PIM-CARE: A Compiler-Assisted Dynamic Resource Allocation Framework for Real-world DRAM PIM

Inyong Hwang*
Yonsei University
Seoul, Republic of Korea
inyonghwang@yonsei.ac.kr

Donghyeon Kim*
Hanyang University
Seoul, Republic of Korea
dhkim9309@hanyang.ac.kr

Seokwon Kang
Yonsei University
Seoul, Republic of Korea
kswon0202@yonsei.ac.kr

Taehyeong Park
Yonsei University
Seoul, Republic of Korea
taehyeongpark@yonsei.ac.kr

Taehoon Kim
Hanyang University
Seoul, Republic of Korea
ted6345@hanyang.ac.kr

Jiwon Seo
Seoul National University
Seoul, Republic of Korea
seojiwon@snu.ac.kr

Hanjun Kim
Yonsei University
Seoul, Republic of Korea
hanjun@yonsei.ac.kr

Youngsok Kim
Yonsei University
Seoul, Republic of Korea
youngsok@yonsei.ac.kr

Yongjun Park
Yonsei University
Seoul, Republic of Korea
yongjunpark@yonsei.ac.kr

Abstract

Processing-In-Memory (PIM) has recently emerged as a promising solution to alleviate the memory bottleneck by integrating computing capabilities into memory chips. Since PIM provides numerous Processing Elements (PEs) and high-bandwidth on-chip data transfers, full utilization of the PEs becomes a critical mission to maximize the performance of PIM applications. However, due to the diverse and complex characteristics of PIM applications, using more resources does not always improve performance. It is therefore important to find the suitable amount of resources to achieve the best performance and to fully utilize the PIM resources.

To address this, we introduce *PIM-CARE*, a framework for dynamic resource allocation across multiple applications with compiler support on real-world PIM systems. PIM-CARE first determines the best amount of PIM resources to allocate for each application. To enable spatial multitasking, the PIM-CARE daemon monitors resource allocation and deallocation requests and estimates total PIM resource utilization at runtime. It then dynamically schedules applications using a priority-based out-of-order policy, considering both available PIM resources and resource requirements for best performance. Evaluation on real-world PIM systems shows that PIM-CARE improves throughput by 5.49x and average turnaround time by 5.71x compared to the baseline.

CCS Concepts

• **Hardware** → **Emerging architectures**; • **Software and its engineering** → **Scheduling**; **Multiprocessing / multiprogramming / multitasking**; **Concurrency control**; *Compilers*.

Keywords

Processing-In-Memory, Spatial Multitasking, Workload Scheduling, Dynamic Resource Allocation

ACM Reference Format:
Inyong Hwang, Donghyeon Kim, Seokwon Kang, Taehyeong Park, Taehoon Kim, Jiwon Seo, Hanjun Kim, Youngsok Kim, and Yongjun Park. 2025. PIM-CARE: A Compiler-Assisted Dynamic Resource Allocation Framework for Real-world DRAM PIM. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3721145.3725777

1 Introduction

Memory-intensive applications such as deep learning, big data analysis, and recommendation systems have recently been widely deployed in various domains. As these applications have become increasingly complex, the need for efficient data processing has become more critical to achieving high performance. However, data transfer has emerged as a primary bottleneck because it cannot keep up with the escalating demands, in contrast to the significant advances in computing power that are achieved in CPUs and GPUs.

*Co-first authors.

ICS '25, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3725777

Recently, in response to this challenge, memory-centric computing approaches, such as Processing-In-Memory (PIM) [1–3, 9, 10, 12, 18, 23, 25, 26, 28–30, 34, 47] and In-Storage Processing (ISP) [8, 13–15, 22, 31–33, 35, 36, 41, 43, 44], have been intensively researched to improve the efficiency of processing memory-intensive tasks. PIM integrates simple processors with memory banks on a single chip, enabling high-bandwidth computations close to the data. In particular, DIMM-based PIM offer high parallelism with fair scalability similar to main memory. Thanks to these features, data access costs for memory-intensive applications can be reduced, and overall performance can be improved by offloading tasks from main processors such as CPUs or GPUs.

UPMEM PIM [6, 45] is one of the commercially available real PIM architectures. UPMEM integrates general-purpose processors, called DRAM processing units (DPUs), with memory banks and achieves high programmability by providing well-designed programming models such as OpenCL and CUDA. In the UPMEM, programmers can easily accelerate existing memory-intensive workloads, unlike most other PIM that only handle predefined operations. They offer a PIM as a DIMM module with 128 DPUs and 8 GB of PIM-enabled memory, enabling simple scalability using a standard memory control interface (currently up to 2,560 DPUs and 160 GB of memory). This enables systems to be built with large PIM resource pools, such as cloud servers, that can handle applications with diverse resource requirements.

Given the architectural characteristics of PIM, such as high internal bandwidth and the proportional increase in parallel computation performance with the increasing number of memory banks, PIM workloads are expected to achieve higher performance as more resources are used. However, current PIM systems are used as additional accelerators rather than fully replacing traditional main memory. Thus, end-to-end performance is affected not only by the capabilities within the PIM banks, but also by the external memory bandwidth. Given these factors, using more PIM resources does not always guarantee performance improvement [7]. Instead, each workload shows different performance trends depending on the degree of workload parallelization and variations in data communication costs. Therefore, it is critical to allocate an appropriate amount of resources to each workload to achieve optimal performance in PIM systems.

By supporting spatial multitasking of multiple PIM workloads, it is possible to achieve higher overall system throughput with full resource utilization while maintaining minimum latency for each workload. However, despite the increasing prevalence of multi-tenancy in high-performance computing [16, 17, 24, 37–39, 48, 49], research on efficient multitasking of PIM applications has not been actively performed [21, 42]. Our prior work, VirtualPIM [21], was the first to enable efficient orchestration of multiple PIM applications by supporting spatial multitasking across the full PIM resource space. To maximize resource utilization, we proposed several key techniques, including temporal optimization to minimize the PIM resource occupancy, a resource monitoring system to enable spatial multitasking, and fine-grained scheduling to divide a single application into multiple small units. Nevertheless, several challenges remain to be addressed for further enhancement of this approach.

While in-order scheduling methods based on execution order can ensure fairness, they are limited in maximizing resource utilization because the performance of PIM systems in multi-tenant environments relies heavily on dynamically launched workloads. To address this issue, VirtualPIM proposes fine-grained scheduling, aiming to fully utilize all PIM resources regardless of workload dynamics. However, this approach introduces significant runtime overhead, along with additional costs to maintain data consistency. For example, in the case of SCAN-RSS from PrIM benchmark [11], data copied once is reused across multiple kernel executions. With fine-grained spatial multitasking, VirtualPIM incurs the overhead of re-transmitting input data for each kernel operation to ensure data consistency (referred to as "input reproduction"). As a result, host-to-PIM data transfers for SCAN-RSS are doubled, leading to increased PIM resource occupancy and a severe latency slowdown.

In contrast, out-of-order (OoO) scheduling can achieve comparable or even higher system throughput than VirtualPIM. The OoO approach rearranges the execution order to prioritize workloads that require fewer resources than available, effectively avoiding resource under-utilization and improving throughput without incurring virtualization or data consistency management overheads. By also using the "best" amount of PIM resources for optimal performance, the OoO approach can further minimize latency for each workload and achieves better overall performance.

Even with out-of-order scheduling, it is not always possible to maximize resource utilization if there are no workloads that can be allocated to available resources. However, because PIM workloads have predictable performance patterns in relation to resource usage, executing tasks immediately with suboptimal resources—assuming minimal latency degradation—often outperforms waiting for sufficient resources. Thus, by minimizing latency degradation, both resource utilization and system throughput can be maximized. This requires an efficient mechanism that can dynamically adjust resource requirements and estimate performance trend at minimal cost to optimize overall system performance.

Based on these insights, we propose a **C**ompiler-**A**ssisted dynamic **RE**source allocation framework for PIM applications, called *PIM-CARE*, which enables near-optimal resource

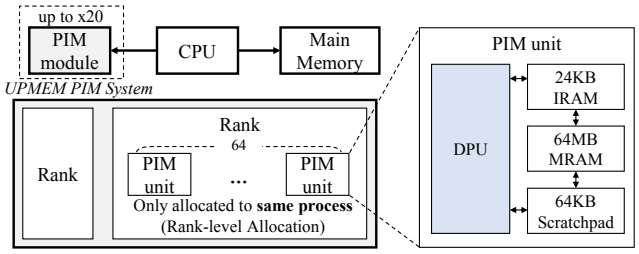

Figure 1: An overview of UPMEM PIM system.

allocation and efficient multitasking for multiple PIM applications. For this, we first construct per-application performance tables using profiling. Based on this, PIM-CARE dynamically determines the best amount of PIM resources to use by considering both currently available resources and application-specific performance variations based on resource usage. A daemon monitors total PIM resource utilization and manages dynamic resource allocation by communicating with all PIM applications. To enable this, the PIM-CARE compiler modifies the PIM control APIs to the communication API with the daemon process. PIM-CARE schedules applications with an out-of-order policy to fully utilize PIM resources while minimizing latency slowdowns. As a result, PIM-CARE supports efficient orchestration of multiple PIM workloads without any virtualization overhead.

