No Cap, This Memory Slaps: Breaking Through the Memory Wall of Transactional Database Systems with Processing-in-Memory

Hyoungjoo Kim
Carnegie Mellon University
hyoungjoo@cmu.edu

Yiwei Zhao
Carnegie Mellon University
yiweiz3@andrew.cmu.edu

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

Phillip B. Gibbons
Carnegie Mellon University
gibbons@cs.cmu.edu

ABSTRACT

Memory channel bandwidth imposes an upper bound on the performance of online transaction processing (OLTP) on in-memory database management systems (DBMS). Emerging processing-in-memory (PIM) hardware has the potential to overcome this barrier by using small cores in DRAM chips that can read and process data in situ, thereby avoiding moving these data across memory channels. However, naïvely offloading all database components to PIM does not solve the problem due to the characteristics of software components and the limitations of PIM hardware.

In this paper, we present OLTPim, the first end-to-end OLTP DBMS designed for PIM systems. We build a formalized model for the affinity of each database operation towards PIM and use it to decide the partitioning of components on different types of memory. We also design a lightweight batching algorithm to overcome the large PIM control latency while minimizing the batching overhead. We implement and evaluate OLTPim on the latest PIM system from UPMEM with 64 worker threads and 2048 PIM modules. Our results show that OLTPim achieves up to 1.71× throughput and up to 6.14× less per-transaction memory channel traffic over MosaicDB, a state-of-the-art in-memory system.

PVLDB Reference Format:
Hyoungjoo Kim, Yiwei Zhao, Andrew Pavlo, and Phillip B. Gibbons. No Cap, This Memory Slaps: Breaking Through the Memory Wall of Transactional Database Systems with Processing-in-Memory. PVLDB, 18(11): 4241 - 4254, 2025.
doi:10.14778/3749646.3749690

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/hyoungjook/OLTPim.

1 INTRODUCTION

Data movement between main memory (DRAM) and the CPU is often the major bottleneck in in-memory database management systems (DBMSs) for online transaction processing (OLTP) workloads [18]. This problem occurs because these DBMSs use pointer-chasing data structures that incur frequent CPU stalls on memory latency [11]. Modern DBMSs employ several tricks to hide this latency and maximize memory bandwidth utilization [29, 35]. However, their performance is bounded by the memory channel bandwidth. Although today's multicore CPUs have large on-chip caches to minimize memory channel traffic, the large working sets of OLTP workloads mean most data accesses hit main memory.

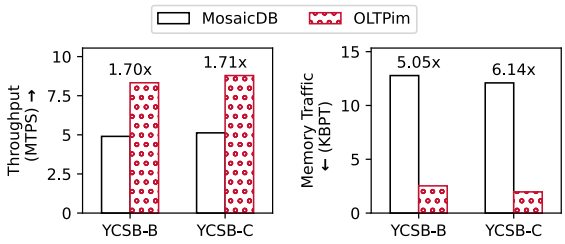


Figure 1: Throughput gain and memory channel traffic reduction of OLTPim over MosaicDB [35], a state-of-the-art research in-memory DBMS, on YCSB [12] with 1B tuples. OLTPim's numbers are measured on a real PIM system [66] with 2048 PIM cores; MosaicDB's numbers are on the same server, but replacing PIMs with DRAMs. *The arrow next to each y-axis legend points to the better direction for the metric.*

Emerging *processing-in-memory* (PIM) hardware overcomes this limitation by allowing in-place computation of memory-resident data without fetching them to the CPU [41, 58, 66]. PIM embeds small programmable cores (*PIM cores*) inside DRAM chips to achieve high-bandwidth/low-latency/low-energy access to data. Instead of processing all data at the CPU and incurring expensive data movement over memory channels, PIM enables offloading memory-intensive operations to PIM cores, reducing memory channel traffic.

However, achieving the benefits of PIM for an OLTP DBMS is not a matter of naïvely offloading its data structures and engines to PIM cores, due to several challenges (C1–C4). First, (**C1**) the PIM cores with their private local memories are akin to a thousand-node *shared nothing* database, inheriting all of its challenges: data placement, load balance, minimizing cross-node communication, etc. Second, (**C2**) PIM cores are quite limited compared to the multicore CPU, typically with slower clock frequencies, no transparent caches, no vector instructions, and limited synchronization primitives. Thus, as we will show, a PIM-optimized OLTP DBMS judiciously divides its components between the multicore CPU and the PIM cores, incorporating their different programming models.

Proceedings of the VLDB Endowment, Vol. 18, No. 11 ISSN 2150-8097.
doi:10.14778/3749646.3749690

But dividing the components raises the third challenge: (**C3**) The DBMS must ensure that any computation offloaded to the PIM cores amortizes its offload overheads. Lastly, (**C4**) despite these offloading overheads and less capable cores, the DBMS on PIM must meet latency constraints and provide superior transaction throughput over state-of-the-art OLTP systems on traditional hardware.

In this paper, we show how to overcome these challenges by presenting **OLTPim**, the first end-to-end OLTP DBMS optimized for PIM hardware. The crux of OLTPim's design is to offload pointer-chasing components (indexes, multi-versioning metadata, garbage collection) to the PIM cores while the CPU executes the rest (transaction management, orchestration, logging). Tuple data are stored in traditional (not PIM-equipped) DRAM in order to be accessed by the CPU without incurring offload overheads. OLTPim intelligently partitions the PIM-side structures to minimize cross-PIM communication. It also uses a lightweight batching algorithm that minimizes the overhead of orchestrating CPU threads and PIM programs. To evaluate OLTPim, we deployed it on a real PIM system from UPMEM [66] and compare against MosaicDB [35], a state-of-the-art in-memory OLTP DBMS, on YCSB [25] and TPC-C [13] workloads. As shown in Figure 1, OLTPim achieves up to 1.71× better transaction throughput with significantly less per-transaction memory channel traffic (up to 6.14×) compared to MosaicDB.

The main contributions of this paper are:

- We formalize the property of PIM-friendly operations and the design rationale for PIM-based OLTP DBMS.
- We identify the limitations of a real PIM system and provide an optimized SDK API and programming tricks to mitigate them.
- We design, implement, and evaluate the first end-to-end OLTP DBMS on a real PIM system. Our evaluation provides insights into the strengths and weaknesses of PIM for OLTP. We demonstrate that even first-generation PIM hardware improves throughput and reduces memory channel traffic.

2 BACKGROUND

We now provide an overview of PIM hardware and software systems and motivate the need for a PIM-optimized OLTP DBMS.

2.1 PIM Hardware

PIM embeds small on-chip processors in the logic layer of DRAM [25, 58]. Current PIM hardware uses single-core, in-order processors [66] (*PIM cores*). Thus, each PIM core is less powerful than a server-grade, multicore, out-of-order CPU (e.g., Intel Xeon). The key advantage of a PIM core is its low-latency access to locally memory-resident data that uses only a small amount of energy for each `LOAD`/`STORE` operation [25]. Moreover, since the accesses are on-chip and do not cross memory channels, the *aggregate memory bandwidth* is capped by neither per-chip pin constraints nor total memory channel bandwidth. Thus, for systems with 1000s of PIM cores, the aggregate memory bandwidth far exceeds the peak memory bandwidth of traditional servers at a fraction of the energy costs.

Figure 2 illustrates a PIM system consisting of a multicore CPU, traditional (passive) DRAM, and PIM components. The CPU has fast access to its on-chip last-level cache (LLC), but it can access traditional DRAM and PIM memory only via (slower) off-chip memory channels. The PIM side is partitioned into *PIM modules*, each

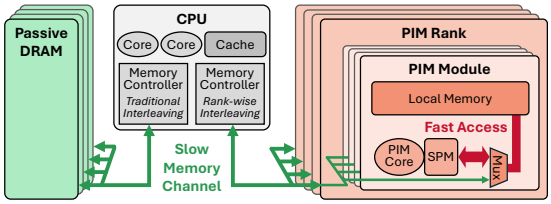


Figure 2: PIM Architecture. A high-level overview of a PIM system with CPU, DRAM, and PIM components. CPU and DRAM have a slow channel (green), whereas the PIM core and its local memory allow fast access (red).

with a *PIM core* and a *local memory*. Each PIM core has fast access to its local memory through its scratchpad memory (SPM) but cannot access the memory in other PIM modules. SPM is an SRAM component next to each PIM core, managed explicitly unlike the transparent CPU cache. The system maps the PIM modules' local memories to the CPU's address space so that the CPU can access them via the memory channels. Communication *between* PIM modules is done by the CPU reading from the source's local memory and writing to the destination's, incurring expensive off-chip accesses.

A fixed number of modules (e.g., 64) constitutes a *PIM rank*. The CPU cannot access each module independently because their addresses follow a *rank-wise interleaving*. Thus, if the CPU has a matrix of data and wants to transfer row i to the ith module in a rank, it should first transpose the matrix to make the interleaved layout and then copy it to the memory-mapped address of the rank. In contrast, the CPU can access each *rank* independently because there is no such interleaving between ranks. Hence, a rank is the smallest unit that can be controlled independently by the CPU. Although the rank-wise interleaving is required to provide a contiguous layout to each PIM core, it differs from traditional (byte-wise) interleaving, which is optimized for cache-line-sized (64B) accesses [15, 54]. Furthermore, a PIM rank's capacity is smaller than an equivalent passive DRAM rank's due to its additional components. Consequently, as depicted in Figure 2, a PIM system also employs passive DRAM with cache-line-optimized access and larger capacity.

Although there are PIM research prototypes (e.g., Samsung [47]), UPMEM [66] is currently the most commonly used commercially available PIM hardware. UPMEM's system provides 2048 PIM modules, each with 64MBs of local memory. Its PIM core is an in-order RISC processor (350 MHz) with a custom instruction set architecture. UPMEM provides an SDK API (C lang) for broadcasting compiled code to each PIM module and then later launching them on demand. The API also enables gathering and scattering data between the CPU and the PIM modules; the rank-wise interleaving transpose discussed above is done transparently to the programmer.

One limitation of the UPMEM hardware is *mux switch latency*. Although the CPU and PIM core can access the same local memory in a module, UPMEM does not allow simultaneous access from both processors. As shown in Figure 2, the mux on each module determines whether the CPU or PIM core has control of the memory. If a PIM program wants to modify the data after copying it from the CPU, it must switch the mux position. The microarchitecture requires this switching to mitigate access conflicts between the CPU and PIM cores [15]. Our measurement shows that the mux switch takes 67 ± 11μs on the default UPMEM SDK (2025.1.0) and 21 ± 4μs on our optimized SDK (see §6.1). To our knowledge, this

work is the first paper emphasizing UPMEM's mux switch latency. Previous works on UPMEM studied batch-oriented applications where millions of elements are loaded and processed at once [3, 25, 42, 53, 57, 62], thereby amortizing the mux switch latency. In contrast, OLTP workloads cannot tolerate the latency of such large batches, so one must explicitly account for mux switch latency.

