Scalable In-Memory Transaction Processing with HTM

Yingjun Wu and Kian-Lee Tan, *National University of Singapore*

978-1-931971-30-0

Scalable In-Memory Transaction Processing with HTM

Yingjun Wu and Kian-Lee Tan
School of Computing, National University of Singapore

ABSTRACT

We propose a new HTM-assisted concurrency control protocol, called `HTCC`, that achieves high scalability and robustness when processing OLTP workloads. `HTCC` attains its goal using a two-pronged strategy that exploits the strengths of HTM. First, it distinguishes between hot and cold records, and deals with each type differently – while accesses to highly contended data are protected using conventional fine-grained locks, accesses to cold data are HTM-guarded. This remarkably reduces the database transaction abort rate and exploits HTM's effectiveness in executing low-contention critical sections. Second, to minimize the overhead inherited from successive restarts of aborted database transactions, `HTCC` caches the internal execution states of a transaction for performing *delta-restoration*, which partially updates the maintained read/write set and bypasses redundant index lookups during transaction re-execution at best effort. This approach is greatly facilitated by HTM's speedy hardware mechanism for ensuring atomicity and isolation. We evaluated `HTCC` in a main-memory database prototype running on a 4 socket machine (40 cores in total), and confirmed that `HTCC` can scale near-linearly, yielding high transaction rate even under highly contended workloads.

1 INTRODUCTION

With the introduction of Intel's newly released Haswell processors, hardware transactional memory (HTM) is finally available in mainstream computing machines. As promised in its initial proposal twenty years ago, HTM greatly simplifies the implementation of correct and efficient concurrent programs.

A major target application for exploiting HTM is modern multicore OLTP databases, where sophisticated concurrency control protocols must be designed to guarantee isolation among threads. To benefit from HTM, effective solutions should reduce HTM's high abort rate caused by capacity overflow, which is a major limitation of Intel's current implementation. A promising approach [34] is to apply HTM to optimistic concurrency control (OCC) protocol, where transaction computation is entirely detached from commitment. While achieving high transaction rate under certain scenarios, this solution unfortunately gives rise to unsatisfactory performance when processing highly contended workloads. This is because conventional OCC is intrinsically sensitive to skewed data accesses, and, in fact, protecting OCC's commitment with the speculative HTM can make the protocol even more vulnerable to contentions.

In this paper, we introduce `HTCC`, a new HTM-assisted concurrency control mechanism that achieves robust transaction processing even under highly contended workloads. The design of `HTCC` is inspired by two observations. First, massive volumes of transactions in a contended workload skew their accesses on a very few popular data records, and optimistically accessing these records are likely to result in high abort rate due to data conflicts. Second, an aborted database transaction may be successively re-executed before it is eventually committed, and such re-executions incur the overhead of repeatedly retrieving the same records via index lookups.

Based on these facts, `HTCC` adopts a two-pronged approach that takes full advantage of the strengths of HTM. On the one hand, `HTCC` splits the data into hot and cold records, and a combination of pessimistic fine-grained locking and optimistic HTM is leveraged to protect a transaction's accesses differentially according to the degree of contentions. This mechanism avoids frequent data conflicts on highly contended records, and fully exploits HTM's effectiveness in executing lowly contended critical sections. On the other hand, to minimize the overhead incurred from successive restarts of aborted database transactions, `HTCC` further maintains a thread-local structure, called *workset cache*, to buffer the accessed records of each operation within a transaction. When an aborted transaction is re-executed, `HTCC` performs an efficient *delta-restoration*: records to be read or written during the re-execution are fetched directly from the cache, and unnecessary overhead caused by redundant index lookups is largely reduced. However, this design requires maintenance of a transaction's read/write sets and handling of deadlocks. With HTM, potential deadlocks are now handled by HTM's hardware scheme; this not only simplifies the design but also minimizes the overhead of deadlock resolution (as there is no longer the need to manage deadlocks for cold data explicitly).

We implemented `HTCC` in CAVALIA (source code: `https://github.com/Cavalia/Cavalia`), a main-memory database prototype that is built from the ground up. Our evaluation confirmed that `HTCC` can yield much higher transaction rate on a 4 socket machine (40 cores in total) compared to the state-of-the-art concurrency control protocols, especially under highly contended workloads.

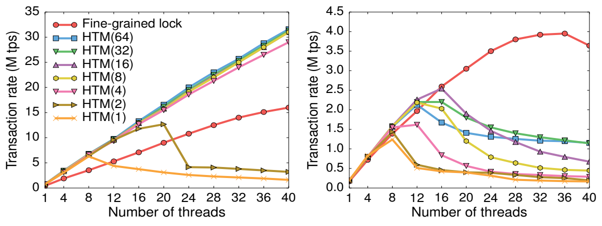
This paper is organized as follows: Section 2 provides a preliminary study to show the design intuition of `HTCC`. Section 3 explains how `HTCC` reduces database transaction abort rate caused by data conflicts, and Section 4 describes how `HTCC` eliminates unnecessary overhead in restarting aborted database transactions. We report extensive experiment results in Section 5. Section 6 reviews related works and Section 7 concludes this work.

2 PRELIMINARY

Hardware transactional memory. Hardware transactional memory (HTM) aims at simplifying the implementation of correct and efficient parallel algorithms. With the release of Intel's Transactional Synchronization Extension (TSX) instructions, programmers nowadays can simply use `xbegin` and `xend` to guard a code segment that should be executed transactionally. Upon finishing an HTM region, all the memory writes encapsulated in this region will appear atomically. TSX instructions also eliminate the need for implementing software mechanisms for deadlock resolution, and its intrinsic hardware-level mechanism efficiently ensures atomicity and isolation with little overhead.

The TSX instructions use the CPU cache to store execution buffers for an HTM region. The cache-coherence protocol is leveraged to track data conflicts at cache-line granularity. Such use of existing features in modern CPU architectures makes it possible to provide low-overhead hardware support for transactional memory. To guarantee forward progress, a fallback routine that acquires coarse-grained exclusive locks will be executed after an HTM region has been aborted for a certain predetermined number of times. We refer to this number as the *restart threshold*.

The major limitation of the current HTM implementation is its high abort rate, which is caused by either capacity overflow or data conflicts. First, the capacity of an HTM region is strictly constrained by both the CPU cache size and the cache associativity, and any violation of such constraint can directly give rise to HTM region abort. Second, HTM executes the protected critical sections in a speculative manner, and two concurrent HTM regions updating to the same memory address can result in either region being aborted. While the capacity-overflow problem can be addressed through a careful design of critical sections, data conflicts among dozens of threads can severely restrict the HTM scalability, making it less effective as a general mechanism to support a wide spectrum of concurrent program workloads. Besides the costly restart overhead, frequent aborts severely hurt HTM's performance because of the well-known *lemming effect* [8]. As the repeated aborts of a single HTM region eventually lead to the acquisition of a global coarse-grained lock, all other HTM regions cannot proceed until the lock is released. This consequently forces all the concurrent HTM regions to abort, resulting in fully serialized execution without parallelism.



(a) Low-contention workload. (b) High-contention workload.

Figure 1: Processing multi-key transactions. Please note the difference in y-axis ranges.

