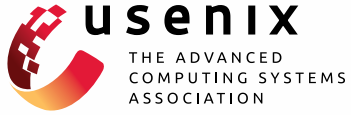
usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION



Phase Reconciliation for Contended In-Memory Transactions

Neha Narula and Cody Cutler, *MIT CSAIL;* Eddie Kohler, *Harvard University;*
Robert Morris, *MIT CSAIL*

https://www.usenix.org/conference/osdi14/technical-sessions/presentation/narula

This paper is included in the Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation.

October 6–8, 2014 • Broomfield, CO

978-1-931971-16-4

Open access to the Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

Phase Reconciliation for Contended In-Memory Transactions

Neha Narula, Cody Cutler, Eddie Kohler[†], and Robert Morris
MIT CSAIL and [†]Harvard University

Abstract

Multicore main-memory database performance can collapse when many transactions contend on the same data. Contending transactions are executed serially—either by locks or by optimistic concurrency control aborts—in order to ensure that they have serializable effects. This leaves many cores idle and performance poor. We introduce a new concurrency control technique, *phase reconciliation*, that solves this problem for many important workloads. Doppel, our phase reconciliation database, repeatedly cycles through *joined*, *split*, and *reconciliation phases*. Joined phases use traditional concurrency control and allow any transaction to execute. When workload contention causes unnecessary serial execution, Doppel switches to a split phase. There, updates to contended items modify *per-core* state, and thus proceed in parallel on different cores. Not all transactions can execute in a split phase; for example, all modifications to a contended item must commute. A reconciliation phase merges these per-core states into the global store, producing a complete database ready for joined-phase transactions. A key aspect of this design is determining which items to split, and which operations to allow on split items.

Phase reconciliation helps most when there are many updates to a few popular database records. Its throughput is up to 38× higher than conventional concurrency control protocols on microbenchmarks, and up to 3× on a larger application, at the cost of increased latency for some transactions.

1 Introduction

The key to good multicore performance and scalability is the elimination of serial execution. Cores should make progress in parallel whenever possible; the implementation should not force cores to wait for one another.

But serial execution sometimes appears to be an inherent feature of a problem. Most databases, for example, guarantee *serializable* results: the effect of executing a set of transactions in parallel should equal the effect of the same transactions executed in some serial order. This requires care when concurrent transactions conflict, which happens when one of them writes a record that the other either reads or writes. Database concurrency control protocols—mostly variants of two-phase locking (2PL) or optimistic concurrency control (OCC)—enforce serializability on conflicting transactions by executing them serially: one transaction will wait for the other, either by spinning on a lock (2PL) or by aborting and retrying (OCC).

Unfortunately, conflicts are common in some important real-world database workloads. For instance, consider an auction web site with skewed item popularity. As a popular item's auction time approaches, and users strive to win the auction, a large fraction of concurrent transactions might update the item's current highest bid. Modern multicore databases will execute these transactions serially, causing huge reductions in throughput.

We present *phase reconciliation*, a new concurrency control technique that can execute some highly conflicting workloads efficiently in parallel, while still guaranteeing serializability; and *Doppel*, a new in-memory database based on phase reconciliation.

Our basic technique is to split logical values across cores. We were inspired by efficient multicore counter designs, such as for packet counters, which partition a logical value into n counters, one per core. To increment the logical counter, a core updates its per-core value; to read it, a core *reconciles* these per-core values into one correct value by adding them together. This design is less contentious than a single global counter as long as writes greatly outnumber reads. But simple value splitting is too restrictive for general database use; splitting every item in the database would explode transaction overhead, and reconciling values on every read is costly. Instead, we dynamically shift data between split and reconciled states, based on observed contention.

A key design decision was to amortize the impact of value reconciliation over many transactions by executing different transactions in different *phases*. In joined phases, the database's structures are accessed using OCC. There are no per-core values and any transaction can execute (albeit with potentially high contention). In split phases, in contrast, updates are applied when possible to split per-core values rather than the global store. This greatly reduces contention on split data, but for correctness not all transactions may execute. Inappropriate uses of split data cause a transaction to block. Finally, short reconciliation phases reconcile these per-core values into the global store. When a reconciliation phase ends, blocked transactions resume and the next joined

phase begins. Thus, conflicting writes operate efficiently in the split phase, reads of frequently-updated data operate efficiently in the joined phase, and the system can achieve high overall performance even for challenging conflicting workloads.

The workloads that work best with phase reconciliation are ones with frequently-updated data items where contentious updates are commutative (they have the same overall effect regardless of order). Commutativity allows different cores to update their per-core values without coordination. Applicable situations include maintenance of the highest bids in the auction example, counts of votes on popular items, and maintenance of "top-*k*" lists for news aggregators such as Reddit [2].

The contributions of this work are the phase reconciliation technique and an implementation of phase reconciliation in Doppel. We show that phase reconciliation improves the overall throughput of various contentious workloads by up to 38× over OCC and 19× over 2PL, and has read throughput comparable to OCC. We port an auction website, RUBiS, to use Doppel and show Doppel improves bidding throughput with popular auctions by up to 3× over OCC.

2 Related Work

The idea of phase reconciliation is related to ideas in transactional memory, executing fast transactions on in-memory databases, and exploiting commutativity to reconcile divergent values, particularly in multicore operating systems and distributed systems.

Transactional memory. In designing Doppel we were inspired by some novel uses of transactional memory. Several designs have been proposed dividing transactions into phases, or rescheduling transactions to avoid aborts. Lev et al. propose the idea of using phases to support executing transactions both on best-effort hardware transactional memory and software transactional memory [21]. We leverage a similar idea to run transactions in different modes which are optimized for the types of transactions in those modes. Sync-Phase splits transactions up into computation and commit phases [25]. We do not split a transaction across phases, but assign transactions to different phases, based on the type of data they access and the operations they perform.

Transactional memory has been used directly for database transactions [20]. These transactions are often too large to use hardware transactional memory in a straightforward manner, so this work develops techniques to split transactions and apply them using timestamp ordering [8]. Still, spurious aborts are common in TM implementations of databases, since some memory writes to index data structures (which abort TM transactions) are irrelevant to database conflicts. One technique for addressing this problem on multicore architectures is rescheduling conflicting operations after detection to avoid continuous retries [7]. On contentious workloads with many conflicting writes, transactional memory would still be forced to abort or run the transactions one at a time. Our techniques would help in this situation.

Main-memory database concurrency control. Conventional wisdom is that when requests in the workload frequently conflict, they must serialize for correctness [16]. Given that, most related work has focused on improving scalability in the database engine for workloads which do not inherently conflict. Several databases try to leverage multiple cores by partitioning the data and running one partition per core. Systems like H-store/VoltDB [28, 29], HyPer [17], and Dora [23] all employ this technique. It is reasonable when the data is perfectly partitionable, but the overhead of cross-partition transactions in these systems is significant, and finding a good partitioning can be difficult. In our problem space (data contention) partitioning won't necessarily help; a single popular record with many writes wouldn't be able to utilize multiple cores. Hyder [9] uses a technique called meld [10], which lets individual servers or cores operate on a snapshot of the database and submit requests for commits to a central log. Each server processes the log and determines commit or abort decisions deterministically. Doppel also processes on local data copies but by restricting transaction execution to phases, can commit without global communication.

Multimed [24] also replicates data per core, but does so for read availability instead of write performance as in Doppel. The central write manager in Multimed is a bottleneck. Doppel partitions *local copies* of data amongst cores for writes and provides a way to re-merge the data for access by other cores.

