A Scalable Lock Manager for Multicores

Hyungsoo Jung  
NICTA

Hyuck Han  
Samsung Electronics

Alan Fekete  
University of Sydney

Gernot Heiser  
NICTA

Heon Y. Yeom  
Seoul National University

@University of Sydney
Moore's Law

“The number of transistors incorporated in a chip will approximately double every 24 months.”

--Gordon Moore, Intel co-founder
Moore’s Law

Microprocessor Transistor Counts 1971-2011 & Moore’s Law

2013: IBM’s System z processor 5.7GHz and with 2.75B transistors.
Moore’s Law goes Multicores

But what about the software?

Database Management Systems: our focus !!!!

Enterprise Software Systems (Not explored completely)

Operating System
“Linux is not scalable,
See [OSDI 2010, EuroSys2012, ASPLOS 2012]”

MULTICORE MACHINES
This research tries to solve ..

- Multi-core scalability problems of DBMS engines (running at SERIALIZABLE isolation) by eliminating latching overhead in a lock manager.
  - Keep overall architecture the same
  - Unlike larger redesigns proposed by Johnson et al. and Thomson et al.

- Now let’s see some background.
Latch protecting Lock table (MySQL)

Lock Acquire in Growing Phase

```c
mutex_enter(lock_table->mutex);
n_lock = lock_create();
n_lock->state = ACTIVE;
lock_insert(n_lock);
for all locks (lock) in hash_bucket
    if (lock is incompatible with n_lock)
        n_lock->state = WAIT;
    if (deadlock_check() == TRUE)
        abort Tx;
    break;
else
    continue;
end if
end for
mutex_exit(lock_table->mutex);
if (n_lock->state == WAIT)
    mutex_enter(Tx->mutex);
    Tx->state = WAIT;
    os_cond_wait(Tx->mutex);
    mutex_exit(Tx->mutex);
end if
```

Lock Release in Shrinking Phase

```c
mutex_enter(lock_table->mutex);
for all locks (lock1) in Tx
    lock_release(lock1);
    for all locks (lock2) following lock1
        if (lock2 doesn't have to wait)
            lock_grant(lock2);
            lock2->state = ACTIVE;
        end if
    end for
end for
mutex_exit(lock_table->mutex);
```

Lock Table Mutex (or Latch)
Lock vs. Latch: Database Lock

Users

Access via transactions

Concurrency

Data

Database Management Systems

Duration is long (S2PL)

Sleeping when locks conflict

Lock conflicts don’t cause the observed performance collapse !!!!
Lock vs. Latch: Latch

Concurrency control by latches

Threads

Access

Duration is usually very short
Spin-waiting on contention
This works fine as long as the duration is really short.

e.g., B+tree

In-memory Data Structures
Lock vs. Latch: High latch contention

In high contention:
1. Latch duration gets longer
2. Spin-waiting incurs the cache invalidation storm on multicores!!
3. This causes performance collapse !!!!
How bad is the performance collapse?

Throughput goes as we expected!!

MySQL (2PL)

Txns/min (million)

MPL

8 cores
4 cores
1 core
1 Socket
How bad is the performance collapse?

MySQL (2PL)

- 16 cores
- 32 cores

Performance collapses !!!

2 and 4 Sockets
What causes this collapse?

Let’s profile databases to peek a little bit deeper inside the system.

**Profiling:**
read-only queries under “SERIALIZABLE” isolation on 32 cores on 4 sockets

Latch contention is the cause!!!
Step back: why do we use latches ???

• Goal: mutual exclusion (ME) between threads

• Mutual Exclusion:
  – (1) prevents data race errors
  – (2) synchronizes update made inside critical section.

• Our intuition is:
  – If we could achieve two objectives with an alternative paradigm, then we can avoid using latches.
We propose

- a scalable lock manager with reduced latching.

- We achieved this by:
  - **Read-After-Write (RAW)** with memory barriers for fast synchronization
  - **Staged allocation and de-allocation** of locks for a lock hash table without dangling pointer dereferences
**RAW-inspired Implementation (Acquire)**

---

**Lock Acquire in Growing Phase**

A1: \[ n\_lock = \text{lock}\_create(); \]
A2: \[ n\_lock->\text{state} = \text{ACTIVE}; \]
A3: \[ \text{atomic\_lock\_insert}(n\_lock); \]
A4: \[ \text{for all locks (lock) in hash\_bucket} \]
A5: \[ \text{if (lock is incompatible with n\_lock)} \]
A6: \[ n\_lock->\text{state} = \text{WAIT}; \]
A7: \[ \text{atomic\_synchronize();} \]
A8: \[ \text{if (lock->\text{state}==\text{OBSOLETE})} \]
A9: \[ \text{\text{continue;}} \]
A10: \[ n\_lock->\text{state}=\text{ACTIVE}; \]
A11: \[ \text{atomic\_synchronize();} \]
A12: \[ \text{if (new\_deadlock() == TRUE)} \]
A13: \[ \text{abort Tx;} \]
A14: \[ \text{break;} \]
A15: \[ \text{end if} \]
A16: \[ \text{end for} \]