Figure 1 compares HTM with conventional fine-grained locking mechanism when processing multi-key transactions on a standard hash map with 1 million data records. Each transaction randomly updates 10 records, and the access contention is controlled by a parameter θ, indicating the skewness of the Zipfian distribution. The fine-grained locking scheme acquires locks associated with all the targeted records and prevents deadlocks by pre-ordering its write sets; such an approach is also used in the validation phase of traditional OCC protocol [32]. The restart threshold for HTM regions is varied from 1 (see `HTM(1)`) to 64 (see `HTM(64)`). Figure 1a shows that, by setting the restart threshold to larger than 4, HTM achieves 2X better performance (31 vs 16 M tps) when processing low-contention workloads ($\theta = 0$) with 40 threads. However, the results exhibited in Figure 1b indicate that the transaction rate achieved by HTM deteriorates significantly under highly contended workloads ($\theta = 0.8$). In particular, with restart threshold set to 64, HTM reaches a transaction rate of 1.1 M tps with 40 threads enabled. This number is remarkably smaller than 3.7 M tps, which is attained by fine-grained locking.

Given the experimental results presented above, we confirm that Intel's HTM implementation is more suitable for protecting lowly contended data accesses. Fine-grained locking, in comparison, is still more attractive for executing high-contention workloads.

Optimistic concurrency control. Modern multicore OLTP database is a major target for applying HTM. However, the capacity limit of Intel's HTM implementation prevents a one-to-one mapping from database transactions to HTM regions. A promising solution [34] to address this problem is to apply HTM to conventional OCC protocol [15], which splits the execution of a database transaction into three phases: (1) a *read* phase, which executes all the read and write operations in the transaction according to the program logic without any blocking; (2) a *validation* phase, which checks the transaction conflicts by certifying the validity of the transaction's read set; and (3) a *write* phase, which installs all the transaction writes to the database atomically. Fine-grained locks for all the records that are accessed by the transaction must be acquired on entering the validation phase and be released when terminating the write phase. A transaction T_1 will be aborted if it fails the validation phase. Such a failure indicates that a certain committed concurrent transaction T_2 has modified the values of certain records in T_1's read set during T_1's read phase. We refer to the time period during T_1's read phase as *vulnerable window*, where inconsistent accesses can occur.

To improve the performance of conventional OCC, existing solution [34] adopts HTM to completely replace the fine-grained locks that are used during the transaction's validation and write phases. However, as HTM executes guarded code segment speculatively, the validation and the write phases consequently become vulnerable to contention. More specifically, any committed updates from a concurrent transaction T_2 can abort a transaction T_1 even if T_1 is within the validation and the write phases that are protected by HTM region, as HTM does not forbid any concurrent accesses. As a result, the vulnerable window in such a HTM-assisted OCC protocol can span across the whole database transaction. Figure 2 depicts the difference between conventional OCC [15] and existing HTM-assisted OCC [34].

The characteristic of the existing HTM-assisted OCC protocol makes it unattractive for processing highly contended workloads. Therefore, we design a new concurrency control protocol that fully exploits the power of HTM to achieve robust transaction processing.

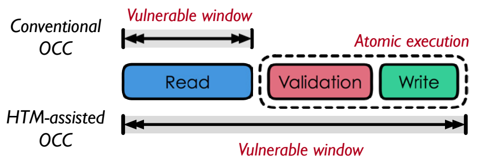


Figure 2: Conventional OCC v.s. HTM-assisted OCC.

3 REDUCING ABORT RATE

HTCC is an HTM-assisted OCC-style protocol that can achieve high transaction rate by leveraging a combination of fine-grained locking and HTM. The key intuitions are that contended OLTP workloads skew their accesses on a small portion of data records in the database, and HTM yields a remarkably higher performance when protecting low-contention critical sections. This section describes how HTCC fully exploits the benefits of HTM and reduces the likelihood of database transaction aborts.

3.1 Data Record Classification

HTCC splits the data into hot and cold records so as to choose the best scheme for locking each single record that is accessed by a certain database transaction. As depicted in Figure 3, HTCC maintains five metadata fields for each data record: (1) a *lock* flag showing the record's lock status; (2) a *ts* attribute recording the commit timestamp of the last writer; (3) a *vis* flag indicating whether the record is visible to inflight transactions; (4) a *hot* flag specifying whether the record is highly contended; and (5) a *cnt* attribute counting the number of validation failures triggered by the conflicting accesses to the record during a certain time slot.

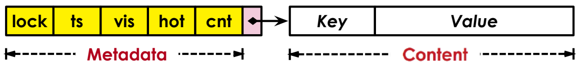


Figure 3: Structure of a data record.

The *vis* flag of a record has three possible states: `public` state indicating that the record is visible to all the inflight transactions; `private` state indicating that the record is newly inserted by an uncommitted transaction; and `dead` state indicating that the corresponding record has been deleted by a committed transaction. During transaction processing, HTCC automatically tracks a transaction's conflicting accesses and atomically increments the *cnt* field for the record to which the access raises the validation failure. To achieve this, the *cnt* field is packed into a single word to allow CAS operation. A background thread is spawned to periodically (e.g., 60 seconds) check the failure counts and mark the most contended records by flipping the associated *hot* flag. A record that is no longer frequently accessed during a constant time duration will be tagged back to cold. The flip of a *hot* flag is performed transactionally, and every in-flight transaction will always see a consistent *hot* flag of each record from its start to its final commitment. That is, the background thread can change a record's *hot* flag only if no concurrent transaction is accessing this record.

3.2 Hybrid Transaction Processing

Like OCC, HTCC divides the execution of a transaction into read, validation, and write phases. However, HTCC adopts a hybrid synchronization mechanism to execute a single transaction: on the one hand, pessimistic fine-grained locking is employed to protect the accesses to hot records; on the other hand, HTM region is leveraged to guard the accesses to cold records. For each transaction, HTCC maintains a thread-local read/write set (rw-set) to store information about the records read or written by the transaction. Each element in the rw-set keeps an array with five basic fields: $\langle record, data, ts, hot, type \rangle$, where *record* stores the pointer to the original record in the database, *data* buffers a local copy of the accessed record's content, *ts* identifies the timestamp value of the record at the time it was accessed by the transaction, *hot* indicates whether the accessed record is contended or not, and *type* shows the access type (read (R), write (W), or both (RW)) to the record. When accessing a hot record in a transaction, HTCC performs updates directly to the original record in the database, and the locally buffered copy is used for content recovery should the database transaction be aborted. However, when accessing a cold record, updates performed by the transaction will be temporarily held in the local copies before they are being installed to the original records. The following subsections describe the three phases of HTCC in detail.

Read phase. HTCC speculatively accesses cold records during a transaction's read phase. The function AccessRecord(r, T) in Figure 4 depicts how an operation with access type T on record r is performed in the read phase. To access a record r in a transaction, HTCC first checks r's *hot* flag and acquires the associated read or write lock if this flag is set to true. Subsequently, a new element e in the rw-set is created for r. All the fields, including *record*, *data*, *ts*, *hot*, and *type* are atomically stored in e. The *type* field in e is updated to T to indicate the access type (i.e., read, write, or both).

