BCC: Reducing False Aborts in Optimistic Concurrency Control with Low Cost for In-Memory Databases

Yuan Yuan[1], Kaibo Wang[1], Rubao Lee[1], Xiaoning Ding[2],
Jing Xing[3], Spyros Blanas[1], Xiaodong Zhang[1]

[1]The Ohio State University
[2]New Jersey Institute of Technology
[3]Institute of Computing Technology, Chinese Academy of Sciences

ABSTRACT

The Optimistic Concurrency Control (OCC) method has been commonly used for in-memory databases to ensure transaction serializability — a transaction will be aborted if its read set has been changed during execution. This simple criterion to abort transactions causes a large proportion of false positives, leading to excessive transaction aborts. Transactions aborted false-positively (i.e. false aborts) waste system resources and can significantly degrade system throughput (as much as 3.68x based on our experiments) when data contention is intensive.

Modern in-memory databases run on systems with increasingly parallel hardware and handle workloads with growing concurrency. They must efficiently deal with data contention in the presence of greater concurrency by minimizing false aborts. This paper presents a new concurrency control method named Balanced Concurrency Control (BCC) which aborts transactions more carefully than OCC does. BCC detects data dependency patterns which can more reliably indicate unserializable transactions than the criterion used in OCC. The paper studies the design options and implementation techniques that can effectively detect data contention by identifying dependency patterns with low overhead. To test the performance of BCC, we have implemented it in Silo and compared its performance against that of the vanilla Silo system with OCC and two-phase locking (2PL). Our extensive experiments with TPC-W-like, TPC-C-like and YCSB workloads demonstrate that when data contention is intensive, BCC can increase transaction throughput by more than 3x versus OCC and more than 2x versus 2PL; meanwhile, BCC has comparable performance with OCC for workloads with low data contention.

1. INTRODUCTION

The rapid increase of memory capacity has made it possible to store the entire OLTP database in the memory of one single server. With memory-resident data, database's performance bottleneck has shifted from disk I/O to software related overhead such as locking and buffer management [15, 9]. This has triggered the re-design of the database systems for the in-memory data. Of the concurrency control methods that have a great impact on database performance, Optimistic Concurrency Control (OCC) [17] has been favored by recent in-memory databases for high performance and scalability [18, 8, 30, 31, 33, 21].

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.
Proceedings of the VLDB Endowment, Vol. 9, No. 6
Copyright 2016 VLDB Endowment 2150-8097/16/02.

With the OCC method, a database executes each transaction in three phases: read, validation, and write. In the read phase, the database keeps track of what the transaction reads into a read set and buffers the transaction's writes into a write set in the transaction's private storage. In the validation phase, the database validates the transaction's read set. If the transaction's read set has been changed, the transaction must be aborted. Otherwise, the transaction proceeds to the write phase, in which the database installs the transaction's writes to the database storage. The validation and write phases must be executed in the critical section.

OCC is *optimistic* in the read phase. It assumes that all the transactions can proceed concurrently. Transactions in the read phase cannot block the execution of other transactions. Being optimistic maximizes concurrency level, leading to high scalability and throughput. However, OCC becomes *pessimistic* in the validation phase. It excessively aborts transactions to ensure serializability. Some aborted transactions may not affect serializability because change in a transaction's read set is not a sufficient condition that the transaction schedule cannot be serialized. Based on the serializability theory, only transactions forming a cycle in their dependency graph cannot be serialized and should be aborted [13]. The paper refers to the transactions aborted false-positively as *false aborts*, to differentiate them from the transactions that actually violate the serializability requirement.

A false abort happens when a transaction is aborted by OCC (i.e. read set changed) but it meets the serializability requirement (i.e., not in a cycle in dependency graph). The differences between these two criteria can be illustrated with the following two transactions: T_1: r(A) w(B) and T_2: r(A) w(A). Figure 1 shows a schedule of T_1 and T_2. According to OCC's validation criterion, T_2 can successfully commit since its read set is not changed, while T_1 must be aborted since its read set has been changed by T_2. However, based on the serializability theory, since there is no cycle in the serialization graph (i.e. $T_1 \xrightarrow{rw} T_2$), both T_1 and T_2 should be committed. If both of them were allowed to commit, the database state would be the same as that after T_1 and T_2 execute serially. Since T_1 is aborted by OCC though it should be allowed to commit based on the serializability requirement, the abort is a false abort.

When data contention is low, aborts, as well as false aborts, are rare, being pessimistic in transaction validation will not cause serious performance issues. For example, in-memory OCC databases can achieve throughput of over 500,000 transactions per second un-

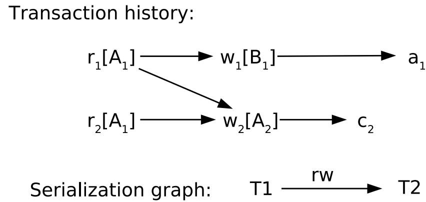


Figure 1: A schedule of the two transactions T_1 and T_2 (r:read, w:write, a:abort, c:commit) and the corresponding serialization graph. T_1 will be unnecessarily aborted by an OCC database.

der low-contention workloads [30, 31]. However, when data contention becomes intensive, an increasing number of transactions may be aborted false-positively. Data contention becomes intensive with the increase of CPU core counts. Data sets with skewed characteristics, e.g., those in OLTP workloads [28], also intensify contention. Based on our experiments, false aborts can reduce system throughput by 3.68x under a TPC-W-like workload. It is important for databases to provide good performance in both low contention and high contention scenarios.

To completely remove false aborts, database systems must abort transactions based on cycle detection in serialization graphs. The idea of detecting partial cycle dependency to guarantee serializability was first proposed by Cahill et al. [5] for disk-based snapshot isolation databases. However, the techniques are not directly applicable to in-memory databases where transactions are usually very short due to their prohibitive cost. Detecting cycles requires the database to operate on shared data structures such as wait-for graphs, which will significantly impact the system's scalability, especially for low contention workloads [6].

The dilemma lies between improving the validation with an accurate criterion to abort transactions and maintaining a low overhead for transaction execution. In this paper, we resolve the dilemma by proposing the **Balanced Concurrency Control (BCC)** method that seeks a sweet spot between being careful and fast. This balances the accuracy and the overhead of transaction validation well. Specifically, in addition to detecting the anti-dependency as OCC does, BCC detects one additional data dependency in a confined search space, which, together with the anti-dependency, forms an *essential dependency pattern*. This pattern more reliably indicates the existence of a cycle in the transaction dependency graph (i.e. unserializable transaction schedule) than OCC's criterion. We will show that by examining one additional dependency BCC can effectively reduce false aborts. At the same time, since BCC limits the search space for the additional dependency, the overhead for dependency detection can be effectively controlled through careful system design and implementation.

The paper makes the following contributions. First, it proposes a new concurrency control method for reducing false aborts while retaining OCC's merits for low contention workloads. Second, the paper proposes an optimized BCC method leveraging the state-of-the-art in-memory database features. Third, the paper studies implementation techniques for minimizing run-time overhead. Fourth, to demonstrate BCC's effectiveness, we implement it in Silo [30], which is a representative OCC-based in-memory database. Our implementation makes a case of how to adopt BCC in an OCC-based concurrency control kernel. Finally, we comprehensively evaluate BCC's performance on a 32-core machine. Our results demonstrate that BCC has a decisive performance advantage over OCC when contention becomes intensive. This advantage is due to a reduction in transaction aborts, and an increase in transaction throughput and workload scalability. Meanwhile, BCC has comparable performance with OCC for low contention workloads.

2. BALANCED CONCURRENCY CONTROL

BCC is an optimistic concurrency control method in nature. The key difference between BCC and other optimistic methods lies in the validation phase — how to determine if a transaction schedule is unserializable. In this section we first review the concepts of transaction history and data dependency in databases. Then we present BCC's transaction model and the essential dependency patterns that BCC utilizes in its validation to guarantee serializability. After that we explain how BCC detects the essential patterns and discuss BCC's overhead.

2.1 Background

Transaction history. A transaction history is an execution of database transactions which specifies a partial order of transactional operations on database tuples. Similar to previous work [4, 3], we use $r_i[x_j]$ to represent that transaction T_i reads the version j of tuple x, $w_i[x_j]$ to represent that T_i writes the version j of tuple x, c_i to represent that T_i is committed and a_i to represent that T_i is aborted. Given a tuple x's two versions, x_i is generated before x_j if $i < j$.

