Polaris: Enabling Transaction Priority in Optimistic Concurrency Control

CHENHAO YE, University of Wisconsin-Madison, USA
WUH-CHWEN HWANG, University of Wisconsin-Madison, USA
KEREN CHEN, University of Wisconsin-Madison, USA
XIANGYAO YU, University of Wisconsin-Madison, USA

Transaction priority is a critical feature for real-world database systems. Under high contention, certain classes of transactions should be given a higher chance to commit than others. Such a prioritization mechanism is commonly implemented in locking-based concurrency control protocols as some lock scheduling mechanisms, but it is rarely supported in the world of optimistic concurrency control.

We present Polaris, an optimistic concurrency control protocol that supports multiple priority levels. To enforce priority, Polaris introduces a minimal amount of pessimism through a lightweight *reservation* mechanism. The protocol is fully optimistic among transactions within the same priority level and preserves the high throughput advantage of optimistic protocols. Our evaluation with YCSB workload shows that Polaris can make the p999 tail latency of high-priority transactions 13× lower than that of low-priority ones. With an abort-aware priority assignment policy, Polaris can deliver 1.9× higher throughput and 17× lower tail latency compared to Silo for high-contention workloads.

CCS Concepts: • **Information systems** → **Database transaction processing**.

Additional Key Words and Phrases: transaction priority, optimistic concurrency control, tail latency, OLTP

ACM Reference Format:
Chenhao Ye, Wuh-Chwen Hwang, Keren Chen, and Xiangyao Yu. 2023. Polaris: Enabling Transaction Priority in Optimistic Concurrency Control. *Proc. ACM Manag. Data* 1, 1, Article 44 (May 2023), 24 pages. https://doi.org/10.1145/3588724

1 INTRODUCTION

It is critical to support multiple priority levels among transactions in an online transaction processing (OLTP) database management system (DBMS). A DBMS uses this mechanism to prioritize certain classes of transactions over others. For example, user-initiated transactions should be prioritized over background system tasks; different priorities can be assigned to user requests based on importance and urgency. Priority is also useful for reducing the tail latency of transactions. In particular, transactions that have been aborted several times should be prioritized in subsequent executions to avoid starvation. As an important mechanism, priority has been supported in multiple practical DBMSs including Microsoft SQL Server [2], CockroachDB [4, 25], Oracle Berkeley DB [23], among others.

Authors' addresses: Chenhao Ye, University of Wisconsin-Madison, 1210 W. Dayton Street, Madison, Wisconsin, USA, 53706-1613, chenhaoy@cs.wisc.edu; Wuh-Chwen Hwang, University of Wisconsin-Madison, 1210 W. Dayton Street, Madison, Wisconsin, USA, 53706-1613, wuh-chwen@cs.wisc.edu; Keren Chen, University of Wisconsin-Madison, 1210 W. Dayton Street, Madison, Wisconsin, USA, 53706-1613, kchen359@wisc.edu; Xiangyao Yu, University of Wisconsin-Madison, 1210 W. Dayton Street, Madison, Wisconsin, USA, 53706-1613, yxy@cs.wisc.edu.

2836-6573/2023/5-ART44 $15.00
https://doi.org/10.1145/3588724

Recent research on transactions has made tremendous progress in optimizing optimistic concurrency control (OCC) protocols and demonstrated substantially higher throughput over conventional designs [17, 20, 27, 29, 30, 34]. However, existing OCC protocols lack general support for setting priority levels, which makes them difficult to be applied to many practical application scenarios. Furthermore, the lack of a priority mechanism can lead to high tail latency due to repeated aborts of transactions [3] in a high-contention workload (e.g., a long transaction being constantly aborted by short writing transactions). This is undesirable in practical systems, where the tail latency is usually as important as throughput.

Enforcing priority is inherently challenging in OCC where conflict detection happens only in the validation phase after execution. Certain pessimism (e.g., locking or preemption) is required to support priority. We make the key observation that only a minimal level of pessimism is required in OCC to enforce priority while still maintaining the performance advantage of optimism and avoiding the overhead in typical pessimistic protocols like lock thrashing [33].

In this paper, we develop **Polaris**, a new concurrency control protocol that extends the widely used Silo [27] protocol to support priority levels. Polaris introduces a lightweight mechanism called *reservation* to enforce priority. Among transactions within the same priority level, the protocol is fully optimistic following Silo. When a high-priority transaction *reserves* a record, a low-priority transaction cannot write to the record (like in a pessimistic protocol) but can still read the record (like in an optimistic protocol). A reserved record can be preempted by another transaction with an even higher priority.

We evaluate Polaris using an open-source DBMS testbed, DBx1000 [32]. We tested two use cases of the priority mechanism. In the case when transactions are statically assigned different priority levels, we observe that high-priority transactions' p999 tail latency is 13× lower than low-priority transactions'. In the second use case, we increment the priority level for transactions that experience repeated aborts, thereby reducing the chance of further aborts. This can reduce the tail latency of YCSB-A benchmark [5] ($\theta = 0.99$) by 2× compared to Silo with only a 1.8% throughput degradation. On the workload with even higher contention (YCSB-A, $\theta = 1.5$), Polaris delivers 1.9× higher throughput and 17× lower p999 tail latency compared to Silo.

Overall, the paper makes the following contributions:

- We present Polaris, an OCC protocol with transaction priority support.
- We formally prove the correctness of Polaris in terms of serializability and liveness.
- We evaluate Polaris against workloads with a variety of contention levels. We show that Polaris successfully enforces priority levels and can reduce tail latency with minimal impact on throughput.

The rest of the paper is organized as follows. Section 2 presents the motivation of transaction priority and related background. Section 3 describes Polaris algorithm in detail. Section 4 formally proves serializability and liveness of Polaris. Section 5 evaluates Polaris against other concurrency control protocols over various workloads. Section 6 covers the related work. Section 7 concludes the paper.

2 BACKGROUND AND MOTIVATION

Database system users may need to prioritize some critical transactions over others so that the critical ones will take precedence when conflicts occur. For example, a transaction that suffers from excessive aborts should be given a higher priority so that the user would not observe high latency.

Concurrency control protocols play an important role in realizing transaction prioritization. Two-phase locking (2PL) and optimistic concurrency control (OCC) are two popular classes of

concurrent control protocols. We describe these two classes of protocols, identify their support for transaction priority, and illustrate our motivation.

Two-Phase Locking. Locking is a widely used concurrency control mechanism that prevents conflicting transactions from accessing the same data. Two-phase locking (2PL) is one catalog of locking-based approaches that provides serializability. The transaction execution is divided into a growing phase and a shrinking phase, where a transaction can only acquire new locks in the growing phase and release locks in the shrinking phase. 2PL schemes can be further classified based on how they deal with deadlock.

For a deadlock-detection scheme, when a deadlock is found, a victim transaction is chosen and aborted to break the deadlock. If users have specified transactions' priority, the lowest-priority one will be chosen as the victim.

No-Wait, Wait-Die, and Wound-Wait are three common deadlock-prevention schemes. Under the No-Wait variant of 2PL, when a transaction encounters a conflict (i.e., lock acquisition denied), it aborts itself without any waiting. Under the Wait-Die scheme, every transaction receives a unique timestamp before execution, and a smaller timestamp indicates higher priority. If a transaction requests a lock but gets denied, it puts itself in the lock wait queue if it has higher priority than the lock holder. Otherwise, it must abort to avoid deadlock. This is a non-preemptive protocol. Lastly, under the Wound-Wait scheme, every transaction also has a timestamp-based priority. In the case of a conflict, if the requesting transaction has higher priority, it preempts the lock and aborts the locker holder. Otherwise, it puts itself in the lock's wait queue.

In practice, many lock-based DBMSs expose interfaces for users to specify transaction priority levels, including Oracle Berkeley DB [23], Microsoft SQL Server [2], and CockroachDB [4, 25]. Although Wait-Die and Wound-Wait schemes may not allow users to set the priority level, they internally use timestamp values to prioritize transactions upon conflicts.

Optimistic Concurrency Control. Another class of concurrency control protocols that ensure serializability is optimistic concurrency control (OCC) [18]. With the assumption that contention is rare, OCC postpones conflict detection to the end of transaction execution. More specifically, OCC tracks the records it reads and keeps the updates to local copies during the execution. After the execution, it enters a global critical section, in which it verifies whether the execution is serializable. If yes, it commits the transaction and publishes all the updates. Otherwise, it gets aborted. OCC has the advantage that it avoids the locking overhead and does not require complicated deadlock prevention or detection mechanism. However, its global critical section hurts the overall scalability.

Silo [27] is a highly optimized modern OCC protocol that eliminates the global critical section. In Silo, every record maintains a 64-bit transaction ID (TID), which encodes its data version number and a latch bit. During the execution, a transaction keeps all updates to its local copies and maintains a read-set and a write-set to track the records it accesses. In the validation phase, a transaction first acquires the latches of all records in its write-set and then validates that the data versions of all records in its read-set remain unchanged. If the validation passes, the transaction proceeds to commit. It chooses a new data version that is larger than those of all records in the write-set. Then it publishes the updates previously kept in its local buffers and applies the new data version with the latch released.