We implemented PIM-CARE based on the core runtime structure of the CASE [4] by adding the custom UPMEM PIM backend API call library. We evaluated our framework on the UPMEM PIM [6, 45], using various applications from the PrIM benchmark [11]. We believe that the approaches proposed in this work can be easily extended to most DIMM-based PIM systems with many PEs. To the best of our knowledge, this is the first work to maximize total resource utilization by applying both dynamic resource adjustment and OoO scheduling of PIM applications on real-world PIM systems.

The contributions of this study are as follows:

- The design of a PIM-CARE framework to efficiently manage multi-application execution with near-optimal resource allocation.
  (1) Determining the best amount of PIM resources for each application with per-application profiling.
  (2) Near-optimal dynamic resource allocation considering the optimal resource requirements and the available resources.
  (3) Out-of-order workload scheduling considering both throughput and ATT within available resources.
- Full-stack implementation of the PIM-CARE framework, including compiler-level code modification.
- Evaluation of the PIM-CARE framework on real PIM systems, achieving a 5.49x improvement in throughput and a 5.71x improvement in average turnaround time.

2 Background And Motivation

2.1 UPMEM PIM System

UPMEM PIM is a DRAM-based real PIM architecture that integrates DPUs closer to the memory bank. Owing to the high bandwidth of up to 1GB/s [45] between memory and DPUs, the UPMEM PIM effectively performs memory-intensive workloads compared to traditional processors. Figure 1 shows an overview of the UPMEM PIM system. A PIM module is connected to host CPUs using DIMM connections, providing two PIM ranks (64 PIM units per rank). The current UPMEM PIM system can accommodate up to 20 PIM modules. In the UPMEM PIM, each DPU operates independently and cannot communicate with others. Communication between the DPUs is achieved by data transfer via the host memory.

The UPMEM PIM system provides a high-level programming model for PIM workloads. It also provides a communication API set that allows host programs to control PIM resources easily, similar to GPUs. When a PIM application requests a predefined number of DPUs or ranks to use, the UPMEM system allocates the requested PIM resources at the rank level if possible. However, since the current UPMEM PIM architecture does not support any spatial multitasking features, such as intra-rank multitasking for different programs, the UPMEM system returns an allocation error if there are not enough PIM resources available for the requests.

The UPMEM communication APIs provide three types of data transfer methods between the host-side main memory and the PIM-side MRAM. The serial method transfers data sequentially to each PIM unit, while the parallel method transfers concurrently. Both methods support bidirectional data transfer. The broadcast is a specialized form of the parallel method that supports unidirectional transfer from the host to entire PIM units. Because current PIM devices are typically used as a new accelerator rather than as main memory, they require explicit data transfers between host-side main memory and PIM-side memory. Parallel transfers can overlap data transfers up to the available bandwidth, whereas serial transfers incur higher costs as more resources are used.

In a PIM system, the performance of each application is affected by several factors, including the type of data transfer, the ratio of kernel to data transfer, and the program flow. As a result, the performance does not always improve as more resources are allocated, and the resource requirements vary depending on different application characteristics.

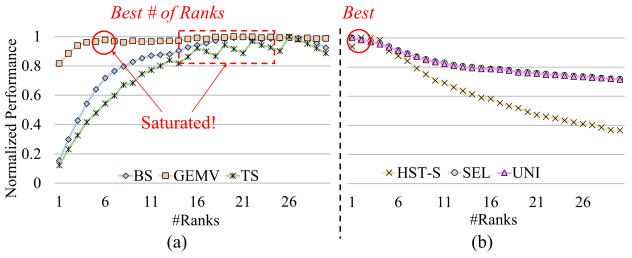

Figure 2: The normalized performance of the PrIM benchmarks [11] with increasing number of ranks.

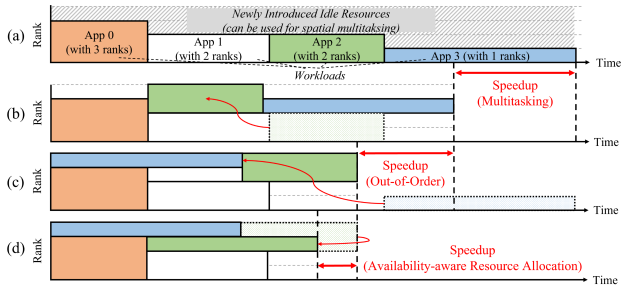

Figure 3: High-level insights for dynamic resource allocation. (a) Allocate the best number of ranks for each application, (b) spatial multitasking support, (c) out-of-order, and (d) availability-aware resource allocation.

2.2 PIM Application Characterization

When considering PIM architectures, it's easy to think that using more resources will improve performance by exploiting more parallelism and bandwidth, but this is not a simple problem. According to the analysis of the PrIM benchmark [11], the execution time of kernel tasks generally decreases in proportion to the amount of resources used. However, since current PIM systems do not fully replace traditional main memory, explicit data transfers between host-side memory and PIM-side memory are required. Thus, while the computational load per PIM core is reduced by using more PIM resources, the data transfer does not scale with proportionally because the host-PIM memory bandwidth does not improve as much as the increasing PIM resources. Depending on the application, some duplicate data transfers to each PIM core may also be required, leading to performance degradation when more PIM resources are used. Given this complexity, using more PIM resources does not always lead to performance improvements from a full system perspective [7]. The performance variation with different resources is influenced by various factors, such as the algorithm of the task and the data access patterns. However, it can generally be categorized into two types: 1) applications that improve performance with more resources and 2) others.

Figure 2 shows the performance changes depending on the resource usage in applications from the PrIM benchmark. The X-axis shows the number of PIM ranks used, and the Y-axis shows the relative performance normalized to the maximum performance. In Figure 2 (a), some applications show overall performance improvements as both kernel execution and data transfer time decrease with increasing number of PIM ranks. However, because data transfer does not improve directly in proportion to resource usage, different patterns of performance improvement are shown depending on the ratio of kernel execution to data transfer. In addition, due to limitations on the amount of data that can be distributed across PIM ranks based on the target algorithm, performance tends to saturate at a certain point ('Saturated' in Figure 2).

On the other hand, some applications, as shown in Figure 2 (b), may not show performance improvements as the number of PIM ranks increases, and may even experience performance degradation. For example, if an application involves excessive redundant data transfers, the data transfer size will increase as more PIM resources are used. Furthermore, as discussed in Section 2.1, applications with numerous serial copy operations, such as SpMV, are unable to fully leverage memory bandwidth. Therefore, for these workloads, there is a trade-off between the performance gains from distributing tasks and the transfer costs affected by data access patterns and bandwidth limitations. Consequently, each workload has a consistent performance trend based on resource utilization.

In summary, using more resources does not always improve performance in PIM systems. There is also the best amount for each application that will maximize performance based on its characteristics. It is therefore important to understand the performance variation and use the best amount of resources for each application to minimize latency.

2.3 Opportunity for Efficient Scheduling to Support Spatial Multitasking

As shown in Figure 3 (a), running all applications with the best amount of resources not only improves the total system performance but also provides new opportunities for spatial multitasking with the remaining resources in multi-tenant environments. In Figure 3 (b), resource utilization can be increased by running other applications on the remaining resources in parallel, resulting in improved overall system throughput (App2). To achieve this, it is essential to support resource monitoring similar to that used in VirtualPIM [21].

Even with spatial multitasking support, the remaining resources are often insufficient to meet the resource requirements of the next applications. First-come, first-served (FCFS) scheduling fairly controls multiple applications, but applications that require more resources than currently available must wait to be executed. Since this also causes resource requests from later applications to be pending, even though some of them could be executed with current resources,

FCFS cannot maximize the speedup from spatial multitasking. However, as shown in Figure 3 (c), running applications that are satisfied with current resources first can maximize utilization and improve system performance (App3). Thus, it is required for the workload scheduler to consider multiple waiting applications and initiate them in an OoO manner.

2.4 Motivation for Availability-aware Resource Allocation

As mentioned in Section 2.3, PIM resource utilization can be improved by supporting spatial multitasking and OoO-based workload scheduling. However, under the constraint that all applications must secure the best amount of resources, the total PIM resources may not be fully utilized. For example, if the available resources do not satisfy the resource requirements of all applications to be scheduled, the resource will remain idle until an application starts its execution by securing sufficient resources. While all applications can achieve a minimum execution time under this constraint, it is worth investigating opportunities to further improve overall system performance by improving overall resource utilization.

In a multi-tenant environment, the latency of an application includes not only execution time but also waiting time for resources to be allocated. If the waiting time for resources is greater than the speedup due to the best resource allocation, always allocating the best resources would degrade the overall system performance. In this case, it would be better for the application to be executed immediately with the available resources, which can improve system throughput by allowing more applications to run simultaneously. Therefore, if an application is expected to be completed earlier than running with the best resources after some waiting time, it is beneficial to immediately execute with the currently available resources, as shown in Figure 3 (d) (App2).

However, predicting at runtime whether allocating insufficient resources to an application is beneficial is difficult and can introduce significant control overhead. In current PIM architectures, traditional scheduling policies, such as the context switching for CPUs/GPUs, are infeasible. Therefore, it is appropriate to prioritize applications that are expected to have the least performance degradation compared to their peak performance when running with available resources. In this priority-based scheduler, since starvation problems can occur, it is important to avoid them while minimizing time loss and ultimately improving system performance.

