Noname manuscript No.
(will be inserted by the editor)

Zen+: a Robust NUMA-Aware OLTP Engine Optimized for Non-Volatile Main Memory

Gang Liu · **Leying Chen** · **Shimin Chen**

(Preprint)

Received: date / Accepted: date

Abstract Emerging Non-Volatile Memory (NVM) technologies like 3DXpoint 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 accesses 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 databases 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. We further propose Zen+ by extending Zen with two mechanisms, i.e., MVCC-based adaptive execution and NUMA-aware soft partition, to robustly and effectively support long running transactions and NUMA architectures. Experimental results on a real machine equipped with Intel Optane DC Persistent Memory show that compared with existing solutions that run an OLTP database as large as the size of NVM, Zen achieves 1.0x-10.1x improvement while attaining fast failure recovery, and supports ten types of concurrency control methods. Experiments also demonstrate that Zen+ robustly supports long running transactions and efficiently exploits NUMA architectures.

Keywords Non-Volatile Memory · OLTP engine · Metadata-Enhanced Tuple Cache · Log-Free Transaction · NUMA

1 Introduction

Byte-addressable, non-volatile memory (NVM) is a new type of memory technology designed to address the DRAM scaling problem [44,55,3,1,21]. 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 removing 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 [5,27,6].

Recent studies in concurrency control methods have advanced the main memory OLTP transaction throughput in a single machine (without persistence) to over one million transactions per second [48,17,38,51,57,33]. However, replacing DRAM with NVM tends to slow down a 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 drastic (e.g., 10x) overhead. In this paper, we rethink the design of the OLTP engine for NVM by fully considering the properties of NVM. Our goal is to achieve transaction performance similar to those of pure DRAM based OLTP engines.

Shimin Chen is the corresponding author.

Gang Liu
SKL of Computer Architecture, ICT, CAS
University of Chinese Academy of Sciences
E-mail: liugang@ict.ac.cn

Leying Chen
SKL of Computer Architecture, ICT, CAS
University of Chinese Academy of Sciences
E-mail: chenleying19z@ict.ac.cn

Shimin Chen
SKL of Computer Architecture, ICT, CAS
University of Chinese Academy of Sciences
E-mail: chensm@ict.ac.cn

We observe three challenges in achieving our goal:

- *Tuple Metadata Modifications*: Main memory OLTP engines typically maintain a small amount of metadata per tuple for supporting concurrency control [48,38,51,57, 33]. 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*: NVM space allocation operations need to be persistent across power failure. Hence, every NVM memory allocation and free call 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 allocation overhead.

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 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 the LP bit. If it is set for one of the tuples, then the transaction is committed. Otherwise, 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 for tuple allocations. Each thread has its own allocation structures to avoid thread synchronization overhead.

Moreover, we propose Zen+ by extending Zen with two mechanisms, namely, MVCC-based adaptive execution and NUMA-aware soft partition, to robustly and effectively support long running transactions and NUMA architectures.

MVCC-Based Adaptive Execution: We aim to (i) efficiently execute long running transactions as much as possible when there is no inherent conflict; (ii) robustly support long running transactions even if there are conflicts; and (iii) effectively manage the system resource usage. For (i), Zen+ adopts MVCC as the concurrency control method to support long-running read-only transactions and light-weight read-write transactions. For (ii) and (iii), we propose an adaptive execution strategy with pre-defined resource usage and roll back thresholds. Zen+ performs normal processing while monitoring the thresholds. If any of the thresholds is triggered, Zen+ stops all other transactions and exclusively executes the privileged long running transaction to completion.

NUMA-Aware Soft Partition: In a machine with multiple CPU sockets, a processor can access its local NVM memory significantly (e.g, 2-3x) faster than remote NVM memory attached to another socket. Therefore, it is desirable to minimize remote NVM accesses in the OLTP design. We propose NUMA-local writes for Zen+. That is, threads in Zen+ write only to local NVM, thereby eliminating remote NVM writes. Moreover, Zen+ divides base tables into partitions, and assigns partitions to NUMA nodes. To run a transaction, Zen+ extracts the target tuple information of the transaction as much as possible, computes the NUMA affinity of the transaction based on the partitions of its target tuples, then executes the transaction on the NUMA node with the highest affinity score. The partition-to-NUMA-node mapping is soft in that Zen+ collects NUMA access statistics for partitions and dynamically adjust the assignment of partitions to NUMA nodes.

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-art NVM based OLTP designs. 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. Third, we propose Zen+ by extending Zen with two novel mechanisms: MVCC-based adaptive execution for long running transactions, and NUMA-aware soft partition for NUMA performance. Fourth, we evaluate the runtime and recovery performance of our proposed solutions using YCSB and TPCC benchmarks on a real machine equipped with Intel Optane DC Persistent Memory. Experimental results show that compared to existing designs, Zen achieves 1.0x-10.1x improvements, while attaining fast recovery and supporting 10 different concurrency control methods. The two mechanisms in Zen+ can robustly and effectively support long running transactions and NUMA architectures.

The rest of the paper is organized as follows. Section 2 provides the background and motivates the study of NVM based OLTP engines. Section 3 presents the design of Zen. Section 4 proposes Zen+ to effectively handle long running transactions and NUMA architectures. Section 5 evaluates our solutions, Zen and Zen+. Then, Section 6 discusses relevant issues, including alternative index designs, optional DRAM-based logs, variable length tuples, and the limitations of Zen/Zen+. Finally, Section 7 concludes the paper.

2 Background and Motivation

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

2.1 NVM Characteristics

There are several competing NVM technologies, including PCM [44], STT-RAM [55], Memristor [3], and 3DXPoint [1, 21]. They share similar characteristics: i) Like DRAM, NVM is byte-addressable; 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 with cache line flush (e.g., `clwb`) and memory fence (e.g., `sfence`) instructions are required to persist data from the volatile CPU cache to NVM, incurring drastically higher (e.g., 10x) overhead than normal writes.

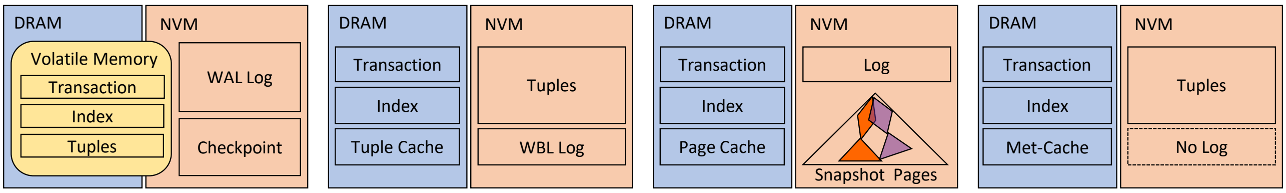
From previous work on NVM based data structures and systems [14, 11, 13, 49, 19, 50, 24, 12, 5, 27, 54, 39, 6, 53, 46, 4, 35, 34], we obtain three common design principles: i) Put frequently accessed data structures in DRAM if they are either transient or can be reconstructed upon recovery; ii) Reduce 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 [48, 17, 38, 51, 57, 33]. Instead of using two phase locking (2PL) [18, 7], which is the standard method in traditional disk-oriented databases, main memory databases typically exploit optimistic concurrency control (OCC) [28] and multi-version concurrency control (MVCC) [7] for higher performance. Silo [48] enhances OCC with epoch-based batch timestamp generation and group commit. MOCC [51] is an OCC based method that exploits locking mechanisms to deal with high conflicts for hot tuples. Tictoc [57] removes the bottleneck of centralized timestamp allocation in OCC and computes transaction timestamps lazily at commit time. Hekaton [17] employs latch-free data structures and MVCC for transactions in memory. Hyper [38] improves MVCC for read-heavy transactions in column stores by performing in-place updates and storing before-image deltas in undo buffers. Cicada [33] 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 concurrency control methods have achieved throughput of over one million transactions per second (TPS) without persistence.

Similar to traditional databases, main memory data-bases (MMDB) store logs and checkpoints on durable storage (e.g., HDDs, SSDs) in order to achieve durability [16, 30, 29, 10, 59, 45]. 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.



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

Fig. 1: OLTP engine designs for NVM.

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 virtual address ranges in the 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 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.

Logging is often one main performance bottleneck in database systems. Various designs have been proposed to exploit NVM for higher logging performance [19,24,50,20,26,23]. In the context of NVM based database systems, recent studies aim to remove or reduce logging by performing out-of-place updates directly on the tuples [41,40,6]. We discuss the latest of these designs, WBL, in the following.

WBL. Write-behind logging (WBL) [6] maintains indices and a tuple cache in DRAM, as shown in Figure 1(b). 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. Thus, 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 [27] 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 handles transactions completely in DRAM. As a result, it avoids 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 [46].

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. Zhou et al. proposes Spitfire that exploits machine learning techniques to automatically tune the policies for data migration in the 3-tier storage design [60]. 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 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 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, avoiding NVM persist operations for tuple allocations and frees.

3 Zen Design

We propose Zen, a high-throughput log-free OLTP engine for NVM. Zen exploits the large capacity of NVM to support OLTP databases much larger than DRAM, while addressing the three design challenges.

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 identify 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.

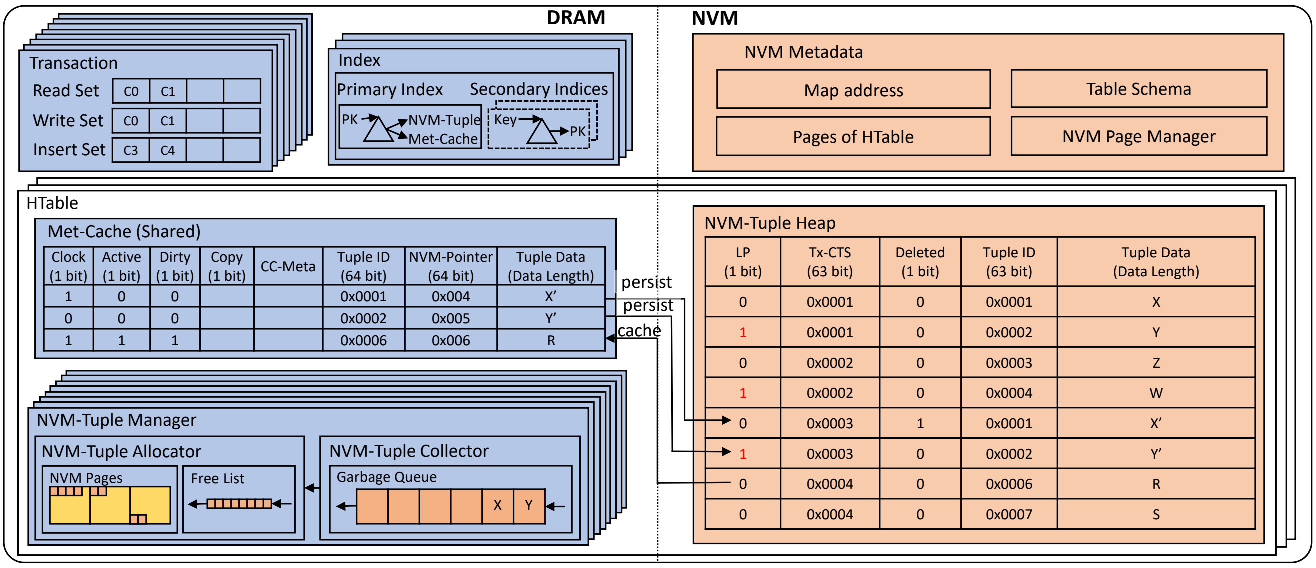
Met-Cache. The Met-Cache manages a tuple-grain cache in DRAM for the corresponding NVM-tuple heap. An Met-Cache entry contains the tuple data and seven metadata fields: a pointer to the NVM-tuple if it exists, the tuple ID, a dirty bit, an active bit to indicate that the entry may be used by an active 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 H-Table 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).

