

Polaris: Enabling Transaction Priority in Optimistic Concurrency Control

Chenhao Ye, Wuh-Chwen Hwang, Keren Chen, Xiangyao Yu



Transaction Priority is a Critical Feature

Transaction Priority is a Critical Feature

Under high contention, certain classes of transactions should be given a higher chance to commit than others

Transaction Priority is a Critical Feature

Under high contention, certain classes of transactions should be given a higher chance to commit than others

Examples:

Repeated aborted transactions

Prioritize for starvation-freedom
& low tail latency

Transaction Priority is a Critical Feature

Under high contention, certain classes of transactions should be given a higher chance to commit than others

Examples:

Repeated aborted transactions

Prioritize for starvation-freedom
& low tail latency

User-initiated transactions

Prioritize over background jobs
to improve user experience

Spectrum of Concurrency Control

	Pessimistic Concurrency Control (Two-Phase Locking)	Optimistic Concurrency Control (OCC)
No Priority		
With Priority		

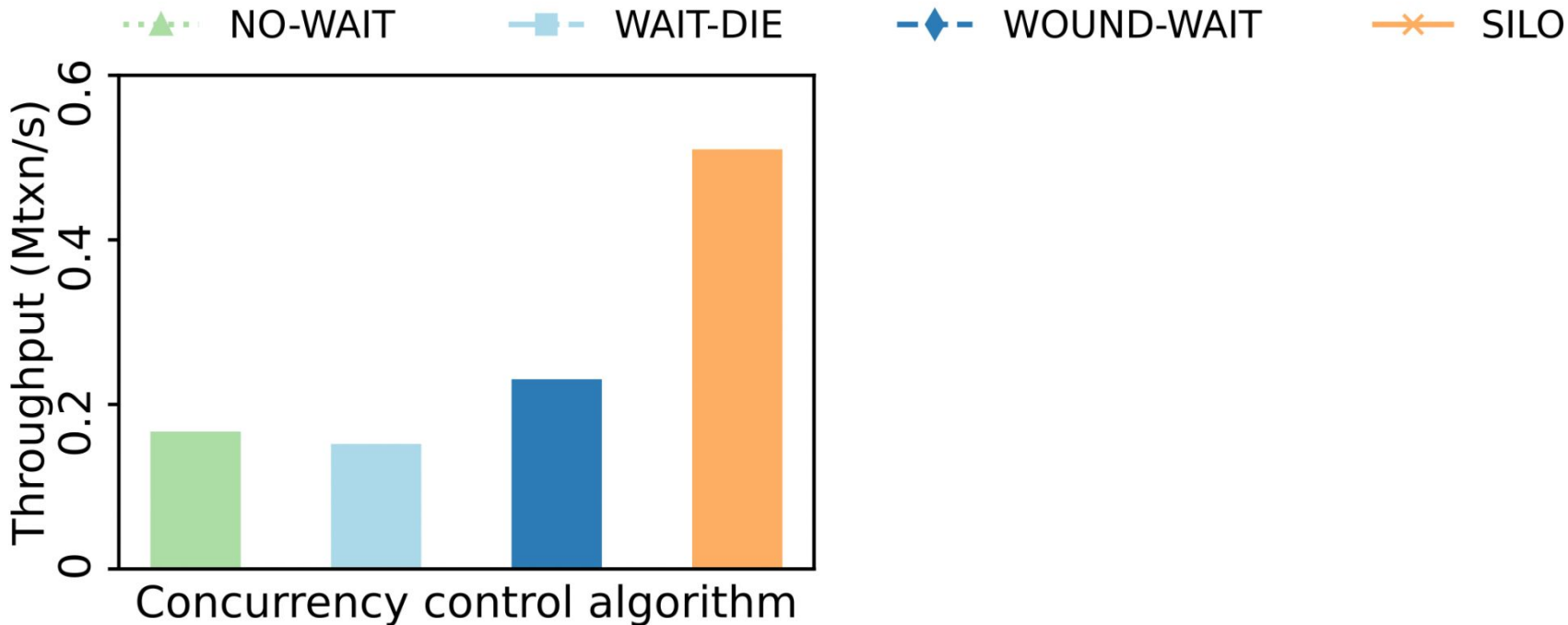
Spectrum of Concurrency Control

	Pessimistic Concurrency Control (Two-Phase Locking)	Optimistic Concurrency Control (OCC)
No Priority	No-Wait	
With Priority	Wait-Die Wound-Wait	

Spectrum of Concurrency Control

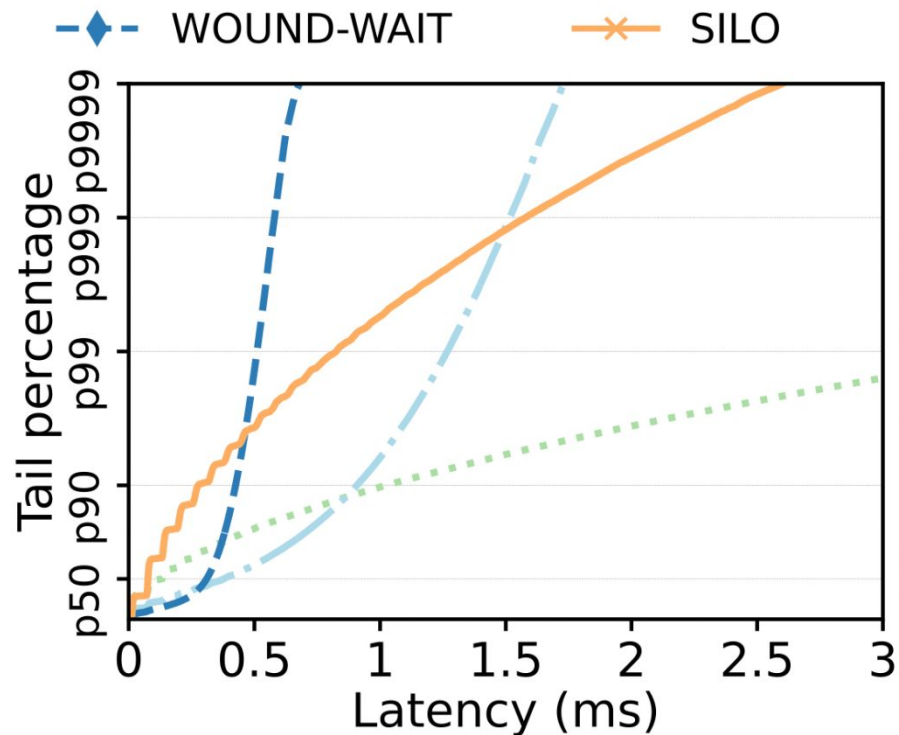
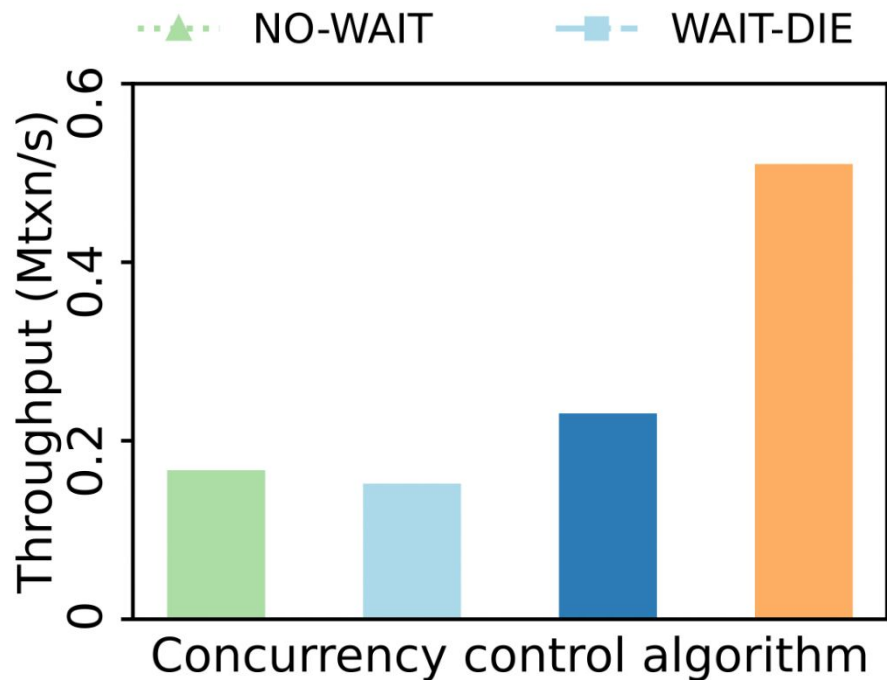
	Pessimistic Concurrency Control (Two-Phase Locking)	Optimistic Concurrency Control (OCC)
No Priority	No-Wait	Version-number-based e.g., Silo [SOSP '13]
With Priority	Wait-Die Wound-Wait	