2.2 PIM-enhanced Software Systems

The PIM core's fast local memory access is ideal for systems that make heavy use of pointer chasing-based indirect structures [34], such as linked lists and B+trees [9, 10, 55]. Prior work on PIM-enhanced data structures uses *range-partitioning* to divide data across 1000s of modules, but such partitioning suffers from heavy load imbalance on skewed workloads. PIM-tree [42] uses provably robust techniques to mitigate such imbalance but it requires multiple rounds of mux-switches to complete one index traversal.

PIM's large number of parallel cores and scalable internal bandwidth makes it an attractive environment for throughput-oriented memory-intensive applications [1, 14, 33, 71]. (However, the workload must not be compute-intensive since each PIM core has limited computing power [25].) Such applications include analytics [2, 3, 37, 46, 54, 73], machine learning (ML) [5, 8, 24, 36, 53, 61, 72, 74], graph processing [14, 25, 51, 62, 64], genomic analysis [7, 50, 57], cryptography [19, 23, 26], and error-correcting codes [20, 21]. A common technique to optimize such workoads for PIM is to convert expensive computations into memory-intensive operations, such as using lookup tables for matrix multiplications in ML systems [53].

OLTP DBMSs are a promising target for PIM because they often incur unavoidable off-chip memory accesses. A CPU's LLC is unable to absorb them due to the OLTP workloads' random access patterns. Since OLTP DBMSs rely on data structures (e.g., B+trees) to find records, they can use PIM cores to accelerate pointer-chasing operations. Moreover, PIM provides higher aggregate memory bandwidth for these memory accesses at lower energy costs (as discussed in §2.1). To our knowledge, no existing OLTP DBMS uses PIM to improve their performance and efficiency.

3 OLTPIM OVERVIEW

We now describe the design of OLTPim, the first end-to-end OLTP DBMS optimized for PIM. OLTPim is designed to address the four challenges (C1–C4) presented in §1 for OLTP on PIM. It targets high-throughput settings where one PIM-enhanced server (e.g., UPMEM) processes up to millions of transactions per second.

3.1 High-Level Design

Figure 3 illustrates the design of OLTPim. The worker threads execute transactions by issuing queries to the CPU engine. Unlike existing in-memory DBMSs, OLTPim bifurcates its data structures in different memory types: PIM and DRAM. As depicted in the right part of Figure 3, indexes and version chains are placed in PIM and partitioned across the modules, whereas tuples are placed in DRAM. As a result, the CPU engine first traverses the index and version chain by offloading the operations to the PIM engines, then fetches the corresponding tuples from DRAM.

Each *query* issued by M worker threads targets one of the N PIM modules. Since any transaction can access data structures in any

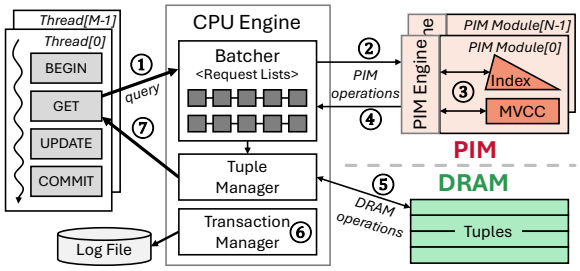


Figure 3: OLTPim Design. Query requests from M worker threads are served by N PIM modules. The batcher ① combines the requests in the request list and ②-④ offloads index and MVCC operations to the PIM engines. Then, the CPU engine ⑤ fetches the tuples from DRAM, ⑥ records them in the transaction context, and ⑦ returns them to the thread.

module, this becomes an M:N multiplexing problem. The *batcher* inside the CPU engine is in charge of the efficient multiplexing. ① When a transaction issues a query to the CPU engine, the engine converts the query into *PIM operations* and appends them to the batcher's *request list*. ② The batcher combines and copies them into the PIM modules. ③ Then, the PIM engine in each PIM core processes the operations, traversing the indexes and version chains stored in its local memory. ④ After this completes, the batcher copies the results back to the CPU where its engine processes the rest of the query. ⑤ Using the value received from the PIM engine, the CPU engine accesses the requested tuples from DRAM. ⑥ It also manages the transaction context, such as appending to the write set and writing the log. ⑦ Lastly, the CPU engine returns the tuple to the transaction, and the worker thread resumes execution.

3.2 Design Rationale

OLTPim stores some data structures in DRAM instead of offloading all operations to PIM. Furthermore, it uses a centralized batcher to coordinate PIM requests instead of allowing worker threads to query modules directly. As we now discuss, these design choices are based on the observation that most OLTP workloads are memory-bound, and thus, memory traffic is the primary factor for determining performance rather than the amount of computations.

Near-Memory Affinity. Naïvely offloading database structures and operations to PIM does not guarantee an advantage over non-PIM designs due to the operations' overheads (Challenge C3). For example, if the DBMS places database tuples in PIM local memories, then fetching a tuple to the CPU in service of a transaction would incur the same memory channel traffic as if it placed the tuple in DRAM. Thus, PIM would have no advantage and, in fact, would hurt performance due to mux switching and rank-wise interleaving.

As the previous example shows, memory channel traffic reduction is an important metric for describing the benefit of PIM offloading. We model this reduction as the *near-memory affinity* A_{near} of an operation. Let m be the number of memory accesses in cache line units that the operation requires. An access that traverses a memory channel is called a *far-memory* access. Assuming the working set size is larger than the CPU LLC, executing the operation without PIM requires m far-memory accesses (i.e., the *far-memory traffic* $M_{\text{noPIM}} = m$). In contrast, if a system instead offloads the operation to a PIM module with the data, these accesses become

Table 1: Near-Memory Affinity A_{near} of Database Operations.

Database Operation	Near-Memory Affinity
Tuple Access	$-i$
B+tree Traversal	$H_{ooc} \log L - 2i$
Version Chain Traversal	$C_{vis} - 2i$

where:

H_{ooc}	Height of B+tree's out-of-cache part
L	B+tree node size in cache line units
C_{vis}	Version chain length until the visible version
i	Size in cache line units of IDs, keys, data fields (e.g. 32bit/64B = 1/16)

near-memory accesses within the module. But one must also account for the memory traffic of sending the operation's arguments from CPU to PIM and retrieving its return values from PIM to CPU. If this incurs r cache lines of traffic then the far-memory traffic $M_{\text{PIM}} = r$. We define A_{near} as the amount of far-memory traffic reduction by using PIM:

$$A_{\text{near}} = M_{\text{noPIM}} - M_{\text{PIM}} = m - r \tag{1}$$

Table 1 shows the estimated A_{near} for operations required to locate a small number of tuples through an index, which is the most common case in OLTP queries [63]. First, fetching a tuple requires far-memory traffic as large as its size, assuming it is not in the CPU's LLC. If a DBMS offloads it to PIM, the CPU sends the tuple's ID to PIM and receives its data. Hence, $r = i + m$ and $A_{\text{near}} = -i$. Next, traversing a B+tree requires searching H nodes from the root to a leaf while binary-searching each node by fetching $\log L$ cache lines from memory. Since the CPU LLC stores the top part of the tree, the required far-memory traffic is $m = H_{ooc} \log L$. To offload the index lookup, the CPU sends the key to PIM and receives the tuple ID/pointer, while the tree traversal becomes near-memory accesses. As a result, $r = 2i$ and $A_{\text{near}} = H_{ooc} \log L - 2i$. Similarly, finding the visible version in a version chain requires $m = C_{vis}$ far-memory accesses, but offloading the operation reduces it to $r = 2i$ for the object ID and the return value, yielding $A_{\text{near}} = C_{vis} - 2i$.

The takeaway from Table 1 is that an operation's near-memory affinity depends on both its memory footprint and input/output sizes. If an operation hides its heavy memory accesses to a PIM module behind small inputs and outputs (e.g., B+tree key/value), it becomes more near-memory-friendly. In contrast, if the operation reads or writes the data (e.g., fetching a tuple), the size of the inputs and outputs becomes as large as the operation's memory footprint, making it less near-memory-friendly.

Note that our near-memory affinity model serves as a guideline for system design decisions, but it is insufficient for modeling the actual performance, which also includes factors such as address interleaving, bulk transfers, and computational intensity.

Condition for Less Memory Channel Traffic. From Equation (1), we have that memory channel traffic decreases if:

$$A_{\text{near}} > 0 \tag{2}$$

Based on Equation (2), OLTPim offloads only the near-memory-friendly operations to PIM and keeps the rest in DRAM to ensure memory channel traffic reduction. According to Table 1, B+tree and version chain traversals are near-memory-friendly, whereas tuple access is not. Hence, OLTPim places indexes and version chains in PIM and leaves tuples in DRAM.

Batching Independent Operations. Although Equation (2) ensures less memory traffic, performance improvement is not guaranteed. Accessing the PIM-side structure requires ① copying the input to the module, ② executing the PIM program, and ③ copying the output back to the CPU. Since the PIM core accesses the local memory during ②, but the CPU does in ① and ③, two mux switches are required. As a result, the latency of accessing the PIM structure is greater than accessing the same structure in DRAM.

Given this, the system should execute the operations in batches to hide this latency and maximize throughput. The CPU sends the inputs for a batch of B operations to PIM, executes them in parallel, and receives their outputs, amortizing the mux switch latency over B operations. In addition to the larger throughput, large-scale batching can improve the load balance across the modules [41, 42].

Condition for Higher Throughput. Batching hides latencies and makes workloads bandwidth-bound. When executing B operations in DRAM, the runtime is $T_{\text{noPIM}} = B \cdot M_{\text{noPIM}}/\mathcal{W}$, where $\mathcal{W}$ is the aggregated memory channel bandwidth in cache line units. On the other hand, if the system offloads them to PIM as a batch of size B, the runtime is $T_{\text{PIM}} = B \cdot M_{\text{PIM}}/\mathcal{W} + \mathcal{L}$, where $\mathcal{L}$ is the *PIM round latency* consisting of two mux switches and additional batching overhead. Hence, the throughput for the batch is improved if $T_{\text{PIM}} < T_{\text{noPIM}}$, which simplifies to:

$$B > \mathcal{W}\mathcal{L}/A_{\text{near}} \tag{3}$$

Equation (3) shows that the batch size for the PIM operations should be larger than $\mathcal{W}\mathcal{L}/A_{\text{near}}$ to achieve better throughput. If the memory channel bandwidth is $\mathcal{W} \approx (256\text{GB/s})/(64\text{B})$ and PIM round latency is $\mathcal{L} \approx 200\mu\text{s}$, then $\mathcal{W}\mathcal{L} \approx 10^6$. Hence, for typical operations where $A_{\text{near}} \ll 10^6$, batched execution is required to hide the PIM latency. In OLTPim, the batcher in Figure 3 is responsible for batch-executing the PIM operations.

