12-13).

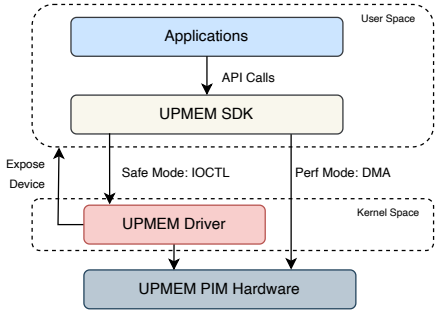


Figure 3: Software Components of a UPMEM System.

Software Stack. The UPMEM applications are written with the UPMEM Software Development Kit (SDK). The SDK provides a set of programming interfaces to manage the UPMEM PIM hardware, which is exposed to the Linux kernel as a device by the UPMEM driver. As Fig 3 shows, the SDK operates the hardware either through the kernel driver (*safe* mode) or directly from the user space (*performance* mode). In safe mode, operations are done through ioctls sent to the driver, providing isolation between host applications. In performance mode, the host application `mmaps` the whole MRAMs and control interfaces, allowing it to bypass the driver completely. `vPIM` uses both modes for different purposes.

UPMEM Hardware Limitations. UPMEM has the following three limitations which pose challenges for virtualization. First, UPMEM does not support direct communication between DPUs. Second, UPMEM needs data to flow through the host CPU before reaching the DPU chips. Third, UPMEM does not support the pause/resume of a task once it has been launched. This aspect is outside the scope of this paper and will be addressed in future work. For the first and second limitations, we propose two data transfer optimizations that alleviate the overhead of virtualized UPMEM devices (see Section 4). However, the virtualization overhead could be further decreased if inter-DPU transfer or direct DRAM-DPU transfer was possible. In

a future version of vPIM, we plan to extend our prototype beyond UPMEM PIM (e.g., adapting our work for [35] and [3]).

3 vPIM DESIGN

vPIM is the first work that virtualizes a PIM device. This step is key to bringing PIM to the cloud, the defacto execution infrastructure for companies.

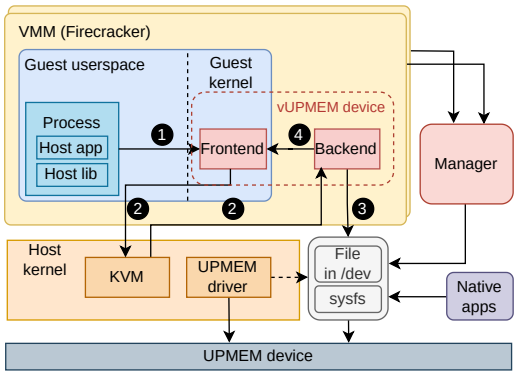


Figure 4: vPIM architecture. The client process in the guest userspace sends requests to the vUPMEM frontend driver, which then forwards them to the backend in the VMM. The backend executes operations on the UPMEM device, while the Manager manages UPMEM allocation across VMs.

3.1 Overview

Although our design is generic, we use Linux KVM [36] (the hypervisor) and Firecracker [12] (the virtual machine monitor) for illustration. We choose these systems for their wide adoption by cloud providers. Fig. 4 presents a high-level view of vPIM's architecture. It depicts the way vPIM's components interact and the path that a request takes when initiated from the VM. vPIM follows the para-virtualization approach, which requires no hardware and no hypervisor changes (R_1) to present a device to VMs. Throughout this section, we use vPIM to virtualize UPMEM devices, which we refer to as **vUPMEM**. VMs see vUPMEM devices thanks to the new virtio-based specification vPIM introduces. virtio is an I/O device para-virtualization standard offering efficiency and extensibility. More details about our new specification can be found in Appendix A.1. vPIM consists of three components:

- **Frontend**. The frontend is a virtio device driver located in the guest kernel. It exposes a virtual UPMEM device to the guest userspace and forwards the requests and data from UPMEM SDK to the backend in Firecracker (Fig. 4 step 1, Section 4.1). The client application uses vUPMEM in safe mode (via the frontend driver).
- **Backend**. The role of the backend is to represent the rank for the VM, decode the host-side requests, perform the desired operation on the rank, and return the request payload (Fig. 4 steps 2 to 4, Section 4.2). It uses UPMEM in performance mode (by `mmaping` the device).
- **Manager**. To enforce isolation between VMs, vPIM relies on a manager which runs on each host OS (Section 3.5). The manager is a userspace program that implements the rank-sharing/allocation policy (R_1). It's role is to monitor the ranks on the system and attach available ranks to virtual machines.

To utilize vPIM, cloud providers are required to install the frontend driver in the virtual machine kernel, while cloud users can directly utilized the untouched UPMEM's SDK within the guest (R_1). The process of device virtualization remains transparent to cloud users, eliminating any need to modify the SDK or the application interface (thus satisfying R_3).

This design is extensible to new features, ensuring compatibility with future SDK updates. To achieve this user space level transparency, the frontend driver exposes identical parameters to the VM's userspace as the native driver does in the host environment. The SDK leverages these provided details for configuration purposes. Within the virtual machine, rank operations (read/write to rank memory) and control interface operations (read/write to the hardware's control interface) are initiated by the SDK. These operations are managed as requests from the guest to the backend with Rank operations having a maximum transfer capacity of 4GB per operation due to hardware limitations.

Virtualizing UPMEM introduces two key challenges. First is the efficient handling of large data transfers without duplication between the guest and host. Second is managing frequent transfers efficiently, as they can significantly amplify virtualization overhead. Both challenges are critical to maintaining performance in the virtualized environment.

3.2 vUPMEM Bootstraping

When a Firecracker VM is launched, a thread establishes a listening socket to handle incoming requests, starting to receives the VM's configuration, such as the path to the kernel, the root file system, the virtio devices (including vUPMEM), and the VM launch command.

During boot, Firecracker passes information about its virtio devices to the VM via the command line. This includes details like the MMIO region allocated for Firecracker-guest communication and the IRQ number assigned for event triggering for vUPMEM. Upon reading the vUPMEM devices' description from the command line, the guest activates the vUPMEM frontend driver to manage the corresponding device.

The device initialization process configures virtio communication settings within the guest, sends a configuration request, and retrieves device attributes such as frequency and the number of available DPUs in the rank. After configuration, the vUPMEM frontend driver exposes the device to userspace through a device file (safe mode). Our evaluation shows that adding a vUPMEM device to a VM increases boot time by up to 2ms, which is negligible.

3.3 vUPMEM Booking

vPIM follows the same approach as Firecracker to specify resource allocation needs (e.g., number of CPUs). More precisely, hosts send requests to the Firecracker API server detailing the requested resources, including the desired amount of vUPMEMs. A VM can request as many vUPMEM devices as needed, up to the maximum number of physical UPMEMs available.

Due to UPMEM hardware limitations, vPIM performs allocation at the rank granularity rather than level of individual DPUs. However, vPIM allows dynamic rank allocation, allowing vUPMEM devices to be linked to different physical ranks during the same VM execution. For instance, once a vUPMEM device completes operations with a rank and no longer needs it, the rank can be released and reallocated to another VM. The vUPMEM device can then request a new available rank from the manager.

3.4 vUPMEM Usage

Within the guest OS, applications utilize the SDK's *safe mode* as the preferred mode. This choice arises from concerns related to rank isolation and the potential overhead associated with migrating a rank from one VM to another. This approach respects the isolation requirement (R_2) compared to the performance mode. Using the performance mode gives the cloud user's application direct access to the rank. This leads to security problems as the cloud application is untrusted. Moreover, the seamless usage and allocation of the device to the VM, as suggested by R_3, aligns well with the characteristics of the safe mode.

However, Firecracker interacts with the physical UPMEM device in *performance mode*. By employing mmap operations, Firecracker bypasses the kernel driver in the host (R_4). Using the performance mode in Firecracker (the VMM) does not lead to security concerns, as it is a trusted environment associated with the cloud provider. In addition, Firecracker works with vPIM's external manager to securely arbitrate rank sharing among VMs.

