Massively Parallel Multi-Versioned Transaction Processing

Shujian Qian and Ashvin Goel, *University of Toronto*

https://www.usenix.org/conference/osdi24/presentation/qian

This paper is included in the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-40-3

Open access to the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

Massively Parallel Multi-Versioned Transaction Processing

Shujian Qian
University of Toronto

Ashvin Goel
University of Toronto

Abstract

Multi-version concurrency control can avoid most read-write conflicts in OLTP workloads. However, multi-versioned systems often have higher complexity and overheads compared to single-versioned systems due to the need for allocating, searching and garbage collecting versions. Consequently, single-versioned systems can often dramatically outperform multi-versioned systems.

We introduce Epic, the first multi-versioned GPU-based deterministic OLTP database. Epic utilizes a batched execution scheme, performing concurrency control initialization for a batch of transactions before executing the transactions deterministically. By leveraging the predetermined ordering of transactions, Epic eliminates version search entirely and significantly reduces version allocation and garbage collection overheads. Our approach utilizes the computational power of the GPU architecture to accelerate Epic's concurrency control initialization and efficiently parallelize batched transaction execution, while ensuring low latency. Our evaluation demonstrates that Epic achieves comparable performance under low contention and consistently higher performance under medium to high contention versus state-of-the-art single and multi-versioned systems.

1 Introduction

There has been a growing need for high-throughput online transaction processing (OLTP) systems capable of executing tens of thousands of transactions per second. In-memory database systems, specifically designed for workloads with datasets that fit entirely in DRAM memory and provide durability and high availability via logging and replication, have been developed to address this demand. Although these systems offer considerable performance advantages over traditional disk-based systems, they suffer under contention, leading to low performance and limited scalability across cores.

Multi-versioning offers a promising solution for contended and read-heavy workloads. Multi-version systems maintain recent past versions of each record, enabling concurrent reads and writes to the same record; reads do not block writes because writes can safely create new versions while reads are accessing the old versions. Consequently, transactions can be serialized in ways unattainable in single-version designs, thereby enabling greater parallelism. Previous work has shown that multi-version systems can outperform single-version systems under high contention [17].

However, current multi-version designs have several drawbacks, including increased overheads during transaction processing, data storage, allocation and garbage collection. These designs store record versions in linked lists, introducing an additional layer of indirection and necessitating list traversal to locate the appropriate version. Accessing the versions results in a larger working set, leading to higher cache miss rates and performance degradation. Multiple versions also lead to higher memory requirements. To reduce the memory footprint, versions are frequently garbage collected, which incurs additional overheads. As a result, a previous study that compared carefully tuned, state-of-the-art multi-version and single-version systems demonstrated that under low contention, a multi-version system has roughly *half* the throughput of single-version systems [14].

Current multi-version designs allocate versions dynamically because transactions may write and thus create versions at any time. Thus, versions are stored in linked lists, reads require searching for versions, and garbage collecting versions has poor locality and requires expensive synchronization.

Our key insight is that deterministic databases employing transaction batching and known transaction read-write sets can avoid most of these multi-versioning costs, thus enabling good performance for all workloads. The transaction batching and known read-write sets requirements are commonly met by most deterministic databases [11, 12, 18, 19, 26, 28, 31, 35].

We introduce Epic, the first multi-versioned, GPU-based deterministic transaction-processing database. Epic batches transactions into epochs and establishes a serial ordering of transactions within a batch before transaction execution, similar to other deterministic databases.

Transaction batching enables splitting an epoch into an *initialization phase* during which concurrency control operations are initialized using the read-write sets, followed by an *execution phase* during which transactions are executed concurrently and synchronized to ensure the deterministic ordering. During the initialization phase, Epic allocates versions based on the write set. These allocation operations are performed efficiently because they do not interfere with transaction execution. In addition, Epic calculates the version location of each read/write operation based on the ordering of transactions and the known read-write sets. This approach enables transactions to access versions directly during the execution phase, without requiring any version search.

Epic's epoch-based design enables efficient garbage collection as well. Since transactions in the next epoch are serialized

after all transactions in the current epoch, only the final write to a record is visible to the transactions in the next epoch. Thus all versions except the last one become obsolete when an epoch ends. Epic stores all intermediate record versions separately from the last version and reclaims them efficiently at the end of an epoch.

The challenge is that Epic's initialization phase is expensive, requiring both significant computation and memory bandwidth. Fortunately, Epic's initialization phase is highly parallelizable. With the rapid commoditization of general-purpose GPU computing, Epic harnesses the thread parallelism offered by modern GPU architectures to significantly accelerate the initialization phase.

Modern GPUs are well suited for Epic's execution phase as well because they offer high-bandwidth memory for memory-bound workloads. In addition, they perform zero-overhead context switching between thread contexts, which allows hiding memory access latency. These advantages help to counter the increased memory footprint and lower cache utilization commonly associated with multi-version systems. Consequently, Epic achieves high throughput and ensures low transaction latency even with its epoch-based execution scheme.

While GPU transaction execution performs well, it is limited by datasets that fit in GPU memory. Thus, Epic also supports larger datasets with a CPU execution model in which the initialization phase runs on the GPU while the execution phase runs on the CPU.

To demonstrate the effectiveness of Epic's design, we conduct extensive evaluation using the TPC-C and YCSB benchmarks and show that Epic significantly outperforms recent single- and multi-version systems on most workloads.

2 Background

This work builds on a rich body of research on multi-version concurrency control, deterministic databases, and GPU-accelerated computation, as discussed below.

2.1 Multi-versioned Concurrency Control

Multi-version concurrency control (MVCC) has a long history [29, 30], with early work evaluating its performance [8], ensuring snapshot isolation [5], providing serializable snapshot isolation [7], using dynamic timestamp assignment [20] and enabling efficient indexing [32], for disk-based databases.

With the advent of machines equipped with high core counts and terabytes of DRAM memory, much work has focused on in-memory database designs, and several MVCC schemes optimized for them have been proposed [15, 16, 22]. MVCC schemes are popular because they provide robust performance under a wide range of workloads. As a result, many commercial in-memory databases implement MVCC [10, 24, 25, 34].

Wu et al. conduct a detailed study of the costs associated with concurrency control, version storage, garbage collection, and index management in various in-memory MVCC schemes [37]. Cicada [17] outperforms previous MVCC schemes with several optimizations, including optimistic multi-versioning, contention regulation, version inlining, and rapid garbage collection. However, a study comparing state-of-the art multi-version and single-version systems showed that while MVCC outperforms OCC under high contention, its throughput is significantly lower under low contention [14]. Epic aims to minimize multi-versioning costs associated with version storage, lookup and garbage collection.

2.2 Deterministic Database Systems

Deterministic databases have gained increasing attention in recent years, driven by the need for efficient replication and improved scalability for distributed transactions [35]. These systems execute transactions deterministically by ensuring that the serial ordering of operations remains consistent across different runs. Determinism enables efficient replication [27, 31, 33] and live migration [18, 19] since all replicas execute transactions independently without coordination. Furthermore, deterministic systems reduce the need for two-phase commit, helping scale the performance of distributed transactions [35]. They can also effectively handle skewed and contended accesses, e.g., orders for popular items [28].

Deterministic systems typically batch transactions into epochs to perform deterministic concurrency control before execution [11, 12, 28, 35]. Thus these system require the read and write sets of transactions to be known before execution. When they are not fully known, they can be determined using reconnaissance queries [35]. Calvin [35] and PWV [12] are single versioned, while Bohm [11] and Caracal [28] utilize MVCC. Calvin uses a centralized lock manager, while PWV employs a more-scalable per-core dependency analysis for concurrency control. Bohm and Caracal allocate versions scalably during the concurrency control initialization phase, but Bohm performs partitioned initialization, while Caracal performs shared memory initialization. Bohm partitions the records in a table across cores. During the initialization phase, all partitions analyze each transaction's write set and insert placeholder versions in a linked list for the records they own. During execution, a read operation traverses the list to find the correct version based on its total order ID. Then, it synchronizes with a write operation that fills the corresponding placeholder version. Caracal uses shared-memory initialization, which enables better handling of skewed workloads. It scales version allocation for contended records by batching the allocations. It stores versions as sorted arrays and uses binary search to reduce version lookup costs during execution. Epic performs shared-memory initialization similar to Caracal. However, Epic avoids any version lookup costs and minimizes version storage and garbage collection overheads.

2.3 GPU Accelerated OLTP Databases

General-Purpose computing on Graphics Processing Units (GPGPU) has become popular with the rapid commoditization of GPUs, the advent of user-friendly programming models and frameworks like CUDA and OpenCL, and the growing demand for high-performance computing on large datasets. Modern GPUs contain an array of streaming multiprocessors (SMs), each of which contains many CUDA cores or stream processors, allowing execution of thousands of active threads concurrently. GPUs uses the Single Program, Multiple Data (SPMD) parallel programming model in which multiple threads execute the same program on different data elements.

GPU-based databases are an active area of research, but most work has focused on accelerating Online Analytical Processing (OLAP) workloads since typical OLAP operators, such as join and sort, are a good fit for parallelization using the GPU's SPMD execution model.

