Parallel Execution for Conflicting Transactions

Neha Narula

Thesis Advisors:
Robert Morris and Eddie Kohler

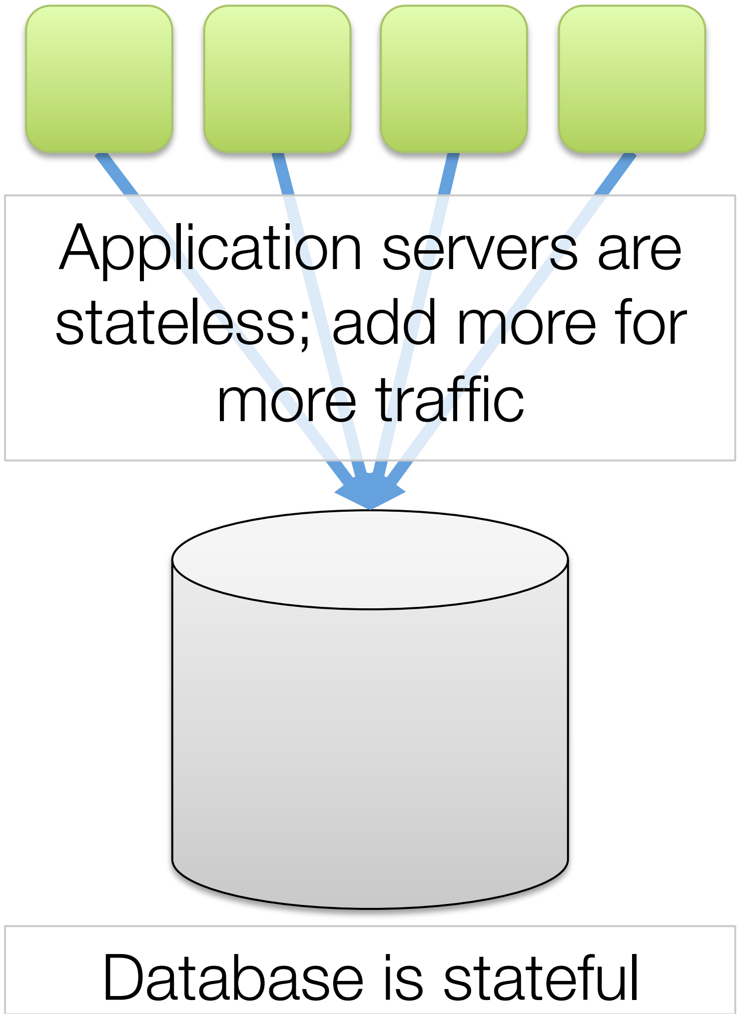
Database-backed applications require good performance

WhatsApp:
- **1M messages/sec**

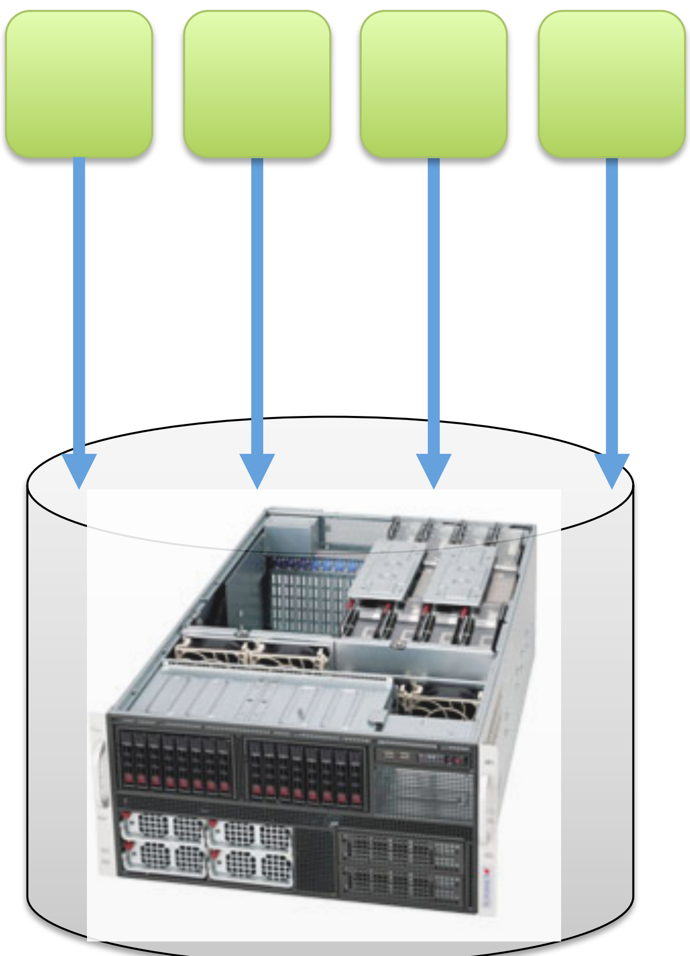
Facebook:
- **1/5 of all page views** in the US

Twitter:
- **Millions of messages/sec** from mobile devices

Databases are difficult to scale



Scale up using multi-core databases



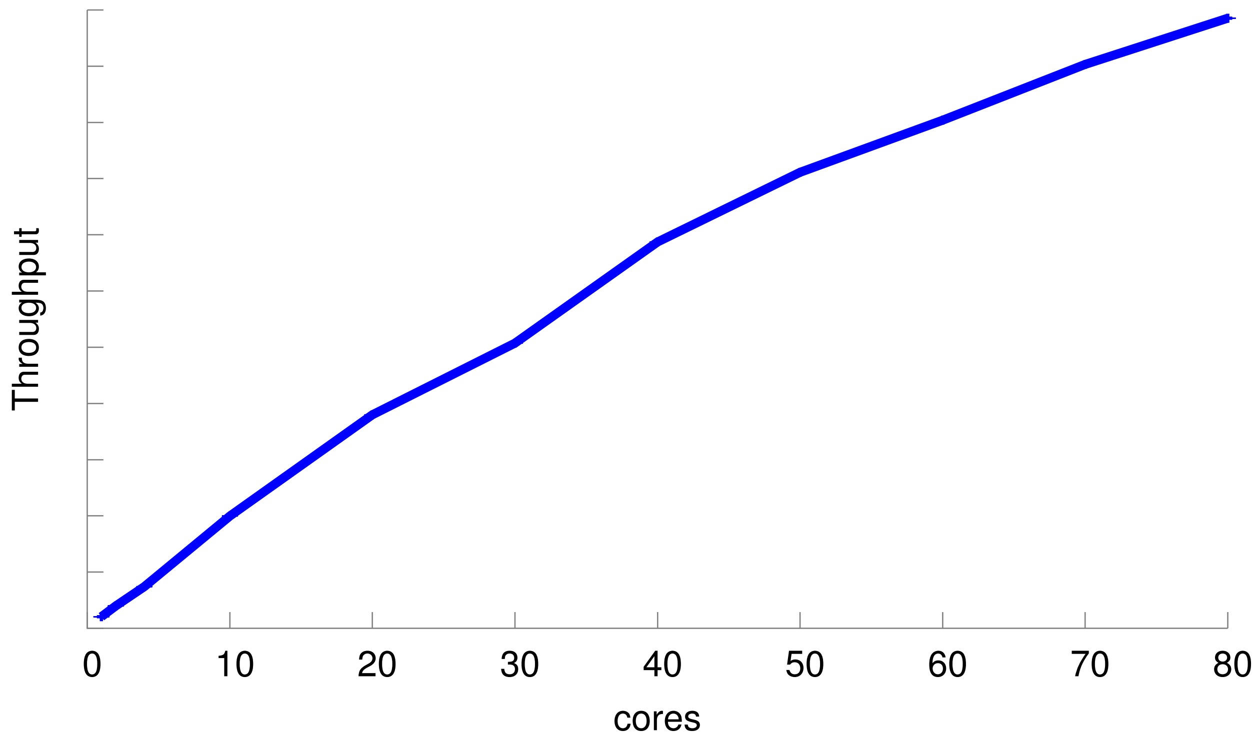
Context

- Many cores
- In-memory database
- OLTP workload
- Transactions are stored procedures

