Accelerating Transactional Execution via Processing-In-Memory

André Lopes
andre.f.lopes@tecnico.ulisboa.pt
INESC-ID & Instituto Superior Técnico, Universidade de Lisboa
Lisbon, Portugal

Daniel Castro
daniel.castro@tecnico.ulisboa.pt
INESC-ID & Instituto Superior Técnico, Universidade de Lisboa
Lisbon, Portugal

Paolo Romano
paolo.romano@tecnico.ulisboa.pt
INESC-ID & Instituto Superior Técnico, Universidade de Lisboa
Lisbon, Portugal

Abstract

Processing-in-Memory (PIM) integrates compute capabilities directly into memory modules to reduce costly data movement, promising large performance gains for data-intensive workloads. In PIM systems, small processors—Data Processing Units (DPUs)—are embedded near memory banks to execute computations close to where data resides. Although PIM has shown strong benefits for analytical tasks, supporting transactional workloads such as online transaction processing (OLTP) remains challenging due to decentralized DPUs and the lack of efficient hardware coordination mechanisms.

We present PIM-TIDE (Processing-in-Memory with Transactional Isolation via Deterministic Execution), a new transaction execution platform designed for PIM architectures. PIM-TIDE provides a lightweight software-based coordination mechanism that enables transactions to span multiple DPUs, while ensuring consistency, atomicity, and isolation. By using the CPU selectively for transaction coordination, PIM-TIDE avoids the performance penalties of frequent inter-DPU communication. We evaluate PIM-TIDE on real PIM hardware (UPMEM) and show that it achieves scalable transaction processing with low overhead, both in terms of performance and energy efficiency, in TPC-C based workloads.

CCS Concepts: • **Hardware** → *Emerging architectures*; • **Information systems** → **Database transaction processing**; **Distributed database transactions**.

Keywords: Processing-in-Memory, Transaction processing systems, Concurrency control, Transactional memory

ACM Reference Format:
André Lopes, Daniel Castro, and Paolo Romano. 2026. Accelerating Transactional Execution via Processing-In-Memory. In *21st European Conference on Computer Systems (EUROSYS '26), April 27–30, 2026, Edinburgh, Scotland Uk*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3767295.3803621

EUROSYS '26, Edinburgh, Scotland Uk

ACM ISBN 979-8-4007-2212-7/2026/04
https://doi.org/10.1145/3767295.3803621

1 Introduction

Recently, Processing-in-Memory (PIM) architectures have attracted significant interest as a means to mitigate the growing disparity between processor speed and memory bandwidth [21, 42, 50]. By integrating compute capabilities directly into memory modules, PIM architectures reduce costly data movement and offer substantial performance gains for memory-bound workloads across domains such as machine learning [27, 36, 40, 52], graph analytics [22, 60], scientific computing [10, 28, 41] and data stores [1, 33, 35, 54].

Among commercial PIM solutions, UPMEM distinguishes itself by offering a flexible general-purpose programming model [58]. Its architecture features thousands of lightweight Data Processing Units (DPUs) embedded within DRAM modules. Each DPU supports multithreaded execution (up to 24 hardware threads) and provides high-bandwidth access to local memory. With over 20,000 hardware threads distributed across thousands of DPUs, applications must exhibit massive parallelism to fully harness UPMEM's potential.

A key limitation of UPMEM—and PIM architectures more broadly—is the absence of direct inter-DPU communication. Any exchange between DPUs must be mediated by the host CPU, incurring high latency and serialization costs. As PIM systems scale to thousands of DPUs, this bottleneck becomes increasingly problematic for transactional in-memory data stores, which require concurrent access to distributed data while preserving strong consistency. Existing work has therefore either focused on OLAP workloads [1, 33, 35, 54], which are more amenable to parallelization, or restricted the expressiveness of the transaction abstraction by limiting its scope to a single DPU [39].

This paper addresses this problem by introducing PIM-TIDE (Processing-in-Memory with Transactional Isolation via Deterministic Execution), the first PIM-accelerated in-memory data store to support cross-DPU transactions. To avoid the high cost of CPU-mediated coordination, PIM-TIDE draws inspiration from speculative techniques commonly employed in the literature on distributed transactions [38, 47] and deterministic concurrency control [49, 56]. In order to hide the cost of inter-node communication and eliminate the need for distributed locking, PIM-TIDE precomputes a global transaction order and decomposes each distributed transaction into subtransactions that execute independently on their

local DPUs. By constraining each subtransaction to access only local data and by enforcing a deterministic serialization order, PIM-TIDE avoids contention among distributed transactions and sidesteps complex synchronization, all while providing strong consistency guarantees with minimal inter-DPU communication. This overall design aligns well with the architectural constraints of UPMEM and enables scalable concurrency across thousands of DPUs.

Via an extensive experimental study, we assess PIM-TIDE's scalability, speedup, and energy gains with respect to a CPU-based system. Our results highlight speedups (up to 6.75x) and energy gains (up to 3.52x) in all tested workload configurations. These efficiency gains are also enabled by the usage of scheduling techniques and concurrency control schemes that optimize processing of local vs distributed transactions. The remainder of the paper is organized as follows. Section 2, presents a short overview of PIM, related work on concurrency control, deterministic transactional systems, and PIM-accelerated databases. Section 3 discusses the design and implementation of PIM-TIDE. Section 4 presents the results of our experimental evaluation. Finally, Section 5 presents the conclusions and future work.

2 Background and Related Work

This section provides background on PIM (§2.1). Next, it discusses related work on concurrency control (§2.2), deterministic transactional systems (§2.3) and PIM-accelerated databases (§2.4).

2.1 Processing-in-Memory

Overview of PIM. One approach to enable in-memory computation is known as Processing Using Memory (PUM). This technique takes advantage of existing logic within DRAM chips to perform data copying, bitwise operations, and basic arithmetic tasks directly inside memory arrays, without requiring significant hardware changes. A more flexible alternative is Processing Near Memory (PNM), which augments memory modules with dedicated, general-purpose compute units positioned in close physical proximity to the memory. Despite PNM processors being less powerful than modern CPUs, they can execute a broader range of application logic than PUM processors. For this reason, this work focuses on the PNM approach. Together, PUM and PNM represent the two predominant architectural strategies for implementing Processing-in-Memory systems [42].

The UPMEM Architecture. The UPMEM PIM platform [29] is the first real-world hardware implementation of a PNM architecture. Each UPMEM module conforms to a standard dual in-line memory module (DIMM) form factor but incorporates several PIM-enabled chips. As depicted in Figure 1, each chip houses eight Data Processing Units (DPUs). Each DPU contains a **64MB** DRAM bank, referred to as **MRAM**,

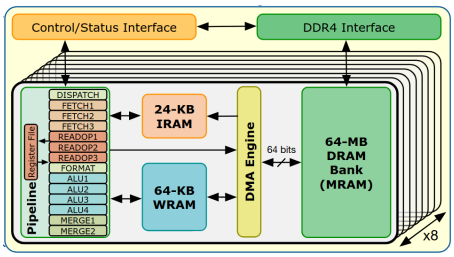


Figure 1. Internal depiction of an UPMEM PIM chip [29].

24KB of instruction memory (**IRAM**), **64KB** of fast scratchpad memory (**WRAM**) and a lightweight general purpose core. This multithreaded, in-order 32-bit RISC core operates at 350 MHz and supports up to **24 hardware threads**. A fully populated UPMEM system contains **2560 DPUs** in total, yielding 160 GB of memory.

Concurrency in UPMEM PIMs. Although each DPU can spawn up to 24 concurrent threads (referred to as tasklets)—matching the 24 hardware threads—empirical evidence indicates that peak instruction throughput is typically reached when running 11 tasklets, which matches the DPU's pipeline depth. Hence, the full UPMEM system with 2560 DPUs enables the concurrent execution of up to 28,160 tasklets.

DPUs operate independently and inter-DPU communication must be explicitly coordinated via the host CPU, whereas tasklets executing on the same DPU can directly communicate through local memories, namely WRAM and MRAM. Consequently, to achieve optimal performance on UPMEM platforms (and PIM architectures more broadly), applications should be designed to maximize memory access locality and minimize costly inter-DPU communication.

A further limitation of the current UPMEM architecture is the mutual exclusivity between computation and communication on a DPU. Specifically, data transfers between the CPU and the DPU-attached DIMMs can only occur when the target DPUs are idle—i.e., not performing any computation [15]. This restriction effectively prevents the overlap of computation and data movement. For example, it is not possible to copy intermediate transaction results from the DPU or to have DPUs wait for a signal from the CPU via shared CPU-DPU variables without first halting computation.

UPMEM Programming. The UPMEM software stack provides a runtime library with architecture-specific primitives—low-level routines for direct hardware interaction—a subset of the standard C library, and a Clang-based compiler. Developers write DPU programs in C using a Single Program Multiple Data (SPMD) model, where tasklets operate on different portions of data and can follow independent control paths. While DPUs function as general purpose processors, they lack native hardware support for complex floating-point

operations. These tasks are instead handled via software emulation, resulting in a higher computational cost compared to standard CPUs. For intra-DPU synchronization, UPMEM exposes two atomic instructions—`acquire` and `release`—that allow tasklets on the same DPU to coordinate using lock-based mechanisms. These primitives operate on a 256-bit atomic register, where each bit can be used as a lock. When a tasklet issues an atomic instruction with a memory address, a hardware hash function maps that address to a specific bit in the register, enabling atomic locking or unlocking of that logical lock. However, the runtime currently only offers these low-level synchronization operations and does not support more advanced constructs like read–write locks. Finally, the limited instruction RAM, restricted to only 24 KB, represents another relevant constraint on the size of programs that can be executed on a DPU.

