Parallel Execution for Conflicting Transactions

by

Neha Narula

B.A., Dartmouth College (2003)
S.M., Massachusetts Institute of Technology (2010)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2015

Author . . . .
Department of Electrical Engineering and Computer Science
May 20, 2015

Certified by . . . .
Robert T. Morris
Professor
Thesis Supervisor

Certified by . . . .
Eddie Kohler
Associate Professor, Harvard University
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Chair, Department Committee on Graduate Theses

Parallel Execution for Conflicting Transactions

by

Neha Narula

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2015, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

Abstract

Multicore main-memory databases only obtain parallel performance when transactions do not conflict. Conflicting transactions are executed one at a time in order to ensure that they have serializable effects. Sequential execution on contended data leaves cores idle and reduces throughput. In other parallel programming contexts—not serializable transactions—techniques have been developed that can reduce contention on shared variables using *per-core* state. This thesis asks the question, can these techniques apply to a general serializable database?

This work introduces a new concurrency control technique, *phase reconciliation*, that uses per-core state to greatly reduce contention on popular database records for many important workloads. Phase reconciliation uses the idea of *synchronized phases* to amortize the cost of combining per-core data and to extract parallelism.

Doppel, our phase reconciliation database, repeatedly cycles through *joined* and *split* phases. Joined phases use traditional concurrency control and allow any transaction to execute. When workload contention causes unnecessary sequential execution, Doppel switches to a split phase. During a split phase, commutative operations on popular records act on per-core state, and thus proceed in parallel on different cores. By explicitly using phases, phase reconciliation realizes two important performance benefits: First, it amortizes the potentially high costs of aggregating per-core state over many transactions. Second, it can dynamically split data or not based on observed contention, handling challenging, varying workloads. Doppel achieves higher performance because it parallelizes transactions on popular data that would be run sequentially by conventional concurrency control.

Phase reconciliation helps most when there are many updates to a few popular database records. On an 80-core machine, its throughput is up to 38$\times$ higher than conventional concurrency control protocols on microbenchmarks, and up to 3$\times$ on a larger application, at the cost of increased latency for some transactions.

Thesis Supervisor: Robert T. Morris
Title: Professor

Thesis Supervisor: Eddie Kohler
Title: Associate Professor, Harvard University

Acknowledgments

First and foremost, I would like to thank my advisors, Robert Morris and Eddie Kohler. I deeply appreciate Robert's thoughtful and insightful feedback. He has been the strongest influence in shaping how I think about systems and research. The way I did research changed after I started working with Eddie. He provided guidance and encouragement at critical points in time, and his intuition is always spot-on. Barbara Liskov's principled systems research has been an inspiration and strong influence on my own work, and I thank her for being on my committee and for our many interesting discussions. I'm honored to have worked with all three of them.

The members of PDOS and the rest of the ninth floor have helped me intellectually grow and learn tremendously. Special thanks to Austin Clements, Cody Cutler, Evan Jones, Yandong Mao, Dan Ports, Stephen Tu, and Irene Zhang for their influence on this work.

Thanks to Russ Cox and especially Dmitry Vyukov for investigating Go's performance on many cores. I'm grateful Dmitry found certain contention problems intriguing enough to pursue and fix. Thanks to Sharon Perl for encouraging me to go to MIT in the first place, and Brad Chen for letting me hang on to Google for years while I found my way.

A special thanks to the community of database and distributed systems enthusiasts in academia and industry. I had the privilege of participating in fun conferences and meeting interesting people, and the combination reignited my passion for the field. Their appetite for research is astounding and their enthusiasm is contagious.

Thank you to all my friends who supported me during this long process, and through all the ups and downs. Without you I would not have made it. Thank you to Brian and my family, Mom, Dad, and Megan, for their unwavering faith and support.

* * *

This research was supported by NSF awards 1065114, 1302359, 1301934, 0964106, 0915164, by grants from Quanta and Google, and by an NSF Graduate Research Fellowship.

* * *

This dissertation incorporates and extends the work in the following paper:

> Neha Narula, Cody Cutler, Eddie Kohler, and Robert T. Morris. Phase Reconciliation for Contended In-Memory Transactions. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, October 2014.

Contents

List of Figures

List of Tables

ONE

Introduction

Databases provide serializable transactions: the effect of executing a set of transactions concurrently should equal the effect of the same transactions executed in some serial order. The guarantee of serializability lets developers write application programs as if their transactions will run one at a time; this makes it easier to program database-backed applications because the developer can write their application as if transactions never run concurrently.

Achieving serializability requires care when concurrent transactions *conflict*, which happens when one writes a record that another either reads or writes. Without care, conflicts could cause one transaction to observe unserializable results—i.e., values that would be impossible for any serial transaction order to create or observe. Database concurrency control protocols enforce serializability on conflicting transactions by executing them one at a time: one transaction will wait for the other, or fail and retry. But the key to good multi-core performance and scalability is the elimination of this kind of serial execution. Cores should make progress in parallel whenever possible.

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, many concurrent transactions might update the item's current highest bid. Modern multi-core databases will execute these transactions sequentially, leaving cores idle. Since many online transaction processing workloads fit in memory and transactions do not stall on the network or disk, sequential processing is the biggest factor affecting performance.

This thesis presents *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.

The basic technique of phase reconciliation is to split logical values across cores, so that cores can update local data, instead of running one at a time because of shared state. The local data copies must be combined to read a value. 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. Doppel uses phase reconciliation to the obtain the parallel performance of split values while getting good performance with arbitrary database transactions. The rest of this chapter goes into more detail about the costs of conflict and describes the challenges in using split values and our solutions.

1.1 System model

Our work concerns high-performance in-memory multi-core databases. Systems that follow this model have several common features. We explain these features in more detail, referring specifically to our Doppel implementation.

Doppel is an in-memory multi-core database that supports arbitrary transactions. We use the model of *one-shot* transactions [6, 50]. Once begun, a transaction runs to completion without network communication or disk I/O. Combined with an in-memory database, this means cores will never block due to user or disk stalls. Some systems assume the set of transactions is known ahead of time for static analysis; we do not.

Doppel is a key/value database built on a hash table, and uses a model of rich types as values, like in the key/value store Redis [4]. For example, a single key's value could hold a blob of bytes, a number, or an ordered list of entries where each entry is a rank and a blob of bytes. Transactions are composed of one or more operations, along with local computation on the data returned by Doppel operations. Each operation takes a key and optional other arguments. For example, the $\text{PUT}(k, v)$ operation looks up a record with key k, and overwrites its value with v. Some operations return values—$\text{GET}(k)$, for example, returns the value of key k—and others do not; some operations modify the database and others do not. Each value type supports a limited number of operations, which are described in chapter 4.

Application developers choose keys, and can embed any structure they choose in their keys. Developers can simulate multiple tables using disjoint key ranges, or indexes using specially designed keys and values, but Doppel does not provide a scan operation.

Figure 1-1 shows an example Doppel transaction for an auction website. This transaction inserts a new bid on an item and if necessary, updates the maximum bid for the item. If the keys and values do not exist, Doppel will create them when the transaction commits.

1.2 Conflict costs

Transactions *conflict* when one transaction reads or writes a record that another transaction writes. To serialize the execution of transactions that conflict, most modern databases use variants of two-phase locking [24] or optimistic concurrency control [32].

```
BidTransaction(bid Bid, item Key) {
  PUT(NewBidKey(bid), bid)
  max_bid = GET(MaxBidKey(item))
  if max_bid == nil || bid.amount >  max_bid {
    PUT(MaxBidKey(item), bid.amount)
  }
  Commit() // applies writes or aborts
}
```

Figure 1-1: A simplified bid transaction in an auction application. `NewBidKey()` and `MaxBidKey()` construct keys. The application developer writes these functions to generate keys based on object identifiers. The key for the max bid on an item is unique for each item, while the key for a new bid is unique for each bid.

Two-phase locking (2PL) ensures that two conflicting transactions have serializable effects by forcing one to wait until the other releases its locks. A transaction acquires read and write locks as it reads and writes data, and releases its locks at the end of the transaction. In contrast, optimistic concurrency control (OCC) ensures serializability by detecting conflicting transactions when they try to commit, and aborting all but one of them. OCC avoids locks during execution time by using a validation protocol to commit.

Both of these protocols force sequential execution when any transactions conflict. A conflicting transaction must either wait until other transactions release locks (in 2PL) or abort and retry until other transactions finish (in OCC); in both cases, parallelism is lost. This can greatly reduce throughput in workloads with conflicts. For example, consider a workload of transactions that concurrently increment a popular counter. These transactions conflict, as they have read/write and write/write conflicts on the record. Using either 2PL or OCC, the database would be forced to execute these transactions one at a time, effectively limiting performance to that of a single-core system.

1.3 Splitting data

Concurrent increments are a problem developers have addressed in scalable operating systems using efficient multi-core counter designs, such as for packet counters [25]. An operating system will frequently provide a counter of the number of packets received so far. This counter is rarely read since it is mostly useful for debugging. But on a multi-core machine, packets might arrive on every core. Updates to a shared counter might become a bottleneck. Therefore, modern operating systems partition a logical packet counter value into J counters, one per core. To increment the logical counter, a core updates its per-core value $c_j \leftarrow c_j + 1$, and we say the counter is *split* between the cores. To read the counter, a core *reconciles* these per-core values into one correct value by adding them together:

counter $\leftarrow \sum c_j$. In order to read the correct counter value amidst other cores performing concurrent increments (a linearizable read), the per-core counter values should have locks, and a reader must acquire a lock for every per-core value, as shown in Figure 1-2. Though the ordering of increment operations is lost, this is still correct because addition is commutative. The increments, in any order, produce the same value.

```
Data: counter $c$, number of cores $J$
$sum \leftarrow 0$;
for $j \leftarrow 1$ to $J$ do
    lock($c_j$);
    $sum \leftarrow sum + c_j$;
for $j \leftarrow 1$ to $J$ do
    unlock($c_j$);
return $sum$
```

Figure 1-2: Pseudocode to correctly reconcile and read a split counter.

The idea of per-core data structures is used many places in the Linux kernel, for example for reference counters and in the virtual memory system [15].

1.4 Joined and split data execution plans

2PL, OCC, and the kinds of per-core counters described in §1.3 provide different mechanisms for making access to objects serializable. We call these mechanisms *execution plans*. At a high level, execution plans tell the database how to safely execute each operation during transaction execution and commit.

For example, consider a database in which each key supports two operations, ADD(k,n) and GET(k). ADD(k,n) adds n to the value of key k, and GET(k) returns the value of k. The typical execution plan for these operations in a 2PL database will obtain a read lock on k for GET(k), and obtain a write lock on k and set $k \leftarrow k+n$ for ADD(k,n); in both cases, the lock will be released at transaction commit time. We call this execution plan the *joined plan* since all cores operate jointly on shared data. A record in this state is *joined*. In the joined plan, GET(k)s do not conflict, but GET(k) conflicts with ADD(k,n), and ADD(k,n)s conflict with each other. Figure 1-3 shows two conflicting ADD(x,n) operations and two GET(x) transactions executing in parallel; the vertical striped bars show the period where a core is blocking, waiting for another transaction to finish. The ADD(x,n) on the second core blocks the ADD(x,n) on the first core. In a 2PL database, this might happen because the second core is holding a lock on x.

The joined plan makes concurrent add operations slow. However, a database might also support another plan—call it the *split plan*—that speeds up concurrent adds using per-

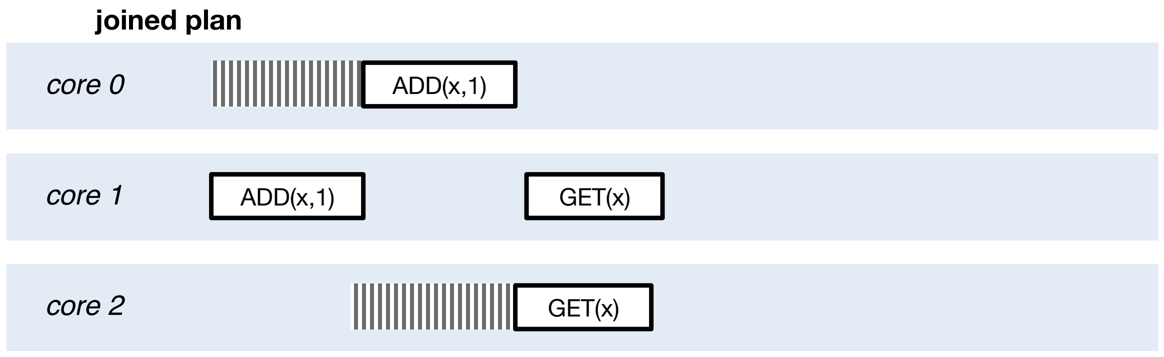


Figure 1-3: The ADD(k,n) operations conflict, so the operation on core 0 must wait for core 1. The GET(k) operation conflicts with ADD(k,n), so the GET(k) on core 2 must wait for the ADD(k,n)s on the other cores to finish. GET(k) operations can proceed in parallel on cores 1 and 2.

core data. The split plan uses a *per-core slice* for a record, with one entry per core. The per-core slice for a record with key k and core j is $v_j[k]$. We say the record for key k is *split*. When core j sees an ADD(k,n) operation in a transaction, it updates its core's value: $v_j[k] \leftarrow v_j[k] + n$. GET$(k)$ reads all the per-core values and sums them: $\sum v_j[k]$. This is exactly like the counter in §1.3.

