Using Restricted Transactional Memory to Build a Scalable In-Memory Database

Zhaoguo Wang†, Hao Qian‡, Jinyang Li§, Haibo Chen‡

† School of Computer Science, Fudan University
‡ Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
§ Department of Computer Science, New York University

Abstract

The recent availability of Intel Haswell processors marks the transition of hardware transactional memory from research toys to mainstream reality. DBX is an in-memory database that uses Intel's restricted transactional memory (RTM) to achieve high performance and good scalability across multi-core machines. The main limitation (and also key to practicality) of RTM is its constrained working set size: an RTM region that reads or writes too much data will always be aborted. The design of DBX addresses this challenge in several ways. First, DBX builds a database transaction layer on top of an underlying shared-memory store. The two layers use separate RTM regions to synchronize shared memory access. Second, DBX uses optimistic concurrency control to separate transaction execution from its commit. Only the commit stage uses RTM for synchronization. As a result, the working set of the RTMs used scales with the meta-data of reads and writes in a database transaction as opposed to the amount of data read/written. Our evaluation using TPC-C workload mix shows that DBX achieves 506,817 transactions per second on a 4-core machine.

1. Introduction

Writing high performance, correctly synchronized software is a challenge. To date, programmers had several unpleasant approaches to choose from. Coarse-grained locks provide a straightforward programming model (mutual exclusion) but can lead to poor performance under load due to contention. To reduce contention, programmers commonly resort to a combination of fine-grained locks and atomic operations. However, the correctness of the resulting code is complex to reason about and relies on the processor's (increasingly complex) memory model.

Recently, Intel has shipped its 4th-generation Haswell processor with support for Hardware Transactional Memory [16]. This opens up a third possibility to scaling multi-core software. Instead of relying on fine-grained locking and atomic operations, one can synchronize using hardware transactions, which offer a programming model that is arguably even more straightforward than mutual exclusion. The promise is that the resulting implementation is much simpler and easier-to-understand while still retaining the performance benefits of fine-grained locking.

Does hardware transactional memory actually deliver its promise in practice? To answer this question, this paper implements a multicore in-memory database for online transaction processing (OLTP) workloads. We use an in-memory database as a case study for two reasons. First, high performance in-memory database is an important application, as indicated by much recent research and commercial activities on in-memory databases [17, 39, 42]. Second, an OLTP database is a sophisticated piece of software, consisting of concurrent data structures as well as complex transaction commit logic that require careful synchronization.

Our system, called DBX, implements a transactional key-value store using Intel's Restricted Transactional Memory (RTM). Using RTM, DBX has achieved similar performance and scalability with state-of-the-art in-memory database using fine-grained locking [41], yet is easier to implement and reason about the correctness. However, while hardware transactional memory theoretically enables effortless coarse-grained synchronization, in practice, the limitations of RTM force us to craft transaction regions carefully. The major technical challenges that shape the design of DBX are 1) ensure that an RTM region's working set does not overflow the allowed hardware limit and 2) avoid spurious aborts due to false sharing of cache lines and system events.

DBX consists of two modular parts: a key-value store with either ordered or unordered access and a transactional

EuroSys 2014, April 13 - 16 2014, Amsterdam, Netherlands.

http://dx.doi.org/10.1145/2592798.2592815

layer for grouping together multiple key-value accesses with serializability property. The key-value subsystem provides a B^+-tree (for ordered tables) or hash table (for unordered tables) implementation. The data structure is shared among many threads and DBX protects each operation such as get and put using a single RTM region. The transaction layer executes each transaction in a separate thread using the three-phase optimistic concurrency control (OCC) algorithm [21]. In the read phase, the transaction reads from the underlying key-value store and buffers its writes. In the validation phase, the transaction is aborted if the records it has previously read have been modified by concurrent transactions. In the write phase, a committed transaction applies its writes to the key-value store. DBX uses a single RTM region to protect the last two phases to synchronize among concurrently executing threads, making it easier to reason about correctness.

DBX is also built with a new read-only transaction protocol that requires no validation phase but still allowing a transaction to read the most recent database snapshot. The key idea is letting read-only transactions advance database snapshot counter, while read-write transactions create most recent snapshot of records for inflight readers only on demand.

We have implemented a prototype and evaluated it using the TPC-C mix benchmark on a 4-core (8-hyperthread) [1] Haswell machine. Our evaluations show that DBX performs similarly to a highly-tuned in-memory database based on fine-grained locking [41] and has slightly better scalability. In particular, under 8 threads, DBX achieves 506,817 transactions per second.

Due to hardware limits, our evaluation is restricted to a small number of threads (i.e., 8), and thus our conclusion on scalability is still speculative. Future hardware enhancement like more RTM cores and larger working set size may affect our current conclusion. Nevertheless, DBX shows that RTM is a powerful hardware primitive that can drastically simplify the design and implementation of high performance multicore software.

2. RTM Performance in Practice

This section gives an overview of RTM and discusses how its characteristics and limitations can impact the design of an in-memory database.

2.1 RTM Background

With Intel's RTM, programmers use *xbegin* and *xend* to mark a code region to be executed transactionally. One can also use *xabort* to explicitly abort in the middle of a transaction's execution. If a transaction commits successfully, all memory writes will appear to have happened atomically. RTM provides strong atomicity [2]: if a transaction conflicts with concurrent memory operations done by other transactional or non-transactional code, the processor will abort the transaction by discarding all its writes and roll back the system state to the beginning of the execution. To avoid confusion with database transactions, we use the term RTM transaction or RTM region to refer to the hardware transaction.

RTM limitations: As a practical hardware transaction mechanism, RTM comes with several limitations. First, the read/write set of an RTM transaction must be limited in size. This is because the underlying hardware uses the CPU cache to track reads/writes. If an RTM transaction reads/writes more memory than the hardware limits, it will be aborted due to overflow of internal processor buffers (like write buffer). Second, the hardware tracks reads and writes at the granularity of a cache line. Consequently, an RTM transaction may be unnecessarily aborted due to false sharing of a cache line and cache set conflict misses. Third, some instructions and system events such as page faults may abort an RTM transaction as well.

Fallback handler: As a best-effort mechanism, an RTM transaction does not have guaranteed progress even in the absence of conflicts. As a result, programmers must write a fallback routine which executes after an RTM transaction aborts for a threshold number of times. To preserve code simplicity, a fallback routine usually acquires a coarse-grained lock. Thus, it is important that RTMs do not abort too frequently; otherwise, the performance will degenerate to that of a coarse lock-based implementation.

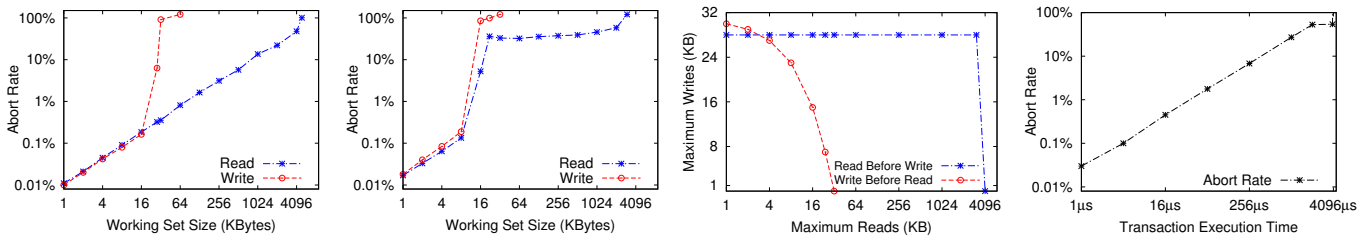
To incorporate the fallback routine, an RTM transaction first checks if the fallback lock has already been acquired upon entering its RTM region. If so, the RTM transaction is aborted explicitly (using *xabort*); otherwise, it proceeds as usual with the lock in its readset. Consequently, the transaction will only commit if there is no concurrent execution of the fallback routine that has acquired the lock. A fallback handler also needs to provide mutual exclusion among multiple threads such that only one thread can successfully execute inside the handler.

2.2 RTM Performance Quirks

We present micro-benchmark results to examine RTM's performance in practice. Our experiments ran on a 4-core (8-hyperthread) Intel Haswell machine.

