Zen: a High-Throughput Log-Free OLTP Engine for Non-Volatile Main Memory

Gang Liu, Leying Chen, Shimin Chen*
SKL of Computer Architecture, ICT, CAS
University of Chinese Academy of Sciences
{liugang,chenleying,chensm}@ict.ac.cn

ABSTRACT

Emerging Non-Volatile Memory (NVM) technologies like 3DX-point promise significant performance potential for OLTP databases. However, transactional databases need to be redesigned because the key assumptions that non-volatile storage is orders of magnitude slower than DRAM and only supports blocked-oriented access have changed. NVMs are byte-addressable and almost as fast as DRAM. The capacity of NVM is much (4-16x) larger than DRAM. Such NVM characteristics make it possible to build OLTP database entirely in NVM main memory.

This paper studies the structure of OLTP engines with hybrid NVM and DRAM memory. We observe three challenges to design an OLTP engine for NVM: tuple metadata modifications, NVM write redundancy, and NVM space management. We propose Zen, a high-throughput log-free OLTP engine for NVM. Zen addresses the three design challenges with three novel techniques: metadata enhanced tuple cache, log-free persistent transactions, and light-weight NVM space management. Experimental results on a real machine equipped with Intel Optane DC Persistent Memory show that Zen achieves up to 10.1x improvement compared with existing solutions to run an OLTP database as large as the size of NVM while achieving fast failure recovery.

PVLDB Reference Format:
Gang Liu, Leying Chen, Shimin Chen. Zen: a High-Throughput Log-Free OLTP Engine for Non-Volatile Main Memory. PVLDB, 14(5): 835 - 848, 2021.
doi:10.14778/3446095.3446105

1 INTRODUCTION

Byte-addressable, non-volatile memory (NVM) is a new type of memory technology designed to address the DRAM scaling problem [1, 3, 18, 29, 39]. NVM delivers a unique combination of near-DRAM speed, lower-than-DRAM power consumption, affordable large (up to 6TB in a dual-socket server) memory capacity, and non-volatility in light of power failure. By eliminating disk I/Os, NVM can substantially improve the performance of systems with persistence requirement. Therefore, OLTP databases using NVM as primary storage is emerging as a promising design choice [5, 6, 20].

Recent studies in concurrency control methods have advanced the single-machine main memory OLTP transaction throughput (without persistence) to over one million transactions per second [15, 24, 27, 32, 36, 41]. However, replacing DRAM with NVM in a system tends to slow down the system because NVM performs modestly (e.g., 2–3x) slower than DRAM, NVM writes have lower bandwidth than reads, and persisting writes from CPU cache to NVM incurs extra overhead. In this paper, we would like to rethink the design of the OLTP engine for NVM by fully considering NVM's characteristics. Our goal is to achieve transaction performance similar to those of pure DRAM based OLTP engines.

We observe three main challenges in achieving our goal:

Tuple Metadata Modifications: Concurrency control methods typically keep a small amount of metadata per tuple in a main memory OLTP engine [24, 27, 32, 36, 41]. The per-tuple metadata is often modified not only by tuple writes but also by tuple reads. As a result, tuple reads in an NVM based OLTP engine can incur expensive NVM writes.

NVM Write Redundancy: OLTP databases typically rely on logs and checkpoints/snapshots to achieve durability. If an NVM based engine takes this approach, there will be substantial NVM write redundancy because the same content is written to the logs, the checkpoints/snapshots, in addition to the base tables. This redundancy not only takes more NVM space, but also negatively impacts the runtime performance.

NVM Space Management: First, NVM space allocation needs to be persistent across power failure. Hence, every NVM memory allocation and free may have to be protected by expensive NVM persistence operations. Unfortunately, OLTP transactions often perform non-trivial numbers of inserts, updates, and/or deletes, potentially incurring significant overhead. Second, NVM may have limited write endurance [29]. It is important yet challenging to remove hot spots in the NVM frequently allocated and freed.

In this paper, we propose Zen, a high-throughput log-free OLTP engine for NVM. Zen addresses the above three challenges with the following three new techniques. It provides general-purpose support for a wide range of concurrency control methods.

Metadata Enhanced Tuple Cache: We store base tables in NVM without per-tuple metadata. Then we propose to build an Met-Cache (Metadata enhanced tuple Cache) in DRAM to (i) cache tuples that are used in currently running transactions or have recently been used, and (ii) augment each tuple with per-tuple metadata required by concurrency control methods. In this way, Zen performs concurrency control mostly in DRAM, avoiding writing per-tuple metadata in NVM for tuple reads, and reduces NVM reads for frequently accessed tuples.

Log-Free Persistent Transactions: We eliminate NVM write redundancy by completely removing logs and checkpoints for transactions in our durability scheme. Each tuple in the base tables in

*Shimin Chen is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 14, No. 5 ISSN 2150-8097.
doi:10.14778/3446095.3446105

NVM has a tuple ID field and a Tx-CTS (Transaction Commit Timestamp) field. Tx-CTS identifies the transaction that produces the version of the tuple. At commit time, Zen persists modified tuples in a transaction from the Met-Cache to the relevant base tables in NVM. It writes to newly allocated or garbage collected space without overwriting the previous versions of the tuples. The most significant bit in Tx-CTS is used as a LP (Last Persisted) bit. After persisting the set of modified tuples in a transaction, Zen sets the LP bit and persists the Tx-CTS for the last tuple in the set. Upon failure recovery, Zen can identify if the modification of a transaction is fully persisted by checking if the LP bit is set for one of the tuples. If yes, then the new tuple versions will be the current versions. If no, then the transaction is considered as aborted, and the previous tuple versions are used.

Lightweight NVM Space Management: We aim to reduce the persistence operations for NVM space management as much as possible. First, we allocate large (2MB sized) chunks of NVM memory from the underlying system, and initialize the NVM memory so that Tx-CTS=0. Second, we manage tuple allocation and free without performing any persistence operations. This is because using the log-free persistence mechanism, Zen can identify the tuple versions that are most recently committed upon recovery. The old tuple versions are then put into the free lists. Third, the allocation structures are maintained in DRAM during normal processing. Zen garbage collects old tuple versions and puts them into free lists for tuple allocations. Each thread has its own allocation structures to avoid thread synchronization overhead.

The contributions of this paper are fourfold. First, we identify the main design principles for NVM based OLTP engines by examining the strengths and weaknesses of three state-of-the-art NVM based OLTP designs (§2). Second, we propose Zen, which reduces NVM overhead by three novel techniques, namely the Met-Cache, log-free persistent transactions, and light-weight NVM space management (§3 and §5). The three techniques push to the extreme of minimizing NVM writes: for every tuple write, the only NVM write is for the modified tuple itself. Third, we evaluate the runtime and recovery performance of Zen using YCSB and TPCC benchmarks on a real machine equipped with Intel Optane DC Persistent Memory. Experimental results show that Zen achieves up to 10.1x improvements over MMDB with NVM capacity, WBL, and FOEDUS, while obtaining almost instant recovery (§4). Finally, we prove the wide applicability of Zen by supporting 10 different concurrency control methods (§3 and §4).

2 BACKGROUND AND MOTIVATION

We provide background on NVM and OLTP, examine existing OLTP engine designs for NVM, then discuss the design challenges.

2.1 NVM Characteristics

There are several competing NVM technologies, including PCM [29], STT-RAM [39], Memristor [3], and 3DXPoint [1, 18]. They share similar characteristics: (i) NVM is byte-addressable like DRAM; (ii) NVM is modestly (e.g., 2–3x) slower than DRAM, but orders of magnitude faster than HDDs and SSDs; (iii) NVM provides non-volatile main memory that can be much larger (e.g., up to 6TB in a dual-socket server) than DRAM; (iv) NVM writes have lower bandwidth than NVM reads; (v) To ensure that data is consistent in NVM upon power failure, special persistence operations using cache line flush and memory fence instructions (e.g., `clwb` and `sfence`) are required to persist data from the volatile CPU cache to NVM, incurring significantly higher overhead than normal writes; and (vi) NVM cells may wear out after a limited number (e.g., 10^8) of writes.

From previous work on NVM based data structures and systems [4–6, 9–12, 17, 19, 20, 25, 26, 28, 33–35, 37, 38], we obtain three common design principles: (i) Put frequently accessed data structures in DRAM if they are either transient or can be reconstructed upon recovery; (ii) Reduce NVM writes as much as possible; (iii) Reduce persistence operations as much as possible. We would like to apply these design principles to the OLTP engine design.

2.2 OLTP in Main Memory Databases

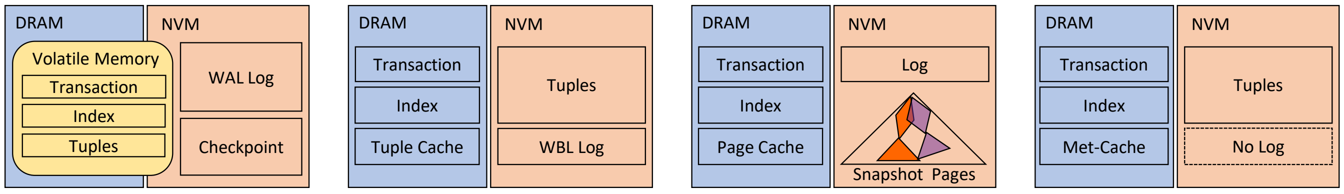
Main memory OLTP systems are the starting point to design an OLTP engine for NVM. We consider concurrency control and crash recovery mechanisms for achieving ACID transaction support.