No stalls due to users, disk, or network

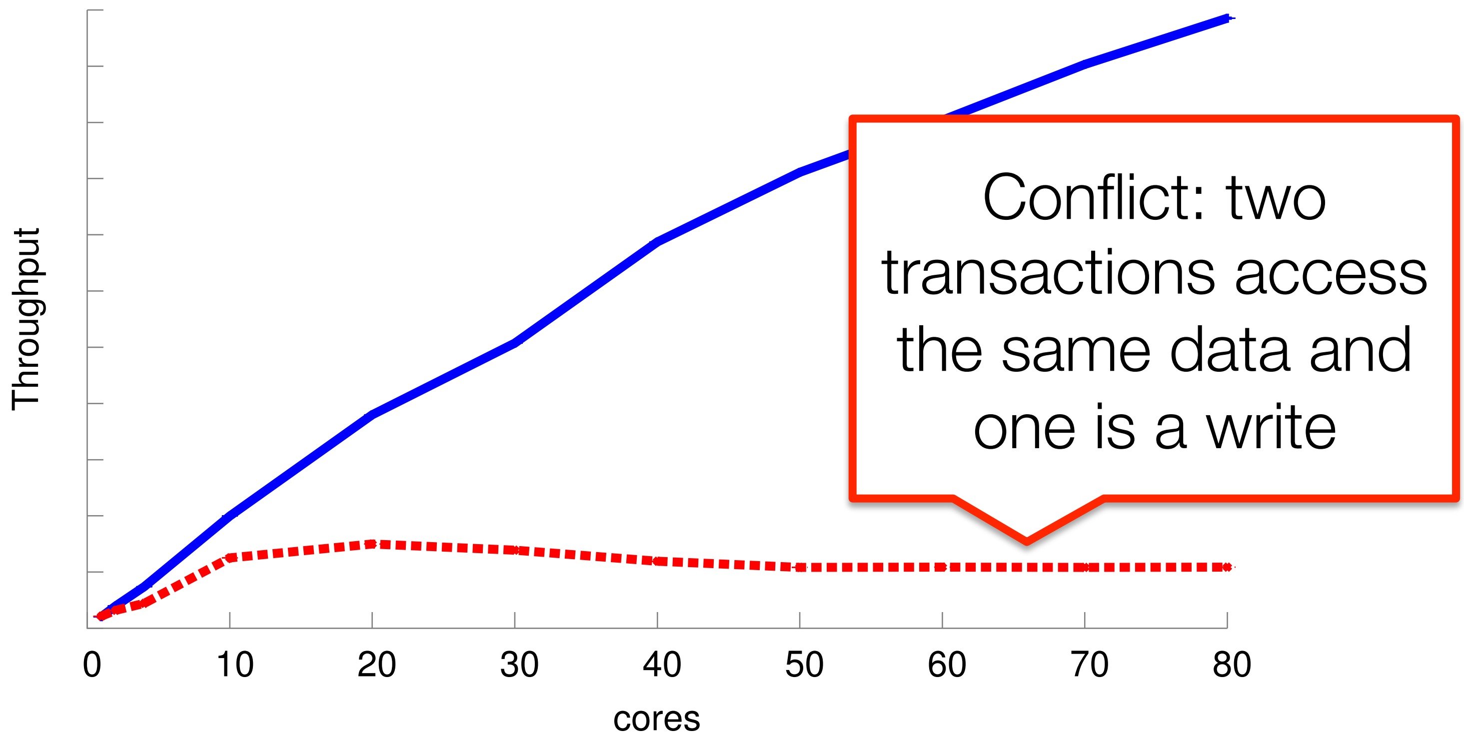
Goal

Execute transactions in parallel



Challenge

Conflicting data access



Database transactions should be serializable

```
TXN1(k, j Key)→(Value, Value) {
    a := GET(k)
    b := GET(j)
    return a, b
}
```

```
TXN2(k, j Key) {
    ADD(k,1)
    ADD(j,1)
}
```

k=0,j=0

To the programmer:

TXN1 TXN2

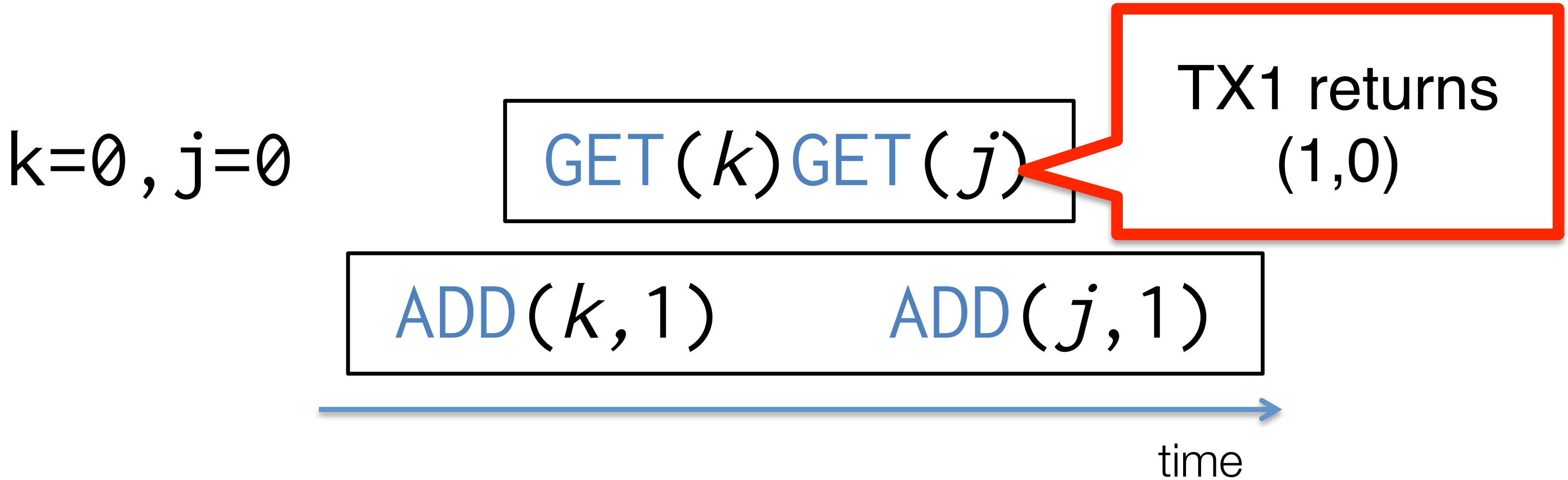
or

TXN2 TXN1

time

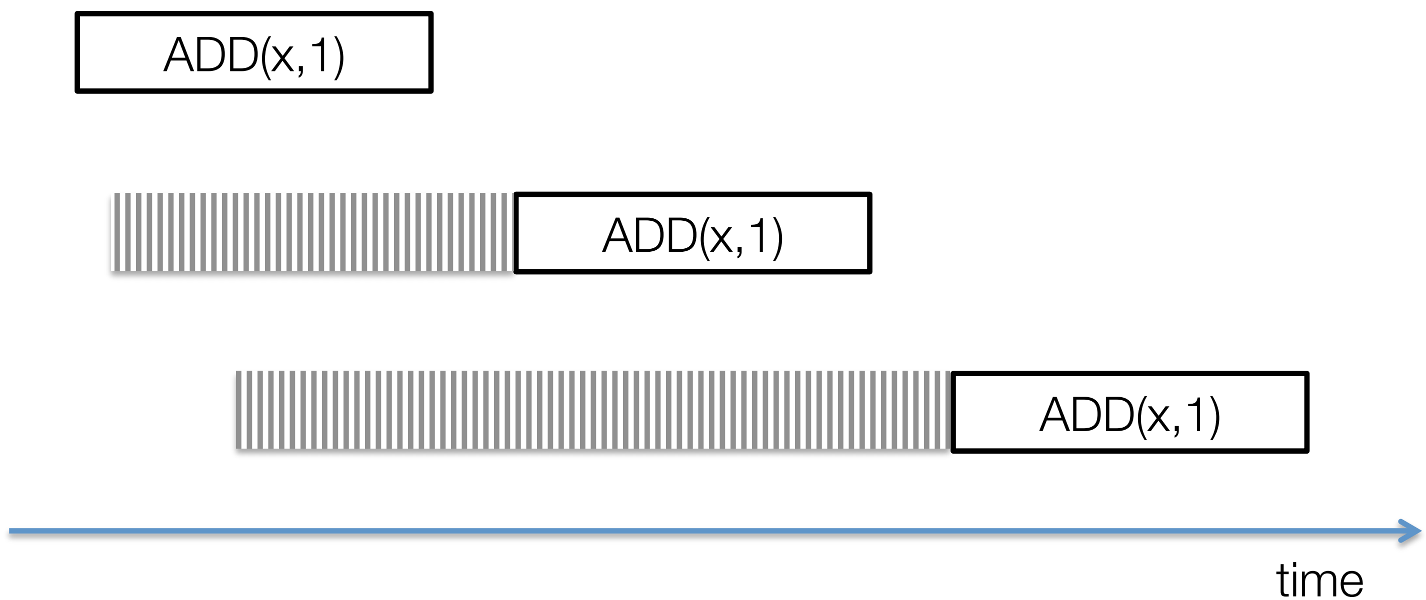
Valid return values for TX1: (0,0) or (1,1)

Executing in parallel could produce incorrect interleavings



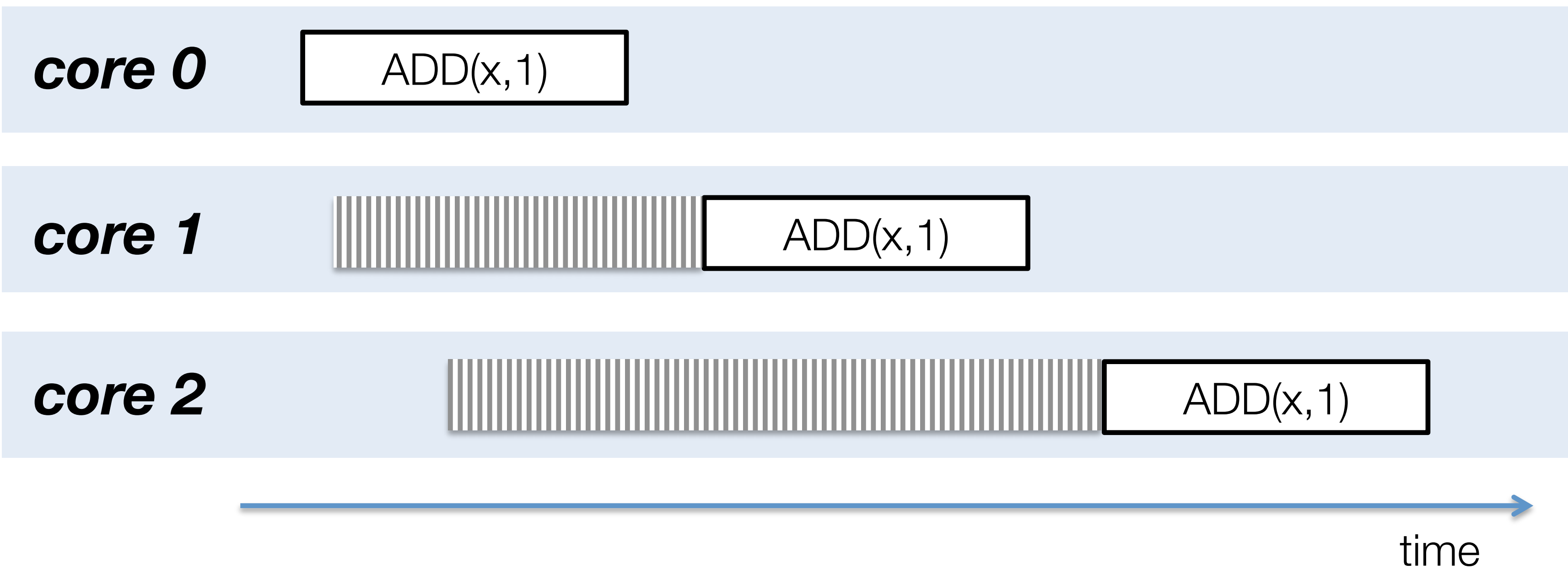
Transactions are incorrectly seeing intermediate values