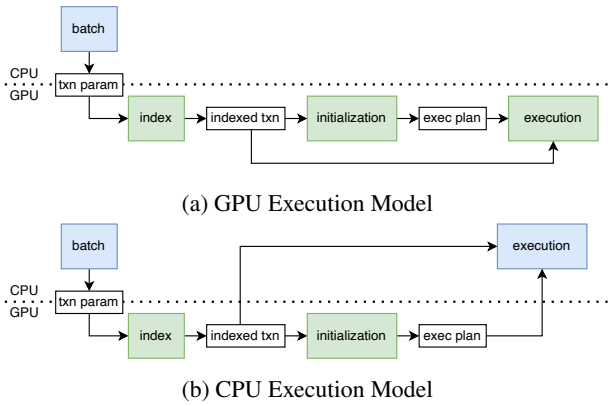
GPU-based transaction processing is relatively unexplored because transactional workloads comprise short-lived transactions with random accesses, and atomicity and isolation require significant synchronization. These requirements makes it hard to exploit the parallelism available in GPUs.

Previously, two GPU-based transaction processing systems, GPUTx [13] and GaccO [6], have been proposed. Similar to Epic, both batch transactions and use epoch-based concurrency control initialization and execution. GPUTx, an early attempt at executing OLTP workloads on GPUs, uses dependency tracking to group transactions into sets; transactions within each set are conflict-free and can execute without synchronization. However, we found that their efficient dependency tracking algorithm, K-Set, does not ensure that transactions in a set are conflict-free, thereby failing to guarantee correctness. GaccO is a deterministic database that uses single-version, deterministic locking, similar to Calvin. We describe Gacco in detail and compare it with Epic in Section 5.

3 Design

Epic is a GPU-accelerated, in-memory deterministic database that employs a novel multi-versioned concurrency control protocol. Epic assumes that transactions are one-shot and use stored procedures, similar to other high-performance in-memory databases [36].

Figure 1 shows the Epic architecture. Epic batches transactions into epochs and splits each epoch into indexing, initialization and execution phases. The transaction inputs, consisting of read-set and write-set keys and other transaction data, are batched on the CPU and then transferred to the GPU for indexing (shown as "txn param" in Figure 1). During indexing, the keys are used to retrieve and store the corresponding record IDs in a per-transaction data structure (shown as "indexed txn" in Figure 1). These record IDs are used during initialization and used as indices for accessing the record tables during transaction execution. The initialization phase performs multi-versioned concurrency control and generates a per-transaction execution plan, which consists of the locations of the record versions that a transaction then directly accesses during execution.



(a) GPU Execution Model

(b) CPU Execution Model

Figure 1: Epic Architecture

While indexing and initialization are always run on the GPU, Epic can execute transactions on the GPU (Figure 1a) or the CPU (Figure 1b). CPU execution is used to support databases larger than GPU memory. In this case, the GPU serves as an accelerator for indexing and initialization.

Sometimes a transaction's read and write sets are not fully known before the indexing phase. For example the TPC-C order-status transaction requires a secondary index to locate a customer's latest order. For these transactions, Epic runs an optional read-write set identification phase on the GPU before the indexing phase. The transaction inputs to the identification phase only contain the read-write keys that are known at transaction generation time. This phase runs reconnaissance queries [35] that use these partial transaction inputs to identify the remaining read-write keys.

The following sections describe Epic's storage scheme and then Epic's initialization and execution phases.

3.1 Epic storage scheme

Epic's storage scheme separates temporary versions created within an epoch from versions that exist across epochs. All writes to a record within an epoch, except the last one, are only read by other transactions within the epoch. This is because transactions from a later epoch are serialized after all transactions in the current epoch and thus can only read the last version of each record. We call the versions that are read by transactions within an epoch *temporary versions*. The final write to a record within an epoch may be read in later epochs and so this last version is saved across epochs.

Figure 2 shows an example of Epic's storage scheme with transactions T1 to T8 reading and writing Record 1 at Epoch 3. Epic places all temporary versions in a scratchpad area. During an epoch, the write transactions on a record, except

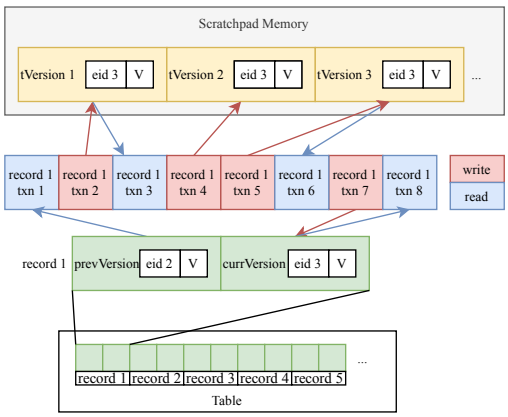


Figure 2: Epic Storage Scheme

the last one, fill these versions, and reads synchronize with the writes to ensure RAW dependencies are satisfied. At the end of an epoch, when all the reads for the temporary versions are done, the scratchpad is reclaimed and used in the next epoch, completely eliminating per-version garbage collection.

The final versions of each record are placed in a dense table area and do not require garbage collection. The last write in an epoch to each record updates the value in the table directly, leading to potential race conditions when transactions in the current epoch need to read data from the previous epoch. Epic addresses this problem by storing two versions for each record in the table: the previous version (*prevVer*) and the current version (*currVer*), as shown in Figure 2. In each epoch, *prevVer* holds the data from the previous epoch (original version). The last write (Txn 7) within an epoch updates *currVer* to avoid overwriting the original version. This write is performed directly on the table, so all temporary versions can be easily collected after an epoch. Reads after the last write to a record (Txn 8) read from *currVer*.

The locations of *prevVer* and *currVer* in each record depend on transaction history, as their positions only change when a record is written during an epoch. Therefore, Epic stores an epoch ID in each version, which helps distinguish the version from previous epochs (*prevVer*) from the version that should be updated in the current epoch (*currVer*). Algorithm 1 is used by transactions to distinguish between *prevVer* and *currVer*. In an epoch, before any write has happened to *currVer*, Epic ensures that the version with a larger epoch ID contains the more up-to-date value and should be used as *prevVer* (Lines 9–12). During the last write, the writer will update the epoch ID of *currVer* to the current epoch's ID (*current_eid*), after which *currVer* will have a larger epoch ID, but it is still distinguishable since its epoch ID matches the current epoch ID (Lines 4–7). The epoch ID is also used for synchronization between reads and writes, as discussed later in Section 3.3.

The record tables and the scratchpad memory are stored in GPU memory for the GPU execution model and in CPU memory for the CPU execution model.

Algorithm 1: Determining the *prevVer* and *currVer*

```
   // Takes the two table versions of a record
1  Function GetTableVersions (V[2]):
2      eid0 ← atomicRead(V[0].eid)
3      eid1 ← atomicRead(V[1].eid)
       // current_eid is the current epoch's ID
4      if eid0 = current_eid then
5          prevVer ← V[1]; currVer ← V[0]
6      else if eid1 = current_eid then
7          prevVer ← V[0]; currVer ← V[1]
8      else
9          if eid0 > eid1 then
10             prevVer ← V[0]; currVer ← V[1]
11         else
12             prevVer ← V[1]; currVer ← V[0]
13     return {prevVer, currVer}
```

3.2 Multi-Version Initialization

During the initialization phase, Epic uses the ordering of transactions and the knowledge of their read-write sets to allocate versions for all writes performed in the epoch. To avoid the expensive version search required in previous multi-versioned systems, Epic calculates the read-write version locations for each transaction in the epoch before any transactions execute. These operations are parallelizable because they are performed in a phase separate from transaction execution.

As shown in Algorithm 2, Epic employs a parallel GPU-based algorithm to perform concurrency control initialization efficiently. Figure 3 provides an example of this algorithm. The initialization phase starts by collecting all the read and write operations within the epoch (Step 1). Each entry in the *all_ops* operations array contains the *record_id* and the *txn_id* associated with the operation, the operation's index within the transaction (*op_id*), and the operation type (read/write). This operation is parallelizable because the order of operations does not matter for the next step, which sorts the operations array by *record_id* and *txn_id* (Step 2).

Then, Epic counts the number of write operations to each record that occur before and after each operation. Since the operations are already grouped by *record_id*, these operations use parallel prefix and postfix sum by key (Steps 3–4). Next, *GetOpType* in Algorithm 3 calculates the read-write location type for each operation (Step 5). A write operation writes to *currVer* for the last write to the record or else to *tempVer*. A read operation will read from the version written by the previous write as follows: *prevVer* if there is no preceding write preceding, *currVer* if there is no succeeding write, and *tempVer* otherwise.