2.5 Summary and Insights

In this section, we have learned some important lessons. First, since PIM applications show different performance trends based on resource usage, it is important to allocate the best amount of resources to each application to maximize its performance. Second, since each application has a different resource requirement, it is necessary to support spatial multitasking to enable efficient execution of multiple applications. Finally, even with spatial multitasking, resources cannot be fully utilized because target applications cannot always allocate their best amount of resources. Therefore, it is required to maximize the resource utilization by introducing OoO-based workload scheduling and availability-aware resource allocation. In this paper, we propose a novel PIM workload scheduling framework to incorporate all these insights.

3 PIM-CARE: Compiler-Assisted Dynamic Resource Allocation Framework

We propose the **PIM-CARE** framework to improve overall performance through spatial multitasking and dynamic resource allocation. As shown in Figure 4, PIM-CARE follows three steps. First, PIM-CARE replaces the PIM resource allocation and deallocation codes in existing applications with the PIM-CARE APIs customized for dynamic resource allocation and deallocation (Figure 4 (a)). Second, PIM-CARE profiles or takes user-provided information of newly launched applications to construct the performance table, and determines the best amount of resources (#ranks$_{best}$ [1]) for each application based on this table (Figure 4 (b)). Finally, the runtime daemon controls spatial multitasking by managing dynamic resource allocation requests via the PIM-CARE API from multiple applications through a task queue to allocate near-best resources to each workload (Figure 4 (c)). Here, taking into account the #ranks$_{best}$ of some requests and the available resources, PIM-CARE updates the priority of each request that can be allocated resources at the same time, and allocates resources to the higher priority requests among them (Figure 4 (d)). In summary, PIM-CARE achieves superior throughput performance by dynamically allocating the near-best number of ranks to workloads. PIM-CARE can further improve overall performance, including both throughput and turnaround time, by introducing a simple but effective priority-based scheduling policy.

3.1 Host Code Modification & Custom PIM control APIs

PIM-CARE modifies the default PIM resource allocation and deallocation operations to custom APIs that can dynamically allocate the best number of ranks based on PIM resource utilization. Specifically, the PIM-CARE compiler identifies the standard PIM resource allocation and deallocation API calls (*dpu_alloc()* and *dpu_free()* in the UPMEM SDK) in the application host code and automatically replaces

[1]For clarity, we have used the term "rank" to mean the smallest allocatable resource unit.

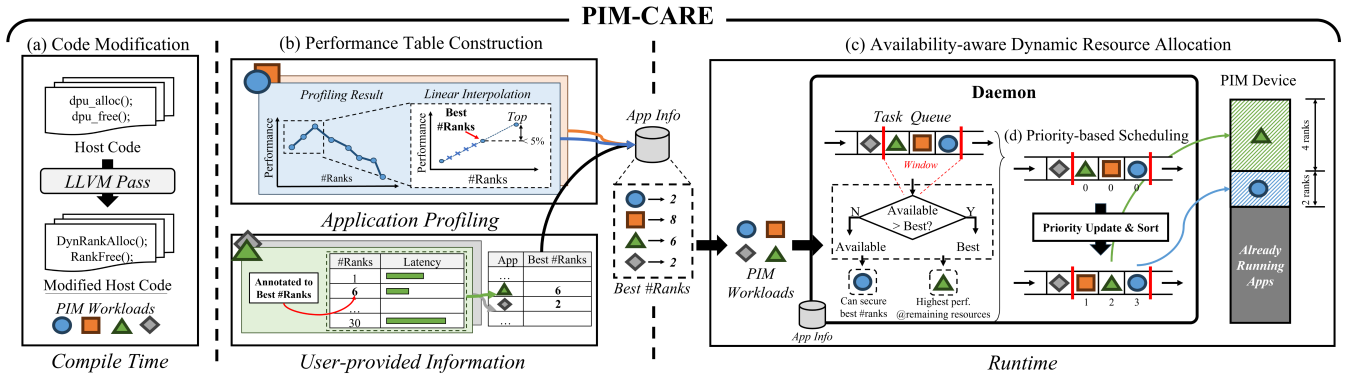

Figure 4: An overview of the PIM-CARE framework. (a) Code modification, (b) performance table construction, (c) availability-aware dynamic resource allocation, and (d) priority-based scheduling.

them with PIM-CARE API calls (*DynRankAlloc()* and *RankFree()*). In PIM-CARE, the dynamic resource allocation (*DynRankAlloc()*) determines the best number of PIM ranks to allocate by considering both $\#ranks_{best}$ and $\#ranks_{available}$, regardless of predefined resource requests in the original program code. As a result, *DynRankAlloc()* dynamically adjusts the number of PIM ranks to use, rather than failing to execute if $\#ranks_{available}$ is less than the original request, as mentioned in Section 2.1. In our implementation, both the *DynRankAlloc()* and *RankFree()* APIs communicate with the runtime daemon. Therefore, *DynRankAlloc()* requests the amount of available PIM resources to the daemon and assigns PIM ranks as the return value. On the other hand, *RankFree()* de-allocates the used ranks and informs the daemon. We have implemented the PIM-CARE compiler as an LLVM pass, following an approach similar to CASE [4]. The original code is automatically modified by the custom pass during compilation of the PIM application. Both *DynRankAlloc()* and *RankFree()* are implemented as a custom library, and linked to the target executable during compilation.

3.2 Performance Table Construction using Profile and User-provided Information

PIM-CARE introduces a profile-based method to predict the execution time of each application on the UPMEM PIM system. Unlike CPUs or GPUs, the current PIM architecture lacks complex elements like cache structures that significantly affect runtime performance. Thus, performance variation of PIM applications based on resource usage is fairly consistent, and it can be estimated with reasonable accuracy using a profiling method. Furthermore, since the UPMEM PIM architecture does not allow simultaneous access of multiple applications to the same PIM resources, the uncertainty in performance prediction is minimized.

Based on this insight, PIM-CARE profiles performance variations according to the number of PIM ranks used for each application and determines $\#ranks_{best}$. When a new application is detected, PIM-CARE performs runtime profiling and constructs a performance table for the application. Based on this, PIM-CARE analyzes performance improvement patterns and estimates the best number of ranks ($\#ranks_{best}$) for optimal performance. To reduce profiling overhead, PIM-CARE only runs profiling once for each application through a subset of the total possible options. In our implementation, we set 20% of the possible options as profiling points at regular intervals, rather than profiling the entire combinations. We use linear interpolation over the nearest profiling points to estimate performance for non-profiled points.

As mentioned in Section 2.2, allocating more resources to applications that have been insufficiently allocated offers a greater opportunity to improve system performance than allocating them to applications that already exceed the saturation point. To do this, PIM-CARE applies a threshold when determining $\#ranks_{best}$, selecting the minimum number of ranks from several candidates with comparable performance.

Algorithm 1 shows the profiling-based $\#ranks_{best}$ determination process. It begins by determining the profiling steps for a given maximum number of PIM ranks available on the system (lines 1-3). For each profiling point, PIM-CARE measures the performance of the target workload and updates the performance table (lines 4-7). To predict the performance of non-profiled points, PIM-CARE performs a linear interpolation based on the collected results (line 8). Then, PIM-CARE normalizes the performance table between 0 and 1 based on the best performance and analyzes the table to look for $\#ranks_{best}$, considering the performance saturation trend (lines 9-17). In our implementation, we set THOpt to 95%.

In addition, PIM-CARE can directly take a user-provided performance table, including annotated $\#ranks_{best}$, to provide a way to more accurately predict performance. Profiling information can be inaccurate when the workload behaves differently depending on the size or shape of the

Algorithm 1 Profile-based Best #Ranks Determination

Require: PIMWorkload w, ProfilingRatio ProfRatio
Ensure: $\#ranks_{best}$, PerformanceTable PerfTable
Let MaxRanks be the maximum number of ranks available.
Let ProfRatio be the ratio of profiling points.
Let THOpt be the performance threshold for $\#ranks_{best}$.

```
Step ← int(1/ProfRatio)
ProfPoints ← [i for i in range(1, MaxRanks, Step)]
ProfPoints.append(MaxRanks)
for Point in ProfPoints do
    Perf ← profile(w, Point)
    PerfTable.AddResult(w, Point, Perf)
end for
PerfTable.LinearInterpolation()
List_best ← []
PerfOpt ← max([PerfTable.RetPerf(w, ProfPoints)])
PerfTable.Normalize(w, PerfOpt)
for item in PerfTable.RetTable(w) do
    if item.Perf > THOpt then
        List_best.append(item.NumRank)
    end if
end for
#ranks_best ← min(List_best)
```

input. Most workloads running on PIM show similar trends in performance change regardless of the input, but for input-dependent applications, profiling cannot be applied. In such cases, the user must manually provide the workload-specific information, such as a performance table with $\#ranks_{best}$ annotations, and PIM-CARE offers an interface for this.

3.3 Availability-aware Dynamic Resource Allocation

PIM-CARE supports availability-aware dynamic resource allocation to improve overall system performance while fully utilizing all PIM resources. The resource monitoring system tracks the availability status of PIM resources, and the priority-based scheduler allocates near-best resources to resource requests and updates resource availability. We implement a priority-based workload strategy to maximize performance while avoiding starvation.