3.5 vUPMEM Sharing

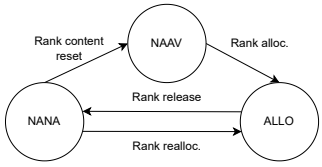


Figure 5: Rank states in the manager. Unallocated ranks start in the NAAV state, and set to the ALLOC state upon allocation. Set to the NANA state after released for content reset.

Multitenancy is a critical feature for cloud computing, yet the current UPMEM PIM hardware does not natively support it. Unlike the approach by Bongjoon Hyun et al. [32], which suggests hardware modifications such as introducing an MMU into UPMEM, vPIM adopts a different strategy by sharing UPMEM at the rank level. This approach requires no hardware changes (R_1), thus allows rapid adoption and deployment of vPIM.

The Manager monitors the availability of physical ranks and allocates them to VMs as needed. Fig. 5 illustrates the life cycle of a rank in vPIM, showing the different state it transits through. The manager maintains a rank table that tracks all ranks in the system, including essential information such as the rank's index, rank status file location, the currently assigned vUPMEM device, and the rank's state. A rank's state can be one of the following: in use (*allocated, ALLO*), available for allocation (*not allocated and available, NAAV*), or temporarily unavailable due to a reset process (*not allocated and not available, NANA*).

VMs initiate rank allocation requests to the manager either during vUPMEM device instantiation or at DPU allocation (called by an application in the VM). The manager listens for such requests via a UNIX domain socket. To effectively handle concurrent requests, the manager employs a thread pool with a configurable number of threads (set at 8 in our prototype) for asynchronous processing.

The allocation strategy is as follows. First, the manager checks any NANA rank requested by its previous user. This strategy optimizes resource usage by avoiding the need for resetting the rank, saving CPU cycle. If no suitable NANA device is found, the manager looks for a NAAV rank using a round-robin algorithm. If no NAAV ranks are available and NANA ranks exist, the manager enters a waiting state until a NANA rank becomes available. If neither type is available, it waits for a configurable timeout before retrying, up to a configurable number of attempts. If all attempts fail, the request is abandoned. In the current prototype, the manager assigns ranks to VMs in a FIFO manner.

Rank releases are managed by a dedicated observer thread which tracks the status of ranks via the sysfs files. Upon detecting a rank release, the manager updates its rank table and triggers the erase process to reset the rank's memory, ensuring no residual data is accessible to subsequent users and preventing information inference between VMs.

Note that the rank release process differs from rank allocation, where VMs explicitly calls the manager. Instead, a VM does not inform the manager when releasing a rank. This design choice enables the detection of UPMEM usage by both VM applications and native applications in the host without requiring any modification to those applications, allowing them to coexist seamlessly with VMs in the vPIM system, thus satisfying requirement R_3.

Data isolation between the VMs is maintained by requiring all rank allocations to go through the manager, which resets rank memory before reassigning it. Additionally, when UPMEM PIM is used as a memory device, byte interleaving prevents DPU programs from accessing rank memory data. This hardware feature makes it impossible for a DPU program to read the memory data of another VM when the device is used as a memory resource, thereby effectively meeting requirement R_2.

Currently, vPIM supports spatial sharing at rank granularity due to the lack of hardware support for application colocation. However, we plan to develop a software-based solution to enable application colocation within ranks. As the hardware evolves, we anticipate that future PIM hardware will incorporate direct support for application colocation, as suggested in related work [32].

4 IMPLEMENTATION

This section presents implementation details and optimizations vPIM integrates into the frontend and the backend to minimize the para-virtualization overhead (R_4).

4.1 Frontend

Data Transfer. In rank operations (read-from-rank, write-to-rank), to facilitate data transfer between the guest and device, the frontend needs to send the *transfer matrix* to the backend.

Fig. 6 provides an overview of this *transfer matrix*, sourced from the SDK. It contains three components: the metadata for the entire transfer, the metadata for each DPU's the data transfer, and arrays of Linux page `struct`, each representing the data for one DPU in the rank. A single rank can have up to 64 DPUs and each DPU has a memory bank of 64MB, so the *transfer matrix* can consist of at most 64 arrays, and each with up to 16,384 pages (64MB/4KB). Since the pages are allocated in the guest userspace, and Firecracker cannot directly access them via the Linux page `struct` in the guest address space, it is not feasible to directly transfer the matrix to the backend using the `virtqueue`.

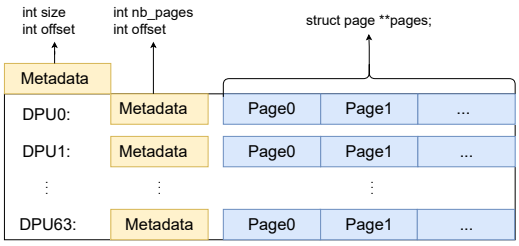


Figure 6: Example of a transfer matrix showing the structure of data and metadata and their distribution to DPUs.

`vPIM` serializes the matrix to fit the data into the virtqueue by breaking down the matrix into two buffer types: metadata buffers and page buffers, which are arrays of 64-bit unsigned integers. Each integer in the array is derived from a Linux page `struct` by converting it to a physical address in the VM (Guest Physical Addresses). This allows Firecracker direct access to the VM pages without copying data.

Fig. 7 illustrates the serialized matrix registered in the virtqueue. The maximum number of buffers is 130, ensuring that the serialized matrix fits within the virtqueue's capacity of 512 pointers, regardless of the data size.

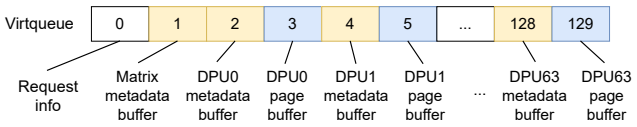


Figure 7: Serialized matrix in virtqueue, showing the alternation of metadata and data in the buffer.

Prefetch Cache In our para-virtualization design, each request triggers message passing between the frontend and backend, incurring a fixed communication overhead regardless of data size. For small data transfers, this overhead can dominate execution time. A critical challenge is thus posed by frequent small-size data transfer operations, which led to repetitive frontend-backend communication, causing up to 53× overhead compared to native. These repetitive transfers typically result from the host application processing DPU data block by block in a loop. Consequently, these transfers generally occur within contiguous memory regions in the MRAM of the DPUs.

Following this observation, we implement a prefetch mechanism within the frontend: ① At boot time, a prefetch cache is allocated within the frontend with the current implementation reserving 16 pages of cache per DPU. ② When a read request below the cache size is received, the frontend checks the cache for the data's availability and validity. If found, it is directly served from the cache, avoiding further backend communication. ③ If a cache miss occurs or the cache is invalidated, the frontend requests a data segment, starting at the request address and matching the cache size, from the backend to repopulate the cache.

The cache remains valid unless the rank is released, data is written to the MRAM (write-to-rank), or DPU programs are launched (via CI operations). Only in these cases is the cache invalidated, prompting a subsequent small read-from-rank to update the cache.

Request Batching. Similarly, to efficiently handle write-to-rank operations with frequent small-size data transfers, a request batching mechanism is introduced in the frontend.

In the UPMEM PIM programming model, data written to the MRAM from the host is not immediately utilized until either a DPU program is launched or a read-from-rank operation is performed. This delay provides an opportunity to batch multiple small write requests into a single message transfer between the frontend and backend. To achieve this, a batch buffer (64 pages per DPU in the current design) accumulates small write-to-rank requests received from the SDK. Once the buffer is full or when a non-write-to-rank request is received, all the buffered write requests are flushed collectively to the backend for processing.

Although this batching mechanism does not reduce the total data writing time, it minimizes virtualization overhead by merging multiple requests into a single interrupt. This reduces the number of frontend-backend interactions, enhancing performance by avoiding frequent VMEXITs.