Performance Comparison: 2PL vs. OCC



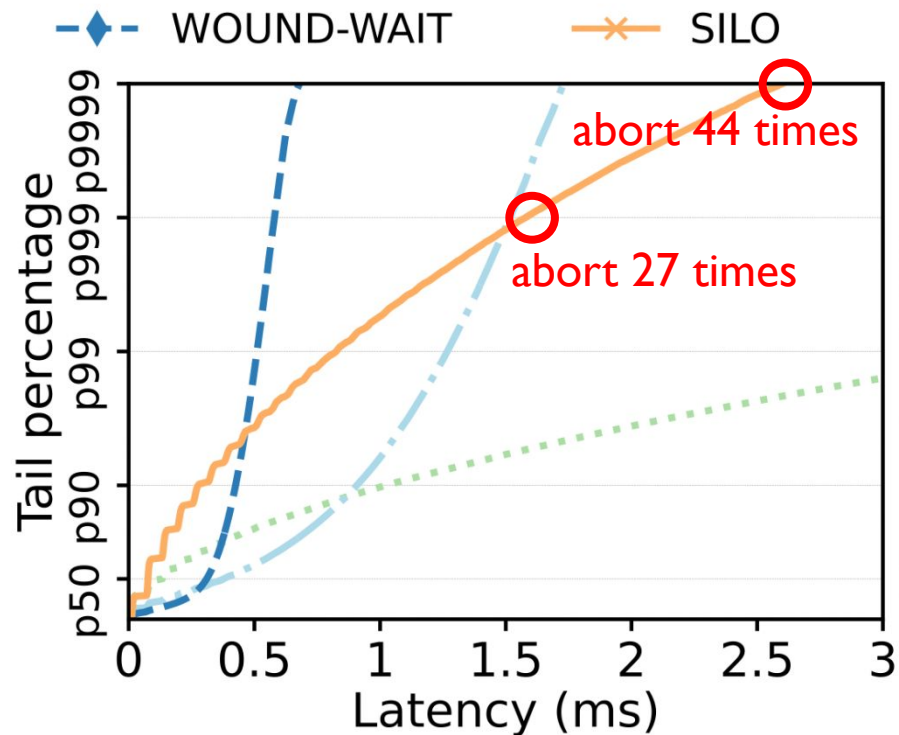
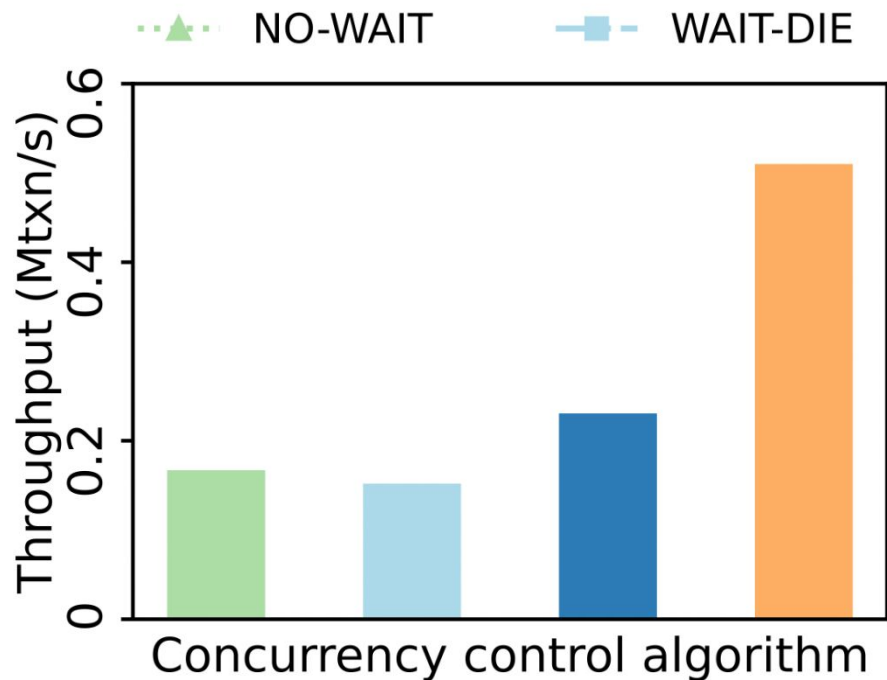
YCSB-A (Read 50%, Write 50%), Zipfian $\theta=0.99$, 64 threads

Performance Comparison: 2PL vs. OCC



YCSB-A (Read 50%, Write 50%), Zipfian $\theta=0.99$, 64 threads

Performance Comparison: 2PL vs. OCC



YCSB-A (Read 50%, Write 50%), Zipfian $\theta=0.99$, 64 threads

Spectrum of Concurrency Control

	Pessimistic Concurrency Control (Two-Phase Locking)	Optimistic Concurrency Control (OCC)
No Priority	No-Wait	Version-number-based e.g., Silo [SOSP '13]
With Priority	Wait-Die Wound-Wait	?

Rich Features Enabled by Priority:
Low tail latency, starvation freedom, user-defined priority, etc.

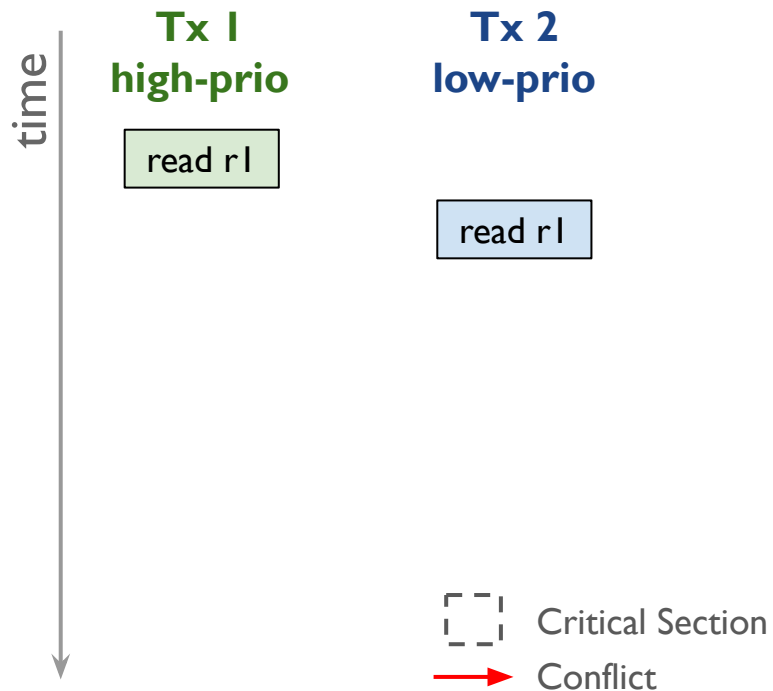
Spectrum of Concurrency Control

	Pessimistic Concurrency Control (Two-Phase Locking)	Optimistic Concurrency Control (OCC)
No Priority	No-Wait	Version-number-based e.g., Silo [SOSP '13]
With Priority	Wait-Die Wound-Wait	Polaris Enable priority while keeping maximum optimistic advantages

Rich Features Enabled by Priority:
Low tail latency, starvation freedom, user-defined priority, etc.

