First A. Author, *Fellow, IEEE*, Second B. Author, and Third C. Author, Jr., Member, IEEE

MemBo: Runtime Memory Borrowing on UPMEM-based Servers

***Abstract*— In modern data centers, the increasing popularity of in-memory computing applications, such as key-value stores and machine learning applications, has led to a higher demand for memory. Meanwhile, these applications exhibit varying memory requirements during runtime, creating challenges for servers with fixed memory capacities. Especially, When the working set of these applications exceeds the available memory, their throughput can experience significant decreases. To tackle this issue, various solutions, including memory hotplug or memory disaggregation, have been proposed. However, these solutions have limitations when it comes to addressing the specific challenges faced by UPMEM-based servers, which are equipped with innovative processing-in-memory (PIM) memory hardware and require special considerations.**

**This paper introduces MemBo, a solution designed specifically for UPMEM-based servers to effectively handle memory demand variance. MemBo enhances the memory management flexibility of UPMEM servers, resulting in significant improvements in the throughput of YCSB workloads, with enhancements of up to 3.43**×**. Additionally, MemBo achieves a remarkable reduction of major page faults, up to 99.9%, under high memory pressure. An important advantage of MemBo is its transparent operation, benefiting both server administrators and application users alike.**

***Index Terms*— memory management, memory pressure, paging**

# INTRODUCTION

UPMEM [1, 2] has introduced UPMEM PIM DIMMs, the first real-world processing-in-memory (PIM) hardware. A UPMEM PIM DIMM integrates multiple programmable 32-bit Reduced Instruction Set Computing (RISC) cores, referred to as data processing units (DPUs), into a dual-rank DDR4-2400 DIMM. In recent times, researchers have been exploring various use cases of the UPMEM PIM architecture. It has demonstrated its potential to accelerate real-life in-memory applications such as genomics [6, 7, 8], databases [3, 4], and machine learning [9, 10] applications. Additionally, benchmarking and methodologies [5] tailored specifically for this architecture have been proposed to provide a comprehensive analysis of its performance and capabilities.

The continuous advancement of big data systems and analytics has led to a significant increase in memory demand [30, 31] within modern data centers. In-memory data computing has become crucial for processing large volumes of data quickly and efficiently. As a result, there is a growing need to address the challenges posed by the ever-increasing memory requirements.

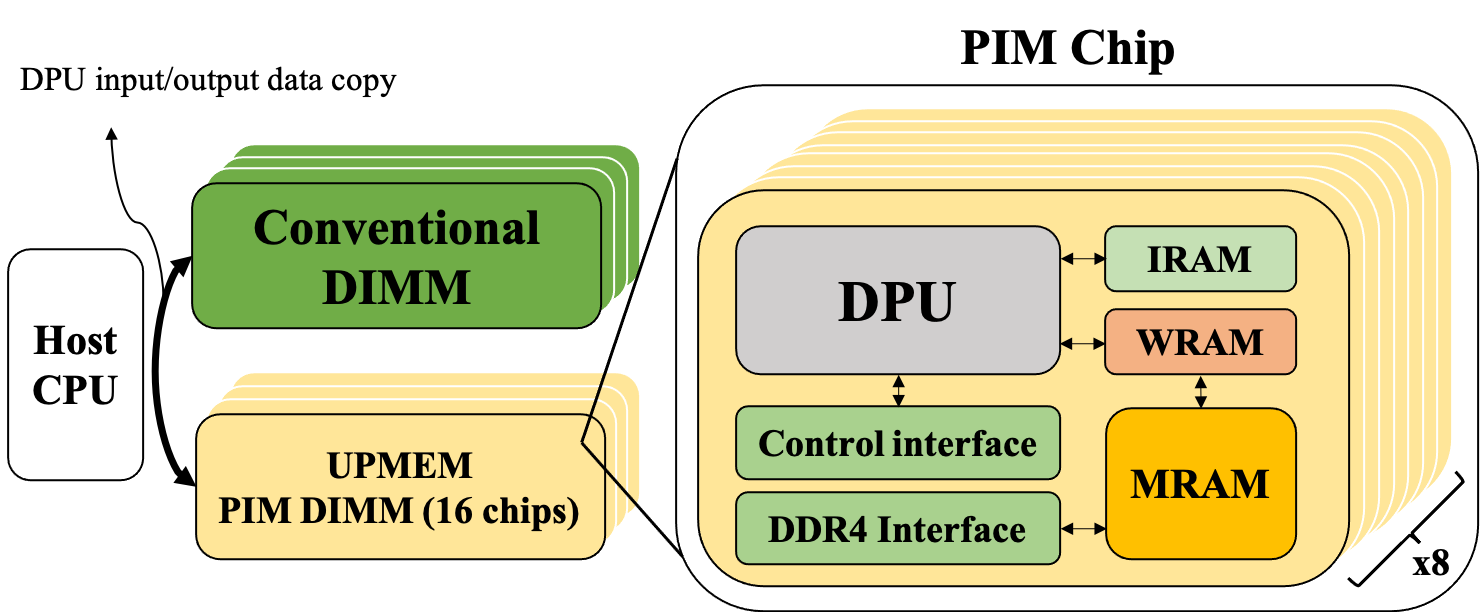


Figure 1. A UPMEM-based System (left) and the Interval Components of a UPMEM PIM Chip

One of the main challenges is the temporal variance in memory demand. Workloads and applications often exhibit varying memory needs over time [30, 31], with peak periods of high memory usage and periods of low or idle memory utilization. Traditional servers with fixed memory capacity struggle to adapt to these variations, leading to performance issues such as excessive paging. This can cause a significant performance drop as the system spends more time accessing data from slower storage devices, leading to increased latency and decreased overall system throughput. Therefore, in response to the increasing memory demand and the observed temporal variance in memory requirements, there is a need for innovative memory management techniques that can dynamically adapt to changing workload demands and prevent excessive paging, ensuring efficient and high-performance in-memory data computing in modern data centers.

To address the demand for memory variance in UPMEM-based servers equipped with UPMEM PIM DIMMs, existing methods such as memory hotplug and memory disaggregation have limitations. These methods were not specifically designed to cater to the unique characteristics of UPMEM-based servers and the specialized requirements of UPMEM PIM DIMMs.

In particular, when examining a UPMEM-based server under high memory pressure, it has been observed that the utilization of UPMEM PIM DIMMs is relatively low. This is primarily attributed to the fact that the DRAM banks on UPMEM PIM DIMMs are typically reserved for DPU acceleration purposes by the OS. This static allocation of memory resources for DPU acceleration limits the flexibility to adapt to the varying memory demands of the system. This highlights the need for a memory management solution that can effectively handle the memory demand variance while considering the specific characteristics and requirements of UPMEM PIM DIMMs. A tailored approach is required to optimize the memory utilization of UPMEM PIM DIMMs and ensure that they can efficiently adapt to the changing memory requirements of the system.

We propose MemBo, a runtime memory borrowing mechanism on UPMEM-based servers. It offers dynamic memory expansion and shrinking capabilities, allowing for flexible allocation of memory resources based on the real-time demands of the system. By intelligently borrowing memory from idle UPMEM PIM DIMMs in the servers and reallocating them to memory-intensive applications, MemBo aims to optimize memory utilization and enhance overall system performance in UPMEM-based servers.

Through its innovative design and tailored approach, MemBo aims to improve memory utilization, mitigate excessive paging, and enable efficient adaptation to changing workload demands in UPMEM-based servers equipped with UPMEM PIM DIMMs. By optimizing memory management for these specialized platforms, MemBo aims to unlock their full potential and deliver enhanced performance for memory-intensive applications.

To evaluate the effectiveness of MemBo, we integrated it into the existing UPMEM system and conducted experiments to assess its performance under memory pressure. In order to simulate a significant memory footprint that induces high memory pressure on the server, we utilized a popular key-value store application, Memcached, and a well-known benchmark set for database systems, YCSB. The experimental results clearly demonstrate the positive impact of MemBo on system performance. When faced with memory pressure, MemBo achieves notable improvements in throughput, with enhancements of up to 3.43× compared to the baseline. Additionally, MemBo effectively reduces major page faults by as much as 99.9%, indicating its ability to mitigate the performance degradation caused by excessive paging.

The remaining sections of the paper are structured as follows: Section II introduces the background of UPMEM-based platforms and the related works, laying the foundation for the discussion on MemBo. Section III highlights the motivation of MemBo. Section IV describes the design of MemBo in detail, outlining its key features and functionality. Section V presents the experimental results obtained from the implementation of MemBo, providing empirical evidence of its effectiveness. Finally, Section VI concludes the paper, summarizing the main findings of MemBo.

# Background and Related Work

## UPMEM-based Computing Platform

UPMEM launched the first real-world processing in memory (PIM) accelerator called UPMEM PIM DIMM, which integrates several programmable 32-bit Reduced Instruction Set Computing (RISC) cores, called data processing units (DPU), into a dual rank DDR4-2400 DIMM. A DPU has direct access to its associated 64-MB DRAM bank on the DIMM, bypassing the need to utilize the off-chip memory bus. This capability allows DPUs to function as co-processors alongside the CPU, effectively accelerating memory-intensive applications.

Figure 1 shows a UPMEM-based system and the interval components of a UPMEM PIM chip. A DPU is equipped with (1) an instruction RAM (IRAM) for storing the instruction information (2) an associated DRAM bank (MRAM) (3) a scratchpad memory (WRAM) serving as temporary storage for working data during computations. Besides, a DPU is controlled by the host CPU, such as launching it, through the memory-mapped control interface. To perform the DPU acceleration, the host CPU follows the following model:

1. loads the instruction information to the DPU’s IRAM
2. copies the input data to the DPU’s MRAM
3. launches the DPU execution through the control interface.
4. retrieves the result from the DPU’s MRAM