Resource Monitoring System: PIM-CARE includes a centralized resource monitoring daemon to support spatial multitasking on PIM systems. The daemon communicates with each application via custom APIs (introduced in Section 3.1). When an application requests resources, the daemon returns the number of ranks determined by the scheduler, considering both the number of ranks available $\#ranks_{available}$ and $\#ranks_{best}$ of the application. To achieve this, the daemon tracks all resource allocation and deallocation requests (*DynRankAlloc()* and *RankFree()*), and maintains $\#ranks_{available}$ at runtime. Through this, PIM-CARE

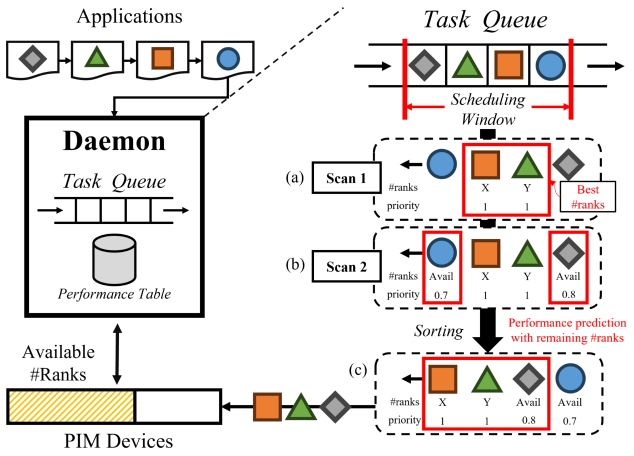

Figure 5: An example of the dynamic resource allocation process in PIM-CARE. (a) Finding the set of requests that can secure $\#ranks_{best}$, (b) performance estimation for the remaining requests, and (c) dynamic resource allocation.

supports software-level spatial multitasking in the PIM system without hardware support.

Priority-based Workload Scheduler: PIM-CARE sets the priority of each resource request and handles the highest priority request first. For this, the scheduler performs two scans of the scheduling window, which is the front part of the task queue. Figure 5 shows an example of the dynamic resource allocation process using priority information in PIM-CARE. In Figure 5 (a), the scheduler first traverses the task queue in order, identifies a combination of applications that can be allocated $\#ranks_{best}$ within $\#ranks_{available}$, and immediately assigns them the highest priority. In Figure 5 (b), applications that did not receive priority in the first scan are assigned lower priority values based on their expected relative performance with the remaining resources from the first round. After these two scans, the scheduler sorts the requests in the window in order of higher priority and dynamically allocates resources starting at the front of the window, as shown in Figure 5 (c). When resource requests from the window are allocated and removed from the queue, the window is updated to cover the remaining pending requests. As a result, PIM-CARE maximizes resource utilization and parallelism, further enhancing overall performance.

Strategy for Starvation Avoidance: In the priority-based scheduler, lower-priority applications may experience prolonged pending states, leading to starvation. For example, applications with a large $\#ranks_{best}$ are likely to have a lower priority with a small $\#ranks_{available}$, so when applications with smaller $\#ranks_{best}$ are continuously enqueued in the task queue, their scheduling would be delayed. This increases the turnaround time of the applications, leading to a degradation in overall system performance. Therefore, PIM-CARE

Algorithm 2 Workload Scheduling Algorithm

Require: PIMWindow w, PerformanceTable PerfTable, $\#ranks_{available}$
Ensure: PIMWindow wt

```
1: wt ← []
2: #ranks_temp ← #ranks_available
3: for app in w do
4:     #ranks_best ← PerfTable.RetBestRanks(app)
5:     if #ranks_temp ≥ #ranks_best then
6:         app.priority ← app.priority + 1
7:         #ranks_temp ← #ranks_temp - #ranks_best
8:         wt.append(app)
9:         w.erase(app)
10:    end if
11: end for
12: for app in w do
13:    if #ranks_temp > 0 then
14:        app.priority ← app.priority +
           PerfTable.RetPerf(app, #ranks_temp)
15:    end if
16:    wt.append(app)
17:    w.erase(app)
18: end for
19: wt.sort()
20: for app in wt do
21:    #ranks_best ← PerfTable.RetBestRanks(app)
22:    if #ranks_available ≥ #ranks_best then
23:        #ranks ← #ranks_best
24:    else
25:        #ranks ← #ranks_available
26:    end if
27:    if #ranks > 0 then
28:        launch(app, #ranks)
29:        #ranks_available ← #ranks_available - #ranks
30:        wt.erase(app)
31:    end if
32: end for
```

avoids the starvation problem by retaining the priorities from the previous scheduling cycle and further reduces the average response time of applications.

Algorithm 2 describes the priority-based workload scheduling algorithm including the starvation avoidance strategy. The algorithm stores $\#ranks_{available}$ in $\#ranks_{temp}$ to assign priorities over two scans, and initializes a space to store resource requests with updated priorities (line 1-2). It then increases the priority by 1 for each request where $\#ranks_{best}$ is less than or equal to $\#ranks_{temp}$ in the window, and decreases $\#ranks_{temp}$ by its $\#ranks_{best}$ (line 3-11). For the remaining resource requests, their priorities are set according to the relative performance predicted by $\#ranks_{temp}$, compared to the best performance (line 12-18). As mentioned in 3.2, the performance table of PIM-CARE returns values greater than 0 and less than 1, except for the minimum execution time. After two scans, the window is sorted by high priority (line 19). It then allocates $\#ranks_{best}$ to resource requests within the sorted window if it can, otherwise it allocates $\#ranks_{available}$ (lines 20-32). To prevent starvation, requests that have not been allocated resources are carried over to the next scheduling cycle as input, maintaining their priority.

3.4 A Running Example of PIM-CARE

Figure 6 shows a working example of multiple workloads running with spatial multitasking on the PIM-CARE framework. In this section, we demonstrate dynamic rank allocation and a priority-based scheduling policy, along with starvation avoidance. First, as shown in Figure 6 (a), applications request dynamic rank allocation, and their initial priorities in the scheduling window are set to 0. PIM-CARE begins by traversing the window from head, identifying the combination of requests that can be allocated $\#ranks_{best}$, and adds a priority of 1 to them. It then adds the normalized performance from the performance table with $\#ranks_{available}$ to the priority for the remaining requests. In Figure 6 (b), the requests in the scheduling window are sorted by priority, with higher priority requests appearing first. PIM-CARE then begins dynamic rank allocation for these sorted requests until $\#ranks_{available}$ reaches 0. If the $\#ranks_{best}$ for the request at the front of the window is smaller than or equal to $\#ranks_{available}$, PIM-CARE assigns $\#ranks_{best}$ to that request; otherwise, it allocates the remaining $\#ranks_{available}$. Requests that do not receive any ranks retain their priorities and wait until ranks are available. As shown in Figure 6 (c), 6 ranks are released by another application's *RankFree()*. Based on updated $\#ranks_{available}$, PIM-CARE performs priority update again. Requests that did not receive resources in the previous scheduling cycle add the new priority to the previous one, regardless of whether they can be allocated $\#ranks_{best}$ or not. In Figure 6 (d), the window is sorted, and dynamic rank allocation begins again. Even if there is a request within the window that can be assigned $\#ranks_{best}$, the front of the window has the highest priority and is assigned all 6 remaining ranks.

3.5 Compatibility with Other PIM Design

PIM-CARE is designed for UPMEM PIM, one of the few publicly available PIM architectures. PIM-CARE can also be adapted to other PIM architectures with minimal modifications, as most PIM architectures share the main characteristics addressed in this paper. For example, both Samsung's HBM-PIM [29] and SK Hynix's AiM [12, 27] allow PEs to access memory only within the same PIM unit. Therefore, depending on the application, duplicate data transfers to multiple PIM units or data transfers between PIM units can

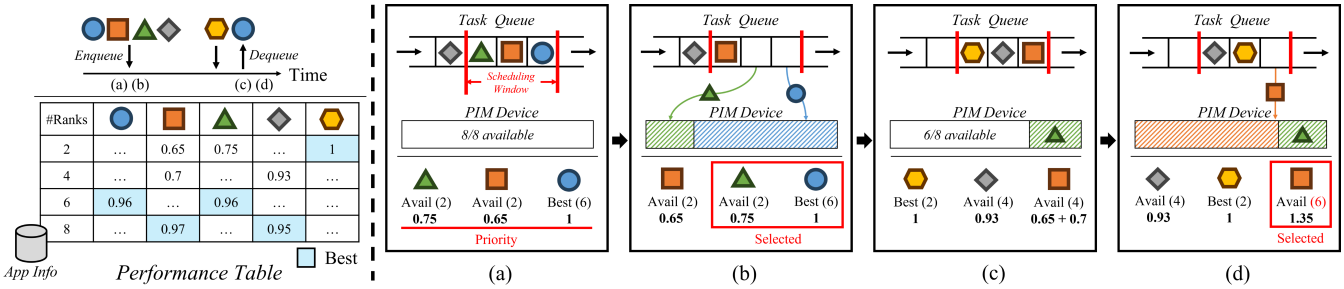

Figure 6: A running example of PIM-CARE.

