Onyx: Efficient Transaction Processing with Real Processing-in-Memory Prototypes

Menglei Chen*
Huazhong University of Science and Technology, China

Yixiao Wang*
Huazhong University of Science and Technology, China

Yu Hua†
Huazhong University of Science and Technology, China

Abstract

A transaction ensures an all-or-nothing guarantee for a set of read/write operations. While modern transaction processing systems provide atomicity and consistency for data center applications, their performance is fundamentally limited by a critical memory bandwidth bottleneck, caused by massive numbers of read/write operations. Processing-in-memory (PIM) architectures offer a promising solution with their high aggregate bandwidth. However, directly applying PIM becomes inefficient in practice due to load imbalance and costly data transfers among PIM processors. To bridge the gap between transaction processing and PIM architectures, this paper presents Onyx, a high-throughput transaction processing system that efficiently executes transactions on a real UPMEM PIM prototype. To fully leverage the high parallelism and bandwidth of PIM, Onyx orchestrates the transaction execution workflow to maximize local accesses while maintaining load balance across PIM processors. It further employs rank-level asynchronous data transfer to reduce communication overhead. Extensive evaluation using real-world YCSB and TPC-C benchmarks shows that Onyx significantly outperforms state-of-the-art transaction processing systems.

CCS Concepts: • **Hardware → Emerging technologies**.

Keywords: Architecture and System Design, Compute/Storage Infrastructure, Emerging Technologies

ACM Reference Format:
Menglei Chen, Yixiao Wang, and Yu Hua. 2026. Onyx: Efficient Transaction Processing with Real Processing-in-Memory Prototypes. In *63rd ACM/IEEE Design Automation Conference (DAC '26), July 26–29, 2026, Long Beach, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3770743.3803987

1 Introduction

Modern data center applications, spanning from e-commerce platforms [14] to financial services [28], heavily rely on online transaction processing (OLTP) systems [11, 29, 31] to ensure critical data properties of atomicity, consistency, isolation, and durability (ACID). To improve performance, state-of-the-art OLTP systems have shifted towards storing entire datasets in main memory [24, 28, 30], thereby eliminating disk I/O bottlenecks. However, they suffer from concurrency control overhead from parallel transaction execution. Recent systems [11, 24, 25] address this problem by adopting deterministic execution, which predetermines a serial execution order for transactions before processing. This approach eliminates runtime conflicts and aborts, leading to predictable and efficient execution. However, these systems remain fundamentally constrained by the memory bandwidth, since transaction execution inherently involves massive, fine-grained random read and write operations.

The processing-in-memory (PIM) paradigm presents an emerging architectural solution to the "Memory Wall" [17, 26]. By integrating processing units with memory banks, PIM architectures provide substantially higher memory bandwidth and significantly reduce data movement, while achieving energy and latency savings. The recent commercialization of real-world PIM prototypes [1, 19, 20] has moved this paradigm from theory to practice. For instance, a single UPMEM PIM prototype [1] integrates thousands of DRAM Processing Units (DPUs), each equipped with its own local memory bank and a multi-threaded processor, delivering an aggregate bandwidth of up to 1.6 TB/s [2]. It is promising to leverage the real-world PIM prototypes to overcome the memory bandwidth limitations in transaction processing.

However, efficiently executing transactions on a real PIM architecture faces several key challenges. First, the limited local memory of each DPU (e.g., 64 MB in UPMEM PIM [2]) requires careful data partitioning across DPUs, complicating data layout and access locality. The write-after-write (WAW) and read-after-write (RAW) dependencies in transactions further exacerbate this challenge. Second, since thousands of DPUs operate in parallel, achieving effective load balancing is critical to fully utilize their high aggregate bandwidth and parallelism. Third, inter-DPU communication relies on the host CPU while requiring equal-sized data transfers across all participating DPUs due to data alignment [2], causing high data transfer overhead.

We present Onyx, a high-throughput transaction processing system designed for real-world PIM architectures. Onyx leverages a co-designed workflow that holistically

*Both authors contributed equally to this research.
†Corresponding Author: Yu Hua (csyhua@hust.edu.cn).

DAC '26, Long Beach, CA, USA

ACM ISBN 979-8-4007-2254-7/2026/07
https://doi.org/10.1145/3770743.3803987

optimizes data partitioning, transaction scheduling, and data movement. To facilitate concurrent transaction execution, Onyx adopts a hybrid multi-version storage scheme to eliminate WAW dependencies and efficiently manages RAW dependencies using a directed acyclic graph. Moreover, Onyx employs a locality-aware dispatcher that maximizes local DPU accesses while maintaining load balance, hence fully utilizing DPU parallelism and bandwidth. Onyx further adopts a rank-level asynchronous data transfer mechanism that pipelines communication with computation to minimize transfer overhead. We have implemented Onyx and evaluated it extensively on a real UPMEM PIM system with 1,020 functional DPUs. Evaluation using industry-standard YCSB and TPC-C benchmarks shows that Onyx significantly outperforms state-of-the-art in-memory transaction processing systems by 6.79× on average across diverse workloads. The source code will be released for public use in the near future.