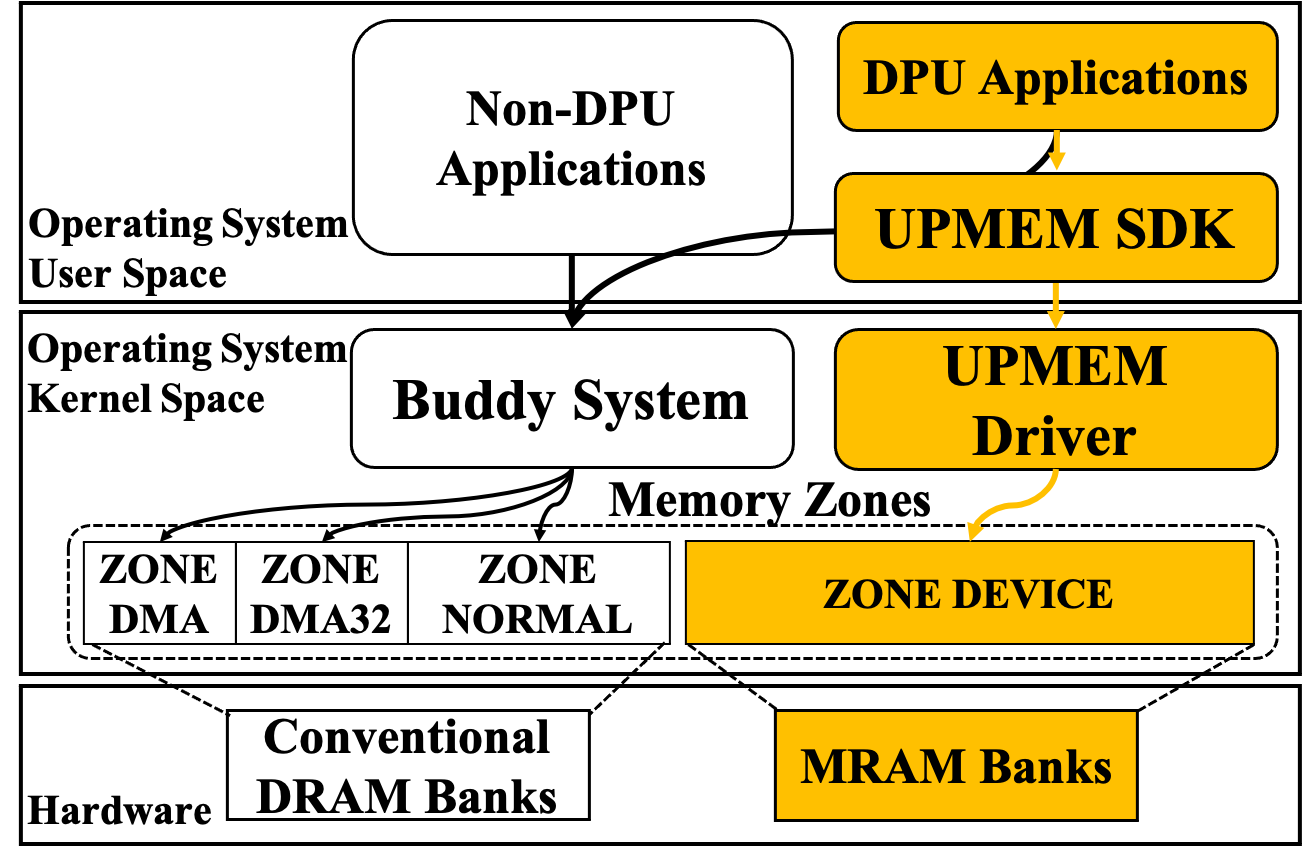


Figure 2. Memory Management of a UPMEM-based

Linux System

Due to hardware limitations, the clock frequency of a DPU (350 MHz) is much lower than that of the host CPU. As a result, real-life applications often employ multiple DPUs for acceleration. In the current DPU acceleration model, DPUs are allocated to an application in ranks, with each rank consisting of 64 DPUs."

Unlike conventional DRAM banks, MRAM banks are reserved by the OS for storing DPU input/output data throughout the system runtime. In other words, these MRAM banks are not utilized as system memory by the operating system. As elaborated in Figure 2, conventional DRAM banks are mapped into the kernel physical space and divided into multiple zones. These zones are managed by the Linux page allocator, also known as the buddy allocator. Similarly, the MRAM banks are mapped into the kernel physical space, but they reside only within specific zones called ZONE\_DEVICEs. This zone is completely hidden from the buddy allocator. MRAM banks must be accessed exclusively through the APIs provided by the UPMEM SDK [1] and the UPMEM driver [1].

## Related Work

### Virtual Machine Memory Management

Memory ballooning [12-19] is a technique employed in virtualization environments to efficiently manage memory resources by enabling the hypervisor to dynamically adjust the memory allocation of virtual machines (VMs) based on their real-time usage. In a virtualized environment, multiple VMs operate on a single physical host, with each VM assigned a specific memory allocation. However, the memory requirements of VMs can vary over time, with some VMs needing more memory during peak usage periods, while others may have idle or unused memory.

To address this, memory ballooning involves the integration of a balloon driver or module into the guest operating system of each VM. This driver acts as an intermediary between the guest OS and the hypervisor, generating artificial memory pressure within the VM by inflating a balloon within the guest's memory space.

When the host system experiences increased memory pressure, indicating a shortage of physical memory, the hypervisor instructs the balloon drivers within the VMs to inflate their balloons. As the balloons inflate, they consume memory within the VMs, effectively reducing the available memory for the VMs' processes. Consequently, the hypervisor can reclaim the memory occupied by the inflated balloons and allocate it to other VMs or processes that require it. The balloon driver maintains communication with the hypervisor, providing information on the amount of memory it has inflated. This allows the hypervisor to monitor memory usage across VMs and make informed decisions regarding memory allocation.

When memory pressure decreases, such as when other VMs release memory or the overall system memory usage decreases, the hypervisor signals the balloon drivers to deflate their balloons. This releases the occupied memory back to the VMs, making it available for their processes once again.

By dynamically adjusting memory allocation based on actual usage, memory ballooning optimizes memory utilization in virtualized environments, facilitating efficient sharing and allocation of memory resources among VMs.

We summarize the differences between memory ballooning and MemBo in three aspects:

***Targeted System and Purpose***

Memory ballooning is a technique primarily used in virtualization environments to manage memory resources within virtual machines (VMs). It aims to dynamically adjust memory allocation based on VMs' actual usage. In contrast, MemBo is specifically designed for native OS on UPMEM platforms and aims to optimize memory utilization within these platforms. This means that MemBo operates at a lower level within the system, directly interacting with the underlying OS.

***Memory Expansion and Shrinkage Mechanism***

Memory ballooning operates by inflating or deflating memory balloons within VMs, dynamically adjusting their memory usage. MemBo, on the other hand, detects memory pressure within UPMEM platforms and automatically expands or shrinks the physical memory capacity of these platforms. It integrates with the UPMEM platform's memory management system to optimize memory utilization.

***Design Issues***

Memory ballooning can introduce some performance overhead due to the inflation and deflation of memory balloons within VMs. This impact needs to be carefully managed to avoid performance degradation. MemBo, being tailored for UPMEM platforms, takes into consideration the performance requirements of DPU applications. It aims to optimize memory utilization without significantly affecting the performance of these applications.

### Memory Disaggregation and Compute Express Link (CXL)

Memory disaggregation [21-27] is a concept in computer systems architecture that aims to decouple memory resources from individual servers or nodes in a distributed system. Traditionally, each server or node in a system has its own dedicated memory, which can lead to inefficiencies in resource utilization. In a disaggregated memory architecture, memory resources are pooled together and made accessible to all servers or nodes in the system. With memory disaggregation, servers or nodes can access and utilize memory resources from the shared pool as needed, rather than being limited to the memory capacity of their individual nodes. This allows for a more flexible and efficient allocation of memory resources across the system, reducing memory underutilization and improving overall system performance.

Computing Express Link (CXL) [11] is an emerging high-speed interconnect technology designed to enable high-performance communication between processors, memory, and accelerators in a system. It is an open industry standard that provides a high-bandwidth, low-latency connection for data transfer. CXL is particularly relevant in memory disaggregation scenarios, as it facilitates communication and access to remote memory resources. It enables processors and accelerators to directly access remote memory, regardless of its physical location in the system, using a unified memory address space. This allows for efficient utilization of memory resources across distributed systems. Overall, CXL plays a crucial role in enabling memory disaggregation by providing a high-performance interconnect that allows for efficient access to remote memory resources and promotes scalable and flexible memory architectures. DirectCXL [21] proposed the first implementation of memory disaggregation based on the CXL specification 2.0.

The differences of memory disaggregation and MemBo are listed as follows:

***Scope and focus***

Memory disaggregation is a broader concept that encompasses the decoupling of memory resources from individual servers or nodes in a distributed system. It aims to create a shared memory pool that can be accessed by multiple servers or nodes. In contrast, MemBo is a specific technique or approach that focuses on optimizing memory utilization within UPMEM platforms, which contain specialized memory devices like UPMEM PIM DIMMs. MemBo is designed to work in conjunction with these specific platforms to improve memory management and performance.

***Design Issues***

Memory disaggregation can introduce increased latency and network communication overhead due to the remote access of memory resources. This impact on performance needs to be carefully managed and optimized. MemBo, being a technique tailored for UPMEM platforms, takes into consideration the impact on application performance, and aims to optimize memory utilization without significantly affecting the performance of DPU applications.

### Memory Hotplug

Memory hotplug [12] is a feature found in computer systems that allows for the seamless addition or removal of physical memory DIMM, eliminating the need for a system reboot. This feature allows server administrators to dynamically adjust the memory capacity of a running system, providing enhanced flexibility in managing memory resources.

Through memory hotplug, server administrators can insert additional memory DIMMs into unoccupied slots, effectively expanding the total available memory of the system. This capability proves particularly advantageous in situations where application or workload memory requirements increase, or when there is a need for additional memory to boost system performance. Similarly, memory hotplug also permits the removal of memory DIMMs while the system remains operational. This functionality proves valuable when troubleshooting faulty memory DIMMs or when reducing the memory capacity of a system that is no longer needed.

The successful execution of memory hotplug relies on hardware support, as well as an operating system and system firmware capable of handling the dynamic addition or removal of memory. When memory DIMMs are hotplugged, the system must promptly recognize and initialize the newly added memory or readjust its configuration to accommodate the reduced memory capacity in cases of DIMM removal. Memory hotplug finds widespread use in enterprise server environments and virtualized systems, where the ability to modify memory configurations without interrupting system operation holds significant advantages. It equips administrators with increased flexibility and scalability in managing memory resources to effectively meet evolving workload demands.

The main differences between memory hotplug and MemBo is: memory hotplug involves physically adding or removing memory DIMMs from a computer system while it is operational. This process requires the intervention of server administrators to adjust the installed DIMMs. In contrast, MemBo is an automated process that expands or shrinks the memory capacity of UPMEM platforms based on memory pressure detection. It does not involve physical intervention or the addition/removal of memory DIMMs.

# MOTIVATION

Memory is one of the most constrained resources in modern computing platforms and data centers. Especially, in-memory applications, such as key-value stores and machine learning applications, have greatly increased the demand for main memory. For instance, Google's servers have reported a 20% rise in average memory usage over recent years [30]. Besides, it has been consistently observed that popular applications demonstrate notable variations in runtime memory demand [30, 31]. Once a system exhausts its available memory, it must resort to swapping, which frees up memory space by transferring data to lower levels of memory, such as disks or remote memory. This process leads to a performance decrease. In some cases, jobs are killed due to memory pressure [30], resulting in time and energy waste.

Previously, various approaches such as memory dis-aggregation, memory ballooning, and memory hot-plug have been proposed to address this scenario. These techniques offer ways to dynamically adjust the main memory size of a virtual or physical machine in response to changing memory demands during runtime. *However, none of these methods have been specifically designed for the UPMEM platform*, which utilizes UPMEM PIM DIMMs and requires special considerations due to its unique hardware architecture.

Besides, in heterogeneous computing environments such as GPU platforms or UPMEM PIM platforms, there is also a dynamic variation in the demand for computing resources [38, 39]. For example, GPU clusters traces from Microsoft, Amazon, and Google all indicate that GPU demands exhibit bursts and that GPU utilization can fluctuate significantly over time [38, 40].

In the current UPMEM design, the OS statically reserves MRAM banks for performing CPU-DPU data exchange during system runtime. While this approach is straightforward, *it fails to effectively utilize the valuable MRAM resources to adapt to fluctuations in main memory and DPU demand*. Particularly, during periods of high memory pressure, the unused MRAM resources remain reserved and are unable to alleviate the memory pressure effectively.