2.2 Concurrency Control

Designing concurrency control schemes has been extensively studied in both the Database Management Systems (DBMS) and, more recently, Transactional Memory (TM) literature. A key distinction is that TM research often adopts opacity [26] as its isolation criterion, which offers stronger guarantees than the consistency levels typically used in DBMSs, such as strict serializability [2]. Unlike opacity, criteria like strict serializability allow transactions that will eventually abort to read from inconsistent states. This is considered acceptable in DBMS environments due to their sandboxed nature [16]. In contrast, TM systems are typically integrated into low-level programming languages (e.g., C/C++), where accessing inconsistent memory can result in severe program behaviors such as crashes or infinite loops [30, 48]. To prevent such issues, opacity requires that even aborted transactions observe only consistent states—specifically, states that correspond to some serial execution that preserves real-time ordering. Since PIM-TIDE targets C programs running in a non-sandboxed environment (the UPMEM system), it adopts opacity as its consistency model. Accordingly, we focus our review on concurrency control schemes from the TM literature, which are designed to provide opacity.

Specifically, we focus on software-based implementations of the transaction abstraction, also known as Software-TM (STM) [53], since existing PIM systems are not equipped with hardware mechanisms for transactional synchronization [31], and it is uncertain if they will be in the near future. STMs have been extensively studied in cache-coherent multi-core CPUs, resulting in numerous algorithms (e.g., NOrec [8], JVSTM [4], SwissTM [12], TinySTM [18]).

Recently, STMs have also been proposed for various hardware platforms, including embedded devices [20], non-cache-coherent many-core systems [24], distributed systems [3, 7], GPUs [45], and heterogeneous systems [6]. In this emerging line of research, PIM-STM represents the closest work to PIM-TIDE. Analogously to PIM-TIDE, PIM-STM provided implementations of the transaction abstraction for the UPMEM PIM system. PIM-TIDE builds upon the most robust STM implementation from PIM-STM (i.e., based on TinySTM [18]) for single DPU transactions and introduces a deterministic concurrency control scheme (see Section 3.2) for transactions spanning multiple DPUs. For this reason, PIM-TIDE achieves similar performance to PIM-STM in single DPU transactions and improves over PIM-STM by enabling inter-DPU transactions, which were previously not possible. Moreover, PIM-STM's single DPU restriction is deliberate as it avoids costly inter-DPU CPU-mediated communications. While this choice is motivated by efficiency concerns, it significantly restricts the generality and applicability of PIM-STM in current PIM systems, as it limits the scope of the atomicity guarantees provided by transactions to datasets that can fit on a single DPU (at most 64MB in the current UPMEM system). PIM-TIDE's hybrid concurrency control scheme overcomes this limitation by combining a local, non-deterministic STM scheme, with a deterministic concurrency control that enables coordinating the execution of transactions spanning multiple DPUs, while circumventing the need for additional inter-DPU distributed commit protocols (see Section 3, *Execution* paragraph).

2.3 Deterministic transactional systems

The concurrency control schemes discussed in the previous section are non-deterministic, meaning they allow transactions to be executed in any serialization order that satisfies the target consistency criterion (e.g., opacity or strict serializability). In contrast, deterministic transactional systems enforce a predefined serialization order. As a result, given the same initial state and inputs, every execution yields an identical final state. This additional guarantee eliminates non-determinism caused by thread scheduling or deadlock resolution, significantly simplifying replication and distribution: each replica or partition processes the same (sub)set of transactions in the same order, avoiding the well-known pitfalls of distributed locking and atomic commit protocols [25].

A canonical example of this approach is Calvin [56], which batches incoming transactions into epochs, determines their order via a distributed sequencer, replicates transaction inputs across nodes (e.g., using consensus [34]), and then uses a deterministic lock-scheduling protocol to execute transactions in that order. By decoupling transaction ordering from execution, Calvin achieves full ACID semantics across partitions without requiring Two-Phase Commit (2PC), enabling both high throughput and fault tolerance.

More recently, Epic [49] has built on Calvin's foundation by combining deterministic sequencing with Multi-Version Concurrency Control (MVCC) and GPU acceleration. Epic preprocesses transaction batches on the GPU to assign versions and precompute read/write dependencies, eliminating

runtime version-chain searches. Once preprocessing is complete, transactions are executed in parallel—on either the CPU or GPU—according to the globally defined order, delivering strong throughput gains under contention.

Sparkle [37] further advanced deterministic transaction processing by removing the assumption that a transaction's read and write sets must be known in advance. It also addressed a key bottleneck in Calvin: the use of a single thread for deterministic scheduling.

PIM-TIDE draws inspiration from these techniques in the deterministic distributed database literature, adapting and specializing them to function efficiently on a real PIM system.

Deterministic transaction ordering has also been proposed as a way to simplify the development, debugging, and testing of multithreaded applications. LiTM [59] achieves determinism by organizing transactions into batches and using a priority-based lock acquisition scheme: when multiple transactions contend for the same lock, only the one with the highest priority succeeds in reserving the data item. Conflicts are detected by verifying whether all required items have been successfully reserved. Transactions that observe no conflicts proceed to commit, while those that encounter conflicts are aborted and retried. LiTM enforces deterministic ordering by decoupling the execution (i.e., reservation) phase from the commit phase within each batch. However, this strict separation can unnecessarily extend transaction lifetimes, increasing contention and introducing conflicts that might not arise under a more relaxed scheduling model. This trade-off highlights the performance challenges of enforcing determinism in concurrent environments.

DeSTM [51] proposes a looser execution model. Instead of introducing full barriers between execution and commit phases, as in LiTM, DeSTM imposes two milder constraints: (i) all transactions in a round must begin before any transaction may initiate its commit phase; and (ii) new transactions can start only after all transactions from the previous round have completed. These constraints, together with a token-passing algorithm, are sufficient to guarantee deterministic execution. However, they still restrict scheduling flexibility and may result in reduced performance [44].

2.4 PIM-Accelerated Databases

Processing-in-Memory (PIM) has recently emerged as a promising approach to accelerate memory-bound database operations by offloading data-intensive tasks to compute units located near memory, especially in OLAP settings [1, 35, 54]. For example, Membrane [54] uses a hybrid CPU-PIM design to evaluate selection predicates directly in memory, reducing data movement. However, such systems primarily target read-heavy workloads with largely immutable state and, unlike PIM-TIDE, are not designed to support OLTP scenarios, which require frequent updates and coordinated concurrency control across multiple DPUs.

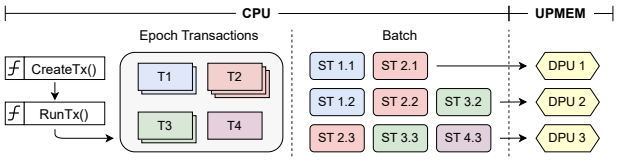


Figure 2. Communication between CPU and PIM

3 PIM-TIDE

This section discusses the design and implementation of PIM-TIDE, a system specifically designed for Processing Near Memory (PNM) architectures. Its architecture is depicted in Figure 2, which provides an overview of PIM-TIDE's main building blocks. PIM-TIDE, executes transactions in rounds, which we call epochs. Each epoch is subdivided into three logical phases: batching, communication, and execution.

Batching. In PIM-TIDE, transactions are submitted in batches to the PIM system. A transaction is composed of one or more subtransactions, where each subtransaction executes on a distinct DPU. A transaction is said to be local if it only accesses one DPU (i.e., contains only one subtransaction). Otherwise, it is said to be distributed. Analogously, we say that a subtransaction is local or distributed depending on the nature of the transaction it is associated with.

To generate a subtransaction, PIM-TIDE's users must specify: i) its target DPU; ii) the computational logic to be executed; iii) its input parameters. The execution of a subtransaction generates a (possibly void) result, which the applications can retrieve programmatically. The computational logic to be executed by each subtransaction on a DPU is a generic code routine for UPMEM that leverages the PIM-TIDE software API to demarcate transaction's start and end, as well as transactional read/write access to memory. In this sense, subtransactions are akin to stored procedures (i.e., a precompiled collection of SQL statements stored within a database designed for execution).

The input and output parameters of a subtransaction are encoded in the form of a bit array, which provides both increased efficiency, since the parameters can be packed, effectively eliminating wasteful padding, and greater flexibility due to the ability to include parameters of different data types. Note that unlike many deterministic concurrency controls (e.g., [13, 32, 49, 56]), PIM-TIDE eschews the assumption of a priori knowledge of the transactions' read and write sets — which increases PIM-TIDE's flexibility, ease of use and avoids the need for reconnaissance queries.