Recent work has investigated concurrency control methods for high-throughput main memory transactions [15, 24, 27, 32, 36, 41]. Instead of using two phase locking (2PL) [7, 16], which is the standard method in traditional disk-oriented databases, main memory OLTP designs exploit optimistic concurrency control (OCC) [21] and multi-version concurrency control (MVCC) [7] for higher performance. Silo [32] enhances OCC with epoch-based batch timestamp generation and group commit. MOCC [36] is an OCC based method that exploits locking mechanisms to deal with high conflicts for hot tuples. Tictoc [41] removes the bottleneck of centralized timestamp allocation in OCC and computes transaction timestamps lazily at commit time. Hekaton [15] employs latch-free data structures and MVCC for transactions in memory. Hyper [27] improves MVCC for read-heavy transactions in column stores by performing in-place updates and storing before-image deltas in undo buffers. Cicada [24] reduces overhead and contention of MVCC with multiple loosely synchronized clocks for generating timestamps, best-effort inlining to decrease cache misses, and optimized multi-version validation. One common feature of the above methods is that they extend every tuple or every version of a tuple with metadata, such as read/write timestamps, pointers to different tuple versions, and lock bits for validation and commit processing. These methods have achieved transaction throughputs of over one million transactions per second (TPS) without persistence.

Similar to traditional databases, main memory databases (MMDB) store logs and checkpoints on durable storage (e.g., HDDs, SSDs) in order to achieve durability [8, 14, 22, 23, 30, 43]. The main difference resides in the fact that all the data fits into main memory in MMDBs. Hence, only committed states and redo logs need to be written to disks. After a crash, an MMDB recovers by loading the most recent checkpoint from durable storage into main memory, then reading and applying the redo log up to the crash point.

2.3 Existing OLTP Engine Designs for NVM

In this paper, we focus on the case where all data and structures of the OLTP engine can fit into NVM memory. We assume that the computer system contains both NVM and DRAM memory, which are mapped to different address ranges in the virtual memory of



(a) MMDB with NVM Capacity (b) WBL (c) FOEDUS (d) Zen (our proposal)

Figure 1: OLTP engine designs for NVM.

software. For example, this corresponds to the App Direct mode in 3DXpoint based Intel Optane DC Persistent Memory (OptanePM). A dual-socket server can have up to 6TB of OptanePM. The ratio P of NVM to DRAM capacity is typically 4–16 in OptanePM.

MMDB with NVM Capacity. As shown in Figure 1(a), MMDB can leverage the NVM capacity by treating part of NVM as slower *volatile* memory when the OLTP database is larger than DRAM. Like existing MMDB designs, the system stores tuples and indices in volatile memory, and processes transactions completely in volatile memory using normal load and store instructions. For durability, the system places the write-ahead logs (WAL) and checkpoints in NVM. It issues special persistence instructions (e.g., `clwb` and `sfence`) to persist log entries and checkpoints. After a crash, tuples and indices in volatile memory are considered lost. The recovery is based on the logs and checkpoints in NVM.

This design suffers from two drawbacks. First, a modified tuple is to be written to both the WAL and checkpoints, incurring two additional NVM writes for the tuple. If it is stored in the volatile part of NVM, the tuple is written three times in NVM. Second, as the database size increases, more and more tuples reside in NVM. Since per-tuple metadata is often modified even for tuple reads, read transactions still perform a large number of NVM writes.

Recent studies propose several improved logging schemes using NVM, including NV-Logging [19], Distributed-Logging [35], and WBL [6]. We discuss WBL in the following.

WBL. As shown in Figure 1(b), write-behind logging (WBL) [6] maintains indices and a tuple cache in DRAM. Tuples are fetched into the tuple cache for transaction processing. WBL supports multiple versions of a logical tuple in NVM by enhancing the tuple with per-tuple metadata, e.g., a transaction ID, commit timestamps, and a reference to previous version of the tuple. A committing transaction persists a modified tuple in the tuple cache by creating a new version of the tuple in NVM. In this way, the previous version of the tuple is available if a crash occurs at commit time. Unlike WAL, the WBL log does not contain modified tuple data. A log entry is written after a set of transactions commit. It contains a persisted commit timestamp (c_p), and a dirty commit timestamp (c_d). Since persist operations issue memory fence instructions (e.g., `sfence`), the existence of this log entry indicates that any transactions with a commit timestamp earlier than c_p must have successfully been persisted to NVM. Upon crash recovery, the system checks the last log entry, and undoes any transactions with a timestamp in (c_p, c_d). It rebuilds the indices in DRAM.

Compared to MMDB in Figure 1(a), WBL significantly reduces the log size and does not maintain checkpoints. Therefore, it writes a modified tuple exactly once to NVM, significantly decreasing the number of NVM writes. However, WBL maintains per-tuple metadata at every tuple in NVM. Therefore, it suffers from frequent per-tuple metadata modifications.

FOEDUS. As shown in Figure 1(c), FOEDUS [20] stores tuple data in snapshot pages in NVM, and employs a page cache in DRAM. The page index in DRAM maintains dual pointers for a page, i.e. a pointer to the page in the NVM snapshots, and a pointer to the page in the page cache (if it exists). FOEDUS runs transactions in DRAM. If the page containing a tuple required by a transaction is not in the page cache, the system loads the page into the page cache and updates the page index. At commit time, the system writes to the redo logs in NVM. A background log gleaner thread periodically collects logs and runs a map-reduce like computation to generate a new snapshot in NVM.

FOEDUS deals with transactions completely in DRAM, thereby avoiding per-tuple metadata writes in NVM. However, there are three significant problems of this design. First, the page granularity of caching results in NVM read amplification. A tuple read incurs the much larger overhead of a page read. Second, the sophisticated map-reduce computation causes many NVM writes. Finally, the FOEDUS implementation uses the I/O interface to access NVM, which does not take full advantage of the byte-addressable NVM.

3-tier Storage Manager with DRAM, NVM, and SSDs. Renen et al. proposes a 3-tier storage manager that uses DRAM and NVM as selective caches for data in SSDs [33]. Pages are loaded into DRAM from SSDs for DB accesses. When a page is evicted from DRAM, it can be placed into NVM for future reuse. In comparison to the 3-tier design, we assume that the OLTP database fits into NVM and propose an Met-Cache in DRAM for data in NVM. To our knowledge, 6TB of NVM is large enough for a significant number of OLTP applications. Exploiting SSDs to support even larger databases is an interesting direction in future work.

2.4 Design Challenges

Given the existing designs, we examine the three design challenges. **1. Tuple Metadata Modifications**: In MMDB and WBL, per-tuple metadata is stored with tuples in NVM. Unfortunately, concurrency control methods (e.g., OCC variants and MVCC variants) may modify the metadata even for tuple reads. **2. NVM Write Redundancy**: In MMDB and FOEDUS, a modified tuple is written to tuple heaps, logs, checkpoints, and/or page snapshots in NVM. The NVM write amplification can negatively impact transaction performance. **3. NVM Space Management**: WBL performs fine-grain space allocation for tuples. The WBL paper does not describe space management in detail. A naïve approach is to persist space allocation metadata to NVM (e.g., with logging) for every allocation and free calls. This may incur significant NVM persist overhead.

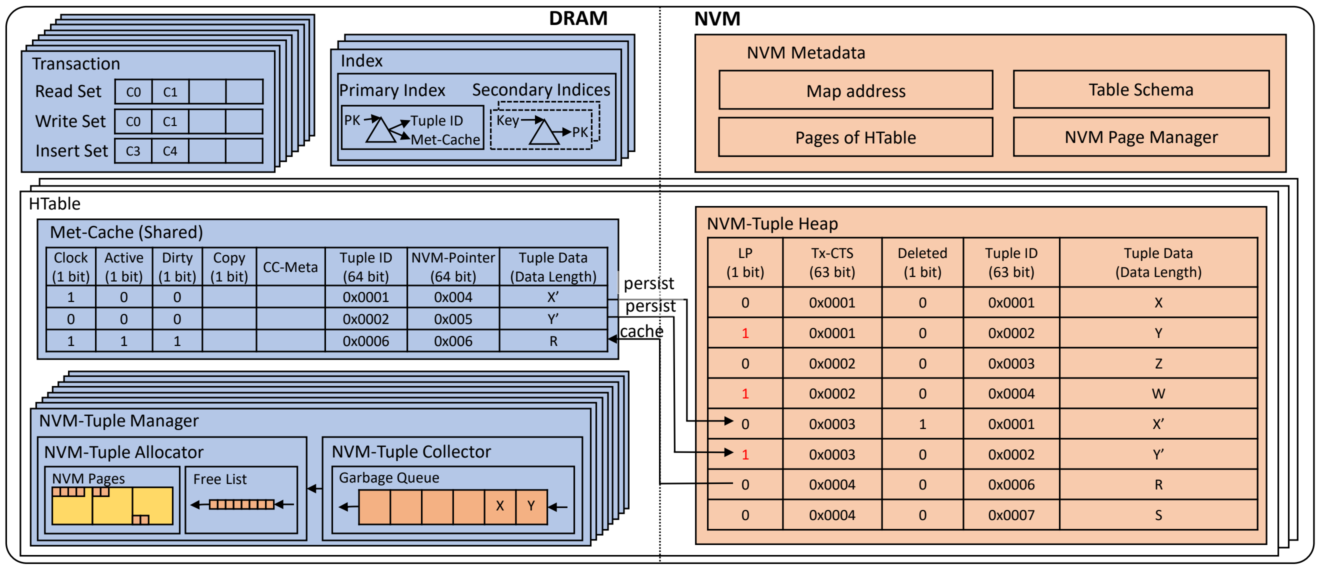


Figure 2: Zen architecture.

Figure 1(d) compares our proposed design, Zen, with the three existing designs side by side. First, Zen maintains the metadata enhanced tuple cache (Met-Cache) in DRAM. Unlike the page cache in FOEDUS, the granularity of Met-Cache is tuple. This avoids FOEDUS's NVM read amplification problem. Unlike WBL, Zen modifies per-tuple metadata *only* in the Met-Cache for concurrency control methods. Second, Zen completely removes logging. There is no NVM write amplification for tuple writes. Finally, Zen proposes a light-weight NVM space allocation design, which does no NVM persist operations for tuple allocations and frees.