Data dependency. Data dependencies happen between transactions when they operate on the same tuple and at least one of the operations is write. The types of dependencies are determined by the operation type (read or write) and the order in which the transactions commit.

There are three types of data dependencies.

- Write-Read (*wr*) dependency: if transaction T_j reads a tuple that has been committed earlier by another transaction T_i, T_j is *wr* dependent on T_i, denoted as $T_i \xrightarrow{wr} T_j$.
- Write-Write (*ww*) dependency: if transaction T_j commits a tuple that has been committed earlier by another transaction T_i, T_j is *ww* dependent on T_i, denoted as $T_i \xrightarrow{ww} T_j$.
- Read-Write (*rw*) dependency: if transaction T_j commits a tuple that has been read earlier by another transaction T_i, T_j is *rw* dependent on T_i (or T_i is anti-dependent on T_j), denoted as $T_i \xrightarrow{rw} T_j$. Here, T_i has already started when T_j commits.

We use $T_i \rightarrow T_j$ to denote that T_j depends on T_i through any of the above dependency types.

2.2 Essential Dependency Patterns

BCC assumes the following transaction model.

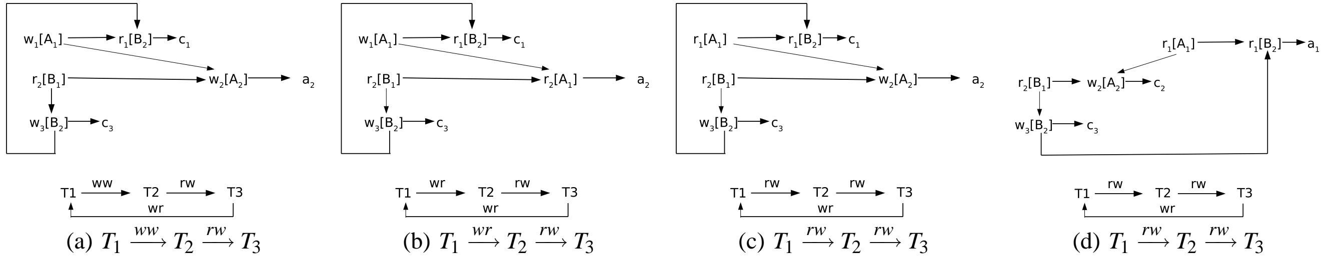
- Each transaction is executed in read, validation and write phases.
- Each transaction can only read committed tuples.
- The validation and write phases must be executed in the critical section.

Note that BCC and OCC [17] have the same transaction model.

In the validation phase, BCC exploits **essential dependency patterns** (or essential patterns for brevity) among transactions to determine unserializable transaction schedules. Each essential pattern specifies that certain data dependencies exist between transactions. We will demonstrate that the existence of an essential pattern is a *necessary* condition that a transaction schedule is unserializable in databases that satisfy BCC's transaction model. In this case, BCC ensures serializability by avoiding the essential patterns.

The essential dependency patterns that BCC detects and prevents are described as follows.

Theorem 1 *In databases that satisfy BCC's transaction model, when an unserializable transaction schedule is created, the schedule must*



(a) $T_1 \xrightarrow{ww} T_2 \xrightarrow{rw} T_3$ (b) $T_1 \xrightarrow{wr} T_2 \xrightarrow{rw} T_3$ (c) $T_1 \xrightarrow{rw} T_2 \xrightarrow{rw} T_3$ (d) $T_1 \xrightarrow{rw} T_2 \xrightarrow{rw} T_3$

Figure 2: Different types of essential patterns when an unserializable transaction schedule is created. The essential patterns from (a) to (c) are created when transaction T_2 commits. The essential pattern in (d) is created when transaction T_1 commits.

contain the following transactions T_1, T_2 and T_3 such that **(1)** *T_3 is the earliest committed transaction in the schedule;* **(2)** *$T_2 \xrightarrow{rw} T_3$; and* **(3)** *$T_1 \rightarrow T_2$ and T_1 commits after T_2 starts.* ***The data dependency patterns formed by T_1, T_2 and T_3 are called the essential patterns****.*

PROOF. When an unserializable transaction schedule is created, a cycle must exist in the transaction dependency graph. Let T_3 be the first transaction committed in the schedule. To form the cycle, T_3 must be dependent on another transaction (i.e. it should be pointed by an arrow in the dependency cycle). Since a transaction can only read committed tuples, this dependency cannot be a *ww* dependency or a *wr* dependency (otherwise there would exist another transaction in the schedule that committed earlier than T_3 committed, which contradicts with the fact that T_3 is the first committed transaction in the schedule). Thus, this dependency must be a *rw* dependency. Let the transaction T_3 *rw* dependent on be T_2 (i.e. $T_2 \xrightarrow{rw} T_3$). T_3 must commit after T_2 starts. To form the cycle, T_2 must also be dependent on a transaction in the cycle. Let the transaction be T_1 (i.e. $T_1 \rightarrow T_2$). T_1 must commit after T_2 starts, because T_3 commits after T_2 starts and T_1 commits later than T_3 commits. □

Theorem 2 *Transactions aborted by OCC may not be aborted by BCC, while transactions aborted by BCC will always be aborted by OCC.*

PROOF. The dependency $T_2 \xrightarrow{rw} T_3$ in the essential patterns is the anti-dependency detected by OCC, and the essential patterns examine additional dependencies to decide whether a transaction should be aborted. □

BCC utilizes the essential patterns to ensure serializability for three reasons. First, based on Theorem 1, validation based on detecting essential patterns only commits serializable transactions. Second, based on Theorem 2, validation based on detecting essential patterns reduces aborts compared to OCC. Third, the overhead of detecting the essential patterns can be effectively controlled by limiting the search space: BCC excludes all transaction T1 that commit before T2 starts.

2.3 Detection of Essential Patterns

A BCC database aborts a transaction if committing the transaction would create an essential pattern. To detect the essential patterns, the database needs to decide: (1) what data dependencies should be examined in each transaction's validation phase; and (2) what data dependency information should be kept for validating other transactions.

Figure 2 shows all possible essential patterns when an unserializable transaction schedule is created. The essential patterns can be divided into two categories based on when they would be created.

The first category contains three essential patterns that would be created at the time a transaction T_2 commits, which are shown in Figures 2(a) to 2(c). To detect these patterns, the database needs to validate if any transaction could be T_2 in the essential pattern by checking: (1) if the transaction T_2 is anti-dependent on a committed transaction T_3, or $T_2 \xrightarrow{rw} T_3$; and (2) if the transaction T_2 is *ww*-, *wr*- or *rw*-dependent on any concurrent transaction T_1, or $T_1 \xrightarrow{ww} T_2$, $T_1 \xrightarrow{wr} T_2$ or $T_1 \xrightarrow{rw} T_2$.

The second category only manifests with snapshot transactions that always operate on a consistent snapshot of the database. The essential pattern that would be created at the time snapshot transaction T_1 commits is shown in Figure 2(d). In this case, when T_1's snapshot time is before T_2's commit time and T_1's read operation happens after T_2 commits, the dependency $T_1 \xrightarrow{rw} T_2$ can only be detected when T_1 commits. To detect this pattern, the database needs to validate if any transaction could be T_1 in the essential pattern by checking if the transaction T_1 is a snapshot transaction and T_1 is anti-dependent on a committed transaction T_2 ($T_1 \xrightarrow{rw} T_2$), which is in turn anti-dependent on another committed transaction T_3 ($T_2 \xrightarrow{rw} T_3$). This requires the database to retain all anti-dependency information.

Algorithm 1 summarizes how BCC validates a transaction T to detect and prevent the essential patterns.