The number of *tempVer* variables created in an epoch is equal to the number of *tempVer* writes. Thus, Epic places the *tempVer* variables in the scratchpad area in the same order as the *tempVer* write operations in the sorted operations array. To calculate the *tempVer* locations, Epic performs a parallel prefix sum over all operations, counting *tempVer* writes before each operation (Step 6). With this information,

Algorithm 2: Multi-Version Initialization Phase

```
1  Function Initialize (txns[NUM_TXN]):
2      all_ops  // all read-write operations in the epoch,
       // contains tuples: {record_id, txn_id, op_id, read_write}
       // All local variables are arrays of size equal to all_ops size

       // Step 1: submit operations
3      parallel foreach txn ∈ txns do
4          op_id = 0
5          foreach record_id ∈ txn.read_record_ids do
6              op_id++
7              all_ops.pushback({record_id, txn.id, op_id, Read})
8          foreach record_id ∈ txn.write_record_ids do
9              op_id++
10             all_ops.pushback({record_id, txn.id, op_id, Write})
       // Step 2: sort first by record_id then by txn_id
11     sorted_ops = Sort(all_ops, key = {record_id, txn_id})
       // Steps 3-4: count writes before/after each op on same record
12     writes_before = PrefixSumByKey(sorted_ops,
13                                    key = record_id,
14                                    value = Write ? 1 : 0)
15     writes_after = PostfixSumByKey(sorted_ops, key = record_id,
16                                    value = Write ? 1 : 0)
       // Step 5: get operation type, can be:
       // prevVer read, currVer read/write, tempVer read/write
17     op_types = GetOpType(sorted_ops, writes_before,
18                          writes_after)
       // Step 6: count tempVer writes before each op in the epoch
19     tw_before = PrefixSum(op_types, value=tempVerWrite ? 1 : 0)
       // Step 7: get read/write location for all ops
20     rw_loc = GetRWLocation(op_types, tw_before)
       // Step 8: scatter rw_loc back to transactions
21     parallel for i = 0 to sorted_ops.size do
22         txn_id = sorted_ops[i].txn_id
23         op_id = sorted_ops[i].op_id
24         txns[txn_id].locations[op_id] = rw_loc[i]
```

GetRWLocation in Algorithm 4 calculates the read-write locations for all operations (Step 7). The *i*th *tempVer* write updates the *i*th *tempVer* in the scratchpad area. A read from *tempVer* reads the previous write in the sorted operations array. Finally, the read-write locations are scattered back to each transaction to be used in the execution phase (Step 8).

3.3 Transaction Execution

Epic's execution phase is considerably simpler than the initialization phase. A transaction accesses versions directly using the locations calculated during initialization, eliminating any version lookup during execution. Due to multi-versioning, write-after-read (WAR) and write-after-write (WAW) dependencies do not require explicit coordination. Epic uses the epoch ID associated with each version to synchronize read-after-write (RAW) dependencies between transactions.

Algorithm 5 shows Epic's transaction execution phase. The transactions in an epoch are scheduled in their predetermined serial order as thread resources become available, as explained further in Section 4.4. The *RunTxn* function shows an example of a transaction. The transaction accesses the versions

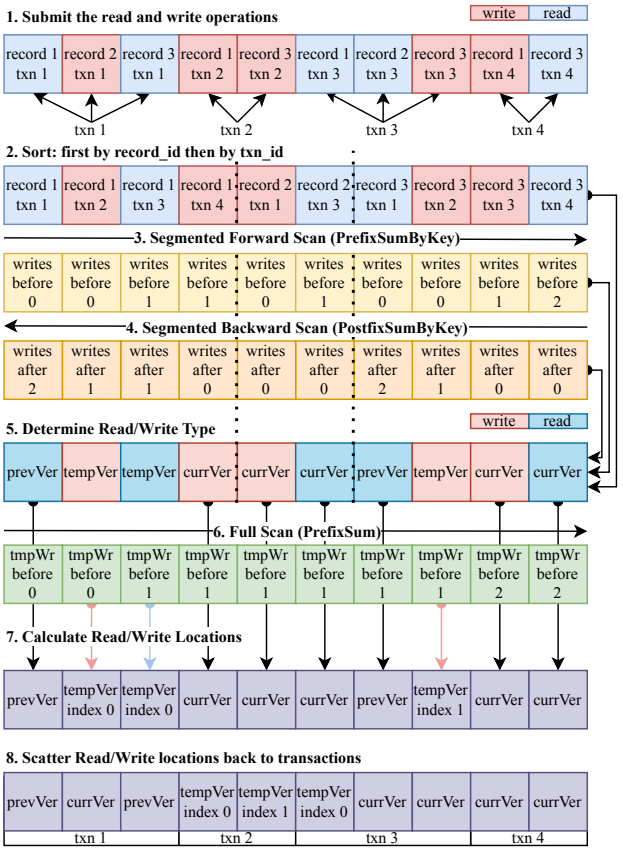


Figure 3: Example of Epic's Initialization Phase

directly using the location information calculated in the initialization phase (lines 11–25 for reads and lines 26–34 for writes). A transaction read waits for a version to be written by an earlier transaction by spinning on the epoch ID of the version until it matches the current epoch ID (lines 21–22). However, reads from *prevVer* do not need any synchronization since this version was updated in a previous epoch. A transaction writes to the data of the version before updating the version's epoch ID (lines 32–34). The GPU weak memory consistency model requires a memory fence between the data write and the epoch ID update to ensure that the data is visible to other threads before the updated version.

CPU-side Execution Epic can also execute transactions on the CPU, which is particularly useful when the database size exceeds GPU memory capacity. In this case, Epic transfers the output of indexing (read and write record IDs) and the initialization phase (read-write locations) to the CPU, as shown in Figure 1. CPU-side execution utilizes the same synchronization mechanism as GPU execution.

Handling Inserts and Deletes Epic treats record insertions and deletions the same way as updates. Both insert and delete operations are considered write operations, so they create a new version of the record, similar to an update. For each version, Epic uses a valid flag to mark whether it contains

Algorithm 3: Calculate Read/Write Type

```
1  Function GetOpType (sorted_ops, writes_before, writes_after):
2      op_types[sorted_ops.size] // type of operations
3      parallel for i = 0 to sorted_ops.size do
4          if sorted_ops[i].read_write == Write then
5              if writes_after[i] == 0 then
6                  op_types[i] = currVerWrite
7              else
8                  op_types[i] = tempVerWrite
9          else // read operation
10             if writes_before[i] == 0 then
11                 op_types[i] = prevVerRead
12             else if writes_after[i] == 0 then
13                 op_types[i] = currVerRead
14             else
15                 op_types[i] = tempVerRead
16     return op_types
```

Algorithm 4: Calculate Read/Write Locations

```
1  Function GetRWLocation (op_types, tw_before):
2      rw_loc[op_types.size] // locations of read/write operations
3      parallel for i = 0 to sorted_ops.size do
4          if op_types[i] ∈ {currVerRead, currVerWrite} then
5              rw_loc[i] = currVer
6          else if op_types[i] == prevVerRead then
7              rw_loc[i] = prevVer
8          else // tempVer read/write, return tempVer index
9              if op_types[i] == tempVerRead then
                   // index is zero-based
10                 rw_loc[i] = {tempVer, index=tw_before[i]−1}
11             else
12                 rw_loc[i] = {tempVer, index=tw_before[i]}
13     return rw_loc
```

valid (V) data, as shown in Figure 2. An update or insert sets and a delete unsets the valid flag of the corresponding version. Read operations use the valid flag to determine if the record exists at the timestamp of the read, preventing transactions from reading invalid data (Algorithm 5, lines 23–24).

Deletion of records can happen at any point within an epoch, and a later write operation to a deleted record will re-insert it. Consequently, the record should be freed only when the last write operation to a record in an epoch is a delete. Epic tracks records that are deleted in an epoch by setting a per-record deleted flag when deletions occur to *currVer*. At the end of the epoch, these flags are scanned to generate a list of deleted records that are subsequently freed, as described later in Section 4.2. A full scan after each epoch is acceptable because the flag is one bit per record and parallel scans are efficient on GPUs.

Handling Aborts Epic eliminates concurrency-control related aborts because transactions are serialized in a predetermined order, similar to other deterministic databases. Epic allows application-level aborts (e.g., constraint violations) before any writes are performed to the database. Transactions

Algorithm 5: Transaction Execution Phase

```
1  Function Execute (txns[NUM_TXN]):
2      parallel for i = 0 to txns.size do
3          RunTxn(txns[i])

4  Function RunTxn (txn):
5      value1 = ReadFromTable(txn.record_id1, txn.read_loc1)
6      value2 = ReadFromTable(txn.record_id2, txn.read_loc2)
       // perform transaction logic
7      if value1 is None or value2 is None then
8          abort()
9      result = SomeOperation(value1, value2)
       // no aborts can happen beyond this point
10     WriteToTable(txn.result_record_id, txn.write_loc, result)

11 Function ReadFromTable (rec_id, read_loc):
12     if rec_id = INVALID_RECORD then
13         return None
14     prevVer, currVer = GetTableVersions(table[rec_id])
15     if read_loc == prevVer then
16         read_ver = prevVer
17     else if read_loc == currVer then
18         read_ver = currVer
19     else // tempVer read
20         read_ver = tempVers[read_loc.index]
21     while read_loc ≠ prevVer and
               atomicRead(read_ver.eid) ≠ current_eid do
22         Spin() // Wait until version is ready
23     if not read_ver.is_valid then
24         return None
25     return read_ver.data

26 Function writeToTable (rec_id, write_loc, data):
27     prevVer, currVer = GetTableVersions(table[rec_id])
28     if write_loc == currVer then
29         write_ver = currVer
30     else // tempVer write
31         write_ver = tempVers[write_loc.index]
32     PerformWrite(write_ver.data, data)
33     __threadfence()
34     atomicWrite(write_ver.eid, current_eid)
```

are expected to perform their reads, buffer writes and issue aborts before any database writes. Since aborts do not occur after the first write, the writes of a transaction are made visible immediately [12].

In previous multi-versioned systems, a sentinel value is used to indicate an aborted version. Subsequent reads skip such versions and read the previous non-aborted version. This approach is not suitable for Epic since there is no version search. Instead, the aborted write operations must copy the previous version to the current version. Thus, for transactions that may abort, Epic also calculates the read location (i.e., of the previous version) for write operations during initialization.

3.4 Field Splitting

Database records often consist of multiple fields. Since Epic eliminates version search, each version of a record must con-

tain a full copy of all of its fields. This approach adds copying overhead when a transaction updates only a few fields of a record since all of its fields must be copied from the previous version. In addition, it introduces unnecessary dependencies because every field update becomes a read-modify-write operation for the record.