3 ZEN DESIGN

We propose Zen, a high-throughput log-free OLTP engine for NVM. Zen supports OLTP databases much larger than DRAM, while addressing the three design challenges to achieve good performance for both transaction processing and crash recovery.

3.1 Design Overview

Figure 2 overviews the architecture of Zen. There is a hybrid table (HTable) for every base table. It consists of a tuple heap in NVM, an Met-Cache in DRAM, and per-thread NVM-tuple managers. Moreover, Zen stores table schemas and coarse-grain allocation structures in the NVM metadata. Furthermore, Zen keeps indices and transaction-private data in DRAM.

NVM-Tuple Heap. An NVM-tuple is a persistent tuple in NVM. Zen stores all tuples in a base table as NVM-tuples in the NVM-tuple heap. The heap consists of fixed-sized (e.g., 2MB) pages. Each page contains a fixed number of NVM-Tuple slots[1]. An NVM-tuple consists of a 16B header and the tuple data. The NVM-tuple heap may contain several versions of a logical tuple. The tuple ID and the transaction commit timestamp (Tx-CTS) uniquely identifies a tuple version. The deleted bit shows if the logical tuple has been deleted. The last persisted (LP) bit shows if the tuple is the last tuple persisted in a committed transaction. The LP bit plays an important role in log-free transactions (cf. Section 3.3). Note that the header contains no field-specific to particular concurrency control methods. The NVM-tuple slots are aligned to 16B boundaries so that an NVM-tuple header always resides in a single 64B cache line. In this way, we can use one `clwb` instruction followed by a `sfence` to persist the NVM-tuple header.

Met-Cache. The Met-Cache manages a tuple-grain cache in DRAM for the corresponding NVM-tuple heap. An Met-Cache entry contains the tuple data and seven metadata fields: a pointer to the NVM-tuple if it exists, the tuple ID, a dirty bit, an active bit to indicate that the entry may be used by an active transaction, a clock bit to support the cache replacement algorithm, a copy bit to indicate if the entry has been copied, and a CC-Meta field that contains additional per-tuple metadata specific to the concurrency control method in use. Zen supports a wide range of concurrency control methods (cf. Section 3.3.2). Using the Met-Cache, Zen performs concurrency control entirely in DRAM.

Indices in DRAM. We maintain indices for each HTable in DRAM. We rebuild the indices upon crash recovery. A primary index is required and secondary indices are optional. For the primary index, the index key is the primary key of a tuple. The value points to the latest version of the tuple in either (i) the Met-Cache or (ii) the NVM-tuple heap. We use an unused bit of the value to distinguish the two cases[2]. For secondary indices, the index value is the primary key of a tuple. Zen requires that the index structures support concurrent accesses, and transactions can see only committed index entries (previously modified by other transactions).

[1]In this paper, for simplicity, we assume that the tuple size is fixed, e.g., by allocating the largest lengths for `varchar` fields. Note that it is easy to extend our design to support variable-sized tuples. One can maintain multiple page types such that the tuple slots in a type-i page are 2^i bytes large. Then, a tuple of length L can be allocated in a type-i page such that $2^{i-1} < L \leq 2^i$.

[2]Only 48 bits in a 64-bit virtual memory address are used in current systems. Moreover, the highest bit is always 0 in user-mode programs.

Transaction-Private Data. Zen supports multiple threads that handle transactions concurrently. Each thread reserves a thread-local space for transaction-private data in DRAM. It records the transaction's read, write, and insert activities. OCC and MVCC variants maintain read, write, and insert sets as separate data structures. 2PL variants store the changes in the form of log entries.

NVM Space Management. Zen uses a two-level scheme to manage NVM space. First, the NVM page manager performs page-level space management. It allocates and manages 2MB sized NVM pages. The map address and the HTable pages in NVM metadata maintain the mapping from NVM pages to HTables. Second, NVM-tuple managers perform tuple-level space management. Each thread owns a thread-local NVM-tuple manager for each HTable that the thread accesses. Each NVM-tuple manager consists of an NVM-tuple allocator and an NVM-tuple collector. The allocator maintains a disjoint subset of free NVM-tuple slots in the HTable. There are two kinds of free slots: empty slots in newly allocated pages or garbage collected slots. We initialize NVM with all 0s at system setup time and use Tx-CTS=0 to indicate empty slots. The collector garbage collects stale NVM-tuples and puts them into the free list. All collectors of the same HTable work cooperatively to recycle NVM-Tuples.

3.2 Metadata Enhanced Tuple Cache

For a HTable, we divide its Met-Cache into multiple equal-sized regions, one per transaction processing thread. The NVM-tuple heap is also divided into per-thread regions. A thread is responsible for managing its Met-Cache region and its NVM-tuple heap region. It can read tuples in all regions, but can only write to its own region. For a cache hit, the thread can read the Met-Cache entry in any region. If the thread wants to modify a tuple in another Met-Cache region, it has to copy the entry into its own Met-Cache entry before modifying it. It sets the copy bit of the original Met-Cache entry. For a cache miss, the thread can bring an NVM-tuple into its own Met-Cache region. If there is no empty entry in the Met-Cache region, the thread has to pick and evict a victim tuple from its region to make space for caching the missed NVM-tuple. This design eliminates thread contention for managing Met-Cache entries, and supports the binding of DRAM and NVM address ranges to specific processor cores.

We employ the CLOCK algorithm for Met-Cache replacement. The algorithm picks as the victim the first encountered entry whose Active and Clock bits are both 0. If Active is set, the entry is being accessed by an active transaction. The algorithm skips such an entry so that it will not replace Met-Cache entries used by other running transactions. If Clock is set, the entry has been used recently. We would like to keep such entries in the cache. Active and clock bits are modified using atomic compare-and-swap instructions.

We decide the Met-Cache size (C_i) for HTable$_i$ given the available DRAM capacity (M), HTable$_i$'s size (S_i), and the average number (f_i) of tuples accessed in HTable$_i$ per transaction. Assuming accesses are uniformly distributed across a HTable, we can estimate the average number of Met-Cache hits per transaction as Met-Cache Hits $= \sum_i \frac{f_i C_i}{S_i}$. We would like to maximize the Met-Cache hits, while satisfying the DRAM capacity constraint: $\sum_i C_i \leq M$. Moreover, we would like to ensure that every HTable gets at least a minimum amount of cache space to support concurrency control in DRAM. That is, $C_i \geq C_{min}$. If we denote $C'_i = C_i - C_{min}$. The resulting problem is a knap-sack problem. We can employ the classical greedy algorithm by assigning cache space to the HTables according to the order of descending $\frac{f_i}{S_i}$.

Zen keeps no per-tuple metadata related to the concurrency control method in NVM-tuples. When an NVM-tuple is fetched from the NVM to Met-Cache, it is enhanced with the CC-Meta in the Met-Cache entry. CC-Meta contains per-tuple metadata specific to the concurrency control method in use. After that, Zen can run the concurrency control method entirely in DRAM because all the tuples accessed by active transactions are in Met-Caches. This design has the following benefits. (i) It shifts fine-grain per-tuple metadata reads and writes from NVM to DRAM. Hence, Zen enjoys fast per-tuple metadata accesses. (ii) Tuple reads will never lead to NVM writes at the NVM-tuples. (iii) Aborted transactions do not incur NVM-tuple write overhead. (iv) In-memory concurrency control decreases the time that a transaction spends in the critical code zone, whether acquiring critical resources or performing consistency validation. Consequently, the overall transaction abort rate may be reduced.

3.3 Log-Free Persistent Transactions

3.3.1 Normal Processing.

Transaction processing in Zen consists of three components: (i) Perform: Zen performs transaction processing in DRAM; (ii) Persist: Zen persists newly written tuples to NVM; (iii) Maintenance: Zen garbage collects stale tuples.

Figure 3 depicts the lifetime of a transaction. Suppose the table keeps account balances for customers. Initially, X has \$500, Y has \$100, and Z has \$100. The transaction transfers \$100 from X to Y and \$100 from X to Z. The upper part of Figure 3 shows the system state before the transaction. The NVM-tuple heap contains five tuples, among which R:d has been deleted and garbage collected. Q:300 is cached in Met-Cache. The index keeps track of the locations of the valid tuples. It points to the Met-Cache entry if exists, as in the case of Q. The allocator records the three empty NVM-tuple slots.

Perform. A transaction obtains a timestamp at start time. For each tuple requested by the transaction, the transaction looks up its location in the primary index. If the tuple is in NVM, the transaction finds a (victim) entry in Met-Cache with the cache replacement algorithm, builds the Met-Cache entry by reading the requested NVM-tuple and enhancing it with per-tuple CC-Meta specific to the concurrency control method in use, and updates the index with the Met-Cache entry location. Note that Zen does not need to write the victim entry to NVM for the following reasons. First, if the entry is only read by previous transactions, then it is not changed and can be discarded. Second, if the entry is generated/modified by a previously committed transaction, then it must have already been persisted to NVM at commit time. Third, if the entry is modified by an aborted transaction, it is invalid and should be discarded.