Memory Overhead. The overhead of the frontend comparing to the native involves the memory use to serialize the matrix, and the buffers for prefetched data and batched requests. The maximum extra memory usage is: (16384×64)B (physical pages) + (16×4)KB (prefetch cache) + (64×4)KB (batch buffer) = 1.37MB per DPU.

4.2 Backend

Zero-copy Request Handling. A trap is triggered in the backend (via KVM) when the frontend sends a request to the event handler in Firecracker. The handler then parses the request and calls a function in the `vPIM` backend module that performs the requested operation on the rank. First, the backend deserializes the matrix received from the virtqueue. To avoid data copy and enable direct access to the pages of the matrix, Guest Physical Addresses (GPAs) in the matrix are translated into Host Virtual Addresses (HVAs) using several threads to accelerate the translation and then improve the overall performance. After the deserialization, the system proceeds with data operations on the DPUs. `vPIM` employs 8 threads to execute operations. This choice aligns with the system's setup, which involves 64 DPUs organized into chips of 8 DPUs. Consequently, operations are performed on 8 DPUs at a time. We empirically validate that using more than 8 threads does not provide additional benefits.

Parallel operations handling. In Firecracker's original implementation, a single loop handles virtio request events sequentially, processing each and injecting interrupt requests (IRQs) sequentially. This design limits parallel handling on multiple rank. vPIM improves this by assigning processing operations to dedicated threads. Upon receiving a request, a thread is spawned, marking the event as complete allowing the event manager to proceed to other tasks. Once the operation completes, the thread injects the IRQ to notify the guest driver to resume execution. This optimizes parallel request processing such that the maximum number of concurrent threads equals the number of ranks for the virtual machine.

AVX512 and C enhancements in Firecracker. Firecracker is developed in Rust. While generally does not impact performance. However, vPIM implementation benefit from rewriting specific code sections in C. We use a C implementation of AVX instructions and matrix management in the `read-from-rank` and `write-to-rank` operations. This design choice was influenced by the instability of the AVX512 support in Rust [4]. We show that implementing these functions in C results in a performance improvement of up to 343% (see Section 5).

Manager's Overhead. At startup, the manager's activities are primarily waiting for rank allocation and free requests. When idle (i.e., no requests received), the manager consumes on average 40% of a CPU core, mostly due to the observer thread. A `dpu_allocation` call from the VM triggers an Firecracker request to the manager, incurring an average overhead of 36 ms on average when the rank is in the NAAV state. If the rank transitions from NANA to NAAV, this overhead is extended by the reset time. In scenarios where all ranks are in the ALLO state, overhead depends on the VM applications' execution time.

When a rank is released, the manager performs a reset, moving the rank to the NANA state, which takes about 597 ms on average. In the worst-case scenario, resetting all ranks can raise CPU usage to 92% of a core, mainly due to the memset operation on 8GB of rank-mapped memory.

5 EXPERIMENTAL EVALUATION

We evaluate vPIM with 16 real-world applications from the PrIM benchmark suite [27], and microbenchmarks from the UPMEM SDK. We focus on comparing vPIM against UPMEM native execution, as the comparison with other types of compute (CPU and GPU) has already been addressed in prior work [27]. We show that:

(1) The overhead of vPIM over native execution is lower than 15% in the majority of PrIM applications and can be as low as 1% (Section 5.2).
(2) vPIM scales with the number of ranks, despite the UPMEM lack of inter-rank communication (Section 5.2).
(3) vPIM incurs minimal overhead over native execution when varying the data transfer size, number of vCPUs, and the number of DPUs (Section 5.3).
(4) Each optimization in vPIM makes a meaningful contribution to the overall system performance (Section 5.4).

5.1 Experimental Environment

Hardware. We use an Intel machine with a 16-core Xeon Silver 4215 CPU at 2.50GHz, with 7 memory modules, including three 64GB DDR4 memory (192 GB in total) and four UPMEM PIM modules. The latter includes 8 ranks with 480 functional DPUs * operating at 350MHz and a total of 30.5GB embedded memory. The machine runs Ubuntu Linux 20.04. Otherwise indicated, all VMs are configured with 128GB memory and 16 vCPUs.

Applications. We evaluate vPIM using the PrIM benchmark suite [27] to show the flexibility of our system on real applications, ranging from linear algebra to data analytics to image processing. Table 1 shows a full description of the PrIM benchmark. We also use two microbenchmarks provided by UPMEM. The first application computes the checksum of a given file. We use it as a microbenchmark to perform a sensitivity analysis of various aspects that can impact vPIM's performance, including varying the number of vCPUs, data transfer size, and degree of parallelism. The second microbenchmark scans an index database of Wikipedia documents.

Methodology. In all plots, vPIM represents the implementation of our prototype with all optimizations enabled (C Code Enhancement, Prefetch Cache, Request Batching, and Parallel Handling). In Section 5.2 and Section 5.3.1, we evaluate the performance of vPIM compared to native. In Section 5.4, we evaluate the effectiveness of each optimization. The native is run in performance mode. Each result is a mean of five runs.

Metrics. The main performance metric is the execution time. Since vPIM is the first system to enable virtualization for processing-in-memory devices, we use native execution time as the baseline and evaluate the virtualization overhead introduced by vPIM. We also use two sets of breakdowns. The first is application-centric, breaking the total execution time into data transfer (CPU-DPU, DPU-CPU, Inter-DPU) and program execution (DPU). The second is driver-centric, where we draw insights into how the workflow for different rank operations contributes to the total execution time.

5.2 PrIM Applications

This section presents the performance of PrIM applications when they run inside a VM using vPIM (with all optimizations enabled). For context, Firecracker's overhead for IO read 4KB-operations is about 26× [12]. We adopt the provided multi-ranks strong scaling configuration [27] with 1 rank (60 DPUs) † and 8 ranks (480 DPUs). In the context of this configuration, the size of the workload corresponds to the dataset size that can be accommodated within one rank, while the number of tasklets is set to the optimal value identified in the PrIM benchmarks [27]. All applications run seamlessly in the vPIM system, where the DPU computed results match accurately with those computed on CPUs. This demonstrates that vPIM works correctly. Fig. 8 displays the execution times of PrIM applications. The execution time is segmented based on application development aspects. These segments represent different components of the UPMEM application logic, such as CPU-to-DPU

*Due to defective DPUs, the total number of DPUs is reduced from the expected 512.
†In our machine, the first UPMEM rank has only 60 functional DPUs.