Epic implements a field splitting optimization by storing different fields of a record separately. Each version now comprises only a single field. As a result, a write to a field does not require copying other fields and introduces no additional dependencies. However, the field splitting optimization adds overhead for full record operations, which need to be split into multiple per-field operations, leading to increased initialization and synchronization costs.

3.5 Recovery

Currently, Epic does not support recovery and replication. However, it can provide durability and high availability by using techniques similar to previous deterministic databases [35]. In each epoch, transaction inputs can be logged to storage on the CPU side concurrently with transaction execution. Once all inputs are logged, transaction results can be made externally visible to applications. Currently, Epic returns these results conservatively at the end of the epoch, which enables handling certain problematic transaction logic, such as infinite loops, by aborting the relevant transaction and its dependent transactions [12].

For recovery, the transaction inputs are used to replay all transactions deterministically until the last logged epoch. The replay uses the same mechanism as normal transaction processing. To reduce recovery time, Epic's two-version tables allow checkpoints to be created efficiently. The checkpointing process can run in parallel with an epoch and create a consistent database snapshot by copying the *prevVer* of each record to a different memory area (e.g., CPU memory). However, the next epoch must start after the checkpointing completes or else the resulting snapshot may be inconsistent. After creating a copy of the tables, they can be transferred to persistent storage in the background. The index and allocation information also needs to be checkpointed or rebuilt during recovery.

4 Implementation

This section describes Epic's GPU-based implementation of indexing, initialization and transaction execution phases.

4.1 Transaction Batching and Ordering

Currently, Epic batches transactions when they are generated and serially orders them by assigning a transaction ID to each transaction. In practice, the batching and ordering process can be performed without contention by batching transactions separately on each core and ordering them using a local counter. Before an epoch starts, transactions from all cores can be serialized based on the core ID and the local counter value. This method is similar to Calvin [35].

4.2 Indexing and Allocation

Epic is capable of executing tens of millions of transactions per second. Its index needs to handle hundreds of millions of operations per second, and so we use GPU-based indexing. Epic uses a hash table index to map keys to record IDs. When needed, range queries are performed in the read-write set identification phase using a range index to obtain all the keys for the read and write sets. The keys are then used to look up the record IDs in the hash table index. Epic implements indexing using CuCollection [23], a GPU-based concurrent hash table. Epic uses a modified version of a GPU B-tree [2,4] for the range index.

Since Epic's indexing operates in parallel, we ensure that read operations see all previously inserted records by performing insert operations before any indexing operations, which also prevents phantom reads. Epic does not distinguish between insert and write operations, and so it first indexes all write operations in an epoch to find the keys to be inserted (keys that are in the write set but are not found in the hash table). To allocate a record for each to-be-inserted key, Epic maintains a ring buffer of free record IDs on the GPU. To ease allocation, these keys are uniquified. Then, Epic allocates record IDs for them by removing the same number of record IDs from the ring buffer. The key-record ID mappings are then inserted in the hash table. Next, Epic indexes all read and write operations. For read operations, if a key is not found, Epic marks the read as invalid by returning a sentinel `invalid_read` value for the record ID. This value is treated as any other record ID during initialization, and then reads detect it during execution (Algorithm 5, lines 12–13). Since Epic performs inserts before read operations, a read of a non-existing record may see an index entry from a later write. A read operation detects this version as invalid during execution (see Section 3.3).

At the end of an epoch, Epic's execution phase returns the deleted record IDs (see Section 3.3). Epic garbage collects these records by appending them to the ring buffer. To free the index entries for these records, Epic also keeps a back-link array that maps record IDs to keys. The hash table and the back-link are stored in GPU memory and are only accessed by the GPU during indexing.

4.3 Multi-Version Initialization

Epic's multi-version concurrency control initialization is implemented using the CUB and Thrust parallel algorithms library. As shown in Algorithm 2, all operations, such as sorting and prefix sum, are highly parallelizable. Epic performs initialization for each table separately for ease of implementa-

tion. Each operation's record ID, transaction ID, operation ID and read-write type are stored in a 64 bit integer for efficient sorting. It is possible to prefix the record ID with a table ID and perform initialization for all tables together.

We implemented an optimized CPU-based initialization phase using Intel's TBB library but its performance was at least an order of magnitude slower than the GPU implementation, motivating our GPU-based approach.

4.4 Transaction Execution

After the concurrency control initialization phase, Epic executes the entire batch of transactions concurrently on the GPU using *warp-cooperative execution*, an approach motivated by previous work on GPU-based concurrent data structures [1, 3, 39]. Next, we provide some background on GPUs to motivate our execution approach.

GPUs provide an array of multi-threaded Streaming Multiprocessors (SMs), with each SM containing simple cores (typically 64–128 per SM). The GPU executes instructions from a group of threads, called a *warp*, in a Single Instruction, Multiple Threads (SIMT) lockstep manner on the cores of an SM, with threads executing the same instruction on different data elements. A warp typically consists of a fixed number of threads, such as 32 threads in Nvidia GPUs.

The warp-based execution model makes branch divergence an important aspect of GPU algorithm design. Branch divergence occurs when thread execution diverges due to control flow statements, such as branches, for threads within a warp. In this case, the GPU serializes the execution of the divergent paths, causing longer execution times per warp.

Instead of running a different transaction on each thread of a warp, Epic's warp-cooperative execution model uses all the threads in a warp to cooperatively execute a *single* transaction, which avoids branch divergence altogether. The threads in a warp read and write versions by accessing consecutive locations of a record. The GPU can coalesce (or combine) these contiguous memory accesses into a single request, which improves memory bandwidth utilization and is especially beneficial when transactions access large records. For example, 32 threads in a warp running the same instruction can access 128 contiguous bytes in parallel from global memory.

Although warp-cooperative execution can lead to reduced concurrency, the amount of parallelism available on modern GPUs is more than sufficient for Epic's transaction processing requirements. For example, Nvidia's A6000 GPU has 84 SMs, each capable of scheduling 1536 threads (48 warps) at a time. With the warp-cooperative execution scheme, Epic can execute $84 \times 48 = 4032$ transactions concurrently. We believe that transaction execution will not benefit from higher concurrency due to dependencies between transactions. Therefore, the benefits of avoiding branch divergence and coalesced memory access outweigh the reduced concurrency.

GPU Transaction Scheduling The GPU hardware scheduler dispatches threads on an SM at the granularity of a group of threads called a thread block. While the GPU does not provide control over the scheduling order of thread blocks (or threads within a thread block), it guarantees that an active thread runs to completion without being preempted.

Since Epic assigns a serial order to each transaction before execution, transactions must be scheduled based on their serial order. Otherwise, a later transaction may depend on an earlier transaction, which never gets to run because the later transaction holds the hardware resources. Epic schedules transactions in serial order by dynamically assigning transactions to threads when they become active. To do so, it uses a next-transaction global counter, that it increments once per block to allocate transactions for all warps within a block. Threads within the block then distribute the allocated transactions using a local counter.

4.5 Other Optimizations

Epic exploits parallelism within a transaction by splitting transactions, when possible, into multiple independent pieces. Due to its deterministic nature, these pieces can be executed concurrently while still ensuring isolation [12, 28].

Epic aims to overlap data transfer and computation on the GPU whenever possible by launching asynchronous tasks on different non-blocking CUDA streams. This approach effectively hides the latency associated with transferring transaction parameters and data. As shown in Figure 1, Epic transfers transaction parameters to the GPU. This transfer is overlapped with the execution of the previous batch of transactions. With CPU-side execution, Epic overlaps the transfer of the indexed transactions to the CPU with the initialization phase.

It is possible to pipeline Epic's CPU-side execution with GPU indexing and initialization. However, this approach complicates the index garbage collection mechanism. If a record is deleted in epoch N, its index information cannot be garbage collected until epoch $N+2$ because the indexing in epoch $N+1$ runs concurrently with the execution of epoch N. However, the same key may be re-inserted in epoch $N+1$. In this case, the index information for the record deleted in epoch N cannot be garbage collected. This issue can be resolved by tracking the epoch ID in an index entry when it is created. Epic currently does not implement this pipelined execution.

5 Evaluation

We compare the overall performance of Epic with several state-of-the-art in-memory transaction processing databases using the TPC-C, TPC-C NP and the YCSB benchmarks. Then, we provide a more detailed analysis of Epic's design.

All experiments are run on cloud server with a 32-core Epyc CPU and 512GB of memory. For all the CPU-based databases except Aria, we use 1 thread per core for a total of

32 threads. For Aria, we use the default 12 worker threads because this configuration achieves the highest throughput. We use the Nvidia A6000 GPU with 10752 CUDA cores and 48GB GDDR6 memory. The operating system is Ubuntu 22.04. All experiments are compiled with NVCC 12.0 with CUDA run time version 12.0.

5.1 Database Systems Comparison

We compare Epic against four state-of-the-art in-memory databases: STOv2 [14], Caracal [28], GaccO [6] and Aria [21]. We use the publicly available implementations of Caracal, STOv2 and Aria. Since GaccO's implementation is not publicly available, we implemented GaccO's GPU-side transaction execution based on the description in their paper. We use the default epoch sizes of 500 for Aria, 100K for Caracal, and 32768 for GaccO as specified in their papers for all experiments except for the latency experiment in Section 5.7. We use an epoch size of 100K transactions for Epic because throughput improvements become smaller beyond this epoch size, which balances throughput and latency.

