Rebirth-Retire: A Concurrency Control Protocol Adaptable to Different Levels of Contention

Qian Zhang[†]
East China Normal University
Shanghai, China
52184501012@stu.ecnu.edu.cn

Yiwen Xiang[†]
East China Normal University
Shanghai, China
51215902109@stu.ecnu.edu.cn

Jianhao Wei
East China Normal University
Shanghai, China
52215902007@stu.ecnu.edu.cn

Yang Yang
East China Normal University
Shanghai, China
51265902054@stu.ecnu.edu.cn

Yifan Li
East China Normal University
Shanghai, China
51265902129@stu.ecnu.edu.cn

Xueqing Gong[*]
East China Normal University
Shanghai, China
xqgong@sei.ecnu.edu.cn

Wanggen Liu
Transwarp Technology (Shanghai)
Co., Ltd
Shanghai, China
wayne.liu@transwarp.io

ABSTRACT

The Wound-Retire concurrency control protocol was proposed to reduce contention for hotspots in in-memory databases. It enhances throughput under high-contention scenarios by allowing transactions to release their locks earlier (referred to as *Retire*), thereby reducing the wait times for other transactions. However, the proactive early release of locks introduces additional overhead, making it less efficient than other lock-based protocols in low-contention scenarios. Moreover, the wound strategy it adopts, while effective at preventing deadlocks, may lead to unnecessary transaction aborts.

To address these issues, this paper proposes the Rebirth-Retire concurrency control protocol as an enhancement to the Wound-Retire protocol. In this protocol, a lock is retired by a younger transaction that requests it, which reduces unnecessary retire costs in low-contention scenarios. Additionally, rather than aborting younger transactions, older transactions are assigned larger timestamps (referred to as *Rebirth*), unless doing so would result in a deadlock. Experimental evaluations demonstrate that the Rebirth-Retire protocol achieves better throughput and lower abort rate than the Wound-Retire protocol across varying levels of contention workloads.

PVLDB Reference Format:
Qian Zhang, Yiwen Xiang, Jianhao Wei, Yang Yang, Yifan Li, Xueqing Gong and Wanggen Liu. Rebirth-Retire: A Concurrency Control Protocol Adaptable to Different Levels of Contention . PVLDB, 18(9): 3162-3174, 2025.
doi:10.14778/3746405.3746435

[*]The corresponding author. [†]These authors contributed equally to this work.
Proceedings of the VLDB Endowment, Vol. 18, No. 9 ISSN 2150-8097.
doi:10.14778/3746405.3746435

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/gitzhqian/RebirthRetire.

1 INTRODUCTION

The concurrency control mechanism is a fundamental component of database systems, designed to coordinate simultaneous operations and ensure the consistency and isolation properties of transactions. Several concurrency control protocols have been developed over the years, each with unique strategies and trade-offs concerning system throughput, transaction abort rate, deadlock handling, and overhead [7, 10, 12, 13, 17, 18, 22, 24, 38, 42].

The ***Two-Phase Locking*** (2PL) protocol [7, 18] is a classic and widely used concurrency control protocol. It employs a "first-come, first-served" strategy to determine the execution order of conflicting operations. The transaction that acquires the data lock first is allowed to execute its operation, while other conflicting transactions must wait for it to release the lock. ***Deadlock-Detection*** is a commonly used method in the 2PL protocol to resolve deadlock issues. A directed wait-for graph is maintained, and the graph is periodically checked for deadlock cycles. If a cycle is detected, one of the transactions in the cycle will be aborted. It performs well in scenarios with fewer conflicts. However, in situations with more conflicts, the waiting time for transactions increases and the occurrence of deadlocks becomes more frequent, leading to many transactions being aborted after waiting for a long time.

Some concurrency control protocols use timestamps to determine the execution order of conflicting operations and avoid deadlocks. In these protocols, each transaction is assigned a unique timestamp that determines its priority for accessing data. For example, in the ***Wound-Wait*** protocol [5], younger transactions wait for older transactions to release their locks, while older transactions do not wait for younger transactions; instead, they directly abort

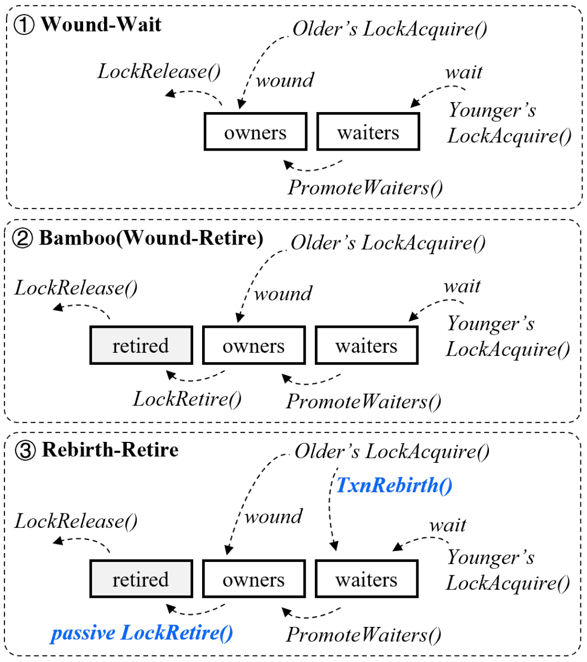

Figure 1: A lock entry and function calls.

the younger transactions. In ***Optimistic Concurrency Control*** (OCC) protocols [27, 34, 36, 43, 44], transactions do not wait for other transactions during execution. Instead, they create and use local copies of the accessed data. During the validation phase, they determine whether they can successfully commit based on timestamp ordering. Although these protocols can avoid deadlocks, in scenarios with a lot of conflicts, a large number of transactions will be aborted (in both Wound-Wait and OCC), and transactions still spend a significant amount of time waiting for locks (in Wound-Wait).

Bamboo[22] is a variant of the Wound-Wait protocol, which we refer to as ***Wound-Retire***. It violates the two-phase locking rule by introducing the lock *Retire* operation (early lock release). In the Bamboo protocol, older transactions retire their locks after accessing the data, enabling younger transactions to access the data without waiting. In scenarios with high contention, this approach can significantly reduce the time transactions spend waiting for locks. However, performing the *Retire* operation and maintaining dependencies between transactions increases the cost of concurrency control, resulting in worse performance than Wound-Wait and Deadlock-Detection in low-contention scenarios.

In this paper, we propose the ***Rebirth-Retire*** protocol, an enhanced version of the Wound-Retire protocol. One notable improvement is that older transactions no longer actively perform the *Retire* operation while holding a lock; instead, the *Retire* operation is initiated by the younger transaction requesting the lock. When no younger transactions request the lock, older transactions can hold the lock until they commit, thus saving the overhead of the *Retire* operation. Our experimental results show that this approach can reduce the cost of concurrency control and improve transaction throughput in low-contention scenarios. Another improvement is that an older transaction does not always abort the younger transactions holding the lock when it requests a lock. Instead, the older transaction is assigned a new and larger timestamp (referred to as *Rebirth*) unless doing so is determined to result in a deadlock. Before performing the *Rebirth* operation on a transaction, we detect potential deadlocks by topologically sorting the transaction and all transactions that directly or indirectly depend on it. Our experimental results demonstrate that *Rebirth* can significantly reduce the transaction abort rate and improve transaction throughput in high-contention scenarios.

In summary, this paper makes the following contributions.

- We developed the Rebirth-Retire concurrency control protocol, which employs *Rebirth* and passive *Retire* strategies to handle conflicting operations between transactions. Rebirth-Retire is provably correct.
- We also designed several optimizations to further improve Rebirth-Retire's performance.
- We conducted comparative experiments between the Rebirth-Retire protocol and other existing protocols under different levels of contention. The results show that Rebirth-Retire offers significant performance advantages across various contention levels, except in scenarios where transaction conflicts are rare.

2 BACKGROUND AND MOTIVATION

2.1 The Bamboo (Wound-Retire) Protocol

In the Wound-Wait protocol, the lock entry for each tuple maintains two lists of transactions: *owners*, representing transactions currently holding the lock, and *waiters*, representing transactions waiting for it, as shown in Figure 1-①. When a younger transaction requests the lock [*LockAcquire()*], it is added to the *waiters*. When an older transaction requests the lock, it kills the younger transactions in the *owners* and then joins the *owners*. Once transactions commit or abort, they are removed from the *owners* [*LockRelease()*]. After that, transactions in *waiters* get a chance to move into owners [*PromoteWaiters()*].