Algorithm 1 BCC's validation and write phases for a committing transaction T

```
1:  if T is anti-dependent on any committed transaction then
2:      record T's anti-dependency information;
3:      if T is wr-, ww-, or rw-dependent on any concurrent trans-
        action then
4:          abort T;
5:      end if
6:      if T is a snapshot transaction and there exists a transac-
        tion T' such that T is anti-dependent on T' and T' is anti-
        dependent on a committed transaction then
7:          abort T;
8:      end if
9:  end if
10: install T's writes and commit T;
```

BCC examines two data dependencies to detect the essential patterns. Theoretically, examining more dependencies can further reduce false aborts. However, the overheads will increase significantly. If detecting more dependencies, the dependencies can happen not only between concurrent transactions, but also between an active transaction and previously committed transactions that were concurrent with other active transactions. This makes the cost of checking dependency increase exponentially. Even disk-based

databases (e.g. PostgreSQL) avoid considering more than two dependencies [26] because of the high overhead [24].

3. AN OPTIMIZED BCC METHOD

In-memory databases can execute transactions as snapshot transactions. In some state of the art in-memory databases, such as [30, 31], read-only transactions are executed as snapshot transactions while write transactions are not. There are two reasons for this design. First, read-only transactions may be continuously aborted when running with other write transactions. Running them as snapshot transactions guarantees that they will never be aborted, although they may read stale data. Second, write transactions dominate the transactions executed by the database. Running them as snapshot transactions could introduce expensive operations like acquiring latches and locks for read operations in some concurrency control kernels, which will degrade the database's performance and scalability when the database has strived to avoid all centralized hotspots and scalability bottlenecks.

The BCC method requires the database to maintain a history of anti-dependency information to detect the essential pattern shown in Figure 2(d). If this is naively implemented, it can add a centralized hotspot to the in-memory database kernel and hurt BCC's scalability. The overhead is caused by the fact that when a transaction T_1 is a snapshot transaction, the dependency $T_1 \xrightarrow{rw} T_2$ may not exist when T_2 commits. In this case, to avoid BCC's overhead of maintaining historical dependency information, the database must guarantee that read-only snapshot transactions may never appear in any essential pattern.

Existing in-memory OCC databases avoid validating the snapshot transaction by taking an early snapshot time. However, this does not work in BCC. The reason is that the dependency cycle that is shown in Figure 2(d) can be created, with transaction T_1 being the snapshot transaction, no matter when the snapshot is taken. Based on BCC's validation criteria, both transactions T_2 and T_3 would be allowed to commit because no essential patterns are detected. To guarantee serializability, the database has to validate the snapshot transaction T_1, detect the essential pattern, and abort T_1.

We solve the problem by adding a light-weight synchronization point for snapshot transactions. The idea is that when a new read-only snapshot transaction begins, the database doesn't immediately start executing the snapshot transaction. Instead, it waits until all the active transactions are finished. During this period, no new transactions will be executed. After all active transactions have finished, the database takes a snapshot for the snapshot transaction and resumes executing transactions as normal.

With the above snapshot mechanism, a read-only snapshot transaction cannot become part of the essential patterns. This can be proved by contradiction. Assume that a read-only snapshot transaction could be part of the essential patterns. Since the snapshot transaction doesn't write any tuple, it cannot be T_3 in the essential patterns. Let the snapshot transaction be T_2. Then the essential pattern would be $T_1 \xrightarrow{wr} T_2 \xrightarrow{rw} T_3$. In this case, T_1 must commit before T_2 takes the snapshot, and T_3 must commit after T_2 takes the snapshot. This means T_1 commits earlier than T_3, which contradicts with the fact that T_3 is the first transaction committed in the dependency cycle. Let the snapshot transaction be T_3. Then essential pattern would be $T_1 \xrightarrow{rw} T_2 \xrightarrow{rw} T_3$. The snapshot transaction can only be *wr* dependent on another transaction to form a cycle. Let the transaction that T_1 is *wr* dependent on be T_0. T_0 must commit before T_1 starts. With our snapshot mechanism, T_2 and T_3 cannot start earlier than T_1. Thus T_0 must commit earlier than T_3, which contradicts with the fact that T_3 is the first committed transaction in the dependency cycle.

In this case, the database only needs to validate if a transaction can become the essential pattern's T_2 to guarantee serializability. The overhead of maintaining history anti-dependency information is avoided. Moreover, there is no need to maintain the read set of read-only snapshot transaction and validate it.

This optimization technique targets long-running read-only transactions where a short execution delay is acceptable. Users can always choose to revert to the original BCC protocol (and accept the additional overhead) if slight latency degradation is unacceptable.

4. DETAILED BCC IMPLEMENTATION

To support BCC in the database, two components must be implemented. One is a global clock to help detect data dependencies between concurrent transactions, the other is efficient management of the tuples accessed by each transaction. These two components are introduced in Section 4.1 and Section 4.2, respectively. Section 4.3 presents how to detect data dependency and Section 4.4 discusses phantom problems. In the last part of the section we explain how a BCC database executes transactions.

4.1 Global Clock

BCC needs a global clock to help decide if a data dependency should be considered as part of the essential pattern.

Our design of the global clock relies on the following in-memory database's features. First, to achieve good scalability, in-memory databases generate Transaction IDs (TIDs) in a decentralized way. For example, in Silo [30], which is a representative in-memory database, each TID can be divided into three parts: (1) *thread index*, which denotes the database thread that generated the TID; (2) the value of the database thread's *local counter*; and (3) the value of *global epoch*, which is a slowly advanced global timestamp in the database. A database thread can generate a TID by reading its local counter and the global epoch without synchronizations. One important property of the TIDs is that TIDs generated by the same database thread increase monotonically. However, this property doesn't hold for TIDs generated by different database threads. Second, each tuple in the database has an associated metadata recording the TID of the latest transaction that has written the tuple.

The global clock is designed as a global TID vector. The number of entries in the vector is the same as the number of available threads in the database. Each thread has a corresponding entry in the global clock, which records the thread's most recently assigned TID. A database thread must update its entry in the global clock every time it assigns a new TID.

The database can determine the order of a global clock value and a TID in two steps. The database first finds the database thread that generated the TID. Then it compares the value in the thread's global clock entry with the TID. The one with a smaller value happened first. The comparison process is shown in Figure 3.

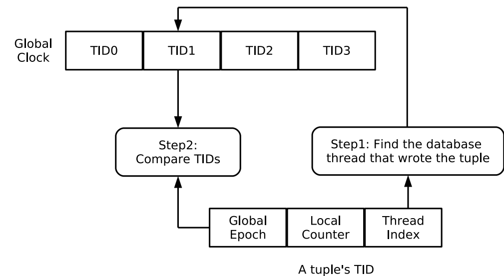


Figure 3: Comparison between the global clock and a tuple's TID.

The global clock is used in data dependency detections, which will be described in Section 4.3.

4.2 Transaction Data Management

BCC requires the database to keep track of each transaction's read set and write set to detect data dependencies. In the validation phase, the database checks if the transaction's read set has been changed and if the transaction's write set overlaps with other concurrent transactions' read sets. In this case, the transaction's write set can be simply kept in the database thread's local storage and will be released when the transaction finishes. On the other hand, the transaction's read set must be stored in the shared memory. Next we discuss how to efficiently manage the read sets.

Organization. We use hash table to organize the read set, since it may be searched by multiple database threads. A common approach is to use a shared hash table to store the tuples read by each transaction, but it requires synchronizations when accessing the hash table. For example, if two transactions read the same tuple, they will modify the same entry in the hash table, which must be synchronized. To ease the synchronization overhead, instead of maintaining one shared hash table across the database, each database thread maintains a separate hash table for each transaction. Each entry in the hash table contains a pointer to a tuple and the tuple's TID. A transaction's hash table must be kept in the memory until the transaction's concurrent peers have finished. Hash tables allocated by the same database thread are organized into a history list. Each entry in the list is a triple $< TID, Address, Release >$, where *TID* specifies which transaction the hash table belongs to; *Address* records the hash table's starting memory address; and *Release* determines when the hash table can be released.

Allocation and release. A database thread allocates a hash table when it starts a transaction T. It also allocates an entry in the history list to store the hash table's memory address and T's TID.

The hash table's *Release* is set with the maximum TID in the global clock after T finishes. In this way, any transaction that has a larger TID than *Release* of T's hash table must start after T finishes and is not concurrent with T.

To release T's hash table, the database thread must guarantee that all T's concurrent transactions have finished. This can be determined by checking if the minimum TID in the global clock is larger than the hash table's *Release*. If minimum TID in the global clock is larger, all the active transactions in the database must start after T finishes. The hash table can be safely released.