Algorithm 1 presents an example of the usage of the programming interface exposed by PIM-TIDE on the CPU side. It depicts the creation of a transaction T, whose logic is inspired by the *Payment* transaction of TPC-C. This transaction T can contain one or two subtransactions (referred to as A and B). In the latter case (i.e., T is a distributed transaction), each

Algorithm 1 Example usage of PIM-TIDE's API (Host side)

```
1:  function CreateTx(w_id, d_id, c_id, c_w_id, amount)
2:      transaction ← create_transaction()
3:      sub_transaction_a ← create_sub_transaction()
4:      sub_transaction_b ← create_sub_transaction()
5:      sub_transaction_a.set_type(PAYMENT)
6:      sub_transaction_a.add_param(w_id)
7:      sub_transaction_a.add_param(d_id)
8:      sub_transaction_a.add_param(c_w_id)
9:      sub_transaction_a.add_param(c_id)
10:     sub_transaction_a.add_param(amount)
11:     tx.add(w_id, sub_transaction_a)
12:     if c_w_id ≠ w_id then                          ▷ This is a remote payment
13:         sub_transaction_b.set_type(PAYMENT_REMOTE)
14:         sub_transaction_b.add_param(w_id)
15:         sub_transaction_b.add_param(d_id)
16:         sub_transaction_b.add_param(c_w_id)
17:         sub_transaction_b.add_param(c_id)
18:         sub_transaction_b.add_param(amount)
19:         tx.add(c_w_id, sub_transaction_b)
20:     RunTx(transaction)
```

Algorithm 2 Pseudo-code of an example program running on UPMEM's DPU, inspired by the TPC-C Payment.

```
1:  function execute_payment_sub_tx(params, commit_ts, execution_mode)
2:      tx ← create_tx_log      ▷ Instantiate a log storing transactional metadata
3:      w_id ← read_next_param(params)
4:      d_id ← read_next_param(params)
5:      c_w_id ← read_next_param(params)
6:      c_id ← read_next_param(params)
7:      amount ← read_next_param(params)
8:      START(tx, commit_ts, execution_mode)
9:      warehouse ← READ_WORD(tx, warehouse_arr[w_id])
10:     WRITE_WORD(tx, warehouse.ytd, READ_WORD(tx, warehouse.ytd) + amount)
11:     district ← READ_WORD(tx, district_arr[d_id])
12:     WRITE_WORD(tx, district.ytd, READ_WORD(tx, district.ytd) + amount)
13:     if w_id == c_w_id then                          ▷ Local payment
14:         c ← READ_WORD(tx, customer_arr[c_id])
15:         WRITE_WORD(tx, c.balance, READ_WORD(tx, c.balance) - amount)
16:     COMMIT(tx)
```

subtransaction targets a different DPU. This example starts by instantiating a transaction object and two subtransaction objects (lines 2–4). The stored procedure executed by A in a specific DPU is defined in line 5. Then, the input parameters for A are appended to the corresponding sub_transaction object (lines 6–10). Next A is added to the encompassing transaction (line 11). In case the condition in line 12 is true, the same process is repeated for B (lines 13–19). After completing the association of A, and possibly B to T, T is handed over to the batching protocol (line 20).

At the end of each epoch, the set of previously received transactions is arranged into a batch. As depicted in Figure 2, batches are composed solely of subtransactions. These subtransactions are arranged into sets, each containing subtransactions aimed at a distinct DPU (e.g., using 2048 DPUs, a batch will have 2048 sets of subtransactions). Within a batch, subtransactions belonging to distributed transactions are assigned a commit timestamp by the CPU. As we will see, during the execution phase, this timestamp (*commit_ts* in line 6 of algorithm 3) is enforced by PIM-TIDE's deterministic concurrency control, sparing distributed transactions from the need to execute additional commit protocols, such as, Two-Phase Commit (2PC) [2].

Communication. Once a batch is constructed, it is transferred to the DPUs. Performing this CPU-DPU communication in batches is essential to maximize performance for two reasons. First, the CPU-DPU bandwidth increases (up to a point) as the data size increases. Therefore, a larger batch size enables faster communication speeds. Second, each epoch requires activating DPU kernels, which imposes a fixed cost. The larger the batch size, the higher the number of transactions that can be executed per epoch. As a result, executing the same number of transactions requires a lower number of epochs, which reduces the overall kernel activation costs. Clearly, larger batch sizes can increase transaction processing latency as individual transactions can only finish after all transactions in the batch have executed. Ultimately, the duration of the batching phase is an application-dependent decision, as this parameter should be tuned to match the maximum admissible latency of the target application — thus striking a good balance between latency and throughput.

Finally, since in the current UPMEM system, communication and computation cannot be overlapped, PIM-TIDE has to wait till the previous batch is processed to start transferring a new batch. However, it still overlaps construction of batch $i + 1$ on the CPU with the processing of batch i on UPMEM.

Execution. In this phase, each DPU executes the set of subtransactions that were assigned to it. This set may include both local and distributed subtransactions. PIM-TIDE provides a programmatic interface that, besides transaction demarcation (begin and commit), allows transactional reads ($read\_word(addr)$) and writes ($write\_word(addr)$) to addresses of the DPU's MRAM, at word-based granularity (4 bytes). This allows C/C++ stored procedures, intended to be executed by subtransactions in DPUs, to safely access shared memory in a multithreaded environment. Despite providing this relatively-low level word-based programming interface, PIM-TIDE represents a foundation over which future work can build interfaces at a higher abstraction level (e.g., key value stores or SQL databases).

Algorithm 2 shows the pseudo-code of an example program, intended to run on a DPU, that encodes the logic of subtransaction T. It starts by instantiating a log object that holds the transactional metadata (e.g., commit timestamp, read/write sets) (line 2). Then, the parameters of T are parsed and stored (lines 3–7). Once all parameters are collected, T demarcates the start of the transactional region by invoking the $start()$ function along with the parameters that define the commit timestamp and execution mode of T (line 8).

Using the routines provided by the concurrency control algorithm, T reads and writes several memory words (lines 9–15). Lastly, T commits the writes performed using $write\_word()$ by calling the commit procedure (line 16).

PIM-TIDE adopts a mixed execution strategy that leverages both deterministic and non-deterministic concurrency control schemes (in two sequential phases), using the former for distributed transactions and the latter for local ones. The key observation is that, since local transactions execute entirely within a single DPU, they are not required to obey global or inter-DPU ordering constraints. As a result, their serialization order can be determined solely based on intra-DPU conflict dynamics, using a lightweight, non-deterministic concurrency control mechanism. [1] As we will demonstrate in Section 4.3, non-deterministic concurrency control achieves up to 1.5× speedup compared to deterministic execution when processing local transactions.

Conversely, subtransactions belonging to distributed transactions are executed using a deterministic concurrency control that executes transactions speculatively (i.e., with no a priori knowledge of their read- and write-sets), while still enforcing that their serialization order is equivalent to the one established by the CPU, at batch construction time. This approach eliminates the need for communication between DPUs involved in a transaction to reach a commit decision, unlike when using traditional distributed commit protocols (e.g., 2PC). Further, it guarantees that every transaction can commit within the same epoch in which it execute (i.e., there is no need for re-execution in other epochs). In the event of a conflict between two subtransactions that access the same position, resolution occurs within the same epoch by having one of them retry. This represents a significant advantage over solutions based on traditional distributed commit protocols, as it reduces the volume of CPU-DPU communication, which is crucial to maximize performance on the UPMEM hardware. Despite the intra-DPU overheads introduced by deterministic execution, this solution largely outperforms alternative approaches based on 2PC (Section 4.3).

Finally, to avoid the complexity of handling the co-existence of subtransactions regulated by both deterministic and non-deterministic concurrency control schemes, PIM-TIDE schedules distributed subtransactions (handled by deterministic concurrency control) before executing local subtransactions (handled by non-deterministic concurrency control).

The remainder of this section is structured as follows: Section 3.1 discusses the particularities of PIM-TIDE's data partitioning scheme. Section 3.2 presents the transaction execution model. Section 3.3 describes the challenges and optimization opportunities associated with the implementation of PIM-TIDE on the UPMEM PIM system. Finally, Section 3.4 explains how PIM-TIDE could recover from crashes by integrating techniques based on SMR.

[1] From the perspective of serializability theory [2], PIM-TIDE ensures that any edge in the conflict graph between the sub-graph, $\mathcal{G}_L$, corresponding to local transactions and the sub-graph, $\mathcal{G}_D$, corresponding to distributed transactions must be oriented from $\mathcal{G}_L$ to $\mathcal{G}_D$. This ensures that the global graph considering all local and distributed committed transactions is cycle free, provided that $\mathcal{G}_L$ and $\mathcal{G}_D$ do not contain cycles.

3.1 Data model

In PIM-TIDE, programmers must define a data partitioning scheme that minimizes the need for subtransactions to access data stored on remote DPUs. While remote data access is supported, it requires synchronizing the execution of dependent subtransactions—i.e., delaying a subtransaction until its input becomes available—which can reduce throughput and increase latency. The difficulty of crafting an effective partitioning strategy is highly application-specific. In general, finer-grained partitioning increases the need for algorithmic changes to adapt the application's logic and adds runtime overheads, such as higher communication costs for distributing inputs and aggregating results as the number of subproblems grows [43]. To balance programmability and performance, PIM-TIDE encourages relatively coarse-grained partitioning, typically aligning with the 64 MB memory available per DPU in the current UPMEM architecture.

Finally, a key design choice of PIM-TIDE is to restrict subtransactions to only access data stored at their target DPU. This choice is based on two observations. First, the UPMEM system does not provide support for direct communication between DPUs and inter-DPU communication has to be mediated by the CPU, which is costly. In fact, we empirically verified that the latency of a CPU-mediated inter-DPU read for a 64-bit memory word is three orders of magnitude larger than a read to the local DPU MRAM (namely $331\mu s$ vs. $231ns$, respectively). Second, communication and computation cannot be overlapped, preventing the use of speculative techniques used in the distributed TM literature to mask inter-node communication latency [38, 47]. For these reasons, the deliberate design decision to limit the scope of subtransactions to the DPU in which they are executed represents an intentional trade-off, which aims to maximize efficiency at the cost of expressiveness of the programming model. Specifically, this enables transaction processing to scale across thousands of DPUs with minimal communication overhead. Note that, despite this restriction, transactions can still span multiple DPUs by using several subtransactions.

