Exploiting Hardware Transactional Memory for Efficient In-Memory Transaction Processing

Hao Qian, Zhaoguo Wang, Haibing Guan, Binyu Zang, Haibo Chen

Shanghai Key Laboratory of Scalable Computing and Systems
Technical Report
January 2015

Shanghai Key Laboratory of Scalable Computing and Systems
Shanghai Jiao Tong University, China
Software Building, 800 Dongchuan Road.
PHN: (86-21) 34204789

Exploiting Hardware Transactional Memory for Efficient In-Memory Transaction Processing

Hao Qian, Zhaoguo Wang, Haibing Guan, Binyu Zang, Haibo Chen

Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University

January 2015

Abstract

The commercial availability of Intel's Haswell processor suggests that hardware transactional memory (HTM), a technique inspired by database transactions, is likely to be widely exploited for in-memory databases in the near future. Its features such as hardware-maintained read/write sets and automatic conflict detection naturally put forward a challenge on how HTM can successfully support concurrency control for fast in-memory transaction processing.

To answer this challenge, we first conduct a detailed analysis on solely using HTM to protect a database transaction without other concurrency controls such as OCC or 2PL. Our analysis reveals that such an approach may lead to scalability issues due to the high probability of aborting transactions. To reduce HTM-protected transaction regions, we propose to leverage the theory of transaction chopping with several workload-inspired optimizations, including recovering commutativity with deferred execution and a database cache for database operations, to reduce HTM aborts while preserving serializability. While transaction chopping effectively reduces HTM aborts due to memory access conflicts, it also creates a new challenge for read-only transactions, which can no longer observe a consistent database snapshot. To address this problem, we propose a lazy snapshot mechanism that constructs a database snapshot during execution which eventually becomes consistent. We implement the proposed techniques in DBX-TC on an Intel Haswell processor and conduct an extensive performance study. Performance evaluation shows that DBX-TC with our mechanisms outperforms a naively constructed database using HTM by 4.77x and 36% over DBX on a 4-core/8-threads machine. The speedup is especially significant under high contention.

1 Introduction

The continuity of Moore's law in multicore era and the progressive hardware innovations have not only led to more CPU cores being squeezed onto a chip and bigger memory size, but also the advancement of innovative hardware features such as hardware transactional memory (HTM) [19]. The abundant CPU and memory resources in a single machine, together with an increasing demand for processing massive data in real time, have shifted the focus from traditional disk-based databases to in-memory databases [11, 18, 26, 39, 45, 28, 46]. Consequently, this has also significantly changed the workload pattern of transaction processing, shifting it from an I/O bound to a CPU-bound workload.

HTM, which was originally inspired by database transactions, has become commercially available after some 20-years of research through Intel's transaction synchronization extension. This has motivated database researchers to explore the possibility of using HTM to substitute traditional concurrency control in its entirety as a means to achieve better performance. The motivation is intrigued by the similarities between concurrency control for databases and those for parallel programming (e.g., HTM), such as tracking of read set and write set and detecting conflicting accesses. Although recent works [46, 28] have started to exploit HTM for in-memory transaction processing, they mostly only leverage the atomicity feature of HTM to protect a part of concurrency control, while the databases still need to track read set and write set and perform conflict detection. Hence, it remains a challenge to completely place concurrency control in hardware, which would lead to notable performance gain.

In this paper, we first conduct a detailed analysis on naively applying HTM to a widely-used online transaction processing workload, namely TPC-C [41]. The analysis reveals that, with the limited working set of commodity HTM, TPC-C would incur frequent HTM aborts (and thus transaction aborts) due to conflicting memory accesses and lengthy transaction executions.

To mitigate transaction aborts while preserving important transaction invariants such as serializability, we propose a model that leverages the theory of transaction chopping [7, 38], by decomposing a set of pre-known transactions into pieces and constructing a chopping graph [38]. A chopping graph is a graph connecting transaction pieces with conflicting edges that represent potentially conflicting accesses between different transactions, and sibling edges that represent consecutive accesses from the same transactions. A cycle containing both conflicting and sibling edges (called SC-cycle) indicates potential unsafe interleavings[38]. To preserve serializability, we need to remove the SC-cycle by merging consecutive pieces connected with sibling edges from the same transaction into one piece.

Our model leverages several workload-inspired heuristics, such as identifying the non-atomicity requirement of individual transactions which avoids unnecessary merging of pieces, and having a database cache layer for caching recent database records which shortens the working set of a single piece. Hence, they may significantly shorten the execution duration for an HTM region, and improve the parallelism for OLTP workloads.

Real-world transactions are complex; nevertheless, they can be categorized into read-only and read-write transactions. Typically, a read-only transaction only reads the states of a database but usually involves a large number of database records. Since large read-only transactions are mainly used for analysis or status check which will take a long time, a typical solution is to make read-only transactions read from a consistent yet a little stale database snapshot provided by read-write transactions [45, 46]. However, with transactions being chopped into multiple pieces, a read-only transaction may no longer observe a consistent state, as the all-or-nothing atomicity of read-write transactions is no longer preserved. While this problem can be simply addressed by treating read-only transactions as read-write transactions, such an approach will significantly limit the parallelism among database transactions.

To provide serializability while preserving the performance advantages of read-only transactions, we present a new snapshot mechanism, namely eventual snapshot. Under such a mechanism, snapshots are constructed during transaction execution, but will evolve from an inconsistent state to a consistent state eventually. To achieve this, each snapshot is associated with a snapshot number, where a larger number indicates the snapshot is fresher. Each transaction is given a snapshot number representing the snapshot upon which it operates on.

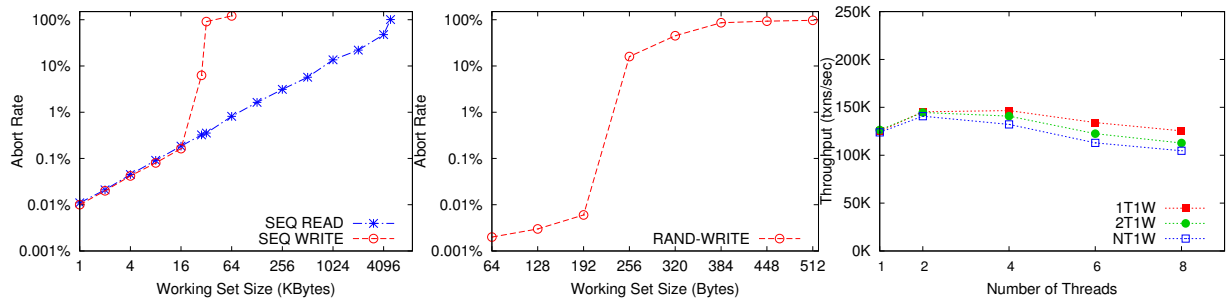
The key idea is to assign the snapshot number according to the serial order of transactions. A transaction should not have a snapshot number larger than the transactions after it. Hence, in our snapshot mechanism, read-write transactions get the global snapshot number, which increases monotonically, in their conflicting pieces. The reason is that the executing order of conflicting pieces decides the serial order of the transactions they belong to. The updates in conflicting pieces are always applied to the latest snapshot.

Such a mechanism is complicated due to commutative pieces. Since the executing order of commutative pieces is not constrained, the latest snapshot may lose updates from transactions with smaller snapshot number. To this end, we leverage a detection and redo process to avoid this inconsistency, where operations in commutative pieces are applied to each snapshot emerged after the transaction starts.

We have extended an HTM-based database system called DBX [46] to incorporate our proposed transaction model, and for ease of reference, we shall call the extended system DBX-TC. We have conducted an extensive performance evaluation. Our evaluation shows that DBX-TC significantly outperforms the naive use of HTM for transaction processing, with a performance speedup by 3X. Detailed analysis shows that the eventual snapshot mechanism contributes up to 67% performance boost. We also compare DBX-TC with the original DBX that uses optimistic concurrency control (OCC) to serialize database transactions. Our evaluation shows that DBX-TC slightly outperforms DBX under no contention due to simplified concurrency control and significantly outperforms DBX under high contention due to fewer aborts caused by access conflicts.

In summary, we make the following contributions:

- The proposal of a transaction model that supports both transaction chopping and HTM-based transaction processing to reduce transaction aborts.
- A set of refinements to avoid unnecessary SC-cycles and reduce the sizes of HTM regions, such as recovering commutativity with deferring execution and building a database cache layer.
- The eventual snapshot mechanism that provides a consistent snapshot while preserving the benefits from read-only transactions.
- The implementation of our proposed model on DBX, which we call DBX-TC, and a comprehensive evaluation that shows DBX-TC outperforms DBX and other baselines.