The acquisition of fine-grained locks for hot records during the read phase can cause deadlocks. Therefore, certain deadlock-resolution scheme must be employed to guarantee forward progress. Our current implementation adopts a no-wait-style strategy [5], which aborts a database transaction if a certain lock cannot be obtained within a certain time range.

```
ACCESSRECORD(r, T):
  // =========== LOCK HOT RECORDS ==========
  if (r.hot == true) then
    r.acquire_lock();
  e = rw_set.insert(r);
  e.set(r, r.data, r.ts, r.hot, T);

COMMIT():                              RTM-guarded region
  //  ============ VALIDATION ==============
  do
    is_success = true;

    HTM_BEGIN(); // begin HTM txn
    foreach e in rw_set do
      if e.hot == false then
        if e.type == R || e.type == RW then
          if e.ts != e.record.ts then
            HTM_END(); // end HTM txn
            Repair transaction;
            is_success = false;
            break;
  while (is_success == false);

  commit_ts = generate_commit_ts();

  // ======== UPDATE COLD RECORDS ==========
  foreach e in rw_set do
    if e.hot == false then
      if e.type == W || e.type == RW then
        install(e.record, e.data);
        e.record.ts = commit_ts;
  HTM_END(); // end HTM txn
  // ======== UPDATE HOT RECORDS ===========
  foreach e in rw_set do
    if e.hot == true then
      if e.type == W || e.type == RW then
        install(e.record, e.data);
        e.record.ts = commit_ts;

  // ======== UNLOCK HOT RECORDS ===========
  foreach e in rw_set do
    if e.hot == true then
      e.record.release_lock();
```

Figure 4: The framework of HTCC.

Validation phase. As HTCC touches cold records in a consistency-oblivious manner through the read phase, a validation phase must be invoked subsequently to check the validity of every cold record that is read by this transaction. Similar with the existing HTM-assisted OCC protocol [34], HTCC leverages HTM region to protect the atomic execution of the validation phase.

As shown in the function Commit() in Figure 4, HTCC begins its validation phase by informing the hardware to start executing in transactional mode with xbegin instruction. After that, HTCC iterates over the rw-set S and checks if any cold record read by the current transaction has been modified by any committed concurrent transaction. The record value loaded at the read phase is determined to be *invalid* if the corresponding element e in S stores a timestamp ts that is different from that attached in the original record (i.e., $e.record.ts$).

Such an invalidation is made by the *inconsistent* read operation that first touches this record. On encountering any invalid element in the rw-set, HTCC will *repair* the transaction instead of re-executing it from scratch. We defer the discussion on the repairing process to the next section. The validation phase terminates when the transaction eventually reads all the records consistently or aborts due to deadlocks (for hot records).

Write phase. A transaction that has passed the validation phase will enter the write phase to make every update visible to all the concurrent transactions. The write phase of HTCC is also described in Figure 4. Under the protection of HTM region, HTCC installs the locally buffered contents back to every accessed cold record, with a commit timestamp *commit_ts* attached to identify the partial dependencies among transactions [32]. In particular, pointer swap [34] is used for local-copy installation, as this mechanism minimizes the HTM region capacity. On exiting the HTM region, the timestamp *commit_ts* is written to all the updated hot records, and fine-grained locks are finally released to make every hot record visible to all other inflight transactions.

The protocol described above leverages the utmost of two distinct synchronization mechanisms: accesses to hot records are protected by fine-grained locking, which yields satisfactory performance when processing high-contention workloads; accesses to cold records are guarded by HTM region, which achieves much higher throughput when executing lowly contended workloads. The next section continues to explore the feature of HTM that further accelerates HTCC's performance.

4 MINIMIZING RESTART OVERHEAD

While the proposed hybrid processing mechanism can effectively reduce the database transaction abort rate, it may still result in expensive re-execution overhead if an inflight database transaction is blindly rejected once validation failure occurs. We develop a *transaction repair* mechanism that greatly benefits from HTM's guarantee of atomicity and isolation. By keeping track of the accesses of each operation at runtime in a *workset cache*, HTCC re-executes a failed transaction only by performing *delta-restoration*, which effectively reutilizes the contents in the cache to speed up the restoration of each operation in the transaction. Thanks to the use of HTM, the accesses to cold records during the repair stage are still processed optimistically, bringing little overhead to the system runtime.

4.1 Workset Cache

The processing cost of a database transaction is dominated by the time to retrieve data records from the corresponding database indexes. On detecting an invalid element in the rw-set during validation, a database transaction has to be (successively) aborted and re-executed from scratch, fetching the same records through index lookups multiple times. The overhead caused by these redundant index lookups can be largely reduced by caching the accessed records in thread-local data structures during the initial execution of a transaction.

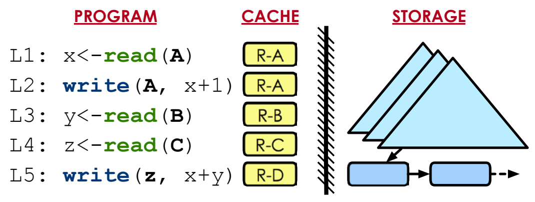


Figure 5: Workset cache for a transaction.

Based on this intuition, HTCC maintains a *workset cache*, which is constructed during the read phase, that buffers a list of record pointers for all the records that are accessed by each operation in a transaction. In particular, we refer to the set of records accessed by a certain operation as the operation's *workset*. Figure 5 shows such a caching structure maintained for a running transaction. The initial execution of the read operation L_1 fetches a data record *R-A* using candidate key *A* and caches the pointer to *R-A* in the corresponding entry of the workset cache. Similarly, the write operation L_5 loads the record *R-D* through the index and buffers its pointer into the workset cache before performing real updates. Should the database transaction be aborted, the subsequent re-execution can tap on the record pointers from the workset cache to minimize index traversal. This approach saves a significant amount of efforts for committing transactions that are prone to abort. Note that for a range operation that accesses multiple records, its workset needs to maintain all the pointers to its targeted records.

4.2 Delta-Restoration

With the workset cache maintained for each transaction, HTCC repairs an aborted database transaction with *delta-restoration*, which re-utilizes the internal states created during the transaction execution without restarting the transaction from scratch. The main idea of delta-restoration is that, by leveraging the maintained workset cache, most of the index lookups can be bypassed by directly accessing the buffered record pointers, and only a small portion of the re-set needs to be updated, and the incurred overhead to ensure atomicity and isolation can be largely reduced through HTM's support.

As shown in Figure 4, on confronting a validation failure of a transaction, HTCC directly exits the HTM region and starts *repairing* the entire transaction, during which stage all the operations in a transaction will be *restored* at best effort. Figure 6 describes how HTCC restores an operation *opt* with access type *T* (i.e., read, write, or both) in a transaction with two distinct schemes: *fast-restoration* and *slow-restoration*.

```
RESTORE(opt, T):
  if is_key_invariant(opt) == false then
  // =========== FAST-RESTORATION ==========
    W = get_workset(opt);
    perform(opt, W); // perform read or write
  else
  // =========== SLOW-RESTORATION ==========
    V = index_lookup(opt);
    W = get_workset(opt);
    set_workset(opt, V);                    Update rw-set
    ins_set, del_set = compute_delta(V, W);
    foreach del in del_set do
      if del.hot == true then
        del.release_lock();
      rw_set.erase(del);
    foreach ins in ins_set do
      if ins.hot == true then
        ins.acquire_lock(); // deadlock?
      e = rw_set.insert(ins);
      e.set(ins, ins.data, ins.ts, ins.hot, T);

    perform(opt, V); // perform read or write
```

Figure 6: Restore a single operation in a transaction.

