Falcon: Fast OLTP Engine for Persistent Cache and Non-Volatile Memory

Zhicheng Ji[♠] Kang Chen[♠♡] Leping Wang[♠] Mingxing Zhang [♠] Yongwei Wu[♠]
[♠]Tsinghua University [♡]Zhongguancun Laboratory

Abstract

Non-volatile memory(NVM) has the properties of both byte addressable and persistence, which provides new opportunities for building on-line transaction processing (OLTP) engines. Recently, a new feature called eADR puts CPU cache also in the persistence domain. Existing OLTP engines are based on *volatile cache* and now have the opportunity to improve performance further and reduce programming complexity with *persistent cache*.

This paper studies the impact of persistent cache on OLTP engines and revisits the existing designs. We have observed that naively removing the flush instructions can trigger the write amplification because of the granularity mismatch between the cache line and NVM access. We propose Falcon, a new OLTP engine for eADR-enabled NVM. Falcon is based on the in-place update architecture. The **small log window** design in Falcon avoids the NVM writes while logging. The **selective data flush** design reduces the data flush and the write amplification while flushing data. Evaluations show that under TPC-C workloads, Falcon achieves 1.21× ~ 1.35× improvement over the state-of-the-art OLTP engine.

CCS Concepts: • **Hardware** → **Emerging architectures**; • **Information systems** → **Database transaction processing**; **DBMS engine architectures**.

Keywords: Non-volatile memory; Persistence; Performance; Persistence Cache; OLTP Engine

ACM Reference Format:
Zhicheng Ji[♠] Kang Chen[♠♡] Leping Wang[♠] Mingxing Zhang [♠] Yongwei Wu[♠]. 2023. Falcon: Fast OLTP Engine for Persistent Cache and Non-Volatile Memory. In *ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP '23), October 23–26, 2023, Koblenz, Germany*. ACM, Koblenz, Germany, 14 pages. https://doi.org/10.1145/3600006.3613141

SOSP '23, October 23–26, 2023, Koblenz, Germany

ACM ISBN 979-8-4007-0229-7/23/10…$15.00
https://doi.org/10.1145/3600006.3613141

1 Introduction

NVM has the properties of both byte addressable and persistence, which provide new opportunities for building online transaction processing (OLTP) engines. FOEDUS[30], WBL[15], and Zen[35] are typical existing OLTP engines designed specifically for NVM. Other related works include persistent indexes [27, 33, 39, 57], key-value stores[18, 24, 51, 52, 59], and performance tuning using NVM[41, 42, 46].

The CPU cache is not persistent in the first commercially available generation NVM, e.g., Intel ADR[10]. ADR only guarantees that data reaching the memory controller is persistent. Data in the CPU cache is still volatile (*volatile cache*), i.e., data in the CPU cache is lost after a power failure. Recent developments have put the CPU cache also in the persistent domain (*persistent cache*), such as Intel's eADR[4]. Persistent cache can significantly reduce the programming complexity. Programmers can safely remove all flush-related instructions (e.g., `clwb`) but keep the implementation correct. Notice that the memory fence instructions, such as `sfence`, are still needed to maintain the memory order.

However, naively removing the flush instructions can be detrimental to performance in persistent cache (§3.3). Although the eADR makes the in-CPU cache persistent and the flush instructions are no longer necessary, there is a write amplification problem when a cache line is evicted. The reason comes from the granularity mismatch between a cache line (typically 64B [13]) and an NVM write (256B [26, 53]). Suppose the evicted data is smaller than the size of an NVM write. In this case, it introduces an additional NVM read to get the data block, modifies it with the evicted data, and writes the block back to NVM, resulting in write amplification. To better use the performance provided by persistent cache, we need to reduce this write amplification.

The distinctive feature of persistent cache also changes the assumption behind the current NVM OLTP engine designs. Current designs assume that *"the logging for in-place update introduces additional NVM writes"*. To achieve crash consistency, OLTP engines with *in-place update* typically ① record logs to make operations persistent before ② in-place apply the changes to tuples. Under volatile cache, recording the logs and updating the tuples are the two unavoidable manual NVM writes. WBL [15] and Zen [35] use the *out-of-place update* approach to reduce the NVM writes. A new version for a tuple is created and appended to the tuple heap, i.e., the log is directly used as data. However, under persistent cache, logs generated in the cache are already persistent.

There is no explicit manual flush needed. Thus, the previous assumption does not hold under persistent cache.

To fully exploit the potential of persistent cache, we propose Falcon, a novel OLTP engine for NVM with persistent cache. Falcon uses the in-place update design, which first records logs and then in-place updates the tuples. One advantage of the in-place update is that it can be integrated with persistent indexes. Since the in-place update does not change the addresses of the tuples, there is no need to modify indexes frequently. Thus, Falcon can integrate recent NVM indexes [22, 38, 39, 57] to achieve fast recovery for the entire OTLP engine. This is not the case for the out-of-place update because the out-of-place update approach changes the address of a tuple when the newly generated version is used as the updated tuple. Indexes must be modified to point to the newest version. Existing NVM OLTP engines have to put indexes in DRAM to avoid frequent NVM writes that modify indexes. As a result, it takes a long time to rebuild indexes during recovery using the out-of-place update approach.

Falcon tries to reduce the amount of data written to NVM, and reduce the overhead while flushing data using the following two designs for logs and tuple data, respectively.

D1: Small log window. Falcon tries to avoid evicting logs to NVM during normal operation. Falcon maintains a reusable small log window in each thread to record logs for active transactions. Each small log window is a circular buffer to hold redo logs for a small number (2~3) of transactions. The total size of all windows is small enough to fit inside the CPU cache. This design has two critical properties: (1) The logs are persistent even in the cache because of persistent cache. This property guarantee crash consistency. (2) Since no explicit flush is required, there is no NVM write overhead as long as the small log window is carefully controlled and not evicted to NVM.

D2: Selective data flush. Unlike logs, the data size is so large that evicting data to NVM is inevitable during normal operations. Falcon has two optimizations on data storage, ① reducing the overhead while data is written to NVM by **hinted flush** and ② reducing the amount of data written to NVM by **hot tuple tracking**. The **hinted flush** reduces the write amplification caused by granularity mismatch (§3.2). Because the CPU does not provide instructions to control the cache line eviction order, counter-intuitively, we bring back `clwb` instruction for performance. Hinted flush uses the instruction sequence of `<sfence + clwbs>`. The multiple `clwb` instructions flush multiple cache lines. If the data of these cache lines are contiguous in memory, the underlying memory module can merge multiple cache line writes into one NVM write. The **hot tuple tracking** uses a small LRU cache to track hot tuples which are never manually flushed, reducing the amount of data written to NVM.

Notice that our two designs are general because similar features can be found in other persistent cache implementations, such as BBB[14] and Global Persistent Flush (GPF) in CXL 3.0[2]. Similar granularity mismatch and uncontrollable cache line eviction problems can also be found in other NVM implementations like Samsung's CXL SSD[43].

Falcon support different concurrency control algorithms, including 2PL, TO, and OCC. To support multi-version concurrency control (MVCC), Falcon places old versions of tuples in DRAM.

This paper has made the following contributions:

1. We analyze the impact of persistent cache on NVM OLTP engines.
2. We propose two essential designs (Small log window and Selective data flush) for persistent cache-enabled NVM OLTP engines. We implement Falcon, a new NVM OLTP engine based on these designs.
3. Evaluations show that under TPC-C workloads, Falcon achieves 1.21× ~ 1.35× improvement over the state-of-the-art OLTP engine.

2 Background

This section provides a brief introduction to OLTP engines, specifically in NVM. To simplify the discussion, we focus on the architecture that separates data from indexes [7, 15, 35]. A few engines (e.g., MYSQL[6]) store data and indexes together. Our design applies to both types.

2.1 NVM OLTP Engine

The OLTP engine manages data storage and provides data access for an OLTP database. Traditional disk-oriented[6, 7] and memory-oriented[23, 34, 40] OLTP engines all use block devices, such as HDDs or SSDs, for persistent storage. Logs are widely used to ensure atomicity and crash consistency. Because block devices are slow, these OLTP engines use the in-DRAM buffer pool to cache tuples and group commits to reduce log overhead. Block devices can only be accessed at block granularity (512 bytes or more), so data marshaling/unmarshaling and read/write amplification is inevitable.