In this paper, we present MemBo, a new MRAM management approach that offers enhanced flexibility. Our key idea is to leverage idle MRAM banks as temporary system memory to alleviate memory pressure when faced with high memory demands. In other words, MemBo enables the buddy system to borrow DRAM banks from DPUs, which are then reclaimed once the memory pressure is released. We claim that MemBo has the potential to substantially reduce major page faults, consequently boosting the throughput of a UPMEM-based platform.

When enabling the MRAM borrowing mechanism, we must consider the situation where the borrowed MRAM banks are demanded by their DPUs for running DPU applications. In such cases, the borrowed banks must be reclaimed immediately (called foreground reclamation in this paper). This requires all the live data remaining on the banks to be migrated to other memory spaces. Unfortunately, data migration can be excessively time-consuming, which can cause a DPU application to block for a long period while waiting for the MRAM banks to be usable by the DPUs. To address this challenge, we propose several optimizations. Which are designed to alleviate the potential delays caused by data migration and minimize the impact on DPU applications.

# Design and implementation

In this section, we introduce the design and implementation of MemBo. The objective of MemBo is to design a system-level runtime MRAM borrowing mechanism that aims to optimize the utilization of MRAM resources for UPMEM-based platforms.

## Overview of MemBo Architecture

Figure 3 illustrates the overview architecture of MemBo, highlighting the components of MemBo in green. MemBo is completely transparent to both non-DPU application and DPU application programmers, where they can seamlessly enjoy the benefit brought by MemBo. Especially, DPU applications within the MemBo environment is supported by the modified UPMEM SDK, a modified version of the UPMEM SDK 2021.3.0 [1].

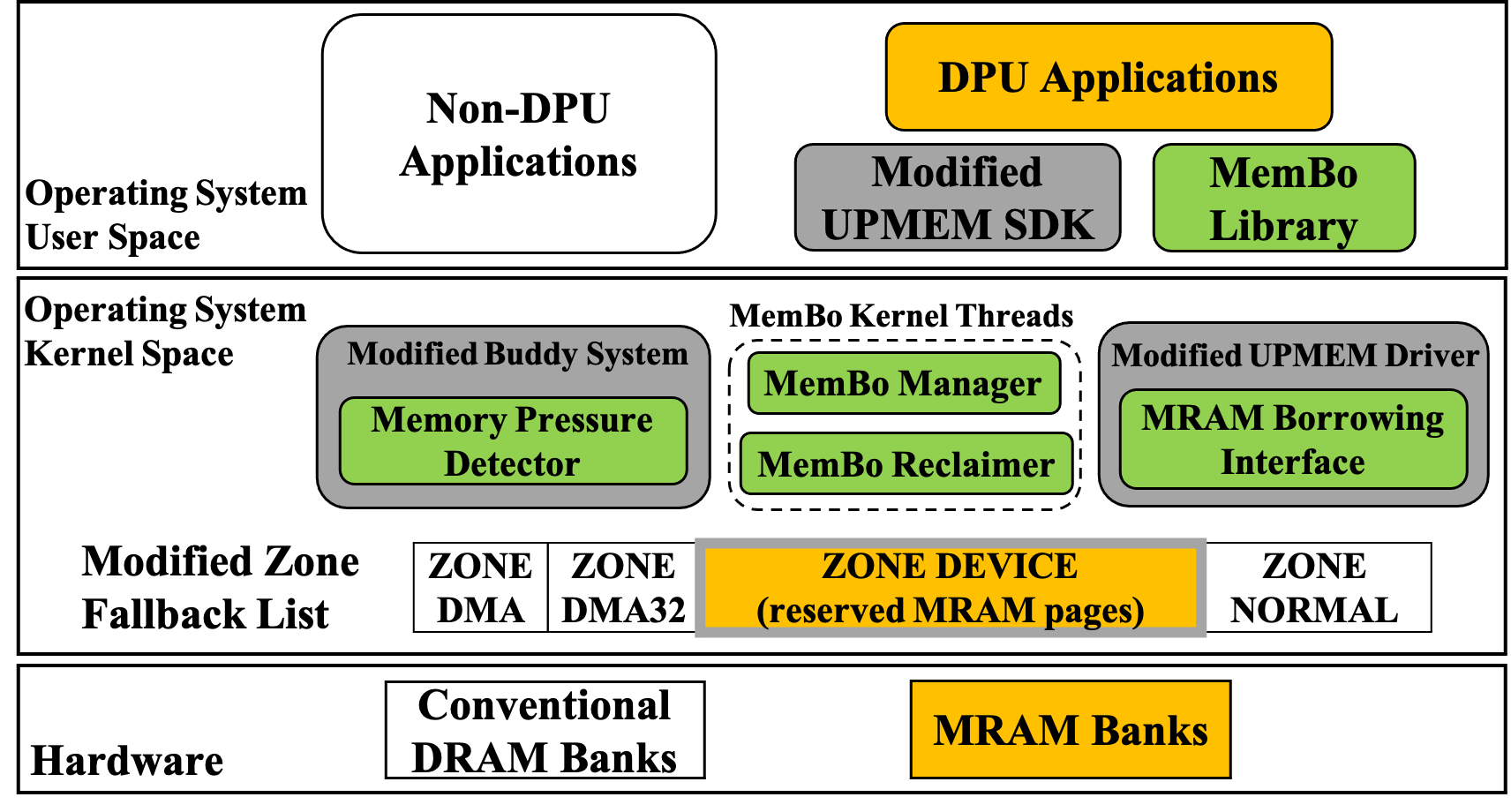


Figure 3. Architecture of MemBo

The initialization process of MemBo consists of two phases. In the first phase, during the buddy system initialization, we make ZONE\_DEVICE visible to the buddy system page allocator by inserting it to the fallback list. This enables the allocator to allocate memory from ZONE\_DEVICE when needed. In the second phase, after the OS initialization, the modified UPMEM driver is loaded into the OS. The modified UPMEM driver is an extension of the UPMEM driver 2021.3.0 [?]. It provides an MRAM borrowing interface, enabling the MRAM borrowing and reclamation operations. It will then activate the MemBo threads, which are responsible for initiating MRAM borrowing or reclamation operations during the system runtime. Especially, we insert a memory pressure detector to the buddy system to collaborate with MemBo kernel threads, triggering MRAM borrowing/reclamation based on the system memory pressure.

As mentioned earlier, MemBo UPMEM SDK minimizes the required changes in DPU programming. However, for programmers who seek to optimize the reclamation time of MRAM, the MemBo library provides additional user-friendly APIs. These APIs are designed to be straightforward to use and allow server managers/DPU application programmers to fine-tune the MRAM reclamation process. Our evaluation has shown that employing these techniques can significantly reduce the latency penalty caused by MRAM reclamation, leading to improved overall performance.

## Memory Zones Setup in MemBo

In this section, we introduce the memory zone setup in MemBo, which takes place during the system initialization phase and is essential for enabling the MRAM borrowing mechanism. Enabling the MRAM borrowing mechanism requires integrating ZONE\_DEVICE into the page allocation process of the buddy system. To achieve this, MemBo includes ZONE\_DEVICE in the zone fallback list during the initialization phase of the buddy system. This modification allows failed page allocations in ZONE\_NORMAL to fall back to ZONE\_DEVICE, attempting to utilize the free pages in ZONE\_DEVICE for the allocation. The modification to the memory zone organization is a crucial step in setting up the infrastructure for the MRAM borrowing mechanism to function effectively.

As shown in Figure 3, in the fallback list, ZONE\_DEVICE is positioned between ZONE\_NORMAL and ZONE\_DMA32. This arrangement establishes the fallback order as follows: ZONE\_NORMAL, ZONE\_DEVICE, ZONE\_DMA32, and finally ZONE\_DMA. The design is based on two reasons. Firstly, as mentioned earlier, MRAM borrowing may introduce latency penalties for DPU applications. Therefore, the buddy system should initiate MRAM borrowing only when the memory pages in ZONE\_NORMAL are nearly exhausted. Secondly, MRAM pages are in high-memory addresses, making them incompatible with devices that can only access low-memory addresses for performing DMA (Direct Memory Access). Consequently, these devices should exclusively allocate DMA memory from ZONE\_DMA or ZONE\_DMA32 but not ZONE\_DEVICE, ensuring compatibility with their memory access requirements.

## The MemBo UPMEM Driver

In this section, we elaborate on how we enable the MRAM borrowing/reclamation mechanism by modifying the UPMEM driver. The UPMEM driver functions as the manager of DPUs and MRAM pages, handling DPU allocations and supporting the UPMEM SDK. To enable the MRAM borrowing and reclamation mechanism, MemBo extends the functionality of the UPMEM driver with an MRAM borrowing interface, enabling the buddy system to adjust the number of borrowed MRAM pages within ZONE\_DEVICE based on the current memory demand during the system runtime.

### MRAM Borrowing/Reclamation Operation

The MRAM borrowing interface provides two operations: the MRAM borrowing operation and the MRAM reclamation operation. The MRAM borrowing operation involves adding a batch of MRAM pages to the buddy system's management, thereby expanding the available free page pool within ZONE\_DEVICE. This operation consists of two phases. During the first phase, kernel metadata such as memmap (i.e., *struct page*s) and page tables for direct mapping are initialized. In the second phase, the MRAM pages are inserted into the free list of ZONE\_DEVICE, allowing them to be used by the buddy system. On the other hand, an MRAM reclamation operation refers to removing a batch of MRAM pages from the buddy system's management and returning control of these pages back to the MemBo UPMEM driver. The MRAM reclamation operation also involves two phases. In the first phase, the pages are removed from the free list and concealed from the buddy system. Any live pages are migrated to other memory locations. In the second phase, the metadata associated with the reclaimed pages is destroyed, completing the reclamation operation.

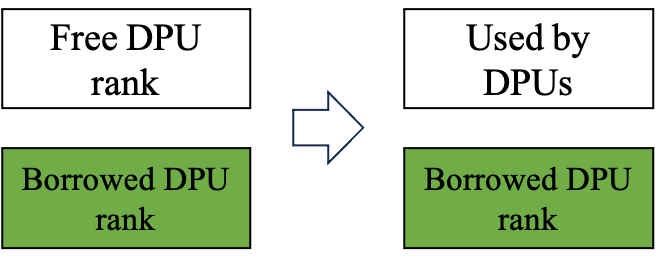
### MRAM Borrowing/Reclamation Operation Granularity

The choice of granularity for MRAM borrowing/reclamation operations is a critical consideration. During an MRAM borrowing/reclamation operation, the process must acquire system locks for the memory zone involved. If the granularity is too large, the MRAM borrowing/reclamation operation can block other operations to that memory zone for a long period of time. On the other hand, if the granularity is too small, the speed of MRAM borrowing may not be able to keep up with high memory demands during periods of significant memory pressure. This situation can trigger costly swapping operations, which can degrade system performance. In our current implementation, the granularity is set to 64MB.