Zen runs concurrency control entirely in DRAM with the help of Met-Cache. If there is no conflict and the transaction can commit, Zen moves the transaction into Persist processing. If the transaction has to abort, Zen checks if any Met-Cache entry accessed by the transaction is dirty. For a dirty entry, Zen restores the entry from

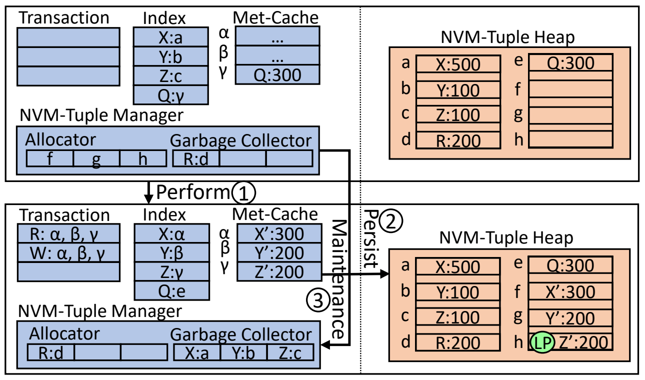


Figure 3: Lifetime of Transaction (X-=200; Y+=100; Z+=100).

the NVM-Tuple pointed by the NVM-Tuple pointer so that the retry of the transaction will find the entry in Met-Cache.

The lower part in Figure 3 shows the system state after the transaction. In Perform processing, Zen brings the three tuples requested by the transaction, i.e. X, Y, Z, into Met-Cache. The index is updated accordingly. The transaction modifies X to 300, Y to 200, and Z to 200 in Met-Cache. The transaction-private data keeps track of the read and write sets.

Persist. Zen persists the generated and modified tuples of a transaction to NVM *with no logs*. The challenge is to persist multiple tuples without writing redo log records and the commit log record. The basic ideas of our solution are as follows. First, we persist a tuple to a free NVM-tuple slot. In this way, the previous version of the tuple is intact during persist processing. Zen can fall back to the previous version in case of a crash. This idea has already been proven successful in WBL. Second, we mark the LP bit of the last tuple to persist in the transaction using an NVM atomic write. We ensure that all the tuples are persisted before persisting the LP bit. In this way, the LP bit plays the same role as a commit log record. During recovery, if the LP bit exists, then the transaction has committed. All the tuples generated/modified by the transaction must have been successfully persisted to NVM. Otherwise, the crash occurs in the middle of persisting the transaction. Therefore, Zen discards any NVM-tuples written by the transaction.

Algorithm 1 shows the persist processing for changed tuples. It persists all the tuples except the line that contains the header of the last tuple (Line 2-8). The cacheline size is 64B. The algorithm persists 64B lines occupied by a tuple using for-loops (Line 4-5 and 7-8). Note that as long as the tuples are flushed to NVM, the order of the flushes is not significant. Therefore, we need to issue only a single `sfence` (Line 9) to ensure that all previous `clwb`s complete. In the end, the algorithm sets the LP of the last tuple (Line 10), and flushes the line that contains the header of the last tuple (Line 11). Note that the header must reside in a single 64B line because NVM-tuple slots are 16B aligned and the header is 16B large.

Interestingly, the algorithm is optimized to not issue `sfence` after the last `clwb`. This is correct because recovery processing can correctly handle either case where LP is set or not, as discussed in the above. Moreover, any `sfence` later issued by this or other thread will ensure the last `clwb` in the algorithm completes. For example, a communication thread can issue a `sfence` before communicating a set of transaction results to database clients.

Algorithm 1: Persist Processing

```
1  Function persistTuples(changed-tuples)
2      for (i=0; i<changed-tuples.size-1; i++) do
3          tuple= changed-tuples[i];
4          for (p=tuple.start; p<tuple.end; p+=64) do
5              clwb(p); /* clwb tuple */
6      last-tuple= changed-tuples[changed-tuples.size-1];
7      for (p=last-tuple.start+64; p<last-tuple.end; p+=64) do
8          clwb(p); /* clwb last-tuple except the first line */
9      sfence();
10     last-tuple.LP= 1;
11     clwb(last-tuple.start); /* clwb last-tuple's header */
```

As shown in the lower-right part of Figure 3, Zen persists the newly modified tuples X', Y', and Z' using the three empty NVM-tuple slots f, g, and h. Z' is the last tuple to persist. Therefore, Zen sets and flushes the LP bit in the header of Z' after persisting X', Y', and all but the first line of Z'.

Maintenance. To reduce contention, each thread has its private NVM-Tuple allocator and garbage queue. A thread garbage collects an NVM-tuple version when it finds that a more recent version exists. The garbage collection decision is made in two situations. First, when it commits a transaction that overwrites a tuple, the thread garbage collects the old NVM-tuple version unless the Met-Cache entry is copied from another region. Second, before it evicts an entry *E* from its Met-Cache region, the thread garbage collects the NVM-tuple pointed by *E* if *E*'s copy bit is set. Note that *E* must have been copied to another region by a committed transaction *T*, and *T* must have written a new version of the tuple[3]. In this way, a thread garbage collects NVM-tuples only in its own region, and an old tuple version is eventually garbage collected.

Entries in the garbage queue cannot be directly freed because the related NVM-tuple versions may still be used by other transactions (e.g., in MVCC). An entry contains the NVM-tuple pointer and its Tx-CTS. Zen computes a global minimum Tx-CTS periodically by taking the min of the last committed transaction's Tx-CTS in every thread. Hence, there are no running transactions that access entries whose Tx-CTS < the minimum Tx-CTS. Such entries can be safely moved from the garbage queue to the allocator free list.

As shown in the lower-left part of Figure 3, Zen puts the old versions of X, Y, and Z into the garbage queue. Moreover, Zen moves the R:d entry from the garbage queue to the allocator free list when it finds that the entry's Tx-CTS < the minimum Tx-CTS.

3.3.2 *Flexible Support for Concurrency Control Methods.*

The above transaction processing design provides a framework to flexibly support wide varieties of concurrency control methods. We show the applicability of Zen to 10 concurrency control methods in our experiments in Section 4.4, including three 2PL variants (2PL with deadlock detection, wait and die, and no waiting [40]), three OCC variants (OCC [21], Silo [32], and Tictoc [41]), three MVCC variants (MVCC [7], Hekaton [15], and Cicada [24]), and a partition-based method (HStore [31]). To support a concurrency

[3]Note that *T* must have committed. If *T* were running, then *E*'s active bit should be 1 and it could not be chosen as the victim. If *T* had aborted, then *T* would have cleared *E*'s copy bit.

control method, we adapt the CC-Meta field of Met-Cache entries to hold per-tuple metadata required by the method. For 2PL variants, CC-Meta stores the locking bits. For OCC variants, CC-Meta can include write timestamp, read timestamp, write lock bit, and/or latest version bit. For MVCC variants, CC-Meta often contains multiple timestamps and version link pointers. The concurrency control method can process the metadata-enhanced tuples in Met-Cache entirely in DRAM.

We consider the support of versions. Concurrency control methods can be divided into two categories: single-version methods and multi-version methods. Note that it is Met-Cache that supports the versions required by concurrency control methods. NVM-tuple heap supports multiple versions for the purpose of removing redo log. As described in Section 3.3, committed versions of NVM-tuple are always persisted to NVM. This is regardless of the number of versions in Met-Cache. For single-version methods, Met-Cache holds a single version for a tuple. While there can be multiple committed NVM-tuple versions in NVM-tuple heap, only the latest version can be cached in Met-Cache. For multi-version methods, Met-Cache holds all the versions that are actively accessed by running transactions. A transaction will create a new version in Met-Cache for an overwrite. The cache replacement algorithm will not replace these Met-Cache entries because their active bits are set. Zen clears the active bit of an Met-Cache entry during garbage collection when the entry's Tx-CTS < the global minimum Tx-CTS. This guarantees that all the tuple versions that are used by any running transactions are kept in Met-Cache. The multi-version methods typically maintain a linked list for the active versions of the same logical tuple in Met-Cache. The primary index points to the most recent version. Old active versions can be found in the version linked list.

3.3.3 *Crash Recovery without Logs.*

After a crash, the data structures in DRAM are lost, including indices, NVM-tuple managers, Met-Caches, and transaction-private data. We need to reconstruct the indices and the tuple-level NVM space management structures in NVM-tuple managers. Met-Caches and transaction-private data do not need to be recovered. NVM persisted data include the NVM metadata (i.e. table schemas and metadata for page-level NVM space management) and committed NVM-tuples in NVM-tuple heaps.

Figure 4 depicts an example NVM-tuple heap after a system failure. We see that the heap contains tuples written by four transactions, i.e. 1000, 1003, 1015, and 1016. The LP bits of (tupleID, Tx-CTS)=(101,1000) and (102,1003) are set. Thus, transaction 1000 and 1003 have committed. However, the other two transactions have not completed because the LP bits of their tuples are all 0.

During recovery, Zen runs multiple threads. Each thread scans an NVM-tuple heap region. A naïve algorithm scans the region twice. The first scan computes the maximum committed transaction timestamp by examining the LP bits. Then the second scan identifies all the committed tuples by comparing their timestamps with the maximum timestamp. We propose an improved algorithm in Algorithm 2 to avoid scanning the data twice. The basic idea is to use the maximum timestamp seen so far to identify as many committed tuples as possible. Only uncertain cases need to be revisited again. We find that the average number of revisits is $O(log(n))$, where n is the number of NVM-tuple slots in the region.

LP	Tx-CTS	Deleted	Tuple ID	Data
1	1000	0	101	X
0	1000	0	102	Y
0	1015	0	103	Z
0	1000	1	104	V

LP	Tx-CTS	Deleted	Tuple ID	Data
0	1016	0	101	X''
0	1016	0	102	Bad Y''
0	1003	0	101	X'
1	1003	0	102	Y'

Figure 4: An NVM-tuple heap region after failure.