Doppel uses optimistic concurrency control, of which there have been many variants [4, 8, 10, 18, 19, 30]. We use the algorithm in Silo [30], which is very effective at reducing contention in the commit protocol, but does not reduce contention caused by conflicting data writes. Larson et al. [19] explore optimistic and pessimistic multiversion concurrency control algorithms for main-memory databases, and this work is implemented in Microsoft's Hekaton [14]. This work presents ideas to eliminate contention due to locking and latches; we go further to address the problem of contention caused by conflicting writes to data. In future work we would like to implement a version of Doppel using pessimistic concurrency control. Doppel's split phase techniques are related to ideas which take advantage of commutativity and abstract data types in concurrency control [15, 31].

Multicore scalability. Linux developers have put a lot of effort into achieving parallel performance on multiprocessor systems. Doppel adopts ideas from the multicore scalability community, including the use of

commutativity to remove scalability bottlenecks [13]. OpLog [11] uses the idea of per-core data structures on contentious write workloads to increase parallelism, and Refcache [12] uses per-core counters, deltas, and epochs. This work tends to shift the performance burden from writes onto reads, which reconcile the per-core data structures whenever they execute. Doppel also shifts the burden onto reads, but phase reconciliation aims to reduce this performance burden in absolute terms by amortizing the effect of reconciliation over many transactions. Our contribution is making these ideas work in a larger transaction system.

Distributed consistency. Some work in distributed systems has explored the idea of using commutativity to reduce concurrency control, usually forgoing serializability. RedBlue consistency [22] uses the idea of blue, eventually consistent local operations which do not require coordination and red, consistent operations which do. Blue phase operations are analagous to Doppel's operations in the split phase. Walter [27] uses the idea of counting sets to avoid conflicts. Doppel could use any Conflict-Free Replicated Data Type (CRDT) [26] with its update operations in the split phase, but does not limit data items to specific operations outside the split phase.

One way of thinking about phase reconciliation is that by restricting operations only *during* phases but not *between* them, we support both scalable (per-core) implementations of commutative operations and efficient implementations of non-commutative operations on the same data items.

3 System model

We implemented phase reconciliation in a multicore, in-memory database called Doppel. Doppel has a low-level key/value store interface, and clients submit transactions in the form of procedures. Doppel provides serializable transactions.

Doppel transactions are *one-shot*: once begun, a transaction runs to completion without communication or disk I/O. Combined with an in-memory database, this means threads will not block due to user or disk stalls. One-shot transactions are used extensively in online transaction processing workloads [5, 28]. *Worker* threads, one per core, run transactions.

Our implementation of Doppel does not currently provide durability. Existing work suggests that asynchronous batched logging could be added to Doppel without becoming a bottleneck [19, 30].

Doppel records have typed values, and each type supports one or more operations. Transactions interact with the database via calls to operations. For example, the MAX(k,n) operation looks up an integer record with key k, and sets its value to the maximum of its current value and n. Some operations return values—GET(k), for example, returns the value of key k—and others do not; some operations modify the database and others do not. Each operation accesses exactly one database record. This isn't a functional restriction: users can build multi-record operations from single-record ones using transactions.

4 Split operations

A phase reconciliation database, such as Doppel, detects contended database records and, during split and reconciliation phases, marks them as *split*. For such records, operations that would normally contend can proceed in parallel.

1. At the beginning of each split phase, Doppel initializes *per-core slices* for each split record. There is one slice per contended record per core.
2. During the split phase, all operations on split records are applied to their per-core slices.
3. During the reconciliation phase, the per-core slices are merged back into the global store.

The combination of applying the operation to a slice and the merge step should have the same effect as the operation would normally. However, the code required to update a slice may be quite different from the code required to update a normal record.

To ensure good performance, per-core slices must be quick to initialize, and operations on slices must be fast. Most critically, the merging step, where per-core slices are merged into the global store, must take $O(J)$ time where J is the number of cores, instead of $O(N)$ time where N is the number of operations applied. This precludes some designs. For instance, one might think that split-phase execution could log updates to per-core slices, with the reconciliation phase applying the logged updates in time order; but this would cause those updates to execute serially, exactly the performance problem we want to avoid.

To ensure correctness, Doppel must ensure serializability. Executing transactions concurrently in a split phase must have the same effects as executing those same transactions in some serial order. Specifically, consider the set of transactions that commit in some split phase. Then there must exist a serial order of those transactions that satisfies:

1. The result of merging per-core slices with the global store is the same as if the transactions had executed, in the serial order, against the global store.
2. Every operation executed on a split record gets the same return value as if it had executed, in the serial order, against the global store.

3. Every operation executed on the global store gets the same return value as it would in the serial order.

An example of an operation that meets these requirements is MAX(k,n) on integer records, which assigns $v[k] \leftarrow \max\{v[k],n\}$ and returns nothing. When Doppel detects contention on MAX(k,n) operations for some key k, it marks k as split for MAX. When entering the next split phase, Doppel initializes per-core slices $c_j[k]$ with the global value $v[k]$. When a transaction on core j commits an operation MAX(k,n), Doppel sets $c_j[k] \leftarrow \max\{c_j[k],n\}$. Key k is temporarily reserved for MAX operations; a transaction that tries to execute another kind of operation on k will block until the following joined phase. When the split phase is over, Doppel merges the per-core slices by setting $v[k] \leftarrow \max_j c_j[k]$.

This implementation of MAX is efficient because per-core slices are fast to initialize, fast to update, and fast to merge. If many concurrent transactions call MAX(k,n) during a split phase, Doppel executes them in parallel on multiple cores with no coordination, getting good parallel speedup over the serial execution of conventional OCC or locking. Another reason for efficiency is that Doppel avoids expensive cache line transfers relating to contended data; these can make OCC and locking on many cores slower than serial execution on a single core.

Doppel's implementation is also correct. The main reason is that MAX *commutes* with itself: the effect of a set of MAX(k,n) operations on $v[k]$ is independent of their order. When operations do not commute, Doppel must enforce a serial order on those operations using global coordination. Per-core slices, which avoid coordination by design, thus work only for commutative operations. It's also important that Doppel restricts key k during the split phase to accept MAX operations only. This means that *all* split-phase operations on k commute, and it's safe to apply them to the per-core slices (even though the slices suppress information about the global execution order). Finally, MAX returns nothing, which is trivially the same as it would return when executed against the global store. We extend this argument in §5.6.

Doppel supports several splittable operations beyond MAX. We ensure these operations are both fast and correct by following some simple guidelines; a more complex implementation could relax these guidelines somewhat, as long as it still achieved the properties above.

1. Every splittable operation must commute with itself.

2. Every splittable operation must return nothing.

3. The system selects one splittable operation per split record per split phase. The selected operation can change between phases—for example, the operation for key k might be MIN in one split phase, and MAX in the next—but within a given phase, any operation but the selected operation causes the containing transaction to abort (and retry in the next joined phase).

4. The size of a per-core slice is independent of the number of operations that executed on that slice.

Doppel's current splittable operations are as follows.

- MAX(k,n) and MIN(k,n) replace k's integer value with the maximum/minimum of it and n.

- ADD(k,n) adds n to k's integer value.