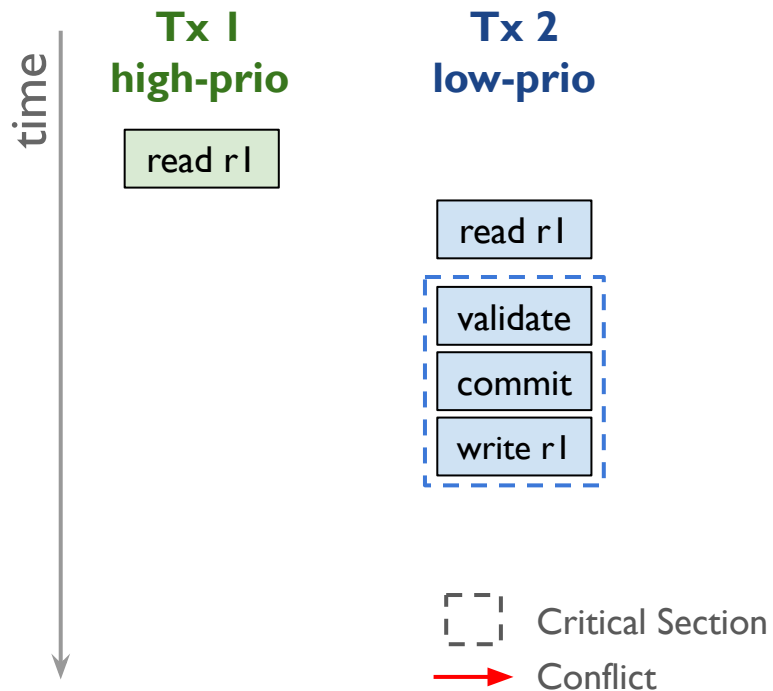
Priority Enforcement in OCC

Example in Silo: Priority Inversion



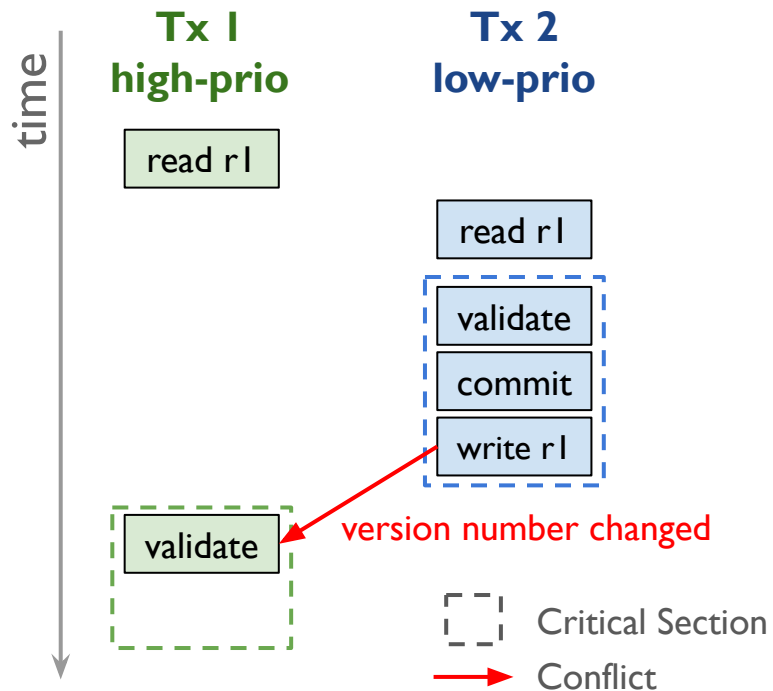
Priority Enforcement in OCC

Example in Silo: Priority Inversion



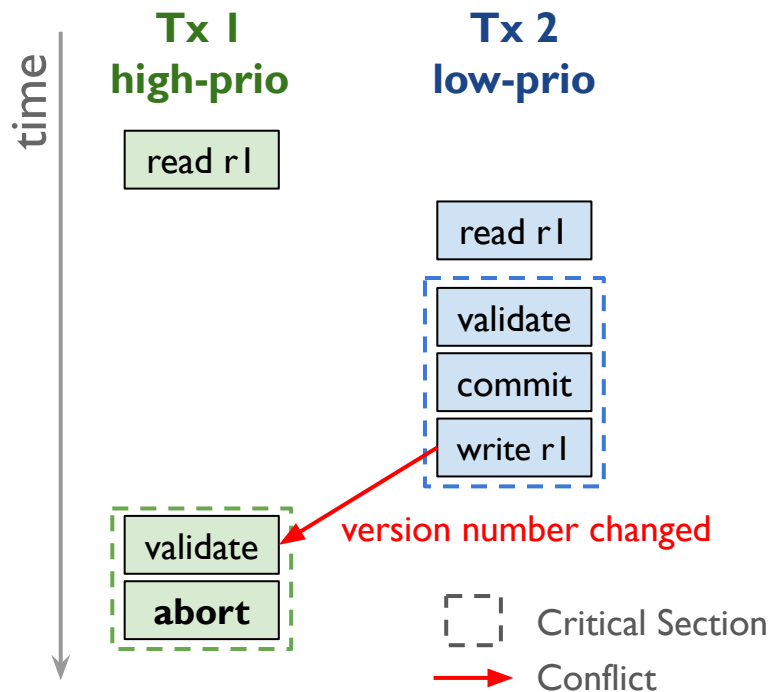
Priority Enforcement in OCC

Example in Silo: Priority Inversion



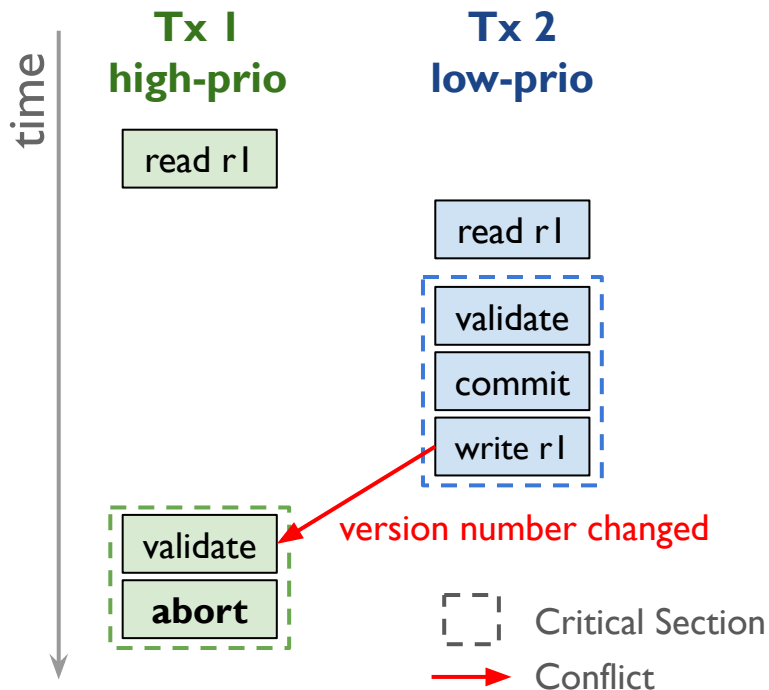
Priority Enforcement in OCC

Example in Silo: Priority Inversion



Priority Enforcement in OCC

Example in Silo: Priority Inversion



Goal: Priority Enforcement

In the case of a conflict, a high-priority transaction must not be aborted by a lower-priority transaction

