Mostly-Optimistic Concurrency Control
for Highly Contended Dynamic Workloads on 1000 cores

Tianzheng Wang, University of Toronto
Hideaki Kimura*, Hewlett Packard Labs

* Currently with Oracle
OLTP on modern & future hardware

**Multi-socket**

- Tens of cores

**HPE Superdome X**

- 16 sockets, 576 HW threads

**HPE The Machine**

- **CRAY XC** with Knights Landing

Very high parallelism

Need **lightweight** concurrency control (CC)
Modern Optimistic CC 101

1. Local R/W
2. Verify
3. Commit/abort

Transaction 1
Read set: \{A, B\}
Write set: \{A'\}

Transaction 2
Read set: \{A, B\}
Write set: \{B'\}

Time

- Read A
- Read B
- Write A'
- Commit?

- Read A
- Read B
- Write B'
- Commit

- Lock(A)
- A, B changed?
- Unlock(A)
- Abort

- Lock(B)
- A, B changed?
- B = B'
- Unlock(B)
Why does OCC work well?

Only lock writes

→ No shared memory writes for reads

Only lock at commit time

Sort writes before locking

→ No deadlock possible

Simplifies lock implementation
High contention: OCC doesn’t work
– 256 threads + 50 records YCSB, 10 ops/tx

Reads not protected

> 98% aborts

Abort ratio

# Writes (out of 10 in total)
Key idea: protect hot records with locks

Only lock hot records (keep OCC’s benefits)

Must lock as of the access

Need better locks

Could deadlock

MOCC: best(2PL) + best(OCC)

New sync. primitive
Must only lock hot records

- **Read-only**, 256 threads

![Graph showing throughput (MTPS) vs. number of CPU cores (number of sockets)]

Throughput (MTPS)

OCC 2PL

2 (1) 16 (2) 60 (4) 288 (16)

# CPU Cores (# Sockets)

Interconnect flooded
Less physical contention with approximate counter*

Real temperature

\[ \sim 2^{p_{\text{Temp}}} \]

Increment upon abort with prob. = \(1/2^{p_{\text{Temp}}}\)

Reduces cache line invalidation

Easy to tell really hot records/pages

Saves space

Lock(hot) re-introduces deadlocks

Hot records

Transaction 1
Read set: \{B\}
Write set: \{A'\}

Transaction 2
Read set: \{A\}
Write set: \{B'\}

Time
Write A
Read B

\textbf{Worse: no control over application footprint}
**Problem:** locks acquired out-of-order

i.e., Some locks acquired too early

What if T2 Unlock(B) now?
Canonical mode (CM): *All* locks acquired *in order*

Alphabetical, address...

Goal: keep transaction in canonical mode

Problems

1. *Restore* canonical mode
2. *Maintain* canonical mode on retry
Restore canonical mode

Transaction 1
Read set: \{A, C\}
Write set: {}
Locks held: \{A, C\}

Time

Read A
Read C
Read B

Unlock C
Lock(B)

Non-twophase

Breaking canonical mode

Commit

Verify A, B, C

Non-twophase locking + OCC verification
Retrospective lock list: A safety net upon retry

Keep the footprint and lock at retry

1st run
Read set: \{A, C\}
Write set: {}
Locks held: \{A, C\}

Retry
Read set: {}
Write set: {}
Locks held: {}
Retro. list: \{A, B, C\}

Read A
Read C
Read B
Abort
Retrospective lock list: A safety net upon retry

Keep the footprint and lock at retry

1st run
Read set: \{A, C\}
Write set: {}
Locks held: \{A, C\}

Retry
Read set: \{A, B, C\}
Write set: {}
Locks held: {}
Retro. list: \{A, B, C\}

Time

Read A
Read C
Read B
Abort

Read(A) – Check RLL, Lock A
Lock(C)?

Check RLL
Lock B
Lock C

No risk of deadlock
Native locking

- No centralized lock tables or blocking
- Synchronization primitive directly as database locks

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
<th>Use in MOCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read/-Write</td>
<td>Allows concurrent readers. Write is exclusive.</td>
<td>All cases</td>
</tr>
<tr>
<td>Unconditional</td>
<td>Indefinitely wait until acquisition.</td>
<td>Canonical mode.</td>
</tr>
<tr>
<td>Try</td>
<td>Instantaneously gives up. Does not leave qnode.</td>
<td>Non-canonical mode. Record access.</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>Leaves qnode for later check. Allows multiple requests in parallel.</td>
<td>Non-canonical mode. Record access and pre-commit (write set).</td>
</tr>
</tbody>
</table>

- MOCC queuing lock = MCS RW + MCS timeout
Evaluation

- HW: four machines of varying scale
  - Models:
    | Model          | EB840 | Z820  | DL580 | GryphonHawk |
    |----------------|-------|-------|-------|-------------|
    | Sockets        | 1     | 2     | 4     | 16          |
    | Cores (HT)     | 2 (4) | 16 (32)| 60 (120)| 288 (576)  |
    | Frequency      | 1.9 GHz | 3.4 GHz | 2.8 GHz | 2.5 GHz    |

- YCSB for high contention workloads
- 10 random RMWs, vary # of writes, 50 records
- More results/CC schemes in the paper
- TPC-C: few conflicts ➞ same as OCC
**MOCC keeps the best of OCC**

**Read-only YCSB**

- MOCC
- OCC
- PCC
- Dreadlock

YCSB Throughput (MTPS)

<table>
<thead>
<tr>
<th># CPU Cores (# Sockets)</th>
<th>2 (1)</th>
<th>16 (2)</th>
<th>60 (4)</th>
<th>288 (16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOCC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OCC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PCC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dreadlock</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 110MTPS
- 0.65MTPS
Keeps away the worst of OCC

Read-write YCSB

Throughput (MTPS)

# Writes (out of 10 in total)

MOCC OCC PCC Dreadlock

~50% abort
>98% abort

Too many deadlocks
## TPC-C results

- Aggregate of all transactions

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Throughput [MTPS+Stdev]</th>
<th>Abort Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOCC</td>
<td>16.9±0.13</td>
<td>0.12%</td>
</tr>
<tr>
<td>FOEDUS</td>
<td>16.9±0.14</td>
<td>0.12%</td>
</tr>
<tr>
<td>PCC</td>
<td>9.1±0.37</td>
<td>0.07%</td>
</tr>
<tr>
<td>ERMIA</td>
<td>3.9±0.4</td>
<td>0.01%</td>
</tr>
</tbody>
</table>

**Almost no overhead under low contention**
Robust CC needed for OLTP

Mostly-optimistic concurrency control

= best(2PL) + best(OCC)

Protect hot records with locks

1. Approx. counter for temperature
2. Non-twophase lock + retrospective lock list
3. MOCC queuing lock

Find out more in our paper and code repo

https://github.com/HewlettPackard/foedus_code