- OPUT(k,o,x) is an operation on *ordered tuples*. An ordered tuple is a 3-tuple (o,j,x) where o, the *order*, is a number (or several numbers in lexicographic order); j is the ID of the core that wrote the tuple; and x is an arbitrary byte string. If k's current value is (o,j,x) and OPUT(k,o',x') is executed by core j', then k's value is replaced by (o',j',x') if $o' > o$, or if $o' = o$ and $j' > j$. Absent records are treated as having $o = -\infty$. The order and core ID components make OPUT commutative. Doppel also supports the usual PUT(k,x) operation for any type, but this doesn't commute and thus cannot be split.

- TOPKINSERT(k,o,x) is an operation on *top-K sets*. A top-K set is like a bounded set of ordered tuples: it contains at most K items, where each item is a 3-tuple (o,j,x) of order, core ID, and byte string. When core j' executes TOPKINSERT(k,o',x'), Doppel inserts the tuple (o',j',x') into the relevant top-K set. At most one tuple per order value is allowed: in case of duplicate order, the record with the highest core ID is chosen. If the top-K contains more than K tuples, the system then drops the tuple with the smallest order. Again, the order and core ID components make TOPKINSERT commutative.

More operations could easily be added (for instance, multiply).

5 Design

This section describes phase reconciliation in the context of Doppel. First, we describe the three phases of phase reconciliation. Second, we describe how updates are reconciled and how records are marked as either *split* or *reconciled*. Next, we describe how the system transitions between phases. We close with a brief argument that Doppel's implementation produces serializable results.

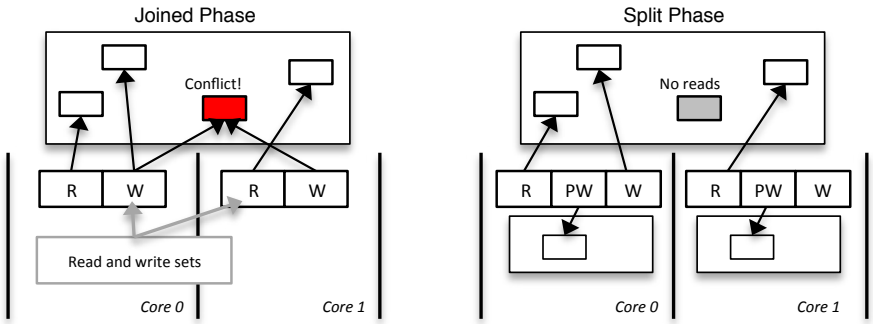


Figure 1: Concurrent transactions executing on different cores, shown in the joined phase and split phase. In the split phase certain data is split so that writes don't conflict.

```
Data: read set R, write set W
// Part 1
for record, operation in sorted(W) do
    lock(record);
commit-tid ← generate-tid()
// Part 2
for record, read-tid in R do
    if record.tid ≠ read-tid
        or (record.locked and record ∉ W)
    then abort();
// Part 3
for record, operation in W do
    apply(operation, record, commit-tid);
    unlock(record);
```

Figure 2: Doppel's joined phase commit protocol. Fences are elided.

5.1 Joined phase execution

A joined phase can execute any transaction. All records are reconciled—there is no notion of split data and there are no per-core slices—so the protocol treats all records the same.

Joined-phase execution could use any concurrency control protocol. However, some designs make more sense for overall performance than others. If all is working according to plan, the joined phase will have few conflicts; transactions that conflict should execute in the split phase. This is why Doppel's joined phase uses optimistic concurrency control (OCC), which performs better than locking when conflicts are rare.

The left side of Figure 1 shows two transactions executing on different cores in a joined phase, and Figure 2 shows the joined-phase commit protocol, which is based on that of Silo [30]. Records have transaction IDs (TIDs); these indicate the ID of the last transaction to write the non-split record, and help detect conflicts. A read set and a write set are maintained for each executing transaction. During execution, a transaction buffers its writes and records the TIDs for all values read or written in its read set. At commit time, the transaction locks the records in its write set (in a global order to prevent deadlock) and aborts if any are locked; obtains a TID; validates its read set, aborting if any values in the read set have changed since they were read, or are concurrently locked by other transactions; and finally writes the new values and TIDs to the shared store.

To avoid overhead and contention on TID assignment, our implementation assigns TIDs locally, using per-core information and the TIDs in the read set. The resulting commit protocol is serializable, but the TID order might diverge from the serial order.

Each transaction executes within a single phase. Any transaction that commits in a joined phase executed completely within that joined phase. Doppel thus cannot leave a joined phase for the following split phase until all current transactions commit or abort. As we see below, this requires coordination across threads.

5.2 Split phase execution

A split phase can execute in parallel some transactions that would normally contend. Accesses to reconciled data proceed much as in a joined phase, using OCC, but split-data operations execute on per-core slices. Split phases cannot execute all transactions, however. As we saw in §4, Doppel selects one operation per split record per split phase. A transaction that invokes an unselected operation on a split record will be aborted and *stashed* for restart during the next joined phase.

The right side of Figure 1 shows a split phase, with each transaction writing to per-core slices. For example, a transaction that executed an ADD$(k, 10)$ operation on a split numeric record might add 10 to the local core's slice for that record.

When a split phase transaction commits, Doppel uses the algorithm in Figure 3. It is similar to the algorithm in Figure 2 with a few important differences. The write set W contains only un-split data, while SW buffers updates to split data. The commit protocol applies the SW updates to the per-core slices. Since these slices are inherently

```
Data: read set R, reconciled write set W, split
      write set SW
// Part 1
for record, operation in sorted(W) do
    lock(record);
commit-tid ← generate-tid()
// Part 2
for record, read-tid in R do
    if record.tid ≠ read-tid
        or (record.locked and record ∉ W)
    then abort();
// Part 3
for record, operation in W do
    apply(operation, record, commit-tid);
    unlock(record);
for slice, operation in SW do
    slice-apply(operation, slice, commit-tid);
```

Figure 3: Doppel's split phase commit protocol.

```
Data: per-core slices S for core j
for record, operation, slice in S do
    lock(record);
    merge-apply(operation, slice, record);
    unlock(record);
S ← ∅
```

Figure 4: Doppel's per-core reconciliation phase protocol.

invisible to concurrently running transactions, there is no need to lock them or check their version numbers. (Any concurrent transaction must be running on another core, since each core runs transactions to completion one at a time.)

Any transaction that commits in a split phase executed completely within that split phase; Doppel does not enter the following joined phase until all of the split-phase transactions commit or abort.

5.3 Reconciliation phase execution

During a reconciliation phase, each core stops processing transactions and merges its per-core slices with the global store. For example, for a split record that used MAX, each core locks the global record, sets its value to the maximum of the previous value and its per-core slice, and unlocks the record. This involves serial processing of the per-core slices, but the expense is amortized over all the transactions that executed in the split phase. The per-core slices are then cleared and the database enters the next joined phase.

5.4 Phase transitions

Transitions between phases are managed by a coordinator thread and apply globally, across the entire database. To initiate a transition from a joined phase to the next split phase, the coordinator begins by publishing the phase change in a global variable. Workers check this variable between transactions; when they notice a change, they stop processing new transactions, acknowledge the change, and wait for permission to proceed. When all workers have acknowledged the change, the coordinator releases them, and workers start executing transactions in split mode. A similar process transitions from a split phase to the next reconciliation phase. When a split-phase worker notices a transition to the reconciliation phase, it stops processing transactions, merges its per-core slices with the global store, and then acknowledges the phase transition and waits for permission to proceed. Once all workers have acknowledged the change, the coordinator releases them to the next joined phase; each worker restarts any transactions it stashed in the split phase and starts accepting new transactions. It is safe for reconciliation to proceed in parallel with other cores' split-phase transactions since reconciliation modifies the global versions of split records, while split-phase transactions access per-core slices. No transactions will start joined phase operations on formerly split data until the coordinator has received acknowledgements from all workers for the phase transition, meaning they all finished their merge.