OLTP engines in NVM avoid the data marshaling / unmarshaling overhead by directly exposing the tuple data based on the byte addressability of NVM. Logs are still required to guarantee atomicity and crash consistency because the CPU can only guarantee the atomicity of a single cache line. A transaction can often span multiple cache lines. After logging, the transaction can either modify the original tuple (*in-place update*) or use the logs as the updated version for the tuple (*out-of-place update*).

2.1.1 In-place Update. The in-place update modifies the tuples directly. To ensure crash consistency, updates are first logged (using redo and/or undo logs) before the corresponding tuples are modified (Figure 1(a)(b)). During recovery, redo logs recover the committed transactions, and undo logs roll back the uncommitted transactions. The in-place update has to perform *the NVM write twice*, once for the log and once for the in-place tuple modification.

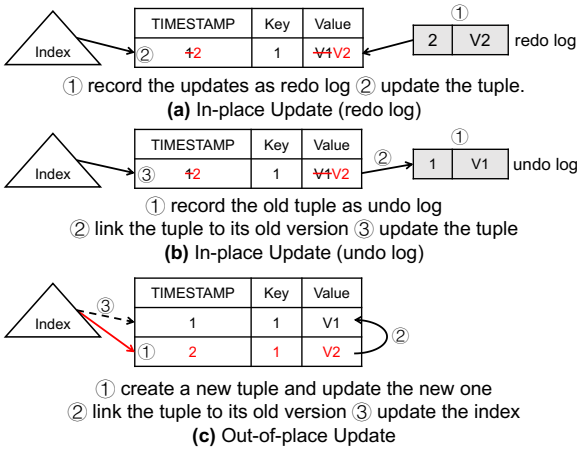

Figure 1. In-place update vs. out-of-place update.

2.1.2 Out-of-place Update.

The out-of-place update (Figure1(c)) does not change the original tuple but generates a new version. Unlike the in-place update, which modifies the tuple data after logging, the out-of-place update uses the new version as the latest version for a tuple. This out-of-place approach is often referred to as log-as-data [24] or log-free [35], as well as an extension to the LSM data structure[18, 51, 52, 59]. The out-of-place update *reduces the NVM write* because it does not modify the original tuple.

2.2 Indexes for Volatile Cache

Indexes are important because they store tuple addresses and are the entry point to find the desired tuples. Indexes can be stored either in DRAM or in NVM. There are many recent works [22, 38, 39, 57] on how to build NVM indexes. Because of the byte-addressability, indexes in NVM can expose the data structure directly for access. NVM indexes can recover fast because indexes are not lost after a system crash.

The ways of updating tuples influence the index location. The in-place update does not change the address of the tuple. No index modification is needed for the update. NVM indexes can be used because there is no frequent NVM write. For the out-of-place update, new versions are generated during transaction execution. The indexes must be modified to point to the latest version because the tuple address has been changed. Indexes need frequent modifications. Recent NVM OLTP engines do not use NVM indexes because of the performance considerations for normal operations. For example, Zen uses DRAM indexes instead of NVM indexes because it uses the out-of-place update approach to run transactions. Though in-DRAM indexes can afford frequent index modifications during normal operations, rebuilding the indexes during recovery is time-consuming.

2.3 Multi-version Concurrency Control

Many NVM OLTP engines use multi-version concurrency control to support non-blocking read-only transactions. As discussed above, both the in-place and out-of-place updates can create version chains (Figure 1) to support MVCC. The in-place update approach uses the undo logs as the old versions of a tuple. Each undo log also points to its predecessor. When a transaction is committed, the tuple in the tuple heap points to the latest old version in the version chain. Similarly, a version chain can be created for the out-of-place update as the new version points to its predecessor. Because NVM is byte-addressable, creating version chains in NVM using the memory addresses is not difficult.

3 Persistent Cache for NVM

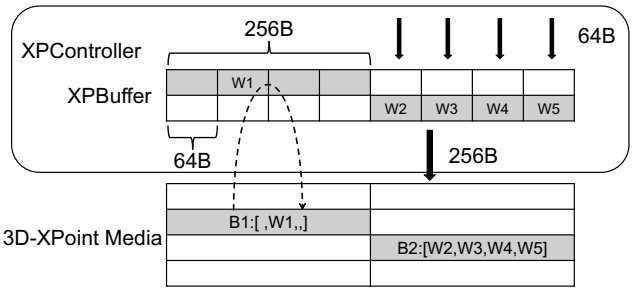

Figure 2. XPBuffer. XPBuffer is a write buffer inside the Optane NVM module to address the granularity mismatch. The 256B block needs to be loaded to XPBuffer from the 3D XPoint storage media first if the size of data for writing is smaller than the storage access granularity.

3.1 Persistent Domains in NVM

The first generation of non-volatile memory implementation has *volatile cache*, such as the Intel ADR mode [10]. Recent implementation has *persistent cache*, such as the Intel eADR mode [4]. ADR mode makes data persistent by flushing them to the write pending queue (WPQ) in the memory controller. Explicit cache line flush instructions (such as `clwb` or `clflush`) or implicit cache line evictions can all persist data. Because cache line eviction is not controllable, programmers must use cache line flush instructions and memory fence instructions (e.g., `sfence`) to flush data manually. The eADR mode puts the CPU cache also in the persistent domain. Data in the CPU cache is already persistent. Therefore, the cache line flush instructions are not needed.

3.2 Granularity Mismatch

A granularity mismatch exists between the cache line size (typically 64B) and the NVM storage media access size (265B in Intel Optane NVM). To address this mismatch, a buffer layer, XPBuffer, is integrated into Optane NVM modules (Figure 2). When a single cache line is evicted (W1), the

XPController fetches one 256B block B1 containing this cache line from the 3D XPoint storage media, modifies B1 with W1, and writes the modified B1 back. This read-modify-write process amplifies the original 64B write with an extra read. If neighboring cache line evictions (W2, W3, W4, W5) are within the same block (B2), the XPController merges them into a single 256B block (B2), which can avoid the write amplification. Previous research tried to reduce such amplification by careful data structure design (such as 256 bytes for a node in B+tree or hash bucket)[38, 39, 57].

3.3 `clwb` for Persistent Cache

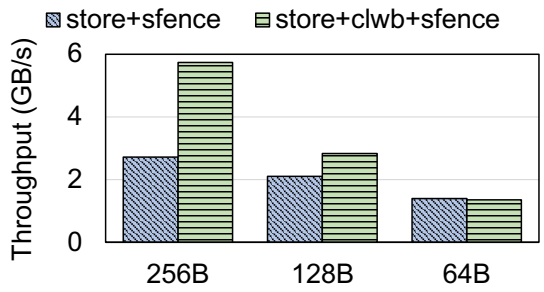

Figure 3. Bandwidth for data stores w/wo `clwb`s.

In persistent cache (such as Intel eADR), `clwb` is not necessary to ensure correctness because data in the CPU cache is already persistent. As modern CPUs usually use set associative cache, there is no correlation between different cache lines in different sets. Thus, if a single cache line or a small number of cache lines are evicted, the granularity mismatch will undoubtedly lead to the write amplification[47, 53] (§3.2). In practice, counterintuitively, we can use `clwb` to proactively flush cache lines to reduce the possibility of write amplification. If multiple cache lines of adjacent addresses are flushed, the underlying memory module has the opportunity to merge cache lines.

Figure 3 shows the results of `clwb` effects for persistent cache (using eADR). The experiment starts by generating a random but aligned address, after which the corresponding amount (X-axis) of data is written. This process repeats one million times. One case only uses the store instructions, while another case uses the combination of <store + `clwbs`>. Because `clwb` can only flush one cache line, multiple `clwbs` are used for flushing multiple cache lines. Results show that `clwb` helps XPController to merge multiple cache lines to achieve higher throughput. Thus, `clwb` is still valuable in persistent cache (eADR) to reduce the write amplification.

4 Design

We revisit the logging mechanisms for NVM OLTP engines with persistent cache and propose two new designs (**small log window** and **selective data flush**).

4.1 Design Decisions

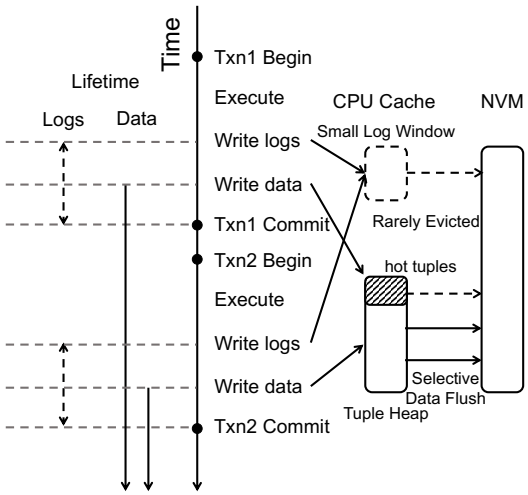

Figure 4. Lifetime of transactions. Small log window keeps logs in the CPU cache avoiding flushing to NVM.