Table 1: System configuration

System	Intel Server System (2 sockets, 24 DIMM slots)
CPU	Intel Xeon Gold 5222 (3.8GHz) 2 EA
Main Memory	64GB DDR4 DRAM 4 EA (256GB)
PIM Memory	8GB DDR4 PIM Modules 20 EA (160GB)
PIM Module	64MB MRAM with DPU (450MHz) 128 EA (8GB)
UPMEM Driver	2021.4.0

Table 2: The benchmarks and workload sets

Group 1		Group 2	
Apps	$\#ranks_{best}$	Apps	$\#ranks_{best}$
BS	11	BFS	1
GEMV	6	HST-L	6
MLP	21	HST-S	1
TS	16	RED	1
		SCAN-RSS	1
		SCAN-SSA	1
		SEL	1
		SpMV	1
		VA	1
		UNI	1

Workload Set	W1	W2	W3	W4	W5
Group1:Group2	1:0	2:1	1:1	1:2	0:1

cause performance degradation as more resources are used. In summary, the main idea of PIM-CARE, finding optimal resource usage and supporting multitasking, can bring significant performance gains in any PIM architecture.

4 Evaluation

4.1 Experimental Setup

System Configuration: We evaluate PIM-CARE and several approaches on the UPMEM system. The detailed configuration is listed in Table 1. The UPMEM system consists of a maximum of 40 PIM ranks, each containing 64 DPUs. However, in real PIM machines, some ranks have fewer than 64 available DPUs. Thus, we use 30 fully functional ranks, assuming that all PIM ranks have the same compute capability.

Workloads: We evaluate PIM-CARE with 14 PIM applications from the PrIM benchmark [11]. As shown in Table 2,

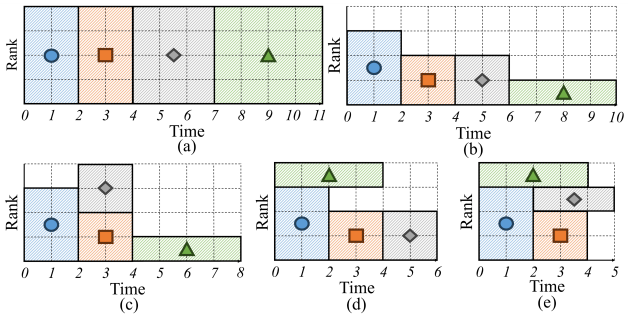

Figure 7: Scheduling examples of (a) SE-Full, (b) SE-Opt, (c) SM-Opt (FCFS), (d) SM-Opt (OoO), and (e) PIM-CARE.

we classify them into two groups based on their performance variation trends depending on the amount of PIM resources. [2] Group 1 consists of applications that show improved performance as more PIM resources are utilized, reaching optimal performance at a higher value. Group 2 consists of applications that achieve best performance at lower amount of resources, with performance degradation occurring when additional resources are used beyond that point. Thus, although GEMV and HST-L have the same $\#ranks_{best}$ of 6, GEMV is classified into Group 1, while HST-L is categorized into Group 2. We create five mixed workload sets with different ratios of applications from both groups to evaluate PIM-CARE in various scenarios as listed in Table 2. Each workload set was completely randomized according to the given ratio for each experiment.

Methodology: To fairly evaluate PIM-CARE with different scheduling approaches, we implemented four different policies: SE-Full, SE-Opt, SM-Opt (FCFS), and SM-Opt (OoO). **SE-Full** allocates all available PIM resources to applications and executes them sequentially. **SE-Opt** allocates $\#ranks_{best}$ to each application based on profile-based resource determination as described in Section 3.2, while still running applications sequentially. **SM-Opt (FCFS)** allocates $\#ranks_{best}$ to each application on a FCFS basis and runs the applications simultaneously using spatial multitasking. **SM-Opt (OoO)** is similar to SM-Opt (FCFS), but it searches for another task

[2] $\#ranks_{best}$ for MLP was determined by user-provided information.

that can be scheduled if the requirements of next application cannot be met. SM-Opt (FCFS) and SM-Opt (OoO) do not support dynamic resource allocation in PIM-CARE, which limits their ability to fully utilize PIM resources. **PIM-CARE** incorporates all the proposed techniques, including availability-aware dynamic resource allocation with the priority-based workload scheduler. We set the size of the scheduling window to 6, which shows a reasonable performance empirically. Figure 7 shows the scheduling examples of the approaches. We compared PIM-CARE and other approaches using throughput and average turnaround time (ATT).

4.2 Performance Evaluation

Figure 8 shows the throughput and ATT of PIM-CARE and other approaches. A total of 24 applications were used for each workload set and all results were normalized to the geometric mean of SE-Full. The Y-axis in Figure 8 (a) shows throughput, while in Figure 8 (b) shows ATT. Figure 9 shows the scheduling timelines of each approaches. In these figures, each workload is shown as a box, with the X-axis and the Y-axis representing time and occupied ranks, respectively.

Throughput: Figure 8 (a) shows that SE-Opt, SM-Opt (FCFS), SM-Opt (OoO), and PIM-CARE achieve throughput improvements of 1.23x, 3.82x, 4.09x, and 5.49x over SE-Full, respectively. In particular, PIM-CARE achieves significant throughput improvements on all workload sets. These results prove that our solution is highly effective and generally applicable across different application domains.

The throughput improvements are primarily achieved through best resource allocation, spatial multitasking with OoO scheduling, and availability awareness. Best resource allocation primarily improves throughput by reducing the execution time of each application with less resource consumption. Figure 9 (b) indicates that SE-Opt achieves higher throughput by reducing the overall execution time, despite not fully utilizing all available resources. Spatial multitasking can further improve throughput by utilizing the remaining resources from the best resource allocation results. As shown in Figure 9 (c), SM-Opt (FCFS) improves throughput by utilizing idle resources by enabling spatial multitasking. SM-Opt (OoO) achieves more throughput improvements than SM-Opt (FCFS) for all workload sets except W5. This is because applications with lower resource requirements can, in most cases, be allocated the best resources immediately if resources are available. Therefore, for W5, OoO scheduling becomes less effective, although it incurs additional scheduling overhead. Availability-aware dynamic resource allocation maximizes throughput by allowing more applications to be spatially multi-tasked. The performance difference between SM-Opt (OoO) and PIM-CARE implies that availability-aware resource allocation is critical to achieve the best throughput. Similarly, there is no additional performance improvement in W5 because resources are already fully utilized, similar to SM-Opt (OoO).

Turnaround Time: Figure 8 (b) shows that SE-Opt, SM-Opt (FCFS), SM-Opt (OoO), and PIM-CARE improve ATT by 1.25x, 3.85x, 5.45x, and 5.71x, respectively. These results indicate that all the proposed techniques, including best resource allocation, spatial multitasking, OoO policy, and availability-aware resource allocation, are effective in reducing ATT. More specifically, the performance difference between SE-Opt and SM-Opt (FCFS) indicates that spatial multitasking is more important in reducing ATT. The introduction of availability-aware resource allocation was not effective in W4. This is due to the loss of execution time for applications allocated insufficient resources is often greater than the corresponding reduction in waiting time.

4.3 Extensibility

Figure 8 (c) shows the performance scalability of PIM-CARE as the number of applications increases to 12, 24, and 36. Each workload set was randomly generated for the given number of applications, and the results were normalized to SE-Full for each number of applications. In this scenario, when all applications in the workload are launched, the total resource requirements increase as the number of applications increases. As shown in the figure, PIM-CARE shows improvements in throughput from 4.23x to 6.16x and in ATT from 4.77x to 5.91x as the number of applications increases from 12 to 36. However, under the same conditions, SM-Opt shows scalable improvements only in ATT and not in throughput. This result indicates that key techniques in PIM-CARE are essential for achieving high throughput, especially under heavy resource pressure. In summary, this experiment proves that PIM-CARE offers scalable performance gains with more applications, both for throughput and ATT.

4.4 Overhead Analysis

As described in Section 3.3, the runtime daemon is responsible for determining the amount of resources during dynamic resource allocation, while each application is responsible for the actual allocation and deallocation of resources with a priority-based scheduling policy. Therefore, the only additional cost comes from inter-process communication (IPC) with the daemon and its decision-making process. Figure 10 shows the overhead of PIM-CARE. Most of the overhead comes from IPC access to POSIX shared memory for communication with the daemon, while the scheduling overhead is negligible. PIM-CARE executes workloads natively without any additional operations for data consistency or task management. Thus, unlike VirtualPIM, which incurs up to 7%

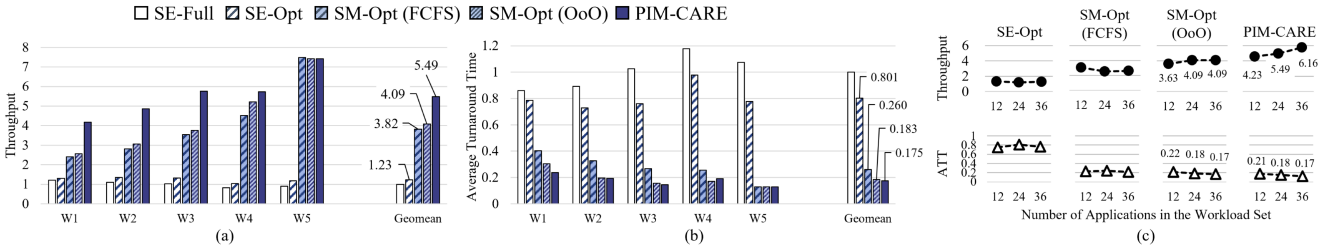

Figure 8: The overall performance of PIM-CARE and other approaches: (a) throughput, (b) ATT, and (c) extensibility.