(a) RTM working set for sequential accesses *(b) RTM working set for random accesses* *(c) Naively Applying HTM*

Figure 1: RTM Working Set and Throughput on TPC-C by Naively Applying HTM (NT1W stands for all threads sharing one warehouse. 2T1W stands for two threads sharing one warehouse. 1T1W stands for each thread accessing its local warehouse)

The rest of this paper is organized as follows. The next section describes necessary background information on HTM and presents a detailed study of naively using HTM to protect database transactions. Section 3 describes the use of transaction chopping and HTM to protect database transactions. Section 4 identifies issues with read-only transactions and how DBX-TC addresses them with the eventual snapshot mechanism. Section 5 discusses some implementation issues. Next, section 6 presents the evaluation results and section 7 discusses related work. Finally, section 8 concludes this paper.

2 Background and Motivation

2.1 Hardware Transactional Memory

The new Intel's Haswell processors provide HTM in the form of restricted transactional memory (RTM) and hardware lock elision (HLE). In this paper, we shall only focus on RTM since it is more flexible and powerful.

RTM generally supports three instructions, *xbegin*, *xend* and *xabort*. The codes between *xbegin* and *xend* are executed transactionally and *xabort* is used to abort an RTM transaction[1] explicitly. The processor provides a set of on-chip storage and extensions to track the read set and write set of an RTM transaction, as well as to detect conflicting accesses among multiple RTM transactions. Like a database transaction, RTM provides atomicity, consistency and isolation properties of an RTM transaction. Unlike a database transaction, there is no durability support in commodity HTMs.

RTM provides a straightforward programming model that combines the benefits of both coarse-grained and fine-grained locks. Coarse-grained locks usually lead to poor performance, while fine-grained locks may risk of deadlocks and are hard to reason about the correctness.

However, RTM, as indicated by its name, also comes with several limitations. First, the working set of an RTM region is limited. As the underlying hardware uses the CPU cache to track read and write sets of RTM transactions, if the sizes of sets exceed the hardware limit, an RTM transaction will be aborted. According to an evaluation conducted in [46], an RTM transaction on a recent Intel Haswell processor can read at most 4MB and write at most 31KB memory on a single core. While these seem abundant for a database transaction, the probability of HTM aborts increases drastically with the increase of working set size.

Figure 1(a) shows the relationship between the HTM abort rate and the working set size from the study in [46]. The abort rate increases exponentially with the increase of write set, especially when the working set size is larger than 16KB. However, this evaluation was done with sequential accesses. In practice, a database transaction may access memory randomly, which will further limit the size of reading/writing sets. To confirm this, we conduct an evaluation on random write by allocating an integer array of 4MB. Each element in the array is assigned a random index to the array, indicating the next element and the cache line to access. As shown in Figure 1(b), an RTM transaction has an abort rate of more than 90% after it writes 448 bytes.

RTM transaction aborts are caused by several reasons. System events such as context switches and interrupts are not allowed in RTM transactions, which lead to an HTM abort (called a system abort). Further, RTM keeps its reading and writing sets during transaction execution at the granularity of a cache line. The RTM transaction may

[1]To differentiate with database transaction, we use *RTM transaction* to denote a piece of code protected using RTM.

experience a conflict abort if other RTM transactions have a conflicting cache line access during its execution. As the on-chip storage is limited, an RTM transaction accessing too much memory may also cause a transaction abort (called a capacity abort). Apart from the total size limit, capacity aborts may be caused by accessing more than *N* cache lines in one cache set. (*N* is the cache associativity.) This is likely to happen with random accessing. When an RTM transaction aborts, the execution flow will be directed to a fallback handler, which may either retry the RTM transaction or acquire a coarse-grained lock.

2.2 Direct Deployment of HTM

The most straightforward way of using HTM to build a database is to put the whole transaction into an RTM transaction region. This, however, may result in suboptimal performance.

Table 1: Transaction working set in TPC-C

Transaction	Read (B)	Write (B)	Total (B)
NEWORDER	7,864	1,938	8,312
DELIVERY	13,567	4,178	16,404
PAYMENT	2,240	656	2,372
STOCKLEVEL	40,101	1,341	40,493
ORDERSTATUS	2,241	469	2,394

We conduct an evaluation on the TPC-C benchmark with all its five transactions. Table 1 gives the average working set of each transaction. From the table, we can see that the actual working set size for each transaction is within the limit of maximally allowed working set size of RTM. However, the random access nature of such transactions actually makes the actual maximum working set much smaller.

Figure 1(c) shows the evaluation results. The number of warehouses is adjusted to represent different levels of contention. As shown in the figure, the scalability is poor even under low contention. Our analysis indicates that too many capacity aborts of RTM transactions due to cache set associativity lead to almost serial execution of transactions. The threshold of capacity abort was set to be one, which means that after the transaction experiences a capacity abort, it is redone with coarse-grained locks.

Our evaluation shows that 88% of database transactions fall back to coarse-grained locks when running on one thread. Although the working size seems to fit with RTM memory access limit, accessing too many cache lines in one cache set results in excessive capacity aborts. Typical CPU L1 cache is 8-way set associated and thus accessing more than 8 cache lines may cause an RTM abort. Note that, we have already leveraged a cache-friendly memory allocator [31] to avoid unnecessary capacity cache misses. Interestingly, the performance is slightly better under high contention, since the total amount of data in the system is less and this results in a better cache locality.

Observation: Even if the studied database transactions fit into the maximally allowable working set of RTM, a database transaction accessing too many cache lines in one set can lead to frequent RTM aborts. This means that it is still desirable to reduce code region protected by an RTM transaction.

3 Refining Database Transactions for RTM

The nice properties like atomicity, consistency and isolation of RTM make it an ideal alternative to ensure the transactional semantics of each transaction. This is because RTM has already provided supports to track the read and write set of a transaction, as well as to detect conflicting accesses. Nevertheless, naive applications of HTM to database transactions may result in poor performance due to excessive HTM aborts. So, it is critical to reduce the HTM protected regions for a database transaction.

This section describes how to leverage the theory of transaction chopping [38], a classical technique to reduce the duration of concurrency, to reduce the HTM regions. Specifically, it leverages static analysis [7] to chop a set of pre-known transactions into small pieces and then construct a chopping graph to analyze cyclic conflicts (i.e., SC-cycles) between transactions. However, trivially applying transaction chopping can usually chop very few transactions into smaller pieces, due to the extensive existence of cyclic conflicts in real-world transactions like TPC-C.

This section first describes chopping graph, the sole algorithm of transaction chopping to preserve the serializability of chopped transactions, and then presents a set of workload-inspired optimizations to avoid SC-cycles and reduce working set sizes.

3.1 Chopping Graph

A chopping graph is an un-directed graph with a set of transaction operations connected with S-edges (sibling edges) and C-edges (conflicting edges) [38]. The steps of building a chopping graph are as follows. First, all transactions

are chopped into pieces. Each piece contains one operation and is represented by a vertex in the chopping graph. Two consecutive pieces in one transaction are connected with a sibling-edge (S-edge). Two conflicting pieces from different transactions are connected with a conflicting-edge (C-edge). If there is an SC-cycle (a cycle containing both S-edges and C-edges), it should be removed by merging pieces connected with S-edges in the cycle. The construction of the chopping graph is not finished until there is no SC-cycle in the graph. In the runtime, each piece in the chopping graph is executed atomically, just as a single transaction.

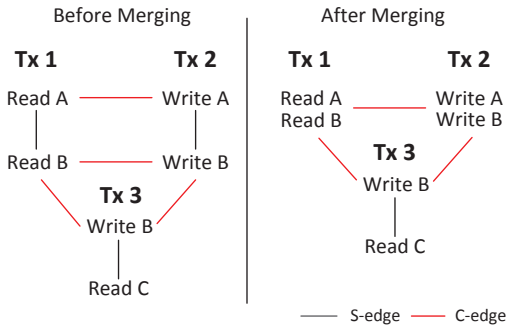


Figure 2: An example of transaction chopping

Figure 2 shows an example of transaction chopping. There are three transactions in total. Each transaction is chopped into two pieces connected with an S-edge as it contains two operations. Transaction 1 (Tx1) and Transaction 2 (Tx2) conflict on *A*, so there is a C-edge between the first pieces of the two transactions which access *A*. Similarly, three C-edges are added due to conflicts on *B*. The pieces from Tx1 and Tx2 form an SC-cycle, leading to the merging of pieces from the same transaction. After merging, both Tx1 and Tx2 contain only one piece with two operations.