The release mechanism is conservative. A transaction T's hash table is not immediately released after T's concurrent transactions have finished. But it guarantees safe release of each hash table without synchronizations.

Synchronizations. Since a hash table can only be modified by one database thread and is not released until all concurrent transactions have finished, the only scenario that needs synchronization is while a thread is inserting into the hash table, another one is searching the hash table for *rw* dependency (i.e. $T_1 \xrightarrow{rw} T_2$). Without synchronization, an actually happened *rw* dependency may not be detected by the searching thread, which can cause serializability problem. Protecting the hash table with latch can solve the problem, but it harms scalability thus we avoid it.

We solve the problem by verifying the tuple after inserting it into the hash table. If any change has happened, a *rw* dependency may not be detected. Thus the database thread must discard the old tuple and re-read the tuple. This guarantees that either the *rw* dependency can be detected later, or the thread will read the newest tuple. The synchronization process is shown in Algorithm 2.

Algorithm 2 Hash table synchronization

```
1: read a tuple from database;
2: insert the tuple into the hash table ;
3: while true do
4:    read the same tuple from database;
5:    if tuple has changed after inserting into the hash table then
6:       discard the old tuple from hash table;
7:       insert the new tuple into the hash table;
8:    else
9:       break;
10:   end if
11: end while
```

Garbage collection. In the BCC database, a transaction T's hash table is kept in the database until T's concurrent transactions have finished. It is possible that some tuples that stored in T's hash table have been garbage collected by the database when a database thread searches T's hash table. This does not cause any problem because the hash table contains sufficient information (tuple's address and tuple's TID) to detect the data dependency. The search thread only needs to check if the tuple is in the hash table. There is no need to access the content of the tuple.

4.3 Data Dependency Detection

Detect anti-dependency. The anti-dependency is detected with OCC's criterion by checking whether T's read set has been changed.

Detect *wr* and *ww* dependencies. The database thread first takes a snapshot of the global clock when T starts and stores the value as *Start* in T's local memory. To detect *wr*, Every time T reads a tuple [1], the thread compares the tuple's TID with *Start* to decide if TID is generated after *Start*. If TID is generated later than *Start*, *wr* has happened. The *ww* dependency is detected after T enters the validation phase in a similar way. If any tuple in the transaction's write set has a TID that is generated later than *Start*, *ww* has happened.

Detect *rw* dependency. The database thread first takes a snapshot of the global clock when T starts and stores the value as *Start* in T's local memory. The thread takes another snapshot of the global clock after T enters the validation phase and stores the value as *End* in T's local memory. *Start* and *End* define the TID range of the concurrent transactions that T may be *rw* dependent on.

With *Start* and *End*, the database thread simply goes through other thread's history lists and check if T's write set overlaps with any hash table whose TID is generated later than *Start* but earlier than *End*. If there is overlap, *rw* has happened.

Since taking the snapshot of the global clock doesn't need synchronizations, it is possible that while the thread is taking the second snapshot, new transactions have started. These transactions may not be considered as concurrent with T. This will not cause any problem. The reason is that these new transactions cannot read the tuples in T's write set since T is in the critical section. Thus T cannot be *rw* dependent on them.

4.4 Phantom

Phantom problem can happen when a transaction is executing a range query while a concurrent transaction inserts a new tuple into the range. In the essential patterns, phantom can happen in two cases: (1) T_2 is the read transaction and T_3 is the insert transaction and (2) T_1 is the read transaction and T_2 is the insert transaction. The BCC database avoids phantom in the same way as recent in-memory OCC database (e.g, [30, 31]) does, which will abort the

[1] The delete operation only marks a tuple as deleted without removing the tuple for snapshot transactions.

read transaction if phantom happens. In the first case, phantom will be detected in the validation of T_2 and T_2 will be aborted. In the second case, there is no need to detect $T_1 \xrightarrow{rw} T_2$ in the validation of T_2 since T_1 will be aborted when validating T_1.

4.5 Put Together: A Transaction's Life

Algorithm 3 How a BCC database works in different phases

```
1:  Transaction Start:
2:  assign a TID and update the global clock;
3:  take a snapshot of the global clock;
4:  allocate a new hash table and release history hash tables;
5:
6:  Transaction Validation:
7:  enter the critical section;
8:  take a snapshot of the global clock;
9:  left_conflict = right_conflict = 0;
10: if there exists anti-dependency then
11:     right_conflict = 1;
12:     find concurrent transactions;
13:     if T is wr, ww or rw dependent on its concurrent transactions
        then
14:         left_conflict=1;
15:     end if
16: end if
17: if right_conflict == 1 and left_conflict == 1 then
18:     set the Release field of the transaction's hash table;
19:     abort the transaction;
20: else
21:     install the writes and commit the transaction;
22: end if
23: leave the critical section;
```

With the above designs, we now illustrate how a BCC database works in different transaction execution phases. The process is shown in Algorithm 3.

When a transaction T starts, the database first assigns T a new TID and updates the corresponding entry in the global clock. Then the database takes a snapshot shot of the global clock, which serves as multiple purposes. First, it is used to help determine the concurrent transactions for the later validation. Second, it is used to set the *Release* field of previous transaction's hash table with the maximum TID value in the global clock. Third, it is used to release unused history hash tables by finding the minimum TID in the global clock and releasing all hash tables whose *Release* are smaller than the minimum TID. The database also allocates a new hash table for the transaction.

When the transaction enters the validation phase, the database first takes a snapshot of the global clock, which can be used together with the clock taken in line 3 to determine concurrent transactions. Then the database checks if T is anti-dependent on any committed transaction. If no anti-dependency exist, T will be committed since no essential pattern will be created. Otherwise the database checks if T is dependent on any of its concurrent transactions. This requires the database to find all the transactions that are concurrent with T and check data dependencies between them. The data dependency is checked in the order of *wr*, *ww* and *rw*. If any data dependency is detected, the transaction will be aborted. The database will also set the *Release* of the aborted transaction's hash table such that the hash table can be immediately released. Otherwise the transaction will be committed.

5. EXPERIMENTAL METHOD

To evaluate BCC's effectiveness, we have implemented BCC and two phase locking (2PL) in Silo [30], which is a multi-threaded, shared in-memory OCC database. Silo generates TIDs in a decentralized way. It maintains a thread-local read-set and write-set for each transaction. Tuples in the transaction's write set are locked in a deterministic order before validation starts. After that, Silo assigns a TID to the transaction if the transaction writes to the database and validates the transaction using OCC's criterion.

5.1 Silo With BCC

Multi-Level Circular Buffers. Each thread in BCC database requires a memory space to store history hash tables, which may be allocated, released, and checked frequently. It is necessary to manage this memory space efficiently.

One way to manage the space is to organize it as a single region and use a free list to record the memory blocks available for new allocations. However, this approach would incur serious cache misses when each newly allocated hash table is filled with read sets. This problem can be addressed by utilizing two special characteristics of BCC thread's memory operations: (1) the hash tables are always released in the same order as they are allocated; (2) the memory demand of each thread for storing history hash tables is usually low, and only occasionally jitters to its maximum requirement (Section 6.3).

In our implementation, each thread partitions its memory space into three smaller areas that are managed with three levels of circular buffers. The lowest-level circular buffer is the smallest and can fit into the L1 CPU cache; the next one is slightly larger but is smaller than the L2 cache; the highest-level buffer is the largest one and can be any size that satisfies the maximum memory requirement of a thread. The database thread always tries to allocate memory from a lower-level circular buffer, and only resorts to a higher one when the lower buffer space becomes depleted. In each circular buffer, the memory is always allocated and released in a *chase-tail* fashion. Since most OLTP transactions are short and the average memory requirement of each database thread is low, this design ensures that most hash table operations can be satisfied in the L1 or L2 (or even L3) CPU cache.

Global clock. We implement the global clock as a set of sub-vectors. The number of sub-vectors equals to the number of CPU sockets and each sub-vector is a continuous array aligned on a single cache line on each socket.