Transaction priority support in OCC-based databases is rare. FoundationDB [36], an OCC-based distributed transactional NoSQL database, provides interfaces to set transactions' priority as "batch" (low) or "immediate" (high) [24]. However, this configuration only affects the transactions' behavior under admission control (throttled or bypassing queuing control), but not on the concurrency control level. We consider this mechanism orthogonal to the problem we are investigating.

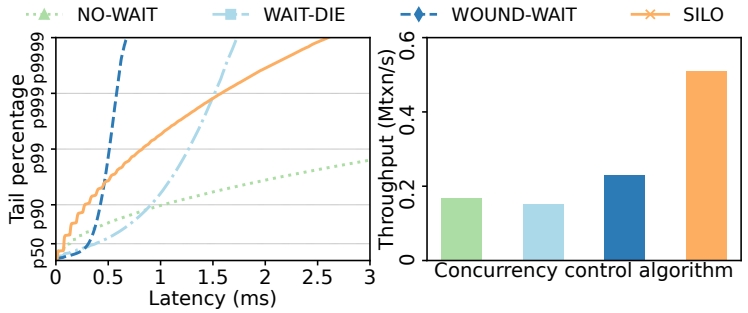

Fig. 1. Latency distribution and overall throughput of four concurrency control algorithms (YCSB-A, r = 50%, w = 50%, θ = 0.99, 64 threads).

Starvation Freedom and Tail-Latency. Some workloads are latency-sensitive. For example, large-scale online services fan out requests to worker servers and collect responses. Servers must respond within a certain time limit to keep services responsive [7]. To prevent indefinitely-growing latency, concurrency control protocols should ensure the execution of transactions is starvation-free.

Wound-Wait and Wait-Die are starvation-free. Wound-Wait scheme keeps an invariant that when a lock conflict occurs, the oldest transaction always wins the lock. The current lock holder will be aborted if an older transaction requests the lock. In the worst case, a repeatedly aborted transaction will become the oldest transaction in the system and eventually acquire all the locks it wants. Under Wait-Die scheme, only transactions older than the lock holder can wait for the lock, which guarantees the lock queue won't grow indefinitely as more new transactions arrive.

In contrast, OCC does not provide starvation freedom. OCC postpones conflict detection to the commit phase, so there is no guarantee that the transaction can pass validation and commit. If a record in the read-set keeps being modified by other transactions, the current transaction may be aborted indefinitely. Figure 1 shows the tail-latency and throughput results of Silo and three 2PL variants under high contention environment. While Silo achieves higher throughput, it bears higher p999 latency than Wound-Wait and Wait-Die. The gap grows even larger when comparing p9999 latency.

Our goal is to get the benefit of both worlds: building priority support on top of OCC, so that some critical transactions (e.g., latency-sensitive) could benefit from prioritization while the system still has high overall throughput.

3 POLARIS ALGORITHM

A key reason that OCC protocols do not have built-in priority is that they delay conflict detection to the commit phase [18, 27]. Although such lazy validation leads to higher throughput, it also poses a challenge in protecting a high-priority transaction from being aborted by a low-priority transaction that commits first.

Polaris resolves this problem by introducing a minimal amount of pessimism back into an OCC protocol, such that priority can be enforced but throughput does not severely degrade. Polaris introduces a notion of *reservation*: a high-priority transaction could reserve a record so that subsequent low-priority transactions will not be able to write to this record; a high-priority transaction could invalidate an earlier low-priority transaction's reservation as a preemption. With reservations, a high-priority transaction is guaranteed to be neither stopped by an earlier low-priority transaction nor aborted by a later one.

The reservation mechanism is very lightweight compared to locking used in pessimistic protocols. First, reads are never blocked by reservation, because a low-priority transaction's reading won't lead to a high-priority transaction's abort. Second, transactions within the same priority level are still operating in an optimistic manner, which can largely preserve the high throughput advantage of OCC protocols.

In the rest of this section, we discuss how to incorporate the idea into an optimistic concurrency control protocol. Our protocol is primarily based on Silo [27], but we believe the idea can be generalized to other OCC protocols as well. We will first show the per-record data structure used in Polaris and then walk through the three phases of transaction processing.

3.1 Data Structure: Transaction ID

Silo [27] introduces a per-record Transaction ID (TID), which contains a *data version* and a *latch bit*. We extend TID to include three extra fields: *priority*, *priority version*, and *reference counter*. To simplify the discussion, we assume all these fields could fit into a 64-bit word for now, so that it could be updated atomically using a single CPU instruction. We defer more extensive discussion on how many bits each field should take to Section 3.6.

The *data version* and *latch bit* work as the ones in Silo [27]. The data version indicates whether a record's data has been updated. The latch bit guards the record data for a short window during the commit phase. If a transaction reads the TID twice and finds the data versions are the same and latch bits are both unset, it knows for sure that the record data is in a consistent state and has no change between two TID reads, even if other fields of the TID have changed.

A reservation on a record is identified by two TID fields: *priority* and *priority version*. In Polaris, a record can be reserved by multiple transactions with the same priority but not cross-priority. We call these transactions "reservees." The *priority* field in the TID indicates the priority of reservees. Reservations on the record can be removed by resetting the *priority* field to zero[1] and incrementing the *priority version*. The field *reference counter* indicates the number of such reservees accessing this record. It is used to decide when a reservation should be removed.

3.2 Execution Phase

In this section, we first walk through the high-level access protocol (Algorithm 1) and then zoom into the reservation operation (Algorithm 2). The access protocol is similar to that in Silo [27], so we highlight the additional code path introduced by Polaris in Algorithm 1.

Access Protocol. Each transaction gets assigned a priority before the execution starts. During the execution, the transaction remembers every record it reads (read-set) and writes (write-set). These two sets will be used in the later commit phase for validation. For simplicity, we assume all writes are read-modify-writes, so records in the write-set are also in the read-set. For every record the transaction reads, it makes a local copy; all writes are applied to local copies.

When the transaction reads a record, it spins if the record's TID has the latch bit set (Algorithm 1, lines 2–4). A set latch bit means another commit-phase transaction may be updating the record so that the record data can be in an inconsistent state. If the record is not locked, the transaction attempts to reserve this record (line 5) using its own priority. The details of the `try_reserve` function will be explained later. Note that this function is the key addition in Polaris compared to Silo when a transaction accesses a record — Silo would proceed with no consideration of priority.

The transaction then proceeds to make a local copy of the record (line 6). To ensure it reads a consistent record value and TID, it uses an atomic compare-and-swap to verify the record TID is still the same as the one it reads earlier (line 7). If this is true, it applies the new TID that contains

[1]In this paper, we use a larger priority number to represent a higher priority level; priority zero is the lowest level.

Algorithm 1: Record Access Protocol

```
   Data: transaction priority tx.prio, record r, read-set R, write-set W, access type is_write
1  do
2      do
3          tid = r.tid // atomic load
4      while tid.latch == LOCKED
5      new_tid, is_reserved = try_reserve(tid, tx.prio, is_write)
6      r_local_copy = r.copy()
7  while !compare_and_swap(r.tid, tid, new_tid)
8  R.add(r, is_reserved, new_tid)
9  if is_write then
10     W.add(r)
11 return r_local_copy
```

Algorithm 2: Reservation Protocol

```
   Data: transaction priority tx.prio, a copy of record TID tid, access type is_write
1  function try_reserve(tid, tx.prio, is_write):
2      new_tid = tid
3      if tid.prio == tx.prio then
           /* reserve with same-priority transactions                    */
4          new_tid.ref_cnt++
5          is_reserved = true
6      else if tid.prio < tx.prio then
           /* preempt from low-priority transactions                     */
7          new_tid.prio = tx.prio
8          new_tid.ref_cnt = 1
9          is_reserved = true
10     else
           /* reserved by high-priority transactions                     */
11         if is_write then
12             ABORT()
13         is_reserved = false
14     return new_tid, is_reserved
```

the reservation. Otherwise, it needs to redo the access protocol. The compare-and-swap instruction uses an acquire-release fence. On the x86 platform, this memory fence is only a compiler fence to prevent instruction reordering optimization.

For every record it reads, the transaction adds the record's TID to its read-set. It also remembers whether it has placed a reservation on the record, which will be used in the later reservation cleanup protocol (line 8).

