Releasing Locks As Early As You Can: Reducing Contention of Hotspots by Violating Two-Phase Locking

Zhihan Guo
University of Wisconsin-Madison
Madison, WI, USA
zhihan@cs.wisc.edu

Kan Wu
University of Wisconsin-Madison
Madison, WI, USA
kanwu@cs.wisc.edu

Cong Yan
Microsoft Research
Redmond, WA, USA
coyan@microsoft.com

Xiangyao Yu
University of Wisconsin-Madison
Madison, WI, USA
yxy@cs.wisc.edu

ABSTRACT

Hotspots, a small set of tuples frequently read/written by a large number of transactions, cause contention in a concurrency control protocol. While a hotspot may comprise only a small fraction of a transaction's execution time, conventional strict two-phase locking allows a transaction to release lock only after the transaction completes, which leaves significant parallelism unexploited. Ideally, a concurrency control protocol serializes transactions only for the duration of the hotspots, rather than the duration of transactions.

We observe that exploiting such parallelism requires violating two-phase locking. In this paper, we propose Bamboo, a new concurrency control protocol that can enable such parallelism by modifying the conventional two-phase locking, while maintaining the same guarantees in correctness. We thoroughly analyzed the effect of cascading aborts involved in reading uncommitted data and discussed optimizations that can be applied to further improve the performance. Our evaluation on TPC-C shows a performance improvement up to 4× compared to the best of pessimistic and optimistic baseline protocols. On synthetic workloads that contain a single hotspot, Bamboo achieves a speedup up to 19× over baselines.

CCS CONCEPTS

• **Information systems → Database transaction processing**.

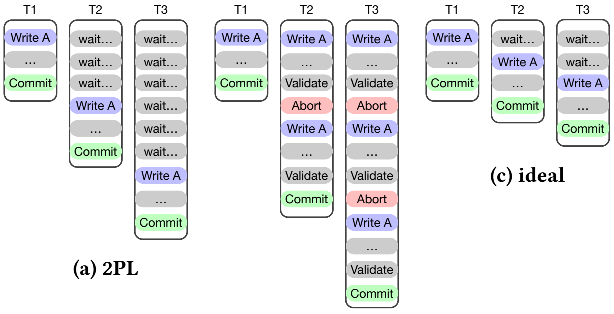
KEYWORDS

concurrency control; two-phase locking; hotspot; cascading abort

ACM Reference Format:
Zhihan Guo, Kan Wu, Cong Yan, and Xiangyao Yu. 2021. Releasing Locks As Early As You Can: Reducing Contention of Hotspots by Violating Two-Phase Locking. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3457294

SIGMOD '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3457294



(a) 2PL

(b) OCC

(c) ideal

Figure 1: Schedules of transactions with a hotspot A under 2PL, OCC, and an ideal case. ("Write" means read-modify-write)

1 INTRODUCTION

Modern highly contentious transactional workloads suffer from hotspots. A hotspot is one or a small number of database records that are frequently accessed by a large number of concurrent transactions. Conventional concurrency control protocols need to serialize such transactions in order to support strong isolation like serializability, even though the hotspot may comprise only a small fraction of a transaction's execution time. Figure 1 illustrates the effect using a single hotspot of tuple *A*. For both pessimistic (Figure 1a) and optimistic (Figure 1b) concurrency control, transactions wait or abort/restart at the granularity of entire transactions.

Ideally, we want a concurrency control protocol to *serialize transactions only for the duration of the hotspots* (e.g., in Figure 1c, transaction T2 can access the hotspot immediately after T1 finishes writing it) but execute the rest of the transactions in parallel. If the hotspot comprises only a small fraction of the transaction's runtime, such an ideal protocol can improve performance substantially.

Many production systems (MS Orleans [12], IBM IMS/VS [16], Hekaton [9, 28], etc.) and research work [3, 10, 14, 17, 21, 24, 26, 32, 36, 45, 46] mitigate hotspots by adding extra complication, but cannot achieve the ideal protocol mentioned above. In particular, the ideal protocol needs to read dirty data written by another transaction that has not committed yet. For pessimistic concurrency control, this violates the conventional definition of 2PL — T1 can acquire new locks after releasing locks to other transactions. For OCC and hybrid concurrency control protocols such as MOCC [42] and CormCC [38], a transaction makes its writes visible only after the execution finishes, which is inherently incompatible with the notion of accessing dirty writes early.

Transaction chopping [36, 44, 48] and its recent improvements (e.g., IC3 [43] and Runtime Pipelining [45]) are a line of research that tried to enable early reads of dirty data. Transaction chopping performs program analysis to decompose a transaction into sub-transactions and allows a sub-transaction to make local updates visible immediately after it finishes. While these techniques can substantially improve performance, they have several severe limitations. **First**, these methods require the *full knowledge of the workload* including the number of transaction templates and the columns/tables each transaction will access. Any new ad-hoc transaction incurs an expensive re-analysis of the entire workload. **Second**, chopping must follow specific criteria to avoid deadlocks and ensure serializability, which limits the level of parallelism can be potentially exploited (see Section 2.2 for details). **Third**, conservative conflict detections based on the limited information before execution can enforce unnecessary waiting. For example, in IC3, two transactions accessing the same column of different tuples may end up causing contention.

In this paper, we aim to explore the design space of allowing dirty reads for general database transactions without the extra assumptions made in transaction chopping. To this end, we propose **Bamboo**, a pessimistic concurrency control protocol allowing transactions to read dirty data during the execution phase (thus violating 2PL), while still providing serializability. Bamboo is based on the Wound-Wait variant of 2PL and can be easily integrated into existing locking schemes. It allows a transaction to *retire* its lock on a tuple after its last update on the tuple so that other transactions can access the data. Annotations of the last write can be provided by programmers or programming analysis. To enforce serializability, Bamboo tracks dependency of dirty reads through the lock table and aborts transactions when the dependency is violated.

One well-known problem of violating 2PL is the introduction of *cascading aborts* [3] — an aborted transaction causes all transactions that have read its dirty data to also abort. If not properly controlled, cascading aborts lead to a significant waste of resources and performance degradation. Through Bamboo, this paper explores the design space and trade-off of cascading aborts, evaluates its overhead, and proposes optimizations to mitigate these aborts. In summary, this paper makes the following contributions.

- We developed Bamboo, a new concurrency control protocol that violates 2PL to improve parallelism for transactional workloads without requiring the knowledge of the workload ahead of time. Bamboo is provably correct.
- We conducted a thorough analysis (both qualitatively and quantitatively) of the cascading abort effect, and proposed optimizations to mitigate such aborts.
- We evaluated Bamboo in the context of both *interactive transactions* and *stored procedures*. In TPC-C, Bamboo demonstrated a performance improvement up to 2× for stored procedures and 4× for interactive transactions compared to the best baseline (i.e., Wait-Die and Silo respectively). Bamboo also outperforms IC3 by 2× when the attributes of hotspot tuples in TPC-C are truly shared by transactions.

2 BACKGROUND AND MOTIVATION

Section 2.1 describes two-phase locking, with a focus on the Wound-Wait variant that Bamboo is based on. Section 2.2 discusses how transaction chopping mitigates the hotspot issue.

2.1 Two-Phase Locking (2PL)

Two-phase locking (2PL) is the most widely used class of concurrency control in database systems. In 2PL, reads and writes are synchronized through explicit locks in shared (*SH*) or exclusive (*EX*) mode. A transaction operates on a tuple only if it has become an "owner" of the corresponding lock.

2PL forces two rules in acquiring locks: 1) conflicting locks are not allowed at the same time for the same data; 2) a transaction cannot acquire more locks once it releases any [4].

The second rule requires every transaction obtaining locks to follow two phases: *growing phase* and *shrinking phase*. A transaction can acquire locks in the growing phase but will enter the shrinking phase if they ever releases a lock. In the shrinking phase, no more locks should be acquired. This rule guarantees serializability of executions by ensuring no cycles of dependency among transactions. (a more rigorous proof is included in the technical report [20]).

For a lock request that violates the first rule, 2PL may put the requesting transaction on the waiting queue until the lock is available. Two major approaches exist to avoid deadlocks due to cycles of waiting: *deadlock detection* and *deadlock prevention*. The former explicitly maintains a central *wait-for* graph and checks for cycles periodically. The graph becomes a scalability bottleneck with highly parallel modern hardware [47]. Deadlock prevention technique instead allows waiting only when certain criteria are met. Wound-Wait is a popular protocols under the category [5, 34].

Wound-Wait Variant of 2PL

In Wound-Wait, each transaction is assigned a timestamp when it starts execution; transactions with smaller timestamps have higher priority. When a conflict occurs, the requesting transaction T compares its own timestamp with the timestamps of the current lock owners — owners whose timestamps are bigger than T are aborted, namely `Wound`. Then T either becomes the new owner (i.e., all current owners are aborted) or waits for the lock (i.e., some owners remain), namely `Wait`. The lock entry for each tuple maintains `owners` and `waiters` as two lists of transactions that are owning or waiting for the lock on the tuple (Figure 2). `waiters` can be sorted based on the transactions' timestamps to simplify the process of moving transactions from `waiters` to `owners`.