Challenge

Silo only detects conflicts at commit time, where a low-priority transaction may have committed already

Idea: Reservation Mechanism

Desired Guarantee

If a transaction is the only highest-priority transaction accessing a record, it will pass the validation on this record

Idea: Reservation Mechanism

Desired Guarantee

If a transaction is the only highest-priority transaction accessing a record, it will pass the validation on this record

Propose: Reservation Mechanism

A transaction can **reserve** a record so that lower-priority transactions cannot write to this record (but can still read it)

A record can be reserved by multiple same-priority transactions, but not cross-priority

Idea: Reservation Mechanism

Desired Guarantee

If a transaction is the only highest-priority transaction accessing a record, it will pass the validation on this record

Propose: Reservation Mechanism

A transaction can **reserve** a record so that lower-priority transactions cannot write to this record (but can still read it)

A record can be reserved by multiple same-priority transactions, but not cross-priority

Reservation Principles:

1. A low-priority transaction **cannot reserve** a record already reserved by high-priority transactions
2. A high-priority transaction can **preempt** a record reserved by a low-priority transaction
3. The protocol remains **fully optimistic** among same-priority transactions

Idea: Reservation Mechanism

Desired Guarantee

If a transaction is the only highest-priority transaction accessing a record, it will pass the validation on this record

Propose: Reservation Mechanism

A transaction can **reserve** a record so that lower-priority transactions cannot write to this record (but can still read it)

A record can be reserved by multiple same-priority transactions, but not cross-priority

Reservation Principles:

1. A low-priority transaction **cannot reserve** a record already reserved by high-priority transactions
2. A high-priority transaction can **preempt** a record reserved by a low-priority transaction
3. The protocol remains **fully optimistic** among same-priority transactions

Idea: Reservation Mechanism

Desired Guarantee

If a transaction is the only highest-priority transaction accessing a record, it will pass the validation on this record

Propose: Reservation Mechanism

A transaction can **reserve** a record so that lower-priority transactions cannot write to this record (but can still read it)

A record can be reserved by multiple same-priority transactions, but not cross-priority

Reservation Principles:

1. A low-priority transaction **cannot reserve** a record already reserved by high-priority transactions
2. A high-priority transaction can **preempt** a record reserved by a low-priority transaction
3. The protocol remains **fully optimistic** among same-priority transactions

Develop Polaris
(from Silo)

Develop Polaris (from Silo): Data structure

Silo: Transaction ID (TID)

Per-record 64-bit word encoding:

- Data version
- Latch bit

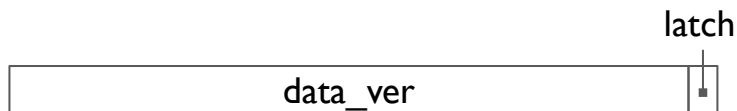


Develop Polaris (from Silo): Data structure

Silo: Transaction ID (TID)

Per-record 64-bit word encoding:

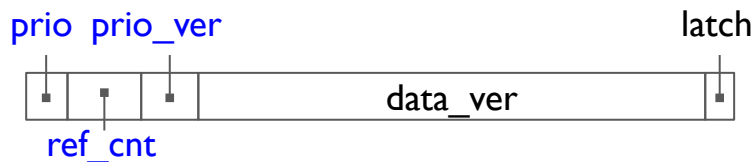
- Data version
- Latch bit



Polaris: Extended TID

Similar 64-bit word encoding:

- Data version
- Latch bit
- **Priority**: which level priority of transactions are reserving this record
- **Reference counter**: how many of reservees
- **Priority version**: for cleanup



Execution Phase

Silo: Execution Phase

In the execution phase, a transaction maintains a read-set $\underline{R}$ and a write-set $\underline{W}$

For a record r accessed by transaction tx :

```
do tid = r.tid
while tid.latch == LOCKED
// ensure tid is unlocked
```

```
r_local = r.copy()
```

Silo: Execution Phase

In the execution phase, a transaction maintains a read-set $\underline{R}$ and a write-set $\underline{W}$

For a record r accessed by transaction tx :

```
do
  tid = r.tid
  while tid.latch == LOCKED
    // ensure tid is unlocked

  r_local = r.copy()
while r.tid != tid

R.add(r, tid)
```

Silo: Execution Phase

In the execution phase, a transaction maintains a read-set $\underline{R}$ and a write-set $\underline{W}$

For a record r accessed by transaction tx :

```
do
  tid = r.tid
  while tid.latch == LOCKED
    // ensure tid is unlocked

  r_local = r.copy()
while r.tid != tid

R.add(r, tid)
if want_write: W.add(r)
return r_local
```

All updates are applied to **r_local**

Polaris: Execution Phase

In the execution phase, a transaction maintains a read-set $\underline{R}$ and a write-set $\underline{W}$

For a record r accessed by transaction tx :

```
do
  tid = r.tid
  while tid.latch == LOCKED
    new_tid, is_reserved = \
      try_reserve(tid, tx.prio, want_write)
    r_local = r.copy()
  while !(r.tid.CAS(tid, new_tid))
```

```
R.add(r, tid)
if want_write: W.add(r)
return r_local
```

All updates are applied to **r_local**

Reservation Protocol:

```
def try_reserve(tid, tx.prio, want_write):
    new_tid = tid
    if tx.prio == tid.prio: // co-reserve
        new_tid.ref_cnt++
        is_reserved = true
    elif tx.prio > tid.prio: // preempt
        new_tid.prio = tx.prio
        new_tid.ref_cnt = 1
        is_reserved = true
    else: // fail to reserve
        if want_write: abort()
        is_reserved = false
    return new_tid, is_reserved
```

Polaris: Execution Phase

In the execution phase, a transaction maintains a read-set $\underline{R}$ and a write-set $\underline{W}$

For a record r accessed by transaction tx :

```
do
  tid = r.tid
  while tid.latch == LOCKED
    new_tid, is_reserved = \
      try_reserve(tid, tx.prio, want_write)
    r_local = r.copy()
  while !(r.tid.CAS(tid, new_tid))
```

```
R.add(r, tid)
if want_write: W.add(r)
return r_local
```

All updates are applied to **r_local**

Reservation Protocol:

```
def try_reserve(tid, tx.prio, want_write):
  new_tid = tid
  if tx.prio == tid.prio: // co-reserve
    new_tid.ref_cnt++
    is_reserved = true
  elif tx.prio > tid.prio: // preempt
    new_tid.prio = tx.prio
    new_tid.ref_cnt = 1
    is_reserved = true
  else: // fail to reserve
    if want_write: abort()
    is_reserved = false
  return new_tid, is_reserved
```

Polaris: Execution Phase

In the execution phase, a transaction maintains a read-set $\underline{R}$ and a write-set $\underline{W}$

For a record r accessed by transaction tx :

```
do
  tid = r.tid
  while tid.latch == LOCKED
    new_tid, is_reserved = \
      try_reserve(tid, tx.prio, want_write)
    r_local = r.copy()
  while !(r.tid.CAS(tid, new_tid))
```

```
R.add(r, tid)
if want_write: W.add(r)
return r_local
```