Reservation Protocol. The goal of the reservation protocol is to ensure that a low-priority transaction does not abort a high-priority transaction. We use the following simple case to illustrate the idea. Consider a high-priority transaction A and a low-priority transaction B both access a record: 1) If A has read the record but has not committed, B should not write to the record, as that

Algorithm 3: Commit Protocol

```
   Data: transaction priority tx.prio, read-set R, write-set W
1  for r in sorted(W) do
2      do
3          tid = r.tid // atomic load
4          if tid.prio > tx.prio or tid.latch == LOCKED then
5              ABORT()
6          locked_tid = tid
7          locked_tid.latch = LOCKED
8      while !compare_and_swap(r.tid, tid, locked_tid)
9  for r, is_reserved, tid in R do
10     curr_tid = r.tid // atomic load
11     if curr_tid.latch == LOCKED and r not in W then
12         ABORT() // locked by another transaction
13     if curr_tid.data_ver != tid.data_ver then
14         ABORT() // data has been updated
   /* validation pass; transaction can commit                              */
15 new_data_ver = W.max_data_ver() + 1
16 for r in W do
17     r.install_write()
18     tid = r.tid // atomic load
19     new_tid = cleanup_write(tid, new_data_ver)
20     r.tid = new_tid // atomic store
21 for r, is_reserved, old_tid in R do
22     if r not in W and is_reserved then
23         do
24             tid = r.tid // atomic load
25             new_tid = cleanup_read(tid, tx.prio, old_tid.prio_ver)
26             if new_tid == tid then
27                 break // no cleanup needed
28         while !compare_and_swap(r.tid, tid, new_tid)
```

would cause A to abort. 2) If B has read the record but has not committed, A must still be able to ignore B and proceed to read/write/commit. Based on this intuition, we describe the reservation protocol in Algorithm 2.

If the record's priority is the same as the requesting transaction's, then earlier same-priority transactions (or no transaction) have already reserved this record. The current transaction can become a reservee by simply incrementing the reference counter (Algorithm 2, lines 3–5).

If the record has a lower priority, this means some low-priority transactions (or no transaction) have reserved this record, and the current transaction should preempt the record from them by setting the record's priority to its own priority and the reference counter to one (lines 6–9).

If the record has a higher priority, this means some higher-priority transactions have reserved this record, so the current transaction cannot write to the record. If the current transaction only requires reading the record, it could still proceed without reservation, hoping to commit before those high-priority transactions making any updates; if it requires writing, it must abort (lines 10–13).

3.3 Commit Phase

When the execution finishes, the transaction enters the commit phase. In this phase, it validates that 1) its execution is serializable with respect to other transactions and 2) its commit will not cause a higher-priority transaction to abort. The pseudo-code of the commit protocol is shown in Algorithm 3. We highlight the code path where Polaris differs from Silo [27].

For every record in the write-set, the transaction tries to acquire the latch (lines 1–8). It checks its priority is equal to or higher than that of the record. If not, it must abort since some high-priority transactions have reserved the record. It also aborts if the latch bit is already set. Note the latch acquisition must be done in sorted order to avoid deadlock.

For every record in the read-set, the transaction validates the record is not being locked and its data version is the same as the one it saw in the execution phase; otherwise, it must abort (lines 9–14). If the record is being locked, it means another transaction that wants to write to this record is in the commit phase, and the data may have been updated. Note that the transaction only checks the data version (with the latch bit) but not the priority-related fields, since the data version is the ground truth of whether the record data has been updated. The priority-related fields are only used for permission checking.

If all validations pass, the transaction designates a data version number that is larger than the data version numbers of all records in its write-set (line 15). For each record in the write-set, it publishes the write it previously made to its local copy (line 17) and then performs reservation cleanup, which returns a new TID for the record (lines 18–20). We defer the detailed discussion of reservation cleanup to Section 3.4. Similarly, it performs the corresponding cleanup for every read-only record (lines 21–28). Note for every record in the write-set, applying new TID only needs an atomic store instruction, instead of compare-and-swap, because the record has been locked by setting the latch bit. With the latch bit set, no other transaction can update the TID, regardless of priority (as shown in Algorithm 1, lines 2–4).

3.4 Reservation Cleanup

After a transaction commits or aborts, it must perform a cleanup to release the reservations made during the execution, so that lower-priority transactions could proceed. The pseudo-code of the reservation cleanup protocol is shown in Algorithm 4, which returns a new TID for each record.

For a read-only record, the transaction verifies whether its reservation is still in the TID. It can happen that its reservation has already been invalidated, e.g., preempted by a higher-priority transaction, so it does not need to release the reservation itself (Algorithm 4, lines 2–5). If the reservation is still in the TID, the transaction decrements the reference counter; if the counter reaches zero, it resets the TID to the lowest priority and increments the priority version by one (lines 6–10). This cleanup is applicable for both a commit and an abort.

Every record in the write-set will receive a new data version in the case of a commit. The transaction must remove all the reservations on the record because the data has been changed, and reservations are no longer meaningful (lines 13–17). If the transaction aborts before acquiring the latch, the cleanup operation is the same as `cleanup_read`. If the latch has been acquired, the cleanup protocol is the same as `cleanup_write`, except the parameter `new_data_ver` is still the current data version of this record. Note in this case, even though the data is unchanged, the reservations must still be removed, because other reservees who have seen the record locked will assume their reservations would be removed and do not perform cleanup themselves (lines 2–3).

Algorithm 4: Reservation Cleanup Protocol

```
Data: a copy of record TID tid, transaction priority tx.prio, previously seen priority version
      of the record prio_ver, new data version for commit new_data_ver
1  function cleanup_read(tid, tx.prio, prio_ver):
2      if tid.latch == LOCKED then
3          return tid // no cleanup needed
4      if tid.prio != tx.prio or tid.prio_ver != prio_ver then
5          return tid // no cleanup needed
6      new_tid = tid
7      new_tid.ref_cnt--
8      if new_tid.ref_cnt == 0 then
9          new_tid.prio = 0
10         new_tid.prio_ver++
11     return new_tid
12 function cleanup_write(tid, new_data_ver):
13     new_tid.data_ver = new_data_ver
14     new_tid.latch = UNLOCKED
15     new_tid.prio = 0
16     new_tid.prio_ver = tid.prio_ver + 1
17     new_tid.ref_cnt = 0
18     return new_tid
```

3.5 Optimization

The idea of the reservation is to protect high-priority transactions from being aborted by low-priority transactions. However, for transactions in the lowest priority level, such a reservation is not necessary at all. Thus, in the reservation protocol (Algorithm 2), if both the record's priority and the transaction's priority are zero, the protocol could simply not increment the reference counter and return false for `is_reserved`.

This optimization is extremely helpful because it avoids a shared-memory write and saves the cost of cache invalidation on other cores. In the case that every transaction is in the lowest priority level, Polaris will fall back to Silo with almost no performance overhead.

3.6 Discussion

Fields Size in TID. The algorithm described above assumes all five fields of TID can fit into a single 64-bit word. We now discuss how many bits these fields should take. In our implementation, we use 10 bits for *reference counter*, 4 bits for *priority*, 4 bits for *priority version*, 1 bit for *latch*, and the rest 45 bits for *data version*.

The number of bits that the *reference counter* should take depends on the maximum worker concurrency in the system. For a machine with fewer than a thousand cores, 10 bits are enough. Alternatively, a transaction could simply skip the reservation if the reference counter has already reached the maximum. The 4-bit *priority* field can support 16 levels of priority, which is sufficient for most use cases. The *priority version* does not need too many bits, either. The priority version is designed for a transaction to identify whether its reservation is still in the TID, so two distinct reservations only need priority versions to be different but do not have to be monotonic increasing. Furthermore, if a wrap-around happens, it may cause a transaction A to falsely believe its reservation

is still in the TID and decrement the reference counter. This could lead to an early removal of the reservation while another same-priority transaction B is still using the record. However, this does not affect serializability at all (note data version is the only guard for serializability) and the consequence is merely a priority inversion (i.e. a transaction C with lower priority than A and B may write to the record and cause B to abort). Since it is rare and does not affect serializability, we do not worry too much about the wrap-around issue. In our implementation, we additionally check that the reference counter is larger than zero before decrementing it (Algorithm 4, line 7) to ensure it won't underflow.

Priority Assignment Policy. Polaris focuses on the mechanism to enforce the given transaction priorities. It is yet interesting to determine how these priorities should be assigned. In a real-world system, there can be two sources of priorities: either users' input or some priority assignment policy within the database.

The DB-assigned priority is similar to the priority used in Wound-Wait and Wait-Die variants of 2PL protocols, where an older transaction can have a higher priority and thus, more chance to commit. Such a priority assignment policy can provide starvation freedom and lower tail latency.

We observed that too many high-priority transactions can hurt the overall throughput because a high-priority transaction's reservation of records would prevent low-priority transactions that attempt to write to those records, which leads to more aborts of low-priority transactions. Moreover, high-priority transactions cannot benefit from the lowest-priority optimization described in Section 3.5. We present a quantitative measurement of such effects in Section 5.2.