Under volatile cache, programmers must manually use flush instructions (e.g., `clwb`) to ensure correctness. However, considering the persistent cache, logs for the in-place update are already persistent, even if they are in the CPU cache. No manual flushing is needed, which effectively reduces NVM writes. Thus, the assumption of "*the logging for in-place update introduces additional NVM writes*" does not hold. Now the in-place update is a reasonable choice and can bring additional benefits for indexes that can stay in NVM for fast recovery because of no tuple address change. Our system, Falcon, follows this analysis to build a high-performance OLTP engine using the in-place update approach and (optional) in-NVM indexes.

4.2 Lifetime of Logs for In-place Update

Figure 4 depicts the flow of committing transactions using the in-place update. Before a transaction modifies the data in NVM, the transaction logs persist. If the system crashes while modifying data in NVM, logs can re-execute the committed transactions or undo any changes made by the uncommitted transactions. Once the transaction has been completed, logs are no longer needed and can be discarded.

With persistent cache, logs generated in caches are already persistent to ensure correctness. After the log lifetime ends, the cache space can be reused for the next transaction. Thus, it is desirable not to evict logs in the cache to NVM, which is the purpose of the small log window design.

4.3 Small Log Window

Without any intervention, logs will eventually be evicted to NVM. Because logs can be safely discarded after committing

or aborting a transaction, such eviction is unnecessary. Falcon uses a local small log window in each thread to keep logs. Each small log window is a circular buffer to hold redo logs for a small number (2~3) of transactions. The total size of all log windows is small enough to fit inside the CPU cache. The window is reused for all transactions in a thread and accessed frequently. Based on the modern CPU cache replacement mechanisms (temporal and spacial locality), the space will be kept inside the CPU cache. As a result, the small log window design can effectively eliminate the NVM writes during logging but still keep logs persistent to ensure correctness. Without persistent cache (like in ADR), logs have to be flushed to NVM manually. Therefore, such NVM writing is unavoidable in volatile cache.

4.4 Selective Data Flush

Unlike the logs, the data size of a database is so large that evicting data to NVM is inevitable during normal operations. Falcon has two optimizations on data storage: ① reducing the NVM write overhead by **hinted flush**, and ② reducing the amount of data written to NVM by **hot tuple tracking**.

The **hinted flush** reduces the write amplification caused by granularity mismatch (§3.2). Unfortunately, there is no direct mechanism in modern CPUs to control the cache line eviction order because it is based on the cache replacement mechanism that is invisible to programmers. Counter-intuitively, we bring back `clwb` instruction for performance. Instead of using the <`clwbs` + `sfence`> instruction sequence, Falcon uses the <`sfence` + `clwbs`> instruction sequence. The performance is slightly better because we do not use `sfence` to wait for the completion of `clwb`. Notice that we must still use `sfence` instruction to ensure correctness. The multiple `clwb` instructions flush multiple cache lines. If the data of these cache lines are contiguous in memory, the underlying memory module can merge multiple cache line writes into one NVM write.

The **hot tuple tracking** uses a small LRU cache to track hot tuples. Contrary to hinted flush, hot tuples are never manually flushed to NVM, which effectively reducing the amount of data written to NVM. The hot tuple tracking does not apply to the out-of-place update because the tuple addresses are changed.

5 Implementation

5.1 Overview

Falcon is implemented with 14000+ lines of Rust code, including the out-of-place update for comparison [1]. Falcon stores all tuples and indexes in NVM (Figure 5), allowing instant recovery. Indexes can also stay in DRAM for performance.

NVM Space Management. Similar to previous systems [35], Falcon uses pages (2MB each) to manage the storage space. To reduce memory contention, pages are dedicated to each thread. The page allocation is NUMA aware that each thread only accesses the pages that belong to its own NUMA node. Each thread manages the internal space inside pages.

Tuple Heap. The in-NVM tuple heap holds all tuples. If multiple versions are used, the in-NVM tuple heap only holds the latest version (§5.2.3).

Version Heap. Version heap is optional. It is required when Falcon uses multi-version concurrency control algorithms. To reduce NVM writes, Falcon stores old versions of a tuple in the version heap in DRAM (§5.2.3).

Index. Falcon keep indexes separate from tuples. We use Dash [38] (Dynamic and Scalable Hashing) and NBTree [57] as our example indexes. Other indexes [22, 39] are also possible under Falcon's architecture. We wrap the source code of Dash and NBTree to call the C++ functions from Rust. We implement the update operation for Dash to support the out-of-place update . We also implement scan operations for NBTree to support TPC-C and YCSB benchmark.

Like other database systems, Falcon uses the indexed field of the tuple as the key and the address of the tuple in NVM as the value. The indexes are used to find the corresponding tuples. Because Falcon uses the in-place update approach, the indexes are not modified during tuple updates. Indexes can be stored in NVM. Indexes can also be configured to stay in DRAM for performance.

Redo Logs. Falcon uses the in-place update design with redo logs.

Read/Write Sets. During the transaction execution, the read and write sets of the transaction are recorded, where the write set also holds the redo logs. Thus, the write sets are in the small log window. Because persistent cache writes the contents to NVM when the system crashes, the write sets are still accessible during recovery.

Catalog. Falcon stores the Catalog in NVM, which records the database metadata, including the addresses of the index roots, tuple heap, and redo logs, along with the table schema, etc. The catalog will be accessed first during recovery.

The design of Falcon uses a very limited amount of DRAM, allowing most DRAM space for tuple, plan, or index cache to improve performance. How to use the DRAM efficiently is orthogonal to the study of this paper.

5.2 Transaction Processing

5.2.1 Concurrency Control. Falcon's design is neutral to concurrency control algorithms. Thus, Falcon supports a variety of concurrency control algorithms, including 2PL, TO, OCC, and MVCC.

Two Phase Locking (2PL). To support 2PL, Falcon uses 8 bytes as a lock in the *metadata* field in the tuple. The first bit acts as the write lock, and the other bits are the read locks [50]. The lock is acquired using `cas` instructions. During the lock-acquiring phase, we apply the no-wait policy to avoid deadlocks [16, 35, 55].

[1] https://github.com/madsys-dev/Falcon

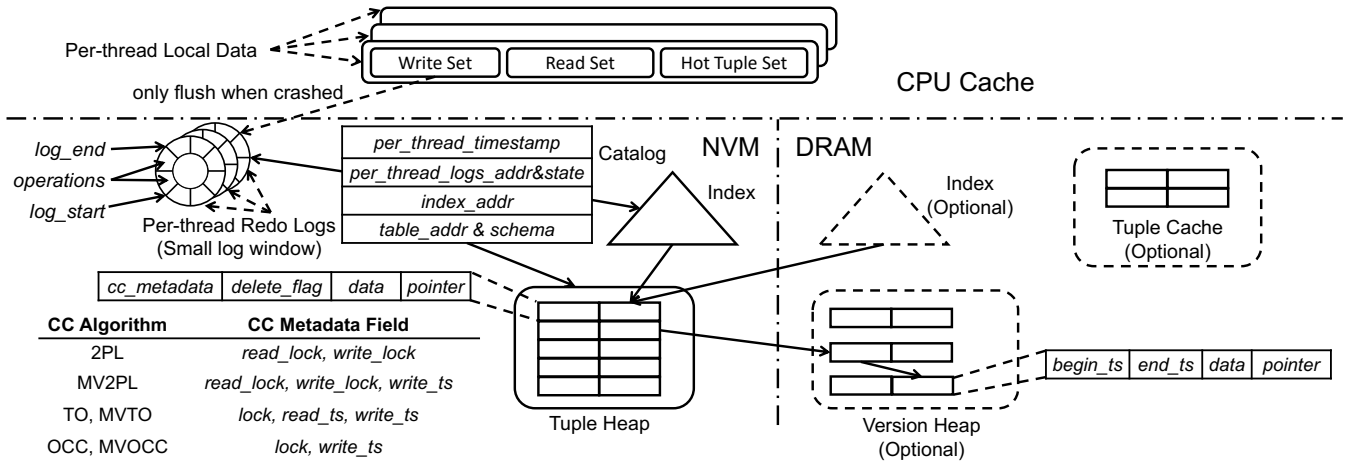

Figure 5. Falcon Architecture.