2 Background and Motivation

2.1 Transaction Processing System

Online transaction processing (OLTP) is a fundamental component of modern datacenter applications. As illustrated in Figure 1, transaction processing provides a high-level abstraction that enables applications to execute multiple operations (e.g., reads and writes) with full ACID guarantees. To achieve high throughput, transactions are processed in parallel with concurrency control mechanisms. With increasing memory capacity, modern OLTP systems tend to store entire datasets in memory for high performance, while ensuring durability and availability through logging and replication [29, 30]. However, these in-memory systems have exposed concurrency control as a major bottleneck due to contention among parallel transactions. To address this problem, recent systems [11, 24, 25] adopt a deterministic execution scheme, which predetermines a serial order for transactions before execution, ensuring serializability and eliminating aborts caused by concurrency conflicts. A key driving force of such deterministic systems is the characteristic that transactions are typically written or decomposed as *one-shot* stored procedures, whose read/write sets are known before execution [11, 25]. Given these advantages, our work focuses on these deterministic transaction processing systems. Since the transaction processing involves substantial memory accesses [11, 25, 30], it poses significant requirements on memory bandwidth.

2.2 The Architecture of UPMEM PIM System

The PIM paradigm has been explored for decades to address the *"Memory-Wall"* [17, 26]. However, its practical prototype was constrained by memory technology limitations. Recently, UPMEM introduced a commercially available PIM hardware [1] for widespread adoption. As shown in Figure 2, this PIM prototype comprises a host CPU, conventional DRAM modules, and specialized PIM modules. Each UPMEM PIM module is structured as a double-rank DIMM with 16

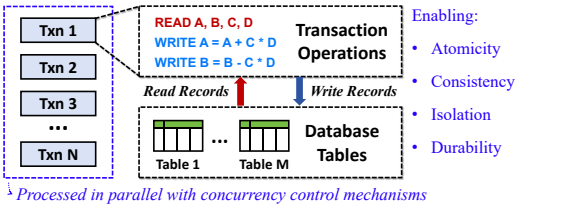


Figure 1. The illustration of transaction processing.

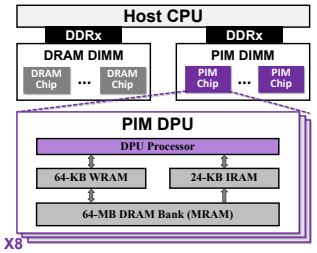


Figure 2. The architecture of a UPMEM-based PIM system.

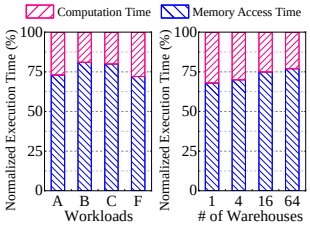


Figure 3. The execution time breakdown for Caracal.

PIM chips, where each chip integrates 8 DRAM processing units (DPUs). A DPU incorporates a 64-MB main DRAM bank (*MRAM*), a 64-KB scratchpad memory (*WRAM*), a 24-KB instruction memory (*IRAM*), and a dedicated processor. This architecture delivers 80 GB/s per DIMM, achieving a total aggregated bandwidth of 1.6 TB/s [13].

A DPU processor achieves high parallelism through 24 hardware threads, which share MRAM and WRAM while synchronizing using primitives such as mutexes. While data must be loaded from MRAM to WRAM before processing, the DPU software development kit [2] simplifies programming by providing a software-managed cache that enables transparent MRAM access. DPU programming follows a conventional accelerator model similar to CUDA [3]. The execution program (similar to a kernel function) is typically preloaded onto DPUs. The host CPU is responsible for transferring task data, launching DPU execution, and collecting results upon completion. The inter-DPU communication relies on the host CPU. For a single transfer, the data sizes sent to all participating DPUs must be *equal* to maintain data alignment and simplify hardware design [2].