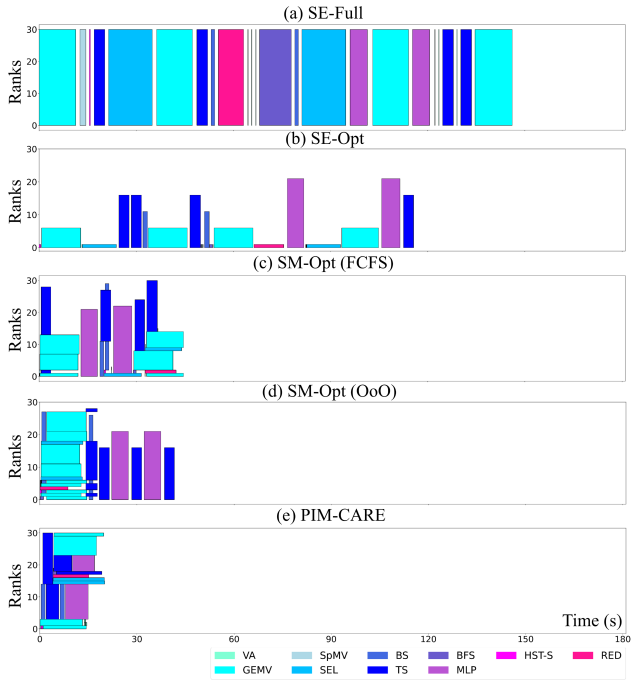

Figure 9: Scheduling timeline comparison for W3.

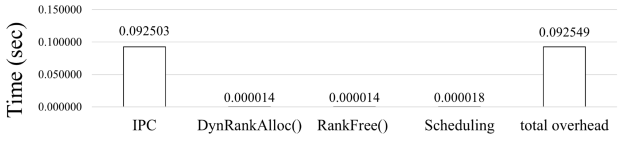

Figure 10: Overheads of PIM-CARE when performing dynamic resource allocation with OoO scheduling.

overhead depending on the workload, PIM-CARE maintains a consistent overhead regardless of the workload or input.

4.5 Comparison with VirtualPIM

We use the VirtualPIM framework to perform a quantitative comparison with PIM-CARE under the same conditions described in the original paper (realistic scenario, 14 ranks, VirtualPIM workload sets VW1–VW5 [21]). We compared throughput and ATT with VirtualPIM using all of VirtualPIM's workload sets. For comparison, we implemented

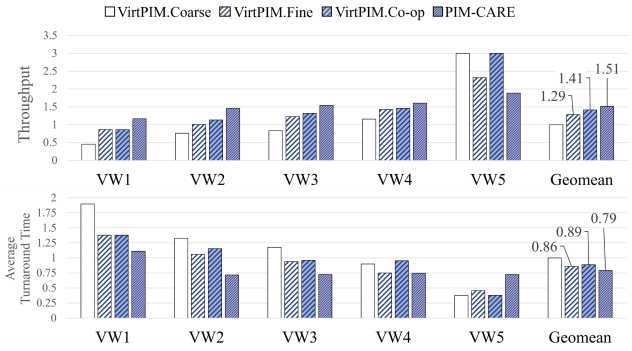

Figure 11: Performance comparison with VirtualPIM.

three different policies within VirtualPIM: VirtPIM.Coarse, VirtPIM.Fine, and VirtPIM.Co-op. **VirtPIM.Coarse** supports spatial multitasking and optimizes the DPU occupancy time of applications by deferring resource allocation through virtualization. **VirtPIM.Fine** is similar to VirtPIM.Coarse, but divides the workload into smaller units and schedules them in a fine-grained manner to maximize resource utilization. **VirtPIM.Co-op** adopts a cooperative policy to apply a different policy from Coarse and Fine to each PIM workload, to maximize performance.

Figure 11 shows the throughput and ATT of PIM-CARE and VirtualPIM. All the results are normalized to the geometric mean of VirtPIM.Coarse. Based on Figure 11, the result shows that PIM-CARE is highly effective in all workload sets except VW5@VirtPIM. This is because VW5@VirtPIM consists only of workloads with significant host operations throughout the entire PIM resource occupancy time, leading to significant performance gains due to reduced resource occupancy via the temporal optimization proposed in VirtualPIM. Except for the VW5 case, PIM-CARE is more effective for two main reasons: 1) better resource utilization with the availability-aware resource allocation and 2) no virtualization overheads with native execution support, such as additional copy operations and control of virtual-to-physical resource mapping.

Table 3: Comparison of key aspects of VirtualPIM [21] and PIM-CARE

Framework	Multitasking	Resource Allocation	Resource Opt.	Scheduling Policy	Primary Goal
VirtualPIM	Spatial	User-defined	Temporal	Fine-grained, FCFS	To Maximize Total Resource Utilization
PIM-CARE	Spatial	Profile-guided	Spatial	Availability-aware, OoO	Efficient Execution of Each Application

5 Related Works

Our prior work, VirtualPIM[21], is the first to efficiently support simultaneous execution of multiple applications on PIM. Both PIM-CARE and VirtualPIM support spatial multitasking to improve PIM resource utilization but provide different optimization techniques to achieve high performance as listed in Table 3. VirtualPIM aims to maximize PIM resource utilization through fine-grained scheduling and performs temporal optimization to minimize PIM resource occupancy per application. However, PIM-CARE determines the best amount of resources for each application, which VirtualPIM cannot do. Moreover, PIM-CARE dynamically allocates resources via a priority-based workload scheduler that considers resource availability. As a result, VirtualPIM ensures that PIM resources are fully utilized, but each application often executes sub-optimally. In contrast, PIM-CARE not only utilizes all resources, similar to VirtualPIM, but also ensures that each application operates at its maximum achievable performance.

vPIM [42] is a framework that leverages virtualization to allocate virtual machines (VMs) to PIM applications instead of physical resources, allowing them to access only the resources they require. It enables the amount of PIM resources used by each VM to change dynamically. However, it does not consider workload characteristics in resource allocation, and its virtualization overhead is relatively high (over 15% of execution time) compared to PIM-CARE, which incurs around 6%.

UPMEM PIM [7, 45], one of the well-known real-world PIM architectures available, provides high programmability and has inspired valuable research [11]. Similarly, several semiconductor industries have also been developing PIM architectures. Aquabolt-XL HBM2-PIM [23, 29] provides programmable in-memory processing and improves the overall performance. AxDIMM also optimizes recommendation frameworks using near-memory [19] scheme and performs database scan operations [28] with high efficiency. GDDR6-AiM [12, 27] accelerates matrix-vector operations using lookup table. These works focus mainly on efficient hardware development and application mapping. However, PIM-CARE emphasizes versatile mapping of multiple PIM applications across PIM hardware, making it applicable.

There have also been many studies on improving the performance of PIM applications using software approaches. SimplePIM [5] is a software framework that reduces lines of code while improving performance by providing several PIM APIs. Infinity-Stream [46] provides an intermediate representation and just-in-time compiler to support in-/near-memory fusion. CHOPPER [40] presents a compiler infrastructure that makes bit-serial processing-using-DRAM more programmable. Cinnamon [20] is a compilation framework that allows device-agnostic and device-aware optimizations on PIM. However, these works provide different optimization techniques for a single target application. Therefore, these approaches are orthogonal to the PIM-CARE framework.

6 Conclusion

We introduce PIM-CARE, a novel framework for dynamic resource allocation of PIM applications with compiler support on a real-world PIM system. PIM-CARE first determines the optimal amount of resources for each application by constructing performance tables. It then supports spatial multitasking, allowing multiple PIM applications to run concurrently with optimal resource allocation, while maximizing resource utilization through a priority-based workload scheduling. PIM-CARE also provides availability-aware dynamic resource allocation that allows PIM resources to be fully utilized. Experimental results show significant improvements in throughput and turnaround time compared to the baseline.

Acknowledgments

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant (2022-0-00441, RS-2020-II201361, RS-2021-II211343, RS-2024-00395134), by the National Research Foundation of Korea(NRF) grant (No. 2022R1A4A1032361, BK21 FOUR (Department of Computer Science and Engineering, Yonsei University)) funded by the Korea government(MSIT), and by the research fund of Hanyang University (HY-201700000002388). Yongjun Park is the corresponding author.