However, we cannot increase B indefinitely as it degrades transaction latencies. We aim to keep P99 latency acceptable (e.g., sub-20ms), while maximizing throughput and reducing memory channel traffic (Challenge C4). Moreover, the system must track the contexts of all B in-flight transactions. If their total size exceeds the CPU LLC capacity, additional LLC misses increase the memory traffic and degrade performance. We study the batch size trade-off in §7.3.

To alleviate the batch size in Equation (3), a system shoud minimize PIM round latency ($\mathcal{L}$). Hence, we optimize the batcher to avoid OS overheads and improve its PIM access API efficiency (§5).

Merging Dependent Operations. Independent operations are combined into a batch to maximize the throughput, but it is impossible if they are dependent on each other. For example, assume the B+ tree nodes are randomly distributed across the PIM modules. If a parent and its child node are in different modules, the CPU should first traverse the parent node, receive the child pointer, and then traverse the child node. This requires runtime of at least $2\mathcal{L}$ since each traversal requires a PIM round latency. As shown in the example, PIM operation dependency arises when different modules have relevant structures. To minimize the dependency, the DBMS merges operations by partitioning the PIM-side structures and placing relevant structures in the same PIM module (Challenge C1). In

addition to reducing the transaction latency, merging operations into one increases its A_{near}, further alleviating Equations (2) and (3).

OLTPim follows two major design rationales that improve the throughput and memory channel traffic. First, the DBMS places its data structures in either DRAM or PIM memory depending on operations' near-memory affinity. The DBMS partitions PIM-side structures across the modules to facilitate dependent operation merging. Second, the DBMS batches operations to hide the PIM round latency while minimizing the coordination overhead. We elaborate on the design choices for OLTPim's structural components and batching mechanism in §4 and §5, respectively.

4 STRUCTURAL COMPONENTS

OLTPim stores its components in either DRAM or PIM memory, spreading PIM-side structures across the modules. As discussed in §3, it stores indexes and version chains in PIM and tuples in DRAM. Figure 4 illustrates the hash-partitioned PIM-side structures and an example workflow for a PIM engine to serve a query. It takes the index key as an input, traverses the index to get an *object ID* (OID), finds and traverses the corresponding version chain, and returns the pointer of the visible tuple to the CPU engine. We now describe the design of OLTPim's DRAM- and PIM-side components.

4.1 Configurably Hash-Partitioned Indexes

Because index traversal requires the DBMS to chase pointers with unpredictable access patterns, it incurs off-chip data movements if the index is larger than the LLC [9]. OLTPim offloads index operations to the PIM cores by partitioning them across modules.

One approach is to use range-partitioning to divide the key space into disjoint subsets and assign each partial index to a module [10, 55]. An extension to this is PIM-tree's skew-resistant paritioning that divides both the key space and the index levels, and then assigns each sub-index to a module to guarantee load balance [42]. However, since the nodes on a traversal path reside on different modules, PIM-tree requires multiple dependent PIM operations per look-up, which is not ideal in latency-sensitive applications.

OLTPim uses a variant of the range-partitioning for its indexes. As depicted in Figure 4, a hash function maps the key right-shifted by R bits into one of the modules, partitioning the key space into disjoint ranges of 2^R keys. When the DBMS needs to access an index, it computes the target *PIM ID* using the hash function and then the CPU engine sends a request to the corresponding PIM module. OLTPim use B+trees as local indexes in each module, but they can be replaced with any other ordered data structures such as radix trees [52], Masstree [56], or skip lists [70].

With a range query, the CPU engine broadcasts PIM operations to all modules that may contain a key in the target range. Although the operations' independence allows parallel execution, the DBMS minimizes the number of accessed modules to reduce resource usage. Because OLTPim maps 2^R consecutive keys to the same module, a larger R reduces the number of modules involved. On the other hand, a larger R risks load imbalance among the PIM modules because queries may target hot keys in a single module. In our experiments, we use $R = \log_2 S$ for indexes supporting scans with range length S and $R = 0$ for indexes without scans.

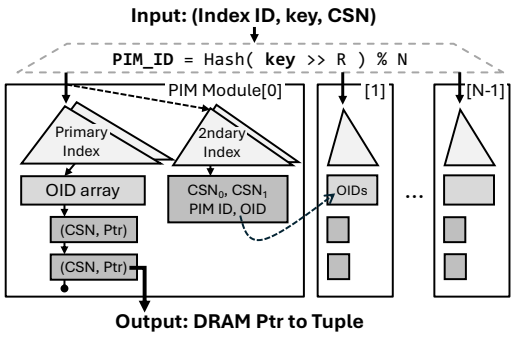


Figure 4: Structural Components of the PIM Engines. A PIM engine uses the index key to return the DRAM pointer to the visible tuple in the key's version chain. Indexes and version chains are partitioned based on `Hash(key»R)` for a configurable R.

OLTPim's hash partitioning randomly distributes the keys across PIM modules to balance local index sizes. In cases of workload skew, the DBMS can reconfigure the hash function to balance the per-module access frequency by placing fewer keys in a module if they are popular. But changing the hash function at runtime requires moving keys to other modules, which incurs expensive off-chip memory traffic (see §2.1). For re-partitioning, OLTPim rebuilds the PIM indexes from scratch instead of partial adjustment.

The DBMS's hash-partitioned indexes serve a query with one PIM operation, minimizing mux switches and off-chip memory traffic. Even with the minimized offloading cost, if the entire index fits in LLC, placing it in DRAM is always better. This also applies if the index's working set is smaller than LLC due to a skewed access pattern. Hence, OLTPim supports selectively placing indexes in either PIM or DRAM to achieve the best performance.

4.2 Tuple Storage

OLTPim places tuples in DRAM because $A_{\text{near}} < 0$ for tuple access operations (§3.2). Furthermore, PIM's rank-wise interleaving (§2.1) exacerbates the disadvantage. Assume OLTPim places each tuple in a PIM module. When it fetches one tuple from a module, it should read data from adjacent modules as well because a cache line from the CPU's address space is striped across the modules in a rank. As a result, the needless data from other modules multiplies the required memory traffic even if the query only needs one tuple.

Placing tuples in DRAM ensures flexible access from the CPU engine, but makes PIM engines less efficient for tuple-heavy operations, such as aggregations. OLTPim targets OLTP workloads where most queries access a small number of tuples at a time via indexes [63]; it is not optimized for more complex OLAP queries.

4.3 Multi-Versioning

OLTPim uses multi-version concurrency control (MVCC) to manage transactions. Previous in-memory MVCC implementations maintain version metadata in each tuple's header [16, 69], including the version timestamp (i.e., commit sequence number, CSN) and the pointer to the next version (i.e., *version chain*). But inlining this metadata in DRAM with tuple data would mean that only the DBMS's CPU engine could traverse a tuple's version chain. A better approach is for the DBMS to store tuples' MVCC metadata in PIM

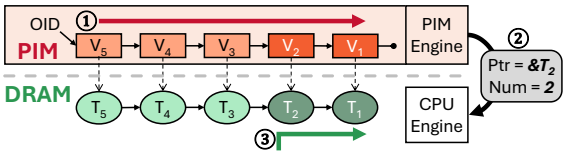


Figure 5: Garbage Collection in OLTPim. The PIM engine ① GC the version chain and ② returns the first pointer and the number of the obsolete tuples, which is used by the CPU engine to ③ GC the tuple chain.

memory since their traversal has a high near-memory affinity (Table 1). The DBMS maintains newest-to-oldest (N2O) version chains in the module with its primary index using the same hash function (§4.1), so that it can merge version chain operations with relevant index operations (Challenge C1). The PIM engine traverses an index and a version chain in the same module, and returns the visible tuple's DRAM pointer to the CPU engine.

MVCC DBMSs include a garbage collection (GC) mechanism to prune obsolete versions and reclaim their storage [69]. In OLTPim, each PIM engine identifies obsolete versions as they traverse version chains, but the CPU engine does not have access to this version metadata to remove DRAM-resident tuples. To overcome this issue, OLTPim maintains a redundant *tuple chain* in DRAM along with tuple data. As shown in Figure 5, after ① the PIM engine traverses the version chain and collects obsolete version metadata (V_2 and V_1), ② it returns (i) the pointer to the first obsolete tuple ($\&T_2$) and (ii) the number of obsolete tuples (n=2) to the CPU engine. From this information, ③ the CPU engine can collect obsolete tuples (T_2 and T_1) without using the version metadata. This only requires 2 DRAM accesses (T_2 and T_1), whereas 5 accesses (T_5 through T_1) would be required if the version metadata were placed in DRAM.

We compare OLTPim with another design where the version metadata is placed in DRAM and connected from oldest to newest (O2N) [69]. During GC, there is no need to traverse the entire chain in such a design (T_1 and T_2), incurring the same number of far-memory accesses as OLTPim. However, OLTPim still wins in the common case of accessing the newest tuple, which requires even more far-memory accesses if traversing the chain from the oldest. In sum, storing version metadata in PIM and separating them from DRAM tuples reduces the memory channel traffic (Challenge C3).

When adding a new version at the end of a tuple's PIM-side chain, the PIM engine acquires a *write lock* on the last version by setting a bit in its metadata. Since only the PIM engine can access its local version chains, a global lock manager is not required. The PIM-side write lock also protects the tuple chain in DRAM because OLTPim GCs it only if the transaction acquires the corresponding write lock for an update query (cooperative GC).

4.4 Secondary Indexes

The DBMS stores a tuple's primary index entry on the same PIM module with its version chain. However, this is impossible for secondary index entries. Assume the primary and a secondary index for the same table use the hash functions h_p and h_s, respectively, to map the key to the PIM modules. To place the secondary index entry on the same module, the DBMS needs a h_s that satisfies $h_p(k_p(t)) = h_s(k_s(t))$ for all tuples t, where k_p and k_s are the primary and the secondary key of the tuple. This is impossible in practice as k_p and k_s are unrelated in general. Hence, the DBMS has to store a tuple's secondary index entry in a different module from the corresponding version chain, and thus querying a secondary index requires at least two dependent PIM operations (Figure 4).

The PIM engine uses the secondary index to return the pair (PIM ID, OID). Then, OLTPim uses the values to locate the version chain in a different module and traverse it. In addition, a secondary index can return invisible entries redundantly if the tuple is removed in another module. OLTPim prevents this by additionally storing two CSNs that describe the visibility of the tuple.

4.5 Transaction Management and Logging

When a transaction commits/aborts, the DBMS makes its changes visible to others using its *write set* (i.e., list of modified tuples). Since OLTPim stores version chains in PIM, the CPU engine sends the OIDs in the write set to the corresponding PIM modules. Then, the PIM engines publish (commit) or remove (abort) the modifications and release the write locks to ensure consistency. Sending the OIDs to PIM incurs additional off-chip memory traffic. However, the benefit of offloading the version chain exceeds the drawback at commits/aborts, improving the end-to-end throughput (§7.6).