STOv2 is a state-of-art in-memory CPU database. STOv2 implements and compares three concurrency control mechanisms: OCC-based Silo [36], timestamp-based TicToc [38], and a variant of MVCC-based Cicada [17]. These mechanisms are called OSTO, TSTO, and MSTO respectively. STOv2's implementations of TicToc and Cicada perform well thanks to careful attention to implementation choices. We enable both the timestamp splitting and deferred updates optimizations in STOv2. Timestamp splitting behaves similar to our field splitting optimization.

Caracal is a multi-versioned, deterministic CPU in-memory database. Similar to Epic, Caracal batches transactions and splits each epoch into an initialization phase and an execution phase. Caracal uses a version array to implement multi-version concurrency control (MVCC). Each record contains an array of versions that are created during the initialization phase and read during the execution phase. Caracal performs well under contention due to transaction batching and MVCC. However, Caracal's concurrency control mechanism keeps the version array sorted by the version ID, which imposes overhead during the initialization phase, and read operations need to perform a binary search through the version arrays. Additionally, the version array requires expensive garbage collection.

GaccO is a single-version, deterministic GPU database that uses lock-based concurrency control [6]. To support databases larger than GPU memory, GaccO proposes running transactions on both the GPU and the CPU. This CPU-GPU co-execution model requires keeping copies of CPU memory tables in GPU memory when the tables are accessed by GPU-side transactions, synchronizing updates to the tables at epoch boundaries, and delaying CPU-side transactions that conflict with GPU-side transactions.

We only compare with GaccO's GPU-based execution, so no synchronization with the CPU is needed. Similar to Epic, GaccO requires transactions' read-write sets in advance. GaccO initializes an epoch by creating a per-record lock table. For each record, all operations are sorted based on the serial ID of the transactions. The corresponding serial IDs are stored in the lock table, representing the order of lock acquisition. During the execution phase, transactions acquire locks on records deterministically by checking the lock table and waiting until the lock value matches the transaction's ID. Upon release, the lock value is advanced to match the next transaction that accesses the record. However, this lock-based concurrency control does not permit readers to share locks.

GaccO executes a transaction per thread and batches transactions by type (e.g., NewOrder in TPC-C) within an epoch to minimize warp divergence (see Section 4.4). This batching also enables GaccO to use a *commutative optimization* when highly-contended items are accessed commutatively. If an operation updates a data item commutatively then the order of performing such updates is flexible, provided the data item is not otherwise observed by its transaction and there are no other conflicting operations on the item. For instance, a transaction that increments a counter in the database row but never reads the value of the counter can implement the update using atomic instructions, without using the deterministic locking protocol. Since GaccO batches transactions by type, conflicts do not occur with other types of transactions.

However, due to this batching of transactions by type, we do not implement the full TPC-C benchmark for GaccO. For the OrderStatus and StockLevel transactions, batching by type would cause these transactions to execute on a snapshot of the database and return the same results within an epoch. Therefore, we only evaluate GaccO on the TPC-C NP and YCSB benchmarks.[1]

Aria is a deterministic database that does not require advance knowledge of read-write sets [21]. It achieves determinism by executing all transactions in a batch against a database snapshot from the previous epoch, while buffering writes and delaying commit until the end of the epoch. After all transactions have executed, Aria deterministically aborts transactions that conflict with an earlier transaction based on transaction ID ordering, and it uses a deterministic reordering optimization to reorder transactions in a batch to reduce the number of aborts. Aria assumes that the read-write sets of transactions are known after the execution phase, and uses Calvin's deterministic locking as a fallback strategy to rerun the aborted transactions after the execution phase.

Aria only implements TPC-C NP. We evaluate the variant with the fallback strategy since their paper reports that it performs better than without the fallback strategy under all contention levels on TPC-C NP.

[1] The GaccO paper also evaluates TPC-C NP on the GPU.

5.2 TPC-C

We use the TPC-C OLTP benchmark to evaluate Epic. The TPC-C benchmark simulates an OLTP workload for a warehouse management system. It consists of five transactions: NewOrder, Payment, OrderStatus, Delivery, and StockLevel.

The NewOrder transaction creates a new order for a customer by incrementing the nextOrderID field in the District table to obtain the order ID. This makes the write-set of NewOrder dependent on the execution-time value of the order ID. OrderStatus retrieves the status of the last order placed by a customer; StockLevel checks the stock level of items ordered in the last 20 transactions in a district; and Delivery processes the oldest undelivered order in a district.

To identify the read-set and write-set keys of these transactions, Epic runs the read-write set identification phase before the indexing phase. Initially, the order ID used by NewOrder is calculated using a per-district counter, which also helps determine the latest order ID for OrderStatus and StockLevel. Then, for each NewOrder transaction, the order information is inserted into a secondary index. The secondary index uses a range index keyed by the customer ID and the order ID. The secondary index also stores the items ordered in each order. OrderStatus performs a backward range scan using the customer ID and the latest order ID in the district as the key to find the last order ID for a customer. StockLevel uses the latest order ID to lookup the ordered item information to check for stock levels. Lastly, Delivery uses a per-warehouse counter to find the oldest undelivered order.

During execution, transactions can validate the read-write sets determined by the identification phase and abort transactions if they do not match the keys that would be accessed during the execution phase [35]. However, since Epic does not cause any concurrency-control related aborts, the read-write sets always match in TPC-C and so no aborts occur [11].

Furthermore, the Payment and OrderStatus transactions in the original TPC-C benchmark can be provided with a customer ID or the customer's last name. In the latter case, the customer ID is retrieved by scanning a read-only index of customers. Since existing GPU range indexes do not support variable length keys needed for scanning the last name, we simplified Payment and OrderStatus to only use the customer ID for all the databases. Other than this change, the behavior and contention level of Epic's TPC-C implementation conforms to the TPC-C specification.

TPC-C has low contention when each warehouse is assigned a separate CPU core. We vary the number of warehouses to evaluate performance under different contention levels. With a single warehouse, TPC-C becomes highly contended due to the per-warehouse Warehouse, District, and Stock tables.

STOv2 and Caracal implement the TPC-C benchmark and we compare Epic against them. Figure 4 shows the throughput of the systems. Epic outperforms the other systems under all contention levels. Under low contention, Epic benefits from the high memory bandwidth and parallelism offered by the GPU, enabling it to outperform all other systems. The two multi-versioned CPU systems, MSTO and Caracal, perform poorly under low contention due to the high overhead of MVCC. However, they perform better under high contention compared to the single version systems. As expected, Epic's performance degrades under high contention. However, due to the deterministic ordering of transactions and its efficient multi-versioning implementation, Epic outperforms the other systems under high contention as well.

5.3 TPC-C NP

The TPC-C NP benchmark is a subset of the TPC-C benchmark that consists of 50% NewOrder and 50% Payment transactions. We use this benchmark to compare with GaccO and Aria as well. The left graph in Figure 5 shows the throughput of the GPU and then the CPU systems for TPC-C NP.

The Epic, STOv2 and Caracal TPC-C NP results are qualitatively similar to TPC-C results. These databases have higher throughput on TPC-C NP under low contention because TPC-C NP has shorter transactions than TPC-C. However, they have lower throughput on TPC-C NP under high contention because TPC-C NP has higher contention than TPC-C. Caracal and Aria have lower throughput than other CPU based databases, but they also support distributed operation.

GaccO performs poorly under all contention levels because it batches transactions by type. For the Payment transaction, updates on the warehouse table require GaccO to serialize all transactions. Also, GaccO cannot run NewOrder transactions concurrently with Payment transactions, resulting in the GPU being underutilized. Additionally, GaccO's lock-based concurrency control has high overhead under contention.

Epic's performance under low contention for TPC-C NP is much higher than for TPC-C for two reasons. First, TPC-C NP does not require scanning for the latest order of a customer and lookup for ordered items and so the overhead of read-write identification is significantly lower. Second, and more importantly, TPC-C NP has short transactions that can be scheduled on GPU thread blocks (see Section 4.4) more efficiently. With TPC-C's mix of short and long transactions, a block needs to wait for the longest transaction to complete. We plan to explore scheduling strategies that co-locate long read-only transactions within blocks.

