Phase Reconciliation for Contended In-Memory Transactions

Neha Narula, Cody Cutler, Eddie Kohler, Robert Morris
MIT CSAIL and Harvard
IncrTxn(k Key) {
   INCR(k, 1)
}

LikePageTxn(page Key, user Key) {
   INCR(page, 1)
   liked_pages := GET(user)
   PUT(user, liked_pages + page)
}

FriendTxn(u1 Key, u2 Key) {
   PUT(friend:u1:u2, 1)
   PUT(friend:u2:u1, 1)
}
IncrTxn(k Key) {
    INCR(k, 1)
}

LikePageTxn(page Key, user Key) {
    INCR(page, 1)
    liked_pages := GET(user)
    PUT(user, liked_pages + page)
}

FriendTxn(u1 Key, u2 Key) {
    PUT(friend:u1:u2, 1)
    PUT(friend:u2:u1, 1)
}
Problem

Applications experience write contention on popular data
If only Bradley's arm was longer. Best photo ever. #oscars

Oscars 2014: Ellen's #Selfie Wins Internet, Breaks Twitter

3,384,862 RETWEETS 2,024,010 FAVORITES

10:06 PM - 2 Mar 2014
Concurrent Control Enforces Serial Execution

Transactions on the same records execute one at a time
Throughput on a Contentious Transactional Workload
Throughput on a Contentious Transactional Workload

![Graph showing throughput (txns/sec) vs. cores for Doppel and OCC.](image)
Transactions on the same record can proceed in parallel on *per-core slices* and be *reconciled* later.

This is correct because INCR commutes.
Databases Must Support General Purpose Transactions

IncrTxn($k \text{ Key}$) {
  \text{INCR}($k$, \text{Must happen atomically})
}

IncrPutTxn($k1 \text{ Key}$, $k2 \text{ Key}$, $v \text{ Value}$) {
  \text{INCR}($k1$, 1)
  \text{PUT}(k2, v)
}

PutMaxTxn($k1 \text{ Key}$, $k2 \text{ Key}$) {
  \text{v1} := \text{GET}(k1)
  \text{v2} := \text{GET}(k2)
  \text{if} \ v1 > v2:
    \text{PUT}(k1, v2)
  \text{else:}
    \text{PUT}(k2, v1)
  \text{return } v1, v2
}
Challenge

Fast, general-purpose serializable transaction execution with per-core slices for contended records
Phase Reconciliation

- Database automatically detects contention to split a record among cores
- Database cycles through phases: split, reconciliation, and joined

Doppel, an in-memory transactional database
Contributions

Phase reconciliation

– Splittable operations
– Efficient detection and response to contention on individual records
– Reordering of split transactions and reads to reduce conflict
– Fast reconciliation of split values
Outline

1. Phase reconciliation
2. Operations
3. Detecting contention
4. Performance evaluation
The *split phase* transforms operations on contended records \((x)\) into operations on per-core slices \((x_0, x_1, x_2, x_3)\).
Transactions can operate on split and non-split records.
Rest of the records use OCC \((y, z)\).
OCC ensures serializability for the non-split parts of the transaction.
### split phase

<table>
<thead>
<tr>
<th>Core 0</th>
<th>INCR($x_0$, 1)</th>
<th>GET($x$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core 1</td>
<td>INCR($x_1$, 1) PUT($y$, 2)</td>
<td>INCR($x_1$, 1)</td>
</tr>
<tr>
<td>Core 2</td>
<td>INCR($x_2$, 1) PUT($z$, 1)</td>
<td>INCR($x_2$, 1)</td>
</tr>
<tr>
<td>Core 3</td>
<td>INCR($x_3$, 1) PUT($y$, 2)</td>
<td></td>
</tr>
</tbody>
</table>

- Split records have assigned operations for a given split phase
- Cannot correctly process a read of $x$ in the current state
- Stash transaction to execute after reconciliation
split phase

core 0
INCR(x₀,1)

core 1
INCR(x₁,1) PUT(y,2) INCR(x₁,1) INCR(x₁,1)

core 2
INCR(x₂,1) PUT(z,1) INCR(x₂,1)

core 3
INCR(x₃,1) PUT(y,2)

GET(x)

• All threads hear they should reconcile their per-core state
• Stop processing per-core writes
• Reconcile state to global store
• Wait until all threads have finished reconciliation
• Resume stashed read transactions in joined phase
Reconcile state to global store
Wait until all threads have finished reconciliation
Resume stashed read transactions in joined phase
- Process new transactions in joined phase using OCC
- No split data
Batching Amortizes the Cost of Reconciliation

