Integrating Non-Volatile Main Memory in a Deterministic Database

Yu Chen Wang
University of Toronto
shirley.wang@mail.utoronto.ca

Angela Demke Brown
University of Toronto
demke@cs.toronto.edu

Ashvin Goel
University of Toronto
ashvin@eecg.toronto.edu

Abstract

Deterministic databases provide strong serializability while avoiding concurrency-control related aborts by establishing a serial ordering of transactions before their execution. Recent work has shown that they can also handle skewed and contended workloads effectively. These properties are achieved by batching transactions in epochs and then executing the transactions within an epoch concurrently and deterministically. However, the predetermined serial ordering of transactions makes these databases more vulnerable to long-latency transactions. As a result, they have mainly been designed as main-memory databases, which limits the size of the datasets that can be supported.

We show how to integrate non-volatile main memory (NVMM) into deterministic databases to support larger datasets at a lower cost per gigabyte and faster failure recovery. We describe a novel dual-version checkpointing scheme that takes advantage of deterministic execution, epoch-based processing and NVMM's byte addressability to avoid persisting all updates to NVMM. Our approach reduces NVMM accesses, provides better access locality, and reduces garbage collection costs, thus lowering the performance impact of using NVMM. We show that our design enables scaling the dataset size while reducing the impacts of using NVMM, achieving up to 79% of DRAM performance. Our design supports efficient failure recovery and outperforms alternative failure recovery designs, especially under contended workloads, by up to 56%.

CCS Concepts: • **Information systems** → **Record and buffer management**; **Main memory engines**; • **Hardware** → **Non-volatile memory**.

Keywords: Deterministic databases, Non-volatile memory, Recovery, Logging, Checkpointing

EuroSys '23, May 8–12, 2023, Rome, Italy

ACM ISBN 978-1-4503-9487-1/23/05…$15.00
https://doi.org/10.1145/3552326.3567494

1 Introduction

Non-volatile main memory (NVMM), such as Intel's Optane Persistent Memory, is a persistent, byte-addressable memory technology that provides higher storage density and lower cost per gigabyte than DRAM. Recently, there have been several proposals for integrating NVMM into existing storage systems and databases to improve performance or reduce costs. For example, tiered caching designs [26, 27] incorporate NVMM as a caching layer between DRAM and SSD. NVMM's byte addressability also enables efficient recovery methods, such as Write-Behind Logging (WBL) [2] and Zen [14], that minimize logging by writing updates to NVMM directly.

In this paper, we describe the design of a novel *deterministic database* (DB) that integrates NVMM storage, enabling efficient support for datasets much larger than DRAM capacity at a lower cost per gigabyte, and fast failure recovery. Deterministic databases establish a serial ordering of transactions *before* transactions are executed and ensure that the final state of the database is consistent with serial execution in that order [22]. These databases are attractive for several reasons. By pre-establishing the transaction serial order, they can ensure serializable execution, while eliminating concurrency-control related deadlocks and aborts. The lack of such aborts reduces the need for two-phase commit to maintain consistency when transactions access data on multiple nodes, helping scale distributed transaction throughput [23]. Deterministic databases use input logging and deterministic replay for failure recovery, which also simplifies replication [21] and live migration [13]. They can also effectively handle skewed and contended accesses, which occur commonly in real workloads, such as orders for popular items in an online store [20].

Currently, deterministic databases are designed as main-memory databases because the predetermined serial ordering of transactions makes them more vulnerable to long and varying disk latencies [22]. For example, consider transactions $T1$, $T2$, and $T3$ with the pre-determined serial order $T1 < T2 < T3$. If $T1$ and $T2$ contend on key X while $T2$ and $T3$ contend on key Y, $T3$ will not be able to commit before both $T1$ and $T2$ finish due to the ordering constraint. If $T1$ waits for a disk access, both $T2$ and $T3$ are delayed, even though $T3$ does not conflict with $T1$. Traditional databases avoid this issue by re-ordering transactions during execution (i.e., run $T3$ before $T2$ to avoid the wait). A main-memory

database reduces transaction latencies by avoiding disk delays during execution, but the dataset size is limited to the main memory size.

While it is feasible to support larger datasets by extending DRAM with NVMM, NVMM has much higher latency and lower throughput than DRAM, causing significant performance overheads in a deterministic database. For example, our evaluation with the TPC-C benchmark shows that storing the row data for the Caracal database [20] in NVMM versus DRAM results in over 4x throughput degradation.

Our approach uses a hybrid DRAM-NVMM design that takes advantage of deterministic execution to efficiently store and access data in NVMM. Deterministic databases such as Calvin [23], Bohm [5], PWV [6] and Caracal [20] support concurrent execution by batching transactions in epochs. Batching makes it possible to determine dependencies across transactions in an epoch, which in turn enables parallelizing the independent data accesses while ensuring the deterministic serial order. Our key observation is that only the final write to a row in an epoch will be needed in future epochs, and all previous (or intermediate) writes to the same row in an epoch can be discarded at the end of the epoch. Thus we only store the final write to NVMM (*persistent data*), while all intermediate writes are stored in DRAM (*transient data*).

This approach works in a deterministic database because a transaction's outcome is uniquely determined by the database's initial state and an ordered set of known previous transactions. Thus, unlike traditional databases that log transaction outputs for recovery (e.g., using redo logging), deterministic databases log transaction inputs and their predetermined serial order each epoch. Upon failure, they can perform recovery by starting from a checkpoint of the last epoch and re-executing the transactions in the failed epoch using the logged inputs. Since the intermediate writes are replayed deterministically, they do not need to be persisted.

Our separation of transient and persistent data lowers the performance impact of using NVMM. Transactions access transient data from DRAM with low latency. Memory management for transient data can be done efficiently, since it is allocated for the duration of an epoch and discarded at the end of the epoch. For persistent data, we take advantage of NVMM's byte addressability to checkpoint each row's persistent data during execution, instead of relying on logging data sequentially and then checkpointing it.

Existing NVMM-based failure recovery schemes, such as write-behind logging (WBL) [2] and Zen [14], propose *log-reducing* designs that minimize logging but require persisting all row updates to NVMM in a multi-versioned store, which increases storage and garbage collection costs and has poor access locality. Instead, our *update-reducing* design persists each updated row only once per epoch, which is especially beneficial for contended workloads that update certain rows frequently. It also enables a novel epoch-based, dual-version checkpointing scheme that reduces storage requirements, garbage collection costs, and improves access locality.

To support our checkpointing scheme, we design efficient NVMM allocation and garbage collection methods. For tracking persistent allocation information, we use a free list that is persisted at epoch granularity. We cache persistent data in DRAM to reduce NVMM reads and perform cache eviction operations once per epoch, which avoids synchronization with row accesses during transaction execution.

We implement our design in the Caracal [20] deterministic database, although our approach can be applied to other batching-based deterministic databases. We evaluate our implementation, called NVCaracal, with the TPC-C, YCSB, and SmallBank benchmarks. We show that NVCaracal enables scaling the dataset size independent of DRAM capacity, while minimizing the performance impact of using NVMM. It achieves throughput up to 79% of an all-DRAM configuration. Our design supports efficient failure recovery and outperforms existing failure recovery designs, especially under contention.

2 Related Work

2.1 Non-Volatile Main Memory in Databases

NVMM is byte-addressable and thus can be used to extend DRAM. NVMM can also dramatically change the way systems store and access persistent data because applications can directly access this data with minimal OS intervention. NVMM can be configured in either memory or app-direct mode. In memory mode, NVMM appears as volatile memory to the OS and the applications – DRAM serves as a cache and NVMM serves as main memory. This configuration is easy to adopt because it does not require changes to application code, but it does not provide the OS or the applications any control over the DRAM caching policy. In app-direct mode, applications directly access NVMM as byte-addressable non-volatile storage.

There are several storage designs that extend DRAM using NVMM in app-direct mode. Van Renen et al. [26] describe a three-tier buffer manager design that uses NVMM as a caching layer between DRAM and SSD, storing hot data in DRAM, warm data in NVMM, and cold data in SSD. Spitfire [27] is another three-tier buffer manager with an adaptive data migration policy. It allows direct access to NVMM pages and reduces duplicate pages cached in both DRAM and NVMM, allowing more SSD pages to be cached. These buffer managers implement traditional write-ahead logging (WAL) and checkpointing in NVMM for failure recovery. These techniques were designed for block storage devices and do not take advantage of NVMM's byte addressability. They require two NVMM writes for each database update, once for logging and then for checkpointing.