Fast-restoration. This scheme re-executes an operation in the transaction without invoking any index lookups. Instead, all the targeted records are directly fetched from the workset cache constructed at runtime. In particular, fast-restoration only works for an operation *opt* if the accessing key has not been updated, which essentially indicates that the corresponding records to be accessed by this operation should be identical to those maintained in the workset cache.

Slow-restoration. This scheme is specifically designed for an operation whose accessing key has been changed due to the restoration of its parent operations. Compared with fast-restoration, slow-restoring an operation *opt* is more expensive, as the targeted records that should be accessed by *opt* have to be re-fetched through index lookup. Consequently, the corresponding workset maintained for *opt* in the caching structure must also be refreshed. As shown in Figure 6, after retrieving all the records required by *opt* through index lookup using a candidate key *key*, the slow-restoration refreshes its workset cache by updating the *opt*'s workset from *W* to *V*. Subsequently, the difference between *W* and *V* is computed. In particular, the delta-set *ins_set* contains all the new records that must be inserted into the rw-set, and the delta-set *del_set* contains all the records that should be removed from the rw-set.

Figure 7 illustrates the two types of restoration mechanisms using the sample transaction shown in Figure 5. During the repair of the transaction, the restoration of L_2 directly fetches the pointer *R-A* from its local cache without resorting to the index lookup. However, the accessing key of L_5 is dependent on the output of L_4, which can be modified during L_4's restoration. Once a change in accessing key is detected, the restoration of L_5 has to traverse the index and update the workset before performing the real write.

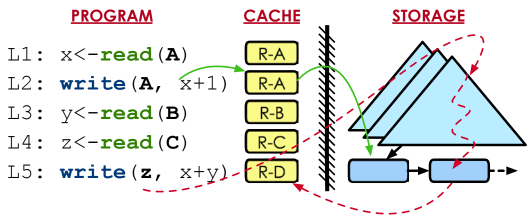


Figure 7: Fast-restoration (solid line) and slow-restoration (dashed line) of an operation.

Delta-restoration, however, is not for free, as the update of the rw-set has to resort to certain synchronization mechanism for protecting records that are newly inserted into the rw-set. In particular, deadlock can occur if fine-grained locking is utilized. HTM's deadlock-free scheme to protect complex critical sections in contrast provides an elegant and efficient way for tackling this problem. Throughout the repair stage, HTCC keeps processing cold records in a consistency-oblivious manner, meaning that no fine-grained locks are acquired to guarantee atomicity. The consistency checking of these newly inserted cold records are deferred until the next round of validation, where HTM region is leveraged to guard the critical section. This design effectively offloads the overhead brought by deadlock resolution to the hardware mechanism, and the software fine-grained locking mechanism is only used for protecting hot records. When updating the rw-set, all the locks associated with the hot records maintained in the delta-set *del_set* should be released during slow-restoration. Similarly, every newly fetched hot record in *ins_set* must be locked before adding them into the rw-set *rw_set*.

HTCC returns back to the validation phase once all the operations within a transaction has been restored. It should be noted that for transactions containing conditional branches, some operations may no longer be executed due to the change of decisions made in those conditional statements. As a result, HTCC further needs to remove all the elements that are no longer accessed by the transaction from the rw-set before validating it again. A transaction can proceed to commit only if all the elements in the rw-set have been successfully validated.

To conclude, the use of HTM has contributed to the efficiency of this repair mechanism and ultimately the superiority of HTCC. The gain in performance comes from two sources. First, there is no need to maintain locks for cold data. This significantly reduces the overhead of the repair mechanism as the number of cold data is much larger than that of hot data. Second, the hardware support provided for HTM to ensure atomicity and isolation of the HTM regions (the part of the transactions that access the cold data) naturally offers superior performance (as compared to a software-based scheme).

4.3 Operation Dependency Extraction

While workset caching can reduce the overhead of transaction re-execution, an invalidation failure will trigger restoration of all the operations within the transaction. As an inconsistent read usually affects only a limited portion of the transaction program, performance can be improved if the code segment to be re-executed can be minimized. In fact, an understanding of the dependencies across operations can determine the subset of operations to be restored when invalidation occurs. HTCC therefore extracts two types of dependencies from the program at compile time: (1) *key dependency*, which occurs when the accessing key of an operation depends on the result of its previous operation; and (2) *value dependency*, which occurs when the non-key value to be used by an operation depends on the result of its previous operation. Figure 8 depicts the dependency relations within the transaction shown in Figure 5. Operations L_2 and L_5 are value-dependent on L_1, as L_1 assigns the variable x that will be used as non-key value in both L_2 and L_5. Operation L_5 is also key-dependent on its previous read operation L_4, which generates the accessing key z for L_5.

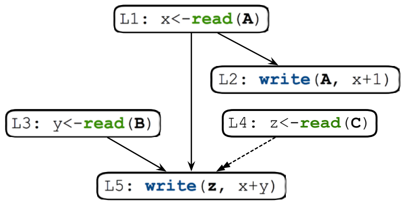


Figure 8: Dependency relations within a transaction.

To identify the read operation that produces the invalid element in the rw-set, HTCC adds an *operation* field to each element. Such a field tracks the operation that first reads the corresponding record, and will be further used for identifying all the affected operations that must be re-executed for preserving correctness.

Figure 9 shows how the extracted dependencies facilitates the repair of a transaction. Given an invalid element *e* that causes the validation failure, the repair mechanism starts by identifying the operation *opt* that performs the first read to *e*. Since *opt* is the initial inconsistent operation, all its descendant operations must be restored to guarantee serializability. Correct restoration mechanism

```
REPAIR(e):
  opt = e.operation;
  Restore(opt); // restore operation
  DS = extract_dependent_set(opt);
  PS = DS;
  while PS.is_empty() == false do
    opt = PS.pop_front();
    Restore(opt); // restore operation
    RS = extract_dependent_set(opt);
    PS.append_set(RS);
```

Figure 9: Repair a transaction with dependency relations.

is selected for each inconsistent operation by checking its dependency relations with its parent operations. After restoring an operation *opt*, all its child operations in the dependent-set *RS* will be inserted into the restoration set *PS*. The repair mechanism terminates once all the potentially inconsistent operations have been restored. Subsequently, HTCC restarts the validation phase, which eventually terminates if all the elements in the transaction's read set pass the validation phase.

In the current version of our system, all the transactions are implemented in C++. We leveraged LLVM Pass framework [1] to extract dependencies from a transaction program at compile time. While the classic interprocedural data-flow analysis techniques [27] can precisely capture operation relations for programs containing function calls, we currently do not yet support dependency extraction for nested transactions, where a transaction can be invoked in the process of another transaction. We leave the support of nested transactions as a future work.

4.4 Extended Database Operations

This section depicts how HTCC supports inserts, deletes, and range queries without sacrificing serializability.

Inserts. To insert a record R in a transaction T_1, HTCC performs an early insertion to hold a place for R in the database during T_1's read phase. The to-be-inserted record R is added to T_1's rw-set, with its *vis* flag set to `private`, indicating that R is not yet ready for access. The *vis* flag is updated to `public` only if T_1 has been successfully committed. If a concurrent transaction T_2 attempts to perform the same insertion, only one transaction can succeed. Should T_1 commit before T_2, T_2 will detect this conflict during the validation phase and trigger repair mechanism to handle failed insertion.