Concurrency control enforces serial execution



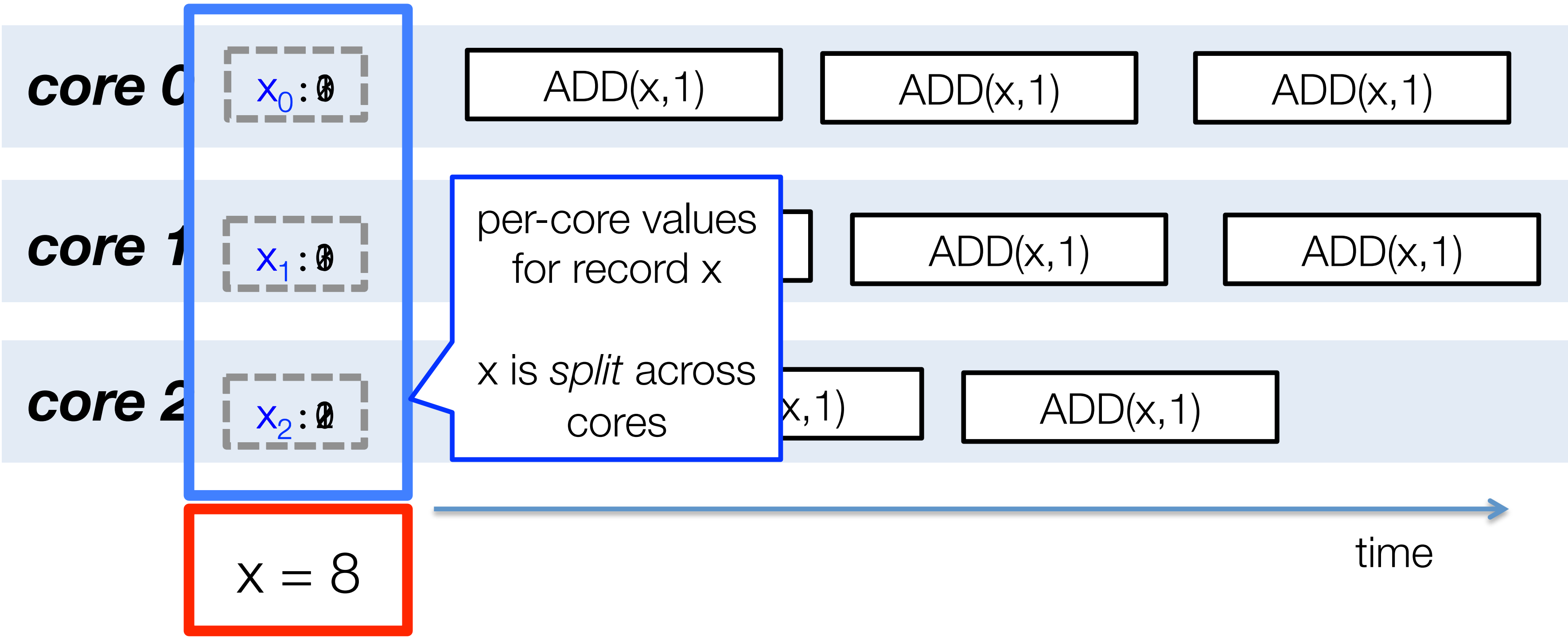
Transactions on the same records execute one at a time

Concurrency control enforces serial execution



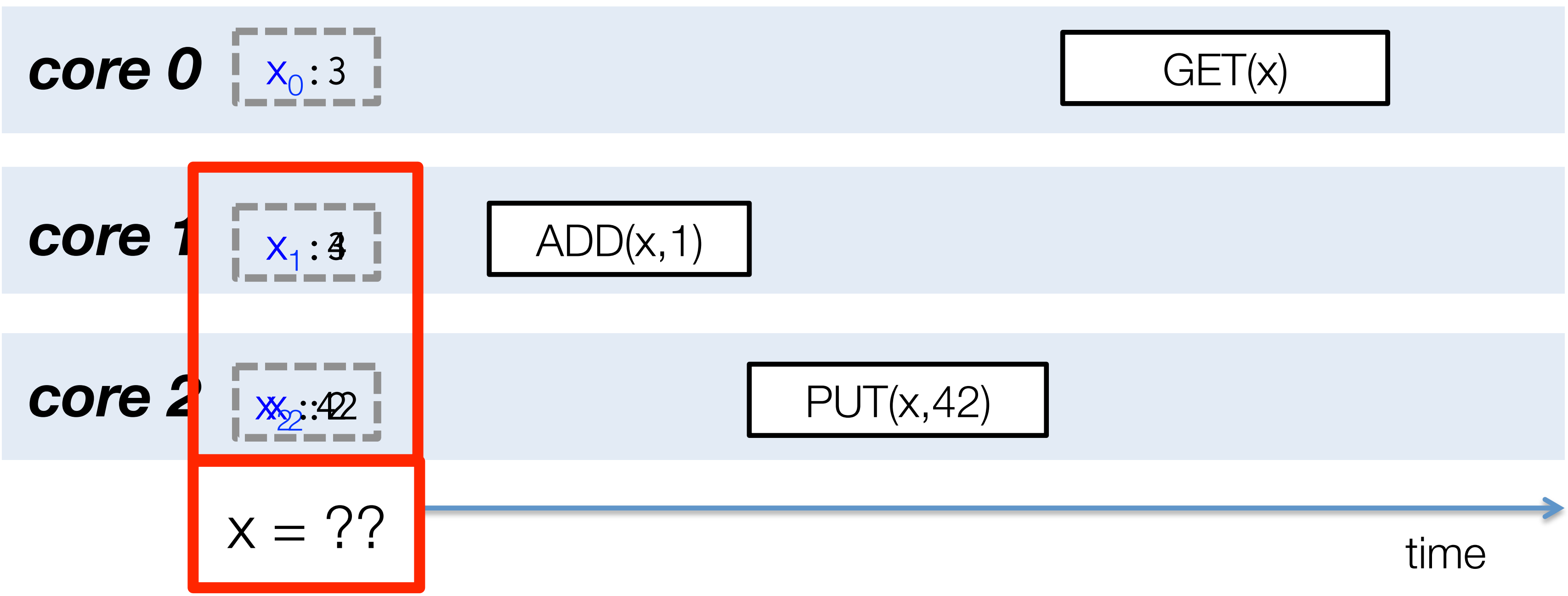
Serial execution results in a lack of scalability

Idea #1: Split representation for parallel execution



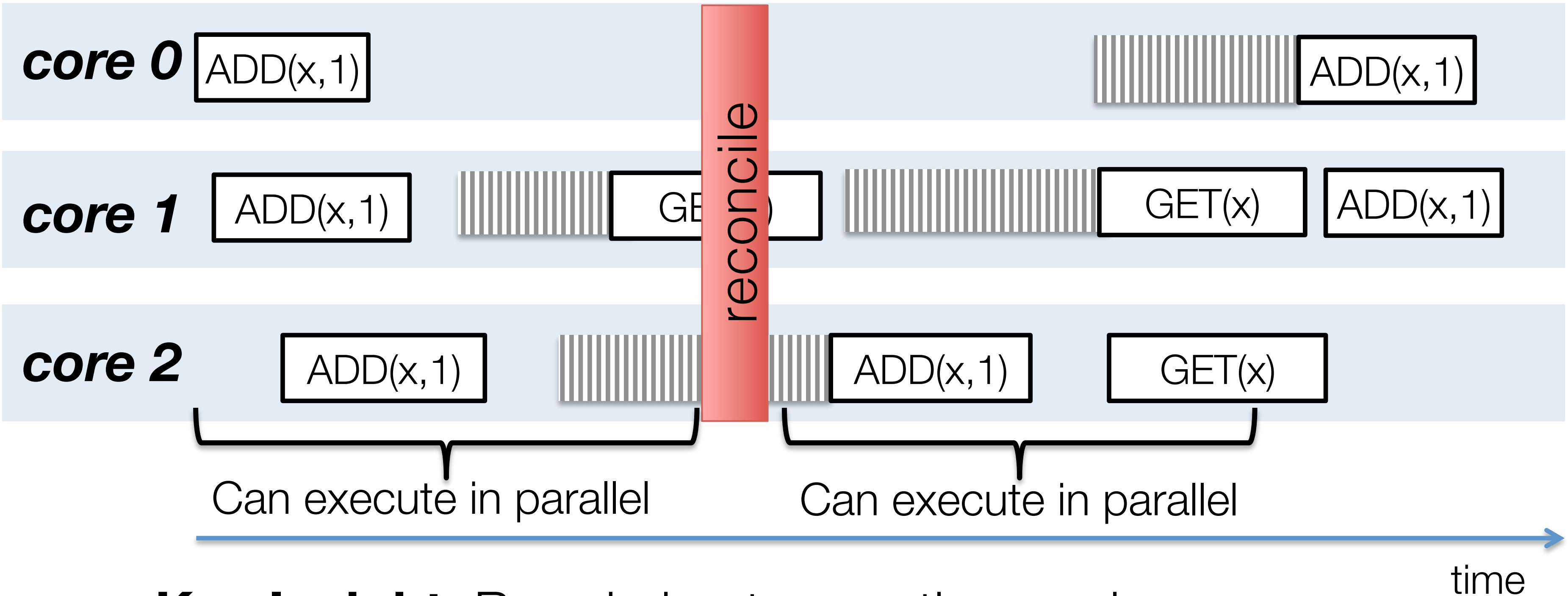
- Transactions on the same record can proceed in parallel on *per-core values*
- *Reconcile* per-core values for a correct value