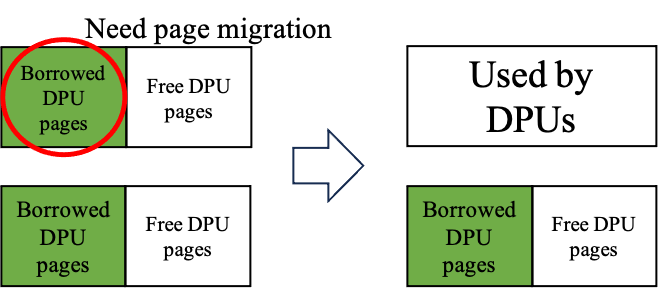
### DPU Rank Management Data Structure in MemBo

Next, we provide an explanation of how DPU ranks are managed in MemBo. As previously mentioned, DPUs are managed and allocated at the rank level, which corresponds to 4GB of MRAM. On the other hand, MRAM borrowing is performed at a granularity of 64MB. Consequently, MemBo needs to carefully consider the data placement of the borrowed MRAM pages. Otherwise, DPU allocations are prone to experiencing additional overhead due to data migration, as depicted in Figure 4.

As shown in Figure 5, MemBo utilizes two linked lists for DPU rank management. The first list, called the "rank list," is responsible for tracking DPU ranks that are either free or used by DPU applications. The second list, known as the "MemBo rank list," is used to manage ranks with borrowed MRAM pages. During MRAM borrowing operations, MemBo follows a specific rule. It first exhausts the MRAM pages in the MemBo rank list before utilizing pages from other free ranks. In other words, when an MRAM borrowing operation is initiated, the MemBo UPMEM driver attempts to obtain free MRAM pages from the MemBo rank list. If all the MRAM pages in the MemBo rank list are already borrowed, the MemBo UPMEM driver then proceeds to move a free rank from the rank list to the MemBo rank list. Conversely, when all the MRAM pages of a DPU rank in the MemBo rank list are reclaimed, that rank is moved back to the rank list.



1. Case 1: No need for data migration when allocating a DPU rank



1. Case 2: Need data migration when allocating a DPU rank

Figure 4. Data placement of the borrowed MRAM pages

## The MemBo Manager Thread and MemBo Reclaimer Thread

TABLE 1

Summary of Memory Watermarks used in MemBo

|  |  |  |
| --- | --- | --- |
| Watermarks | Value | Usage |
| WMARK\_LOW | Calculated based on the system memory capacity | Determine the timing to initiate an MRAM borrowing operation |
| WMARK\_HIGH\_MEMBO | WMARK\_LOW + 512MB | Determine the timing to initiate an MRAM reclamation operation |

In this section, we explain how MemBo monitors the system’s memory pressure and initiates MRAM borrowing/reclamation operations automatically. In particular, we will explain the timing of initiating MRAM borrowing/reclamation operations and the MemBo components responsible for performing those operations.

As shown in Figure 6, the Linux buddy system uses a memory watermark known as WMARK\_LOW to control the timing of the fallback mechanism and swapping operations. Whenever a page allocation would cause the number of free pages to fall below WMARK\_LOW, the allocation is redirected to other zones. In cases where the buddy system fails to find a suitable zone for that allocation, it wakes up kswapd, the swapping daemon thread. kswapd continues to execute swapping operations until the free page number reaches WMARK\_HIGH.

一張含有 螢幕擷取畫面, 正方形, 鮮豔, Rectangle 的圖片

自動產生的描述

Figure 5. MRAM Management of MemBo

The main goal of MemBo is to minimize the occurrence of swapping operations by making use of idle MRAM pages whenever feasible. This is achieved by redirecting failed allocations in ZONE\_NORMAL to ZONE\_DEVICE. If ZONE\_DEVICE successfully handles the allocation with borrowed MRAM pages, it eliminates the need for kswapd to perform swapping operations, allowing it to remain in a dormant state. As a result, MemBo must dynamically adjust the size of the free page pool in ZONE\_DEVICE based on the system memory demand.

To dynamically adjust the size of the free page pool in ZONE\_DEVICE, MemBo introduces two threads. The MemBo manager thread is awakened when necessary to initiate MRAM borrowing operations. Its role is to ensure that an adequate number of MRAM pages are available for allocation. On the other hand, the MemBo reclaimer thread operates in the background and is responsible for periodically reclaiming unused borrowed MRAM pages. This reclamation process helps maintain efficient memory utilization by freeing up MRAM pages that are no longer in active use.

The timing of initiating MRAM borrowing/reclamation operations in MemBo is determined using memory watermarks set by MemBo. Table 1 provides a summary of the memory watermarks relevant to MemBo. During page allocations, if the memory pressure detector detects that the number of free memory pages in both ZONE\_NORMAL and ZONE\_DEVICE falls below their respective WMARK\_LOW values, it triggers the awakening of the MemBo manager thread. Subsequently, the MemBo manager thread attempts to initiate an MRAM borrowing operation.

Once the first 64MB of MRAM pages has been borrowed, the MemBo reclaimer thread is activated. It periodically checks on the number of free pages in ZONE\_NORMAL. If it exceeds the WMARK\_HIGH\_MEMBO watermark, indicating a surplus of free ZONE\_NORMAL pages, the MemBo reclaimer thread calls the reclaim\_mram\_pages operation to reclaim 64MB of MRAM. It is worth noting that the WMARK\_HIGH\_MEMBO watermark is set to a high value (WMARK\_LOW + 512MB) to ensure that the reclamation operation will not be triggered due to the memory usage fluctuation when the system is still under high memory pressure.

一張含有 寫生, 圖畫, 黑色, 藝術 的圖片

自動產生的描述

Figure 6. Linux memory watermark

The main objective of the MemBo manager thread and MemBo reclaimer thread is to dynamically adjust the quantity of borrowed MRAM pages based on the memory demand of the host system. However, it is essential to address situations where the borrowed MRAM may be required by a DPU application. In such cases, the MemBo UPMEM SDK offers a DPU allocation API that triggers immediate MRAM reclamation during the DPU allocation process. Unfortunately, this can potentially result in significant blocking time, leading to latency penalties for the DPU application. In the next section, two optimization techniques are introduced to mitigate this issue and minimize the potential impact on DPU applications' latency.

## Reclamation Time Optimization Techniques

### Data Migration Pipelining

Figure 7 illustrates the fundamental concept of data migration pipelining (DMP), which involves pipelining the reclamation process and the input data copy time based on the rank level. Instead of waiting for all the necessary ranks to be reclaimed before initiating the input data copy operation, DMP activates the data copy for each DPU rank as soon as it becomes available. This pipelining approach allows for overlapping the reclamation process with the input data copying, resulting in improved efficiency. Especially, the hardware can parallelize the data copy operations of different DPU ranks, effectively concealing the latencies associated with data copying.

TABLE 2

APIs for DPU Usage Predictor Implementation

|  |  |
| --- | --- |
| APIs | Usage |
| *get\_dpu\_usage()* | Get the current DPU usage in ranks. |
| *set\_membo\_*  *rank\_reservation(nr\_ranks)* | Set the number of ranks to be reserved to *nr\_ranks*. If MemBo determines that there are insufficient ranks available to fulfill the reservation requirement, it initiates rank reclamation operations. |

To facilitate data migration pipelining, the MemBo UPMEM SDK introduces an event-driven DPU rank allocation API. This API enables DPU application programmers to register a callback function during DPU rank allocations. Whenever a single DPU rank is reclaimed, the registered callback function is invoked. This allows programmers to initiate data migration

一張含有 螢幕擷取畫面, 鮮豔 的圖片

自動產生的描述

(a) DPU application latency without DMP

一張含有 螢幕擷取畫面, 鮮豔 的圖片

自動產生的描述

(b) DPU application latency with DMP

Figure 7. Concept of DMP

operations for the reclaimed DPU rank within the callback function. By utilizing this event-driven API, DPU application programmers can take advantage of the callback function to trigger data copying for the already-reclaimed DPU ranks, enabling efficient pipelining of the reclamation process and the input data copy time. The event-driven DPU rank allocation API allows programmers to reduce the idle time of the reclaimed DPU ranks and seamlessly integrate data migration pipelining techniques into their applications.

### Prediction-based Rank Reservation Mechanism

The main concept of the technique is to make periodic predictions of DPU usage in the near future. Based on these predictions, the mechanism determines the number of DPU ranks that can be borrowed. The objective is to reclaim MRAM pages in advance, ensuring they are available when needed by the DPUs and reducing any potential delays in the startup of DPU applications.

Utilizing the historical record of resource usage is a widely adopted method for resource usage prediction [28, 32-35]. To facilitate this, the MemBo UPMEM SDK offers a programming framework with a set of APIs, as listed in Table 2. These APIs allow user space programmers to implement their own prediction algorithms for optimizing MemBo:

In this paper, we introduce LiRDUP, a Linear Regression-based DPU Usage Prediction algorithm, which utilizes the provided APIs. Our approach is inspired by the idea proposed in LiRCUP [28]. Algorithm 1 presents the pseudocode for LiRDUP. To begin, we initialize a ring buffer called "historical\_usages" with a size of k, which is used to store the past k DPU usage data. LiRDUP periodically samples the DPU usage and stores the sample data in "historical\_dpu\_usage" (lines 3-4). Whenever a new record is collected, LiRDUP invokes the prediction function "lirdup\_predict" (line 8) to predict the DPU usage for the next period using the historical data stored in the ring buffer. Subsequently, LiRDUP calls the "set\_rank\_reservation" API to update the number of ranks to be reserved based on the prediction result. If LiRDUP has not collected enough records (less than k), the prediction is skipped for that round. In this paper, the prediction function "lirdup\_predict" (line 7) is treated as a black box and not further explained. It is important to note that alternative prediction functions can be utilized in place of "lirdup\_predict".

|  |  |
| --- | --- |
| **Algorithm 1.** LiRDUP | |
|  | ***Input:*** |
| *k: number of recent records used for the prediction* |
| *period: sample period in seconds* |
| *historical\_usages: a ring buffer of size k* |
| ***Output****:* |
| *None* |
|  | ***while*** *true* ***do*** |
|  | *// Sample and store the current DPU usage* |
|  | ***uint8\_t*** *current\_dpu\_usage ←* ***get\_dpu\_usage()****;* |
|  | *historical\_usages.push(current\_dpu\_usage);* |
|  |  |
|  | *// Predict the DPU usage* |
|  | *if (historical\_usages.size == k) {* |
|  | ***uint8\_t*** *prediction =* ***lirdup\_predict();*** |
|  | ***set\_rank\_reservation****(prediction);* |
|  | *}* |
|  | *// Sleep for period of seconds* |
|  | *sleep(period);* |
|  | ***end*** |

TABLE 3

Evaluation Platform Configuration

|  |  |
| --- | --- |
| CPU | Intel Xeon Silver 4208 CPUs @ 2.10GHz [20] |
| Conventional  DIMMs | 16GB \* 4 |
| UPMEM PIM DIMMs | 16 dual rank UPMEM PIMM DIMMs  Total DPUs: 1024 DPUs @350GHz  Total MRAM: 64GB |
| Operating System | Distro: Ubuntu 20.04  **Linux kernel: 5.15.45 (MemBo)**  **UPMEM driver: 2021.3.0 [1] (MemBo)** |
| UPMEM SDK | **UPMEM SDK 2021.3.0 [1] (MemBo)** |
| SSD | SAMSUNG 980 PRO NVMe SSD 2TB (used as swap space) |

# Evaluation

In this section, we assess the advantages and trade-offs of MemBo and address the following two questions:

1. Performance of MemBo in In-Memory Computing Applications under High Memory Pressure:

We evaluate how MemBo performs in in-memory computing applications when faced with high memory pressure. This involves examining the effectiveness of MemBo in dynamically expanding the memory capacity of UPMEM platforms to accommodate the increased memory demands. We assess its ability to alleviate memory pressure and maintain efficient performance in these applications. We analyze metrics such as throughput and major page fault number to quantify the benifit of MemBo in high-memory-pressure scenarios.