Recent recovery techniques designed specifically for NVMM, such as Write-Behind Logging (WBL) [2] and

Zen [14], minimize logging by taking advantage of NVMM's byte addressability to persist updates to NVMM directly, thus avoiding the need for logging updates and checkpointing them. Our design also takes advantage of NVMM's byte addressability to update NVMM directly, thus eliminating a separate checkpointing phase.

WBL [2] is a logging and recovery technique that uses group commit and only logs a pair of timestamps per commit to NVMM. These timestamps are used during recovery to determine the changes made by un-committed transactions. After a failure, WBL resumes regular execution while garbage collecting NVMM changes made by the group of un-committed transactions, which has poor locality. This delayed garbage collection is not required in our design because deterministic replay during recovery writes the same changes to the same locations. Zen [14] is an OLTP engine that eliminates logging by using a per-row bit stored in NVMM to determine whether a transaction has committed and should be recovered after a crash. On recovery, Zen needs to scan the database more than once to rebuild the database correctly.

Both WBL and Zen use multi-versioning in NVMM. WBL can support single-versioned systems, but stores multiple copies in NVMM to support transaction rollbacks. Zen performs garbage collection by managing free lists in DRAM, which adds memory and compute costs, while WBL does not specify a garbage collection design. Our novel dual-version checkpoint design stores at most two versions per row in NVMM, significantly reducing garbage collection costs.

In WAL, WBL or Zen, all updates must be written to persistent storage at some point to support failure recovery. Our design eliminates this requirement and reduces NVMM accesses by taking advantage of determinism and batching.

Several recent works have proposed NVMM-based index designs [9, 11, 15, 19, 28] for capacity expansion and fast failure recovery. Currently, our design stores indexes in DRAM and rebuilds them during recovery. Incorporating NVMM-based indexes would enable faster recovery.

LeanStore [12] is an efficient buffer manager designed for extending main memory databases beyond DRAM capacity using fast SSDs. Haubenschild et al. [7] propose logging and checkpointing recovery techniques for such SSD-based storage engines. While we currently target NVMM technology, we plan to explore extending to fast block-based storage.

2.2 Deterministic Databases

Calvin [23] is a shared-memory deterministic database that batches transactions to enable concurrent deterministic execution. Bohm [5] improves on Calvin's centralized lock manager by decoupling lock management from transaction execution. Piece-wise-visibility (PWV) [6] proposes making transaction writes visible on transaction commit, before the transaction finishes execution. Caracal [20] is a shared-memory deterministic database that performs well under both skewed and contended workloads. All these main-memory deterministic databases use batching and can incorporate our NVMM design.

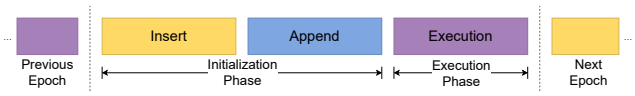


Figure 1. The Caracal Architecture

3 Background

We implement and evaluate our NVMM design in the Caracal [20] deterministic database. This section provides background on Caracal and deterministic concurrency control.

3.1 The Caracal Deterministic Database

The main challenge with deterministic databases is supporting concurrent execution while ensuring the predetermined serial ordering of transactions. Caracal uses two techniques, multiversion concurrency control (MVCC) and transaction batching, to support concurrent execution. With multi-versioning, transactions can read and write different versions of rows concurrently, allowing writers to create new row versions safely while readers are accessing old versions. Caracal supports multiple writes per item per transaction using private buffers. Transaction batching enables determining dependencies across transactions in a batch and parallelizing the independent data accesses while ensuring the serial order. Next, we describe Caracal's transaction model and outline its deterministic concurrency control mechanism.

3.1.1 Deterministic Transaction Model. Caracal's transaction model is similar to other deterministic databases'. It supports one-shot transactions in which a client provides all transaction inputs to the database at the start of the transaction [4, 5, 8, 10, 23, 25]. This approach enables logging or replicating transaction inputs and re-executing the transactions deterministically for failure recovery or replication.

Caracal requires the write-set keys of transactions before transaction execution. These keys are either inferred from the transaction code or provided as transaction inputs, and they are used for concurrency control as discussed below. For transactions where the write-set cannot be inferred before execution, Caracal supports reconnaissance transactions to obtain the write-set by first running a read-only transaction and then validating the reads when running the transaction [20, 23]. Caracal supports a limited form of user-level aborts in transaction logic [4, 6]. For instance, an order placement transaction should abort if there is not enough stock left. In Caracal, these user-level aborts must be issued before the transaction performs any writes. To do so, the transaction performs reads and the relevant checks before issuing write operations.

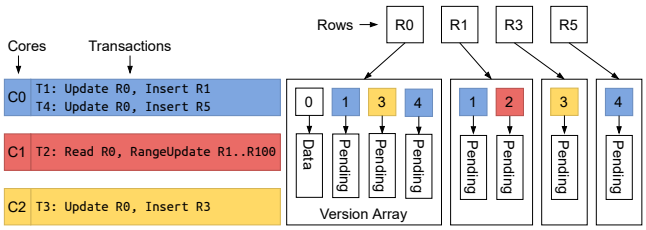


Figure 2. Epoch-Based Transaction Processing in Caracal

3.1.2 Deterministic Concurrency Control. Caracal batches transactions into epochs and executes these transactions in two phases, *initialization* and *execution*, as shown in Figure 1. The initialization phase performs concurrency control for all the transactions in the epoch. The execution phase runs the transactions in the epoch. Transactions within an epoch can be assigned any arbitrary serial ordering, while transactions across epochs are strictly ordered.

Figure 2 shows an example in which Caracal batches four transactions and processes them on three cores in an epoch. The initialization phase uses the write sets of the transactions to create corresponding row versions with a PENDING value that indicates that this version is a placeholder whose data has not been produced yet. The write set of a transaction contains the keys of all the rows that are to be written (inserted, updated or deleted) by the transaction. For example, transactions T_1, T_3 and T_4 update Row R_0 and so the initialization phase creates three PENDING versions for Row R_0. Each version stores a version number that is the serial ID of the writer transaction and a pointer to row version data. Figure 1 shows that Caracal splits the initialization phase into two steps, insert and append. The insert step creates rows and their first pending version (insert operations) while the append step creates additional pending row versions (update and delete operations). This approach allows determining the write set of certain operations that depend on row insertions, such as range updates, during the initialization phase. As an optimization, if the insert operation's row data is available, then the insert step initializes the version's data.

During the execution phase, writers update PENDING row versions with row data without any synchronization. Caracal makes writes visible *before* transactions complete, similar to PWV [6]. This is safe because transactions in Caracal do not abort after issuing their first write, ensuring that transactions read committed data. Readers execute concurrently with writers because they can observe all the row versions that will be created in the epoch (due to the initialization phase) and thus can determine the correct previous version to read based on the serial order. This guarantees that readers always read the latest committed version. For example, Transaction T_2 will read the version of Row R_0 that is written by Transaction T_1 in Figure 2. Unlike traditional MVCC schemes that use a linked list for storing versions, Caracal stores all the row versions in an epoch in a sorted *version array*, allowing more efficient read access to the versions during execution. Readers synchronize with writers by waiting when this version is PENDING, until it is written. No further synchronization is needed since the execution is deterministic, deadlock-free, and has no concurrency-control related aborts.

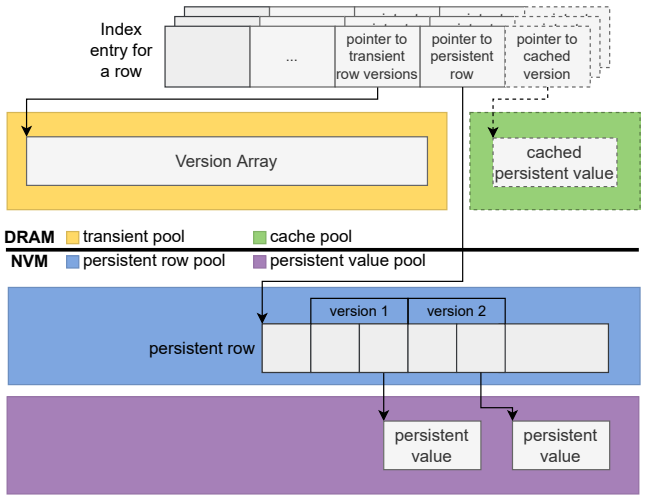


Figure 3. Row Structures in DRAM and NVMM

3.1.3 Failure Recovery. Caracal implements input logging but not checkpointing. We extend Caracal with our dual-version NVMM checkpoint design to fully support failure recovery.