- Wait to accumulate stashed transactions, batch for joined phase
- Amortize the cost of reconciliation over many transactions
- Reads would have conflicted; now they do not

```
INCR(x_0,1)
GET(x)

INCR(x_1,1) INCR(y,2)
INCR(x_1,1)

INCR(x_2,1) INCR(z,1)
GET(x)
INCR(x_2,1) INCR(z,1)

INCR(x_3,1) INCR(y,2)
GET(x)
```

```
Phase Reconciliation Summary

- Many contentious writes happen in parallel in split phases
- Reads and any other incompatible operations happen correctly in joined phases
Outline

1. Phase reconciliation
2. Operations
3. Detecting contention
4. Performance evaluation
Operation Model

Developers write transactions as stored procedures which are composed of operations on keys and values:

- Traditional key/value operations:
  - value GET\((k)\)
  - void PUT\((k, v)\)

- Operations on numeric values which modify the existing value:
  - void INCR\((k, n)\)
  - void MAX\((k, n)\)
  - void MULT\((k, n)\)

- Ordered PUT and insert to an ordered list:
  - void OPUT\((k, v, o)\)
  - void TOPK_INSERT\((k, v, o)\)

- Not splittable
- Splittable
MAX Can Be Efficiently Reconciled

- Each core keeps one piece of state $x_i$
- $O(#\text{cores})$ time to reconcile $x$
- Result is compatible with any order
What Operations Does Doppel Split?

Properties of operations that Doppel can split:

– Commutative
– Can be efficiently reconciled
– Single key
– Have no return value

However:

– Only one operation per record per split phase
Outline

1. Phase reconciliation
2. Operations
3. Detecting contention
4. Performance evaluation
Which Records Does Doppel Split?

- Database starts out with no split data
- Count conflicts on records
  - Make key split if \#conflicts > conflictThreshold
- Count stashes on records in the split phase
  - Move key back to non-split if \#stashes too high
Outline

1. Phase reconciliation
2. Operations
3. Detecting contention
4. Performance evaluation
Experimental Setup and Implementation

• All experiments run on an 80 core Intel server running 64 bit Linux 3.12 with 256GB of RAM
• Doppel implemented as a multithreaded Go server; one worker thread per core
• Transactions are procedures written in Go
• All data fits in memory; don’t measure RPC
• All graphs measure throughput in transactions/sec
Performance Evaluation

• How much does Doppel improve throughput on contentious write-only workloads?
• What kinds of read/write workloads benefit?
• Does Doppel improve throughput for a realistic application: RUBiS?
Doppel Executes Conflicting Workloads in Parallel

Throughput (millions txns/sec)

- Doppel
- OCC
- 2PL

20 cores, 1M 16 byte keys, transaction: INCR(x,1) all on same key
Doppel Outperforms OCC Even With Low Contention

Throughput (txns/sec)

5% of writes to contended key

20 cores, 1M 16 byte keys, transaction: INCR(x,1) on different keys
Contentious Workloads Scale Well

1M 16 byte keys, transaction: INCR(x,1) all writing same key
LIKE Benchmark

• Users liking pages on a social network
• 2 tables: users, pages
• Two transactions:
  – Increment page’s like count, insert user like of page
  – Read a page’s like count, read user’s last like
• 1M users, 1M pages, Zipfian distribution of page popularity

Doppel splits the page-like-counts for popular pages
But those counts are also read more often
Benefits Even When There Are Reads and Writes to the Same Popular Keys

Throughput (millions txns/sec)

Doppel

OCC

20 cores, transactions: 50% LIKE read, 50% LIKE write
Doppel Outperforms OCC For A Wide Range of Read/Write Mixes

Throughput (txns/sec)

% of transactions that read

Doppel does not split any data and performs the same as OCC!

More stashed read transactions

20 cores, transactions: LIKE read, LIKE write
RUBiS

• Auction application modeled after eBay
  – Users bid on auctions, comment, list new items, search
• 1M users and 33K auctions
• 7 tables, 17 transactions
• 85% read only transactions (RUBiS bidding mix)

• Two workloads:
  – **Uniform** distribution of bids
  – **Skewed** distribution of bids; a few auctions are very popular
StoreBid Transaction

StoreBidTxn(bidder, amount, item) {

  INCR(NumBidsKey(item),1)
  MAX(MaxBidKey(item), amount)
  OPUT(MaxBidderKey(item), bidder, amount)
  PUT(NewBidKey(), Bid{bidder, amount, item})
}

All commutative operations on potentially conflicting auction metadata

Inserting new bids is not likely to conflict
Doppel Improves Throughput on an Application Benchmark

Throughput (millions txns/sec)

Uniform

Skewed

8% StoreBid Transactions

3.2x throughput improvement

80 cores, 1M users 33K auctions, RUBiS bidding mix
Related Work

• Commutativity in distributed systems and concurrency control
  – [Weihl ’88]
  – CRDTs [Shapiro ’11]
  – RedBlue consistency [Li ’12]
  – Walter [Lloyd ’12]

• Optimistic concurrency control
  – [Kung ’81]
  – Silo [Tu ’13]

• Split counters in multicore OSes
Conclusion

Doppel:

• Achieves parallel performance when many transactions conflict by combining per-core data and concurrency control

• Performs comparably to OCC on uniform or read-heavy workloads while improving performance significantly on skewed workloads.

http://pdos.csail.mit.edu/doppel