All updates are applied to r_local

Reservation Protocol:

```
def try_reserve(tid, tx.prio, want_write):
  new_tid = tid
  if tx.prio == tid.prio: // co-reserve
    new_tid.ref_cnt++
    is_reserved = true
  elif tx.prio > tid.prio: // preempt
    new_tid.prio = tx.prio
    new_tid.ref_cnt = 1
    is_reserved = true
  else: // fail to reserve
    if want_write: abort()
    is_reserved = false
  return new_tid, is_reserved
```

Polaris: Execution Phase

In the execution phase, a transaction maintains a read-set $\underline{R}$ and a write-set $\underline{W}$

For a record r accessed by transaction tx :

```
do
  tid = r.tid
  while tid.latch == LOCKED
    new_tid, is_reserved = \
      try_reserve(tid, tx.prio, want_write)
    r_local = r.copy()
  while !(r.tid.CAS(tid, new_tid))
```

```
R.add(r, tid)
if want_write: W.add(r)
return r_local
```

All updates are applied to r_local

Reservation Protocol:

```
def try_reserve(tid, tx.prio, want_write):
    new_tid = tid
    if tx.prio == tid.prio: // co-reserve
        new_tid.ref_cnt++
        is_reserved = true
    elif tx.prio > tid.prio: // preempt
        new_tid.prio = tx.prio
        new_tid.ref_cnt = 1
        is_reserved = true
    else: // fail to reserve
        if want_write: abort()
        is_reserved = false
    return new_tid, is_reserved
```

Validation Phase

Silo:Validation Phase

First acquire latches of all records in write-set in sorted order

```
for r in sorted(W):  
    tid = r.tid  
    if tid.latch == LOCKED: abort()  
  
    r.tid.CAS(tid, locked(tid))
```

Then validate every record in read-set is unlatched and has data version unchanged

```
for r, old_tid in R:  
    if r.latch == LOCKED and r not in W:  
        abort()  
    if r.tid.data_ver != old_tid.data_ver:  
        abort()
```

Silo:Validation Phase

First acquire latches of all records in write-set in sorted order

```
for r in sorted(W):  
    tid = r.tid  
    if tid.latch == LOCKED: abort()
```

```
    r.tid.CAS(tid, locked(tid))
```

Then validate every record in read-set is unlatched and has data version unchanged

```
for r, old_tid in R:  
    if r.latch == LOCKED and r not in W:  
        abort()  
    if r.tid.data_ver != old_tid.data_ver:  
        abort()
```

If it passes validation, the transaction **commits**

It publishes local writes to records in write-set, together with TID encoding new data version

```
new_data_ver = W.max_data_ver() + 1  
for r in W:  
    r.install_write()  
    r.tid = {new_data_ver, UNLOCKED}
```

Polaris: Validation Phase

First acquire latches of all records in write-set in sorted order

```
for r in sorted(W):
    tid = r.tid
    if tid.latch == LOCKED: abort()
    if tid.prio > tx.prio: abort()
    r.tid.CAS(tid, locked(tid))
```

Then validate every record in read-set is unlatched and has data version unchanged

```
for r, old_tid in R:
    if r.latch == LOCKED and r not in W:
        abort()
    if r.tid.data_ver != old_tid.data_ver:
        abort()
```

If it passes validation, the transaction **commits**

It publishes local writes to records in write-set, together with TID encoding new data version
Every reserved record must perform cleanup (next slide)

```
new_data_ver = W.max_data_ver() + 1
for r in W:
    r.install_write()
    r.tid = cleanup_write(r.tid, \
                          new_data_ver)

// read-only records
for r, old_tid in R if r not in W:
    r.tid = cleanup_read(tid, tx.prio, \
                        old_tid.prio_ver)
```

Polaris: Cleanup Protocol (Committed)

For records in write-set:

```
def cleanup_write(tid, new_data_ver):
    new_tid = {
        .data_ver = new_data_ver
        .latch    = UNLOCKED
        .prio     = 0
        .ref_cnt  = 0
        .prio_ver = tid.prio_ver + 1
    }
    return new_tid
```

For read-only records:

```
def cleanup_read(tid, tx.prio, \
                  old_tid.prio_ver):
    // check if reservation still in TID
    if tid.latch == LOCKED or \
        tid.prio != tx.prio or \
        tid.prio_ver != old_tid.prio_ver:
        return tid // no cleanup needed

    new_tid = tid
    new_tid.ref_cnt--
    if new_tid.ref_cnt == 0:
        new_tid.prio = 0
        new_tid.prio_ver++
    return new_tid
```

Polaris: Cleanup Protocol (Committed)

For records in write-set:

```
def cleanup_write(tid, new_data_ver):
    new_tid = {
        .data_ver = new_data_ver
        .latch    = UNLOCKED
        .prio     = 0
        .ref_cnt  = 0
        .prio_ver = tid.prio_ver + 1
    }
    return new_tid
```

For read-only records:

```
def cleanup_read(tid, tx.prio, \
                  old_tid.prio_ver):
    // check if reservation still in TID
    if tid.latch == LOCKED or \
        tid.prio != tx.prio or \
        tid.prio_ver != old_tid.prio_ver:
        return tid // no cleanup needed

    new_tid = tid
    new_tid.ref_cnt--
    if new_tid.ref_cnt == 0:
        new_tid.prio = 0
        new_tid.prio_ver++
    return new_tid
```

Polaris: Cleanup Protocol (Committed)

For records in write-set:

```
def cleanup_write(tid, new_data_ver):
    new_tid = {
        .data_ver = new_data_ver
        .latch    = UNLOCKED
        .prio     = 0
        .ref_cnt  = 0
        .prio_ver = tid.prio_ver + 1
    }
    return new_tid
```

For read-only records:

```
def cleanup_read(tid, tx.prio, \
                  old_tid.prio_ver):
    // check if reservation still in TID
    if tid.latch == LOCKED or \
        tid.prio != tx.prio or \
        tid.prio_ver != old_tid.prio_ver:
        return tid // no cleanup needed

    new_tid = tid
    new_tid.ref_cnt--
    if new_tid.ref_cnt == 0:
        new_tid.prio = 0
        new_tid.prio_ver++
    return new_tid
```

Polaris: Cleanup Protocol (Committed)

For records in write-set:

```
def cleanup_write(tid, new_data_ver):
    new_tid = {
        .data_ver = new_data_ver
        .latch    = UNLOCKED
        .prio     = 0
        .ref_cnt  = 0
        .prio_ver = tid.prio_ver + 1
    }
    return new_tid
```

For read-only records:

```
def cleanup_read(tid, tx.prio, \
                  old_tid.prio_ver):
    // check if reservation still in TID
    if tid.latch == LOCKED or \
        tid.prio != tx.prio or \
        tid.prio_ver != old_tid.prio_ver:
        return tid // no cleanup needed

    new_tid = tid
    new_tid.ref_cnt--
    if new_tid.ref_cnt == 0:
        new_tid.prio = 0
        new_tid.prio_ver++
    return new_tid
```

Polaris: Cleanup Protocol (Committed)

For records in write-set:

```
def cleanup_write(tid, new_data_ver):
    new_tid = {
        .data_ver = new_data_ver
        .latch    = UNLOCKED
        .prio     = 0
        .ref_cnt  = 0
        .prio_ver = tid.prio_ver + 1
    }
    return new_tid
```