4 Approach

Our design supports large datasets and failure recovery by integrating non-volatile main memory (NVMM) storage in a deterministic database. We leverage the recovery mechanism in a deterministic database to efficiently store and access data in NVMM. For recovery, a deterministic database logs the inputs and the pre-determined serial order of all the transactions in an epoch before processing the transactions. Upon failure, recovery is performed by starting from a checkpoint of the last epoch and deterministically replaying the logged transactions in the failed epoch.[1]

Our key idea is that we can minimize writes to NVMM by only storing the data needed for a checkpoint in NVMM. When a row is updated by any transaction in an epoch, it is immediately visible to other transactions [6]. However, we only need to store the final write to the row in an epoch to NVMM. All the previous (or intermediate) writes can be stored in DRAM and discarded at the end of the epoch. Unlike a non-deterministic database, which must store all committed writes to NVMM, either by logging and then writing them to the database [26, 27] or in the database directly [2, 14], we can avoid storing the intermediate writes

[1]While checkpoints do not need to be taken each epoch, we assume that the epoch is sized based on the checkpoint interval.

to NVMM because they can be recreated when transactions are replayed deterministically during recovery.

We can efficiently distinguish between the intermediate writes and the final write in a deterministic database since we know the write sets of transactions and their serial order before transaction execution (see Section 3.1.1). Figure 3 shows the row structure in our design. We store the intermediate writes as versions in a version array in the transient pool in DRAM (transient data), and the final write in the persistent row and persistent value pools in NVMM (persistent data). Our separation of transient and persistent data lowers the performance impact of using NVMM in a deterministic database since transactions access transient data from DRAM with low latency.

We use several different pools in DRAM based on usage. The transient pool is created each epoch and discarded at the end of the epoch because the intermediate writes are not needed for checkpointing or by transactions in later epochs. We optimize reads by using a cache pool that caches the persistent row value (cached version). This pool is sized based on DRAM capacity. Currently, we store the row index in DRAM for performance. Our design reduces DRAM requirements significantly because only the row index and the transient pool must fit in DRAM. The row index is much smaller than the row data stored in NVMM, and the transient pool size only depends on the epoch size.

4.1 Epoch-Based Transaction Processing

Algorithm 1 shows our epoch-based deterministic transaction processing algorithm. The insert and append steps perform concurrency control and the execute phase runs the transactions in the epoch, as explained in Section 3.1.2. During the insert step, we create new persistent rows in NVMM directly. We do not create any transient data or a cached version for these rows until they are accessed later, thus ensuring that only hot rows are cached in DRAM.

During the append step, the first thread that tries to append to a row creates the version array for the row in the transient pool. Then it creates an initial version in the version array by copying existing data from the persistent row. If the row is cached, it copies the initial version from the cached version instead, and then deletes the cached version since it will be updated during the execution phase. After that, the first thread and later threads append new pending versions to the version array in the transient pool.

During the execution phase, transactions write to rows by creating version values in the version array in the transient pool. The final write creates a cached version so that it remains cached in the next epoch, links the final version in the version array to the cached value, and then writes to the persistent row. The cached value is created before the persistent value so that the write is visible to other transactions earlier and they can read the value from DRAM. A transaction reads the latest version from the version array that has a serial ID smaller than its own serial ID. When there are no writes to the row during the current epoch, then the version array will not exist. In this case, the transaction reads from the cached version or the persistent row. In this design, a row is written at most once, and read at most once (zero times if cached) from NVMM per epoch. The other operations shown in Algorithm 1 are explained in the following subsections.

Algorithm 1: Epoch-Based Deterministic Transaction Processing

```
init_database()
for each epoch do
    log_transaction_inputs()
    /* Initialization phase */
    insert_step()
    GC_major()
    evict_cache()
    append_step()
    /* Execution phase */
    execute_phase()
    fence()
    persist_epoch_number()
    fence()
    /* Epoch checkpointed */
    transient_pool_free()
end
```

4.2 Evicting Cached Versions

The cached version of a persistent row helps reduce NVMM reads. We evict cached versions using an epoch-based LRU policy. Cached versions that have not been updated or accessed in the last user-configurable K epochs are evicted, ensuring that hot rows accessed in recent epochs are kept cached. As shown in Algorithm 1, we evict cached versions during the initialization phase when transactions are not executing, thus requiring no synchronization between eviction and transaction execution.

4.3 Failure Recovery

Our design supports efficient failure recovery by logging the transaction inputs to NVMM at the beginning of each epoch. In many transaction processing workloads, these inputs are much smaller than the transaction writes that are logged by traditional write-ahead logging, which reduces logging overheads. Furthermore, since all transaction inputs are available at the beginning of the epoch, logging can be performed efficiently at close to maximum NVMM throughput.

We ensure that all transaction inputs in the epoch are logged and persisted before the execution phase shown in Algorithm 1. As a result, all writes during transaction execution can be made visible immediately, even before the

transaction ends, since on a failure, we can replay these writes deterministically.

To support failure recovery, we need to checkpoint each epoch so that we can replay the failed epoch starting from the previous epoch's checkpoint. Thus, as shown in Figure 3, we store two versions in the persistent row in NVMM, the most recent checkpointed version from a previous epoch, and a version that is either written in the currently executing epoch or is a stale checkpointed version. We require at most two versions in the persistent row because each row is updated at most once by the final write in an epoch. The most recent checkpointed versions of all the rows provide a checkpoint of the last epoch and are used for recovery after a crash.

At the end of each epoch, we issue a memory fence to ensure that all changes made to NVMM in this epoch are persisted. Then, we persist the current epoch number to NVMM and issue another fence. After this step is complete, the epoch has been checkpointed.

During recovery, we first rebuild the row indexes by scanning the keys of all the persistent rows. As discussed later in Sections 5.4 and 5.5, our persistent row and persistent value allocator pools ensure that any row allocations or deletions that were performed in the crashed epoch are reverted. Then, we replay the transactions using their logged inputs and serial order using our regular epoch-based deterministic concurrency control, which guarantees that the same transactions will update the same rows during recovery.

4.4 Persistent Row Garbage Collection (GC)

We need to garbage collect row versions since we store at most two row versions in NVMM. Our basic garbage collector, which we call the *major* collector, tracks the updated persistent rows each epoch. Then in the next epoch, during the initialization phase (see Listing 1), the major collector deletes the stale version of each updated row, making the version available for this epoch's execution phase. This collector imposes two overheads. First, each updated persistent row is written twice per epoch, once when it is updated, and then when its stale version is collected, which increases the number of NVMM writes. Second, tracking the updated persistent rows has memory and computation costs.

We reduce the cost of GC by using a *minor* collector that does not require tracking the updated persistent rows. Instead, it triggers when a thread writes a final version to the persistent row and finds two existing versions. The thread deletes the stale version immediately, thus making a version available for the final version. By performing GC as part of the update, the minor collector avoids cold NVMM accesses to the persistent rows during major GC. Section 5.3 describes when we use the two garbage collectors.

4.5 Recovery Correctness

We store the two versions in the persistent row, shown in Figure 3, in the same cache line. Each version has a serial ID (SID) of the transaction that wrote the version and a pointer to the version value. Our design ensures that the first version in the persistent row always has a smaller SID than the second version, which avoids comparing version SIDs to determine the latest version.

Both the major and minor collectors ensure that we copy the second (most recent checkpointed) version to the first (stale) version before resetting or overwriting the newly available version. Thus, on failure, the most recent checkpointed version will not be lost (either one or two versions will exist) and can be used during recovery.

When updating a version in the persistent row, both the SID and the pointer to the version need to be modified, but a crash may occur between the modifications to the SID and the pointer. To handle such intervening crashes, we always modify the SID before the pointer. During recovery we can detect whether the pointer was updated or not before the crash by comparing the SID of both the versions. There are three situations when an intervening crash could occur during a persistent version update:

1. GC modifies the first SID version by copying the second version to the first. In this case, we would observe both versions have the same SID (and it is not the crashed epoch's SID). If they have different pointers, then we simply copy the second version's pointer to the first version.
2. GC modifies the second SID version by resetting it to null. In this case, we would observe a null SID in the second version. If the pointer is not null then we simply reset the pointer to null.
3. A transaction writes the final row version in the second SID version. In this case, the second version's SID would match the crashed epoch's SID. During replay, when a transaction writes a final version to the persistent row, it checks whether the persistent row may have already been written during the crashed epoch by checking the serial ID of the second version. If the serial ID was updated during the crashed epoch, the transaction overwrites the version, thus updating the pointer. Otherwise, it performs the write normally, e.g., it may perform minor GC.