Algorithm 2: Scan NVM-tuple heap region

```
1  Function updateIndexGC(ntup)
2      ptup= searchIndex(ntup);
3      if (ptup == NULL) then
4          insertIndex(ntup); return; /* no existing version */
5      if (ntup.Tx-CTS < ptup.Tx-CTS) then
6          putIntoFreeList(ntup); return; /* ntup is old */
7      updateIndex(ntup); putIntoFreeList(ptup); /* ptup is old */
8  Function scanRegion(region)
9      ts-commit= 0; pending-list= {};
10     foreach (ntup ∈ region) do
11         if (ntup.LP) then
12             ts-commit= max(ts-commit, ntup.Tx-CTS);
13         if (ntup.Deleted or ntup.Tx-CTS == 0) then
14             putIntoFreeList(ntup); continue;
15         if (ntup.Tx-CTS ≤ ts-commit) then
16             updateIndexGC(ntup); /* transaction committed */
17         else
18             put ntup into pending-list; continue; /* uncertain */
19     foreach (ntup ∈ pending-list) do
20         if (ntup.Tx-CTS ≤ ts-commit) then
21             updateIndexGC(ntup); /* transaction committed */
22         else
23             /* Crash occurs at commit time. Mark the slot empty */
               putIntoFreeList(ntup); ntup.Tx-CTS=0; clwb(ntup);
24     sfence();
```

Algorithm Description. Algorithm 2 uses ts-commit to compute the maximum committed timestamp seen so far. Zen updates ts-commit whenever it encounters a tuple with LP set (Line 11-12). If a tuple's timestamp ≤ ts-commit, Zen considers the associated transaction has committed. Zen updates the index with the tuple (Line 16). If the index contains a version of the tuple, Zen compares the current tuple with the version in the index. Zen keeps the new version in the index, and puts the old version (if exists) into the free list (Line 3-7). When a tuple's timestamp > ts-commit, Zen cannot tell the state of the associated transaction at this moment. It puts the tuple into a pending list (Line 18).

After scanning the region, ts-commit is the maximum committed timestamp in this region. Then, Zen revisits the tuples in the pending list. If a tuple's timestamp ≤ ts-commit, then Zen updates the index with the tuple and possibly garbage collects an old version of the tuple in the index (Line 21). If a tuple's timestamp > ts-commit, the crash must occur when the associated transaction is being persisted. Since a thread can write to only its own region, all the tuple writes of a transaction go to the same region. This means the scan has seen all the tuples written by the transaction, but none of them has the LP set. Therefore, the transaction has not completed. Zen discards the tuple by marking the tuple slot empty (with Tx-CTS=0) and puts it into the garbage queue.

Correctness. Algorithm 2 correctly identifies all committed tuples in the region. First, if Tx-CTS ≤ ts-commit, then transaction Tx-CTS has committed. This is because the tuples in the region are written by a single thread, and the transaction timestamp of the same thread monotonically increases (though timestamps across different threads may not have a total order in certain concurrency control schemes). Second, the algorithm performs the checking in the main scan loop, then it checks the uncertain pending cases again. As a result, all the committed tuples are identified.

Moreover, the algorithm correctly reconstructs the index. It calls `updateIndexGC` for committed tuples that are not deleted, which puts the latest version of the tuple in the index. The algorithm also collects all the unused NVM-tuple slots (i.e. old tuple versions, deleted tuples, and empty tuple slots) into the free list.

Furthermore, Algorithm 2 is idempotent. It does not modify committed tuples. It marks uncommitted tuples as empty slots. As a result, when there is a crash during recovery, we can re-run the algorithm to compute the same ts-commit, and rebuild the index and the free list in the same way.

Finally, we consider the case where a crash occurs, the system recovers and processes transactions for a while, then a second crash occurs. The normal transaction processing and the recovery after the second crash will not see any uncommitted tuples resulted from the first crash because they have been marked as empty slots.

Efficiency. A tuple in the pending list is examined twice in Algorithm 2. Therefore, the size of the pending list decides the benefit of the proposed algorithm compared to the naïve algorithm. We can prove the following theorem, which shows that the pending list is quite small.

THEOREM 3.1. *The size L of the pending list is $O(ln(n))$ on average, where n is the number of NVM-tuple slots in the region.*

3.3.4 Support for Long Running Transactions.

A long running transaction that updates a lot of tuples may prevent the garbage collector from freeing up the tuples on NVM, and may exceed the Met-Cache capacity. Zen identifies a long running read-write transaction and switches to the exclusive mode to run the transaction.

Detecting Long Running Transactions. A long running transaction is detected if one of the following conditions is true: (i) an Met-Cache region runs out of available entries; or (ii) the number of entries in a garbage queue is beyond a pre-defined threshold (The threshold shows the NVM space is about to be used up). Zen finds the running transaction that accesses the largest number of tuples as the long running transaction.

Exclusive Mode. Zen has a global exclusive flag (G-EX). When a long running transaction is detected, Zen atomically sets G-EX. Each thread checks G-EX periodically. If it sees that G-EX is set, a thread aborts its transaction if the transaction is not the long running transaction. The long running transaction waits for all other transactions to complete aborting. Then it resumes execution in the exclusive mode. In the exclusive mode, the transaction enjoys all the resources in the system. After it completes, the transaction atomically clears G-EX. Then Zen threads resume normal execution.

3.4 Lightweight NVM Space Management

Our two-level NVM space management design incurs little NVM persist overhead. First, only the page-level manager persists metadata. Since we allocate 2MB NVM pages, the persist operations for recording the page allocation and page-to-HTable mapping in NVM are infrequent. Second, the tuple-level manager performs garbage collection and NVM-tuple allocation entirely in DRAM without accessing NVM during normal processing. This is feasible because the writing of a committed tuple serves the purpose of marking the NVM-tuple slot as occupied. We do not need to record separate per-tuple metadata in NVM for tuple allocations. During crash recovery, Zen scans the NVM-tuple heap and is able to determine the state of each NVM-tuple slot by examining its header, as described in Section 3.3.3. Consequently, Zen completely removes the cost of NVM write and persist operations for tuple-level NVM allocation.

We design the NVM-tuple manager to be decentralized to decrease thread contention. Each thread manages its own NVM-tuple heap region. It allocates NVM-tuple slots from its region. It collects garbage and frees NVM-tuple slots in its region. When the free list is empty, and there is a tuple allocation request, the NVM-tuple manager asks the NVM page manager to allocate a new 2MB NVM page. It divides the newly allocated page into empty slots and put them into the free list. As describe previously, empty slots' Tx-CTS are 0 since NVM space is initialized with 0 at setup time.

Two implementation details help reduce the impact of garbage collection on individual transaction latencies. (i) The per-thread garbage queue and free list are implemented in DRAM without any NVM overhead. Garbage collection does not have thread contention. (ii) We limit the number of items to scan in the garbage queue per transaction unless NVM space is used up. This bounds the impact of garbage queue scan on the latency of a single transaction.

Moreover, Zen persists a tuple to a location different from its previous version in NVM. This helps wear-leveling for hot tuples because Zen decreases hot spot writes in NVM.

4 EVALUATION

We run real-machine experiments to compare the performance of Zen with existing OLTP engine designs for NVM in this section.

4.1 Experimental Setup

Machine Configuration. The machine is equipped with 2 Intel Xeon Gold 5218 CPUs (16 cores/32 threads, 32KB L1I, 32KB L1D, and 1MB L2 per core, and a shared 22MB L3 cache). There are 384GB(12x32GB) DRAM and 1.5TB(12x128GB) 3DXPoint based Intel Optane DC Persistent Memory NVDIMMs in the system. We configure the system to run in the App Direct mode where both NVM and DRAM can be mapped to the virtual address of software. The machine runs Ubuntu 18.04.3 LTS with the 4.15.0-70-generic Linux kernel. We install file systems with fs-dax mode to the NVM, then use libpmem in PMDK to map NVM files to the virtual memory of a process. We issue `clwb` and `sfence` to persist data to NVM. All code is written in C/C++ and compiled with gcc 7.5.0. To avoid NUMA effects, by default, we run the experiments on a single CPU socket with its associated NVM and DRAM. There are 768GB NVM and 192GB DRAM available to a single CPU socket. In our experiments, we set the NVM size according to the database sizes in the benchmarks, and vary the DRAM size to model the machines with different ratio P of NVM to DRAM. Presently, P can be 4, 8, and 16 for OptanePM [1]. For space constraint, we report results with P=4 or 16. Results with P=8 show similar trends.

OLTP Engine Designs to Compare. We compare the following four designs: (i) MMDB with NVM capacity (*mmdb*), (ii) Write-behind logging (*wbl*), (iii) FOEDUS (*foedus*), and (iv) Zen (*zen*).

We control the NVM usage to a specific size by using `pmem_mmap`. We limit the DRAM usage by allocating a large DRAM from the system and manage DRAM space by ourselves. Note that the engines run in main memory without accessing any data files on disk. Therefore, they cannot leverage other DRAM space available in the system, such as the OS page cache. We keep the indices and transaction-private data in DRAM, and adapt the size of other data structures (e.g., Met-Caches) to the remaining DRAM. For FOEDUS [20], we obtain the code from the author, and modify it to store logs and snapshots in real NVM hardware. FOEDUS implements its own concurrency control method. For MMDB, WBL, and Zen, we write two implementations based on Cicada [24] and DBx1000 [40], respectively. We measure transaction processing and crash recovery performance using the Cicada based implementations. Then, we use the DBx1000 based implementations to demonstrate the applicability of our design to other 9 concurrency control methods besides Cicada. For MMDB, we optimize the logging procedure to combine the log records of a transaction, and write them together at commit time using sequential NVM writes, `clwb`s, and a single `sfence`. We implement decentralized logs to reduce contention. That is, each thread writes its log to a separate NVM space. When the database cannot fit into DRAM, MMDB uses part of NVM as volatile memory to store base tables. We disable checkpoints when measuring transaction throughputs of MMDB. For WBL, our implementation follows the description of the WBL paper closely for persisting tuples, writing WBL logs, and recovery. It writes a WBL log record roughly every 100us. We apply the light-weight NVM space management to WBL. The Zen design is described in detail in Section 3.

Benchmarks. we run YCSB [13] and TPCC [2] benchmarks.

