Transaction Healing: Scaling Optimistic Concurrency Control on Multicores

Yingjun Wu, Chee-Yong Chan, Kian-Lee Tan
School of Computing, National University of Singapore
{yingjun, chancy, tankl}@comp.nus.edu.sg

ABSTRACT

Today's main-memory databases can support very high transaction rate for OLTP applications. However, when a large number of concurrent transactions contend on the same data records, the system performance can deteriorate significantly. This is especially the case when scaling transaction processing with optimistic concurrency control (OCC) on multicore machines. In this paper, we propose a new concurrency-control mechanism, called *transaction healing*, that exploits program semantics to scale the conventional OCC towards dozens of cores even under highly contended workloads. Transaction healing captures the dependencies across operations within a transaction prior to its execution. Instead of blindly rejecting a transaction once its validation fails, the proposed mechanism judiciously restores any non-serializable operation and heals inconsistent transaction states as well as query results according to the extracted dependencies. Transaction healing can partially update the membership of read/write sets when processing dependent transactions. Such overhead, however, is largely reduced by carefully avoiding false aborts and rearranging validation orders. We implemented the idea of transaction healing in THEDB, a main-memory database prototype that provides full ACID guarantee with a scalable commit protocol. By evaluating THEDB on a 48-core machine with two widely-used benchmarks, we confirm that transaction healing can scale near-linearly, yielding significantly higher transaction rate than the state-of-the-art OCC implementations.

1. INTRODUCTION

Optimistic concurrency control (OCC) is gaining popularity in the development of modern main-memory databases that target at supporting OLTP transactions on multicore machines [21, 35, 42, 55]. By clearly detaching the computation of a transaction from its commitment, OCC greatly shortens its lock-holding duration and therefore yields very high transaction rate when processing low-contention workloads.

Unfortunately, such performance benefits diminish for workloads with significant data contention, where multiple concurrent transactions access the same data record with at least one transaction modifying the record. A transaction using OCC protocol has to validate the consistency of its read set before commitment in order to ensure that no other committed concurrent transaction has modified any record that is read by the current transaction. If a transaction fails the validation, the transaction has to be aborted and restarted from scratch. Moreover, any partial work done prior to the abort will be discarded, wasting the resources that have been put into running the transaction. Such performance penalty can be exacerbated if the accessed records are highly contended, forcing a transaction to repeatedly abort and restart.

In this paper, we present *transaction healing*, a new concurrency-control mechanism that scales the conventional OCC towards dozens of cores even under highly contended workloads. The key observation that inspires our proposal is the fact that most OLTP applications contend on a few hot records [37], and the majority of transactions failing OCC's validation phase is due to the inconsistency of a very small portion of its read set. By exploiting program semantics of the transactions, expensive transaction aborts-and-restarts can be prevented by restoring only those *non-serializable operations*, whose *side effects*, i.e., the value returned by a read operation or the update performed by a write operation, are (indirectly) affected by a certain inconsistent read. Subsequently, inconsistent transaction states as well as query results can be healed without resorting to the expensive abort-and-restart mechanism. This approach significantly improves the resource-utilization rate in transaction processing, yielding superior performance for any type of workloads.

A key design decision in transaction healing is to maintain a thread-local structure, called *access cache*, to track the runtime behavior of each operation within a transaction. This structure facilitates the operation restoration in transaction healing from two aspects. First, the recorded side effects of each operation can be re-utilized to shorten the critical path for healing transaction inconsistencies; second, the cached memory addresses of the accessed records can be leveraged to eliminate any unnecessary index lookups for accessing targeted records. In particular, the maintenance of this data structure is very lightweight, which is confirmed by our experimental studies.

Transaction healing can partially update the membership of read-/write sets when processing dependent transactions[1]. Observing the high expense brought by such update, transaction healing avoids unnecessary overhead by carefully analyzing the false invalidation. While membership update can result in transaction abort due to deadlock prevention, our proposed schema-based optimization mechanism leverages the access patterns within the database applications to greatly reduce the likelihood of deadlock occurrences.

Different from the state-of-the-art OCC optimization techniques

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-3531-7/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2882903.2915202

[1]A dependent transaction is a transaction where its read/write set cannot be determined from a static analysis of the transaction [53].

that address scalability bottlenecks caused by redundant serial-execution points [21, 35, 55], the emphasis of transaction healing is to reduce the high cost of aborts-and-restarts from data contentions. This essentially renders OCC effective for a wider spectrum of OLTP workloads. The design of transaction healing is also a departure from existing hybrid OCC schemes [28, 52, 60]. Instead of executing restarted transactions with lock-based protocols, transaction healing attempts to re-utilize the execution results without restarting the invalidated transactions from scratch.

We implemented transaction healing in THEDB, a main-memory database prototype built from the ground up. Results of an extensive experimental study on two popular benchmarks, TPC-C and Smallbank, confirmed THEDB's remarkable performance especially under highly contended workloads.

This paper is organized as follows: Section 2 demonstrates transaction healing through a running example. Section 3 introduces the static analysis mechanism and Section 4 describes the runtime execution of transaction healing. We report extensive experiment results in Section 5. Section 6 reviews related works and Section 7 concludes this work.

2. TRANSACTION HEALING OVERVIEW

Transaction healing aims at scaling the conventional optimistic concurrency control (OCC) towards dozens of cores even under highly contended workloads. Inheriting the success of the state-of-the-art OCC protocols [21, 35, 55] in eliminating redundant serial-execution points, transaction healing further strengthens OCC's capability in tackling data conflicts by exploiting program semantics.

Optimistic concurrency control. The conventional OCC proposed by Kung and Robinson [34] splits the execution of a transaction into three phases: (1) a read phase, which tracks the transaction's read/write set using a thread-local data structure; (2) a validation phase, which certifies the consistency of its read set; and (3) a write phase, which installs all its updates atomically. While the detachment between computation and commitment shortens the lock-holding time during execution, the absence of lock protection in the read phase can compromise the consistency of an uncommitted transaction if certain record in its read set is modified by any committed concurrent transaction. Conventional OCC tackles such problem with a straightforward abort-and-restart strategy once inconsistency is detected. We illustrate the mechanism with a running example depicted in Figure 1a[2].

```
 1. PROCEDURE Transfer(srcId){
 2.    dstId<-read(Client, srcId)
 3.    srcVal<-read(Balance, srcId)
 4.    dstVal<-read(Balance, dstId)
 5.    tmp<-0.01*srcVal
 6.    write(Balance, srcId, srcVal-tmp)
 7.    write(Balance, dstId, dstVal+tmp)
 8.    bonus<-read(Bonus, srcId)
 9.    write(Bonus, srcId, bonus+1)
10. }
```

(a) Stored procedure.

Table: Client	
Amy	*Dan*

Table: Balance	
Amy	*2,000*
Dan	*1,200*

Table: Bonus	
Amy	*18*

(b) Initial database state.

Figure 1: Bank-transfer example.

Given the initial database state shown in Figure 1b, a transaction T_1 issued with argument Amy first assigns `dstId` with the value Dan (Line 2) and then transfers \$20 to Dan's Balance account (Lines 3-7). Finally, \$1 is returned back to Amy's Bonus account (Lines 8-9). During the validation phase, T_1 will be determined as inconsistent if a concurrent transaction T_2 gets committed with Amy's balance modified from \$2,000 to, say, \$2,500. In this scenario, abort-and-restart mechanism is applied to ensure a serializable execution of T_1. Unfortunately, such a scheme can severely degrade the system performance if a certain data record is intensively updated, causing invalidated transactions to be repeatedly restarted from scratch.

Transaction Healing. Confronting the pros and cons of conventional OCC, transaction healing leverages program semantics to remedy OCC's weakness in addressing data conflicts. This is achieved with the help of static analysis at compile time that extracts operation dependencies hidden within the stored procedures.

Figure 2 compares the runtime execution of transaction healing with that of conventional OCC. Instead of directly rejecting an invalidated transaction, transaction healing resorts to an additional healing phase to handle any detected inconsistency by restoring the transaction's non-serializable operations. Given an invalidated transaction T, a read/write operation o in T is defined to be a *non-serializable operation* if the outcome of o would be different when T is re-executed. The healing phase aims to re-utilize as many of an invalidated transaction's execution results as possible to heal its inconsistent transaction state as well as its query results according to the extracted dependencies. The forward progress of any in-flight transaction is guaranteed by the design principle of transaction healing, as will be elaborated further in the following sections.

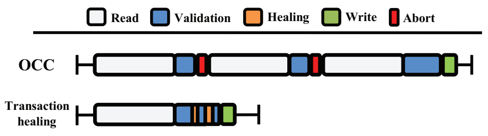


Figure 2: A comparison between OCC and transaction healing.

As an illustration, we discuss how transaction healing addresses the data conflicts exhibited in the running example with minimal execution overhead. Transaction healing maintains a thread-local access cache to track the behavior of every operation that is executed by T_1. On detecting the modification of Amy's balance during the validation phase of T_1, the operations in Line 3 and Lines 6-7 (see Figure 1a) are determined to be non-serializable. This is because the operation in Line 3 assigns Amy's balance to `srcVal`, which is subsequently used in Lines 6-7. Transaction healing therefore directly corrects the side effects made by these three operations without restarting the whole transaction. This strategy works as the maintained access cache records the runtime behavior of every operation in the transaction, and the results generated by those serializable operations can still be reused. Meanwhile, the invoked operation restoration does not trigger any expensive index lookups, since all the records that are read or written by the corresponding operation are logged in the access cache. Hence, the system overhead is greatly reduced.

THEDB overview. We implemented transaction healing in a main-memory database prototype called THEDB, which is specifically designed for modern multicore architecture. THEDB is designed to optimize the execution of transactions that are issued from *stored procedures* and it provides full support for ad-hoc queries. THEDB maintains locks with a per-record strategy. For each database record, THEDB maintains the following three metadata fields: (1) a *timestamp* field indicating the commit timestamp of the last transaction that writes the record; (2) a *visibility* bit in-

[2]For simplicity, we respectively abstract the *read* and *write* operations in a stored procedure as `var←read(Tab, key)` and `write(Tab, key, val)`. Both operations search records in table `Tab` using the accessing key called `key`. The read operation assigns the retrieved value to a local variable `var`, while the write operation updates the corresponding value to `val`.

dicating whether the record is visible to other transactions[3]; and (3) a *lock* bit indicating the lock status of the record. As we shall see, these additional fields enable an efficient implementation of the healing phase in THEDB.

In the following sections, we formalize the mechanism of transaction healing and show how this proposed technique improves the performance of THEDB without bringing costly runtime overhead.

3. STATIC ANALYSIS

THEDB performs static analysis [7] to extract operation dependencies from each predefined stored procedure prior to transaction processing. The goal is to help identify the inconsistent transaction states as well as the query results for any uncommitted transaction that fails its validation, which is elaborated in Section 4. For ease of presentation and to focus on the key ideas, this paper considers only stored procedures without conditional branches; details for handling more general procedures are given in the extended version of this paper [57].

THEDB classifies the dependencies among program operations into two categories: *key dependencies* and *value dependencies*. A key dependency captures the relation between two operations where the preceding operation directly determines the accessing key of the subsequent operation. A value dependency captures the relation between two operations where the generated output of the preceding operation determines the non-key value to be used in the subsequent operation. The dependencies in a program are extracted using a static analysis process and they are represented by a graph referred to as a *program dependency graph*. Figure 3 shows such a graph for the bank-transfer example listed in Figure 1a. We say that the operations in Lines 4 and 7 are key-dependent on the preceding operation in Line 2, because Line 2 generates `dstId` that is further used as accessing key in Lines 4 and 7. Operations in Lines 8 and 9, in contrast, depict a value-dependency relation, since the preceding read operation defines the variable `bonus` that is later used as update value in the subsequent write operation.