Note that modifications to new rows inserted in the crashed epoch will not affect recovery since they are discarded by our persistent allocators after a crash, explained in Section 5.

4.6 Supporting Transaction Aborts

Deterministic databases eliminate concurrency-control-related aborts. However, transactions can still abort based on application logic, such as aborting a transaction that results in a negative bank account balance. When a transaction aborts, it writes IGNORE markers in all of the pending row versions created by the transaction. Other transactions that are reading the version array skip the IGNORE markers and read from a previous non-ignore version.

If an aborted write was originally the final write to a row in the epoch, then it searches for the latest non-ignore version in the version array written in this epoch and writes that version to the persistent row as the final write of this epoch. If the latest non-ignored version is the initial version in the version array then the persistent row will not be updated because the initial version was written during previous epochs and already exists in the persistent row. The cached version is also updated to the latest non-ignore version.

5 Implementation

5.1 Transient Pool Design

The transient pool stores the row version array and the transient version values. We allocate memory for the transient pool using per-core bump allocators. At the end of the epoch, we discard the entire transient pool by simply resetting the bump allocator offset. The row index contains a pointer to the version array (see Figure 3), but version arrays are deallocated at the end of the epoch. Instead of resetting this pointer for every updated row, we detect stale pointers by maintaining a version array epoch number in the row index.

5.2 Cache Eviction

We associate an epoch number with each cached version. When a cached version is created, its epoch number is set to the current epoch, and it is added to the cache eviction list of the current epoch. Whenever the cached version is read or updated by a transaction, we update its epoch number to the current epoch.

In the initialization phase of an epoch, we perform cache eviction on the *current - K - 1* epoch's eviction list. If we see a cached version with a more recent epoch number than *current - K - 1*, we add it to the eviction list of its epoch number and do not evict it. This design evicts all cached versions that were not accessed in the last K epochs.

By using epoch-based eviction outside the execution phase, we can avoid synchronization with row accesses during transaction execution. While our current design does not evict cached versions during execution phases, we can incorporate existing methods [14] to handle heavy memory pressure during execution.

5.3 Persistent Row Design and Garbage Collection

Each persistent row has a fixed size of 256 bytes (by default), which is also the internal access granularity of Intel Optane Persistent Memory. We optimize our storage design by using an inline heap within the persistent row (168 bytes). If a persistent value (either or both of the versions shown in Figure 3) fits within the inline heap, we store it there to improve locality and avoid allocating additional NVMM.

Recall from Section 4.4 that both the minor and major garbage collectors delete the first (stale) version. The minor collector is efficient and operates when the final version is being written during execution, while the major collector operates during the initialization phase. We use the minor collector for persistent rows whose first version is in the inline heap. Otherwise, we use the major collector. We will explain this choice in Section 5.5.

5.4 Persistent Row Pool

The persistent row pool handles allocation and freeing of persistent rows. Each persistent row is fixed size, making it is easy to locate and scan allocated rows during recovery.

To support failure recovery, the persistent row pool must be able to revert the allocation status of the persistent rows to the last checkpointed epoch. Thus any deleted rows in the current epoch must remain allocated and any allocated rows must remain free. Instead of using existing NVMM allocators, we design our own allocator to support epoch-granularity undo efficiently.

For the persistent row pool, we could store a free list in DRAM and track the allocation status of a persistent row by storing an allocation and a deletion epoch number in each persistent row. This would allow rebuilding the state of the free list at the last checkpointed epoch during recovery. However, each allocation and deallocation would require costly modification of the corresponding epoch number field in NVMM.

Instead, we optimize persistent row allocation and freeing by using a per-core persistent bump allocator and free list to track the allocation status of the persistent rows, as shown in Figure 4. The bump allocator uses an offset stored in DRAM (shown as `offset`) to track the size of the allocated region (similar to the `sbrk` system call), and it is used when the free list is empty. Allocations are performed by simply updating this offset in DRAM. The bump allocator also maintains two offsets in NVMM, a checkpointed version and a previously checkpointed version (shown as `offset 1` and `offset 2` respectively, in Figure 4). When the execution phase completes, we persist the DRAM offset to the previously checkpointed version and issue a fence. After the epoch is checkpointed, the previously checkpointed version becomes the checkpointed version. On a crash, we use the checkpointed version to load the DRAM offset, thus reverting the allocated region to its checkpointed state. For example, offset version 1 is persisted during odd epochs and version 2 is persisted during even epochs. During recovery, if the last checkpointed epoch is epoch 5, we would recover the offset from version 1, as it is the most recent checkpointed offset.

We use a per-core ring buffer to store the free list for the core's bump allocator. This list uses a head and tail offset stored in DRAM (shown as `head` and `tail` in Figure 4) and stores pointers to deleted persistent rows. Allocations are performed from the head of the list while deletions are added to the tail of the list. Similar to the bump allocator, the head and tail offsets have two versions in NVMM for recovery. We maintain two invariants in order to be able to revert the free

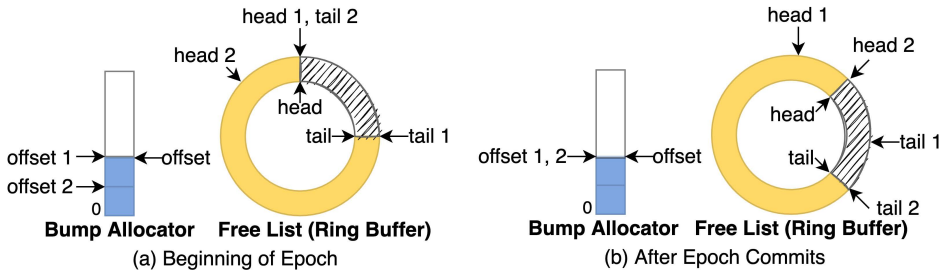


Figure 4. Persistent Row Pool Allocator and Free List

list to the state of the latest checkpointed epoch after a crash. First, the free list of the last checkpointed epoch must not be modified until the current epoch is checkpointed. This ensures that we can revert the free list to the last checkpointed state. Second, the persistent rows that are deleted in the current epoch must not be re-allocated in the same epoch. This ensures that the persistent rows will not be modified and thus their deletions can be reverted if the epoch crashes.

We allocate a persistent row from the free list by incrementing the DRAM head offset, which does not modify the free list in NVMM, thus ensuring the first invariant. We delete a persistent row by adding it to the tail of the free list and increasing the DRAM tail offset. During an epoch, allocations are not allowed beyond the last checkpointed tail offset (head cannot cross `tail 1` in Figure 4), since the persistent rows beyond that tail offset were deleted in the current epoch. These rows must not be reused until the epoch is checkpointed, which ensures the second invariant. We use the bump allocator when an allocation cannot be fulfilled from the free list. Our current implementation assumes that the bump allocator is configured to be large enough for the database and will not run out of space.

Figure 4(a) shows the state of the persistent row pool at the beginning of an epoch. Offset 1, head 1, and tail 1 show the checkpointed state from the previous epoch. Figure 4(b) shows that offset 2, head 2 and tail 2 have been updated from their corresponding DRAM values at the end of the current epoch. Items in the free list between head 1 and head 2 were re-allocated in the current epoch and are no longer part of the free list. Similarly, items between tail 1 and tail 2 were deleted and added to the free list.

Our design is efficient because allocations do not require any persistent writes. A deletion appends a pointer to the free list in NVMM, but these appends are sequential and can be flushed in batches, such as at the 256 byte NVMM access granularity. It is also possible to store the deleted pointers in a temporary buffer in DRAM and then sequentially copy the buffer to the free list at the end of the epoch before the epoch is checkpointed.

Our implementation maps the NVMM memory regions used by the ring buffers and the bump allocators to fixed addresses in each run. This allows us to use the persistent row and persistent value pointers without modification after a crash. If mapping to a fixed address is not possible then we can store offsets instead of pointers and calculate the pointer address using the starting address of the memory region.

5.5 Persistent Value Pool

The persistent value pool handles allocations and frees for persistent values in NVMM. It has the same general design as the persistent row pool, consisting of a per-core persistent bump allocator and free list. We currently allocate persistent values using a fixed size of 1024 bytes. This design can be extended to support multiple sizes by using multiple persistent value pools, such as one pool for each power of two size.