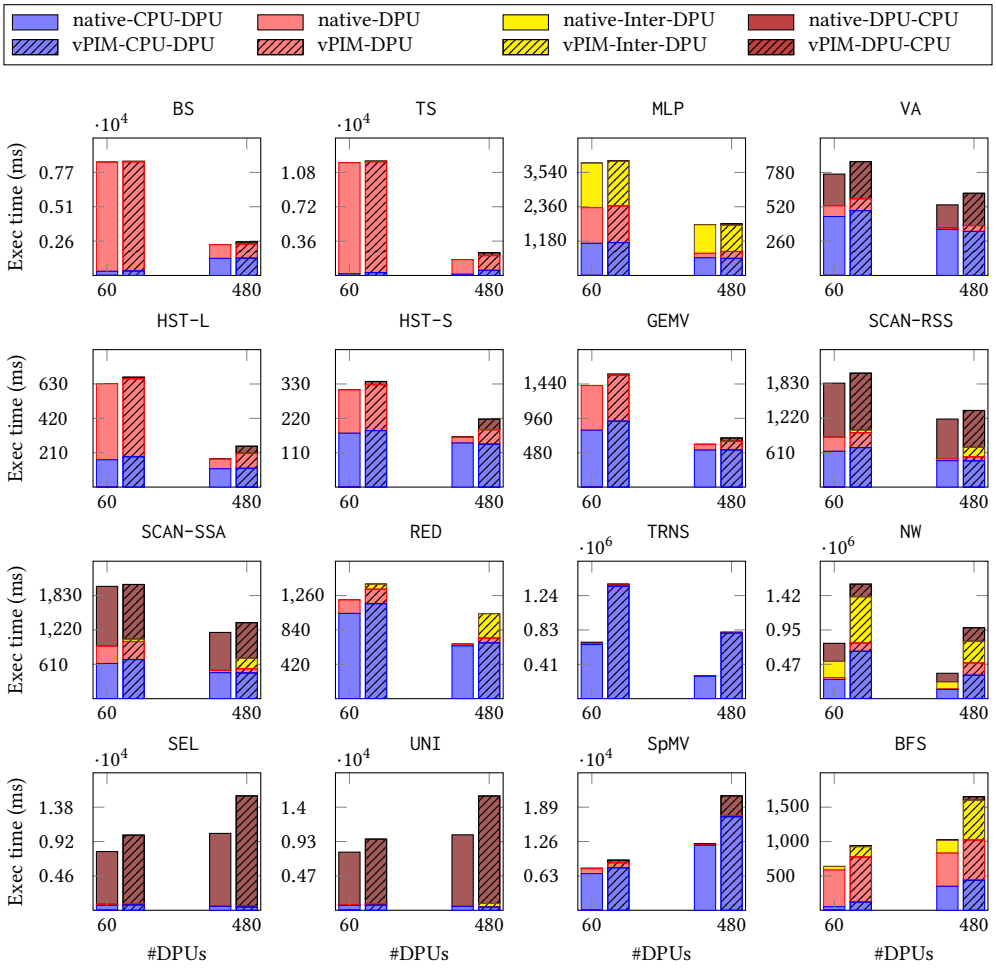


Figure 8: Execution time of vPIM running PrIM applications with one rank (60 DPUs) and 8 ranks (480 DPUs) using strong-scaling configuration. The execution time is segmented into four steps: data loading (CPU-DPU), task execution (DPU), synchronization (Inter-DPU), and result retrieval (DPU-CPU).

Table 1: PrIM Applications [27].

Domain	Benchmark	Short name
Dense linear algebra	Vector Addition	VA
	Matrix-Vector Multiply	GEMV
Sparse linear algebra	Sparse Matrix-Vector Multiply	SpMV
Databases	Select	SEL
	Unique	UNI
Data analytics	Binary Search	BS
	Time Series Analysis	TS
Graph processing	Breadth-First Search	BFS
Neural networks	Multilayer Perceptron	MLP
Bioinformatics	Needleman-Wunsch	NW
Image processing	Image histogram short	HST-S
	Image histogram long	HST-L
Parallel primitives	Reduction	RED
	Prefix Sum: scan-scan-add	SCAN-SSA
	Prefix Sum: reduce-scan-scan	SCAN-RSS
	Matrix Transposition	TRNS

transfer of input data ("CPU-DPU"), DPU execution time ("DPU"), inter-DPU communication via the host CPU ("Inter-DPU"), and DPU-to-CPU transfer of final results ("DPU-CPU") [27].

First, comparing the execution time of vPIM to native in the 60 DPUs configuration, the overhead reaches as low as 1.01× (with BS) and never goes beyond 2.07× (for NW), with an average of 1.24×. Nine of the sixteen applications have an overhead less than 1.15× and fourteen applications remain below the 1.5× overhead. In the 480 DPUs configuration, the overhead ranges from 1.02× (with MLP) to 2.89× (with TRNS), with an average overhead of 1.54×. Five applications show overhead of less than 1.15× and ten applications maintain overhead under 1.5×.

Note that we observe a trend where the overhead in the vPIM system increases with the number of utilized DPUs. This is a result of the workload partition in PrIM benchmarks. Dividing the workload across multiple DPUs results in smaller portions of data being transferred to each individual DPU. As will be detailed in

Section 5.3.1, small-size data transfers suffer more from the virtualization overhead. Consequently, this leads to higher overhead in vPIM when using 480 DPUs compared to 60 DPUs.

Second, we observe the reduction in total execution time for both native and vPIM systems when the number of DPUs is increased from 60 to 480 for twelve in sixteen applications (BS, TS, MLP, VA, HST-S, HST-L, GEMV, SCAN-RSS, SCAN-SSA, RED, TRNS, and NW). This trend aligns with the expected performance scaling when increasing the number of DPUs. We note exceptions in the last row of Fig. 8 for SEL, UNI, SpMV, and BFS for both native and vPIM. This results from their data transfer implementation by PrIM developers. The DPU-CPU step of SEL and UNI, and the CPU-DPU step for SpMV and BFS, are implemented in a serial way, which handles data one DPU at a time. Therefore, with an increase in the number of DPUs, there is a corresponding increase in the data transfer time, leading to extended total execution times in configurations utilizing 480 DPUs. This observation highlights the importance of choosing data transfer methods to fully leverage the system's capabilities, especially in larger-scale DPU configurations.

Third, in the Inter-DPU step of RED, we observe a significant overhead of 33.3× and 145.5× for the 60 DPUs and the 480 DPUs configurations, respectively. This step of the application only involves a read-from-rank operation of size 256 bytes. However, in the vPIM system, it triggers the Prefetch Cache, which proactively fetches a larger chunk of data. This Prefetch Cache strategy, intended to enhance efficiency for potential subsequent data transfers, leads to a more intensive task, resulting in exceptionally high overhead in the vPIM system. A similar situation is observed in the Inter-DPU step of SCAN-SSA and SCAN-RSS, and the DPU-CPU step of HST-S and HST-L. In these steps, one small-size reading operation triggers the Prefetch Cache Menchanism, resulting in a higher overhead than expected.

Takeaway 1

vPIM developers should disable the Prefetch Cache when their code lacks frequent small-size data transfers patterns to prevent unnecessary data fetching.

Fourth, the Inter-DPU step of the BFS algorithm also exhibits considerable overhead. Unlike the previous observation, this overhead is primarily a result of frequent synchronization handshakes among the DPUs. In the BFS algorithm, processing each level of the input graph requires a sequence of `read-from-rank` and `write-to-rank` operations to synchronize the node levels across the DPUs. Consequently, this leads to overheads of 3.0× and 3.2× in the Inter-DPU step for the 60 and 480 DPUs configurations, respectively.

Fifth, NW and TRNS workloads exhibit a significant overhead. These two applications are characterized by their extensive use of small-size data transfers. A data transfer is produced for each element in the large input matrix in their implementations. For instance, each data transfer step (DPU-CPU, CPU-DPU, Inter-DPU) of NW involves more than 650000 operations of 160 Bytes on average. In the CPU-DPU step of the TRNS, more than 980000 `write-to-rank` operations are performed in the 480 DPU configuration of 512 bytes on average. This considerable number of data transfers significantly strains the request handling of the PIM devices, leading to high overheads. Despite the substantial overhead in these applications shown in Fig. 8, it is important to note that this represents an already optimized scenario within the vPIM system. The Prefetch Cache and Request Batching optimizations implemented in the Frontend (Section 4.1) have been fully utilized in these cases. The analysis of the effectiveness of these optimizations, detailed in Section 5.4.2, reveals the challenges faced by the vPIM architecture in managing the high overhead caused by the frequent and small-scale data transfers in workloads such as NW and TRNS.

Takeaway 2

Applications with frequent small-size data transfers face the highest overhead in vPIM. To mitigate this, developers should minimize transfer operations, such as by aggregating data.

5.3 UPMEM Microbenchmarks

The Index Search and Checksum programs are microbenchmarks provided by UPMEM. These benchmarks offer essential insights into the system's efficiency and overhead for basic data processing operations. They establish a baseline for understanding the performance dynamics of vPIM.