The Doppel coordinator usually starts a phase change every 20 milliseconds, but feedback mechanisms allow it to flexibly adjust to the workload. If, in a joined phase, no records appear contended—or they contend on unsplittable operations—the coordinator delays the next split phase. A worker can also delay a split phase by refusing to acknowledge it, and our workers delay acknowledging a split phase until they have committed or aborted all previously-stashed transactions. Finally, if, in a split phase, workers have to abort and stash too many transactions, the coordinator hurries the next joined phase.

5.5 Classification

Doppel automatically decides how records should be split. During joined execution, Doppel samples transactions' conflicting record accesses, and keeps a count of which records are most conflicted (are causing the most aborts) and by which operations. During the transition to the split phase, a coordinator thread examines these counts and marks the most conflicted records as split data for the next phase. Each core reads this list before the start of the next split phase in order to know which records are restricted. Doppel also samples which transactions are stashed due to incompatible operations on split data during the split phase, and uses this to consider whether to move a split record back to reconciled or change its assigned operation. Since split records in the split phase will not cause conficts, Doppel uses write sampling to estimate if a split record might still be con-

tended.

Doppel also supports manual data labeling ("this record should be split for this operation"), but we only use automatic detection in our experiments.

5.6 Serializability

This section sketches an argument that Doppel transactions are serializable.

Since transactions don't cross phases—any committed transaction executes entirely within a single phase—we can consider phases as units. Joined phases are clearly serializable since they just implement OCC, but to show that split and reconciliation phases are serializable, we must consider per-core slices. So consider a split–reconciliation phase pair that commits a set of transactions. We will show that there is a serial execution of those transactions against the global store—without using per-core slices—that produces the same output global store and the same operation results as the concurrent execution. Since the operations produce identical results, any conditional logic inside the transactions will make identical decisions in concurrent execution as in the serial order, so the transactions as a whole will behave identically.

Consider the transactions that commit in a split phase. These transactions can access both split records and non-split records. The non-split records use OCC, so the transactions are serializable with respect to non-split records. It remains to be shown that at least one serial order valid for non-split records is valid for split records as well. We show that, in fact, *any* serial order that works for non-split records also works for split records. Consider a split record r with currently selected operation Op. (We can consider one record at a time because each operation affects only one record.) Since it is splittable, Op commutes with itself and returns nothing. All committed split-phase operations on r must use Op, since Doppel aborts transactions that use non-selected operations. So these operations trivially return the same results in any serial order as in the concurrent execution: Op always returns nothing! Commutativity shows that the final value produced by applying the Op operations to the global store is the same regardless of the serial order chosen. This value also equals the outcome of applying the Ops to per-core slices and then merging those slices into the global store, though the reasons why depend on the operation. This concludes the argument.

6 Implementation

Doppel is implemented as a multithreaded server written in Go. Go made thread management and RPC easy, but caused problems with scaling to many cores, particularity in the Go runtime's scheduling and memory management. In our experiments we carefully managed memory allocation to avoid this contention at high core counts.

```
func max-merge(key Key) {
  val := local-get(key)
  g-val := global-get(key)
  global-set(key, max(g-val, val))
}

func oput-merge(key Key,
    phase TID) {
  order, coreid, val := local-get(key)
  // note that coreid == system.MyCoreID()
  g-order, g-coreid, g-val := global-get(key)
  if order > g-order ||
      (order == g-order && coreid > g-coreid) {
    global-set(key, (order, coreid, val))
  }
}
```

Figure 5: Doppel Max and OPut merge functions.

Doppel runs one worker thread per core, and one coordinator thread which is responsible for changing phases and synchronizing workers when progressing to a new phase. Doppel uses channels to synchronize phase changes and acknowledgements between the coordinator and workers. It briefly pauses processing transactions while moving between phases; we found that this affected throughput at high core counts. Another design could execute transactions that do not read or write past or future split data while the system is transitioning phases.

Workers read and write to a shared store, which is a set of key/value maps, using per-key locks. The maps are implemented as hash tables. Clients submit transactions written in Go to any worker, indicating the transaction to execute along with arguments. Doppel supports RPC from remote clients over TCP, but we do not measure this in §8. All workers have per-core slices for the split phases.

Developers write transactions in Go with no knowledge of reconciled data, split data, per-core slices, or phases. They access data using a key/value get and set interface or using the operations mentioned in §4.

7 Application Experience

We implemented two test applications: a feature of a social networking site where users can like pages, and a version of the RUBiS auction site benchmark.

The LIKE application simulates a set of users "liking" profile pages. Each update transaction writes a record inserting the user's like of a page, and then increments a per-page sum of likes. Each read transaction reads the user's last like and reads the total number of likes for some page. With a high level of skew, this application

```
func StoreBid(bidder, item, amt) (*Bid, TID) {
  bidkey := NewKey()
  bid := Bid {
    Item: item,
    Bidder: bidder,
    Price: amt,
  }
  Put(bidkey, bid)
  highest := Get(MaxBidKey(item))
  if amt > highest {
    Put(MaxBidKey(item), amt)
    Put(MaxBidderKey(item), bidder)
  }
  numBids := Get(NumBidsKey(item))
  Put(NumBidsKey(item), numBids+1)
  tid := Commit()  // applies writes or aborts
  return &bid, tid
}
```

Figure 6: Original RUBiS `StoreBid` transaction.

```
func StoreBid(bidder, item, amt) (&Bid, TID) {
  bidkey := NewKey()
  bid := Bid {
    Item: item,
    Bidder: bidder,
    Price: amt,
  }
  Put(bidkey, bid)
  Max(MaxBidKey(item), amt)
  OPut(MaxBidderKey(item),
      ([amt, GetTimestamp()], MyCoreID(), bidder))
  Add(NumBidsKey(item), 1)
  TopKInsert(BidsPerItemIndexKey(item),
             amt, bidkey)
  tid := Commit()  // applies writes or aborts
  return &bid, tid
}
```

Figure 7: Doppel `StoreBid` transaction.

explores the case where there are many users but only a few popular pages; thus the increments often conflict, but the inserts of individual records recording user likes do not. We expect the per-page sums for the popular page records to be marked as split data in the split phase, for use with the Add operation.

We used RUBiS [6], an auction website modeled after eBay, to evaluate Doppel on a realistic application. RUBiS users can register items for auction, place bids, make comments, and browse listings. RUBiS has 7 tables (users, items, categories, regions, bids, buy_now, and comments) and 26 interactions based on 17 database transactions. We ported a RUBiS implementation to Go for use with Doppel.

There are a few notable transactions in the RUBiS workload for which Doppel is particularly suited: `StoreBid`, which inserts a user's bid and updates auction metadata for an item, and `StoreComment`, which publishes a user's comment on an item and updates the rating for the auction owner. RUBiS materializes the `maxBid`, `maxBidder`, and `numBids` per auction, and a `userRating` per user based on comments on an owning user's auction items. We show RUBiS's `StoreBid` transaction in Figure 6.