Unlike the persistent row pool, the persistent value pool needs to cooperate with GC. While a persistent row is only deleted when a transaction deletes a row, a persistent value can be deleted when a transaction deletes a row or due to GC. Deletions of rows or values due to transaction logic should be reverted on a crash since the transactions will be replayed deterministically during recovery. However, when GC deletes a persistent value, it overwrites the pointer in the persistent row. If this deletion is reverted after a crash, an NVMM leak would occur since the old pointer value cannot be reverted. Thus, the persistent value pool needs to handle both revertible deletions from transactions and non-revertible deletions from GC. In the first case the persistent free list must be reverted after a crash, but in the second case the free list must not be reverted after a crash.

We simplify the persistent value pool design by ensuring that the two types of deletions do not occur simultaneously. Transaction execution performs revertible deletions, and we do not perform minor GC for non-inline versions. Instead, these versions are deleted by the major collector during the initialization phase, when transactions are not executing (see Section 4.4). With this design, we ensure correct operation by persisting the free list during major GC before the execution phase. A crash during execution reverts the free list to its state after major GC. A crash during major GC does not need to revert any deletions made by transactions in the previous checkpointed epoch.

We add a third non-revertible offset in NVMM called current tail offset to track the free list during major GC. We elide this implementation detail due to space constraints.

If a crash happens during major GC, we need to rebuild the GC list since we do not persist the list during the execution phase. During recovery, when scanning the persistent rows to rebuild the index, we add rows to the GC list when they have two versions and the first one is non-inline (Section 5.3).

We need to ensure that major GC operates idempotently so that we can complete a failed collection correctly. While major GC prevents memory leaks, a crash in between can leave dangling pointers in the persistent rows and cause duplicate deletes upon recovery. On recovery, we build a hash table of the pointers that were deleted in the crashed epoch to detect and avoid adding duplicates to the free list.

6 Evaluation

We implement our NVMM design in Caracal (called *NVCaracal*) and compare its performance with Zen, a state of the art OLTP engine designed for NVMM (see Section 2.1). We also measure the throughput of NVCaracal and alternative NVMM designs in Caracal to compare NVCaracal's epoch-based recovery scheme with existing recovery schemes in a deterministic database. We break down NVCaracal's DRAM and NVMM usage to evaluate how well our design can support larger datasets. We also compare the throughput of NVCaracal against Caracal in DRAM to evaluate the impact of NVMM on performance. Finally, we evaluate the impact of deterministic replay on recovery time in NVCaracal.

We show that NVCaracal can effectively expand the dataset size to NVMM capacity while minimizing NVMM's impact on performance, and it performs better under contention than alternative designs that provide failure recovery.

6.1 Machine setup

We run our experiments on a machine equipped with an Intel Xeon Silver 4215 CPU with 8 cores and 1 NUMA zone, 11 MB of last level cache, 96 GB of DRAM, and 256 GB of Intel Optane Persistent Memory. The NVMM is configured in App Direct mode with an Ext4 filesystem in fsdax mode mounted on it. The machine runs Ubuntu20.04.1 LTS with 5.4.0-54-generic Linux kernel. Measured on our machine, DRAM has 11.9X and 3.2X higher throughput than NVMM for random writes and reads, respectively.

6.2 Benchmarks

We evaluate NVCaracal using both uncontended and contended workloads in three benchmarks: YCSB, SmallBank, and TPC-C. For all benchmarks, we run Caracal's default of 49 epochs with 100,000 transactions per epoch. Using larger epoch sizes can improve throughput at the cost of higher latencies [20]. Our design will also perform better with larger epochs since rows are likely to be updated more times per epoch and so a higher percentage of updates will be written to DRAM compared to NVMM. We set the default cache eviction parameter K to 20, so that rows not accessed in the most recent 20 epochs are evicted from the DRAM cache.

dataset size	16M rows, ≈40 GB 64M rows, ≈150 GB (**YCSB-large**)
value size	1000 64 (**YCSB-smallrow**)
hotspot rows	256 rows
low contention	0/10 accesses to hotspot rows
medium contention	4/10 accesses to hotspot rows
high contention	7/10 accesses to hotspot rows

Table 1. YCSB Configurations. Dataset sizes in GB refer to space used by NVCaracal.

dataset size (customers)	18 million, ≈10 GB 180 million, ≈100 GB (**SmallBank-large**)
value size	8
low contention	1 million hotspot customers
high contention	10,000 hotspot customers

Table 2. SmallBank Configurations. Dataset sizes in GB rfefer to space used by NVCaracal.

6.2.1 YCSB. The YCSB microbenchmark was designed to evaluate key-value stores [3]. Its original implementation does not have multi-key transactions, so we use Caracal's YCSB implementation, which groups 10 read-modify-write operations to unique keys into one transaction. Our default YCSB configuration uses one table with 16M rows, where each row is 1,000 bytes and each write operation updates the first 100 bytes of the row to represent updates to a subset of columns. To evaluate larger datasets, we increase the number of rows to 64M (YCSB-large); to evaluate value inlining, we decrease the row size to 64 bytes and update the entire row (YCSB-smallrow). To change the contention level, we designate 256 rows as "hot" and select varying numbers of keys in a transaction from the hot rows; the remaining keys are chosen randomly from a uniform distribution. Table 1 summarizes our YCSB configurations.

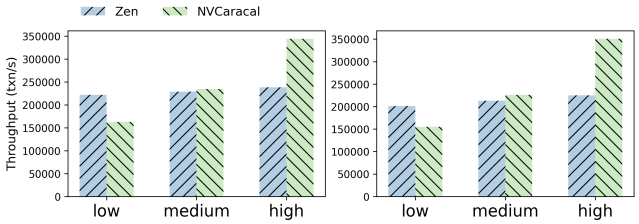
6.2.2 SmallBank. SmallBank is an OLTP benchmark that simulates transactions to bank accounts [1]. It includes five types of transactions that are chosen randomly and uniformly. Two of the transaction types may abort at a rate of 10%. The original SmallBank benchmark consists of 18,000 customers, a subset of which are designated as hotspot customers whose accounts are targeted by 90% of the transactions. In our experiments, we scale the benchmark size by 1000 so it is more realistic for a database; to evaluate larger-than-DRAM datasets we increase the size by an additional 10x. Table 2 summarizes our SmallBank configurations.

low contention	256 warehouses
high contention	1 warehouse

Table 3. TPC-C Configurations

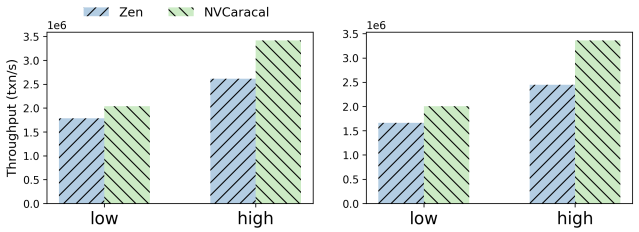
	YCSB	**SmallBank**
NVCaracal Persistent Row Size	2304 bytes	128 bytes
Zen Persistent Row Size	1024 bytes	32 bytes
Max Number of Cache Entries	16M 20M (**YCSB-large**)	6M
NVCaracal Cache Size	17 GB 20 GB (**YCSB-large**)	384 MB
Zen Cache Size	17 GB 20 GB (**YCSB-large**)	768 MB

Table 4. NVCaracal and Zen Configurations



(a) 16M rows, 1K row size (b) 64M rows, 1K row size

Figure 5. YCSB Throughput with (a) Default and (b) Larger Than DRAM Datasets.



(a) 18 million customers (b) 180 (NVCaracal) or 167 (Zen) million customers

Figure 6. SmallBank Throughput with (a) Default and (b) Larger Than DRAM (SmallBank-large) Datasets.

6.2.3 TPC-C. TPC-C is an OLTP benchmark with five transactions that simulates a wholesale supplier with multiple warehouses [24]. We use both 256-warehouse TPC-C and single-warehouse TPC-C to evaluate low and high contention workloads, respectively, as summarized in Table 3. We use Caracal's implementation of TPC-C, which includes two modifications for deterministic execution [20]. For Payment transactions, Caracal removes the customer name lookup and instead provides the customer ID as a transaction input. For NewOrder transactions, Caracal uses an atomic counter in the insert phase to generate the order id (instead of incrementing a field in the District table) since it requires the write sets of transactions before execution.