3.2 Transaction execution

During the execution phase, each DPU independently processes its assigned subtransactions. As noted earlier, when both deterministic (i.e., distributed) and non-deterministic (i.e., local) subtransactions are assigned to the same DPU within an epoch, PIM-TIDE enforces a serialization order in which deterministic subtransactions are executed first. Only

Algorithm 3 PIM-TIDE's Concurrency Control

```
1:  Global data structures:
2:      Int CLOCK                          ▷ Global clock shared by all subtransactions
3:      Set<LockEntry> LOCK_TABLE          ▷ Lock table shared by all subtransactions
4:  Data structures local to each tasklet:
5:      Int ts                             ▷ Value of the global clock read at the start
6:      Int commit_ts                      ▷ Timestamp that specifies the order for deterministic
    subtransactions
7:      Enum<DET, NON_DET> mode            ▷ Execution mode
8:      Set<ReadLogEntry> read_log         ▷ Log that keeps track of every read
9:      Set<WriteLogEntry> write_log       ▷ Log that keeps track of every write
10: function START(tx, commit_ts, mode)
11:     tx.ts ← CLOCK
12:     tx.c_ts ← commit_ts
13:     tx.mode ← mode
14: function READ_WORD(tx, addr)
15:     lock ← MAP_ADDR_TO_LOCK(addr)
16:     if OWNER(lock) == tx then
17:         return READ(addr)
18:     do
19:         if WAS_ABORTED(tx) then
20:             break
21:         version ← GET_VERSION(lock)
22:         value ← READ(addr)
23:         if ¬LOCKED(lock) ∧ GET_VERSION(lock) == version ∧ version ≤ tx.ts then
24:             ADD_TO_READ_LOG(tx, lock)
25:             return value
26:     while LOCKED(lock) ∧¬CM_SHOULD_ABORT(tx, OWNER(lock))
27:     ROLLBACK(tx)
28: function WRITE_WORD(tx, addr, value)
29:     lock ← MAP_ADDR_TO_LOCK(addr)
30:     if OWNER(lock) == tx then
31:         WRITE_TO_MEMORY(addr, value)
32:         return
33:     while true do
34:         if WAS_ABORTED(tx) then
35:             ROLLBACK(tx)
36:         lock_value ← GET_VALUE(lock)
37:         version ← GET_VERSION(lock)
38:         if LOCKED(lock_value) then
39:             if CM_SHOULD_ABORT(tx, OWNER(lock)) then
40:                 ROLLBACK(tx)
41:             else
42:                 continue
43:         log_entry ← ADD_TO_WRITE_LOG(tx, lock, addr, value)
44:         if CAS(lock, lock_value, log_entry) == lock_value then
45:             break
46:     if version > tx.ts then
47:         ROLLBACK(tx)
48: function COMMIT(tx)
49:     if tx.mode == DET then
50:         while CLOCK < tx.c_ts do
51:             if WAS_ABORTED(tx) then
52:                 ROLLBACK(tx)
53:     t ← ATOMIC_INCREMENT_AND_GET(CLOCK)
54:     if IS_READ_ONLY(tx) then
55:         return
56:     if t > tx.ts + 1 ∧¬VALIDATE(tx) then
57:         ROLLBACK(tx)
58:     for log_entry in tx.write_log do
59:         WRITE_VERSION_AND_DROP(log_entry.lock, t)
60: function CM_SHOULD_ABORT(self_tx, other_tx)
61:     if self_tx.mode == NON_DET then
62:         return true
63:     if self_tx.c_ts > other_tx.c_ts then
64:         return true
65:     ABORT(other_tx)
66:     return false
```

after completing the distributed transactions do DPUs execute local ones in parallel, ensuring correctness. The remainder of this section details the concurrency control mechanisms used in PIM-TIDE. Our design builds on PIM-STM [39], which evaluated a broad range of non-deterministic concurrency control algorithms on single-DPU UPMEM-like systems. That study found that timestamp-based algorithms with fine-grained encounter-time locking and write-through policy—such as the one in TinySTM [18]—consistently deliver strong performance across diverse workloads. Accordingly, PIM-TIDE adopts the TinySTM variant developed in PIM-STM for local, non-deterministic execution, and extends it with additional mechanisms to support the deterministic guarantees required for distributed transactions.

The logic executed in every DPU (where the transactions are executed) is coded using C, where it is not possible to sandbox the state of the STM from the application's (unlike in a traditional database that exposes a query language interface like SQL). In such an environment, to avoid anomalous application behavior (that can lead to crashes, array overruns, etc.), the correctness guarantees offered by opacity [16, 26] are essential. Specifically, opacity guarantees that every transaction, including the ones that eventually abort, observes a state that can be explained via a sequential execution, ruling out the possibility of externalizing the writes of uncommitted transactions to concurrent transactions.

Algorithm 3 presents the pseudo-code for both deterministic and non-deterministic concurrency control algorithms used in PIM-TIDE, where the choice of which algorithm to use is established at transaction start, via a boolean flag passed as input to the start transaction primitive. We begin by introducing the key data structures used to regulate concurrent transaction execution, before describing the concurrency control logic.

Main data structures. In PIM-TIDE all subtransactions that execute on a target DPU share a global clock, *CLOCK*, which is incremented upon every commit. Every memory word that is read/written in a transactional context is mapped to an entry in a global lock table. Each entry e, is a memory word that can be in one of two modes: locked and unlocked. When locked, the least significant bit of e is set to one, symbolizing that a subtransaction is trying to write to an address that maps to e. When e is unlocked, its value represents the version of the memory address. This version is dictated by the timestamp of the subtransaction that last updated the address (this design was originally proposed by TL2 [11]). The size of the lock table (which is determined at compile time) dictates a trade-off between memory usage and aliasing. Aliasing occurs when different memory positions are mapped to the same lock table entry. Using a larger lock table leads to less aliasing (and thus less unnecessary aborts). However, a larger lock table also takes up more space. This is a particularly important consideration in the UPMEM hardware, given its limited storage capacity. Every subtransaction maintains local data structures that store 1) the value of the global clock read at the start and 2) two sets that track every

memory address read and written (the write-set also contains the value prior to the transaction's update, thus serving as an undo log), which is used to validate the transaction and rolling it back, if necessary. In addition, the metadata of deterministic subtransactions also includes a commit timestamp that specifies the order in which they must be committed.

Start. Every subtransaction, upon start, reads and stores the global clock (line 11); records the commit timestamp, which defines the serialization order of deterministic subtransactions (line 12); and sets the execution mode (line 13), which determines whether the subtransaction behaves deterministically (DET) or non-deterministically (NON_DET).

Read. When reading the memory position *addr*, subtransaction T starts by reading the state of the corresponding *lock* to identify possible read-after-write scenarios. If T is the owner of *lock*, it returns the value of *addr*, which corresponds to the last value written by T to that memory position (lines 15–17). If T is not the owner of *lock* or it is unlocked, it checks if it has been aborted (keep in mind that only deterministic subtransactions can be aborted by concurrent subtransactions). If that is the case, T rolls back by undoing all its writes and releasing all locks being held (lines 19–20 and 27). If T has not been aborted, it reads the version *version* corresponding to *addr* in the lock table (when *lock* is unlocked, it maintains the version of *addr*, i.e., the timestamp of the last transaction to have updated *addr*) the value of *addr*, and the version from *lock* again. If 1) the two versions read from *lock* are the same, indicating that *version* and *addr* were read atomically, 2) *lock* is in the unlocked state, and 3) *version* is lower or equal to the global clock value observed when T started, which means that *addr* has not been changed since T started (and opacity is guaranteed), T adds this read to the read log and returns the previously read (in line 22) value of *addr* (lines 21–25). Otherwise, if the read was not successful because *lock* was locked, T invokes the contention manager (line 26) to determine whether to rollback and retry the entire subtransaction again (line 27) or wait for the current owner of *lock* to release it.

Write. When subtransaction T writes to the memory position *addr* the value *value*, it starts by checking if it owns the lock *lock* that covers *addr*. If T is the owner, it writes *value* directly to memory and returns (lines 30–32). If T is not the owner of *lock* or it is unlocked, T checks if it has been aborted. If that is the case, T performs a rollback, undoing all its writes and releasing all locks being held (lines 34–35). Otherwise, T tries to atomically acquire *lock* using a compare-and-swap (CAS) primitive, which replaces the old value of *lock* with the address of T's write log entry that stores the new value of *addr* (lines 43–44). If the CAS fails, T invokes the contention manager to determine whether to rollback and retry the entire subtransaction or wait for the current owner of *lock* to release it (line 39). If the CAS succeeds, T checks if the version of *addr* is lower or equal to the global clock value observed by T when it started. If not, T has to rollback (lines 46–47). Note that the pseudo-code relies on CAS only to aid presentation. However, UPMEM does not provide the CAS primitive, but only weaker atomic operations, which we leverage to implement the CAS abstraction, as we will discuss in Section 3.3.)