Bamboo extends the Wound-Wait protocol by introducing an additional list called *retired* in each lock entry, as shown in Figure 1-②. After a transaction has finished accessing a tuple, the transaction can be moved from *owners* to *retired* [*LockReire()*]. This allows other transactions to join *owners*. Once transactions commit or abort, they are removed from the *retired* [*LockRelease()*]. All transactions that have read the dirty updates of retired transactions or modified the tuples read by retired transactions must wait until the retired transactions they depend on commit before they can commit. Bamboo introduces a new variable, *commit_semaphore*, in transaction objects to track dependencies among transactions. A transaction T increments its own 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*. In principle, each write can be immediately followed by *LockRetire()* 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. For better performance, a lock can be retired after the transaction's last write

Table 1: The System Time Proportion, Throughput and Abort Rate of concurrency control protocols under different levels of contention.

Level	Protocol	Committed	Abort	Concurrency Control	Lock Wait	Commit Wait	Throughput	Abort Rate
Low	***DD***	48.97%	0.00%	51.03%(L)	0.00%	-	**2,896,840**	0.00%
	WW	54.69%	0.00%	45.31%(L)	0.00%	-	2,886,750	0.00%
	WR	51.15%	0.00%	35.58%(L)3.58%(R)3.05%(LC)	0.00%	0.63%	2,549,320	0.00%
	Silo	71.88%	0.00%	1.42%(L)+26.70%(LC)	-	0.00%	2,620,910	0.00%
Medium	***DD***	12.48%	8.60%	6.38%(L)7.25%(D)	65.30%	-	142,193	26.33%
	WW	37.89%	8.95%	15.84%(L)	37.31%	-	575,844	14.39%
	WR	40.12%	11.01%	31.12%(L)4.19%(R)6.95%(LC)	0.45%	5.12%	**832,912**	**12.41%**
	Silo	54.36%	32.61%	7.10%(L)5.91%(LC)	-	0.03%	366,109	32.51%
High	***DD***	1.00%	11.12%	0.79%(L)8.17%(D)	78.91%	-	12,246	86.94%
	WW	16.85%	17.53%	5.03%(L)	60.59%	-	**254,421**	**43.70%**
	WR	7.11%	50.30%	18.96%(L)4.42%(R)4.84%(LC)	4.84%	7.43%	208,507	63.47%
	Silo	25.78%	65.66%	6.23%(L)2.17%(LC)	-	0.16%	206,311	68.18%

to the tuple if the tuple may be updated more than once by the same transaction. Bamboo rely on programmer annotation or program analysis to find the last access and insert *LockRetire()* after it[22].

In Bamboo, each transaction maintains local copies of its read and write tuples, which can be read by other transactions. Essentially, it functions as a multi-version protocol. Bamboo employs two optimizations to reduce overhead. One is to directly move a read operation to *retired* whenever it can become the owner, reducing the latch overhead on *owners*. The other is to give up retiring a write operation if it brings little benefit, such as writes at the tail of a transaction. Bamboo also employs two optimizations to reduce transaction aborts. One is allowing older transactions to read the local copies of younger transactions to avoid aborting the younger transactions. The other is assigning timestamps to transactions only when they first encounter a conflict. These optimizations are not unique to Bamboo, and some are also applied in other protocols.

2.2 System Time Proportion

We conducted a detailed breakdown analysis of the time spent on various tasks during transaction execution under different levels of contention for four typical concurrency control protocols: Deadlock-Detection(**DD**) , Wound-Wait(**WW**), Wound-Retire(**WR**), and **Silo**[34]. The first three are lock-based protocols, while Silo is an optimistic protocol. Deadlock-Detection and Wound-Wait are single version protocols, while Wound-Retire and Silo are multi-version protocols. The transaction execution time is divided into the following parts:

- Processing Time: The time spent on transaction processing includes accessing tuples, performing computations, etc.
  - **Committed**: Time spent on committed transactions.
  - **Abort**: Time spent on aborted transactions.
- Concurrency Control Time: The time spent on concurrency control processing.
  - DL-Detect(**D**): Time spent on deadlock detection in the Deadlock-Detection protocol.
  - Latching(**L**): Time spent on internal data structure operations, such as operations on lock structures in lock-based protocols and operations on tuple version timestamps in timestamp-based protocols.
  - Retire(**R**): Time spent on performing *Retire* operations in the Wound-Retire protocol.
  - Local Copy(**C**): Time spent on creating local copies of tuples for transactions in multi-version protocols.
- **Lock Wait**: Time spent waiting for locks in lock-based protocols.
- **Commit Wait**: Time spent waiting for the transaction commit conditions to be satisfied in both Wound-Retire and Silo protocols.

Table 1 presents the results of the analysis experiment on YCSB workloads[1]. In the low-contention scenario, the transaction abort rates of all four protocols are zero. The primary overhead for Deadlock-Detection and Wound-Wait protocols comes from acquiring and releasing locks (Latching). For the Wound-Retire protocol, Latching and the *Retire* operation constitutes a significant portion of the overhead. For the Silo protocol, the main overhead arises from creating local copies. In the medium-contention scenario, both Deadlock-Detection and Silo experience higher abort rates. Deadlock-Detection and Wound-Wait spend a significant amount of time waiting for locks, while Wound-Retire spends minimal time waiting for locks but incurs additional overhead from *Retire* operations and commit waiting. In the high-contention scenario, all protocols exhibit a high abort rate, resulting in a significant increase in the proportion of time spent on aborted transactions. Deadlock-Detection and Wound-Wait still spend the majority of their time waiting for locks, while Wound-Retire sees an increase in both lock wait time and commit wait time.

According to the analysis experiment, we conclude the following:

- In low-contention scenarios, acquiring and releasing locks (Latching) constitute the primary overhead for lock-based protocols while creating local copies of data is the main

[1]The experimental setup is described in Section 5.1.

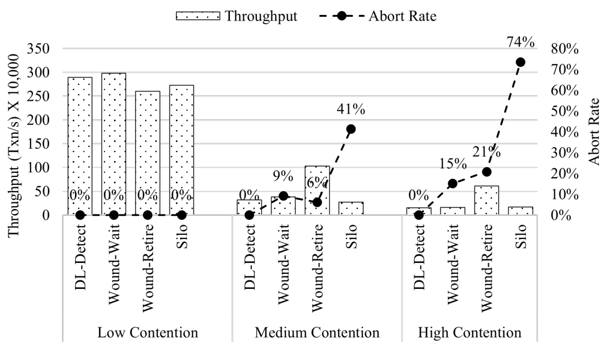

Figure 2: The Throughput and Abort Rate of concurrency control protocols under different levels of contention (Each transaction accesses tuples in Key Order).

overhead for Silo. Other single-version optimistic protocols should have better performance than Silo.

- As contention increases, lock waiting becomes the dominant factor affecting the performance of lock-based protocols. In contrast, Silo experience a significant rise in transaction abort rates, resulting in poorer performance.
- The *Retire* operation can significantly reduce the time spent waiting for locks. However, it introduces additional overhead and increases commit waiting time due to transaction dependencies. As a result, the performance of the Wound-Retire protocol is worse than that of Wound-Wait in both low-contention and high-contention scenarios.
- No concurrency control protocol can achieve optimal performance across all scenarios. Simple protocols such as Deadlock-Detection, Wound-Wait and Silo are better suited for low-contention scenarios, while more complex, well-designed protocols like Wound-Retire perform better in high-contention scenarios.

In this paper, we propose an improvement to the Wound-Retire protocol, where transactions no longer actively perform the *Retire* operations. Instead, the Retire operation is passively initiated by the transaction requesting the lock. This modification significantly enhances the performance of the protocol in low-contention scenarios.

2.3 Unnecessary Abort

Deadlock handling is a fundamental task of concurrency control protocols. Deadlock-Detection uses a detection-based approach to identify deadlocks, while Wound-Wait and Wound-Retire avoid deadlocks by adopting an "*Older Kill Younger*" strategy based on timestamp ordering. Since the probability of deadlock occurrence is low in most scenarios [19, 20], the time spent on deadlock detection in Deadlock-Detection protocol does not constitute a significant proportion in high-contention scenario, as shown in Table 1. In the Wound-Wait and Wound-Retire protocols, while the "*Older Kill Younger*" strategy can prevent deadlocks, it inevitably results in the premature termination of some transactions that would not have caused a deadlock. For optimistic protocols, although the optimistic strategy can prevent lock waiting, it leads to the abort of some pre-committed transactions.

We demonstrate the occurrence of unnecessary transaction aborts in the protocols through an experiment on YCSB workloads[2]. In this experiment, each transaction accesses the tuples sequentially in key order, thereby preventing deadlocks. According to the experimental results shown in Figure 2, as the level of contention increases, the number of transactions that are aborted also increases. All of these aborts are unnecessary.

In this paper, we propose an improvement to the deadlock handling method of the Wound-Wait and Wound-Retire protocols. Older transactions will only abort younger transactions when a deadlock is imminent; otherwise, we avoid aborting younger transactions by performing a *Rebirth* operation, which assigns a larger timestamp to the older transaction. In high-contention scenarios, the *Rebirth* mechanism can reduce unnecessary transaction aborts while preventing deadlocks, thereby improving system performance.