If an auction is very popular, there is a greater chance two users are bidding or commenting on it at the same time, and that their transactions will issue conflicting writes. At first glance it might not seem like Doppel could help with the `StoreBid` transaction; the auction metadata is contended and could potentially be split, but each `StoreBid` transaction requires reading the current bid to see if it should be updated, and reading the current number of bids to add one. Recall that split data cannot be read during a split phase, so as written in Figure 6 the transaction would have to execute in a joined phase, and would not benefit from local per-core operations.

But note that the `StoreBid` transaction does not *return* the current winner, value of the highest bid, or number of bids to the caller, and the only reason it needs to *read* those values is to perform commutative Max and Add operations. Figure 7 shows the Doppel version of the transaction that exploits these observations. The new version uses the maximum bid in OPut to choose the correct core's `maxBidder` value (the logic here says the highest bid should determine the value of that key). This changes the semantics of `StoreBid` slightly. In the original `StoreBid` if two concurrent transactions bid the same highest value for an auction, the first to commit is the one that wins. In Figure 7, if two concurrent transactions bid the same highest value for an auction at the same coarse-grained timestamp, the one with the highest core ID will win. Doppel can execute Figure 7 in the split phase.

Using the *top-K set* record type, Doppel can support inserts to contended lists. The original RUBiS benchmark does not specify indexes, but we use top-K sets to make browsing queries faster. We modify `StoreItem` to insert new items into top-K set indexes on `category` and `region`, and we modify `StoreBid` to insert new bids on an item into a top-K set index per item, `bidsPerItemIndex`. `SearchItemsByCategory`, `SearchItemsByRegion`, and `ViewBidHistory` read from these records. Finally, we modify `StoreComment` to use Add on the `userRating`.

These examples show how Doppel's commutative operations allow seemingly conflicting transactions to be re-cast in a way that allows concurrent execution. This

pattern apppears in many other Web applications. For example, Reddit [2] also materializes vote counts, comment counts, and links per subreddit [3]. Twitter [1] materializes follower/following counts and ordered lists of tweets for users' timelines.

8 Evaluation

This section presents measurements of Doppel's performance, supporting the following hypotheses:

- Doppel increases throughput for transactions with conflicting writes to split data (§8.2).
- Doppel can cope with changes in which records are contended (§8.3).
- Doppel makes good decisions about which records to split when key popularity follows a smooth distribution (§8.4).
- Doppel can help workloads with a mix of read and write transactions on split data (§8.5).
- Doppel transactions which read split data have high latency (§8.6).
- Doppel increases throughput for a realistic application (§8.8).

8.1 Setup

All experiments are executed on an 80-core Intel machine with 8 2.4Ghz 10-core Intel chips and 256 GB of RAM, running 64-bit Linux 3.12.9. In the scalability experiments, after the first socket, we add cores an entire socket at a time. We run most fixed-core experiments on 20 cores.

The worker thread on each core both generates transactions as if it were a client, and executes those transactions. If a transaction aborts, the thread saves the transaction to try at a later time, chosen with exponential backoff, and generates a new transaction. backoff, and generates a new transaction. Throughput is measured as the total number of transactions completed divided by total running time; at some point we stop generating new transactions and then measure total running time as the latest time that any existing transaction completes (ignoring saved transactions). Each point is the mean of three consecutive 20-second runs, with error bars showing the min and max.

The Doppel coordinator changes the phase every 20 milliseconds. Doppel uses the technique described in §5.5 to determine which data to split. The benchmarks omit many costs associated with a real database; for example we pre-allocate all the records and do not incur any costs related to network, RPC, or disk.

In most experiments we measure phase reconciliation (Doppel), optimistic concurrency control (OCC), and two-phase locking (2PL). Doppel and OCC transactions abort and later retry when they see a locked item; 2PL uses Go's read-write mutexes. Both OCC and 2PL are implemented in the same framework as Doppel.

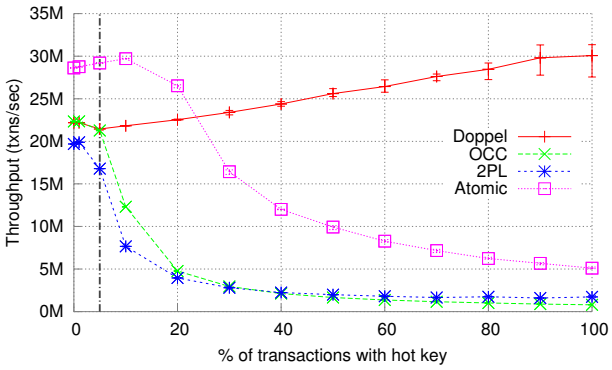


Figure 8: Total throughput for INCR1 as a function of the percentage of transactions that increment the single hot key. 20 cores. The vertical line indicates when Doppel starts splitting the hot key.

8.2 Parallelism versus Conflicts

This section shows that Doppel improves performance on a workload with many conflicting writes, using the following microbenchmark:

INCR1 microbenchmark. There are 1M 16-byte keys, and each transaction increments the value of a single key. There is a single popular key and we vary the percentage of transactions which increment that key; each other transaction randomly chooses from the not-popular keys.

This experiment compares Doppel with OCC, 2PL, and a system called Atomic. Doppel without split keys and OCC read the value of a key, compute the new value, and try to lock the key and validate that it hasn't changed since it was first read. If the key is locked or its version has changed, both abort the transaction and save it to try again later. 2PL waits for a write lock on the key, reads it, and then writes the new value. 2PL never aborts. Atomic uses an atomic increment instruction with no other concurrency control. Atomic represents an upper bound for locking schemes.

Figure 8 shows the throughputs of these schemes with INCR1 as a function of the percentage of transactions that write the single hot key.

At the extreme left of Figure 8, when there is little conflict, Doppel does not split the hot key, causing it to behave and perform similarly to OCC. With few conflicts, all of the schemes benefit from the 20 cores available.

As one moves to the right in Figure 8, OCC, 2PL, and Atomic provide decreasing total throughput. The high-level reason is that they must execute operations on the hot key serially, on only one core at a time. Thus their throughputs ultimately drop by roughly a factor of 20, as they move from exploiting 20 cores to doing useful

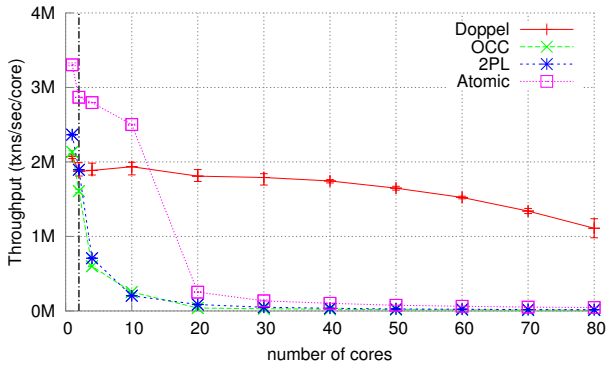


Figure 9: Throughput per core for INCR1 when all transactions increment a single hot key. The y-axis shows per-core throughput, so perfect scalability would result in a horizontal line.