RTM can perform more reads than writes. What is the working set size limit in practice? To find the answer, we ran a micro-benchmark where all threads read or write varying amounts of memory sequentially in an RTM region. All threads touch different cache-aligned memory regions so that their RTM regions have no conflicts. Figure 1 (a) shows the RTM abort rate as a function of working set size when running 4 threads each pinned to a physical core. The abort rate is measured by dividing the number of aborted transactions by the total number of transactions (both aborted and committed). The occasional aborts due to system events or internal buffer overflow cause the abort rate to increase linearly with the the execution time of an RTM transaction, as

[1] This is currently the maximum number of cores/threads for a machine with RTM.



(a) Abort Rate With Working Set (No HT) *(b) Abort Rate With Working Set (With HT)* *(c) Feasible Write Set and Read Set Combination* *(d) Abort Rate With Execution Time*

Figure 1: *RTM Features Study*

reflected by the working set size. As the working set size passes a certain limit, abort rates shoot up drastically. Interestingly, the size limits differ for reads and writes: an RTM transaction can read at most 4MB with an abort rate of 47%, but can only write at most 31KB memory with an abort rate of 90%. This is consistent with our understanding that RTM tracks writes using L1 cache (32KB on our CPU) and uses an implementation-specific structure to track reads.

Hyperthreading reduces maximum working set size. Figure 1 (b) corresponds to the same experiments as Figure 1 (a) but with 8 threads, each running on a hyperthreaded core. The maximal read set is about 2MB with an abort rate of 58% and the maximal write set is about 22KB with an abort rate of 99%. This is because two hyperthreaded cores share critical processor resources such as caches and even compete with each other for the resources.

RTM prefers reads before writes: The order of reads and writes done in an RTM region affects the working set size. Figure 1 (c) shows the feasible write set and read set combination in two types of workloads: read-before-write performs reads before any writes, and write-before-read performs writes before any reads. The points on the line mark the maximum reads and writes for an RTM transaction. The region at the bottom left of each line marks all feasible read/write sizes for each workload. As we can see, performing reads before writes results in a larger feasibility region: 3MB memory read with 28KB memory write. This is because when performing writes before reads, a later read might evict some write set entry from the L1 cache, resulting in an RTM abort. Such a scenario does not happen when evicting a read set entry from the cache.

RTM should execute promptly: Besides working set size limits, an RTM transaction can also be aborted due to system events such as a timer interrupt. Consequently, the duration of an RTM execution also impacts its abort rate. We ran experiments in which each RTM transaction spins inside a while loop for a number of CPU cycles. Figure 1 (d) shows that the abort rate increases with the execution time due to increased likelihood of the timer firing during RTM execution. The timer duration of our test machine is 4 ms, which is the maximum time an RTM can execute without being aborted with certainty.

3. DBX Overview

The setup. DBX is an in-memory transactional storage engine. DBX is designed for OLTP applications with frequent database updates. On a machine with n cores, DBX employs n worker threads each of which runs on a separate physical or hyperthreaded core. Each thread executes and commits a single database transaction at a time, synchronizing with other threads using RTM regions. Committed transactions are asynchronously logged to local disks/SSDs in the background, as done in other systems like VoltDB [19, 42], Silo [41] and MassTree [28].

Our Approach. The biggest challenge in using RTM is to constrain the size of an RTM's working set to not overflow the hardware limits. Therefore, we cannot use the naive approach of enclosing each database transaction within a single RTM region (i.e., having a single critical section for the lifetime of a transaction, including both execution and commit). Doing so makes the RTM's working set size impractically large: it not only contains all the data read and written in a database transaction but also the additional memory accesses required to traverse the in-memory data structure such as a B-tree.

To limit RTM working set size, we build DBX out of two independent components: a shared in-memory store and a transaction layer. The shared in-memory store exposes a key/value abstraction with either ordered access (using a B^+-tree) or unordered access (using a hash table). The transaction layer builds on top of the key-value store and implements typed tables with named records that can be accessed from within a transaction. For fast access of data records, this layer keeps the memory references of records in the read and write set. By decoupling the two layers, each layer can use smaller RTM regions within itself to ensure correct synchronization among threads. Furthermore, the RTM regions used by one layer never conflict with those used in the other.

To further reduce the RTM region size used in the transaction layer, we separate the execution of a transaction from its commit, using a variant of optimistic concurrency con-

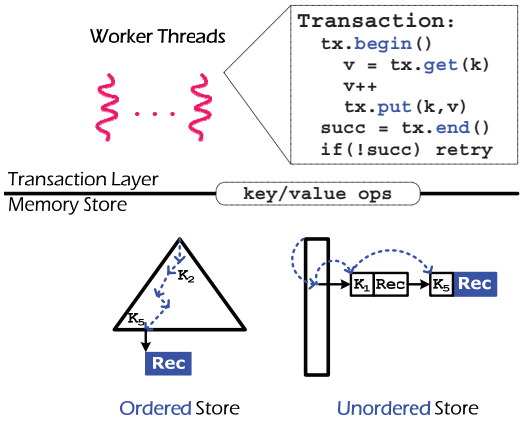


Figure 2: The architecture of DBX

trol (OCC). With OCC, we can employ separate, smaller RTM regions to handle the execution and commit phase of a database transaction separately. Different RTM regions used in different phases collectively guarantee the overall serializability of concurrent database transactions.

4. Shared Memory Store

The in-memory key-value store of DBX is a shared data structure among all worker threads. To optimize for different access patterns [1, 25, 27], DBX provides both an ordered store and an unordered store. We first explain how these data structures are implemented using RTM regions and then discuss some custom features for supporting the upper transactional layer more efficiently.

Ordered store. An ordered key/value store supports ordinary lookups as well as iterator-based range scans. We implement such a store using a B^+-tree. Programming a high performance concurrent B^+-tree has been notoriously difficult [23]. Under RTM, such a task is almost as straightforward as writing a single-threaded tree: we simply enclose each B^+-tree operation with a coarse-grained RTM region. We have also evaluated a skiplist-based implementation and found that B^+-tree achieves much higher performance due to better cache behavior.

Several efforts in reducing RTM aborts are important for the overall performance. First, to avoid false conflict due to different accesses to the same cache line, we make each tree node cache-line aligned. Specifically, our current implementation supports 8-byte fixed length keys and uses a node size of 256 bytes to pack 16 key-value pairs in each interior node, achieving a fan-out of 15. Another important detail is to ensure that the 8-byte pointer to the root of the tree occupies an exclusive cache line (64-byte). Second, we adjust the threshold for initiating lock-based fallbacks. In particular, we use a smaller threshold (i.e., retrying the RTM region for fewer times) if the height of the tree is big, indicating more wasted CPU cycles for each aborted RTM. Third, like other systems [28], we do not rebalance the B^+-tree upon deletion, which may result in a larger RTM region. There has been both theoretical and empirical work showing deletion without rebalancing works well in practice [34].

Unordered store. A hash table is better than a B-tree for performing random lookups. To implement a concurrent hash table, we simply use an RTM region to protect the entirety of each lookup, insert or delete operation. Collisions in hash lookups are resolved using simple chaining. We use a global lock to perform hash table resizing, which occurs rarely. The RTM regions of normal hash table operations always read the status of the global lock and abort if the lock is held. To avoid false conflicts, we make each hash table entry cache aligned.

Custom features for the transaction layer. The key-value stores operate independently of the transaction layer, however, two aspects of their designs are influenced by the need to support the transaction layer efficiently.

First, apart from the basic key/value interface such as `get(k)`, `insert(k,v)` and `delete(k)`, DBX supports a new operation called `get-with-insert(k,v)`. This operation inserts/updates the key k with the new value and returns the old value if k already exists. The `get-with-insert` operation is essential in helping the transaction layer detect conflicts between a read for non-existing records and a concurrent put with the same key.

Second, since the transaction layer looks up pointers instead of the actual content of the table record in the tree or hash table (see Figure 2), the "value" stored in the key-value store is small (8-bytes). Thus, when we make the hash table entry cache-line aligned (64-bytes), the memory utilization is low. To alleviate this problem, we provide a modified `put` interface which DBX's transaction layer uses to instruct the hash table to allocate a memory chunk large enough to store a table record. The hash table allocates the chunk in the same cache line as the key's hash entry if it fits. By contrast, when using a B-tree based store, the transaction layer performs record allocation and simply passes pointers to the key-value store. This is because each leaf node of B-tree is already big enough to occupy multiple cache lines.

5. Transaction Layer

The transaction layer of DBX runs atop the shared in-memory store to implement serializable transactions. The key challenge is to restrict the working sets of RTM regions to below the hardware limit while still maintaining code simplicity. DBX uses a variant of optimistic concurrency control protocol (OCC) [21]. The advantage of OCC is that it cleanly separates the execution of a transaction from its commit. Since the underlying shared memory key-value store correctly synchronizes concurrent accesses, only the commit phase requires synchronization among concurrent threads, resulting in a relatively small RTM working set containing only the metadata of all the read and write accesses done by a transaction.