Deletes. A transaction marks the deletion of an existing record R by setting R's *vis* flag to `dead`. A garbage collection thread periodically reclaims all the deleted records that are still maintained in the database. A deleted record R can be safely reclaimed if R is no longer referred to in any running transaction's rw-set.

Range queries. Range queries can lead to phantom problem [9]. Instead of resorting to the classic *next-*

key locking mechanism [23] for tackling such problem, HTCC leverages a solution that is first introduced by Silo [32]. HTCC maintains both the version number and the leaf pointers in a transaction's rw-set. Any structural modification caused by concurrent inserts or deletes will be detected by checking the version's validity. Once an invalidity is discovered, HTCC restores any inconsistent operations by invoking the repair mechanism.

5 SERIALIZABILITY

In this section, we give an informal argument on why HTCC is capable to enforce full serializability.

First, HTCC's a combination of fine-grained locking and HTM yields serializable results. We reduce the proposed scheme to two-phase locking (2PL) protocol. Let us consider a transaction T accessing two records R_1 and R_2. If R_1 and R_2 are both tagged as hot records, then HTCC locks both records using fine-grained locking scheme, which is equivalent to 2PL, hence database serializability is guaranteed. If R_1 and R_2 are both tagged as cold records, then HTCC leverages HTM region to protect the atomic access to them. This scheme is serializable, as is proved in the previous work [34]. If R_1 is a hot record and R_2 is a cold record, then HTCC acquires the lock on R_1 before entering the HTM region where R_2's atomic access is protected. On passing the validation, HTCC exits the HTM region and eventually releases the lock on R_1. During this stage, no new lock is acquired, and hence the protocol obeys the constraints set in the 2PL scheme. Therefore, this protocol is serializable.

Second, HTCC's repair mechanism in the validation phase does not compromise database serializability. HTCC executes the repair mechanism in a consistency-oblivious manner, that is, no synchronization schemes are applied during this stage. However, once the repair mechanism completes, HTCC falls back to the very beginning of the validation phase, and all the records read by the transaction will be validated within an HTM region. Hence, the serializability is guaranteed.

To conclude, HTCC yields serializable results when processing transactions.

6 EXPERIMENTS

In this section, we evaluate the performance of HTCC. All the experiments are performed in a main-memory database prototype, called CAVALIA, that is implemented from the ground up in C++.

We deployed CAVALIA on a multicore machine running Ubuntu 14.04 with four 10-core Intel Xeon Processor E7-4820 clocked at 1.9 GHz, yielding a total of 40 cores. Each core owns a private 32 KB L1 cache and a private 256 KB L2 cache. Every 10 cores share a 25 MB L3 cache and a 64 GB local DRAM. We have enabled hyper-threading, and the machine provides 80 hardware threads in total. Note that due to hyper-threading, the experiments with more than 40 threads may yield sub-linear scalability, as we shall see later in this section. We compare HTCC with a collection of concurrency control protocols, as listed below:

2PL: The classic two-phase locking with wait-die strategy adopted for resolving deadlocks [5].

OCC: The classic OCC protocol with scalable timestamp generation mechanism applied [32].

SOCC: A variation of conventional OCC protocol implemented in Silo [32].

HOCC. An HTM-assisted OCC protocol proposed by Wang et al [34].

HTO. An HTM-assisted timestamp ordering protocol proposed by Leis et al [17].

All the protocols in comparison are implemented in CAVALIA. Similar with the original implementation of Silo, our SOCC implementation also adopts the epoch-based timestamp generation mechanism and eliminates the need of tracking anti-dependency relations. However, instead of using Masstree [22] as the underlying index, we leveraged libcuckoo [19] for indexing primary keys. The implementation of HOCC is adapted from that of OCC. Following the description of Wang et al. [34], we directly protected the validation phase using HTM region and adopted pointer swap to minimize the likelihood of hardware transaction abort caused by capacity overflow. Our implementation of HTO basically follows the original paper [17], but we ignored the proposed storage-layer optimizations, as the storage layout in a database system should be orthogonal to the upper-layer concurrency control protocol. We use the TPC-C benchmark [2] to evaluate the performance.The workload contention is controlled by changing the number of warehouses, and decreasing this number can increase the contention. In our experiments, HTCC's automatic scheme (see Section 3.1) is applied for identifying hot records.

Existing bottlenecks. We begin our experiments with a detailed analysis on the state-of-the-art HTM-assisted concurrency control protocol [34]. We first measure the performance of HOCC to see whether this protocol is robust to workload contentions. Figure 10 shows the transaction rate achieved by the HOCC protocol with the HTM region's restart threshold increased from 1 (HOCC(1)) to 64 (HOCC(64)). The number of warehouses is set to 40, yielding a low data contention. As the result indicates, with the restart threshold set to 64, HOCC can successfully scale on 80 threads, achieving a transaction rate at over 2.1 M transactions per second (tps). This number is 15% higher than that achieved by OCC, showing that

HTM can provide a more efficient scheme for executing low-contention critical sections.

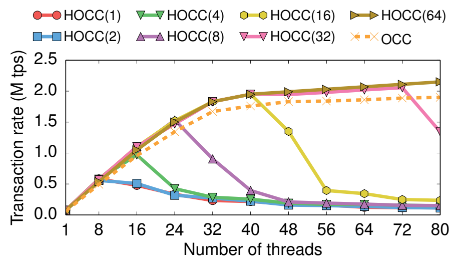


Figure 10: Performance of HOCC with different restart thresholds. The number of warehouses is set to 40.

While the previous work [34] as well as our experiment shown above confirmed the efficiency of the existing HTM-assisted protocol under low-contention workloads, the performance can drop drastically when processing highly contended workloads. Figure 11 shows the experiment result. By setting the number of warehouses to 4, the performance of HOCC deteriorates drastically regardless of the restart threshold. In particular, with the thread count set to 80, HOCC only yields around 200 K tps, while OCC under the same scenario achieves over 550 K tps, exhibiting a much better scalability. This is mainly because the speculative execution of HTM can give rise to high abort rate, and frequent aborts subsequently result in costly restart overhead as well as the lemming effect, which eventually leads to pure serialized execution without parallelism. This result is also consistent with that reported by Makreshanski et al [21].

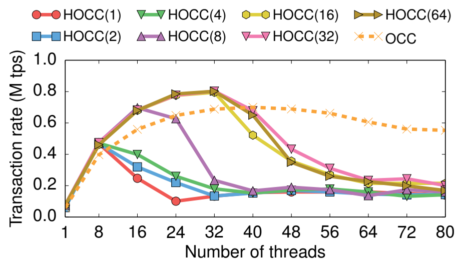


Figure 11: Performance of HOCC with different restart thresholds. The number of warehouses is set to 4.

An interesting observation in the experiment above is that, as shown in Figure 12, the database transaction abort rate[1] generated by HOCC drops with the increase of thread count. With the restart threshold set to 64, HOCC yields an abort rate of 0.45, which is much smaller than that attained with 56 threads enabled (1.18). This result is different from that achieved by OCC, which suffers higher contention with more threads accessing the same records. This observation is still a consequence of the lemming effect, which results in pure serialized execution of database transactions, and therefore fewer transactions get aborted.