An alternative design would keep redundant write sets in PIM engines. This could reduce memory traffic at commit/abort by sending a single transaction ID instead of potentially multiple OIDs to each module. In our setting, however, the PIM write set typically has at most one entry per module since tens of entries (average 18 on TPC-C [13]) are spread across thousands of modules (N=2048), so we chose our simpler design. Keeping redundant write sets in PIM would be beneficial for workloads with larger write sets.

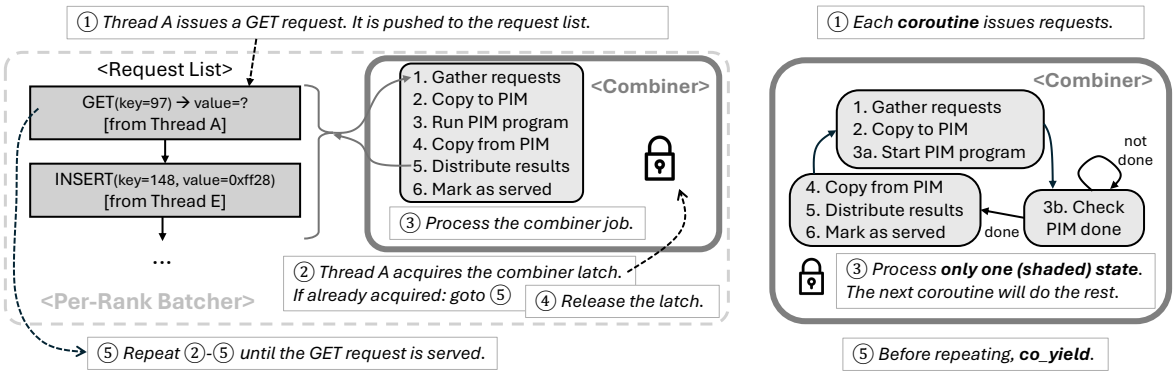
OLTPim guarantees durability by logging the information already available to the CPU engine. Before a transaction commits, for each tuple in the write set, the CPU engine writes a log consisting of its index ID, PIM ID, OID, and tuple data. Note that the tuple data is stored in DRAM, and the rest are available in the write set. The updates on PIM-side structures (indexes and version chains) are not flushed to the logs. Instead, on recovery, the PIM engine rebuilds the indexes for valid tuples and initializes the version chains to be single-entry. The index keys and visibility information are computed from the tuple data and update history available in the log. Although rebuilding PIM-side structures degrades the recovery performance, we optimize OLTPim for normal operation by avoiding further data movements from PIM to CPU and disks.

5 BATCHED EXECUTION

The batcher in OLTPim executes PIM operations in batches to hide PIM round latency (recall Figure 3). It should be lightweight since it is on the performance critical path, being called as frequently as the rate of all PIM operations. Although our batcher is designed for OLTP, it can also be used on other PIM-based latency-critical applications that serve multiple concurrent streams of requests. This section elaborates on the basic execution algorithm of the batcher and the optimizations to minimize batching overheads.

5.1 Per-Rank Batching

When a transaction issues a query, the CPU engine converts it into one or more *PIM requests* that target certain PIM modules. A PIM



(a) Flat combining for the batching job and PIM program.

(b) Modifications for using coroutines.

Figure 6: Batcher Execution. (a) Our base algorithm uses flat combining to process the combiner job without additional worker threads. (b) In our modified algorithm, each worker thread schedules multiple coroutines, and while the PIM rank is busy, it switches to other coroutines to interleave the executions.

request contains all the information required to serve an operation, such as operation type, input arguments, output placeholders, and target PIM ID. Since all clients in the system issue the requests concurrently to different modules, OLTPim serves the concurrent M:N multiplexing of requests from all clients to all modules (§3.1).

Consider two extreme design choices for the batcher: (i) a system-wide batcher that processes all requests bulk-synchronously, limiting the concurrency between requests on different modules and degrading the response time, or (ii) per-module batchers that maximize fine-grained concurrency but cannot be controlled independently due to the interleaving of modules in a rank (§2.1). Residing in a sweet spot between these two, OLTPim uses a batcher for each rank. Because the DBMS can control different ranks independently, the per-rank batchers maximize concurrency as far as the hardware permits. They are also more robust to load imbalance (stragglers) than a system-wide batcher since lightly-loaded ranks need not wait for heavily-loaded ranks to complete.

5.2 Flat Combining Avoids Oversubscription

Each per-rank batcher executes a *combiner job*, consisting of CPU-side tasks to collect the batch of requests, copy them to and execute them on the rank's modules, and distribute the results to the source transactions. Using designated threads for combiner jobs, however, would cause core oversubscription—incurring OS overhead and degrading the performance [35]—because the CPU cores are already fully occupied by worker threads for executing transactions.

To avoid oversubscription, our base algorithm (Figure 6a) uses *flat combining* [31, 55], which enables concurrency between transactions and combiner jobs without introducing additional threads. Each worker thread executes a stream of transactions, and each per-rank batcher has a combiner latch in addition to a request list. ① When a transaction issues a PIM request, the thread appends it to the request list of a per-rank batcher and ② tries to acquire the combiner latch. ③ The thread with the latch becomes the combiner thread, executing the combiner job for all requests in the rank and then it ④ releases the latch. ⑤ If the thread fails to acquire the latch, it waits for the other threads to execute its request. This approach avoids OS overhead if the combiner job does not use system calls or spawn threads. We assume that all PIM accesses are through `LOAD`/`STORE` instructions on the memory-mapped addresses of the PIM rank. This is not the case in the default SDK from UPMEM [66], but is enabled by our custom API (see §6.1).

5.3 Coroutines for CPU-PIM Interleaving

The algorithm in Figure 6a is designed to avoid OS overhead. However, it still wastes CPU cycles waiting for the combiner job and the PIM program to complete (③ in the Figure) if another thread is already working on the same rank. Our improved algorithm, depicted in Figure 6b, avoids such inefficiency using coroutines–i.e., lightweight user-level threads. In-memory DBMSs often use coroutines to hide DRAM latency by switching to another coroutine while prefetching the desired data [29, 35].

OLTPim uses coroutines to hide the latency of combiner jobs and PIM programs (instead of hiding DRAM latency). First, ① OLTPim uses multiple coroutines for each worker thread, and each coroutine issues requests to per-rank batchers and tries to acquire the combiner latch. After ② acquiring the latch, the coroutine ③ executes a part of the combiner job instead of executing them all. As depicted in the three-state finite state machine in Figure 6b, after collecting the requests and starting the PIM program, the coroutine ④ releases the latch and ⑤ yields the control to another coroutine (co_yield). Hence, the worker thread can utilize the CPU with other coroutines while the PIM program is running, maximizing the throughput (Challenge C4). The later parts of the combiner job (e.g., distributing the results) are executed by other coroutines or the same coroutine when it is scheduled later.

Using multiple coroutines improves throughput by interleaving CPU work with PIM. However, as noted in §3.2, too large batching (of coroutines) degrades transaction latency and increases the total transaction context size. We evaluate this trade-off in §7.3.

5.4 NUMA-Aware Workload Partitioning

The flat combining algorithm in Figure 6 performs well on single-socket servers, but its performance is suboptimal on multi-socket machines. The algorithm assumes that any worker thread can be the combiner for any rank, and the combiner thread accesses the structures of the per-rank batcher (the request list and the combiner latch). Hence, if threads from different NUMA nodes become the

combiners of the same rank one after the other, the structures of the batcher are moved to the other socket's LLC over the cross-socket interconnect. Such ping-ponging can be as frequent as the rate of PIM requests, significantly degrading performance.

OLTPim avoids this problem by partitioning the tables for each NUMA node and allowing each worker thread to query only its local partitions [60]. By using NUMA-partitioned tables, the DBMS prevents a worker thread from becoming the combiner of NUMA-remote ranks, minimizing inter-socket data movement.

6 IMPLEMENTATION CHALLENGES

We implemented OLTPim targeting UPMEM hardware [66], the most commonly used commercial PIM system. We found, however, that the official UPMEM SDK was suboptimal for OLTP workloads. As such, we extended the SDK to improve its API and implement additional features [44]. This section lists the challenges and our solutions for implementing OLTPim using the UPMEM SDK v2025.1.0.

6.1 Reducing API Overheads

The CPU interacts with UPMEM modules via `LOAD`/`STORE` instructions [66]. Both the local memory and control interface are mapped to predefined addresses in the virtual memory on the CPU, with rank-wise interleaving. A CPU-resident thread should be able to interact with each rank using the `LOAD`/`STORE` instructions without system calls or helper threads. However, the UPMEM SDK offloads the work to helper threads for parallelism. This benefits the copying of a large amount of data in batch-oriented applications, but it incurs unnecessary OS overhead on latency-sensitive applications.

Instead of offloading the copy operations to helper threads, our extension has the calling thread copy the data to/from the mapped addresses using SIMD instructions. We apply this optimization to both the data movement and the control interface, thereby removing OS overheads and enabling the flat-combining optimization (§5.2). We also inline our extension into a single executable to avoid external library calls and allow compiler optimizations.

Another problem we faced is that the UPMEM SDK uses a *busy-loop polling* protocol to check the completion of both PIM programs and mux switches. Each status poll uses memory-mapped I/O, and the protocol requires multiple rounds of read/write commands. Because the CPU repeatedly polls the PIM status over the memory channel, unnecessary memory channel traffic grows linearly with PIM program running time. Furthermore, the protocol checks the activeness of each module every time it accesses the rank, adding redundant complexity and latency since this status does not change during runtime. After exploring several mitigations, including decreasing the polling frequency, we implemented an optimized API that removes redundant steps in the I/O protocol. Because this API is in the critical path of small-batch applications like OLTPim, these optimizations improve end-to-end transaction latency (see §7.6).

6.2 PIM Programming

PIM modules have limited resources and abilities, making programming systems for them difficult (Challenge C2). UPMEM's SDK provides useful libraries for PIM programs [66], but as we now describe, it lacks functionality necessary for OLTPim (see §4).

The first problem is that the SDK does not support dynamic memory allocation on PIM modules. To implement variable-sized structures (e.g., B+trees), OLTPim uses a static array for them with a predefined number of fixed-sized blocks. The system manually adjusts the capacity of each array (e.g., for tree nodes, versions, and list nodes) to store the maximum amount of data in PIM.

Second, the PIM core supports atomic instructions on only a designated 256-bit memory region, not on arbitrary addresses. This makes it difficult to implement in-place, fine-grained latches on B+tree nodes. Instead, OLTPim uses per-PIM-module bitmaps of (software) latch status, protected by one of the hardware latches.

As mentioned in prior work [25, 42], the UPMEM PIM core has limited instruction memory (4K instructions). Thus, we extract commonly used code into functions and un-inline them to fit all PIM engine codes within the instruction memory limit.