Below, we explain the transaction layer by discussing the metadata layout, the commit protocol, and other key components like range query, read-only transactions, garbage collections and durability.

5.1 Metadata Layout

The transaction layer implements typed database tables and stores the pointers to table records in the underlying key-value store. To facilitate OCC, DBX stores the following metadata with each record.

- *Sequence Number* tracks the number of updates on the record. It is used to detect read-write conflict during commit.
- *Local Snapshot Number* identifies which snapshot this record belongs to. It is used to support read-only transactions (§5.4).
- *Previous Version Pointer* points to the old versions of the record in the past snapshots. This allows DBX to find records from old snapshots quickly when serving read-only transactions.
- *Record Data Pointer* points to the actual data of the record. We use a separate memory chunk to store the actual data to reduce the working set of RTM. This comes at the cost of an additional memory fetch during record read.

5.2 Basic Transaction Execution and Commit

We describe the design for an update transaction that only reads or writes existing records. How DBX handles insertion/deletion and read-only transactions will be discussed in later sections (§5.3 and §5.4).

Under OCC, a transaction proceeds through three phases: read, validate and write. The read phase can be viewed as the execution phase of a transaction while the latter two phases constitute the commit of the transaction.

Transaction execution. In the read phase of OCC, DBX executes the user's transaction code which makes a series of read and write accesses. To read a record, DBX performs a `get` in the underlying key-value store and obtains the pointer to the record metadata. DBX stores the pointer as well as the sequence number of the record in the transaction's read-set. To modify a record, DBX similarly performs a `get` to obtain the pointer to the record and stores the pointer as well as the actual written value in the transaction's write-set. In other words, writes are simply buffered and not visible to other threads during a transaction's execution. When performing a read, DBX checks in the transaction's write-set first so that a transaction always sees the value that it has previously written.

With one exception, the read phase requires no additional synchronization among worker threads beyond what is already provided by the shared key-value store. The one place where synchronization is needed is to guarantee the consis-

Algorithm 1: Commit()

```
   input: TxContext pointer tx
 1 //CommitLock is grabbed for RTM fallback
   RTMBegin(CommitLock);
 2     //validate phase
 3     for ⟨ p, seqno⟩ ∈ tx→readSet do
 4         //p is the saved pointer to the record meta-data
 5         if p→seqno ≠ seqno then
 6             abort transaction;
 7         end
 8     end
 9     for ⟨ p, key, pValue⟩ ∈ tx→writeSet do
10         //p is the saved pointer to the record meta-data
11         if p→status = REMOVED then
12             abort transaction;
13         end
14     end
15     //write phase
16     for ⟨ p, key, pValue ⟩ ∈ tx→writeSet do
17         //line 18–22 is for supporting read-only tx
18         if p→lsn ≠ globalSnapshot then
19             ov := new Version(p→data_pointer,
               p→lsn);
20             p→oldVersions := ov;
21             p→lsn := globalSnapshot;
22         end
23         //pValue points to the new record value
           p→data_pointer := pValue;
24         p→seqno++;
25     end
26 RTMEnd(CommitLock);
```

Algorithm 2: Read()

```
   input: TxContext pointer tx, key
 1 if key ∈ tx→writeSet then
 2     return corresponding value;
 3 end
 4 //look up pointer to the record to be read
 5 p := kvStore→get(key);
 6 //CommitLock is grabbed for RTM fallback
   RTMBegin(CommitLock);
 7     seqno := p→seqno;
 8     value := *(p→ data_pointer);
 9 RTMEnd(CommitLock);
10 add ⟨p,seqno⟩ to tx→readSet;
11 return value
```

tency of sequence number and the actual record data read. DBX uses an RTM region to group together the memory reads of the record's sequence number and its corresponding data, as shown in algorithm 2. Without the protection of an RTM, the sequence number in the read-set might not correspond to the actual record data read.

Transaction commit. Committing a transaction proceeds through two OCC phases (algorithm 1). In the validation phase, DBX first iterates through the read-set and checks if any record has been modified since last read by comparing the current sequence number with the sequence number remembered in the read-set. Second, DBX iterates through the write-set to check if any record has been removed by checking its deletion flag (i.e., *REMOVED*). If the validation fails, DBX aborts the transaction.

A successfully validated transaction proceeds to the write phase to make its writes visible globally. Since the write-set of a transaction keeps track of the pointer to the record's metadata, DBX updates the record's metadata by swapping pointers and incrementing the record's sequence number by one. DBX remembers the reference of old record data in the transaction's write-set for garbage collection later (§5.5).

To ensure correctness, DBX protects the entire commit process (i.e. both validation and write phases) within a single RTM region. The resulting code, as shown in algorithm 1, is easy to read (For now, we ignore lines 18-22 which are used to support read-only transactions).

Discussion. The size of the RTM region in the commit phase scales with the meta-data of a transaction's reads and writes. The asymmetric read/write limits of RTM hardware are beneficial since a database transaction tends to do much more reads and writes. Further, RTM's preference for "reads before writes" also matches the memory access pattern of the commit.

As RTMs do not guarantee progress, DBX must implement a fallback routine. Furthermore, for those transactions performing too many reads or writes to exceed the hardware limit, they rely on the fallback routine to proceed. In algorithm 1, `RTMBegin(CommitLock)` and `RTMEnd(..)` are utility functions wrapping the RTM region with a fallback routine that acquires the global `CommitLock` if the RTM region aborts a threshold number of times. As part of `RTMBegin`, the RTM always checks `CommitLock` and blocks if the lock is being held.

Very large read-write transactions are relatively rare; we have not encountered any that exceeds the RTM working set limit in the TPC-C benchmark. On the other hand, large read-only transactions can be common. Therefore, DBX provides separate support for read-only transactions that do not require RTM-based commits (§5.4).

Correctness Proof. We give an informal argument on why our RTM-based commit phase ensures conflict serializability.

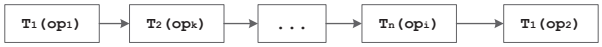


Figure 3: Unserializable Dependency Graph

If the execution result is not serializable, the dependency graph of the transaction will form a cycle. Then, there is at least one transaction T_1: one of its operations op_2 depends one its another operation op_1 (Figure 3). According to the type of operation, there are the following cases.

1. op_1 and op_2 are both write operations. It happens when some concurrent transactions observe op_1's result, but miss op_2's result. It is impossible, because the write phase is protected by RTM. This guarantees that all local updates of a transaction are committed atomically.

2. op_1 and op_2 are both read operations. It happens when some concurrent transactions update the tuples read by op_1, then op_2 observes the concurrent update. It is impossible because if a concurrent update happened between two read operations, the conflict will be detected in the validation phase at the end of execution by checking the sequence number of the record.

3. op_1 writes a record, but op_2 reads a record. It happens when some concurrent transactions observe op_1's result before op_2's execution. It is impossible because a transaction's update can be observed by others only after the transaction commits which is after all the read operations' execution.

4. op_1 reads a record, but op_2 writes a record. It happens before op_2 writes some records, some concurrent transactions have already updated the records read by op_1. In this case, T_1 can not commit successfully, because RTM is used to protect the transaction's validation and commit to provide the atomicity . If there is a concurrent update between read and write operations, a transaction can detect the conflict in the validation phase.

5.3 General Database Operations

The previous section described DBX transactions that only read or write existing records. This section discusses how other types of operations are supported within a transaction.

Read a non-existent record. A transaction that has read a non-existent key conflicts with other concurrent transactions that inserts a record with the same key. To detect this conflict, DBX uses the key-value store's `get-with-insert` interface to insert the pointer to a newly allocated empty record with a sequence number of zero if the key does not exist and tracks the read in the transaction's read-set. If another transaction inserts the same key, the conflict will be detected during the validation phase.

Insert. To insert a new record, DBX uses `get-with-insert` to insert the pointer to a new record if the key does not exist. The actual record value to be inserted is buffered in the write-set. Doing so allows DBX to detect conflicts among transactions that attempt to read the same key.

Delete. Deletion of a record is treated in the same way as a normal write. In the write phase of the commit protocol, DBX simply marks the status of the record as having been deleted. We cannot immediately unlink the record from the memory store since it holds the reference of old versions which may be needed by read only transactions. DBX relies on a separate garbage collection phase to unlink and reclaim deleted records.

