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

Tianzheng Wang, University of Toronto
Hideaki Kimura, Hewlett Packard Labs (currently with Oracle)
https://github.com/hewlettpackard/foedus_code

What? **High-contention workloads** are common, but not well supported under 1000 cores

Why? Optimistic concurrency control (OCC) **doesn’t protect reads**: extremely hard to commit

How? Hybrid approach: **lock hot records upon access** + OCC verification for serializability

---

OLTP on modern and future hardware

**HPE Superdome X**
16 sockets, 288 cores

**HPE The Machine**
1000 cores

Very high parallelism ➔ favor lightweight, optimistic concurrency control (OCC)

---

**Mostly-Optimistic Concurrency Control = best(2PL) + best(OCC)**

Key idea: **protect hot accesses with pessimistic locks**
- Hot records under contention: lock **upon** access
- Prevents clobbered reads ➔ more likely to succeed verification during pre-commit
- Cold records: same as in OCC

---

Know the real hot: approximate counter*

<table>
<thead>
<tr>
<th>Page</th>
<th>pTemp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

- Per-page (preferred) or per-record temperature field
- Real temperature \( \approx 2^{pTemp} \)
- Increment upon abort with probability \( 1/(2^{pTemp}) \)

- Larger \( pTemp \) ➔ harder to increment ➔ easy to tell “real” hot records
- Need not be accurate ➔ no concurrency control needed ➔ less cacheline invalidation


---

**Canonical mode and retrospective locking**

Reason for deadlock: **locks acquired out-of-order**

**def**: a transaction in **canonical mode** if all of its locks are acquired in order (so far)

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

Retry
Read set: \( \{ A, C, B \} \)
Write set: \{ \}
Locks held: \( \{ A, B, C \} \)

Ways to recover:
- Unlock \( C \)
- Lock(B, C)
- or try-lock(B)

**Bottom line**: verify upon commit

Retrospective locking: a safety net

---

**Challenges**:
- Accurately and cheaply detect “real” hot records
- Lock-upon-access can lead to deadlocks
- Need an efficient cancellable, reader-writer lock
  **Must not revive all the overheads of traditional 2PL**

---

**Efficient native locking**

- Desired properties
  - **Native locking**: synchronization primitives directly as database locks
  - **Decentralized**: co-locate locks with records
  - **Scalable, cancellable, reader-writer locks**
    - Cancel a request in case of possible deadlock
  - Solution: **MOCC queuing lock**
    = MCS Reader-Writer + MCS Timeout

- **Keeps OCC’s best and keeps away its worst**

**HW**: 288-core, 16-socket, Intel E7-8890, 12TB RAM

**Read-write TPS**: MOCC: 65k, OCC: 7k
Pure locking (PCC): too many deadlocks
Dreadlock: too many interthread communications

---

**Read-only**

MOCC matches OCC:
No overhead for low contention
Pure locking (PCC): too much physical contention

---

**Reason for deadlock**:
locks acquired out-of-order

**def**: a transaction in **canonical mode** if all of its locks are acquired in order (so far)

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

**Time**
Read A
Read C
Read B

**Ways to recover**:
- Unlock \( C \)
- Lock(B, C)
- or try-lock(B)

**Bottom line**: verify upon commit

**Retrospective locking**: a safety net

**Time**
Read A – Lock (A)
Read C – Lock (B, C)
Read B – already locked

---

**Key idea**: **protect hot accesses with pessimistic locks**

- Hot records under contention: lock **upon** access
- Prevents clobbered reads ➔ more likely to succeed verification during pre-commit
- Cold records: same as in OCC

---

Page dimensions: 1800.0x2880.0

[Image 278x1908 to 519x2031]
[Image 283x2051 to 461x2170]
[Image 91x2051 to 186x2184]