Other types of operations do not work with split data



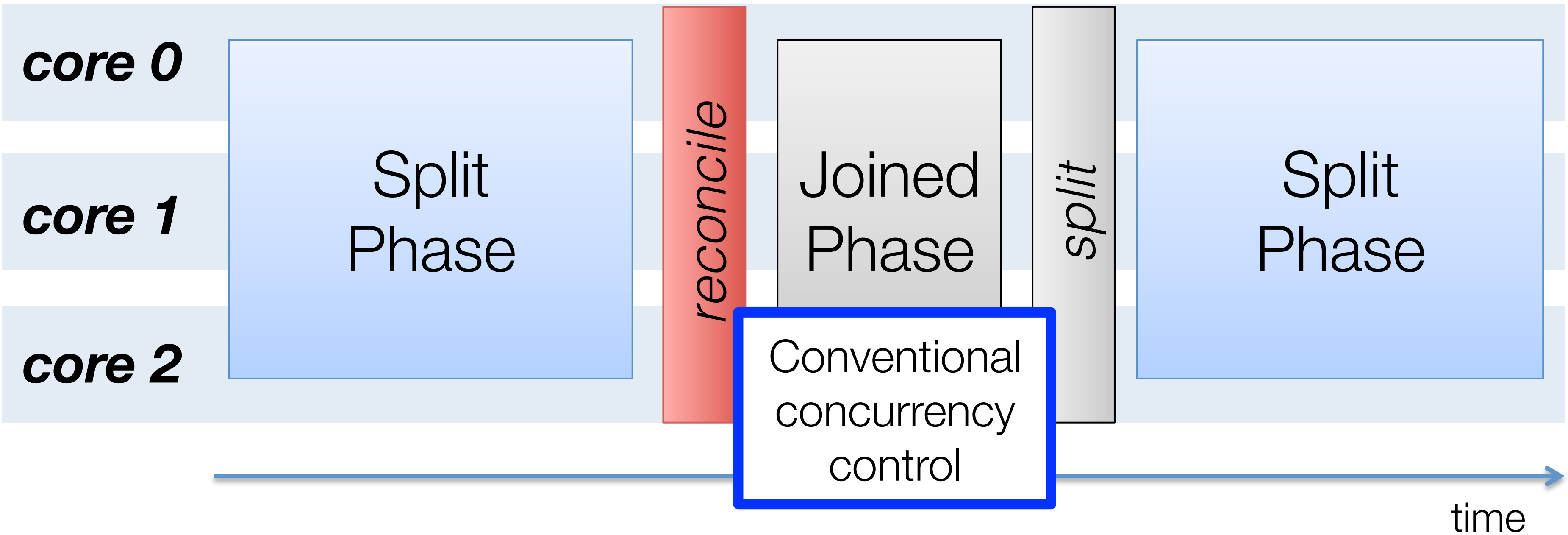
- Executing with split data does not work for all types of operations
- In a workload with many reads, better to *not* use per-core values

Idea #2: Reorder transactions



- **Key Insight**: Reordering transactions reduces
  - Cost of reconciling
  - Cost of conflict
- Serializable execution

Idea #3: Phase reconciliation



- Database automatically detects contention to split a record between cores
- Database cycles through *phases*: split and joined
- Doppel: An in-memory key/value database

Challenges

Combining split data with general database workloads:

1. How to handle transactions with multiple keys and different operations?
2. Which operations can use split data correctly?
3. How to dynamically adjust to changing workloads?

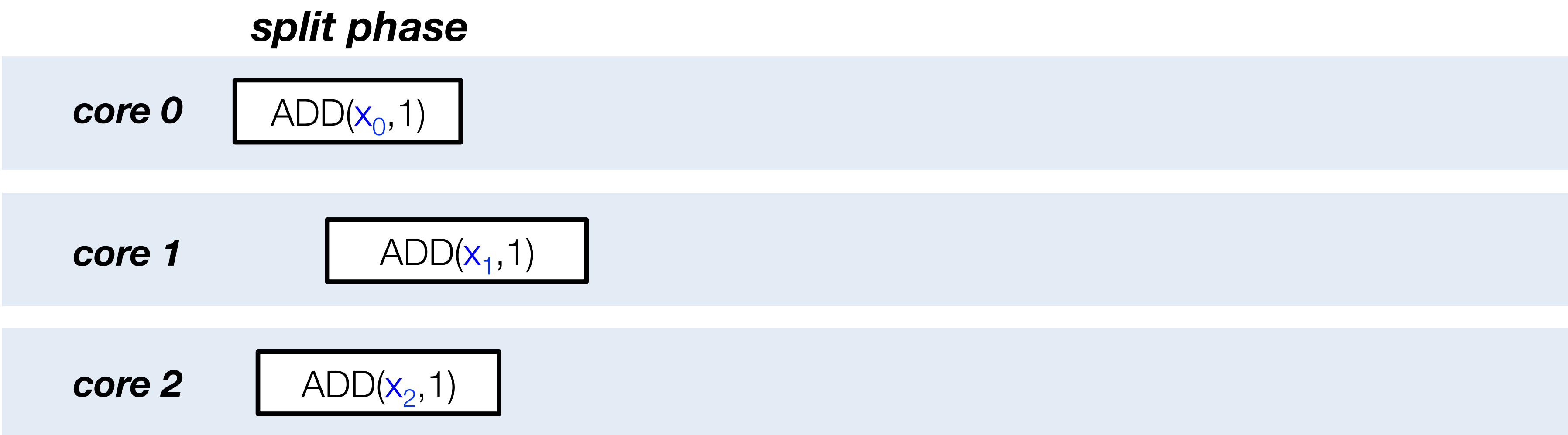
Contributions

- Synchronized phases to support any transaction and reduce reconciliation overhead
- Identifying a class of splittable operations
- Detecting contention to dynamically split data

Outline

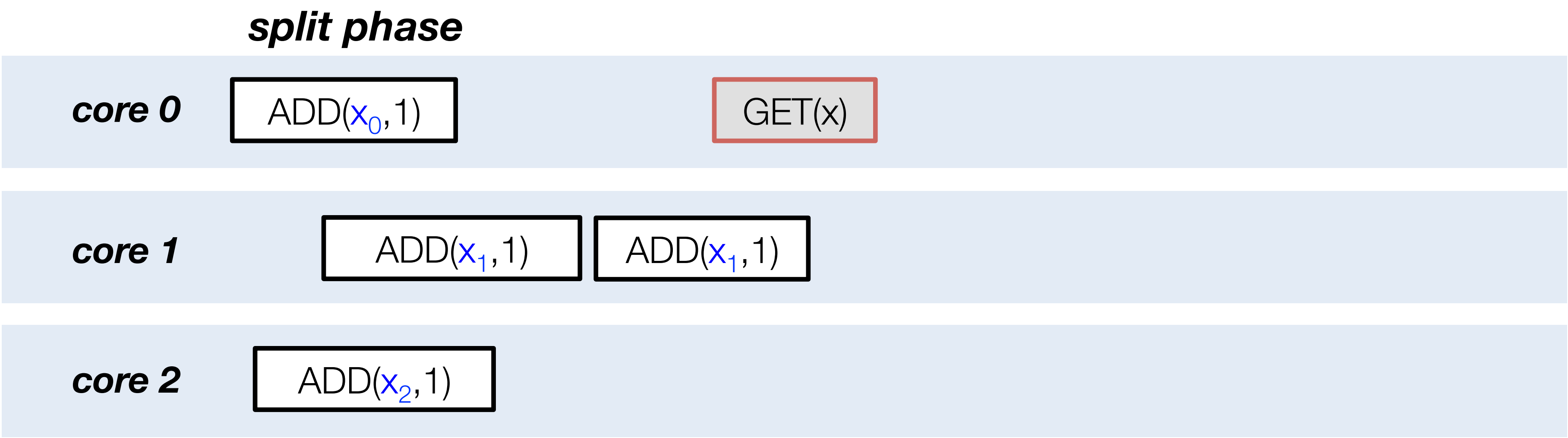
- Challenge 1: Phases
- Challenge 2: Operations
- Challenge 3: Detecting contention
- Performance evaluation
- Related work and discussion