Range query. DBX provides range queries with an iterator interface if the underlying key-value store is ordered. Range query faces the phantom problem [12] as concurrent insertions to the range being scanned might violate serializability. Commodity databases usually use next-key locking [30] to prevent insertion in the queried range. Since next-key locking is at odds with our OCC-based commit protocol, we use a mechanism introduced by Silo [41], which also uses OCC-based commit.

In order to detect insertions in the scanned range, the underlying B^+-tree keeps track of modifications on each leaf node using a version number. Any structural modification like a key insertion, node split or key deletion, will increase the version number. During the range scan, the B^+-tree exposes a set of pointers that point to the version number field in each leaf node visited. The version numbers as well as their corresponding pointers are copied to a transaction's read-set. During the validation phase, we abort a transaction if any leaf node has been changed as indicated by the mismatch between the version number remembered in the read-set and the current version number.

5.4 Read-only Transactions

It is important to have a custom design for read-only transactions. Huge transactions, ones that can exceed RTM's hardware limit, tend to be read-only in practice. DBX' design allows these read-only transactions to read the most-recent snapshot of data while only using fixed-size RTM regions.

DBX uses a global snapshot number *S* to indicate the current snapshot. *S* is updated only by read-only transactions. Each read-only transaction atomically reads *S* into local variable *rs* and increases *S* by one (e.g., using *atomic xadd* in x86). Later, this read-only transaction will only read records whose snapshot numbers are not larger than *rs*.

To support snapshot reads, DBX adds extra logic to the write phase of the commit protocol (see Figure 1 lines 18-22). Specifically, the read-write transaction checks if the local snapshot number (i.e., *lsn*) of the modified record is equal to *S* (i.e., *globalSnapshot*). If not, some concurrent read-only transactions may need to read the current version of this record. Thus, the read-write transaction creates a new version to accommodate the write. The current snapshot number is stored in the new version of the record. Because the checking and version creation are done within the RTM for commit, all the writes of the same transaction will have same snapshot number.

DBX's read-only transactions satisfy two important properties: 1) Unlike the previous design [41], DBX's read-only transactions read the most-recent snapshot. Since read-only transactions are responsible for updating the global snapshot number, they are guaranteed to see the most recent snapshot. 2) New snapshots are created only when needed. When a read-only transaction is not in progress, read-write transactions will not trigger the creation of a new version.

5.5 Garbage Collection

Whenever a record is removed from the memory store or an old version is no longer needed, it is put on a per-core garbage list. To safely reclaim memory for objects in the list, we implement a quiescent-state based reclamation scheme [15]. When a thread *T* finishes a transaction, it enters into a quiescent state where *T* holds no references to shared objects. DBX uses a vector timestamp to track the number of transactions executed by each worker thread. We define a grace period as the interval of time during which all threads have finished at least one transaction. This ensures that no inflight transaction holds references to removed or staled records. DBX always waits for a grace period before reclaiming memory and a thread can determine whether a grace period has passed by examining the vector timestamp.

Reclaiming old data. There are two kinds of old data: one is old records and the other is old versions created for read-only transactions. For the former case, as DBX removes the reference of the record from the memory store, no thread will hold its reference after every thread finishes at least one transaction. For the latter case, when entering a grace period, all read-only transactions run on the snapshot smaller than or equal to the global snapshot number will eventually finish and drop references to the old versions. Hence, after a grace period, it is safe to reclaim both kinds of old data.

Unlinking old records. When a record is marked as deleted, DBX cannot remove it from memory store immediately since it holds the reference to old versions. However, when entering a grace period, all old versions created before the deletion can be reclaimed. Thus, when a record is marked as deleted, DBX waits a grace period, before removing it from the memory store. After a record is removed from the memory store, DBX marks the status of the record as removed. If another transaction concurrently attempts to write to a removed record, it will be aborted during the validation phase. This record will be put into garbage collection list for reclamation.

Figure 4 illustrates an example. When a transaction tries to delete record K_3, it marks the record as logically deleted and creates an old copy of this record (old V_3) for concurrent read-only transactions. After a grace period, old V_3 is no longer needed by the read-only transactions and DBX can reclaim its memory. Meanwhile, the deleted record (K_3) is unlinked from the memory store and is marked as removed. After another grace period, DBX can safely reclaim the memory of K_3.

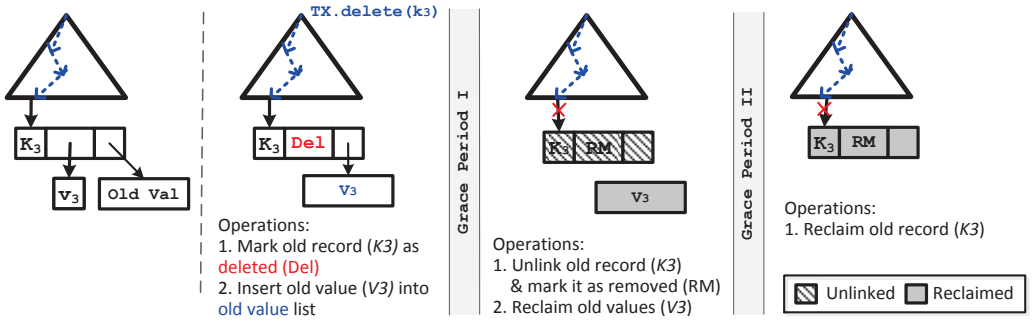


Figure 4: Steps in deleting a record from DBX:1). logically delete the record; 2). unlink the record from DBX; 3). reclaim the memory of the record.

5.6 Durability

To ensure correct recovery from a crash, DBX logs all update transactions to the local disk, similar to that of [41].

When a worker thread commits a transaction, it records all transaction's update information, including the key, value and sequence number in its local buffer. When the buffer is full or a new snapshot is created, the local buffer and the snapshot number of the records in the buffer will be sent to a logger thread. The logger thread periodically flushes the updates from worker threads into durable storage and calculates the latest persistent snapshot s. All transactions committed earlier than s can be returned to client. Hence, the frequency of log flushing and snapshot updating affect client-perceived latency.

During recovery, DBX calculates the latest snapshot by scanning the log file. Then, it sorts the update entries belonged to the same record using sequence number and reapplies the latest update of the record.

6. Implementation

We have built DBX from scratch, which comprises of about 2800 C++ SLOCs. DBX supports transactions as a set of pre-defined stored procedures [19, 42]. A stored procedure consists of a sequence of `get` and `put` operations and runs as a single transaction. Currently DBX does not provide a SQL interface; the stored procedures are hard-coded with the put and get interfaces provided by DBX. DBX currently supports a one-shot request model, under which all parameters are ready for a single transaction and hence transactions can avoid stalls due to interaction with users. This implementation choice allows us to saturate the raw hardware performance much easier.

The rest of this section will discuss two specific optimizations that further boost the performance of DBX.

6.1 Fallback handler

When a transaction aborts, the CPU will restart the execution from a fallback handler specified by the *xbegin* instruction. Since RTM never guarantees a transaction will succeed, DBX needs to combine RTM with a global lock in the fallback handler (i.e., transactional lock elision [32]) to ensure forward progress. A simple scheme is when a transaction restarts, the thread tries to acquire a fallback lock instead of retrying the RTM region. However, frequently acquiring the fallback lock is costly and may notably degrade performance.

DBX uses different retrying thresholds according to abort reasons. When an RTM aborts, its abort reason is stored in the *eax* register, which includes: shared memory access (*conflict abort*), internal buffer overflow (*capacity abort*) and system event (*system event abort*). The threshold for *system event abort* is 2, according to the maximal abort rate (i.e., 50%) for systems events only (Figure 1 (d) in §2). This means that if a transaction continually aborts twice due to systems events, either its execution time is larger than the timer interrupt interval or there may be unrecoverable system events (e.g., page fault) during execution. Hence, it will continually fail with retrying, and thus the transaction should acquire the fallback lock. The threshold for *capacity abort* is set to 10, as the maximal abort rate is 90% (Figure 1 (a) in §2) when writing 31 KB memory. This means if a transaction continually aborts more than 10 times due to capacity aborts, it is likely caused by cache line eviction due to working set overflow. Thus, the transaction should acquire the lock instead of retrying. The threshold of *conflict abort* is set to 100, as the RTM transaction may likely succeed upon retry.