Lastly, each PIM core has a local scratchpad (SPM) but without hardware support for transparent caching (§2.1). OLTPim statically partitions the SPM and manually specifies the addresses and sizes of data to temporarily fetch them into the SPM. There is too little instruction memory to implement complicated software caching. As a result, our implementation (possibly redundantly) fetches PIM-side data into the SPM every time they are accessed.

7 EVALUATION

We evaluate OLTPim to analyze the performance and efficiency improvement of PIM under different OLTP workloads.

7.1 Experiment Setup

We evaluate OLTPim on a dual-socket server with 2 × Intel Xeon Silver 4216 CPUs (32 threads, 2.1 GHz, 22 MB LLC). It has 12 memory channels where eight are equipped with UPMEM DIMMs and the other four are DRAM DIMMs, which are all DDR4 2400MT/s. The server has a total 32 UPMEM ranks or 2048 modules with 128 GB capacity and PIM cores at 350 MHz. Since UPMEM DIMMs take up 2/3 of the memory channels and aggregate bandwidth, evaluating the baseline system on the same server degrades the performance by 5–10%. Instead, we compare the baseline system on the same server when all 12 channels are equipped with DRAM DIMMs (DDR4 2400MT/s). We evaluate both systems in a Docker container with Ubuntu v22.04 (kernel v5.10.0) and UPMEM SDK v2025.1.0.

Baseline In-Memory DBMS. We compare the performance and efficiency of OLTPim with MosaicDB [35], a state-of-the-art research OLTP DBMS with memory-optimized data structures and latency hiding techniques. It inherits the source code of Silo [65], ERMIA [48], and CoroBase [29]. Although MosaicDB supports on-disk queries, we disable them for a fair comparison with OLTPim.

Workloads. We primarily use YCSB [12] to evaluate the systems under various scenarios. We use the following workload mixes:

- `YCSB-C` is a read-only workload. Each transaction reads 10 tuples.
- `YCSB-B` is a read-intensive workload (95% reads, 5% updates). Each update transaction updates 10 tuples.
- `YCSB-A` is an update-intensive workload (50% updates, 50% reads).

Additionally, we use TPC-C [13] (1024 warehouses) to represent complex OLTP workloads. In addition to the original TPC-C, we use its variants with different ratios of write transactions (new order, payment, and delivery) and read-only transactions (order status

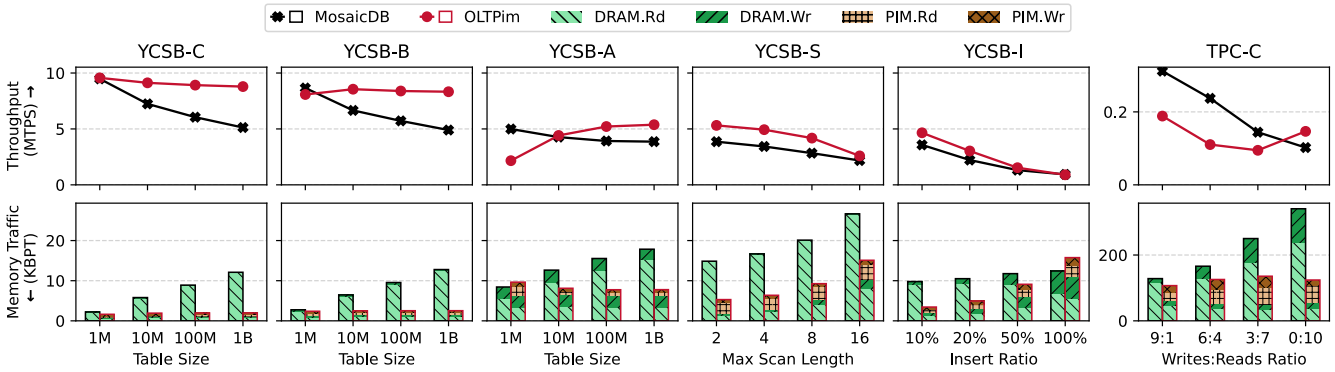


Figure 7: Performance Overview. Transaction throughput and memory channel traffic of OLTPim and MosaicDB on YCSB and TPC-C workloads. In the memory traffic plots, the left bars are from MosaicDB, and the right ones are from OLTPim. Each bar is broken down into DRAM read/write and PIM read/write traffic from bottom to top using different colors. *In this and later figures, the arrow next to each y-axis legend points to the better direction for the metric.*

and stock level). The original benchmark uses the write-read ratio of 9:1, and the ratio of 0:10 is the read-only TPC-CR [29].

To further increase the batch size on OLTPim, we modify the workloads to issue as many queries in each transaction simultaneously as possible, similar to the *MultiGet* [40] and JDBC's `executeBatch` [22] interfaces. The benchmark driver simultaneously issues 10 independent queries in each YCSB transaction so the CPU engine processes them in one batch. Similarly, each transaction in TPC-C is optimized to use the minimum number of PIM rounds by computing the dependency graph of the queries in each transaction and issuing all queries at the same level simultaneously.

Measurement. We use two evaluation metrics: (i) *Transaction throughput* is the rate of committed transactions, excluding aborts; its unit is million transactions per second (MTPS), and (ii) *Per-transaction memory channel traffic*, or *memory traffic*, is the amount of memory channel traffic (for both commits and aborts) amortized over the number of commits; its unit is kilobytes per transaction (KBPT) and we collect it via hardware performance counters, separating DRAM and PIM traffic using per-channel values. All reported results are for 60-second runs of each system, except for TPC-C and YCSB-A with table size 1B where we use 30-second runs.

DBMS Configurations. MosaicDB uses multiple coroutines per thread to hide DRAM latencies. OLTPim also uses coroutines to interleave CPU/PIM execution and hide the PIM latencies (§5.3). Since PIM's latency is higher than DRAM's, we configure OLTPim to use coroutines with a larger batch size to achieve maximum throughput. MosaicDB's optimal batch is eight coroutines per thread. OLTPim shows different maxima for each workload. On YCSB, OLTPim uses 256 coroutines per thread, except for YCSB-S, which uses 128. On TPC-C, it uses 64 coroutines per thread. We measure the impact of batch size in §7.3. Both DBMSs write the recovery logs to a RAM disk. Since its latency is negligible compared to SSD, the DBMSs asynchronously flush log buffers without pipelined commits [39].

7.2 Overall Performance

We first compare the transaction throughput and memory channel traffic of OLTPim and MosaicDB on different workloads.

Read-Only Workload. We use a read-only workload to evaluate the read performance of OLTPim and its dependency on the table size. The YCSB-C column in Figure 7 shows that OLTPim achieves up to 1.71× throughput with 6.14× less memory traffic over MosaicDB (as highlighted in Figure 1). MosaicDB's throughput and efficiency degrade on large tables because the CPU cache cannot store the entire working set as it grows, which increases the number of LLC misses. In contrast, OLTPim's throughput and memory traffic remain constant on larger tables since it avoids far-memory traffic by exploiting near-memory accesses.

Workloads with Updates. We add update transactions to the workload to evaluate the update performance of OLTPim on different table sizes. The results for YCSB-B and YCSB-A show OLTPim achieves up to 1.70× higher throughput with 5.05× less memory traffic over MosaicDB. Compared to YCSB-C, DRAM write traffic increases on both systems. However, the trend of total memory traffic over the increasing table sizes remains similar; it increases for larger tables in MosaicDB but remains constant in OLTPim. Although OLTPim performs well on large tables, its throughput decreases on small tables and underperforms MosaicDB on tables with less than 10M tuples. This slowdown is due to the GC of each DBMS, which incurs significant overhead if used with OLTPim's larger batch sizes. We further discuss this inefficiency in §7.4.

Workload with Scans. To evaluate the range scan performance with different scan lengths, we create a custom workload in which each transaction issues 10 scan queries. As shown in the YCSB-S column, OLTPim shows higher throughput than MosaicDB using less memory traffic on all scan lengths. Since the tuples are stored in DRAM, both DBMSs' DRAM read memory traffic increases by the same amount as the scan length becomes large.

Workload with Inserts. To evaluate the insert performance of OLTPim, we create another workload YCSB-I with varying ratios of inserts and reads. It starts with 100M tuples, and each insert transaction inserts 10 tuples. The final table size is 300M–800M tuples, depending on the insert ratio. The YCSB-I results show that OLTPim outperforms MosaicDB on workloads with less than 50% insertions. We will analyze the reason behind the inefficiency of OLTPim on insertion-intensive workloads in §7.5.

TPC-C Benchmark. Lastly, we use the TPC-C benchmark to evaluate the DBMSs on more complex transaction workloads, with varying write-reads ratios. As described in §4.1, OLTPim places

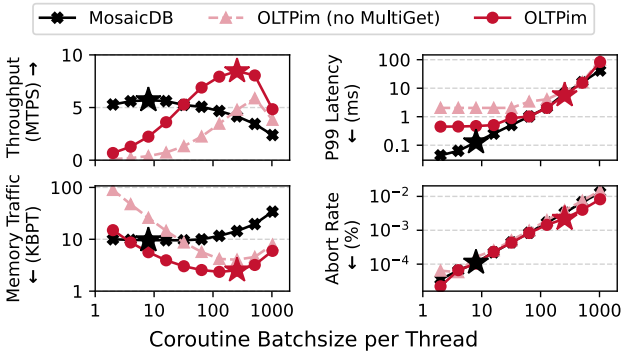


Figure 8: Impact of Batch Size. Evaluation of varying batch sizes on transaction throughput, P99 latency, memory channel traffic, and abort rate for YCSB-B with 100M tuples. The star points denote the throughput-optimal batch size for each system.

small indexes (i.e., warehouse, district) in DRAM and others in PIM. OLTPim reduces memory traffic per transaction on all write-read mixes, up to 2.75× on the read-only mix. TPC-C's write transactions are insert-heavy, highly-skewed, and exhibits good data locality [29]. On higher read ratios, MosaicDB's throughput decreases because the overall workload's data locality decreases with the write ratio, increasing LLC misses. However, OLTPim's throughput decrease is smaller than MosaicDB as PIM enables efficient processing of workloads without locality. Furthermore, OLTPim outperforms MosaicDB on the read-only workload, showing that the read-only transactions are more suitable for OLTPim than the write transactions.

7.3 Impact of Batch Size and P99 Latency

OLTPim requires a large batch size to hide PIM round latency and improve throughput, but it incurs a larger transaction latency, a higher conflict rate, and a larger working set for transaction contexts. To measure this tradeoff, we evaluate YCSB-B with 100M tuples for varying batch sizes and with/without the *MultiGet* optimization (§7.1). Figure 8 shows the throughput, P99 latency, memory traffic, and abort rate depending on the batch size. The stars denote the throughput-optimal batch size for each system.