5.3.1 Checksum. The checksum application features a host application that generates a random file of a specified size and transfers it to each allocated DPU for checksum computation. Each execution includes one write-to-rank and 60 read-from-rank operations, along with 8000 to 28000 CI (Control Interface) operations, depending on the running time. In contrast to the PrIM benchmarks and the Index Search application, where the workload is partitioned and executed in parallel across multiple DPUs, in the checksum program, all DPUs perform the same task on the same dataset. Note that Previous work focused on in-DPU performance, without evaluating the cost of these individuia operations. As noted in prior research [24], and demonstrated in our results, these operations can significantly affect UPMEM's overall performance.

Fig. 9 shows the total execution time of vPIM (with all optimizations enabled) for different configurations. The configuration in Fig. 9.a involves 60 DPUs and a 60MB input file size for each DPU. We vary the number of vCPUs. In Fig. 9.b, we vary the number of DPUs while maintaining a constant input file size of 60MB for each DPU and 16 vCPUs. In Fig. 9.c, we use 60 DPUs and 16 vCPUs with different input file sizes. Fig. 9.a indicates that the execution time is independent of the number of vCPUs. In Fig. 9.b, we see that the execution time increases when the VM uses more DPUs, owing to the additional data transfer cost, even though all DPUs are executing in parallel.

Fig. 9.c shows that in vPIM, the overhead decreases as the data size increases, ranging from 2.33× for 8MB to 1.29× for 60MB. This trend is attributed to the fixed time spent on message passing between the VM and Firecracker: as data transfer operations become lengthier, the proportion of time dedicated to message passing diminishes, thereby reducing the relative overhead.

5.3.2 Wikipedia Index Search. The UPMEM PIM Index Search application [7] is designed to scan an index database of documents to

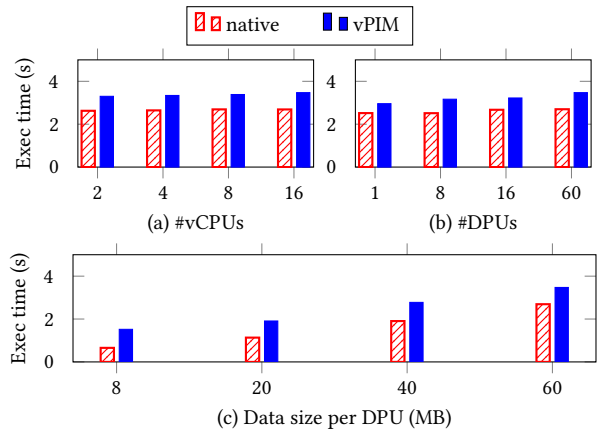


Figure 9: Checksum sensitivity analysis varying (a) #vCPUs, (b) #DPUs, and (c) transfer size. Configurations: 60 DPUs with 60 MB input per DPU and varying #vCPUs, 16 vCPUs with 60 MB input and varying #DPUs, and 16 vCPUs with 60 DPUs and varying input file size per DPU.

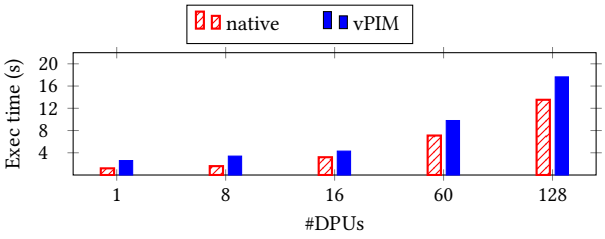


Figure 10: Execution time of the Index Search application.

find locations of a target sequence of words and return the document IDs and the positions.A benchmark configuration is provided, measuring the execution time of executing 445 index searching requests on 4305 different files from a subset of the English Wikipedia dataset. The application first builds the index for the corresponding database and writes it to the DPUs' MRAM. Requests are sent in batches of 128 elements. Thus 4 batches of requests are sent to the DPUs to compute the index search.

Fig. 10 shows the execution time of vPIM and native varying the number of utilized DPUs. The data size is 63MB. For both native and vPIM the execution time increases with the number of DPUs, as the data transfer time increases. The overhead, on the other hand, decreases with the number of DPUs, going from 2.1x for 1 DPU, to 1.3x for 128 DPUs.

5.4 Optimizations Evaluation

This section evaluates the effectiveness of each vPIM optimization we presented in Section 4. Table 2 shows the features that were enabled and disabled for different versions of vPIM, to better isolate the effect of each optimization.

5.4.1 C Enhancement. To justify our C enhancement optimization within Firecracker, two versions of vPIM are implemented: (1) vPIM-rust consists solely of Rust code and uses AVX2 for byte-interleaving, and (2) vPIM-C implements the C optimizations. We use the checksum program mentioned above for evaluation.

Table 2: Optimization strategies enabled for different versions of vPIM used in this section to evaluate the effectiveness of each optimization.

	C Code Enhancement	Prefetch Cache	Request Batching	Parallel Handling
vPIM-rust	✗	✗	✗	✗
vPIM-C	✓	✗	✗	✗
vPIM+P	✓	✓	✗	✗
vPIM+B	✓	✗	✓	✗
vPIM+PB	✓	✓	✓	✗
vPIM-Seq	✓	✓	✓	✗
vPIM	✓	✓	✓	✓

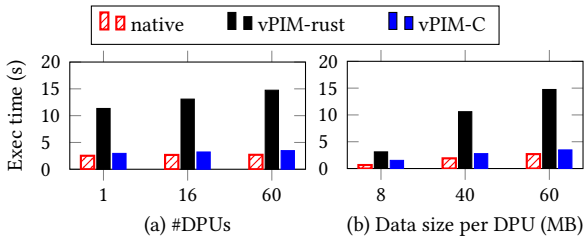


Figure 11: Execution time of checksum comparing native and vPIM implementations, using a 60 MB input file per DPU with varying #DPUs and 60 DPUs with varying input file sizes.

Fig. 11 shows the total execution time of each vPIM implementation across various configurations. In Fig. 11.a, we vary the number of DPUs while maintaining a constant input file size of 60MB for each DPU. In Fig. 11.b, we use 60 DPUs with different input file sizes. We observe that in both plots, vPIM-rust demonstrates considerably slower execution times with 5.2× overhead on average compared to native. Conversely, the execution time of vPIM-C is closer to the baseline with an average overhead of 1.4×.

We further analyze the effectiveness of the optimization by conducting a driver-centric breakdown, showing how three types of operation (write-to-rank, read-from-rank, and CI) contribute to the total execution time. We use 60 DPUs and 8MB input file size. This breakdown identifies latency sources and assesses vPIM 's efficiency in handling each operation type. The results are presented in Fig. 12, which includes execution times within the guest driver (frontend) and Firecracker (backend), and does not incorporate the execution time in the SDK. Unlike write-to-rank, the execution times of read-from-rank and CI exhibit a similar trend across both implementations. Therefore, the main factor influencing the overall execution time is the write-to-rank operation. Fig. 13) further breaks down write-to-rank into the following steps: page management (the frontend reallocates user space pages to kernel space pointers), matrix serialization (the frontend serializes the transfer matrix), virtio interrupt handling, matrix deserialization (the backend reassembles the transfer matrix), and data transfer to UPMEM.

Fig. 13.a reveals that the data transfer is the dominate step, representing 98.3% and 69.3% of the write-to-rank execution for

vPIM-rust and vPIM. The zoomed-in view of Fig. 13.b shows that the execution time of other steps remains relatively constant across implementations, which aligns with expectations, as the primary differences lie in the implementation of low-level operations.

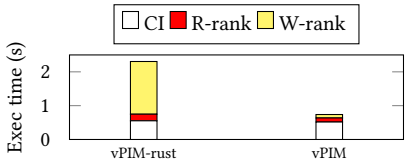


Figure 12: Breakdown of execution time for control interface (CI) operations, `read-from-rank` (R-rank), and `write-to-rank` (W-rank) operations, using 60 DPUs, 16 vCPUs, and an 8 MB file size. Only the execution time within the guest driver and Firecracker is shown.