Split phase

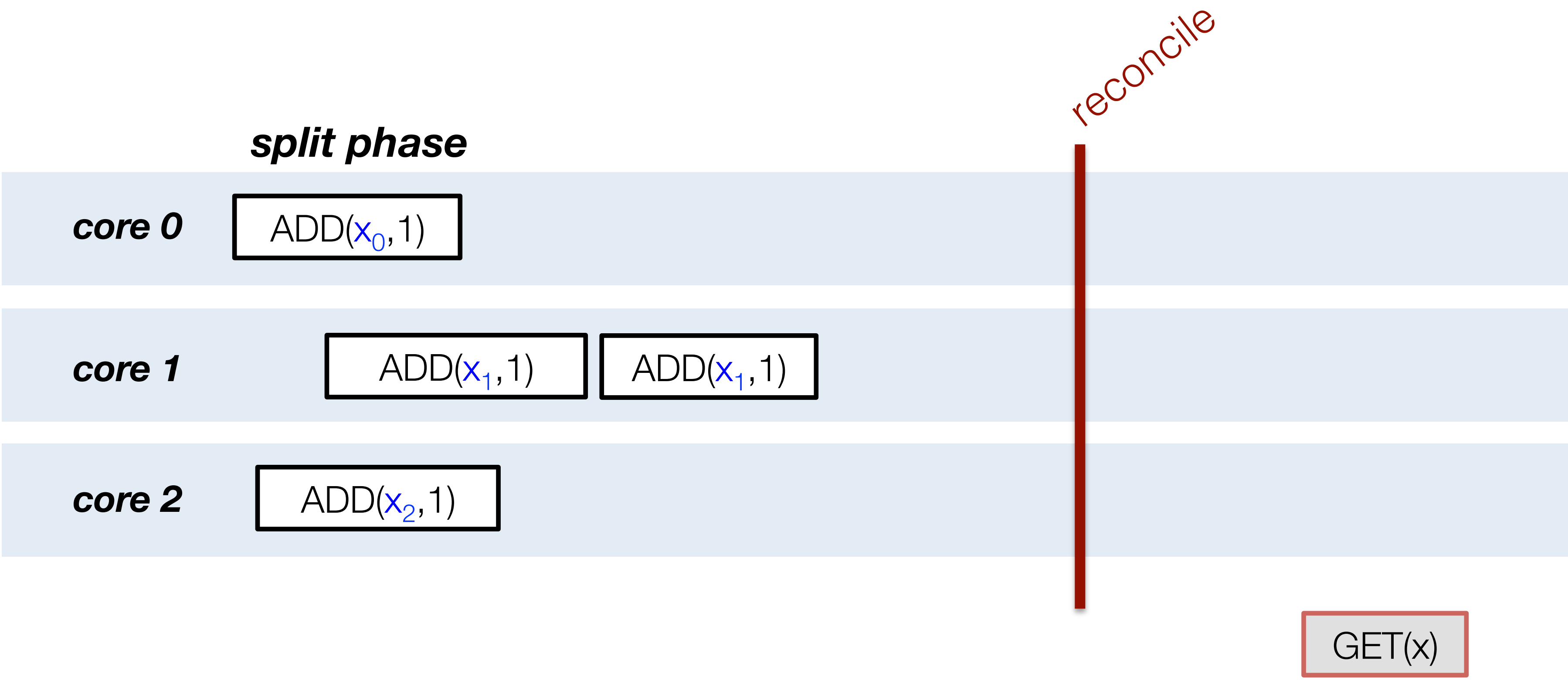


- The *split phase* executes operations on contended records on per-core slices (x_0, x_1, x_2)

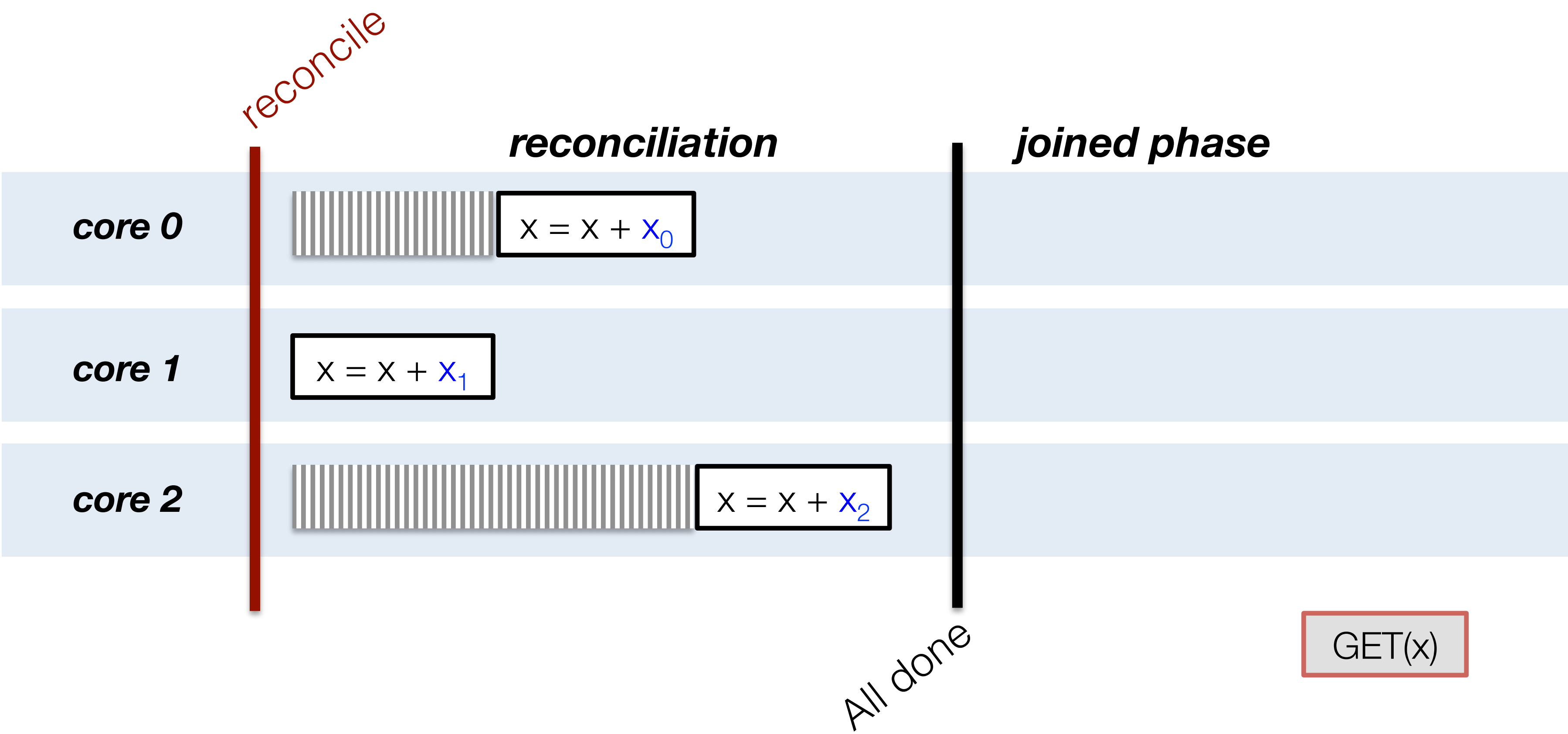
Reordering by stashing transactions



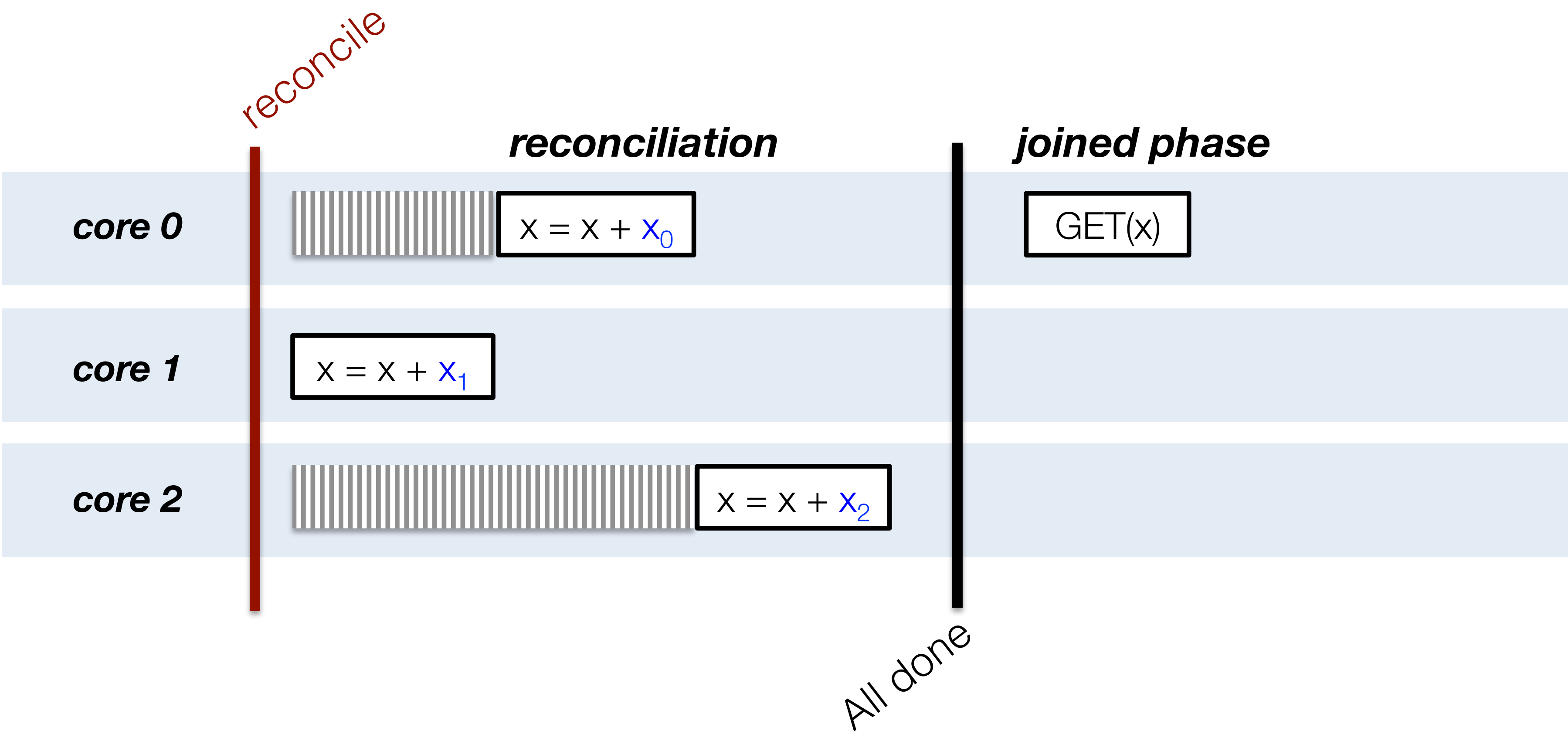
- Split records have **selected operations** for a given split phase
- Cannot correctly process a read of x in the current state
- *Stash* transaction to execute after reconciliation