6.2 Buffer Node

Updating a record is common in OLTP workload. For an update operation, DBX first gets the value from the system, modifies the value and then puts the value into the system. Intuitively, it will search the record by traversing the memory store twice (for both get and put). To avoid this, DBX maintains a one-node buffer for each transaction in a database table. For each get or put operation, the node buffer will be checked first by comparing the key. Upon miss, the system will try to get the node by traversing the memory store and store the pointer of the node in the node buffer. According to our evaluation, the buffer has nearly 50% hit rate for

TPCC-neworder benchmark and 6% performance improvement when using B^++ tree as the memory store.

7. Evaluation

This section presents our evaluation of DBX, with the goal of answering the following questions:

- Can DBX provide good performance and scalability on a real RTM machine for typical OLTP workloads?
- Can DBX and its memory store perform comparably with other highly-tuned fine-grained locking counterparts?
- How does each design decision affect the performance of DBX?

7.1 Experimental Setup

All experiments were conducted on an Intel Haswell i7-4770 processor clocked at 3.4GHz, which has a single chip with 4 cores/8 hardware threads and 32GB RAM. Each core has a private 32 KB L1 cache and a private 256KB L2 cache, and all four cores share a 8MB L3 cache. The machine has two 128GB SATA-based SSD devices and runs 64-bit Linux-3.10.0.

We use two standard benchmarks for key-value store and databases: YCSB [5] and TPC-C [37]. The YCSB and TPC-C benchmarks are derived from Silo [41]. To make a direct comparison with the B^+-tree part of Masstree (a state-of-the-art trie-structured tree highly tuned for multicore), and Silo, we fix the key length to 64 bit for both YCSB and TPC-C. The read/write ratio for YCSB is 80/20 and the size of a record value is 100 bytes. The write operation for YCSB is read-modify-write. For TPC-C, DBX uses the B^+-tree memory store by default.

For experiments with durability enabled, two logger threads are used and each is responsible for logging database records on a separate SSD. Each SSD has 200MB/s maximal throughput for sequential write, which accumulatively provide 400MB/s write throughput.

We use SSMalloc [26] as the memory allocator for all evaluated systems, as it has superior performance and scalability than other allocators evaluated. To eliminate networking overhead, each workload generator directly issues requests to a database worker. By default, we execute 5 million transactions concurrently to evaluate the throughput of the system with TPC-C. Logging is disabled by default, §7.6 will show the performance of DBX with durability enabled.

7.2 End-to-end Performance

We first present the end-to-end performance of DBX compared to other multicore in-memory databases. We compare DBX against two different implementations, Silo and Silo-Masstree. Silo [41] is a most scalable and fastest in-memory database previously published. During evaluation, we found that Silo's B^+-tree implementation is not as optimized as that of Masstree [28]. Hence, we used a variant of Silo, referred to as Silo-Masstree, which used the B^+-tree implementation of Masstree. To make the results directly comparable, DBX only uses B^+-tree instead of mixing it with hash tables by default. All three systems are evaluated using the TPC-C standard mix benchmark. All logging functionalities were disabled.

As shown in Figure 5, DBX performs better than both Silo-Masstree and Silo. Compared with Silo-Masstree, DBX has 52% improvement (119,868 vs. 78,816 transactions per second, TPS) when running 1 worker thread and 40% improvement (442,230 TPS vs. 314,684 TPS) for 8 worker threads. One source of overhead from Silo-Masstree is from the byte stream in the tree structure of Masstree, which requires it to encode the tree node structure into byte stream. Further, Silo-Masstree allocates a new memory chunk for both put and get operations. In contrast, DBX only allocates memory chunks for put operations. To study the overhead of these differences, we re-implement DBX to use byte stream in memory store and copy records for get operations (DBX').

After aligning the implementation difference, DBX still has 13% improvement (356,911 TPS vs. 314,684 TPS) over Silo-Masstree when running 8 threads. Our analysis uncovered that the overhead of Silo-Masstree mainly comes from its OCC implementation using two-phase locking: 1). Before validation, it acquires all the locks of the records in the write set. Thus, it needs to sort the locks to avoid deadlock; 2). Using fine-grained locking also means using more atomic instructions and thus memory barriers, while DBX essentially groups a number of memory barriers inside an RTM region into one; 3). During validation, it still needs to traverse the write set to check if any validated record is locked by itself. Beside these, RTM-based B^+-tree also performs slightly better than the concurrent B^+-tree implementation in Masstree (§7.3).

To understand the overhead from RTM, we removed RTM in the transaction layer, which, however cannot guarantee the correctness. As shown in figure 5, RTM in DBX has only less than 10% overhead.

7.3 RTM vs. Traditional Synchronization Means

This section compares DBX with traditional synchronization methods for both transaction layer and data structures used in the memory store.

7.3.1 Benefits of RTM in Memory Store

B^+-tree. We use YCSB to compare the memory store performance in DBX with its alternatives. First, we compare our RTM B^+-tree with Masstree and Silo's default B^+-tree. Masstree uses a trie-like concatenation of B^+-trees and uses read-copy-update like technique to support non-blocking read during writes. Masstree is optimized for comparison of variable-length keys with common prefixes, which DBX currently does not support. However, when the key length is no larger than 64 bit, Masstree is essentially a scalable B^+-tree (Mass-B^+-tree for short). As Figure 6 shows, our RTM-based B^+-tree performs slightly better than Mass-B^+-tree

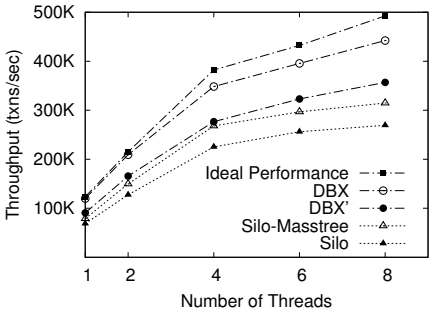


Figure 5: *DBX vs. Silo (TPC-C)*

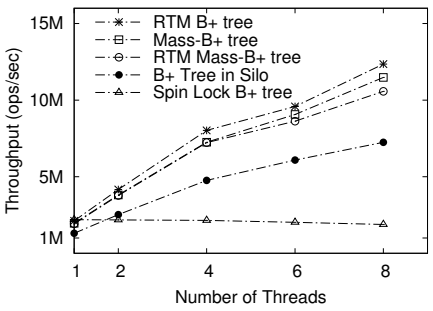


Figure 6: *RTM B^+tree vs. Fine-grained locking B^+tree in masstree (YCSB)*

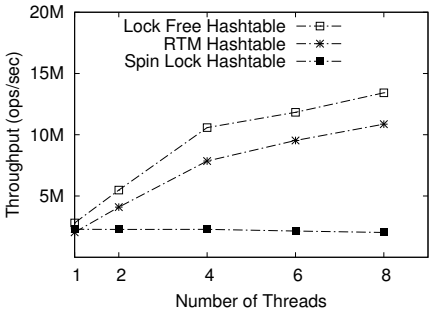


Figure 7: *RTM Hashtable vs. Lockfree Hashtable (YCSB)*

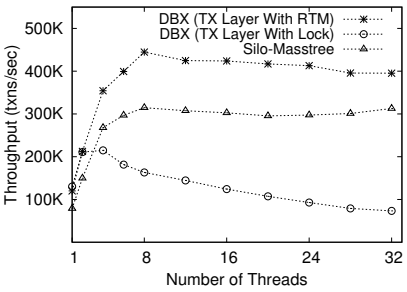


Figure 8: *Scalability of DBX (TPC-C)*

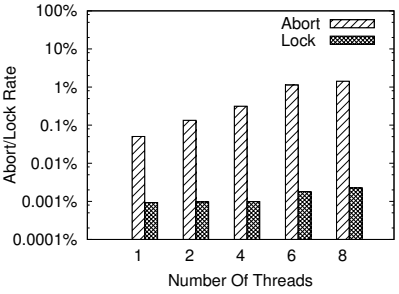


Figure 9: *RTM Abort/Lock Rate*

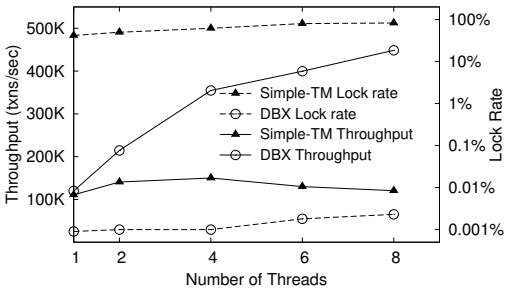


Figure 10: *DBX vs. protecting whole transactions using RTM (Simple-TM)*

(about 7.6% for 8 threads). According to our profiling results, though the tree in DBX has larger average cycles per instruction (CPI) than Mass-B^+-tree (3.67 vs. 2.76 for 8 threads) due to worse cache locality, its simple implementation results in much less instructions executed (only around 68% of Mass-B^+-tree). One major source of more instructions in Mass-B^+-tree is from checking conflict among concurrent operations. For example, because the readers do not lock against concurrent writers, it needs to check versions to avoid reading an inconsistent record during traversing the tree (§4.6 of [28]). As for the B^+-tree used in Silo, it is slower than Masstree by 36% as it is not as optimized as Masstree.