In the split plan, ADD(k,n) operations on the same record also do not conflict; they can execute in parallel by updating different per-core slices. Figure 1-4 shows three transactions running on different cores, the first and second cores executing ADD(k,n) operations on the same record in parallel. A GET(k) on the same record conflicts with an ADD(k,n), since it must read and sum all the per-core values. The third core in Figure 1-4 is waiting for the ADD(k,n)s on cores one and two.

This technique does not work with all operations—it is critical here that ADD(k,n) operations commute, meaning that they return the same results, and cause the same changes to the database state, regardless of the order they are executed. This is true for ADD(k,n) since addition commutes in the mathematical sense, and since ADD(k,n) does not return a result. If ADD(k,n) were not commutative—for instance, if we were operating on floating-point numbers (the result of floating-point addition can differ depending on the order of addition), or if ADD(k,n) returned the new value of the counter—using per-core values would not produce serializable results.

Since each plan incurs conflict between different operations, the two plans experience sequential execution on different workloads. In the joined plan, GET(k)s can run in parallel, but ADD(k,n)s are run one at a time on the record being updated. In the split plan, ADD(k,n)s run in parallel as well, but GET(k)s are more expensive. Neither plan is ideal for all workloads. If there are mostly ADD(k,n)s, then the joined plan is slow because oper-

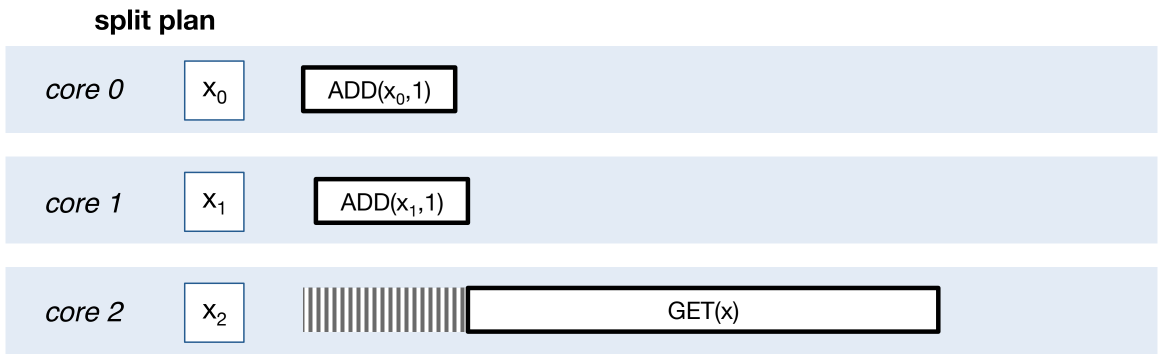


Figure 1-4: The ADD(k,n) operations can proceed in parallel. The GET(k) operation conflicts with ADD(k,n), and also needs to sum the per-core values.

ations on the same record must run one at a time, and if there are mostly GET(k)s, then the split plan is slow because it requires reading all of the per-core values. This is unfortunate because database workloads contain mixes of transactions, and only very rarely does a mix allow the exclusive use of one plan or another. Furthermore, the best mix of plans for a record often changes over time.

1.5 Solution

The overall goal of the thesis is to combine these two plans in a dynamic way, getting the good parallel performance of the split plan when possible, and the general-purpose flexibility of the joined plan otherwise.

Our key insight is that a database that changes records' plans in response to contention and operations can achieve much better parallelism than a database that uses a single plan for each record. In a workload with a mix of ADD(k,n) and GET(k) operations, instead of statically using one plan, it is more efficient to switch a record between plans as transactions require it. Doppel executes ADD(k,n)s on per-core values for contended keys using the split plan, and then sums the per-core values once, writes the sum to a shared location as a *reconciled* value, and executes the GET(k)s on the shared value using the joined plan. The ADD(k,n)s execute in parallel using per-core slices, and the GET(k)s execute in parallel in the absence of writes. A critical piece of Doppel's performance story is that the cost of reconciling the record and switching plans is amortized over many transactions.

In Doppel, records use different plans over time, but stay in one plan over the course of many transactions. Doppel switches phases every few tens of milliseconds, and re-evaluates plans for records roughly every tenth of a second, in order to respond to quickly changing workloads. Transactions can execute on records using different plans. Transactions that execute operations tuned to work well with records' current plans can execute; a transaction with an operation that doesn't work with a record's current plan is deemed *incompatible* and delayed. Batching transactions so that a record uses either the split or joined plan for many transactions reduces the cost of supporting both plans on one record. For example, if a record were using a split plan, transactions executing $\text{ADD}(k,n)$ operations could proceed, but transactions issuing $\text{GET}(k)$ operations would be delayed until the record moved to a joined plan. By reordering transactions, Doppel can reduce the cost of reconciling data for reads, and a record can benefit from split data.

But how can Doppel support transactions that perform operations on *multiple* contended records? Unless such a transaction happened to use the right operations on all data items, it seems like the transaction could be delayed indefinitely.

Doppel handles this situation by imposing synchronized *phases* on transaction execution; all records change phase at the same time. The database cycles between two kinds of phases, *split phases* and *joined phases*. During a split phase, records that might benefit use split plans with per-core slices. During a joined phase, all records use joined plans so the database can use conventional concurrency control. As a result, during joined phases any transaction can execute correctly—if not necessarily efficiently—and there is no chance that a transaction is delayed for unbounded time by incompatible operations.

When Doppel observes contention, it operates as much as possible in split phases, and only switches to a joined phase when there are delayed transactions. In the applications we investigated, only a few records were contended, so most records in the database always use the joined plan. Contended records usually suffered from a mix of one type of update operation and reads in multi-key transactions. Using synchronized phases, many transactions execute using a record's split plan while reads are delayed. But the cost of reconciling a record only happens at the phase boundary, instead of for every read. This helps performance because at the beginning of a joined phase *all* records are reconciled, and transactions with many reads can execute efficiently, and in parallel.

Not all records would benefit from using a split plan in the split phase. Doppel must choose whether or not to use a split plan for a record during a given split phase. Doppel decides when to do so based on observed conflicts and operations. For example, if Doppel were using the joined plan for a record but many transactions executed conflicting $\text{ADD}(k,n)$ operations on it, Doppel might choose to use a split plan on the record during the next split phase. Over time, records that were once popular and contended may become less so, and using per-core data for a non-contended record just makes reads more expensive without providing a benefit to parallelism. Similarly, new records may become contended, and could benefit from per-core data. Doppel continually *classifies* records to make sure it is using a good plan.

The workloads that work best with phase reconciliation are ones with frequently-updated data items where contentious updates have certain properties. Doppel uses conven-

```
BidTransaction(bid Bid, item Key) {
  PUT(NewBidKey(bid), bid)
  MAX(MaxBidKey(item), bid.amount)
  Commit() // applies writes or aborts
}
```

Figure 1-5: The bid transaction from Figure 1-1 rewritten to use a commutative Doppel operation.

tional concurrency control on joined records, which induces some set of sequential orderings on an execution of concurrent transactions. The results of executing operations on split data in those transactions must match at least one of those orderings. To ensure this, Doppel will only execute operations that *always* commute on a record when it is using a split plan. This way, the result of many operations on per-core data for a record, coupled with reconciliation, matches *any* ordering, including one imposed by the containing transactions.

Figure 1-5 shows the `BidTransaction` from Figure 1-1 rewritten to use a commutative operation, $\textsc{Max}(k,n)$. This operation replaces the value of key k with the maximum of its current value and n. The new `BidTransaction` has two operations—a commutative operation on the current winning bid key, and a non-commutative operation on the new bid's key. By using the commutative $\textsc{Max}(k,n)$ operation instead of $\textsc{Get}(k)$ and $\textsc{Put}(k,v)$, which do not commute, the max bid key could benefit from being split in Doppel's split phase.

In order for Doppel to execute split operations efficiently, i.e., get a parallel speedup on multi-core hardware, the operations should be able to be *summarized* per core, so the work is done in parallel. Chapter 4 describes these types of operations in more detail. There are many workloads that have these properties, for example those that update aggregated or derived data. Examples 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].

In summary, Doppel moves records dynamically between joined and split plans so transactions can take advantage of per-core data for parallel writes and reconciled data for fast reads. With these solutions, 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 that vary over time.

1.6 Contributions

The contributions of this work are as follows:

- A model of database execution based on split and joined execution plans. The split plan efficiently uses per-core data to execute conflicting operations in parallel, while the joined plan can correctly execute any operation.
- The idea of batching transactions with similar kinds of operations into phases to efficiently amortize the cost of switching records between split and joined plans.
- Synchronized phases, which ensure that transactions that read multiple split records can find a single time at which none are split, and that writes to multiple split records in a single transaction are revealed atomically.
- A design for dynamically moving data between split and joined states based on observed contention.
- An in-memory database, Doppel, which implements these ideas and improves the overall throughput of various contentious workloads by up to 38× over OCC and 19× over 2PL.

We also demonstrate that on a workload modeled on real auction sites, the RUBiS web auction software, Doppel improves bidding throughput with popular auctions by up to 3× over OCC, and has comparable read throughput.

1.7 Outline

Chapter 2 relates this work to other ideas in executing transactions in phases, commutative concurrency control, and techniques for parallelizing multi-core transactions. Doppel's design is described in more detail in chapter 3. Chapter 4 describes the class of operations that can be efficiently parallelized using phase reconciliation. Chapters 5 and 6 describe the system we implemented and our application experience. Chapter 7 evaluates performance hypotheses about the workloads where phase reconciliation provides a benefit. The last chapter looks at future directions of this work and concludes.

TWO

Related work

The idea of phase reconciliation is related to ideas in executing fast transactions on multi-core databases, using commutativity in concurrency control and to reconcile divergent values in distributed systems, and using scalable data structures in multi-core operating systems.

2.1 Database transactions on multi-core

Several databases achieve multi-core speedup by partitioning the data and running one partition per core. In this domain conventional wisdom is that when requests in the workload conflict, they must serialize for correctness [29]. Given that, work has focused on improving parallelism in the database engine for workloads without much conflict. Systems like H-store/VoltDB [50, 51], HyPer [31], and Dora [40] 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. Horticulture [41] and Schism [22] are two systems developed to aid in choosing a partitioning to maximize single-partition transactions, using workload traces. This approach is brittle to changing workloads and does not help with non-partitionable workloads, two situations Doppel handles. Also, in our problem space (data contention) partitioning will not necessarily help; a single popular record with many writes would not be able to utilize multiple cores.

Other systems operate on copies of the database on different cores, but still contend for writes. Hyder [12] uses a technique called meld [13], 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 [44] 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 records among cores for writes and provides a way to re-merge the data for access by other cores.

Bernstein and Newcomer describe a design for dividing a popular record into partitions, so transactions update different copies to reduce hotspots in a workload with many conflicting writes [11]. They also discuss batching updates on hot records across many transactions to reduce the number of writes to the contended data. They note that reading a partitioned record is more expensive, and might cause performance problems. Doppel addresses this situation and makes these ideas practical by combining them with the idea of synchronized phases, in order to efficiently read and write keys.

Doppel uses optimistic concurrency control, of which there have been many variants [5, 10, 13, 32, 34, 52]. We use the algorithm in Silo [52], which is very effective at reducing contention in the commit protocol, but does not reduce contention caused by conflicting reads and data writes. Shore-MT [30] removes scalability bottlenecks from Shore [16] found in the the buffer pool manager, lock manager, and log manager. Larson et al. [34] explore optimistic and pessimistic multiversion concurrency control algorithms for main-memory databases, and this work is implemented in Microsoft's Hekaton [23]. Both of these pieces of work present ideas to eliminate contention in the database engine due to locking and latches; we go further to address the problem of contention caused by conflicting writes to data.