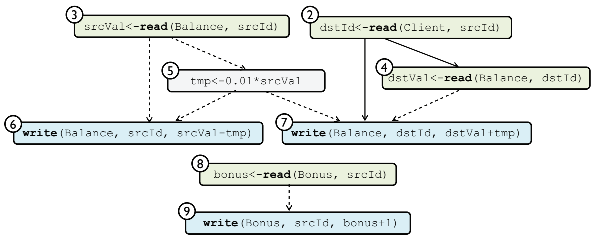

3
srcVal<-read(Balance, srcId)
2
dstId<-read(Client, srcId)
5
tmp<-0.01*srcVal
4
dstVal<-read(Balance, dstId)
6
write(Balance, srcId, srcVal-tmp)
7
write(Balance, dstId, dstVal+tmp)
8
bonus<-read(Bonus, srcId)
9
write(Bonus, srcId, bonus+1)

Figure 3: Program dependency graph. Solid lines represent key dependencies, while dashed lines represent value dependencies.

Given a stored procedure's program dependency graph, THEDB can leverage the extracted dependency information for healing the procedure's transactions that fail to pass the validation phase. The detailed mechanism is discussed in Section 4.

THEDB aborts any transaction that violates the integrities enforced by either application logic (e.g., user-defined constraints) or database constraints (e.g., functional dependencies). This is achieved by encoding additional dependencies for any enforced integrities in the program dependency graph. The whole transaction will be aborted once the restoration of any non-serializable operation results in the violation of integrities.

[3]The visibility bit for a record R is set to 0 iff R has been deleted by a committed transaction or R is newly inserted by a yet-to-be-committed transaction.

4. RUNTIME EXECUTION

This section describes the runtime execution of THEDB, our multicore database prototype that supports scalable transaction processing on multicores. THEDB serializes concurrent transactions using the transaction-healing protocol, which generally splits the execution of a transaction into three phases, including a read phase, a validation phase, and a write phase. During the validation phase, an additional healing phase is invoked to restore non-serializable operations once any inconsistent read is detected. This is achieved by leveraging a combination of the statically extracted dependency graph and the dynamically obtained execution information that is explicitly monitored during the transaction's execution.

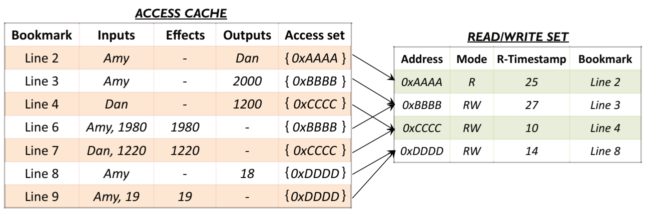
In this section, we first explain transaction healing by modeling transactions using simple read and write operations where records are accessed given their key values (Sections 4.1-4.6). Specifically, Section 4.1 explains how transaction healing tracks runtime information during the read phase of the transaction execution, and Sections 4.2-4.6 show how the validation, healing, and write phases are designed and optimized to facilitate transaction processing under highly contended workloads. To show the generality of transaction healing, we discuss the support for generic database operations (e.g., inserts, deletes, and range queries) as well as ad-hoc transactions in Sections 4.7-4.8.

4.1 Tracking Operation Behaviors

Similar with conventional OCC, transaction healing tracks the read/write set of a transaction and buffers all the write effects during the read phase of its execution [11]. In particular, a read/write set is a thread-local data structure (i.e., a structure that is privately updated by a single thread) where each element in the set is represented by the main-memory address of some data record accessed by the transaction. In addition, the following metadata is maintained for each accessed record in the transaction's read/write set: (1) a *mode* field indicating the access type (i.e., read (R), write (W), or read-write (RW)) to the data record; (2) an *R-timestamp* field recording the value of the timestamp metadata of the data record at the time it was read; and (3) a *bookmark* field uniquely identifying the transaction's operation that first reads the record; if the record is created by a blind-write operation, its bookmark value is $null$. For simplicity, throughout the paper, we represent a bookmark value by the line number in the corresponding stored procedure.

In addition to the read/write set, transaction healing further maintains a lightweight thread-local *access cache* to keep track of the runtime behavior of each operation. Each operation invokes an index lookup to retrieve a certain number of database records. By using the outputs of preceding operations or the input arguments to its stored procedure, a read operation *op* returns certain values that will be either consumed by the operations that are dependent on *op* or used as query results, while a write operation yields update effects that will be buffered to the local copy of its accessed record. In transaction healing, the access cache monitors inputs, outputs, as well as update effects to capture each operation's behavior. Each operation further maintains an access set in the access cache to log the memory addresses of all the records it reads or writes. The access cache facilitates the restoration of an operation as follows: on the one hand, recording the runtime behavior for each operation helps re-utilize the execution results yielded by those serializable operations; on the other hand, caching the memory addresses of the accessed records eliminates the need for invoking an index lookup to access a record as long as the accessing key of the operation remains the same.

Figure 4 shows the thread-local data structures maintained for transaction T_1 that is created in the bank-transfer example (see Sec-



ACCESS CACHE

Bookmark	Inputs	Effects	Outputs	Access set
Line 2	*Amy*	-	*Dan*	{ *0xAAAA* }
Line 3	*Amy*	-	*2000*	{ *0xBBBB* }
Line 4	*Dan*	-	*1200*	{ *0xCCCC* }
Line 6	*Amy, 1980*	*1980*	-	{ *0xBBBB* }
Line 7	*Dan, 1220*	*1220*	-	{ *0xCCCC* }
Line 8	*Amy*	-	*18*	{*0xDDDD*}
Line 9	*Amy, 19*	*19*	-	{*0xDDDD*}

READ/WRITE SET

Address	Mode	R-Timestamp	Bookmark
0xAAAA	*R*	*25*	*Line 2*
0xBBBB	*RW*	*27*	*Line 3*
0xCCCC	*RW*	*10*	*Line 4*
0xDDDD	*RW*	*14*	*Line 8*

Figure 4: Thread-local data structures.

tion 2). The execution of the read operation in Line 2 accesses a single data record stored at address `0xAAAA` and produces the value Dan that will be used by subsequent operations dependent on this read operation. Similarly, the write operation in Line 6 consumes two input arguments and updates the local copy of its corresponding record to $1,980. In this example, although each entry in the access cache is associated with exactly one element in the read-/write set, in general, range queries in a transaction could retrieve multiple records and therefore each entry in the access cache can map to multiple elements.

As we shall see shortly, with the runtime information maintained in these thread-local data structures, transaction healing is able to restore any non-serializable operation efficiently during the validation phase without resorting to abort-and-restart mechanism that can lead to extremely low resource-utilization rate.

4.2 Restoring Non-Serializable Operations

4.2.1 Validation Phase

The read phase in the transaction execution is performed in a consistency-oblivious manner. That is, any committed concurrent transaction can modify the global copy of a data record in the database without notifying any concurrent transaction that has a local copy of the same record in its read/write set. Thus, transaction healing, similar to conventional OCC, resorts to a validation phase to check the consistency of every record that is read by a transaction before committing that transaction. We briefly depict transaction healing's validation phase in Algorithm 1.

Algorithm 1: Validation phase in transaction healing.

```
Data: Read/write set $\mathcal{S}$ of the current transaction.

Validation Phase:
foreach $r$ in sorted($\mathcal{S}$) do
    Lock data record located at $r.address$;
    if $r$ is accessed by any read operation then
        if Validation of $r$ fails then
            Invoke healing phase for $r$;
```

In the validation phase for a transaction T, the data record corresponding to each element in T's read/write set will be locked and the locks are only released after the commit or abort of T. Locking of data records during the validation, healing, and write phases is necessary as multiple transactions could be concurrently validated and committed. Since the data records accessed by a transaction are known from its read/write set, deadlocks due to locking is avoided in THEDB by ordering the lock acquisitions following a global order that is applied to all transactions. In our implementation, the global order is based on an ascending order of the memory addresses of the data records [55].

For each element r in the read/write set $\mathcal{S}$, the validation phase first locks the data record R corresponding to r by turning on its lock bit. If R was retrieved by a read operation, the consistency of r is then validated by comparing the timestamps of R and r. A read inconsistency is detected if these timestamps are not equal, implying that a committed concurrent transaction has updated the same record. In this case, conventional OCC would abort and restart the entire transaction from scratch, wasting resources that have been put into running the transaction. Our proposed protocol, in contrast, detects and restores non-serializable operations by leveraging the data structures maintained in the thread-local workspace. This is achieved with the assistance of the healing phase.

4.2.2 Healing Phase

Algorithm 2: Healing inconsistent access during validation.

```
Data: Inconsistent element $r$, read/write set $\mathcal{S}$, access cache $\mathcal{C}$,
      and program dependency graph $\mathcal{G}$ of the current transaction.

Healing Phase:
Retrieve operation $op = r.bookmark$;
Restore $op$;
Retrieve child operation list $\mathcal{O}$ for $op$ w.r.t. $\mathcal{G}$;
Initialize FIFO healing queue $\mathcal{H} = \mathcal{O}$;
while $\mathcal{H} \neq \emptyset$ do
    $heal\_op$ = PopFront($\mathcal{H}$);
    if $heal\_op$ is key-dependent on its parent operation then
        Update $heal\_op$'s access set $\mathcal{M}$ through re-execution;
        foreach $m$ in $\mathcal{M}$ do
            if $m.address < r.address$ then
                if Attempting to lock $m$ fails then
                    Abort();
            Insert $m$ into $\mathcal{S}$ and update $\mathcal{C}$;
    else
        Restore $heal\_op$;
    Retrieve child operation list $\mathcal{P}$ for $heal\_op$;
    foreach $p$ in $\mathcal{P}$ do
        Insert $p$ into $\mathcal{H}$;
```

Algorithm 2 shows how the healing phase works. On detecting an inconsistent element r that is read by an operation *op* in the transaction, the healing phase first corrects the outputs for *op*, which is the initial non-serializable operation whose side effect is influenced by the inconsistency of r. The modification on the record pointed by r can affect *op*'s outputs, subsequently influencing the behavior of the operations that are dependent on *op*. Instead of restoring *op* with a straightforward operation re-execution, transaction healing corrects *op*'s outputs by directly visiting the memory addresses maintained in the access cache. This approach fully eliminates the potential overhead brought by index lookup. Meanwhile, transaction serializability is still preserved. The key reason is that *op*'s accessing key remains the same despite of the raised inconsistency, and therefore the corresponding access set is still unchanged[4]. The effect of *op*'s restoration must be propagated to all operations dependent on *op*, which can be identified using the statically-extracted program dependency graph. On retrieving an operation list $\mathcal{O}$ comprising the operation that are directly dependent on *op*, transaction healing selects the correct healing strategy

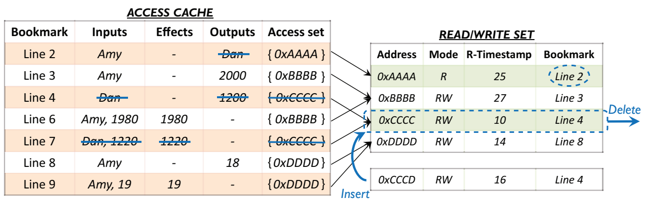
[4]This statement is still valid even if inserts, deletes, or range queries exist. See Section 4.7 for a detailed explanation.

for each operation according to the dependency type with *op*, as described below.