OLTPim achieves its best throughput with batch size 256, whereas MosaicDB's best size is eight. On small batch sizes, OLTPim's throughput is less than MosaicDB due to the large PIM round latency for each of many batches; larger batch sizes mitigate this overhead. Performance decreases for batch sizes larger than 1K because the total context size exceeds the CPU LLC capacity. Additionally, issuing multiple queries in one batch improves throughput since the no-*MultiGet*'s peak throughput is smaller than OLTPim.

The P99 measurements in Figure 8 show that OLTPim's latency is higher than MosaicDB with small batch sizes due to the large PIM round latency. The impact is higher for the no-*MultiGet* instance as it requires 10 batches or 20 mux switches to complete a transaction. However, large batch sizes hide PIM's impact as the DBMSs' P99 converges to similar values. Although OLTPim's P99 with its throughput-optimal large batch size is higher than MosaicDB's with its small batch size, OLTPim still achieves a <10 ms P99 latency (star point), which is acceptable in most applications.

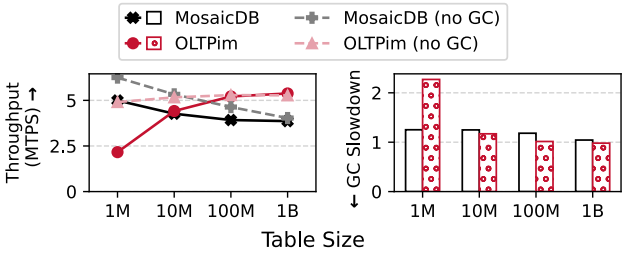


Figure 9: Impact of Garbage Collection. Comparison of transaction throughput for YCSB-A. The GC slowdown is the ratio of the throughput without GC over the throughput with GC.

For batch sizes smaller than 100, OLTPim's per-transaction memory channel traffic decreases with larger batch sizes while MosaicDB's traffic remains constant. This is because a portion of memory channel traffic unrelated to each transaction (e.g., PIM control signals) is amortized over the increasing throughput of OLTPim but not over MosaicDB's constant throughput. This impact is larger for the no-*MultiGet* OLTPim with a smaller throughput. On the other hand, memory channel traffic of both systems increases for batch sizes larger than 100 since the total context size exceeds the CPU LLC capacity, as mentioned above.

The abort rate increases with the batch size due to the higher chance of conflicts between more in-flight coroutines. Thus, OLTPim has a higher abort rate under the throughput-optimal batch size than MosaicDB. Although its value is negligible on the read-mostly YCSB-B, the difference can be large on update-heavy workloads. For example, in YCSB-A with 1M tuples, the abort rate of MosaicDB and OLTPim are 1.1% and 21%, respectively. We report the *commit* throughput and the memory traffic *per commit* (§7.1), and hence account for OLTPim's higher abort rate in the reported numbers.

7.4 Updates and Garbage Collection

Since update operations extend version chains that the GC has to then reclaim, we next evaluate YCSB-A without GC to analyze its impact on OLTPim. As shown in Figure 9, enabling GC degrades throughput for both OLTPim and MosaicDB, especially on small tables. This is because applying the same rate of random updates on a smaller table is more likely to create a longer version chain.

After removing the impact of GC, the trend of dotted lines in Figure 9 is the same as the trend of the (read-only) YCSB-C column in Figure 7. MosaicDB's throughput decreases on large tables due to the larger working set and LLC misses, whereas OLTPim's throughput remains constant by converting them to near-memory accesses. Thus, the advantage of using OLTPim on update-intensive workloads without GC is as effective as on read-only workloads.

We also observe that the slowdown due to GC on OLTPim is smaller on large tables. It shows the advantage of OLTPim on GC as described in §4.3. However, OLTPim shows a larger slowdown on the 1M case. The larger batch size of OLTPim causes more in-flight transactions, increasing the length of the non-obsolete version chains (V_5 to V_3 in Figure 5). Because the PIM engine traverses them redundantly at a high rate, it incurs significant overhead even as near-memory accesses within the PIM module.

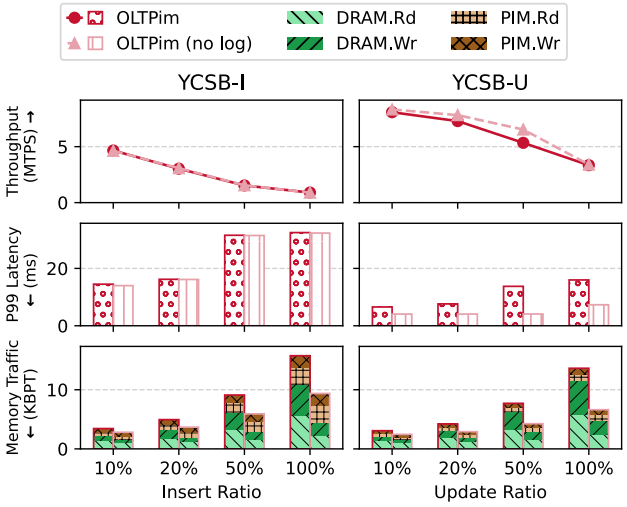


Figure 10: Impact of Logging on Insert/Update. Transaction throughput, P99 latency, and memory channel traffic of OLTPim on YCSB-I and YCSB-U. Unlike the left bars, the right bars are measured without logging.

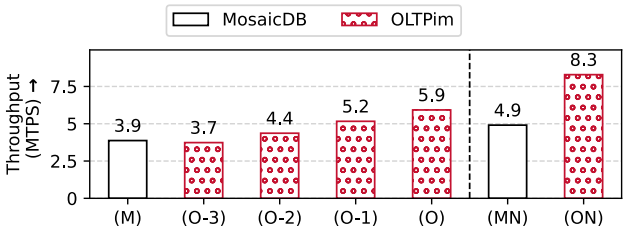


Figure 11: Impact of Optimizations. Transaction throughput of YCSB-B with 1B tuples, incrementally applying different optimizations in Table 2.

7.5 Impact of Logging on Insert Performance

We next evaluate YCSB-I with and without logging to measure its impact on performance. We also create an additional workload YCSB-U that replaces the insert queries in YCSB-I with update queries. By comparing the two workloads without logging, we analyze the impact of the insert operations themselves.

Figure 10 shows the throughput, P99 latency, and memory traffic of YCSB-I and -U, with/without logging. When logging is disabled, the throughput increases while the P99 decreases. Unlike YCSB-U, the changes are negligible in YCSB-I, indicating that logging is not the bottleneck for inserts. Furthermore, insert transactions have larger latency than update transactions, even though they have the same number of queries with similar CPU code complexity. Hence, the latency difference is due to the longer execution time of insert operations in the PIM engine compared to updates, caused by the need for coarse-grained latches in B+tree inserts (§6.2).

Comparing the PIM traffic (the two upper boxes of each memory traffic bar) between YCSB-I and YCSB-U, inserts require more PIM traffic than updates. UPMEM hardware requires the CPU to polling-wait for PIM completion (§6.1). Although our optimized API minimizes the memory traffic for each poll, longer PIM execution consumes more PIM traffic. Hence, the larger PIM traffic also indicates that inserts takes longer than updates on the PIM engine.

Table 2: Description of each optimization level in Figure 11.

Level	Description
(M)	MosaicDB
(O-3)	Offload only the index to PIM with default SDK
(O-2)	Offload both the index and MVCC to PIM (§4.3)
(O-1)	(O-2) + Optimized PIM access (§6.1)
(O)	OLTPim: (O-1) + Interleave CPU & PIM (§5.3)
(MN)	(M) with NUMA-partitioned workload
(ON)	(O) with NUMA-partitioned workload (§5.4)

7.6 Impact of Optimizations

To evaluate the impact of our design choices, we measure the throughput of YCSB-B by selectively enabling our optimizations. For each optimization level described in Table 2, its throughput is shown in Figure 11. We incrementally add optimizations from left to right in the figure. Starting from (M) MosaicDB, (O-3) is the naive version of offloading only the index to PIM. The throughput decreases because using PIM incurs significant batching overhead. (O-2) further offloads the version chains to PIM, following the discussion in §4.3, resulting in higher throughput.

The later optimization levels minimize the batching overhead. (O-1) uses the optimized interface for accessing PIM, as discussed in §6.1. (O) uses coroutines to interleave CPU and PIM executions, as discussed in §5.3. After applying these optimizations, the system becomes OLTPim, which has higher throughput than MosaicDB.

The (MN) and (ON) execute the NUMA-partitioned workload to minimize inter-socket communication, as described in §5.4. Comparing them to (M) and (O), the speedup of using the NUMA-partitioned workload for OLTPim is 1.41×, whereas the speedup for MosaicDB is 1.26×. Because OLTPim's batcher uses more inter-socket communication than MosaicDB for the non-NUMA-partitioned workload, OLTPim gets greater benefit from NUMA partitioning.

7.7 Impact of Skewness

Existing in-memory DBMSs already handle skewed workloads well because the CPU LLC can exploit their data locality. In contrast, the design of OLTPim assumes uniform workloads where the working set does not fit in the CPU cache. Figure 12 shows throughput and memory traffic of OLTPim and MosaicDB on YCSB-B under the zipfian distribution. $\theta = 0$ refers to the uniform distribution, and higher θ means more skewed. The result shows that OLTPim has higher throughput and lower memory traffic than MosaicDB on low-to-moderate skew ($\theta \leq 0.5$), but not on high skew ($\theta = 0.99$).

We also observe that the two systems have different trends regarding the skewness of the workload. The throughput and memory traffic of MosaicDB are improved on more skewed workloads because the CPU LLC is more friendly to them. Conversely, in OLTPim, these metrics are improved on more uniform workloads because the requests are evenly distributed across the PIM cores, exploiting their massive parallelism. Hence, each system has an advantage in different workload characteristics due to the difference in their hardware properties. Future work will explore using ideas from PIM-tree [42] to increase skew-resistance, while seeking to minimize the number of PIM rounds.

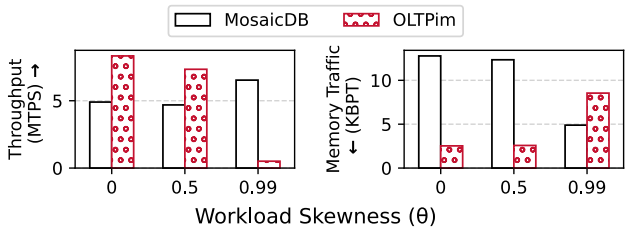


Figure 12: Impact of Skewness. Transaction throughput and memory channel traffic of YCSB-B with 1B tuples under different workload skew. Higher θ means a more skewed workload.

8 RELATED WORK

OLTPim is the first end-to-end OLTP DBMS designed for PIM systems. However, it is closely related to prior work on designing database systems for heterogeneous hardware.

PIM-Optimized Index Structures. Database indexes are the primary components of an OLTP DBMS. PIM-optimized data structures include range partitioning indexes [10, 55] and skew-resistant indexes [42, 43], as discussed in §2.2. In addition, HybriDS [9] improves the range-partitioning indexes by storing the frequently-used upper part of the B+tree in the CPU cache. As discussed in §4.1, OLTPim uses a variant of the range-partitioning index instead of the skew-resistant index to minimize the batching overhead, which is critical to latency-sensitive applications.