Caracal's implementation of TPC-C is not completely deterministic due to these counters, so we modify failure recovery for TPC-C slightly. First, we persist the atomic counter values to NVMM at the end of each epoch so we can recover to the state of the latest committed epoch. Second, since different order IDs might be assigned to transactions during replay, we do not make results visible to the user until the epoch is checkpointed. Finally, we revert all persistent values written in the crashed epoch before replaying the epoch during recovery. Changes are reverted while scanning each persistent row to rebuild the index. We use the serial ID of the persistent row's second version to check if it was written during the crashed epoch, and if so, we reset the second version to null. We evaluate the impact on recovery time due to reverting changes in Section 6.8.

6.3 Transaction Throughput Comparison with Zen

We compare the throughput of NVCaracal and Zen using the YCSB and SmallBank benchmarks with different contention levels, and with datasets that fit in DRAM and exceed DRAM capacity (see Tables 1 and 2). We omit TPC-C since it is not supported by Zen's open-sourced code. For Zen's SmallBank-large, we decrease the dataset size slightly to 167 million customers, since its DRAM allocation exceeds the DRAM capacity of our machine at 180 million customers. Table 4 shows various configurations used for this experiment. For a better comparison, we use the optimal persistent row sizes that result in the highest throughputs for both NVCaracal and Zen. For NVCaracal, these sizes allow inlining all persistent values to the persistent rows. We also configure the same limit of DRAM cache entries for NVCaracal and Zen. We have evaluated Zen with double the number of cache entries and did not observe a significant impact on performance (at most 4%).

Figure 5 shows the YCSB throughput of NVCaracal and Zen. Under low contention, NVCaracal is only able to write 3% of the updates as transient values to DRAM since most rows are only updated once within an epoch. Thus for each update, NVCaracal must write to NVMM twice (once for input logging, and once for writing the update during execution), causing NVCaracal to perform 30-36% worse than Zen under low contention. However, as the contention level increases, NVCaracal outperforms Zen despite having additional logging requirements due to the reduced number of NVMM writes during execution that result from writing transient values. At high contention, NVCaracal is able to write 70% of the updates as transient values to DRAM, whereas Zen must write all updates to NVMM regardless of

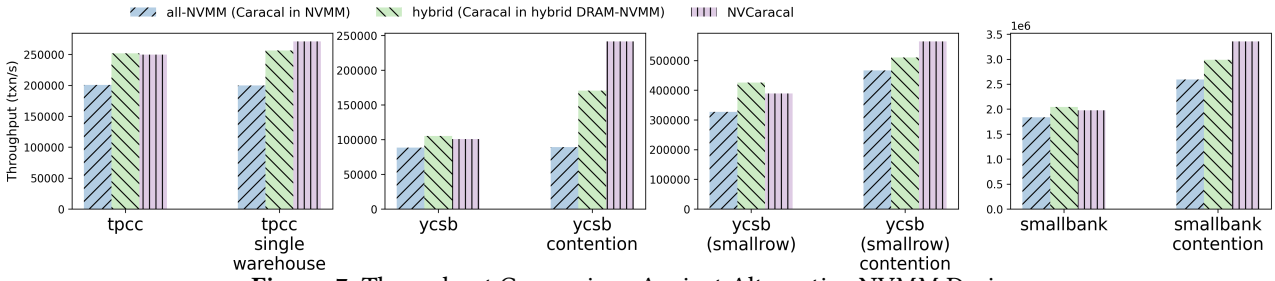


Figure 7. Throughput Comparison Against Alternative NVMM Designs

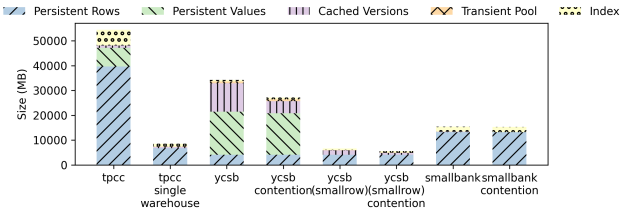


Figure 8. DRAM and NVMM Consumption in NVCaracal

contention level. Therefore, NVCaracal outperforms Zen by 45-56% under high contention.

Figure 6 shows the SmallBank throughput of NVCaracal and Zen. NVCaracal outperforms Zen by 14-21% under low contention and 31-37% under high contention. The SmallBank transaction input sizes are much smaller than in YCSB, which reduces input logging for NVCaracal. Both Zen and NVCaracal show improved performance under high contention due to higher hit rates in their DRAM caches, which reduce NVMM reads, especially for the read-only table in SmallBank. NVCaracal also reduces NVMM writes using transient updates under high contention, on top of reducing NVMM reads with the DRAM cache. Therefore, NVCaracal outperforms Zen by a wider margin under high contention.

For both YCSB and SmallBank, both NVCaracal and Zen experience small reductions in throughput when comparing the larger data set to the default data set due to a lower DRAM cache hit rate. For both benchmarks, NVCaracal's throughput drops by 2% on average but Zen's throughput drops by 7% on average since Zen is more reliant on the DRAM cache for reducing NVMM accesses.

6.4 NVCaracal Design Analysis

We evaluate the performance of NVCaracal, which uses input logging for failure recovery and avoids writing transient versions to NVMM, against two alternative NVMM Caracal designs using low and high contention TPC-C, YCSB, YCSB-smallrow, and SmallBank benchmarks (Figure 7). *Caracal in NVMM* (*all-NVMM*) is a simple baseline design where all values and version arrays are stored in NVMM. *Caracal in hybrid DRAM-NVMM* (*hybrid*) improves on the baseline design by storing version arrays in DRAM and writing all updates to NVMM, but with caching in DRAM similar to Zen. Both *all-NVMM* and *hybrid* omit logging since it is theoretically possible to support failure recovery without logging when all updates are written to NVMM. We use the default persistent row size of 256 bytes for all benchmarks to show the impact of inlining values in persistent rows vs. allocating them from the persistent value pool. We use default datasets that fit in DRAM to compare with *Caracal in DRAM* in Section 6.7.

In all benchmarks except for YCSB, almost all persistent values can be inlined within the persistent rows. In YCSB, all values are non-inline. Inlining reduces NVMM allocations for persistent values and improves NVMM access locality because the persistent row size is within the internal access granularity of Intel Optane Persistent Memory (256 bytes).

Figure 7 shows that the simple baseline design always has the worst performance since it stores all data in NVMM, which has much lower throughput than DRAM. Both *hybrid* and NVCaracal outperform *all-NVMM* since they leverage DRAM to reduce the impact of NVMM on performance. In NVCaracal, the reduction in NVMM accesses is proportional to the number of transient version values stored in DRAM, which increases with higher contention in the workload. We see that NVCaracal performs better under higher contention for all benchmarks. For the low / high contention YCSB and SmallBank benchmarks, roughly 3% / 70% of the versions are transient, while for TPC-C roughly 8% / 30% of the versions are transient.

When version values are large like in YCSB, *all-NVMM* performs significantly worse than NVCaracal under contention (2.9x slower) due to writing large transient values to NVMM. When version values are small (e.g., SmallBank), *all-NVMM* improves under higher contention and is only 1.38x slower than NVCaracal. NVCaracal and *hybrid* have similar performance under low contention, but NVCaracal outperforms *hybrid* for all high-contention workloads. This demonstrates the trade-off of reducing logging versus reducing NVMM writes during execution in a deterministic database. NVCaracal performs more writes to NVMM than *hybrid* for YCSB-smallrow and SmallBank, since (a) the transaction inputs are not smaller than their outputs, and (b) the