We also compare an RTM version of Mass-B^+-tree by using RTM to protect the single-threaded version of Mass-B^+-tree, which contains no synchronization means to coordinate concurrent accesses. Note that even if we use the same fallback handling policy for the RTM version of Masstree, it has worse performance than Masstree (8% throughput slowdown) mainly due to more frequent RTM aborts, which account for 16% of total CPU cycles.

We also vary the ratio of the YCSB workload to 50% get, 25% insert and 25%put. Such workload taxes more on the updates and structural modifications of the tree. RTM B^+-tree still slightly outperforms Masstree (by 1%), while Masstree using RTM is around 16% worse than Masstree and our RTM B^+-tree.

Hash table. We also implement a lock-free hash table, which is refined and tuned from the same code base with the RTM version. The hash table is chained with a lock-free linked list [13]. Because deletion and resizing can make the lock-free algorithm much more complex, they are not supported in the current lock-free algorithm. To insert a node, only one atomic instruction (*cmpxchg*) is used to change the predecessor's pointer in the linked list. No synchronization instruction is needed to get a node. As Figure 7 shows, the RTM version has lower throughput than the lock-free version (19% slowdown for 8 threads). The reason is that DBX uses RTM to protect both put and get functions, which causes more overhead than the atomic instruction (20 cycles vs. 70 cycles to protect a single write operation), as the critical section size is very short. The cost of RTM in the get operation accounts for more than 90% of the total overhead. However, if *get* uses no synchronization means, there will be subtle order requirements of pointer operation in *put* to prevent readers from reading an inconsistent linked list.

Third, we also use a spinlock to replace RTM. As shown in the Figure 6, this implementation will get significantly worse performance scalability than other alternatives.

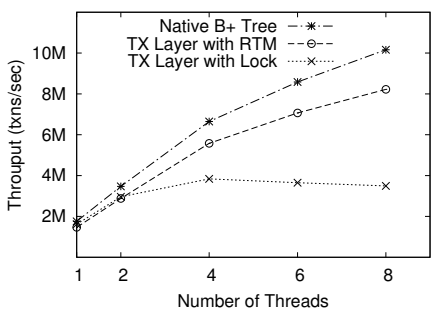


Figure 11: *Overhead for small transactions*

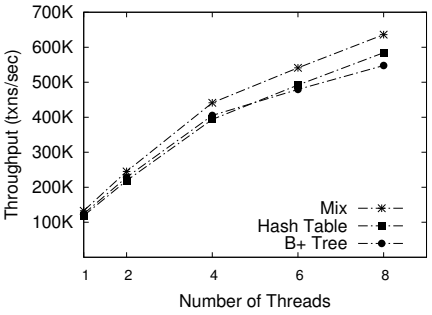


Figure 12: *Various data structures (TPC-C New-Order)*

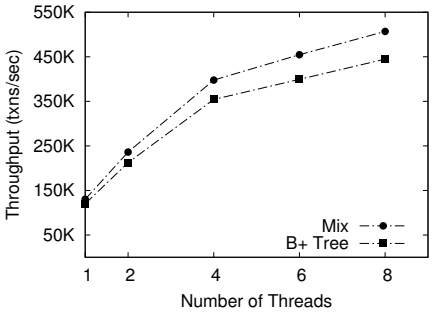


Figure 13: *Various data structures (TPC-C)*

7.3.2 Cost and Benefit of RTM in DB Transactions

We use the standard workload of TPC-C with a mix of five transactions, which includes read-only transactions and range queries to evaluate the performance of DBX transactions. Each worker thread is configured with a local warehouse, as done in prior work [41].

Transaction layer. We compare DBX's throughput with its spinlock counterpart by scaling the number of worker threads from 1 to 32. We use spinlock to replace RTM only for the transaction layer, by using spinlock to protect the commit phase and record read in the transaction layer. The memory store still uses the scalable RTM B^+tree implementation.

As shown in Figure 8, DBX with RTM can scale well from 1 to 8 threads, which is the maximal number of hardware threads on our machine. At 8 threads, the throughput for TPC-C reaches 442,230 transactions per second (TPS), projecting to 110,557 per core or 55,278 per thread. If the number of worker threads exceeds the number of hyper-threaded cores, DBX incurs a little performance slowdown, e.g., 10% under 32 threads. This is mainly caused by cache misses and RTM aborts due to frequent context switches. By contrast, the spinlock version cannot scale beyond 4 threads due to contention on spinlock, which causes notable overhead and unstable performance scalability. When the number of threads exceeds 8, the slowdown is also caused by the fact that a thread holding a lock may be preempted without releasing the lock. If a thread tries to acquire the lock, it has to wait until the lock-holder is re-scheduled to release the lock.

With RTM, there is no such lock-holder preemption problem. This is because an RTM transaction will be aborted if the thread is preempted and thus will not block other running threads trying to enter the critical section. DBX also consistently outperforms Silo-Masstree. However, the scalability of DBX slightly degrades with more database threads than hardware threads, because of more RTM aborts due to frequent context switches. In contrast, Silo-Masstree does not suffer from this and thus scales slightly better than DBX.

Abort Rate and Lock Rate. To evaluate if DBX can fit into RTM, we study the RTM transaction abort rate and fallback lock acquiring rate. Fallback lock acquiring rate is calculated by dividing the times of acquiring fallback locks by the number of total transactions.

Figure 9 shows the abort rate and lock acquiring rate. Both rates are very small. For one thread, the abort rate is 0.05% and lock acquiring rate is 0.001%. Around 80% abort is capacity abort that may succeed on retry. For 8 threads, the abort rate increases to 1.4%, and the lock acquiring rate increases to 0.002%. There are two major reasons for the increase: on one hand, more concurrent threads means more conflict aborts due to share data accesses. For 8 threads, conflict abort is around 46% of the total aborts, as the maximal working set of RTM is reduced when two hardware threads share the L1 cache. For 8 threads, capacity aborts account for around 53% of the total aborts.

Protecting Whole Transactions with RTM. An interesting question is whether RTM can protect an entire database transaction to ensure serializability. To answer this question, we use RTM to protect five transactions in TPC-C. In each RTM region, the updates are applied on the records in the memory store directly.

We evaluate the throughput and the fallback lock acquiring rate when using RTM to protect the whole transactions in TPC-C, as shown in figure 10. With 1 thread, the lock acquiring rate is 42%, which is mainly caused by capacity aborts. This is because the memory access pattern of TPC-C is nearly random, which shrinks the maximal RTM working set and increases the chance of cache set conflicts. For example, the *neworder* transaction accesses less than 9KB memory but still has more than 70% lock acquiring rate, with more than 99% from cache set conflicts. Further, the read-only transaction *stocklevel* accesses more than 40KB memory, which leads to 100% lock acquiring rate. This is because the read only-transaction also has memory writes and its large working set causes cache lines in the write-set to be evicted from L1 cache, which will abort the transaction. The lock acquiring rate increases for 8 threads (82%), due to further decreases of maximum RTM working set per

thread. As a result, the throughput cannot scale up with the number of worker threads.

Overhead of Small Transactions. To evaluate the overhead associated with small transactions, we compare the performance of DBX using B^+ with the raw B^+-tree of the memory store using YCSB. Both B^+-trees are protected using RTM. We also compare the transaction layer using spinlock. We form one transaction by enclosing a single *get* operation for the get workload, or a *get* and a *put* operation for the read-modify-write workload. The benchmark comprises 80% get and 20% read-modify-write.

Figure 11 shows the throughput of the three configurations. Both raw RTM B^+-tree and DBX scales to 8 threads, while the spinlock version scales poorly. The overhead of DBX for small transactions is modest (20% for 8 threads). RTM is not only used to protect the commit phase but also to guarantee the consistency between the sequence number and record value for each get operation,. As a result, about 40% overhead is from starting and ending an RTM transaction. This is because the cost to form an RTM region is still large in current Haswell processors (e.g., around 70 cycles to protect a single write operation), which is costly for a short transaction region. However, removing RTM instructions in the get operation will result in more code complexity. We believe that future processors with less RTM cost may further reduce the overhead of DBX for small transactions.