Empirically, we find the following policy works well: every transaction starts with its initial priority p_0 and remains in p_0 until it has been aborted t times; after the abort counter reaches the threshold t, the transaction increments priority by 1 for every s aborts. Formally, a transaction's priority is

$$p = \begin{cases} p_0, & \text{if } abort\_cnt < t \\ p_0 + \lfloor (abort\_cnt - t)/s \rfloor, & \text{otherwise} \end{cases}$$

User-specified priority can be combined with DB-assigned priority. For example, the user could specify one high-priority transaction A and two low-priority transactions B and C. A should always be prioritized over B and C; between B and C, they can follow the database priority assignment policy. We will show an example case in Section 5.3.

Logging and Durability. Polaris focuses on transaction prioritization in a concurrency control protocol, which is mostly orthogonal to durability. The system can be made durable by applying the durability constructs described in Silo [27]. Silo encodes an epoch number and a sequence number in its data version number. This can also be applied to Polaris. More sophisticated logging schemes are possible but beyond the scope of this paper.

Read/Write Intention in Reservation. The current design of reservation only encodes priority-related information but does not specify whether the reservation is intended for reading or writing. It is possible to use one bit in TID to encode such intention, which enables more fine-grained control of conflict handling policy, e.g., early detection of a write-write conflict at reservation time. This is not a foundational design choice for Polaris, and we do not include it in the main algorithm for simplicity.

3.7 Extensions

We then discuss the potential extension of Polaris, including application in distributed systems and support for lower isolation levels.

Application in Distributed Databases. Polaris is applicable to distributed databases, where concurrency happens among multiple machines. TID is per-record, so the machine that stores the record is also responsible for managing its TID. There is no centralized logic that requires additional coordination in a distributed context. Locking the write-set at the commit phase could piggyback over the two-phase commit protocol as in other distributed OCC (e.g., Sundial [35]).

Support for Lower Isolation Levels. The isolation levels that Polaris can support depend on the underlying concurrency control protocol. The current design of Polaris is based on Silo, and it mostly inherits the isolation levels that Silo can support.

In particular, read committed isolation allows read operations to happen at different timestamps as long as the data read is committed. Silo can support read committed isolation by only validating records in the write-set. Such support is similar in Polaris but with additional accommodation for the reservation protocol: the transaction only reserves the record if it intends to write (Algorithm 2) and in the validation phase, it only traverses through the write-set instead of the read-set (Algorithm 3, line 9).

4 FORMAL PROOF

In this section, we prove the correctness of Polaris in terms of serializability and liveness. For simplicity, we consider a re-executed transaction as a new transaction; we assume there is no user-initiated abort. We say a transaction becomes active when it starts execution; it turns inactive when it finishes the commit phase with reservation cleanup. In what follows, we present and prove three theorems.

Theorem 4.1. *Transactions are serializable in Polaris. One equivalent serialization order is the real-time order of the moments when committed transactions acquire all the latches for write-sets.*

Theorem 4.2. *When there is no transaction active, every record's priority is zero.*

Theorem 4.3. *A transaction will not be aborted if it is the only active highest-priority transaction.*

4.1 Serializability

Proof of Theorem 4.1

Proof. An aborted transaction will never publish its writes, so no other transaction will see uncommitted data. Let T_1 and T_2 be two transactions that commit, t_1 be the timestamp of T_1 having all the latches acquired for its write-set, t_2 be that of T_2, and $t_1 < t_2$.

We first show T_2 must see every record T_1 writes. This is trivially true if T_1's write-set is disjointed with T_2's read-set; we then consider cases where there is a record r^* that is in both T_1's write-set and T_2's read-set. Suppose r^* is locked by T_1 at t_1^* and not released until T_1 commits and finishes publishing the updated r^*. Since T_1 has acquired latches of all records in the write-set at t_1, we know $t_1^* \leq t_1 < t_2$. According to Algorithm 3, the validation of the read-set (lines 9–14) is done after the latch acquisition (lines 1–8). When T_2 performs validation on r^*, it must be after t_2 and then also after t_1^*. During the validation, a record being locked by another transaction will lead to an abort (lines 11–12), so T_2's validation on r^* must happen after the latch on r^* being released, at which point T_1 has published the updated r^*. Then T_2 must see T_1's write on r^*, otherwise, the validation will fail due to data version mismatch (lines 13–14).

We then show that T_1 must not see any record T_2 writes. T_1 reads the records (in the execution phase) before t_1 (in the commit phase), so it is also before t_2. T_2 does not publish its write before t_2, so the data T_1 read must not contain anything written by T_2. □

4.2 Liveness

We first formally define a *reservee* as below.

Definition 4.4. A transaction becomes a reservee of a record r when it successfully reserves r. It stops being a reservee when either 1) it finishes the reservation cleanup on r, or 2) the priority or the priority version of r change, whichever comes first.

Note the identity as a reservee is automatically revoked when the record is preempted by a higher-priority transaction, even though the current transaction may not notice immediately. We then introduce Lemmas 4.5 and 4.6 before the proof of the theorems.

Lemma 4.5. *A record r's priority is never higher than the highest priority of transactions active in the system.*

Proof. A record r can only be in one of the three possible states: 1) unlocked and at priority zero; 2) unlocked and at non-zero priority; 3) locked. All records start with state 1, and the lemma holds trivially for state 1. We then show that the lemma still holds during the state transition.

The record transitions from state 1 to state 2 if and only if a transaction T with priority $p > 0$ reserves the record. According to Algorithm 2, lines 3–4 and 6–8 and Algorithm 4, lines 4–7, the following invariant of state 2 always holds: the reference counter in r is equal to the number of reservees of the record r. When the reference counter is not zero, there is at least one reservee; according to Definition 4.4, the record's priority is the same as the reservee. The lemma holds. When the reference counter becomes zero, Algorithm 4, lines 8–9 guarantee this will trigger the transition from state 2 to state 1, and the lemma still holds.

The record transitions from state 2 to state 3 if and only if a transaction T locks the record, according to Algorithm 3, line 7. At this point, the record's priority must be no higher than T's; otherwise, the latch acquisition will be denied by lines 4–5. When T finishes the cleanup phase and becomes inactive, the record transitions from state 3 to state 1, according to Algorithm 4, lines 14–15. The lemma still holds.

Finally, the record transitions from state 1 to state 3 if and only if a zero-priority transaction T locks the record. In this case, the priority of the record will remain zero. The lemma still holds. □

Lemma 4.6. *If a transaction T_1 has higher priority than T_2, then for every record r of which T_1 is a reservee, T_2 cannot write to r.*

Proof. The access protocol ensures the latch was not set when T_1 first read and reserved r (Algorithm 1, lines 2–5). After T_1 has reserved r, T_2 cannot acquire the latch on r, as that will be denied by Algorithm 3, lines 4–5. However, T_2 must acquire r's latch first (Algorithm 3, lines 1–8) before it can commit and apply write to r (line 17), so T_2 cannot write to r. □

Proof of Theorem 4.2

Proof. Consider a moment when there is no active transaction in the system; the goal is to show that all records at this moment have priority zero. Suppose at this moment, we start a trivial zero-priority transaction $T_{trivial}$ that does not read or write any record. According to Lemma 4.5, all records must not have a priority higher than $T_{trivial}$, so they must all have priority zero. □

Proof of Theorem 4.3

Proof. There are four paths that can lead to abort: 1) fail to reserve a record for writing (Algorithm 2, line 12); 2) fail to acquire the latch of a record in the write-set (Algorithm 3, line 5); 3) in commit-phase validation, a record in the read-set is locked by another transaction (Algorithm 3,

line 12); 4) in commit-phase validation, a record's data version number is found changed (Algorithm 3, line 14). Let transaction T_{high} be the only highest-priority transaction. Let T_{low} be a random transaction, which must have lower priority than T_{high}. We then show that none of these four paths can lead to T_{high} being aborted.

Since T_{high} has the highest priority, whenever it wants to access a record, it will always make a reservation successfully: according to Lemma 4.5, it will always reach either line 3's or line 6's branch in Algorithm 2 and never reach the abort path 1 (lines 10–12). We have shown in Lemma 4.6 that T_{low} cannot acquire the latch for every record of which T_{high} is a reservee. Combined with Lemma 4.5, T_{high} will not be aborted through paths 2 and 3. The data version of a record can only be changed between T_{high}'s read in the execution phase and its validation in the commit phase if there is a transaction T_{low} that has committed and written to the record, as shown in Algorithm 3, lines 16–20. Lemma 4.6 shows this cannot happen, so T_{high} will not be aborted through path 4. □

5 EXPERIMENTAL EVALUATION

We present our evaluation of Polaris with two use cases: DB-assigned priority and user-specified priority. In our evaluations, we would like to answer the following questions:

- In the case of DB-assigned priority, how should the database assign priority to reduce tail latency while keeping high throughput?
- With a proper priority assignment policy, can Polaris deliver high throughput and low tail latency under various workloads?
- In the case of user-specified priority, can Polaris distinguish high-priority transactions from low-priority transactions well?