Restoring value-dependent operations. The restoration of an operation $heal\_op$ that is value-dependent on *op* simply requires a direct access to the corresponding memory addresses maintained in the access cache. This is because while the restoration can modify *op*'s outputs that will be consumed as inputs by $heal\_op$, the access set cached for $heal\_op$ remains the same, due to the invariance of $heal\_op$'s accessing key.

For a transaction issued from the stored procedure in Figure 1a, transaction healing merely restores operations in Lines 8 and 9 once detecting the inconsistency of Amy's bonus account. Such restoration is lightweight, as the access cache maintains the corresponding record pointers that will be used by these operations, and the index lookup overhead is consequently eliminated.

Restoring key-dependent operations. The restoration of an operation $heal\_op$ that is key-dependent on *op* calls for a more sophisticated mechanism. This is because *op*'s output directly serves as the accessing key for $heal\_op$, and therefore the correction of *op*'s output can affect the composition or even the size of $heal\_op$'s access set. Consequently, the maintained access cache should not be used for accelerating the restoration of $heal\_op$. Transaction healing solves this problem by invoking a complete re-execution of $heal\_op$, where the latest access set is retrieved through index lookup. Such re-execution also updates the membership of the transaction's read/write set.



ACCESS CACHE

Bookmark	Inputs	Effects	Outputs	Access set
Line 2	*Amy*	-	~~*Dan*~~	{ *0xAAAA* }
Line 3	*Amy*	-	*2000*	{ *0xBBBB* }
Line 4	~~*Dan*~~	-	~~*1200*~~	~~{ *0xCCCC* }~~
Line 6	*Amy, 1980*	*1980*	-	{ *0xBBBB* }
Line 7	~~*Dan, 1220*~~	~~*1220*~~	-	~~{ *0xCCCC* }~~
Line 8	*Amy*	-	*18*	{*0xDDDD*}
Line 9	*Amy, 19*	*19*	-	{*0xDDDD*}

READ/WRITE SET

Address	Mode	R-Timestamp	Bookmark
0xAAAA	*R*	*25*	*Line 2*
0xBBBB	*RW*	*27*	*Line 3*
0xCCCC	*RW*	*10*	*Line 4*
0xDDDD	*RW*	*14*	*Line 8*

0xCCCD	*RW*	*16*	*Line 4*

Figure 5: Healing inconsistency for the bank-transfer example.

We still use the bank-transfer procedure in Figure 1a to give a detailed explanation. Figure 5 shows the scenario where the validation of an instantiated transaction fails due to the detection that a committed concurrent transaction has updated Amy's client to $Dave$. The healing phase first corrects the output value from Dan to $Dave$ for the operation in Line 2. As this output is used as the accessing key in Lines 4 and 7, transaction healing further re-executes these two operations to retrieve the correct access set. In particular, the re-execution triggers an index lookup with the accessing key $Dave$. This also leads to a partial update to the membership of the read/write set, where the original element pointing to the memory address `0xCCCC` is replaced by a new one referring to the address `0xCCCD`. The healing phase terminates by correcting the update effects and outputs for these two operations.

Note that membership updates can cause deadlocks. Let us consider that a healing phase is invoked after detecting an inconsistent element r in the read/write set during validation. At this point, every element with a smaller memory address compared to r would have been locked, as is guaranteed by the global order of the validation phase. However, if a new element r_n containing a smaller memory address than r is inserted into the read/write set during the healing phase, the global order will be violated when attempting to lock r_n. Consequently, potential deadlocks can occur. Transaction healing resolves this problem using a no-waiting deadlock prevention technique [11, 61]. On confronting a failure when attempting to acquire the lock for r_n, transaction healing directly aborts the whole transaction instead of blindly spinning. This mechanism can be further optimized by setting an upper bound controlling the maximum number of times the lock request is attempted.

The read inconsistency within a transaction can be propagated through (indirect) operation dependencies. Transaction healing therefore recursively checks and restores all the possibly non-serializable operations by traversing the statically extracted program dependency graph in a breath-first approach. This essentially guarantees that any non-serializable operation is restored exactly once, and the healing overhead is minimized. The execution of a transaction resumes its validation once the healing phase completes. The forward progress of the validation is guaranteed because of the finite capacity of a transaction's read/write set. A transaction is allowed to commit if all the elements in its read/write set have been successfully validated. Transaction healing aborts a transaction only if the deadlock-prevention mechanism is triggered during the healing phase, where the membership of the read/write set is partially updated.

4.3 Committing Transactions at Scale

The commitment of a transaction installs all the locally buffered write effects to the database state. In addition, all the updates will also be flushed to the persistent storage for ensuring database durability. Transaction healing leverages a variation of epoch-based protocol [55] for committing transactions in a concurrent manner. The detailed mechanism is presented in Algorithm 3.

Algorithm 3: Commit protocol in transaction healing.

```
Data: Read/write set $\mathcal{S}$ of the current transaction.
Write Phase:
Generate global timestamp $glocal\_ts$;
Compute commit timestamp $commit\_ts$;
foreach $r$ in sorted($\mathcal{S}$) do
    if $r$ is accessed by any write operation then
        Install writes for data record $R$ located at $r.address$;
        Dump writes to persistent storage;
        Overwrite timestamp of $R$ with $commit\_ts$;
foreach $r$ in sorted($\mathcal{S}$) do
    Unlock data record located at $r.address$;
```

In transaction healing, a commit timestamp is a 64-bit unsigned integer, where the higher order 32 bits contain a global timestamp and the lower order 32 bits contain a local timestamp. The global timestamp is assigned with the value of a global epoch number E that is periodically (e.g., every 10 ms) advanced by a designated thread in the system, and the local timestamp is generated according to the specific thread ID. As an example, given three threads for executing transactions, the first thread generates a local timestamp from the list $0, 3, 6, ..., 3m, ...$, while the third thread generates a local timestamp from the list $2, 5, 8, ..., 3n + 2, ...$ When committing a transaction, the corresponding thread will assign the transaction with the smallest commit timestamp that is larger than (a) the commit timestamp attached in any record read or written by the transaction and (b) the thread's most recently generated commit timestamp. On obtaining the commit timestamp $commit\_ts$, a transaction installs all the buffered writes to the database and assigns the corresponding records with $commit\_ts$. In particular, each thread persists its committed transactions independently, and updates from transactions assigned with a same global epoch number E can be dumped to the persistent storage as a group. When a transaction finishes its commitment, it releases all its locks.

4.4 Guaranteeing Serializability

This section sketches an argument that transaction healing provides full serializability for transaction processing.

Compared with OCC, transaction healing restores non-serializable operations to heal inconsistent transaction states and query results. When processing a transaction whose read/write set does not overlap with that of any concurrent one, the effect of transaction healing is essentially equivalent to that of OCC. Now let us assume that an inconsistent element r that is read by the transaction is detected during the validation. At this point, we denote the set of elements with smaller memory address than r as E_s, and the set of elements with larger memory address as E_l. To restore the initial non-serializable operation *op* that first reads the inconsistent r, transaction healing directly reloads the latest value of the record's global copy, which is referred to by r. This action does not compromise database serializability, because the lock bit associated with this record has already been acquired by the current thread. After restoring *op*, transaction healing begins to restore all the operations that are (possibly indirectly) dependent on *op*. Restoring operation op_v that is value-dependent on its parent operation only requires retrieving the latest values of the records pointed by the access cache. This action does not affect the serializability. On the one hand, if a record accessed by op_v is pointed by an element in E_s, then the record access is consistent, because the current thread has exclusive privilege for accessing the record; on the other hand, if a record accessed by op_v is pointed by an element in E_l, then the record access can be inconsistent. However, the remaining part of the validation phase will lock and validate any element in E_l. Therefore, the raised inconsistency will be healed. The restoration of an operation that is key-dependent on its parent operation, however, can partially update the membership of read/write set. The potential deadlock brought by such membership update is prevented by transaction healing, as transaction healing attempts to lock any newly inserted element r_i with a smaller memory address than that of r. An uncommitted transaction will be aborted once any lock-acquisition attempt fails during the membership update. Transaction healing guarantees forward progress and final termination of transaction processing. This is because the read/write set of any transaction maintains a finite number of elements, and the validation phase certifies the consistency of any element for exactly once. To conclude, transaction healing guarantees serializability when restoring non-serializable operations, and the effect of transaction healing is equivalent to that of OCC.

4.5 Optimizing Dependent Transactions

Transaction healing optimizes the execution of dependent transactions, which must perform reads for determining their full read-/write sets [53, 54]. Compared with their independent counterparts, dependent transactions usually require more efforts for resolving conflicting accesses, since the restoration of non-serializable key-dependent operations can partially update the membership of the read/write set, and transaction aborts can be invoked by the deadlock-prevention mechanism. Transaction healing reduces these overheads by (1) avoiding unnecessary membership update by eliminating false invalidations and (2) reducing the likelihood of deadlock occurrences by rearranging global validation orders.

Eliminating false invalidations. During the validation phase, a false invalidation can occur if any concurrent transactions accessing the same record modify a column that is not read by the current transaction. Figure 6 shows a simplified example of such a case.

While transactions T_1 and T_2 both access record R, T_1's read is not compromised by T_2's write. However, conventional OCC can still invalidate T_1's access when checking R's timestamp field.

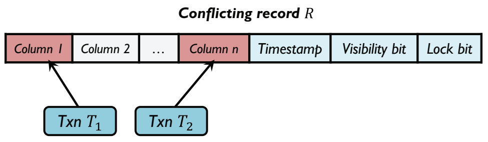


Figure 6: False invalidation. Transaction T_1 reads the first column while transaction T_2 writes the nth one. T_1 is invalidated although the write installed by T_2 does not affect T_1's correctness.

Such false invalidation is tolerable when processing independent transactions, but the invoked healing phase can bring high overhead for dependent transactions because of the partial membership update of the read/write set. Transaction healing eliminates such overhead by maintaining a local copy for each read operation. Once the validation of a certain element fails, transaction healing directly checks the value of the read column to determine whether a false invalidation occurs. This proposed mechanism may incur additional overhead inherited from memory allocation. However, our experiments confirm that such overhead is negligible.

Rearranging validation orders. Transaction healing can abort an uncommitted transaction when partially updating the membership of the read/write set, which is invoked by the restoration of an inconsistent key-dependent operation. The key reason is that the record accessing order in the healing phase may not be aligned with the global validation order, and therefore attempts will be made to lock any record with comparatively smaller memory address (see Section 4.2). Observing that most stored procedures in certain applications access tables based on a *tree schema* [20, 50], transaction healing consequently sorts the elements in the read/write set according to any topological order of the tree structure. In particular, elements pointing to the records extracted from the same table are ordered based on the memory address. In this way, only records with larger order are inserted into the read/write set during the membership update. As a result, the likelihood of deadlock occurrences is greatly reduced.

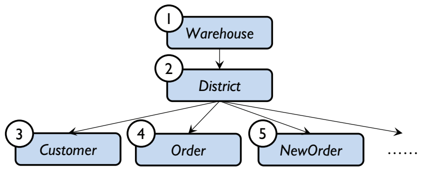


Figure 7: Validation order in the TPC-C benchmark. The stored procedures modeled in this benchmark touch `Warehouse` table and `District` table before accessing any other tables.