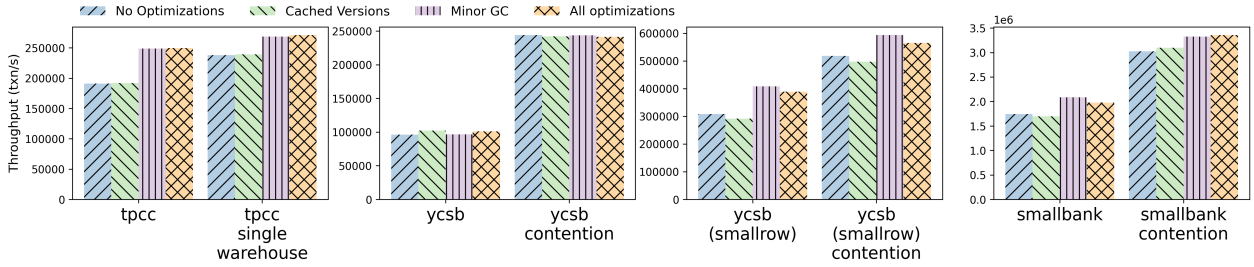


Figure 9. Impact of Optimizations on Throughput

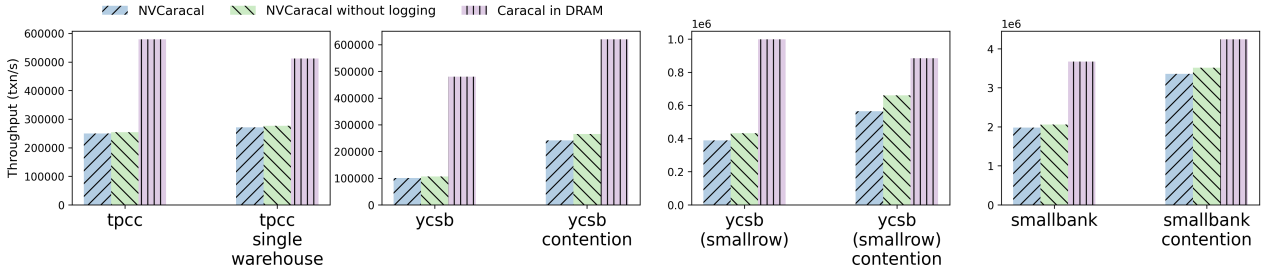


Figure 10. Failure Recovery Impact on Throughput

final writes in each epoch are written to NVMM during execution as well as being logged as input. However, NVCaracal still outperforms *hybrid* since it reduces long latency NVMM writes on transactions' critical paths during execution. This can greatly affect the performance of a deterministic database due to its limited re-ordering capability.

6.5 Memory Consumption

Figure 8 breaks down the DRAM and NVMM usage for the data structures in NVCaracal. In all benchmarks, most of the storage is allocated from NVMM. The index and transient pool in DRAM consume 12% of the total storage on average, with a maximum of 15.5%. This allows NVCaracal to support larger datasets compared to a DRAM-only design. The transient pool size does not expand with dataset size since it is bounded by the epoch. For YCSB, the cached versions consume a large amount of DRAM since the value sizes are large, but the cached versions are not required for database operations and so the dataset size is not limited by the cache footprint. Cached versions can be evicted more aggressively by setting a smaller value of K if memory pressure becomes an issue. In all benchmarks except YCSB, most persistent values can be inlined, thereby reducing NVMM usage for persistent values.

6.6 Impact of Optimizations

Figure 9 shows the impact of the minor GC and cached versions optimizations on throughput. When it can be applied, minor GC is the most beneficial optimization. Minor GC is not triggered for YCSB since the values are too large to be inlined. For the other benchmarks, the improvement due to minor GC ranges from 9.8% in contended SmallBank to 32.4% in uncontended YCSB-smallrow. The cached version optimization improves performance by reducing NVMM reads. Improvements range from 0.5% for TPC-C to 6% for YCSB. However, creating cached versions has memory and CPU overhead, and they are not always effective. In the worst case there is a -5.2% drop in throughput for YCSB-smallrow. We plan to explore creating cached versions only for hot rows, which can be identified during epoch initialization.

6.7 Impact of Supporting Failure Recovery

We compare the throughput of NVCaracal with *NVCaracal without logging* (*no-logging*) and *NVCaracal in DRAM* (*all-DRAM*) in Figure 10. The latter two designs cannot support failure recovery. We use NVCaracal for *all-DRAM* instead of Caracal since it performs slightly better due to our minor garbage collector optimization. For TPC-C, we see that input logging only adds a 2% overhead in NVCaracal compared to *no-logging*, since transaction inputs are much smaller than transaction outputs. For SmallBank and YCSB the transaction inputs are not much smaller, so logging adds a larger overhead ranging from 4% to 17%. The amount of logging is not affected by contention level, but its overhead increases since the number of NVMM writes during execution is reduced with contention.

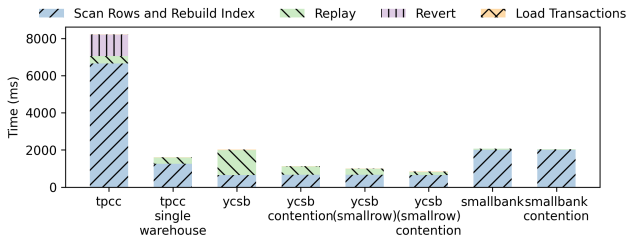


Figure 11. Recovery Time

Comparing NVCaracal to *all-DRAM*, NVCaracal is less than 2x slower in most benchmarks, which is small compared to the read and write throughput difference between DRAM and NVMM. NVCaracal is the most efficient for the contended SmallBank benchmark and is only 1.26x slower than *all-DRAM* thanks to transient versions and inlining.

6.8 Recovery Performance

To measure the recovery performance of NVCaracal, we run 4 million transactions and crash the database before the epoch is checkpointed. Figure 11 breaks down the recovery time into loading transactions, scanning persistent rows and rebuilding the index, reverting changes for TPC-C, and replaying the epoch. In most workloads, it takes 2 seconds or less to recover the database. The largest cost is scanning the persistent rows and rebuilding the index, which depends on the total size of persistent rows. Persistent values are not scanned during recovery, so their size does not affect recovery time. Replaying transactions is generally the next largest contributor to recovery time, but it is bounded by the epoch size and not the database size. Reverting changes for TPC-C adds ≈1 second to recovery time under low contention, but has almost no overhead under high contention because fewer persistent values are written under contention and therefore less reversions are needed. Note that Zen's recovery design does not require replaying transactions, but it requires scanning the database rows more than once. As the database size grows, Zen's recovery performance will scale worse than our design since we only scan the rows once and the epoch replay time does not increase with database size.

6.9 Effects of Epoch Size

We evaluate NVCaracal with different epoch sizes ranging from 5,000 to 100,000 transactions per epoch. Figure 12 shows the throughput and epoch latency trade-off for the different epoch sizes. Overall, a larger epoch size results in a higher throughput at the cost of higher latency. The differences in throughput between the smallest and the largest epoch sizes range from 3% (contended YCSB) to 51% (contended SmallBank). Larger epochs result in higher throughputs for two reasons. First, epoch synchronization is performed less frequently and causes less overheads. Second, the number

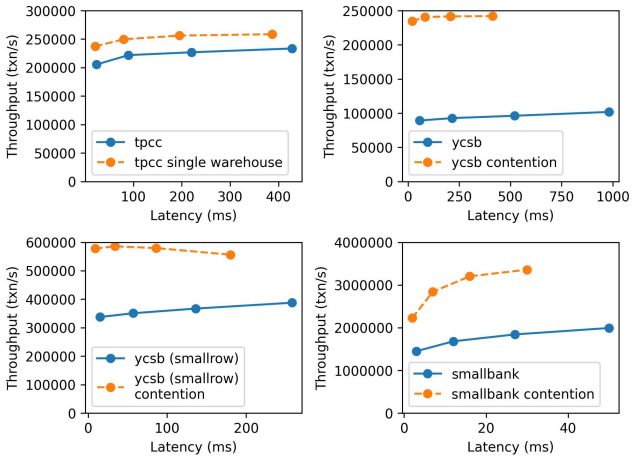


Figure 12. Effects of Epoch Size on Throughput and Latency

of updates for each row in an epoch increases, so the percentage of transient updates increases as well, and thus a higher percentage of updates can be written to DRAM instead of NVMM. However, one exception is the contended YCSB (smallrow) workload. For this workload, the largest epoch size results in a 4% lower throughput than the smallest epoch size. We observed that this is due to a slower append phase. YCSB has a small set of hot rows that are updated frequently in the contended workload. The version arrays of these hot rows are much longer for larger epochs, and the append phase slows down due to these long sorted version arrays. This issue can be addressed using the batch append optimization [20] that is currently not supported by our implementation.

7 Conclusions

Non-volatile main memory presents an opportunity to expand the dataset sizes of in-memory databases beyond DRAM capacity. However the higher latency and lower throughput of NVMM requires a careful redesign of the database structures and memory allocation strategies. In this work, we have integrated NVMM with a multi-versioned deterministic database. We take advantage of the epoch-based deterministic execution model to store transient versions in DRAM and checkpoint the final version to NVMM at the end of each epoch. Our approach stores at most two versions in NVMM, reduces NVMM accesses, provides better access locality and reduces garbage collection costs. Our approach also enables efficient schemes for persistent allocation, caching data in DRAM and failure recovery. Our results show that our approach lowers the cost of using NVMM, allowing us to scale the dataset size, support efficient failure recovery, and achieve good performance even with contended workloads.