For read-only records:

```
def cleanup_read(tid, tx.prio, \
                  old_tid.prio_ver):
    // check if reservation still in TID
    if tid.latch == LOCKED or \
        tid.prio != tx.prio or \
        tid.prio_ver != old_tid.prio_ver:
        return tid // no cleanup needed

    new_tid = tid
    new_tid.ref_cnt--
    if new_tid.ref_cnt == 0:
        new_tid.prio = 0
        new_tid.prio_ver++
    return new_tid
```

Polaris: Cleanup Protocol (Committed)

For records in write-set:

```
def cleanup_write(tid, new_data_ver):
    new_tid = {
        .data_ver = new_data_ver
        .latch    = UNLOCKED
        .prio     = 0
        .ref_cnt  = 0
        .prio_ver = tid.prio_ver + 1
    }
    return new_tid
```

For read-only records:

```
def cleanup_read(tid, tx.prio, \
                  old_tid.prio_ver):
    // check if reservation still in TID
    if tid.latch == LOCKED or \
        tid.prio != tx.prio or \
        tid.prio_ver != old_tid.prio_ver:
        return tid // no cleanup needed

    new_tid = tid
    new_tid.ref_cnt--
    if new_tid.ref_cnt == 0:
        new_tid.prio = 0
        new_tid.prio_ver++
    return new_tid
```

Polaris: Cleanup Protocol (Committed)

For records in write-set:

```
def cleanup_write(tid, new_data_ver):
    new_tid = {
        .data_ver = new_data_ver
        .latch     = UNLOCKED
        .prio      = 0
        .ref_cnt   = 0
        .prio_ver  = tid.prio_ver + 1
    }
    return new_tid
```

For read-only records:

```
def cleanup_read(tid, tx.prio, \
                  old_tid.prio_ver):
    // check if reservation still in TID
    if tid.latch == LOCKED or \
        tid.prio != tx.prio or \
        tid.prio_ver != old_tid.prio_ver:
        return tid // no cleanup needed

    new_tid = tid
    new_tid.ref_cnt--
    if new_tid.ref_cnt == 0:
        new_tid.prio = 0
        new_tid.prio_ver++
    return new_tid
```

If the transaction aborts, it also needs to perform cleanup (more details in paper)

Optimization

Note that reservation is to protect a high-priority transactions from the lower-priority ones

If a transaction is already in the lowest priority level, it does not need to make reservation at all

Optimization

Note that reservation is to protect a high-priority transactions from the lower-priority ones

If a transaction is already in the lowest priority level, it does not need to make reservation at all

Benefit:

This avoids a shared-memory write to TID during execution and saves the cost of cache coherence protocol

When all transactions are in the lowest priority level, Polaris gracefully falls back to Silo

Evaluation

Evaluation

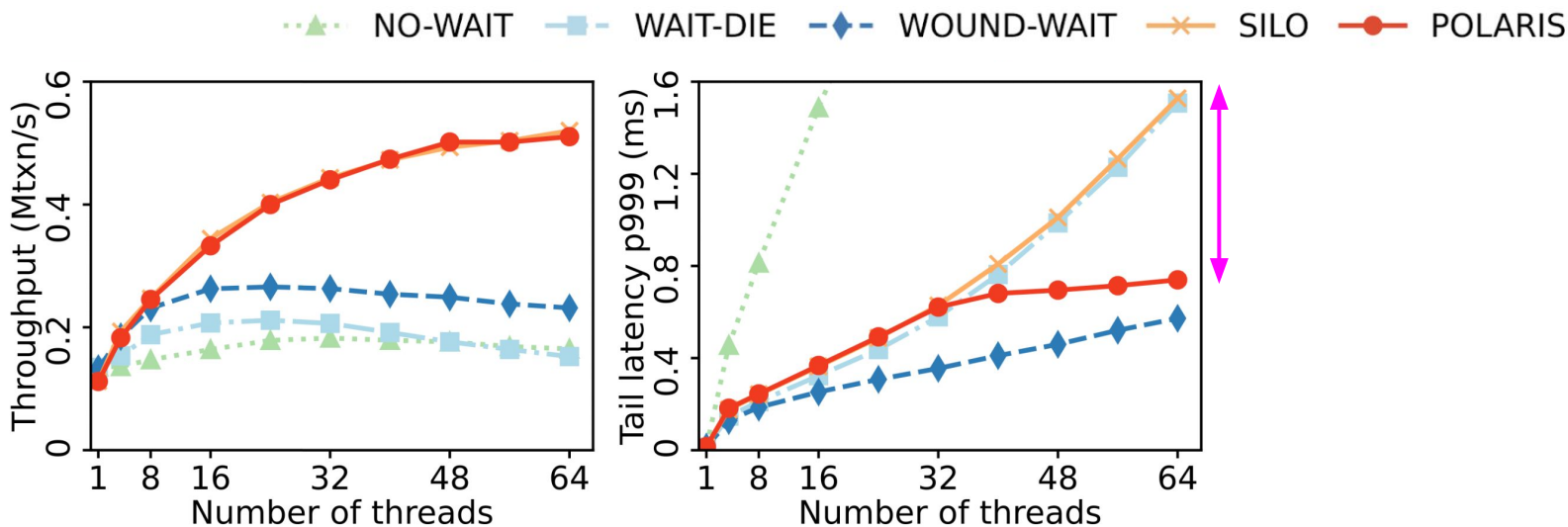
Polaris is implemented on DBx1000

The experiments runs on a machine with 64 logical cores

We use the following simple (but powerful) priority assignment policy:

- All transactions begin with priority 0
- After being aborted 11 times, increment its priority by 1 for every 3 aborts
- Maximum priority is 16

Evaluation: Varying Threads

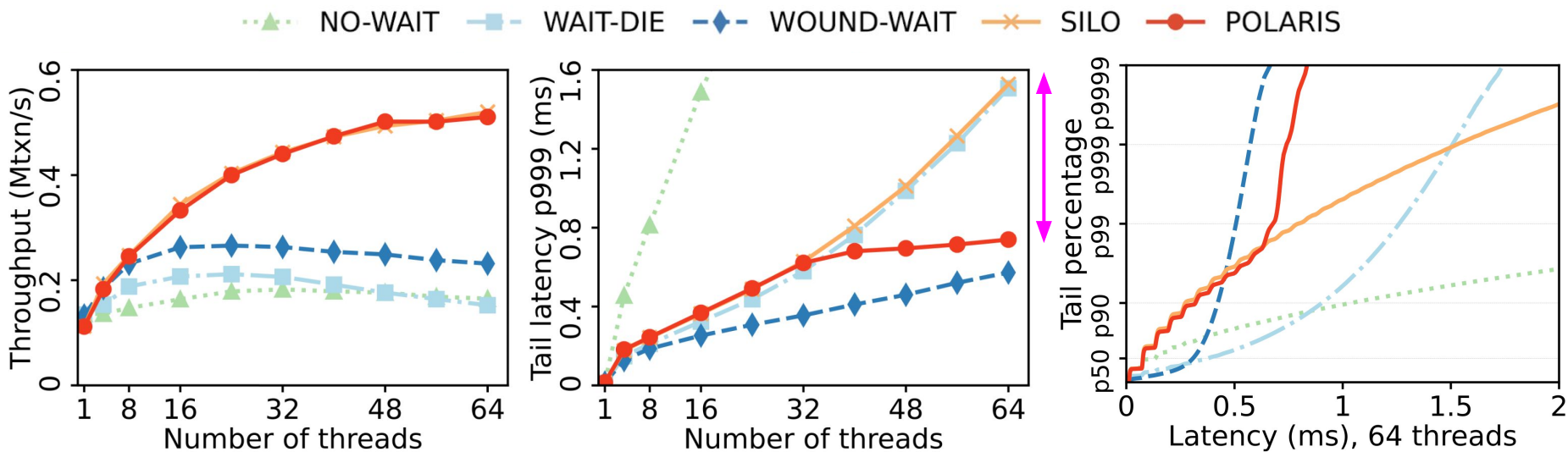


YCSB-A (Read 50%, Write 50%), Zipfian $\theta=0.99$

At 32 threads, only 0.12% transactions in Polaris have non-zero priority

At 64 threads, 2.5% transactions in Polaris commits with priority 1 and 0.01% commits with priority 2