In practice, transactions are chopped at the granularity of database tables. During static analysis, the actual records accessed in the runtime are unknown. When two pieces access the same table, and at least one of them contains write operations, they are considered to be in conflict, unless they are commutative. As there may be multiple instances of a transaction executing simultaneously, it is a common practice to construct a chopping graph by using two instances of each transaction [47]. Further, there are also a set of optimizations to reduce SC-cycles and piece size, like reordering of independent operations and leveraging commutativity [38, 16, 47, 32].

3.2 Exposing More Concurrency

Transaction chopping may improve parallelism by chopping transactions into small pieces, which may be executed in parallel and may have shorter HTM regions. However, traditional analysis may result in multiple SC-cycles to be merged and hence transaction pieces that are still relatively very large, which may still cause excessive RTM aborts.

Here, we describe several workload-inspired optimizations to expose more opportunities for chopping.

Making Use of Snapshots: Read-only transactions accessing many records are likely to cause conflicts, which lead to merging of many pieces from not only the read-only transactions but also other read-write transactions. Hence, we extend the traditional transaction chopping by letting them read from a consistent but a little stale snapshot (section 4). As a result, read-only transactions can be removed from the chopping graph.

Recovering Commutativity with Deferred Execution: A database operation is commutative not only if its operation type is commutative, but also if its result does not need to be visible by other transactions. For example, the DELIVERY transaction in TPC-C increases the balance of a customer, which is commutative between two instances of the DELIVERY transaction. However, the PAYMENT transaction will read the customer's balance, which turns it into a non-commutative but conflicting operation.

We convert such non-commutative operations into commutative ones by proposing a publish-subscribe scheme. Specifically, the publisher transactions (e.g., DELIVERY) aggregate the commutative updates locally and only publish this result after a period of time (i.e., an epoch). Hence, the subscriber (e.g., the PAYMENT) transactions may only conflict with the publisher transactions (e.g., DELIVERY) at the publishing time. To further remove such a conflict, we enforce a barrier during publishing such that all publisher transactions are serialized before the on-going subscriber transactions. This essentially defers the execution of publisher transactions until the barrier and the subscriber transactions may only see the results after the barrier.

The key premise of this deferring scheme is that there are no other C-edges connecting the publisher and the subscriber transactions (possibly involving other transactions) in the chopping graph after applying the scheme. Oth-

erwise, the subscriber may still be able to observe the publisher's immediate states through other transitive dependence. Under this premise, we only need to include the conflicting operations between the two transactions in a separate piece and defer it, while letting the results of other pieces be immediately visible to other transactions.

To implement our scheme, the database periodically increases a global epoch number and only publishes the local updates when the global epoch number increases. At this time, each thread will first snapshot the publisher's local results to a predefined location and then synchronize its local epoch number with the global one. A new subscriber transaction will first see if the global epoch number has been changed or not. If so, it will wait until the local epoch numbers of all other threads (may include the publisher transactions) have been synchronized as the global epoch number. Next, it collects all pending results and applies them to the database tables. Afterwards, it can start its own operations. This essentially serializes all subscriber transactions that observe the new global snapshot number after the pending publisher transactions, and all subscriber transactions that do not observe the new global snapshot number before the pending publisher transactions.

Figure 3 shows the chopping graph after applying this optimization. As a by-product, the accesses to the Order and Orderline tables can also be removed from the conflicting piece.

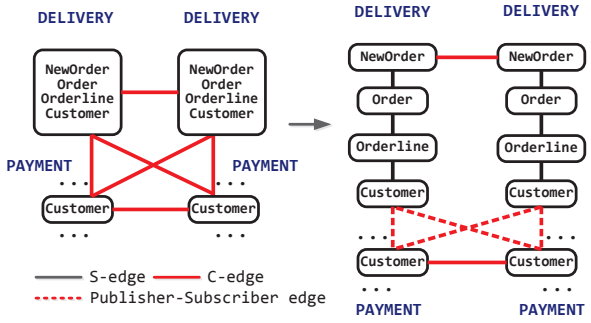


Figure 3: Recovering Commutativity with Deferred Executions

3.3 Reducing HTM Aborts

Besides the refinement of the chopping graph, we also use some optimizations to reduce HTM Aborts.

Database Caches for Database Operations: A transaction may frequently access the database tables, which usually involves a large working set as it may first need to search and update the table.

During the static analysis, we find that some tables are static during execution, as there is no insertion or deletion of records in such tables. As a result, searching the underlying data structures of these tables will not cause conflicts, and can be safely removed from the RTM transaction region protecting the corresponding piece.

For ordinary tables that will be updated during transaction execution, we provide a small database cache for each table. When executing a transaction piece, record updates are written to this cache instead of directly searching and updating the database, which contains a large amount of records. After committing a transaction piece, the cached records may be written back to the underlying database in RTM transactions to keep the cache small. A transaction piece will first read records from the cache before reading from the database. This optimization may decrease the working set of RTM transactions, and thus reduce the possibility of conflict aborts and capacity aborts.

For example, we provide a cache for the Orderline table. The NEWORDER transaction will insert Orderline records to the cache in its conflicting piece, and write these records back to the underlying database table in separate RTM transactions after the piece is finished.

Lock Granularity in Fallback Handler: RTM transactions provide no guarantees on the eventual success of transactions. A fallback handler is usually necessary in case of too many aborts. The common way is to acquire locks in the fallback handler. Using a global lock on each table is a straightforward choice. When an RTM transection begins, it checks whether the lock is held by the others. If so, the transaction aborts explicitly by executing an *xabort* instruction. Nevertheless, this causes false conflicts. Transactions are aborted as long as the global lock is held, even though it has no conflicts with the transaction holding the lock. We propose the per-partition lock to serialize accesses to each partition in the fallback handler. Specifically, in the fallback handler, all the locks of the table partitions accessed by the transaction will be acquired. To avoid deadlock, these locks will be sorted in a fixed order before acquiring. In TPC-C, we use warehouse id as the partition index, which keeps the balance between parallelism and programming complexity.

4 Eventual Snapshot

This section first discusses issues with read-only transactions and then describes an eventual snapshot mechanism to provide a consistent snapshot.

4.1 Issues with Read-only Transactions

Prior approaches usually leverage snapshot isolation [3], a multi-version concurrency control mechanism, to provide read-only transactions with a consistent database state [5]. Specifically, a read-only transaction read records from a snapshot at the beginning of it and will run to completion on this snapshot. Transaction chopping also raises a new issue for read-only transactions. As a transaction will commit its updates at the end of each piece instead of the whole transaction, a read-only transaction can no longer see a consistent snapshot.

Figure 4 shows an example. Suppose X and Y are two records and initially X=0 and Y=0. Transaction1 (Tx1) reads Y's value and then updates X's value to 1. Transaction2 (Tx2) updates Y's value to 1. Transaction3 (Tx3) is a read-only transaction. It reads X and Y's value from snapshot. The interleavings among all operations are shown in the figure. When Tx3 is absent, the execution of the other two transactions are serializable. The equivalent serial execution is to execute Tx1 first, then Tx2. However, after Tx3 is involved, an equivalent serial execution can not be found. Since Tx2's write on Y is after Tx1's read, Tx3's read on Y is after Tx2's write, and Tx1's write on X is after Tx3's read, it forms a loop.

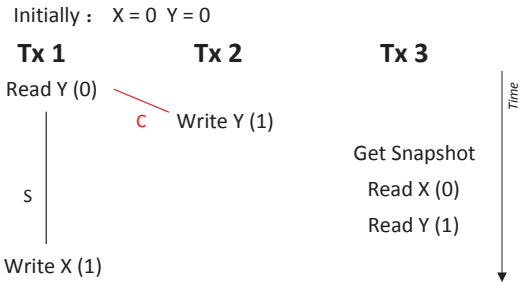


Figure 4: An Anomaly for Read-only Transactions

4.2 Consistency Requirement

Like prior approaches [45, 46], we leverage an epoch-based scheme to support a consistent yet a little stale snapshot for read-only transactions. The basic idea is using a global snapshot number (GSN) which is advanced periodically (e.g., every 20 ms). For each transaction update, a new version will be created if the snapshot number is advanced. Each record is also assigned with a snapshot number to identify which snapshot it belongs to.