Currently, we store the row indexes in DRAM. We plan to explore persisting the row indexes to NVMM to improve

recovery time and reduce DRAM requirements further. We expect that our epoch-based design will allow persisting index updates in batches efficiently. Our results show that caching values in DRAM is not always beneficial for cold rows. We plan to explore strategies to selectively cache values using the write-set information that is available during epoch initialization.

Our design targets determinisitic databases that know or can speculatively infer the write set of transactions before their execution. Recently proposed deterministic concurrency control schemes such as Aria [16] and BCDB [18] eliminate this requirement by executing a batch of transactions against a snapshot of the database and by using deterministic cycle prevention to ensure serializability. We plan to explore integrating NVMM in these databases.

We also plan to extend our design to non-deterministic databases that support epoch-based commit and replication [17]. Unlike deterministic databases, these databases cannot make results visible to users until the epoch commits and must revert all changes made in a crashed epoch during recovery due to the lack of deterministic replay. We expect that these differences will only have a small impact on our design.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Shimin Chen, for their valuable feedback. We specially thank Michael Stumm, Ding Yuan, and the members of the Caracal group, especially Dai Qin and Zhiqi He, for their insightful suggestions. This research was supported by an NSERC Discovery Grant.

References

[1] Mohammad Alomari, Michael Cahill, Alan Fekete, and Uwe Rohm. 2008. The Cost of Serializability on Platforms That Use Snapshot Isolation. In *2008 IEEE 24th International Conference on Data Engineering*. 576–585. https://doi.org/10.1109/ICDE.2008.4497466

[2] Joy Arulraj, Matthew Perron, and Andrew Pavlo. 2016. Write-behind Logging. *Proc. VLDB Endow.* 10, 4 (Nov. 2016), 337–348. https://doi.org/10.14778/3025111.3025116

[3] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[4] James Cowling and Barbara Liskov. 2012. Granola: Low-Overhead Distributed Transaction Coordination. In *Proceedings of the 2012 USENIX Annual Technical Conference* (Boston, MA) *(USENIX ATC'12)*. USENIX Association, USA, 21–33.

[5] Jose M. Faleiro and Daniel J. Abadi. 2015. Rethinking Serializable Multiversion Concurrency Control. *Proc. VLDB Endow.* 8, 11 (July 2015), 1190–1201. https://doi.org/10.14778/2809974.2809981

[6] Jose M. Faleiro, Daniel J. Abadi, and Joseph M. Hellerstein. 2017. High Performance Transactions via Early Write Visibility. *Proc. VLDB Endow.* 10, 5 (Jan. 2017), 613–624. https://doi.org/10.14778/3055540.3055553

[7] Michael Haubenschild, Caetano Sauer, Thomas Neumann, and Viktor Leis. 2020. Rethinking Logging, Checkpoints, and Recovery for High-Performance Storage Engines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 877–892. https://doi.org/10.1145/3318464.3389716

[8] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. 2020. Opportunities for Optimism in Contended Main-Memory Multicore Transactions. *Proc. VLDB Endow.* 13, 5 (Jan. 2020), 629–642. https://doi.org/10.14778/3377369.3377373

[9] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. USENIX Association, Oakland, CA, 187–200. https://www.usenix.org/conference/fast18/presentation/hwang

[10] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1496–1499. https://doi.org/10.14778/1454159.1454211

[11] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 462–477. https://doi.org/10.1145/3341301.3359635

[12] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 185–196. https://doi.org/10.1109/ICDE.2018.00026

[13] Yu-Shan Lin, Shao-Kan Pi, Meng-Kai Liao, Ching Tsai, Aaron Elmore, and Shan-Hung Wu. 2019. MgCrab: Transaction Crabbing for Live Migration in Deterministic Database Systems. *Proc. VLDB Endow.* 12, 5 (Jan. 2019), 597–610. https://doi.org/10.14778/3303753.3303764

[14] Gang Liu, Leying Chen, and Shimin Chen. 2021. Zen: A High-Throughput Log-Free OLTP Engine for Non-Volatile Main Memory. *Proc. VLDB Endow.* 14, 5 (Jan. 2021), 835–848. https://doi.org/10.14778/3446095.3446105

[15] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable Hashing on Persistent Memory. *Proc. VLDB Endow.* 13, 8 (April 2020), 1147–1161. https://doi.org/10.14778/3389133.3389134

[16] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: A Fast and Practical Deterministic OLTP Database. *Proc. VLDB Endow.* 13, 12 (July 2020), 2047–2060. https://doi.org/10.14778/3407790.3407808

[17] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2021. Epoch-Based Commit and Replication in Distributed OLTP Databases. *Proc. VLDB Endow.* 14, 5 (Jan. 2021), 743–756. https://doi.org/10.14778/3446095.3446098

[18] Senthil Nathan, Chander Govindarajan, Adarsh Saraf, Manish Sethi, and Praveen Jayachandran. 2019. Blockchain Meets Database: Design and Implementation of a Blockchain Relational Database. *Proc. VLDB Endow.* 12, 11 (July 2019), 1539–1552. https://doi.org/10.14778/3342263.3342632

[19] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 371–386. https://doi.org/10.1145/2882903.2915251

[20] Dai Qin, Angela Demke Brown, and Ashvin Goel. 2021. Caracal: Contention Management with Deterministic Concurrency Control. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association

for Computing Machinery, New York, NY, USA, 180–194. https://doi.org/10.1145/3477132.3483591

[21] Kun Ren, Dennis Li, and Daniel J. Abadi. 2019. SLOG: Serializable, Low-Latency, Geo-Replicated Transactions. *Proc. VLDB Endow.* 12, 11 (July 2019), 1747–1761. https://doi.org/10.14778/3342263.3342647

[22] Alexander Thomson and Daniel J. Abadi. 2010. The Case for Determinism in Database Systems. *Proc. VLDB Endow.* 3, 1–2 (Sept. 2010), 70–80. https://doi.org/10.14778/1920841.1920855

[23] 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* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2213836.2213838

[24] Transaction Processing Performance Council (TPC). 2010. TPC Benchmark C Standard Specification, Revision 5.11.

[25] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-Memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 18–32. https://doi.org/10.1145/2517349.2522713

[26] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. 2018. Managing Non-Volatile Memory in Database Systems. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1541–1555. https://doi.org/10.1145/3183713.3196897

[27] Xinjing Zhou, Joy Arulraj, Andrew Pavlo, and David Cohen. 2021. Spitfire: A Three-Tier Buffer Manager for Volatile and Non-Volatile Memory. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD/PODS '21)*. Association for Computing Machinery, New York, NY, USA, 2195–2207. https://doi.org/10.1145/3448016.3452819

[28] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. 2019. DPTree: Differential Indexing for Persistent Memory. *Proc. VLDB Endow.* 13, 4 (Dec. 2019), 421–434. https://doi.org/10.14778/3372716.3372717

A Artifact Appendix

A.1 Abstract

This appendix provides information for our artifact, the implementation of NVCaracal.

A.2 Description & Requirements

The implementation of NVCaracal is publicly available on GitHub at https://github.com/uoft-felis/felis/tree/pmem. To compile and run this artifact, your machine should be equipped with a CPU with at least 8 cores, at least 128GB of Intel Optane Persistent Memory, and at least 64GB of DRAM.

A.2.1 Hardware dependencies.

Intel Optane Persistent Memory (at least 128GB)

A.2.2 Software dependencies.

Linux operating system

A.2.3 Benchmarks.

The artifact includes implementations for all three benchmarks evaluated in the paper (TPC-C, YCSB, and SmallBank).

A.3 Set-up

Please refer to the instructions provided in the artifact at https://github.com/uoft-felis/felis/blob/pmem/doc/artifact_instructions.md.

A.4 Evaluation workflow[2]

A.4.1 Major Claims.

- (C1): NVCaracal achieves higher throughput as the contention level in the workload increases, due to less NVMM writes for persistent values. This is proven by the experiment (E1) described in Section 6.4 whose results are illustrated in Figure 7.

A.4.2 Experiments.

Experiment (E1): [NVCaracal Throughput Measurements] [30 human-minutes + 30 compute-minutes]: measures NVCaracal's throughput with the TPC-C, YCSB, and SmallBank benchmarks under low and high contention.

[How to] Follow the instructions provided in the artifact at https://github.com/uoft-felis/felis/blob/pmem/doc/artifact_instructions.md to run the TPC-C, YCSB, and SmallBank benchmarks with low and high contention. The throughput will be displayed at the end of each run.

[Results] Results should follow a similar trend as figure 7, where NVCaracal's throughput increases with higher contention level in the workload.

[2]Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://sysartifacts.github.io/eurosys2023/.