2. Impact of MemBo on DPU Applications and Optimization Techniques for Reclamation Time:

TABLE 4

YCSB Workloads

|  |  |  |
| --- | --- | --- |
| Workload | Operation | Request Distribution |
| A | 50% Read, 50% Write | Zipfian |
| B | 95% Read, 5% Write | Zipfian |
| C | 100% Read | Zipfian |
| D | 95% Read, 5% Write | Latest |
| F | 50% Read,  50% Read-Modify-Write | Zipfian |

We investigate the impact of MemBo on DPU applications and study the effects of memory expansion and shrinkage on application performance. We analyze the overhead introduced by the memory management operations of MemBo, such as memory reclamation. Additionally, we explore optimization techniques employed by MemBo to minimize the impact on DPU applications, particularly the reclamation time.

By addressing these two questions, we provide a comprehensive evaluation of MemBo's benefits and costs in terms of its performance in in-memory computing applications under high memory pressure and its impact on DPU applications, including the effectiveness of optimization techniques to mitigate any potential drawbacks.

## Evaluation Setup

Table 3 displays the configuration of our evaluation platform, which includes 4 conventional 16GB DIMMs and 16 UPMEM PIM DIMMs with 4GB of MRAM, allowing for a total memory capacity expansion of up to 128GB when utilizing MemBo. The implementation of MemBo in our evaluation is based on the Linux kernel version 5.15.45, UPMEM driver version 2021.3.0, and UPMEM SDK version 2021.3.0. These software components provide the necessary support and functionality for MemBo to operate effectively within the system, enabling dynamic memory capacity adjustments and optimizing memory utilization on the UPMEM platform.

To assess the performance under memory pressure, we employ the YCSB (Yahoo! Cloud Serving Benchmark) workloads [37]. The YCSB workloads are executed on Memcached [36], a widely used in-memory key-value cache system. In Table 4, we provide a summary of the YCSB workloads used in our experiments. The experiments are conducted using the default Memcached item size of 1KB, and item and operation counts are configured to 50M or 100M to generate a substantial memory footprint, triggering the MemBo MRAM borrowing operations. These large counts ensure significant memory pressure and allow us to evaluate the effectiveness of MemBo in managing memory resources.

Additionally, to evaluate the performance of DPU applications in MemBo environments, we select eight DPU applications from the PrIM [5] benchmark suite. Table 5 provides a summary of the DPU applications employed in our experiments.

By running these applications in the MemBo environment, we can assess the impact of MemBo on their performance. These evaluations help us understand the effects of MemBo on the execution of DPU applications.

TABLE 5

DPU Applications

|  |  |
| --- | --- |
| DPU Application | Input Size |
| VA | 25M elements/DPU rank |
| SEL |
| UNI |
| RED |
| SCAN-RSS |
| SCAN-SSA |
| TRNS | 12288 × 16 × 1024 × 8/16 DPU ranks |
| BS | 25M elements, 25M queries/DPU rank |

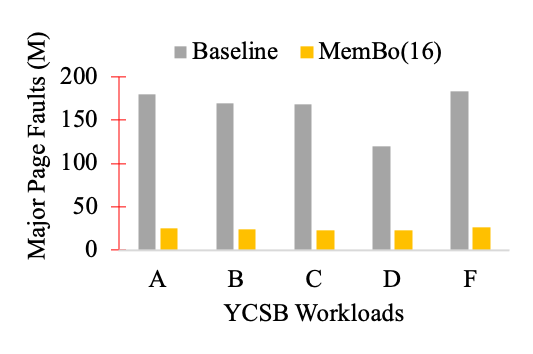


Figure 11. YCSB Major Page Faults Reduction (100M items/operations)

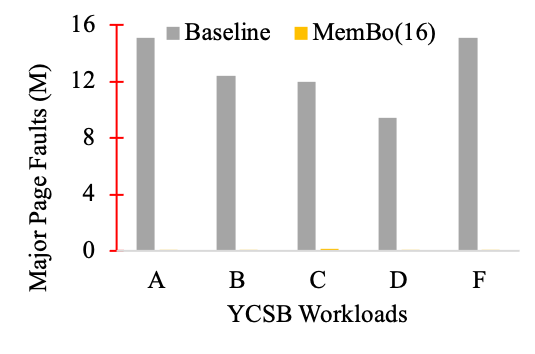


Figure 9. YCSB Major Page Faults Reduction (50M items/operations)

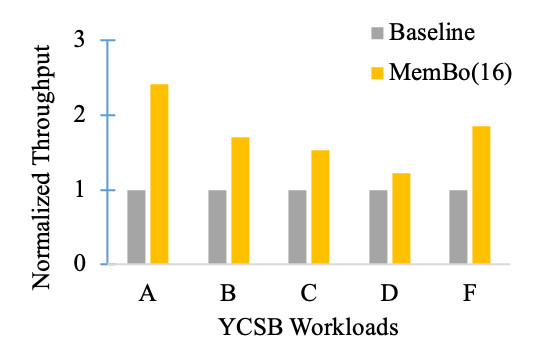


Figure 8. YCSB Throughput Improvement (50M items/operations)

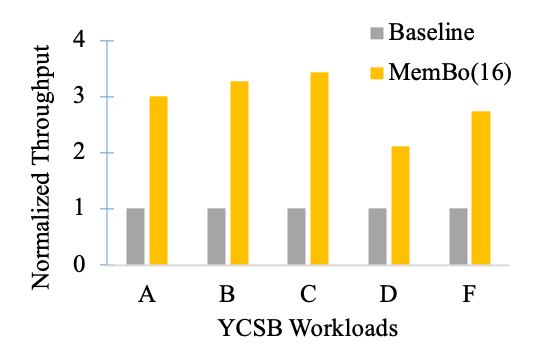


Figure 10. YCSB Throughput Improvement (100M items/operations)

***Experimental Groups Denotation***

In our experimental setup, the baseline group represents the native UPMEM platform without the MemBo mechanism. This group serves as a reference point for comparison.

To evaluate the benefits and costs of MemBo, we introduce experimental groups where MemBo is enabled with different numbers of ranks available for MRAM borrowing. We define a parameter, , which denotes the number of DPU ranks allocated for MRAM borrowing during system runtime. Therefore, MemBo() represents an experimental group where MemBo is enabled, and DPU ranks are dedicated to the MRAM borrowing process. By varying , we can assess the impact of different levels of DPU rank allocation on the performance and efficiency of the MemBo mechanism. This analysis helps us evaluate the trade-offs and costs associated with different configurations of MemBo in terms of memory management and overall system performance.

## Performance Improvement of YCSB workloads

In this section, we conduct a performance evaluation of MemBo(16) compared to the baseline using the YCSB workloads listed in Table 4. Figure 8 illustrates the throughput improvement achieved by MemBo(16) when the item and operation count of the YCSB workloads are set to 50M. MemBo(16) exhibits a notable improvement, with throughput enhancements of up to 2.42× compared to the baseline. It is particularly noteworthy that MemBo(16) achieves more significant improvements in write-operation-intensive workloads (A and F). This can be attributed to the fact that MemBo(16) effectively reduces major page faults in these workloads, as shown in Figure 9. The baseline experiences a higher number of major page faults in these workloads compared to others, while MemBo(16) leverages idle MRAM resources effectively, avoiding the issuance of major page faults in a significant majority of cases (over 99.9%). However, in YCSB workload D, where requests are issued with the latest distribution, MemBo(16) achieves a relatively smaller throughput improvement of only 1.22×. This is because the workload exhibits better spatial locality, resulting in fewer major page faults being issued by the baseline, limiting the benefit of MemBo(16) in this scenario.

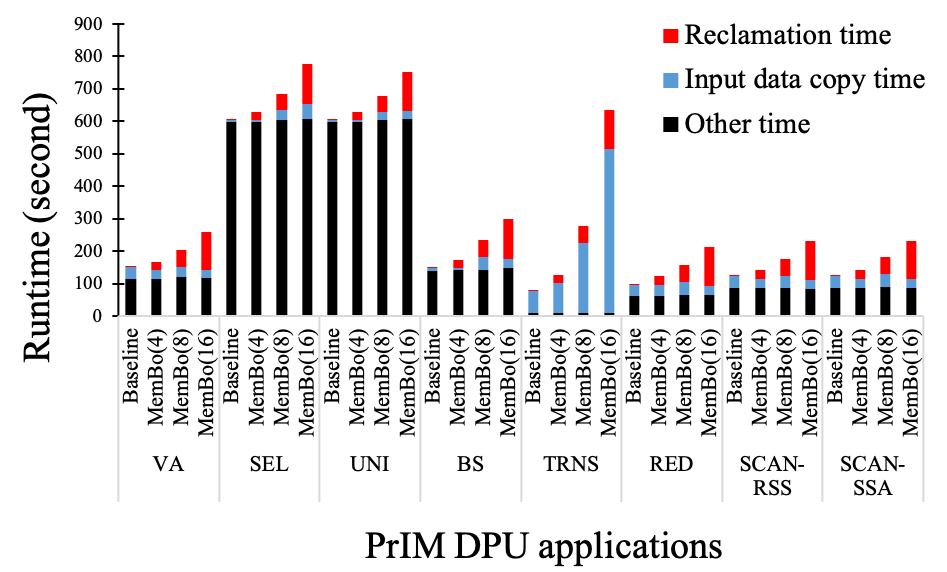


Figure 13. DPU Applications Runtime (When executed concurrently with the YCSB A workload)

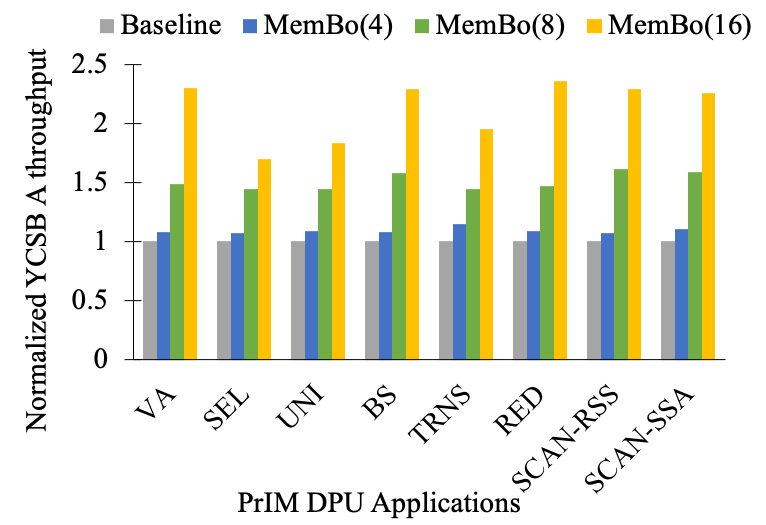


Figure 12. YCSB A Throughput

(When executed concurrently with the DPU applications in the y-axis)