Transaction-Private Data. Zen runs multiple threads to handle transactions concurrently. Each thread reserves a thread-

[1] For simplicity, Zen assumes that the tuple size is fixed. For example, `varchar(n)` can be regarded as `char(n)`. We discuss how to support variable sized tuples in Section 6.

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



Met-Cache (Shared)

Clock (1 bit)	Active (1 bit)	Dirty (1 bit)	Copy (1 bit)	CC-Meta	Tuple ID (64 bit)	NVM-Pointer (64 bit)	Tuple Data (Data Length)
1	0	0			0x0001	0x004	X’
0	0	0			0x0002	0x005	Y’
1	1	1			0x0006	0x006	R

NVM-Tuple Heap

LP (1 bit)	Tx-CTS (63 bit)	Deleted (1 bit)	Tuple ID (63 bit)	Tuple Data (Data Length)
0	0x0001	0	0x0001	X
1	0x0001	0	0x0002	Y
0	0x0002	0	0x0003	Z
1	0x0002	0	0x0004	W
0	0x0003	1	0x0001	X’
1	0x0003	0	0x0002	Y’
0	0x0004	0	0x0006	R
0	0x0004	0	0x0007	S

Fig. 2: Zen architecture.

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, while 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. NVM is initialized with all 0s and Tx-CTS=0 indicates 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, each for a 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 H-Table$_i$ per transaction. Assuming accesses are uniformly distributed across a HTable, we can estimate the average number of Met-Cache hits per transaction as:

$$\text{Met-Cache Hits} = \sum_i \frac{f_i C_i}{S_i}.$$

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 for tuples stored in NVM. 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 concurrency control 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. The allocator records the three empty NVM-tuple slots.

Perform. A transaction obtains a timestamp when it starts. For each tuple that it requests, 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.

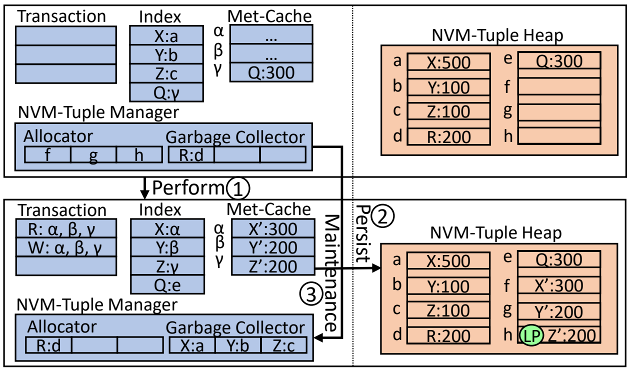


Fig. 3: Illustration of perform, persist, and maintenance using a transaction (X-=200; Y+=100; Z+=100).

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 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

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); /* flush the 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); /* except the first line */
9      sfence();
10     last-tuple.LP= 1;
11     clwb(last-tuple.start); /* flush the header */
```

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 `clwbs` 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 any 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.

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 allocator and garbage queue for NVM-Tuple allocation. 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, no running transactions access entries with Tx-CTS $<$ the minimum Tx-CTS. Such entries can be safely moved from the garbage queue to the allocator free list.

For normal OLTP workloads, the garbage queues are often quite short. This is because every transaction thread tries to garbage collect and recycle stale NVM-tuple versions after it completes each transaction. Stale NVM-tuple versions are often reclaimed in a short period of time, and consume only a small amount of NVM space. On the other hand, if there is a long running transaction, it may prevent the minimum Tx-CTS from being updated, and the garbage queues can grow drastically. We discuss how to robustly support long running transactions in Section 4.1.

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 Wide Varieties of Concurrency Control Methods

Our 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 5.4, including three 2PL variants (2PL with deadlock detection, wait and die, and no waiting [56]), three OCC variants (OCC [28], Silo [48], and Tictoc [57]), three MVCC variants (MVCC [7], Hekaton [17], and Cicada [33]), and a partition-based method (H-Store [47]). To support a concurrency 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. Importantly, the concurrency control method can process the metadata enhanced tuples in Met-Cache and entirely in DRAM.

We consider the support of versions. Concurrency control methods can be divided into two classes: 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 al-

[3] 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.

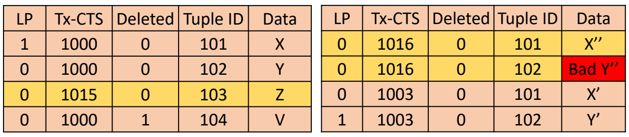
ways persisted to NVM. This is regardless of the number of versions in Met-Cache. For single-version methods, Met-Cache holds a single version for a tuple. While there can be multiple committed NVM-tuple versions in NVM-tuple heap, only the latest version can be cached in Met-Cache. For multi-version 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 consists of the NVM metadata (i.e. table schemas and metadata for page-level NVM space management) and committed tuple versions 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’

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

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 6-7). If a tuple's timestamp $\leq$ ts-commit, Zen considers the associated transaction has committed. Zen updates the index with the tuple (Line 11). 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 24-31). 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 13).

After the per-thread region is scanned, ts-commit is the maximum committed timestamp in this region. Since a thread can write only to 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 transactions in this region. Therefore, a transaction with Tx-CTX $>$ ts-commit must have not committed successfully. The crash must occur when the transaction was being persisted. Then, Zen revisits the tuples in the pending list. If a tuple's timestamp $\leq$ ts-commit, then Zen updates the index with the tuple and possibly garbage collects an old version of the tuple in the index (Line 16). If a tuple's timestamp $>$ ts-commit, Zen discards the tuple by marking the tuple slot empty (with Tx-CTS=0) and puts it into the garbage queue (Line 19-20).

Correctness. Algorithm 2 correctly identifies all committed tuples in the region. First, if Tx-CTS $\leq$ 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

Algorithm 2: Scan NVM-tuple heap region.