Timestamp Ordering (TO). In TO, a transaction obtains a 64-bit unique transaction ID (TID) from the hardware clock (`clock_gettime` in Linux) [2] at the beginning of the transaction [29, 31]. The sequence numbers generated by this counter also act as timestamps. Each tuple records the write timestamp (*write_ts*) from the TID of the latest writer and the read timestamp (*write_ts*) from the TID of the latest reader [50]. The first bit of the write timestamp also acts as a lock bit, indicating some transaction is updating the tuple. Transaction execution has to guarantee the timestamp order.

Optimistic Concurrency Control (OCC). Falcon also runs OCC in three phases [16, 32, 45, 56]. The write timestamp (*write_ts*), same as in TO, is used as the *version number*. The first bit of the version number acts as a lock bit.

Multi-Version Concurrency Control (MVCC). MVCC algorithms are often used to support the non-blocking read. Each tuple keeps the *write_ts* as the version number. When a transaction updates a tuple, it creates an old version. The old version also points to its predecessor. Thus, all old versions of a tuple form a *version chain*. Different from the tuple, we use two timestamps for an old version of the tuple. The begin timestamp (*begin_ts*) is the *write_ts* from the tuple before the update. It is used for the non-blocking read. When a transaction reads a tuple, it always selects the latest version with the *begin_ts* smaller than the transaction's TID or the tuple. The end timestamp (*end_ts*) is the TID of the writer transaction. It is used for garbage collection (§5.4). Falcon stores version chains in DRAM (§5.2.3).

[2]We use $\{Timestamp << 8|thread\_id\}$ as TID to avoid different transactions getting the same timestamp. After system crash and recovery, Falcon requires that `clock_gettime` gets a larger value. In rare cases, such as the broken of RTC (real time clock) or NTP (network time protocol), this condition might not hold. Falcon recovers monotonic increasing timestamps by scanning the logs, which takes negligible time.

Falcon combines MVCC with other concurrency control algorithms to implement MV2PL, MVTO, and MVOCC. In MV2PL, read-write transactions still lock tuples, while read-only transactions can access old versions using the version chain without being blocked by read-write transactions. In MVTO, tuples still keep the read timestamps. A transaction cannot modify tuples with read timestamps greater than its TID. In MVOCC, read-only transactions can access old tuple versions and do not require validation.

Algorithm 1: Transaction updates in Falcon

```
Thread-local Variables:
tid: generated at the beginning of a transaction.
write-set: all updates in the local small log window.
hot-tuple-set: a small LRU cache to track hot tuples.
Before Update:
write-set.state = UNCOMMITTED
write-set.timestamp = tid
write-set.operations = updates // redo logs.
1  Function Update(tuples, updates)
2      write-set.state = COMMITTED;
3      for (tuple, update) in (tuples, updates) do
4          tuple.do_update(update);
5          release_lock(tuple);
6      end
7      sfence;
8      for (tuple, update) in (tuples, updates) do
9          if tuple.id ∉ hot-tuple-set then
10             tuple.do_clwb(update);
11         hot-tuple-set.cache(tuple.id);
12     end
```

5.2.2 Logging and In-place Update.

Algorithm 1 shows the tuple update performed in the final stage of transaction

execution. Before the update, the modifications are buffered in the write-set in the small log window. During the update, the transaction is first committed (line 2). Later, the tuples get updated in place (line 3 to line 5). The `sfence` in line 7 ensures the updated tuples are in persistent cache. Line 8 to line 11 runs the selective flush (hinted flush in line 10 and hot tuple tracking in line 9 and line 11).

Persistent cache ensures that the write-set inside the CPU cache will flush to NVM at a system crash. During recovery, Falcon checks *write-set.state*. If it is UNCOMMITTED, there is no need to replay the updates because the tuples are not touched. The logs (updates in *write-set.operations*) can be safely discarded. If the state is COMMITTED, the logs will be replayed to guarantee that the transaction updates the corresponding tuples.

To ensure correct recovery, the updates in the write-set must be idempotent. For non-idempotent operations (such as read-modify-write operations, e.g., $value = value + 1$), they must be converted to idempotent ones such as by recording the updated values.

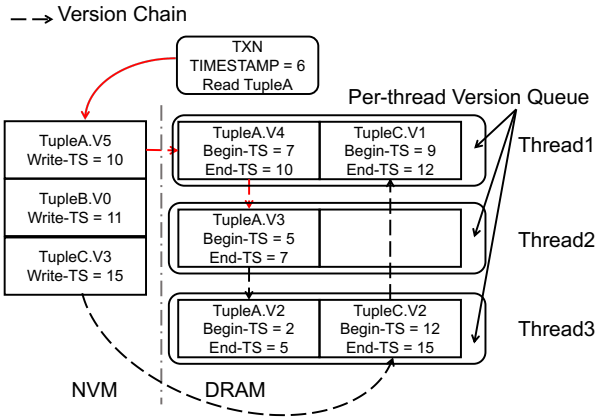

Figure 6. Version traversal. Old versions are stored in a per-thread version queue. Transaction(TS=6) travels the version chain to find out that TupleA.V3(TS=5) is the latest version in its read snapshot.

5.2.3 In-DRAM MVCC. Old versions are not restored after a system crash. Thus, Falcon does not keep old versions in NVM. Instead, old versions are stored in DRAM to form version chains. In-DRAM multiple versions can improve the performance of creating new versions during normal execution and simplify the recovery process during recovery. As shown in Figure 6, a transaction uses its TID (timestamp) to find the appropriate version to read. Figure 6 also shows the per-thread version queue design, which is used to manage the memory space for the version heap to ease the garbage collection (details in §5.4).

5.3 Recovery

Falcon's recovery has two steps: log replay, and index recovery.

Log Replay. The logging process is described in detail in Section 5.2.2. During recovery, Falcon checks the *write-set.state*. If the state is UNCOMMITTED, the transaction is not committed, and the corresponding tuples are not touched. There is no need to replay the logs in the *write-set.operations*. The space for the *write-set* is then recycled. The transaction is committed if the state is COMMITTED, but the corresponding tuples might not be modified. Thus, during recovery, the logs in *write-set.operations* are replayed.

Index Recovery. Index recovery has two steps if indexes are stored in NVM. The first step is to recover the index based on the current NVM index implementation. Many existing NVM indexes try to achieve instant recovery [27, 33, 39, 57]. The second step is to update the indexes based on the committed but possibly not completed transactions. Same as log replay, these transactions have the state as COMMITTED. After replaying the logs to modify the tuples, the indexes are updated accordingly. Notice that the index modification is idempotent.

For the performance consideration, Falcon only needs to process the tuples in the redo log (*write_set*) during recovery. The total size of all redo logs is very small ($< 1MB$). The size is also independent of the whole tuple heap size. With indexes in NVM, the recovery of indexes is also only related to the redo logs. Thus, the data involved in the recovery process is very small. Falcon can recover in milliseconds.

5.4 Garbage Collection

Falcon may generate garbage in two ways: ① one is from the deleted tuples, ② the other is from the old versions of tuples in MVCC. Falcon uses the worker threads that run transactions to do the garbage collection, i.e., no dedicated recycling threads are needed.

Deleted Tuples. Falcon does not immediately clean up the deleted tuples as they might be in another transaction's *read_set*. Instead, a *delete flag* indicates that the tuple can be recycled. The delete operation is translated to an update operation to raise the delete flag. The timestamp of the tuple is set to the TID of the transaction that deletes it.

Deleted tuples are organized by a thread in a local deleted list. Thanks to persistent cache, this deleted list is persisted in the tuple heap so it can still be found after a system crash. A deleted tuple is appended to the end of the local deleted list. Thus, the tuples in the list are naturally sorted by tuple timestamps. When a new tuple needs to be allocated, the transaction first finds a deleted tuple from the head of the local deleted list and checks its timestamp. If it can be reclaimed, the tuple space can be recycled for a new tuple. Otherwise, the original allocation method is followed.