Commit. When subtransaction T commits, if it is deterministic, T waits until the global clock reaches its assigned commit timestamp, while simultaneously checking whether it has been aborted in the meantime (lines 49–51). If T was aborted, it rolls back and reties (line 52). When the global clock reaches T's commit timestamp, in the case of a deterministic transaction, or if T is non-deterministic, the global clock is incremented, and its updated value, t, determines the serialization order of T (line 53). If T is read-only, it can be committed immediately, since its observed snapshot is guaranteed to be consistent (lines 54–55). Otherwise, if the global clock changed since T started (signaling the commit of a concurrent subtransaction that may have written to a position read by T), T revalidates its read log, to ensure that it is consistent. If validation fails, T rollbacks and retries (lines 56–57). If validation is successful, T releases all locks after updating the version of every entry in the lock table that covers the memory positions written to by T. This is done by writing the serialization timestamp t, obtained during commit (lines 58–59), to each entry.

Contention Management. When contention arises between two subtransactions T and T' that are trying to access memory addresses mapped to the same lock table entry, the contention manager determines which of them must abort and restart. Upon conflict, if T is non-deterministic, it immediately aborts itself (lines 61–62) — this simple contention management strategy ensures deadlock-freedom and is frequently adopted for its lightweight nature in STM environments [17, 39]. If T is deterministic and its commit timestamp is greater than the commit timestamp of T', that is, T must be serialized after T', T aborts itself (lines 63–64). Otherwise, if the commit timestamp of T is smaller, T aborts T' (line 65). During execution, if T' detects that it has been aborted, it rolls back and retries.

3.3 UPMEM-related challenges and optimizations

This section describes the challenges and optimizations related to the implementation of PIM-TIDE on UPMEM.

Hardware synchronization primitives. So far, we have assumed the availability of the CAS primitive to implement PIM-TIDE's lock table. However, since the CAS instruction is not available on the UPMEM system, we implement it in software by relying on the *acquire* and *release* instructions, see Section 2.1. More precisely, we first acquire a lock on the address targeted by the CAS operation, then we check if

the current value matches the expected one and finally, we release the lock. Recall that the acquire and release atomic instructions are implemented via a 256-bit atomic register. Thus, when two tasklets try to acquire locks on distinct addresses that are mapped to the same bit of the atomic register, the two tasklets may suffer lock aliasing and be unnecessarily serialized. However, in PIM-TIDE, this serialization occurs only for the time needed to consult and possibly update the shared data. This is a relatively short period of time compared to the duration of a subtransaction. Furthermore, since the acquire/release primitives of UPMEM operate on a hardware register (i.e., they do not access WRAM or MRAM), their overhead is minimal in practice.

WRAM management. The UPMEM system features two types of memory for application data: WRAM and MRAM. WRAM is faster, but has limited capacity, whereas MRAM offers larger storage at the cost of lower speed (see Section 2.1). To enhance efficiency, PIM-TIDE places in WRAM its lock table, which has a statically defined size and is accessed multiple times for every transactional access. Conversely, since the number of data accesses performed by a transaction can vary widely, we opt for storing the read and write logs (which may easily exceed the capacity of WRAM) in MRAM.

Further, every access to MRAM requires first transferring the target data to WRAM via a DMA engine. These transfers achieve maximum sustainable bandwidth only when operating with granularity above 1 KB [23]. PIM-TIDE faces this exact challenge, since the subtransactions scheduled for execution in an epoch are initially located in MRAM (the host cannot access WRAM and must copy them to MRAM) and must be transferred to WRAM to be processed. To maximize efficiency, in PIM-TIDE every DMA transfer retrieves multiple (i.e., a chunk) of transactions, where the size of each chunk is chosen to allow the MRAM bandwidth to be maximized. These chunks are then cached in WRAM for future accesses. By default, the data structures maintained in WRAM by PIM-TIDE are sized in order to occupy about 75% of WRAM, leaving the remainder available for applications. However, PIM-TIDE's design allows for easily reducing the sizes of these data structures, if additional WRAM is required for application purposes.

Pipelining. As mentioned in Section 2.1, the UPMEM system does not allow for overlapping data transfer with processing (on the PIM side). However, each epoch in PIM-TIDE requires three sequential phases: batching, communication and computation. This creates the opportunity to pipeline the PIM computation phase of epoch i, with the batching phase (performed by the host) of epoch $i + 1$. This optimization is highly effective at high load, where the execution phase is normally longer than batching — thus allowing for totally removing the batching latency from the critical path.

3.4 Fault-tolerance

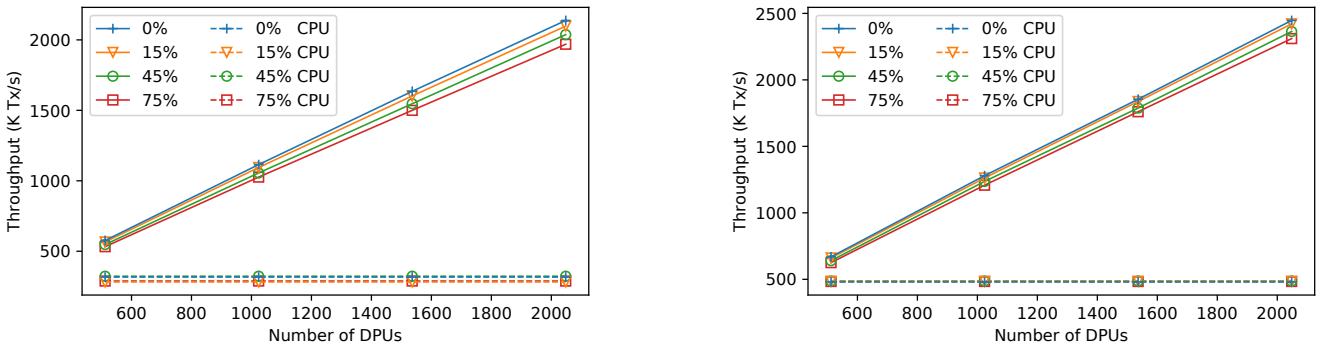
A key advantage of deterministic concurrency control schemes is that they enable the employment of well-studied State Machine Replication (SMR) techniques [34, 46, 56, 61], which provide fault-tolerance and high availability.

The current implementation of PIM-TIDE does not guarantee fault-tolerance, but its design allows for straightforwardly integrating additional techniques based on SMR. Specifically, since PIM-TIDE employs deterministic concurrency, multi-DPU transactions are already assigned a serialization order by the CPU. This order can be easily disseminated to any replica (i.e., other independent machines equipped with UPMEM DPUs), ensuring they serialize transactions in the same order as the main node.

The only source of non-determinism in PIM-TIDE is the processing of local (i.e., single DPU) transactions, whose serialization order is not determined prior to their submission to the target DPU. However, to maintain system-wide correctness, this serialization order must be disseminated to all replicas, ensuring they execute transactions in the same order as the primary node. One simple way to deal with this source of non-determinism is to elect a master that operates using the mixed (deterministic and non-deterministic) execution strategy described in Section 3.2 and to replicate its states on a set of backup replicas. At the end of an epoch the master disseminates the serialization order not only of deterministic/distributed transactions, but also of non-deterministic/local transactions to its replicas. The backups execute all (both local and distributed) transactions using the deterministic strategy, following the order established by the master.

Backup replicas could be deployed on distinct UPMEM systems to survive the failure of an entire system. Alternatively, one could replicate the state of a DPU on one or more local DPUs of the same PIM system. The latter approach would imply sacrificing the effective memory capacity available at each DPU, but would allow for tolerating partial failure of the PIM system — provided that the replicas of a given DPU are deployed on distinct DIMMs of the PIM system (see Section 2.1), so they can be assumed to fail independently.

The fault-tolerance mechanisms described above require including the serialization timestamp of each non-deterministic subtransaction upon its submission to a backup PIM. Arguably, though, this would impose negligible overhead since the additional information (i.e., a timestamp) is minimal compared to the amount of metadata each subtransaction already requires. Furthermore, determining the serialization timestamp requires no additional computation as each subtransaction already needs to obtain it at commit time.



(a) STD: 44% *Payment*, 43% *NewOrder*, 13% *OrderStatus*

(b) CUST: 25% *Payment*, 25% *NewOrder*, 50% *OrderStatus*

Figure 3. Throughput for a mix workload of TPCC with multiple DPUs. Solid and dashed lines correspond to PIM-TIDE and to the CPU baseline (52 cores Intel Xeon Gold 5320), resp. Different colors indicate different percentages of *Payment* transactions accessing multiple DPUs.

4 Experimental Evaluation

This section evaluates the performance of PIM-TIDE across a variety of workloads and deployment scenarios. Our study is structured to answer the following questions:

- Q1 What performance and energy gains can be achieved using PIM-TIDE to accelerate OLTP applications? (§4.2)
- Q2 What is the performance cost of executing distributed transactions across multiple DPUs? (§4.2)
- Q3 How do different concurrency control strategies — deterministic and non-deterministic — compare in terms of performance under varying workload characteristics? (§4.3)

We start by assessing the performance and energy gains achievable by using PIM-TIDE to accelerate OLTP workloads originally developed for CPUs (Q1). We then determine the scalability and the cost of distributed transactions (Q2). Finally, to evaluate the performance characteristics of both non-deterministic and deterministic execution strategies, we perform single-DPU evaluation. This allows us to examine both approaches under increasing intra-DPU parallelism, without the complexities of distributed coordination (Q3).