Figure 7 shows the tree schema for the TPC-C benchmark. As the restoration of an operation accessing `District` table never affects those accessing `Warehouse` table, no deadlock can occur when healing the inconsistency of an element pointing to the record from `District` table. Based on this principle, the possibility of transaction abort caused by deadlock prevention can be significantly reduced.

4.6 Optimizing Independent Transactions

Transaction healing achieves optimal performance when processing independent transactions, whose read/write sets can be determined according to the input arguments prior to execution [18, 53, 54]. As transaction abort happens only when confronting membership update during the healing phase, independent transactions processed using the transaction healing protocol are guaranteed to be committed due to the absence of key-dependency relations. Based on this observation, transaction healing further optimizes the execution of such transactions by combining the validation phase with its subsequent write phase. Accordingly, any write effect of a transaction can be directly applied to the database state once the corresponding element in the read/write set has passed validation. As a result, transaction healing reduces lock-holding duration for independent transactions, increasing the overall level of concurrency when supporting OLTP workloads.

4.7 Supporting Database Operations

As a general database system, THEDB supports the full spectrum of database operations that are expressible by the SQL language. In this subsection, we discuss how different operations are implemented in THEDB.

4.7.1 Inserts and Deletes

When committing a transaction T, the timestamp of each record modified or created by T is updated to the transaction's commit timestamp. In addition, the visibility bit of each record that is deleted by T is turned off. THEDB further relies on a garbage collector to periodically clean up all the deleted records. To guarantee the correctness of garbage collection, a reference counter is maintained for each record to count the number of transactions that are currently accessing the record. A deleted record can be safely removed from the database once its reference counter drops to 0.

We further explain how THEDB handles insert operations in the presence of conflicting operations using three scenarios.

In the first scenario, consider an insertion of a new record R by transaction T_1 followed by a read operation by another concurrent transaction T_2 to read R. To insert R, T_1 performs the insertion during its read phase, with the visibility bit of R set to false. When T_2 reads R, although R is added to T_2's read set, R is not visible to T_2 due to its visibility value (i.e., R does not exist from T_2's perspective). When T_1 commits, the visibility bit of R will be turned on, indicating that R is now visible to other transactions. During the validation phase of T_2, T_2's read operation that accesses R would be detected to be non-serializable, and the healing phase will be triggered to restore all the affected non-serializable operations.

In the second scenario, consider the reverse of the first scenario where a transaction T_1 first attempts to read a non-existent record R followed by a concurrent transaction T_2 that inserts R. When T_1 attempts to read the non-existent R, THEDB will create a dummy empty record R_e to represent R with the visibility bit of R_e set to off, and an element corresponding to R_e is inserted to T_1's read set. If T_2 attempts to insert R into the database, it must acquire the lock on R_e before performing the real insertion. Once T_2 has passed the validation, record insertion is executed by directly copying R's content to R_e. Suppose that T_1 commits before T_2. Both transactions can commit successfully without confronting validation failure. However, if T_2 commits before T_1, T_1 will detect the modification of R's timestamp during its validation phase, and a healing phase will be triggered to heal the detected inconsistency.

In the third scenario, we consider the case where two concurrent transactions attempt to insert the same record R. Suppose that T_1's insertion is performed before T_2's insertion during their read phases. Similar to the discussion for the second scenario, a dummy empty record R_e would be created by T_1's insertion with its visibility bit turned off. Subsequently, when T_2 attempts to insert R, it would detect the presence of R_e and an element that points to R_e will be added to T_2's read/write set. Should T_2 validate and commit before T_1, T_2's insertion will be committed and the visibility bit for R_e will be turned on. Subsequently, T_1's validation will fail on detecting the modification of R_e's timestamp; in this case, T_1 will be aborted due to the integrity constraint violation.

We have demonstrated that THEDB guarantees serializability in all the cases where inserts are performed concurrently with conflicting operations. We conclude that database serializability can be preserved with the existence of inserts and deletes.

4.7.2 Range Queries and Phantoms

The design of transaction healing naturally supports range queries that access a collection of records in a table. However, range queries can result in the phantom problem [23]. Instead of utilizing the *next-key locking* mechanism [40] that is specifically designed for two-phase locking protocol, THEDB solves this problem by leveraging a mechanism that is first proposed by Silo [55]. THEDB records a version number on each leaf node of a B+-tree to detect structural modifications to the B+-tree. Any structural modification caused by inserts, deletes, or node splits will increase the version number. When performing a range query in a transaction, transaction healing records both the version number and the leaf node pointers to the read/write set. During the validation phase, on detecting a structural change that is indicated by the version mismatch, transaction healing attempts to heal the inconsistency by restoring the corresponding non-serializable operations.

4.8 Supporting Ad-Hoc Transactions

In real-world applications, a database user can submit ad-hoc transactions without invoking stored procedures that are defined prior to execution. THEDB processes such type of transactions using the conventional OCC scheme, which is fully compatible with the transaction-healing mechanism. In the case that all the incoming transactions are ad-hoc, THEDB is equivalent to a database system that implements conventional OCC for serializing transactions. While it is technically possible to enable transaction healing for ad-hoc transactions by building an efficient program analyzer that extracts dependency graphs at runtime, there still exists two factors that may restrict the system's effectiveness. First, a database user may issue SQL statements within a transaction interactively, making the extraction of dependency graphs difficult due to the absence of complete knowledge of the transaction program. Second, most ad-hoc transactions may be executed only once, and the overhead introduced by runtime program analysis can potentially outweigh the benefits brought by transaction healing, making it unnecessary to perform transaction healing to execute ad-hoc transactions. At current stage, we restrict the scope of transaction healing to transactions that are issued from stored procedures, and leave the investigation of supporting ad-hoc transactions as a future work.

5. EVALUATION

We implemented THEDB from the ground up in C++. As a highlight, we automated the static analysis in THEDB with the LLVM Pass framework [1]. In this section, we evaluate the effectiveness of THEDB, by seeking to answer the following key questions:

1. Why do the state-of-the-art OCC protocols not scale well under highly contended workloads?
2. Can THEDB scale linearly under different workloads?

All the experiments were performed on a multicore machine running Ubuntu 14.04 with four 12-core AMD Opteron Processor 6172 clocked at 2.1 GHz, yielding a total of 48 physical cores. Each core owns a private 64 KB L1 cache and a private 256 KB L2 cache. Every 6 cores share a 5 MB L3 cache and a 8 GB local

DRAM. The machine has a 2 TB SATA hard disk. We compare THEDB with the following five systems:

THEDB-OCC. THEDB-OCC implements the conventional OCC with several optimization techniques applied [61]. We have implemented the scalable timestamp-allocation mechanism proposed by Silo [55] to improve system concurrency.

THEDB-SILO. THEDB-SILO faithfully implements Silo's design [55] on THEDB. It adopts a variation of OCC and improves concurrency level by eliminating the necessity for tracking anti-dependency relations.

THEDB-2PL. THEDB-2PL implements the widely accepted two-phase locking (2PL) mechanism [11]. We adopt no-waiting strategy for avoiding transaction deadlocks. We note that this strategy is reported as the most scalable deadlock-prevention approach for 2PL-based protocols [61].

THEDB-HYBRID. THEDB-HYBRID is a system that adopts a hybrid concurrency-control mechanism [28, 52, 60] for optimized performance. THEDB-HYBRID first executes an incoming transaction using OCC, and switches over to executing it using 2PL protocol should the transaction aborts due to OCC validation failure.

THEDB-DT. THEDB-DT is a partitioned deterministic database that follows the design of existing works [32, 33, 54]. It leverages coarse-grained partition-level locks to serialize transaction executions. In particular, several optimization mechanisms, including replication of read-only tables, were adopted [19, 45].

We adopted two well-known benchmarks, namely TPC-C [4] and Smallbank [6], to evaluate the system performance. For the TPC-C benchmark, we control the workload contention by varying the number of warehouses. Specifically, the contention degree increases with the decrease in the number of warehouses. For the Smallbank benchmark, the degree of workload contention is controlled by a parameter θ, which indicates the skewness of the Zipfian distribution. Increasing θ yields more contended workload. Our query-generation approach faithfully follows that employed by several previous works [55, 61].

5.1 Existing Performance Bottlenecks

We begin our evaluation with a detailed performance analysis on the state-of-the-art OCC protocols. We measure the transaction throughput of THEDB-OCC and THEDB-SILO with different degrees of workload contentions using the TPC-C benchmark. Figure 8 shows the results produced with 46 cores[5]. By decreasing the warehouse count from 48 to 2, the performance of both systems drops drastically. Specifically, when setting the number of warehouses to 2, these two systems respectively yield only 150 K and 60 K transactions per second (tps), reflecting high system sensitivity to workload contentions. To investigate how transaction aborts influence system performance, we disable the validation phase of the OCC protocols in both systems. Such modification can result in non-serializable results due to the absence of consistency checking, but the achieved transaction rates essentially indicate the peak performance that could be attained without any aborts. As shown in Figure 8, disabling the validation phase essentially yields 3 (THEDB-OCC) to 12 (THEDB-SILO) times higher transaction rate for highly contended workloads (see THEDB-OCC$^-$ and THEDB-SILO$^-$). In particular, the peak performance achieved by THEDB-SILO can be 10-15% higher than that of THEDB-OCC after disabling the validation phase. The key reason is that the commit protocol of THEDB-SILO by design eliminates the necessity for tracking anti-dependency relations [55], consequently leading to reduced locking overhead. Note that the transaction rate of both systems can still deteriorate even after disabling the validation phase. This is mainly because of lock thrashing effects [11, 61], where concurrent transactions are waiting for the access privilege of contended locks. Such a phenomenon exists universally in modern concurrency-control mechanisms that require fine-grained locking scheme [61]. While the recently proposed deterministic partitioned databases can prevent such overhead [32, 33, 53, 54], the management of coarse-grained locks in these databases incurs costly overhead when processing cross-partition transactions. This is confirmed by our experiments presented later in Section 6.2.

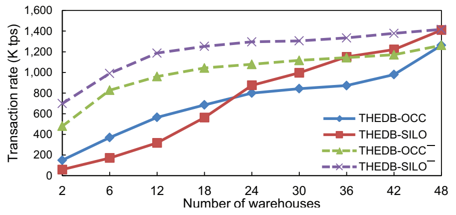
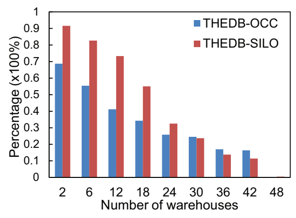
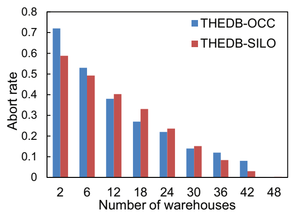


Figure 8: Transaction rate with different degree of contentions. The number of cores is set to 46.

We next analyze the overheads incurred by OCC protocols due to their abort-and-restart mechanism. Figure 9a depicts the percentage of the total execution time spent on transaction abort-and-restart. With the number of warehouses set to 2, THEDB-OCC and THEDB-SILO respectively spent 69% and 91% of their execution time on aborting-and-restarting transactions due to validation failure. This result confirms that the abort-and-restart mechanism is the key contributor to the inefficiency of the state-of-the-art OCC protocols. Figure 9b illustrates that both THEDB-OCC and THEDB-SILO achieve similarly high abort rate[6] which increases as expected with increasing data contention (i.e., lower number of warehouses). Given the relatively weaker performance of THEDB-SILO under highly contended workloads compared to THEDB-OCC, this indicates that THEDB-SILO is more sensitive to high abort rate. The main reason is that THEDB-SILO starts its validation phase for a transaction only after locking its entire write set, which therefore incurs more wasted effort for an aborted transaction. Indeed, the concurrency-control protocol adopted in Silo can be considered as a *more optimistic* OCC scheme. While this design achieves comparatively higher transaction throughput for low-contention workloads, its design suffers significant performance penalty when the workload is highly contended.