2.3 Motivation

Transaction processing is inherently bandwidth-intensive, involving numerous read/write operations to database tables [11, 25, 30]. Our analysis of a transaction processing system Caracal [25] under YCSB [9] and TPC-C [4] benchmarks reveals that memory access accounts for 68–81% of execution time across various workloads. The evaluation results demonstrate the memory bandwidth as a primary performance bottleneck for transaction processing. The PIM architecture offers TB-scale high memory bandwidth, making it a promising solution to this bottleneck. Furthermore, UPMEM reports [1] indicate that its real PIM platform can reduce total cost of ownership by 10× and lower energy consumption by nearly 60%, making transaction

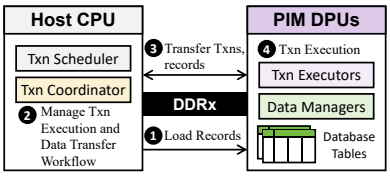


Figure 4. The system overview of Onyx.

processing on PIM prototypes highly cost-effective. However, the limited MRAM capacity of individual PIM DPUs necessitates partitioning database tables across multiple DPUs. This introduces the challenges for efficient PIM-based transaction processing: (1) orchestrating the transaction execution workflow to fully utilize the high parallelism and bandwidth of DPUs, and (2) minimizing data transfer overhead between the host CPU and DPUs.

3 The Onyx Design

We propose Onyx, an efficient multi-versioned transaction processing system on real PIM prototypes.

3.1 Onyx Overview

Figure 4 depicts the system overview of Onyx, which comprises two cooperative components, (1) *host CPU* for orchestrating execution control, data transfer, and transaction scheduling among PIM DPUs, and (2) *PIM DPUs* for storing database tables and executing transactions.

Workflow. We present the overall workflow of Onyx, consisting of four steps. ❶ The database tables are partitioned at the granularity of the table record (e.g., a row of the table) and distributed across PIM DPUs. ❷ The host CPU schedules transaction batches, dispatches them to DPUs, generates execution plans, and manages DPU transfer and execution. ❸ The transaction and record data are transferred between DPUs and the host CPU. The inter-DPU data transfer relies on the host CPU. ❹ DPUs execute transactions in parallel via multiple DPU threads, performing read/write operations on corresponding records. Onyx adopts multi-versioning to avoid synchronization overheads from WAW dependencies.

To enable high parallelism, Onyx batches transactions into *epochs* and employs epoch-based concurrency control. The transactions across epochs are serialized. Each epoch consists of two phases: *initialization* and *execution* phases. In the initialization phase (§3.3), Onyx performs multi-versioned concurrency control and generates a per-transaction execution plan, identifying locations of record versions for each read/write operation. In the execution phase (§3.4), Onyx dispatches transactions to DPUs and launches DPU kernels to execute the transactions.

3.2 Onyx Storage Scheme

Figure 5 illustrates the storage scheme of Onyx. Database tables are partitioned and distributed across DPUs, with each DPU storing a configurable, fixed number of records. Onyx achieves efficient multi-versioning based on *temporary versions* and a double-version structure for records (i.e., regular versions). Within an epoch, all writes to a record, except the last one, are stored as temporary versions, while

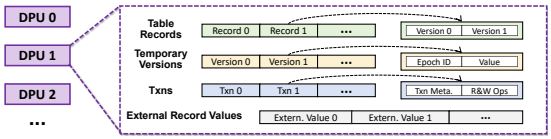


Figure 5. The storage schemes of Onyx.

the final write to this record is stored as a regular version. This design stems from the observation that only the final write remains visible to reads in subsequent epochs. Temporary and regular versions are stored in two separate contiguous memory regions. This separated scheme allows Onyx to reuse temporary versions in later epochs, completely eliminating per-version garbage collection overhead. Both temporary and regular versions carry an *epoch ID*. A temporary version is valid and readable only when its epoch ID matches the current epoch. A regular version uses its epoch ID to identify the newer version.

In addition, each DPU maintains a memory region to store the transaction metadata, such as the numbers of read/write operations and corresponding record locations. Since records are distributed across DPUs, transactions need to read/write records on remote DPUs. To facilitate such cross-DPU accesses, each DPU further reserves a memory region for storing external records. The sizes of these two memory regions depend on the epoch size (the number of batched transactions per epoch), which can be pre-configured to support various concurrency levels.

3.3 Multi-Versioning Initialization

Onyx analyzes the read/write sets of transactions within an epoch (Figure 6(a)) to generate execution plans. Records are partitioned and distributed across DPUs based on the hash values of their compacted keys (comprising table ID and original key), as shown in Figure 6(b). Each record is assigned a unique record ID, which serves as its access index within the corresponding DPU.

Onyx counts the number of writes for each record to compute its temporary version offset. For example, as shown in Figure 6(c), Record 0 has one temporary version in Epoch 0 since there are two writes on it. To avoid costly traversals upon version chains, Onyx pre-calculates the version locations (for both temporary and regular versions) of records involved in read/write operations. For instance, Transaction 0 reads previous versions of Records 0 and 1, while writing the temporary version of Record 0 (Figure 6(d)).