Other database research has investigated how to achieve intra-query parallelism using multi-core systems, for example on queries that read and aggregate large amounts of data. This research uses adaptive aggregation algorithms [8, 9, 17, 18, 48, 57]. In these algorithms, the database uses per-core data structures for queries that require large scans and have group by aggregations. Most of this work focuses on large read queries; Doppel targets executing conflicting update workloads in parallel.

2.2 Commutativity

Commutative concurrency control. Doppel's split phase techniques are related to ideas that take advantage of commutativity and abstract data types in concurrency control. Gawlick and Reuter first developed concurrency control algorithms that allowed concurrent access to the same data by overlapping transactions [26, 27, 42]. These techniques were limited to aggregate quantities. Escrow transactions extend this idea; an escrow system carefully keeps track of potential values for a data item given the concurrent set of transactions and data criteria [39]. In these systems, data is not partitioned per-core, so writes still happen on a shared data item. Doppel also takes advantage of commutativity, but performs commutative operations on per-core data to avoid contention. All of these techniques use the values of the contended data to aid in determining whether a transaction could commit or abort; this would also be useful in Doppel.

Weihl et al. created a concurrency control algorithm that determined conflict in concurrency control by examining the specification of atomic abstract data types [53–56], implemented in Argus [36] and based on other work in abstract data types [37]. Doppel does not use the return values of an operation to determine commutativity, sacrificing some concurrency. Transactional boosting [28] uses data types and commutativity to reduce conflict when using highly-concurrent linearizable objects in software transactional memory. During split phase, Doppel only performs certain operations on split data items, reminiscent of following a specification on that datatype, and relies on these operations being commutative for correctness. However, in the joined phase transactions can execute arbitrary reads and writes on any datatype. Synchronized phases provide a way for Doppel to transition between different semantic descriptions for objects, in an efficient way.

Commutativity in distributed systems. Some work in distributed systems has explored the idea of using commutativity to reduce coordination, usually forgoing serializability. RedBlue consistency [35] uses the idea of blue, eventually consistent local operations, which do not require coordination, and red, consistent operations, which do. Blue phase operations are commutative, and are analagous to Doppel's operations in the split phase. Walter [49] uses the idea of counting sets to avoid conflicts. Doppel could use any Conflict-Free Replicated Data Type (CRDT) [45] with its split operations in the split phase, but does not limit data items to specific operations outside the split phase. None of these systems provide multi-key transactions.

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.

2.3 Scalability in multi-core operating systems

Linux developers have put a lot of effort into achieving parallel performance on multiprocessor systems. Doppel adopts ideas from the multi-core scalability community, including the use of commutativity to remove scalability bottlenecks [20]. OpLog [14] uses the idea of per-core data structures on contentious write workloads to increase parallelism, and Refcache [19] uses per-core counters, deltas, and epochs. Tornado used the idea of clustered objects, which involves partitioning a data item across processors [25]. Per-core counters is a technique widely used in the Linux kernel [21]. 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.

THREE

Phase reconciliation

This chapter explains how Doppel structures phases and transitions between them. Phases are useful for the following reasons: First, in the split phase some operations that would normally conflict execute quickly, since those operations can update per-core data rather than shared data. Second, the cost of reconciling per-core data is amortized over many transactions. Finally, using phases the system can still efficiently support transactions that read multiple records in the joined phase, when all records are guaranteed to be reconciled.

The database cycles between phases, and each transaction executes on one core, entirely within a single joined or split phase. The rest of this chapter describes each phase in detail, how updates are reconciled, how records are classified, and how the system transitions between phases. It concludes with a discussion about what kinds of transactions might not work well with Doppel.

3.1 Split operations

To understand how Doppel executes transactions in phases, we must first know how Doppel executes operations. As described in §1.1, each operation takes as an argument a key, and an optional set of additional arguments. Doppel has a built-in set of operations which always commute, and for which it has efficient split implementations. The implementation of each split operation must provide three parts:

- $\text{OP}_{\text{init}}(j,k)$ initializes core j's per-core slice for k's record.
- $\text{OP}_{\text{delta}}(j,k,\dots)$ applies the operation to core j's per-core slice.
- $\text{OP}_{\text{reconcile}}(j,k)$ merges core j's per-core slice back into k's global record.

During the split phase, whenever a transaction executes $\text{OP}(k,\dots)$, Doppel will instead execute $\text{OP}_{\text{delta}}(j,k,\dots)$ with the same arguments passed to the original operation, where j is the id of the core on which the transaction is running. At the end of the split phase, to reconcile the per-core slices, Doppel will execute $\text{OP}_{\text{reconcile}}(j,k)$ for each per-core slice.

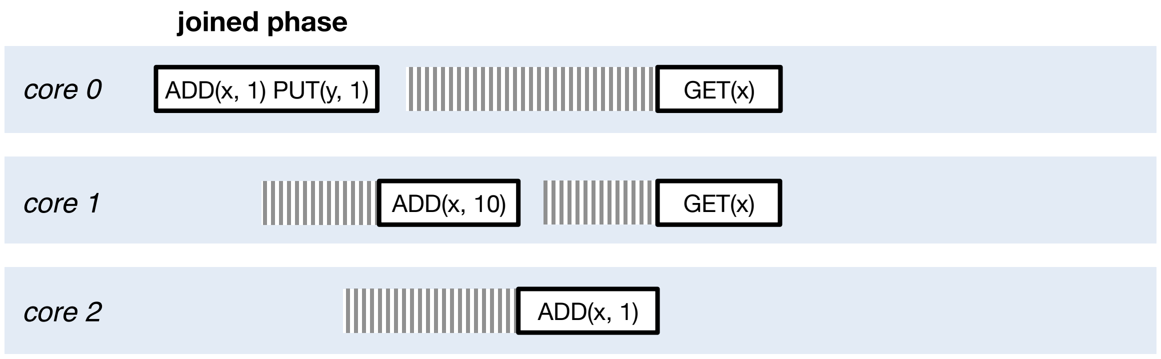


Figure 3-1: Several transactions executing in Doppel using OCC in the joined phase. The $\textsc{Add}(k,n)$ operations conflict with each other, and with $\textsc{Get}(k)$ operations on the same record, x.

The combination of applying the delta function to a slice and the reconcile function to a slice and record should have the same effect as the operation would if applied to the shared record instead of a per-core slice.

For example, here are the functions for $\textsc{Max}(k,n)$:

- $\textsc{Max}_{\text{init}}(j,k) = c_j[k] \leftarrow v[k]$
- $\textsc{Max}_{\text{delta}}(j,k,n) = c_j[k] \leftarrow \max\{c_j[k],n\}$
- $\textsc{Max}_{\text{reconcile}}(j,k) = v[k] \leftarrow \max\{v[k],c_j[k]\}$

To ensure good performance, per-core slices must be quick to initialize, and operations on slices must be fast. Chapter 4 describes the properties of split operations in more detail.

3.2 Joined phase

A joined phase executes all transactions using conventional concurrency control. All records are reconciled—there is no per-core data—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.

Doppel's joined phase concurrency control protocol is based on that of Silo [52]. Figure 3-2 shows the joined-phase commit protocol. Records have transaction IDs (TIDs); these indicate the ID of the last transaction to write the non-split record, and help detect

```
Data: read set $R$, write set $W$
// Part 1
for Record, operation in sorted($W$) do
    if (lock(Record) $\neq$ true) then abort();
commit-tid $\leftarrow$ generate-tid($R$,$W$)
// Part 2
for Record, read-tid in $R$ do
    if Record.tid $\neq$ read-tid
        or (Record.locked and Record $\notin W$)
    then abort();
// Part 3
for Record, operation in $W$ do
    Record.value $\leftarrow$ apply(operation, Record, commit-tid);
    Record.tid $\leftarrow$ commit-tid;
    unlock(Record);
```

Figure 3-2: Doppel's joined phase commit protocol. Fences are elided.

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. If a transaction executes an operation like ADD(k,n) in the joined phase, it puts the record for the key k in its read and write sets. 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 unique 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. Using a shared counter to assign TIDs would cause contention between cores. To avoid this, our implementation assigns TIDs locally, using a per-core counter, with the core id in the low bits. The resulting commit protocol is serializable, but the actual execution may yield different results than if the transactions were executed in TID order [52].

Figure 3-1 shows five transactions executing on three cores in a joined phase. Each transaction runs on a single core and maintains a read and write set. Initially, all three transactions are running ADD(k,n) on the same record x, and will conflict and run sequentially. Similarly, ADD(k,n) conflicts with GET(k) on the same record, so the GET(k) operations are delayed. The GET(x) operations can later execute in parallel.

Since each transaction executes completely within a single phase, Doppel cannot leave a joined phase for the following split phase until all current transactions commit or abort.

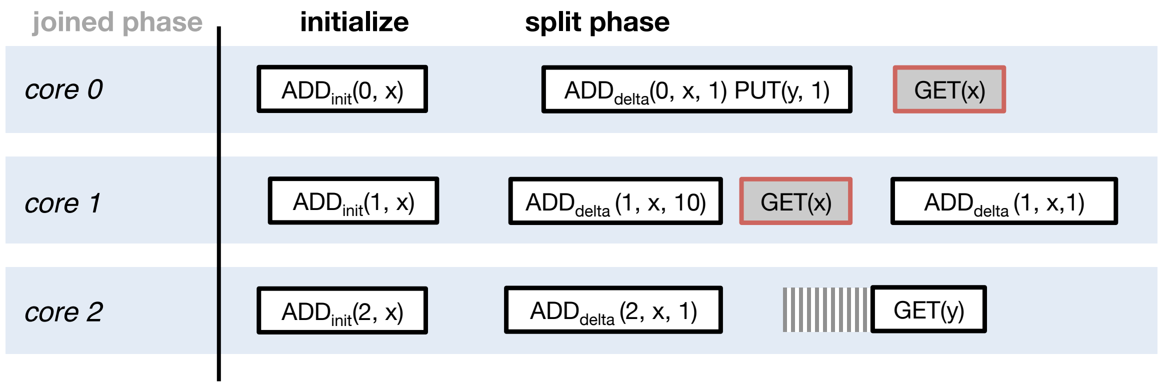


Figure 3-3: Cores initialize per-core slices at the beginning of a new split phase, to split x. The transactions use $\text{ADD}_{\text{delta}}(k,n)$ operations on per-core slices. Transactions with $\text{GET}(k)$ operations on x are stashed.

3.3 Split phase

A split phase can execute in parallel some transactions that conflict. Not all records are split during a split phase; non-contended records remain joined. Accesses to joined data proceed much as in a joined phase, using OCC.

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. During the split phase, as a transaction executes, all operations on split records are buffered in a new split-data write set. When the transaction commits, the updates are applied to their per-core slices. At the end of the split phase, the per-core slices are merged back into the global store. For example, a transaction that executed an $\text{ADD}(k,10)$ operation on a split numeric record might add 10 to the local core's slice for that record.

Figure 3-3 shows a split phase in Doppel where x is split. First, Doppel initializes per-core slices for x. Then, three transactions, one on each core, execute in parallel. Each includes an $\text{ADD}(k,n)$ on the split record x. Doppel translates this to the $\text{ADD}_{\text{delta}}(k,n)$ function on x. Since the $\text{ADD}(k,n)$ operations now use per-core slices, they can execute in parallel, and the three transactions do not conflict. The transaction on the first core also writes to y, which is not split. Doppel will execute the operation on y using OCC. This forces it to serialize with the later transaction on the third core, which reads y.

Split phases cannot execute all transactions, however. Operations on the same split keys in the split phase must commute. For simplicity, Doppel actually selects only one *selected operation* per split record per split phase. Different records might have different selected operations, and the same record might have different selected operations in different split phases. A transaction that invokes an unselected operation on a split record will be aborted and *stashed* for restart during the next joined phase, at which point Doppel can execute

```
Data: joined read set $R$, joined write set $W$, split write set $SW$
// Part 1
for Record, operation in sorted($W$) do
    if (lock(Record) $\neq$ true) then abort();
commit-tid $\leftarrow$ generate-tid()
// Part 2
for Record, read-tid in $R$ do
    if Record.tid $\neq$ read-tid
        or (Record.locked and Record $\notin W$)
    then abort();
// Part 3
for Record, operation in $W$ do
    Record.value $\leftarrow$ apply(operation, Record, commit-tid);
    Record.tid $\leftarrow$ commit-tid;
    unlock(Record);
for slice, operation in SW do
    slice $\leftarrow$ delta(operation, slice);
```

Figure 3-4: Doppel's split phase commit protocol.

any combination of operations on the record using OCC. In Figure 3-3, $\text{ADD}(k,n)$ is the selected operation on x, so the two $\text{GET}(k)$ operations are stashed. Doppel saves them to execute in the next joined phase.