4.1 Evaluation settings

All experiments were conducted on systems equipped with UPMEM PIM hardware. Our primary testbed includes two Intel Xeon Silver 4215 CPUs, 256 GB of DRAM, and 160 GB of PIM-enabled memory distributed across 2560 DPUs (see §2.1). For the CPU benchmarks, used to compare the performance and energy efficiency of PIM-TIDE, we used a system with an Intel Xeon Gold 5320 CPU (52 hardware threads) and 250 GB of DRAM. The Xeon Gold 5320 offers higher performance than the Xeon Silver 4215 and arguably provides a more balanced comparison against the PIM setup. Unless otherwise stated, each data point represents the average of 10 independent runs.

The code for PIM-TIDE, along with all the benchmarks presented in this paper, is publicly available[2].

Most of our evaluation is based on the TPC-C benchmark [57], a widely adopted standard for assessing the performance of online transaction processing (OLTP) systems. TPC-C captures the characteristics of real-world transactional workloads, including a mix of read and write operations, contention hotspots, and skewed access patterns. Our implementation builds on a Transactional Memory-based port used in prior works [5, 19], which itself follows the specification by M. Stonebraker et al. [55]. The baseline CPU implementation of TPC-C uses the original implementation of TinySTM [17].

Due to the memory limitations of the UPMEM architecture—particularly the instruction memory size per DPU—we focus on a representative subset of the benchmark consisting of the *Payment*, *NewOrder*, and *OrderStatus* transactions. By considering different mixes of these three transaction profiles, we can evaluate PIM-TIDE with workloads that generate diverse access patterns.

PIM-TIDE partitions the TPC-C schema by by assigning each warehouse and all its associated records (districts, customers, orders, and stock) to a distinct DPU. Since most TPC-C transactions target a single warehouse, this partitioning strategy naturally minimizes distributed transactions (and cross-DPU interactions) while preserving the benchmark's semantics.

To characterize the performance of PIM-TIDE, we vary the number of threads per DPU, the number of DPUs (same as the number of warehouses), and the mix of transaction

[2]https://github.com/Andre12Lopes/PIM-TIDE

profiles. We report throughput and abort rates, which together capture system efficiency, scalability, and robustness under varying degrees of contention and parallelism.

For the sake of simplicity, we refer to each workload by its mix of transaction types. The mix 44/43/13 (also referred to as STD), represents a workload with 44% *Payment* transactions, 43% *NewOrder* transactions and 13% *OrderStatus* transactions. The mix 25/25/50 (i.e., 25%, 25%, 50% for *Payment*, *NewOrder*, and *OrderStatus* transaction profiles, resp.) is referred to as CUST (or custom). In some experiments, we also adjust the probability with which *Payment* transaction can be distributed over multiple DPUs. For example, if the STD workload is configured to have 75% of the *Payment* transaction to access data in multiple DPUs, then the workload will be denoted as STD 75.

Further, we also use a synthetic benchmark, *Bank*, in which each transaction simulates a transfer between 2-100 bank accounts, determined uniformly at random from a set of 200 000 bank accounts. In our study we will vary the number of bank transfers in order to easily synthesize workloads with different characteristics.

Across all experiments, PIM-TIDE uses a 32KB lock table (the largest lock table size that fits in a DPU's WRAM) and a batch size of 128 transactions per batch.

4.2 PIM-TIDE Performance and Energy advantage

This section assesses the performance and energy gains achievable by using PIM-TIDE to accelerate data intensive workloads originally developed for CPUs. Section 4.2.1 analyses performance. Section 4.2.2 presents energy gains.

4.2.1 Performance gains. Figure 3 shows the transaction throughput of the TPC-C benchmark when considering the STD and CUST mixes (Figures 3a and 3b, resp.). In each plot, different color lines correspond to different percentages of *Payment* transactions that access multiple DPUs (from 0% to 75%, where 15% is the standard value according to the benchmarks' specifications). The solid lines correspond to PIM-TIDE and the dashed lines represent the CPU baseline. Each curve reports throughput as a function of the number of DPUs for increasing percentages of distributed *Payment* transactions (0%, 15%, 45%, and 75%).

Across both STD and CUST workloads, PIM-TIDE consistently outperforms the CPU baseline, with the performance gap widening as the number of DPUs increases. The peak gains (6.75×) are achieved when the fraction of distributed *Payment* transactions is low (STD 0, STD 15, CUST 0 and CUST 15), as these scenarios incur less CPU-mediated coordination overhead. Although the performance difference narrows somewhat as the proportion of distributed transactions rises, reflecting the increased communication cost, PIM-TIDE continues to deliver substantial throughput gains even with 75% of distributed *Payment* transactions (6.82× in STD 75 and 4.78× in CUST 75) .

Comparing the two workloads, PIM-TIDE achieves higher absolute throughput in the 25/25/50 configuration (Figure 3b). This improvement is primarily due to the higher fraction of read-only *OrderStatus*, leading to reduced contention.

Overall, the results demonstrate that PIM-TIDE scales efficiently with the number of DPUs and maintains superior performance across different workload compositions and contention levels. At 2000 DPUs, PIM-TIDE achieves up to a 10× throughput improvement over the CPU baseline. These gains stem from two key factors: (i) the massive parallelism enabled by UPMEM's thousands of DPUs and (ii) the ability of DPUs to perform computation closer to memory, reducing data access latency compared to the host CPU.

In order to shed additional light on the sources of performance gains for PIM-TIDE, in Figure 4 we report the speedup of the PIM-TIDE-based implementation relative to an equivalent CPU baseline for each individual TPC-C transaction profile, under a workload with 0% distributed transactions and varying numbers of DPUs.

The *OrderStatus* transaction exhibits the highest speedup, surpassing 10× at 2,048 DPUs. This strong scaling behavior stems from its read-mostly nature and lack of contention, enabling near-perfect distribution and parallel execution across DPUs.

The *NewOrder* transaction also achieves substantial gains, approaching 9× speedup at 2,048 DPUs. Although it introduces more data dependencies and write operations than *OrderStatus*, its entirely local execution in the absence of distributed transactions allows it to scale efficiently as DPU count increases.

In contrast, the *Payment* transaction improves more modestly, with performance gains remaining below 2× even at the largest DPU configuration. Its higher write intensity, combined with greater contention, diminishes the benefits of intra-DPU parallelism and amplifies coordination overhead, thereby limiting scalability.

These results highlight that PIM-based systems can deliver the largest performance gains in read-heavy and moderately

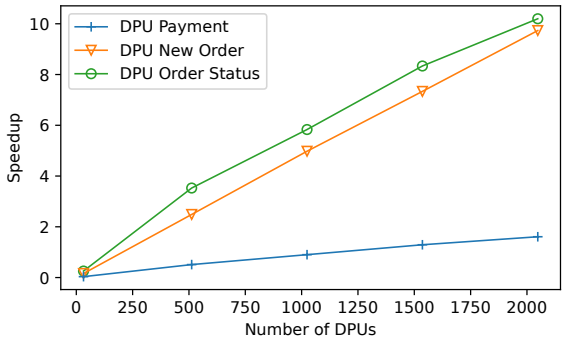


Figure 4. Speedup w.r.t. CPU for the *Payment*, *NewOrder* and *OrderStatus* transactions with multiple DPUs.

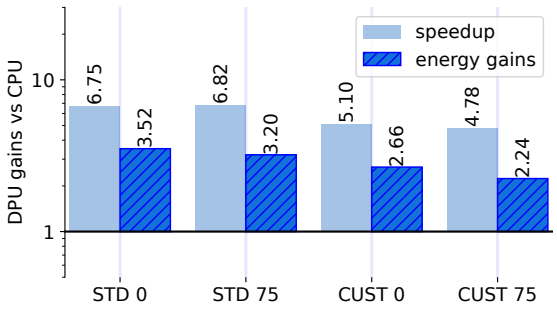


Figure 5. Speedup and energy gains for 2048 DPUs with respect to CPU-based implementation TPC-C.

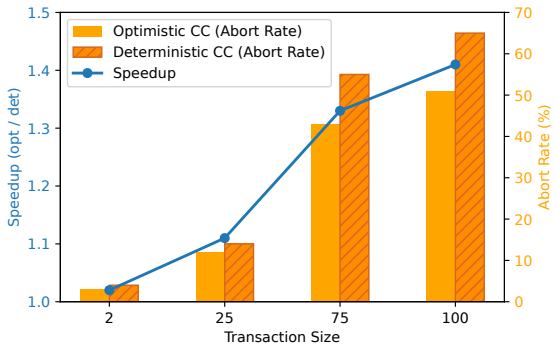


Figure 6. Speedup of optimistic over deterministic TM (blue, left y-axis) and abort rate (orange, right y-axis) for bank.

complex transactions. However, the degree of scalability can be strongly influenced by transaction characteristics such as contention levels and the balance of reads and writes.

4.2.2 Energy efficiency. Figure 5 presents the speedup and energy gain when using 2048 DPUs for varying workload configurations. Energy gains are determined as the ratio between the energy consumed by the CPU and the DPU. Due to the absence of energy counters on the UPMEM system, we estimate its energy consumption for a given workload by multiplying the thermal design power (TDP) (370W when utilizing all DPUs, as reported in [14]) by the workload's execution time. Conversely, for the CPU-based implementations, we measure the energy consumed both for the CPU and memory subsystems via the RAPL [9] library. Note that since the UPMEM estimation of energy consumption is an over-approximation, the reported energy gains represent a conservative lower bound on the actual gains.