While Onyx efficiently handles WAW dependencies via multi-versioning, RAW dependencies still pose a challenge in synchronization among concurrent transactions. In Onyx, transactions are distributed across DPUs, exacerbating this synchronization overhead due to significant cross-DPU communication costs. To address this challenge, Onyx groups transactions within an epoch into micro-batches, ensuring that no RAW dependencies exist within each micro-batch. Onyx constructs a directed acyclic graph (DAG) representing RAW relationships among transactions. For

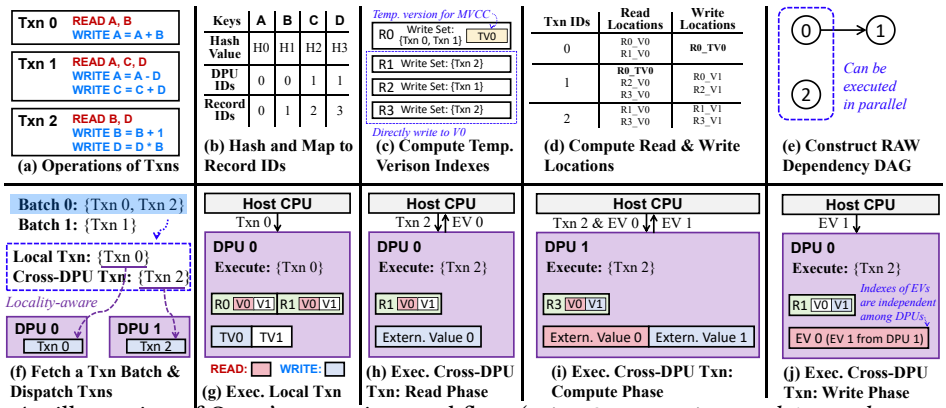


Figure 6. An illustration of Onyx's execution workflow *(using 3 transactions and 4 records as an example).*

example, in Figure 6(e), since Transaction 0 writes the temporary version of Record 0 and Transaction 1 reads it, an edge is added from Transaction 0 to Transaction 1. The Transaction 0 and Transaction 2 form a micro-batch and can be executed in parallel. Onyx leverages the Kahn algorithm [15] to efficiently generate batched topological orders and corresponding micro-batches in $O(N + M)$ time complexity, where N is the number of transactions and M is the number of RAW dependencies.

3.4 Locality-Aware Execution

Onyx orchestrates the transaction execution workflow to efficiently utilize the high parallelism and bandwidth of PIM DPUs while minimizing data transfer overheads.

Transaction Scheduling. As shown in Figure 6(f), Onyx sequentially processes micro-batches while concurrently executing all transactions within each micro-batch. Transaction dispatch is affinity-driven: each transaction is prioritized to be assigned to the DPU that maximizes its local data accesses, hence minimizing cross-DPU data transfer. To achieve load balance, Onyx distributes transactions evenly across DPUs, constraining the number of transactions per DPU not to exceed the total transactions divided by the number of DPUs (referred to as the DPU capacity). Onyx employs a greedy-based algorithm to efficiently dispatch transactions, including (1) selecting an unscheduled transaction, (2) computing its affinity to each relevant DPU, and (3) assigning this transaction to the DPU with both the highest affinity and available capacity. This algorithm achieves a sweet spot between the load balance and access locality.

Intra-DPU Execution. Transactions in Onyx fall into two categories: (1) *local transactions*, which only access the records within a single DPU, and (2) *cross-DPU transactions*, which involve the records across multiple DPUs. Onyx separately executes these two types of transactions within a micro-batch. For local transactions, once a DPU receives the relevant transaction metadata, it leverages multiple DPU threads to efficiently carry out all operations without host CPU involvement, as depicted in Figure 6(g). The read/write operations directly access the pre-calculated version locations with only trivial costs for identifying the newer regular versions. This intra-DPU execution mechanism is highly efficient and delivers high performance.

Cross-DPU Execution. Onyx processes cross-DPU transactions in three phases. (1) *Read Phase*: Each DPU reads the records required by transactions on other DPUs and writes these records to external values (Figure 6(h)). (2) *Compute Phase*: The host CPU transfers data across DPUs, and each DPU executes its transactions (Figure 6(i)). Local records are directly written, while remote records are written to external values. (3) *Write Phase*: Each DPU writes external values back to the corresponding versions (Figure 6(j)). Similar to previous phases, the inter-DPU data transfer relies on the host CPU. This cross-DPU execution mechanism is necessary for PIM-based transaction processing, since records are distributed across DPUs. While cross-DPU execution introduces overheads due to extra reads/writes on external values and inter-DPU data transfer, Onyx mitigates these costs by maximizing local data access via locality-aware dispatching. As a result, Onyx enhances overall transaction processing performance with high aggregate bandwidth and parallelism of PIM DPUs.