5.1 Experimental Setup

We implemented Polaris on top of DBx1000 OLTP DBMS [10, 11, 32, 33], a multi-threaded, shared-everything in-memory database system that stores data in a row-oriented manner with hash table indexes. DBx1000 has several concurrency control protocols implemented, which enables fair comparison between them on the same codebase. The source code of Polaris is publicly available [31]. For experiments in Sections 5.3 and 5.4, we evaluated three 2PL protocols and two OCC protocols:

(1) **NO-WAIT**: The No-Wait variant of 2PL protocols where a transaction aborts immediately if its lock acquisition is denied due to contention.
(2) **WAIT-DIE**: The Wait-Die variant of 2PL protocols where a transaction only waits for a lock if it is older than the lock holder and aborts itself if younger.
(3) **WOUND-WAIT**: The Wound-Wait variant of 2PL where a transaction preempts the lock if it is older than the lock holder and waits if younger.
(4) **SILO**: A state-of-art OCC protocol [27].
(5) **POLARIS**: Our OCC protocol with priority support.

An alternative approach to support priority is through batching: the system executes a batch of transactions but delays their commits to a batch-commit phase; in the batch-commit phase, the system resolves conflicts by aborting the ones with lower priority. To compare Polaris with such a batching-based approach, we implemented Aria [21] on DBx1000. In Aria, each transaction is pre-assigned a sequence number, whose order represents the serial execution order, and a lower sequence number is analogous to a higher priority. Our implementation includes the reordering optimization described in Aria [21]. The comparison between Polaris and the batching-based priority is covered in Section 5.5.

DBx1000 implements the stored procedure of Yahoo! Cloud Serving Benchmark (YCSB) [5] and TPC-C benchmarks [6]. In YCSB, transactions are randomly generated from a Zipfian distribution

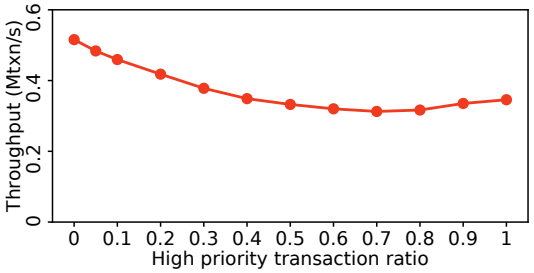

Fig. 2. Polaris throughput with varying ratio of high priority transaction (YCSB-A, $r = 50\%$, $w = 50\%$, $\theta = 0.99$, 64 threads).

with a tunable parameter θ; each transaction accesses 16 records in a 100-million-record table. We use two workloads: YCSB-A consists of 50% read and 50% write transactions, and YCSB-C is read-only. TPC-C is a widely used standard OLTP benchmark, and DBx1000 only implements the *Payment* and *NewOrder* transactions. TPC-C contains 5% of user-initiated transaction aborts and contains nine types of tables with a tunable number of warehouses.

We ran experiments on a CloudLab c6420 machine [9], which has two sixteen-core 2.6 GHz Intel Xeon Gold 6142 processors and 384 GB of ECC DDR4-2666 memory, running on Ubuntu 20.04. With hyperthreading enabled, the machine has 64 logical cores in total.

In our experiments, each database worker thread was pinned to a logical core. We used `numactl --interleave all` to allocate memory between NUMA nodes in a round-robin manner. Whenever a transaction is aborted, it is re-executed after a random back-off time within 1 us. We have tuned the system and found this back-off time yields good overall performance. The exception is Aria, which places aborted transactions into the next batch and does not perform back-off. For Wait-Die, Wound-Wait, and Aria, we do not allocate a new timestamp/sequence number for re-execution, so the re-executed transaction will have higher priority than the newly arrived ones. We collected each transaction's latency as the time elapsed between the start of its first execution and the end of its commit; the time spent in aborted execution is included in the latency.

5.2 Priority Sensitivity Analysis

We quantitatively evaluate the impact of introducing high-priority transactions with YCSB-A workload ($\theta = 0.99, r = 50\%, w = 50\%$) with 64 threads. In this experiment, every transaction is assigned a binary priority (high/low) before its first execution, and this priority does not change during the life cycle of this transaction. We use priority 0 for low-priority transactions so that they can benefit from the lowest-priority optimization. The priority assignment is uniformly random, and we vary the ratio of high-priority transactions. The result is shown in Figure 2.

When there are only low-priority transactions, Polaris behaves exactly like Silo, and its throughput is 516 Ktxn/s. When there are 5% of high-priority transactions, the overall throughput drops by 6% (483 Ktxn/s); when there are 10% high-priority transactions, the throughput drops by 11% (459 Ktxn/s). The overall throughput reaches its lower bound (313 Ktxn/s) when 70% of transactions are high-priority, making the rest of 30% transactions difficult to commit. The throughput goes up when every transaction is high-priority (346 Ktxn/s). However, this is still lower than the case when every transaction is low-priority because high-priority transactions cannot benefit from the lowest-priority optimization — they frequently write to TIDs, triggering cache coherence traffic, which is costly in the case of 64 threads.

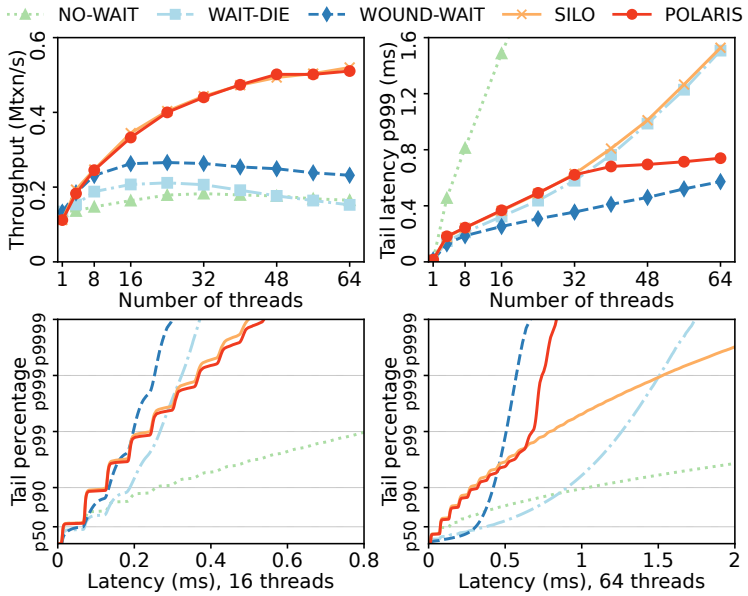

Fig. 3. Throughput and p999 tail latency over a spectrum of thread numbers; latency distribution in the cases of 16 and 64 threads (YCSB-A, $r = 50\%, w = 50\%, \theta = 0.99$).

From the above measurement, we draw a key takeaway: when the database has the flexibility to assign priority, it should leave the majority of transactions at priority zero and only assign non-zero priority to a small number of transactions when necessary. A typical example is the transactions that have been aborted multiple times and could result in high tail latency. Since only a small fraction of transactions has non-zero priority, assigning them high priority will have a limited impact on the throughput, but could significantly reduce the tail latency.

For the rest of the evaluation, we use the priority assignment policy described in Section 3.6 with $p_0 = 0, t = 8, s = 3$ for DB-assigned priority. Namely, a transaction will remain at priority 0 until it has been aborted 11 times; then its priority is incremented by one for every three aborts. Though there is a large space to tune the priority assignment policy, we find this simple policy already works well enough.

5.3 YCSB Results

We first evaluate Polaris with DB-assigned priority against YCSB over a spectrum of thread numbers and access distribution skewness. Then, we consider the case where the user specifies the priority.

Varying Number of Threads (Read/Write-Mix). Figure 3 shows the throughput and p999 tail latency of five concurrency control protocols over a spectrum of thread numbers with YCSB-A workload ($\theta = 0.99$); we further zoom into the latency CDF in the cases of 16 threads and 64 threads.

Up to 32 threads, Polaris behaves similarly to Silo, both in terms of throughput and p999 tail latency. At 32 threads, only 0.12% of transactions in Polaris have been aborted 11 times or more and get prioritized, which explains its similar performance to Silo's.