When a split phase transaction commits, Doppel uses the algorithm in Figure 3-4. It is similar to the algorithm in Figure 3-2 with a few important differences. The read set R and write set W contain only joined records, while *SW* buffers updates on split data. The decision as to whether the transaction will abort is made before applying the *SW* writes; operations on split data cannot cause an abort, and are not applied if the transaction aborts. In the split phase the commit protocol applies the *SW* delta operations to per-core slices. Since per-core slices are not shared data, they are not locked and the commit protocol does not check version numbers.

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

3.4 Reconciliation

At the end of the split phase, each core stops processing transactions and begins reconciliation; each merges its per-core slices with the global store using the $\text{OP}_{\text{reconcile}}$ function for each record's selected operation. Figure 3-5 shows the reconciliation algorithm Doppel

```
Data: per-core slices S for core j
for Record, operation, slice in S do
    lock(Record);
    Record.value ← reconcile(operation, slice, Record);
    unlock(Record);
S ← ∅
```

Figure 3-5: Doppel's per-core reconciliation protocol.

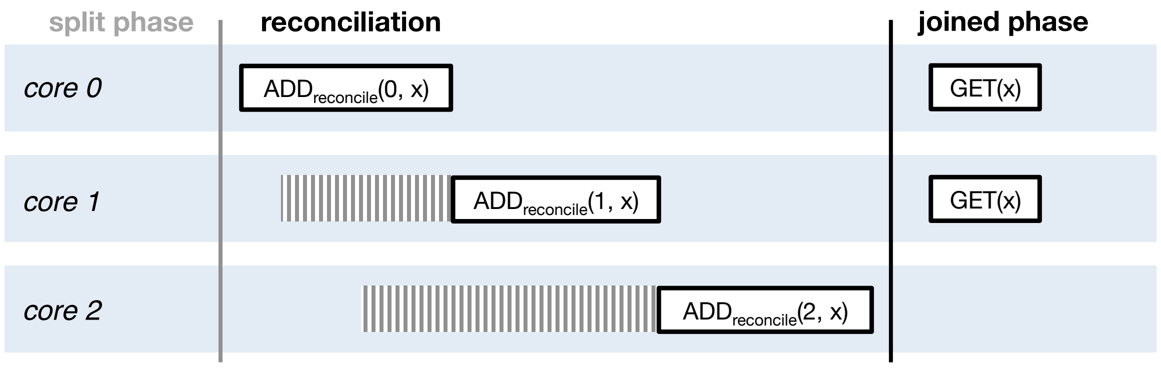


Figure 3-6: Each core j reconciling its per-core values for x, using $\text{ADD}_{\text{reconcile}}(j,x)$. Each core must reconcile sequentially. When done, x's value will be incremented by the sum of all the per-core values.

uses to do so. For example, for a split record with selected operation MAX, each core locks the global record, sets its value to the maximum of the previous value and its per-core slice, unlocks the record, and clears its per-core slice. Figure 3-6 shows three cores reconciling per-core values for record x. This involves sequential processing of the per-core slices, but the expense is amortized over all the transactions that executed in the split phase.

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. Once a core begins reconciliation, it will not process any more split phase operations on per-core slices until the next split phase. However, a core that has finished reconciliation cannot begin processing joined phase transactions again until all cores have finished reconciliation. Once the last core has finished reconciliation, each core can resume, now in the joined phase. At the end of reconciliation the database reflects a correct, serializable execution of all split phase transactions.

3.5 Phase transitions

When Doppel is initialized, there is no split data. Every t_c milliseconds, Doppel checks statistics it is maintaining to see if it should split any records. If so, then Doppel begins a new split phase. Phase transitions apply globally, across the entire database. All cores stop processing transactions and read the list of new split records. Once all cores have synchronized with the list of split records and initialized their per-core slices, Doppel is in split phase, and operations on split records execute on per-core slices.

Doppel only considers switching from a split phase to a joined phase when there are stashed transactions. Since each core synchronizes and stops processing transactions during a phase change, changing phases too often can be costly. However, transactions stashed during a split phase have to wait until the next joined phase, and to ensure low latency for these transactions, phase changing should be frequent. Chapter 7 evaluates this tradeoff between throughput and latency. Doppel balances this issue by bounding the length of a split phase to t_s, if there are stashed transactions. When t_s time has passed since the first stashed transaction, Doppel will initiate a phase change from split to joined. Doppel might transition to a joined phase earlier if there are many stashed transactions. This threshold is set by a parameter, n_s. In experiments in Chapter 7 n_s is 100,000, t_s is 20ms, and t_c is 200ms; these parameters were determined through our experiments.

When Doppel transitions from a split phase to a joined phase, it re-evaluates its statistics to determine if it should change the set of split records or selected operations. This algorithm is described in §3.6. If at this point Doppel decides to split a new record, move a split record back to a joined state, or change the selected operation for a record, then it synchronizes all the cores, reconciling records that are changing from split mode to joined or are changing operations and creating per-core slices for records that are moving from joined to split. If there are no stashed records, then Doppel checks statistics every t_c milliseconds during the split phase to see if it should change the set of split records.

Doppel only stays in joined phase long enough to execute the previously stashed transactions. Once it begins, all cores execute transactions in their stashed lists using OCC. When a core finishes it blocks, and Doppel waits for the last core to finish until moving to the next split phase. No core will start using per-core slices again until all cores have synchronized to move to the next split phase.

3.6 Classification

Doppel classifies records to decide whether or not to split a record in the split phase by estimating the cost of conflict. Doppel's cost estimation algorithm tries to estimate how often a record will cause an abort, and if that value is high, it will split the record. Doppel aborts one transaction when two transactions are executing conflicting operations on the same record. The cost of an abort is running the aborted transaction again, which wastes CPU cycles. Split records never cause aborts during a split phase, so if the record can be split, the abort cost will be eliminated.

When a record is split for a selected operation, then in split phase, any transactions issuing another operation on that record must be stashed. This has a latency cost for the stashed transactions, and a CPU cost to save the transaction and run it later. Doppel's algorithm seeks to minimize the cost of stashing transactions while maximizing the cost saved by splitting records that were causing aborts.

While executing, Doppel samples all transactions' accesses, and in particular conflicting record accesses. Doppel keeps two counts for each operation on the record: the number of times that operation was used on that record, and the number of times that record caused a transaction to abort because of a conflict using that operation. For operations with enough conflicts, Doppel estimates how much of an impact splitting a record with that operation could save in conflict costs; we call this estimate $savings(op, record)$.

$savings(op, record)$ has two components. First is the sampled count of conflicts on the record for op, c_{op}, and second is the sampled count of times op is issued on that record, n_{op}. When a record is split, it no longer causes the conflicts it might cause were it to be always joined, so Doppel uses the number of times the operation is issued on the split record as a proxy for conflict. This effectively assumes that all operations that did not conflict could operate in parallel should the conflicts be removed. w_c and w_i are weights assigned to these factors.

$$savings(op, record) = w_c \times c_{\text{op}} + w_i \times n_{\text{op}}$$

If an operation cannot execute correctly under the split plan, then its $savings(op, record)$ is defined to be zero. If a record is receiving many different operations it might not be worth splitting, even if it causes some conflicts, because it will delay transactions until the next joined phase. Doppel looks at a ratio of the benefit of $savings(op, record)$ to the other operations on the record to estimate the total *impact factor*. A record receives a total number of operations n.

$$impact(op, record) = \frac{savings(op, record)}{n - n_{\text{op}}}$$

The denominator is the sampled count of every operation other than op on the record. This approximates the *benefit per delayed operation*. Doppel then looks at the highest savings $(impact_i, op_i)$ for the record. This operation op_i is the operation that, if split, could provide the highest cost savings despite delayed transactions. If $impact_i$ is higher than some T_{split} threshold, Doppel will split the record. Once a record is split, if $impact_i$ falls below a different threshold, T_{join}, Doppel will put the record back into joined mode.

Good values for the thresholds and weights will vary between systems. The weights represent what ratio to use when considering the number of conflicts, reads, and writes an operation on a record might cause. We found good values for w_c and w_i by running different workloads with different weights; in our experiments $w_i = 1$ and $w_c = 2$. Once Doppel splits the first record that occasionally experiences an incompatible operation, the system will have to change phases. After this, the marginal cost of splitting the next record is much lower, while the savings is the same, so Doppel uses a lower threshold after the first record.

Generally, Doppel will only split records that cause a lot of conflict. Fortunately, in most cases, only a few records cause problems, for example an aggregate count or a popular item receiving bids. In order for a record to force many transactions to abort, many transactions must be accessing that record at a high rate; this implies the transactions are not accessing other records as much.

Doppel continously samples transactions. Every t_c milliseconds Doppel evaluates the formula for all split records and the most conflicted non-split records, to consider changing their status or operation. Records might switch back and forth between being marked as split or not across these evaluations. This is not a problem for performance because it implies that neither scheme for executing transactions on the record (using per-core slices or not) provides much better performance than the other. Doppel also supports manual data labeling ("this record should be split for this operation"), but we only use automatic detection in our experiments.

3.7 Discussion

This formula described above requires tuning to find good threshold numbers and good weights for each of the factors. A better algorithm might derive weights from running a workload trace and measuring performance. We leave this to future work. Also, the benefit gained by splitting might not be linear in the factors; and this is only a rough approximation of conflict. For example, though many transactions might include a splittable operation on a record, perhaps they would not cause many aborts in joined phase. This could happen if a record is written mostly by one core.

Conflict counts might not even be the best feature to minimize in all cases. For example, consider the case where two different operations on a record force a lot of aborts, but one is used in large, CPU-intensive transactions while the other is used in short, inexpensive transactions. It might be more prudent to split the record for the operation used in the large transaction to avoid wasted work, but Doppel could not detect this since it only counts the number of conflicts. Also, our current algorithm does not take into account having multiple contended records in a transaction. If two records in a transaction were heavily contended, then splitting one would only help a small amount; the conflicting transactions would still run one at a time because of the second contended record.

Only certain types of transactions work well with phases. Doppel requires transactions to run entirely within a single phase. This means that long-running transactions would not work well with phases, as they would delay a phase change and leave the system mostly idle while running. If transactions had a lot of variance in length, this could cause some cores to be idle waiting for a phase change while other cores are finishing longer transactions.

In Doppel, phases are designed to work best with mostly small, fast transactions that only access a few records. Transactions that use the disk or communicate with the client over the network might benefit from phase reconciliation, but we have not investigated this. Phases could potentially be useful for bigger, more expensive transactions, since the cost for aborting and retrying expensive transactions is high.

FOUR

Parallel operations

Doppel applies operations to per-core slices to achieve parallel speedup during a split phase. However, not all operations can correctly be performed in parallel, and of those that can, not all offer a performance benefit. Figure 4-1 illustrates which operations Doppel can run correctly, and quickly, in its split phase. Operations that are *always* commutative, no matter database state or operation arguments, can run correctly in Doppel's split phase. Only some of the commutative operations will get a parallel speedup. The rest of this chapter discusses the class of operations that Doppel can split to take advantage of phase reconciliation, and when it can achieve parallel speedup.

4.1 Properties for correctness

In order to achieve correctness when operations execute on per-core slices, the results of the containing transactions should be as though they had executed in some sequential order. These results include modifications to the database and return values from the operations. The tricky part of that is making sure that transactions' split operations (after reconciliation) appear to have executed in a compatible sequential order.

In general, Doppel can correctly execute split operations that *commute*. To show this, we now present a semi-formal development of operations, transactions, and commutativity. The goal is to show that Doppel's execution strategy produces serializable results when operations that execute on per-core state in split phases are commutative.

4.1.1 States and operations

A *database state* $s \in S$ is a mapping from keys to values. The notation $s' = s[k \mapsto v]$ means that s and s' are identical except for key k, and key k has value v in s' (k might not exist in s). The notation $v = s[k]$ means that v is the value of key k from s.