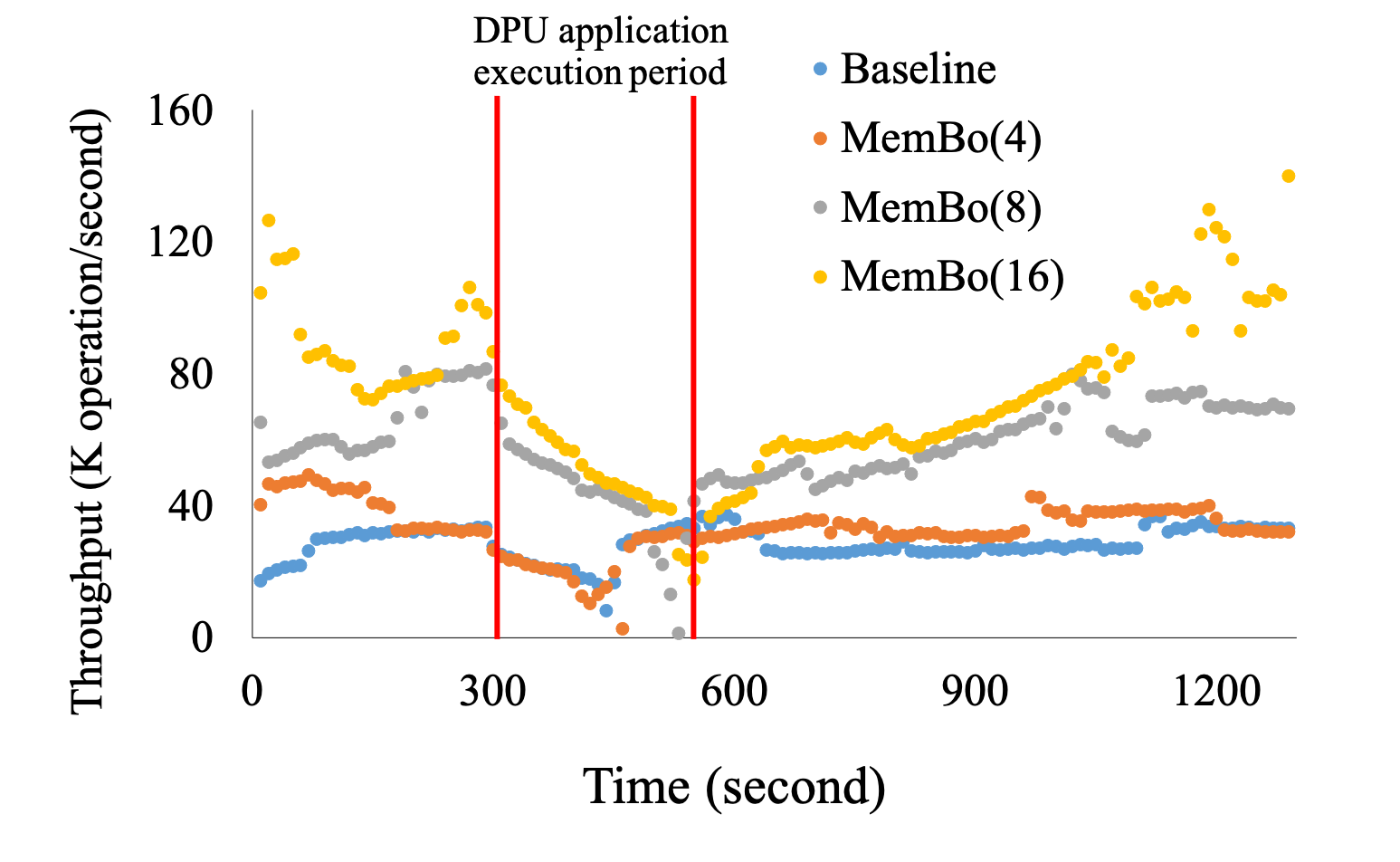


Figure 14. Throughput Variation of YCSB A

(When executed concurrently with VA)

Figure 10 demonstrates the throughput improvement when the item and operation count of the YCSB workloads are incr eased to 100M. Additionally, Figure 11 showcases the reduction in major page faults achieved by MemBo(16) in this scenario. MemBo(16) exhibits a larger magnitude of throughput improvement over the baseline, ranging from 2.11× to 3.4×. This significant improvement is due to the substantial performance drop experienced by the baseline, caused by a huge number of major page faults issued by Memcached. It is particularly noteworthy that MemBo(16) archives more significant improvements in read-operation-intensive workloads (B and C) in this case. Overall, the evaluation indicates that MemBo(16) provides significant throughput improvements compared to the baseline. It effectively reduces major page faults and optimizes the utilization of idle MRAM resources. However, it's important to note that the magnitude of improvement can vary depending on factors such as the size of the memory footprint and the memory access pattern. In summary, while MemBo(16) generally delivers notable throughput improvements and efficient utilization of MRAM resources, it is important to consider workload-specific factors that may influence the actual magnitude of improvement in each scenario.

## YCSB Throughput Improvement vs. DPU Application Reclamation time Tradeoff

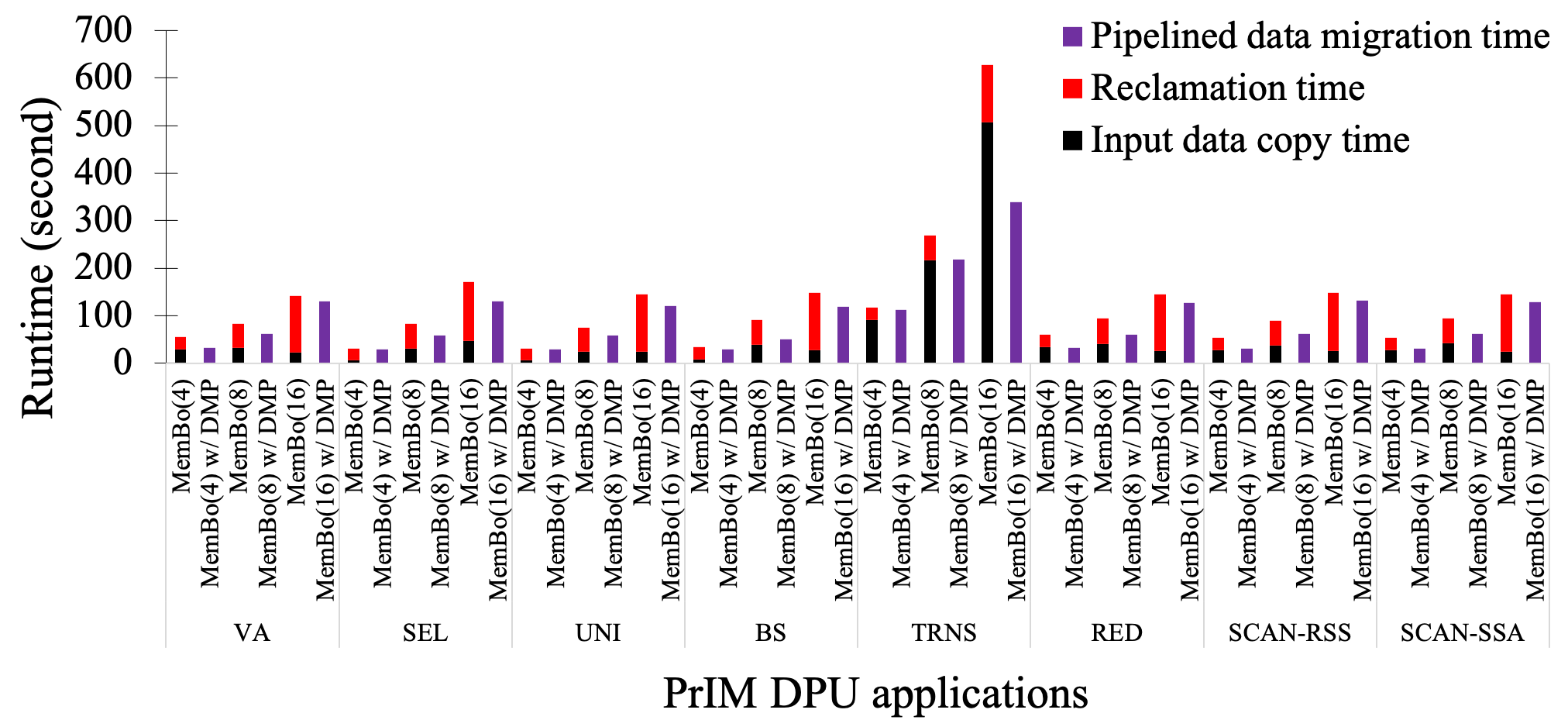


Figure 15. Runtime Reduction of DMP (sum of input data copy time and reclamation time)

In this section, we evaluate the performance of YCSB workloads and DPU applications when one YCSB workload and one DPU application are executed concurrently. The purpose is to analyze the impact of MRAM reclamation time

and the benefit of MemBo in this scenario. The experiment is conducted as follows:

1. Launch YCSB workload A (100M items and operations).

2. After 5 minutes, activate a 16-rank DPU application, triggering MRAM reclamation.

3. Collect the overall throughput of YCSB workload A and the runtime of the 16-rank DPU application.

Figure 12 illustrates the throughputs of YCSB workloads, and Figure 13 shows the runtime breakdown of each DPU application, where runtime of a DPU application is divided into three parts:

1. Reclamation Time: Time required to perform MRAM reclamation for the required DPU ranks.

2. Input Data Copy Time: Time taken for DPU input data copy.

3. Other Time: Remaining execution time, including input data setup, DPU execution, and output data copy.

It can be observed that MemBo() achieves better performance in YCSB workloads as increases. MemBo(16) achieves a throughput improvement of more than 2×, MemBo(8) achieves a throughput improvement of 1.5×, and MemBo(4) achieves a throughput improvement of 1.1×. Figure 14 shows the runtime throughput variation of YCSB workload A when executed concurrently with the DPU application. It can be observed that MemBo groups can detect memory pressure and automatically trigger MRAM borrowing operations using idle DPU ranks.