7.4 Benefit from Other Design Choices

This section shows the benefit of two major design choices of DBX: memory store customization and read-only transaction optimization.

7.4.1 Memory Store Customization

This section shows the performance of DBX based on the memory store implemented using two different data structures (hash table and B^+-tree), and how they compare with DBX's mixture of them. Because the hash table does not support range queries, we only run the TPC-C new-order transaction to do the comparison for all database tables. As shown in Figure 12, both data structures can scale up to 8 threads and hash table performs better when number of threads is more than 4.

As other transactions will perform range query on ORDER-LINE and NEW-ORDER tables, we use B^+-tree to store the records of these two tables and secondary indexes. The records in other seven tables are stored using hash table. Interestingly, even there is no range query in new-order transaction, a mixture of B^+-tree and hash table still performs better than hash table only implementation. The reason is that the new-order transaction will perform sequential access on the ORDER-LINE table, which is more suitable for B^+-tree than hash table. Figure 13 also compares DBX's mixture of data structures with B^+-tree only implementation using the standard TPC-C benchmark. At 8 threads, the former one achieves 506,817 TPS, projecting to 126,704 per core or 63,352 per thread, while the latter only achieves 442,230 TPS. This confirms the benefit of customization enabled by the modular design of DBX.

7.4.2 Read-Only Transactions

To evaluate the effectiveness of read-only transactions, we run a mixed transaction of 50% new-order (read-write transaction) and 50% stock-level. Stock-level is the largest read-only transaction of TPC-C that will touch hundreds of records. Each warehouse is assigned to 2 threads. We compare the throughput and abort rate of DBX with and without read-only transaction support. In the later case, a read-only transaction is treated as normal read-write transaction, which has the validation phase and can be aborted due to conflict.

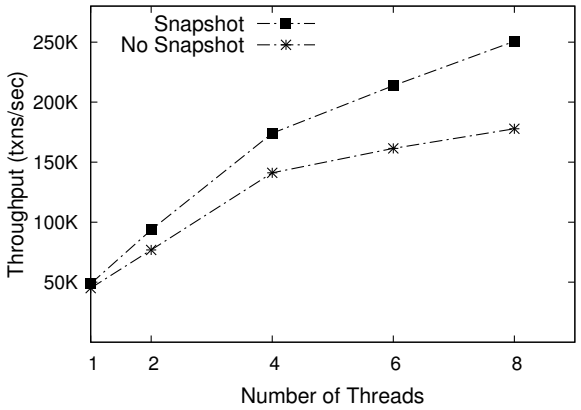


Figure 14: Effect of read only optimization (TPC-C Read Only)

Figure 14 shows the result of this experiment. The support of read-only transaction improves the performance by 1.41x at 8 threads. The abort rate is reduced from 10% to 1.2%.

7.5 Factor Analysis

To understand the performance impact of various aspects of DBX, we present an analysis of multiple factors in Figure 15. The workload is the standard TPC-C mix running with 8 warehouses and 8 worker threads.

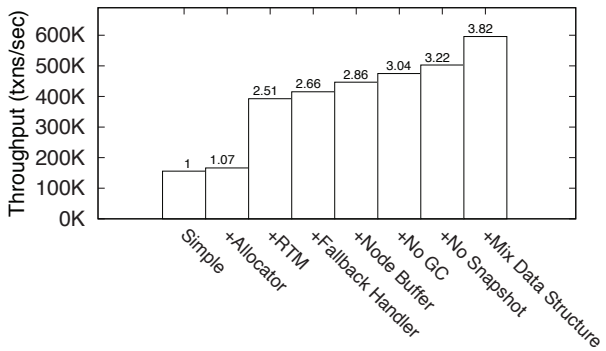


Figure 15: Factor Analysis

Baseline refers to DBX running with the default allocator with a global lock. +Allocator adds the SSMalloc. +RTM means using RTM to protect the commit protocol instead of spinlock; when an RTM transaction aborts, it just simply retries one hundred times before acquiring the lock. +Fallback handler means using the abort reason to adjust the thresholds in the fallback handler. +Node buffer means using the node buffer optimization. +No GC means disabling the garbage collection. +Nosnapshot means disabling the read-only transaction optimization. This also slightly improves the performance for standard-mix benchmark because the memory allocation is reduced. While read-only transaction support can reduce abort rate for read-only transactions, the standard-mix benchmark only contains 4% stock-level transactions, whose benefit does not exceed the overhead from additional memory allocation. This phenomenon is similar to other systems like Silo [41].

7.6 Durability and Its Impact on Latency

We also evaluated DBX with durability enabled. Figure 16 shows the throughput of both DBX and Silo-Masstree when running TPC-C with durability enabled. Both of them can scale up to 8 threads, DBX with durability has 30% performance improvement over Silo-Masstree with durability on average. However, when the number of worker threads is more than 2, the scalability of DBX with durability is a little bit worse than DBX without logging. This is because our test machine has only 4 cores, two logger threads start to contend for cores when the number of working threads is more than 2.

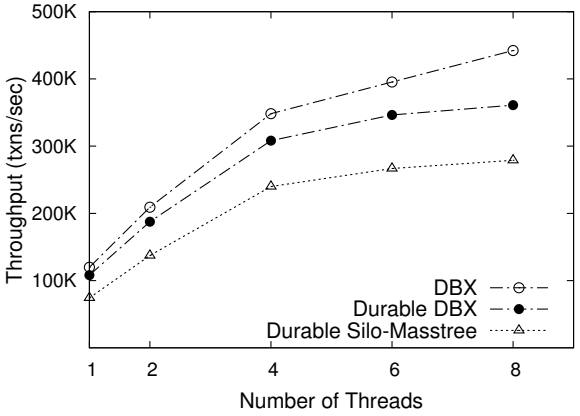


Figure 16: Throughput of DBX with durability enabled (TPC-C)

Figure 17 shows the transaction latency for both systems. Since Silo-Masstree advances the epoch number for every 40 ms, the transaction latency is more than 40ms (117ms∼480ms). For DBX, the snapshot is advanced by both of a separate thread for every 40ms and the read-only transaction. As a result, DBX has lower transaction latency, which ranges from 38 ms to 63 ms. For DBX, it has higher latency with 1 thread. This is because only one SSD is used

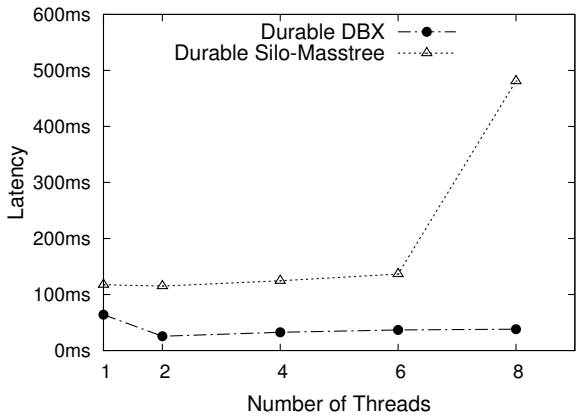


Figure 17: Laytency of DBX with durability enabled (TPC-C)

when running 1 worker thread, and the I/O becomes the bottleneck. Silo-Masstree has higher latency with 8 threads. This is because worker threads and the thread advancing epoch contend with logger threads. However, DBX does not have higher latency with 8 threads. This is because read-only transactions also advance the snapshot.

8. Related Work

Multicore in-memory databases: The dominance of multicore processors and the need of running speedy in-memory transactions [14] have stimulated much work on multicore in-memory databases. Generally, they can be classified into three types: shared nothing [31, 35, 42], shared everything [7, 18, 22, 41] and hybrid [36]. Shared-nothing designs have lower synchronization overhead but may cause imbalanced load and expensive cross-partition transactions. Shared-everything systems usually require a sophisticated design to reduce synchronization overhead and can be difficult to reason about correctness. DBX is essentially a shared-everything design but uses RTM to simplify the design and implementation.

DBX's design is influenced by Silo [41], a shared-everything database based on optimistic concurrency control [21]. Silo uses uses fine-grained locking with atomic instructions to ensure the atomicity of the validation and commit stage, while DBX leverages RTM. Further, Silo allows read-only transactions to read a potentially stale snapshot, while DBX ensures that a read-only transaction always reads the most recent committed transaction by creating a database snapshot on-demand. Our evaluation shows that DBX has higher throughput and smaller latency over Silo on a 4-core machine due to increased concurrency from RTM. Nevertheless, Silo has been shown to scale up to 32 cores while DBX's evaluation is limited to 4 cores due to current hardware limits.