In Figure 5 we observe that PIM-TIDE is able to achieve energy efficiency gains in all workload configurations. The observed energy gains range from 2.24× to 3.52×, reflecting the improvements achieved over the CPU baseline. Overall, the results demonstrate that the current UPMEM system offers strong performance potential. However, the energy gains are not totally on par with the performance gains.

4.3 Impact of determinism on concurrency control

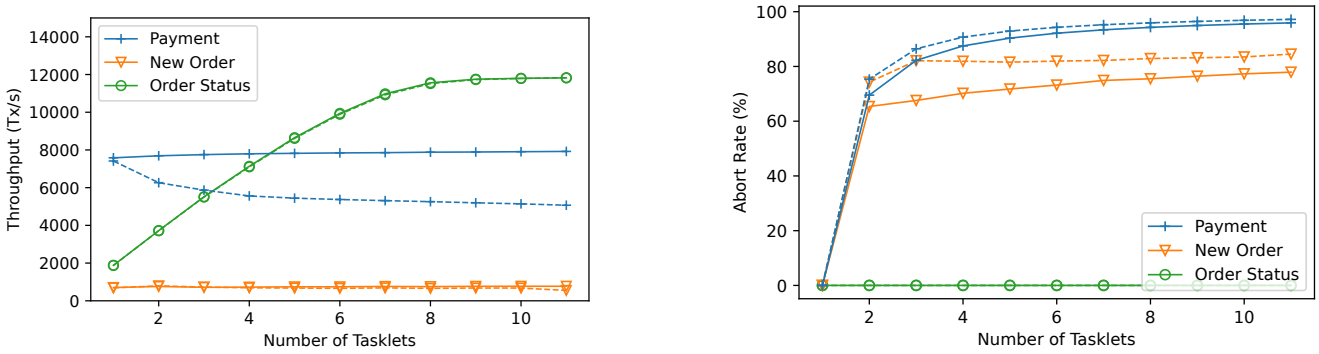
Figure 6 compares PIM-TIDE's deterministic and non-deterministic concurrency controls on a single DPU using the bank benchmark. We vary the number of bank transfers performed in each transaction on the x-axis, varying the corresponding contention level. The results show that the non-deterministic concurrency control consistently outperforms its deterministic counterpart when dealing with local transactions, achieving up to 1.5× speedup for largest transaction sizes. However, the advantage narrows as transaction sizes decrease, with speedups falling to approximately 1.1× for the smallest transactions.

The abort rate, shown as the orange bars against the right y-axis, remains low for smaller transactions but increases sharply as transaction sizes grow, reaching its highest levels at the largest transaction size. Deterministic execution generally exhibits higher abort rates than optimistic execution in these larger workloads. Interestingly, at smaller transaction sizes, deterministic execution experiences slightly less contention than its optimistic counterpart. Nonetheless, its lower throughput is primarily explained by the additional overhead of enforcing a strict transaction ordering, which forces transactions that are ready to commit to wait for the completion of every transaction that is serialized before it — which can severely hinder parallelism.

Figures 7a and 7b extend this study on intra-DPU parallelism to the TPC-C benchmark, showing throughput and abort rates for individual transaction profiles for the deterministic (dashed lines) and non-deterministic concurrency controls, as a function of the number of tasklets per DPU.

As shown in Figure 7a, *OrderStatus* achieves the highest throughput and scales nearly linearly with the number of tasklets. The two concurrency control schemes perform similarly with this transaction profile, which incurs negligible abort rates (see Figure 7b). This behavior is expected: *OrderStatus* is a read-only transaction with low contention probability (at least in all workloads tested), making it a good candidate for parallel execution.

The *Payment* transaction exhibits a markedly different pattern. Throughput remains relatively stable as tasklets increase with non-deterministic concurrency control, declining significantly with the deterministic approach. This degradation stems from higher contention and the overhead of enforcing a global commit order. Figure 7b presents abort rates exceeding 90% at higher tasklet counts for both modes, reflecting severe write contention. Non-deterministic execution partially mitigates this impact through rapid retries, whereas deterministic execution suffers more acutely from its stricter ordering requirements. These elevated contention levels can be attributed to two factors. 1) Each DPU processes



(a) Throughput

(b) Abort Rate

Figure 7. Throughput and abort rate for the *Payment*, *NewOrder* and *OrderStatus* transactions of the TPC-C benchmark in a single DPU. Solid lines represent optimistic concurrency control. Dashed lines represent deterministic concurrency control.

the subtransactions for a single warehouse and 2) every *Payment* transaction requires updating the balance of the target warehouse. As a consequence, all *Payment* transactions try to update, and contend on the same field.

Finally, the *NewOrder* transaction profile demonstrates limited intra-DPU scalability under both execution modes, as shown in Figure 7a. Abort rates exceed 80% across tasklet counts (Figure 7b), severely constraining progress. These results suggest that *NewOrder* is highly sensitive to intra-DPU contention (recall that each DPU stores a different warehouse in our implementation), and that neither concurrency control approach effectively handles its complex data dependencies under increased parallelism. This is due to the fact that the *NewOrder* transaction is relatively long, which increases the likelihood of conflicts. Furthermore, these transactions perform a large number of inserts, each of which requiring a significant number of writes.

In summary, non-deterministic execution is generally more resilient to contention and parallelism in write-heavy workloads, while the deterministic approach performs competitively only for read-mostly local transactions. Across all transactions, high abort rates remain the primary obstacle to single DPU throughput scaling.

5 Conclusions and Future Work

We presented PIM-TIDE, the first in-memory data store to support cross-DPU transactions on real PIM hardware. PIM-TIDE introduces a mixed-mode execution scheme that combines deterministic execution for distributed transactions with lightweight, non-deterministic concurrency control for local ones. We evaluated PIM-TIDE using a real-world OLTP benchmark ported to UPMEM-based PIM hardware, showing that PIM-TIDE can deliver up to a 10× performance improvement over traditional CPU-based systems, as well as energy efficiency gains, by leveraging massive parallelism and in-memory computation.

Overall, PIM-TIDE demonstrates the viability and benefits of bringing transactional processing to PIM environments today, while also opening several promising avenues for future research, such as efficient fault-tolerant schemes tailored for PIM systems and data partitioning schemes to reduce the frequency of distributed transactions.

As a final note, recently UPMEM has been taken over by a large corporation, which announced that the UPMEM technology will foster the development of their future PIM systems. We believe that this development will increase the industrial relevance of PIM systems and stimulate research on how to implement transactional processing systems that match the unique characteristics of PIM systems.

Acknowledgments

Work supported by national funds through Fundação para a Ciência e a Tecnologia, I.P. (FCT) under projects UID/50021/2025 (DOI: https://doi.org/10.54499/UID/50021/2025) and UID/PRR/50021/2025 (DOI: https://doi.org/10.54499/UID/PRR/50021/2025), through the Portuguese Recovery and Resilience Plan through project C645008882-00000055 (Center for Responsible AI) and through the EU's Horizon Europe research and innovation programme under Grant Agreement No 101189689 (ACHILLES project).

References

[1] Alexander Baumstark, Muhammad Attahir Jibril, and Kai-Uwe Sattler. 2023. Accelerating large table scan using processing-in-memory technology. *Datenbank-Spektrum* 23, 3 (2023), 199–209.

[2] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley.

[3] Robert L Bocchino, Vikram S Adve, and Bradford L Chamberlain. 2008. Software transactional memory for large scale clusters. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. 247–258.

[4] Joao Cachopo and António Rito-Silva. 2006. Versioned boxes as the basis for memory transactions. *Science of Computer Programming* 63, 2 (2006), 172–185.

[5] Daniel Castro, Alexandro Baldassin, João Barreto, and Paolo Romano. 2021. SPHT: Scalable Persistent Hardware Transactions. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 155–169.

[6] Daniel Castro, Paolo Romano, Aleksandar Ilic, and Amin M Khan. 2019. HeTM: transactional memory for heterogeneous systems. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 232–244.

[7] Maria Couceiro, Paolo Romano, Nuno Carvalho, and Luis Rodrigues. 2009. D2STM: Dependable distributed software transactional memory. In *2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*. IEEE, 307–313.

[8] Luke Dalessandro, Michael F Spear, and Michael L Scott. 2010. NOrec: Streamlining STM by abolishing ownership records. *ACM Sigplan Notices* 45, 5 (2010), 67–78.

[9] Howard David, Eugene Gorbatov, Ulf R Hanebutte, Rahul Khanna, and Christian Le. 2010. RAPL: Memory power estimation and capping. In *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*. 189–194.

[10] Safaa Diab, Amir Nassereldine, Mohammed Alser, Juan Gómez Luna, Onur Mutlu, and Izzat El Hajj. 2023. A framework for high-throughput sequence alignment using real processing-in-memory systems. *Bioinformatics* 39, 5 (2023), btad155.

[11] Dave Dice, Ori Shalev, and Nir Shavit. 2006. Transactional locking II. In *International Symposium on Distributed Computing*. Springer, 194–208.

[12] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. 2009. Stretching transactional memory. *ACM sigplan notices* 44, 6 (2009), 155–165.

[13] Jose M Faleiro and Daniel J Abadi. 2014. Rethinking serializable multiversion concurrency control. *arXiv preprint arXiv:1412.2324* (2014).