YCSB is a widely used key-value workload representative of transactions handled by web companies. In our experiments, the YCSB database consists of a single table. Every tuple contains an 8B primary key and ten 100-byte columns of random string data. The size of a tuple is approximately 1KB. Each YCSB transaction consists of 16 random requests by default. Given the primary key, a read request retrieves a tuple, and a write request modifies a tuple. We vary two parameters in the workload: (1) Percentage of read requests: Read-Only(RO, 100% read), Read-Heavy (RH, 90% read, 10% write), Balanced (BA, 50% read, 50% write), and Write-Heavy (WH, 10% read, 90% write); and (2) The θ parameter of the Zipfian distribution: No-Skew ($\theta = 0$), Low-Skew ($\theta = 0.6$), and High-Skew ($\theta = 0.95$). Note that No-Skew has no request locality. It models the worst case scenario for cross visiting different Met-Cache regions. High-Skew models the scenario of high transaction contentions. We use a 256GB YCSB database in most experiments so that the database can fit into the NVM, but is larger than the available DRAM. We use another 100GB YCSB database in the recovery experiments to understand the impact of data size on recovery performance.

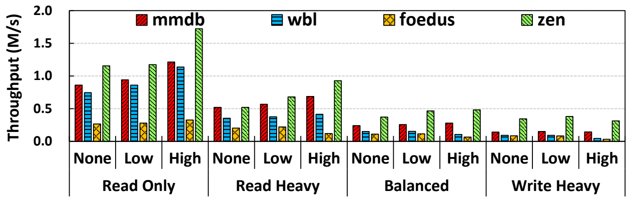


Figure 5: YCSB performance with P=4 and 16 threads.

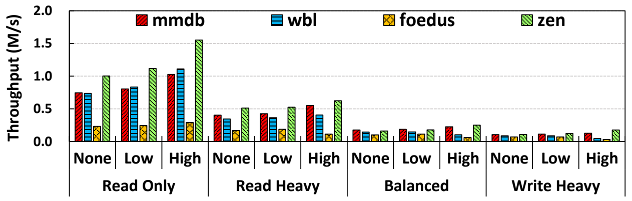
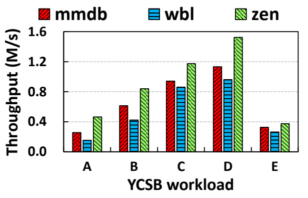


Figure 6: YCSB performance with P=16 and 16 threads.



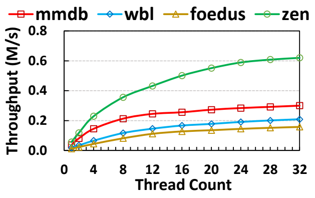


Figure 7: Standard workloads. (Low skew, P=4, 16 threads) **Figure 8: YCSB scalability. (High skew, Balanced, P=4)**

TPCC simulates an order-entry application of a wholesale supplier. There are five transaction types. Among the five types, New-Order and Payment transactions modify the database and account for 88% of all transactions. We conduct New-Order and Payment transactions (TPCC-NP) in our experiments. The ratio of New-Order transactions to Payment transactions is 45:43 as specified in TPCC. We configure the benchmark to use 2048 warehouses and 100,000 items. The initial footprint of the database is about 205GB.

4.2 Transaction Performance

4.2.1 *YCSB Performance.*

Varying Read/Write Ratio and Data Skew. We run the YCSB benchmark while varying the percentage of read requests and the Zipf θ parameter. The main result is shown in Figure 5.

Among the four OLTP engines, FOEDUS has the worst performance. It suffers from the NVM read amplification problem due to page-grain caching. Moreover, the map-reduce computation to merge logs to snapshots incur computation overhead as well as NVM write cost. Finally, the implementation uses heavy-weight file system interface and persists pages to NVM with `msync`. As a side effect, we do not count `clwb` and `sfence` for FOEDUS.

WBL gives the second worst performance. WBL maintains per-tuple metadata in NVM. Hence, it incurs a large number of NVM writes and persists for the per-tuple metadata. This is confirmed by Figure 9 and 10. First, WBL sees drastically more `sfence`s than MMDB and Zen. Second, when the workload has high skews, WBL sees significantly more `clwb`s than MMDB and Zen.

MMDB achieves better performance than WBL. Unlike WBL, MMDB considers the database to be in volatile memory. Therefore, it does not persist per-tuple metadata. The main overhead is the NVM write amplification problem. If a tuple is in the (volatile) part

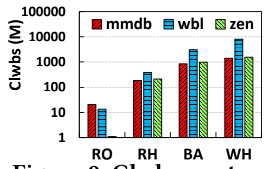


Figure 9: Clwb counts.

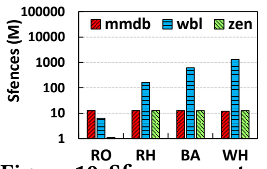


Figure 10: Sfence counts.

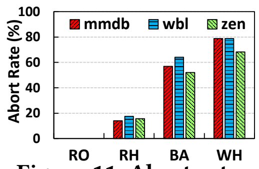


Figure 11: Abort rate.

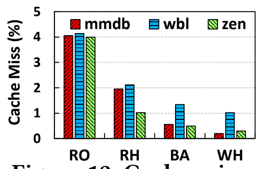


Figure 12: Cache miss.

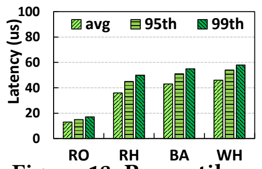


Figure 13: Percentiles.

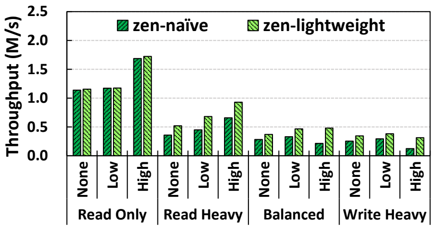


Figure 14: NVM space management.

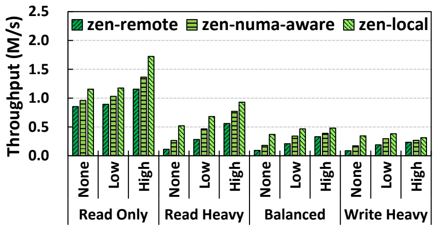


Figure 15: NUMA effects.

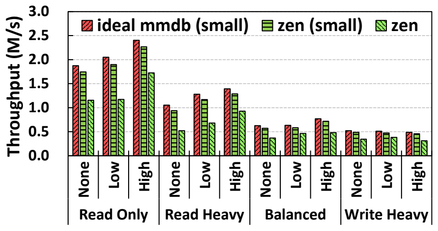


Figure 16: Zen vs. ideal MMDB.

(Figure 9–12: High skew, P=4, 16 threads; Figure 13: No skew, P=4, 16 threads Figure 14–16: P=4, 16 threads;)

of NVM, it is written to NVM twice, i.e. to the (volatile) part of NVM and to the log in NVM. (Note that this set of experiments do not perform checkpoints.)

Zen achieves better performance than MMDB mainly because Zen performs fewer NVM writes. For a committed transaction, Zen performs 1 NVM write per tuple write, and 0 NVM write for tuple reads. In the case of MMDB with NVM capacity, when $P = 4$, 75% of data reside in NVM. MMDB writes per-tuple concurrency control metadata even for tuple reads. A tuple write also creates a new version and incurs logging on NVM. Hence, MMDB performs an average 0.75×(1+1)+1=2.5 NVM writes per tuple write and 0.75×1=0.75 NVM write per tuple read. This explains the significant advantage of Zen over MMDB for Read Only or No Skew/Low Skew cases. For an aborted transaction, Zen frees resources without writing to NVM. In contrast, MMDB still writes the per-tuple metadata and new tuple versions. Hence, it incurs an average 1.5 NVM writes per tuple write and 0.75 NVM write per tuple read. This explains why Zen out-performs MMDB under High Skew.

Our proposed design, Zen, achieves the best performance. Compared with MMDB, WBL, and FOEDUS, Zen achieves 1.25–5.29x speedup for Read-Only, 1.00-7.86x speedup for Read-Heavy, 1.54x-7.50x speedup for Balanced, and 2.16x-10.12x speedup for Write-Heavy. Zen successfully addresses the three design challenges with Met-Cache, log-free transactions, and light-weight NVM space management. As shown in Figure 9, Zen issues much fewer `clwb` instructions than MMDB and WBL for Read-Only because Zen does no NVM writes for read requests, while MMDB and WBL still need to persist their logs. As shown in Figure 10, Zen issues at most one `sfence` per transaction. This is the same as MMDB, but much fewer than WBL, which writes and persists per-tuple metadata to NVM. Furthermore, the speedups of Zen over the other designs increase as the workload becomes more write intensive. This shows the benefit of Zen in reducing overhead for write requests.

We see two general trends for all the engine designs. First, transaction throughputs increase as the percentage of read requests because there are fewer NVM writes and persists as shown in Figure 9. Second, higher skews bring two effects. Higher skews give better performance for Read-Only, Read-Heavy, and Balanced because more data are found in DRAM. However, higher skews result in more contention for Write-Heavy as shown in Figure 11. As a result, we see lower transaction throughputs. Interestingly, Zen has fewer aborts in Balanced and Write-Heavy for skewed workload compared to MMDB and WBL. As writes become more, Met-Cache may be more frequently updated. As a result, Zen has better cache performance as shown in Figure 12. We attribute the reason to our fine-grained Met-Cache because the cache provides data lower accesses latency for hot tuples, which makes the process to detect conflicts faster. In average, Zen spends less time in critical region for concurrency control. Hence, transactions in Zen face less conflicts because the writes keep the Met-Cache updated in time.