PIM-Based OLAP DBMSs. §2.2 also mentioned several works on PIM implementations of OLAP operators. Kepe et al. [46] replaces CPU SIMD instructions with PIM and compares their TPC-H performance. PimDB [3] implements PIM-based filters and aggregates, and Lim et al. [54] designs a join algorithm exploiting PIM's interleaved addresses. Those works target the OLAP operators, which are read-only and batch-friendly. In contrast, OLTPim targets the read-write and latency-sensitive OLTP workloads. Note that typical OLAP systems store most of their data on disk, whereas practical OLTP workloads have working sets that fit in memory, making PIM more suitable for OLTP.

GPU-Based OLTP DBMSs. Due to the powerful batch processing capability of GPUs, several OLAP DBMSs use GPUs to accelerate analytic workloads [28, 30, 32]. On the other hand, GPUTx [27], GaccO [4], and LTPG [68] use GPUs for accelerating OLTP workloads. They group the concurrent transactions into a single batch and resolve their conflicts in advance using the dependency graph. Although GPUs are computationally stronger than PIM cores, PIM consumes significantly less energy and manufacturing cost. Furthermore, PIM is easily scalable to larger memory capacity, making it more suitable for large-scale OLTP workloads.

DBMSs for Other NDP Hardwares. PIM is an example of *near-data processing* (NDP) [58] that pushes code to data instead of fetching data to code. Prior work on on-disk DBMSs exploits NDP-capable SSDs to offload atomic page operations [45], individual OLAP operators [17, 38], the entire SQL engine [59], or a hybrid transactional-analytical engine [67]. For disaggregated memory DBMSs, Farview [49] exploits the NDP in the memory node to improve OLAP performance. OLTPim exploits the NDP concept on the DRAM level within the memory hierarchy.

9 FUTURE PIM HARDWARE

We now discuss how OLTPim's design choices could change with future PIM hardware.

Foremost is OLTPim's data placement scheme in §4, which assumes that each PIM core can only access its local memory. But if PIM hardware supported inter-module communication [75], the DBMS could employ more dynamic run-time operations and placement strategies. For example, secondary index traversal (Figure 4, §4.4) could employ direct PIM-to-PIM communication, avoiding the extra PIM round OLTPim incurs. This capability would also enable the DBMS to perform online re-partitioning (§4.1) without incurring off-chip memory traffic. However, even with enhanced PIM hardware, accessing local memory will still be cheaper than accessing remote memory. As such, OLTPim's fine-grained partitioning scheme will likely remain the best choice for PIM systems.

Second, future PIM hardware could allow independent access to PIM modules. UPMEM's current rank-wise address interleaving prevents the DBMS from controlling individual modules, making PIM-side tuple storage less practical (§4.2) and requiring control of at least rank granularity (§5.1). This interleaving arises because each PIM module resides in a physical chip, while the CPU performs parallel access across multiple chips and banks [15]. But independent module access would alleviate these limitations and allow more fine-grained concurrency.

Third, OLTPim's batcher (§5) assumes that the mux switch latency is larger than memory access latency. Future PIM hardware could optimize mux switches or support asynchronous PIM access [6]. These improvements could enable the batcher for smaller batch sizes. However, the CPU-PIM interleaving and the NUMA partitioning in §5.3 and §5.4 are still effective for hiding PIM execution latency and reducing inter-socket accesses, respectively. Additionally, even with asynchronous PIM, a DBMS will still require rank-wise batching (§5.1) to pack the requests into an interleaved array, unless the CPU can send data to each module independently.

10 CONCLUSION

This paper presents OLTPim, the first end-to-end OLTP DBMS designed for a real PIM system. The evaluation shows that it achieves up to 1.71× throughput speedup with up to 6.14× less per-transaction memory channel traffic than the state-of-the-art in-memory DBMS. Its design includes the formalized partitioning strategy of database components on different types of memory and a lightweight batching algorithm, which are likely useful for designing other PIM-optimized latency-sensitive applications.

ACKNOWLEDGMENTS

We thank Patrice Lacouture and others from UPMEM for providing extensive technical support. We also thank the reviewers of PVLDB for their invaluable feedback. This work was supported by NSF grants CCF-1919223 and CNS-2211882, and the Parallel Data Lab (PDL) Consortium (Amazon, Bloomberg, Datadog, Google, Honda, Intel, Jane Street, LayerZero, Meta, Microsoft, Oracle, Pure Storage, Salesforce, Samsung, and Western Digital). Hyoungjoo is supported by the Korea Foundation for Advanced Studies and a Northrop Grumman Fellowship. Yiwei's support includes a Lee-Stanziale Ohana Fellowship and a Michel and Kathy Doreau Fellowship.

REFERENCES

[1] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*. 105–117.

[2] Alexander Baumstark, Muhammad Attahir Jibril, and Kai-Uwe Sattler. 2023. Adaptive Query Compilation with Processing-in-Memory. In *2023 IEEE 39th International Conference on Data Engineering Workshops (ICDEW)*. 191–197.

[3] Arthur Bernhardt, Andreas Koch, and Ilia Petrov. 2023. pimDB: From main-memory DBMS to processing-in-memory DBMS-engines on intelligent memories. In *Proceedings of the 19th International Workshop on Data Management on New Hardware (DaMoN)*. 44–52.

[4] Nils Boeschen and Carsten Binnig. 2022. Gacco - A GPU-Accelerated OLTP DBMS. In *Proceedings of the 2022 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 1003–1016.

[5] Pedro Carrinho, Oscar Ferraz, João Dinis Ferreira, Yann Falevoz, Vitor Silva, and Gabriel Falcao. 2024. Processing Multi-Layer Perceptrons In-Memory. In *2024 IEEE Workshop on Signal Processing Systems (SiPS)*. 7–12.

[6] Liyan Chen, Dongxu Lyu, Jianfei Jiang, Qin Wang, Zhigang Mao, and Naifeng Jing. 2025. AsyncDIMM: Achieving Asynchronous Execution in DIMM-Based Near-Memory Processing. In *Proceedings of the 2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 518–532.

[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 *Proceedings of the 60th ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[8] Sitian Chen, Haobin Tan, Amelie Chi Zhou, Yusen Li, and Pavan Balaji. 2024. UpDLRM: Accelerating Personalized Recommendation using Real-World PIM Architecture. In *Proceedings of the 61st ACM/IEEE Design Automation Conference (DAC)*. 211:1–211:6.

[9] Jiwon Choe, Andrew Crotty, Tali Moreshet, Maurice Herlihy, and R Iris Bahar. 2022. Hybrids: Cache-conscious concurrent data structures for near-memory processing architectures. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 321–332.

[10] Jiwon Choe, Amy Huang, Tali Moreshet, Maurice Herlihy, and R Iris Bahar. 2019. Concurrent data structures with near-data-processing: An architecture-aware implementation. In *Proceedings of the 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 297–308.

[11] Russell Clapp, Martin Dimitrov, Karthik Kumar, Vish Viswanathan, and Thomas Willhalm. 2015. Quantifying the performance impact of memory latency and bandwidth for big data workloads. In *Proceedings of the 2015 IEEE International Symposium on Workload Characterization (IISWC)*. 213–224.

[12] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SOCC)*. 143–154.

[13] Transaction Processing Performance Council. 2010. TPC Benchmark C. https://www.tpc.org/TPC_Documents_Current_Versions/pdf/tpc-c_v5.11.0.pdf. Accessed on 2025.07.13.

[14] Guohao Dai, Tianhao Huang, Yuze Chi, Jishen Zhao, Guangyu Sun, Yongpan Liu, Yu Wang, Yuan Xie, and Huazhong Yang. 2018. GraphH: A processing-in-memory architecture for large-scale graph processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 4 (2018), 640–653.

[15] Alexandar Devic, Siddhartha Balakrishna Rai, Anand Sivasubramaniam, Ameen Akel, Sean Eilert, and Justin Eno. 2022. To PIM or not for emerging general purpose processing in DDR memory systems. In *Proceedings of the 49th International Symposium on Computer Architecture (ISCA)*. 231–244.

[16] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 1243–1254.

[17] Jaeyoung Do, Yang-Suk Kee, Jignesh M Patel, Chanik Park, Kwanghyun Park, and David J DeWitt. 2013. Query processing on smart SSDs: Opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 1221–1230.

[18] Franz Faerber, Alfons Kemper, Per-Åke Larson, Justin Levandoski, Thomas Neumann, and Andrew Pavlo. 2017. Main Memory Database Systems. *Foundations and Trends® in Databases* 8, 1-2 (2017), 1–130.

[19] Dina Fakhry, Mohamed Abdelsalam, M Watheq El-Kharashi, and Mona Safar. 2023. An HBM3 Processing-In-Memory Architecture for Security and Data Integrity: Case Study. In *Green Sustainability: Towards Innovative Digital Transformation*, Dalia Magdi, Ahmed Abou El-Fetouh, Mohamed Mamdouh, and Amit Joshi (Eds.). 281–293.

[20] Oscar Ferraz, Gabriel Falcao, and Vitor Silva. 2024. In-Memory Bit Flipping LDPC Decoding. In *2024 32nd European Signal Processing Conference (EUSIPCO)*. 706–710.

[21] Oscar Ferraz, Yann Falevoz, Vitor Silva, and Gabriel Falcao. 2023. Unlocking the Potential of LDPC Decoders with PiM Acceleration. In *2023 57th Asilomar Conference on Signals, Systems, and Computers*. 1579–1583.

[22] Maydene Fisher, Jonathan Ellis, Jon Ellis, and Jonathan Bruce. 2003. *JDBC API tutorial and reference*. Addison-Wesley Professional.

[23] Sahar Ghoflsaz Ghinani, Jingyao Zhang, and Elaheh Sadredini. 2025. Enabling Low-Cost Secure Computing on Untrusted In-Memory Architectures. arXiv:2501.17292 [cs.CR] https://arxiv.org/abs/2501.17292

[24] Christina Giannoula, Ivan Fernandez, Juan Gómez-Luna, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. 2022. SparseP: Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Architectures. In *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 288–291.

[25] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F Oliveira, and Onur Mutlu. 2022. Benchmarking a new paradigm: Experimental analysis and characterization of a real processing-in-memory system. *IEEE Access* 10 (2022), 52565–52608.

[26] Harshita Gupta, Mayank Kabra, Juan Gómez-Luna, Konstantinos Kanellopoulos, and Onur Mutlu. 2023. Evaluating Homomorphic Operations on a Real-World Processing-In-Memory System. In *Proceedings of the 2023 IEEE International Symposium on Workload Characterization (IISWC)*. 211–215.