References

[1] Sathwika Bavikadi, Purab Ranjan Sutradhar, Mark Indovina, Amlan Ganguly, and Sai Manoj Pudukotai Dinakarrao. 2024. ReApprox-PIM: Reconfigurable Approximate Look-Up-Table (LUT)-Based Processing-in-Memory (PIM) Machine Learning Accelerator. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2024), 1–1. doi:10.1109/TCAD.2024.3367822

[2] Arthur Bernhardt, Andreas Koch, and Ilia Petrov. 2023. pimDB: From Main-Memory DBMS to Processing-In-Memory DBMS-Engines on Intelligent Memories. In *Proceedings of the 19th International Workshop on Data Management on New Hardware* (Seattle, WA, USA) *(DaMoN*

'23). Association for Computing Machinery, New York, NY, USA, 44–52. doi:10.1145/3592980.3595312

[3] Weidong Cao, Yilong Zhao, Adith Boloor, Yinhe Han, Xuan Zhang, and Li Jiang. 2022. Neural-PIM: Efficient Processing-In-Memory With Neural Approximation of Peripherals. *IEEE Trans. Comput.* 71, 9 (2022), 2142–2155. doi:10.1109/TC.2021.3122905

[4] Chao Chen, Chris Porter, and Santosh Pande. 2022. CASE: A Compiler-Assisted SchEduling Framework for Multi-GPU Systems. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 17–31. doi:10.1145/3503221.3508423

[5] 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. In *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 99–111. doi:10.1109/PACT58117.2023.00017

[6] F. Devaux. 2019. The true Processing In Memory accelerator. In *2019 IEEE Hot Chips 31 Symposium (HCS)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–24. doi:10.1109/HOTCHIPS.2019.8875680

[7] Birte Friesel, Marcel Lütke Dreimann, and Olaf Spinczyk. 2023. A Full-System Perspective on UPMEM Performance. In *Proceedings of the 1st Workshop on Disruptive Memory Systems* (Koblenz, Germany) *(DIMES '23)*. Association for Computing Machinery, New York, NY, USA, 1–7. doi:10.1145/3609308.3625266

[8] Donghyun Gouk, Miryeong Kwon, Hanyeoreum Bae, and Myoungsoo Jung. 2024. DockerSSD: Containerized In-Storage Processing and Hardware Acceleration for Computational SSDs. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 379–394. doi:10.1109/HPCA57654.2024.00036

[9] Saransh Gupta, Mohsen Imani, Harveen Kaur, and Tajana Simunic Rosing. 2019. NNPIM: A Processing In-Memory Architecture for Neural Network Acceleration. *IEEE Trans. Comput.* 68, 9 (2019), 1325–1337. doi:10.1109/TC.2019.2903055

[10] Saransh Gupta, Mohsen Imani, Behnam Khaleghi, Venkatesh Kumar, and Tajana Rosing. 2019. RAPID: A ReRAM Processing in-Memory Architecture for DNA Sequence Alignment. In *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. 1–6. doi:10.1109/ISLPED.2019.8824830

[11] 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), 52565–52608. doi:10.1109/ACCESS.2022.3174101

[12] Mingxuan He, Choungki Song, Ilkon Kim, Chunseok Jeong, Seho Kim, Il Park, Mithuna Thottethodi, and T. N. Vijaykumar. 2020. Newton: A DRAM-maker's Accelerator-in-Memory (AiM) Architecture for Machine Learning. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 372–385. doi:10.1109/MICRO50266.2020.00040

[13] Ali HeydariGorji, Siavash Rezaei, Mahdi Torabzadehkashi, Hossein Bobarshad, Vladimir Alves, and Pai H. Chou. 2022. Leveraging Computational Storage for Power-Efficient Distributed Data Analytics. *ACM Trans. Embed. Comput. Syst.* 21, 6, Article 82 (oct 2022), 36 pages. doi:10.1145/3528577

[14] Ali HeydariGorji, Mahdi Torabzadehkashi, Siavash Rezaei, Hossein Bobarshad, Vladimir Alves, and Pai H. Chou. 2020. Stannis: Low-Power Acceleration of DNN Training Using Computational Storage Devices. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 1–6. doi:10.1109/DAC18072.2020.9218687

[15] Ali HeydariGorji, Mahdi Torabzadehkashi, Siavash Rezaei, Hossein Bobarshad, Vladimir Alves, and Pai H. Chou. 2022. In-storage Processing of I/O Intensive Applications on Computational Storage Drives. In *2022 23rd International Symposium on Quality Electronic Design (ISQED)*. 1–6. doi:10.1109/ISQED54688.2022.9806270

[16] Muhammad H. Hilman, Maria A. Rodriguez, and Rajkumar Buyya. 2020. Multiple Workflows Scheduling in Multi-tenant Distributed Systems: A Taxonomy and Future Directions. *ACM Comput. Surv.* 53, 1, Article 10 (feb 2020), 39 pages. doi:10.1145/3368036

[17] Ru Jia, Yun Yang, John Grundy, Jacky Keung, and Li Hao. 2021. A systematic review of scheduling approaches on multi-tenancy cloud platforms. *Information and Software Technology* 132 (2021), 106478. doi:10.1016/j.infsof.2020.106478

[18] Qingcai Jiang, Shaojie Tan, Junshi Chen, and Hong An. 2024. A3PIM: An Automated, Analytic and Accurate Processing-in-Memory Offloader. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1–6. doi:10.23919/DATE58400.2024.10546698

[19] Liu Ke, Xuan Zhang, Jinin So, Jong-Geon Lee, Shin-Haeng Kang, Sukhan Lee, Songyi Han, YeonGon Cho, Jin Hyun Kim, Yongsuk Kwon, KyungSoo Kim, Jin Jung, Ilkwon Yun, Sung Joo Park, Hyunsun Park, Joonho Song, Jeonghyeon Cho, Kyomin Sohn, Nam Sung Kim, and Hsien-Hsin S. Lee. 2022. Near-Memory Processing in Action: Accelerating Personalized Recommendation With AxDIMM. *IEEE Micro* 42, 1 (2022), 116–127. doi:10.1109/MM.2021.3097700

[20] Asif Ali Khan, Hamid Farzaneh, Karl F. A. Friebel, Clément Fournier, Lorenzo Chelini, and Jeronimo Castrillon. 2023. CINM (Cinnamon): A Compilation Infrastructure for Heterogeneous Compute In-Memory and Compute Near-Memory Paradigms. arXiv:2301.07486 [cs.AR]

[21] Donghyeon Kim, Taehoon Kim, Inyong Hwang, Taehyeong Park, Hanjun Kim, Youngsok Kim, and Yongjun Park. 2023. Virtual PIM: Resource-Aware Dynamic DPU Allocation and Workload Scheduling Framework for Multi-DPU PIM Architecture. In *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 112–123. doi:10.1109/PACT58117.2023.00018

[22] Junkyum Kim, Myeonggu Kang, Yunki Han, Yang-Gon Kim, and Lee-Sup Kim. 2023. OptimStore: In-Storage Optimization of Large Scale DNNs with On-Die Processing. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 611–623. doi:10.1109/HPCA56546.2023.10071024

[23] Jin Hyun Kim, Shin-haeng Kang, Sukhan Lee, Hyeonsu Kim, Woongjae Song, Yuhwan Ro, Seungwon Lee, David Wang, Hyunsung Shin, Bengseng Phuah, Jihyun Choi, Jinin So, YeonGon Cho, JoonHo Song, Jangseok Choi, Jeonghyeon Cho, Kyomin Sohn, Youngsoo Sohn, Kwangil Park, and Nam Sung Kim. 2021. Aquabolt-XL: Samsung HBM2-PIM with in-memory processing for ML accelerators and beyond. In *2021 IEEE Hot Chips 33 Symposium (HCS)*. 1–26. doi:10.1109/HCS52781.2021.9567191

[24] Seah Kim, Hasan Genc, Vadim Vadimovich Nikiforov, Krste Asanović, Borivoje Nikolić, and Yakun Sophia Shao. 2023. MoCA: Memory-Centric, Adaptive Execution for Multi-Tenant Deep Neural Networks. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 828–841. doi:10.1109/HPCA56546.2023.10071035

[25] Seongguk Kim, Subin Kim, Kyungjun Cho, Taein Shin, Hyunwook Park, Daehwan Lho, Shinyoung Park, Kyungjune Son, Gapyeol Park, Seungtaek Jeong, Youngwoo Kim, and Joungho Kim. 2021. Signal Integrity and Computing Performance Analysis of a Processing-In-Memory of High Bandwidth Memory (PIM-HBM) Scheme. *IEEE Transactions on Components, Packaging and Manufacturing Technology* 11, 11 (2021), 1955–1970. doi:10.1109/TCPMT.2021.3117071

[26] Daehan Kwon, Seongju Lee, Kyuyoung Kim, Sanghoon Oh, Joonhong Park, Gi-Moon Hong, Dongyoon Ka, Kyudong Hwang, Jeongje Park, Kyeongpil Kang, Jungyeon Kim, Junyeol Jeon, Nahsung Kim, Yongkee Kwon, Vladimir Kornijcuk, Woojae Shin, Jongsoon Won, Minkyu Lee, Hyunha Joo, Haerang Choi, Guhyun Kim, Byeongju An, Jaewook Lee,

Donguc Ko, Younggun Jun, Ilwoong Kim, Choungki Song, Ilkon Kim, Chanwook Park, Seho Kim, Chunseok Jeong, Euicheol Lim, Dongkyun Kim, Jieun Jang, Il Park, Junhyun Chun, and Joohwan Cho. 2023. A 1ynm 1.25V 8Gb 16Gb/s/Pin GDDR6-Based Accelerator-in-Memory Supporting 1TFLOPS MAC Operation and Various Activation Functions for Deep Learning Application. *IEEE Journal of Solid-State Circuits* 58, 1 (2023), 291–302. doi:10.1109/JSSC.2022.3200718