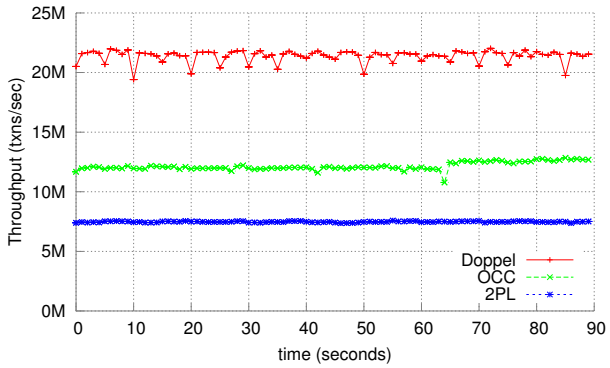


Figure 10: Throughput over time on INCR1 when 10% of transactions increment a hot key, and that hot key changes every 5 seconds.

work on only one core. The differences in throughput among the three schemes stem from differences in concurrency control efficiency: Atomic uses the hardware locking provided by the cache coherence and interlocked instruction machinery; 2PL uses Go mutexes which yield the CPU; while OCC saves and re-starts aborted transactions. The drop-off starts at an x value of about 5%; this is roughly the point at which the probability of more than one of the 20 cores using the hot item starts to be significant.

Doppel has the highest throughput for most of Figure 8 because once it splits the key, it continues to get parallel speedup from the 20 cores as more transactions use the hot key. Towards the left in Figure 8, Doppel obtains parallel speedup from operations on different keys; towards the right, from split operations on the one hot key. The vertical line indicates where Doppel starts splitting the hot key. Doppel throughput gradually increases as a smaller fraction of operations apply to non-popular keys, and thus a smaller fraction incur the DRAM latency required to fetch such keys from memory. When 100% of transactions increment the one hot key, Doppel performs 6.2× better than Atomic, 19× better than 2PL, and 38× better than OCC.

We also ran the INCR1 benchmark on Silo to compare Doppel's performance to an existing system. Silo has lower performance than our OCC implementation at all points in Figure 8, in part because it implements more features. When the transactions choose keys uniformly, Silo finishes 11.8M transactions per second on 20 cores. Its performance drops to 102K transactions per second when 100% of transactions write the hot key.

To illustrate the part of Doppel's advantage that is due to parallel speedup, Figure 9 shows multi-core scaling when all transactions increment the same key. The y-axis shows transactions/sec/core, so perfect scalability (perfect parallel speedup) would result in a horizontal line. Doppel falls short of perfect speedup, but nevertheless yields significant additional throughput for each core added. The lines for the other schemes are close to $1/x$ (additional cores add nothing to the total throughput), consistent with essentially serial execution. The Doppel line decreases because phase changes take longer with more cores; phase change must wait for all cores to finish their current transaction.

In summary, Figure 8 shows that when even a small fraction of transactions write the same key, Doppel can help performance. It does so by parallelizing update operations on the popular key.

8.3 Changing Workloads

Data popularity may change over time. Figure 10 shows the throughput over time for the INCR1 benchmark with 10% of transactions writing the hot key, with the identity of the one hot key changing every 5 seconds. Doppel throughput drops every time the popular key changes and a new key starts gathering conflicts. Once Doppel has measured enough conflict on the new popular key, it marks it as split. The adverse effect on Doppel's throughput is small since it adjusts quickly to each change.

8.4 Deciding What to Split

Doppel must decide whether to split each key. At the extremes, the decision is easy: splitting a key that causes few aborts is not worth the overhead, while splitting a key that causes many aborts may greatly increase parallelism. Section 8.2 explored this spectrum for a single popular key. This section explores a harder set of situations, ones in which there is a smooth falloff in the distribution of key popularity. That is, there is no clear distinction between hot keys and non-hot keys. The main question is whether Doppel chooses the right number (if any) of most-popular keys to split.

This experiment uses a Zipfian distribution of popularity, in which the kth most popular item is accessed in

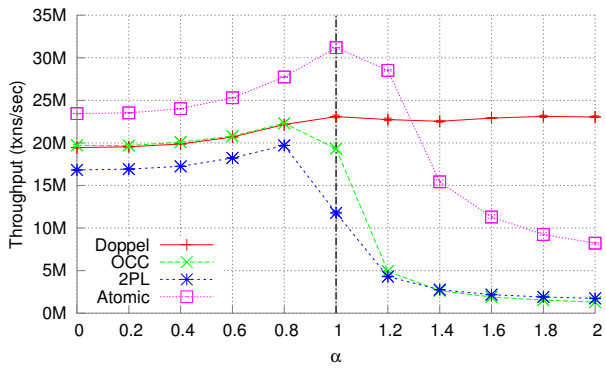


Figure 11: Total throughput for INCRZ as a function of α (the Zipfian distribution parameter). The skewness of the popularity distribution increases to the right. 20 cores. The vertical line indicates when Doppel starts splitting keys.

α	1st	2nd	10th	100th
0.0	.0001%	.0001%	.0001%	.0001%
0.2	.0013%	.0011%	.0008%	.0005%
0.4	.0151%	.0114%	.0060%	.0024%
0.6	.1597%	.1054%	.0401%	.0101%
0.8	1.337%	.7678%	.2119%	.0336%
1.0	6.953%	3.476%	.6951%	.0695%
1.2	18.95%	8.250%	1.196%	.0755%
1.4	32.30%	12.24%	1.286%	.0512%
1.6	43.76%	14.43%	1.099%	.0276%
1.8	53.13%	15.26%	.8420%	.0133%
2.0	60.80%	15.20%	.6079%	.0061%

Table 1: The percentage of writes to the first, second, 10th, and 100th most popular keys in Zipfian distributions for different values of α, 1M keys.

α	# Moved	% Reqs
< 1	0	0.0
1.0	2	10.5
1.2	4	35.9
1.4	4	56.1
1.6	4	70.5
1.8	4	80.1
2.0	3	82.7

Table 2: The number of keys Doppel moves for different values of α in the INCRZ benchmark.

proportion to $1/k^{\alpha}$. We vary α to explore different skews in the popularity distribution, using INCRZ:

INCRZ microbenchmark. There are 1M 16-byte keys. Each transaction increments the value of one key, chosen with a Zipfian distribution of popularity.

Figure 11 shows total throughput as a function of α. At the far left of the graph, key access is uniform. Atomic performs better than Doppel and OCC, and both better than 2PL, for the same reasons that govern the left-hand extreme of Figure 8.

As the skew in key popularity grows—for α values up to about 0.8—all schemes provide increasing throughput. The reason is that they all enjoy better cache locality as a set of popular keys emerge. Doppel does not split any keys in this region, and hence provides throughput similar to that of OCC.

Figure 11 shows that Doppel starts to display an advantage once α is greater than 0.8, because it starts splitting. These larger α values cause a significant fraction of transactions to involve the most popular few keys; Table 1 shows some example popularities. Table 2 shows how many keys Doppel splits for each α. As α increases to 2.0, Doppel splits the 2nd, 3rd, and 4th-most popular keys as well, since a significant fraction of the transactions modify them. Though the graph doesn't show this region, with even larger α values Doppel would return to splitting just one key.

In summary, for this workload Doppel does a good job of identifying which and how many keys are worth splitting, despite the gradual transition from popular to unpopular keys.

8.5 Mixed Workloads

This section shows how Doppel behaves when workloads both read and write popular keys. The best situation for Doppel is when there are lots of update operations to the contended key, and no other operations. If there are other operations on a split key, such as reads, Doppel's phases essentially batch writes into the split phases, and reads into the joined phases; this segregation and batching increases parallelism, but incurs the expense of stashing the read transactions during the split phase. In addition, the presence of the non-update operations makes it less clear to Doppel's algorithms whether it is a good idea to split the hot key. To evaluate Doppel's performance on a more challenging, but still understandable, workload, we use the LIKE benchmark from §7 that simulates users "liking" pages on a social networking site.