Rank-based Asynchronous Data Transfer. Onyx employs rank-granularity data transfer to optimize the data transfer between the host CPU and DPUs. This transfer approach strikes a balance between two inefficient extremes. Specifically, a coarse-grained strategy, such as transferring data to all DPUs simultaneously, incurs significant transfer amplification. This is because the transfer sizes must be uniform across DPUs to maintain data alignment, forcing some to transmit more data than necessary. Conversely, a fine-grained, per-DPU transfer scheme results in severe bandwidth underutilization. Prior designs demonstrate that DPU-level transfers sustain only 0.27 GB/s (CPU-to-DPU) and 0.12 GB/s (DPU-to-CPU), a mere 4.0% and 2.5% of the theoretical peak bandwidths [13, 22]. By adopting rank-level transfers, Onyx allows DPUs within a rank to pad their buffers only to the largest size within this rank, thereby minimizing padding overhead while saturating the available bandwidth. Furthermore, Onyx pipelines data transfer with

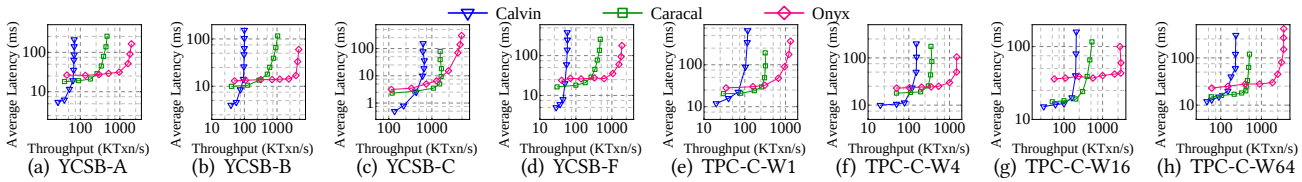


Figure 7. The throughputs and average latencies of different schemes under various workloads.

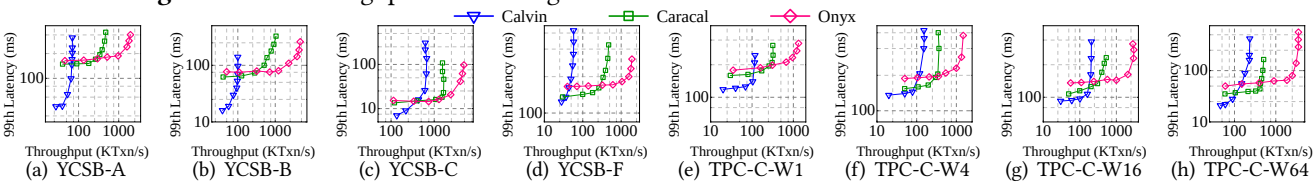


Figure 8. The throughputs and 99th latencies of different schemes under various workloads.

DPU execution through asynchronous copy and execution calls, leading to substantial performance improvements.

Handling Inserts, Deletes, and Aborts. Onyx treats insert and delete operations as writes. It uses a version's epoch ID to determine its validity: an insert marks the version as valid, while a delete invalidates it. If the final write to a record is a delete, Onyx reclaims the record by freeing its ID at the end of epoch. Moreover, although the deterministic execution scheme eliminates concurrency control-related aborts, Onyx still handles application-related aborts (e.g., constraint violations). These application-related aborts can be detected before writes operations. Upon an abort, Onyx replaces the values of write operations with the previous versions of corresponding records. This is necessary because version locations for subsequent reads are pre-calculated and fixed in Onyx. Consequently, Onyx also reads previous versions for write operations in transactions that may abort.

4 Performance Evaluation

4.1 Experimental Setup

Platform. Our experiments are conducted on a real PIM prototype equipped with two Intel Xeon Silver 4216 CPUs, 192 GB of DRAM memory, and 8 UPMEM PIM DIMMs [1]. The platform hosts 1,020 functional DPUs running at 350 MHz, with 4 DPUs being unavailable for allocation[1].

Comparisons. We compare Onyx with two state-of-the-art in-memory deterministic transaction processing systems: (1) ***Calvin*** [27], which is a representative single-versioned deterministic transaction processing system, and (2) ***Caracal*** [25], which employs multi-versioning to minimize concurrency control overheads.

Benchmarks. We evaluate Onyx and the compared schemes using two representative benchmarks: **(1)** ***YCSB*** [9]. The benchmark is configured with a single database table containing 1 million records. Each record uses the standard YCSB size of 1 KB, structured as ten 100-byte fields. Evaluations are conducted using four core YCSB workloads: *A* (50% Read, 50% Update), *B* (95% Read, 5% Update), *C* (100% Read), and *F* (50% Read, 50% Read-Modify-Write). The workloads are generated using a Zipfian distribution with the default skewness parameter ($\theta = 0.99$). Each transaction consists of a batch of 10 operations. **(2)** ***TPC-C*** [4]. The TPC-C benchmark models a complex OLTP environment for a warehouse management system with 9 tables. The workload is characterized by a 92% rate of read-write transactions and records of up to 672 bytes. To evaluate performance under varying contention levels, we scale the number of warehouses in the benchmark.