To implement GaccO's commutative optimization for TPC-C NP, we changed the NewOrder transaction to use atomic CAS instructions to update the District and Stock tables, and we changed the Payment transaction to use atomicAdd to increment the balances of the warehouse, district, and customers tables. Since the updated values are not used after the update or read by other transactions, the order of updates is flexible. The right graph in Figure 5 shows that GaccO with this optimization outperforms all systems. The throughput

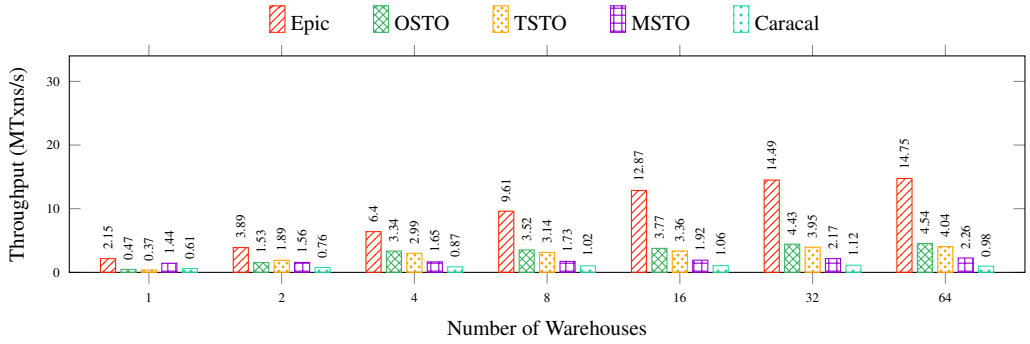


Figure 4: TPC-C Throughput

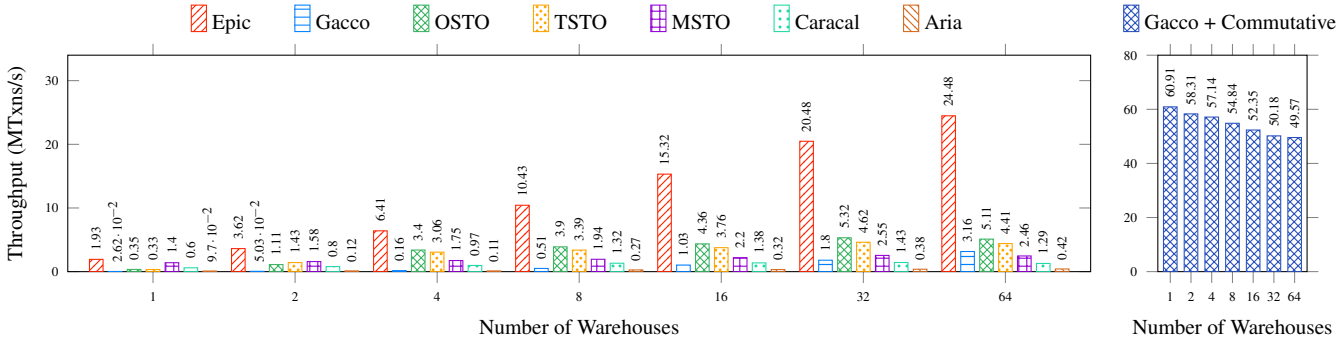


Figure 5: TPC-C NP Throughput

drops slightly with more warehouses due to decreased cache locality. This optimization eliminates concurrency control in TPC-C NP since both the NewOrder and the Payment transactions do not hold any locks. However, this optimization is not general-purpose, e.g., it doesn't allow reading the District table to validate the order ID in the NewOrder transaction.

5.4 YCSB

Next, we conduct experiments using the Yahoo! Cloud Serving Benchmark (YCSB) [9]. For the experimental setup, we use a single table consisting of 1,000,000 records. We used the standard record size in YCSB, where each record is 1000 bytes and consists of ten 100 byte fields. We performed experiments using four YCSB workloads, as shown in Figure 6. In all workloads, a read operation reads the entire record. An update operation replaces the value of one randomly chosen field. A read-modify-write (RMW) operation reads a record and updates a randomly chosen field. For our evaluation, we group 10 operations to form a transaction. We vary the Zipfian skew factor θ from 0 to 0.99 to vary contention levels.

Figure 7 shows the throughput of the six databases for the four YCSB workloads with increasing contention levels. Epic outperforms all other databases for all workloads. In YCSB-A, Epic's performance drops significantly under high contention. Epic performs a read-modify-write operation for each update operation. Even when an update only writes to a part of the record, the entire record needs to be copied from the previous version. As a result, the read-modify-write operations form long dependency chains under high contention. In the YCSB-B benchmark, where the write ratio is low, Epic's performance drops more gently under high contention. In the read-only YCSB-C benchmark, Epic achieves high throughput due to the high memory bandwidth of GPUs. Finally, in the YCSB-F benchmark, Epic shows a similar trend as YCSB-A, where performance drops significantly under high contention because Epic performs the same read-modify-write operations for both YCSB-A and YCSB-F. In some workloads, Epic's throughput increases slightly from low to medium contention level (skew factor 0.0 to 0.5) due to better cache locality that improves GPU indexing performance. The execution phase in Epic also benefits from this better cache locality, especially for read-only YCSB-C.

Workload	Description	Operations
YCSB-A	Update heavy	Read: 50%, Update: 50%
YCSB-B	Read heavy	Read: 95%, Update: 5%
YCSB-C	Read only	Read: 100%
YCSB-F	Read-modify-write	Read: 50%, RMW: 50%

Figure 6: YCSB Workload Configurations

We also evaluate the performance of Epic with field splitting, as described in Section 3.4. In this case, each record is divided into ten fields, and each field is treated as a separate

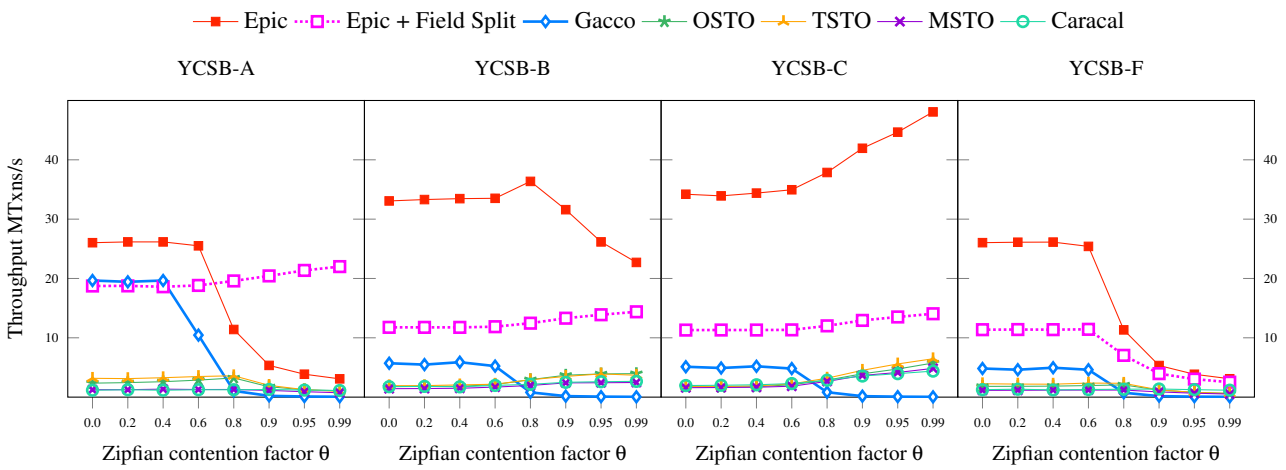


Figure 7: YCSB Throughput

data item from the perspective of concurrency control. As a result, each full-record read operation needs to perform 10 field reads, each requiring separate synchronization. As a result, the number of read operations in the initialization phase increases by 10x, and read performance decreases. On the other hand, since each field is treated separately, an update operation on a single field does not require copying the rest of the fields from the previous versions, improving update performance. As shown in Figure 7, Epic with field splitting performs better than default Epic under YCSB-A with high contention. However, Epic's performance is lower in YCSB-B, YCSB-C, and YCSB-F, where the read ratio is higher.

GaccO shows similar trends under all workloads, performing well under low contention, but its performance drops significantly under high contention due to its lock-based concurrency control. GaccO's initialization phase is simpler and faster than Epic's MVCC initialization but its lock-based concurrency control does not allow readers to share locks, causing its performance to drop significantly under contention, even under a read-only workload. GaccO's assigns each transaction to a single GPU thread, which causes non-coalesced memory accesses that reduce memory bandwidth utilization. As a result, GaccO's performance decreases when the ratio of read operations increases (YCSB-A and YCSB-B) because read operations retrieve the entire record. GaccO's commutative operation optimization cannot be applied to YCSB workloads (except YCSB-C) because other transactions read the values of the data items updated. Therefore, we did not implement this optimization for the YCSB workloads.

Both multi-versioned systems (MSTO and Caracal) suffer from the same extra dependency as Epic in YCSB-A. Therefore, they exhibit similar trends for YCSB-A and YCSB-F. OSTO and TSTO perform well under low contention, but their performance drops significantly under high contention with write-heavy workloads (YCSB-A and YCSB-F). This is due to increased aborts resulting from a high conflict rate. In read-heavy workloads (YCSB-B and YCSB-C), OSTO and TSTO outperform MSTO and Caracal due to their lightweight concurrency control mechanisms. However, Caracal achieves higher throughput than OSTO and TSTO in YCSB-A and YCSB-F under high contention because its MVCC-based concurrency control allows readers to run in parallel with writers.

5.5 CPU-side Execution

Next, we evaluate the performance of Epic's CPU-side execution using the same setup for the TPC-C, TPC-C NP and YCSB benchmarks. As mentioned in Section 3.3, the GPU performs indexing and initialization for the epoch and then transfers the execution plan to the CPU. This data transfer takes roughly 4 ms for the TPC-C NP and YCSB benchmarks and 6 ms for TPC-C, which contains long running queries with more operations. The transactions are then executed on the CPU. The throughput reported in Figure 8 includes the time for indexing, initialization, data transfer and execution because Epic currently does not implement pipelining.

With TPC-C and TPC-C NP, CPU-side execution achieves higher throughput than GPU-side execution with a single warehouse. We believe that the contended Payment transaction limits Epic from utilizing the parallelism of the GPU effectively. On the CPU, Epic's execution time synchronization is more efficient as the atomic flags can be directly communicated through the CPU cache. However, with more warehouses, GPU-side execution achieves higher throughput due to the higher parallelism and memory bandwidth of the GPU.

With CPU-side execution, Epic achieves lower throughput in TPC-C than TPC-C NP under low contention due to the longer data transfer time. However, Epic performs better for TPC-C with a single warehouse because TPC-C has lower contention than TPC-C NP.