- All cores hear they should reconcile their per-core state
- Stop processing per-core writes

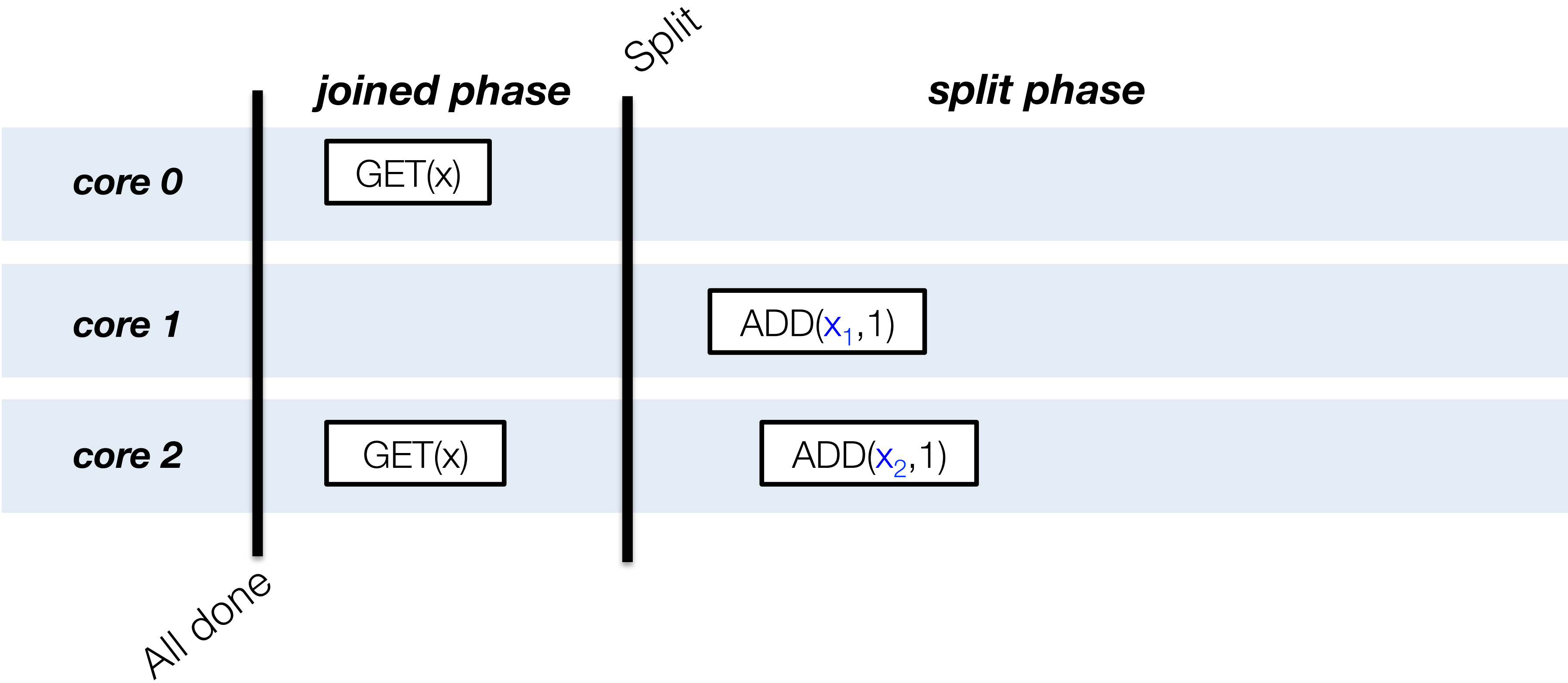


- Reconcile state to global store
- Wait until all cores have finished reconciliation
- Resume stashed read transactions in joined phase



- Reconcile state to global store
- Wait until all cores have finished reconciliation
- Resume stashed read transactions in joined phase

Transitioning between phases



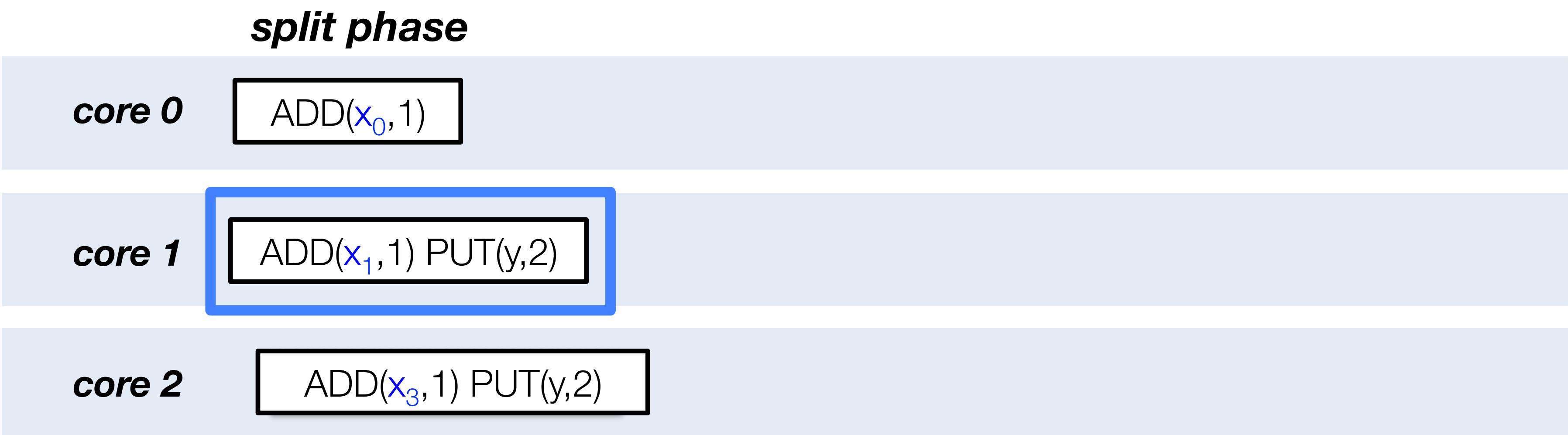
- Process stashed transactions in joined phase using conventional concurrency control
- Joined phase is short; quickly move on to next split phase

Challenge #1

How to handle transactions with multiple keys and different operations?

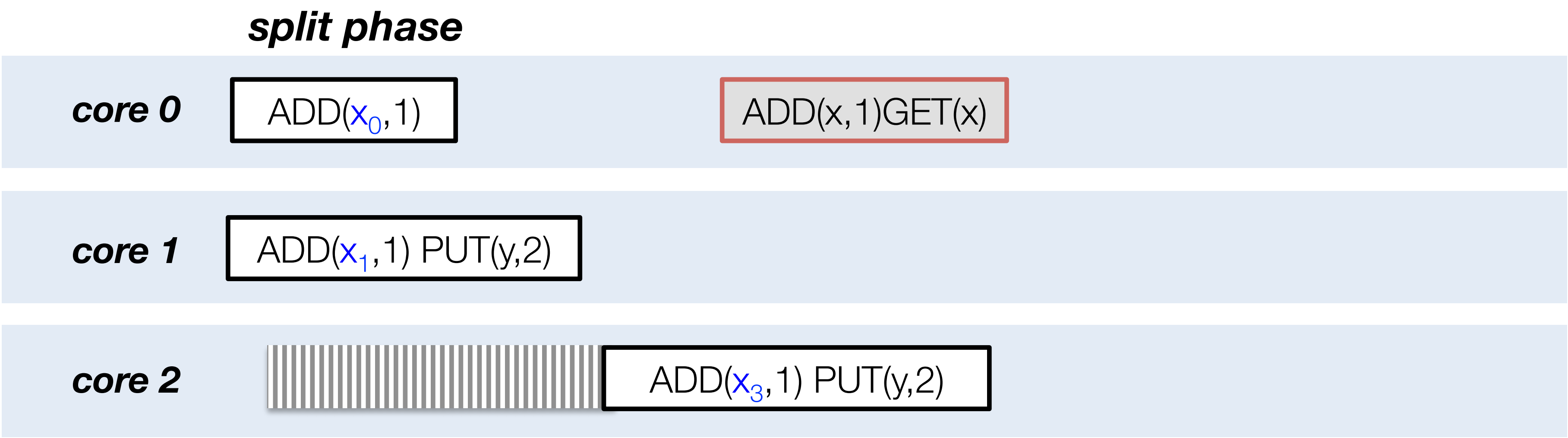
- Split and non-split data
- Different operations on a split record
- Multiple split records