(a) Percentage of allocated time.



(b) Abort rate.

Figure 9: Overhead of the abort-and-restart mechanism with different degree of contentions. The number of cores is set to 46.

In the experiments above, we confirm that the existing OCC protocols are not scalable on multicore architectures due to the expensive abort-and-restart mechanism. Given this, THEDB is designed

[5]We leave two cores for handling OS-related tasks.

[6]The abort rate is calculated as the number of transaction restarts divided by the number of committed transactions.

and implemented to achieve high scalability even under highly contended workloads by reducing the abort-and-restart overhead.

5.2 Scalability

This subsection evaluates the scalability of TheDB. Specifically, we attempt to address the following questions: (1) whether TheDB yields high transaction rate under workloads with different contentions; (2) whether TheDB achieves low latency when processing transactions; (3) whether TheDB sustains high performance in the presence of ad-hoc transactions; (4) whether TheDB achieves satisfactory performance in benchmarks comprising short-duration transactions; and (5) how each proposed mechanism affects the system performance.

5.2.1 Transaction Rate

We first investigate the robustness of TheDB using the TPC-C benchmark with 46 cores. We set the percentage of cross-partition transactions to 0 and change the number of warehouses from 2 to 48 to decrease the workload contention. Figure 10 shows the results. All the systems in comparison achieve near-linear scalability with the number of warehouses set to 48. In particular, TheDB-Dt yields the highest transaction rate, due to the absence of cross-partition transactions. However, with the increase of workload contention, the performance of TheDB-Occ, TheDB-Silo, TheDB-2pl, TheDB-Hybrid, and TheDB-Dt drop sharply, especially when the number of warehouses is set to 2. TheDB, in contrast, sustains a relatively high transaction rate that is very close to TheDB-Occ's peak performance (denoted as TheDB-Occ^{-}) where the validation phase of the OCC protocol is disabled. This observation essentially confirms that the transaction healing protocol adopted in TheDB brings little overhead to the system runtime, and it can scale well even when the workload is highly contended.

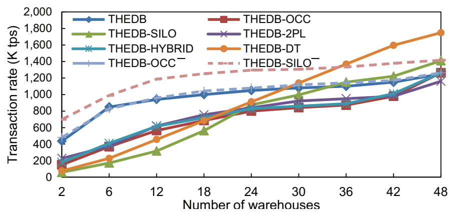


Figure 10: Transaction rate with different degree of contentions. The number of cores is set to 46.

Figure 11 further presents the system scalability using the same benchmark. Under the highly contended workload shown in Figure 11a, TheDB achieves much higher transaction rate than that of the other five systems. In contrast, TheDB-Silo achieves the worst performance. This is because TheDB-Silo's commit protocol is vulnerable to frequent transaction aborts, as explained in Section 6.1. Although TheDB-2pl achieves 25% higher transaction rate compared to TheDB-Occ, its long-duration locks decrease the concurrency degree, making it less effective on multicore architecture. TheDB-Hybrid also achieves unsatisfactory results, since its performance is severely restricted by the combination of OCC and 2PL protocols. While the percentage of cross-partition transactions is set to 0 in this experiment, TheDB-Dt still yields a low performance. The major reason is that the execution model of TheDB-Dt forbids concurrent execution on a single partition, and therefore the number of cores that can be utilized for processing transactions is strictly limited by the number of warehouses in the TPC-C benchmark, consequently resulting in low resource utilization rate. Compared to these systems, TheDB scales near linearly towards 46 cores, achieving respectively 2.3 and 6.2 times higher throughput than that of TheDB-2pl and TheDB-Silo. This is because the transaction-healing protocol adopted by TheDB heals any inconsistency that is detected during the validation phase, and the expensive overhead caused by abort-and-restart is completely eliminated with the help of the proposed optimization mechanisms. Figures 11b and 11c further illustrate that, while the performance of the other five systems improves under low-contention workload, TheDB maintains a high transaction throughput when scaling to 46 cores, demonstrating TheDB's high scalability and robustness.

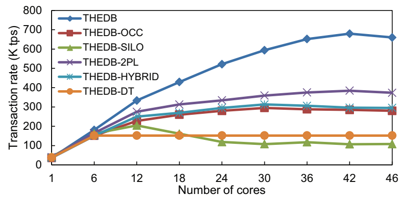
While TheDB achieves a comparatively high transaction rate when supporting high-contention workloads, the experimental results reported above indicate that TheDB still suffers from performance degradation due to lock thrashing [11, 61]. While recent research has proposed deterministic database systems to overcome this problem, the management of coarse-grained locks in such database systems incurs additional overhead when processing cross-partition transactions. Figure 12 shows the transaction rate of each system with different percentage of cross-partition transactions. In this set of experiments, the number of cores is set to 46. While all the other systems achieve a stable performance that is not affected by the percentage of cross-partition transactions, TheDB-Dt suffers from a significant drop in performance when cross-partition transactions are introduced. Specifically, regardless of the workload contentions, TheDB-Dt achieves a low transaction rate when the percentage of cross-partition transactions increases to 10%. This is because the coarse-grained locking mechanism adopted by TheDB-Dt requires a transaction to lock all the partitions that it accesses until it completes. Consequently, any concurrent transaction that needs to access one of the locked partitions would be blocked. This experiment demonstrates that existing deterministic databases cannot perform well when supporting cross-partition transactions.

5.2.2 Transaction Latency

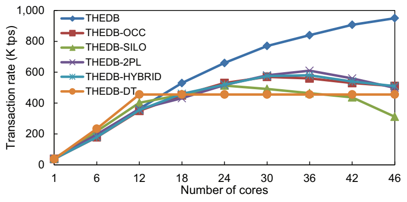
Next, we analyze the transaction latency of TheDB when processing highly contended workloads. We execute the TPC-C benchmark with the number of warehouses set to 4, and measure the processing durations for `NewOrder` transactions and `Delivery` transactions. Both types of transactions are dependent transactions, which must perform read operations to obtain its full read/write set. In particular, the program logic of `Delivery` transactions is more complicated, and the processing latency can be much longer compared to `NewOrder` transactions.

Table 1 shows the transaction latencies of different systems for processing `NewOrder` transactions. Compared with TheDB-Occ TheDB-Silo, and TheDB-2pl, TheDB incurs a much shorter latency with over 95% of the `NewOrder` transactions committed within 80 µs. In contrast, the latencies for TheDB-Occ, TheDB-Silo, and TheDB-2pl are more varied, ranging from below 20 µs to over 640 µs. This is because the conventional abort-and-restart mechanism adopted by these two systems could incur a high overhead when the same transaction has to be re-executed multiple times. Table 1 also presents the latencies achieved by TheDB-Occ and TheDB-Silo with the validation phase disabled (denoted as TheDB-Occ^{-} and TheDB-Silo^{-}). The reported numbers are very close to that obtained by TheDB, showing that the adopted transaction-healing protocol incurs little overhead to the system runtime. To conclude, TheDB enables efficient transaction processing as any transaction that fails the validation will be healed without getting restarted from scratch.

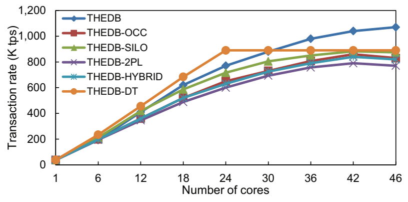
We further analyze the latencies achieved by different systems when processing `Delivery` transactions, which comprise



(a) 4 warehouses.

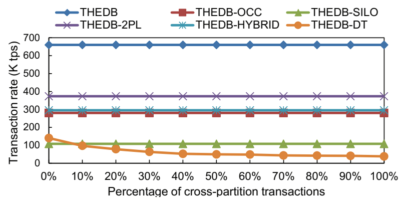


(b) 12 warehouses.

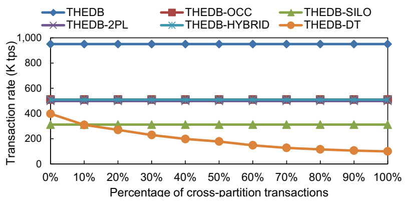


(c) 24 warehouses.

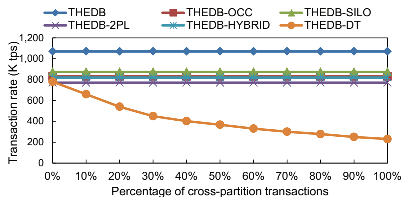
Figure 11: Transaction rate for TPC-C benchmark with different degree of workload contentions.



(a) 4 warehouses.



(b) 12 warehouses.



(c) 24 warehouses.

Figure 12: Transaction rate for TPC-C benchmark with different percentage of cross-partition transactions.

complex dependencies among operations. As shown in Table 1, by disabling the validation phase, TheDB-Occ commits 84.1% of the `Delivery` transactions within 320 µs (denoted as TheDB-Occ$^-$). In this scenario, no consistency check is performed during the execution, and therefore transactions will always be committed without being any restarts. However, enabling the validation phase significantly increases the transaction latency, and only 14.1% and 16.0% of the transactions are committed within 320 µs respectively by TheDB-Occ and TheDB-Silo. This result demonstrates the inefficiency of the abort-and-restart mechanism. Compared with these two systems, TheDB could achieve a much lower transaction latency. While the healing of inconsistencies for dependent transactions could cause membership updates of the read/write sets, TheDB is still able to commit nearly 90% of the transactions within 640 µs. In addition, the transaction latency is strictly bounded within 1280 µs, and hence the overall system performance is much better than that achieved by TheDB-Occ, TheDB-Silo, and TheDB-2pl.

The experiments reported above demonstrate that the transaction-healing protocol adopted by TheDB does not incur high latency when processing different types of transactions.

5.2.3 *Ad-Hoc Queries*

TheDB processes ad-hoc transactions using conventional OCC protocol, which is fully compatible with transaction healing. On detecting inconsistency during validation phase, ad-hoc transactions will be directly aborted and restarted from scratch. In this experiment, we randomly taint some transactions as ad-hoc transactions, and examine how the transaction rate of TheDB is influenced by the percentage of ad-hoc transactions. Figure 13 shows the result with the number of warehouses set to 4. By changing the percentage of ad-hoc transactions from 0% to 100%, the performance of TheDB deteriorates smoothly, and finally degrades to the performance of conventional OCC protocol. This is because TheDB's transaction-processing scheme is essentially equivalent to that of TheDB-Occ when all the incoming transactions are ad-hoc. Given the fact that most transactions in modern applications are generated from stored procedures [50], we conclude that transaction healing can provide a great performance boost when supporting real-world OLTP workloads.

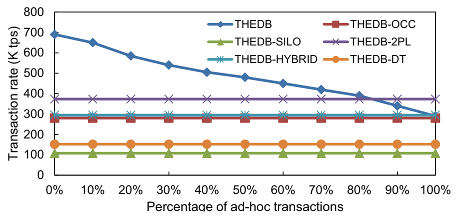


Figure 13: Transaction rate with different percentages of ad-hoc transactions. The number of cores is set to 46.