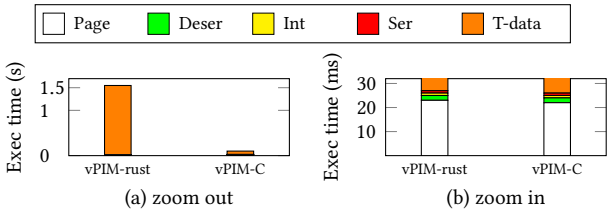


Figure 13: Breakdown of `write-to-rank` (W-rank) of Fig. 12 (checksum program, 60 DPUs and 8MB input). The second graph is a zoom-in of the first graph.

5.4.2 Prefetch Cache and Request Batching. In our initial evaluation of the vPIM prototype, before integrating Prefetch Cache and Request Batching optimizations, we observed significant overhead in certain scenarios. As shown in Section 5.3.1, smaller-size data transfer operations suffers higher virtualization overhead. In this early version of vPIM, each operation requires a message exchange between the guest and the VMM (Firecracker). Consequently, such workloads resulted in a substantial high virtualization overhead. An illustrative example is the strong-scaling single-rank configuration of the NW application from the PrIM benchmarks, which entails over 15000 small-size data transfer operations, each averaging approximately 109 Bytes per DPU.

Fig. 14 shows the execution time of NW after the implementation of Prefetch Cache & Request Batching optimizations in vPIM. These include the control version (vPIM-C), the Prefetch Cache only implementation (vPIM-P), the Request Batching only implementation (vPIM-B), and vPIM with both optimizations (vPIM-PB). The first two bars in Fig. 14 highlight the impact of the Prefetch Cache. We note a 89.3% reduction in read time (DPU-CPU), resulting in an overall improvement of 1.4×. This improvement is due to the order-of-magnitude reduction in the number of message exchanges between the VM and Firecracker, from 5000 to 125. The third bar shows the effectiveness of Request Batching. By batching `write-to-rank` operations, the frequency of context switches between the guest and Firecracker was reduced by two orders of magnitude, from 10000 to just 402. This optimization cut down the execution times for CPU-DPU and inter-DPU transfers by 95.8% and 95.3%, respectively.

The combined version of Prefetch Cache and Request Batching yielded an overall performance boost of 10.8×. This result verifies the performance benefits of these optimization strategies implemented in the vPIM system.

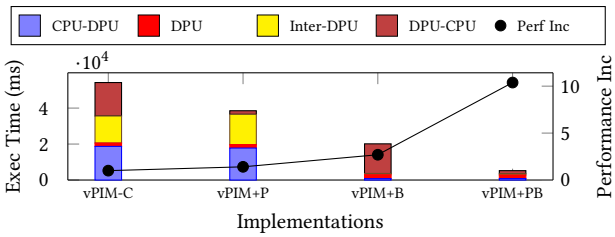


Figure 14: Execution time (left axis) and performance improvement (right axis) of the NW application with different vPIM optimizations. The unoptimized version (vPIM-C) shows a 53× overhead compared to native execution.

Takeaway 3

Prefetch Cache and Request Batching effectively reduce overhead from frequent data transfers by combining multiple operations. Developers can further minimize overhead by aggregating data transfers within their applications.

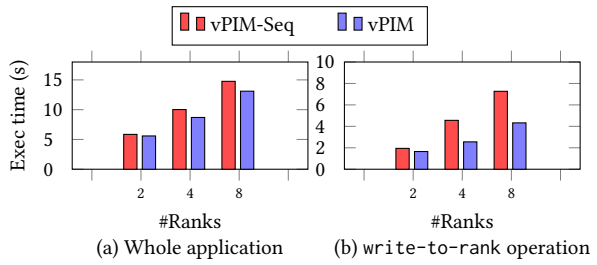
5.4.3 Parallel Operation Handling on Multi-Rank. This optimization addresses Firecracker's default sequential handling of virtio requests, enabling parallel execution in vPIM. We evaluated this optimization using the checksum program. Fig. 15 shows the results, where vPIM-Seq refers to the system without the optimization and vPIM includes the optimization. Experiments varys the number of ranks used by the VM.

Fig. 15.a displays the total execution time of the application, showing that the optimization achieves a 1.13× speedup on average, which increases with the number of ranks. Fig. 15.b showcases the effect of parallelization on the `write-to-rank` operation, depicting an average 1.4× speedup by parallelization.

To further understand the results, Fig. 16 shows the virtio request execution time for each rank involved in a single write operation. In the Sequential version, write requests is handled one after the other, resulting in increasing execution times for subsequent ranks. In contrast, the multithreaded version achieves nearly uniform execution times across all ranks, with the total time determined by the longest individual rank's operation.

6 RELATED WORK

Unlike PIM hardware devices, hardware accelerators such as GPUs (Graphic Processing Units) and FPGAs (Field Programmable Gate Arrays) are already heavily used in cloud production systems [51] thanks to a long history of academia and commercial solutions[1, 10, 15, 20, 22, 26, 28, 30, 33, 38, 40, 43, 44, 47, 49, 50, 52, 53, 56] to



(a) Whole application (b) `write-to-rank` operation

Figure 15: Execution time of checksum with Parallel Operation Handling on Multi-rank enabled. (a) shows the total application execution time, while (b) displays the write operation execution time.

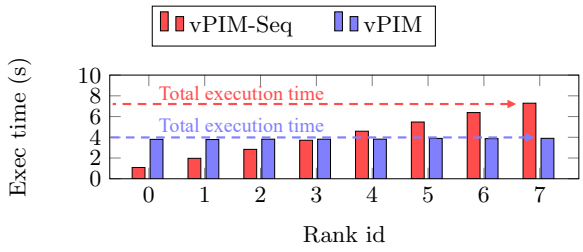


Figure 16: Execution time of virtio requests for a write-to-rank operation across 8 ranks. In the non-parallel version (red), requests are processed sequentially in the backend, while in the multithreaded version (blue), they are handled in parallel.

virtualize them. More recent solutions [2, 55] are converging to generic platforms able to integrate all types of existing accelerators.

AVA [55] is a framework that exposes well-known prior accelerators to VMs. Unlike most other solutions that work on virtualizing a single type of accelerator, AVA can virtualize multiple accelerators, including GPUs, TPUs, and FPGAs. However, the AVA framework is not a generic virtualization support and cannot include any new hardware accelerator (e.g., UPMEM ranks). In addition, the AVA design targets compute offload accelerators APIs. It does not provide guest VMs with a virtual representation of the accelerator but with an endpoint that routes communication through the hypervisor. `vPIM`, on the other hand, aims to fully virtualize the UPMEM ranks while allowing concurrent utilization by host applications. Moreover, AVA requires developer effort to integrate AVA's API components with the guest driver, while `vPIM` necessitates no work from users.

vAccel [2] is a library framework allowing serverless functions, containers, and VMs to access accelerators on cloud platforms. vAccel is close to AVA in that it provides virtualization support for many hardware accelerators, including GPUs, TPUs, and FPGAs. When using vAccel, users call a function from the vAccel API, and vAccel uses plugins to map this function call to an existing hardware-specific implementation. The vAccel API proposes functions for predefined and given types of operations, including image classification, object detection, matrix-to-matrix multiplication, Tensorflow operations, etc. vAccel virtualizes the accelerator at the granularity of *accelerate-able* functions, while `vPIM` exposes the entire DPU to the guest. `vPIM` can be considered an upstream work that can further be integrated into vAccel or AVA to extend the number of accelerators they support.

Although `vPIM` is, to the best of our knowledge, the first work to virtualize real PIM hardware, recent work is looking into the advantage of PIM architectures for cloud environments. In this vein, ***PIMCloud*** [17] explores how datacenters can make latency-critical applications benefit from PIM architectures. Datacenters services continuously impose microsecond-level tail latency quality of service constraints due to the increasing prevalence of serverless applications in the cloud. Nonetheless, PIM architectures, like prior accelerators (e.g., GPUs), have been studied and exploited for memory-intensive workloads. PIMCloud then changes the tone and investigates the impact of PIM architectures on latency critical and best-effort jobs, proposing a scheduling and data placement algorithm to *manage PIM resources to maximize their benefit for this emerging type of cloud services*. However, PIMCloud is evaluated in simulation and does not provide a virtualization solution. The focus of this work is showing the interest of PIM resources for the cloud, which emphasizes `vPIM`'s importance. PIMCloud results can further be applied to `vPIM` in a cloud environment.