Old Versions of Tuples. Old versions will be recycled in MVCC if they are not referenced by any transaction. Recycling old versions uses a timestamp-based algorithm, i.e., versions with timestamps smaller than TIDs of all running transactions can be recycled. The old versions in the version chain are ordered by *end_ts*. When a thread generates a new version for a tuple, the thread not only inserts the version into the version chain but also inserts it to a thread-local *version queue* to facilitate version recycling (Figure 6). Tuples in the thread local version queue are also sorted by the *end_ts* because the transaction sets the *end_ts* of the version to its own TID before inserting it into the version queue. When the length of the version queue is above a predefined threshold, the worker thread checks the version queue to recycle the versions which can be reclaimed, i.e., with *end_ts* smaller than TIDs of all running transactions.

Old versions are stored in DRAM and are automatically discarded if the system crashes. Hence, each thread only needs to create a new empty version queue during recovery.

Compared to other in-place update databases, Falcon does not need to clean up the logs due to the small log window design. Compared to out-of-place databases[15, 35], Falcon does not need to scan the tuple heap to clean up old versions during recovery.

5.5 Limitations

Falcon's design has limitations. ① The small log window design limits the redo log size of one transaction. Fortunately, this design should work for most OLTP workloads with the relatively large size of the cache in modern CPUs and the relatively small amount of data touched by a single transaction in an OLTP workload. ② As there is no explicit way to control the order of cache eviction, the hinted flush design hints at the underlying memory controller. But there is no guarantee to flush the cache lines following the hinted order. Hardware-assisted hints may help by providing more precise control without proactive flush. Enlarging the XPBuffer size can also alleviate this problem because the memory module has more space to merge cache lines.

6 Evaluation

6.1 Setup

Configurations. We use a server with Intel Optane NVM for testing. The server has two Intel Xeon Gold 5320 processors with **eADR mode enabled**. Each socket has 6 interleaved 128GB persistent memory, 768GB in total. The cache size of the server is 1280K for L2 cache (per core) and 39MB for L3 cache (per NUMA node). The persistent memory is set in Direct Access (DAX) [3, 8] mode.

Benchmarks. Same as existing works, we use TPC-C [9] and YCSB [20]. **TPC-C** simulates the activity of a wholesale supplier. The benchmark contains 9 tables and 5 transaction types. NewOrder transactions account for 45% of all transactions, and Payment transactions account for 43%. These two short read/write transactions account for 88% of the total TPC-C workload. The remaining transactions, 4% each, consist of long read/write Deliver transactions, and two read-only transactions, OrderStatus and StockLevel. TPC-C is widely used as the OLTP benchmark. TPC-C has a configurable number of warehouses. Same as Zen [35], our benchmark contains 2048 warehouses and 100,000 items. **YCSB** is another widely used key-value store benchmark. It is also used to evaluate transnational systems. Our YCSB workload contains a single table with keys of 8 bytes and 10 columns with 100 bytes for each. The tuple size is about 1KB bytes. Before testing, we put 256 million tuples ($\sim 256GB$) during table initialization. Each transaction reads and updates all fields. YCSB consists of six core workloads. Workloads A and F are read-write workloads. Workloads B, D and E are read-heavy workloads (5% write) and workload C is a read-only workload. YSCB-A describes a balanced read-write workload (read:write = 1:1). Updates in this workload do not require the original record to be read first. In YCSB-F, half of the transactions do read-modify-write operations, so there are more conflicts than in YCSB-A. In each workload, we use both Uniform and Zipfian distributions (θ = 0.99).

6.2 Overall Performance

6.2.1 OLTP Engines to Compare. We compare Falcon with the following systems. They all use the same Tuple Heap design. Table 1 lists their features.

ZenS: Zen [35] is the state-of-the-art NVM OLTP engine. We re-implement Zen's storage engine, as ZenS, for testing. ZenS uses the out-of-place update approach, an in-DRAM index[3] and a buffer pool for tuple cache. ZenS is flexible and can be combined with different concurrency control algorithms and whether to manually flush or not.

ZenS (No Flush): Same as ZenS but all flush instructions are removed. It is used for observing the `clwb` instruction effects for persistent cache.

Outp: Outp is a pure out-of-place update engine. `clwb` instructions are used.

Inp: Inp is a pure in-place update engine. `clwb` instructions are used.

Falcon (No Flush): Falcon (No Flush) is the same as Falcon with `clwb` instructions removed.

Falcon (All Flush): Falcon (All Flush) is the same as Falcon with hot tuple tracking removed, i.e., flushing all touched tuples.

Falcon (DRAM Index): Falcon (DRAM Index) is the same as Falcon but putting indexes in DRAM not in NVM.

[3]We use DashMap [12], an implementation of a concurrent hashmap in Rust, as the DRAM index. For NBTree, we use `malloc` to allocate memory instead of mmap NVM files.

Table 1. Comparison of NVM OLTP Engines.

OLTP Engine	DRAM Data	NVM Data	Flush
Out-of-place Update			
ZenS	Index+Tuple Cache	Tuple Heap (Multiple Version)	All
ZenS (No Flush)	Index+Tuple Cache	Tuple Heap (Multiple Version)	No
Outp	/	Index+Tuple Heap (Multiple Version)	All
In-place Update			
Inp	/	Index+Tuple Heap (Single Version)+Logs (Old Versions)	All
Falcon (No Flush)	Version Heap	Index+Tuple Heap (Single Version)+Small Log Window	No
Falcon (All Flush)	Version Heap	Index+Tuple Heap (Single Version)+Small Log Window	All
Falcon	Version Heap	Index+Tuple Heap (Single Version)+Small Log Window	Selective
Falcon (DRAM Index)	Index+Version Heap	Tuple Heap (Single Version)+Small Log Window	Selective

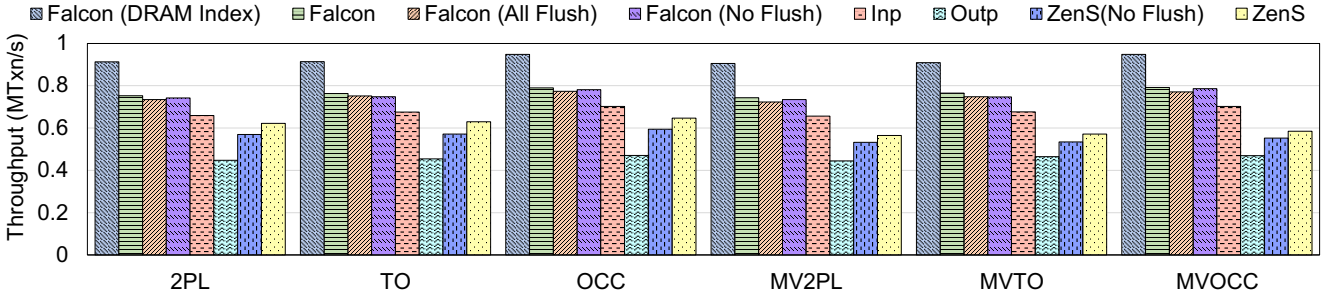

Figure 7. TPC-C throughput. (48 threads)

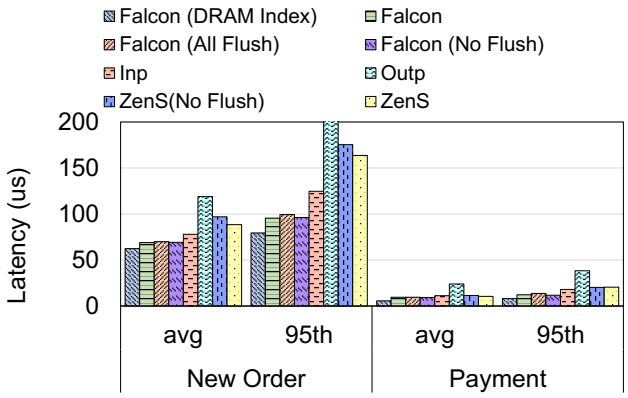

Figure 8. TPC-C Latency. (48 threads, OCC)

6.2.2 TPC-C. Figure 7 and Figure 8 show the throughput and latency of all OLTP engines with the TPC-C workload using different concurrency control algorithms.

In-place Update. For in-place update OLTP engines, Falcon reduces logging overhead with small log windows and reduces hot data writes with selective flush, improving throughput by 12.5% ~ 14.2% and decreasing latency by 13.1% ~ 18.6% over Inp. Falcon (All Flush) adds small log windows to Inp, avoiding log writes to NVM, which achieved 10.1% ~ 11.5% improvement in throughput and 11.2% ~ 17.8% decreasing in latency. Falcon and Falcon (All Flush) work similarly because only a small percentage of update operations are performed on hot tuples in TPC-C.