5.2.4 *Short-Duration Transactions*

In the following experiments, we use the Smallbank benchmark to evaluate the performance of TheDB for workloads with short-duration transactions. Recall that the workload contention of the Smallbank benchmark is controlled by a parameter θ, which indicates the skewness of the Zipfian distribution. Table 2 shows the percentage of accesses to different keys based on the various Zipfian distributions by varying θ. Here, the number of records in each table is set to 1,000. Note that the workload contention grows exponentially with θ. The results in Table 2 show that the abort rates of TheDB-Occ and TheDB-Silo climb rapidly from 0.007 to 0.324 and 0.403, respectively. Different from these two systems, TheDB did not abort any transaction as all the detected validation failures were resolved with the healing phase.

Figure 14 shows the transaction rate of different systems with θ varying from 0.1 to 0.9. In this experiment, the number of cores is set to 24. With θ set to 0.1, TheDB-Silo achieves around 5% higher throughput compared to TheDB and TheDB-Occ. This is because the design of TheDB-Silo's concurrency-control protocol eliminates the necessity for checking anti-dependency relations. However, the tradeoff for such an extreme optimistic protocol is that it underperforms for high-contention workloads. In particular, when $\theta = 0.9$, TheDB-Silo yields the lowest transaction throughput among all the systems being compared. However, the performance of TheDB remains stable for different workload contentions. Under highly contended workload, the transaction throughput achieved by TheDB is 4.5 times higher than other systems, and this performance is very close to the peak throughput

Transaction type	Latency (µs)	TheDB	TheDB-Occ	TheDB-Silo	TheDB-2pl	TheDB-Occ^{-}	TheDB-Silo^{-}
`NewOrder`	10 - 20	0.2%	0%	3.5%	1.1%	0%	4.8%
	20 - 40	36.7%	13.7%	25.9%	29.2%	34.0%	45.3%
	40 - 80	59.1%	32.1%	28.8%	41.4%	62.8%	42.0%
	80 - 160	2.7%	28.4%	28.1%	19.9%	3.2%	7.7%
	160 - 320	1.3%	17.9%	7.8%	6.7%	0%	0.2%
	320 - 640	0%	5.6%	4.7%	1.5%	0%	0%
	640 - INF	0%	2.3%	1.2%	0.3%	0%	0%
`Delivery`	10 - 80	0%	0.3%	0.8%	1.4%	0%	0%
	80 - 160	0.3%	0%	0%	1.6%	0.6%	1.4%
	160 - 320	41.1%	14.1%	16.0%	29.6%	84.1%	69.3%
	320 - 640	48.4%	31.4%	24.2%	38.1%	14.0%	22.7%
	640 - 1280	10.2%	34.6%	37.9%	21.3%	1.2%	6.6%
	1280 - 2560	0%	13.8%	16.8%	7.5%	0%	0%
	2560 - 5120	0%	4.0%	3.9%	0.6%	0%	0%
	5120 - INF	0%	1.7%	0.5%	0%	0%	0%

Table 1: Transaction latency for TPC-C benchmark. The number of warehouses is set to 4, and the number of cores is set to 46.

θ	1st	2nd	10th	100th	Abort rate
0.1	0.25%	0.24%	0.20%	0.16%	0 / 0.007 / 0.007
0.2	0.45%	0.39%	0.29%	0.18%	0 / 0.008 / 0.008
0.3	0.78%	0.63%	0.40%	0.19%	0 / 0.009 / 0.009
0.4	1.34%	1.02%	0.55%	0.22%	0 / 0.013 / 0.010
0.5	2.26%	1.60%	0.74%	0.22%	0 / 0.016 / 0.012
0.6	3.70%	2.45%	0.95%	0.24%	0 / 0.024 / 0.023
0.7	5.86%	3.60%	1.20%	0.23%	0 / 0.047 / 0.084
0.8	8.91%	5.17%	1.48%	0.23%	0 / 0.251 / 0.347
0.9	13.01%	7.06%	1.72%	0.21%	0 / 0.324 / 0.403

Table 2: The percentage of accesses to the first, second, 10th, and 100th most popular keys in Zipfian distributions for different values of θ. The last column shows the abort rates of TheDB, TheDB-Occ, and TheDB-Silo respectively.

that is achieved by disabling OCC's validation phase. This result essentially demonstrates the low overhead of transaction healing.

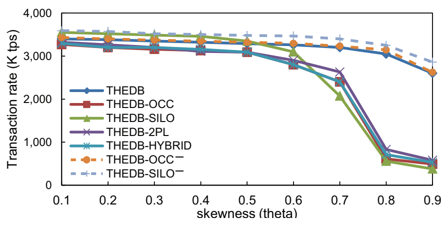


Figure 14: Transaction rate with different degree of contentions. The number of cores is set to 24.

We further compare the transaction latency of TheDB with that of TheDB-Occ and TheDB-Silo in Table 3. When θ = 0.5, the three systems in comparison yield similar transaction latency. In TheDB, 25% of the transactions are committed within 4.86 µs. This number is very close to that achieved by TheDB-Occ and TheDB-Silo, which are 4.79 µs and 5.45 µs, respectively. This result essentially indicates that transaction healing brings little overhead to the system runtime when processing workloads with low contentions. When supporting highly contended workloads, TheDB generates remarkably lower latency compared to the other systems. Specifically, 95% of the transactions executed by TheDB complete within 11.45 µs when θ = 0.9; in contrast, the latency for TheDB-Occ and TheDB-Silo increases to 36.14 µs and 42.54 µs, respectively. This result indicates that the state-of-the-art OCC

θ	Percentile	TheDB	TheDB-Occ	TheDB-Silo
0.5	25%	4.86 µs	4.79 µs	5.45 µs
	80%	8.52 µs	8.57 µs	9.02 µs
	95%	10.63 µs	11.12 µs	11.58 µs
0.7	25%	4.55 µs	4.25 µs	3.20 µs
	80%	9.12 µs	8.43 µs	7.75 µs
	95%	11.84 µs	12.74 µs	12.34 µs
0.9	25%	4.57 µs	2.60 µs	2.21 µs
	80%	9.14 µs	5.22 µs	4.50 µs
	95%	11.45 µs	36.14 µs	42.54 µs

Table 3: Transaction latency for Smallbank benchmark. The number of cores is set to 24.

protocols cannot process each transaction uniformly, and the abort-and-restart mechanism severely hurts the latency of some transactions in the workload, hence causing degradation to the overall system performance.

5.2.5 *Runtime Overhead*

In this section, we analyze the runtime overhead incurred by transaction healing in TheDB. Compared with OCC, transaction healing maintains an additional access cache during the read phase of the transaction execution. In addition, a local copy of any record that is read by the transaction is held to eliminate potential overhead caused by false invalidation when processing dependent transactions. Such overheads can potentially cause observable performance degradation when processing low-contention workloads. To precisely measure the performance overhead associated with transaction healing, we executed the TPC-C benchmark with the number of warehouses set to be equal to the core count. To minimize conflicting actions, we allocate each thread to be responsible for processing transactions associated with a single warehouse. Table 1 shows the experimental results. Without maintaining the access cache and local copies for read operations, TheDB yields 1139 K tps when processing transactions with 46 cores (denoted by Normal). Maintaining the access cache incurs little overhead to the system runtime, and TheDB still achieves 1087 K tps with 46 cores enabled (denoted by +Access Cache). Similarly, the overhead caused by the maintenance of local copies for read operations is also negligible, and less than 2% performance degradation is observed (denoted by +Read Copy). Hence, we conclude that transaction healing brings little overhead to the system runtime when processing low-contention workloads.

The experiments presented above demonstrate that the

#cores	8	16	24	32	40	46
Normal (K tps)	328	606	840	971	1088	1139
+Access Cache (K tps)	325	602	811	937	1036	1087
+Read Copy (K tps)	314	588	790	924	1015	1067

Table 4: Transaction rate when processing the TPC-C benchmark. The number of warehouses is set to be equal to the core count.

transaction-healing protocol can achieve both high scalability and robustness for transaction processing on multicores, with little performance overhead brought to the system runtime.

6. RELATED WORK

Multicore databases. Concurrency-control protocols have been investigated in the last thirty years [11]. To facilitate the performance in disk-based OLTP databases, several works [29, 30, 31, 43, 44] have been proposed to remove centralized locking bottlenecks. With the evolution of hardware architecture, researchers have shifted their attentions to improve performance in multicore main-memory databases. Ren et al. [48] removed contention bottlenecks in centralized lock manager by proposing a lightweight per-tuple 2PL scheme. Yao et al. [59] extracted concurrency control from execution to accelerate the multicore processing of OLTP workloads. Larson et al. [35] recently revisited two multi-version concurrency-control (MVCC) algorithms, and their study further settled the foundation for Microsoft's Hekaton database [21]. Faleiro et al. [25] proposed a technique for lazily evaluating transactions, and this technique improves database performance for certain kinds of workloads. Based on a similar design principle, the same authors improved the MVCC performance by decoupling concurrency control and version management from transaction execution [24]. Levandoski et al. [36] presented an efficient range concurrency-control scheme that extends multi-version timestamp ordering to support range resources and fully supports phantom prevention. As a departure from the traditional database architectures, several deterministic databases, including H-Store [32], Hyper [33], and Calvin [53, 54], have been proposed. These databases divide the underlying storage into multiple partitions, each of which is protected by a lock and is assigned a single-threaded execution engine with exclusive access. To optimize system performance, different partitioning schemes [19, 45] were proposed to reduce the number of cross-partition transactions. Unlike these databases, THEDB leverages static analysis to optimize database performance.

Optimistic concurrency control. Optimistic concurrency control (OCC) was first proposed by Kung and Robinson [34]. Witnessing its vulnerability to contended workloads, several works have been introduced to reduce OCC's abort rate. Agrawal et al. [5] adopted a multi-versioned protocol to allow inconsistent access to the database records. Herlihy [28] eliminated successive abort-and-restart in OCC by resorting to pure lock-based protocol once transaction abort occurs. Different from these proposals, THEDB exploits program semantics to partially re-execute transactions instead of blindly restarting the entire transaction from the scratch. In recent years, OCC have been widely adopted in main-memory databases. As a representative system, Silo [55] achieves high transaction rate by avoiding anti-dependency tracking and taking advantage of a main-memory index [39]. Hekaton [21] facilitates the OCC's performance by exploiting multi-versioning to avoid installing writes until commit time. Hyder [12] adopts a variant of OCC protocol called meld [13] that is specifically designed for log-structured databases. Absorbing the design benefits from these systems, THEDB lays its major contribution to improve OCC's scalability and robustness to the contended workloads by leveraging static analysis to the transaction programs. Some other recent works focus on optimizing OCC in distributed environments and new system architecture. For example, Bernstein et al. [9, 10] improved OCC's performance in distributed log-structured databases without storage partitioning. Ding et al. [22] introduced a new elastic distributed transaction processing mechanism that separates the validation layer from storage layer. Wang et al. [56] leveraged restricted transactional memory to optimize the OCC performance.

Program analysis. Many works have been proposed to adopt program partitioning and transformation to optimize system performance [8, 15, 16, 17, 26, 27, 46]. Among them, one widely adopted technique is transaction chopping [49], which analyzes possible transaction conflicts using SC-cycles. In fact, the database community has investigated various types of transaction partitioning mechanisms for improved system performance [8, 26, 27]. Transaction chopping is also applied to several modern database applications. Zhang et al. [62] proposed transaction chain to achieve serializability in geo-distributed databases, and Mu et al. [41] tracked dependencies between concurrent transactions to optimize distributed transactions in high contention scenario. Several other works also resorted to program analysis for improved performance. Doppel [42] splits transaction execution into two phases, and processes commutative operations in parallel for higher transaction rate. Wu et al. [58] leveraged program analysis to parallelize command-log recovery on multicores. Unlike these previous works, THEDB only requires simple static analysis that extracts the dependencies within a program, and ad-hoc queries are naturally supported.