Wound-Wait is deadlock-free because a transaction can only wait for other transactions that have smaller timestamps. In the wait-for graph, this means all the edges are from transactions with larger timestamps to transactions with smaller timestamps, which inherently prevents cycles. Besides deadlock-freedom, Wound-Wait is also starvation-free as the oldest transaction has the highest priority and will never abort due to a conflict. Wound-Wait is the concurrency control protocol used in Google Spanner [8, 30].

2.2 Transaction Chopping

Similar to Bamboo, transaction chopping [36] aims to increase concurrency when hotspots are present. In particular, it chops a transaction into smaller sub-transactions and allow an update to be visible after the sub-transaction finishes but before the entire transaction commits. In particular, an SC-graph is created based on static analysis of the workload, where each sub-transaction represents a node in the graph. Sub-transactions of the same transaction are connected by sibling (S) edges. Sub-transactions of different transactions are connected by conflict (C) edges if they have potential conflicts. Chopping requires no cycle in the graph and only the first piece can roll-back or abort. It obtains the finest chopping that can guarantee safeness based on static information.

IC3 [43] is the state-of-the-art concurrency control protocol in this line of research. IC3 achieves fine-grained chopping through column-level static analysis. Sub-transactions accessing different columns of the same table will no longer introduce C-edges. As IC3 allows cycles in the SC-graph, during runtime, it tracks dependencies of transactions and enforces pieces involving C-edges to execute in order to maintain serializability. Moreover, it proposes optimistic execution to enforce waiting just on validation and commit phases for non-conflicting transactions accessing same columns of different tuples. Although inducing more aborts, the optimistic approach still show advantages under high contention.

However, IC3 still has several limitations. First, it assumes column accesses of all transactions to be known before execution to identify possible conflicts in advance. Second, chopping must guarantee no crosses of C-edges to avoid potential deadlocks. For example, if one transaction accesses table A before B while the other accesses table B before A. The accesses of tables A and B must be merged into one piece, limiting concurrency. Third, column-level static analysis does not exploit the concurrency when transactions accessing same columns of different tuples; it helps reduce more contention only when transactions access different columns of the same tuple. Section 5.6 shows more quantitative evaluations.

3 BAMBOO

The basic idea of Bamboo is simple — in certain controlled circumstances, we allow other transactions to violate an exclusive lock held by a particular transaction. A transaction's dirty updates can be accessed after it has finished its updates on the tuple, following the idea shown in Figure 1c.

3.1 Challenges of Violating 2PL

Although violating 2PL offers great performance potential, it also brings two key challenges that we highlight below.

Challenge 1: Dependency Tracking

A conventional 2PL protocol uses locks to track dependencies among transactions. 2PL protocols use various techniques (cf. Section 2.1) to prevent/break a cycle in the dependency graph. We call an edge in a conventional dependency graph a *lock-induced* edge.

In contrast, Bamboo allows a transaction to read dirty value without waiting for locks. Such a *read-after-write* dependency can be part of a cycle and yet is not captured by a conventional lock. For example, T1 may read T2's dirty write on record A and T2 may read T1's dirty write on record B. Such a cycle is not captured by the "wait-for" relationship. We call such a dependency edge a *dirty-read-induced* edge.

For Bamboo to work efficiently, we need a new deadlock avoidance mechanism that can avoid cycles caused by both lock-induced and dirty-read-induced edges uniformly. Section 3.2 will describe the detailed protocol we build that achieves this goal.

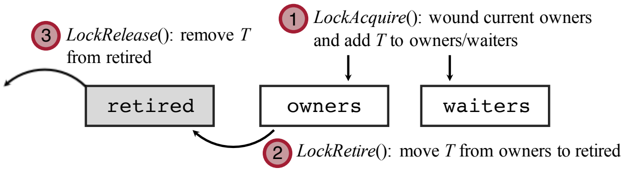

3 LockRelease(): remove T from retired
1 LockAcquire(): wound current owners and add T to owners/waiters
retired
owners
waiters
2 LockRetire(): move T from owners to retired

Figure 2: A lock entry in Bamboo — Transaction T moves between lists (i.e., owners, waiters, and retired) through function calls. The retired list does not exist in baseline Wound-Wait.

Challenge 2: Cascading Aborts

Allowing a transaction to read dirty data may lead to cascading aborts, as pointed out in multiple previous protocols [12, 28]. Specifically, if T2 reads T1's update before T1 commits, a commit dependency between the two transactions is established — T2 is able to commit only if T1 has successfully committed. If T1 decides to abort (e.g., due to conflicts or integrity violation) then all the transactions that have commit dependencies on T1 must also abort; this includes T2 and all the transactions that have read T2's dirty writes and so forth. This means potentially a long chain of transactions with commit dependencies need to cascadingly abort, causing waste of work and performance degradation. In Section 4, we present a deeper analysis of cascading aborts.

3.2 Protocol Description

This section describes the basic Bamboo protocol in detail. In particular, we focus on addressing the first challenge in Section 3.1 (i.e., dependency tracking). Bamboo is developed based on Wound-Wait (cf. Section 2.1); our description mainly focuses on the differences between the two. We firstly describe the new data structures Bamboo requires to track dependencies of dirty reads, followed by a detailed description of the pseudocode of the protocol.

3.2.1 Data Structures. All edges in the dependency graph of a conventional 2PL protocol are lock-induced edges. They are captured by locks and maintained in lock entries of individual tuples. For Bamboo, we try to uniformly handle both lock-induced and dirty-read-induced edges by adding extra metadata into each lock entry and transaction.

tuple.retired: Bamboo adds a new list called retired in each lock entry next to the existing lists of owners and waiters, as shown in Figure 2. retired is sorted based on the timestamps of transactions in it. After a transaction has finished updating a tuple, the transaction can be moved from owners to retired. This allows other transactions to join owners and read the dirty updates of the retired transactions. By maintaining the retired list, a dirty-read dependency can be captured in the lock entry — if a retired transaction T has an exclusive lock, then all transactions in retired after T and all transactions in owners depend on T. Adding retired allows both lock-induced and dirty-read-induced dependencies to be maintained in the lock entry.

Transaction.commit_semaphore: Bamboo uses a new variable commit_semaphore to ensure that transactions with dirty-read dependencies commit in order. A transaction T increments its own

Algorithm 1: A transaction's lifecycle in Bamboo — Differences between Bamboo and Wound-Wait are highlighted in gray.

```
   # req_type is SH or EX
1  LockAcquire(txn, req_type1, tuple1)
   ...
   # Lock can retire after the last write to the tuple
2  LockRetire(txn, tuple1)
3  LockAcquire(txn, req_type2, tuple2)
   ...
   # Wait for transactions that txn depends on
4  while txn.commit_semaphore $\neq$ 0 do
5      pause
6  if(!abort) writeLog() # Log to persistent storage device
7  LockRelease(txn, tuple1, is_abort)
8  LockRelease(txn, tuple2, is_abort)
9  txn.terminate(is_abort)
```

semaphore when it conflicts with any transaction in `retired` of any tuple. The semaphore is decremented only when the dependent transaction leaves `retired` so that `T` becomes one of the leading non-conflicting transactions in `retired`. The semaphore is implemented using a 64-bit integer for each transaction and incremented/decremented through atomic operations. The number of accesses to the semaphore is bounded by the number of tuple accesses of a transaction. The overhead of the semaphore is within 0.2% of the total execution time with 120 threads under a high-contention workload. Details of how this variable is operated will be discussed in the concrete protocol.

3.2.2 Locking Logic. Algorithm 1 shows the lifecycle of how the database executes a transaction with Bamboo. It largely remains the same as a conventional Wound-Wait 2PL protocol, with the differences highlighted using gray background color. For the basic protocol, we assume each transaction is assigned a timestamp when it first started similar to Wound-Wait; we will later optimize the timestamp assignment process in Section 3.5.

Different from conventional 2PL, Bamboo allows a transaction to immediately retire a tuple that will not be written again by the transaction (line 2); the transaction can still read the tuple since Bamboo keeps a local copy of the tuple for each read request. The transaction can acquire more locks on other tuples after retiring some lock (line 3). After the execution finishes, a transaction must wait for its `commit_semaphore` to become 0 before it can start committing. The transaction then moves forward to perform logging (line 6) and release locks (lines 7–8). Finally, the transaction either commits or aborts depending on the execution result. If an abort occurs during the transaction execution, the transaction directly jumps to line 7 to release locks.

Algorithm 2 shows the detailed implementation of the functions in Bamboo, i.e., *LockAcquire()*, *LockRetire()*, *LockRelease()*, as well as an auxiliary function *PromoteWaiters()* which is called by the other three functions. Note that the first three functions are in critical sections protected by latches, same as other 2PL protocols.

The baseline Wound-Wait is the algorithm in Algorithm 2 ignoring the code in gray. Bamboo adds extra logic to *LockAcquire()* and *LockRelease()*, and adds a new function *LockRetire()*. In the following, we walk through these functions step-by-step.