Out-of-place Update. For out-of-place update OLTP engines, ZenS accelerates hot data reads by caching them in DRAM. In addition, ZenS uses DRAM indexes to speed up indexing operations, improving throughput by 22.9% ~ 38.9% and decreasing latency by 25.7% ~ 56.5% over Outp.

Flush Instructions. TPC-C only modifies one of a dozen or so fields in a tuple. Hinted flush is useful for NewOrder transactions that involve the insert operations. However, a lot of time is taken by complex transactions (OrderStatus, Deliver). The improvement from hinted flush is limited. As a result, Falcon and Falcon (No Flush) work similarly.

For ZenS, throughput drops by 5.5% ~ 9.2% and latency increases by 9.7% ~ 10.5% after removing the flush instructions. Removing the flush instructions is not the optimal choice in eADR-enabled NVM.

NVM Index vs. DRAM Index. The performance of NVM index is lower than DRAM index due to the performance gap between DRAM and NVM. Replacing NVM index with DRAM index in TPC-C brings a 18.8% ~ 21.8% throughput improvement and 9.4% ~ 40.1% latency decrease.

In-place Update vs. Out-of-place Update. Requests in TPC-C typically access only a small number of columns

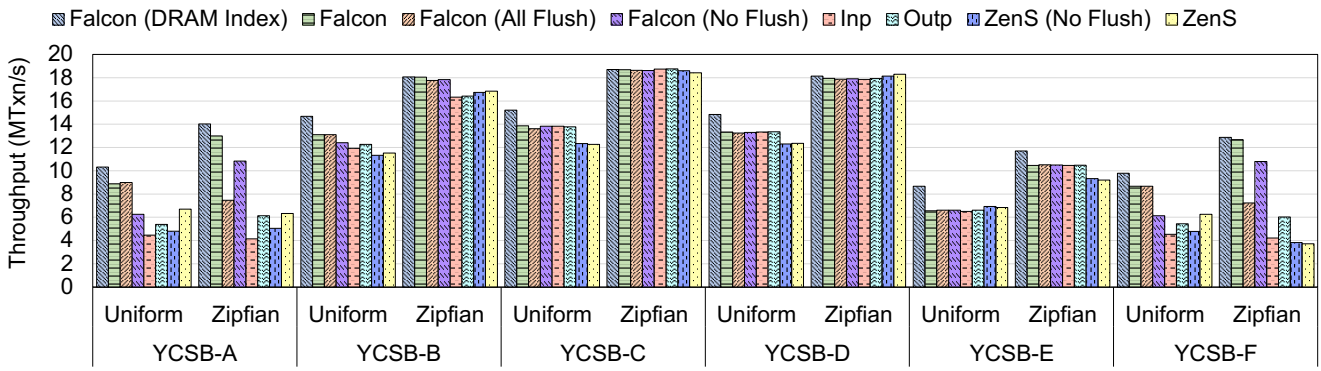

Figure 9. YCSB-A and YCSB-F throughput (48 threads, Uniform and Zipfian (θ = 0.99)).

(1 ~ 2 columns) in tuples. Out-of-place update systems need to copy the entire tuple after an update, which leads to write amplification. Therefore, the in-place update has the advantage over the out-of-place update in reducing NVM writes.

Concurrency Control Algorithms. The bottleneck of OLTP engines is NVM reads and writes. Therefore, different OLTP engines perform similarly under different concurrency control algorithms. The performance of Falcon with the multi-version algorithms is similar to that of the single-version algorithms, with a small degradation (less than 1%). This is because there are a small number of modifications, and each modification involves a small amount of data in the TPC-C workload. The throughput of ZenS in the multi-version algorithm is about 10.2% lower than in the single-version algorithm. This is due to the maintenance of extra multiple versions in the DRAM buffer pool.

6.2.3 YCSB. TPC-C modifies only one of a dozen or so fields in a tuple, which is not friendly to engines with the out-of-place update. Long read-only transactions in TPC-C weaken the effects of small log window and selective data flush design. Therefore, we chose a configuration in which all ten fields get updated in YCSB. It is more friendly to the engines with the out-of-place update because it avoids the write amplification caused by partial modifications. Figure 9 shows the results. As all of the optimizations in Falcon are for writes, we only report results for OCC and mainly focus on YCSB-A and YCSB-F. Results for other concurrency control algorithms are similar.

In-place Update. Inp has the same logging overhead as the update overhead. Falcon and Falcon (All Flush) use small log windows to avoid logging overhead, increasing throughput by 1.71× ~ 2.01× under YCSB-A and YCSB-F Uniform workloads. For YCSB-A and YCSB-F Zipfian workloads with hot tuples, Falcon also uses selective flush to avoid writing hot tuples to NVM, achieving in a throughput improvement of 3.14× over Inp and 1.75× over Falcon (All Flush).

Out-of-place Update. Under YCSB-A and YCSB-F Uniform workloads, like TPC-C, ZenS increases throughput by up to 1.24× via in-memory caching over Outp. However, under Zipfian workloads, especially in YCSB-F, the results for ZenS throughput drop up to 41.6%. It is because Zen doesn't allow a thread to modify a tuple belonging to other threads directly. A thread has to copy the tuple and invalidate the original tuple before applying the changes, which leads to high copying overhead in high contention workloads.

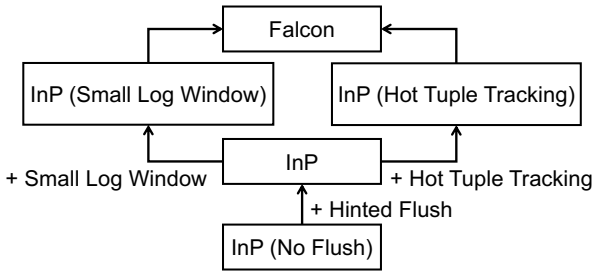

Figure 10. Engine relationships for testing individual optimization and scalability.

Flush Instructions. Under Uniform workloads, Falcon and Falcon (All Flush) actively flush writes to the NVM using `clwb`, allowing neighboring modifications to be merged, improving the throughput by 41.3% and 42.0% over Falcon (No Flush), respectively. For the out-of-place update OLTP engines, ZenS with `clwb` improves the throughput by 39.5% over ZenS (No Flush). However, under the Zipfian workloads, actively using flush instructions causes the hot tuples to be unnecessarily written to the NVM. The hot tuple tracking alleviates this problem. As a result, Falcon and Falcon (No Flush) have 74.3% and 49.4% higher throughput than Falcon (All Flush), respectively. The hot tuple tracking does not apply to out-of-place update.

In-place Update vs. Out-of-place Update. Inp runs slower than the out-of-place update engines because the out-of-place update engines use the log-free design, but Inp needs tuple modifications in addition to redo logs, which incurs more NVM writes. Falcon and Falcon (No Flush) outperform

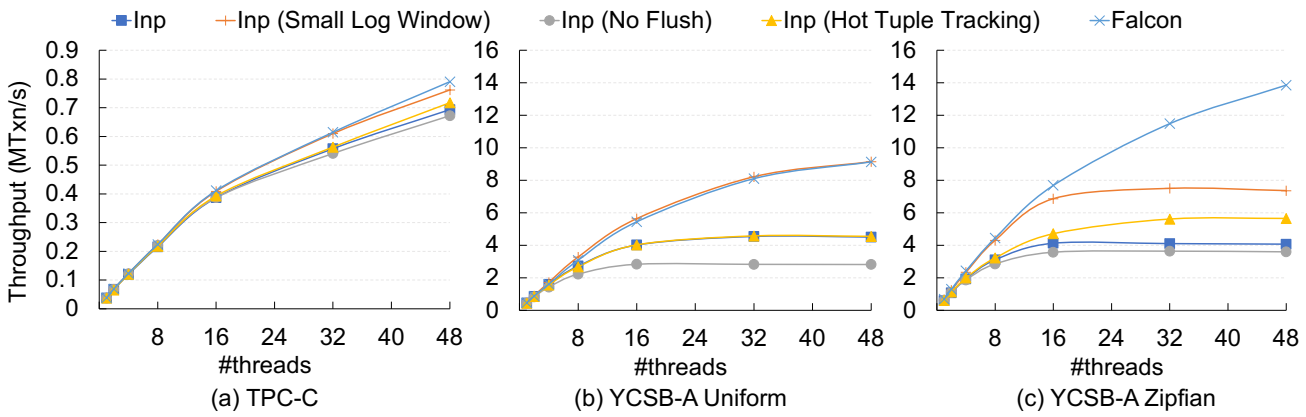

Figure 11. TPC-C and YCSB-A Scalability.

out-of-place update engines by avoiding the NVM writes of logs by using the small log window and reducing the NVM writes of data by using the hot tuple tracking.