However, when there are more threads, Silo's p999 tail latency grows almost linear with the thread number, while Polaris will prioritize more transactions at the tail to bound the tail latency. At 64 threads, the throughput of Polaris is only 1.8% lower than Silo's, but its p999 tail latency is half of Silo's; 2.5% of transactions in Polaris commit with priority 1 and 0.01% commit with

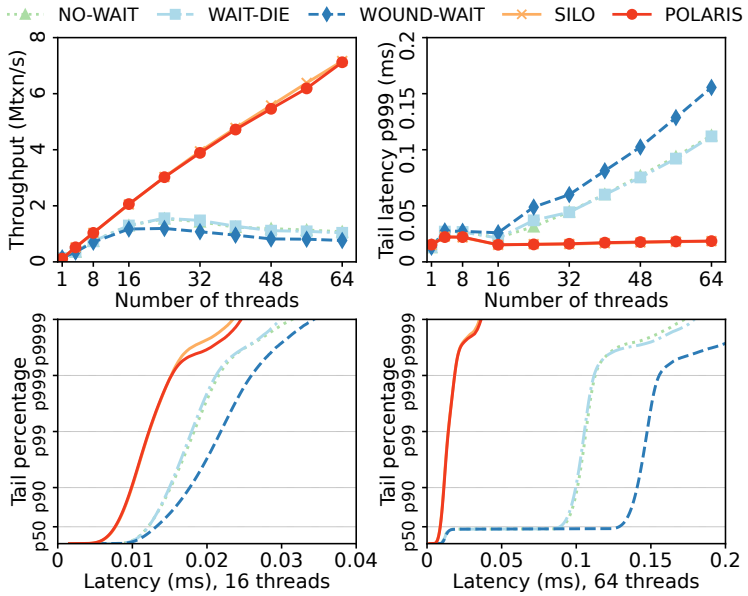

Fig. 4. Throughput and p999 tail latency over a spectrum of thread numbers (YCSB-C, $r = 100\%$, $\theta = 0.99$); latency distribution in the cases of 16 and 64 threads. Note the curves of Silo and Polaris are mostly overlapped; same for No-Wait and Wait-Die.

priority 2; this can also be seen from the latency distribution graph where Silo and Polaris diverge at around p98. Wound-Wait has the lowest tail latency as it strictly follows the priority based on timestamps. Wait-Die also uses timestamp-based priority, but its lock scheduling mechanism is not friendly to the tail latency. Whenever a transaction commits and is deciding whom to grant the lock next, Wait-Die has to maintain the invariant that the waiting transactions have higher priority than the lock holder, so it will grant the lock to the youngest waiting transaction and leave older transactions waiting.

In terms of the median (p50) latency, Silo, Polaris, and No-Wait perform better than Wound-Wait and Wait-Die, because the major contributor to the median latency is waiting instead of aborts.

Varying Number of Threads (Read-Only). Figure 4 demonstrates the performance of five concurrency control protocols with the read-only YCSB-C workload ($\theta = 0.99$). Since there is no abort, all transactions in Polaris remain in the lowest priority. As a result, Polaris and Silo have similar performance, and both scale well. In contrast, three 2PL variants do not scale, as lock acquisition requires a write operation to shared memory, which will trigger expensive cache coherence protocol to invalidate cache on other cores.

Varying Data Access Distribution. We then compare five concurrency control protocols at 64 threads over a spectrum of access distribution skewness with YCSB-A. Figure 5 shows the cases from low contention to high, where a larger Zipfian θ value implies a more skewed distribution. When contention is low, all protocols perform well but OCC variants perform slightly better. When contention is high, we found that Silo is much less resistant to the growing skewness. All three 2PL variants have stable throughput and tail latency when θ grows from 1.3 to 1.5, while Silo suffers from a dramatic throughput drop and a tail latency increase. For Polaris under $\theta = 1.5$, we observe the transaction at p999 tail has been aborted 28 times and commits with priority 6. Around 91%

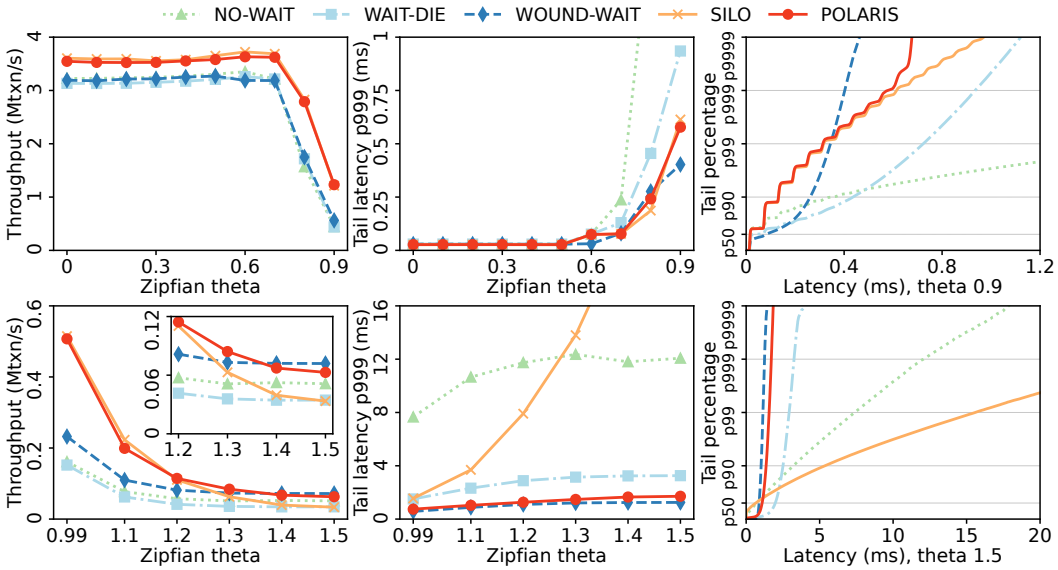

Fig. 5. Throughput and p999 tail latency with varying contention levels; latency distribution when Zipfian θ equals to 0.9 and 1.5 (YCSB-A, $r = 50\%, w = 50\%$, 64 threads).

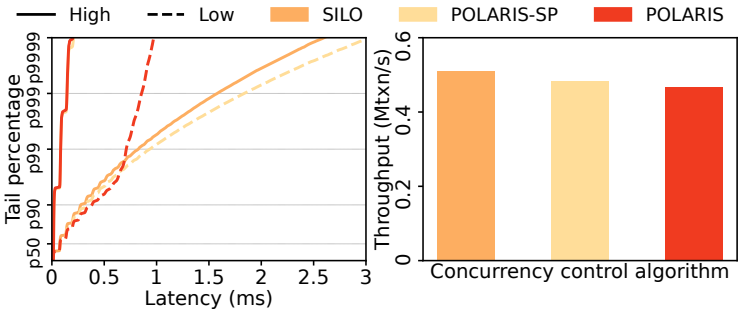

Fig. 6. Latency distribution of high/low-priority transactions and overall throughput (YCSB-A, $r = 50\%, w = 50\%, \theta = 0.99$, 64 threads). Note the high-priority latency CDF curves of Polaris-SP and Polaris are mostly overlapped. Silo does not distinguish transactions based on priority.

of transactions eventually commit with non-zero priority. In such a case with high contention, Polaris behaves more like 2PL, and priority makes it more resistant to skewness. As a result, Polaris delivers 1.9× higher throughput and 17× lower tail latency compared to Silo.

User-Specified Priority. To evaluate the capability of Polaris handling priorities provided by users, we conduct an experiment with the following settings on Silo, Polaris with static priority (Polaris-SP), and Polaris default: 1) We make 5% of the transactions (from uniform random distribution) have high priority and the rest have low. 2) For Polaris with static priority, we assign high-priority transactions with priority 8 and low-priority transactions with priority 0; such a priority does not change during the life cycle of these transactions. 3) For Polaris default, we use the priority assignment policy described in Section 5.2 but $p_0 = 8$ for high priority and $p_0 = 0$ for low; we further

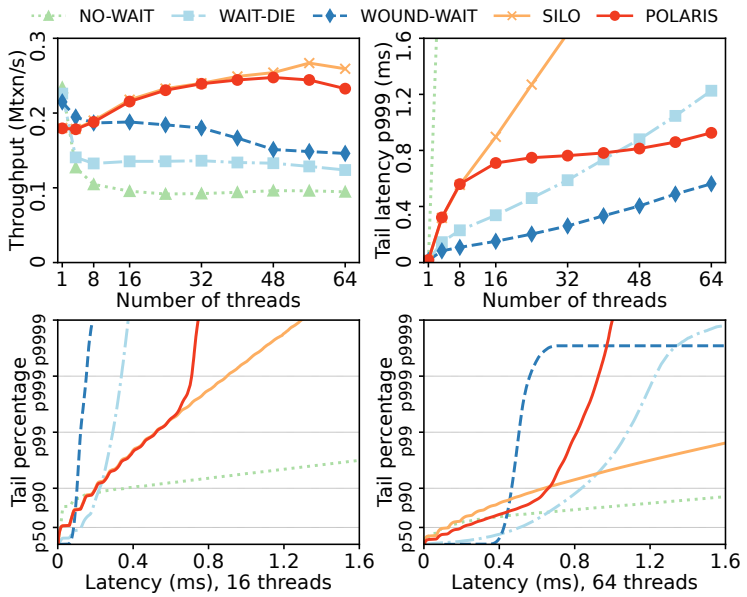

Fig. 7. Throughput and p999 tail latency over a spectrum of thread numbers; latency distribution in the cases of 16 and 64 threads (TPC-C, 1 warehouse).

restrict the low-priority transaction to have the maximum priority at 7, so that the user-specified priority will always be respected.