A Doppel *operation* o is modeled as comprising two parts, $\langle o^s, o^v \rangle$. $o^s : S \to S$ is a function on database state that returns a new database state, and $o^v : S \to V$ is a function on

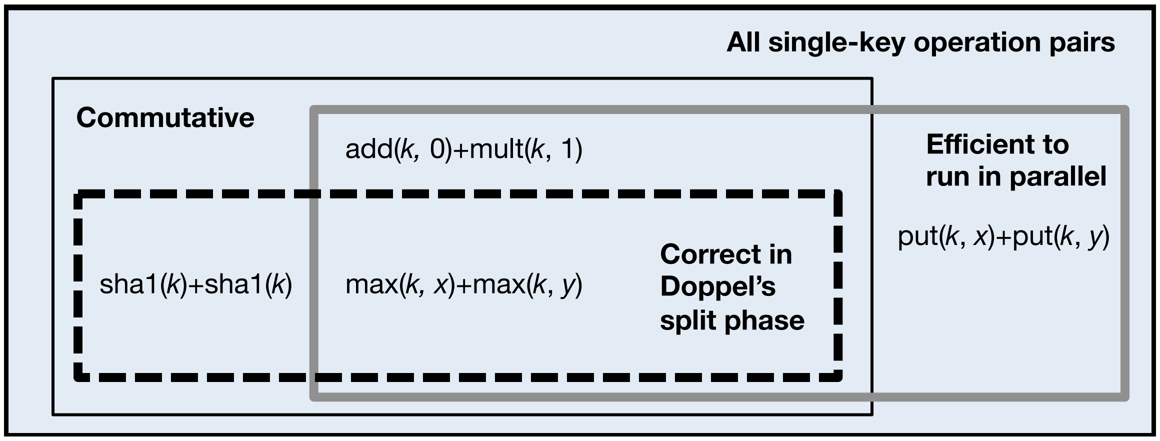

All single-key operation pairs
Commutative
add(k, 0)+mult(k, 1)
Efficient to run in parallel
put(k, x)+put(k, y)
sha1(k)+sha1(k)
max(k, x)+max(k, y)
Correct in Doppel's split phase

Figure 4-1: A categorization of sets of single-key operations. The box with the dashed lines indicates what combinations of operations Doppel can execute correctly in the split phase.

database state that returns a value to the caller.

An operation might change the database state. For example, an operation $\langle o^s, o^v \rangle$ which sets a key x's value to 5 would be expressed in the following way: $o^s(s) \equiv s[x \mapsto 5]$, and since it does not return anything, $o^v(s) \equiv nil$. An operation $\langle p^s, p^v \rangle$ which returned the value of x, without changing any database state, would be defined as $p^s(s) \equiv s$ and $p^v(s) \equiv s[x]$.

An operation $\langle o^s, o^v \rangle$ is *independent* of key k iff $\forall s \in S, v \in V$,

$$o^s(s[k \mapsto v]) = (o^s(s))[k \mapsto v] \quad \text{and} \quad o^v(s[k \mapsto v]) = o^v(s).$$

In other words, o never modifies key k, nor does it read its value. If an operation is not independent of a key k, it is *dependent* on k. Two operations $\langle o^s, o^v \rangle$ and $\langle p^s, p^v \rangle$ are independent if their sets of dependent keys do not intersect.

Two operations a and b *commute* if and only if for any database state s,

$$a^s(b^s(s)) = b^s(a^s(s)) \quad \text{and} \quad a^v(s) = a^v(b^s(s)) \quad \text{and} \quad b^v(s) = b^v(a^s(s)).$$

That is, their execution on any database state, in either order, results in the same new database state and the same return values.

It is also useful to reason about *classes* of operation that all commute. For example, any two operations that only read the database commute, since the operations' state functions are the identity. We can also prove that operations which act on different keys always commute, as follows:

Lemma. Two operations that are independent commute.

Proof sketch. Two operations o and p that are independent operate on different, non-intersecting sets of keys. p never reads any keys that o modifies, so p^v must produce the same result on $o^s(s)$ as on s, and the same holds for o^v on $p^s(s)$. Since o and p read and write different sets of keys, $o^s(p^s(s)) = p^s(o^s(s))$. Thus, o and p commute.

4.1.2 Transactions and histories

Now we describe the transaction model we use to argue about correctness. A *transaction* is a sequence of operations, each of which operates on the database. In execution, operations from different transactions are interleaved. (Operations might also execute in parallel, but we assume here that the effects of parallel operations are always the same as those operations in some serial order—that is, that operations implement proper local concurrency control, such as fine-grained locks. This allows us to focus on the more difficult problem of transaction serializability.)

A *history* h is a sequence of operations. For a history $h = [o_1, \ldots, o_n]$ we define two functions: $h^s(s) = o_n^s(o_{n-1}^s(\cdots(o_1^s(s))\cdots))$ and $h^v(i,s) = o_i^v(o_{i-1}^s(\cdots(o_1^s(s))\cdots))$.

Two histories $h = [o_1, \ldots, o_n]$ and $h' = [o'_1, \ldots, o'_n]$ are *equivalent* if and only if for any database state s:

- They contain the same operations. That is, there exists a permutation π over $\{1, \ldots, n\}$ where for all i with $1 \le i \le n$, $o_i = o'_{\pi(i)}$.
- Corresponding operations have the same return values. That is, for all i with $1 \le i \le n$, $h^v(i,s) = h'^v(\pi(i),s)$.
- The final state of each history is the same. That is, $h^s(s) = h'^s(s)$.

Clearly h is equivalent to itself. Furthermore, if h and h' are equivalent, then for any other history p, the concatenated histories $p \,\|\, h$ and $p \,\|\, h'$ are equivalent, as are the histories $h \,\|\, p$ and $h' \,\|\, p$.

A history is a *transaction history* for some set of committed transactions T if and only if the folowing properties hold:

- The history contains exactly the operations from the set of transactions T; there is a one-to-one mapping between operations in h and operations in T. Since aborted transactions do not return values and operations in aborted transactions do not affect database state, from here we will assume all transactions from T committted.
- If operations appear in some order in a single transaction $t \in T$, they are in the same order in h. In other words, h restricted to the subhistory of transaction t, $h|t$, has the operations of t in transaction order.

A history h is a *serial history* of a set of transactions T if and only if h is a transaction history for T, and no two transactions in h are interleaved.

Given this model, we can begin to reason about the ways in which commutativity affects the equivalence of different transaction histories. For example, this lemma will be useful:

Lemma (commutative equivalence). Given a history $h = [o_1, \ldots, o_m, p, p', q_1, \ldots, q_n]$ where operations p and p' commute, h and $h' = [o_1, \ldots, o_m, p', p, q_1, \ldots, q_n]$ are equivalent.

Proof sketch. Consider the two histories $[p, p']$ and $[p', p]$. These two histories are equivalent—a direct consequence of the definition of commutativity. The lemma follows from the equivalence facts above.

4.1.3 Doppel execution

Doppel executes transactions in phases. Every transaction executes in exactly one phase—split or joined—and never straddles phases. Doppel's concurrency control algorithm for joined phases is exactly optimistic concurrency control, which generates serializable results. Thus, when Doppel executes a set of transactions T in a joined phase, any history h it generates for those transactions will be equivalent to a serial history of T.

Doppel's concurrency control algorithm for split phases is a different story. We wish to show that our algorithm for executing transactions in Doppel's split phase is serializable: that when Doppel executes a set of transactions T in a split phase, any history h it generates for those transactions will be equivalent to a serial history of T.[1] But split-phase execution is not equivalent to OCC. In the rest of this section, we first model the histories actually produced by Doppel in split phases, and then show that each such history is equivalent to a serial history.

At the beginning of a split phase, for each core and split key, Doppel initializes the key's per-core slice to *nil*. During the split phase, operations on split keys are recorded as deltas; these deltas modify the per-core slice (and read or write no other database state) and return *nil*. At the end of the split phase, for each core and split key, Doppel executes the corresponding reconciliation operation that merges the per-core slice into the database state, then erases the per-core slice.

Doppel requires the following rules hold for split-phase execution:

1. During a split phase, there is a set of split records, $s_{\text{split}} \subseteq s$.

2. All operations that depend on split records depend on at most one key.

[1]Note that by "Doppel" we mean "an ideal implementation of Doppel's concurrency control algorithms that doesn't have any implementation bugs."

3. All operations that depend on split records commute. This is because for each split record, Doppel will use an execution plan that is specialized for a single commutative operation class. That is, every operation that executes during the split phase on the record commutes with all other operations that might execute on the record; if any other operation is attempted, the relevant transaction is aborted and delayed to the next joined phase.

4. No operation that depends on a split record returns a value.

5. When a core receives an operation that depends on a split record, it does not apply it to the database. Instead, it applies the delta for that split operation to its per-core slice.

6. At the end of the split phase, Doppel reconciles the per-core slices with the database. Each core's reconciliation operations are its final operations in the phase; it executes no other operations.

Example split phase history

We first use an example to demonstrate the effects of these rules, and then argue about general properties of histories generated by Doppel's execution in split phase.

Consider two transactions $t_1 = [a,b,c]$ and $t_2 = [d,e,f]$, where a through f are operations. Executing t_1 and t_2 in a joined phase might produce a history like the following:

$$h = [a,d,b,c,e,f]$$

The operations from different transactions are interleaved, but since Doppel's joined phase uses OCC and is serializable, h is equivalent to some serial history.

Now assume that b and e are compatible split operations on the same key, k. If Doppel decided to split k for operations like b and e, executing t_1 and t_2 in a split phase might produce a history like the following (assuming that t_1 executes on core 1 and t_2 on core 2):

$$h_{\text{split}} = [i_1, i_2, a, d, b_{\text{delta}}, c, e_{\text{delta}}, r_1, f, r_2]$$

Here, the i operations initialize the per-core slices, and the r operations are reconcile operations; there is one i and one r per core and per split key. The i and r operations do not correspond to any transactions.

Properties of split-phase operations

Any operations in a split-phase history have the following properties. Assume that o is a normal joined operation (e.g., a or d in our example), and σ_{jk} is an operation on core j on split key k (e.g., i_1, e_{delta}, or r_2).

- Any o operation in a split phase is independent of, and therefore commutes with, all σ_{jk} operations in that split phase. This is because all operations on a split key k in a split phase are executed in split mode.
- Any σ_{jk} and $\sigma_{jk'}$ operations on different keys are independent, and therefore commute.
- Any delta operations δ_{jk} and $\delta_{j'k'}$ commute. If they are on different cores and/or keys, this is because they are independent; if they are on the same core and key, this is because the underlying operation class is commutative.
- Any reconciliation operations r_{jk} and $r_{j'k}$ on different cores, but the same key, commute. This is true even though the operations affect the same database state because the underlying operation class is commutative.
- Let $a_1, \ldots, a_m$ be a sequence of split operations on core j and key k. Then the sequence of operations

  $$h = [i_{jk}, (a_1)_{\text{delta}}, \ldots, (a_m)_{\text{delta}}, r_{jk}]$$

  is essentially equivalent to the sequence of operations

  $$h' = [a_1, \ldots, a_m],$$

  in the following sense: Given any initial database state s, the states $h^s(s)$ and $h'^s(s)$ are equal; and the return value of each $(a_i)_{\text{delta}}$ is the same as that of a_i. (Essential equivalence is the same as equivalence, except that it allows the system-generated operations i_{jk} and r_{jk} to be dropped.) This is true by the definition of Doppel's delta and reconciliation operations: Doppel ensures that a set of deltas, plus the corresponding reconciliation, has the same effect on database state as the original operations; and both the original operations and the deltas return nil.

Split equivalence

These properties let us show that split-phase histories are essentially equivalent to simpler histories, in which the *original* operations—not their deltas—execute out of transaction order at the end of the history.

Lemma (split equivalence). Let T be a set of transactions executing in a Doppel split phase, and let T' equal T with all split-mode operations omitted. (Thus, in our example,

$T = \{[a,b,c],[d,e,f]\}$ and $T' = \{[a,c],[d,f]\}$.) Then Doppel's execution of T will produce a history that is equivalent to a history $h = [o_1,\dots,o_m,p_1,\dots,p_n]$, where the o_i are the operations of T' in an order compatible with an OCC execution of T', and the p_i are the split-mode operations in T in any order.

Proof sketch. We sketch a proof with reference to h_{split}, our example history, but the argument applies to any split-phase history. Recall that

$$h_{\text{split}} = [i_1, i_2, a, d, b_{\text{delta}}, c, e_{\text{delta}}, r_1, f, r_2].$$

First, consider the restriction of h_{split} to joined-mode operations, which is

$$h_{\text{joined}} = [a,d,c,f].$$

Doppel executes transactions in split phases using OCC, just as it does in joined phases, with the exception that split-mode operations do not participate in the validation protocol. Put another way, Doppel in split phase acts like OCC on T'. So we know that h_{joined} contains the operations of T' in an order compatible with OCC execution.