3 REBIRTH-RETIRE PROTOCOL

This section describes the Rebirth-Retire protocol in detail, which is developed based on Wound-Retire[22]. We first describe the ideas of passive *Retire* and *Rebirth*, followed by a detailed description of the data structures and the pseudocode of the protocol. Finally, proof of the protocol's correctness and a discussion of compatibility are provided.

3.1 Passive Retire and Rebirth

3.1.1 ***Passive Retire.*** According to Table 1, when there are numerous operation conflicts among concurrent transactions, lock waiting becomes the primary factor affecting system performance. The Wound-Retire protocol breaks the two-phase locking protocol by employing the *Retire* mechanism, which allows transactions holding locks to release them early, thereby reducing lock waiting times for other transactions. The *Retire* operation enables Wound-Retire to achieve better performance than other lock-based protocols in high-contention scenarios.

In the Wound-Retire protocol, transactions need to perform *Retire* operations in addition to acquiring and releasing locks, which introduces additional overhead. Intuitively, the *Retire* mechanism benefits system performance when the overhead introduced by *Retire* operations is less than the reduction in waiting times for other transactions. However, when the overhead of *Retire* operations exceeds the reduction in waiting times for other transactions, it can negatively impact system performance. For example, when no other transactions are waiting for a transaction to release its locks, performing *Retire* operation provides no benefit at all. This explains why Wound-Retire performs worse than Deadlock-Detection and Wound-Wait in the low-contention scenario.

To reduce unnecessary *Retire* operations, we propose a passive *Retire* strategy. The basic idea is that transactions holding locks no longer actively perform *Retire* operations; instead, *Retire* operations are initiated by other transactions waiting for the locks.

[2]The experimental setup is described in Section 5.1 except accessing tuples in key order in transactions.

For transactions holding shared locks, the timing and initiator of the *Retire* operations do not affect their normal execution. For transactions holding exclusive locks, the *Retire* operations must be performed after they have completed their current write operations on the tuples. To address this, we introduce a flag in the metadata of the new version generated by the transaction to indicate the completion of the write operation. When using the passive *Retire* strategy, the locks held by a transaction will not be retired when no other transactions are waiting, thus avoiding unnecessary overhead.

For better performance, an exclusive 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, same as Wound-Retire, we can rely on *programmer annotation* or *program analysis* to find the last write and perform passive *Retire* after it.

3.1.2 ***Rebirth****.* In the Wound-Wait and Wound-Retire protocols, each transaction is assigned a timestamp representing its priority. When a transaction with the smaller timestamp (*Older*) acquires a lock and another transaction with a larger timestamp (*Younger*) holds the lock, the older transaction will abort the younger transaction. Although this strategy can prevent deadlocks, it may lead to unnecessary transaction aborts (see section 2.3). The basic idea of *Rebirth* is to assign a new, larger timestamp to the older transaction, thereby avoiding aborting the younger transaction holding the lock.

One challenge in performing *Rebirth* on transactions is handling deadlocks. To track dependencies between transactions, unlike Wound-Retire, we do not maintain a dependency counter for each transaction. Instead, each transaction maintains a list of transactions it depends on (*Parents*) and a list of transactions that depend on it (*Children*).

When transaction A acquires a lock, it discovers that a younger transaction B holds the lock and conflicts with it. Before performing the *Rebirth* operation on A, we first perform a topological sort on A and A's descendants (all transactions that directly or indirectly depend on A), based on transactions' children lists. The topological sorting produces a sorted list of A and A's descendants. If B appears in the sorted list, it indicates that the *Rebirth* operation would result in a deadlock. In this case, the transaction B is aborted. After checking all the owners of the lock, the transactions in the sorted list are assigned a larger timestamp one by one. This strategy can reduce transaction aborts while avoiding deadlocks. The time complexity of the topological sort algorithm (Kahn's Algorithm) [25] is $O(V+E)$, where V is the number of transactions involved and E is the number of dependencies between these transactions.

3.2 Protocol Description

3.2.1 ***Data Structures****.* Similar to the Wound-Retire protocol, in the Rebirth-Retire protocol, each tuple's lock entry contains three lists: **tuple.waiters**, **tuple.owners**, and **tuple.retired**(see Figure 1-③). Each entry in the lists is a $(txn, lock\_type)$ pair, where txn is a transaction object, and $lock\_type$ is the type of lock requested by the transaction. Each list is sorted based on the timestamps of transactions in it.

To track dependencies between transactions, unlike Wound-Retire, which uses the *commit_semaphore* variable, we maintain two lists in a transaction object: **txn.parents** and **txn.children**, which respectively contain the transactions it depends on and the transactions that depend on it directly. Similarly, these two lists are also sorted according to the transactions' timestamps. Additionally, each transaction maintains a **txn.status** variable, which indicates whether the transaction has been committed or aborted by another transaction.

The advantage of the *commit_semaphore* variable is that it can be updated using atomic instructions. However, it only records the number of dependencies between transactions rather than the actual dependency relationships. For example, if transaction x depends on transaction y for both tuple A and tuple B, x's *commit_semaphore* will be incremented twice. After y commits, x's *commit_semaphore* must also be decremented twice. Although updating the *parents* and *children* lists in *Rebirth-Retire* incurs higher overhead than updating *commit_semaphore*, it enables tracking of actual dependency relationships between transactions, which can be used to detect potential deadlocks. Additionally, a transaction does not need to notify other transactions when it finishes. Each transaction can independently determine whether it can commit based on the status of the transactions in its *parents* list. In Section 4.1, we propose an optimization that use an 8-byte word to implement the *children* list, which can also be updated using atomic instructions.

In order to determine whether a transaction holding a lock has completed its current write operations before performing a passive *Retire* operation, each tuple version has a **version.ready** variable. This variable serves as an indicator to track the completion status of the current write operation, ensuring that the lock can only be safely retired once the transaction has finished its operation.

3.2.2 ***Function Calls****.* The Rebirth-Retire protocol is an improvement to the Wound-Retire protocol. See Figure 1-③, one difference is that *LockRetire()* is no longer initiated proactively by the transaction. Another difference is that an older transaction first executes *TxnRebirth()* when requesting a lock and only kills the youngers if a deadlock is confirmed. We Implement the protocol based on the algorithm of the Wound-Retire protocol, with the differences highlighted using a gray background color. In the following, we walk through the algorithm step-by-step.

Lifecycle Algorithm 1 illustrates the lifecycle of how the database executes a transaction using the Rebirth-Retire protocol. The differences from the Wound-Retire protocol are as follows:

- A transaction no longer actively performs *Retire* operations. (The *LockRetire()* statement is removed from the algorithm.)
- A transaction determines whether it satisfies the commit condition by checking the statuses of transactions in its *Parents* list (line 4).

LockAcquire()

In the Rebirth-Retire protocol, when a transaction encounters a conflict while requesting a lock, it no longer directly aborts the younger conflicting transactions. Instead, it first chooses to perform a *Rebirth* operation on itself (Algorithm 2 lines 9-10). During a single *LockAcquire()* call, a transaction will be rebirthed only once (line 11). If the transaction's timestamp remains unchanged after performing *Rebirth*, it indicates that the lock request would result in a deadlock, and the younger conflicting transactions will be aborted (lines 12-13). However, if the transaction's timestamp has been modified

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

```
    # req_type is SH or EX
1   LockAcquire(txn, req_type1, tuple1)
    ...
2   # LockRetire(txn,tuple1)
3   LockAcquire(txn, req_type2, tuple2)
    ...
    # Wait for transactions that the txn depends on
4   while all parents have terminated & txn.status≠abort do
5       pause
6   if txn.status≠abort then
7       writeLog() # Log to persistent storage device
8   LockRelease(txn, tuple1, txn.status)
9   LockRelease(txn, tuple2, txn.status)
10  txn.terminate(txn.status)
```

Algorithm 2: LockAcquire() and LockRelease() —- Differences between Wound-Retire and Rebirth-Retire are highlighted in gray .

```
1   Function LockAcquire(txn, req_type, tuple)
2       has_conflicts = FALSE
3       rebirthed = FALSE
4       youngers = NIL
5       for (t, type) in concat(tuple.retired, tuple.owners) do
6           if conflict(req_type, type) then
7               has_conflicts = TRUE
            # A transaction rebirth only once
8           if has_conflicts & !rebirthed & txn.ts < t.ts then
9               youngers = The tail of concat(retired, owners)
                    starting from t
10              TxnRebirth(txn, youngers)
11              rebirthed = TRUE
12          if has_conflicts & txn.ts < t.ts then
13              t.set_abort()
14      tuple.waiters.add(txn)
15      PromoteWaiters(tuple)
16  Function LockRelease(txn, tuple, txn_status)
17      if txn_status==abort & txn.getType(tuple)==EX then
            # Cascading abort
18          abort all transactions in txn.children
19      remove txn from tuple.retired or tuple.owners
20      PromoteWaiters(tuple)
```

after performing *Rebirth*, it is now younger than all conflicting transactions, and those conflicting transactions will not be aborted.
LockRelease()