Evaluation: Varying Threads

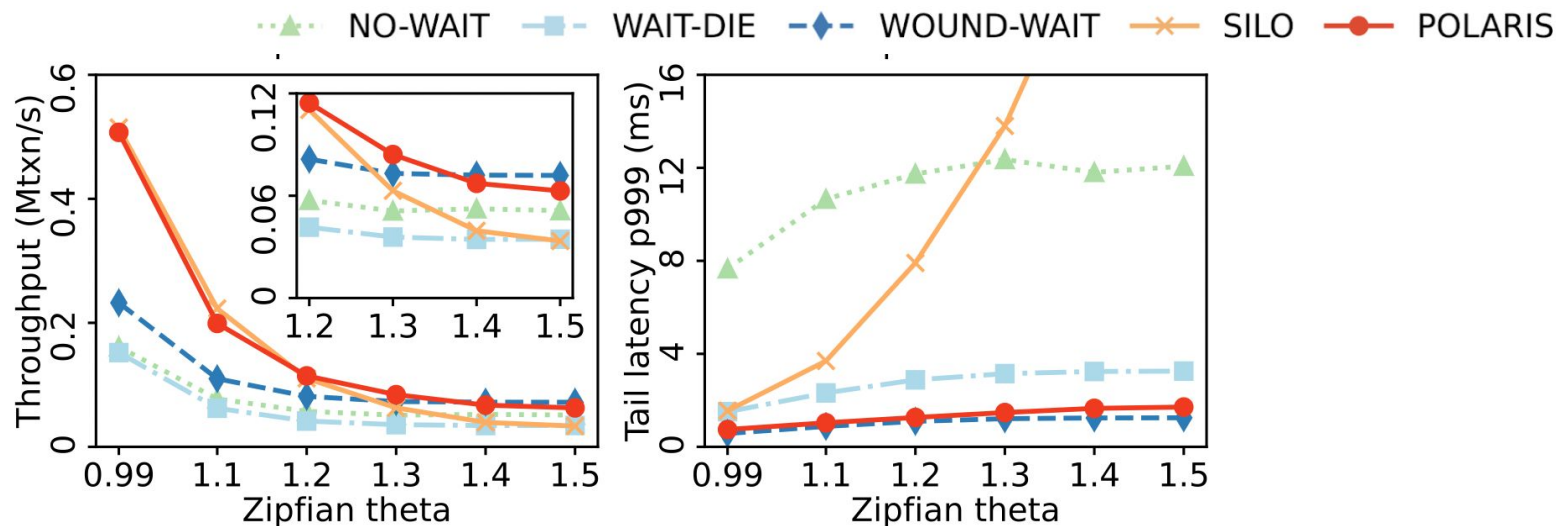


YCSB-A (Read 50%, Write 50%), Zipfian $\theta=0.99$

At 32 threads, only 0.12% transactions in Polaris have non-zero priority

At 64 threads, 2.5% transactions in Polaris commits with priority 1 and 0.01% commits with priority 2

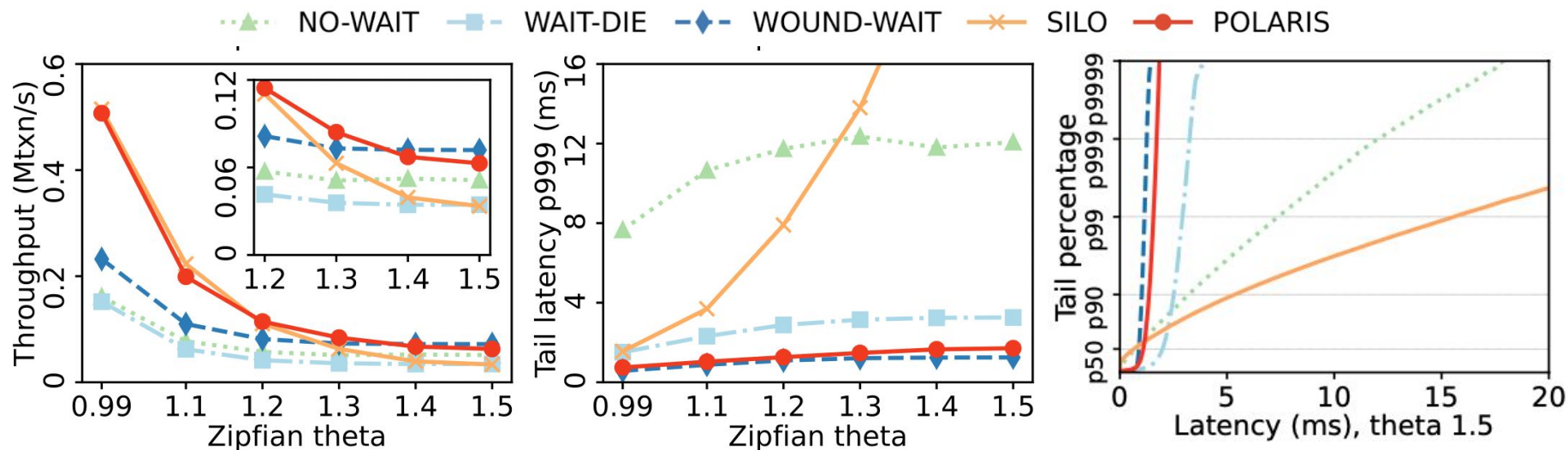
Evaluation: Varying Data Access Distribution



YCSB-A (Read 50%, Write 50%), 64 threads

At $\theta=1.5$, 91% of transaction in Polaris commits with non-zero priority

Evaluation: Varying Data Access Distribution



YCSB-A (Read 50%, Write 50%), 64 threads

At $\theta=1.5$, 91% of transaction in Polaris commits with non-zero priority

Polaris behaves similarly to 2PL with priority under high contention

More Experiments in Paper

- Priority sensitivity analysis
- User-define priority
- Read-only workload
- Comparison with batching

46:10

Chenhuo Yu, Wuh Chen, Hong, Xuan Chen, and Bingqiang Yu

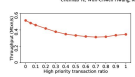


Fig. 2. Polar throughput with varying ratio of high priority transaction (PCSB-A, $r = 30\%$, $w = 30\%$, $f = 0\%$, 4 threads).

with a tunable parameter f of each transaction account is results in a 100-million-round table. We use two workloads: UCSB-A consists of 50 read and 50 write transactions, and UCSB-C is a read-only workload. TPC-C is a widely used standard benchmark. TPC-C contains 15 of user-initiated transaction short and medium size types of table with a tunable number of tables.

We use experiments on a CloudAI 400 machine (15, which has six times-core 16 GHz Intel Xeon Gold 6248 processors and 64 GB of UCSB-A read-only memory, running on Ubuntu 20.04). With hyperthreading enabled, the machine has 48 logical cores in total.

In our experiments, each database workload thread is pinned to a logical core. We used `reactor1` + `interleave` as all to allocate memory between NMMIO nodes in a round-robin manner. Whenever a transaction is aborted, it is re-executed after a random back-off time within 1 sec. We have tuned the system and found this back-off time 1600 good overall performance. The exception is Aita, which shows aborted transactions can the next batch and does not perform back-off. For Wal-Silo, Wal-Silo, and Aita, we do not allocate a new timestamp sequence number for re-execution, so the re-executed transactions will have higher priority than the newly created ones. We collected such transaction latency as the time elapsed between the start of the first execution and the end of its commit; the time spent to abort transaction is included in the latency.