Now, using repeated invocations of commutative equivalence and the properties of split-phase operations, we can show that h_{split} is equivalent to h_{reorder}, which orders all split operations at the end of execution, grouped by core and key:

$$h_{\text{reorder}} = [a,d,c,f,i_1,b_{\text{delta}},r_1,i_2,e_{\text{delta}},r_2]$$

The last property of split-phase operation then shows that the reordered execution is essentially equivalent to the following execution without delta operations:

$$h_{\text{simple}} = [a,d,c,f,b,e]$$

Although this execution has fewer operations than the original, it's reasonable to consider it equivalent to the original, since the omitted *i* and *r* operations were introduced by Doppel and do not correspond to user-visible transactions. h_{simple} starts with h_{joined}, and ends with the split-mode operations in some order, so the lemma is proved.

4.1.4 Serializability of split-phase execution

Now we can argue that Doppel's execution strategy during a split phase produces serializable results.

Theorem. Doppel's operation in split phases is serializable: any history produced by executing a set of transactions T in a split phase is essentially equivalent to a serial history of those transactions.

Proof sketch. Consider a single Doppel split phase that executes a set of transactions T starting from a database state s, generating some history h. Some of the keys are split.

By the split equivalence lemma above, h is essentially equivalent to a new history h_1 that looks like the following:

$$h_1 = [o_1, \ldots, o_m, p_1, \ldots, p_k]$$

Each o_i operation is independent of all split keys and each p_j is an operation from the commutative class associated with its split key.

We show that h_1 is equivalent to a serial history. T' is the set of transactions from T with their split operations omitted. Doppel uses OCC for all operations when executing T', which would produce a history equivalent to a serial history of the transactions in T'. Operations on split keys do not affect Doppel's OCC behavior. This means the joined operation prefix of h_1:

$$h_{\text{joined}} = [o_1, \ldots, o_m]$$

is equivalent to a serial history of T':

$$h_{\text{serial}} = [q_1, \ldots, q_m]$$

where the q_i are the o_i permuted into some serial transaction order.

This in turn means that $[o_1, \ldots, o_m, p_1, \ldots, p_k]$ is equivalent to $[q_1, \ldots, q_m, p_1, \ldots, p_k]$. Since the p_j commute with each other and with the o_i (q_i), we can reorder them arbitrarily without affecting equivalence. In particular, we can reorder them into an order consistent with the order of operations in each $t \in T$.

Transactions in T that had no operations on joined keys can be reordered in between other transactions. This produces a serial transaction history of T, which means h_1 is equivalent to a serial history of T and h is essentially equivalent to a serial history of T. $\square$

4.1.5 Serializability of alternating split and joined phases

Doppel's joined phase is serializable because it uses OCC. We just showed that Doppel's split phase is serializable. Since no transactions ever execute over phase boundaries, all of the transactions in the next split phase will start after all joined phase transactions complete, and all transactions in a following joined phase will start and finish after all split phase transactions complete. Let X be a serial history for a split phase, and Y a serial history for the following joined phase. A serializable order for multiple phases is a concatenation of any of the serializable orders produced for each phase, so in this case a serial history for the two phases is $X \parallel Y$. This means the combination of the split and joined phases is serializable.

4.2 Efficient operation implementations

Commutativity is a sufficient condition to run a split operation correctly in the split phase. However, there is a distinction between operations Doppel can execute correctly in the split phase, and operations for which Doppel gets parallel speedup. This section describes when it is possible to achieve good performance.

Commutative operations can execute correctly on per-core slices, but for some operations, the process of reconciling the per-core slices can take time linear in the number of operations. Running the operations one at a time when reconciling is not likely to yield much parallel speedup.

An example of an operation that we do know how to execute with parallel speedup is $\textsc{Max}(k,n)$, which replaces k's value with the max of the value and n. $\textsc{Max}(k,n)$ acts on integer values, assigns $v[k] \leftarrow \max\{v[k],n\}$, and returns nothing. In order to execute many of these operations in parallel, each core initializes per-core slices $c_j[k]$ with the global value $v[k]$. When a core j executes an operation $\textsc{Max}(k,n)$, it actually sets $c_j[k] \leftarrow \max\{c_j[k],n\}$. To merge the per-core slices, a core applies $v[k] \leftarrow \max_j c_j[k]$. With $\textsc{Max}(k,n)$, merging the per-core slices takes $O(j)$ time, where j is the number of cores.

An example of an operation where Doppel does not get parallel speedup is $\textsc{Sha1}(k)$, which sets k's value to the $\textsc{Sha1}$ of its current value. Doppel could execute $\textsc{Sha1}(k)$ correctly in the split phase, like so: Each core initializes its own per-core slices $c_j[k]$ to 0. When a core j executes an operation $\textsc{Sha1}(k)$, it actually sets $c_j[k] \leftarrow c_j[k]+1$. When a core wishes to read k, it must merge the per-core slices by executing $v[k] \leftarrow \textsc{Sha1}^{\Sigma_j c_j[k]}(k)$. Though this scheme is correct, it is unlikely to achieve parallel speedup; reconciliation is $O(n)$ where n is the number of $\textsc{Sha1}(k)$ operations. We do not know of a good way to execute many instances of $\textsc{Sha1}(k)$ on the same record in parallel. Because of this, Doppel cannot get a parallel speedup with this operation, even though it commutes.

Definition. A *summary a* for a sequence of commutative operations $[o_1, o_2, ..., o_n]$ is an operation such that for all database states $s \in S$, $a^s(s) = o_n^s(o_{n-1}^s \cdots (o_1^s(s)))$. If for each possible sequence of operations in commutative class O there is a summary a that we can compute in constant time and that runs in constant time with respect to the number of operations in the sequence, we say O can be *summarized.*

The key difference between $\textsc{Max}(k,n)$ and $\textsc{Sha1}(k)$ is that $\textsc{Max}$ can be summarized. When we can summarize a set of operations (such as all $\textsc{Max}$ operations), Doppel can perform the computation of the operations in the delta function, instead of just recording the operations to perform in the merge function. When reconciling $\textsc{Max}$, Doppel runs max on each $c_j[k], v[k]$, so it runs a total of j $\textsc{Max}$ operations. When reconciling $\textsc{Sha1}$, Doppel must run $\textsc{Sha1}$ n times serially, where n is the total number of $\textsc{Sha1}$ operations issued. If an operation can be summarized, then during a split-phase execution with n operations on a split key, Doppel can record a summary function on each core for the sequence of split operations that have executed on that core so far. During reconciliation, instead of executing the n split operations, Doppel executes j summary functions, and thus the time to reconcile the per-core values is $O(j)$ instead of $O(n)$.

In summary, Doppel requires operations on split data in the split phase to execute on a single record, not return a value, and to be commutative. Commutativity is required so that no matter in what order operations are applied to per-core slices, the order will match a valid order of their containing transactions. If the operation can be summarized, Doppel can get parallel performance.

4.3 Limitations

One situation where Doppel does not help is when a sequence of operations on a record commute due to their arguments, though the operations with all possible arguments do not commute. For example, during a split phase, $\textsc{Mult}(k,1)$ operations multiply a record's value by one, while $\textsc{Add}(k,0)$ operations add zero to the value. Technically these commute; all operations leave the value unchanged and none return a value, so they can be performed in any order, with the same result. But Doppel will not execute such a sequence of operations on per-core values during one split phase. Doppel does not detect commutativity; instead, Doppel knows that certain operations *always* commute, and if the developer uses those operations in transactions, Doppel may execute them in the split phase.

Our demand that operations commute rules out some situations in which parallel speed-up would be possible. For example, suppose in a set of transactions, each performs only blind writes to a specific record. These blind writes do not commute, yet they could be implemented in a way that achieves parallel speedup; statically analyzing the set of running transactions would show that it is correct to execute those writes in parallel and then choose a "winning" writer depending on how the transactions were ordered based on the other operations in the transactions. Doppel does not allow this, because Doppel cannot execute non-commutative operations on per-core slices.

Doppel does not parallelize within an operation; Doppel only gets parallel speedup with many different instances of operations. For example, consider storing matrices in records.

Though multiple matrix multiplication operations on the same record do not commute, a single matrix multiplication operation can be performed in parallel on many cores; Doppel will not do this.

Finally, a system like Doppel might be able to achieve a performance improvement for even non-commutative operations, but without parallel execution. For example, split-phase execution could log updates to per-core slices, with the merge step applying the logged updates in time order; this would cause those updates to execute serially. In some cases this is faster than executing the operations using locking in parallel on multiple cores, for example by avoiding cache line transfers relating to the contended data.

4.4 Operations in Doppel

Doppel supports traditional GET and PUT operations, which are not splittable:

- GET(k) returns the value of k.
- PUT(k,v) overwrites k's value with v.

Doppel's current set of splittable operations is 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 a commutative form of PUT(k,x). It operates 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 be added (for instance, multiply).

All operations in Doppel access only one record. This is not a functional restriction since application developers can build multi-record operations from single-record ones using transactions. It is an open question whether or not Doppel could be extended to work with multi-record operations on split records in split phase.

FIVE

Doppel 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.

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. Workers read and write to a shared store, which is a key/value map, using per-key locks. The map is implemented as a concurrent hash table. All workers have per-core slices for the split phases.

Developers write transactions 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 chapter 4. 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 chapter 7.

Our implementation does not currently provide durability, and is not fault tolerant. Existing work suggests that asynchronous batched logging could be incorporated with phase reconciliation without becoming a bottleneck [34, 52, 58].

5.1 Measuring contention and stashed operations

Doppel samples the number of operations and the number of conflicts for each record in a given phase. These are kept in per-worker storage, and during every 10th phase change, are accumulated. At this time the formula in Chapter 3 is evaluated, and records are moved from split to joined, or vice versa. If Doppel is not changing phases because no records are split or no transactions are stashed, then it collects per-worker statistics and evaluates the formula every ten phase lengths.

Doppel also samples the number of stashed transactions for a given split phase. Each worker keeps a separate count. If a worker's count rises past some threshold, it asks the coordinator to initiate a phase change. If enough workers have reached this level of stashed transactions, the coordinator initiates an early phase change.

5.2 Changing phases

Doppel uses channels to synchronize phase changes and acknowledgments between the coordinator and workers. 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 over a channel, and wait for permission to proceed. When all workers have acknowledged the change, the coordinator releases them over another channel, and workers start executing transactions in split mode. A similar process ends the split phase to begin reconciliation. When a split-phase worker notices a global variable update, it stops processing transactions, merges its per-core slices with the global store, and then acknowledges the merge on a channel 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. Because of the coordination, Doppel briefly pauses transaction processing 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.

5.3 Reconciling operations

Workers merge their per-core slice for a record by using the record's selected operation's reconcile function. Each worker locks the joined record, applies its per-core slice, and then unlocks the joined record. Workers reconcile concurrently with other workers which might still be processing split-phase transactions; this interleaving is still correct because each reconcile function is run atomically, the per-core slices can be reconciled in any order, and a worker will not process any transactions in the split phase after it has reconciled. Doppel provides initialization, split, and reconcile implementations for the built-in splittable operations. Figure 5-1 shows two reconcile functions, one for $\textsc{Max}(k,n)$ and one for $\textsc{OPut}(k,v)$.

```
func max-reconcile(j int, k Key) {
  val := local[j][k]
  g-val := v[k]
  v[k] = max(g-val, val))
}

func oput-reconcile(j int, k Key) {
  order, val := local[j][k]
  g-order, g-coreid, g-val := v[k]
  if order > g-order ||
      (order == g-order && j > g-coreid) {
    v[k] = order, j, val
  }
}
```

Figure 5-1: Doppel MAX and OPUT reconcile functions.

SIX

Application experience

This chapter describes our experience implementing an auction site, RUBiS, to use Doppel's operations and transaction model. This experience shows that Doppel's operations are powerful enough to use for real applications, and that applications have many opportunities to use operations that can be split on contentious records. An interesting takeaway is that only a few records cause most of performance problems related to contention in RUBiS, yet it only applies one or two different operations on these contended records.

6.1 RUBiS

We used RUBiS [7], 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 two 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 6-1.