```
global global-ts-commit[thread-num] = {0};
global global-pending-list[thread-num][dynamic];
Function scanRegion(region, thread-id)
    ts-commit= 0; pending-list= {};
    foreach (ntup ∈ region) do
        if (ntup.LP) then
            ts-commit= max (ts-commit, ntup.Tx-CTS);
        if (ntup.Deleted or ntup.Tx-CTS==0) then
            putIntoFreeList(ntup); continue;
        if (ntup.Tx-CTS ≤ ts-commit) then
            updateIndexGC(ntup); /* committed */
        else
            pending-list.push(ntup); /* uncertain */
    foreach (ntup ∈ pending-list) do
        if (ntup.Tx-CTS ≤ ts-commit) then
            updateIndexGC(ntup); /* committed */
        else
            /* Crash occurs at commit time */
            putIntoFreeList(ntup);
            ntup.Tx-CTS=0; clwb(ntup);
    sfence();
Function updateIndexGC(ntup)
    ptup= searchIndex(ntup);
    if (ptup == NULL) then
        insertIndex(ntup); /* no existing version */
        return;
    if (ntup.Tx-CTS < ptup.Tx-CTS) then
        putIntoFreeList(ntup); /* ntup is old */
        return;
    updateIndex(ntup);
    putIntoFreeList(ptup); /* ptup is old */
```

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 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 Optimization for accessing multiple tables

For multiple tables, Zen still sets and persists the LP flag only for the last committed tuple regardless of the associated tables in a transaction, rather than setting the LP flag for one tuple per table. However, randomly distributed LP flags at tables incur overhead for crash recovery because our recovery algorithm needs to scan more tuples to find LP flags and faces more uncertain tuple versions.

To optimize recovery performance, Zen chooses the table to persist LP in a pre-defined order. During recovery, Zen scans all tuples of tables following the same order. In this way, Zen identifies all the LP flags faster and earlier in fewer tables. This order could be set globally by the database administrator for each database instance. It is advisable to set the table that serves as application logging or is frequently modified as the first table. For example, in the TPCC benchmark, we put the history table and the order table as the first two tables in the pre-defined order to persist LP. Besides, Zen provides a default persist order, e.g., the lexicographical order of table name.

When adapting this optimization, Zen needs to sort the write set of each transaction at its commit stage. It incurs little overhead because most short OLTP transactions write a small number of tuples. Typically, their write set is small enough to fit in CPU cache.

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 our page granularity for NVM is 2MB, 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 Zen+: Improving Robustness of Zen

In this section, we study how to robustly and effectively support long running transactions and NUMA architectures. We propose Zen+ that extends Zen with two novel techniques: MVCC-based adaptive execution and NUMA-aware soft partition. In the following, Section 4.1 describes the support for long running transactions, while Section 4.2 focuses on NUMA-aware solutions.

4.1 Support for Long Running Transactions

A typical OLTP transaction performs a small number of reads and writes. In contrast, a long running transaction may read or write a large number of tuples. This may severely impact the performance of an OLTP system for the following two reasons. First, a long running transaction may occupy resource (such as locks, memory, and CPU cores) for a long period of time, stalling or slowing down other concurrent transactions. Second, the amount of work required for a long running transaction could be several orders of magnitude higher than that for a typical OLTP transaction. The two factors combined can significantly reduce the transaction throughput of the system. This is especially painful for a main memory based OLTP engine, which is capable of supporting millions of transactions per second. As a result, many existing main memory based OLTP solutions tend to trade long running transactions for better throughput [25]. While challenging, it is important for production systems to robustly support long running transactions. Therefore, we consider how to robustly and effectively handle long running transactions in Zen+.

4.1.1 Challenges

Long running transactions pose two main types of challenges to (NVM) main memory optimized OLTP engines. In the following, we discuss the challenges with an emphasis on Zen's structures as described in Section 3.

Concurrency Control Challenge. A long running transaction may conflict with concurrent short transactions. That is, the read set and write set of a long running transaction may overlap with those of concurrent short transactions. Let us consider the behavior of long running transactions under different concurrency control methods. First, when a locking based concurrency control method is in use, the long running transaction has to obtain a large number of locks (even for reads), incurring significant locking overhead. It can be easily stalled by other concurrent short transactions holding relevant locks. Then the long running transaction can block newly issued short transactions because of the locks that it holds. Such poor behavior can cause severe system performance degradation. Second, when an OCC-style or MVCC-style concurrency control method is in use, a long running transaction may be frequently rolled back because short transactions encounter few conflicts and are more likely to succeed. This can lead to a considerable amount of wasted computation.

Resource Usage Challenge. A long running transaction consumes and occupies a large amount of resource in an OLTP system. This may exceed the capacity of certain internal data structures and lead to abnormal behaviors even if there is no conflict across transactions. In Zen, the Met-Cache data structure in DRAM holds the subset of tuples being actively used in transactions. It is possible that the number of tuples accessed by a long running transaction may exceed the capacity of the Met-Cache. This problem can block the execution of not only the long running transaction itself, but also other concurrent short transactions. Moreover, the garbage collector plays a key role in NVM space management. It reclaims an NVM-tuple entry only if its Tx-CTS is less than the global minimum Tx-CTS, which guarantees the entry is not used by any active transactions. Note that the global minimum Tx-CTS is periodically computed by taking the minimum of last committed Tx-CTS for all threads. However, the last committed Tx-CTS of the thread running the long transaction can be much smaller than that of other threads running short transactions. Consequently, the long running transaction can prevent the update of the global minimum Tx-CTS, and hence stop the garbage collector from reclaiming old NVM-tuple versions in the whole system.

Böttcher et al. propose to eagerly prune obsolete tuple versions in garbage collection in MVCC-based in-memory databases [9]. In this way, resource usage for both short and long transactions can be reduced. Please note that the eager pruning technique is orthogonal to our proposed solution in Section 4.1.2. While eager pruning mitigates the resource usage problem, our proposed solution detects the cases that the resources are about to be used up, and provides a safety net to robustly support such cases. Combining eager pruning and our solution is an interesting research direction to investigate in the future.

4.1.2 Our Solution: MVCC-Based Adaptive Execution

Given the above challenges, our design goal is threefold: (i) efficiently execute long running transactions as much as possible when there is no inherent conflict; (ii) robustly support long running transactions even if there are conflicts; and (iii) effectively address the resource usage challenge.

For (i), we consider two types of long running transactions: read-only long running transactions, and read-write long running transactions. The former is quite common. Examples include real-time analytics and ad-hoc queries. In contrast, it is less common for a transaction to modify a large number of tuples. Therefore, we would like to efficiently support read-only long running transactions, while robustly run read-write long running transactions.

For this purpose, we choose an MVCC-style concurrency control method[4]. For a read-only transaction, Zen+ sets its commit timestamp to the current global minimum Tx-CTS-1. In this way, read-only transactions see a consistent snapshot of the database before the start of all the other concurrently running read-write transactions. Moreover, when a read-only transaction incurs a Met-Cache miss, the transaction directly reads the target NVM-tuple without fetching the tuple into the Met-Cache, thereby reducing its resource usage. This is correct because all tuple versions with time stamps $\geq$ the global minimum Tx-CTS are still cached in the Met-Cache. However, a read-only transaction may still not proceed to completion in some cases because the transaction stalls the garbage collection of other concurrent transactions and the other transactions may use up NVM. Hence, Zen+ is obliged to handle the resource usage challenge for even read-only transactions.

For (ii) and (iii), we propose an adaptive execution strategy with pre-defined resource usage and roll back thresholds. Our adaptive execution strategy includes two stages: the detecting stage and the exclusive stage. In the detecting stage, Zen+ performs normal processing for transactions, while monitoring the resource usage and counting the number of retries of transactions. If the resource usage or the amount of wasted work due to retries goes beyond the pre-defined thresholds, Zen+ marks a long running transaction as a privileged transaction and switches into the exclusive stage. In the exclusive stage, Zen+ stops all other transactions and exclusively runs the transaction to completion.

Figure 5 depicts the two stages. In the detecting stage, a thread executes a long running transaction, starting at timestamp 100. The length of the garbage queue of the thread is 10. Meantime, two other threads execute a number of normal transactions. As the long running transaction blocks the update of the global minimum timestamp, the garbage queues of the threads running normal transactions increase considerably. Then, certain pre-defined threshold(s) are met and the long running transaction is detected as the privileged transaction. Zen+ switches to the exclusive stage and runs the privileged transaction to completion before resuming normal execution in the detecting stage.

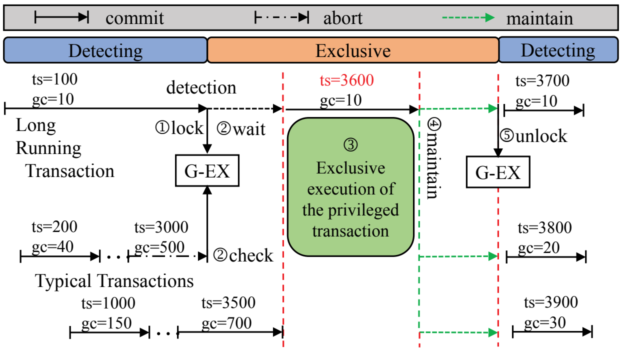


Fig. 5: Illustration of the two stages: the detecting stage and the exclusive stage. (ts: timestamp of a transaction, gc: garbage queue length)

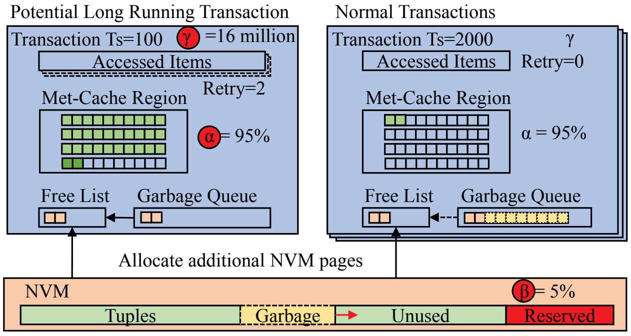


Fig. 6: Three conditions that trigger the exclusive stage.

Detecting Long Running Transactions. As illustrated in Figure 6, a transaction is detected as a privileged long running transaction if at least one of the following three conditions is satisfied:

Condition 1: The transaction is a read-write transaction and the number of accessed tuples of the transaction is beyond a pre-defined threshold α. This condition constrains the consumption of the Met-Cache by the transaction.

Condition 2: The size of the available NVM memory in the NVM page allocator is below a pre-defined threshold β. This condition indicates that Zen+ is about to use up the available NVM space.

Condition 3: The amount of wasted work is beyond a pre-defined threshold γ. This condition guarantees that any long running transaction can commit successfully after a limited number of tries.

For Condition 1 and 3, the relevant transaction is marked as the privileged transaction. For Condition 2, the transaction with the largest number of accessed tuples is marked as the privileged transaction.

Choice of α. α is specified as a pre-defined percentage of a Met-Cache region, as shown in Figure 6. This threshold

[4] Please note that the choice of MVCC-style concurrency control method is only required by Zen+'s support for long running transactions. Other techniques in this paper can flexibly support a wide range of concurrency control methods.

limits the maximal Met-Cache space that a read-write transaction can consume in normal execution during the detecting stage. A naïve way is to set α to be 100% in hope that a long running transaction spends as much time as possible in normal execution, and reduces the chance to switch to the expensive exclusive execution state. However, the Met-Cache region can hold tuple versions that are copied to other regions and actively used by other concurrently running transactions. As a result, the Met-Cache region could be filled before the 100% threshold is reached. Hence, we need to choose a lower threshold. If we have knowledge of the OLTP workload, we can estimate the average number of tuples accessed by a normal (short) transaction. Then we can compute the amount of Met-Cache region space that should be reserved for concurrently running transactions. Note that a Met-Cache region can be of several GBs in a typical server machine. This capacity is often large enough to cache millions of tuples. Therefore, the reserved space will be a small percentage of the Met-Cache region. In practice, we find that reserving 5% of the space is often good enough. We set α to 95% in our experiments.