The latency distribution and overall throughput are shown in Figure 6. Both Polaris-SP and Polaris default can make 99.99% high-priority transactions commit within 3 aborts and thus, have p9999 latency lower than 0.2 ms. The low-priority transactions in Polaris-SP have slightly higher tail latency (3.0 ms) than transactions in Silo (2.6 ms) because they behave like those in Silo but can additionally be aborted by high-priority transactions. With DB-assigned priority enabled, Polaris can even bound the tail latency of low-priority transactions within 1 ms. With a significant improvement in tail latency, Polaris delivers throughput that is comparable to Silo and Polaris-PS (only 8.6% and 5.1% lower, respectively).

5.4 TPC-C Results

We then evaluate the five concurrency control protocols using the TPC-C workload under two different contention levels by using 1 and 64 warehouses.

Varying Number of Threads at High Contention. Figure 7 shows the performance with TPC-C workload with 1 warehouse. Both Silo and Polaris deliver high throughput (1.6×–1.8× compared to Wound-Wait) at 64 threads. Polaris could bound the p999 tail latency within 1.0 ms, while Silo's p999 tail latency reaches 4.9 ms.

The tail latency of Silo and Polaris diverge early at 16 threads. At 16 threads, Polaris has 21% lower p999 tail latency, but its throughput is almost the same as Silo. Both Silo and Polaris have 0.48% of transactions aborts 11 times or more. Polaris starts to prioritize those transactions, so most of them can commit at their next re-execution; only 0.0021% of transactions have been aborted 12 times or more. In contrast, 0.33% of transactions in Silo have been aborted 12 times or more; the transaction at the p999 tail is aborted 15 times and the one at the p9999 tail is aborted 21 times.

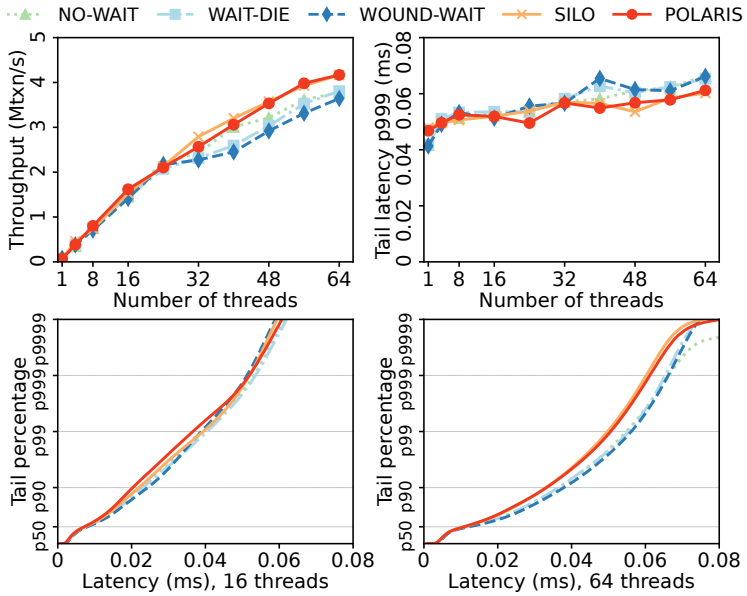

Fig. 8. Throughput and p999 tail latency with over a spectrum of thread numbers; latency distribution in the cases of 16 and 64 threads (TPC-C, 64 warehouses).

The throughput of Silo and Polaris diverge at 56 threads. At 56 threads, 12% of transactions in Polaris have non-zero priority; at 64 threads, 18% of transactions have non-zero priority. The throughput of Polaris is 10% lower than Silo's, but still significantly better than 2PL variants.

Varying Number of Threads at Low Contention. Figure 8 demonstrates the case where contention is low (TPC-C with 64 warehouses). All protocols scale well with the number of threads, but Silo and Polaris still deliver higher throughput and lower tail latency than 2PL variants.

5.5 Comparison with Batching

We now compare the performance of Polaris with Aria, a batching-based deterministic concurrency control protocol that can express the priority of a transaction through its serial execution order in the batch. The results are shown in Figures 9 and 10. Legend "Aria-X" indicates each thread executes X transactions in a batch. We term X as "per-thread batch size." For example, Aria-2 at 64 threads means each thread executes two transactions in a batch, so there are 128 transactions executed in total within a batch. We say the batch size is 128 and the per-thread batch size is 2.

Varying Number of Threads at High Contention. Figure 9 shows the performance of Polaris and Aria with YCSB-A workload ($\theta = 0.99$). Under this workload, Aria does not perform well due to high abort rates: within a batch, a record can only be written by one transaction and read by transactions with serial orders before the writer; every transaction after the writer must be aborted. When contention is high, a large number of transactions would be aborted because they fail to read or write some hot records. Larger batch sizes can exacerbate such a high abort rate issue, which explains the throughput drop when the thread number and the per-thread batch size increase.

A large per-thread batch size hurts median latency because transactions that finish execution cannot commit immediately but have to wait until the batch-commit phase. It also increases the

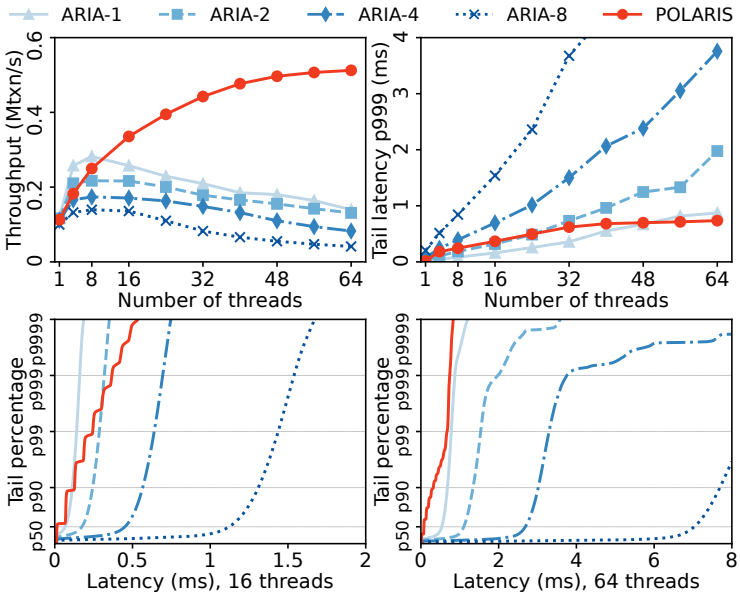

Fig. 9. Throughput and p999 tail latency over a spectrum of thread numbers; latency distribution in the cases of 16 and 64 threads (YCSB-A, $r = 50\%, w = 50\%, \theta = 0.99$).

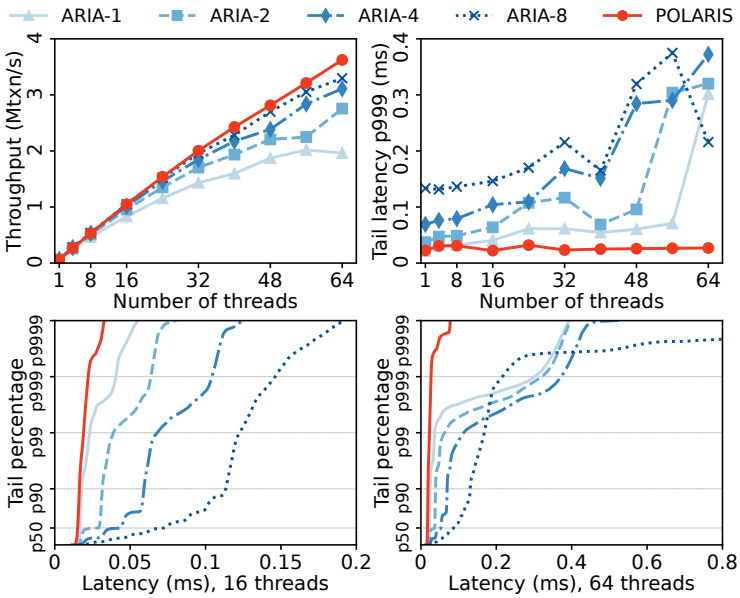

Fig. 10. Throughput and p999 tail latency over a spectrum of thread numbers; latency distribution in the cases of 16 and 64 threads (YCSB-A, $r = 50\%, w = 50\%, \theta = 0.5$).

cost of each abort, as re-execution has to wait until the next batch. When the abort rate is high, this further results in high tail latency.

Varying Number of Threads at Low Contention. Figure 10 shows the results with YCSB-A workload ($\theta = 0.5$). When contention is low, a large batch size in Aria does not suffer from the high abort rate. The major factor that affects throughput is the global barriers across all threads. Each batch contains one barrier for the execution phase and another for the commit phase. A larger per-thread batch size reduces the frequency of barriers and thus, yields better throughput.

We observe Aria has fluctuating p999 tail latency when there are more than 32 threads at $\theta = 0.5$. Under this workload, only < 0.08% of transactions have experienced aborts, so the transaction at p999 tail commits at its first execution. We think it is the large batch sizes that amplify noise (e.g., one slow transaction causes all transactions in the batch to wait) and lead to fluctuation.