Algorithm 2: Function calls in Bamboo — Difference between Bamboo and Wound-Wait is highlighted in gray.

```
    tuple.retired # List of retired transactions ordered by ascending
      timestamp order
    tuple.owners # List of owners of the lock
    tuple.waiters # List of transactions waiting for the lock, sorted by
      ascending timestamp order
    # Each list above contains {txn, type} where type is either SH or EX

1   Function LockAcquire(txn, req_type, tuple)
2       has_conflicts = false
3       for (t, type) in concat(tuple.retired, tuple.owners) do
4           if conflict(req_type, type) then
5               has_conflicts = true
6           if has_conflicts and txn.ts < t.ts then
7               t.set_abort()
8       tuple.waiters.add(txn)
9       PromoteWaiters(tuple)

    # move txn from tuple.owners to tuple.retired
10  Function LockRetire(txn, tuple)
11      tuple.owners.remove(txn)
12      tuple.retired.add(txn)
13      PromoteWaiters(tuple)

14  Function LockRelease(txn, tuple, is_abort)
15      all_owners = tuple.retired ∪ tuple.owners
16      if is_aborted and txn.getType(tuple) == EX then
17          abort all transactions in all_owners after txn
18      remove txn from tuple.retired or tuple.owners
19      if txn was the head of tuple.retired and
          conflict(txn.getType(tuple), tuple.retired.head) then
            # heads: leading non-conflicting transactions
            # Notify transactions whose dependency is clear
20          for t in all_owners.heads do
21              t.commit_semaphore−−
22      PromoteWaiters(tuple)

23  Function PromoteWaiters(tuple)
24      for t in tuple.waiters do
25          if conflict(t.type, tuple.owners.type) then
26              break
27          tuple.waiters.remove(t)
28          tuple.owners.add(t)
29          if ∃(t', type) ∈ tuple.retired s.t. conflict(type, t.type) then
30              t.commit_semaphore ++
```

LockAcquire()

As discussed in Section 2.1, in Wound-Wait when a conflict occurs, the requesting transaction would abort (i.e., wound) current owners that have a bigger timestamp than the requesting transaction (lines 2–7). In Bamboo, we need only a small change which is to wound transactions in *both* `owners` and `retired` (line 3). After some or all of the current owners are aborted, some waiting transaction(s) or the requesting transaction may become the new owner. In the pseudo code, for brevity, we show this logic by always adding the requesting transaction to the waiter list (line 8) and then try to promote transactions with small timestamps from `waiters` to `owners` (by calling *PromoteWaiters()* in line 9). In our actual implementation, unnecessary movement between `waiters` and `owners` are avoided.

Inside *PromoteWaiters()*, the algorithm scans waiters in the growing timestamp order (line 24). For each transaction that does not conflict with the current owner(s) (line 25), it is moved from waiters to owners (lines 27–28). Otherwise the loop breaks when the first conflict is encountered (line 26). In Bamboo, we also need to increment the *commit_semaphore* for each transaction that just became an owner if it conflicts with any transaction in retired (lines 29–30). This allows the transaction to be notified when transactions that it depends on have committed.

LockRetire()

This function simply moves the transaction from owners to retired of the tuple (lines 11–12). It then calls *PromoteWaiters()* to potentially add more transactions to owners (line 13). It is important to note that the *LockRetire()* function call is completely optional. If the function is never called for all transactions, then Bamboo degenerates to Wound-Wait. Bamboo also allows any particular transaction to choose whether to call *LockRetire()* on any particular tuple. Such compatibility allows the system to choose between Bamboo and Wound-Wait at a very fine granularity.

LockRelease()

In Wound-Wait, releasing a lock simply means removing the transaction from owners of the tuple (line 18) and promoting current waiters to become new owners (line 22). In Bamboo, the *LockRelease()* function does more work: (1) handling cascading aborts and (2) notifying other transactions when their dependency is clear.

Specifically, we define a list all_owners to be the concatenation of retired and owners (line 15). If the releasing transaction decides to abort and its lock on the tuple has type *EX*, then all the transactions in all_owners after the releasing transaction must abort cascadingly. In Bamboo, these transactions will be notified abort; the abort logic will be later performed by the corresponding worker threads. Note that if the aborting transaction locks the tuple with type *SH*, then cascading aborts are not triggered — an *SH* lock has no effect on the following transactions.

The algorithm then removes the transaction from retired or owners depending on where it resides (line 18). If the removed transaction was the old head of retired and has a lock type that conflicts with the new head of retired (line 19), then the algorithm notifies all the current leading non-conflicting transactions in retired (i.e., heads of retired) that their dependency on this tuple is clear, by decrementing their corresponding *commit_semaphore* (lines 29–30).

3.3 Deciding Timing for Lock Retire

In principle, every write can be immediately followed by *lock_retire()* without affecting correctness. If a transaction writes a tuple for a second time after retiring the lock, it can still ensure serializability by aborting all transactions that have seen its first write. It will not affect performance if transactions update each tuple only once.

For better performance, a lock can be retired after the transaction's last write to the tuple if the tuple may be updated more than once by the same transaction. To determine where the last write is, Bamboo can rely on *programmer annotation* or *program analysis* to find the last write and insert *lock_retire()* after it. In this section, we discuss the latter approach.

Determining the last write to a tuple can be challenging since the position may depend on the query parameters or tuples accessed earlier in the transaction. We illustrate the challenge using a transaction snippet shown in Listing 1, where op_1 and op_2 (line 1 and 5) both work on some tuple from the same table table1. In the example, ideally we want to add LockRetire() immediately after op_1, yet we cannot be sure whether the later operation op_2 will execute and access the same tuple or not at the desired retire point. To solve this challenge, Bamboo synthesizes a condition to add to the transaction program that dynamically decides whether to retire a lock. The process is described below.

Program analysis. Bamboo first performs standard control and data flow analysis [33] to obtain all the control and dataflow dependencies in the transaction program. It inlines all functions to perform inter-procedural analysis, and constructs a single dependency graph for each transaction.

Identify queries. Bamboo next identifies every tuple access by recognizing the database query API calls. It analyzes the query (e.g., the SQL query string passed to the API call) as well as parameters to understand the table involved and the variable that stores the key of the tuple being accessed. We assume most queries access a single tuple by primary key and Bamboo can check potential tuple-level re-access using the key. For non-key-access queries we assume it touches all tuples and detect table-level re-access.

Synthesizing retire condition. If an operation *op* works on a table that is no longer accessed after *op* in the transaction, then the lock in *op* can safely retire. Otherwise, Bamboo will synthesize a condition to decide whether to retire the lock. This condition checks any later access will be executed and touching the same tuple as *op* as an example shown on line 3 in Listing 2. In the condition, the lock in op_1 can safely retire either when cond evaluates to false, which means the later access op_2 will not happen, or when cond evaluates to true but the key of tup1 and tup2 are not equal, which means op_2 will happened but not touch the same tuple.

To generate such condition, the value of cond and the key of tup2 must be computed before the LockRetire() call. To do so, Bamboo traces the data source along the data dependency path of cond and the key, and then moves any computation on the path that happens later than op_1 to an early position, without changing the program semantic. Internally, Bamboo tries to move the computation to every position later than op_1, and stops when it finds the earliest one where all the data dependencies hold after the movement. Then it adds the synthesized condition as well as the LockRetire() call to a position after the computation. For instance, line 3 in Listing 1 originally computes the key of tup2 late in the transaction. Bamboo moves the computation of tup2.key to line 2 in Listing 2, the earliest position after op_1 where no data dependency is violated.

```
1 LockAcquire(table1, tup1, EX); // op1
2 ... // other queries and computations
3 tup2.key = f(input);
4 if (cond)
5   LockAcquire(table1, tup2, EX); // op2
```

Listing 1: A transaction program snippet

```
1 LockAcquire(table1, tup1, EX); // op1
2 tup2.key = f(input);
3 if (!cond || (cond && tup1.key!=tup2.key)) //synthesized
4   LockRetire(table2, tup2)
5 ... // other queries and computations
```

```
if (cond)
  LockAcquire(table1, tup2, EX); // op2
```

Listing 2: An example of synthesized condition from Listing 1

Handling loops. Bamboo performs loop fission to allow synthesizing condition for lock retire in loops. Listing 3 shows an example. Because the keys used in later accesses are computed in later loop iterations, Bamboo breaks the loop into two parts, as shown in Listing 4, where the first loop computes all the keys and the second loop executes the tuple accesses. Bamboo then adds a nested loop to produce the retire condition, as shown from lin 6-9. This nested loop checks the keys of the rest of the iterations and sets the variable added by Bamboo, `can_retire`, if any later key is the same as the key in the current iteration. Bamboo only handles `for` loops where the number of iteration is fixed (i.e., not changed inside the loop). For other types of loop, we do not retire locks inside the loop for now and leave it as future work.

```
for(i=0; i<input1; i++) {
  key[i] = f(input2[i]);
  tup.key = key[i];
  LockAquire(table, tup, EX);
}
```

Listing 3: A transaction snippet involving for loop

```
1 for(i=0; i<input1; i++)
2   key[i] = f(input2[i]);
3 for(i=0; i<input1; i++) {
4   tup.key = key[i];
5   LockAquire(table, tup, EX);
6   bool can_retire = true;
7   for(j=i+1; j<input1; j++)
8     can_retire &&= (key[j]!=tup.key);
9   if (can_retire) LockRetire(table, tup);
10 }
```

Listing 4: An example of synthesized condition from Listing 3

3.4 Discussions

This section discusses a few other important aspects of Bamboo. An important feature of Bamboo is the strong compatibility with Wound-Wait, meaning an existing database can be extended to use Bamboo without a major rewrite. Many important design aspects can be directly inherited from 2PL.