Choice of β. β is specified as a pre-defined percentage of the available page-grained NVM space, as shown in Figure 6. As a long running transaction prevents the update of the global minimum timestamp, NVM-tuple versions can no longer be reclaimed. Consequently, the NVM-tuple space managers cannot reuse reclaimed NVM slots for newly written tuple versions. They have to request the NVM page manager to allocate new NVM pages. Therefore, the extensive consumption of NVM pages is an indication of long running transactions. On the one hand, we must ensure that the NVM space is not used up. Otherwise, allocation requests would fail and no transactions that write new tuple versions could commit. The whole system would come to a stop, without making forward progress. On the other hand, we do not want to have a threshold that is too loose. Otherwise, the expensive exclusive stage may be falsely triggered. We find that β=5% is a good setting, which we use in our experiments.

Choice of γ. γ is specified as a pre-defined tuple count to limit the wasted work due to retries, as shown in Figure 6. If a transaction aborts and retries, Zen+ counts the total number of tuples accessed by the transaction in all its retries. This count is maintained as transaction-private data. It measures the total amount of wasted work because of the retries of the transaction. If the characteristics of the OLTP workload are known, we can estimate the maximum number of normal rollbacks and retries of a transaction because of conflicts with concurrent transactions. Then, the multiplication of this number of retries and the average number of tuples accessed by a transaction gives a lower bound of γ. On the other hand, we can empirically set a threshold for the wasted work in terms of wall-clock time. Given the latencies and bandwidths of NVM reads and NVM writes, it is easy to compute the number of NVM tuples that can be visited within the specified time. This will give an upper bound of γ. We set γ to 16 million in our experiments. This translates to about 10s of wasted work due to retries.

Exclusive Stage. In the exclusive stage, Zen+ runs the privileged transaction exclusively to completion. The transaction enjoys all the system resources.

The middle part of Figure 5 depicts the long running transaction in the exclusive stage. Zen+ uses a global exclusive flag (G-EX) to protect the entrance and exit of the exclusive stage. ①When a triggering condition is met and a long running transaction is detected, Zen+ locks G-EX using a `compare_and_swap`. ②The privileged transaction waits for other concurrent transactions to end. Each transaction checks G-EX periodically. If it sees that G-EX is locked, a transaction aborts unless it is already in the process of committing its changes. In a short period of time, all other transactions either commit or quickly abort. ③The privileged transaction enjoys all resources and executes its transaction logic to completion. The privileged transaction follows the same procedure in the log-free persistence transactions except that it directly writes NVM if its Met-Cache region is used up. ④ Since the thread cooperative garbage collection mechanism stops in the exclusive stage, the privileged transaction is responsible for reclaiming stale NVM-tuples for all threads in the maintenance phase at the end of the exclusive stage. It checks the garbage queue of every thread, and moves NVM-tuple versions in the queue to the corresponding free list. If all NVM-tuples in a NVM page are freed, then it frees the entire NVM page[5]. Note that we can employ the implementation detail (ii) in Section 3.4 to limit the amount of space management work, and leave part of the work to the corresponding threads in normal execution. ⑤The privileged transaction unlocks G-EX and completes. Zen+ switches back to the detecting stage. All threads resume to handle new or aborted transactions.

Zen+ is recoverable when the system crashes during the exclusive stage. This is because the privileged transaction follows the same persistence procedure as in normal execution. Therefore, the same arguments as in Section 3.3.3 show that Zen+ can correctly recover from crashes.

4.2 Support for NUMA Architectures

Multi-socket (e.g., dual-socket/quad-socket) machines are popular today. NUMA architectures allow more CPUs and memory resources to be integrated in a single machine. A k-socket machine can provide k times as large NVM memory capacity, and k times as high CPU computing power

[5] A per-page counter can be kept in the NVM-tuple manager to keep track of the number of allocated slots in the page. The counter is updated for tuple allocations and frees. When the counter decreases to 0, we can return the page to the NVM page manager.

as a single-socket machine. We would like Zen+ to effectively exploit NUMA architectures to support larger OLTP databases with higher transaction performance.

In Section 4.2.1, we first review background on existing optimizations for NUMA systems, and consider their applicability to Zen+. Then, we propose our NUMA-Aware soft partition design in Section 4.2.2.

4.2.1 Background

In a multi-socket machine, NVDIMMs are attached to different sockets. For Intel Optane DC Persistent Memory, the NVM attached to different sockets are identified as different special devices (e.g., in Linux), and can be mapped to different virtual address regions in software. We use the term NVM NUMA node i to denote the NVM attached to CPU socket i and also the corresponding virtual memory region. Hence, software can control the space allocation and data accesses to different NVM NUMA nodes.

Previous work studies the placement and migration of threads and data to reduce remote memory accesses and balance CPU loads for operating systems [32,8], in-memory indices [36], execution of analytical queries [31,42,43,37], and graph analytics [22] in DRAM-based NUMA systems. In the context of NUMA systems equipped with NVM memory, Wang et al. investigate NUMA-aware thread migration in file systems [52]. They propose to migrate only threads but not data because (i) NVM writes are more expensive than NVM reads and (ii) NVM has limited write endurance.

H-Store [47] assumes that an OLTP database can be fully partitioned. It runs a dedicated thread per partition that handles the transactions accessing tuples in the partition. With this assumption, H-Store achieves high throughput for transactions that access only one partition. It can support NUMA architectures nicely by co-locating each partition and its dedicated thread and distributing the partitions across NUMA nodes. However, when this assumption is not satisfied, transactions that access multiple partitions suffer from significantly lower performance. Other mainstream OLTP engines mainly rely on the operating system for NUMA-aware thread scheduling and data placement.

From the related work, we extract the following three design principles: i) Remote NVM accesses should be reduced as much as possible; ii) Data migration incurs expensive NVM writes, and therefore should not be extensively used; iii) Workload characteristics are important (e.g., if transactions mostly focus on one of the partitions, the database partitioning approach can support NUMA architectures well). We follow these principles to consider NVM and NUMA properties, and workload characteristics in our NUMA-aware optimization in Zen+.

4.2.2 Our Design: NUMA-Aware Soft Partition

We begin our description with a naïve NUMA-agnostic solution. Then, we exploit the Zen structures to perform only NUMA-local writes. Finally, we extend NUMA-local writes and propose the technique of NUMA-aware soft partition.

Naïve NUMA-Agnostic Design. The naïve design is shown in Figure 7(a). It maintains a shared pool of DRAM memory and a shared pool of NVM memory. All threads allocate DRAM or NVM resources from the shared pools. DRAM and NVM pages are allocated in an interleaved fashion across NUMA nodes[6]. With this design, Zen allocates the Met-Cache and the NVM-Tuple heap pages from the shared DRAM pool and the shared NVM pool, respectively.

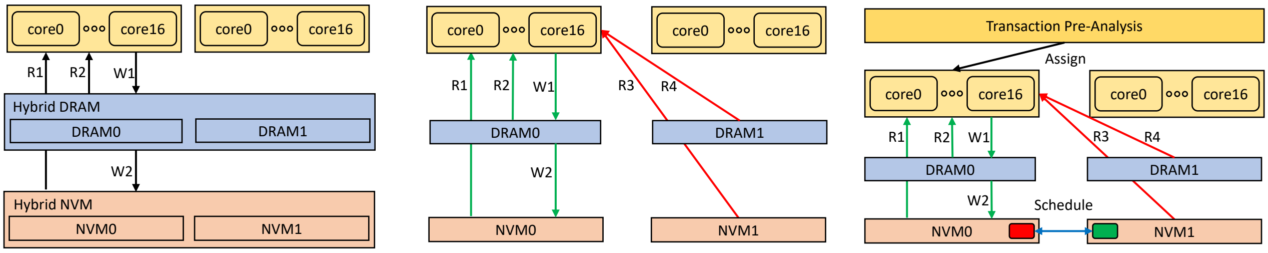
The naïve design is NUMA-agnostic in that it does not specially co-locate threads and data to reduce remote memory accesses. Suppose there are k NUMA nodes (e.g., k=2 for a dual-socket machine). Because of the interleaved allocation policy, an allocated DRAM/NVM page is equally likely to reside in one of the k NUMA nodes. Therefore, the probability of local DRAM/NVM accesses is $\frac{1}{k}$, and that of remote accesses is $\frac{k-1}{k}$. We would like to devise NUMA-aware solutions that significantly improve upon the naïve design.

NUMA-Local Write. NVM writes are more expensive than NVM reads. Remote NVM writes are even more costly. Thus, the first idea that comes to our mind is to eliminate remote NVM writes and perform only NUMA-local writes. As depicted in Figure 7(b), the design manages separate DRAM and NVM pools for NUMA nodes, and binds threads to CPU cores in different CPU sockets. A thread running on CPU socket i allocates DRAM and NVM space from the local pools of NUMA node i. A thread can read local or remote DRAM/NVM. However, it can write to only local DRAM or local NVM, thereby completely avoiding remote writes.

This design fits the architecture of Zen well. First, a Zen thread can allocate its Met-Cache region and its NVM-tuple heap pages from the local DRAM and NVM pools, respectively. Second, Zen's Met-Cache does not perform remote writes. A thread can read entries in other Met-Cache regions, but it cannot write to other Met-Cache regions. This is designed originally for reducing thread contentions. We can leverage the same facility to achieve NUMA-local DRAM writes. Third, when committing a transaction, a thread allocates NVM-tuple space from the NVM-tuple manager, which in turn allocates pages from the local NVM pool. Therefore, Zen performs only NUMA-local NVM writes.

While this design removes remote writes, it does not optimize for remote reads. Note that tuples can be migrated across NUMA nodes. When it wants to update a tuple that

[6] This is similar to the interleaved NUMA allocation policy in the operating system. However, when NVM is in the App Direct mode, the OS policy cannot be directly applied to NVM.



(a) Naïve Design (b) Thread-Local Write Design (c) NUMA-Aware Soft Partition

Fig. 7: NUMA related design and optimizations

is cached in another Met-Cache region, a thread copies the Met-Cache entry to its local Met-Cache region. At commit time, it writes the new version of the tuple to its local NVM NUMA node. This essentially migrates the tuple. In a pathological case, a sequence of back-to-back transactions Txn_1, Txn_2, Txn_3, ... update the same tuple. Odd numbered transactions are scheduled to run on NUMA node 1, while even numbered transactions are scheduled to run on NUMA node 0. Then the new versions of the tuple ping-pong between the two NUMA nodes, incurring a great many remote reads. We would like to avoid such cases as much as possible.

NUMA-Aware Soft Partition. The static database partitioning approach as described in Section 4.2.1 provides a simple solution to the above problem if a transaction always visits data in a single partition. However, real-world OLTP workloads are often more dynamic. It is likely that a transaction visits tuples in more than one partition.

We propose NUMA-aware soft partition to cope with the dynamic workload behaviors. Zen+ divides base tables into partitions, and maps partitions to NUMA nodes. For an incoming transaction, Zen+ computes its NUMA affinity based on the partitions that it is about to visit, then assigns the transaction to the most relevant NUMA node. By “soft”, we mean that Zen+ collects partition statistics and *dynamically* adjust the mapping from partitions to NUMA nodes

In the following, we describe the construction of partitions, the assignment of transactions, the dynamic partition mapping, and the parameter choices in details.

(1) Partitions. A table is partitioned by applying a hash function on the partition key (e.g., $hash(key)\%part\_size$). Please note that the partition is *logical*. There is no physical partitioning step that copies tuples into contiguous memory regions. Instead, through the NUMA-aware transaction assignment, the new versions of tuples in a partition tend to be written to the same NUMA node by NUMA-local writes. In this way, the tuples in a partition are likely to eventually co-locate in the same NUMA node.

The default partition key of a table is its primary key, which leads to even distribution of tuples to partitions. However, in some cases, tuples of multiple related base tables are frequently visited together in a transaction. Hence, it is desirable to have the related tuples in the same partition. We observe that the relationship between tuples is typically expressed as foreign key references. That is, the primary key of one table is referenced in other related tables as foreign keys (e.g., warehouse ID in TPCC). Consequently, we can judiciously select foreign keys as the partition key for the related tables so that related tuples belong to the same partition.

Algorithm 3: Transaction assignment.