Transactional memory. In the area of transactional memory systems, researchers also adopt OCC to facilitate the performance of transactional execution. Blundell et al. [14] adopted symbolic tracking to commit transactions in the case of data conflicts by re-executing read instructions in the programs. However, their approach is restricted to only "non-critical conflicts" occurring on auxiliary or bookkeeping data. Litz et al. [38] resorted to snapshot isolation for reduced abort rate, but their approach inevitably sacrifices program consistency. Ramadan et al. [47] proposed conflict serializability to reduce aborts by relaxing concurrency control. While their approach reduce abort rates in certain cases, it is not general enough to provide full capability of tolerating all the read-write conflicts. Different from these works, THEDB is specifically designed for supporting OLTP applications and can be generalized to support all kinds of database operations. Meanwhile, system durability is guaranteed by its effective commit protocol.

7. CONCLUSION

We have introduced a new concurrency-control mechanism, called *transaction healing*, that scales the conventional OCC towards dozens of cores even under highly contended workloads. Transaction healing leverages the statically extracted program dependency graph to restore any non-serializable operations once inconsistency is detected during validation. By maintaining a thread-local access cache, the overhead for committing conflicting transactions is significantly reduced. Our experimental study confirms that transaction healing can scale near-linearly, yielding much higher transaction rate than the state-of-the-art OCC implementations.

Acknowledgment

We appreciate the reviewers' insightful comments. This research is supported in part by MOE Grant R-252-000-583-112. The source code is available at https://github.com/Cavalia/Cavalia.

8. REFERENCES

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

[2] https://github.com/apavlo/h-store.

[3] https://github.com/stephentu/silo.

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

[5] D. Agrawal, A. J. Bernstein, P. Gupta, and S. Sengupta. Distributed optimistic concurrency control with reduced rollback. *Distributed Computing*, 2(1), 1987.

[6] M. Alomari, M. Cahill, A. Fekete, and U. Röhm. The cost of serializability on platforms that use snapshot isolation. In *ICDE*, 2008.

[7] T. M. Austin and G. S. Sohi. Dynamic dependency analysis of ordinary programs. In *ISCA*, 1992.

[8] A. J. Bernstein, D. S. Gerstl, and P. M. Lewis. Concurrency control for step-decomposed transactions. *Information Systems*, 24(8), 1999.

[9] P. A. Bernstein and S. Das. Scaling optimistic concurrency control by approximately partitioning the certifier and log. *IEEE Data Eng. Bull*, 38(1), 2015.

[10] P. A. Bernstein, S. Das, B. Ding, and M. Pilman. Optimizing optimistic concurrency control for tree-structured, log-structured databases. In *SIGMOD*, 2015.

[11] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. 1987.

[12] P. A. Bernstein, C. W. Reid, and S. Das. Hyder-a transactional record manager for shared flash. In *CIDR*, 2011.

[13] P. A. Bernstein, C. W. Reid, M. Wu, and X. Yuan. Optimistic concurrency control by melding trees. In *VLDB*, 2011.

[14] C. Blundell, A. Raghavan, and M. M. Martin. Retcon: transactional repair without replay. In *ISCA*, 2010.

[15] A. Cheung, S. Madden, O. Arden, and A. C. Myers. Automatic partitioning of database applications. In *VLDB*, 2012.

[16] A. Cheung, S. Madden, and A. Solar-Lezama. Sloth: Being lazy is a virtue (when issuing database queries). In *SIGMOD*, 2014.

[17] A. Cheung, S. Madden, A. Solar-Lezama, O. Arden, and A. C. Myers. Using program analysis to improve database applications. *IEEE Data Eng. Bull.*, 37(1), 2014.

[18] J. Cowling and B. Liskov. Granola: low-overhead distributed transaction coordination. In *USENIX ATC*, 2012.

[19] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. In *VLDB*, 2010.

[20] S. Das, D. Agrawal, and A. El Abbadi. Elastras: An elastic, scalable, and self-managing transactional database for the cloud. *TODS*, 38(1), 2013.

[21] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *SIGMOD*, 2013.

[22] B. Ding, L. Kot, A. Demers, and J. Gehrke. Centiman: Elastic, high performance optimistic concurrency control by watermarking. In *SoCC*, 2015.

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

[24] J. M. Faleiro and D. J. Abadi. Rethinking serializable multiversion concurrency control. In *VLDB*, 2015.

[25] J. M. Faleiro, A. Thomson, and D. J. Abadi. Lazy evaluation of transactions in database systems. In *SIGMOD*, 2014.

[26] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *TODS*, 8(2), 1983.

[27] H. Garcia-Molina and K. Salem. Sagas. In *SIGMOD*, 1987.

[28] M. Herlihy. Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *TODS*, 15(1), 1990.

[29] T. Horikawa. Latch-free data structures for dbms: design, implementation, and evaluation. In *SIGMOD*, 2013.

[30] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-mt: A scalable storage manager for the multicore era. In *EDBT*, 2009.

[31] H. Jung, H. Han, A. D. Fekete, G. Heiser, and H. Y. Yeom. A scalable lock manager for multicores. In *SIGMOD*, 2013.

[32] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: A high-performance, distributed main memory transaction processing system. In *VLDB*, 2008.

[33] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *ICDE*, 2011.

[34] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *TODS*, 6(2), 1981.

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

[36] J. Levandoski, D. Lomet, S. Sengupta, R. Stutsman, and R. Wang. Multi-version range concurrency control in deuteronomy. In *VLDB*, 2015.

[37] J. J. Levandoski, P.-A. Larson, and R. Stoica. Identifying hot and cold data in main-memory databases. In *ICDE*, 2013.

[38] H. Litz, D. Cheriton, A. Firoozshahian, O. Azizi, and J. P. Stevenson. Si-tm: reducing transactional memory abort rates through snapshot isolation. In *ASPLOS*, 2014.

[39] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *EuroSys*, 2012.

[40] C. Mohan. Aries/kvl: A key-value locking method for concurrency control of multiaction transactions operating on b-tree indexes. In *VLDB*, 1990.

[41] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting more concurrency from distributed transactions. In *OSDI*, 2014.

[42] N. Narula, C. Cutler, E. Kohler, and R. Morris. Phase reconciliation for contended in-memory transactions. In *OSDI*, 2014.

[43] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. In *VLDB*, 2010.

[44] I. Pandis, P. Tözün, R. Johnson, and A. Ailamaki. Plp: page latch-free shared-everything oltp. In *VLDB*, 2011.

[45] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *SIGMOD*, 2012.

[46] K. Ramachandra, R. Guravannavar, and S. Sudarshan. Program analysis and transformation for holistic optimization of database applications. In *SOAP*, 2012.

[47] H. E. Ramadan, C. J. Rossbach, and E. Witchel. Dependence-aware transactional memory for increased concurrency. In *MICRO*, 2008.

[48] K. Ren, A. Thomson, and D. J. Abadi. Lightweight locking

for main memory database systems. In *VLDB*, 2012.

[49] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. Transaction chopping: Algorithms and performance studies. *TODS*, 20(3), 1995.

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

[51] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *TODS*, 4(2), 1979.

[52] A. Thomasian. Distributed optimistic concurrency control methods for high-performance transaction processing. *TKDE*, 10(1), 1998.

[53] A. Thomson and D. J. Abadi. The case for determinism in database systems. In *VLDB*, 2010.

[54] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *SIGMOD*, 2012.

[55] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, 2013.

[56] Z. Wang, H. Qian, J. Li, and H. Chen. Using restricted transactional memory to build a scalable in-memory database. In *EuroSys*, 2014.

[57] Y. Wu, C.-Y. Chan, and K.-L. Tan. Transaction healing: Scaling optimistic concurrency control on multicores (extended version). In *CoRR*, 2016.

[58] Y. Wu, W. Guo, C.-Y. Chan, and K.-L. Tan. Parallel database recovery for multicore main-memory databases. In *CoRR*, 2016.

[59] C. Yao, D. Agrawal, P. Chang, G. Chen, B. C. Ooi, W.-F. Wong, and M. Zhang. Dgcc: A new dependency graph based concurrency control protocol for multicore database systems. In *CoRR*, 2015.

[60] P. S. Yu and D. M. Dias. Analysis of hybrid concurrency control schemes for a high data contention environment. *TSE*, 18(2), 1992.

[61] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. In *VLDB*, 2014.

[62] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *SOSP*, 2013.

APPENDIX

A. BENCHMARKS

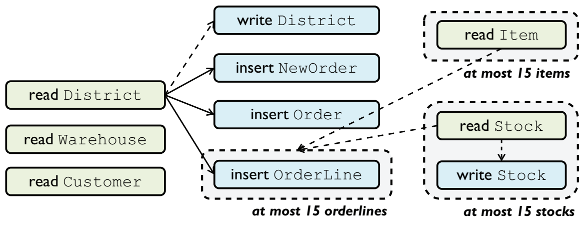
In our experimental study, we use two benchmarks, namely, TPC-C and Smallbank, to evaluate the system performance. We briefly describe these benchmarks here.

TPC-C. This benchmark is a widely-recognized industry standard for evaluating the performance of OLTP workloads. It contains nine tables and simulates a warehouse-centric order processing application. Three out of the five stored procedures, namely, `NewOrder`, `Delivery`, and `Payment`, contain write operations. The workload contention is controlled by the number of warehouses, and smaller number of warehouses indicates higher level of contention. The ratio of cross-partition transactions is controlled by changing the probability of accessing remote warehouses in the `NewOrder` transactions.

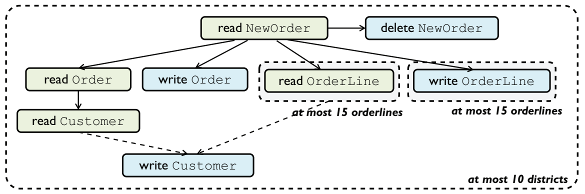
Smallbank. The Smallbank benchmark models a simple banking application containing three tables and six stored procedures. Transactions in this benchmark perform simple read and update operations on customers' accounts. Only a small number of items are involved in each transaction. The workload contention is controlled by a parameter θ, which indicates the skewness of the Zipfian distribution.

B. PROGRAM DEPENDENCY GRAPH

Figure 15a and Figure 15b respectively show the program dependency graphs for `NewOrder` transactions and `Delivery` transactions. In these two figures, solid lines represent key dependencies, while dashed lines represent value dependencies. Compared with `NewOrder` transactions, `Delivery` transactions access more data records, and the dependency relations among operations are more complicated. Therefore, healing the inconsistencies that are detected in `Delivery` transactions can require more efforts compared to `NewOrder` transactions.



(a) `NewOrder` transactions.



(b) `Delivery` transactions.

Figure 15: Program dependency graphs.

C. DURABILITY

This section first discusses how THEDB achieves system durability with minimum runtime overhead, and then demonstrates the efficiency of the adopted mechanisms through careful experiments.