Algorithm 3: PromoteWaiters()—Differences between Wound-Retire and Rebirth-Retire are highlighted in gray .

```
1   Function PromoteWaiters(tuple)
2       for t in tuple.waiters do
3           if conflict(t.type, tuple.owners.type) then
4               if tuple.owners.type == EX then
                    # Wait for the owner to complete its
                      current write operation
5                   while !tuple.owners.version.ready do
6                       pause
                # Move transactions in tuple.owners to
                  tuple.retired (Retire)
7               for t' in tuple.owners do
8                   tuple.owners.remove(t')
9                   tuple.retired.add(t')
            # Promote trandaction t
10          tuple.waiters.remove(t)
11          tuple.owners.add(t)
            # Tracking dependencies
12          for
              t' in tuple.retired in descending timestamp order
             do
13              if conflict(t'.type,t.type) then
14                  t.parents.add(t')
15                  t'.children.add(t)
16                  break
```

When an aborted transaction releases an exclusive lock, all transactions in its *Children* list will be aborted (Algorithm 2 lines 17-18). We no longer use a counter, as in the Wound-Retire protocol (*transaction.commit_semaphore*), to track dependencies between transactions. Therefore, all statements related to counter maintenance are removed from the algorithm.
PromoteWaiters()
Inside PromoteWaiters(), the algorithm scans waiters in the growing timestamp order (Algorithm 3 line 2). For each transaction, if it conflicts with the owner(s) and the owner holds an exclusive lock (lines 3-4), the transaction will wait for the owner to complete its current write operation (lines 5-6). Afterward, the transaction will *Retire* the owner(s) (lines 7-9). It will then be moved from the waiters list to the owners list (lines 10-11). In Rebirth-Retire, we only need to maintain the dependencies between the transaction that just became an owner and the last transaction that conflicted with it in the retired list. (lines 12–16).
TxnRebirth()
Inside TxnRebirth(), We first perform a topological sort on the requesting transaction and all transactions that depend on it (Algorithm 4 line 3). Then, for each transaction t in the youngers

Algorithm 4: TxnRebirth()

```
1 Function TxnRebirth(txn,youngers)
2     sort_list = NIL
      # Perform topological sorting on the transaction
        and its dependencies using Kahn's algorithm.
3     sort_list = KahnTopologicalSorting(txn)
4     for (t, type) in youngers do
5         if t in sort_list then
              # A deadlock is found
6             abort transaction t
7     for t in sort_list do
          # Rebirth the tansactions (Largest)
8         t.ts = ++global_ts
```

list, if it appears in the *sort_list*, it will be aborted (lines 6). Subsequently, all transactions that depend on t will also be aborted in a cascading manner. Finally, we assign new timestamps to the active transactions in the *sort_list* one by one (lines 7-8). Here, we apply the *Largest* strategy, with further optimization of the timestamp assignment process discussed in Section 4.3.

3.3 Proof of Correctness

In this section, we prove that the Rebirth-Retire protocol is able to enforce serializability correctly. According to the serializability theory, a schedule of transactions is serializable if and only if there's no cycle in its serialization graph. The serialization graph represents the schedule as a directed graph, where each node represents a committed transaction, and each edge represents a conflict between two committed transactions.

The Rebirth-Retire protocol is an improvement of the Wound-Retire protocol [22]. We prove serializability following the proof of Wound-Retire.

Definition 1. ***[Commit Point]*** *A transaction's commit point is a point in time when it recorded the operations in log. (In Algorithm 1, the commit point is after finishing line 8 but before starting line 9.)*

Lemma 1. ***[Commit Point Ordering]*** *In Rebirth-Retire, if the serialization graph contains $T_i \rightarrow T_j$, then T_j reaches the commit point after T_i.*

Proof. Without loss of generality, we assume T_i and T_j conflict on tuple x. It means that T_j depends on T_i. In other words, T_j is in T_i.children and T_i is in T_j.parents. We know that T_j can reach its commit point only when all its parents have terminated (lines 5–6 in Algorithm 1). T_i can terminate only after it has released its lock on x , which can happen only after T_i has reached its commit point (line 11 in Algorithm 1). Together, this means T_j reaches its commit point after T_i. □

Theorem 1. *Every schedule in Rebirth-Retire is serializable.*

Proof. According to Lemma 1, every edge $T_i \rightarrow T_j$ in the serialization graph means T_j reaching the commit point after T_i. Therefore, no cycle may exist since a transaction cannot reach the commit point after it has already reached the commit point, finishing the proof. □

4 OPTIMIZATIONS

As an improvement over *Wound-Retire*, all its optimizations are also applicable to *Rebirth-Retire*. A read operation is moved directly to the *retired* list whenever it can become the owner. We also give up retiring a write operation if it brings little benefit under the same conditions as *Wound-Retire*. Similar to *Wound-Retire*, we minimize *Rebirth* operations by allowing older transactions to read the old versions of tuples. In *Rebith-Retire*, a transaction is assigned its first timestamp only when it encounters a conflict for the first time. In addition, we introduce some new optimization methods.

4.1 O1: Latch-free Dependency Tracking

In *Rebirth-Retire*, we track dependencies between transactions using the *parents* and *children* lists maintained within each transaction object. Since a transaction's *parents* is updated only by itself, we simply implement it using the sorted list provided by the *C++ STL* library. Since a transaction's *children* may be updated concurrently by multiple transactions, we initially implemented it using *Intel TBB*'s open-source lock-free concurrent data structures.

However, we found that implementing a latch-free *Children* list can be further simplified by manipulating the bits of an 8-byte atomic word. Specifically, each worker thread is assigned a unique bit within this atomic word, with the bit's offset determined by its thread ID modulo 64. When a transaction needs to be added to the *children* list, it sets the corresponding bit of its worker thread to 1 using an atomic instruction. The transaction manager maintains the transaction object currently being executed by each worker thread. During the topological sorting, a transaction needs to check the worker threads corresponding to the bits set to 1 in the *Children* atomic word to determine whether it is present in the *Parents* list of the transactions on those worker threads. If not, the transaction resets the bit to zero, indicating that the dependency has been resolved. Since multiple worker threads may be mapped to the same bit, all these worker threads need to be checked simultaneously. Although this may incur some overhead, it does not affect the correctness of the algorithm.

4.2 O2: Optimistic Read Descendant

When performing a *Rebirth* operation on a transaction, all its descendant transactions must first undergo topological sorting. Then, based on the sorting result, each descendant transaction is assigned a new timestamp in sequence. To ensure that the timestamps of descendant transactions are not modified by other transactions during this process, we need to lock these transaction objects. However, this approach may delay the rebirth operations of other transactions.

To mitigate this challenge, we adopt an optimistic read strategy [30] to access descendant transaction object. Each transaction object has a *Version* counter. During the topological sorting process, instead of locking descendant transactions objects, we record their *Versions*. A new timestamp can only be assigned to a descendant transaction if its *Version* has not been modified; otherwise, the topological sorting must be re-executed. Now, we lock the descendant transaction objects only when updating their timestamps and *Versions*, minimizing the impact on other transactions.

4.3 O3: Assign Larger Timestamps

When performing a *Rebirth* operation on a transaction, we assign new timestamps to it and all its descendant transactions. As shown in algorithm 4, we always assign the current largest timestamp in the system to these transactions in sequence (the "*Largest*" strategy), which would result in the lowest priority for these transactions. Meanwhile, this approach may cause a timestamp allocation bottleneck [42] and lead to tail latency issues [41]. Here, we present another approach, which assigns timestamps that are slightly larger than those of all other transactions holding locks and conflicting with it (the "*Larger*" strategy). For example, when performing *Rebirth* on transaction x, if the largest timestamp among all transactions that hold locks and conflict with x is t, then we sequentially assign $t+1, t+2, \ldots$ to x and its descendants. To prevent assigning the same timestamp to two different transactions in the system, we use a hybrid timestamp scheme, which incorporates the worker thread ID of the executing transaction as part of the timestamp[3].

4.4 O4: Version Prefetching

Similar to *Wound-Retire*, in *Rebirth-Retire*, a tuple may have multiple versions, and transactions are allowed to read the corresponding version based on their timestamps. When there are a large number of concurrent updates to hotspot tuples in the system, their version chains may grow very long. Since tuple versions are typically stored in a heap structure, transactions may experience a significant number of cache misses while traversing these version chains.