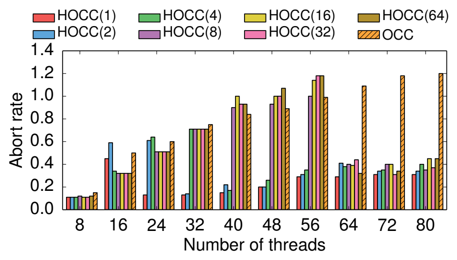


Figure 12: Database transaction abort rate of HOCC. The number of warehouses is set to 4.

To conclude, the state-of-the-art HTM-assisted OCC protocol may not be an attractive solution for supporting contended OLTP workloads.

Transaction rate. Next, we compare HTCC with several other concurrency control protocols. We set the restart threshold of all the HTM-assisted protocols to 64.

The following experiment evaluates the performance of each protocol under low-contention workloads. Figure 13 shows the result with the number of warehouses set to 40. All the four OCC-style protocols, including HTCC, OCC, SOCC, and HOCC, scale near-linearly when the thread count is increased from 1 to 40. With over 40 threads enabled, the performance stablizes, and stops improving. As explained earlier, this is because the hyper-threading scheme can produce sub-linear scalability, hence constraining the performance of the tested protocols. Thanks to the efficiency of HTM and workset caching, HTCC yields over 10% higher transaction rate than OCC. Under the same scenario, the HTM-assisted HOCC only produces a similar performance with that of SOCC. Despite the absence of hardware transactional support, SOCC still generates a very competitive result, as it fully eliminates the necessity for tracking anti-dependency relations. Compared with these OCC-style protocols, 2PL achieves a lower transaction rate. The main reason is that it requires a longer locking duration for every record read or written by a transaction, and hence the overall performance is bounded. As a variation of timestamp ordering (TO) protocol, HTO still suffers high overhead for maintaining internal data structures [41], consequently leading to unsatisfactory results, even if HTM is utilized.

Next, we compare the performance of these protocols with the number of warehouses set to 4. In this scenario, the workload is highly contended. Figure 14 shows the transaction rate of each protocol with the thread count

[1]The abort rate equals to the ratio the number of transaction re-executions to the number of committed transactions.

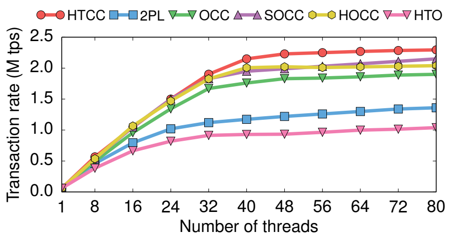


Figure 13: Performance comparison with the number of warehouses set to 40.

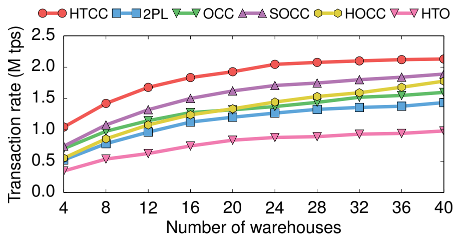


Figure 15: Performance comparison with the number of threads set to 40.

varied from 1 to 80. With 80 threads enabled, HTCC can process around 1,000 K transactions per second (tps), and this number is respectively 2 times and 6 times higher than that attained by SOCC and HOCC. OCC yields similar performance compared to 2PL, but the overhead for tracking anti-dependency makes it 10% slower than SOCC. The low performance of HOCC and HTO also indicate that the state-of-the-art HTM-assisted protocol cannot scale well under high-contention workloads.

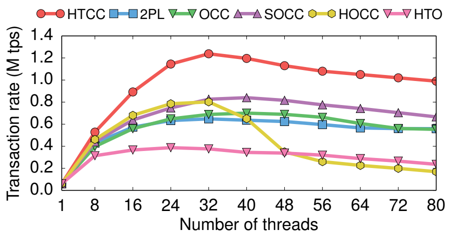


Figure 14: Performance comparison with the number of warehouses set to 4.

To further validate the scalability of HTCC, we measure the transaction rate achieved by each protocol with the number of warehouses changed from 4 to 40. Figure 15 shows the result with 40 threads enabled. Compared with all the other protocols, HTCC yields a superior transaction rate even under highly contended workloads. Specifically, the performance of HTCC is constantly 1.2 to 2 times higher than that achieved by SOCC. While HOCC produces good performance when the number of warehouses is set to 40, its unoptimized use of HTM makes it vulnerable to workload contentions.

All the experiment results reported above have confirmed that HTCC achieves much better performance than the existing (HTM-assisted) concurrency control protocols, especially under highly contended workloads.

Transaction latency. The following experiment measures the latency achieved by each OCC-style protocol. We omit the measurement for 2PL and HTO, as our previous experiments have already shown that these two protocols cannot attain high performance under various types of workloads. Figure 16 shows the corresponding results with 40 threads enabled. The number of warehouses is set to 4. Compared with the other three protocols, HTCC incurs a comparatively lower latency when processing NewOrder transactions. Specifically, 89% of the transactions can reach the final commitment within 160 µs. This number is much larger than that achieved by OCC (78%) and SOCC (73%). This is because HTCC attempts to repair any invalid accesses instead of restarting the entire transaction when an validation failure occurs. This approach essentially saves the resources that have been allocated for executing the transaction. In contrast to the other three protocols, HOCC gives rise to significantly higher transaction latency, and only 37% of the transactions can commit within 640 µs. The key reason is that the unoptimized use of HTM makes HOCC extremely vulnerable to conflicting accesses, and the lemming effect caused by frequent aborts eventually leads to pure serialized execution without parallelism, making the resultant transaction latency intolerable.

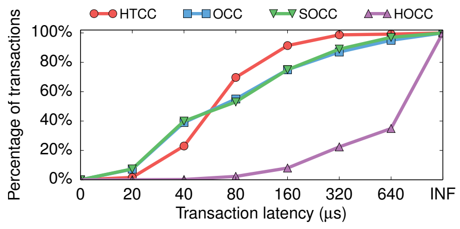


Figure 16: Comparison of transaction latency. The number of warehouses is set to 4.

Performance breakdown. HTCC attains robust transaction processing due to the use of a hybrid synchronization scheme, a workset-caching strategy, and a static-analysis-facilitated optimization. We next examine how these mechanisms contribute to HTCC's performance.

We first study the effectiveness of the hybrid synchronization mechanism. As shown in Table 1, when processing transactions on a database with 4 warehouses using OCC, 62.9% and 31.0% of the database transaction aborts are respectively caused by the conflicting accesses to the records in the WAREHOUSE and the DISTRICT tables. Therefore, applying fine-grained locks on these records can potentially reduce the performance degradation caused by high abort rate. Keeping this observation in mind, we disable HTCC's automatic scheme for data classification and manually select different sets of hot records and measure the influence to HTCC's transaction rate. Figure 17 shows the result with the number of warehouses set to 4. When all the records are labeled as cold (i.e., HTM region is utilized to guard atomic accesses to the targeted data records), the performance of HTCC drops drastically once the thread count reaches 32 (see HTCC (None)). However, if all the records in the WAREHOUSE table are labeled as hot data, the performance degradation is remarkably mitigated (see HTCC (W)). In particular, HTCC attains a comparatively high performance with every record in the WAREHOUSE and the DISTRICT tables tagged as hot (see HTCC (W+D)). These results directly confirmed the effectiveness of fine-grained locking in protecting conflicting records. The same figure also reports HTCC's performance when all the records in the database are labeled as hot (see HTCC (All)). In this case, the effects of HTCC is indeed equivalent to that of 2PL. As shown in Figure 17, the resulting performance is still much lower than that of HTCC (W+D), and this essentially indicates the superior performance of HTM region when executing low-contention critical sections.