C.1 Design

THEDB employs both checkpointing and value-logging during transaction execution to avoid loss of committed transaction updates in the event of machine failure or power outage. THEDB periodically checkpoints its in-memory database states to bound the maximum failure-recovery time. Meanwhile, all the update effects along with their timestamps are logged to the persistent storage, as is described in Section 4.3. In particular, transactions assigned with the same epoch number are persisted together, reducing the overhead that is caused by frequent storage access. Query result generated by a transaction is returned to the client only until the transaction's write effects have already been dumped to the underlying storage. Once machine failure occurs, THEDB reloads the most recent transaction-consistent checkpoint from the persistent storage. The reloading time is correlated to the checkpoint size and storage access speed, and parallelism can be achieved by using multiple storage devices or other new hardware. THEDB's logging mechanism ensures that log-recovery can be performed in parallel.

After reloading the corresponding log file to its local workspace, each thread re-instates the values of committed records concurrently. Specifically, each record in the database is attached with a timestamp indicating its last updater. When attempting to update a record with timestamp t_1, the write effect will be ignored if the updater's timestamp t_2 is smaller than t_1. The correctness of this approach is guaranteed by the Thomas write rule [51].

C.2 Experiments

This subsection evaluates the effectiveness of THEDB's durability. While the logging mechanism in a database system is largely influenced by the throughput of the underlying persistent storage, we focus on investigating whether the proposed commit protocol in transaction healing can bring any bottleneck that can bound the system performance. Figure 16 shows the logging overhead in THEDB when processing the TPC-C benchmark. Each thread directly dumps its logging data to an in-memory data structure, therefore, the generated result will not be influenced by the hardware (of the persistent devices like disk, SSD) throughput. As shown in the figure, the adopted value-logging mechanism yields similar transaction rate compared with command logging, indicating that the commit protocol in transaction healing is scalable. In fact, without considering the hardware throughput, the major overhead in THEDB's logging mechanism is memory allocation and string serialization, which can be addressed with sophisticated implementation. This result confirms that the logging mechanism in THEDB will not become a system bottleneck if provided with high-throughput persistent devices.

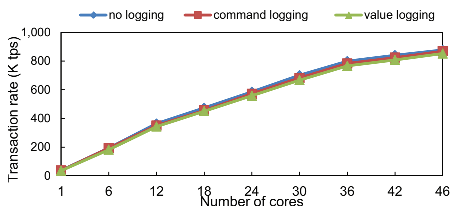


Figure 16: Transaction rate with different logging mechanisms. The number of warehouses is set to 12.

D. SYSTEM COMPARISON

In our experimental study, all the systems in comparison share exactly the same underlying framework. Therefore, the fairness of our experiments are guaranteed.

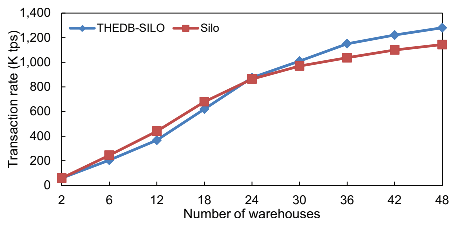


Figure 17: Performance comparison between THEDB-SILO and Silo. The number of cores is set to 46.

We adopted a self-implemented THEDB to compare transaction healing with the concurrency-control protocol that is proposed by Silo [55]. To confirm that our own implementation can precisely capture the behavior of Silo, we show the transaction rate of THEDB and the open-sourced version of Silo [3] in Figure 17.

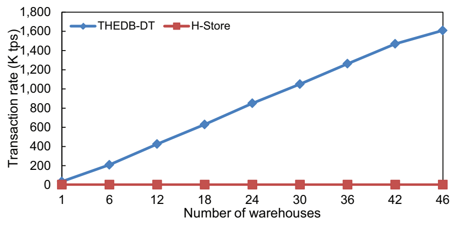


Figure 18: Performance comparison between THEDB-DT and H-Store. The number of cores is set to 46.

In this experiment, we set the number of cores to 46, and vary the number of warehouses to change the workload contention. As the result indicates, neither of these two systems can achieve a satisfactory performance when processing highly contended workloads. Meanwhile, the transaction rates of both systems scale smoothly with the decrease of workload contention. When the number of warehouses is set to 48, THEDB-SILO achieves around 10% higher transaction rate compared to Silo. This result confirms that our implementation of THEDB-SILO is efficient, and the runtime behavior of Silo's concurrency-control protocol can be precisely captured.

Next, we show a performance comparison between THEDB-DT and H-Store [32], which is a deterministic database that resorts to coarse-grained partition-level locking for achieving high performance. THEDB-DT faithfully follows the design of H-Store, and is specifically optimized for multicore architecture. In our experimental study, we did not directly use H-Store to measure the performance of deterministic databases. This is because H-Store cannot achieve satisfactory performance on multicore machines, and it suffers from additional performance penalty from network communication. Figure 18 shows a comparison between THEDB-DT and the open-sourced version of H-Store [2]. In this experiment, the ratio of cross-partition transactions is set to 0. That is, the workload can be perfectly partitioned. By varying the number of warehouses from 1 to 46, the transaction rate of THEDB-DT increases linearly. However, H-Store only achieves around 4.8 K tps regardless of the number of warehouses. This is because the performance of H-Store is strictly bounded by the network client. Therefore, we adopt the self-implemented THEDB-DT in our study.

E. TRANSACTION LATENCY

In this section, we analyze the transaction latency of THEDB when processing low-contention workloads. Table 5 shows the transaction latencies of different systems when processing `NewOrder` and `Delivery` transactions.

Similar with the results achieved when processing high-contention workloads, THEDB still yields a shorter latency compared with THEDB-OCC, THEDB-SILO, and THEDB-2PL. Specifically, around 95% of `NewOrder` and `Delivery` transactions processed by THEDB are committed within 80 ns and 640 ns, respectively. While the generated latency is much shorter than that achieved under high-contention workloads (see Section 5.2.2), THEDB-OCC, THEDB-SILO, and THEDB-2PL yet cannot maintain a stable transaction latency, and the produced latency can range from less than 20 ns to over 640 ns. In fact, the reported result generated by THEDB is very close to that achieved by THEDB-OCC and THEDB-SILO with the validation phase disabled (denoted as THEDB-OCC$^-$ and THEDB-SILO$^-$). This result confirms that THEDB can achieve high performance when processing low-contention workloads.

Transaction type	Latency (µs)	THEDB	THEDB-OCC	THEDB-SILO	THEDB-2PL	THEDB-OCC$^-$	THEDB-SILO$^-$
`NewOrder`	10 - 20	0.9%	2.0%	9.0%	4.4%	2.1%	9.8%
	20 - 40	42.4%	37.4%	47.9%	38.6%	50.2%	53.7%
	40 - 80	54.2%	47.3%	36.9%	44.2%	45.3%	35.2%
	80 - 160	2.4%	12.2%	4.9%	11.9%	1.9%	1.2%
	160 - 320	0%	1.0%	1.3%	0.7%	0.2%	0%
	320 - 640	0%	0%	0%	0%	0.1%	0%
	640 - INF	0%	0%	0%	0.1%	0%	0%
`Delivery`	10 - 80	0%	1.5%	1.3%	4.2%	0%	0%
	80 - 160	0%	0%	0%	0%	0.7%	0.5%
	160 - 320	65.7%	28.2%	54.9%	40.3%	83.3%	74.3%
	320 - 640	29.0%	50.7%	29.8%	49.6%	14.9%	24.9%
	640 - 1280	5.2%	18.3%	13.4%	3.7%	0.8%	0.3%
	1280 - 2560	0%	0.5%	0.5%	2.2%	0.3%	0%
	2560 - 5120	0%	0.8%	0%	0%	0%	0%
	5120 - INF	0%	0%	0%	0%	0%	0%

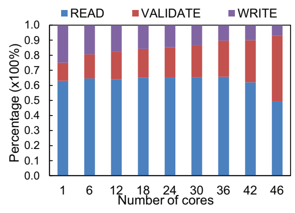
Table 5: Transaction latency for TPC-C benchmark. The number of warehouses is set to 24, and the number of cores is set to 46.

F. RUNTIME BREAKDOWN

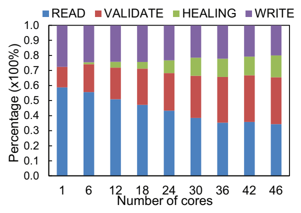
Figure 19 depicts the time breakdown of THEDB and THEDB-OCC using the TPC-C benchmark. In this set of experiments, we set the number of warehouses to 4 in order to generate contended workloads.

As shown in Figure 19a, THEDB-OCC spends respectively over 60% and 20% of its runtime in executing the read and write phases when the number of cores is set to 1. However, with 46 cores utilized in processing the workload, over 40% of the time is taken by the execution of the validation phase. This is because, under highly contended workloads, more transactions fail the validation phase due to inconsistent reads, and consequently much time is wasted in performing resource cleanups, such as lock releasing and memory deallocation. As a result, the transaction rate (i.e., the proportion of transactions that successfully completed the write phase) is reduced as contention increases.

Figure 19b shows the runtime breakdown of THEDB using the same workload configurations. As the number of cores increases, the proportion of time spent on the healing phase also increases. However, the percentage of time spent on the write phase is still maintained at around 20%, as most of the transactions that pass the validation phase can be committed. This result confirms the effectiveness of transaction healing when processing highly contended workloads.



(a) THEDB-OCC.



(b) THEDB.

Figure 19: Breakdown of transaction processing time using the TPC-C benchmark.

G. ABORT RATE

he following experiment exploits the performance improvement brought by validation-order rearrangement using the TPC-C benchmark. While we can hardly control the likelihood of deadlock occurrence, we can measure the *worst case* scenario where validation-order rearrangement is disabled. To construct such a scenario, we directly reverse the validation order. That is, instead of validating the consistency of the records following the table order: `Warehouse` → `District` → `Customer` → ... (shown in Figure 7, as is done when validation-order rearrangement is enabled), THEDB validates the consistency of the accessed records in a reversed order: ... → `Customer` → `District` → `Warehouse`. This is essentially the worst case, as once any update to the read-/write set's membership occurs, the healing phase is more likely to trigger transaction abort for the prevention of deadlocks. Figure 20 shows the result. Even under the worst case (see THEDB-W), THEDB still achieves 2X higher transaction rate compared to THEDB-OCC, especially when the workload is highly contended. With validation-order rearrangement enabled, THEDB's performance is improved by around 25%. This result confirms the effectiveness of order rearrangement. Moreover, the performance benefits brought by transaction healing is further confirmed.

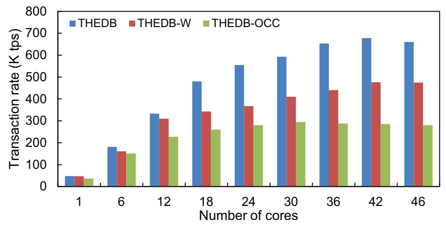


Figure 20: Performance improvement brought by rearranging validation orders.

Table 6 exhibits the abort rate caused by deadlock-prevention strategy in the transaction healing protocol. With order rearrangement disabled, in the worst case, the abort rate in THEDB can raise to 0.16 with the number of cores set to 46. However, with order rearrangement enabled, the abort rate is drastically dropped to less than 0.01 even under highly contended workloads. This result further confirms the effectiveness of validation-order rearrangement.

#cores	8	16	24	32	40	46
THEDB	0.0009	0.0017	0.0019	0.0021	0.0023	0.0024
THEDB-W	0.01	0.03	0.09	0.11	0.14	0.16

Table 6: Transaction abort rate caused by deadlock prevention.