Transactions on split and non-split data



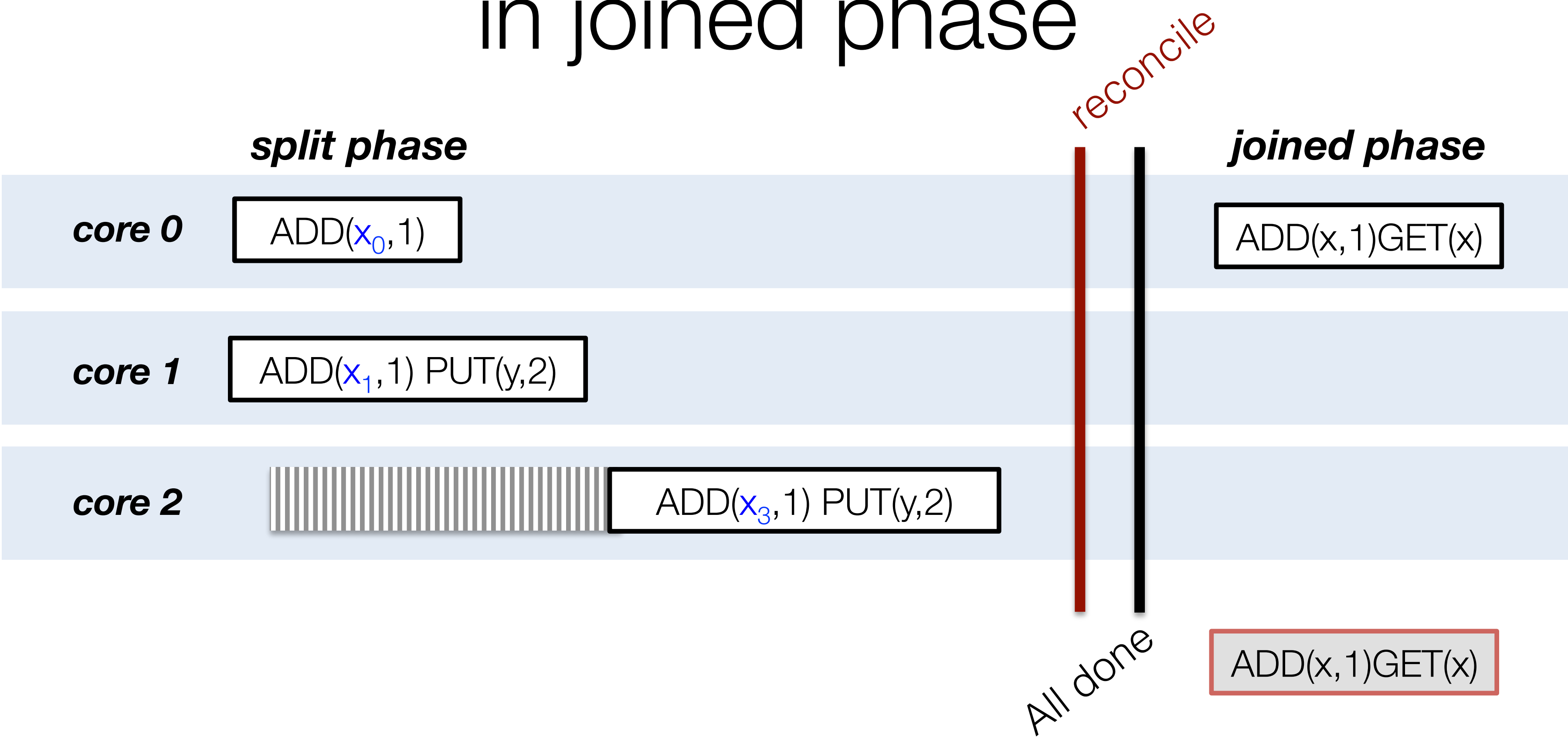
- Transactions can operate on split and non-split records
- Rest of the records (y) use concurrency control
- Ensures serializability for the non-split parts of the transaction

Transactions with different operations on a split record



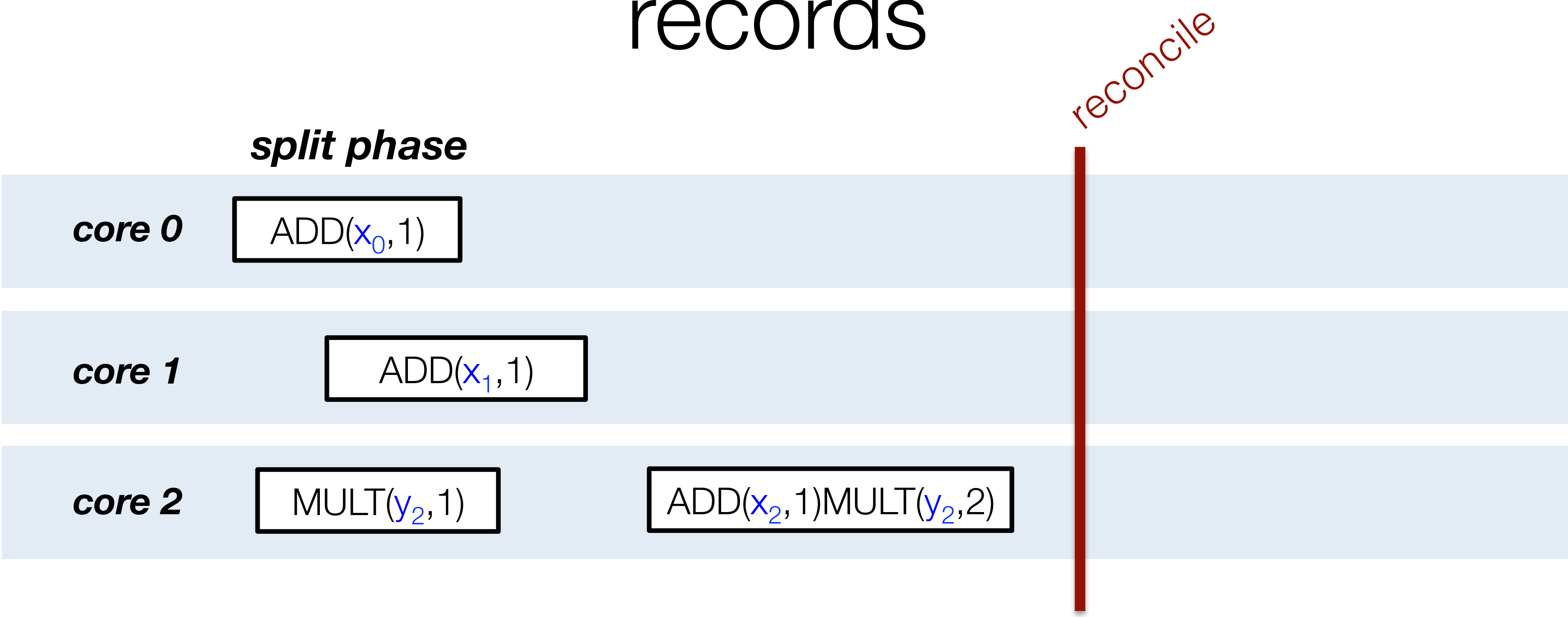
- A transaction which executes *different* operations on a split record is also stashed, even if one is a selected operation

All records use concurrency control in joined phase



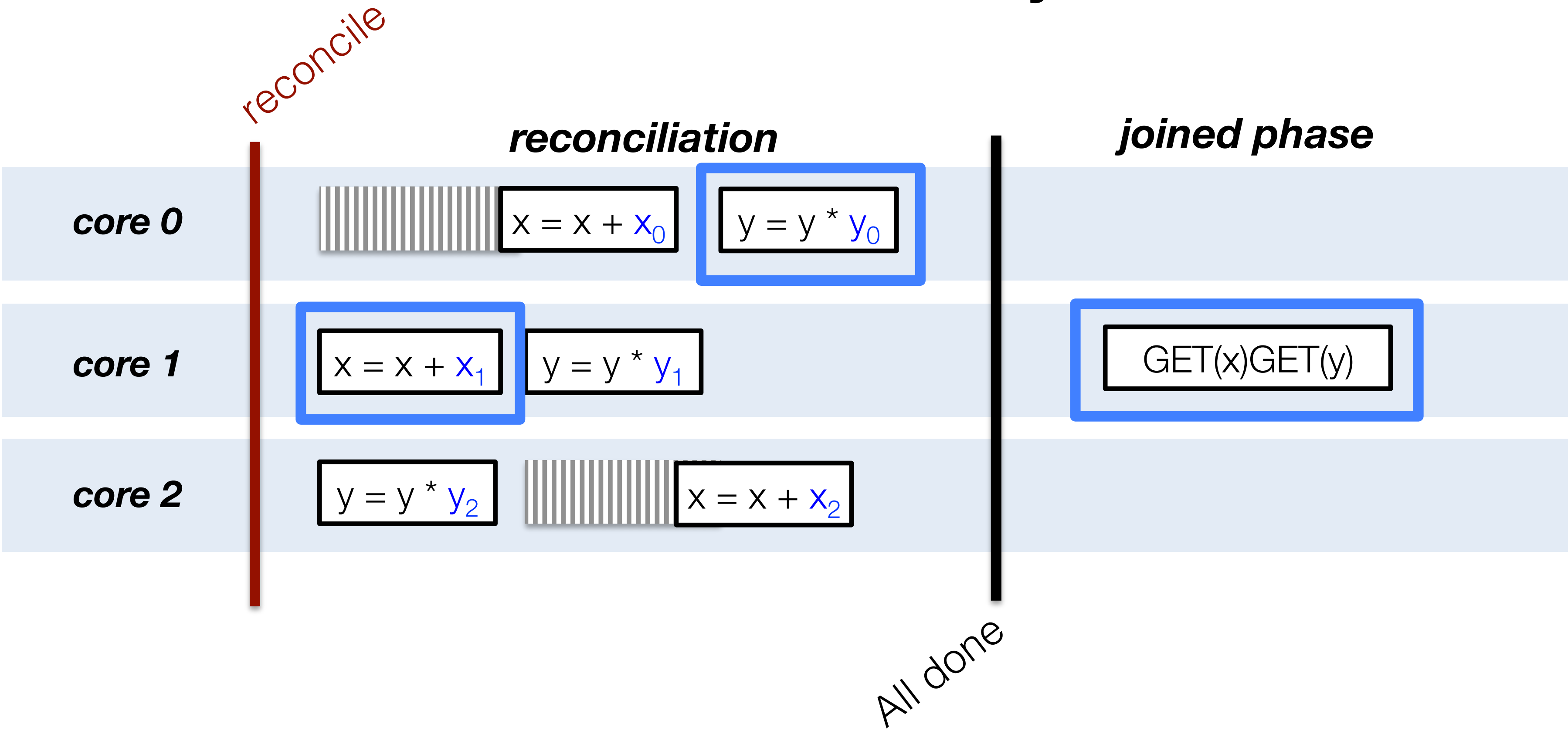
- In joined phase, no split data, no split operations
- ADD also uses concurrency control