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. Unfortunately, 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 can only use one selected operation in a split phase, so as written in Figure 6-1 the transaction would have to execute in a joined phase, and would not benefit from local per-core operations.

```
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 6-1: 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 6-2: Doppel `StoreBid` transaction.

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 6-2 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 6-2, 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 6-2 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(k,n) on the `userRating`.

6.2 Discussion

This example shows how Doppel's commutative operations allow seemingly conflicting transactions to be re-cast in a way that allows concurrent execution. This pattern appears 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.

SEVEN

Performance evaluation

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

- Doppel increases throughput for transactions with conflicting writes to split data (§7.2).
- Doppel can cope with changes in which records are contended (§7.3).
- Doppel makes good decisions about which records to split when key popularity follows a smooth distribution (§7.4).
- Doppel can help workloads with a mix of read and write transactions on split data (§7.5).
- Doppel transactions that read split data have high latency (§7.6).
- Doppel increases throughput for a realistic application (§7.8).

7.1 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. 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 according to the algorithm described in §3.5. Doppel uses the technique described in §3.6 to determine which data to split. The benchmarks omit many costs associated with a real database; for example we do not incur any costs related to network, RPC, or disk. There is evidence that contention could be the bottleneck in a system that had network, RPC, and disk costs [38]; however, in a full system, the relative cost of contention would be lower.

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.

7.2 Parallelism versus conflict

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

INCR1 microbenchmark. The database contains 1M 16-byte keys with integer values. One of these keys is designated "hot." A configurable fraction of the transactions increment the hot key (that is, they increment the value stored under the hot key), while the remaining transactions each increment a key chosen uniformly from the remaining keys. In all transactions, the increment is done using an $\text{ADD}(k,1)$ operation.

This experiment compares Doppel with OCC, 2PL, and a system called Atomic. To execute an increment, OCC reads the key's value, computes the new value, and then at commit time, tries 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, OCC aborts the transaction and saves it to try again later. Doppel also behaves this way in joined phases (and in split phases for non-split keys). 2PL waits for a write lock on the key, reads it, and then writes the new value and releases the lock. 2PL never aborts. Atomic uses an atomic increment instruction with no other concurrency control. Atomic represents an upper bound for schemes that do not use split data.

Figure 7-1 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 7-1, 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 7-1, OCC, 2PL, and Atomic provide decreasing total throughput. The high-level reason is that they must execute operations on the hot key sequentially, on only one core at a time. Thus OCC and 2PL's throughput ultimately drop by roughly a factor of 20, as they move from exploiting 20 cores to doing useful 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

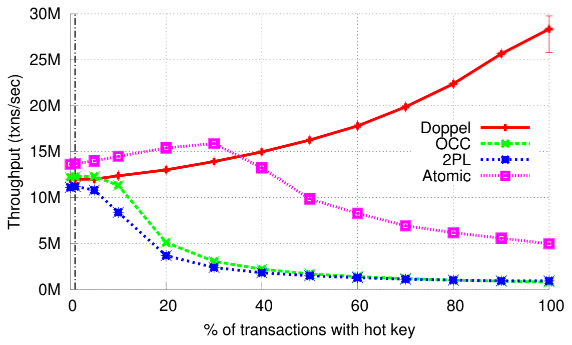


Figure 7-1: Total throughput for INCR1 as a function of the percentage of transactions that increment the single hot key. 20 cores. The vertical line indicates when Doppel starts splitting the hot key (x=2%).

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 7-1 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 7-1, 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$\times$ better than Atomic, 19$\times$ better than 2PL, and 38$\times$ 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 7-1, 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 7-2 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 sequential 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 7-1 shows that when even a small fraction of transactions write the

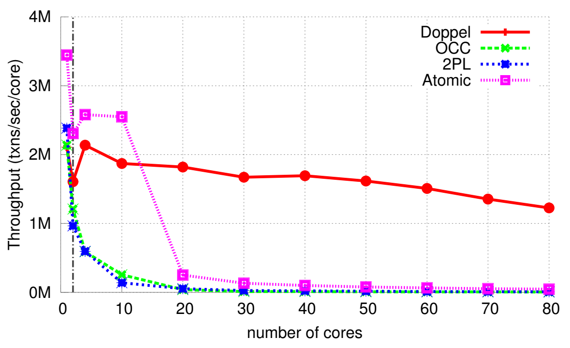


Figure 7-2: Throughput per core for INCR1 when all transactions increment a single hot key. The y-axis shows per-core throughput, so perfect scalability would result in a horizontal line. The dashed line is the number of cores where Doppel starts splitting the hot key, here x=2.

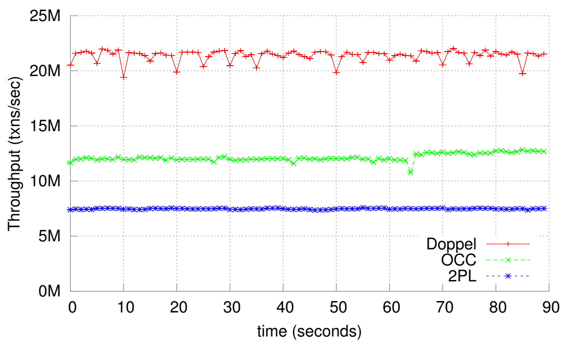


Figure 7-3: Throughput over time on INCR1 when 10% of transactions increment a hot key, and that hot key changes every 5 seconds.

same key, Doppel can help performance. It does so by parallelizing update operations on the popular key.

7.3 Changing workloads

Data popularity may change over time. Figure 7-3 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.

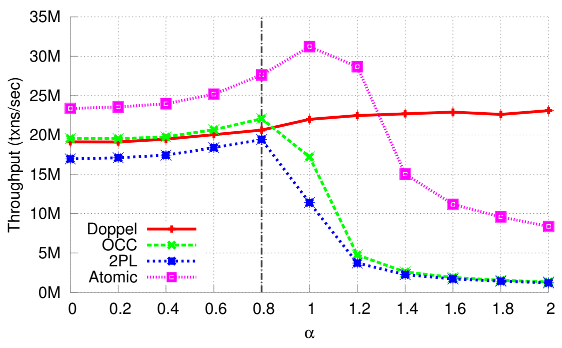


Figure 7-4: Total throughput for INCRZ as a function of α (the Zipfian distribution parameter). The skewness of the popularity distribution increases to the right. 20 cores. The vertical line indicates when Doppel starts splitting keys.

7.4 Classifying records as 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 7.2 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 a sensible number (if any) of most-popular keys to split.

This experiment uses a Zipfian distribution of popularity, in which the *k*th most popular item is accessed in 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 7-4 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 7-1.

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 emerges. Doppel does not split any keys in this region, and hence provides throughput similar to that of OCC.

Figure 7-4 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 7.1 shows some example popularities. Table 7.2 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

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 7.1: 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 7.2: The number of keys Doppel moves for different values of α in the INCRZ benchmark.

would return to splitting just one key.

In summary, Doppel's steady performance as α changes indicates that it is doing a reasonable job of identifying good candidate keys to split.

7.5 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,

```
func ReadLike(user, page int) (int, int) {
  user_key := UserLastLikeKey(user)
  v1 := GET(user_key)
  page_count_key := PageCountKey(page)
  v2 := GET(page_count_key)
  Commit()  // might abort
  return v1, v2
}
```