With YCSB, each transaction reads several records, and so CPU-side execution is limited by memory bandwidth and la-

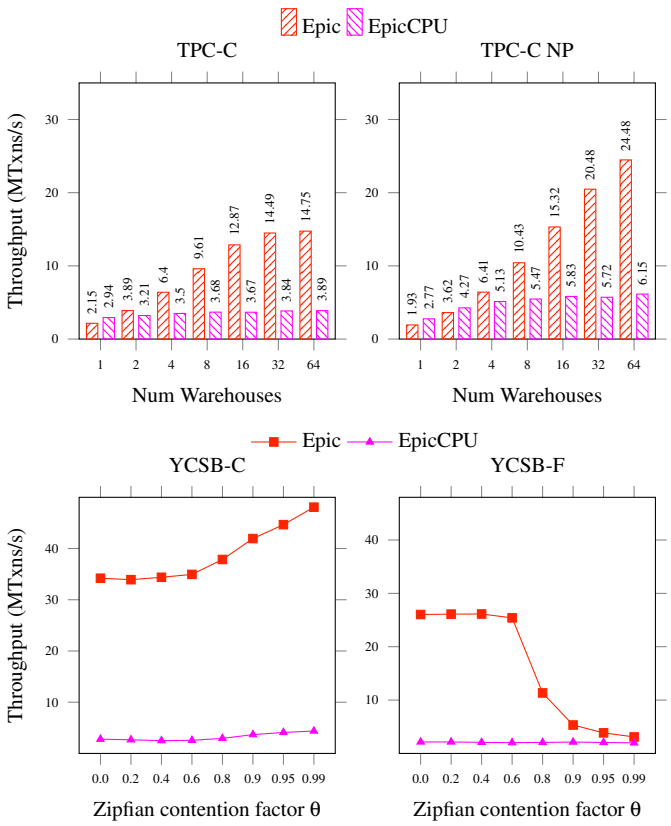


Figure 8: Throughput with CPU-side Execution

tency. For read-only YCSB-C, CPU-side execution has much lower throughput than GPU-side execution. Throughput increases slightly under contention due to cache locality. For YCSB-F, CPU-side execution throughput is bottlenecked by memory bandwidth at low contention and achieves similar throughput as GPU-side execution under high contention. YCSB-A and YCSB-B show similar trends so we omit them.

For all the three benchmarks, Epic's CPU-side execution achieves comparable throughput to OSTO and TSTO under low contention because Epic's GPU initialization is efficient. Under high contention, Epic outperforms OSTO by 6.2x and TSTO by 7.9x for TPC-C single warehouse and both by 3.2x for YCSB-F with a 0.99 skew factor due to its multi-versioning. Epic-CPU outperforms both multi-version systems, MSTO and Caracal, under all workloads because Epic's MVCC initialization is efficient and, unlike MSTO and Caracal, Epic's CPU-side execution runs without performing expensive version search.

5.6 Run Time Breakdown

Figure 9 shows the breakdown of per-epoch run time for Epic running TPC-C with the CPU- and GPU-execution model. The figure shows that the initialization time is similar for both low and high contention levels because Epic's initialization

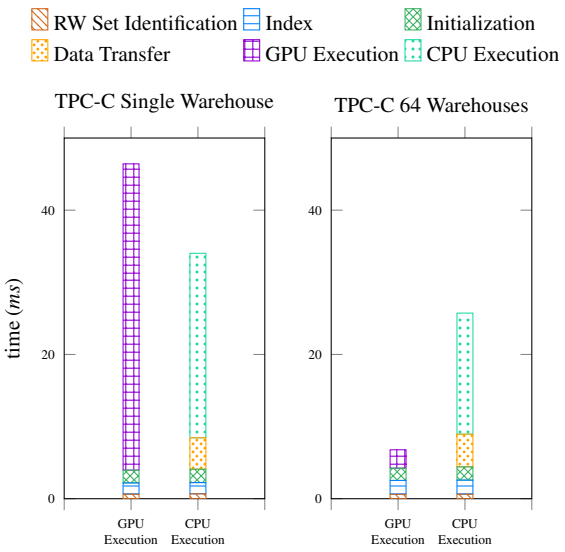


Figure 9: Epic Run Time Breakdown

phase is unaffected by the contention level. The GPU execution time is significantly longer under high contention because transaction dependencies reduce GPU utilization.

For CPU execution, the indexed transactions and the transaction execution plans need to be transferred from the GPU to the CPU. Depending on the complexity of the transaction, the data transfer time can vary but is a significant portion of the total run time. Pipelining the GPU and CPU phases will help reduce the epoch run time.

5.7 Latency

In this experiment, we evaluate Epic's throughput and latency for different epoch sizes by comparing against the GaccO, Caracal, and Aria deterministic databases. We show TPC-C NP results because our GaccO implementation and Aria implement TPC-C NP. We also show YCSB-F results (but not for Aria, which doesn't implement it). For both workloads, we show results under low and high contention. Epic's results for TPC-C are not shown but they are similar to TPC-C NP.

We vary the epoch size from 500 to 200K transactions/epoch. Epic batches transactions during the previous epoch and the benchmarks do not cause aborts, so Epic's average transaction latency is 1.5× the epoch run time.

Figure 10 shows the throughput and average latency of the four systems. Each point on a line represents an epoch size. The lines start at 5000 for Caracal (which crashes at lower epoch sizes) and 500 for all other systems. The lines also show some key epoch sizes, e.g., at maximum throughput and at the knee of the curve. In all workloads, Epic achieves higher throughput with increasing epoch size. Intuitively, a larger epoch enables higher parallelism and amortizes overheads at the cost of transaction latency. Similarly, Caracal's throughput increases with larger epoch sizes.

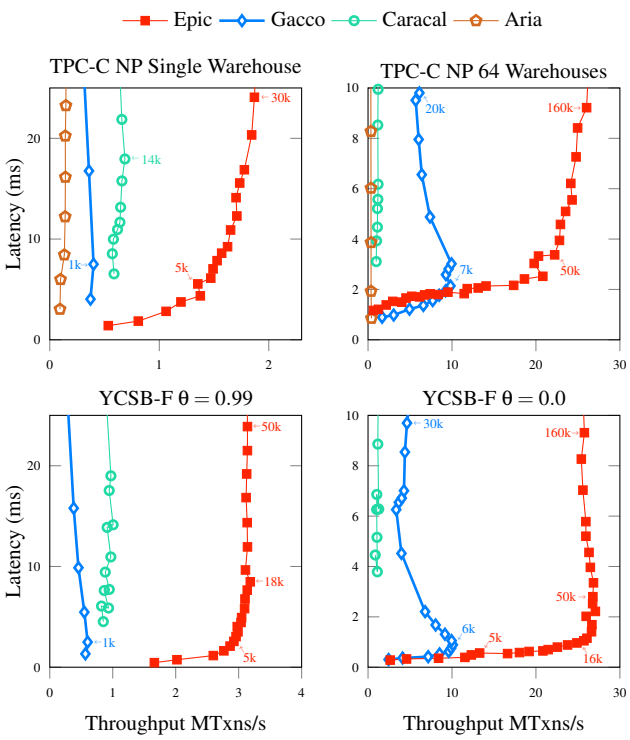


Figure 10: Latency vs. Throughput

GaccO's throughput increases with larger epoch sizes initially but then decreases. We believe that GaccO's lock-based scheduling performance degrades with increasing number of concurrent transactions. We plan to investigate this issue.

Aria's throughput decreases with larger epoch sizes under low contention because more transactions are deterministically aborted. However, Aria benefits from a larger epoch size under high contention. In this case, Aria's deterministic scheduling mechanism aborts a majority of transactions. The aborted transactions are rerun using the deterministic locking fallback strategy, which is more efficient at larger epoch sizes.

Overall, Epic achieves comparable latency to other systems at small epoch sizes. Epic has higher latency than GaccO at small epoch sizes because its multi-version initialization phase is slower and the small epoch size does not allow it to amortize this overhead. However, beyond roughly 2 ms average transaction latency, Epic outperforms all other systems.

5.8 Impact of Aborts

To evaluate the impact of aborts on Epic's performance, we run a micro-benchmark where each transaction reads and updates 10 records. The keys are generated using a Zipfian distribution with $\theta = 0.8$ for medium contention. Transactions abort when the read-set or the write-set is predicted incorrectly, and aborted transactions are rerun in the next epoch. We vary the abort rate for the experiments. We assume that the read-set and write-set are known after a transaction executes,

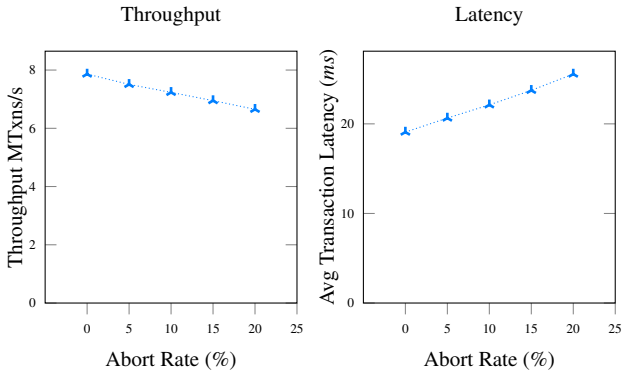


Figure 11: Impact of Abort Rate

and so an aborted transaction will not abort again when rerun, similar to Aria's assumption for its fallback strategy [21].

Figure 11 shows Epic's throughput and average latency. As the abort rate increases, Epic's throughput decreases and latency increases roughly linearly. Aborted transactions are rerun in the next epoch, which increases their latency and requires additional work.