Table	4 WHs	10 WHs	40 WHs
WAREHOUSE	62.9%	56.6%	46.8%
DISTRICT	31.0%	37.2%	42.6%
Others	6.1%	6.2%	10.6%

Table 1: Data contentions in each table with 80 threads.

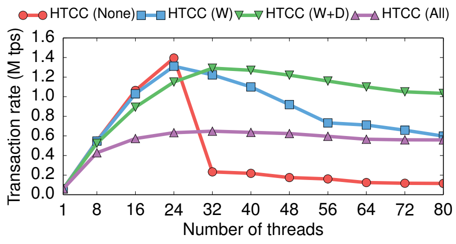


Figure 17: Effectiveness of hybrid concurrency control scheme. The number of warehouses is set to 4.

Figure 18 provides a performance analysis on how the workset caching strategy as well as the static-analysis-facilitated optimization benefit the transaction processing. The number of warehouses in this experiment is still set to 4, generating a highly contended workload. We measure the transaction rate produced by: (1) HTCC, (2) HTCC with static-analysis optimization disabled (see HTCC(-S)), (3) HTCC with both the workset caching and the static-analysis optimization disabled (see HTCC(-SC)), and (4) OCC. In this experiment, we have enabled hybrid processing scheme. Please note that HTCC(-SC) essentially tests the performance purely achieved by the hybrid processing scheme. As shown in Figure 18, the performance gaps among HTCC, HTCC(-S), HTCC(-SC), and OCC widen with the increase of thread count. When scaling to 40 threads, HTCC yields a transaction rate at around 1.2 M tps, which is respectively over 10% and 40% higher than that of HTCC(-S) and HTCC(-SC). This confirmed the effectiveness of both the workset caching and static-analysis-facilitated optimization in HTCC. The same experiment also shows that HTCC(-SC) achieves around 20% higher performance compared to OCC, and this further reaffirmed the efficiency of the hybrid concurrency control scheme in HTCC.

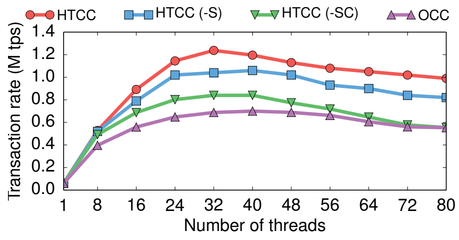


Figure 18: Effectiveness of workset caching and static-analysis-facilitated optimization. The number of warehouses is set to 4.

The delta-restoration mechanism adopted by HTCC incurs a certain amount of overhead to the system runtime. To quantify the overhead, we compare HTCC with HTCC(NV), a variation of HTCC that commits a transaction without invoking HTCC's validation phase. It should be noted that, due to the absence of the validation phase, HTCC(NV) will yield inconsistent query result. However, as it fully bypasses HTCC's overhead in restoring inconsistent operations, it essentially provides a higher bound on the performance that HTCC can potentially achieve. Figure 19 shows the performance comparison between HTCC and HTCC(NV). The number of threads is set to 40. As the number of warehouses increases, the performance of HTCC becomes closer to that of HTCC(NV). This is because increasing the number of warehouses will decrease the workload contention, and a transaction processed under lower contention is less likely to fail the validation phase. With the number of warehouses set to 4, HTCC(NV) can yield over 40% higher transaction rate, which reflects the overhead of operation restoration.

To sum up, the hybrid synchronization mechanism, the workset caching strategy, and the static-analysis-

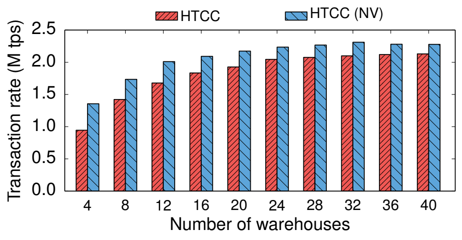


Figure 19: Overhead of delta-restoration. The number of threads is set to 40.

facilitated optimization scheme altogether provide a remarkable performance boost for HTCC.

7 RELATED WORK

Transaction processing. Extensive research efforts have been invested into the development of main-memory database systems. A pioneering work is Silo [32], a shared-everything database that attains high transaction rate using a variation of the conventional OCC protocol [15]. Researchers from Microsoft recently analyzed the performance of multi-version OCC [16, 18], and their study further lays the foundation for the Hekaton database [7]. However, neither Silo nor Hekaton is robust to contended workload, and Yu et al. [41]'s thorough scalability evaluation of traditional concurrency control protocols on 1000 cores further confirmed this point. Several recent works have proposed partitioning scheme to achieve higher scalability on multicore architectures [28]. Both DORA [25] and PLP [26], which are built on top of Shore-MT [12], partition data among cores to reduce long lock waiting time on a centralized lock manager. Several databases [13, 14, 31] employ a deterministic execution model that achieves high scalability using partition-level locks. While these databases can yield high transaction rate when the workload is well partitioned, the performance may severely degrade with an increasing number of distributed transactions. Leveraging program semantics can boost the performance of OLTP databases. Several works [24, 37, 42] leveraged transaction chopping [29] to speed up transaction processing and failure recovery. While HTCC optimizes its performance by using a similar static-analysis mechanism adopted in Wu et al. [36], its HTM-assisted hybrid protocol and caching mechanism can be directly applied to any ad-hoc transactions.

Transactional memory. Transaction memory is well studied in recent years [10, 11]. Sonmez et al. [30] proposed a mechanism that allows STM to dynamically select the best scheme for individual variables. While their design is similar with the hybrid protocol introduced in this paper, their work is restricted in the field of STM and did not consider the overhead caused by re-executing aborted transactions. Xiang et al. [39] also observed a high abort rate of HTM transactions and presented a consistency-oblivious (i.e., OCC-like) solution [3, 4] for reducing the HTM abort rate caused by capacity overflow, Their following work [38] further mitigated the conflict-caused abort problem using advisory lock. Litz et al. [20] borrowed the idea of snapshot isolation from the database community to reduce the abort rate. Different from these works, HTCC relies on a hybrid protocol and a lightweight caching mechanism to reduce the abort rate as well as the incurred abort overhead. Several recent works have exploited HTM to improve the performance of OLTP databases. Yoo et al. [40] utilized Intel's TSX to build efficient indexes, and Makreshanski et al. [21] further studied the interplay of HTM and lock-free indexing methods. Wang et al. [33] also employed HTM to build a concurrent skiplist. These studies on concurrent database indexes revealed that high abort rate due to capacity overflow and data contention can severely restrict HTM's performance. To deal with the high abort rate caused by HTM's capacity overflow, Leis et al. [17] and Wang et al. [34] respectively modified the timestamp ordering and OCC protocols to fully explore HTM's benefits in atomic execution. While achieving satisfactory performance when processing low-contention workloads, neither of them is able to sustain high transaction rate if the workload is contended. Wei et al. [35] and Chen et al. [6] exploited HTM and RDMA to build speedy distributed OLTP databases. As a departure from these works, HTCC focuses on exploiting the benefits of HTM for scalable and robust transaction processing on multicores.