Varying NVM/DRAM Size Ratio. We show YCSB transaction performance for P=16 in Figure 6. Zen is best performing OLTP engine design among the four designs. Compared with MMDB, WBL, and FOEDUS, Zen achieves 1.34-5.35x speedup for Read-Only, 1.13-5.59x speedup for Read-Heavy, 1.02x-4.20x speedup for Balanced, and 1.02x-5.58x speedup for Write-Heavy. We also observe similar trends as P=4 compared with Figure 5. Moreover, as P increases and DRAM becomes smaller compared to NVM, all engine designs see decreasing throughput because more accesses have to visit NVM. Furthermore, for the case of P=16, Write-Heavy or Balanced, and No Skew, Zen and MMDB show similar performance because the bottleneck is the NVM persist operations. Zen persists a modified tuple to NVM-tuple heap, while MMDB persists the tuple to the log. Both designs issue a single `sfence` per transaction.

Standard YCSB Workloads. We run experiments for the five standard workloads of YCSB [13] under low skew, P=4, and 16 threads: (A) 50% read, 50% update; (B) 95% read, 5% update; (C) 100% read; (D) 95% read recent, 5% insert; and (E) 95% scan, 5% insert. Note that the Read Only case in the previous experiments is workload (C), and the Balanced case is workload (A). Figure 7 compares the performance of MMDB, WBL, and Zen under the five standard YCSB workloads. We see that Zen achieves 1.15–1.82x improvements over MMDB, and 1.36–3.04x improvements over WBL.

YCSB Scalability. We study the scalability of the four OLTP engine designs in Figure 8. We set P=4 and use Balanced, High Skew requests in the experiments. We vary the number of threads from 1 to 32. From the figure, we see that Zen scales up better than MMDB, WBL, and FOEDUS. First, Zen conducts concurrency control completely in DRAM. Second, the Met-Cache hit rate is 85% for High

Skew workloads. This makes Zen's efficiency close to that of pure in-memory database. Third, Zen incurs less cost for aborts because an aborted transaction does not write to NVM.

Benefit of Light-Weight NVM Space Management. We compare the transaction performance of Zen with and without the light-weight NVM space management in Figure 14. The naïve design records and persists the metadata of every tuple allocation in NVM. From the figure, we see that the two designs have similar performance for Read-Only because there is few NVM-tuple allocation requests. On the other hand, Zen significantly out-performs the naïve design for the other cases. Compared to the naïve design, Zen achieves 1.41–1.52x speedup for Read-Heavy, 1.32–2.26x speedup for Balanced, and 1.30–2.52x speedup for Write-Heavy. Moreover, Figure 13 studies the impact of garbage collection activities on transaction latencies. Note that we choose no skew to minimize the impact of transaction conflicts and aborts. The figure shows that 99th percentile latencies are only slightly larger than the average latencies, indicating that garbage collection works smoothly.

NUMA effects. We demonstrate the NUMA effects in Figure 15. Zen-local represents the Zen results in previous experiments, where 16 threads run on a single CPU socket using the affiliated local NVM and DRAM. Zen-remote runs 16 threads on CPU socket 0 but uses remote NVM and DRAM affiliated to socket 1. Zen-numa-aware runs 8 threads on CPU socket 0, and 8 threads on CPU socket 1. The threads allocate Met-Cache regions and NVM-tuple heaps from their affiliated local DRAM and NVM. A thread may perform remote reads but always write to its local Met-Cache and NVM-tuple heap. As shown in Figure 15, zen-local is the best performing configuration. Using NUMA-aware designs, zen-numa-aware achieves 1.12–2.34x improvements over zen-remote.

Comparison to ideal MMDB. We compare the performance of Zen with ideal MMDB in Figure 16. For ideal MMDB, we reduce the database size to 100GB and fit it into DRAM, then we run MMDB without checkpointing. In this way, ideal MMDB performs no NVM reads, and almost optimal number of NVM writes and persists for write requests. Zen's database is 256GB large. Zen (small) has a 100GB database as MMDB. Both Zen (small) and Zen can use 64GB DRAM. As shown in Figure 16, we see that Zen (small) is only slightly (6%–11%) slower than ideal MMDB, showing the benefits of our proposed optimization techniques. Note that Zen (small) performs better than Zen because a larger fraction of tuples of Zen (small) are in Met-Cache.

4.2.2 *TPCC-NP Performance.*

TPCC-NP performance. We run the TPCC-NP benchmark while varying the memory configuration and using 16 threads. As shown in Figure 17, the TPCC-NP experiment shows the same trend as the YCSB experiment. Zen is best performing OLTP engine design among the four designs. Compared with MMDB, WBL, and FOEDUS, Zen achieves 1.44x–3.92x speedup when P=4, 1.67x–4.61x speedup when P=8, and 1.65x–4.77x speedup when P=16.

TPCC-NP scalability. We study the scalability of the 4 OLTP engine designs using TPCC-NP benchmark. We set P=4 and vary the number of threads from 1 to 32. As shown in Figure 18, all designs scale well to 32 threads. We attribute the good scalability to the modestly low contention in the TPCC-NP workload. Zen performs

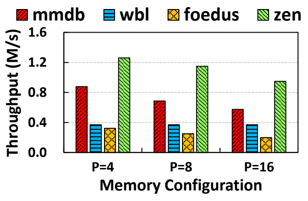


Figure 17: TPCC-NP performance. (16 threads)

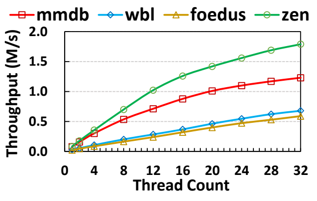
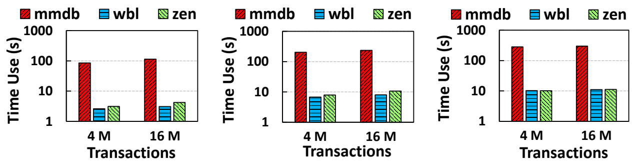


Figure 18: TPCC-NP scalability. (P=4)



(a) YCSB 100GB (b) YCSB 256GB (c) TPCC-NP 205GB

Figure 19: Recovery performance. (P=4, 16 parallel threads)

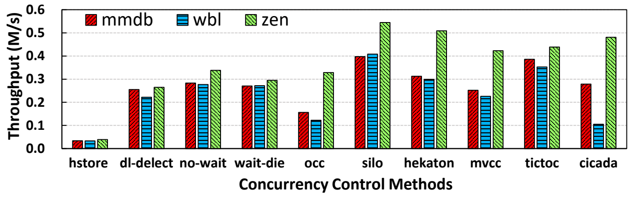


Figure 20: YCSB performance with 10 concurrency control methods. (P=4, High Skew, Balanced, 16 threads)

the best in all designs. Zen achieves a transaction throughput of 1.8 million transactions per second using 32 threads.

4.3 Recovery Performance

In this section, we evaluate the recovery time of the OLTP engine designs using YCSB and TPCC benchmark. For each benchmark, we first execute a fixed number of transactions, then force a hard shutdown of the DBMS (SIGKILL). After that, we measure the time for the system to restore to a consistent state, where the effects of all committed transactions are durable and the effects of the uncommitted transactions are removed. We configure the ratio of NVM to DRAM capacity to be 4 and use 16 parallel threads for recovery processing. We consider MMDB, WBL, and Zen for recovery performance. We omit FOEDUS because it has the worst transaction performance and the implementation does not provide a straightforward way to perform recovery. For MMDB, we assume that there is a checkpoint before running the benchmark. We do not take more checkpoints during the benchmark. Therefore, the recovery process reads the checkpoint and applies redo logs to bring the database state up to date. For WBL, we read the WBL to obtain the pairs of persisted commit timestamp (c_p), and dirty commit timestamp (c_d). Then we scan the tuples in NVM while comparing the tuple timestamps with the (c_p, c_d) pairs to identify committed tuples. The reconstruction of indices is similar to Zen.

Recovery for YCSB. We use two database sizes, i.e. 100GB and 256GB, in the YCSB recovery experiments. We run 4 million and 16 million transactions before the system failure. As shown in Figure 19(a), MMDB takes 85.9 seconds to recover from the system failure for the 100GB database. In contrast, WBL and Zen take 2.6 seconds and 3.1 seconds for recovery, respectively. They are orders of magnitude faster than MMDB. MMDB spends most time in loading the checkpoint and redoing logs. WBL and Zen spend most time in scanning tuples in NVM and restoring the indices. However, because of the uncertainty of the maximum committed transaction timestamp, Zen needs to check the LP flag, Deleted flag, and Tx-CTS for each NVM-Tuple, which accounts for additional time compared with WBL. When we increase the number of transactions from 4 million to 16 million, MMDB takes additional 28.2 seconds to redo logs, while WBL and Zen takes merely additional 0.49 and 1.12 seconds, respectively.

Comparing Figure 19(b) with Figure 19(a), as the size of the database increases, MMDB, WBL, and Zen all take significantly more time for recovery. Note that Zen spends most of the time in scanning tuples and rebuilding indices. Therefore, persistent indices may help significantly improve the recovery time of Zen. We discuss persistent indices in Section 5.

Recovery for TPCC-NP. We conduct the TPCC-NP recovery experiment. The TPCC database contains 2048 warehouses. We use 16 parallel threads. As shown in Figure 19(c), we observe similar trends as the YCSB recovery experiment. MMDB takes 282.6 and 298.8 seconds for 4 million and 16 million transactions, respectively. WBL takes 10.3 and 11.1 seconds for 4 million and 16 million transactions, respectively. Zen takes 10.1 and 11.2 seconds for 4 million and 16 million transactions, respectively. Zen and WBL are dramatically faster than MMDB for crash recovery.