LIKE. The database contains a row for each user and a row for each page. Each transaction involves a user and a page. The user is always chosen uniformly at random. A write transaction chooses a page from a Zipfian distribution, increments the page's count of likes, and updates the user's row; the user's row is rarely contended, but the page's count might be. A read transaction chooses a page using the same Zipfian distribution, and reads the page's count and the user's row. There are 1M users and 1M pages, and unless specified otherwise the transaction mix is 50% reads and 50% writes.

Figure 12 shows throughput for Doppel, OCC, and 2PL with LIKE on 20 cores as a function of the fraction of transactions that write, with $\alpha = 1.4$. This setup causes the most popular page key to be used in 32% of transactions.

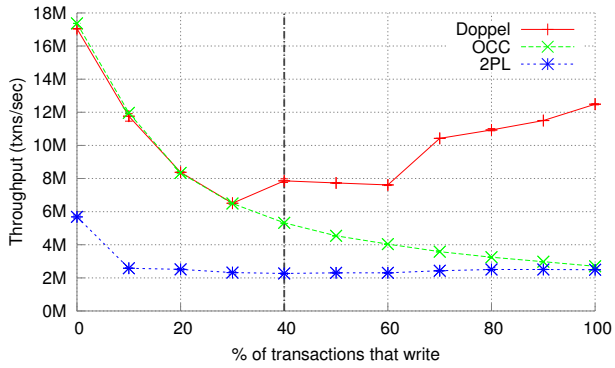


Figure 12: Throughput of the LIKE benchmark with 20 cores as a function of the fraction of transactions that write, $\alpha = 1.4$.

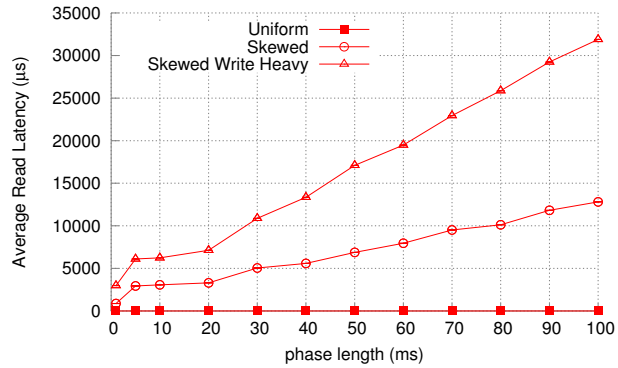


Figure 13: Average read transaction latencies in Doppel with the LIKE benchmark, varying phase length. A uniform workload, a skewed workload with 50% reads and 50% writes, and a skewed workload with 10% reads and 90% writes. 20 cores.

We would expect OCC to perform the best on a read-mostly workload, which it does. Until 30% writes Doppel does not split, and as a result performs about the same as OCC.

Doppel starts splitting data when there are 30% write transactions. This situation is tricky for Doppel because the split keys are read even more than they are written, so many read transactions have to be stashed. Figure 12 shows that Doppel nevertheless gets the highest throughput for all subsequent write percentages.

This example shows that Doppel's batching of transactions into phases allows it to extract parallel performance from contended writes even when there are many reads to the contended data.

8.6 Latency

Doppel stashes transactions which read split data in the split phase. This increases latency, because such transactions have to wait up to 20 milliseconds for the next joined phase. We use the LIKE benchmark to explore latency on two workloads (uniform popularity and skewed popularity with Zipf parameter $\alpha = 1.4$), separating latencies for read-only transactions and transactions that write. To measure latency, we measure the difference between the time each transaction is first submitted and when it commits. The workload is half read and half write transactions.

Table 3 shows the results. Doppel and OCC perform similarly with the uniform workload because Doppel does not split any data. In the skewed workload Doppel's write latency is the lowest because it splits the four most popular page records, so that write transactions that update those records do not need to wait for serial access to the data. Doppel's read latencies are high because reads of hot data during split mode have to wait up to 20 milliseconds for the next joined phase. This delay is the price Doppel pays for achieving almost twice the throughput of OCC.

8.7 Phase Length

When a transaction tries to read split data during a split phase, its expected latency is determined by the phase length; a shorter phase length results in less latency, but potentially lowered throughput. Figures 13 and 14 show how phase length affects read latency and throughput on three LIKE workloads. "Uniform" uses uniform key popularity and has 50% read transactions; nothing is split. "Skewed" has Zipfian popularity with $\alpha = 1.4$ and 50% read transactions; once the phase length is > 2ms, which is long enough to accumulate conflicts, Doppel moves either 4 or 5 keys to split data. "Skewed Write Heavy" has Zipfian popularity with $\alpha = 1.4$ and 10% read transactions; Doppel moves 20 keys to split data.

Figure 13 shows that the phase length directly determines the latency of transactions that read hot data and have to be stashed. Shorter phases are better for latency, but too short reduces throughput. The throughputs are low to the extreme left in Figure 14 because phase change takes about half a millisecond (waiting for all cores to finish split phase), so phase change overhead dominates throughput at very short phase lengths. For these workloads, the measurements suggest that the smallest phase length consistent with good throughput is five milliseconds.

8.8 RUBiS

Do Doppel's techniques help in a complete application? We measure RUBiS [6], an auction Web site implementation, to answer this question.

Section 7 describes our RUBiS port to Doppel. We modify six transactions to use Doppel operations; `StoreBid`, `StoreComment`, and `StoreItem` to use MAX, ADD, OPUT, and TOPKINSERT, and `SearchItemsByCategory`, `SearchItemsByRegion`, and `ViewBidHistory` to read from *top-K set* records as indexes. This means Doppel can potentially mark

	Uniform workload			Skewed workload		
	Mean latency	99% latency	Txn/s	Mean latency	99% latency	Txn/s
Doppel	1µs R / 1µs W	1µs R / 2µs W	11.8M	1262µs R / 4µs W	20804µs R / 2µs W	10.3M
OCC	1µs R / 1µs W	1µs R / 2µs W	11.9M	26µs R / 1069µs W	22µs R / 1229µs W	5.6M
2PL	1µs R / 1µs W	2µs R / 2µs W	9.5M	1µs R / 8µs W	3µs R / 215µs W	3.7M

Table 3: Average and 99% read and write latencies for Doppel, OCC, and 2PL on two LIKE workloads: a uniform workload and a skewed workload with $\alpha = 1.4$. Times are in microseconds. OCC never finishes 156 read transactions and 8871 write transactions in the skewed workload. 20 cores.

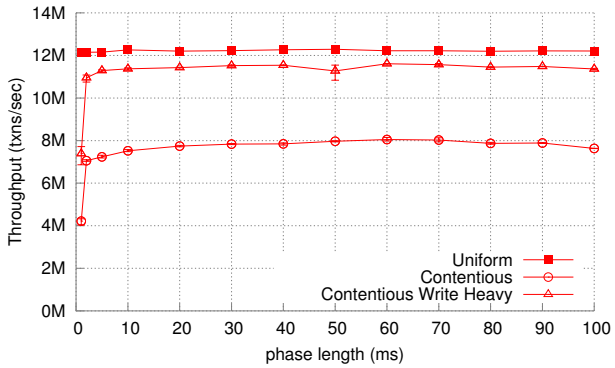


Figure 14: Throughput in Doppel with the LIKE benchmark, varying phase length. A uniform workload, a skewed workload with 50% reads and 50% writes, and a skewed workload with 10% reads and 90% writes. 20 cores.