7 CONCLUSION

We presented `vPIM`, the first PIM virtualizing solution. This work constitutes the first foundational step in virtualizing PIM devices. `vPIM` follows a virtio-based para-virtualization approach to facilitate its quick adoption by cloud users and providers. `vPIM` includes a set of optimizations to minimize virtualization overhead. We evaluated `vPIM` using unmodified applications provided by PrIM [31] and UPMEM. The results showed that some applications generate many guest-hypervisor-VMM transitions, drastically increasing virtualization overhead. For future work, we plan to investigate the vhost_vsock approach [8] to reduce the cost of performing guest-hypervisor-VMM transitions and enhance PIM device sharing across workloads. Although the current hardware limitations of UPMEM prevent `vPIM` from sharing devices at the DPU granularity, efficient pause-resume and checkpoint-restore mechanisms could enable dynamic workload consolidation without hardware changes. Additionally, a VMM module similar to the UPMEM simulator could support oversubscription by running applications at reduced performance.

8 ACKNOWLEDGEMENTS

This work would not have been possible without the substantial contributions of all the authors. We gratefully acknowledge UPMEM for providing the hardware that we used in the context of this research. This work was supported by the PAI2021 project "Fault Tolerance for Disaggregated Rack-Scale Computing" from *La Région Auvergne-Rhône-Alpes* and the Natural Sciences and Engineering Research Council of Canada, whose funding was critical to the successful completion of this study.

REFERENCES

[1] [n. d.]. GVTg Setup Guide. https://github.com/intel/gvt-linux/wiki/GVTg_Setup_Guide.
[2] [n. d.]. Hardware Acceleration for Serverless Computing. https://vaccel.org/.

[3] [n. d.]. SK hynix Develops PIM, Next-Generation AI Accelerator. https://news.skhynix.com/sk-hynix-develops-pim-next-generation-ai-accelerator/.
[4] [n. d.]. Status of AVX512. https://github.com/rust-lang/portable-simd/issues/28.
[5] [n. d.]. UPMEM. https://www.upmem.com/.
[6] [n. d.]. UPMEM Checksum. https://github.com/upmem/dpu_demo.
[7] [n. d.]. UPMEM PIM Index Search. https://github.com/upmem/usecase_UPIS.
[8] [n. d.]. Virtio and Vhost Architecture - Part 2. https://insujang.github.io/2021-03-15/virtio-and-vhost-architecture-part-2/.
[9] [n. d.]. Worldwide IDC Global DataSphere Forecast, 2023-2027: It's a Distributed, Diverse, and Dynamic (3D) DataSphere. https://www.marketresearch.com/IDC-v2477/Worldwide-IDC-Global-DataSphere-Forecast-33986214/.
[10] 2023. Using Remote GPU Virtualization Techniques to Enhance Edge Computing Devices. *Future Generation Computer Systems* 142 (2023).
[11] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture. In *Proceedings of ISCA*.
[12] Alexandru Agache and Marc Brooker and Alexandra Iordache and Anthony Liguori and Rolf Neugebauer and Phil Piwonka and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *Proceedings of NSDI*.
[13] George Amvrosiadis, Ali R Butt, Vasily Tarasov, Erez Zadok, Ming Zhao, Irfan Ahmad, Remzi H Arpaci-Dusseau, Feng Chen, Yiran Chen, Yong Chen, et al. [n. d.]. Data Storage Research Vision 2025: Report on NSF Visioning. https://par.nsf.gov/servlets/purl/10086429.
[14] Aurelia Augusta and Stratos Idreos. 2015. JAFAR: Near-Data Processing for Databases. In *Proceedings of SIGMOD*.
[15] Stuart Byma, J Gregory Steffan, Hadi Bannazadeh, Alberto Leon-Garcia, and Paul Chow. 2014. Fpgas in the Cloud: Booting Virtualized Hardware Accelerators with Openstack. In *A Case Study of FCCM*.
[16] Jinfan Chen, Juan Gómez-Luna, Izzat El Hajj, Yuxin Guo, and Onur Mutlu. 2023. SimplePIM: A Software Framework for Productive and Efficient Processing-in-Memory. arXiv:2310.01893 [cs.AR]
[17] Shuang Chen, Yi Jiang, Christina Delimitrou, and José F. Martínez. 2022. PIM-Cloud: QoS-Aware Resource Management of Latency-Critical Applications in Clouds with Processing-in-Memory. In *Proceedings of HPCA*.
[18] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory. In *Proceedings of ISCA*.
[19] Pietro Cicotti, Sarp Oral, Gokcen Kestor, Roberto Gioiosa, Shawn Strande, Michela Taufer, James H. Rogers, Hasan Abbasi, Jason Hill, and Laura Carrington. 2016. *Data Movement in Data-Intensive High Performance Computing*. Springer International Publishing, Cham, 31–59. https://doi.org/10.1007/978-3-319-33742-5_3
[20] Micah Dowty and Jeremy Sugerman. 2009. GPU Virtualization on VMware's Hosted I/O Architecture. *ACM SIGOPS Operating Systems Review* 43, 3 (2009).
[21] Jeff Draper, Jacqueline Chame, Mary Hall, Craig Steele, Tim Barrett, Jeff LaCoss, John Granacki, Jaewook Shin, Chun Chen, Chang Woo Kang, Ihn Kim, and Gokhan Daglikoca. 2002. The Architecture of the DIVA Processing-in-Memory Chip. In *Proceedings of SC*.
[22] Suhaib A Fahmy, Kizheppatt Vipin, and Shanker Shreejith. 2015. Virtualized FPGA Accelerators for Efficient Cloud Computing. In *A Case Study of CloudCom*.
[23] Fernandez, Ivan and Quislant, Ricardo and Gutiérrez, Eladio and Plata, Oscar and Giannoula, Christina and Alser, Mohammed and Gómez-Luna, Juan and Mutlu, Onur. 2020. NATSA: A Near-Data Processing Accelerator for Time Series Analysis. In *Proceedings of ICCD*.
[24] Friesel, Birte and Lütke Dreimann, Marcel and Spinczyk, Olaf. 2023. A Full-System Perspective on UPMEM Performance. In *Proceedings of DIMES*.
[25] Ghose, S. and Boroumand, A. and Kim, J. S. and Gómez-Luna, J. and Mutlu, O. 2019. Processing-in-Memory: A Workload-Driven Perspective. *IBM Journal of Research and Development* 63, 6 (2019).
[26] Giulio Giunta, Raffaele Montella, Giuseppe Agrillo, and Giuseppe Coviello. 2010. A GPGPU Transparent Virtualization Component for High Performance Computing Clouds. In *Proceedings of Euro-Par Conference*.
[27] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F Oliveira, and Onur Mutlu. 2022. Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System. *IEEE Access* 10 (2022).
[28] Nelson Mimura Gonzalez and Tonia Elengikal. 2021. Transparent I/O-Aware GPU Virtualization for Efficient Resource Consolidation. In *Proceedings of IPDPS*.
[29] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. 2016. Biscuit: A Framework for Near-Data Processing of Big Data Workloads. In *Proceedings of ISCA*.
[30] Vishakha Gupta, Ada Gavrilovska, Karsten Schwan, Harshvardhan Kharche, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. 2009. GViM: GPU-Accelerated Virtual Machines. In *Proceedings of HPCVirt*.
[31] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. 2021. Benchmarking Memory-centric Computing Systems: Analysis of Real Processing-in-Memory Hardware. In *Proceedings of IGSC*.
[32] B. Hyun, T. Kim, D. Lee, and M. Rhu. 2024. Pathfinding Future PIM Architectures by Demystifying a Commercial PIM Technology. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 263–279. https://doi.org/10.1109/HPCA57654.2024.00029
[33] Jason Kennedy, Vishal Sharma, Blesson Varghese, and Carlos Reaño. 2023. Multi-Tier GPU Virtualization for Deep Learning in Cloud-Edge Systems. *IEEE Transactions on Parallel and Distributed Systems* 34, 7 (2023).
[34] Gokcen Kestor, Roberto Gioiosa, Darren J. Kerbyson, and Adolfy Hoisie. 2013. Quantifying the energy cost of data movement in scientific applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*. 56–65. https://doi.org/10.1109/IISWC.2013.6704670
[35] Kim, Jin Hyun and Kang, Shin-haeng and Lee, Sukhan and Kim, Hyeonsu and Song, Woongjae and Ro, Yuhwan and Lee, Seungwon and Wang, David and Shin, Hyunsung and Phuah, Bengseng and Choi, Jihyun and So, Jinin and Cho, YeonGon and Song, JoonHo and Choi, Jangseok and Cho, Jeonghyeon and Sohn, Kyomin and Sohn, Youngsoo and Park, Kwangil and Kim, Nam Sung. 2021. Aquabolt-XL: Samsung HBM2-PIM with in-Memory Processing for ML Accelerators and Beyond. In *A Case Study of HCS*.
[36] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. KVM: The Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium*, Vol. 1.
[37] Mai, Ken and Paaske, Tim and Jayasena, Nuwan and Ho, Ron and Dally, William J and Horowitz, Mark. 2000. Smart Memories: A Modular Reconfigurable Architecture. In *Proceedings of ISCA*.
[38] Joel Mandebi Mbongue, Festus Hategekimana, Danielle Tchuinkou Kwadjo, and Christophe Bobda. 2018. Fpga Virtualization in Cloud-Based Infrastructures Over virtio. In *A Case Study of ICCD*.
[39] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. 2019. Processing Data where it Makes Sense: Enabling In-Memory Computation. *Microprocessors and Microsystems* 67 (2019).
[40] Diana M. Naranjo Delgado, Manuel Contreras, Germán Moltó, Sebastián Risco, Ignacio Blanquer, Javier Prades, and Federico Silla. 2023. On the Acceleration of FaaS Using Remote GPU Virtualization. In *Proceedings of ICPE*.
[41] Joel Nider, Craig Mustard, Andrada Zoltan, John Ramsden, Larry Liu, Jacob Grossbard, Mohammad Dashti, Romaric Jodin, Alexandre Ghiti, Jordi Chauzi, and Alexandra Fedorova. 2021. A Case Study of Processing-in-Memory in off-the-Shelf Systems. In *Proceedings of USENIX ATC*.
[42] Onur Mutlu and Saugata Ghose and Juan G'omez-Luna and Rachata Ausavarungnirun. 2020. A Modern Primer on Processing in Memory. *ArXiv* abs/2012.03112 (2020).
[43] Anna Panagopoulou, Michele Paolino, and Daniel Raho. 2023. Virtio-FPGA: A Virtualization Solution for SoC-Attached FPGAs. In *A Case Study of ESARS-ITEC*.
[44] Michele Paolino, Sébastien Pinneterre, and Daniel Raho. 2017. FPGA Virtualization with Accelerators Overcommitment for Network Function Virtualization. In *A Case Study of ReConFig*.
[45] Rusty Russell. 2008. virtio: Towards a De-facto Standard for Virtual I/O Devices. *ACM SIGOPS Operating Systems Review* 42, 5 (2008).
[46] Yasas Seneviratne, Korakit Seemakhupt, Sihang Liu, and Samira Khan. 2023. NearPM: A Near-Data Processing System for Storage-Class Applications. In *Proceedings of EuroSys*.
[47] Lin Shi, Hao Chen, Jianhua Sun, and Kenli Li. 2011. vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines. *IEEE Trans. Comput.* 61, 6 (2011).
[48] Stone, Harold S. 1970. A Logic-in-Memory Computer. *IEEE Trans. Comput.* 100, 1 (1970).
[49] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. 2015. Gpuvm: Gpu Virtualization at the Hypervisor. *IEEE Trans. Comput.* 65, 9 (2015).
[50] Kun Tian, Yaozu Dong, and David Cowperthwaite. 2014. A Full GPU Virtualization Solution with Mediated Pass-through. In *Proceedings of ATC*.
[51] Anuj Vaishnav, Khoa Dang Pham, and Dirk Koch. 2018. A survey on FPGA Virtualization. In *Proceedings of FPL*.
[52] Lan Vu, Hari Sivaraman, and Rishi Bidarkar. 2014. GPU Virtualization for High Performance General Purpose Computing on the ESX Hypervisor. In *Proceedings of HPC*.
[53] Wei Wang, Miodrag Bolic, and Jonathan Parri. 2013. pvFPGA: Accessing an FPGA-based Hardware Accelerator in a Paravirtualized Environment. In *A Case Study of CODES+ISSS*.
[54] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon Wei. 2021. RecSSD: Near Data Processing for Solid State Drive Based Recommendation Inference. In *Proceedings of ASPLOS*.
[55] Hangchen Yu, Arthur Michener Peters, Amogh Akshintala, and Christopher J Rossbach. 2020. Ava: Accelerated Virtualization of Accelerators. In *Proceedings of ASPLOS*.