6.3 Individual Optimization and Scalability

This section investigates how the different optimizations in Falcon can influence performance and scalability.

The engines for this test start with ① Inp (No Flush), which is the pure in-place update implementation but with all the `clwb` instructions removed. This engine is the baseline because our Falcon's basic design uses the in-place update approach. Though all `clwb` instructions are removed, Inp (No Flush) can still work correctly for persistent cache. ② Inp adds `clwb` instructions to Inp (No Flush), which can reduce the write amplification. Starting from Inp, we have two independent optimizations leading to different engines. ③ Inp (Small Log Window) is the engine of Inp with the small log window optimization to avoid log flushing. ④ Inp (Hot Tuple Tracking) is the engine of Inp with the hot tuple tracking optimization to reduce the amount of tuple data written to NVM. Finally, with all optimizations turned on, we have ⑤ Falcon. Their relationships are in Figure 10.

We evaluate the effects of these optimizations on TPC-C, YCSB-A Uniform, and YCSB-A Zipfian workloads. The results for YCSB-F are similar.

Figure 11 shows the experiment results. As expected, with all optimizations turned on, Falcon performs the best for all workloads in both overall performance and scalability.

TPC-C workload (Figure 11(a)). Inp works better than the baseline (Inp (No Flush)) because sequential logs flushing reduces write amplification. This is because each transaction has to access one hot tuple in Warehouse Table. Inp (Hot Tuple Tracking) reduces the flush for this hot tuple, so it's better than Inp. Inp (Small Log Window) reduces the NVM write for the logs and performs better than Inp and Inp (Hot Tuple Tracking), which means the improvement of Small Log Window is more significant in TPC-C workloads.

YCSB A Uniform workload (Figure 11(b)). There are no hot tuples. The hot tuple tracking has little impact. Therefore, Inp and Inp (Hot Tuple Tracking) perform similarly, while Inp (Small Log Window) and Falcon also perform similarly.

YCSB A Zipfian workload (Figure 11(c)). With 48 threads, Falcon achieves 2.44× speed up compared to Inp (Hot Tuple Tracking), due to the effect of the small log window optimization. The small log window can cut half of the writes. However, the performance gain here is over 2×. This is due to the shorter duration for transactions to hold tuples, which effectively reduces conflicts.

6.4 Impact of Different Tuple Sizes

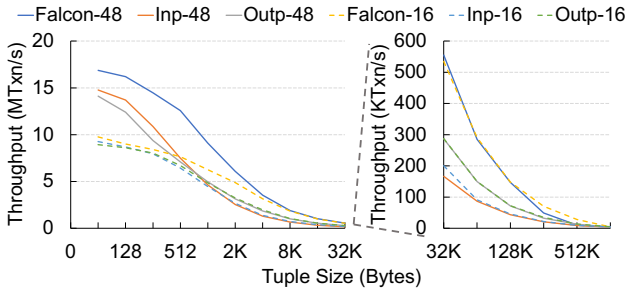

Figure 12. YCSB-A throughput with different tuple size. (16 threads and 48 threads, Uniform)

The small log window design limits the redo log size of one transaction. We show Falcon's performance with larger log sizes by increasing the tuple size in the YCSB workload. Figure 12 shows the performance of Falcon, Inp and Outp under YCSB-A Uniform workload with different tuple sizes. The small log window effect diminishes when the tuple size reaches 512KB. When more logs are evicted, the performance gets closer to Inp. With larger tuple sizes, the out-of-place update design is better.

Larger tuples result in an increase in NVM writes during transactions. A higher number of concurrent transactions

increases the likelihood of encountering cache thrashing in the underlying cache layer within the NVM module, such as Optane XPBuffer. Consequently, when dealing with larger tuple sizes, 16 threads deliver better performance compared to 48 threads. Workload throttling can help here [60].

6.5 Recovery

We use about 256*GB* of YCSB data to test the recovery performance. Similar to previous studies [19, 22], we run Falcon and inject crashes using `SIGKILL`. The test runs 100 times, and we report the results for the longest recovery instance.

Falcon takes 3.276*ms* in total for recovery. During recovery, Falcon takes 1.272*ms* to initialize in-DRAM data structures (e.g., catalog and table schemas) and 1.057*ms* to recover the NVM index by calling the Dash's `Recovery()` function. Falcon then replays the log and clears the lock bits in a single thread, which takes about 0.97*ms*. Other recovery work takes negligible time.

On the contrary, ZenS takes 9.4*s* for recovery. Its recovery time is proportional to the data size in the NVM tuple heap. Most of the time is spent scanning tuples in NVM to rebuild the in-DRAM index.

7 Related Work

Persistent Cache. Intel has discontinued the Optane business[11], but other work besides eADR explores putting the CPU cache into persistent domains. CXL 3.0[1] introduces Global Persistent Flush (GPF)[2] to provide similar functionality to the eADR, i.e., flushing the cache to persistent storage after a system crash. BBB[14] (Battery-Backed Buffer) uses batteries to make the CPU cache persistent, achieving almost identical results to eADR in terms of performance while using two orders of magnitude less power and time. These can work as a replacement for eADR.

Reducing logging overhead. To reduce the write amplification of logs, many existing works modify hardware to control the flushing of logs. Proteus [44] attempts to remove the logs of completed transactions in the memory controller so that they are not written to the NVM. Silo [58] provides on-chip logs that do not rely on eADR. Silo stores logs in on-chip space and writes the on-chip logs back to the NVM only in the case of a system crash. LOAD [61] and Hercules [54] achieve the same goal by leveraging hardware logging and eADR. All these hardware approaches try to control the log flushing precisely. Our software-based approach works very well under persistent cache without precise control.

OLTP Engines for NVM: Previous works also optimize the NVM OLTP engine by reducing NVM writes. WBL [15] uses the byte-addressable feature of NVM to allow transactions to create a new version of tuples before writing logs. This method removes the data from the log, which reduces the size of the logs. Zen[35] uses a Metadata Enhanced Tuple Cache (Met-Cache) in DRAM to move the per-tuple metadata used by concurrency control algorithms from NVM to DRAM, reducing the overhead of writing metadata in NVM. NVCaracal [48] uses epoch-based transaction processing, which writes at most one update per tuple per epoch to NVM. Unlike other OLTP engines optimized for volatile cache, Falcon's optimization is specially designed to target persistent cache.

Persistent Transactional Memory: Persistent transactional memory (PTM) is another type of transaction processing system in NVM. PTM supports all ACID attributes. The underlying implementation is similar to the NVM OLTP storage engine. PTM operates on objects instead of structured tuples. PMDK [5], NV-heap [19], Atlas [17], JUSTDO [28], iDO [37] use the in-place update and undo logs to back up all modified objects during transaction execution. Pisces [25], Romulus [21], and DudeTM [36] use the in-place update and redo logs. All of them incur extra writing in NVM. TimeStone [31] and ArchTM [49] use out-of-place update storage with an in-DRAM index to avoid frequent index updates in NVM. In general, the storage of PTM is similar to that of an OLTP engine. Similarly, existing PTMs are designed and optimized for volatile cache. The optimizations in Falcon can help improve PTMs for persistent cache.

8 Conclusion

Persistent cache provides a new opportunity to ease the programming for NVM and improve the overall performance. However, as shown in this paper, the improved hardware architecture does not guarantee higher performance. Falcon is a new OLTP engine specially designed for NVM with persistent cache. Falcon tries to reduce the amount of data that is written to the underlying storage media by two designs. For logs, Falcon uses the small log window to reuse the log space for multiple transactions. Logs are never manually flushed to NVM. For data, Falcon uses the selective flush to reduce the write amplification when tuples are flushed to NVM (using the hinted flush) and avoid flushing hot tuples (using the hot tuple tracking to identify hot tuples).

Acknowledgments

We thank the anonymous reviewers and our shepherd, Ryan Stutsman, for their valuable comments and helpful suggestions. The authors from Tsinghua University are all in the Department of Computer Science and Technology, Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University, China. This Work is supported by National Key Research & Development Program of China (2022YFB2404202), Natural Science Foundation of China (62141216, 61877035) and Tsinghua University Initiative Scientific Research Program, Young Elite Scientists Sponsorship Program by CAST (2022QNRC001), and Beijing HaiZhi XingTu Technology Co., Ltd. Correspondence to: Kang Chen(chenkang@tsinghua.edu.cn), Mingxing Zhang (zhang_mingxing@mail.tsinghua.edu.cn).