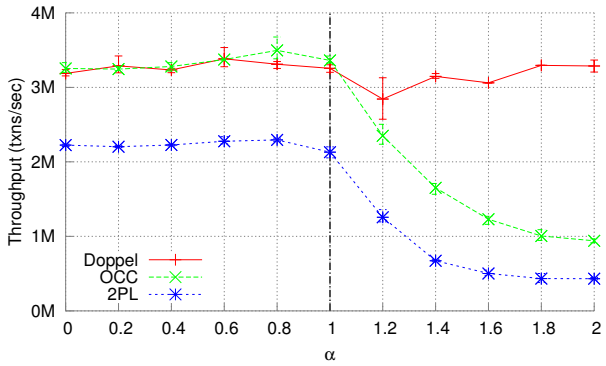


Figure 15: The RUBiS-C benchmark, varying α on the x-axis. The skewness of the popularity distribution increases to the right. 20 cores.

	RUBiS-B	RUBiS-C
Doppel	3.4	3.3
OCC	3.5	1.1
2PL	2.2	0.5

Table 4: The throughput of Doppel, OCC, and 2PL on RUBiS-B and on RUBiS-C with Zipfian parameter $\alpha = 1.8$, in millions of transactions per second. 20 cores.

auction metadata as split data. The implementation includes only the database transactions; there are no web servers or browsers.

We measured the throughput of two RUBiS workloads. One is the Bidding workload specified in the RUBiS benchmark, which consists of 15% read-write transactions and 85% read-only transactions; this ends up producing 7% total writes and 93% total reads. We call this RUBiS-B. In RUBiS-B most users are browsing listings and viewing items without placing a bid. There are 1M users bidding on 33K auctions, and access is uniform, so when bidding, most users are doing so on different auctions. This workload has few conflicts and is read-heavy.

We also created a higher-contention workload called RUBiS-C. 50% of its transactions are bids on items chosen with a Zipfian distribution and varying α. This approximates very popular auctions nearing their close. The workload executes non-bid transactions in correspondingly reduced proportions.

Table 4 shows how Doppel's throughput compares to OCC and 2PL. The RUBiS-C column uses a somewhat arbitrary $\alpha = 1.8$. As expected, Doppel provides no advantage on uniform workloads, but is significantly faster than OCC and 2PL when updates are applied with skewed record popularity.

Figure 15 explores the relationship between RUBiS-C record popularity skew and Doppel's ability to beat OCC and 2PL. Doppel gets close to the same throughput up to $\alpha = 1$. Afterwards, Doppel gets higher performance than OCC. When $\alpha = 1.8$ Doppel gets approximately 3× the performance of OCC and 6× the performance of 2PL.

Doppel's techniques make the most difference for the `StoreBid` transaction, shown in Figures 6 and 7. Doppel marks the number of bids, max bid, max bidder, and the list of bids per item of popular products as split data. It's important that the programmer wrote the transaction in a way that Doppel can split all of these data items; if the update for any one of the items had been programmed in a non-splittable way (e.g., with explicit read and write operations) Doppel would execute the transactions serially and get far less parallel speedup.

In Figure 15 with $\alpha = 1.8$, OCC spends roughly 67% of its time running `StoreBid`; much of this time is consumed by retrying aborted transactions. Doppel eliminates almost all of this 67% by running the transactions in parallel, which is why Doppel gets three times as much throughput as OCC with $\alpha = 1.8$.

These RUBiS measurements show that Doppel is able to parallelize substantial transactions with updates to multiple records and, skew permitting, significantly outperform OCC.

9 Conclusion

Doppel is an in-memory transactional database which uses phase reconciliation to increase throughput. The key idea is to execute certain types of conflicting operations on local per-core data, in parallel, and to reconcile the per-core states periodically. On workloads with many writes to a small number of popular records, Doppel can increase throughput by a factor related to the number of available cores.

Acknowledgments

We thank the anonymous reviewers, and our shepherd Allen Clement, for their helpful feedback. This research was supported by NSF awards 1065114, 1302359, 1301934, 0964106, 0915164, by Quanta, and by Google.

References

[1] Cassandra @ Twitter: An interview with Ryan King. `http://nosql.mypopescu.com/post/407159447/cassandra-twitter-an-interview-with-ryan-king`.

[2] Reddit. `http://reddit.com`.

[3] Reddit codebase. `https://github.com/reddit/reddit`.

[4] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. *ACM SIGMOD Record*, 24(2):23–34, 1995.

[5] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 159–174. ACM, 2007.

[6] C. Amza, A. Chanda, A. L. Cox, S. Elnikety, R. Gil, K. Rajamani, W. Zwaenepoel, E. Cecchet, and J. Marguerite. Specification and implementation of dynamic web site benchmarks. In *International Workshop on Workload Characterization*, pages 3–13. IEEE, 2002.

[7] M. Ansari, M. Luján, C. Kotselidis, K. Jarvis, C. Kirkham, and I. Watson. Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In *High Performance Embedded Architectures and Compilers*, pages 4–18. Springer, 2009.

[8] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, 1981.

[9] P. A. Bernstein, C. W. Reid, and S. Das. Hyder—a transactional record manager for shared flash. In *CIDR*, volume 11, pages 9–20, 2011.

[10] P. A. Bernstein, C. W. Reid, M. Wu, and X. Yuan. Optimistic concurrency control by melding trees. *PVLDB*, 4(11):944–955, 2011.

[11] S. Boyd-Wickizer. *Optimizing Communication Bottlenecks in Multiprocessor Operating System Kernels*. PhD thesis, Massachusetts Institute of Technology, 2013.

[12] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. RadixVM: Scalable address spaces for multithreaded applications. In *Eurosys*, pages 211–224. ACM, 2013.

[13] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: designing scalable software for multicore processors. In *SOSP*, pages 1–17. ACM, 2013.

[14] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD*, pages 1243–1254. ACM, 2013.

[15] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, 1983.

[16] R. Johnson, I. Pandis, and A. Ailamaki. Eliminating unscalable communication in transaction processing. *PVLDB*, 23(1):1–23, 2014.

[17] A. Kemper and T. Neumann. HyPer: A hybrid OLTP and OLAP main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206. IEEE, 2011.

[18] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[19] P.-Å. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *PVLDB*, 5(4):298–309, 2011.

[20] V. Leis, A. Kemper, and T. Neumann. Exploiting hardware transactional memory in main-memory databases. In *ICDE*, 2014.

[21] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased transactional memory. In *Workshop on Transactional Computing*, 2007.

[22] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *OSDI*, pages 265–278. USENIX Association, 2012.

[23] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *PVLDB*, 3(1-2):928–939, 2010.

[24] T.-I. Salomie, I. E. Subasu, J. Giceva, and G. Alonso. Database engines on multicores, why parallelize when you can distribute? In *Eurosys*, pages 17–30. ACM, 2011.

[25] J. Schneider, F. Landau, and R. Wattenhofer. Synchronization phases (to speed up transactional memory). Technical report, July 2011.

[26] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400. Springer, 2011.

[27] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP*, pages 385–400. ACM, 2011.

[28] M. Stonebraker, S. Madden, J. D. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*, pages 1150–1160, 2007.

[29] M. Stonebraker and A. Weisberg. The VoltDB main memory DBMS. *IEEE Data Engineering Bulletin*, 36(2), 2013.

[30] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, pages 18–32. ACM, 2013.

[31] W. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, 1988.