4.4 Support for Concurrency Control

We demonstrate that Zen supports a wide variety of concurrency control methods in Figure 20, including three 2PL variants (2PL with deadlock detection, wait and die, and no waiting [40]), three OCC variants (OCC [21], Silo [32], and Tictoc [41]), three MVCC variants (MVCC [7], Hekaton [15], Cicada [24]), and a partition-based method (HStore [31]). Our implementation for 9 of the above 10 methods (except Cicada) is based on DBx1000, an in-memory OLTP testbed for concurrency control research. For completeness, we include the Cicada results from Figure 5 in Figure 20. However, note that the results are not directly comparable because the implementations in DBx1000 and Cicada are different. For example, DBx1000 simplifies space management by not reclaiming space for old tuples (which is only OK for short test runs). We configure P to be 4. We use a 160GB YCSB benchmark under High Skew and Balanced configuration with 16 threads.

Comparing Zen with MMDB, we find the following. First, Zen achieves 1.10x-2.46x speedup in all concurrency control methods. Second, Zen achieves higher speedup for OCC and MVCC methods compared with 2PL variants. This is because Zen has more chance to process transactions in DRAM and writes no NVM. Moreover, Zen costs less for aborts because aborted transactions do not write to NVM in Zen. In contrast, in 2PL methods, conflicts are handled at each data access, which limits the performance of Zen. Third, Zen shows limited improvement in partition-based concurrency control method because cross-partition transactions become the bottleneck under High Skew. The coarse-grained lock of partition limits the performance of all OLTP engine designs.

Comparing Zen with WBL, we see that Zen achieves 1.11x-4.58x speedup in all concurrency control methods. Moreover, in OCC and MVCC variants, the performance gains of Zen are larger. Zen fully exploits DRAM for concurrency control, while WBL maintains concurrency control related per-tuple metadata in NVM. Hence, WBL sees small grained accesses in NVM, which limits its throughput.

5 DISCUSSION

In this section, we discuss two further design issues for Zen. To support faster recovery, we discuss the use of persistent indices. To support scaling out to a cluster of machines, we consider RDMA network and log shipping. We also discuss the limitations of Zen.

Alternative Index Designs. In the current design, we put the indices in DRAM and prove the scheme reasonable. Note that index design is orthogonal to the three main techniques of Zen, i.e. Met-Cache, log-free persistent transactions, and light-weight NVM space management. It is possible to employ persistent indices like NV-Tree [38], WB-Tree [10], FP-Tree [28], HiKV [37], and LB+tree [25] to improve recovery performance. Moreover, we can exploit previous index designs to reduce DRAM space consumption for indices. The dual-stage hybrid index architecture [42] saves space by placing aged index entries into a more compact structure. Selective persistence in NV-Tree [38], FP-Tree [28], and LB+-Tree [25] places non-leaf nodes of B+-Trees in DRAM and leaf nodes in NVM. Note that these alternative designs have been shown to have similar index performance compared to original DRAM-based indices.

Optional DRAM-Based Logs. Zen removes logging on NVM to reduce NVM writes for better OLTP throughput. However, we can optionally write *DRAM-based logs* for supporting log shipping to a hot standby, and archive the logs for supporting point-in-time recovery and disaster recovery. Since such logs are not required for persistence in Zen, the logs do not need to be "write-ahead" and can be handled with little impact on transaction performance.

Limitations of Zen. First, Zen cannot support OLTP databases larger than NVM memory. It would be interesting to study Zen's optimizations to improve the 3-tier design [33]. Second, a long running read-write transaction may trigger the exclusive mode and delay other transactions. It is challenging to support such transactions well in any OLTP design. Users may be advised to rewrite the transaction as smaller tasks for better performance.

6 CONCLUSION

In this paper, we propose Zen, a high-throughput log-free OLTP engine for NVM. Zen employs three novel techniques to reduce NVM overhead, namely the Met-Cache, log-free persistent transactions, and light-weight NVM space management. Experimental results show that Zen achieves up to 10.1x improvements over MMDB with NVM capacity, WBL, and FOEDUS, for YCSB and TPCC benchmarks. Zen achieves a transaction throughput of 1.8 million transactions per second using 32 threads. The recovery time of Zen is on the order of a few seconds for a database of a few hundred GB in size. In conclusion, we believe Zen is a viable solution to support OLTP transactions in NVM memory.

ACKNOWLEDGMENTS

This work is partially supported by National Key R&D Program of China (2018YFB1003303) and NSFC (61572468).

REFERENCES

[1] 2019. Intel Optane DC Persistent Memory Architecture and Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[2] 2020. TPC benchmark C. http://www.tpc.org/tpcc/.

[3] Dmytro Apalkov, Alexey Khvalkovskiy, Steven Watts, Vladimir Nikitin, Xueti Tang, Daniel Lottis, Kiseok Moon, Xiao Luo, Eugene Chen, Adrian Ong, Alexander Driskill-Smith, and Mohamad Krounbi. 2013. Spin-transfer torque magnetic random access memory (STT-MRAM). *ACM J. Emerg. Technol. Comput. Syst.* 9, 2 (2013), 13:1–13:35.

[4] Joy Arulraj, Justin J. Levandoski, Umar Farooq Minhas, and Per-Åke Larson. 2018. BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory. *Proc. VLDB Endow.* 11, 5 (2018), 553–565.

[5] Joy Arulraj, Andrew Pavlo, and Subramanya Dulloor. 2015. Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 707–722.

[6] Joy Arulraj, Matthew Perron, and Andrew Pavlo. 2016. Write-Behind Logging. *Proc. VLDB Endow.* 10, 4 (2016), 337–348.

[7] Philip A. Bernstein and Nathan Goodman. 1981. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.* 13, 2 (1981), 185–221.

[8] Tuan Cao, Marcos Antonio Vaz Salles, Benjamin Sowell, Yao Yue, Alan J. Demers, Johannes Gehrke, and Walker M. White. 2011. Fast checkpoint recovery algorithms for frequently consistent applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011.* 265–276.

[9] Shimin Chen, Phillip B. Gibbons, and Suman Nath. 2011. Rethinking Database Algorithms for Phase Change Memory. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings.* 21–31.

[10] Shimin Chen and Qin Jin. 2015. Persistent B+-Trees in Non-Volatile Main Memory. *Proc. VLDB Endow.* 8, 7 (2015), 786–797.

[11] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011.* 105–118.

[12] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin C. Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, Jeanna Neefe Matthews and Thomas E. Anderson (Eds.). ACM, 133–146.

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010.* 143–154.

[14] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, and David A. Wood. 1984. Implementation Techniques for Main Memory Database Systems. In *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984.* 1–8.

[15] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013.* 1243–1254.

[16] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. 1976. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM* 19, 11 (1976), 624–633.

[17] Ru Fang, Hui-I Hsiao, Bin He, C. Mohan, and Yun Wang. 2011. High performance database logging using storage class memory. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan (Eds.). IEEE Computer Society, 1221–1231.

[18] D. H. Graham. 2019. Intel optane technology products - what's available and what's coming soon. https://software.intel.com/en-us/articles/3d-xpointtechnology-products.

[19] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. 2014. NVRAM-aware Logging in Transaction Systems. *Proc. VLDB Endow.* 8, 4 (2014), 389–400.

[20] Hideaki Kimura. 2015. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 691–706.

[21] H. T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.* 6, 2 (1981), 213–226.

[22] Juchang Lee, Kihong Kim, and Sang Kyun Cha. 2001. Differential Logging: A Commutative and Associative Logging Scheme for Highly Parallel Main Memory Databases. In *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany.* 173–182.

[23] Tobin J. Lehman and Michael J. Carey. 1987. A Recovery Algorithm for A High-Performance Memory-Resident Database System. In *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, CA, USA, May 27-29, 1987.* 104–117.

[24] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 21–35.

[25] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+-Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. *Proc. VLDB Endow.* 13, 7 (2020), 1078–1090.

[26] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. (2017), 329–343.

[27] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 677–689.

[28] 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, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 371–386.

[29] Simone Raoux, Geoffrey W. Burr, Matthew J. Breitwisch, Charles T. Rettner, Yi-Chou Chen, Robert M. Shelby, Martin Salinga, Daniel Krebs, Shih-Hung Chen, Hsiang-Lan Lung, and Chung Hon Lam. 2008. Phase-change random access memory: A scalable technology. *IBM J. Res. Dev.* 52, 4-5 (2008), 465–480.

[30] Kun Ren, Thaddeus Diamond, Daniel J. Abadi, and Alexander Thomson. 2016. Low-Overhead Asynchronous Checkpointing in Main-Memory Database Systems. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016.* 1539–1551.

[31] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2007. The End of an Architectural Era (It's Time for a Complete Rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007.* 1150–1160.

[32] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, 18–32.

[33] 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, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 1541–1555.

[34] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*, Rajiv Gupta and Todd C. Mowry (Eds.). ACM, 91–104.

[35] Tianzheng Wang and Ryan Johnson. 2014. Scalable Logging through Emerging Non-Volatile Memory. *Proc. VLDB Endow.* 7, 10 (2014), 865–876.

[36] Tianzheng Wang and Hideaki Kimura. 2016. Mostly-Optimistic Concurrency Control for Highly Contended Dynamic Workloads on a Thousand Cores. *Proc. VLDB Endow.* 10, 2 (2016), 49–60.

[37] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, Dilma Da Silva and Bryan Ford (Eds.). USENIX Association, 349–362.

[38] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015*, Jiri Schindler and Erez Zadok (Eds.). USENIX Association, 167–181.

[39] J. Joshua Yang and R. Stanley Williams. 2013. Memristive devices in computing system: Promises and challenges. *ACM J. Emerg. Technol. Comput. Syst.* 9, 2 (2013), 11:1–11:20.

[40] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proc. VLDB Endow.* 8, 3 (2014), 209–220.

[41] Xiangyao Yu, Andrew Pavlo, Daniel Sánchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 1629–1642.

[42] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. 2016. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. 1567–1581.

[43] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. 2014. Fast Databases with Fast Durability and Recovery Through Multicore Parallelism. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*. 465–477.