Fault Tolerance: Bamboo does not require special treatment of logging. As shown in Algorithm 1, a transaction does not log its commit record until it has satisfied the concurrency control protocol. This is similar to conventional 2PL.

Phantom Protection: Phantom protection [13] in Bamboo uses the same mechanism as in other 2PL protocols, namely, *next-key locking* [31] in indexes; this technique achieves the same effect as *predicate locking* but is more widely used in practice. In this context, lock retiring can also be applied to inserts or deletes to an index in the same way as read/writes to tuples.

Weak Isolation: Bamboo can support isolation levels weaker than serializability. For example, *repeatable read* is supported by giving up phantom protection; *read committed* (RC) is supported by releasing shared locks early. For RC, Bamboo needs to retire only writes since read locks are always immediately released. Finally, *read uncommitted* means each retire becomes a release.

Other Variants of 2PL: To ensure deadlock freedom, Bamboo permits `T2` to read `T1`'s dirty write only if such a dependency edge is permitted in the underlying 2PL protocol. Bamboo can be extended to any variants of 2PL but some variants fit better than others. Wait-Die, for example, allows only older transactions to wait for younger transactions. When applying retiring and dirty reads to this setting, the older transactions are subject to cascading aborts, meaning an unlucky old transaction may starve and never commit. Such problems do not exist in Wound-Wait.

Compatibility with Underlying 2PL: As we pointed out in Section 3.2, it is possible to smoothly transition between Bamboo and the underlying 2PL protocol. Specifically, the *LockRetire()* function call is completely optional for any transaction on any tuple. When it is not called, dirty reads are disabled for the particular lock and the system behavior degenerates to 2PL. This allows a system to dynamically turn on/off the dirty read optimization of Bamboo based on the performance and frequency of cascading aborts.

Opacity: Opacity is a property that reads must be consistent even before commit. By definition, ensuring opacity means a transaction is not allowed to read uncommitted data. If opacity is required for a transaction, Bamboo can enforce it by running the transaction in Wound-Wait (i.e., wait on a tuple until the retired and owners lists are empty). Note that some production systems [8, 9, 12, 28, 30] also do not support further optimizations for transactions with opacity.

3.5 Optimizations

Here we introduce four optimizations for Bamboo— the first two are to reduce extra overheads and the rest are to reduce aborts. These ideas are not entirely new but we discuss them here since they can substantially improve the performance of Bamboo. We also apply them to baseline protocols when applicable.

Optimization 1: No extra latches for read operations. In Bamboo, read operations retire automatically in `LockAcquire()`. We keep a local copy for every new read unless the data is written by the transaction itself. A read operation can be moved to the retired list directly whenever it can become the owner. This optimization requires no extra latches for retiring and will not cause more aborts since aborting a transaction holding a read lock does not cause cascading aborts. Although copying may incur extra overhead, existing work shows such overhead scales linearly with the core count [45] and the introduced overhead is less than 0.1% of the runtime with 120 cores under high contention. With the optimization, the cost of accessing extra latches is within 0.8% of the total execution time.

Optimization 2: No retire when there is no benefit. Write operations do not need to retire if they bring little benefit but increase the chances of cascading abort or the overhead of acquiring latches; read operations can always safely retire since they cannot cause cascading effects. Different heuristics can be used to decide which writes may or may not be retired. We use a simple heuristic where writes in the last δ ($0 \leq \delta \leq 1$) fraction of accesses are not retired. The intuition is that hotspots at the end of a transaction should not cause long blocking and retiring them has no benefit. However, if a transaction turns out to spend significant time (i.e., longer than δ of the total execution time) waiting on the

Algorithm 3: Support for dynamic timestamp assignment — lines 1–5 are added to the beginning of function *LockAcquire()* in Algorithm 2.

```
1 all_txns = concat(tuple.retired, tuple.owners, tuple.watiers)
2 if txn conflicts with any transaction in all_txns then
3     for t in all_txns do
4         set_ts_if_unassgined(t)
5     set_ts_if_unassigned(txn)

6 Function set_ts_if_unassigned(txn)
7     if txn.ts = UNASSIGNED then
8         atomic_compare_and_swap(&txn.ts, UNASSIGNED,
            atomic_add(global_ts))
```

`commit_semaphore`, we will retire those write operations at the end of a transaction.

Optimization 3: Eliminate aborts due to read-after-write conflicts. In the basic Bamboo protocol, when a transaction tries to acquire a shared lock, it needs to abort all the write operations of low-priority transactions in both the retired list and the owner list. We observed that such aborts are unnecessary. As Bamboo keeps local copies for reads and writes, such a new read-operation can read the local copy of other operations without triggering aborts. The optimization naturally fits Bamboo as it allows for read-modify-write over dirty data and multiple uncommitted updates can exist on a tuple. However, the idea cannot be easily applied to existing 2PL. As reading uncommitted data is not allowed, there exists only one copy of the data. Some existing works [28, 35] share a similar idea that a transaction can choose which version to read as they allow reading uncommitted data under certain scenarios. But they typically support up to one uncommitted version.

Optimization 4: Assign timestamps to a transaction on its first conflict. Here we explain how to extend Bamboo to support dynamic timestamp assignment to avoid aborting on the first conflict. The pseudocode is shown in Algorithm 3. Specifically, lines 1–5 in Algorithm 3 are inserted to the beginning of function *LockAcquire()* in Algorithm 2. If the incoming transaction conflicts with any other transaction in `retired`, `owners`, or `waiters` (line 1–2), we assign timestamps for all transactions in the three lists, in the order specified by the algorithm (lines 3–4), and then assign timestamp for the incoming transaction (line 5). Timestamp assignment is a single compare_and_swap() call (lines 6–8). Bamboo may be further improved through more complex timestamping strategies [29].

4 CASCADING ABORTS

This section analyzes the effect of cascading aborts qualitatively and discusses an optimization that we propose to mitigate the effect.

4.1 Cases Inducing Cascading Aborts

Cascading aborts (also called *cascading rollback*) is a situation where the abort of a transaction causes other transactions to abort. In Bamboo, transactions can read uncommitted data, if the transaction that wrote the data aborts, all dependent transactions must also abort cascadingly. In our algorithm, the procedure of cascading aborts corresponds to the line 17 of Algorithm 2.

In Bamboo, a transaction T may abort in three cases: (1) when T is wounded by another transaction with higher priority to prevent deadlocks, (2) when a transaction that T depends on aborts and T aborts cascadingly, or (3) when T self-aborts due to transactional logic or user intervention. As cases (1) and (3) can occur in the baseline Wound-Wait protocol as well, we focus on discussing the difference between case (2) and the other two cases.

4.2 Effects of Cascading Aborts as a Trade-off of Reducing Blocking

The effect of cascading aborts can be evaluated through three metrics — length of abort chain, abort rate, and abort time. The *length of abort chain* depicts the number of transactions that must abort cascadingly due to one transaction's abort; our empirical result shows the number can be as large as the number of concurrent transactions at high contention. The *abort time* describes the total CPU time wasted on executing transactions that aborted in the end. The throughput of a transaction processing system is largely determined by the number of CPU cycles performing useful vs. useless work; *abort time* is an example of such useless work. The other example is the time spent on waiting for a lock, which we defined as *Wait Time*. Unlike the first two indicating the magnitude of the effect, the time measurements illustrate the tradeoff between waits and aborts in a more direct way. We use this metric to show the trade-off in the evaluation section.

Compared to Wound-Wait, Bamboo substantially reduces the *wait time* but increases *abort time*. Although Bamboo may have more aborts than Wound-Wait due to cascading aborts, trading waits for aborts may be a good deal in many cases.

There are two reasons why aborts may be preferred in certain cases. First, with hotspots, all the transactions aborted cascadingly are those that have speculatively read dirty data. These transactions would have been waiting in Wound-Wait without making forward progress in the first place. A large portion of cycles wasted on cascading aborts in Bamboo would also be wasted on waiting in Wound-Wait. Second, even if a transaction aborts, it warms up the CPU cache with accessed tuples, so subsequent executions become faster [27, 41]. We observe this effects on Silo, an OCC-based protocol, as described in Section 5 — Silo has higher abort rate than many 2PL protocols but also higher throughput.