6 RELATED WORK

In this section, we describe some previous work on transaction priority and hybrid concurrency control protocols.

Transaction Priority. The support for transaction priority in OCC has been studied in real-time databases [1, 12, 13, 15], where each transaction has a deadline, and an earlier deadline implies a higher priority. Haritsa et al. [12] present a few priority-aware OCC variants. Built on top of the original OCC [18], these variants require a global critical section for validation and global visibility of each transaction's read-set and write-set, which can hurt scalability. Polaris differs from this work by embedding the priority-related conflict detection into each record, which avoids global operations and data structures.

Ding et al. [8] propose to use transaction batching and reordering to enable priority in OCC. By delaying the decision of transaction serialization order within a batch, a validator can reorder transactions to give certain ones a better chance to commit. However, this approach introduces non-trivial overhead for searching reordering schemes, and batching can hurt median latency.

Hybrid Transactional and Analytical Processing (HTAP) databases are systems that serve both transactional and analytical queries. To prevent one type of workload from starving due to the others, existing HTAP systems [14, 16, 19, 22] typically use MVCC. Though Polaris can also achieve such starvation prevention through priority, it targets transactional workload and is not designed as a counterpart to the existing HTAP systems.

Hybrid Concurrency Control. Combining 2PL and OCC to get the best of both worlds is not a nascent idea. A classical approach is to dynamically switch between concurrency control protocols based on the workload. MOCC [29] dynamically partitions records, and each partition can be managed by either OCC or 2PL. MOCC uses OCC by default to obtain high throughput but can switch to a No-Wait variant 2PL once hot spots are detected in the partition. Similarly, CormCC [26] presents a framework to dynamically switch between multiple concurrency control protocols. Polyjuice [28] focuses on decomposing a transaction into smaller actions. It dynamically chooses the optimal variant for each action with learning-based techniques and allows optional validations after each read/write access to avoid wasting work.

Plor [3] is recent work that integrates optimism into Wound-Wait. In Plor, read operations are mostly optimistic; a write lock only blocks readers when the owner enters the commit phase. Such optimism brings the benefit of higher throughput while preserving the low tail latency of Wound-Wait. Polaris differs from Plor in two aspects: 1) Polaris is a lightweight add-on to OCC and generally applicable to many OCC protocols, while Plor makes deeper changes and functions as a standalone protocol. 2) Polaris is based on OCC with minimal pessimism, and thus, inherits the

advantages of reducing shared-memory write; Plor is based on Wound-Wait with added optimism, but it does not avoid lock acquisition overhead.

7 CONCLUSION

In this paper, we present Polaris, a new optimistic concurrency control protocol that supports multiple priority levels. Polaris introduces a lightweight reservation mechanism to enforce priority. Transactions within the same priority level can proceed in a fully optimistic way. A low-priority transaction cannot write to a record reserved by a high-priority transaction, but it can read it as in OCC protocols. A high-priority transaction can preempt records reserved by low-priority transactions. With a minimal amount of pessimism, Polaris preserves the throughput advantage of OCC, while enabling the powerful feature of transaction priority.

ACKNOWLEDGMENTS

We appreciate the valuable feedback from the anonymous reviewers. We thank Guanzhou Hu for the insightful discussion. This work was supported by NSF grant 2144588.

REFERENCES

[1] Robert K. Abbott and Hector Garcia-Molina. 1992. Scheduling Real-Time Transactions: A Performance Evaluation. *ACM Trans. Database Syst.* 17, 3 (sep 1992), 513–560. https://doi.org/10.1145/132271.132276

[2] William Assaf. n.d.. SET DEADLOCK_PRIORITY (Transact-SQL) - SQL Server. https://docs.microsoft.com/en-us/sql/t-sql/statements/set-deadlock-priority-transact-sql?view=sql-server-ver15

[3] 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* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 19–33. https://doi.org/10.1145/3514221.3517879

[4] CockroachDB. n.d.. Transaction | CockroachDB Docs. https://www.cockroachlabs.com/docs/v21.2/transactions#transaction-priorities

[5] 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) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[6] Transaction Processing Performance Council. 1992. TPC-C: an On-Line Transaction Processing Benchmark. http://www.tpc.org/tpcc/

[7] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56 (2013), 74–80. http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext

[8] Bailu Ding, Lucja Kot, and Johannes Gehrke. 2018. Improving Optimistic Concurrency Control through Transaction Batching and Operation Reordering. *Proc. VLDB Endow.* 12, 2 (oct 2018), 169–182. https://doi.org/10.14778/3282495.3282502

[9] 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 *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 1–14. https://www.usenix.org/conference/atc19/presentation/duplyakin

[10] Zhihan Guo, Kan Wu, Cong Yan, and Xiangyao Yu. 2021. Bamboo codebase. https://github.com/ScarletGuo/Bamboo-Public

[11] 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*. Association for Computing Machinery, New York, NY, USA, 658–670. https://doi.org/10.1145/3448016.3457294

[12] J.R. Haritsa, M.J. Carey, and M. Livny. 1990. Dynamic real-time optimistic concurrency control. In *[1990] Proceedings 11th Real-Time Systems Symposium*. 94–103. https://doi.org/10.1109/REAL.1990.128734

[13] Jayant R. Haritsa, Michael J. Carey, and Miron Livny. 1990. On Being Optimistic about Real-Time Constraints. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Nashville, Tennessee, USA) *(PODS '90)*. Association for Computing Machinery, New York, NY, USA, 331–343. https://doi.org/10.1145/298514.298585

[14] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: A Raft-Based HTAP Database. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3072–3084. https://doi.org/10.14778/3415478.3415535

[15] S. L. Hung and K. Y. Lam. 1992. Locking Protocols for Concurrency Control in Real-Time Database Systems. *SIGMOD Rec.* 21, 4 (dec 1992), 22–27. https://doi.org/10.1145/141818.141822

[16] Jongbin Kim, Jaeseon Yu, Jaechan Ahn, Sooyong Kang, and Hyungsoo Jung. 2022. Diva: Making MVCC Systems HTAP-Friendly. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 49–64. https://doi.org/10.1145/3514221.3526135

[17] Hideaki Kimura. 2015. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 691–706. https://doi.org/10.1145/2723372.2746480

[18] 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

[19] Juchang Lee, Michael Muehle, Norman May, Franz Faerber, Vishal Sikka, Hasso Plattner, Jens Krueger, and Martin Grund. 2013. High-Performance Transaction Processing in SAP HANA. *IEEE Data Eng. Bull.* 36, 2 (2013), 28–33.

[20] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 21–35. https://doi.org/10.1145/3035918.3064015

[21] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: A Fast and Practical Deterministic OLTP Database. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2047–2060. https://doi.org/10.14778/3407790.3407808

[22] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 677–689. https://doi.org/10.1145/2723372.2749436

[23] Oracle. 2019. Berkeley DB C++ API Reference. https://docs.oracle.com/database/bdb181/html/api_reference/CXX/txnset_priority.html

[24] FoundationDB project authors. 2022. Python API — FoundationDB 7.1. https://apple.github.io/foundationdb/api-python.html#transaction-options

[25] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1493–1509. https://doi.org/10.1145/3318464.3386134

[26] Dixin Tang and Aaron J. Elmore. 2018. Toward Coordination-free and Reconfigurable Mixed Concurrency Control. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 809–822. https://www.usenix.org/conference/atc18/presentation/tang

[27] 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) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 18–32. https://doi.org/10.1145/2517349.2522713

[28] 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 (OSDI 21)*. USENIX Association, 198–216. https://www.usenix.org/conference/osdi21/presentation/wang-jiachen

[29] Tianzheng Wang and Hideaki Kimura. 2016. Mostly-Optimistic Concurrency Control for Highly Contended Dynamic Workloads on a Thousand Cores. *Proc. VLDB Endow.* 10, 2 (oct 2016), 49–60. https://doi.org/10.14778/3015274.3015276

[30] 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) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 279–294. https://doi.org/10.1145/2815400.2815430

[31] Chenhao Ye, Wuh-Chwen Hwang, Keren Chen, and Xiangyao Yu. 2022. Polaris codebase. https://github.com/chenhao-ye/polaris

[32] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. DBx1000 codebase. https://github.com/yxymit/DBx1000

[33] 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

[34] 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) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1629–1642. https://doi.org/10.1145/2882903.2882935

[35] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sanchez, Larry Rudolph, and Srinivas Devadas. 2018. Sundial: Harmonizing Concurrency Control and Caching in a Distributed OLTP Database Management System. *Proc. VLDB Endow.* 11, 10 (jun 2018), 1289–1302. https://doi.org/10.14778/3231751.3231763

[36] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. 2021. *FoundationDB: A Distributed Unbundled Transactional Key Value Store.* Association for Computing Machinery, New York, NY, USA, 2653–2666. https://doi.org/10.1145/3448016.3457559

Received April 2022; revised July 2022; accepted August 2022