5.2 Priority Sensitivity Analysis

We quantitatively evaluate the impact of introducing high-priority transactions with UCSB-A workload ($r = 0\%$, $w = 30\%$, $f = 30\%$) with 4 threads. In this experiment, every transaction with the highest priority (high priority) has the highest latency. In this experiment, the priority queue is uniformly random, and we say the ratio of high-priority transactions. The result is shown in Figure 2.

When there are only low-priority transactions, the result is shown in Figure 3. The throughput is 100 Kops. When there are 5% of high-priority transactions, the overall throughput drops to 45 Kops (45%), when there are 10% high-priority transactions, the throughput drops to 35 Kops (35%). The overall throughput results is low based on 100 Kops (100%) of high-priority transactions and high priority, making the rest of the transactions difficult to commit. The throughput goes up when every transaction is high priority (50 Kops). However, this is still lower than the case when every transaction is low priority because high-priority transactions cannot benefit from the lower-priority optimization – they frequently write TIDs, triggering cache coherence invalid, which is costly in the case of 64 threads.

Proc. ACM Manag. Data, Vol. 1, No. 1, Article 46. Publication date: May 2023.

46:10

Chenhuo Yu, Wuh Chen, Hong, Xuan Chen, and Bingqiang Yu

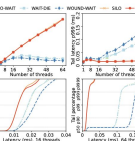


Fig. 3. Throughput and p99 latencies over a spectrum of thread numbers (TPC-C, $r = 0\%$, $w = 40\%$, $f = 0\%$). Latency distribution in the case of 0 and 40 threads (TPC-C, $r = workloads$).

priority. This can also be seen from the latency distribution graph where Silo and Polar drops at around p99. Wal-Silo has the lowest tail latency as it strictly follows the priority-based scheduling and timestamp. Wal-Silo also uses the timestamp but its lock scheduling mechanism is not friendly to the tail latency. Whenever a transaction commits and it is waiting for the lock back, Wal-Silo has to wait the time that the waiting transactions have higher priority than the lock holder, so it will get the lock in the longest waiting transactions and from other transactions waiting.

In terms of the median latency (Silos, Silos, and NoWait perform better than Wal-Silo and Wal-Silo, because the major contributors to the median latency is waiting instead of aborts.

Varying Number of Threads (Read-Only). Figure 4 demonstrates the performance of the concurrency control protocols with the read-only UCSB-C workload ($r = 0\%$). Since there is no abort, all transactions in Polar commit in the lowest priority. As a result, Polar and Silo have similar performance, and both scale well. In contrast, their 95% variants do not scale as lock acquisition requires a writer operation to start recovery, which will trigger expensive cache flushing.

Varying Data Access Distribution. We also compare five concurrency control protocols at 64 threads under a spectrum of user distribution workloads with UCSB-A. Figure 5 shows the cases from low contention to high, where a larger value of f implies a more skewed distribution. When the contention is low, the performance of all protocols is similar. As the contention increases, the contention is high, we found that Silo is much less resistant to the growing contention. All three 95% variants have similar throughput and tail latency. Polar and NoWait have the lowest latency under a dramatic throughput drop and tail latency increase. For Polar under $f = 1.5$, we observe the transaction of the p99 tail has been delayed 15 times and the one of the p99 tail is delayed 11 times.

Proc. ACM Manag. Data, Vol. 1, No. 1, Article 46. Publication date: May 2023.

46:10

Chenhuo Yu, Wuh Chen, Hong, Xuan Chen, and Bingqiang Yu

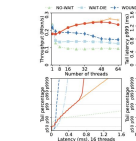


Fig. 4. Throughput and p99 latencies over a spectrum of thread numbers, latency distribution in the case of 0% and 40 threads (TPC-C, $r = workloads$).

inside the low-priority transactions to have the maximum priority at t , so that the user-specified priority will always be respected.

The latency distribution and overall throughput are shown in Figure 6. Both Polar and Silo and Polar deliver low latency (p99) high-priority transactions within 1.5 ms and thus have p99 latency lower than 1.5 ms. The low-priority transactions in Polar-95 have slightly higher tail latency (15 ms) than transactions in Silo (12 ms) and because they before them in the lock, but are additionally be delayed by high-priority transactions. With 100 unpaired priority enabled, Polar can even bound the tail latency of low-priority transactions within 1 ms. With a significant improvement for tail latency, Polar delivers throughput that is comparable to Silo and Polar-95 (only 8.5% and 1.5% 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 10 and 44 workloads.

Varying Number of Threads at High Contention. Figure 7 shows the performance with TPC-C workload at 44 threads. Polar could bound the p99 tail latency within 1.5 ms, while Silo's p99 tail latency is 1.5 ms.

The tail latency of Silo and Polar drops early on 16 threads. At 16 threads, Polar has 21% lower p99 latency than Silo. As the contention is increased, the gap is widened. Both Silo and Polar have 64% of transaction short 11 times or more. Polar starts to prioritize those transactions, so most of those are completed in short time. In contrast, only 50% of transactions have been delayed 12 times or more. In contrast, 53% of transactions in Silo have been delayed 12 times or more, the transaction of the p99 tail is delayed 13 times and the one of the p99 tail is delayed 11 times.

Proc. ACM Manag. Data, Vol. 1, No. 1, Article 46. Publication date: May 2023.

46:10

Chenhuo Yu, Wuh Chen, Hong, Xuan Chen, and Bingqiang Yu

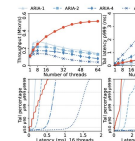


Fig. 5. Throughput and p99 latencies over a spectrum of thread numbers, latency distribution in the case of 0% and 40 threads (TPC-C, $r = workloads$).

the throughput of Silo and Polar drops at 16 threads. At 16 threads, 12% of transactions in Polar have non-zero priority at 64 threads, 16% of transactions have non-zero priority. The throughput of Polar is 9% lower than Silo's, but still significantly better than 95% variants.

Varying Number of Threads at Low Contention. Figure 8 demonstrates the cases where contention is low (TPC-C with 10 workloads). All protocol scale well with the number of threads, but Silo and Polar still deliver higher throughput and lower tail latency than 95% variants.

Proc. ACM Manag. Data, Vol. 1, No. 1, Article 46. Publication date: May 2023.

46:10

Chenhuo Yu, Wuh Chen, Hong, Xuan Chen, and Bingqiang Yu

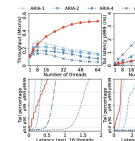


Fig. 6. Throughput and p99 latencies over a spectrum of thread numbers, latency distribution in the case of 0% and 40 threads (TPC-C, $r = workloads$).

the throughput of Silo and Polar drops at 16 threads. At 16 threads, 12% of transactions in Polar have non-zero priority at 64 threads, 16% of transactions have non-zero priority. The throughput of Polar is 9% lower than Silo's, but still significantly better than 95% variants.

Varying Number of Threads at Low Contention. Figure 8 demonstrates the cases where contention is low (TPC-C with 10 workloads). All protocol scale well with the number of threads, but Silo and Polar still deliver higher throughput and lower tail latency than 95% variants.

Proc. ACM Manag. Data, Vol. 1, No. 1, Article 46. Publication date: May 2023.

Conclusion

We present **Polaris**, an OCC protocol with priority support

Through a lightweight reservation mechanism, Polaris can

- preserve high-throughput advantage of OCC
- guarantee starvation-freedom
- provide lower tail latency

Thank You!