We build a model based on previous theoretical analysis on 2PL [6, 18, 19] to illustrate when the benefits of Bamboo outweigh its overhead. We define K as the number of lock requests per transaction, N as the number of transactions running concurrently, D as the number of data items, and t as the average time spent between lock requests. The throughput is proportional to $\frac{N}{(K+1)t} \times (1 - AP_{conflict} - BP_{abort})$, where $P_{conflict}$ and P_{abort} denote the probability a transaction encounters a conflict and an abort, respectively; A denotes the fraction of execution time that a transaction spends waiting given a conflict; B denotes the fraction of time spent on aborted execution. Bamboo (bb) can reduce $AP_{conflict}$ (due to early retire) but increase BP_{abort} (due to cascading aborts). It has positive gains over Wound-Wait (ww) when the benefits outweigh the overhead.

The gain in $AP_{conflict}$ is $(A_{bb} - A_{ww})P_{conflict}$, where $P_{conflict}$ is a property of the workload and is approximately $NK^2/(2D)$ [18, 19] in both protocols; A_{bb} is approximately $1/(K+1)$ (i.e., wait for only the duration of one access) and A_{ww} is on average 1/2 (i.e., wait for half of the transaction execution time).

To model BP_{abort}, we observe that Bamboo and Wound-Wait share two common sources of aborts, i.e. aborts due to deadlock and user-initiated aborts. Bamboo introduces another source of aborts due to cascading, represented as BP_{cas_abort}, where P_{cas_abort} is the probability that a transaction aborts cascadingly. We calculate an upper bound of this cost. We can bound B by 1 and bound P_{cas_abort} by $(1 - P_{deadlock}) \times P_{conflict} \times P_{deadlock} \times (N - 1)$ (i.e., the current transaction experiences a conflict while some other transaction experiences a deadlock), which is bounded by $NP_{conflict}P_{deadlock}$. The value of $P_{deadlock}$ is approximately $NK^4/4D^2$ [18, 19] given uniform distribution.

Combining the above, Bamboo has performance advantage when $(A_{ww} - A_{bb})P_{conflict} > BP_{cas\_abort}$, which is satisfied when $(\frac{1}{2} - \frac{1}{K+1})P_{conflict} > NP_{conflict}P_{deadlock}$, which is satisfied when $\frac{N^2K^4}{2D^2} < \frac{K-1}{K+1}$. For most databases, the data size D is orders of magnitude larger than N and K; so the equation will hold. The high-level intuition here is that $P_{deadlock}$ is much lower than $P_{conflict}$ [18, 19]. Bamboo optimizes for the common case by reducing the cost of a conflict and sacrifices performance of the conner case by increasing the cost of aborts during deadlocks.

In Section 5, we performed quantitative evaluations on the impact of cascading aborts and the tradeoff between aborts and waits using the metrics described above. We will show how the evaluation results corroborate the arguments and modeling here.

5 EXPERIMENTAL EVALUATION

This section evaluates the performance of Bamboo. We first introduce the experimental setup in Section 5.1, followed by demonstrations of the performance of Bamboo without cascading aborts in Section 5.2. In Section 5.3, we evaluate Bamboo under different scenarios to understand the effect of cascading aborts. We then report the performance of Bamboo on YCSB and TPC-C workloads in Sections 5.4 and 5.5 respectively to evaluate Bamboo on different distributions and workloads with user-initiated aborts. Section 5.6 compares Bamboo and IC3 with TPC-C at high contention.

5.1 Experimental Setup

We implement Bamboo in DBx1000 [1, 47], a multi-threaded, in-memory DBMS prototype. It stores data in a row-oriented manner with hash table indexes. The code is open sourced [2]. In this paper, we extended DBx1000 to run transactions in both stored-procedure and interactive modes. In the stored-procedure mode, all accesses in a transaction and the execution logic are ready before execution.

The interactive mode involves two types of nodes: (1) the *DB server* processes requests like `get_row()`, and `update_row()`, and (2) the *client server* executes transaction logic and sends requests to the DB server through gRPC. As Bamboo does not require knowing the position of the last write for correctness (cf. Section 3.3), DB server immediately retires the lock after completing each write request, essentially treating every write as the last write. Optimization 2 introduced in Section 3.5 does not apply in this mode.

DBx1000 includes a pluggable lock manager that supports different concurrency control schemes. This allows us to compare Bamboo with various baselines **within the same system**. We implemented 5 approaches described as follow:

- **WOUND_WAIT** [5]: The Wound-Wait variant of 2PL (Section 2.1).
- **NO_WAIT** [5]: The No-Wait variant of 2PL where any conflict causes the requesting transaction to abort.
- **WAIT_DIE** [5]: The Wait-Die variant of 2PL.
- **SILO** [41]: An in-memory database for fast and scalable transaction processing. It implements a variant of OCC.
- **IC3** [43]: State-of-the-art transaction chopping-based concurrency control protocol as described in Section 2.2.

Experiments in stored-procedure mode were run on a machine with four Intel Xeon CPU at 2.8GHz (15 cores) with 1056GB of DRAM, running Ubuntu 16.04. Each core supports two hardware threads. For the interactive mode, experiments were run on workstations provided by cloudlab [11] with each machine containing two Intel Xeon CPU at 2.6GHz (32 cores) with 376GB of DRAM, running Ubuntu 16.04. Each core supports two hardware threads. We collect transaction statistics, such as throughput, latency, and abort rates by running each workload for at least 30 seconds.

In this paper, we assume that each hotspot contains one tuple and treat a set of hot tuples as multiple hotspots. For the experiments, transactions log to main memory — modern non-volatile memory would offer similar performance. Bamboo applies all the optimizations introduced in Section 3.5. To decide the choice of δ, we ran microbenchmark with a wide range of δ. In general, as δ increases, the overhead in Bamboo decreases, which improves performance in low-contention cases. However, a larger δ also increases the time spent on waiting for locks under high contention, which leads to less than 13% drop in performance in our experiments. To balance the contrary effects under different workloads, we chose a δ of 0.15 across all workloads. As the dynamic timestamp assignment can also be applied to other 2PL-based protocols, we turn on the optimizations whenever they gain improvements from it. However, as only Bamboo involves cascading aborts, the other protocols barely benefit from the optimization.

5.2 Experimental Analysis on Bamboo without Cascading Aborts (Single Hotspot)

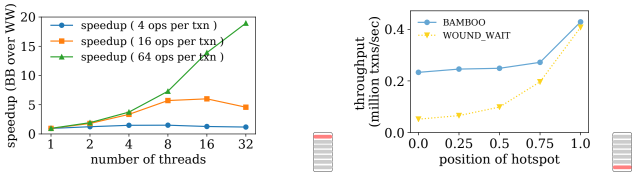
In this section, we evaluate the potential benefits of Bamboo in the ideal cases with only one hotspot and induce no cascading aborts.

Single Hotspot at Beginning

We firstly design a synthetic workload with 16 random reads but a single read-modify-write hotspot at the beginning. In stored-procedure mode, Bamboo shows 6× improvements against the best-performing 2PL-based protocols (Wait-Die) due to savings on waiting. In the interactive mode, Bamboo is up to 7× better than the best baseline (Wound-Wait).

Varying Transaction Length

In Figure 3a, we vary the length of the transactions and report the speedup of Bamboo (BB) over Wound-Wait (WW). Firstly, the results shows that Bamboo has greater speedup for longer transactions by up to 19×, which corresponds to a larger A increasing the benefit in the modeling shown in Section 4.2. Secondly, the speedup first increases as the number of threads increases, and then



(a) Varying transaction length **(b) Varying hotspot position**

Figure 3: Performance on synthetic benchmark with one hotspot at the beginning with various settings, stored-procedure mode

saturates or drops when even more threads come in. This is due to the limitation of the inherent level of parallelism in the workload.

Varying Hotspot Position

Instead of fixing the hotspot at the beginning, we vary the position of the hotspots in this experiment. The result is shown in Figure 3b. The two small figures on each side of the x-axis corresponding to the position of the hotspot when $x = 0$ (beginning of transaction) and $x = 1$ (end of transaction), respectively. We show the results for workloads with transactions of 16 operations, while the results for other transaction lengths have similar observations. Bamboo provides a higher speedup against Wound-Wait when the access of hotspot is earlier in a transaction. The result also aligns with the modeling as an early access gives larger A_{ww} and thus greater benefit from $A_{ww} - A_{bb}$.

Summary: Through reducing lock waiting time, Bamboo can improve performance significantly (up to 19× over Wound-Wait). Some factors have an impact on the performance gain — Bamboo shows larger speedup under a higher level of parallelism (i.e., more threads), longer transactions, and "earlier" hotspot accesses.

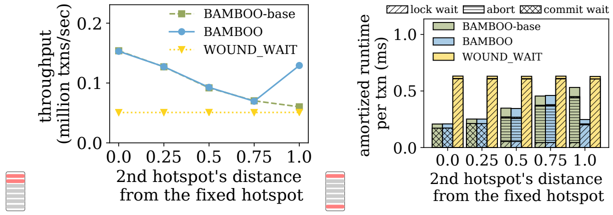
5.3 Experimental Analysis on Bamboo with Cascading Aborts (Multiple Hotspots)

Next, we present empirical evaluation of Bamboo serving workloads that can induce cascading aborts to understand their effects. We start with synthetic workloads with 2 read-modify-write hotspots and 14 random reads. Here we use a dataset of more than 100 GB.