However, simply using the epoch-based scheme may still violate the consistency requirement shown below:

- *Atomicity.* The updates from the same transaction should be applied on the same snapshot. This requirement can be preserved by simply enforcing that all the updates from the same transaction are assigned with the same snapshot number.
- *Serializability.* The order of different snapshots should agree with the serial order of committed transactions. For example, if a transaction Tx1 commits its updates on snapshot S_n, then all transactions depending on Tx1 should commit with a snapshot number equal or larger than S_n.

To preserve the second requirement, optimistic concurrency control (OCC) usually buffers the updates locally [45, 8]. The GSN is assigned to a transaction before the transaction commits the buffered updates at the end of the transaction. Such a method can guarantee the atomicity of snapshot assignment and update commitment. However, under transaction chopping, guaranteeing the atomicity becomes more complex as the update needs to be committed at the end of a piece instead of the whole transaction.

Yet, simply acquiring the snapshot number at the beginning of a transaction is also not feasible. In Figure 5, Tx1's second piece conflicts with Tx2's first piece. If both transactions get the snapshot number at the beginning, Tx1 may get a snapshot number (GSN_1) which is smaller than Tx2's (GSN_2). After they apply the updates to the corresponding snapshots, *SN1* gets a newer value for X, while *SN2* gets a newer value for Z. This violates the consistency requirements.

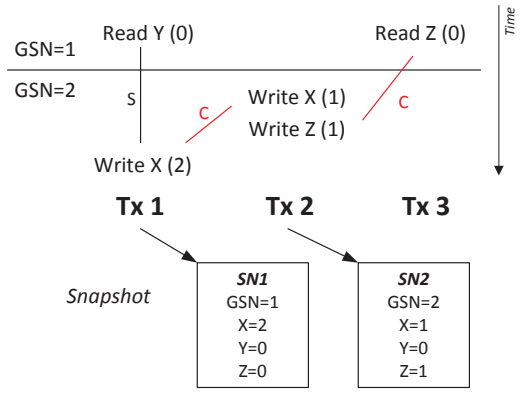


Figure 5: Inconsistent Snapshot caused by acquiring SN at the beginning of a transaction

4.3 Lazy Snapshot Assignment

To tackle the above issues, we leverage a lazy snapshot assignment scheme based on an interesting observation from the chopping graph of chopped transactions:

Theorem 1. *For each transaction, when two instances are used to construct the chopping graph, each transaction has at most one conflicting piece in the chopping graph.*

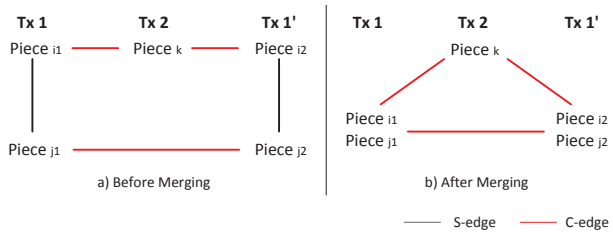


Figure 6: Remove the SC-cycle produced by 2 instances by merging conflicting pieces in one transaction

Proof. Suppose a transaction has more than one conflicting piece. Two of them are $Piece_{i1}$ and $Piece_{j1}$ in the first instance, $Piece_{i2}$ and $Piece_{j2}$ in the second.

As a conflicting piece, $Piece_{i1}$ has a C-edge with another piece $Piece_k$. If $Piece_k$ is $Piece_{i2}$, $Piece_{i1}$ and $Piece_{i2}$ is connected by a C-edge directly. Otherwise, $Piece_{i2}$ has a C-edge with $Piece_k$, since it does the same operation as $Piece_{i1}$ which conflicts with $Piece_k$.

As a result, there is a path from $Piece_{i1}$ to $Piece_{i2}$, composed of C-edges. It is the same with $Piece_{j1}$ and $Piece_{j2}$. As $Piece_{i1}$ and $Piece_{j1}$ are from the same transaction, there is a path connecting them with S-edges. Similarly, there is a path with S-edges for $Piece_{i2}$ and $Piece_{j2}$.

The four paths form an SC-cycle. To remove the SC-cycle, the two conflicting pieces from the same instance should be merged. Hence, each transaction only has one conflicting piece, after merging conflicting pieces iteratively. (Figure 6 shows an example of merging conflicting pieces in one transaction). □

According to Theorem 1, after merging, every transaction (Tx_i) has at most one piece (Cp_i) conflicting with other transactions. This means, during the runtime, if a transaction (Tx_j) conflicts with Tx_i, Tx_j should have exactly one piece (Cp_j) conflicting with Cp_i. Meanwhile, *the execution order of the conflicting pieces (Cp_i, Cp_j) decides the serial order of their parent transactions (Tx_i, Tx_j)*: if Cp_i executes before Cp_j, Tx_i should be serialized before Tx_j. Thus, in Figure 5, even Tx2 starts after Tx1, it should still be serialized before Tx1 according to the execution order of the conflicting pieces.

To preserve the second requirement (serializability) of the snapshot, we lazily assign the snapshot number to a transaction in the conflicting piece. Thanks to the help from RTM, the atomicity of conflicting piece execution and the snapshot assignment can be simply preserved. Since the conflicting piece decides the serializing order, all transactions

serialized before Tx_i cannot get a snapshot number larger than Tx_i's snapshot number. As a result, in Figure 5, both Tx1 and Tx2 will be assigned with *SN2*.

After a transaction is assigned with a snapshot number, the transaction will commit all the updates to the according snapshot. For the updates before the conflicting piece, as they have no conflict with any other transactions, we can keep them in a local buffer before the snapshot number is assigned. The transaction will check if the assigned snapshot number is larger than the snapshot number of the updated records. It will create a new version with the assigned snapshot number if so.

A read-only transaction should read the snapshot on which the read-write transaction will not commit the update afterwards. To ensure this property, each worker thread keeps a local snapshot number (LSN). It indicates the current snapshot number used by the read-write transaction. The read-only transaction can only read the record with the snapshot number smaller than $min(LSN)$.

4.4 Commutative Operations

As an important optimization to eliminate C-edges in a chopping graph, commutative operations also introduce challenges to build a consistent snapshot. This is because the execution order of commutative operations may be different with the serial order. For example, both Tx1 and Tx2 blindly update record A and increase record B. Since the increment is commutative, Tx1 and Tx2 only conflict on the first piece. Figure 7 shows a possible runtime interleaving. Tx2 should be serialized before Tx1 according to the execution order of the blindly write. However, the commutative operations can still happen in a different order: Tx2 will increase B on SN_1 based on the initial value of B (which is zero). As a result, the value of B on SN_2 is 1 instead of 2, which violates the consistency requirement.

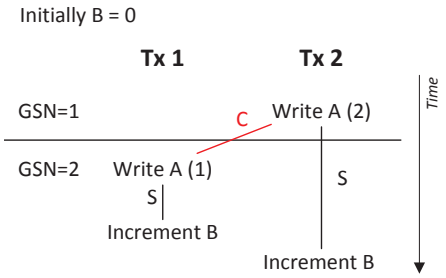


Figure 7: Execution of transactions containing commutative pieces

Based on the observation that the order of the commutative operations does not affect the final result, we employ a *redo* scheme to solve this challenge. The basic idea is, after getting a snapshot number N in the conflicting piece, all commutative operations should be redone on the snapshots whose numbers are larger than N. In Figure 7, Tx2 should apply the update of record B on both SN_1 and SN_2. Then, the final result of record B on SN_2 will be 2.

Algorithm 1: Execution of Commutative Piece

```
Input: Transaction t, Commutative piece p, Record r
//Normal Execution
p.op(r.nosn_version);
//Redo after snapshot number is acquired
curVersion := r.sn_version;
while curVersion.sn > t.sn do
    p.op(curVersion);
    if curVersion.oldVersion == NULL then
        break;
    else
        curVersion := curVersion.oldVersion;
if curVersion.sn < t.sn then
    curVersion := newVersion(t.sn);
p.op(curVersion);
```

To avoid the impact of the redo scheme on the execution of read-write transactions, we isolate the update on the latest value with the update on the snapshot. For each record involved in commutative operations, we keep a version without the snapshot number (i.e., the latest version). This version always has the latest value of the record. The snapshot versions of the record are linked with its latest version. Algorithm 1 shows the basic algorithm of handling commutative operations. For each operation, it will update the latest version of the record which has no snapshot number (Line 2). After the snapshot number is assigned to the transaction, all commutative operations need to be redone on the snapshot whose number is equal or larger than the assigned snapshot number (Line 5 – 16).