```
1  global load[node_num];
2  global access_count[thd_num][part_num];
3  global part_map[part_num];
4  Function assignTransaction(txn)
5      if $max(load) - min(load) > \delta$ then
6          node = argmin (load); /* if load is very unbalanced */
7      else
8          node = getAffinityNode (txn);
9      thd_id = getNextThread (node);
10     access_count[thd_id][part_id] ++;
11     return thd_id;
12 Function getAffinityNode(txn)
13     vote[0..node_num-1] = 0;
14     foreach $t \in txn.requests$ do
15         if $(k = extractKey(t)) \neq None$ then
16             part_id = hash(k) % part_size; /* partition rule */
17             t_node = part_map[part_id];
18             vote[t_node] ++;
19     if $max(vote) == 0$ then
20         return getRandomNode (); /* complex transaction */
21     else
22         return argmax(vote); /* affinity can be computed */
```

Zen+ has two global data structures for partitions. There is a global hash map `part_map` that maps every partition to a NUMA node. Then, Zen+ keeps `access_count` for each thread and each partition.

(2) NUMA-Aware Transaction Assignment. Algorithm 3 considers load balancing and NUMA affinity for assigning transactions to NUMA nodes.

For load balancing purpose, if the loads across NUMA nodes differ greatly, Zen+ chooses the node with the lowest load to run the transaction (Line 6). Zen+ obtains CPU utilization by reading `/proc/stat`. From our experience, unbalanced load is often caused by changing workload.

Algorithm 4: Dynamic Mapping of Partitions.