To study the tradeoff between waits and aborts, we precisely control the workloads as two types: (1) we fix the first hotspot at the beginning of each transaction while moving the second around. In this case, the benefit Bamboo gained over Wound-Wait is fixed and the chance of having cascading aborts increases as the distance between the two hotspots increases. We use the case to study how different magnitude of cascading aborts can affect the gains. (2) we fix the second hotspot at the end of the transactions and move the other around to study the case where the benefits and the chance of cascading aborts increase simultaneously for Bamboo.

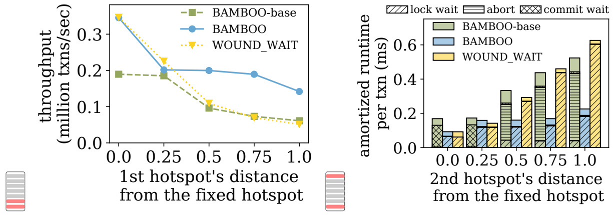
Fix One Hotspot at the Beginning

In this experiment, we also show BAMBOO-base without the Optimization 2 introduced in Section 3.5. As shown in Figure 4a, Bamboo outperforms Wound-Wait for all distances. Figure 4b shows how Bamboo gains speedup by trading more aborts for less blocking. The improvements of Bamboo can be up to 3×. When the distance $x = 0.75$ (i.e., there are 10 operations between the two hotspots), Bamboo outperforms Wound-Wait by 37%, although the abort rate



(a) throughput **(b) runtime analysis**

Figure 4: One hotspot at the beginning for Bamboo (left) and Wound-Wait (right), stored-procedure mode (32 threads)



(a) throughput **(b) runtime analysis**

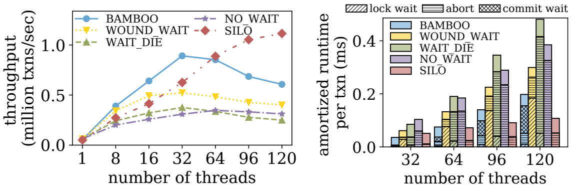
Figure 5: The second hotspot at the end for Bamboo (left) and Wound-Wait (right), stored-procedure mode (32 threads)

of Bamboo is 72% higher. The result can be explained by the model, which indicates an improvement of at least 21.7% in Bamboo following $(A_{ww} - A_{bb})P_{conflict} - B_{bb} * P_{cas\_abort} = 15/16 * 1 - B_{bb} * 0.72 \geq 0.217$. The two versions of Bamboo differ only when the second hotspot is at the end of the transaction ($x = 1.0$). With the optimization, the last hotspot will not be retired, which greatly reduces the bookkeeping overhead. Figure 4b also illustrates that Bamboo reduces blocking while not increasing aborts with the optimization.

Fix One Hotspot at the End

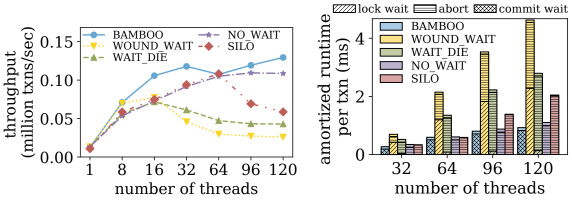
We now fix the second hotspot at the end and change the position of the first hotspot. Compared to Figure 5a, this workload here has less advantage for Bamboo to begin with and yet introduces more cascading aborts as the benefit increases. Figure 5b shows that the time spent on aborts in Bamboo never exceeds the time spent on waiting in Wound-Wait. However, Bamboo without the second optimization (i.e., BAMBOO-base) may suffer from the overhead when it barely has benefits when $x = 0$, where the theoretical improvement is only 1/16. We note that such cost is a function of the workload and underlying system. It may be significant with stored-procedure as shown here, which makes our optimization necessary in mitigating the problem. With other system setups such as the interactive mode shown in later experiments, the trade-off can change greatly.

Summary: The potential benefit of Bamboo against Wound-Wait is a tradeoff between the benefit of reducing lock waiting time and the cost of cascading aborts and other overhead. Our measurements show that the benefits of reducing lock waiting time is usually greater than the cost of abort. However, Bamboo can suffer from overhead with certain system setup when the benefit is minimal. In this case, our optimization of conditionally retiring some write operations should be applied for such cases.



(a) throughput **(b) runtime analysis**

Figure 6: YCSB with varying thread count, stored-procedure mode ($\theta = 0.9$, $read\_ratio = 0.5$)



(a) throughput **(b) runtime analysis**

Figure 7: YCSB with 5% long read-only transactions accessing 1000 tuples, stored-procedure mode ($\theta = 0.9$, $read\_ratio = 0.5$)

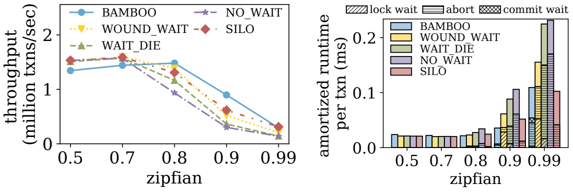
5.4 Experiments on YCSB

We now move to an even more complex workload — YCSB with zipfian distribution. We will show how Bamboo performs compared to other baselines as the number of threads, data accessing distribution, and read ratio vary. Note that in general Bamboo only targets the high-contention setup in this workload as it is where hotspots (that most transactions would access) are present.

The Yahoo! Cloud Serving Benchmark (YCSB) [7] is a collection of workloads that are representative of large-scale services created by Internet-based companies. For all experiments in this section, we use a large scale database of more than 100 GB, containing a single table with 100 million records. Each YCSB tuple has a single primary key column and then 10 additional columns each with 100 bytes of randomly generated string data. The DBMS creates a single hash index for the primary key.

Each transaction in the YCSB workload by default accesses 16 records in the database. Each access can be either a read or an update. We control the overall read/write ratio of a transaction by a specified *read_ratio*. We also control the workload contention level through θ, a parameter controlling the Zipfian data distribution. For example, when $\theta = 0$, all tuples have equal chances to be accessed. When $\theta = 0.6$ or $\theta = 0.8$, 10% of the tuples in the database are accessed by ~40% and ~60% of all transactions respectively.

Varying Number of Threads. Figure 6a demonstrates that Bamboo's improvement against Wound-Wait with different number of threads in highly contentious YCSB ($\theta = 0.9$) configured in stored-procedure mode. Figure 6b shows Bamboo's benefits come from reducing waiting time without introducing many aborts. With 64 threads, Bamboo achieves the maximum speedup against Wound-Wait, which is up to 1.77×. All 2PL-based protocols show degradation after 32 threads and Bamboo underperforms SILO when the thread count more than 96. This is mainly due to the intrinsic lock thrashing problem in 2PL [40].



(a) throughput **(b) runtime analysis**

Figure 8: YCSB with varying distribution — throughput vs. Zipfian skew level for YCSB, stored-procedure mode, ($read\_ratio = 0.5$, 16 threads)

Long Read-Only Transaction. This experiment uses a workload with 5% long read-only transactions accessing 1000 tuples and 95% read-write transactions accessing 16 tuples. Figure 7a shows that Bamboo outperforms all other protocols most of the time. Compared with waiting-based protocols, Bamboo benefits much from reducing waiting while rarely aborts. It shows an improvement of up to 5× against Wound-Wait. Optimization 3 (Section 3.5) in Bamboo also contributes to the scenario as long read-only transactions will not block writes nor cause cascading aborts. SILO experience performance degradation in this case since long transactions may starve and aborts dominate the runtime as shown in Figure 7b. Bamboo also outperforms No-Wait as Bamboo ensures the priorities of transactions and commit 20% more long transactions than No-Wait when the thread count is 120.

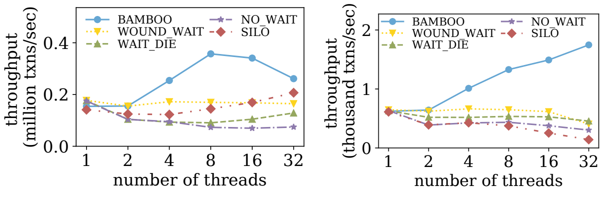
Varying Read Ratio. We examine how varying *read_ratio* would influence the performance of different protocols in stored-procedure mode. Bamboo shows improvements against all other protocols regardless of the read ratio. The percentage of improvement ranges from 27% to 71%

Varying Data Accessing Distribution. Figure 8 shows how Bamboo performs in both stored-procedure mode and interactive mode as θ of Zipfian distribution changes. As showed in Figure 8a, Bamboo outperforms all 2PL-based protocols under high contention (e.g. $\theta > 0.7$). Compared to Wound-Wait, Bamboo provides up to 72% improvements in throughput. For cases of lower contention, Bamboo has ~10% degradation in throughput compared with Wound-Wait due to overhead. However, such degradation diminishes in the interactive mode where the expensive network communication dominates. In the interactive mode, Bamboo is comparable with Wound-Wait with ~8% improvements when $\theta \leq 0.8$ and shows up to 2× speedup over Wound-Wait.