```
func Like(user, page int) {
  user_key := UserLastLikeKey(user)
  Put(user_key, page)
  page_count_key := PageCountKey(page)
  Add(page_count_key, 1)
  Commit()  // applies writes or aborts
}
```

Figure 7-5: LIKE read and write transactions.

workload, we use the LIKE benchmark

LIKE. The LIKE benchmark 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. Figure 7-5 shows examples of these transactions. With a high level of skew, this application explores the case where there are many users but only a few popular pages; thus the increments of page counts often conflict, but are also read frequently.

The database contains a row for each user and a row for each 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. 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 7-6 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.

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, where 70% of transactions are reads, is tricky for Doppel because the split keys are read even more

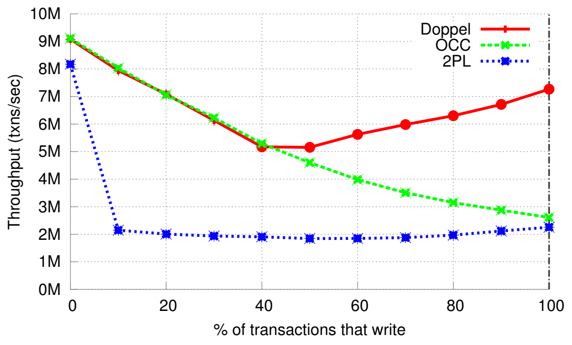


Figure 7-6: Throughput of the LIKE benchmark with 20 cores as a function of the fraction of transactions that write, $\alpha = 1.4$.

Uniform workload

	Mean latency	**99% latency**	**Txn/s**
Doppel	1µs R / 1µs W	1µs R / 2µs W	11.8M
OCC	1µs R / 1µs W	1µs R / 2µs W	11.9M
2PL	1µs R / 1µs W	2µs R / 2µs W	9.5M

Skewed workload

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

Table 7.3: 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.

than they are written, so many read transactions have to be stashed. Figure 7-6 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.

7.6 Latency

Doppel stashes transactions that 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 7.3 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 sequential 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.

7.7 Phase length

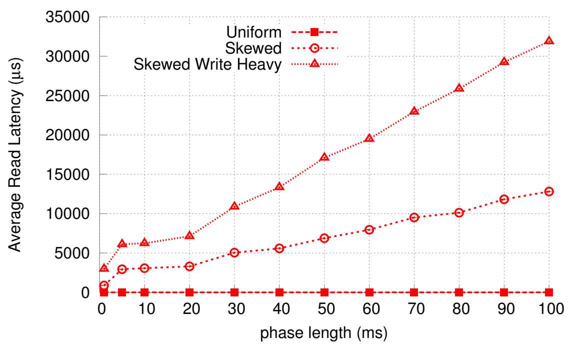
When a transaction tries to read split data during a split phase, its expected latency is determined by the phase length; a shorter phase length results in less latency, but potentially lowered throughput. Figure 7-7 shows how phase length affects read latency and throughput on three LIKE workloads. "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.

The first graph in Figure 7-7 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 the second graph in Figure 7-7 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.

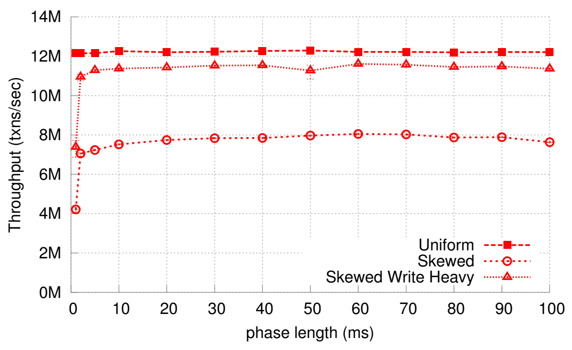
7.8 RUBiS

Do Doppel's techniques help in a complete application? We measure RUBiS [7], an auction Web site implementation, to answer this question.

Section 6 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 `View-BidHistory` to read from top-K set records as indexes. This means Doppel can potentially mark auction metadata as split data. The implementation includes only the database



(a) Average read transaction latencies with varying phase length.



(b) Throughput with varying phase length.

Figure 7-7: Latency and throughput in Doppel with the LIKE benchmark. Each includes a uniform workload, a skewed workload with 50% reads and 50% writes, and a skewed workload with 10% reads and 90% writes. 20 cores.

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

Table 7.4: 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.

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 7.4 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 7-8 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

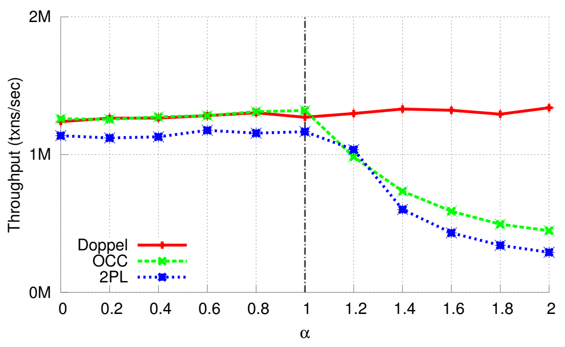


Figure 7-8: The RUBiS-C benchmark, varying α on the x-axis. The skewness of the popularity distribution increases to the right. 20 cores.

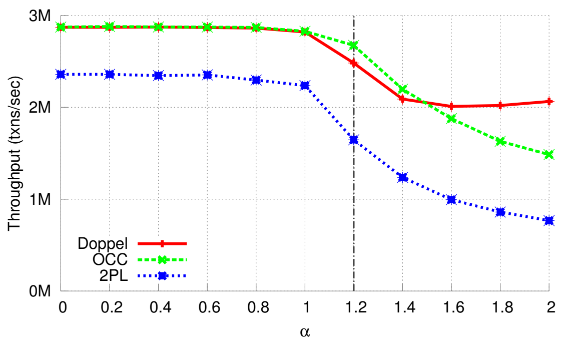


Figure 7-9: The RUBiS-B benchmark, varying α on the x-axis. The skewness of the popularity distribution increases to the right. 80 cores.

$\alpha = 1$. Afterwards, Doppel gets higher performance than OCC. When $\alpha = 1.8$ Doppel gets approximately $3\times$ the performance of OCC and $6\times$ the performance of 2PL.

Doppel's techniques make the most difference for the `StoreBid` transaction, whose code is shown in Figures 6-1 and 6-2. 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 sequentially and get far less parallel speedup.

In Figure 7-8 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 out-perform OCC.

EIGHT

Future directions

The idea of using different execution plans for different records, and transitioning between those plans using phases, could be helpful in the designs of other systems. We have just begun to explore the set of plans that Doppel could execute, and the types of operations that can execute in Doppel's split phase. This chapter describes a few key areas of future work. It includes a discussion of how phases might work in a distributed database, a summary of static and dynamic analysis techniques that could work with Doppel to increase commutativity, a discussion of what kinds of operations Doppel could execute given a source of synchronized time, and how to extend the set of execution plans to include partitioning data.

8.1 Distributed Doppel

Phase reconciliation and execution plans might improve the performance of distributed transactions. What are the key differences between execution plans on multi-core and in a distributed database? One important difference is that the synchronization for phase changing is more expensive on multiple servers, due to network latencies for communication. Also, the phase change protocol currently requires hearing from all participants before moving to the next phase. In a distributed system this would be infeasible because it would need to be tolerant to faults. We would like to explore using consensus protocols like Paxos [33] to coordinate phase changes in a fault-tolerant way.

Another point is that there are other plans that would be useful in a distributed database, but did not make sense in a multi-core database. One example is replicating data for reads. On a multi-core server the cache coherence protocol automatically replicates data among processor caches and keeps cache lines up-to-date. In a distributed system, reading remote records requires a network RPC, and it might make sense to replicate frequently read data to avoid the RPC cost.

8.2 Static and dynamic transaction analysis

Phase reconciliation could be combined with static analysis techniques, like transaction chopping [46, 47], to release locks early and increase concurrency. We could also use static analysis to find commutative parts of application code and turn them into operations. A tool like commuter [20] could help developers identify and increase commutativity in their transactions. Other work on parallelizing compilers does commutativity analysis to generate parallel code [43]. All of these techniques would expand the set of operations that could be executed in parallel in Doppel's split phase.

8.3 Synchronized time

Doppel requires operations to commute in order to execute them correctly on per-core data in the split phase. This is required to obtain an ordering of the operations consistent with their surrounding transactions. But there are other ways to achieve a correct ordering without communication, such as using the synchronized timestamp counters provided by some machines. Reading a synchronized counter would always producing increasing timestamps, even among different cores. If each core had access to always-increasing counters, these could be used as ordered transaction IDs. Per-core operations could be logged with transaction IDs, and later could be applied in transaction ID order to the database state, during reconciliation. For example, Doppel could split $\text{PUT}(k,v)$ operations by keeping a per-core slice on each core that contained a transaction ID and value for the transaction with the highest transaction ID to issue a write to that record. When reconciling, Doppel would overwrite the record's global value with the value from the core that had the highest transaction ID.

8.4 Partitioning

Many databases partition data among cores, and run single-partition transactions without concurrency control. This technique is useful for workloads which have a good partitioning, meaning they do not incur many multi-partition transactions.

This thesis only explored two dynamic execution plans, split and joined. A concurrency control scheme that partitions data among different cores and runs single-partition transactions without validating or locking might perform better than using OCC or 2PL. Doppel could implement a *partition* plan as well as split and joined plans, in which a record is pinned to a core (only that core is allowed to access it) and transactions are directed accordingly. Single-partition transactions that executed entirely on partitioned data could run without the costs of concurrency control.

The partition plan might work well on mixed workloads, where most of the workload is partitionable, but there are a few distributed transactions. The distributed transactions could run in the joined phase using concurrency control, while the single-partition transactions run quickly in the partitioned phase.

NINE

Conclusion

As the scale of transaction processing grows and the cost of in-memory storage drops, databases must be able to take advantage of multiple cores for increased performance. Though some workloads appear to fundamentally serialize because of conflicting operations on the same data, we have shown that many of them actually fit a model where using per-core data can increase parallelism. Per-core data, however, is not a complete solution—incorporating its use in database transactions is challenging due to the mix of workloads databases must execute.

In this dissertation, we introduce phase reconciliation, which provides a framework for using different execution plans on different data, depending upon how it is accessed. The plans in Doppel combine data layout decisions and concurrency control algorithms that shift over time based on workload properties. Our technique reduces conflict in transaction workloads by executing commutative operations in parallel, on multiple cores. We demonstrated Doppel's performance benefits scaling a microbenchmark to 80 cores and implementing RUBiS, an auction web application.

Bibliography

[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] Redis. `http://redis.io`.

[5] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. *ACM SIGMOD Record*, 24(2):23–34, 1995.

[6] Marcos K Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. *ACM SIGOPS Operating System Review*, 41:159–174, 2007.

[7] Cristiana Amza, Anupam Chanda, Alan L. Cox, Sameh Elnikety, Romer Gil, Karthick Rajamani, Willy Zwaenepoel, Emmanuel Cecchet, and Julie Marguerite. Specification and implementation of dynamic web site benchmarks. In *IEEE International Workshop on Workload Characterization*, pages 3–13, 2002.

[8] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. *ACM SIGMOD Record*, 29:261–272, 2000.

[9] Srikanth Bellamkonda, Hua-Gang Li, Unmesh Jagtap, Yali Zhu, Vince Liang, and Thierry Cruanes. Adaptive and big data scale parallel execution in oracle. *Proceedings of the VLDB Endowment*, 6(11):1102–1113, September 2013.

[10] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, 1981.

[11] Philip A. Bernstein and Eric Newcomer. *Principles of transaction processing*. Morgan Kaufmann, 2009.

[12] Philip A. Bernstein, Colin W. Reid, and Sudipto Das. Hyder—a transactional record manager for shared flash. In *Proceedings of the Conference on Innovative Database Research (CIDR)*, Asilomar, CA, January 2011.

[13] Philip A. Bernstein, Colin W Reid, Ming Wu, and Xinhao Yuan. Optimistic concurrency control by melding trees. In *Proceedings of the 37th International Conference on Very Large Data Bases (VLDB)*, Seattle, WA, August–September 2011.

[14] Silas Boyd-Wickizer. *Optimizing Communication Bottlenecks in Multiprocessor Operating System Kernels*. PhD thesis, Massachusetts Institute of Technology, February 2014.

[15] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert T. Morris, and Nickolai Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, October 2010.

[16] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, CK Tan, Odysseas G. Tsatalos, et al. Shoring up persistent applications. In *Proceedings of the 1994 ACM Special Interest Group on the Management of Data (SIGMOD)*, Minneapolis, MN, June 1994.

[17] John Cieslewicz and Kenneth A Ross. Adaptive aggregation on chip multiprocessors. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB)*, pages 339–350, Vienna, Austria, September 2007.

[18] John Cieslewicz, Kenneth A Ross, Kyoho Satsumi, and Yang Ye. Automatic contention detection and amelioration for data-intensive operations. In *Proceedings of the 2010 ACM Special Interest Group on the Management of Data (SIGMOD)*, Indianapolis, IN, June 2010.

[19] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. RadixVM: Scalable address spaces for multithreaded applications. In *Proceedings of the ACM EuroSys Conference*, Prague, Czech Republic, April 2013.

[20] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: designing scalable software for multicore processors. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, Framingham, PA, October 2013.

[21] Jonathan Corbet. The search for fast, scalable counters. `http://lwn.net/Articles/170003/`, May 2010.

[22] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: a workload-driven approach to database replication and partitioning. In *Proceedings of the 36th International Conference on Very Large Data Bases (VLDB)*, Singapore, September 2010.

[23] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM Special Interest Group on the Management of Data (SIGMOD)*, New York, NY, June 2013.

[24] Kapali P. Eswaran, Jim N Gray, Raymond A. Lorie, and Irving L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.

[25] Benjamin Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, New Orleans, LA, February 1999.

[26] Dieter Gawlick. Processing "hot spots" in high performance systems. In *CompCon*, pages 249–251, 1985.

[27] Dieter Gawlick and David Kinkade. Varieties of concurrency control in ims/vs fast path. *IEEE Database Engineering Bulletin*, 8(2):3–10, 1985.

[28] Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, Salt Lake City, UT, February 2008.

[29] Ryan Johnson, Ippokratis Pandis, and Anastasia Ailamaki. Eliminating unscalable communication in transaction processing. In *Proceedings of the 40th International Conference on Very Large Data Bases (VLDB)*, Hangzhou, China, September 2014.

[30] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. Shore-MT: A scalable storage manager for the multicore era. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, New York, NY, 2009.

[31] Alfons Kemper and Thomas Neumann. HyPer: A hybrid OLTP and OLAP main memory database system based on virtual memory snapshots. In *Proceedings of the IEEE International Conference on Database Engineering*, Hannover, Germany, April 2011.

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

[33] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[34] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M Patel, and Mike Zwilling. High-performance concurrency control mechanisms for main-memory databases. In *Proceedings of the 37th International Conference on Very Large Data Bases (VLDB)*, Seattle, WA, August–September 2011.

[35] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, October 2012.

[36] Barbara Liskov. Distributed programming in argus. *Communications of the ACM*, 31(3):300–312, 1988.

[37] Barbara Liskov and Stephen Zilles. Programming with abstract data types. *ACM Sigplan Notices*, 9:50–59, 1974.

[38] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the ACM EuroSys Conference*, Bern, Switzerland, April 2012.

[39] Patrick E O'Neil. The escrow transactional method. *ACM Transactions on Database Systems (TODS)*, 11(4):405–430, 1986.

[40] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. Data-oriented transaction execution. In *Proceedings of the 36th International Conference on Very Large Data Bases (VLDB)*, Singapore, September 2010.

[41] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. Skew-aware automatic database partitioning in shared–nothing, parallel OLTP systems. In *Proceedings of the 2012 ACM Special Interest Group on the Management of Data (SIGMOD)*, Scottsdale, AZ, May 2012.

[42] Andreas Reuter. Concurrency on high-traffic data elements. In *Proceedings of the 1st ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Ottawa, Canada, August 1982.

[43] Martin C Rinard and Pedro C Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(6):942–991, 1997.

[44] Tudor-Ioan Salomie, Ionut Emanuel Subasu, Jana Giceva, and Gustavo Alonso. Database engines on multicores, why parallelize when you can distribute? In *Proceedings of the ACM EuroSys Conference*, Salzburg, Austria, April 2011.

[45] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems*, Grenoble, France, October 2011.

[46] Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. Transaction chopping: Algorithms and performance studies. *ACM Transactions on Database Systems (TODS)*, 20(3):325–363, 1995.

[47] Dennis Shasha, Eric Simon, and Patrick Valduriez. Simple rational guidance for chopping up transactions. *ACM SIGMOD Record*, 21:298–307, 1992.

[48] Ambuj Shatdal and Jeffrey F. Naughton. Adaptive parallel aggregation algorithms. *ACM SIGMOD Record*, 24:104–114, 1995.

[49] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, October 2011.

[50] Michael Stonebraker, Samuel Madden, J. Daniel Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB)*, Vienna, Austria, September 2007.

[51] Michael Stonebraker and Ariel Weisberg. The VoltDB main memory DBMS. *IEEE Data Engineering Bulletin*, 36(2), 2013.

[52] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, Framingham, PA, October 2013.

[53] William Weihl. *Specification and implementation of atomic data types*. PhD thesis, Massachusetts Institute of Technology, 1984.

[54] William Weihl and Barbara Liskov. Implementation of resilient, atomic data types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(2):244–269, April 1985.

[55] William E Weihl. Data-dependent concurrency control and recovery. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing (PODC)*, 1983.

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

[57] Yang Ye, Kenneth A Ross, and Norases Vesdapunt. Scalable aggregation on multicore processors. In *Proceedings of the 7th International Workshop on Data Management on New Hardware*, June 2011.

[58] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, CO, October 2014.