To address this, we leverage the software prefetching technique [2, 23] by introducing an additional *jump pointer* field in each tuple version, which points to the version that should be prefetched next. This method could reduces a transaction's cache misses when traversing long version chains by using prefetch instructions (e.g., `__mm_prefetch`).

5 EXPERIMENTAL EVALUATION

5.1 Experimental Setup

We implement Rebirth-Retire in DBx1000 [1, 42], a multi-threaded, in-memory DBMS prototype. It stores all data in a row-oriented manner and employs worker threads to invoke transactions from a fixed-length queue in stored-procedure mode. Each transaction contains multiple query invocations. In our experiments, each worker thread is assigned a maximum of 100,000 transactions to execute. The experiment stops as soon as any worker thread completes its maximum transaction count. This is to ensure that the number of concurrent transactions in the experiment does not change due to some worker threads completing their tasks.

DBx1000 includes a pluggable lock manager that supports different concurrency control protocols. This allows us to compare *Rebirth-Retire* with various baselines within the same system. The concurrency control protocols used in the experiment include:

- **Deadlock-Detection (DL-Detection)**: a 2PL protocol with deadlock detection, which is implemented by [42].
- **Wound-Wait**: a variant of 2PL with deadlock prevention strategy, Which is implemented by [22].

[3]The hybrid timestamp schema is used in many systems [1, 4, 26, 29, 34].

Table 2: Workloads with varying levels of contention

Benchmark	Factors	Low	Medium	High
YCSB	***#Tuples(×1000)***	10,000	1,000	1,000
	#Transactions	10	20	40
	#Queries	4	16	16
	%Writes	10%	50%	90%
	Access Skew	$\theta = 0$	$\theta = 0.85$	$\theta = 0.9$
TPC-C	***#Warehouses***	100	1	1
	#Transactions	10	10	40

- **Wound-Retire (Bamboo)**: a variant of Wound-Wait with releasing locks early, which is implemented by [22].
- **Silo**: an optimistic concurrency control protocol, which is implemented by [42].
- **TicToc**: an optimistic concurrency control protocol, which dynamically adjusts transaction timestamps during execution. It is implemented by [42].
- **MOCC**: a concurrency control protocol that combines the benefits of optimistic concurrency control with pessimistic locking, which is implemented by [11].
- **Rebirth-Retire (RR)**: Our concurrency control protocol[4].

The experiments were run on a server equipped with an Intel Xeon Gold 5218R CPU (20 physical cores@2.10GHz and 28 MB last level cache) and 96 GB DDR4 DRAM, running Ubuntu 20.04 LTS. Each core supports two hardware threads, for a total of 40 threads. We collect transaction statistics, including throughput and abort rates, by running each workload at least five times and averaging the results. The throughput is calculated by dividing the total number of committed transactions by the total running time, measured in transactions per second. The abort rate is calculated by dividing the total number of aborted transactions by the total number of transaction attempts, which includes both committed and aborted transactions.

Two benchmarks are implemented in DBx1000, YCSB (The Yahoo! Cloud Serving Benchmark)[14] and TPC-C[15]. YCSB is representative of large-scale online services. Each query accesses a single random tuple based on a Zipfian distribution with a parameter (θ) that controls the contention level in the benchmark [21]. Each tuple has a primary key column and 10 additional columns, each containing 100 bytes of randomly generated string. TPC-C is the current industry standard for evaluating OLTP systems. It consists of nine tables that simulate a warehouse order processing application. In our experiment, all five transactions of TPC-C are used, with their default proportions as follows: *NewOrder* (45%), *Payment* (43%), *OrderStatus* (4%), *Delivery* (4%), and *StockLevel* (4%).

According to the previous studies[6, 20], many factors influence the probability of conflicts between transactions in a workload. It is positively correlated with the number of concurrently running transactions(***#Transactions***), the average number of queries per transaction(***#Queries***), the skew of data access(***Access Skew***), and the proportion of write operations in queries(***%Writes***). On the other hand, it is inversely proportional to the number of tuples in

[4]https://github.com/gitzhqian/RebirthRetire

Table 3: The Throughput of optimizations

TPS(×1000)	Low	Medium	High
Basic	2,773	1,239	538
+O1	(+1.27%) 2,808	(+3.70%) 1,284	(+8.22%) 582
+O2	(+0.79%) 2,795	(+1.89%) 1,262	(+4.16%) 560
+O3	(+0.68%) 2,792	(+3.54%) 1,282	(+7.57%) 579
+O4	(+0.08%) 2,775	(+0.54%) 1,245	(+2.06%) 549
+All	(+1.97%) 2,827	(+5.18%) 1,303	(+12.36%) 605

the database(***#Tuples***). Together, these factors determine how likely conflicts are to occur during transaction execution. We designed workloads with varying levels of contention by adjusting parameters related to these factors. TPC-C primarily controls the level of contention in a workload through two parameters: the number of warehouses(***#Warehouses***) and the number of concurrent transactions(***#Transactions***). Table 2 presents details of the workloads with varying levels of contention.

5.2 Evaluation of Optimizations

We first evaluate the optimizations presented in Section 4 to assess their individual effect on the performance of *Rebirth-Retire*. To do this, we conduct experiments using the YCSB workload at different contention levels, running *Rebirth-Retire* multiple times with each optimization enabled individually. We use six configurations:

- **Basic**: *Rebirth-Retire* without any optimizations;
- **+O1**: *Rebirth-Retire* with Latch-free Dependency Tracking;
- **+O2**: *Rebirth-Retire* with Optimistic Read Descendant;
- **+O3**: *Rebirth-Retire* with Assign Larger Timestamps;
- **+O4**: *Rebirth-Retire* with Version Prefetching;
- **+All**[5]: *Rebirth-Retire* with all optimizations.

The results in Table 3 show the throughput performance. Under low contention, *Rebirth-Retire* benefits less from the optimizations due to fewer conflicts. As contention increases, more dependencies between transactions need to be tracked. The *Latch-free Dependency Tracking* optimization(O1) provides an 8.22% throughput gain in high-contention scenario. This improvement is attributed to the lower update cost of the *Children* list in the latch-free implementation. The *Optimistic Read Descendant* optimization(O2) increases throughput by up to 4.16%, primarily by reducing unnecessary transaction aborts. The *Assign Larger Timestamps* optimization(O3) also delivers a significant performance gain in high-contention workload. However, the *Version Prefetching* optimization(O4) offers limited improvement. In low, medium, and high contention scenarios, applying all four optimizations together results in 1.97%, 5.18%, and 12.36% throughput improvements, respectively.

5.3 Evaluation of Passive Retire and Rebirth

Next, we compare the performance impact of passive *Retire* and *Rebirth* mechanisms in the Rebirth-Retire protocol. We refer to the *Retire* mechanism in Wound-Retire as active *Retire*.

Compared to active *Retire*, passive *Retire* introduces no additional overhead and does not increase transaction waiting time. In low

[5]**+All** is the default configuration for all other experiments in this paper.

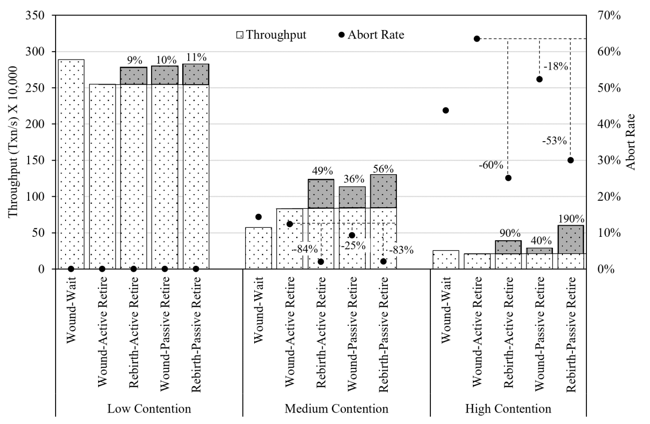

Figure 3: The evaluation of Passive Retire and Rebirth

contention scenarios, it helps reduces unnecessary *Retire* overhead. Unlike the *Wound* mechanism, *Rebirth* only terminates the younger transactions when a deadlock is confirmed. It effectively reduces unnecessary transaction aborts in high-contention scenarios. For clarity, we define the evaluated protocols as follows:

- **Wound-Active Retire** corresponds to *Wound-Retire*;
- **Rebirth-Active Retire** represents *Rebirth-Retire* without passive *Retire*;
- **Wound-Passive Retire** denotes *Rebirth-Retire* without *Rebirth*;
- **Rebirth-Passive Retire** denotes *Rebirth-Retire.*

Effects of Passive Retire. As shown in Figure 3, passive *Retire* outperforms active *Retire* across all YCSB workloads under different contention levels. In low and medium contention workloads, passive *Retire* improves throughput by 10% and 36%, respectively, compared to active *Retire*, as it performs fewer *Retire* operations. This is further supported by the proportion of time spent on *Retire* operations. As shown in Table 4, Wound-Passive Retire spends significantly less time on *Retire* operations under these workloads. Even under high contention, passive *Retire* continues to achieve higher throughput and lower abort rates than active *Retire*. These results demonstrate that passive *Retire* consistently outperforms active *Retire* across all scenarios.