Similarly, SILO's low-level performance optimizations (e.g., lock-free data structures) and the cache warming-up effect due to many aborts makes its performance significantly better than other 2PL-based protocols in stored-procedure mode. However, the performance advantage of Silo disappears in the interactive mode (not shown in the figures) where aborts are significantly more expensive due to expensive gRPC calls. Bamboo outperforms all other protocols including SILO in interactive mode where the cache warming-up effect has less impact.

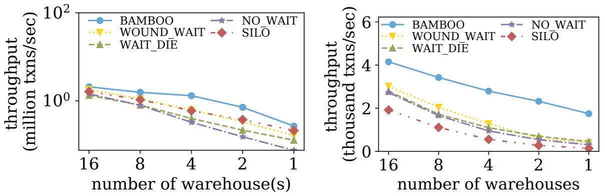
5.5 Experiments on TPC-C Results

Finally, we compare Bamboo with other concurrency control schemes on the TPC-C benchmark [39]. We only ran experiments with 50% new-order transactions and 50% payment transactions. Note in the



(a) stored-procedure mode **(b) interactive mode**

Figure 9: vary # of threads in TPC-C (1 warehouse)



(a) stored-procedure mode **(b) interactive mode**

Figure 10: vary # of warehouses in TPC-C (32 threads)

benchmark, 1% of new order transaction are chosen at random to simulate user-initiated aborts.

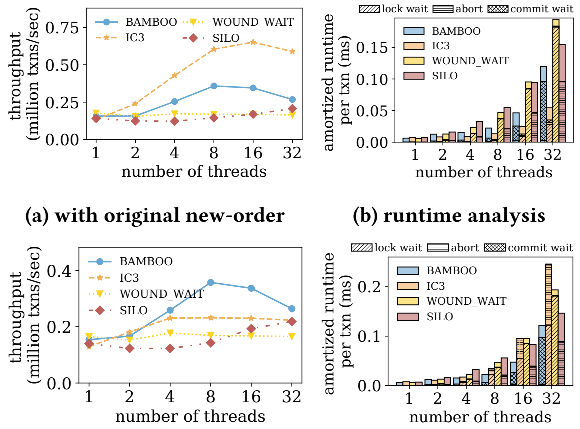
We first vary the number of threads. Figure 9 presents the behavior of Bamboo under high contention in both modes. Bamboo can obtain up to 2× improvements against Wound-Wait in stored-procedure mode. Similar to previous observations, SILO outperforms other 2PL protocols given its cache warming-up effect in stored-procedure mode. In interactive mode, Bamboo's performance scales up till 32 threads and achieves up to 4× and 14× improvements against Wound-Wait and SILO respectively.

Figure 10 presents how Bamboo performs with a different number of warehouses with 32 threads. In stored-procedure mode, Bamboo outperforms other 2PL-based protocols in high-contention cases. The improvement depends on the number of warehouses. For example, when the number of warehouses is one (which is similar to the single hotspot cases), Bamboo outperforms Wound-Wait by up to 2×. When the workload is less contentious (e.g. with more warehouses), the difference between Bamboo and other protocols is smaller. It is expected since Bamboo targets at high-contention cases. Figure 10b shows that Bamboo has more improvements of up to 4× over the best baseline when running in interactive mode.

5.6 Comparison with IC3

We choose to compare the performance of Bamboo and IC3 in a separate section due the fact that IC3 requires the knowledge of the entire workload — an assumption not made in the other protocols. We implemented IC3 with all its optimizations in DBx1000 and show the results with the best setting. Note that we omit the optimizations for commutative operations for a fair comparison to all algorithms.

Figure 11a shows the comparison between Bamboo and IC3 on the mix of payment and new-order transactions with a global warehouse table. As payment and new-order accesses different columns of warehouse table and district table (the most contentious tables in TPC-C), IC3's prior knowledge on all column accesses help it get rid of much contention. Given this, IC3 outperforms Bamboo though it enforces some waiting when two transactions access the same column of different tuples.



(a) with original new-order **(b) runtime analysis**

(c) with modified new-order **(d) runtime analysis**

Figure 11: Bamboo vs. IC3 in TPC-C, stored-procedure mode (1 warehouse)

However, for workloads where contentious transactions access the same columns of hotspot tuples, IC3 cannot gain such benefits and underperforms Bamboo due to its column-level static analysis. To illustrate the effect, we simply modified new-order transactions to read one more column (W_YTD) that will be updated by payment transactions. It is conceivable for a real-world transaction to read a hot field updated by other transactions. The result and runtime analysis are shown in Figure 11c and Figure 11d. While the performance of Bamboo is barely affected, the performance of IC3 drops significantly. As expected, IC3 spends more time on waiting than Bamboo due to its column-level static analysis. Note that, the increase in aborts is due to IC3's optimistic execution. The version without optimistic execution shows worse performance with more waiting. It would serialize the execution of potentially but not actually conflicting sub-transactions instead of just serializing the validation phases of these sub-transactions. Overall, Bamboo has up 1.5× improvement against IC3 on the slightly modified TPC-C workload that has "true" conflicts between payment and neworder transactions on the warehouse table.

Summary. In stored-procedure mode, Bamboo is better than IC3 when most transactions truly conflict on a global hotspot. The column-level static analysis in IC3 reduces more contention when transactions access different columns of the same tuple, however, it makes IC3 non-applicable for interactive modes.

6 RELATED WORK

6.1 Violating Two-Phase Locking

Previous work has explored mechanisms that violate 2PL to improve transaction performance but targeted different aspects than Bamboo. In distributed systems, Jones et al. [25] proposed a locking scheme that allows dependent transactions to execute when one transaction is waiting for its execution to finish in multiple partitions. The technique avoids making transactions wait for earlier transactions' distributed coordination. Gupta et al. [21] proposed a distributed commit protocol where transactions are permitted to read the uncommitted writes of transactions in the prepare phase of two-phase commit, but not during transaction execution.

Locking violation is also used to avoid holding locks while logging. Early Lock Release (ELR) [26, 37] is based the observation that a canonical transaction holds locks after the execution (i.e., has pre-committed) while waiting for the log to be flushed. ELR allows such pre-committed transactions to release their locks early so that other transactions can proceed and uses tags to enforce the commit order among dependent transactions. ELR has been applied in both research protocols [24] and commercial systems [12].

Controlled lock violation (CLV) [17] achieved the same goal as ELR. But instead of using tags to enforce the commit order, CLV extends data structures in the lock table to track and enforce the dependency order, therefore working for more general cases. The dependency tracking mechanism of Bamboo is inspired by CLV. In contrast to ELR and CLV, Bamboo explores violating 2PL to avoid holding locks on hotspots during the *execution phase* before a transaction pre-commits to exploit more parallelism.

Ordered shared locks [3] explores violating 2PL in the execution phase but lacks specifications on key design components such as how to track dependency and avoiding deadlocks effectively. It has no qualitative or quantitative analysis on cascading aborts, a major concern of the approach, on modern systems. In contrast, this paper thoroughly analyzes the effect of cascading aborts and the inherent tradeoff between waits and aborts both qualitatively and quantitatively with proposed optimizations. We further discussed both key designs and new techniques in details such as safe retire with program analysis used in Bamboo. We also performed evaluations on modern systems comparing with state-of-the-art baselines.

6.2 Reading Uncommitted Data

Previous work also proposed non-locking concurrency control protocols that can read uncommitted data. Faleiro et al. [15] proposed a protocol for deterministic databases that enables early write visibility, meaning that a transaction's writes are visible prior to the end of the execution. This protocol leverages the determinism where transaction execution is ordered prior to execution and the writes can be immediately visible to later transactions if the writing transaction will certainly commit. In contrast, Bamboo does not rely on the assumption of determinism.

Hekaton [9, 28] proposed two protocols — pessimistic version and optimistic version — for main memory databases based on multiversioning. The pessimistic protocol allows for eager updates. Operations like appending updates to a read-locked data or reading the last-committed version of a write-locked data will not be blocked. If the owner of uncommitted dirty data is in *preparing state*, dirty reads are allowed as well. However, as dirty data is not visible if its owner is in *active state*, a write operation over write locked data by an active transaction will still be blocked. Bamboo makes uncommitted data visible to reduce blocking time for transactions with write-after-write conflicts. Similarly to Hekaton, Bamboo also tracks dependencies of transactions due to visible dirty data for serializable correctness, but in a different way.

In addition to IC3, runtime pipelining [45] (RP) is another variant of transaction chopping. It is based on table-level static analysis combined with runtime enforcement. Specifically, they firstly derive a total ranking of all the read-write tables. Then it orders the sub-transactions based on the rank and enforces the execution to follow the order. However, sub-transactions still cannot be arbitrarily small to allow for more concurrency for two reasons. First, similar to IC3, accesses must be merged into one piece if they cause crossing of C-edges. Second, table-level analysis allows for less concurrency than column-level analysis.