```
1  global access_count[thd_num][part_num];
2  global part_map[part_num];
3  Function mapPartitions(void)
4      pt_cnt[0..part_num-1] = 0;
5      nd_pt[0..node_num-1][0..part_num-1] = 0;
6      for $t = 0; t < thd\_num; t++$ do
7          for $p = 0; p < part\_num; p++$ do
8              pt_cnt[p] += access_count[t][p];
9              node= getNodeForThread(t);
10             nd_pt[node][p] += access_count[t][p];
11     mapped_pt[0..node_num-1] = 0;
12     mapped_cnt[0..node_num-1] = 0;
13     foreach part ∈ descending order of pt_cnt[] do
14         min_load= min(mapped_cnt);
15         node= -1;
16         for $nd = 0; nd < node\_num; nd++$ do
17             if (mapped_cnt[nd] < 1.1 * min_load) and
               (mapped_pt[nd] < $\lceil \frac{part\_num}{node\_num} \rceil$) then
18                 if (node < 0) or (nd_pt[node][part] >
                   nd_pt[nd][part]) then
19                     node= nd;
20         orig_node = part_map[part];
21         if nd_pt[node][part] > nd_pt[orig][part] * $\eta$ then
22             part_map[part] = node; /* remap the part */
23         else
24             node= orig_node; /* mapping is unchanged */
25         mapped_pt[node] ++;
26         mapped_cnt[node] += nd_pt[node][part];
```

For NUMA affinity, Zen+ examines each request (data operation, e.g., SQL select, update, insert, delete) in the transaction (Line 14). It attempts to extract the base table and the primary/foreign key of each accessed tuple, then computes its partition ID (Line 15-16). This is feasible for point operations (e.g., when keys are specified in the "where" clause) but may fail for more complex requests. If the partition ID is identified, the algorithm maps the partition to its NUMA node and accumulates a vote for the node (Line 17-18). It chooses the NUMA node with the highest vote (Line 22). In case that all votes are 0, which means that all requests are complex, the algorithm chooses a node randomly (Line 19-20). The time complexity of this procedure is $O(n)$, where n is the number of requests in a transaction.

Finally, Zen+ assigns the transaction to a thread in the chosen NUMA node in a round robin fashion (Line 9).

(3) Dynamic Mapping of Partitions to NUMA Nodes. The system periodically invokes Algorithm 4 to dynamically map partitions to NUMA nodes. The algorithm aims to (i) distribute hot spots across NUMA nodes, and (ii) balance the number NVM accesses for the NUMA nodes.

First, the algorithm computes per-partition access counts (`pt_cnt`) and per-node-partition access counts (`nd_pt`) based on the global `access_count` (Line 4-10). Second, it examines the partitions from the hottest to the coldest (Line 13). That is, it follows the descending order of the accumulated per-partition access counts. Third, in every iteration, the algorithm checks all the NUMA nodes to find a candidate node to map the current partition. The algorithm excludes any node whose assigned load is already unbalanced (i.e., over 10% higher than the minimum load) (Line 17). It also guarantees that the number of partitions assigned to each node is roughly the same (Line 17). Then, the candidate is chosen as the node that sees the largest number of accesses for the partition (Line 18-19). Fourth, the algorithm compares the access counts of the candidate and the original node. If the candidate sees much higher counts, then the global mapping is changed (Line 22). Finally, the algorithm accumulates mapped partitions and access counts for the chosen node (Line 25-26) before processing the next partition.

The algorithm complexity is O($P(log(P)+T)$, where P is the number of partitions and T is the number of threads.

(4) Parameter Choices. δ and η are both tunable parameters. δ controls the threshold to detect unbalanced load. η sets the difference between the candidate and original nodes to remap partitions. Another parameter is the number of partitions. As the number of partitions increases, the number of tuples in a partition decreases. It is more flexible to schedule the partitions across NUMA nodes. However, the number of partitions cannot be too large. Otherwise, the global mapping structure (`part_map`) and the partition statistics (`part_map`) cannot fit into the last level CPU caches. In our experiments, we set δ=0.3, η=1.2, and use 2048 partitions.

5 Evaluation

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

5.1 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, except for the experiments in Section 5.6 where we examine our optimizations for NUMA architectures. In our experiments, we set the NVM size according to the database sizes in the benchmarks, and vary the DRAM size to model different ratio *P* of NVM to DRAM. Presently, *P* can be 4, 8, and 16 for OptanePM [1].

OLTP Engine Designs to Compare. In Section 5.2–5.4, 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-Cache) to the remaining DRAM.

For FOEDUS [27], we obtain the code from the author, and modify it to store logs and snapshots in real NVM hardware. FOEDUS implements its own method of concurrency control. For MMDB, WBL, and Zen, we write two implementations based on Cicada [33] and DBx1000 [56], 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 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 buffer. 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 throughput 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.

In Section 5.5 and 5.6, we implement Zen+ by extending the Cicada based implementation with support for long running transactions and NUMA architectures. As previous OLTP engines (i)–(iii) do not support similar features, we mainly compare various design choices in Zen+.

Benchmarks. we run YCSB [15] 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.

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 configure the benchmark to use 2048 warehouses and 100,000 items. The initial footprint of the database is approximately 205GB, which is larger than the available DRAM.

Unless otherwise noted, for each experiment, each thread runs 500 thousand random transactions to warm up the system, then we measure the throughput by running 500 thousand random transactions per thread.

5.2 Transaction Performance

5.2.1 YCSB Performance

Varying Read/Write Ratio and Data Skew. We run the YCSB benchmark while varying the percentage of read requests and the Zipf's θ parameter in Figure 8.

Among the four OLTP engines, FOEDUS has the worst performance. It suffers from the NVM read amplification problem due to page-grained caching. Moreover, the map-reduce computation, which merges logs to snapshots, incurs computation overhead and NVM write cost. Finally, the implementation employs 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 11 and 12. Compared with MMDB and Zen, WBL sees drastically more `sfence`s in most cases, and sees significantly more `clwb`s when the workload has high skews.

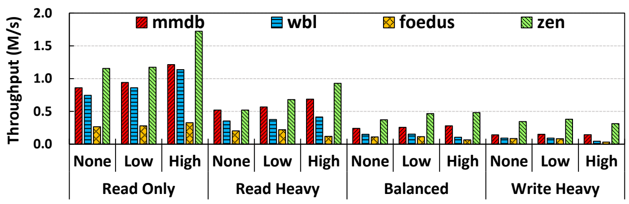


Fig. 8: YCSB performance with P=4 and 16 threads.

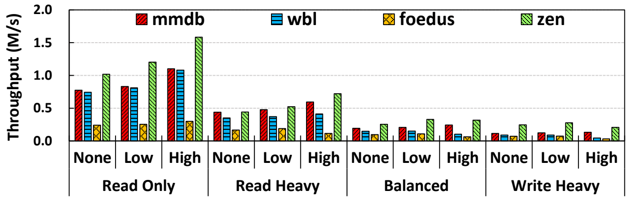


Fig. 9: YCSB performance with P=8 and 16 threads.

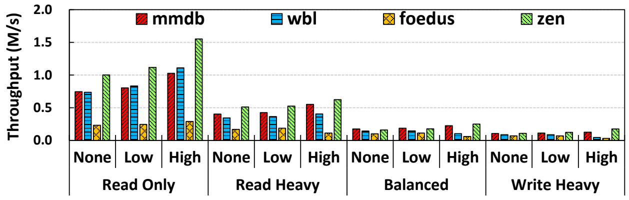


Fig. 10: YCSB performance with P=16 and 16 threads.

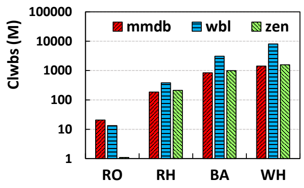


Fig. 11: Clwb counts.

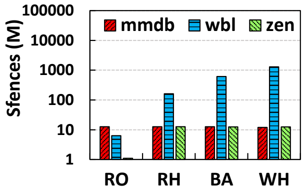


Fig. 12: Sfence counts.

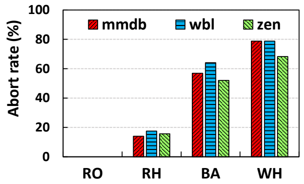


Fig. 13: Abort rate.

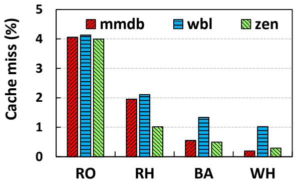


Fig. 14: Cache miss.

(Figure 11–14: High skew, P=4, 16 threads)

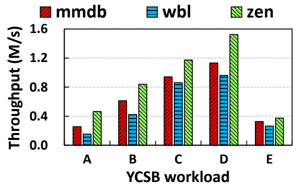


Fig. 15: Standard workloads. (Low skew, P=4, 16 threads)

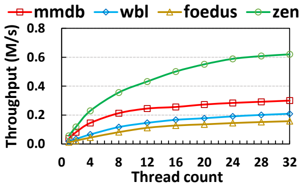


Fig. 16: YCSB scalability. (High skew, Balanced, P=4)

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 problem is NVM write amplification. If a tuple is in the (volatile) part 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 $0.75\times(1+1)+1=2.5$ NVM writes per tuple write and $0.75\times 1=0.75$ NVM write per tuple read. This explains the significant advantage of Zen over MMDB for Read Only or, No Skew and 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 outperforms MMDB under High Skew.

Our proposed design, Zen, achieves the best performance. Compared with MMDB, WBL, and FOEDUS, Zen achieves 1.25x–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. From Figure 11, we see that Zen issues much fewer `clwb` instructions than MMDB and WBL for Read-Only because Zen incurs no NVM writes, while MMDB and WBL still need to persist their logs. Moreover, Zen issues at most one `sfence` per transaction. This is the same as MMDB, but much better than WBL, which writes and persists per-tuple metadata to NVM. Furthermore, the speedups of Zen over the other designs increase as the percentage of writes, showing the benefit of Zen in reducing NVM write overhead.

We see two general trends for all the engine designs. First, transaction throughput increases as the percentage of read requests because there are fewer NVM writes and persists as shown in Figure 11. Second, higher skews bring two effects. The performance for Read-Only, Read-Heavy, and Balanced is better because more data are found in DRAM. However, higher skews result in more contention for Write-Heavy as shown in Figure 13. As a result, we see lower transaction throughput. 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. 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 10. Zen is best performing OLTP engine design among the four designs. Com-

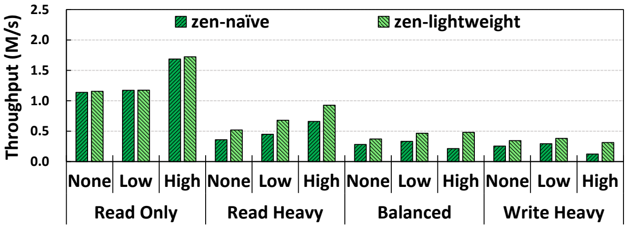


Fig. 17: NVM space management. (P=4, 16 threads)

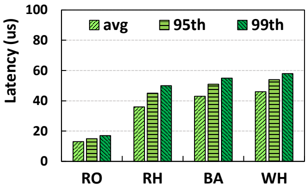


Fig. 18: Percentiles. (No skew, P=4, 16 threads)

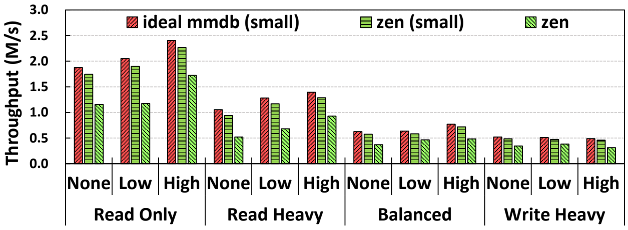


Fig. 19: Zen vs. ideal MMDB. (P=4 16 threads)

pared 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 8. 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. In another typical setting P=8, we observe similar results as shown in Figure 9.

Standard YCSB Workloads. We run experiments for the five standard workloads of YCSB [15] under low skew, P=4, and 16 threads in Figure 15: (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). From Figure 15, we see that Zen achieves 1.15x–1.82x improvements over MMDB, and 1.36x–3.04x improvements over WBL.

YCSB Scalability. We study the scalability of the four OLTP engine designs in Figure 16. 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.

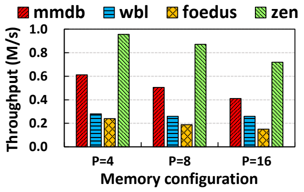


Fig. 20: TPCC performance. (16 threads)

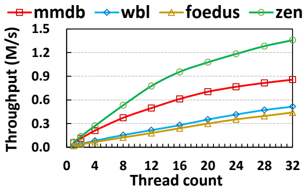
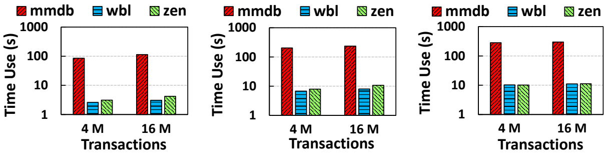


Fig. 21: TPCC scalability. (P=4)



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

Fig. 22: Recovery performance. (P=4, 16 parallel threads)

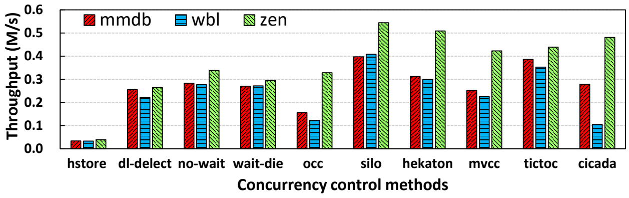


Fig. 23: YCSB performance with 10 concurrency control methods. (P=4, High Skew, Balanced, 16 threads)

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 17. 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. For the other cases, Zen significantly out-performs the naïve design. Compared to the naïve design, Zen achieves 1.41x–1.52x speedup for Read-Heavy, 1.32x–2.26x speedup for Balanced, and 1.30x–2.52x speedup for Write-Heavy. Moreover, Figure 18 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.

Comparison to ideal MMDB. We compare the performance of Zen with ideal MMDB in Figure 19. 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 19, 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.

5.2.2 TPCC Performance

TPCC performance. We run the TPCC benchmark using 16 threads while varying the memory configuration. As shown in Figure 20, the TPCC experiment shows the same trend as the YCSB experiment. Zen is the best performing OLTP engine design among the four designs. Compared with MMDB, WBL, and FOEDUS, Zen achieves 1.56x–3.98x speedup when P=4, 1.72x–4.63x speedup when P=8, and 1.75x–4.79x speedup when P=16.

TPCC scalability. We study the scalability of the four OLTP engine designs using TPCC benchmark. We set P=4 and vary the number of threads from 1 to 32. As shown in Figure 21, all designs scale well to 32 threads. We attribute the good scalability to the modestly low contention in the TPCC workload. Zen performs the best in all designs. Zen achieves a transaction throughput of 1.36 million transactions per second using 32 threads.