Effects of Rebirth. The results in Figure 3 show that Rebirth-Active Retire consistently outperforms Wound-Active Retire in both throughput and abort rate across all workloads. In low contention scenarios, where conflicts are minimal, both protocols achieve low abort rates, but Rebirth-Active Retire achieves slightly higher throughput. Under medium contention, Rebirth-Active Retire improves throughput by 49% and reduces the abort rate by 84% compared to Wound-Active Retire, as the *Rebirth* mechanism more effectively resolves conflicts. In high contention workloads, Rebirth-Active Retire achieves approximately ~1.90× higher throughput than Wound-Active Retire, which suffers from excessive aborts. This performance gap occurs because Wound-Active Retire frequently aborts younger transactions unnecessarily, whereas Rebirth-Active Retire prevents such false aborts by accurately detecting true cycle

Table 4: The time proportion of Retire, Rebirth and Topological Sorting

Level	Protocol	Retire	Rebirth	Sort
Low	**Wound-Active Retire**	3.58%	-	-
	Rebirth-Active Retire	4.22%	0%	0%
	Wound-Passive Retire	0.09%	-	-
	Rebirth-Passive Retire	0%	0%	0%
Medium	**Wound-Active Retire**	4.19%	-	-
	Rebirth-Active Retire	5.34%	0.44%	0.85%
	Wound-Passive Retire	0.97%	-	-
	Rebirth-Passive Retire	2.54%	0.49%	0.97%
High	**Wound-Active Retire**	4.42%	-	-
	Rebirth-Active Retire	5.13%	1.88%	1.62%
	Wound-Passive Retire	1.05%	-	-
	Rebirth-Passive Retire	2.07%	2.38%	2.64%

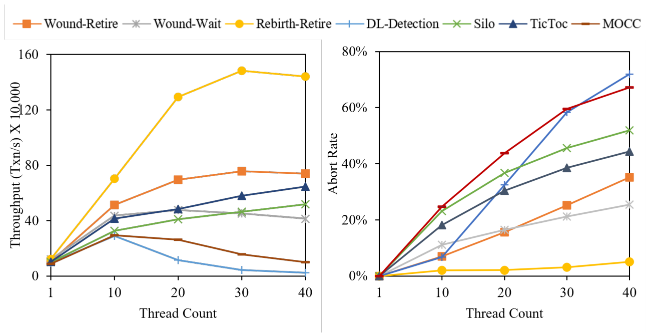

Figure 4: YCSB performance with varying the number of work thread (*θ*=0.9, *read_ratio*=0.5, 10 million tuples).

dependencies, leading to a 60% reduction in abort rate. By combining passive *Retire* and *Rebirth*, Rebirth-Retire achieves 2.9× higher throughput than Wound-Retire while reducing the abort rate by 53%. This demonstrates that the *Rebirth* mechanism effectively mitigates unnecessary aborts and significantly enhances throughput, especially in conflict-heavy transaction scenarios.

Topological Sorting Cost. From the proportion of time in Table 4, we see that the time spent on *Rebirth* operations is minimal across all scenarios, indicating that its overhead is negligible. Furthermore, profiling the time spent on topological sorting confirms that its overhead is also low and does not constrain performance throughput. This is because Rebirth-Retire prevents deadlocks and minimizes the waiting time for requesting transactions.

5.4 Evaluation on Various Workloads

We now evaluate how Rebirth-Retire performs in comparison to other protocols under varying thread counts, data access distribution, and read ratio. These workloads are commonly studied in concurrency control research, ensuring a comprehensive and comparative evaluation.

Varying Number of Threads. Figure 4 shows the improvement of Rebirth-Retire over Wound-Retire with varying thread counts in YCSB, configured with *θ*=0.9/*read_ratio*=0.5. Rebirth-Retire reduces waiting time and minimizes transaction aborts. With 40 threads, Rebirth-Retire achieves approximately 3× lower abort rates and 2× higher throughput than Wound-Retire. Both Wound-Wait and OCC exhibit similar throughput, with Wound-Wait being hindered by blocking and lock contention, and OCC suffering from high abort rates during validation. Deadlock-Detection has the worst scalability due to its deadlock detection mechanism, which results in a high abort rate and increased lock waiting times.

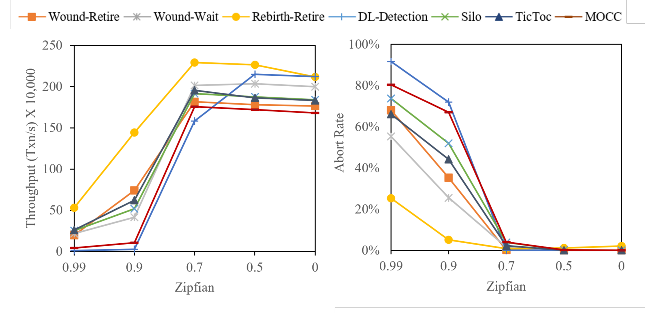

Figure 5: YCSB performance with varying access distribution (*40 work threads, read_ratio=0.5,* 10 million tuples).

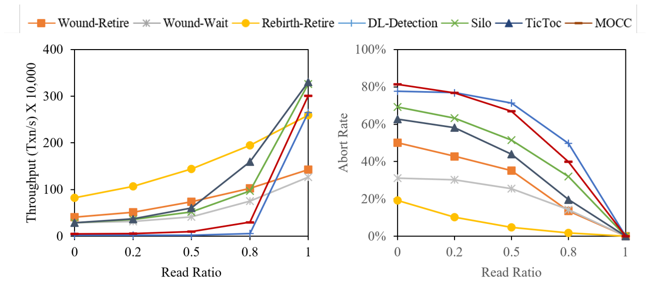

Figure 6: YCSB performance with varying the read ratio (*40 work threads, θ*=0.9, 10 million tuples).

Varying Data Access Distribution. Figure 5 illustrates the impact of the parameter *θ* in the Zipfian distribution on protocol performance. The results indicate that for *θ* values below 0.7, the skewness introduced by the distribution has little effect on performance. However, for higher *θ* values, increased skewness leads to higher contention, causing a sudden drop in throughput and making all protocols non-scalable. Rebirth-Retire consistently outperforms the others, with its performance improvement being most significant under highly skewed workloads. This is due to Rebirth-Retire's ability to reduce unnecessary aborts and minimize rebirth overhead. Wound-Wait is slower due to increased lock waiting time. Wound-Retire performs worse than Wound-Wait because it suffers from cascading aborts and commit wait times. Both Deadlock-Detection and OCC continue to perform poorly due to their high abort rates.

Varying Read Ratio. Figure 6 illustrates the impact of varying *read_ratio* on the performance of different protocols. As the read ratio decreases, throughput for all protocols declines. This is because

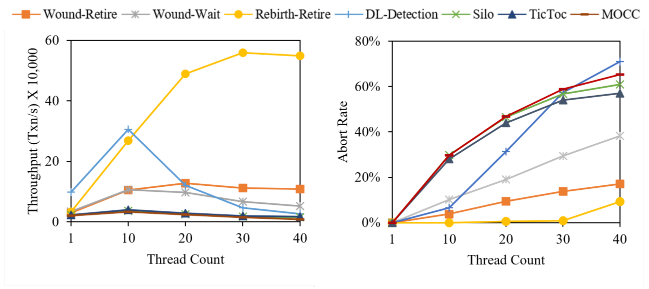

Figure 7: YCSB performance with 5% long read-only transactions (θ=0.9, *read_ratio=0.5*, 10 million tuples).

a higher proportion of write operations increases contention, leading to more frequent transaction aborts due to conflict resolution. Rebirth-Retire consistently outperforms all other protocols across different read ratios, except when *read_ratio* = 1, where transactions do not conflict with each other. In this scenario, OCC protocols, such as Silo and Tictoc, achieve better performance than the other algorithms due to their lower concurrency control overhead. Both Wound-Wait and Wound-Retire incur additional overhead from concurrency control mechanisms, such as timestamp allocation and the maintenance of wait, owner, and retire lists. Rebirth-Retire's passive retire mechanism, dynamic timestamp allocation, and lock-free dependency tracking effectively reduce this overhead, especially in less contentious workloads.