Deferred runtime pipelining [32] (DRP) extends runtime pipelining to support both transactions where the access sets are known and unknown. Similar to Bamboo, DRP allows transactions to read *tame* transactions' uncommitted data whenever the updates are done. However, DRP imposes stronger assumptions on the tame transactions that all the accesses must be known before execution to ensure serializability and being deadlock-free. DRP also introduces deferred execution to know when the updates are done and to reduce cascading aborts. However, the technique can be applied only when later operations do not depend on the previous ones.

There are works redesigning the architecture to allow reading uncommitted data for hardware transaction. For example, Jeffrey et al. proposed Swarm [22, 23] which divides a sequential program into many small ordered transactions and speculatively runs them in parallel. Unlike Swarm, Bamboo is implemented in software and can be easily integrated into existing 2PL-based database systems.

6.3 Transaction Scheduling

Quro [46] changes the order of operations within a transaction to make hotspots appear as close to commit as possible to reduce the duration of locking period. However, Qura is subject to data dependency in the transaction thus often not able to flexibly move hotspots arbitrarily. Bamboo does not require changing the order of transaction operators, but changes the concurrency control to handle hotspots, making it able to improve performance for a wider range of transactions than aforementioned work.

Ding et al. [10] reorders the transactions within a batch to minimize inter-transaction conflicts and improve OCC for highly contentious cases. It models the problem of finding the best order while preserving the correctness as a feedback vertex set problem over directed graph. For each batch, they run a proposed greedy algorithm to approximate the solution of the NP-hard problem. However, empirical evaluation shows the reordering process can take up to 17× of the transaction processing time. making the end-to-end performance lower than Bamboo.

7 CONCLUSION

We proposed Bamboo, a concurrency control protocol that extends traditional 2PL but allows the two-phase rule to be violated by retiring locks early. Through extensive analysis and performance evaluation, we demonstrated that Bamboo can lead to significant performance improvement when the workload contain hotspots. Evaluation on TPC-C shows a performance advantage of up to 3×.

ACKNOWLEDGMENTS

Support for this research was provided by the University of Wisconsin-Madison, Office of the Vice Chancellor for Research and Graduate Education with funding from the Wisconsin Alumni Research Foundation. We also thank Phil Bernstein for his sapient feedback.

REFERENCES

[1] [n.d.]. DBx1000. https://github.com/yxymit/DBx1000.

[2] [n.d.]. DBx1000 with Bamboo Implemented. https://github.com/ScarletGuo/Bamboo-Public.

[3] Divyakant Agrawal, Amr El Abbadi, Richard Jeffers, and Lijing Lin. 1995. Ordered shared locks for real-time databases. *The VLDB Journal* 4, 1 (1995), 87–126.

[4] Philip A Bernstein, Philip A Bernstein, and Nathan Goodman. 1981. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)* 13, 2 (1981), 185–221.

[5] Philip A. Bernstein and Nathan Goodman. 1981. Concurrency Control in Distributed Database Systems. *CSUR* (1981), 185–221.

[6] Philip A Bernstein and Eric Newcomer. 2009. *Principles of transaction processing.* Morgan Kaufmann.

[7] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*. 143–154.

[8] James C. Corbett and et al. 2012. Spanner: Google's Globally-Distributed Database. In *OSDI*. 251–264.

[9] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *SIGMOD*. 1243–1254.

[10] Bailu Ding, Lucja Kot, and Johannes Gehrke. 2018. Improving optimistic concurrency control through transaction batching and operation reordering. *Proceedings of the VLDB Endowment* 12, 2 (2018), 169–182.

[11] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19

[12] Tamer Eldeeb and Phil Bernstein. 2016. *Transactions for Distributed Actors in the Cloud.* Technical Report.

[13] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. 1976. The Notions of Consistency and Predicate Locks in a Database System. *CACM* (1976), 624–633.

[14] Jose M. Faleiro and Daniel J. Abadi. 2015. Rethinking Serializable Multiversion Concurrency Control. *PVLDB* (2015), 1190–1201.

[15] Jose M Faleiro, Daniel J Abadi, and Joseph M Hellerstein. 2017. High performance transactions via early write visibility. *Proceedings of the VLDB Endowment* 10, 5 (2017), 613–624.

[16] Dieter Gawlick and David Kinkade. 1985. Varieties of concurrency control in IMS/VS fast path. *IEEE Database Eng. Bull.* 8, 2 (1985), 3–10.

[17] Goetz Graefe, Mark Lillibridge, Harumi Kuno, Joseph Tucek, and Alistair Veitch. 2013. Controlled lock violation. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 85–96.

[18] J. Gray, Pete Homan, H. Korth, and R. Obermarck. 1981. A Straw Man Analysis of the Probability of Waiting and Deadlock in a Database System. In *Berkeley Workshop*.

[19] Jim Gray and Andreas Reuter. 1992. *Transaction Processing: Concepts and Techniques* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[20] Zhihan Guo, Kan Wu, Cong Yan, and Xiangyao Yu. 2021. Releasing Locks As Early As You Can: Reducing Contention of Hotspots by Violating Two-Phase Locking (Extended Version). arXiv:2103.09906 [cs.DB]

[21] Ramesh Gupta, Jayant Haritsa, and Krithi Ramamritham. 1997. Revisiting Commit Processing in Distributed Database Systems. In *SIGMOD*. 486–497.

[22] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. 2015. A Scalable Architecture for Ordered Parallelism. In *MICRO*. 228–241.

[23] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez. 2016. Unlocking Ordered Parallelism with the Swarm Architecture. *IEEE Micro* (2016), 105–117.

[24] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. 2010. Aether: a scalable approach to logging. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 681–692.

[25] Evan P.C. Jones, Daniel J. Abadi, and Samuel Madden. 2010. Low Overhead Concurrency Control for Partitioned Main Memory Databases. In *SIGMOD*. 603–614.

[26] Hideaki Kimura, Goetz Graefe, and Harumi A Kuno. 2012. Efficient locking techniques for databases on modern hardware.. In *ADMS@ VLDB*. 1–12.

[27] Hsiang-Tsung Kung and John T Robinson. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)* 6, 2 (1981), 213–226.

[28] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *VLDB* (2011), 298–309.

[29] David Lomet, Alan Fekete, Rui Wang, and Peter Ward. 2012. Multi-Version Concurrency via Timestamp Range Conflict Management. In *ICDE*. 714–725.

[30] Dahlia Malkhi and Jean-Philippe Martin. 2013. Spanner's concurrency control. *ACM SIGACT News* 44, 3 (2013), 73–77.

[31] C Mohan. 1990. *ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes.* VLDB.

[32] Shuai Mu, Sebastian Angel, and Dennis Shasha. 2019. Deferred runtime pipelining for contentious multicore software transactions. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16.

[33] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 2010. *Principles of Program Analysis.* Springer Publishing Company, Incorporated.

[34] Daniel J Rosenkrantz, Richard E Stearns, and Philip M Lewis. 1978. System Level Concurrency Control for Distributed Database Systems. *ACM Transactions on Database Systems (TODS)* 3, 2 (1978), 178–198.

[35] Mohammad Sadoghi, Mustafa Canim, Bishwaranjan Bhattacharjee, Fabian Nagel, and Kenneth A. Ross. 2014. Reducing Database Locking Contention through Multi-Version Concurrency. *Proc. VLDB Endow.* 7, 13 (Aug. 2014), 1331–1342. https://doi.org/10.14778/2733004.2733006

[36] Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. 1995. Transaction Chopping: Algorithms and Performance Studies. *ACM Transactions on Database Systems (TODS)* 20, 3 (1995), 325–363.

[37] Eljas Soisalon-Soininen and Tatu Ylönen. 1995. Partial strictness in two-phase locking. In *International Conference on Database Theory*. Springer, 139–147.

[38] Dixin Tang and Aaron J Elmore. 2018. Toward coordination-free and reconfigurable mixed concurrency control. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 809–822.

[39] The Transaction Processing Council. 2007. TPC-C Benchmark (Revision 5.9.0).

[40] Alexander Thomasian. 1993. Two-phase locking performance and its thrashing behavior. *ACM Transactions on Database Systems (TODS)* 18, 4 (1993), 579–625.

[41] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-Memory Databases. In *SOSP*.

[42] Tianzheng Wang and Hideaki Kimura. 2016. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *Proceedings of the VLDB Endowment* 10, 2 (2016), 49–60.

[43] Zhaoguo Wang, Shuai Mu, Yang Cui, Han Yi, Haibo Chen, and Jinyang Li. 2016. Scaling multicore databases via constrained parallel execution. In *Proceedings of the 2016 International Conference on Management of Data*. 1643–1658.

[44] Gerhard Weikum and Gottfried Vossen. 2001. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery.* Elsevier.

[45] Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. 2015. High-performance ACID via modular concurrency control. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 279–294.

[46] Cong Yan and Alvin Cheung. 2016. Leveraging Lock Contention to Improve OLTP Application Performance. *Proceedings of the VLDB Endowment* 9, 5 (2016), 444–455.

[47] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *VLDB*, 209–220.

[48] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K Aguilera, and Jinyang Li. 2013. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *SOSP*. 276–291.