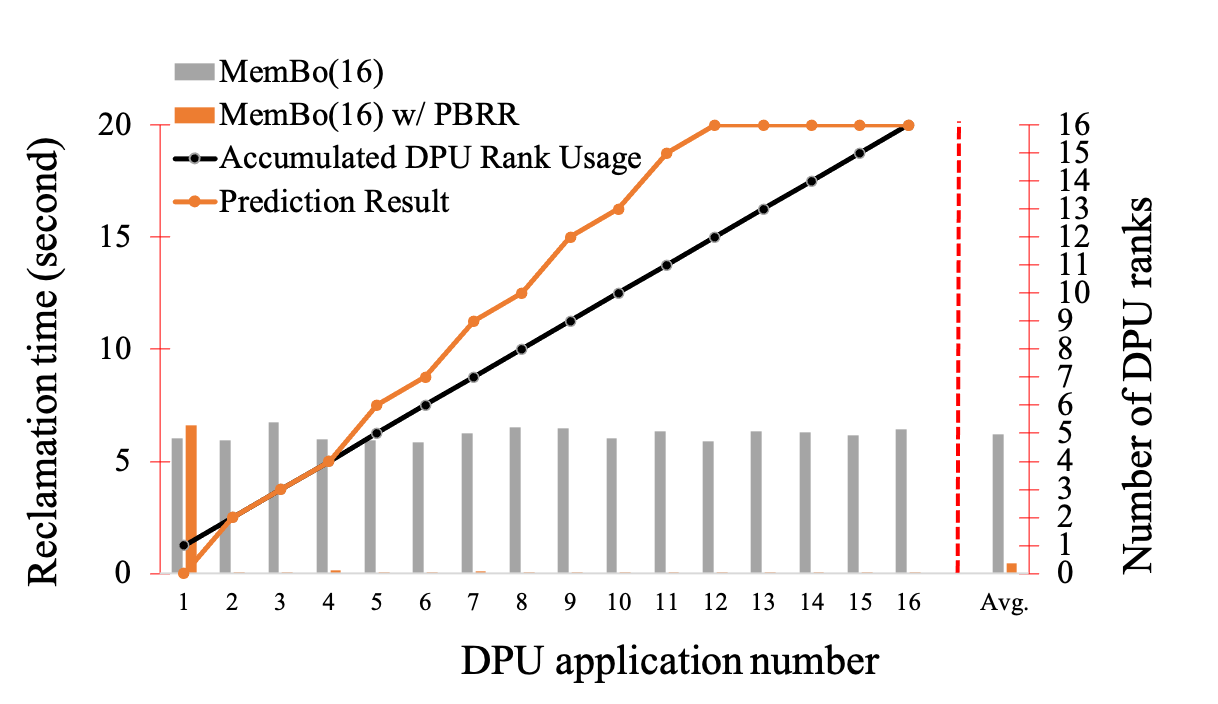


Figure 16. Reclamation Time Reduction and Prediction Result (Configuration A)

However, it is worth noting that the reclamation time also increases as increases, significantly impacting the runtime of DPU applications. For latency-sensitive applications like RED, the time taken for reclaiming 16 DPU ranks exceeds its runtime, resulting in a more than 2× increase in runtime. From these observations, we conclude that a larger can provide more benefits to memory-hungry host-side processes. However, it can significantly impact the performance of DPU applications. Therefore, it is essential to strike a balance between the values chosen for to optimize the overall system performance without significantly hurting the DPU application performance, which motivates the idea of prediction-based rank reservation mechanism.

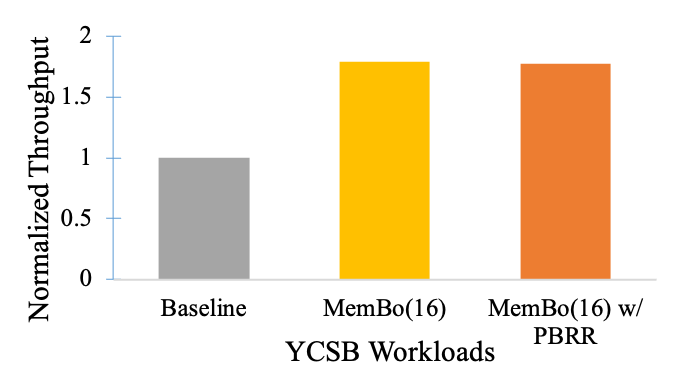


Figure 17. YCSB A Throughput Comparison (Configuration A)

## Analysis of Data Migration Pipelining

In this section, we evaluate the impact of the data migration pipelining (DMP) technique. The experiment follows the same setup as the previous section, but we apply DMP to each MemBo experimental group and collect the reduction in the sum of input data copy time and reclamation time.

Figure 15 presents the reduction achieved in the sum of input data copy time and reclamation time with the application of DMP. It can be observed that DMP effectively hides the latency introduced by reclamation time, reducing the overall latency of sum of input data copy time and reclamation time by up to 55%.

Furthermore, the effectiveness of DMP depends on the proportion of input data copy time compared to the reclamation time. When the proportion of input data copy time is larger or equal to the reclamation time, the reclamation time can be completely hidden, resulting in a significant reduction in overall latency. This highlights the effectiveness of DMP in concealing

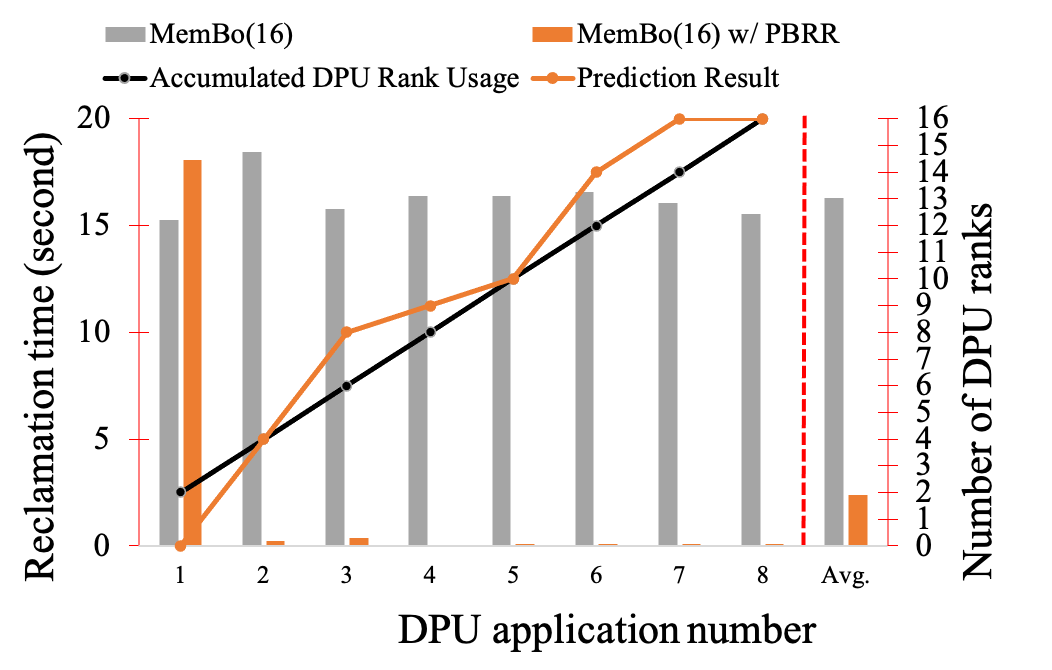


Figure 18. Reclamation Time Reduction and Prediction Result (Configuration B)

the reclamation latency for input-data-copy-intensive DPU applications.

In conclusion, the results demonstrate that DMP is an effective technique for hiding the latency associated with reclamation time during the input data copy phase. The extent of the reduction depends on the proportion of input data copy time relative to reclamation time, emphasizing its effectiveness for input-data-copy-intensive DPU applications.

## Prediction-based Rank Reservation Mechanism

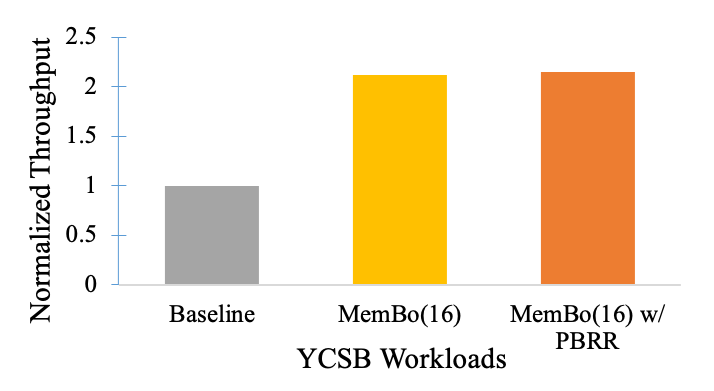


Figure 19. YCSB A Throughput Comparison (Configuration B)

In this section, we evaluate the effectiveness of the prediction-based rank reservation (PBRR) mechanism. To produce runtime DPU usage variation, we divide the 16-rank VA application in Table 5 into multiple DPU applications and launch them sequentially, with each DPU application allocating DPU ranks and occupying them for a duration of minutes before releasing them. The experiment is conducted as follows:

1. Launch YCSB workload A (100M items and operations).

2. After 5 minutes, launch a -rank application every 1 minute, where each DPU application occupies its allocated DPU ranks for minutes before releasing them.

3. Collect the overall throughput of YCSB workload A and the reclamation time of each DPU application

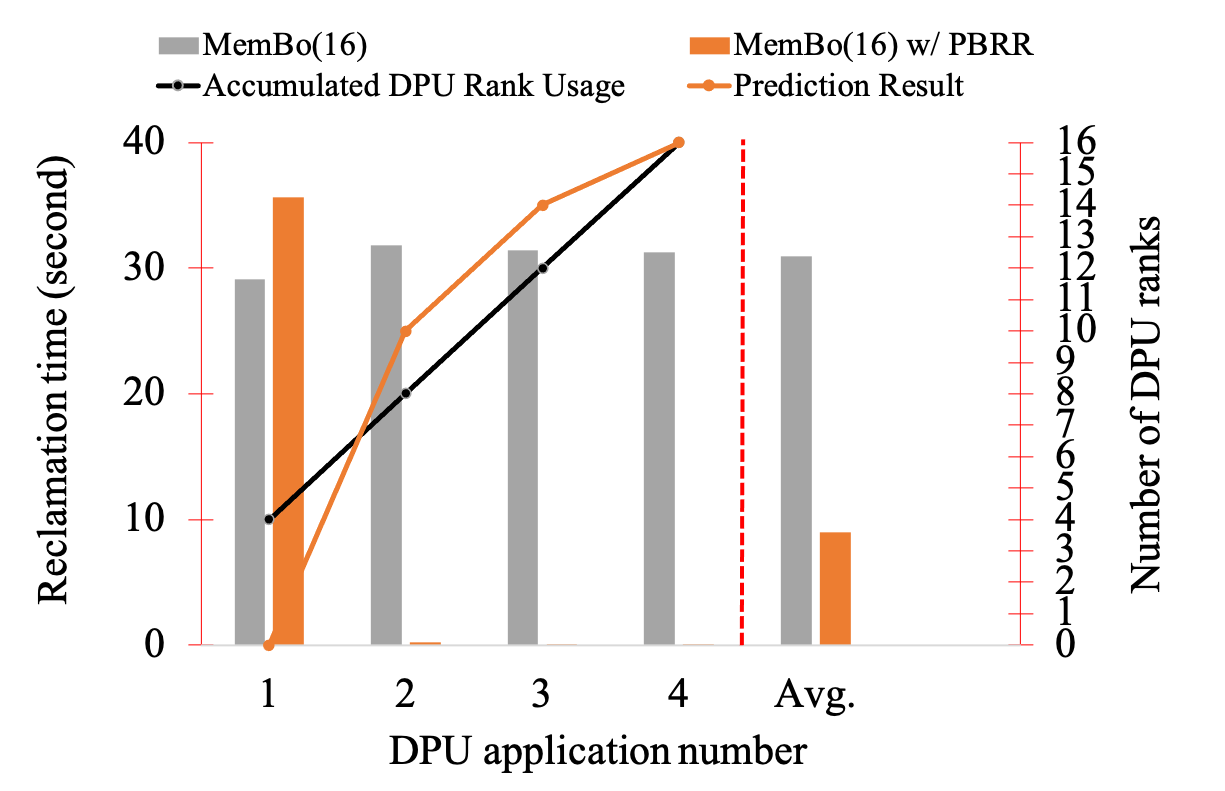


Figure 20. Reclamation Time Reduction and Prediction Result (Configuration C)

Table 6 provides the configurations of the two parameters used in our experiments: and .