4.2 Results and Analysis

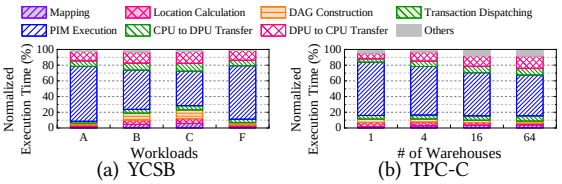
Figures 7 and 8 present the throughput-latency curves of different transaction processing systems. To plot a throughput-latency curve, we record the throughput and latency (e.g., average or 99^{th} latency) via varying epoch sizes (for Caracal and Onyx) and request rates (for Calvin). For Onyx, we leverage all 1,020 available DPUs by default to prevent out-of-memory (OOM) errors across workloads.

The results show that Onyx significantly outperforms CPU-based transaction processing systems by 4.62× and 8.13× on average for YCSB and TPC-C benchmarks, respectively. The improvements stem from the fact that Onyx fully leverages the high bandwidth and parallelism of PIM DPUs via an orchestrated execution workflow and an efficient multi-versioning scheme. Although Caracal also uses multi-versioning to resolve WAW dependencies and achieves higher throughput than Calvin, its array-based scheme incurs non-trivial overheads for maintaining version order, searching version arrays for read operations, and carrying out expensive garbage collection for per-record version arrays. In contrast, Onyx separately manages temporary and regular versions, enabling an efficient initialization phase and avoiding garbage collection costs. Onyx further employs DAG-based micro-batching to handle RAW dependencies while fully utilizing the high parallelism of PIM.

4.3 Performance Breakdown

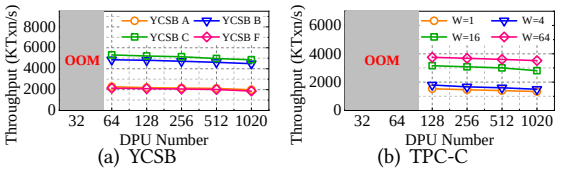
Figure 9 presents the latency breakdown of Onyx. The DAG construction and locality-aware dispatching account for 4.8% and 4.6% of the execution time on average, respectively. Although these mechanisms introduce runtime cost, they are critical for Onyx to manage the RAW dependencies and

[1]This is a common case consistent with prior studies [6, 13].



(a) YCSB (b) TPC-C

Figure 9. The execution time breakdown for Onyx.



(a) YCSB (b) TPC-C

Figure 10. The performance impacts of DPU number.

maximize local accesses for fully leveraging DPUs. Moreover, by adopting rank-level asynchronous data transfer, Onyx effectively limits the transfer costs to 25.1%. The PIM execution dominates the total execution time because Onyx maximizes local access and reduces transfer overheads.

4.4 Sensitivity Analysis

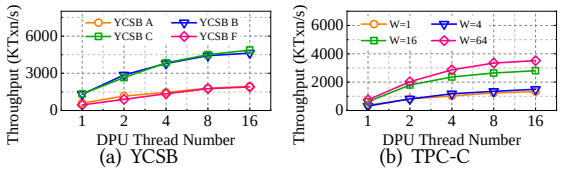
The number of DPUs. As shown in Figure 10, when the number of DPUs increases from 64 to 1020, Onyx exhibits only a marginal drop in throughput (i.e., 11.1%). This is because, although more DPUs hinder data access locality, they also deliver higher bandwidth and parallelism. Onyx explores and exploits this salient feature by using its locality-aware execution scheme. Moreover, when the number of DPUs falls below 47 for YCSB and 105 for TPC-C benchmarks, DPUs cannot store all records, raising out-of-memory (OOM) errors. Therefore, we use all 1,020 available DPUs in practice to ensure all records can be fully stored within the DPUs across various workloads.

The number of DPU threads. Figure 11 shows that scaling the number of DPU threads from 1 to 8 brings a 4.16× reduction in execution time. Further increasing threads beyond 8 offers marginal performance benefits, since 16 threads already fully utilize the DPU processor pipeline and exhaust available computational resources. These observations are consistent with existing studies [13].

Skewness of workloads. We evaluate the impact of workload skewness on performance, as shown in Figure 12. Higher Zipfian parameters indicate more skewed workloads. Onyx ensures load balancing across DPUs at the cost of more remote DPU accesses. Overall, Onyx provides higher throughputs than CPU-based transaction processing systems across different skewness levels.