4.5 Correctness

This section shows that our snapshot scheme can preserve the serializability of transactions. The theory of transaction chopping ensures the serializability of read-write transactions. Hence, we show that adding read-only transactions continues to preserve the serializability.

$$P_{ro1} \rightarrow P_{rw1} \rightarrow \cdots \rightarrow P_i \rightarrow P_j \rightarrow \cdots \rightarrow P_{rwn} \rightarrow P_{ro2}$$

Figure 8: Happen Before Relationship among Pieces

If read-only transactions break serializability, there must be a dependence chain shown in Figure 8. The arrow between pieces represents read-after-write, write-after-read, or write-after-write relationships between two conflicting pieces. ($A \rightarrow B$ means that A is after B.) For short, a piece's snapshot number is set to the snapshot number of the transaction it belongs to.

The first piece and the last piece are from the same read-only transaction, T_{ro}. $Piece_{ro1}$ reads the result of $Piece_{rw1}$, so the snapshot number of $Piece_{rw1}$, SN_1, is no larger than T_{ro}'s. As $Piece_{ro2}$ reads oldder versions of records than the versions $Piece_{rwn}$ updates, according to the policy which a read-only transaction uses to get the snapshot number, the snapshot number of $Piece_{rwn}$, SN_n, is larger than T_{ro}'s. So, SN_1 is smaller than SN_n. As a result, there must be two adjacent pieces, $Piece_i$ and $Piece_j$, in the chain that $Piece_i$'s snapshot number is smaller than $Piece_j$'s. Namely, $SN_i < SN_j$. Otherwise, we will get $SN_1 \geq SN_n$ by iteration ($SN_1 \geq SN_2 \geq ... \geq SN_n$).

At least one of the two pieces is from a read-write transaction, as pieces from read-only transactions do not conflict with each other.

If $Piece_i$ comes from a read-only transaction, it reads the result of $Piece_j$, which has a larger snapshot number. This violates the process that a read-only transaction acquires records.

If $Piece_j$ comes from a read-only transaction, as it does not read the results from $Piece_i$, SN_i is larger than SN_j, which contradicts the precondition, $SN_i < SN_j$.

As a result, both pieces are from read-write transactions. As the pieces are in conflict, $Piece_j$, which executes earlier than $Piece_i$, should have a snapshot number no larger than $Piece_i$. This leads to a contradiction with the precondition, $SN_i < SN_j$. Hence, our snapshot assignment scheme still preserves the serialzability of the chopped transactions.

5 Implementation

We have implemented our mechanisms in the DBX [46] system (denoted as DBX-TC for convenience). This section describes some key implementation details, including the record structure, range queries and durability.

Record Structure: The record structure is carefully designed to facilitate transaction execution and save memory space.

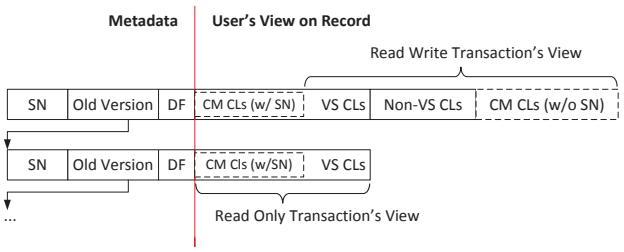


Figure 9: The record structure of DBX-TC (SN : Snapshot Number, DF : Delete Flag, CM : Commutative, CL : Column, VS : Versioned)

Figure 9 shows the structure of a record. Each version of a record has a snapshot number field indicating the snapshot for a particular version, an old version pointer that points to the previous version, and a delete flag indicating whether the record is logically deleted or not. As transactions in DBX-TC access records at the granularity of a column, the columns of a record are divided into two groups according to their types. The versioned columns contain columns that will be accessed by read-only transactions; Other columns are in the non-versioned columns.

Except the newest version, other previous versions do not contain the non-versioned columns, as old versions are only read by read-only transactions. This optimization saves memory used to store old versions. Since the old versions are only read by read-only transactions, there is no need to store them in old versions.

To support consistent snapshots, two copies of commutative columns are used. Read-write transactions apply the commutative operations on a special version of columns without the snapshot number. Hence, a replica of commutative columns(w/o SN) is added to the end of the record. The other replica of commutative columns(w/ SN), which is updated in the redo process, is arranged to the front of the versioned columns.

As a result, the record structure seen by read-only transactions and read-write transactions are different. The record pointer returned to read-only transactions locates at the start of versioned columns, while the record pointer returned to read-write transactions is at the end of commutative columns in the versioned columns. For commutative columns, read-write transactions always operate on the version containing the newest value to ensure the correctness of transaction execution. Read-only transactions get a version that is updated lazily and may be inconsistent before it can be read.

The record structure is carefully designed such that user codes can update the records returned by DBX-TC in-place and be blind to the actual version it operates on.

Range queries: DBX-TC also significantly simplifies the support of range queries. Without HTM, other high-performance in-memory databases resort to fine-grained locks by attaching each record with a lock to prevent concurrency accesses. Range lock which involves records and index locks is necessary to support range queries. The use of RTM in its entirety mostly eliminates such complexity. A single pair of *xbegin* and *xend* provides several levels of atomicity for range queries: the accesses to record's metadata like the snapshot number and the record value are atomic; multiple operations in one piece are also executed atomically. RTM also provides the atomicity of updates in data structures, such as record insertion and search. Hence, this essentially avoids the phantom problem during range queries.

Durability: The durability support is similar to the original DBX. The results of transactions are logged and flushed to disk in batch. Each local thread keeps a log buffer. Transactions' results are written into the buffer after they are committed. When the buffer is full, or the snapshot number of transactions is increased, the buffer is sent to a logger thread. The logger thread remembers the largest snapshot number of buffers it has flushed for each thread. Suppose the smallest among them is SN_{min}, the durable snapshot number, SN_d is set to SN_{min} -1. It means that all transactions with a snapshot number no larger than SN_d are durable.

Since our snapshot support ensures that a transaction's snapshot number is always no larger than the transactions it depends on. The durability support can guarantee that if a transaction's result has been flushed to disk, all results of transactions it depends on have been flushed, which provides consistency.

6 Evaluation

This section presents the evaluation of DBX-TC by comparing it to the counterparts.

6.1 Evaluation Setup

We conducted all experiments on an Intel Haswell i7-4770 processor with the clock rate at 3.4GHz. The machine 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 is equipped with two 128GB SATA-based SSD devices.

From subsection 6.2 to subsection 6.6, we use a standard OLTP benchmark TPC-C [41] with standard mix of transactions. It has five types of transactions, two of which are read-only transactions. The level of contention is configured by adjusting the number of warehouses: 1) low contention: each thread has its local warehouse; 2) medium contention: two threads share one warehouse; 3) high contention: all threads share one warehouse.

In subsection 6.7, we use the Articles benchmark from H-Store [1]. It simulates an on-line article posting and commenting site and has four types of transactions.

By default, 1 million transactions are executed concurrently to evaluate the database throughput.

6.2 Impact of Optimizations on Chopping Graph

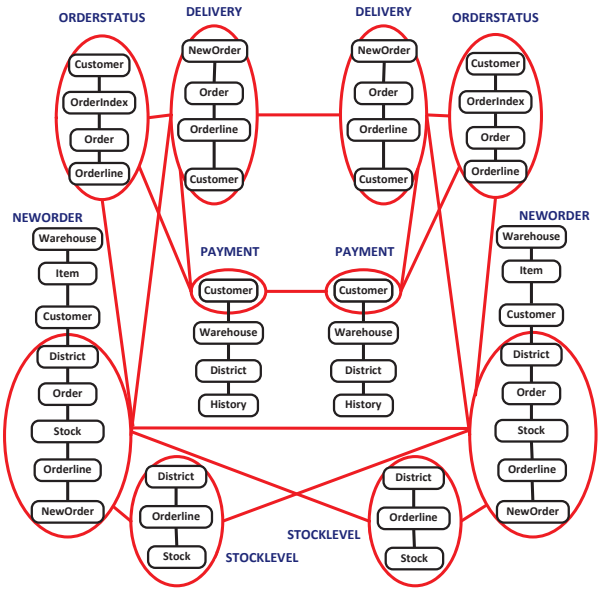


Figure 10: Default Chopping Graph of TPC-C (crossed lines are omitted for clarity)