6 Conclusions

Multi-versioning schemes for transaction processing systems have traditionally been popular because they provide good performance for a range of workloads, including for long-running transactions and contended workloads. With in-memory databases increasingly being used for applications requiring high-throughput transaction processing, several multi-version schemes have been proposed for in-memory databases. However, these schemes have significant costs associated with version search and storage, garbage collection, index management.

This work proposes a novel design for multi-versioning that takes advantage of the predetermined ordering of transactions and known read-write sets in deterministic databases to eliminate version search by efficiently pre-calculating the version location of each read/write operation. Our batching design helps reduce version allocation, garbage collection and indexing overheads as well. Our design is parallelizable and so we explore accelerating transaction processing on GPUs. Our evaluation shows that our multi-versioned, GPU database performs well under both low and high contention workloads and significantly outperforms state-of-the-art systems.

Acknowledgments

We thank our shepherd, Eddie Kohler, and the anonymous reviewers for their valuable feedback.

References

[1] Saman Ashkiani, Martin Farach-Colton, and John D. Owens. A Dynamic Hash Table for the GPU. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 419–429, May 2018.

[2] Muhammad A. Awad, Saman Ashkiani, Rob Johnson, Martín Farach-Colton, and John D. Owens. Engineering a high-performance GPU B-tree. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2019, pages 145–157, February 2019.

[3] Muhammad A. Awad, Saman Ashkiani, Rob Johnson, Martín Farach-Colton, and John D. Owens. Engineering a high-performance GPU B-Tree. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 145–157, New York, NY, USA, February 2019. Association for Computing Machinery.

[4] Muhammad A. Awad, Serban D. Porumbescu, and John D. Owens. A GPU multiversion B-tree. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, PACT 2022, October 2022.

[5] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. *ACM SIGMOD Record*, 24(2):1–10, 1995.

[6] Nils Boeschen and Carsten Binnig. GaccO - A GPU-accelerated OLTP DBMS. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, pages 1003–1016, New York, NY, USA, June 2022. Association for Computing Machinery.

[7] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. Serializable isolation for snapshot databases. *ACM Trans. Database Syst.*, 34(4), dec 2009.

[8] Michael J. Carey and Waleed A. Muhanna. The performance of multiversion concurrency control algorithms. *ACM Transactions on Computer Systems*, 4(4):338–378, September 1986.

[9] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, June 2010. Association for Computing Machinery.

[10] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *Proceedings of the International Conference on Management of Data - SIGMOD*, pages 1243–1254. ACM, 2013.

[11] Jose M. Faleiro and Daniel J. Abadi. Rethinking serializable multiversion concurrency control. *Proceedings of the VLDB Endowment*, 8(11):1190–1201, July 2015.

[12] Jose M. Faleiro, Daniel J. Abadi, and Joseph M. Hellerstein. High performance transactions via early write visibility. *Proceedings of the VLDB Endowment*, 10(5):613–624, January 2017.

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

[14] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. Opportunities for optimism in contended main-memory multicore transactions. *The VLDB Journal*, January 2022.

[15] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1675–1687, New York, NY, USA, June 2016. Association for Computing Machinery.

[16] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proceedings of the VLDB Endowment*, 5(4):298–309, December 2011.

[17] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 21–35, New York, NY, USA, May 2017. Association for Computing Machinery.

[18] Yu-Shan Lin, Shao-Kan Pi, Meng-Kai Liao, Ching Tsai, Aaron Elmore, and Shan-Hung Wu. Mgcrab: Transaction crabbing for live migration in deterministic database systems. *Proc. VLDB Endow.*, 12(5):597–610, jan 2019.

[19] Yu-Shan Lin, Ching Tsai, Tz-Yu Lin, Yun-Sheng Chang, and Shan-Hung Wu. Don't look back, look into the future: Prescient data partitioning and migration for deterministic database systems. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1156–1168, New York, NY, USA, 2021. Association for Computing Machinery.

[20] David Lomet, Alan Fekete, Rui Wang, and Peter Ward. Multi-version Concurrency via Timestamp Range Conflict Management. In *2012 IEEE 28th International Conference on Data Engineering*, pages 714–725, April 2012.

[21] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. Aria: A fast and practical deterministic OLTP database. *Proceedings of the VLDB Endowment*, 13(12):2047–2060, July 2020.

[22] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 677–689, New York, NY, USA, May 2015. Association for Computing Machinery.

[23] Nvidia. Cucollection. https://github.com/NVIDIA/cuCollections, 2023.

[24] Oracle. TimesTen In-Memory Database FAQ. https://www.oracle.com/database/technologies/timesten-faq.html, 2021.

[25] Dan R. K. Ports and Kevin Grittner. Serializable snapshot isolation in postgresql. *Proc. VLDB Endow.*, 5(12):1850–1861, aug 2012.

[26] Thamir M. Qadah and Mohammad Sadoghi. Quecc: A queue-oriented, control-free concurrency architecture. In *Proceedings of the 19th International Middleware Conference*, Middleware '18, page 13–25, New York, NY, USA, 2018. Association for Computing Machinery.

[27] Dai Qin, Angela Demke Brown, and Ashvin Goel. Scalable replay-based replication for fast databases. *Proceedings of the VLDB Endowment*, 10(13):2025–2036, September 2017.

[28] Dai Qin, Angela Demke Brown, and Ashvin Goel. Caracal: Contention Management with Deterministic Concurrency Control. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 180–194, New York, NY, USA, October 2021. Association for Computing Machinery.

[29] D. P. Reed. Naming and synchronization in a decentralized computer system. Technical report, Massachusetts Institute of Technology, 1978.

[30] David P. Reed. Implementing atomic actions on decentralized data. *ACM Trans. Comput. Syst.*, 1(1):3–23, feb 1983.

[31] Kun Ren, Dennis Li, and Daniel J. Abadi. Slog: Serializable, low-latency, geo-replicated transactions. *Proc. VLDB Endow.*, 12(11):1747–1761, jul 2019.

[32] Mohammad Sadoghi, Mustafa Canim, Bishwaranjan Bhattacharjee, Fabian Nagel, and Kenneth A. Ross. Reducing database locking contention through multi-version concurrency. *Proceedings of the VLDB Endowment*, 7(13):1331–1342, August 2014.

[33] Weihai Shen, Ansh Khanna, Sebastian Angel, Siddhartha Sen, and Shuai Mu. Rolis: a software approach to efficiently replicating multi-core transactions. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 69–84, 2022.

[34] Vishal Sikka, Franz Färber, and Wolfgang Lehner. Efficient transaction processing in SAP HANA database: The end of a column store myth. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012.

[35] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 1–12, New York, NY, USA, May 2012. Association for Computing Machinery.

[36] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, New York, NY, USA, November 2013. Association for Computing Machinery.

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

[38] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. TicToc: Time Traveling Optimistic Concurrency Control. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1629–1642, New York, NY, USA, June 2016. Association for Computing Machinery.

[39] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. Mega-KV: A case for GPUs to maximize the throughput of in-memory key-value stores. *Proceedings of the VLDB Endowment*, 8(11):1226–1237, July 2015.

A Artifact Appendix

Abstract

We implement Epic, the first multi-versioned GPU-based deterministic OLTP database. Epic batches transactions into epochs and establishes a serial ordering of transactions within a batch before transaction execution. Epic performs concurrency control initialization for a batch of transactions before execution, avoiding version search and reducing version allocation and garbage collection overheads. Epic runs on the GPU to accelerate concurrency control initialization and parallelize batched transaction execution. In addition, Epic supports larger datasets with a CPU execution model. We evaluate Epic using the TPC-C and YCSB benchmarks and compare it with state-of-the-art systems: STOv2, Caracal, Gacco, and Aria.

Scope

The artifact allows reproduction of the results of the paper, including the performance evaluation of Epic using the TPC-C and YCSB benchmarks, the latency and throughput comparison, and the performance evaluation of Epic with varying abort rates.

All the experiments except the runtime breakdown in Figure 9 can be reproduced using the artifact. The runtime breakdown is created by retrieving the runtime information manually, and we do not have a script to automate this process.

Additionally, the artifact cannot perform the performance evaluation for Aria due to the conflict of dependencies. Therefore, the Aria results in Figure 5 and Figure 7 are not reproducible using the artifact.

Contents

The artifact repository contains the source code of Epic, STOv2, and Caracal as separate submodules. We used our best-effort implementation of Gacco, and the source code is included in the Epic submodule. The repository contains scripts to run the experiments and generate the figures in the paper. The repository also contains scripts to install the necessary dependencies and set up the experiment environment. The README file in the repository provides detailed instructions on how to run the artifact.

Hosting

Our artifact repository is hosted on GitHub at https://github.com/ShujianQian/epic-artifact/commit/9303f4d2b1fa8368de0dbdc24bcd798585ceb920.

More details on how to set up the experiment environment, run the experiments, and reproduce the results are provided in the README file in the repository.

Requirements

Our experiments require running on servers equipped with GPUs. We used FluidStack to host on-demand virtual GPU servers. Our artifact repository contains instructions on how to set up the virtual servers and run the experiments.

Alternatively, the experiments can be run on machines with NVIDIA GPUs. The artifact repository is tested for machines with more than 32 CPU cores, 128GB of RAM, and NVIDIA GPUs of compute capability 8.6 and GPU memory of 48GB. The artifact repository contains scripts to install the necessary dependencies and run the experiments.