TID generation. We use Silo's distributed TID generator to generate TIDs for every transaction. The original Silo only assigns TIDs to transactions that write to the database. We modify the TID generator such that every transaction will be assigned a TID. Every time a database thread generates a TID, it will update its entry in the global clock. For each database thread, we use the last TID generated by the thread to identify the current transaction's hash table since Silo generates TIDs in the validation phase.

Snapshot transactions. We add a synchronization point in the database before a snapshot transaction begins. After the synchronization, the database first advances the global epoch and then starts executing transactions. The snapshot is created based on the current Epoch value. Only tuples written in the previous Epoch can be read by the snapshot transactions.

5.2 Silo With 2PL

Our 2PL implementation is motivated by [25]. We avoid the centralized lock manager which generates suboptimal performance. Instead, we implement the per-tuple lock, and associate each tuple with a shared read lock and an exclusive write lock. No lock lists

are used. To avoid the deadlock detection overhead, we adopt the wait-die 2PL mechanism [27]. A global timestamp allocator, which is implemented as an atomic variable, assigns timestamp to each transaction to differentiate the precedence of transactions.

5.3 Experimental Setup

All experiments are conducted on a 32-core machine with four 2.13GHz Intel Xeon E7-8830 CPUs and 128GB memory. Hyperthreading is disabled to yield the best base performance of Silo [30]. The operating system is 64-bit Linux with 2.6.32 kernel. The version of GCC compiler is 4.8.2.

To avoid stalls due to user interaction, no network clients are involved in our experiments. Each database thread runs on a dedicated CPU core and has a local workload generator to generate input transactions for itself. Database logging is also disabled. All table data are resident in main memory and no disk activities are involved during each measurement. For each measurement, we run the experiment for 10 times, each lasting for 30 seconds, and the median results are reported.

6. EXPERIMENT RESULTS

In this section we present the performance results of BCC, OCC, and 2PL based on the prototype implementation in Silo. The experiment results confirm our expectations for BCC performance as follows:

- *BCC achieves comparable performance and scalability with OCC when the workload contention is low* (Section 6.1).
- *BCC significantly improves transaction throughput for high-contention workloads: BCC improves the throughput by 3.68x over OCC and by 2x over 2PL* (Section 6.2).
- *BCC's overhead on memory consumption and increased transaction latency is acceptable* (Section 6.3).

The performance results demonstrate BCC's usefulness in practice, which can provide good performance in both low contention and high contention scenarios.

6.1 Low Contention

We first evaluate how BCC performs when data contention is low. TPC-C [1] and YCSB [7] benchmarks are used in the experiments. Due to limited space, we only present TPC-C's results here. YCSB's results are similar.

TPC-C. TPC-C is an industry-standard benchmark for evaluating transaction database performance. It models the operations in a wholesale store that consists of a number of warehouses. In the Silo implementation, TPC-C tables are partitioned across the warehouses. We set the number of warehouses (i.e., the scale factor) to be the same with the number of database threads. In this configuration, each thread will mostly operate on the data items in its own warehouse, which makes the chance of data contention rare.

Figure 4 shows the transaction throughputs achieved by BCC, OCC and 2PL as the number of threads increases. As can be seen from the figure, BCC performs comparably and scales near-linearly as OCC does for the TPC-C workload. When there are 32 threads, BCC delivers an overall throughput of 1.15M transactions per second, which is only 7.29% lower than that achieved by OCC (1.24M). Despite the extra operations introduced by BCC for detecting the essential patterns, inter-core communication induced for checking history hash tables is rare when there are few data contentions. This makes BCC's overhead low, retaining OCC's performance benefits for low-contention workloads.

To better understand the causes of BCC's slight overhead compared to OCC, we further break down the slowdown of BCC at 32

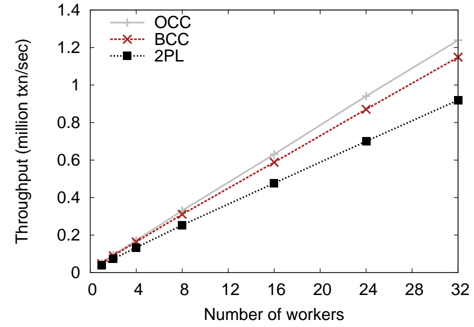


Figure 4: TPC-C transaction throughputs achieved by BCC, OCC and 2PL with up to 32 threads (cores). The scale factor is set to be the same with the number of threads.

Table 1: Breakdown of BCC overhead for TPC-C workload at low contention with 32 threads.

Operations	Contributions to slowdown (%)
Mm	4.76
Clock	2.42
Others	0.11

threads and show how different kinds of operations in BCC contribute to the throughput degradation. The result is listed in Table 1. It can be seen that the overhead mainly comes from two sources: memory management (Mm), which lowers the throughput by a delta of 4.76%, and accessing global clocks (Clock), which contributes 2.42% to the performance degradation. When the data contention is low, memory management operations mainly include bookkeeping the history list, and allocating and releasing memory for history hash tables. In our implementation each database thread uses a small memory region residing on local NUMA node to store the history list and hash tables. In this case no inter-core communication is needed for memory management operations. Since most OLTP transactions are short, the overhead due to memory management should remain almost constant regardless of the number of cores on the target platform.

On the other hand, the overhead of accessing the global clock is affected by the number of sockets on the machine. This is because the global clock in BCC is implemented as a distributed vector spread among the sockets. Each database thread needs to read all the distributed vectors at the beginning of a transaction, incurring inter-socket communication. This overhead is mainly determined by the number of sockets in the machine. However, since the number of sockets in a system is typically small, we believe this overhead (only 2.42% with 4 sockets) is acceptable in practice.

Compared to 2PL, BCC achieves better performance and scalability. With 32 threads, 2PL only delivers a throughput of 0.92M transactions per second, which is 19.9% and 25.8% lower than BCC and OCC respectively. In general, 2PL introduces extra overheads in two aspects. First, 2PL incurs extra locking operations for read operations compared to BCC and OCC. Each read operation needs to acquire and release a latch-protected read lock. Second, 2PL needs a centralized timestamp allocator to accurately determine the order of transactions to avoid deadlock, which becomes a bottleneck with the increase of number of the threads.

6.2 High Contention

This section compares the performance of BCC with OCC and 2PL when data contention is high. A modified TPC-W [2], TPC-C

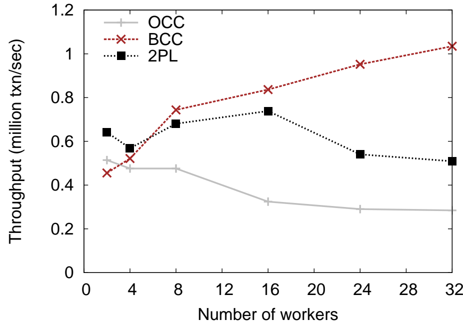


Figure 5: Throughput of modified TPC-W *DoCart* and *OrderProcess* transactions as the per-thread contention probability is 100% and the number of threads varies.

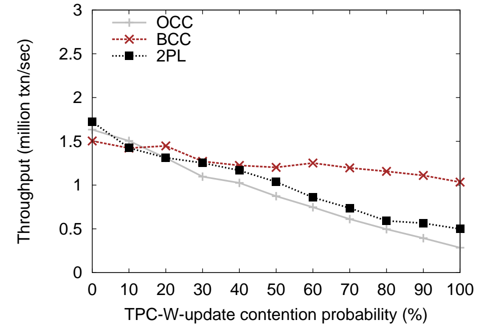


Figure 6: Throughput of TPC-W transactions with fixed 32 threads as per-thread contention probability varies.

and YCSB [7] are used in the evaluation.

TPC-W. TPC-W is a popular OLTP benchmark simulating the operations of an online bookseller. Compared with TPC-C, TPC-W has more complex read-only transactions. Since read-only transactions are executed as snapshot transactions which Silo never aborts with either OCC or BCC, their performance is similar under both concurrency control methods. We thus exclude them from our experiments with TPC-W, otherwise they would dominate the measured system throughput.

We experiment with the two update-intensive transactions from the TPC-W benchmark: (1) *DoCart* adds a set of random items to the shopping cart and displays the cart; (2) *OrderProcess* processes a set of random orders and updates the database (e.g., updating the stock numbers of the ordered items). To simulate the high contention scenario, we use slightly modified versions of the two transactions: there is one hot item in the orders processed by each OrderProcess transaction, and each *DoCart* transaction has a certain probability to display the hot item. In all our experiments, we let one database thread execute the *OrderProcess* transactions, while all other threads execute the *DoCart* transaction.