References

[1] CXL SPECIFICATION | Compute Express Link. https://www.computeexpresslink.org.

[2] CXL™ 3.0 Specification. https://www.computeexpresslink.org/download-the-specification.

[3] Indradead. 2021. Direct Access for files. https://www.infradead.org/~mchehab/kernel_docs/filesystems/dax.html.

[4] Intel Corporation. 2021. eADR: New Opportunities for Persistent Memory Applications. https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html.

[5] Intel. Persistent Memory Development Kit. . https://pmem.io/.

[6] Mysql. https://www.mysql.com/.

[7] Postgresql. https://www.postgresql.org/.

[8] The Linux kernel archives. 2021. DAX-Direct access for files. https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[9] TPC Benchmark C. https://www.tpc.org/tpcc/.

[10] Build Persistent Memory Applications with Reliability Availability and Serviceability., 2020. https://www.intel.cn/content/www/cn/zh/developer/articles/technical/build-pmem-apps-with-ras.html?wapkw=Asynchronous%20DRAM%20Refresh%20(ADR)%20domain.

[11] Intel Reports Second-Quarter 2022 Financial Results, 2022. https://www.intc.com/news-events/press-releases/detail/1563/intel-reports-second-quarter-2022-financial-results.

[12] Blazing fast concurrent HashMap for Rust., 2023. https://github.com/xacrimon/dashmap.

[13] Intel® 64 and ia-32 architectures optimization reference manual, 2023. https://www.intel.com/content/www/us/en/content-details/671488/intel-64-and-ia-32-architectures-optimization-reference-manual.html?wapkw=IntelÂő%2064%20and%20IA-32%20Architectures%20Optimization%20Reference%20Manual.

[14] Mohammad Alshboul, Prakash Ramrakhyani, William Wang, James Tuck, and Yan Solihin. Bbb: Simplifying persistent programming using battery-backed buffers. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 111–124. IEEE, 2021.

[15] Joy Arulraj, Matthew Perron, and Andrew Pavlo. Write-behind logging. *Proceedings of the VLDB Endowment*, 10(4):337–348, 2016.

[16] Philip A Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, 1981.

[17] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. *ACM SIGPLAN Notices*, 49(10):433–452, 2014.

[18] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1077–1091, 2020.

[19] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. *ACM SIGARCH Computer Architecture News*, 39(1):105–118, 2011.

[20] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[21] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 271–282, 2018.

[22] Laxman Dhulipala, Guy E Blelloch, Yan Gu, and Yihan Sun. Pactrees: Supporting parallel and compressed purely-functional collections. *arXiv preprint arXiv:2204.06077*, 2022.

[23] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. Sap hana database: data management for modern business applications. *ACM Sigmod Record*, 40(4):45–51, 2012.

[24] Caixin Gong, Chengjin Tian, Zhengheng Wang, Sheng Wang, Xiyu Wang, Qiulei Fu, Wu Qin, Long Qian, Rui Chen, Jiang Qi, et al. Tairpmem: a fully durable non-volatile memory database. *Proceedings of the VLDB Endowment*, 15(12):3346–3358, 2022.

[25] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. Pisces: A scalable and efficient persistent transactional memory. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 913–928, 2019.

[26] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. Understanding the idiosyncrasies of real persistent memory. *Proceedings of the VLDB Endowment*, 14(4):626–639, 2020.

[27] Daokun Hu, Zhiwen Chen, Jianbing Wu, Jianhua Sun, and Hao Chen. Persistent memory hash indexes: an experimental evaluation. *Proceedings of the VLDB Endowment*, 14(5):785–798, 2021.

[28] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via justdo logging. *ACM SIGARCH Computer Architecture News*, 44(2):427–442, 2016.

[29] Sanidhya Kashyap, Changwoo Min, Kangnyeon Kim, and Taesoo Kim. A scalable ordering primitive for multicore machines. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.

[30] Hideaki Kimura. Foedus: Oltp engine for a thousand cores and nvram. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 691–706, 2015.

[31] R Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. Durable transactional memory can scale with timestone. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 335–349, 2020.

[32] Hsiang-Tsung Kung and John T Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[33] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. Evaluating persistent memory range indexes. *Proceedings of the VLDB Endowment*, 13(4):574–587, 2019.

[34] Hyeontaek Lim, Michael Kaminsky, and David G Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 21–35, 2017.

[35] Gang Liu, Leying Chen, and Shimin Chen. Zen: a high-throughput log-free oltp engine for non-volatile main memory. *Proceedings of the VLDB Endowment*, 14(5):835–848, 2021.

[36] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. Dudetm: Building durable transactions with decoupling for persistent memory. *ACM SIGPLAN Notices*, 52(4):329–343, 2017.

[37] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L Scott, Sam H Noh, and Changhee Jung. ido: Compiler-directed failure atomicity for nonvolatile memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 258–270. IEEE, 2018.

[38] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: scalable hashing on persistent memory. *Proceedings of the VLDB Endowment*, 13(8):1147–1161, 2020.

[39] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. Roart: Range-query optimized persistent art. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 1–16, 2021.

[40] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 677–689, 2015.

[41] Gihwan Oh, Sangchul Kim, Sang-Won Lee, and Bongki Moon. Sqlite optimization with phase change memory for mobile applications. *Proceedings of the VLDB Endowment*, 8(12):1454–1465, 2015.

[42] Jong-Hyeok Park, Gihwan Oh, and Sang-Won Lee. Sql statement logging for making sqlite truly lite. *Proceedings of the VLDB Endowment*, 11(4):513–525, 2017.

[43] Next-Generation Memory Solutions at Flash Memory Summit, 2022. https://news.samsung.com/global/samsung-electronics-unveils-far-reaching-next-generation-memory-solutions-at-flash-memory-summit-2022.

[44] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. Proteus: A flexible and fast software supported hardware logging approach for nvm. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 178–190, 2017.

[45] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.

[46] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. Managing non-volatile memory in database systems. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1541–1555, 2018.

[47] Alexander Van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. Persistent memory i/o primitives. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*, pages 1–7, 2019.

[48] Yu Chen Wang. *NVCaracal: A Deterministic Database for Non-Volatile Main Memory*. PhD thesis, University of Toronto (Canada), 2022.

[49] Kai Wu, Jie Ren, Ivy Peng, and Dong Li. Archtm:architecture-aware, high performance transaction for persistent memory. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 141–153, 2021.

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

[51] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. Hikv: A hybrid index lsm store for dram-nvm memory systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 349–362, 2017.

[52] Baoyue Yan, Xuntao Cheng, Bo Jiang, Shibin Chen, Canfang Shang, Jianying Wang, Gui Huang, Xinjun Yang, Wei Cao, and Feifei Li. Revisiting the design of lsm-tree based oltp storage engine with persistent memory. *Proceedings of the VLDB Endowment*, 14(10):1872–1885, 2021.

[53] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, 2020.

[54] Chongnan Ye, Meng Chen, Qisheng Jiang, and Chundong Wang. Enabling atomic durability for persistent memory with transiently persistent cpu cache. *arXiv preprint arXiv:2210.17377*, 2022.

[55] Xiangyao Yu. *An evaluation of concurrency control with one thousand cores*. PhD thesis, Massachusetts Institute of Technology, 2015.

[56] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1629–1642, 2016.

[57] Bowen Zhang, Shengan Zheng, Zhenlin Qi, and Linpeng Huang. Nbtree: a lock-free pm-friendly persistent b+-tree for eadr-enabled pm systems. *Proceedings of the VLDB Endowment*, 15(6):1187–1200, 2022.

[58] Ming Zhang and Yu Hua. Silo: Speculative hardware logging for atomic durability in persistent memory. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 651–663. IEEE, 2023.

[59] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. Chameleondb: a key-value store for optane persistent memory. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 194–209, 2021.

[60] Diyu Zhou, Yuchen Qian, Vishal Gupta, Zhifei Yang, Changwoo Min, and Sanidhya Kashyap. Odinfs: Scaling pm performance with opportunistic delegation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 179–193, 2022.

[61] Taiyu Zhou, Yajuan Du, Fan Yang, Xiaojian Liao, and Youyou Lu. Efficient atomic durability on eadr-enabled persistent memory. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 124–134, 2022.