In a position paper, Tran et al. [40] compared the performance of using spinlocks or transactional memory in place of database latches. However, they used a much stripped

down database with only 1,000 records and the database transaction size is not large enough to uncover the RTM limits. It is also not clear how the commit protocol is implemented and whether it guarantees serializability. In contrast, DBX is a full-fledged OLTP database with ACID properties that can run complete TPC-C workloads.

A parallel effort [24] also investigates how to use RTM in the Hyper main-memory database [20]. Their approach is different from DBX. For example, it uses a variant of timestamp ordering to group together small RTM regions, while DBX uses a variant of OCC [21] to separate transaction execution from its commit. Further, DBX is carefully designed to leverage the performance characteristics of RTM (like asymmetric read/write set and "read before write" access pattern), while [24] is oblivious to these issues.

DBX is also influenced by prior modular database designs, such as Monet [3], Stasis [33], Anvil [27] and Calvin [38]. Specifically, the separation of transactional layer and the data store allows DBX to use different RTM regions and RTM-friendly data structures for different stores.

Concurrent Data Structures using HTM: Transactional memory [8, 16] has long been the focus of the research community to provide scalable performance with less complexity. Wang et al. [43] provide a comparative study of RTM with fine-grained locks and lock-free designs for a concurrent skip list. This paper further studies the performance of concurrent B+ trees and hash tables under RTM.

Dice et al. [9, 10] have implemented concurrent data structures such as hash tables and red-black trees using HTM in Sun's Rock prototype processor [4]. Matveev and Shavit [29] also conduct an emulated study of different fallback schemes on the scalability of common data structures and propose reduced hardware transactions to eliminate instrumentation in hybrid transactional memory. There have also been studies using HTM to simplify synchronization primitives such as read-write lock [11] and software transactional memory [6]. These studies may provide a different way to scale up the data store of DBX.

9. Conclusion

We have presented DBX, a serializable in-memory database using restricted transactional memory. DBX centralizes its design by using RTM to protect memory store and transactional execution accordingly. By using an optimistic concurrency control algorithm and only protecting the validation and write phases, DBX restricts RTM's working set to only the meta-data of the touched records instead of the data, which significantly reduces the RTM abort rate. DBX is also built with a new read-only transaction protocol that requires no validation phase but still allows a read-only transaction to read the most-recent database snapshot. Our implementation and evaluation with DBX are encouraging: DBX has less complexity than prior systems but still has notably better performance and scalability on a 4-core 8-thread Haswell machine.

10. Acknowledgement

We thank our shepherd Pascal Felber, Frank Dabek and the anonymous reviewers for their insightful comments, and Han Yi for evaluating DBX against its counterparts. This work is supported in part by a research gift from Intel Corp., the Program for New Century Excellent Talents in University of Ministry of Education of China, Shanghai Science and Technology Development Funds (No. 12QA1401700), a foundation for the Author of National Excellent Doctoral Dissertation of PR China, China National Natural Science Foundation (No. 61303011) and Singapore NRF (CREATE E2S2). Jinyang Li is partially supported by NSF CNS-1065169.

References

[1] D. Batoory, J. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. Twichell, and T. Wise. GENESIS: An extensible database management system. *IEEE Transactions on Software Engineering*, 14(11):1711–1730, 1988.

[2] C. Blundell, E. C. Lewis, and M. M. Martin. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5(2):17–17, 2006.

[3] P. A. Boncz. *Monet; a next-Generation DBMS Kernel For Query-Intensive Applications*. 2002.

[4] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Rock: A high-performance SPARC CMT processor. *Micro, IEEE*, 29(2): 6–16, 2009.

[5] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc. SoCC*, pages 143–154. ACM, 2010.

[6] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid norec: A case study in the effectiveness of best effort hardware transactional memory. In *Proc. ASPLOS*, pages 39–52, 2011.

[7] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Servers memory-optimized OLTP engine. In *Proc. SIGMOD*, 2013.

[8] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Distributed Computing*, pages 194–208. Springer, 2006.

[9] D. Dice, Y. Lev, M. Moir, D. Nussbaum, and M. Olszewski. Early experience with a commercial hardware transactional memory implementation. In *Proc. ASPLOS*, 2009.

[10] D. Dice, Y. Lev, V. J. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Simplifying concurrent algorithms by exploiting hardware transactional memory. In *Proc. SPAA*, pages 325–334. ACM, 2010.

[11] D. Dice, Y. Lev, Y. Liu, V. Luchangco, and M. Moir. Using hardware transactional memory to correct and simplify and readers-writer lock algorithm. In *Proc. PPoPP*, pages 261–270, 2013.

[12] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Comm. of the ACM*, 19(11):624–633, 1976.

[13] M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *Proc. PODC*, 2004.

[14] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, 1992.

[15] T. E. Hart, P. E. McKenney, and A. D. Brown. Making lockless synchronization fast: Performance implications of memory reclamation. In *Proc. IPDPS*. IEEE, 2006.

[16] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. ISCA*, 1993.

[17] IBM. IBM solidDB. http://www-01.ibm.com/software/data/soliddb/.

[18] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *Proc. EDBT*, pages 24–35. ACM, 2009.

[19] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker, Y. Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB*, 1(2):1496–1499, 2008.

[20] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proc. ICDE*, pages 195–206, 2011.

[21] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM TODS*, 6(2):213–226, 1981.

[22] P.-Å. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. In *Proc. VLDB*, pages 298–309, 2011.

[23] P. L. Lehman et al. Efficient locking for concurrent operations on B-trees. *ACM TODS*, 6(4):650–670, 1981.

[24] V. Leis, A. Kemper, and T. Neumann. Exploiting Hardware Transactional Memory in Main-Memory Databases. In *Proc. ICDE*, 2014.

[25] B. Lindsay, J. McPherson, and H. Pirahesh. A data management extension architecture. In *Proc. SIGMOD*, pages 220–226, 1987.

[26] R. Liu and H. Chen. SSMalloc: A Low-latency, Locality-conscious Memory Allocator with Stable Performance Scalability. In *Proc. APSys*, 2012.

[27] M. Mammarella, S. Hovsepian, and E. Kohler. Modular data storage with Anvil. In *Proc. SOSP*, pages 147–160, 2009.

[28] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Proc. EuroSys*, pages 183–196, 2012.

[29] A. Matveev and N. Shavit. Reduced hardware transactions: a new approach to hybrid transactional memory. In *Proc. SPAA*, pages 11–22. ACM, 2013.

[30] C. Mohan. ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes. In *Proc. VLDB*, pages 392–405, 1990.

[31] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *Proc. SIGMOD*, pages 61–72. ACM, 2012.

[32] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proc. MICRO*, pages 294–305, 2001.

[33] R. Sears and E. Brewer. Stasis: flexible transactional storage. In *Proc. OSDI*, pages 29–44, 2006.

[34] S. Sen and R. E. Tarjan. Deletion without rebalancing in balanced binary trees. In *Proc. SODA*, pages 1490–1499, 2010.

[35] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era:(it's time for a complete rewrite). In *Proc. VLDB*, pages 1150–1160, 2007.

[36] T. Subasu and J. Alonso. Database engines on multicores, why parallelize when you can distribute. In *Proc. Eurosys*, 2011.

[37] The Transaction Processing Council. TPC-CBenchmark (Revision 5.9.0). http://www.tpc.org/tpcc/, 2007.

[38] A. Thomson and D. J. Abadi. Modularity and Scalability in Calvin. *IEEE Data Engineering Bulletin*, page 48, 2013.

[39] C. TimesTen Team. In-memory data management for consumer transactions the timesten approach. *ACM SIGMOD Record*, 28(2):528–529, 1999.

[40] K. Q. Tran, S. Blanas, and J. F. Naughton. On Transactional Memory, Spinlocks, and Database Transactions. In *Proc. Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, 2010.

[41] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy Transactions in Multicore In-Memory Databases. In *Proc. SOSP*, 2013.

[42] L. VoltDB. Voltdb technical overview, 2010.

[43] Z. Wang, H. Qian, H. Chen, and J. Li. Opportunities and pitfalls of multi-core scaling using hardware transaction memory. In *Proc. Apsys*. ACM, 2013.