Figure 16 presents the reclamation time reduction and the prediction results of the experiment with configuration A. It shows that the application of the PBRR technique significantly

reduces the average reclamation time of the DPU applications, with reductions of up to 96%. The predictor employed by PBRR is able to reserve a sufficient number of ranks for all the DPU applications, except for the first one. This is due to the lack of historical DPU usage records when the first DPU application arrives. Similar phenomena can be observed in the results of experimental configuration B and C, as shown in Figure 18 and Figure 20, respectively, where the predictor underestimates the DPU rank demand for the first few DPU applications. However, PBRR still manages to reduce the average reclamation latency by 64% and 51% respectively. Furthermore, it is worth noting that PBRR occasionally overestimates the actual demand for DPU ranks. To assess if these situations impact the benefits of MemBo, Figure 17 compares the throughput of the YCSB workload A in the baseline, MemBo(16), and MemBo(16) with PBRR. It can be observed that PBRR does not significantly diminish the benefits of MemBo(16) and can still achieve significant improvements over the baseline. Similar results can be observed in other experimental configurations, as shown in Figure 19 and Figure 21.

In conclusion, PBRR proves to be an effective mechanism for reducing the average reclamation time of DPU applications while preserving the benefits of MemBo. Although there are instances where the predictor underestimates or overestimates the demand for DPU ranks, PBRR still achieves substantial reductions in reclamation latency and maintains the overall advantages of MemBo in terms of performance and throughput.

# CONCLUSION

In this paper, we propose MemBo, a runtime memory borrowing mechanism that provides a powerful solution to address memory demand variance on UPMEM servers, equipped with PIM memory hardware. By increasing memory management flexibility, MemBo significantly improves throughput and reduces major page faults for a UPMEM server, enhancing the performance of popular in-memory computing applications. Moreover, MemBo's transparency to administrators and users further enhances its appeal and ease of integration.

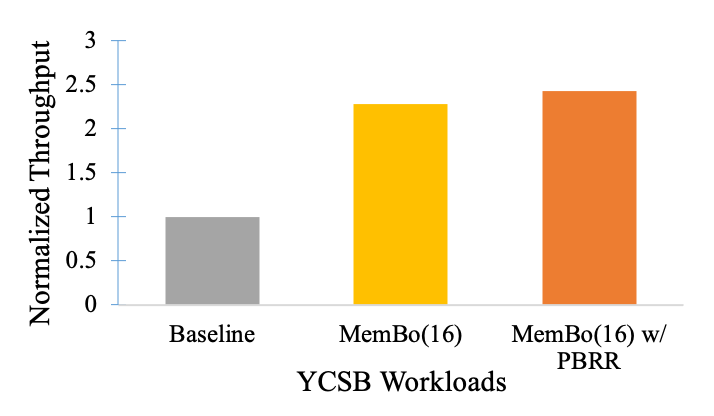


Figure 21. YCSB A Throughput Comparison (Configuration C)

References

1. UPMEM, “UPMEM Website,” https://www.upmem.com, 2023.
2. F. Devaux, “The true Processing In Memory accelerator”, in *2019 IEEE Hot Chips 31 Symposium (HCS)*, 2019, pp. 1–24.
3. J. Nider *et al.*, “A Case Study of Processing-in-Memory in off-the-Shelf Systems”, in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 117–130.
4. J. Nider, C. Mustard, A. Zoltan, and A. Fedorova, “Processing in Storage Class Memory”, in *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020.
5. J. Gómez-Luna, I. El Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu, “Benchmarking Memory-Centric Computing Systems: Analysis of Real Processing-In-Memory Hardware”, in *2021 12th International Green and Sustainable Computing Conference (IGSC)*, 2021, pp. 1–7.

TABLE 6

PBRR Experimental Configurations

|  |  |
| --- | --- |
| Configuration | (, ) |
| A | (1, 16) |
| B | (2, 8) |
| C | (4, 4) |

1. D. Lavenier, R. Cimadomo, and R. Jodin, “Variant Calling Parallelization on Processor-in-Memory Architecture”, in *2020 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, 2020, pp. 204–207.
2. D. Lavenier, J.-F. Roy, and D. Furodet, “DNA mapping using Processor-in-Memory architecture”, in *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, 2016, pp. 1429–1435.
3. L.-C. Chen *et al.*, “RNA-seq Quantification on Processing in memory Architecture: Observation and Characterization”, in *2022 IEEE 11th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, 2022, pp. 26–32.
4. G. F. Oliveira, J. Gómez-Luna, S. Ghose, A. Boroumand, and O. Mutlu, “Accelerating Neural Network Inference With Processing-in-DRAM: From the Edge to the Cloud”, *IEEE Micro*, vol. 42, no. 6, pp. 25–38, 2022.
5. J. Gómez-Luna *et al.*, “Machine Learning Training on a Real Processing-in-Memory System”, in *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2022, pp. 292–295.
6. Compute Express Link (CXL). <https://www.computeexpresslink.org/>.
7. J. H. Schopp, K. Fraser, and M. J. Silbermann, “Resizing memory with balloons and hotplug”, in *Proceedings of the Linux Symposium*, 2006, vol. 2, pp. 313–319.
8. H. Liu, H. Jin, X. Liao, W. Deng, B. He, and C.-Z. Xu, “Hotplug or Ballooning: A Comparative Study on Dynamic Memory Management Techniques for Virtual Machines”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 5, pp. 1350–1363, 2015.
9. C. A. Waldspurger, “Memory Resource Management in VMware ESX Server”, *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 181–194, Dec. 2003.
10. J. Kim, V. Fedorov, P. V. Gratz, and A. L. N. Reddy, “Dynamic Memory Pressure Aware Ballooning”, in *Proceedings of the 2015 International Symposium on Memory Systems*, Washington DC, DC, USA, 2015, pp. 103–112.
11. J. Hu, X. Bai, S. Sha, Y. Luo, X. Wang, and Z. Wang, “HUB: Hugepage Ballooning in Kernel-Based Virtual Machines”, in *Proceedings of the International Symposium on Memory Systems*, Alexandria, Virginia, USA, 2018, pp. 31–37.
12. T.-I. Salomie, G. Alonso, T. Roscoe, and K. Elphinstone, “Application Level Ballooning for Efficient Server Consolidation”, in *Proceedings of the 8th ACM European Conference on Computer Systems*, Prague, Czech Republic, 2013, pp. 337–350.
13. J.-H. Chiang, H.-L. Li, and T.-C. Chiueh, “Working Set-based Physical Memory Ballooning”, in *10th International Conference on Autonomic Computing (ICAC 13)*, 2013, pp. 95–99.
14. Q. Zhang, L. Liu, J. Ren, G. Su, and A. Iyengar, “iBalloon: Efficient VM Memory Balancing as a Service”, in *2016 IEEE International Conference on Web Services (ICWS)*, 2016, pp. 33–40.
15. Intel, “Intel Xeon Silver 4110 Processor,” https://ark.intel.com/content/www/us/en/ark/products/123547/intel-xeon-silver-4110-processor-11m-cache-2-10- ghz.html, 2017.
16. D. Gouk, S. Lee, M. Kwon, and M. Jung, “Direct Access, High-Performance Memory Disaggregation with DirectCXL”, in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 287–294.
17. J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, “Efficient Memory Disaggregation with Infiniswap”, in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 649–667.
18. Z. Guo, Y. Shan, X. Luo, Y. Huang, and Y. Zhang, “Clio: A Hardware-Software Co-Designed Disaggregated Memory System”, in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Lausanne, Switzerland, 2022, pp. 417–433.
19. H. Li *et al.*, “Pond: CXL-Based Memory Pooling Systems for Cloud Platforms”, in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Volume 2*, Vancouver, BC, Canada, 2023, pp. 574–587.
20. M. K. Aguilera *et al.*, “Remote regions: a simple abstraction for remote memory”, in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 775–787.
21. E. Amaro *et al.*, “Can Far Memory Improve Job Throughput? ”, in *Proceedings of the Fifteenth European Conference on Computer Systems*, Heraklion, Greece, 2020.
22. I. Calciu *et al.*, “Rethinking Software Runtimes for Disaggregated Memory”, in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Virtual, USA, 2021, pp. 79–92.
23. F. Farahnakian, P. Liljeberg, and J. Plosila, “LiRCUP: Linear Regression Based CPU Usage Prediction Algorithm for Live Migration of Virtual Machines in Data Centers”, in *2013 39th Euromicro Conference on Software Engineering and Advanced Applications*, 2013, pp. 357–364.
24. M. Tirmazi *et al.*, “Borg: The next Generation”, in *Proceedings of the Fifteenth European Conference on Computer Systems*, Heraklion, Greece, 2020.
25. K. Rzadca *et al.*, “Autopilot: Workload Autoscaling at Google”, in *Proceedings of the Fifteenth European Conference on Computer Systems*, Heraklion, Greece, 2020.
26. C. Lu, K. Ye, G. Xu, C.-Z. Xu, and T. Bai, “Imbalance in the cloud: An analysis on Alibaba cluster trace”, in *2017 IEEE International Conference on Big Data (Big Data)*, 2017, pp. 2884–2892.
27. S. Gupta and D. A. Dinesh, “Resource usage prediction of cloud workloads using deep bidirectional long short term memory networks”, in *2017 IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS)*, 2017, pp. 1–6.
28. K. Sato, M. Samejima, and N. Komoda, “Dynamic optimization of virtual machine placement by resource usage prediction”, in *2013 11th IEEE International Conference on Industrial Informatics (INDIN)*, 2013, pp. 86–91.
29. S. Sudevalayam and P. Kulkarni, “Affinity-Aware Modeling of CPU Usage for Provisioning Virtualized Applications”, in *2011 IEEE 4th International Conference on Cloud Computing*, 2011, pp. 139–146.
30. J. Gao, H. Wang, and H. Shen, “Machine Learning Based Workload Prediction in Cloud Computing”, in *2020 29th International Conference on Computer Communications and Networks (ICCCN)*, 2020, pp. 1–9.
31. Memcached. <https://memcached.org/>.
32. B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking Cloud Serving Systems with YCSB”, in *Proceedings of the 1st ACM Symposium on Cloud Computing*, Indianapolis, Indiana, USA, 2010, pp. 143–154.
33. K. Mahajan et al., “Themis: Fair and efficient {GPU} cluster scheduling,” in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020, pp. 289–304.
34. W. Xiao et al., “{AntMan}: Dynamic scaling on {GPU} clusters for deep learning,” in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 533–548.
35. P. Zheng, R. Pan, T. Khan, S. Venkataraman, and A. Akella, “Shockwave: Fair and Efficient Cluster Scheduling for Dynamic Adaptation in Machine Learning,” in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 703–723.