---

\[ \text{if (n\_lock->\text{state} == \text{WAIT})} \]
S4
\[ \text{mutex\_enter(Tx->mutex);} \]
\[ \text{atomic\_synchronize();} \]
S5
\[ \text{mutex\_exit(Tx->mutex);} \]

---

**Write -> Barrier -> Read**
Lock Release in Shrinking Phase

R1: for all locks (lock1) in Tx

R2: lock1->state = OBSOLETE;

R3: atomic_synchronize();

S6 <Write

R4: for all locks (lock2) that follow lock1

R5: mutex_enter(lock2->Tx->mutex);

R6: if (lock2->Tx->state==WAIT &&

R7: lock2 does not have to wait )

R8: lock2->Tx->state=ACTIVE;

R9: lock2->state=ACTIVE;

R10: atomic_synchronize();

S7 <Write

R11: os_cond_signal(lock2->Tx);

R12: end if

R13: mutex_exit(lock2->Tx->mutex);

R14: end for

R15: end for

<-Barrier
<-Read
<-Write
<-Barrier
<-Read
Staged allocation and de-allocation

Traditional Lock Manager

Transaction begins
Enter a lock hash table
Lock table latch is held
Exit a lock hash table
Transaction commits
-physical lock de-allocation-

Insert a lock into the lock table

TIME

The lifetime of a lock in legacy systems
Staged allocation and de-allocation

New Lock Manager

Transaction begins
Pre-allocate “lock” objects
Enter a lock hash table

Latch-free !!!!
Atomic lock insert

TIME

The lifetime of a lock in legacy systems
The lifetime of a lock in a new lock manager

Exit a lock hash table
Transaction commits
Logical lock release

A lock exists, but non-blocking

Garbage-collector
physical
lock de-allocation

NICTA Copyright 2011
From imagination to impact
Two important operations

• **Atomic lock insert**
  – Unique insert order must be ensured

• **Garage-collection**
  – No dangling pointer dereference !!!
Atomic lock insert

(1) old_tail = atomic_fetch_and_store(&tail, NewLock)

(2) old_tail -> next = NewLock
Garage-collection

Correctness: transactions started after the head is advanced can NEVER see “Lock A” since it is INVISIBLE to him.

(1) Logical release by changing the state of a lock A
(2) Advance the head pointer
(3) Garbage-collect “OBsolete” locks
The Architecture of New Lock Manager

Transaction Table

<table>
<thead>
<tr>
<th></th>
<th>Committed Tx</th>
<th>Active Tx</th>
<th>Committed Tx</th>
</tr>
</thead>
<tbody>
<tr>
<td>head₁</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>head₂</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Physical De-allocation</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Temporally Persistent Locks

Logical Lock Release

Pool of Free Locks

Lock Hash Table

lock index
Experimental Setup

• Databases
  – MySQL-5.6.10, Our system (only the lock manager has been rewritten); also but not for comparison: Wisconsin Shore-MT and commercial DBMS X

• Micro-benchmark
  – Read-only
    \[
    \text{SELECT } \text{sum(b\_int\_value)\*rand\_number FROM txbench-i} \\
    \text{WHERE } b\_int\_key > :id \text{ and } b\_int\_key \leq :id+S
    \]
  – Update
    \[
    \text{UPDATE txbench-((i+1)\%3) SET b\_value-k = rand\_str} \\
    \text{WHERE } b\_int\_key = :id1} \\
    \text{OR } b\_int\_key = :id2
    \]
Experimental Setup (cont.)

- Multicore machines

<table>
<thead>
<tr>
<th>Component</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processors</td>
<td>8-Core Intel Xeon CPU E7-8837</td>
</tr>
<tr>
<td>Processor Sockets</td>
<td>4 Sockets</td>
</tr>
<tr>
<td>Hardware Threads</td>
<td>32 (No HyperThreading Support)</td>
</tr>
<tr>
<td>Clock Speed</td>
<td>2.66 GHz</td>
</tr>
<tr>
<td>L1 D-Cache</td>
<td>32 KiB (per core)</td>
</tr>
<tr>
<td>L1 I-Cache</td>
<td>32 KiB (per core)</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>256 KiB (per core)</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>24 MiB (per socket)</td>
</tr>
<tr>
<td>Memory</td>
<td>128 GiB DDR3 1066 MHz</td>
</tr>
<tr>
<td>Network</td>
<td>Ethernet 1 Gbps</td>
</tr>
</tbody>
</table>

- Isolation: “SERIALIZABLE”
Performance Evaluation (throughput)

10S 100% read-only workload

- 32core
- 16core
- 8core
- 4core
- 1core

Our system

MySQL (2PL)
Performance Evaluation (throughput)

100S 100% read-only workload

MySQL (2PL)  Our system

Note Y-axes differ
Performance Evaluation (profiled)

MySQL (2PL)  
Our system

32 cores on 4 sockets with 10S

Idle  Kernel  Mutex  Database

32 cores on 4 sockets with 100S

NICTA Copyright 2011
Degradation is due to lock conflicts, not latch contention.
Conclusion

• We identified that latch contention in the lock manager is a major cause for the performance collapse problems in multicore environments.

• We proposed a scalable lock manager with reduced latching, and demonstrated the performance.

Thank You & Questions?