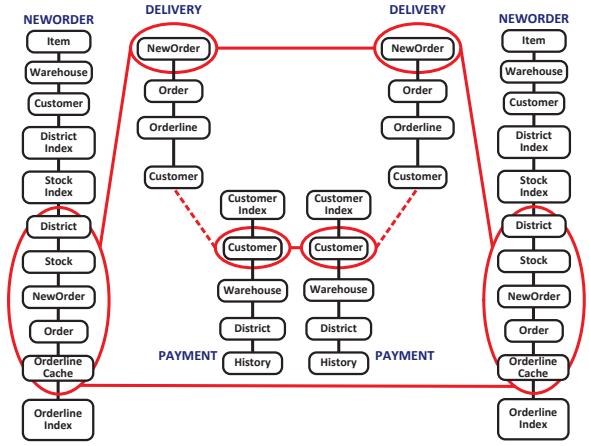
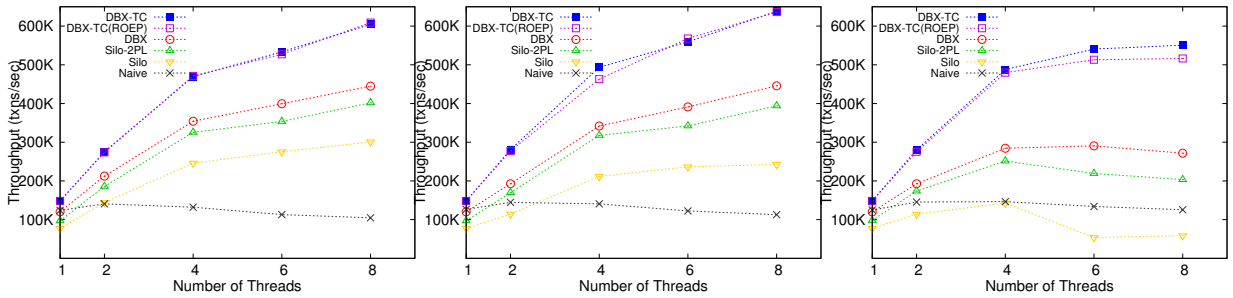


Figure 11: Chopping Graph of TPC-C in DBX-TC (crossed lines are omitted for clarity)

Figure 10 shows the default chopping graph of TPC-C, with both reordering of independent operations and optimization on commutative operations. Two instances of each transaction type are used to construct the chopping graph. The black line represents an S-edge, while the red line represents a C-edge. The dashed line indicates the publisher-subscriber relationship. A conflicting piece is embraced in an oval, which may contain multiple operations due to merging pieces. As shown in Figure 10, the traditional chopping algorithm can chop very few transactions into smaller pieces and thus can hardly reduce the working sets of RTM transactions. However, our mechanism successfully chops transactions into smaller pieces through our optimizations, as shown in Figure 11.

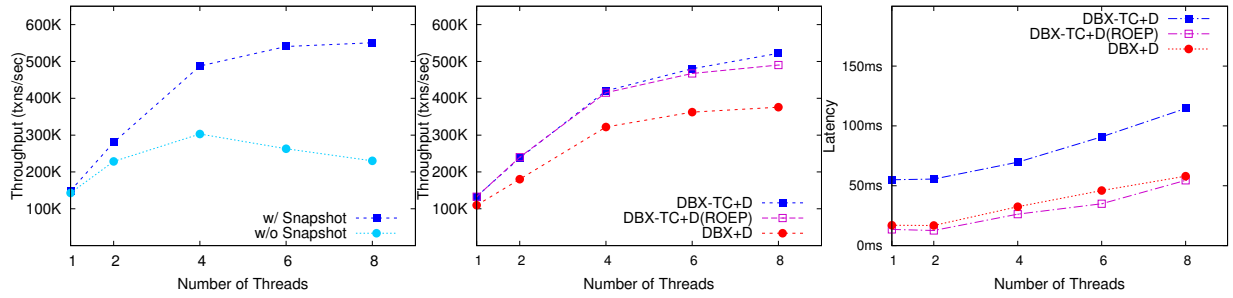
6.3 Performance and Scalability

Figure 12 shows the throughput of DBX-TC under different contention levels, and its comparison with naively applying RTM to transactions, the original DBX, Silo and an implementation of 2PL based on Silo. The original DBX uses an aggressive epoch advancing scheme by letting each read-only transaction advance the epoch as well to avoid read-write transactions updating on the snapshot a read-only transaction is going to read. Though DBX-TC no longer needs such a scheme, we align DBX-TC with the same epoch scheme (called DBX-TC(ROEP)) to make an apple-to-apple



(a) Low Contention *(b) Medium Contention* *(c) High Contention*

Figure 12: Throughput of TPC-C with different contention



(a) Throughput w/ and w/o Snapshot Support *(b) Throughput w/ and w/o Durability Support* *(c) Per-Epoch Latency w/ Durability*

Figure 13: The impact of snapshot and durability Support on throughput and latency of TPC-C (by default that GSN is updated every 20ms. ROEP means that GSN is also updated by read-only transactions)

comparison with DBX. Note that the 2PL implementation should be considered as an optimal one as it currently does not contain deadlock detection, which may further reduce its performance.

The naive approach always shows poor scalability and low performance. Under low contention, DBX-TC achieves the peak throughput of 604,220 txns/sec at 8 threads, which is 5.77 times compared to that of naive applying RTM. This is because the transactions are chopped into smaller pieces, which notably reduces capacity aborts of RTM transactions. Moreover, a large transaction is chopped into multiple pieces, which leads to a higher level of parallelism. It is thus less likely that a conflict arises before a piece is finished and only a small part of a transaction has to be redone in case of conflicts. Actually, the total number of RTM transaction aborts at 8 cores is reduced by 92%.

We also compare DBX-TC with the original DBX. Both DBX-TC and DBX scale well under low and medium contention levels, while DBX-TC outperforms DBX by 36% to 43% at 8 cores. This is because DBX-TC does not need a validation phase in software (which takes 8% execution time of DBX) and the tracking of read set and write set is done by hardware rather than software. Under high contention, the performance of DBX-TC only marginally increases after 6 threads. This is because the machine only has four physical cores and the use of hyper-threading almost halves the working set and increases the probability of aborts. However, DBX-TC still shows 102% speedup compared to DBX, which stops scaling after 4 threads.

Our analysis shows that the user transaction abort rate of DBX is 64% at 8 threads due to excessive conflicts under high contention. In contrast, the RTM transaction abort rate of DBX-TC is 38% with 8 threads under high contention. Moreover, the whole transaction in DBX needs to be redone during a a conflict. In contrast, the use of transaction chopping leads to an early abort mechanism. Further, only pieces are redone during a conflict, which reduces the cost for aborts.

DBX-TC and DBX-TC(ROEP) have similar throughput under both low and medium contentions. Under high contention, DBX-TC(ROEP) performs slightly worse than DBX-TC as faster epoch advancing may lead to faster creation of new versions and thus more aborts. Still, DBX-TC(ROEP) outperforms DBX by 90%.

Silo [45] is a modern in-memory database, which utilizes OCC as DBX does. According to the paper [46], Silo has the overhead of encoding and memory copying of records compared with DBX. However, DBX still outperforms Silo after aligning the implementation [46]. Silo cannot scale well under high contention due to the similar reasons

with DBX.

DBX-TC consistently outperforms Silo-2PL due to two reasons. First, as DBX-TC chops transactions into smaller pieces and thus exposes more parallelism. Second, the tracking of read set and write set as well as conflict detection are done in RTM entirely, leading to higher per-core performance.

6.4 Benefits from Snapshot Support

The throughput of TPC-C with and without snapshot support is shown in Figure 13(a). All threads share one warehouse. Without the snapshot support, DBX-TC stops scaling after 4 threads, as the number of conflict aborts of RTM transactions increases 3.3 times at 8 threads compared with DBX-TC with snapshot support. The STOCKLEVEL read-only transaction has a high possibility of capacity aborts. Once it aborts, an RTM transaction falls back to locking. Other transactions may be aborted as the lock is held, even though there may be no true conflicts. Other transactions will be blocked for a long time as the STOCKLEVEL transaction takes a long time to execute.