In our first experiment, we evaluate the performance of BCC, OCC and 2PL when the *DoCart* transaction has the highest contention probability with the *OrderProcess* transaction. We set the probability of DoCart adding the hot item to 100%, and measure transaction throughputs as the number of threads varies. The result is presented in Figure 5.

It can be seen that BCC scales much better than OCC in this experiment. As the number of threads increases, BCC gains increasingly higher performance advantage. With 32 threads, BCC achieves a throughput of 1.03M transactions per second, which is 3.68x over the throughput with OCC (0.28M).

The performance improvement achieved by BCC over OCC is mainly attributed to the reduction of *false-aborted DoCart* transactions. We can understand this conclusion from the following observations. First, there can be no data dependencies between two *DoCart* transactions because each *DoCart* only modifies its own private shopping cart. Second, the hot item displayed (read) by a *DoCart* transaction has a high probability of having been modified by an *OrderProcess* transaction when the *DoCart* transaction tries to commit. In this case, OCC must abort the *DoCart* transaction due to the appearance of a anti-dependency on a committed transaction. However, it is actually unlikely that a data dependency cycle would form because the rest tuple accesses in both *DoCart* and *OrderProcess* are random, making it a *false abort* to abort the *DoCart* transaction. BCC effectively reduces such false aborts by checking for one more data dependency besides the anti-dependency.

2PL performs differently. As can be seen, 2PL's throughput goes through three stages: decrease-increase-decrease. When using 2PL, *OrderProcess*'s throughput decreases as the number of threads increases because the hot tuple is more likely to be read locked by *DoCart*, which blocks *OrderProcess*. The reason for the decrease of 2PL's throughput when the number of threads increases from 1 to 2 is that the decreased throughput of *OrderProcess* is larger than the added throughput of *DoCart*. Note that 2PL's throughput with one database thread is *OrderProcess*'s throughput. As the number of threads increases from 2 to 16, 2PL's throughput increases. The reason is that *DoCart*'s throughput has increased and it outweighs the decrease of *OrderProcess*'s throughput. As the number of threads further increases, both *DoCart*'s throughput and *OrderProcess*'s throughput decrease because of the high contention. Thus 2PL's throughput decreases.

Compared to 2PL, BCC doesn't perform as well as 2PL when the number of threads is 4 or less. However, as the number of threads increases, BCC significantly outperforms 2PL. With 32 threads, the throughput of BCC is 2.03x over that of 2PL (0.5M). The performance differences are mainly determined by the relationship between 2PL's synchronization overhead and BCC's overhead of detecting essential patterns. With 2PL, the workloads are dominated by *DoCart* transactions when the number of threads is 32. 2PL has to synchronize between different threads because each one tries to add a read-lock to the hot tuple. In an optimized in-memory database, the synchronization cost is non-trivial. On the other hand, BCC's validation doesn't incur synchronizations for read contention.

The above experiment demonstrates how BCC performs under the highest intensity of per-thread contention. To understand how different contention intensity affects the transaction throughput, we fix the number of threads to 32 and vary the probability that *DoCart* adds the hot item to shopping cart. When the probability is 100%, it is the same with the previous experiment at 32 threads and the contention reaches the highest. When the probability is 0%, all items in a *DoCart* transaction are randomly chosen and the contention is the lowest. The result is shown in Figure 6.

Compared to OCC, BCC performs slightly lower when the contention probability is less than 10%, (by up to 7.95%) due to the overhead of shared memory management and inter-socket communication incurred by accessing the global clock. With the increase of contention probability, the throughput of OCC drops sharply, bottoming at only 285k transactions per second when the probability reaches 100%. On the other hand, BCC's throughput decreases at a much a slower rate. When the contention probability is 20% or greater, BCC's benefit of reducing false aborts outweighs its overhead for detecting the essential patterns, which improves the overall throughput.

We can see that 2PL has a similar performance trend with OCC,

although 2PL performs better than OCC. When the contention probability is less than 40%, 2PL has comparable performance with BCC. However, as the contention probability continuously increases, the performance of 2PL decreases much faster as that of BCC. This is because of 2PL's higher synchronization cost as we have discussed previously.

TPC-C. In the TPC-C experiments we use the update-intensive transactions, *NewOrder* and *Payment*, which comprises of most transactions in the TPC-C benchmark. We set the scale factor of TPC-C to 2 and the workload mix executed by each thread to {50%, 50%}. Figure 7 shows the throughput of this TPC-C workload as we increase the number of threads.

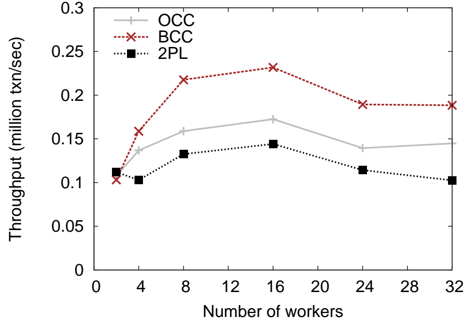


Figure 7: Throughput of TPC-C *NewOrder* and *Payment* with a workload mix of 50%-50% as the number of threads varies.

It can be seen that BCC outperforms OCC when the number of threads exceeds the number of warehouses. With up to 16 threads, the throughput of BCC and OCC both increase with the increase of thread number, but BCC scales better than OCC. BCC improves the throughput by 37% over OCC with 16 threads. As the number of threads further increases beyond 16 threads, the performance of both BCC and OCC start degrading with similar trends, but BCC still maintains good performance improvement (up to 35.8%) above OCC. This again confirms BCC's advantage over OCC through reducing false aborts.

The transactions *NewOrder* and *Payment* in TPC-C have much more complex data dependency patterns than the transactions *DoCart* and *OrderProcess* in TPC-W do. When operating on the same warehouse, all types of data dependencies can happen between any two concurrent transactions, each of which can be either *NewOrder* or *Payment*. Thus it is possible that a cycle would be created in the transaction dependency graph. For example, when two threads are executing the *Payment* transactions on the same warehouse, they both need to read and update the year-to-date payment, a dependency cycle containing *rw* and *wr* dependencies would likely be formed and thus one of the *Payment* transaction will be aborted by both BCC and OCC. This explains the performance decrease of both BCC and OCC when the contention becomes severe (with more than 16 threads).

BCC also performs better than 2PL. The throughput is improved by up to 1.84x. The poor performance of 2PL is mainly caused by its high synchronization overhead and the lock thrashing behavior. For example, when multiple database threads are executing *Payment* on the same warehouse, they need to acquire both read lock and write lock on the contended tuple. It is likely that the tuple is read locked by multiple threads thus only one thread can wait for the write lock while the rest are aborted. These aborted transactions cause unnecessary synchronization for others which limits the number of transactions processed to the database. We observe that with 32 threads, the total number of transaction throughput processed by 2PL (including both committed and aborted transactions) is 0.31M per second, which is significantly lower than that of BCC (0.9M).

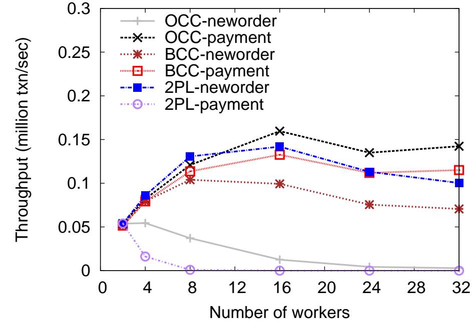


Figure 8: Breakdown of TPC-C throughput by transaction types with OCC, BCC and 2PL.

To better understand BCC's performance improvement over OCC and 2PL on TPC-C, we break down the overall throughput by the numbers contributed by different transaction types. The results are shown in Figure 8.

OCC, BCC and 2PL perform differently for *NewOrder* and *Payment*. OCC favors *Payment* transactions over *NewOrder* transactions while 2PL commits much more *NewOrder* transactions than *Payment* transactions. BCC's performance for *NewOrder* and *Payment* lie between.