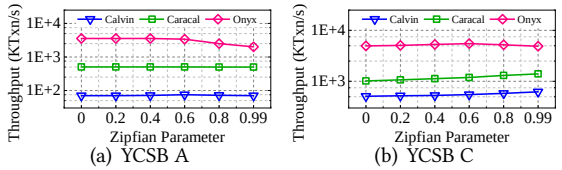
5 Related Work

Real-world PIM architectures. The PIM paradigm has been explored to mitigate the "Memory Wall" problem [17, 26]. However, its practical implementation has to address the constraints in memory technology. In recent years, various practical hardware architectures have been introduced [1, 18–20], moving PIM closer to widespread implementation. Specifically, Samsung [20] and SK Hynix AiM [19] integrate



(a) YCSB (b) TPC-C

Figure 11. The performance impacts of DPU thread number.



(a) YCSB A (b) YCSB C

Figure 12. The performance impacts of workload skewness.

compute units in 3D-stacked HBM and GDDR6, respectively, targeting machine learning workloads. AxDIMM [18] embeds FPGA logic in DIMM buffer chips to boost performance. In contrast, UPMEM [1] offers a commercial general-purpose PIM architecture by embedding processor cores directly into standard 2D DRAM arrays, providing a flexible platform for diverse applications such as our Onyx system.

Applications using real PIM prototypes. Existing studies [5–8, 10, 12, 16, 21–23] have leveraged commercial UPMEM PIM to accelerate diverse applications, including database engines (pimDB [5]), tree and learned indexes (PIM-tree [16], PIMLex [10]), graph processing (PimPam [6]), and deep learning frameworks (PIM-DL [21]). Among them, PIM-STM [23] provides a software transactional memory library that is orthogonal to our work and can be leveraged by Onyx to enhance intra-DPU transaction processing. Unlike these prior efforts, Onyx specifically demonstrates an efficient transaction processing system that executes transactions near memory, illustrating PIM's effectiveness in alleviating bandwidth bottlenecks for applications involving massive memory accesses.

6 Conclusion

This paper presents Onyx, an efficient transaction processing system that leverages a real-world PIM architecture to overcome the memory bandwidth bottleneck. Onyx is designed to fully exploit DPU parallelism and bandwidth by optimizing the transaction execution workflow for local memory access and balanced workload distribution, complemented by a rank-level asynchronous mechanism to deliver efficient host-DPU communication. Experimental results on a real UPMEM prototype demonstrate that Onyx achieves substantial performance gains over state-of-the-art systems. This work underscores the significant potential of PIM in accelerating data-intensive workloads and is a case in point for building high-performance transaction processing engines on real-world PIM hardware.

Acknowledgments

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant No. 62125202 and U22B2022.

References

[1] 2025. UPMEM PIM. https://www.upmem.com/technology/.

[2] 2025. UPMEM DPU SDK Documentation. https://sdk.upmem.com/2025.1.0/.

[3] 2025. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[4] 2025. TPC-C Benchmark. http://www.tpc.org/tpcc.

[5] Arthur Bernhardt, Andreas Koch, and Ilia Petrov. 2023. pimDB: From Main-Memory DBMS to Processing-In-Memory DBMS-Engines on Intelligent Memories. In *DaMoN*.

[6] Shuangyu Cai, Boyu Tian, Huanchen Zhang, and Mingyu Gao. PimPam: Efficient Graph Pattern Matching on Real Processing-in-Memory Hardware. *Proc. ACM Manag. Data* 2, 3 (2024).

[7] Liang-Chi Chen, Chien-Chung Ho, and Yuan-Hao Chang. 2023. UpPipe: A Novel Pipeline Management on In-Memory Processors for RNA-seq Quantification. In *DAC*.

[8] Sitian Chen, Haobin Tan, Amelie Chi Zhou, Yusen Li, and Pavan Balaji. 2024. UpDLRM: Accelerating Personalized Recommendation using Real-World PIM Architecture. In *DAC*.

[9] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *SoCC*.

[10] Lixiao Cui, Kedi Yang, Yusen Li, Gang Wang, and Xiaoguang Liu. 2025. PIMLex: A High-Performance Learned Index with Processing-in-Memory. In *FAST*.

[11] Jinkun Geng, Shuai Mu, Anirudh Sivaraman, and Balaji Prabhakar. 2025. Tiga: Accelerating Geo-Distributed Transactions with Synchronized Clocks. In *SOSP*.

[12] Christina Giannoula, Ivan Fernandez, Juan Gómez-Luna, Nectarios Koziris, Georgios I. Goumas, and Onur Mutlu. 2022. SparseP: Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Architectures. In *ISVLSI*.

[13] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. Benchmarking a new paradigm: An experimental analysis of a real processing-in-memory architecture.