6.5 Impact from Durability

Figure 13(b) shows the impact from durability on throughput. The evaluation is done with TPC-C under low contention. DBX-TC and DBX-TC(ROEP) outperform DBX by 39% and 30% on throughput at 8 threads. The overhead of logging is 11-16% and 11-24% on DBX-TC and DBX-TC(ROEP), and 9-18% on DBX. DBX-TC(ROEP) has slightly higher overhead since the epoch is advanced faster than DBX-TC and thus more old versions are created. Compared with DBX-TC with no durability support, the scalability from 6 threads to 8 threads is a little bit worse, since the machine only has 8 hyperthreads and the two logger threads begin to contend with the worker threads.

The evaluation results of per-epoch latency are shown in Figure 13(c). We use the algorithm in Silo [45] to calculate per-epoch latency. Hence, a larger epoch duration may mean higher latency as a transaction may need to wait until an epoch expire before being returned to users.

Hence, the latency of DBX-TC at 8 threads is 114ms, while DBX has a lower latency of 58ms. This is because the epoch is updated relatively slow and thus all transactions need to wait an epoch to finish before they can be returned. In contrast, by aligning the epoch scheme difference, the latency of DBX-TC(ROEP) is 54ms at 8 threads, which is slightly better than DBX.

6.6 Factor Analysis on Optimizations

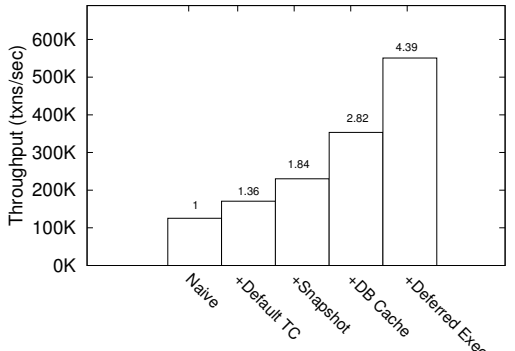


Figure 14: Factor Analysis on DBX-TC

To understand the performance impact of various optimizations on DBX-TC, we present an analysis of multiple factors in Figure 14. The evaluation in done on TPC-C with 8 threads sharing one warehouse to show why DBX-TC can have a high throughput even with high contention.

The baseline refers to naively including the whole user transaction in a RTM transaction. +Default TC means applying the default transaction chopping algorithm with reordering and commutative optimizations. +Snapshot makes read-only transactions read from snapshots. +DB Cache adds a small cache for database operations, so that transactions update to this cache instead of the large underlying database in RTM region. +Deferred Execution converts the DELIVERY transaction's operation on the Customer table into commutative operations.

6.7 Articles Benchmark

To prove DBX-TC also works well for other workload, we tried Articles benchmark derived from H-Store.

Figure 15 shows the evaluation results. Naively applying HTM cannot scale due to large amounts of capacity aborts. However, as it has low contention, all other systems can scale to 8 cores. DBX-TC achieves 475875 txns/sec at 8 threads, which outperforms DBX by about 10% and Silo-2PL by 30%. Since the read write transactions in Articles are relatively small, the acceleration by introducing more concurrency is not so much as TPC-C. However, DBX-TC still has the advantage of simpler execution process.

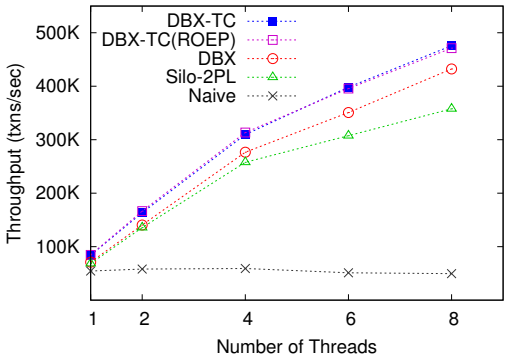


Figure 15: Throughput of Articles Benchmark

7 Related Work

Leveraging HTM for Database: With the emergence of HTM, researchers have started to exploiting HTM for databases. In a position paper, Tran et al. [44] conducted a performance comparison of implementing latches with spinlocks and HTM for a disk-based database. However, the database is a much stripped down one with only 1,000 records and it is not clear whether serializability is guaranteed or not. Wang et al. [46] and Leis [28] both leverage HTM to protect a portion of transactional execution for in-memory databases. Specifically, Wang et al. [46] use RTM to protect the commit phase of a variant of OCC [25], while Leis et al. [28] use hardware lock elision to implement a variant of timestamp ordering. Compared to DBX-TC, both still need to track read/write set and implement conflict detection in software, while DBX-TC completely offload such work to HTM. Hence, according to our evaluation, DBX-TC performs notably better for TPC-C on a Haswell processor.

Transaction Chopping: DBX-TC is made efficient with static analysis and transaction chopping, which was proposed several decades ago [7, 6, 17, 4, 10, 38]. For example, Bernstein et al. [7, 6] describe a conflict graph to statically analyze transaction conflicts such that the orders of transactions are predefined to preserve serializability. Garcia-Molina [16] further shows that there will be a safe interleaving if all pieces of a decomposed transaction commute. Sasha et al. [38] further propose using chopping graph to analyze transactions and show that serializability can be guaranteed when there is no SC-cycle. Zhang et al. [47] and Mu et al. [32] further leverage static analysis on chopping graph to reduce latency and improve parallelism of distributed transactions. DBX-TC is built on the theory of transaction chopping, but with a set of workload-inspired refinements, and more importantly an eventual snapshot mechanism to support read-only transactions.

Concurrency Control: Concurrency control has been intensively studied for decades and there are a number of schemes in the form of 2-phase locking [5], timestamp ordering [9, 28] and commit ordering [35]. Recent research also attempt to reduce or even eliminate concurrency control at its entirety [23, 43, 2]. For example, H-store and its relatives [39, 23] uses a run-to-completion model to execute local transactions without any concurrency control, and relies on distributed transaction protocols or a global lock to handle cross-partition transactions. This, however, makes its performance dependent on whether a database is well partitioned [45]. Calvin and its descendants [43, 42] leverages deterministic lock ordering to preassign distributed transactions to eliminate a commit protocol. Compared to these approaches, DBX-TC leverages a set of techniques to completely offload concurrency control to HTM.

In-memory Databases. There have been numerous efforts optimizing transaction processing for in-memory databases [34, 24, 26, 40, 27, 30, 11, 45, 48, 33]. With the increasing number of cores, researchers have used various approaches to improve multicore scalability by mitigating contentions on locks and latches [21, 22, 36, 20] for 2PL-based databases. Others use fine-grained locks [45] or HTM to protect the commit phase [46]. Due to carefully reducing RTM transaction regions, DBX-TC enjoys better per-core transaction throughput, as demonstrated in this paper.

Read-only Transactions and Snapshot: Generally, many prior designs [12, 37, 13, 29, 24, 15, 45, 46, 48] leverage snapshot isolation [3] to insulate read-only transactions from read-write ones. Fekete et al. [14] point out that when read-write transactions are serializable, read-only transactions reading snapshot may still violate serializability. Hyper [15] leverages the copy-on-write semantics of virtual memory to implement efficient snapshot. DBX-TC departs from prior epoch-based snapshot schemes [45, 46, 48] to support read-only transactions under transaction chopping.

8 Conclusion

This paper tried to answer a research question: whether HTM can substitute concurrency control in database transactions in its entirety? This paper showed that naively doing so resulted in poor performance and scalability. To this end, this paper proposed leveraging the theory of transaction chopping and refining it with workload inspired optimisations to avoid unnecessary conflicts to reduce RTM transaction regions. This paper further described an eventual snapshot assignment scheme that decouples read-only transactions from the chopping graph. With the above optimizations, this paper showed that offloading concurrency control to HTM can result in better performance and scalability than its alternatives on a 4-core Intel Haswell machine.

References

[1] Articles benchmark. http://hstore.cs.brown.edu/documentation /deployment/benchmarks/.

[2] BAILIS, P., FEKETE, A., FRANKLIN, M. J., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. Coordination avoidance in database systems. *Proc. VLDB 8*, 3 (2014).

[3] BERENSON, H., BERNSTEIN, P., GRAY, J., MELTON, J., O'NEIL, E., AND O'NEIL, P. A critique of ansi sql isolation levels. *ACM SIGMOD Record 24*, 2 (1995), 1–10.

[4] BERNSTEIN, A. J., GERSTL, D. S., AND LEWIS, P. M. Concurrency control for step-decomposed transactions. *Information Systems 24*, 8 (1999), 673–698.

[5] BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. *Concurrency control and recovery in database systems*, vol. 370. Addison-wesley New York, 1987.