Compared to OCC, BCC's performance advantage comes from the improved throughput of the *NewOrder* transaction. The reason is that many of the *rw* dependencies that happen between *NewOrder* and *Payment* that do not actually form a dependency circle, thus suffering false aborts with OCC. Examining additional dependency can greatly avoid the aborts and thus improve the overall throughput. However, BCC cannot improve the throughput of *Payment* transactions. It performs even worse than OCC. This is because each OCC-aborted *Payment* transaction is likely to reside in a dependency cycle with another transaction of the same type (i.e., *true abort*). Therefore the OCC-aborted *Payment* transactions will also be aborted by BCC. In this case, BCC's effort of examining additional dependencies only increases transaction execution latency, which in turn degrades the overall throughput.

On the other hand, BCC outperforms 2PL because it performs much better on the *Payment* transactions. With 32 threads, 2PL can barely commit *Payment* transactions. 2PL's poor performance on *Payment* is caused by the following two reasons. First, there is read write contention between *NewOrder* and *Payment* when they operate on the same warehouse. When the contended tuple is read locked by a NewOrder transaction, other *NewOrder* transactions can continue adding read locks to the tuple while *Payment* transaction has to wait. In this case, *Payment* transaction is likely to be aborted to avoid deadlock. Second, the contention two *Payment* transactions on the same warehouse cause aborts of the *Payment* transactions because they create dependency cycle.

YCSB. YSCB (Yahoo Cloud Serving Benchmark) benchmark [7] models the workload generated from online key-value and cloud serving stores. The benchmark contains a single table with ten String columns and populated with one million data items. Each transaction randomly accesses 16 tuples with each one having a 20% probability of being an update. Accesses to the tuples follow a Zipfian distribution. We set the conflict factor θ to 100 to make the level of data contention high. In this case, all types of data de-

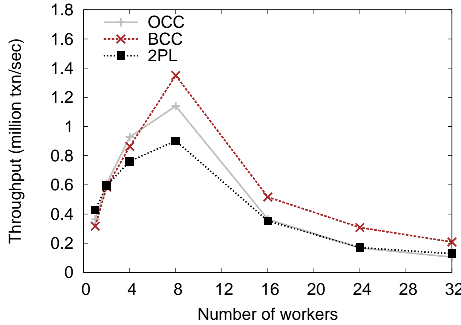


Figure 9: YCSB throughputs achieved with BCC, OCC and 2PL as the number of threads (cores) increases. The level of workload contention (θ) is set to 100.

pendencies can happen between two transactions. The results are shown in Figure 9.

As can be seen from Figure 9, BCC performs better than both OCC and 2PL when the number of thread is 8 or greater. With 32 threads, BCC's throughput is 1.99x over that of OCC and 1.63x over that of 2PL. The reason for the different performance behaviors is similar to the previous high contention benchmarks. BCC's performance improvement over OCC comes from BCC's reduction of unnecessary transaction aborts. On the other hand, BCC outperforms 2PL because of 2PL's high synchronization cost and lock thrashing behaviors.

6.3 Memory Consumption and Latency

BCC improves transaction throughput through detecting the essential patterns, with shared memory usage and extra operations. In this part we illustrate BCC's memory consumption and its impact on transaction latency.

Memory Consumption. With BCC, each thread maintains a memory area to store (1) a list of entries for recent history transactions, and (2) the hash tables of these transactions needed for detecting the essential patterns.

The size of saved history hash tables determines the memory consumption of BCC. For a given workload with a fixed number of threads, this overhead is usually stably low. Figure 10 shows the average and maximum sizes of memory occupied by history hash tables in each thread, executing the TPC-C *NewOrder* and *Payment* workload mix used in the previous subsection. It can be seen that the average per-thread memory consumption stays below 56KB consistently across all thread counts, with the maximum memory usage not exceeding 1.6MB. The high variance of the memory consumption between the average and the maximum is caused by the conservative hash table release mechanism, which achieves good performance but relies on the process of all database threads to determine when a hash table can be released. When a transaction T's hash table is released, T's concurrent transactions may have already finished for some time. Similar results are observed with other workloads used in our experiments as well.

In our experiments we set the number of entries in the history transaction list to 16K and the total size of memory for storing history hash tables to 4MB, which are more than enough for all the workloads encountered in our experiments. This amounts the total memory consumption of BCC with each thread to about 4.38MB, which is negligible compared with the tens of hundreds of gigabytes of memory present on a typical enterprise server. This also justifies BCC's design of using one hash table per transaction for the benefit of performance with low-contention workloads.

Latency. To illustrate how BCC affects transaction execution

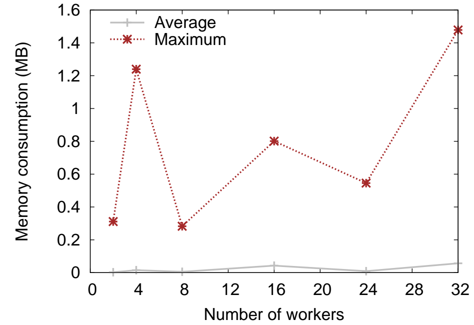


Figure 10: BCC memory consumption for saving history hash tables in each thread as the total number of threads varies for the 50%-50% workload mix of TPC-C *NewOrder* and *Payment* transactions.

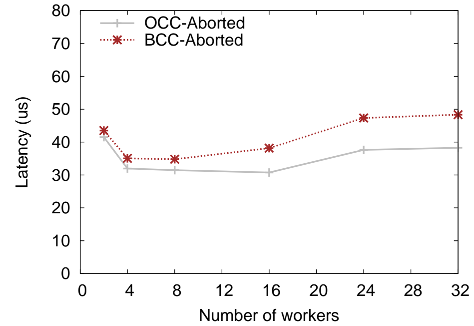


Figure 11: Latency of *NewOrder* transactions aborted by both OCC and BCC. The workload is the same with that used in Figure 10.

latency, we divide the *NewOrder* transactions processed with BCC in the previous experiment into the following three categories: (1) transactions that are committed with OCC's validation criterion; (2) transactions that are aborted even after BCC checks; and (3) transactions that are aborted with OCC but committed with BCC. These three types of transactions do not overlap.

For the first type of transactions, BCC's overhead mainly includes memory management and accessing the global clock, which are similar as the low contention workloads we discussed earlier in Section 6.1. For the second type of transactions, they are aborted by both OCC and BCC. Besides memory management and global clock operations, BCC performs extra data dependency checking before aborting a transaction. Figure 11 shows the total latency of each aborted *NewOrder* transaction in this case. BCC increases the latency by up to 26% with 32 threads. However, since the database needs to cleanup an aborted transaction for re-execution, this makes BCC's overhead negligible.

For the third type of transactions, BCC commits a transaction that would otherwise be aborted by OCC (i.e., BCC *saves* a transaction). This comes at the overhead of increased latency because the database thread needs to validate the transaction's write set with the history hash tables on other threads. Figure 12 shows the latency of transactions in this type, compared with transactions that OCC commits. As can be seen, the latency of transaction saved by BCC is almost twice as that committed by OCC. However, this overhead is acceptable for two main reasons. First, in high contention scenario, an OCC-aborted transaction may be aborted several times before it can actually commit. Considering the high cost of transaction re-executions, the latency of BCC-saved transactions is often justified. Second, the increased latency is still within tens of microseconds, which is sufficiently small for most real-world applica-

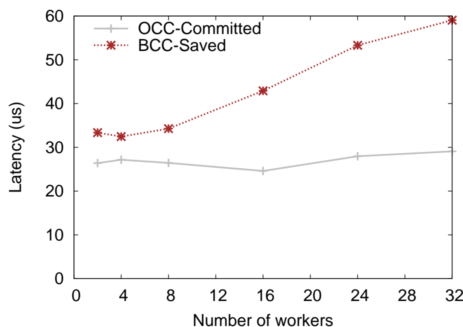


Figure 12: Latency of *NewOrder* transactions saved by BCC compared with that committed by OCC. The workload is the same with that used in Figure 10.

tions. We thus believe it is reasonable to trade the small increasing of latency for the high improvement of transaction throughput.