[27] Bingsheng He and Jeffrey Xu Yu. 2011. High-throughput transaction executions on graphics processors. *Proceedings of the VLDB Endowment* 4, 5 (2011), 314–325.

[28] Dong He, Supun C Nakandala, Dalitso Banda, Rathijit Sen, Karla Saur, Kwanghyun Park, Carlo Curino, Jesús Camacho-Rodríguez, Konstantinos Karanasos, and Matteo Interlandi. 2022. Query processing on tensor computation runtimes. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2811–2825.

[29] Yongjun He, Jiacheng Lu, and Tianzheng Wang. 2020. CoroBase: coroutine-oriented main-memory database engine. *Proceedings of the VLDB Endowment* 14, 3 (2020), 431–444.

[30] heavy.ai. 2025. HeavyDB (formerly OmniSciDB). https://github.com/heavyai/heavydb. Accessed on 2025.07.13.

[31] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 355–364.

[32] heterodb. 2025. PG-Strom Manual. https://heterodb.github.io/pg-strom/. Accessed on 2025.07.13.

[33] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W Keckler. 2016. Transparent offloading and mapping (TOM) enabling programmer-transparent near-data processing in GPU systems. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 204–216.

[34] Kevin Hsieh, Samira Khan, Nandita Vijaykumar, Kevin K Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu. 2016. Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation. In *Proceedings of the 2016 IEEE 34th International Conference on Computer Design (ICCD)*. 25–32.

[35] Kaisong Huang, Tianzheng Wang, Qingqing Zhou, and Qingzhong Meng. 2023. The Art of Latency Hiding in Modern Database Engines. *Proceedings of the VLDB Endowment* 17, 3 (2023), 577–590.

[36] Bongjoon Hyun, Taehun Kim, Dongjae Lee, and Minsoo Rhu. 2024. Pathfinding Future PIM Architectures by Demystifying a Commercial PIM Technology. In *Proceedings of the 2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 263–279.

[37] Muhammad Attahir Jibril, Hani Al-Sayeh, and Kai-Uwe Sattler. 2024. Accelerating Aggregation Using a Real Processing-in-Memory System. In *Proceedings of the 2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 3920–3932.

[38] Insoon Jo, Duck-Ho Bae, Andre S Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel DG Lee, and Jaeheon Jeong. 2016. YourSQL: a high-performance database system leveraging in-storage computing. *Proceedings of the VLDB Endowment* 9, 12 (2016), 924–935.

[39] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. 2010. Aether: a scalable approach to logging. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 681–692.

[40] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin Levandoski, and Gor Nishanov. 2018. Exploiting coroutines to attack the "killer nanoseconds". *Proceedings of the VLDB Endowment* 11, 11 (2018), 1702–1714.

[41] Hongbo Kang, Phillip B Gibbons, Guy E Blelloch, Laxman Dhulipala, Yan Gu, and Charles McGuffey. 2021. The processing-in-memory model. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 295–306.

[42] Hongbo Kang, Yiwei Zhao, Guy E Blelloch, Laxman Dhulipala, Yan Gu, Charles McGuffey, and Phillip B Gibbons. 2022. PIM-Tree: A Skew-Resistant Index for Processing-in-Memory. *Proceedings of the VLDB Endowment* 16, 4 (2022).

[43] Hongbo Kang, Yiwei Zhao, Guy E Blelloch, Laxman Dhulipala, Yan Gu, Charles McGuffey, and Phillip B Gibbons. 2023. PIM-trie: A Skew-resistant Trie for Processing-in-Memory. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 1–14.

[44] Hongbo Kang, Yiwei Zhao, and Hyoungjoo Kim. 2024. upmem-sdk-light. https://github.com/Loremkang/upmem-sdk-light. Accessed on 2025.07.13.

[45] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Gi-Hwan Oh, and Changwoo Min. 2013. X-FTL: transactional FTL for SQLite databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 97–108.

[46] Tiago R Kepe, Eduardo C de Almeida, and Marco AZ Alves. 2019. Database processing-in-memory: An experimental study. *Proceedings of the VLDB Endowment* 13, 3 (2019), 334–347.

[47] Jin Hyun Kim, Shin-Haeng Kang, Sukhan Lee, Hyeonsu Kim, Yuhwan Ro, Seungwon Lee, David Wang, Jihyun Choi, Jinin So, YeonGon Cho, et al. 2022. Aquabolt-XL HBM2-PIM, LPDDR5-PIM with in-memory processing, and AXDIMM with acceleration buffer. *IEEE Micro* 42, 3 (2022), 20–30.

[48] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. Ermia: Fast memory-optimized database system for heterogeneous workloads. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 1675–1687.

[49] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan S Milojicic, and Gustavo Alonso. 2022. Farview: Disaggregated Memory with Operator Off-loading for Database Engines. In *Proceedings of the 12th Conference on Innovative Data Systems Research (CIDR)*. 14.

[50] Dominique Lavenier, Jean-Francois Roy, and David Furodet. 2016. DNA mapping using Processor-in-Memory architecture. In *Proceedings of the 2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. 1429–1435.

[51] Dongjae Lee, Bongjoon Hyun, Taehun Kim, and Minsoo Rhu. 2024. PIM-MMU: A Memory Management Unit for Accelerating Data Transfers in Commercial PIM Systems. In *Proceedings of the 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 627–642.

[52] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *Proceedings of the 2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 38–49.

[53] 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 *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Volume 2*. 879–896.

[54] Chaemin Lim, Suhyun Lee, Jinwoo Choi, Jounghoo Lee, Seongyeon Park, Hanjun Kim, Jinho Lee, and Youngsok Kim. 2023. Design and Analysis of a Processing-in-DIMM Join Algorithm: A Case Study with UPMEM DIMMs. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–27.

[55] Zhiyu Liu, Irina Calciu, Maurice Herlihy, and Onur Mutlu. 2017. Concurrent data structures for near-memory computing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 235–245.

[56] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys)*. 183–196.

[57] Meven Mognol, Dominique Lavenier, and Julien Legriel. 2024. Parallelization of the Banded Needleman & Wunsch Algorithm on UPMEM PiM Architecture for Long DNA Sequence Alignment. In *Proceedings of the 53rd International Conference on Parallel Processing (ICPP)*. 1062–1071.

[58] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. 2022. A modern primer on processing in memory. In *Emerging computing: from devices to systems: looking beyond Moore and Von Neumann*. Springer, 171–243.

[59] Jong-Hyeok Park, Soyee Choi, Gihwan Oh, and Sang-Won Lee. 2021. SaS: SSD as SQL database system. *Proceedings of the VLDB Endowment* 14, 9 (2021), 1481–1488.

[60] Danica Porobic, Ippokratis Pandis, Miguel Branco, Pınar Tözün, and Anastasia Ailamaki. 2012. OLTP on hardware islands. *Proceedings of the VLDB Endowment* 5, 11 (2012), 1447–1458.

[61] Steve Rhyner, Haocong Luo, Juan Gómez-Luna, Mohammad Sadrosadati, Jiawei Jiang, Ataberk Olgun, Harshita Gupta, Ce Zhang, and Onur Mutlu. 2024. PIM-Opt: Demystifying Distributed Optimization Algorithms on a Real-World Processing-In-Memory System. In *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 201–218.

[62] Shunchen Shi, Xueqi Li, Zhaowu Pan, Peiheng Zhang, and Ninghui Sun. 2024. CoPIM: A Collaborative Scheduling Framework for Commodity Processing-in-memory Systems. In *Proceedings of the 2024 IEEE 42nd International Conference on Computer Design (ICCD)*. 44–51.

[63] Michael Stonebraker, Samuel Madden, Daniel J Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2007. The End of an Architectural Era: (It's Time for a Complete Rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB)*. 1150–1160.

[64] Dufy Teguia, Jiaxuan Chen, Stella Bitchebe, Oana Balmau, and Alain Tchana. 2024. vPIM: Processing-in-Memory Virtualization. In *Proceedings of the 25th ACM International Middleware Conference (Middleware)*. 417–430.

[65] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. 18–32.

[66] UPMEM. 2025. UPMEM Software Development Kit (SDK). https://sdk.upmem.com/. Accessed on 2025.07.13.

[67] Tobias Vinçon, Christian Knödler, Leonardo Solis-Vasquez, Arthur Bernhardt, Sajjad Tamimi, Lukas Weber, Florian Stock, Andreas Koch, and Ilia Petrov. 2022. Near-data processing in database systems on native computational storage under HTAP workloads. *Proceedings of the VLDB Endowment* 15, 10 (2022), 1991–2004.

[68] Jianpeng Wei, Yu Gu, Tianyi Li, Jianzhong Qi, Chuanwen Li, Yanfeng Zhang, Christian S Jensen, and Ge Yu. 2024. LTPG: Large-Batch Transaction Processing on GPUs with Deterministic Concurrency Control. In *Proceedings of the 2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 3865–3877.

[69] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment* 10, 7 (2017), 781–792.

[70] Zhongle Xie, Qingchao Cai, HV Jagadish, Beng Chin Ooi, and Weng-Fai Wong. 2017. Parallelizing skip lists for in-memory multi-core database systems. In *Proceedings of the 2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 119–122.

[71] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L Greathouse, Lifan Xu, and Michael Ignatowski. 2014. TOP-PIM: Throughput-oriented programmable processing in memory. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing (HPDC)*. 85–98.

[72] Yiwei Zhao, Jinhui Chen, Sai Qian Zhang, Syed Shakib Sarwar, Kleber Hugo Stangherlin, Jorge Tomas Gomez, Jae-Sun Seo, Barbara De Salvo, Chiao Liu, Phillip B Gibbons, and Ziyun Li. 2025. H4H: Hybrid Convolution-Transformer Architecture Search for NPU-CIM Heterogeneous Systems for AR/VR Applications. In *Proceedings of the 30th Asia and South Pacific Design Automation Conference (ASPDAC)*. 1133–1141.

[73] Yiwei Zhao, Hongbo Kang, Yan Gu, Guy E Blelloch, Laxman Dhulipala, Charles McGuffey, and Phillip B Gibbons. 2025. Optimal Batch-Dynamic kd-trees for Processing-in-Memory with Applications. In *Proceedings of the 37th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[74] Yiwei Zhao, Ziyun Li, Win-San Khwa, Xiaoyu Sun, Sai Qian Zhang, Syed Shakib Sarwar, Kleber Hugo Stangherlin, Yi-Lun Lu, Jorge Tomas Gomez, Jae-Sun Seo, Phillip B Gibbons, Barbara De Salvo, and Chiao Liu. 2024. Neural Architecture Search of Hybrid Models for NPU-CIM Heterogeneous AR/VR Devices. *arXiv preprint arXiv:2410.08326* (2024).

[75] Zhe Zhou, Cong Li, Fan Yang, and Guangyu Sun. 2023. DIMM-Link: Enabling Efficient Inter-DIMM Communication for Near-Memory Processing. In *Proceedings of the 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 302–316.