[6] BERNSTEIN, P. A., AND SHIPMAN, D. W. The correctness of concurrency control mechanisms in a system for distributed databases (sdd-1). *ACM TODS 5*, 1 (1980), 52–68.

[7] BERNSTEIN, P. A., SHIPMAN, D. W., AND ROTHNIE JR, J. B. Concurrency control in a system for distributed databases (SDD-1). *ACM TODS 5*, 1 (1980), 18–51.

[8] CAO, T., VAZ SALLES, M., SOWELL, B., YUE, Y., DEMERS, A., GEHRKE, J., AND WHITE, W. Fast checkpoint recovery algorithms for frequently consistent applications. In *Proc. SIGMOD* (2011).

[9] CAREY, M. J. *Modeling and evaluation of database concurrency control algorithms*. PhD thesis, University of California, Berkeley, 1983.

[10] DAVIES, C. T. Data processing spheres of control. *IBM Systems Journal 17*, 2 (1978), 179–198.

[11] DIACONU, C., FREEDMAN, C., ISMERT, E., LARSON, P.-A., MITTAL, P., STONECIPHER, R., VERMA, N., AND ZWILLING, M. Hekaton: Sql server's memory-optimized oltp engine. In *Proc. SIGMOD* (2013), ACM, pp. 1243–1254.

[12] ELNIKETY, S., PEDONE, F., AND ZWAENEPOEL, W. Generalized snapshot isolation and a prefix-consistent implementation. Tech. rep., 2004.

[13] FEKETE, A., LIAROKAPIS, D., O'NEIL, E., O'NEIL, P., AND SHASHA, D. Making snapshot isolation serializable. *ACM TODS 30*, 2 (2005), 492–528.

[14] FEKETE, A., O'NEIL, E., AND O'NEIL, P. A read-only transaction anomaly under snapshot isolation. *ACM SIGMOD Record 33*, 3 (2004), 12–14.

[15] FUNKE, F., KEMPER, A., NEUMANN, T., ET AL. Hyper-sonic combined transaction and query processing. In *Proc. VLDB* (2011).

[16] GARCIA-MOLINA, H. Using semantic knowledge for transaction processing in a distributed database. *ACM TODS 8*, 2 (1983), 186–213.

[17] GARCIA-MOLINA, H., AND SALEM, K. Sagas. In *Proc. SIGMOD* (1987), ACM.

[18] HARIZOPOULOS, S., ABADI, D. J., MADDEN, S., AND STONEBRAKER, M. Oltp through the looking glass, and what we found there. In *Proc. SIGMOD* (2008), ACM, pp. 981–992.

[19] HERLIHY, M., AND MOSS, J. E. B. Transactional memory: Architectural support for lock-free data structures. In *Proc. ISCA* (1993).

[20] HORIKAWA, T. Latch-free data structures for DBMS: design, implementation, and evaluation. In *Proc. SIGMOD* (2013).

[21] JOHNSON, R., PANDIS, I., HARDAVELLAS, N., AILAMAKI, A., AND FALSAFI, B. Shore-MT: a scalable storage manager for the multicore era. In *Proc. EDBT* (2009), ACM, pp. 24–35.

[22] JUNG, H., HAN, H., FEKETE, A. D., HEISER, G., AND YEOM, H. Y. A scalable lock manager for multicores. In *Proc. SIGMOD* (2013).

[23] KALLMAN, R., KIMURA, H., NATKINS, J., PAVLO, A., RASIN, A., ZDONIK, S., JONES, E. P., MADDEN, S., STONEBRAKER, M., ZHANG, Y., ET AL. H-store: a high-performance, distributed main memory transaction processing system. In *Proc. VLDB* (2008).

[24] KEMPER, A., AND NEUMANN, T. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proc. ICDE* (2011), pp. 195–206.

[25] KUNG, H.-T., AND ROBINSON, J. T. On optimistic methods for concurrency control. *ACM TODS 6*, 2 (1981), 213–226.

[26] LARSON, P.-Å., BLANAS, S., DIACONU, C., FREEDMAN, C., PATEL, J. M., AND ZWILLING, M. High-performance concurrency control mechanisms for main-memory databases. In *Proc. VLDB* (2011).

[27] LEE, J., MUEHLE, M., MAY, N., FAERBER, F., SIKKA, V., PLATTNER, H., KRUEGER, J., AND GRUND, M. High-performance transaction processing in sap hana. *IEEE Data Eng. Bull. 36*, 2 (2013), 28–33.

[28] LEIS, V., KEMPER, A., AND NEUMANN, T. Exploiting Hardware Transactional Memory in Main-Memory Databases. In *Proc. ICDE* (2014).

[29] LIEDES, A.-P., AND WOLSKI, A. Siren: A memory-conserving, snapshot-consistent checkpoint algorithm for in-memory databases. In *Proc. ICDE* (2006), IEEE, pp. 99–99.

[30] LINDSTRÖM, J., RAATIKKA, V., RUUTH, J., SOINI, P., AND VAKKILA, K. Ibm soliddb: In-memory database optimized for extreme speed and availability. *IEEE Data Eng. Bull. 36*, 2 (2013), 14–20.

[31] LIU, R., AND CHEN, H. Ssmalloc: a low-latency, locality-conscious memory allocator with stable performance scalability. In *Proc. APSYS* (2012).

[32] MU, S., CUI, Y., ZHANG, Y., LLOYD, W., AND LI, J. Extracting more concurrency from distributed transactions. In *Proc. OSDI* (2014).

[33] NARULA, N., CUTLER, C., KOHLER, E., AND MORRIS, R. Phase reconciliation for contended in-memory transactions. In *Proc. OSDI* (2014), pp. 511–524.

[34] PANDIS, I., JOHNSON, R., HARDAVELLAS, N., AND AILAMAKI, A. Data-oriented transaction execution. In *Proc. VLDB* (2010).

[35] RAZ, Y. The principle of commitment ordering, or guaranteeing serializability in a heterogeneous environment of multiple autonomous resource managers using atomic commitment. In *Proc. VLDB* (1992), pp. 292–312.

[36] REN, K., THOMSON, A., AND ABADI, D. J. Lightweight locking for main memory database systems. In *Proc. VLDB* (2012).

[37] RIEGEL, T., FELBER, P., AND FETZER, C. A lazy snapshot algorithm with eager validation. In *Distributed Computing*. Springer, 2006, pp. 284–298.

[38] SHASHA, D., LLIRBAT, F., SIMON, E., AND VALDURIEZ, P. Transaction chopping: Algorithms and performance studies. *ACM TODS 20*, 3 (1995), 325–363.

[39] STONEBRAKER, M., MADDEN, S., ABADI, D. J., HARIZOPOULOS, S., HACHEM, N., AND HELLAND, P. The end of an architectural era:(it's time for a complete rewrite). In *Proc. VLDB* (2007).

[40] SUBASU, T., AND ALONSO, J. Database engines on multicores, why parallelize when you can distribute. In *Proc. Eurosys* (2011).

[41] THE TRANSACTION PROCESSING COUNCIL. TPC-CBenchmark (Revision 5.9.0). http://www.tpc.org/tpcc/, 2007.

[42] THOMSON, A., AND ABADI, D. J. Modularity and Scalability in Calvin. *IEEE Data Engineering Bulletin* (2013), 48.

[43] THOMSON, A., DIAMOND, T., WENG, S.-C., REN, K., SHAO, P., AND ABADI, D. J. Calvin: fast distributed transactions for partitioned database systems. In *Proc. SIGMOD* (2012).

[44] TRAN, K. Q., BLANAS, S., AND NAUGHTON, J. F. On Transactional Memory, Spinlocks, and Database Transactions. In *Proc. ADMS* (2010).

[45] TU, S., ZHENG, W., KOHLER, E., LISKOV, B., AND MADDEN, S. Speedy Transactions in Multicore In-Memory Databases. In *Proc. SOSP* (2013).

[46] WANG, Z., QIAN, H., LI, J., AND CHEN, H. Using restricted transactional memory to build a scalable in-memory database. In *Proc. EuroSys* (2014).

[47] ZHANG, Y., POWER, R., ZHOU, S., SOVRAN, Y., AGUILERA, M. K., AND LI, J. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Proc. SOSP* (2013), pp. 276–291.

[48] ZHENG, W., TU, S., KOHLER, E., AND LISKOV, B. Fast databases with fast durability and recovery through multicore parallelism. In *Proc. OSDI* (2014), pp. 465–477.