7. RELATED WORK

Partial dependency detection. The idea of detecting partial graph cycles to guarantee serializability was first proposed in the snapshot isolation concurrency control method [5, 11]. Snapshot isolation (SI) has been implemented in major database systems, such as Oracle and PostgreSQL. SI guarantees that read and write transactions won't block each other to increase system throughput. However, Fekete et al. [12] showed that SI could not guarantee transaction serializability, and Fekete et al. [11] further found a data dependency pattern (dangerous structure) that will always happen when transactions cannot be serialized in snapshot isolation (SI). Cahill et al. [5] demonstrated how to implement the dangerous structure in Berkeley DB. Ports et al. [24] further optimized this method for PostgreSQL. Han et al. [14] further optimized SI for multicore systems. BCC's essential patterns contain different data dependencies compared to the dangerous structure, which is caused by different record visibilities between SI and the optimistic concurrency control model. In SI a transaction cannot see writes which happen after the transaction starts. However, in BCC, any data dependency may exist between concurrent transactions. Moreover, BCC is designed and optimized for the short latencies that are encountered in main-memory OLTP workloads and not for disk-based implementations.

OCC for in-memory databases. The optimistic concurrency control (OCC) method was originally proposed by Kung and Robinson [17] and has been implemented in recent in-memory databases [18, 30, 31, 32]. These works mainly study how to efficiently implement the OCC method in in-memory databases. Larson et al. [18] proposed OCC for Microsoft SQL Server's in-memory OLTP engine "Hekaton" [8] and compared the performance of OCC with two-phase locking. Their implementation of OCC used a centralized timestamp allocator. Silo [30] introduced an implementation of OCC without centralized bottlenecks which can achieve near-linear scalability for low contention workloads. Tran et al. [29] studied transaction behaviors on hardware transactional memory. Wang et al. [31] explored how to build high performance OCC for in-memory databases with hardware transactional memory. Yu et al. [32] studied the scalability of different concurrency control methods on up to 1024 cores with a simulator. Despite the implementation differences among these databases, their OCC-based nature unavoidably causes spurious false aborts.

OCC for distributed systems. Recently OCC has also been studied in distributed systems. Maat [19] re-designed OCC for distributed systems and removed the need of locking during two-phase commit. ROCOCO [20] broke transactions into atomic pieces and executed them out of order by tracking dependencies, which significantly outperformed OCC. In comparison, BCC focuses on transaction execution for single-node in-memory databases.

OLTP on modern hardware. Our design and implementation benefits from existing in-memory OLTP systems. Databases such as Hyper [16] and H-Store [15] adopt the partitioning approach to scale. Harizopoulos et al. [15] analyzed the overheads of the Shore database. Pandis et al. [22] eliminated the overhead of centralized lock manager with partitioning. Porobic et al. [23] systemtically compared the performance of shared-nothing and shared-everyting OLTP system designs on multi-socket, multi-core CPUs. Faleiro et al. [10] redesigned the multiversion concurrency control method for in-memory databases by avoiding bookkeeping operations for read and global timestamp allocator, but it requires all the transactions to be submitted to the database before they can be processed.

Doppel [21] introduced an in-memory database designed for transactions that contend on the same data item. It proposed splitting the contended data item across cores such that each core can continue updating the data item in parallel. The per-core value was reconciled before the data item can be read. Doppel's optimization is orthogonal to BCC: Doppel improves performance when *ww* dependencies happen, while BCC avoids false aborts caused by *rw* dependencies.

8. CONCLUSION

In this paper we have presented the Balanced Concurrency Control (BCC) mechanism for in-memory databases. Unlike OCC that aborts a transaction based on whether the transaction's read set has changed, BCC aborts transactions based on the detection of essential patterns that will always appear in unserializable transaction schedules. We implemented BCC in Silo, a representative OCC-based in-memory database and comprehensively compared BCC with OCC and 2PL with TPC-W-like, TPC-C-like and YCSB benchmarks. Our performance evaluations demonstrate that BCC outperforms OCC by more than 3x and 2PL by more than 2x when data contention is high; meanwhile, BCC has comparable performance to OCC in low-contention workloads.

9. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their constructive comments. The work was supported in part by the National Science Foundation under grants OCI-1147522, CNS-1162165, and CCF-1513944.

10. REFERENCES

[1] TPC-C benchmark. `http://www.tpc.org/tpcc/`.

[2] TPC-W benchmark. `http://www.tpc.org/tpcw/`.

[3] A. Adya, B. Liskov, and P. O'Neil. Generalized isolation level definitions. In *Proceedings of the 16th International Conference on Data Engineering*, ICDE '00, pages 67–78, Washington, DC, USA, 2000. IEEE Computer Society.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[5] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 729–738, New York, NY, USA, 2008. ACM.

[6] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 1–17, New York, NY, USA, 2013. ACM.

[7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[8] 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 *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1243–1254, New York, NY, USA, 2013. ACM.

[9] X. Ding, S. Jiang, and X. Zhang. Bp-wrapper: A system framework making any replacement algorithms (almost) lock contention free. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ICDE '09, pages 369–380, Washington, DC, USA, 2009. IEEE Computer Society.

[10] J. M. Faleiro and D. J. Abadi. Rethinking serializable multiversion concurrency controll. *Proc. VLDB Endow.*, 8(11):1190–1201, 2015.

[11] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, June 2005.

[12] A. Fekete, E. O'Neil, and P. O'Neil. A read-only transaction anomaly under snapshot isolation. *SIGMOD Rec.*, 33(3):12–14, Sept. 2004.

[13] J. Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, UK, 1978. Springer-Verlag.

[14] H. Han, S. Park, H. Jung, A. Fekete, U. Röhm, and H. Y. Yeom. Scalable serializable snapshot isolation for multicore systems. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 700–711, 2014.

[15] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 981–992, New York, NY, USA, 2008. ACM.

[16] A. Kemper and T. Neumann. Hyper: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 195–206, Washington, DC, USA, 2011. IEEE Computer Society.

[17] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[18] P.-A. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow.*, 5(4):298–309, Dec. 2011.

[19] H. A. Mahmoud, V. Arora, F. Nawab, D. Agrawal, and A. El Abbadi. Maat: Effective and scalable coordination of distributed transactions in the cloud. *Proc. VLDB Endow.*, 7(5):329–340, Jan. 2014.

[20] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting more concurrency from distributed transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 479–494, Broomfield, CO, Oct. 2014. USENIX Association.

[21] N. Narula, C. Cutler, E. Kohler, and R. Morris. Phase reconciliation for contended in-memory transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 511–524, Broomfield, CO, Oct. 2014. USENIX Association.

[22] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *Proc. VLDB Endow.*, 3(1-2):928–939, Sept. 2010.

[23] D. Porobic, I. Pandis, M. Branco, P. Tözün, and A. Ailamaki. OLTP on hardware islands. *Proc. VLDB Endow.*, 5(11):1447–1458, July 2012.

[24] D. R. K. Ports and K. Grittner. Serializable snapshot isolation in PostgreSQL. *Proc. VLDB Endow.*, 5(12):1850–1861, Aug. 2012.

[25] K. Ren, A. Thomson, and D. J. Abadi. Lightweight locking for main memory database systems. *Proc. VLDB Endow.*, 6(2):145–156, Dec. 2012.

[26] S. Revilak, P. O'Neil, and E. O'Neil. Precisely serializable snapshot isolation (PSSI). In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 482–493, Washington, DC, USA, 2011. IEEE Computer Society.

[27] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, II. System level concurrency control for distributed database systems. *ACM Trans. Database Syst.*, 3(2):178–198, June 1978.

[28] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing. *Proc. VLDB Endow.*, 8:245–256, November 2014.

[29] K. Q. Tran, S. Blanas, and J. F. Naughton. On transactional memory, spinlocks, and database transactions. In R. Bordawekar and C. A. Lang, editors, *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2010, Singapore, September 13, 2010.*, pages 43–50, 2010.

[30] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, New York, NY, USA, 2013. ACM.

[31] Z. Wang, H. Qian, J. Li, and H. Chen. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 26:1–26:15, New York, NY, USA, 2014. ACM.

[32] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stronebraker. Staring into the abyss an evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.*, 8(3):209–220, Nov. 2014.

[33] W. Zheng, S. Tu, E. Kohler, and B. Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 465–477, Broomfield, CO, Oct. 2014. USENIX Association.