The Long Transaction. This experiment uses a workload consisting of 5% long read-only transactions accessing 1000 tuples and 95% read-write transactions accessing 16 tuples. Figure 7 shows that when the workload is less contentious, Deadlock-Detection performs the best due to its lower latch overhead compared to the other protocols. As the number of threads exceeds 20, Rebirth-Retire outperforms all other protocols. Compared to Deadlock-Detection, OCC, and Wound-Wait, Rebirth-Retire benefits from reduced waiting times and rarely transaction aborts, achieving up to a 10× performance improvement. Under 40 threads, both Wound-Retire and Rebirth-Retire exhibit similar abort rates; however, Rebirth-Retire achieves 5× higher throughput than Wound-Retire. This improvement is attributed to Rebirth-Retire's optimization in hiding the memory latency of long read-only transactions, as these transactions do not block concurrent read-write transactions and can target visible versions faster than the traditional linear version traversal. In contrast, although Wound-Retire optimizes read operations, it is still constrained by local read copies. OCC exhibits the worst performance across all scenarios, as long transactions may suffer from starvation, and aborts dominate the execution time.

5.5 Experimental Analysis on Workloads at Different Contention Levels

Finally, we compare Rebirth-Retire with other protocols on YCSB and TPC-C workloads at different contention levels.

Figure 8 shows the performance of the seven protocols across different YCSB workloads. In the low contention workload, all protocols perform similarly, achieving nearly identical throughput with no transactions aborts. In the medium contention workload,

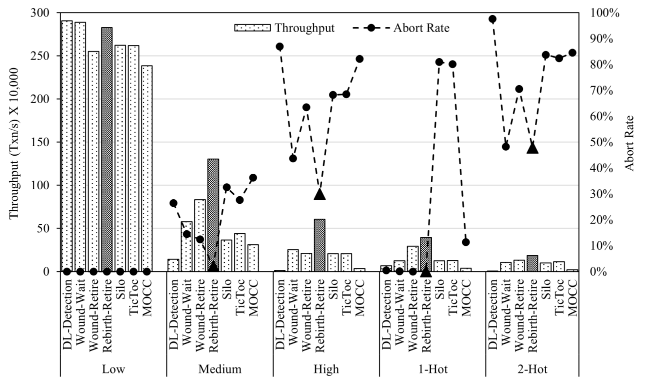

Figure 8: Throughput and Abort Rate of concurrency control protocols under different levels of contention (YCSB)

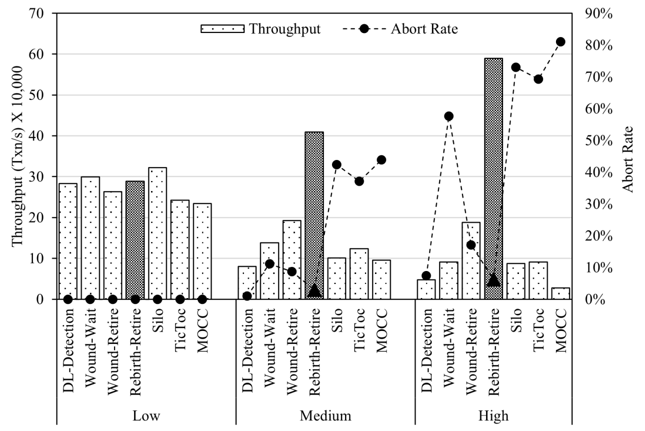

Figure 9: Throughput and Abort Rate of concurrency control protocols under different levels of contention (TPC-C)

Rebirth-Retire achieves the highest throughput, followed by Wound-Retire, Wound-Wait, OCC, and Deadlock-Detection. Rebirth-Retire outperforms Wound-Retire by 33%, and Wound-Wait achieves only about half of Rebirth-Retire's throughput. Rebirth-Retire also has the lowest abort rate (~2%), while Wound-Retire suffers from a higher abort rate (~12%). Wound-Wait, Deadlock-Detection, and OCC protocols have even higher abort rates (14%, 26%, 33%, 28% and 36%, respectively). In the high contention workload, Rebirth-Retire achieves the highest throughput (~600K transactions/s), while the others perform poorly, with throughput below 300K transactions/s. It also maintains the lowest abort rate (~30%) across all protocols.

Moreover, to assess the impact of conflicts due to hotspot accesses, we design hotspot workloads similar to those in Bamboo. Unlike fixed hotspot accesses, our approach allows hotspot tuple locations to vary within transactions, better reflecting real-world conditions. Figure 8 shows that Rebirth-Retire consistently outperforms Wound-Retire in throughput and achieves a lower abort rate, demonstrating its superior handling of hotspot-induced conflicts.

Figure 9 presents the performance of all protocols on the TPC-C workloads. In the low-contention TPC-C workload, Silo achieves the best performance, completing 100,000 transactions in the shortest time. In contrast, TicToc and MOCC perform worse as they spend more time detecting conflicts and computing timestamps during the validation phase. As contention increases to a medium level, Deadlock-Detection performs the worst due to the overhead of lock waiting and deadlock detection. OCC also struggles, as a significant portion of execution time is spent on transaction aborts. Both Wound-Retire and Rebirth-Retire reduce the time spent by transactions waiting for locks but increase the time spent waiting for commit; their overall performance still surpasses that of other protocols. In high-contention workloads, Deadlock-Detection and Wound-Wait suffer from extremely low throughput due to excessive lock waiting. Long transactions, such as StockLevel and OrderStatus, prolong lock holding, increasing conflicts and causing frequent aborts in Wound-Wait. OCC continues to underperform, as most transactions fail during the validation phase. Rebirth-Retire, however, outperforms Wound-Retire by up to ~3.1× by minimizing unnecessary aborts and reducing concurrency control overhead.

6 RELATED WORK

Dirty Reads and Dirty Writes. Many works have explored concurrency control protocols that allow transactions to access uncommitted data for performance improvement. Hekaton [16, 28] proposed two protocols — pessimistic version and optimistic version for main memory databases based on multiversioning. IC3 [37] and Runtime Pipelining (RP) [32, 40] are two popular variants of transaction chopping techniques. Both methods allow early reads of dirty data but require prior knowledge of the workload and do not support ad-hoc transactions. Bamboo [22] extends Wound-Wait [5] to permit dirty reads and writes without prior workload knowledge. However, it aborts transactions with larger timestamps than the requesting transaction, even in the absence of dependency cycles. In contrast, Rebirth-Retire enables dirty reads and writes without requiring workload knowledge, using dead-dependency detection to prevent cycles and avoid unnecessary aborts.

Dynamic Timestamp Assignment. Given the significant bottleneck introduced by centralized timestamp allocation in multi-core systems, several works have explored alternative approaches for transaction ordering. Silo [34] employs a decentralized validation phase using batched atomic addition timestamps, which avoids bottlenecks from global locks but increases latency under high contention. Dynamic Timestamp Allocation (DTA) [3] dynamically assigns a timestamp range to each transaction at the start, with a final commit timestamp assigned within that range during validation. TicToc [43] calculates the commit timestamp based on the accessed tuples during validation, eliminating the need for a static timestamp. Both DTA and TicToc reduce aborts compared to traditional OCC, but they are limited by the fact that transaction validation and aborts can only occur during the validation phase, which may lead to frequent aborts under high contention. Recent works in distributed database systems [31, 39] further explore dynamic timestamp mechanisms. In contrast, Rebirth-Retire assigns timestamps when conflicts occur and can abort transactions at any point during execution.

Hybrid Concurrency Control. Several existing approaches combine multiple concurrency control mechanisms to enhance system performance across various contention workloads. MOCC [36] dynamically selects between 2PL and OCC based on the hotness of tuples. CormCC [33] formalizes concurrency control into four phases, allowing any CC policy within each phase, provided they follow the same phase order. Polyjuice [35] uses a learning framework to optimize algorithm selection for specific workloads. However, both CormCC and Polyjuice rely on prior workload knowledge, while MOCC requires statistical analysis of data access patterns. Plor [11] and Polaris [41] focus on reducing latency's impact on throughput. Plor adopts an optimistic approach within the Wound-Wait framework but does not eliminate lock acquisition overhead. Polaris integrates a reservation mechanism to enable priority levels in the Silo protocol, specifically targeting tail latency reduction while maintaining throughput. Recent approaches like R-SMF [12], Morty [8], and TSkd [9] explore the scheduling space systematically, proactively identifying and executing optimal schedules to boost throughput. In contrast, Rebirth-Retire adapts to varying contention levels through its passive retire and rebirth mechanisms.

7 CONCLUSION

We proposed the Rebirth-Retire concurrency control protocol, which incorporates a passive *Retire* mechanism that not only reduces the time other transactions spend waiting for locks but also avoids unnecessary *Retire* overhead. Its *Rebirth* mechanism prevents deadlocks and avoids unnecessary transaction aborts. The protocol is provably correct. Experimental results demonstrate that, compared to other protocols, Rebirth-Retire offers significant performance advantages in scenarios with varying levels of contention, except in cases where conflicts between transactions are rare.