[14] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An Optimized Storage Engine for Large-scale E-commerce Transaction Processing. In *SIGMOD*.

[15] Arthur B. Kahn. Topological sorting of large networks. *Commun. ACM* 5, 11 (1962).

[16] Hongbo Kang, Yiwei Zhao, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, Charles McGuffey, and Phillip B. Gibbons. PIM-tree: A Skew-resistant Index for Processing-in-Memory. *Proc. VLDB Endow.* 16, 4 (2022).

[17] William H. Kautz. Cellular Logic-in-Memory Arrays. *IEEE Trans. Comput.* 18, 8 (1969).

[18] Jin Hyun Kim, Shinhaeng Kang, Sukhan Lee, Hyeonsu Kim, Yuhwan Ro, Seungwon Lee, David Wang, Jihyun Choi, Jinin So, YeonGon Cho, Joon-Ho Song, Jeonghyeon Cho, Kyomin Sohn, and Nam Sung Kim. Aquabolt-XL HBM2-PIM, LPDDR5-PIM With In-Memory Processing, and AXDIMM With Acceleration Buffer. *IEEE Micro* 42, 3 (2022).

[19] Dae-Han Kwon, Seongju Lee, Kyuyoung Kim, Sanghoon Oh, Joonhong Park, Gimoon Hong, Dongyoon Ka, Kyu-Dong Hwang, Jeongje Park, Kyeong Pil Kang, Jungyeon Kim, Junyeol Jeon, Nahsung Kim, Yongkee Kwon, Kornijcuk Vladimir, 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. 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 J. Solid State Circuits* 58, 1 (2023).

[20] Young-Cheon Kwon, Sukhan Lee, Jaehoon Lee, Sang-Hyuk Kwon, Je-Min Ryu, Jong-Pil Son, Seongil O, Hak-soo Yu, Haesuk Lee, Soo Young Kim, Youngmin Cho, Jin Guk Kim, Jongyoon Choi, Hyunsung Shin, Jin Kim, BengSeng Phuah, Hyoungmin Kim, Myeong Jun Song, Ahn Choi, Daeho Kim, Sooyoung Kim, Eun-Bong Kim, David Wang, Shinhaeng Kang, Yuhwan Ro, Seungwoo Seo, Joon-Ho Song, Jaeyoun Youn, Kyomin Sohn, and Nam Sung Kim. 2021. 25.4 A 20nm 6GB Function-In-Memory DRAM, Based on HBM2 with a 1.2TFLOPS Programmable Computing Unit Using Bank-Level Parallelism, for Machine Learning Applications. In *ISSCC*.

[21] Cong Li, Zhe Zhou, Yang Wang, Fan Yang, Ting Cao, Mao Yang, Yun Liang, and Guangyu Sun. 2024. PIM-DL: Expanding the Applicability of Commodity DRAM-PIMs for Deep Learning via Algorithm-System Co-Optimization. In *ASPLOS*.

[22] Chaemin Lim, Suhyun Lee, Jinwoo Choi, Jounghoo Lee, Seongyeon Park, Hanjun Kim, Jinho Lee, and Youngsok Kim. Design and Analysis of a Processing-in-DIMM Join Algorithm: A Case Study with UPMEM DIMMs. *Proc. ACM Manag. Data* 1, 2 (2023).

[23] André Lopes, Daniel Castro, and Paolo Romano. 2024. PIM-STM: Software Transactional Memory for Processing-In-Memory Systems. In *ASPLOS*.

[24] Shujian Qian and Ashvin Goel. 2024. Massively Parallel Multi-Versioned Transaction Processing. In *OSDI*.

[25] Dai Qin, Angela Demke Brown, and Ashvin Goel. 2021. Caracal: Contention Management with Deterministic Concurrency Control. In *SOSP*.

[26] Harold S. Stone. A Logic-in-Memory Computer. *IEEE Trans. Comput.* 19, 1 (1970).

[27] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*.

[28] Qing Wang, Youyou Lu, Zhongjie Wu, Fan Yang, and Jiwu Shu. 2020. Improving the Concurrency Performance of Persistent Memory Transactions on Multicores. In *DAC*.

[29] Zhao Wang, Yiqi Chen, Cong Li, Yijin Guan, Dimin Niu, Tianchan Guan, Zhaoyang Du, Xingda Wei, and Guangyu Sun. 2025. CTXNL: A Software-Hardware Co-designed Solution for Efficient CXL-Based Transaction Processing. In *ASPLOS*.

[30] Ming Zhang, Yu Hua, and Zhijun Yang. 2024. Motor: Enabling Multi-Versioning for Distributed Transactions on Disaggregated Memory. In *OSDI*.

[31] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. 2022. FORD: fast one-sided rdma-based distributed transactions for disaggregated persistent memory. In *FAST*.