5.3 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 run. 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 22(a), MMDB takes 85.9s to recover from the system failure for the 100GB database. In contrast, WBL and Zen take 2.6s and 3.1s, respectively. They are an order of magnitude faster than MMDB. MMDB spends most time in loading the checkpoint and redoing logs, while 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 the additional time compared with WBL. When we increase the number of transactions from 4 million to 16 million, MMDB takes an additional 28.2s, while WBL and Zen takes merely 0.49s and 1.12s more time, respectively.

When the data size increases from 100GB in Figure 22(a) to 256GB in Figure 22(b), all three solutions take significantly longer to recover. The time complexity of the recovery algorithm is $O(n+ln(n))$, where n is the number of tuple versions in a NVM-tuple region. Therefore, the recovery time grows linearly as the number of tuples per NVM-tuple region increases. Suppose the entire 1.5TB of NVM in the experimental machine is filled with tuples and all the 64 threads are used to scan the data in parallel. Then the recovery time can be roughly estimated to be 1.5TB/100GB * 16 threads/64 threads = 3.75x as large as that for an 100GB database. Since the main operations of Zen are scanning tuples and rebuilding indices, persistent indices may effectively reduce the recovery time of Zen. We discuss persistent indices in Section 6.

Recovery for TPCC. We conduct the TPCC recovery experiment. The TPCC database contains 2048 warehouses. We use 16 parallel threads. As shown in Figure 22(c), we observe similar trends as the YCSB recovery experiment. MMDB takes 282.1s (297.1s) for 4 million (16 million) transactions. WBL takes 10.2s (11.1s) and Zen takes 10.1s (11.3s) for 4 million (16 million) transactions. Zen and WBL recover dramatically faster than MMDB.

5.4 Wide Applicability to Concurrency Control Methods

Figure 23 demonstrates that Zen supports a wide variety of concurrency control methods, including three 2PL variants (2PL with deadlock detection, wait and die, and no waiting [56]), three OCC variants [28,48,57], three MVCC variants [7,17,33], and a partition-based method (H-Store [47]). 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 8 in Figure 23. 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 is more likely to run transactions in DRAM. Aborts are also less costly in Zen because aborted transactions do not write to NVM. 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.

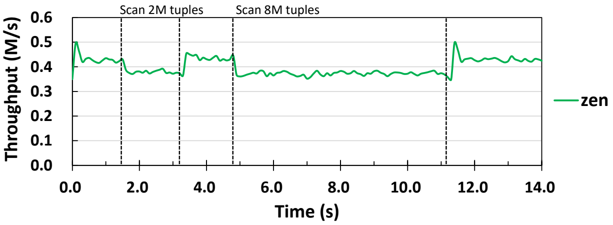


Fig. 24: Experiment with long running scan transactions. (Low skew, Balanced, P=4, 16 threads)

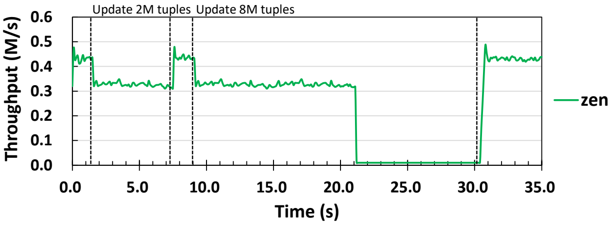


Fig. 25: Experiment with long running update transactions. (Low skew, Balanced, P=4, 16 threads)

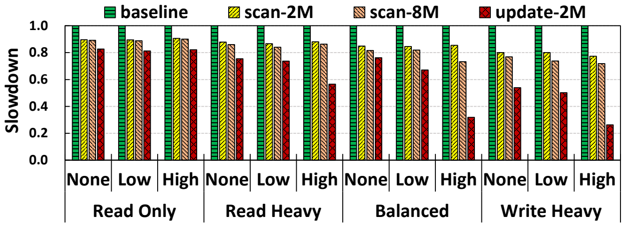


Fig. 26: Performance with long running transactions. (P=4, 16 threads)

5.5 Support for Long Running Transactions

In this section, we evaluate Zen+'s support for long running transactions. We use YCSB benchmark with 256GB database, the low skew balanced setting, P=4, and 16 threads. A Met-Cache region is 4GB large, which can cache 4 million tuples. We set $\alpha = 95\%$, $\beta = 5\%$, $\gamma = 16$ million.

Impact of Read-Only Long Running Transactions. The first experiment studies the behavior of read-only long running transactions in Zen+. One thread (denoted S) performs large scan transactions from time to time, while the other 15 threads run normal YCSB transactions. As shown in Figure 24, thread S first executes 50 thousand YCSB transactions. Then, it goes into a long running transaction that scans 2GB data (i.e., 2 million tuples). After that, it executes another 50 thousand YCSB transactions, and then a second long running transaction that scans 8GB data (i.e., 8 million tuples). The exclusive stage is not triggered in the experiment. Overall, we see a merely 11% throughput loss.

Impact of Read-Write Long Running Transactions. The second experiment studies the impact of read-write long running transactions. Thread S performs long running update transactions, while the other 15 threads run normal YCSB transactions. Note that we must avoid any conflicts between normal transactions and the long running transaction. Otherwise, the commit of a conflicting normal transaction would abort the long running transaction. For this purpose, we configure the normal transactions and the long running transactions to use disjoint key ranges. Thread S performs 50 thousand normal YCSB transactions, then updates 2GB data (2 million tuples), then runs another 50 thousand normal YCSB transactions, then updates 8GB data (8 million tuples). As shown in Figure 25, we observe that the 2GB update is handled in normal execution while the 8GB data update triggers the exclusive stage. Zen+ switches to the exclusive stage at around 21s. The throughput curve drops to 0 at this point. After the long running transaction completes, Zen+ resumes normal execution. The system throughput rises to the normal level. While its performance is poor, the exclusive stage improves the robustness of Zen+. Nevertheless, we expect such case is rare.

Slowdowns Due to Long Running Transactions. We use one thread to perform back-to-back long running transactions. We consider three types of long running transactions: scan 2GB, scan 8GB, and update 2GB data. We omit the update of 8GB data because it constantly triggers the exclusive stage. As shown in Figure 26, the scan based long running transactions cause 9.4%–24.9% slowdowns compared with the baseline without long running transactions. Because Zen+ employs MVCC, the influence of read-only long running transactions is limited. In comparison, the update based long running transactions cause 17.3%–73.7%

slowdowns compared with the baseline. As the write ratio increases and the workload skew increases, the throughput decreases significantly. This is because read-write long running transactions incur more conflicts between transactions.

5.6 Support for NUMA Architecture

In this section, we evaluate Zen+'s support for NUMA architectures. The experimental machine has two NUMA nodes.

Overall YCSB Performance of Zen+. We compare four cases: *remote*, *local*, *naïve*, and *soft partition*. One extreme is *remote*, where 16 threads run in CPU socket 0 but use remote DRAM and NVM in NUMA node 1. It gives the performance lower bound. The other extreme is *local*, where 16 threads run in CPU socket 0 and use local DRAM and NVM in NUMA node 0. It shows the performance upper bound. Note that the result of *local* is directly comparable with the experimental results in previous subsections. *naïve* and *soft partition* run 8 threads in either of the two NUMA nodes. *naïve* is NUMA-agnostic. It allocates memory using a shared pool of DRAM and a shared pool of NVM. In comparison, our proposed solution, NUMA-aware *soft partition* aims to reduce remote memory accesses while balancing the load across NUMA nodes. We empirically set the number of partitions to 2048 so that the global partition map and statistics collection structures can reside in the L3 cache.

As shown in Figure 27, *remote* and *local* achieve the worst and the best performance as expected. *naïve* achieves 1.03x-2.17x speedups compared with *remote*. *soft partition* achieves 1.13x-3.72x speedups compared with the *remote*. It performs NUMA-local writes to eliminate remote NVM writes. It also decreases remote reads by NUMA-aware transaction assignment. In general, as the workload skew increases, the performance of both *naïve* and *soft partition* increases because of better memory locality. As the write ratio increases, their performance decreases because NVM writes are more costly than NVM reads.

NUMA-Aware Soft Partition vs. NUMA-Local Write. We compare the NUMA-aware soft partition design with the NUMA-local write design. As shown in Figure 28, *soft partition* outperforms *numa-local write* in most workload setting by up to 1.32x. In the read-only cases, soft partition can not adjust the store position of tuples. The NUMA-aware transaction assignment and dynamic mapping of partitions may incur at most 5% overhead.

Scalability. Figure 29 studies the scalability of the four cases by varying the number of threads from 1 to 64. Note that there are 32 hardware threads in each CPU socket. We extend *remote* and *local* curves beyond 32 threads as flat lines. From the figure, we see that *soft partition* scales up to 64 threads. It outperforms *naïve* by up to 1.36x because of its better read-write locality.

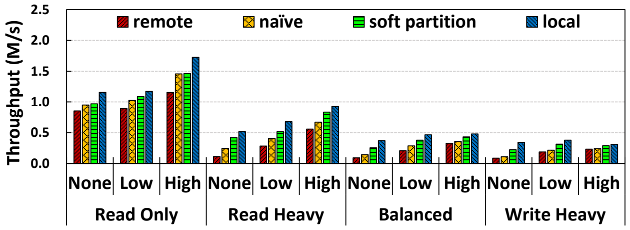


Fig. 27: Performance of NUMA designs. (P=4, 16 threads)

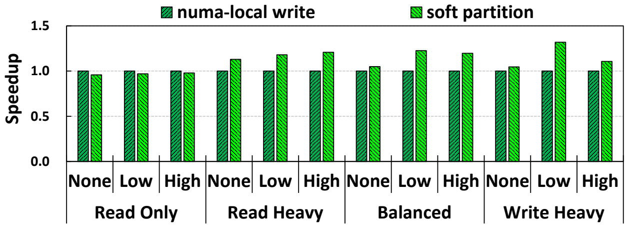


Fig. 28: Comparison with NUMA-local Design (YCSB, P=4, 16 threads)

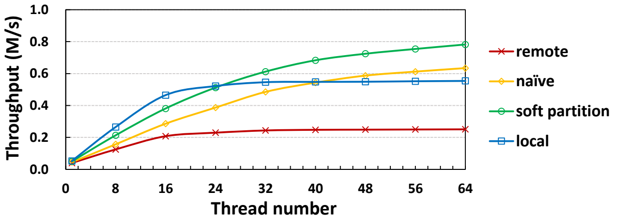


Fig. 29: Scalability of NUMA designs. (Low skew,Balanced, P=4)

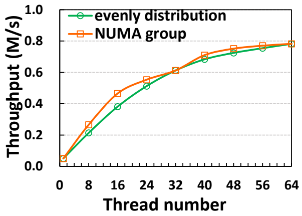


Fig. 30: Thread distribution strategies. (Low skew, Balanced, P=4)

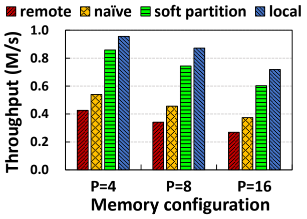


Fig. 31: TPCC NUMA performance. (P=4, evenly distributed)

Thread to Node Binding Strategy. We consider two strategies to bind threads to NUMA nodes for *soft partition*. Suppose the OLTP engine is allowed to run T threads. The *evenly distribution* strategy binds $T/2$ threads to each of the NUMA nodes. In contrast, the *NUMA group* strategy exploits as many CPU cores as possible in one NUMA node, before considering the other NUMA node. For example, if T=20, then *evenly distribution* binds 10 threads to either NUMA node, while *NUMA group* puts 16 threads to NUMA 0 and 4 threads to NUMA 1. As shown in Figure 30, we observe that *NUMA group* outperforms *evenly distribution* by up to 1.24x. As the number of threads increases from 1 to 16, *NUMA group* significantly outperforms *evenly distribution* because *NUMA group* exploits better memory locality. Note that when there are 32 or 64 threads, the two strategies are the same. As the number of threads increases from 32 to 64, *NUMA group* slightly outperforms *evenly distribution*, but the benefit is less significant because the distribution strategies become similar with more threads. Overall, it is advisable to use *NUMA group* if possible.

TPCC Performance of Zen+. Finally, we conduct the experiment in a benchmark with multiple related tables. We run the TPCC benchmark as specified in the previous section. There are 2048 warehouses. We use the warehouse id to partition the tables and evenly distribute the partitions between NUMA nodes. As shown in Figure 31, compared with the ideal *local* case, *soft partition* shows only 10.2% slowdown. Compared to *naïve*, *soft partition* achieves a factor of 1.54x improvement. This shows that *soft partition* captures the characteristics of the TPCC workload well, and it can effectively reduce remote accesses across NUMA nodes.

6 Discussion

In this section, we discuss a number of interesting design issues for Zen/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 [54], WB-Tree [12], FP-Tree [39], HiKV [53], and LB+tree [34] to improve recovery performance. Besides, we can exploit previous index designs to reduce DRAM space consumption for indices. The dual-stage hybrid index architecture [58] saves space by placing aged index entries into a more compact structure. Selective persistence in NV-Tree [54], FP-Tree [39], and LB+-Tree [34] 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 to original DRAM-based indices.

Optional DRAM-Based Logs. Zen removes NVM-based logging 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.