ACKNOWLEDGMENTS

This work was supported by NSF of China (61572194,61672233). Additional funding was provided by Transwarp Technology (Shanghai) Co., Ltd.

REFERENCES

[1] 2020. DBx1000. https://github.com/yxymit/DBx1000.

[2] Sam Ainsworth. 2018. *Prefetching for complex memory access patterns.* Technical Report. University of Cambridge, Computer Laboratory.

[3] Vaibhav Arora, Ravi Kumar Suresh Babu, Sujaya Maiyya, Divyakant Agrawal, Amr El Abbadi, Xun Xue, Zhiyanan, and Zhujianfeng. 2018. Dynamic Timestamp Allocation for Reducing Transaction Aborts. In *2018 IEEE 11th International Conference on Cloud Computing*. 269–276. https://doi.org/10.1109/CLOUD.2018.00041

[4] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. 2016. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *Proceedings of the 2016 International Conference on Management of Data*. 583–598. https://doi.org/10.1145/2882903.2915231

[5] Philip A. Bernstein and Nathan Goodman. 1981. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.* 13, 2 (jun 1981), 185–221. https://doi.org/10.1145/356842.356846

[6] P. A. Bernstein and E. Newcomer. 2009. *Principles of Transaction Processing.* Morgan Kaufmann.

[7] Philip A. Bernstein, David W. Shipman, and Wing S. Wong. 1979. Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering* 3 (1979), 203–216. https://doi.org/10.1109/TSE.1979.234182

[8] Matthew Burke, Florian Suri-Payer, Jeffrey Helt, Lorenzo Alvisi, and Natacha Crooks. 2023. Morty: Scaling Concurrency Control with Re-Execution. In *Proceedings of the Eighteenth European Conference on Computer Systems.* 687–702. https://doi.org/10.1145/3552326.3567500

[9] Yang Cao, Wenfei Fan, Weijie Ou, Rui Xie, and Wenyue Zhao. 2023. Transaction Scheduling: From Conflicts to Runtime Conflicts. *Proc. ACM Manag. Data* 1, 1, Article 26 (may 2023), 26 pages. https://doi.org/10.1145/3588706

[10] Chen Chen, Xingbo Wu, Wenshao Zhong, and Jakob Eriksson. 2024. Fast Abort-Freedom for Deterministic Transactions. In *2024 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 692–704. https://doi.org/10.1109/IPDPS57955.2024.00067

[11] Youmin Chen, Xiangyao Yu, Paraschos Koutris, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Jiwu Shu. 2022. Plor: General transactions with predictable, low tail latency. In *Proceedings of the 2022 International Conference on Management of Data*. 19–33. https://doi.org/10.1145/3514221.3517879

[12] Audrey Cheng, Aaron Kabcenell, Jason Chan, Xiao Shi, Peter Bailis, Natacha Crooks, and Ion Stoica. 2024. Towards Optimal Transaction Scheduling. *Proceedings of the VLDB Endowment* 17, 11 (2024), 2694–2707. https://doi.org/10.14778/3681954.3681956

[13] Audrey Cheng, Jack Waudby, Hugo Firth, Natacha Crooks, and Ion Stoica. 2023. Mammoths Are Slow: The Overlooked Transactions of Graph Data. *Proceedings of the VLDB Endowment* 17, 4 (2023), 904–911. https://doi.org/10.14778/3636218.363624

[14] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA). Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[15] Transaction Processing Council. 2007. *TPC-C Benchmark (Revision 5.9.0)*. Technical Report. Transaction Processing Council. Available at: https://www.tpc.org/tpcc/.

[16] 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 *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA). Association for Computing Machinery, 1243–1254. https://doi.org/10.1145/2463676.2463710

[17] Zhiyuan Dong, Zhaoguo Wang, Xiaodong Zhang, Xian Xu, Changgeng Zhao, Haibo Chen, Aurojit Panda, and Jinyang Li. 2023. Fine-Grained Re-Execution for Efficient Batched Commit of Distributed Transactions. *Proceedings of the VLDB Endowment* 16, 8 (2023), 1930–1943. https://doi.org/10.14778/3594512.3594523

[18] 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. *Commun. ACM* 19, 11 (nov 1976), 624–633. https://doi.org/10.1145/360363.360369

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

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

[21] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. 1994. Quickly Generating Billion-Record Synthetic Databases. *SIGMOD Rec.* 23, 2 (may 1994), 243–252. https://doi.org/10.1145/191843.191886

[22] 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* (Virtual Event, China). Association for Computing Machinery, New York, NY, USA, 658–670. https://doi.org/10.1145/3448016.3457294

[23] Kaisong Huang, Tianzheng Wang, Qingqing Zhou, and Qingzhong Meng. 2023. The Art of Latency Hiding in Modern Database Engines. *Proceedings of the VLDB Endowment* 17, 3 (2023), 577–590. https://doi.org/10.14778/3632093.3632117

[24] Evan PC Jones, Daniel J Abadi, and Samuel Madden. 2010. Low overhead concurrency control for partitioned main memory databases. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 603–614. https://doi.org/10.1145/1807167.1807233

[25] Arthur B Kahn. 1962. Topological sorting of large networks. *Commun. ACM* 5, 11 (1962), 558–562. https://doi.org/10.1145/368996.369025

[26] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. Ermia: Fast memory-optimized database system for heterogeneous workloads. In *Proceedings of the 2016 International Conference on Management of Data*. 1675–1687. https://doi.org/10.1145/2882903.2882905

[27] H. T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.* 6, 2 (jun 1981), 213–226. https://doi.org/10.1145/319566.319567

[28] Per Ake Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M Patel, and Mike Zwilling. 2011. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *Proc. VLDB Endow.* 5, 4 (dec 2011), 298–309. https://doi.org/10.14778/2095686.2095689

[29] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-memory data management beyond main memory. In *2018 IEEE 34th International Conference on Data Engineering*. 185–196. https://doi.org/10.1109/ICDE.2018.00026

[30] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*. 1–8. https://doi.org/10.1145/2933349.2933352

[31] Yishuai Li, Yunfeng Zhu, Chao Shi, Guanhua Zhang, Jianzhong Wang, and Xiaolu Zhang. 2024. Timestamp as a Service, not an Oracle. *Proceedings of the VLDB Endowment* 17, 5 (2024), 994–1006. https://doi.org/10.14778/3641204.3641210

[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* (Dresden, Germany). Association for Computing Machinery, New York, NY, USA, Article 40, 16 pages. https://doi.org/10.1145/3302424.3303966

[33] Dixin Tang and Aaron J Elmore. 2018. Toward coordination-free and reconfigurable mixed concurrency control. In *2018 USENIX Annual Technical Conference*. 809–822.

[34] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-Memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania). Association for Computing Machinery, New York, NY, USA, 18–32. https://doi.org/10.1145/2517349.2522713

[35] Jiachen Wang, Ding Ding, Huan Wang, Conrad Christensen, Zhaoguo Wang, Haibo Chen, and Jinyang Li. 2021. Polyjuice:High-Performance transactions via learned concurrency control. In *15th USENIX Symposium on Operating Systems Design and Implementation*. 198–216.

[36] 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. https://doi.org/10.14778/3015274.3015276

[37] 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* (San Francisco, California, USA). Association for Computing Machinery, New York, NY, USA, 1643–1658. https://doi.org/10.1145/2882903.2882934

[38] Jack Waudby. 2024. *High Performance Concurrency Control and Commit Protocols in OLTP Databases*. Ph.D. Dissertation. Newcastle University.

[39] Xingda Wei, Rong Chen, Haibo Chen, Zhaoguo Wang, Zhenhan Gong, and Binyu Zang. 2021. Unifying Timestamp with Transaction Ordering for MVCC with Decentralized Scalar Timestamp. In *18th USENIX Symposium on Networked Systems Design and Implementation*. 357–372.

[40] 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* (Monterey, California). Association for Computing Machinery, New York, NY, USA, 279–294. https://doi.org/10.1145/2815400.2815430

[41] Chenhao Ye, Wuh Chwen Hwang, Keren Chen, and Xiangyao Yu. 2023. Polaris: Enabling transaction priority in optimistic concurrency control. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–24. https://doi.org/10.1145/3588724

[42] 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. *Proc. VLDB Endow.* 8, 3 (nov 2014), 209–220. https://doi.org/10.14778/2735508.2735511

[43] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA). Association for Computing Machinery, New York, NY, USA, 1629–1642. https://doi.org/10.1145/2882903.2882935

[44] Yuan Yuan, Kaibo Wang, Rubao Lee, Xiaoning Ding, Jing Xing, Spyros Blanas, and Xiaodong Zhang. 2016. Bcc: Reducing false aborts in optimistic concurrency control with low cost for in-memory databases. *Proceedings of the VLDB Endowment* 9, 6 (2016), 504–515. https://doi.org/10.14778/2904121.2904126