8 CONCLUSION

We have proposed HTCC, an HTM-assisted concurrency control protocol that aims at providing scalable and robust in-memory transaction processing. HTCC attains its goal by reducing the transaction abort rate and minimizing the overhead of restarting aborted transactions. Our experiments confirmed that HTCC can yield high performance even under highly contended workloads. As a future work, we will explore how HTM can be leveraged to improve the performance of modern multi-version database systems.

Acknowledgment

We sincerely thank our shepherd Tim Harris and the anonymous reviewers for their insightful suggestions. This research is supported in part by a MOE/NUS grant R-252-000-586-112.

References

[1] http://llvm.org/docs/passes.html.

[2] http://www.tpc.org/tpcc/.

[3] AFEK, Y., AVNI, H., AND SHAVIT, N. Towards Consistency Oblivious Programming. In *OPODIS* (2011).

[4] AVNI, H., AND KUSZMAUL, B. C. Improving HTM Scaling with Consistency-Oblivious Programming. In *TRANSACT* (2014).

[5] BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. *Concurrency Control and Recovery in Database Systems*. 1987.

[6] CHEN, Y., WEI, X., SHI, J., CHEN, R., AND CHEN, H. Fast and General Distributed Transactions using RDMA and HTM. In *EuroSys* (2016).

[7] 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 *SIGMOD* (2013).

[8] DICE, D., HERLIHY, M., LEA, D., LEV, Y., LUCHANGCO, V., MESARD, W., MOIR, M., MOORE, K., AND NUSSBAUM, D. Applications of the Adaptive Transactional Memory Test Platform. In *TRANSACT* (2008).

[9] ESWARAN, K. P., GRAY, J. N., LORIE, R. A., AND TRAIGER, I. L. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM 19*, 11 (1976).

[10] HARRIS, T., LARUS, J., AND RAJWAR, R. Transactional Memory. Morgan and Claypool Publishers.

[11] HARRIS, T., MARLOW, S., PEYTON-JONES, S., AND HERLIHY, M. Composable Memory Transactions. In *PPoPP* (2005).

[12] JOHNSON, R., PANDIS, I., HARDAVELLAS, N., AILAMAKI, A., AND FALSAFI, B. Shore-MT: A Scalable Storage Manager for the Multicore Era. In *EDBT* (2009).

[13] KALLMAN, R., KIMURA, H., NATKINS, J., PAVLO, A., RASIN, A., ZDONIK, S., JONES, E. P. C., MADDEN, S., STONEBRAKER, M., ZHANG, Y., HUGG, J., AND ABADI, D. J. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. In *VLDB* (2008).

[14] KEMPER, A., AND NEUMANN, T. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *ICDE* (2011).

[15] KUNG, H.-T., AND ROBINSON, J. T. On Optimistic Methods for Concurrency Control. *TODS 6*, 2 (1981).

[16] 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 *VLDB* (2011).

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

[18] LEVANDOSKI, J., LOMET, D., SENGUPTA, S., STUTSMAN, R., AND WANG, R. Multi-Version Range Concurrency Control in Deuteronomy. In *VLDB* (2015).

[19] LI, X., ANDERSEN, D. G., KAMINSKY, M., AND FREEDMAN, M. J. Algorithmic improvements for fast concurrent cuckoo hashing. In *EuroSys* (2014).

[20] LITZ, H., CHERITON, D., FIROOZSHAHIAN, A., AZIZI, O., AND STEVENSON, J. P. SI-TM: Reducing Transactional Memory Abort Rates through Snapshot Isolation. In *ASPLOS* (2014).

[21] MAKRESHANSKI, D., LEVANDOSKI, J., AND STUTSMAN, R. To Lock, Swap, or Elide: On the Interplay of Hardware Transactional Memory and Lock-Free Indexing. In *VLDB* (2015).

[22] MAO, Y., KOHLER, E., AND MORRIS, R. T. Cache craftiness for fast multicore key-value storage. In *EuroSys* (2012).

[23] MOHAN, C. ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes. In *VLDB* (1990).

[24] MU, S., CUI, Y., ZHANG, Y., LLOYD, W., AND LI, J. Extracting More Concurrency from Distribted Transactions. In *OSDI* (2014).

[25] PANDIS, I., JOHNSON, R., HARDAVELLAS, N., AND AILAMAKI, A. Data-Oriented Transaction Execution. In *VLDB* (2010).

[26] PANDIS, I., TÖZÜN, P., JOHNSON, R., AND AILAMAKI, A. PLP: Page Latch-Free Shared-Everything OLTP. In *VLDB* (2011).

[27] REPS, T., HORWITZ, S., AND SAGIV, M. Precise interprocedural dataflow analysis via graph reachability. In *POPL* (1995).

[28] SALOMIE, T.-I., SUBASU, I. E., GICEVA, J., AND ALONSO, G. Database Engines on Multicores, Why Parallelize When You Can Distribute? In *EuroSys* (2011).

[29] SHASHA, D., LLIRBAT, F., SIMON, E., AND VALDURIEZ, P. Transaction Chopping: Algorithms and Performance Studies. *TODS 20*, 3 (1995).

[30] SÖNMEZ, N., HARRIS, T., CRISTAL, A., ÜNSAL, O. S., AND VALERO, M. Taking the Heat Off Transactions: Dynamic Selection of Pessimistic Concurrency Control. In *IPDPS* (2009).

[31] THOMSON, A., DIAMOND, T., WENG, S.-C., REN, K., SHAO, P., AND ABADI, D. J. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *SIGMOD* (2012).

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

[33] WANG, Z., QIAN, H., CHEN, H., AND LI, J. Opportunities and Pitfalls of Multi-Core Scaling using Hardware Transaction Memory. In *APSys* (2013).

[34] WANG, Z., QIAN, H., LI, J., AND CHEN, H. Using Restricted Transactional Memory to Build a Scalable In-Memory Database. In *EuroSys* (2014).

[35] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast In-memory Transaction Processing using RDMA and RTM. In *SOSP* (2015).

[36] WU, Y., CHAN, C.-Y., AND TAN, K.-L. Transaction Healing: Scaling Optimistic Concurrency Control on Multicores. In *SIGMOD* (2016).

[37] WU, Y., GUO, W., CHAN, C.-Y., AND TAN, K.-L. Parallel Database Recovery for Multicore Main-Memory Databases. In *CoRR* (2016).

[38] XIANG, L., AND SCOTT, M. L. Conflict Reduction in Hardware Transactions Using Advisory Locks. In *SPAA* (2015).

[39] XIANG, L., AND SCOTT, M. L. Software Partitioning of Hardware Transactions. In *PPoPP* (2015).

[40] YOO, R. M., HUGHES, C. J., LAI, K., AND RAJWAR, R. Performance Evaluation of Intel® Transactional Synchronization Extensions for High-Performance Computing. In *SC* (2013).

[41] YU, X., BEZERRA, G., PAVLO, A., DEVADAS, S., AND STONEBRAKER, M. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. In *VLDB* (2014).

[42] 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 *SOSP* (2013).