[27] Yongkee Kwon, Kornijcuk Vladimir, Nahsung Kim, Woojae Shin, Jongsoon Won, Minkyu Lee, Hyunha Joo, Haerang Choi, Guhyun Kim, Byeongju An, Jeongbin Kim, Jaewook Lee, Ilkon Kim, Jaehan Park, Chanwook Park, Yosub Song, Byeongsu Yang, Hyungdeok Lee, Seho Kim, Daehan Kwon, Seongju Lee, Kyuyoung Kim, Sanghoon Oh, Joonhong Park, Gimoon Hong, Dongyoon Ka, Kyudong Hwang, Jeongje Park, Kyeongpil Kang, Jungyeon Kim, Junyeol Jeon, Myeongjun Lee, Minyoung Shin, Minhwan Shin, Jaekyung Cha, Changson Jung, Kijoon Chang, Chunseok Jeong, Euicheol Lim, Il Park, Junhyun Chun, and Sk Hynix. 2022. System Architecture and Software Stack for GDDR6-AiM. In *2022 IEEE Hot Chips 34 Symposium (HCS)*. 1–25. doi:10.1109/HCS55958.2022.9895629

[28] Donghun Lee, Jinin So, MINSEON AHN, Jong-Geon Lee, Jungmin Kim, Jeonghyeon Cho, Rebholz Oliver, Vishnu Charan Thummala, Ravi shankar JV, Sachin Suresh Upadhya, Mohammed Ibrahim Khan, and Jin Hyun Kim. 2022. Improving In-Memory Database Operations with Acceleration DIMM (AxDIMM). In *Proceedings of the 18th International Workshop on Data Management on New Hardware* (Philadelphia, PA, USA) *(DaMoN '22)*. Association for Computing Machinery, New York, NY, USA, Article 2, 9 pages. doi:10.1145/3533737.3535093

[29] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, Jinhyun Kim, O Seongil, Anand Iyer, David Wang, Kyomin Sohn, and Nam Sung Kim. 2021. Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology : Industrial Product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 43–56. doi:10.1109/ISCA52012.2021.00013

[30] Seongju Lee, Kyuyoung Kim, Sanghoon Oh, Joonhong Park, Gimoon Hong, Dongyoon Ka, Kyudong Hwang, Jeongje Park, Kyeongpil Kang, Jungyeon Kim, Junyeol Jeon, Nahsung Kim, Yongkee Kwon, Kornijcuk Vladimir, Woojae Shin, Jongsoon Won, Minkyu Lee, Hyunha Joo, Haerang Choi, Jaewook Lee, Donguc Ko, Younggun Jun, Keewon Cho, Ilwoong Kim, Choungki Song, Chunseok Jeong, Daehan Kwon, Jieun Jang, Il Park, Junhyun Chun, and Joohwan Cho. 2022. A 1ynm 1.25V 8Gb, 16Gb/s/pin GDDR6-based Accelerator-in-Memory supporting 1TFLOPS MAC Operation and Various Activation Functions for Deep-Learning Applications. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 65. 1–3. doi:10.1109/ISSCC42614.2022.9731711

[31] Yunjae Lee, Jinha Chung, and Minsoo Rhu. 2022. SmartSAGE: training large-scale graph neural networks using in-storage processing architectures. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 932–945. doi:10.1145/3470496.3527391

[32] Minje Lim, Jeeyoon Jung, and Dongkun Shin. 2021. LSM-tree Compaction Acceleration Using In-storage Processing. In *2021 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia)*. 1–3. doi:10.1109/ICCE-Asia53811.2021.9641965

[33] Jilan Lin, Ling Liang, Zheng Qu, Ishtiyaque Ahmad, Liu Liu, Fengbin Tu, Trinabh Gupta, Yufei Ding, and Yuan Xie. 2022. INSPIRE: in-storage private information retrieval via protocol and architecture co-design. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 102–115. doi:10.1145/3470496.3527433

[34] André Lopes, Daniel Castro, and Paolo Romano. 2024. PIM-STM: Software Transactional Memory for Processing-In-Memory Systems. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 897–911. doi:10.1145/3620665.3640428

[35] Vikram Sharma Mailthody, Zaid Qureshi, Weixin Liang, Ziyan Feng, Simon Garcia de Gonzalo, Youjie Li, Hubertus Franke, Jinjun Xiong, Jian Huang, and Wen-mei Hwu. 2019. DeepStore: In-Storage Acceleration for Intelligent Queries. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 224–238. doi:10.1145/3352460.3358320

[36] Nika Mansouri Ghiasi, Jisung Park, Harun Mustafa, Jeremie Kim, Ataberk Olgun, Arvid Gollwitzer, Damla Senol Cali, Can Firtina, Haiyu Mao, Nour Almadhoun Alserr, Rachata Ausavarungnirun, Nandita Vijaykumar, Mohammed Alser, and Onur Mutlu. 2022. GenStore: A High-Performance in-Storage Processing System for Genome Sequence Analysis. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 635–654. doi:10.1145/3503222.3507702

[37] Dimosthenis Masouros, Sotirios Xydis, and Dimitrios Soudris. 2021. Rusty: Runtime Interference-Aware Predictive Monitoring for Modern Multi-Tenant Systems. *IEEE Transactions on Parallel and Distributed Systems* 32, 1 (2021), 184–198. doi:10.1109/TPDS.2020.3013948

[38] Joel Mandebi Mbongue, Alexis Maya-Isabelle Shuping, Pankaj Bhowmik, and Christophe Bobda. 2020. Architecture Support for FPGA Multi-tenancy in the Cloud. In *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 125–132. doi:10.1109/ASAP49362.2020.00030

[39] Isaac Odun-Ayo, Sanjay Misra, Olusola Abayomi-Alli, and Olasupo Ajayi. 2017. Cloud Multi-Tenancy: Issues and Developments. In *Companion Proceedings of The10th International Conference on Utility and Cloud Computing* (Austin, Texas, USA) *(UCC '17 Companion)*. Association for Computing Machinery, New York, NY, USA, 209–214. doi:10.1145/3147234.3148095

[40] Xiangjun Peng, Yaohua Wang, and Ming-Chang Yang. 2023. CHOPPER: A Compiler Infrastructure for Programmable Bit-serial SIMD Processing Using Memory in DRAM. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 1275–1288. doi:10.1109/HPCA56546.2023.10071070

[41] Zhenyuan Ruan, Tong He, and Jason Cong. 2019. INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 379–394. https://www.usenix.org/conference/atc19/presentation/ruan

[42] Dufy Teguia, Jiaxuan Chen, Stella Bitchebe, Oana Balmau, and Alain Tchana. 2024. vPIM: Processing-in-Memory Virtualization. In *Proceedings of the 25th International Middleware Conference* (Hong Kong, Hong Kong) *(Middleware '24)*. Association for Computing Machinery, New York, NY, USA, 417–430. doi:10.1145/3652892.3700782

[43] Mahdi Torabzadehkashi, Siavash Rezaei, Vladimir Alves, and Nader Bagherzadeh. 2018. CompStor: An In-storage Computation Platform for Scalable Distributed Processing. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1260–1267. doi:10.1109/IPDPSW.2018.00195

[44] Mahdi Torabzadehkashi, Siavash Rezaei, Ali Heydarigorji, Hosein Bobarshad, Vladimir Alves, and Nader Bagherzadeh. 2019. Catalina: In-Storage Processing Acceleration for Scalable Big Data Analytics. In *2019 27th Euromicro International Conference on Parallel, Distributed*

and Network-Based Processing (PDP). 430–437. doi:10.1109/EMPDP.2019.8671589

[45] UPMEM. 2022. UPMEM Processing In-Memory (PIM) (Tech Paper).

[46] Zhengrong Wang, Christopher Liu, Aman Arora, Lizy John, and Tony Nowatzki. 2023. Infinity Stream: Portable and Programmer-Friendly In-/Near-Memory Fusion. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 359–375. doi:10.1145/3582016.3582032

[47] Sheng Xu, Xiaoming Chen, Xuehai Qian, and Yinhe Han. 2020. TUPIM: A Transparent and Universal Processing-in-Memory Architecture for Unmodified Binaries. In *Proceedings of the 2020 on Great Lakes Symposium on VLSI* (Virtual Event, China) *(GLSVLSI '20)*. Association for Computing Machinery, New York, NY, USA, 199–204. doi:10.1145/3386263.3406896

[48] Yuqi Xue, Yiqi Liu, Lifeng Nai, and Jian Huang. 2023. V10: Hardware-Assisted NPU Multi-tenancy for Improved Resource Utilization and Fairness *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 24, 15 pages. doi:10.1145/3579371.3589059

[49] Fuxun Yu, Di Wang, Longfei Shangguan, Minjia Zhang, Chenchen Liu, and Xiang Chen. 2022. A Survey of Multi-Tenant Deep Learning Inference on GPU. arXiv:2203.09040 [cs.DC]