Dealing with Variable Sized Tuples. In our current implementation, we treat the maximum size of the variable sized tuples in a table as the "fixed size" in NVM-tuple allocation. To better utilize main memory space, we discuss an alternative design in the following. In DRAM, we exploit a memory heap to store variable sized fields. A tuple in the Met-Cache can store pointers to these fields. Hence, variable sized tuples can be handled as fixed-sized tuples. In NVM, we maintain multiple NVM page types such that the tuple slots in a type-i page are of 2^i bytes large. Then, a tuple of length L is stored in a type-i page such that $2^{i-1} < L \leq 2^i$. We can persist variable sized tuples in proper fixed sized slots. Like our current design, NVM tuple allocation incurs no NVM writes. One problem of this approach is that we now have multiple NVM page types for every table. To simplify the management of NVM pages, we can use part of the 63-bit tuple ID to include the table ID so that the tuple ID is unique across the OLTP database. In this way, we only need to manage a single set of NVM page types because tuples from different tables can be mixed in the same page.

Limitations of Zen/Zen+. First, Zen/Zen+ cannot support OLTP databases larger than NVM memory. It would be interesting to study Zen's optimizations to improve the 3-tier design [46, 60]. 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. For Zen+, the parameters of the MVCC-based adaptive execution technique can be better tuned. However, this often requires a good knowledge of the OLTP workload. It is interesting to automatically tune the parameters by dynamically observing the workload. Finally, NUMA-aware soft partition in Zen+ requires users to declare partition keys. This optimization also relies on the identification of partition IDs for transaction operations. Therefore, it mainly targets short/simple transactions.

7 Conclusion

In this paper, we study the OLTP engine design for NVM memory. After examining the existing OLTP engine designs for NVM, we find three design challenges: (i) tuple metadata modifications, (ii) NVM write redundancy, and (iii) NVM space management. 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. Moreover, we propose Zen+, which extends Zen with MVCC-based adaptive execution and NUMA-aware soft partition to robustly and effectively support long running transactions and NUMA architectures. Experimental results on a real machine equipped with Intel Optane DC Persistent Memory show that Zen achieves 1.0x-10.1x improvements over existing NVM based designs for YCSB and TPCC benchmarks. The recovery time of Zen is on the order of a few seconds for a database of a few hundred GB in size. Moreover, experimental results also show that Zen+ supports both read-only and read-write long running transactions with reasonable cost. NUMA-Aware soft partition adapts to workloads and achieves the performance close to the local NUMA setting. In conclusion, we believe Zen/Zen+ is a viable solution to support OLTP transactions in NVM based system.

Acknowledgements This work is partially supported by National Key R&D Program of China (2018YFB1003303) and Natural Science Foundation of China (62172390). Shimin Chen is the corresponding author.

References

1. Intel Optane DC persistent memory architecture and technology. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html (2019)
2. TPC benchmark C. http://www.tpc.org/tpcc/ (2020)
3. Apalkov, D., Khvalkovskiy, A., Watts, S., Nikitin, V., Tang, X., Lottis, D., Moon, K., Luo, X., Chen, E., Ong, A., Driskill-Smith, A., Krounbi, M.: Spin-transfer torque magnetic random access memory (STT-MRAM). ACM J. Emerg. Technol. Comput. Syst. **9**(2), 13:1–13:35 (2013)
4. Arulraj, J., Levandoski, J.J., Minhas, U.F., Larson, P.: Bztree: A high-performance latch-free range index for non-volatile memory. Proc. VLDB Endow. **11**(5), 553–565 (2018)
5. Arulraj, J., Pavlo, A., Dulloor, S.: 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, pp. 707–722 (2015)
6. Arulraj, J., Perron, M., Pavlo, A.: Write-behind logging. Proc. VLDB Endow. **10**(4), 337–348 (2016)
7. Bernstein, P.A., Goodman, N.: Concurrency control in distributed database systems. ACM Comput. Surv. **13**(2), 185–221 (1981)
8. Blagodurov, S., Zhuravlev, S., Dashti, M., Fedorova, A.: A case for numa-aware contention management on multicore systems. In: 2011 USENIX Annual Technical Conference, Portland, OR, USA, June 15-17, 2011 (2011)
9. Böttcher, J., Leis, V., Neumann, T., Kemper, A.: Scalable garbage collection for in-memory MVCC systems. Proc. VLDB Endow. **13**(2), 128–141 (2019)
10. Cao, T., Salles, M.A.V., Sowell, B., Yue, Y., Demers, A.J., Gehrke, J., White, W.M.: 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, pp. 265–276 (2011)
11. Chen, S., Gibbons, P.B., Nath, S.: 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, pp. 21–31 (2011)
12. Chen, S., Jin, Q.: Persistent b+-trees in non-volatile main memory. Proc. VLDB Endow. **8**(7), 786–797 (2015)
13. Coburn, J., Caulfield, A.M., Akel, A., Grupp, L.M., Gupta, R.K., Jhala, R., Swanson, S.: 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, pp. 105–118 (2011)
14. Condit, J., Nightingale, E.B., Frost, C., Ipek, E., Lee, B.C., Burger, D., Coetzee, D.: 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, pp. 133–146 (2009)
15. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: 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, pp. 143–154 (2010)
16. DeWitt, D.J., Katz, R.H., Olken, F., Shapiro, L.D., Stonebraker, M., Wood, D.A.: Implementation techniques for main memory database systems. In: SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984, pp. 1–8 (1984)
17. Diaconu, C., Freedman, C., Ismert, E., Larson, P., Mittal, P., Stonecipher, R., Verma, N., Zwilling, M.: 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, pp. 1243–1254 (2013)
18. Eswaran, K.P., Gray, J., Lorie, R.A., Traiger, I.L.: The notions of consistency and predicate locks in a database system. Commun. ACM **19**(11), 624–633 (1976)
19. Fang, R., Hsiao, H., He, B., Mohan, C., Wang, Y.: 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, pp. 1221–1231 (2011)
20. Gao, S., Xu, J., Härder, T., He, B., Choi, B., Hu, H.: Pcmlogging: Optimizing transaction logging and recovery performance with PCM. IEEE Trans. Knowl. Data Eng. **27**(12), 3332–3346 (2015)
21. Graham, D.H.: Intel optane technology products - what's available and what's coming soon. https://software.intel.com/en-us/articles/3d-xpointtechnology-products (2019)
22. Hasanzadeh-Mofrad, M., Melhem, R.G., Ahmad, M.Y., Hammoud, M.: Graphite: A numa-aware HPC system for graph analytics based on a new MPI * X parallelism model. Proc. VLDB Endow. **13**(6), 783–797 (2020)
23. Haubenschild, M., Sauer, C., Neumann, T., Leis, V.: Rethinking logging, checkpoints, and recovery for high-performance storage engines. In: Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020, pp. 877–892 (2020)
24. Huang, J., Schwan, K., Qureshi, M.K.: Nvram-aware logging in transaction systems. Proc. VLDB Endow. **8**(4), 389–400 (2014)
25. Kim, J., Cho, H., Kim, K., Yu, J., Kang, S., Jung, H.: Long-lived transactions made less harmful. In: Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020, pp. 495–510 (2020)
26. Kim, W., Kim, J., Baek, W., Nam, B., Won, Y.: NVWAL: exploiting NVRAM in write-ahead logging. In: Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2016, Atlanta, GA, USA, April 2-6, 2016, pp. 385–398 (2016)
27. Kimura, H.: 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, pp. 691–706 (2015)
28. Kung, H.T., Robinson, J.T.: On optimistic methods for concurrency control. ACM Trans. Database Syst. **6**(2), 213–226 (1981)
29. Lee, J., Kim, K., Cha, S.K.: 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, pp. 173–182 (2001)
30. Lehman, T.J., Carey, M.J.: 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, pp. 104–117 (1987)

31. Leis, V., Boncz, P.A., Kemper, A., Neumann, T.: Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In: International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014, pp. 743–754. ACM (2014)
32. Lepers, B., Quéma, V., Fedorova, A.: Thread and memory placement on NUMA systems: Asymmetry matters. In: 2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA, pp. 277–289 (2015)
33. Lim, H., Kaminsky, M., Andersen, D.G.: 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, pp. 21–35 (2017)
34. Liu, J., Chen, S., Wang, L.: Lb+-trees: Optimizing persistent index performance on 3dxpoint memory. Proc. VLDB Endow. **13**(7), 1078–1090 (2020)
35. Liu, M., Zhang, M., Chen, K., Qian, X., Wu, Y., Zheng, W., Ren, J.: Dudetm: Building durable transactions with decoupling for persistent memory pp. 329–343 (2017)
36. Maas, L.M., Kissinger, T., Habich, D., Lehner, W.: BUZZARD: a numa-aware in-memory indexing system. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013, pp. 1285–1286. ACM (2013)
37. Memarzia, P., Ray, S., Bhavsar, V.C.: The art of efficient in-memory query processing on NUMA systems: a systematic approach. In: 36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020, pp. 781–792. IEEE (2020)
38. Neumann, T., Mühlbauer, T., Kemper, A.: 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, pp. 677–689 (2015)
39. Oukid, I., Lasperas, J., Nica, A., Willhalm, T., Lehner, W.: Fp-tree: 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, pp. 371–386 (2016)
40. Oukid, I., Lehner, W., Kissinger, T., Willhalm, T., Bumbulis, P.: Instant recovery for main memory databases. In: Seventh Biennial Conference on Innovative Data Systems Research, CIDR 2015, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings (2015)
41. Pelley, S., Wenisch, T.F., Gold, B.T., Bridge, B.: Storage management in the NVRAM era. Proc. VLDB Endow. **7**(2), 121–132 (2013)
42. Psaroudakis, I., Scheuer, T., May, N., Sellami, A., Ailamaki, A.: Scaling up concurrent main-memory column-store scans: Towards adaptive numa-aware data and task placement. Proc. VLDB Endow. **8**(12), 1442–1453 (2015)
43. Psaroudakis, I., Scheuer, T., May, N., Sellami, A., Ailamaki, A.: Adaptive numa-aware data placement and task scheduling for analytical workloads in main-memory column-stores. Proc. VLDB Endow. **10**(2), 37–48 (2016)
44. Raoux, S., Burr, G.W., Breitwisch, M.J., Rettner, C.T., Chen, Y., Shelby, R.M., Salinga, M., Krebs, D., Chen, S., Lung, H., Lam, C.H.: Phase-change random access memory: A scalable technology. IBM J. Res. Dev. **52**(4-5), 465–480 (2008)
45. Ren, K., Diamond, T., Abadi, D.J., Thomson, A.: 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, pp. 1539–1551 (2016)
46. van Renen, A., Leis, V., Kemper, A., Neumann, T., Hashida, T., Oe, K., Doi, Y., Harada, L., Sato, M.: 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, pp. 1541–1555 (2018)
47. Stonebraker, M., Madden, S., Abadi, D.J., Harizopoulos, S., Hachem, N., Helland, P.: 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, pp. 1150–1160 (2007)
48. Tu, S., Zheng, W., Kohler, E., Liskov, B., Madden, S.: 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, pp. 18–32 (2013)
49. Volos, H., Tack, A.J., Swift, M.M.: 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, pp. 91–104 (2011)
50. Wang, T., Johnson, R.: Scalable logging through emerging non-volatile memory. Proc. VLDB Endow. **7**(10), 865–876 (2014)
51. Wang, T., Kimura, H.: Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. Proc. VLDB Endow. **10**(2), 49–60 (2016)
52. Wang, Y., Jiang, D., Xiong, J.: Numa-aware thread migration for high performance nvmm file systems. In: 36th Symposium on Mass Storage Systems and Technologies, MSST 2020, Santa Clara, CA, USA, October 29-30, 2020 (2020)
53. Xia, F., Jiang, D., Xiong, J., Sun, N.: 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, pp. 349–362 (2017)
54. Yang, J., Wei, Q., Chen, C., Wang, C., Yong, K.L., He, B.: 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, pp. 167–181 (2015)
55. Yang, J.J., Williams, R.S.: Memristive devices in computing system: Promises and challenges. ACM J. Emerg. Technol. Comput. Syst. **9**(2), 11:1–11:20 (2013)
56. Yu, X., Bezerra, G., Pavlo, A., Devadas, S., Stonebraker, M.: Staring into the abyss: An evaluation of concurrency control with one thousand cores. Proc. VLDB Endow. **8**(3), 209–220 (2014)
57. Yu, X., Pavlo, A., Sánchez, D., Devadas, S.: 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, pp. 1629–1642 (2016)
58. Zhang, H., Andersen, D.G., Pavlo, A., Kaminsky, M., Ma, L., Shen, R.: 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, pp. 1567–1581 (2016)
59. Zheng, W., Tu, S., Kohler, E., Liskov, B.: 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, pp. 465–477 (2014)
60. Zhou, X., Arulraj, J., Pavlo, A., Cohen, D.: Spitfire: A three-tier buffer manager for volatile and non-volatile memory. In: SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021, pp. 2195–2207 (2021)