[56] Deze Zeng, Andong Zhu, Lin Gu, Peng Li, Quan Chen, and Minyi Guo. 2023. Enabling Efficient Spatio-Temporal GPU Sharing for Network Function Virtualization. *IEEE Trans. Comput.* (2023).
[57] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L. Greathouse, Lifan Xu, and Michael Ignatowski. 2014. TOP-PIM: Throughput-Oriented Programmable Processing in Memory. In *Proceedings of HPDC*.

A APPENDIX

A.1 Virtio PIM Specification

We define the first specification for PIM devices following the `virtio` guidelines [45]. We intend to work with the VIRTIO technical committee to include our specification in a future VIRTIO specification release.

Device ID. The `virtio` PIM device is assigned a free ID (at this time). the `virtio` device ID 42.

Virtqueues. The `virtio` PIM device consists of two queues: *transferq* and *controlq*. Transferq is used to transfer data and commands to and from the PIM device. Whenever `vPIM` can transfer commands directly through this queue, transferring data should be done by avoiding copies. The data transfer must include the Guest Physical Addresses (GPAs) of the pages that contain data and their respective metadata. This queue has 512 slots. Controlq handles the synchronization with the manager. A boolean is sufficient to communicate this notification.

Feature bits. No feature bits are needed in this implementation of `vPIM`. This is because at this time, there are no specific features related to this devices that should be mentioned to the guest. Moreover, there are no other PIM devices that need to be specifically handled.

Device configuration layout. The PIM device requires the driver to be aware of the following configurations and hardware characteristics, including clock division, memory region size, number of control interfaces, processing units frequency, and power management information. These configurations are presented to the userspace by the native PIM device driver in the host environment.

Device initialization. Virtual PIM initialization consists of the following steps: instantiating the device structure, gathering PIM configuration details, creating a link between the real PIM device and the `virtio` PIM device through `mmap` operations, and establishing the two virtqueues. The driver must wait until the completion of device initialization before sending any requests. Additionally, the `virtio` PIM device must ensure that the underlying device is not being utilized by another application during initialization.

Device operations. The `virtio` PIM device supports five operations: requesting configuration, sending commands, reading commands, writing to the PIM device, and reading from the PIM device. It is crucial for the driver to refrain from sending any requests when the `virtio` PIM device is not linked to a physical PIM device. This is because the request will not be handled by any rank beneath and will be lost. Simultaneously, the device itself must ensure that it is consistently linked to a PIM device. If linking is not feasible at a given time, the device should attempt to establish a connection after a designated countdown period.