[14] Yann Falevoz and Julien Legriel. 2023. Energy Efficiency Impact of Processing in Memory: A Comprehensive Review of Workloads on the UPMEM Architecture. In *2023 International Congress on Power, Energy, and Computer Systems (PECS 2023)*.

[15] Yann Falevoz and Julien Legriel. 2023. Keynote: UPMEM PIM platform for Data-Intensive Applications. In *Minisymposium on Applications and Benefits of UPMEM commercial Massively Parallel Processing-In-Memory Platform (ABUPIMP), co-located with EUROPAR23*.

[16] Pascal Felber, Christof Fetzer, Rachid Guerraoui, and Tim Harris. 2008. Transactions are back—but are they the same? *ACM SIGACT News* 39, 1 (2008), 48–58.

[17] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. 2010. Time-based software transactional memory. *IEEE Transactions on Parallel and Distributed Systems* 21, 12 (2010), 1793–1807.

[18] Pascal Felber, Christof Fetzer, and Torvald Riegel. 2008. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. 237–246.

[19] Pascal Felber, Shady Issa, Alexander Matveev, and Paolo Romano. 2016. Hardware read-write lock elision. In *Proceedings of the Eleventh European Conference on Computer Systems* (London, United Kingdom) *(EuroSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 34, 15 pages. doi:10.1145/2901318.2901346

[20] Cesare Ferri, Samantha Wood, Tali Moreshet, R Iris Bahar, and Maurice Herlihy. 2010. Embedded-TM: Energy and complexity-effective hardware transactional memory for embedded multicore systems. *J. Parallel and Distrib. Comput.* 70, 10 (2010), 1042–1052.

[21] Saugata Ghose, Amirali Boroumand, Jeremie S Kim, Juan Gómez-Luna, and Onur Mutlu. 2019. Processing-in-memory: A workload-driven perspective. *IBM Journal of Research and Development* 63, 6 (2019), 3–1.

[22] Christina Giannoula, Peiming Yang, Ivan Fernandez, Jiacheng Yang, Sankeerth Durvasula, Yu Xin Li, Mohammad Sadrosadati, Juan Gomez Luna, Onur Mutlu, and Gennady Pekhimenko. 2024. PyGim: An Efficient Graph Neural Network Library for Real Processing-In-Memory Architectures. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 8, 3 (2024), 1–36.

[23] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F Oliveira, and Onur Mutlu. 2021. Benchmarking a new paradigm: An experimental analysis of a real processing-in-memory architecture. *arXiv preprint arXiv:2105.03814* (2021).

[24] Vincent Gramoli, Rachid Guerraoui, and Vasileios Trigonakis. 2012. TM2C: a software transactional memory for many-cores. In *Proceedings of the 7th ACM european conference on Computer Systems*. 351–364.

[25] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. 1996. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD international Conference on Management of Data*. 173–182.

[26] Rachid Guerraoui and Michal Kapalka. 2008. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. 175–184.

[27] Saransh Gupta, Mohsen Imani, Harveen Kaur, and Tajana Simunic Rosing. 2019. Nnpim: A processing in-memory architecture for neural network acceleration. *IEEE Trans. Comput.* 68, 9 (2019), 1325–1337.

[28] Saransh Gupta, Mohsen Imani, Behnam Khaleghi, Venkatesh Kumar, and Tajana Rosing. 2019. RAPID: A ReRAM processing in-memory architecture for DNA sequence alignment. In *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, 1–6.

[29] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. 2021. Benchmarking Memory-centric Computing Systems: Analysis of Real Processing-in-Memory Hardware. In *2021 12th International Green and Sustainable Computing Conference (IGSC)*. IEEE.

[30] Jayant R Haritsa, Krithi Ramamritham, and Ramesh Gupta. 2000. The PROMPT real-time commit protocol. *IEEE Transactions on Parallel and Distributed Systems* 11, 2 (2000), 160–181.

[31] Maurice Herlihy and J Eliot B Moss. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on Computer architecture*. 289–300.

[32] Shady Issa, Miguel Viegas, Pedro Raminhas, Nuno Machado, Miguel Matos, and Paolo Romano. 2020. Exploiting symbolic execution to accelerate deterministic databases. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 678–688.

[33] 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*. 1–14.

[34] Leslie Lamport. 2001. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001), 51–58.

[35] Donghun Lee, Jinin So, Minseon Ahn, Jong-Geon Lee, Jungmin Kim, Jeonghyeon Cho, Rebholz Oliver, Vishnu Charan Thummala, Ravi shankar JV, Sachin Suresh Upadhya, et al. 2022. Improving in-memory Database Operations with Acceleration DIMM (AxDIMM). In *Proceedings of the 18th International Workshop on Data Management on New Hardware*. 1–9.

[36] 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, Volume 2*. 879–896.

[37] Zhongmiao Li, Paolo Romano, and Peter Van Roy. 2019. Sparkle: Speculative deterministic concurrency control for partially replicated transactional stores. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 164–175.

[38] Zhongmiao Li, Peter Van Roy, and Paolo Romano. 2018. Transparent Speculation in Geo-Replicated Transactional Data Stores. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing* (Tempe, Arizona) *(HPDC '18)*. Association for Computing Machinery, New York, NY, USA, 255–266. doi:10.1145/3208040.3208055

[39] André Lopes, Daniel Castro, and Paolo Romano. 2024. PIM-STM: Software transactional memory for processing-in-memory systems. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 897–911.

[40] Sparsh Mittal. 2018. A survey of ReRAM-based architectures for processing-in-memory and neural networks. *Machine learning and knowledge extraction* 1, 1 (2018), 75–114.

[41] 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*. 1062–1071.

[42] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. 2023. *A Modern Primer on Processing in Memory*. Springer Nature Singapore, Singapore, 171–243. doi:10.1007/978-981-16-7487-7_7

[43] Jacob Nelson and Roberto Palmieri. 2019. Don't forget about synchronization! a case study of k-means on gpu. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*. 11–20.

[44] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2014. Deterministic galois: on-demand, portable and parameterless. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 499–512. doi:10.1145/2541940.2541964

[45] Diogo Nunes, Daniel Castro, and Paolo Romano. 2023. CSMV: A highly scalable multi-versioned software transactional memory for GPUs. *J. Parallel and Distrib. Comput.* (2023), 104701.

[46] Fernando Pedone, Rachid Guerraoui, and André Schiper. 2003. The database state machine approach. *Distributed and Parallel Databases* 14, 1 (2003), 71–98.

[47] Sebastiano Peluso, Joao Fernandes, Paolo Romano, Francesco Quaglia, and Luís Rodrigues. 2012. SPECULA: Speculative replication of software transactional memory. In *2012 IEEE 31st Symposium on Reliable Distributed Systems*. IEEE, 91–100.

[48] Xiaodong Qi, Jiao Jiao, and Yi Li. 2023. Smart Contract Parallel Execution with Fine-Grained State Accesses. In *2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 841–852.

[49] Shujian Qian and Ashvin Goel. 2024. Massively Parallel Multi-Versioned Transaction Processing. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 765–781. https://www.usenix.org/conference/osdi24/presentation/qian

[50] Moinuddin K Qureshi, M Aater Suleman, and Yale N Patt. 2007. Line distillation: Increasing cache capacity by filtering unused words in cache lines. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE, 250–259.

[51] Kaushik Ravichandran, Ada Gavrilovska, and Santosh Pande. 2014. DeSTM: harnessing determinism in STMs for application development. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. 213–224.

[52] 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* (Long Beach, CA, USA) *(PACT '24)*. Association for Computing Machinery, New York, NY, USA, 201–218. doi:10.1145/3656019.3676947

[53] Nir Shavit and Dan Touitou. 1997. Software transactional memory. *Distributed Computing* 10, 2 (1997), 99–116.

[54] Akhil Shekar, Kevin Gaffney, Martin Prammer, Khyati Kiyawat, Lingxi Wu, Helena Caminal, Zhenxing Fan, Yimin Gao, Ashish Venkat, José F Martínez, et al. 2025. Membrane: Accelerating Database Analytics with Bank-Level DRAM-PIM Filtering. *arXiv preprint arXiv:2504.06473* (2025).

[55] 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* (Vienna, Austria) *(VLDB '07)*. VLDB Endowment, 1150–1160.

[56] 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 *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 1–12.

[57] Transaction Processing Performance Council (TPC). 2011. *TPC Benchmark™ C Standard Specification*. Technical Report Version 5.11. Transaction Processing Performance Council. https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf Available at https://www.tpc.org/tpcc/.

[58] UPMEM. 2023. UPMEM — UPMEM is releasing a true Processing-in-Memory (PIM) acceleration solution. https://www.upmem.com/. Accessed: 2023-08-02.

[59] Yu Xia, Xiangyao Yu, William Moses, Julian Shun, and Srinivas Devadas. 2019. LiTM: A Lightweight Deterministic Software Transactional Memory System. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*. 1–10.

[60] Sheng Xu, Chun Li, Le Luo, Ming Zheng, Liang Yan, Xingqi Zou, and Xiaoming Chen. 2025. Balancing Graph Processing Workloads in Heterogeneous CPU-PIM System. *IEEE Transactions on Emerging Topics in Computing* (2025), 1–14. doi:10.1109/TETC.2025.3563249

[61] Xuanhe Zhou, Guoliang Li, Jianhua Feng, Luyang Liu, and Wei Guo. 2023. Grep: A graph learning based database partitioning system. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–24.