Transactions with multiple split records



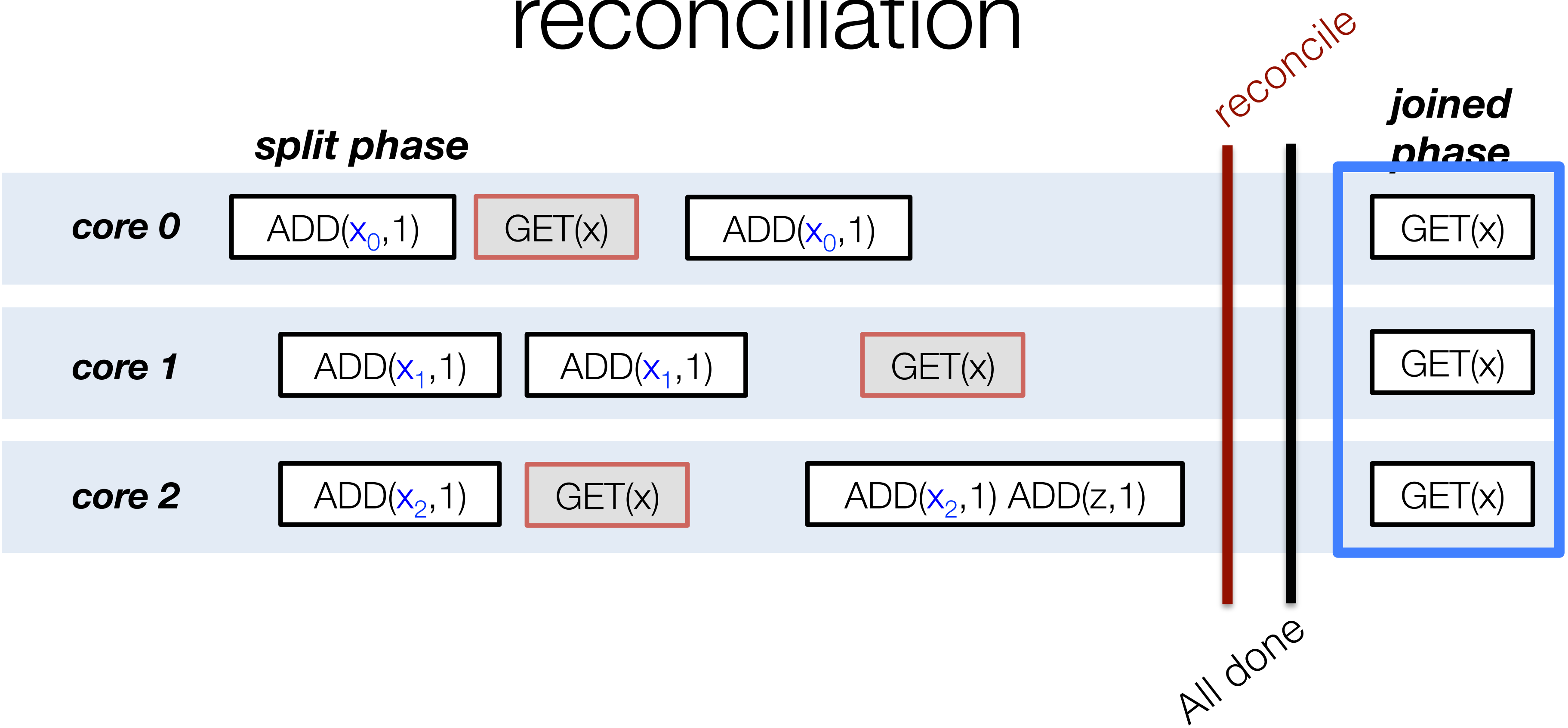
- x and y are split and operations on them use per-core slices (x_0, x_1, x_2) and (y_0, y_1, y_2)
- Split records all use the same synchronized phases

Reconciliation must be synchronized



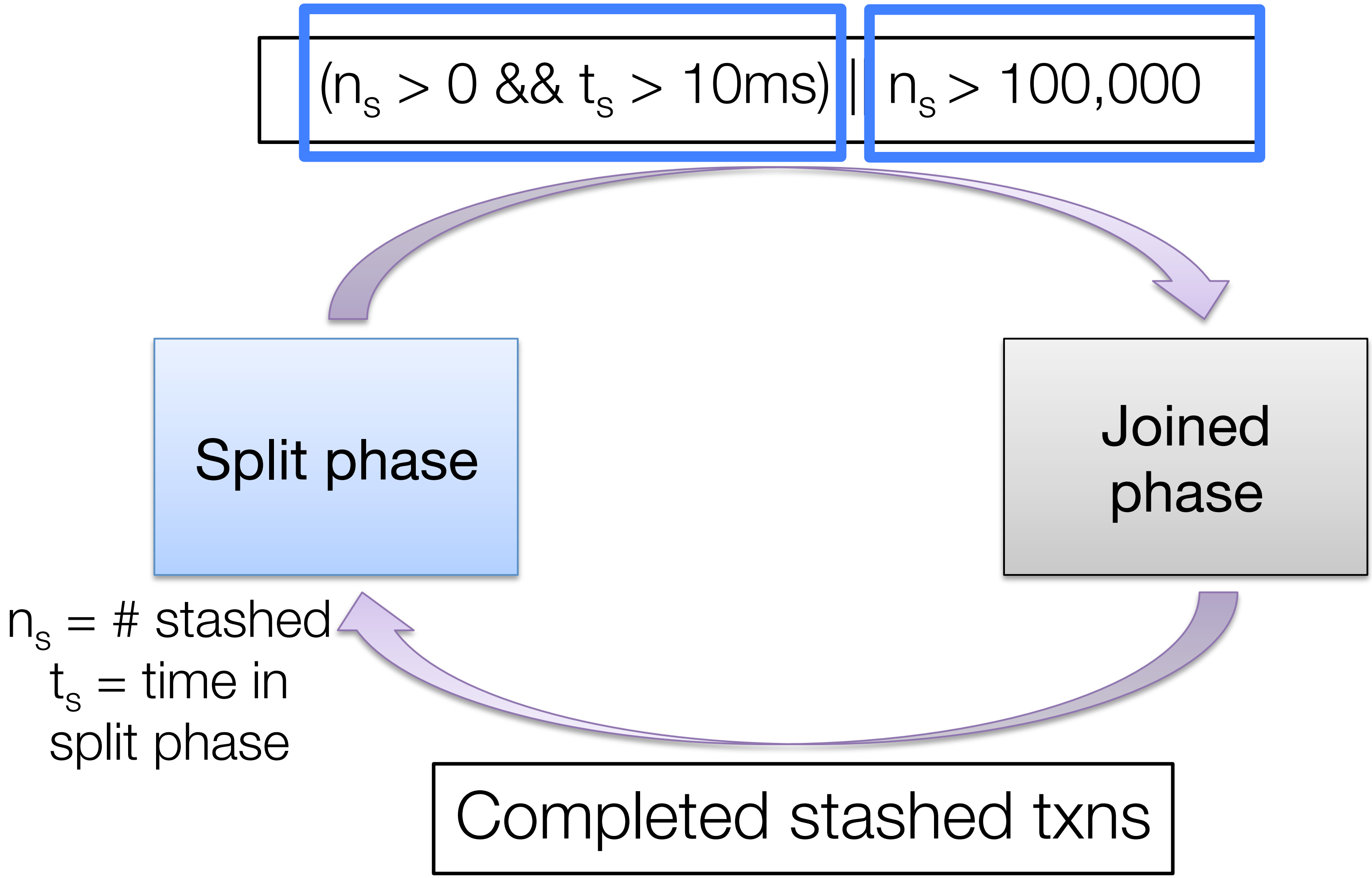
- Cores reconcile all of their split records: ADD for x and MULT for y
- Parallelize reconciliation
- Guaranteed to read values atomically in next joined phase

Delay to reduce overhead of reconciliation



- Wait to accumulate stashed transactions, many in joined phase
- Reads would have conflicted; now they do not

When does Doppel switch phases?



Outline

- Challenge 1: Phases
- Challenge 2: Operations
- Challenge 3: Detecting contention
- Performance evaluation
- Related work and discussion

Challenge #2

Define a class of operations that is correct and performs well with split data.

Operations in Doppel

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

Operations on numeric values which modify the existing value

```
void ADD(k,n)
void MAX(k,n)
void MULT(k,n)
```

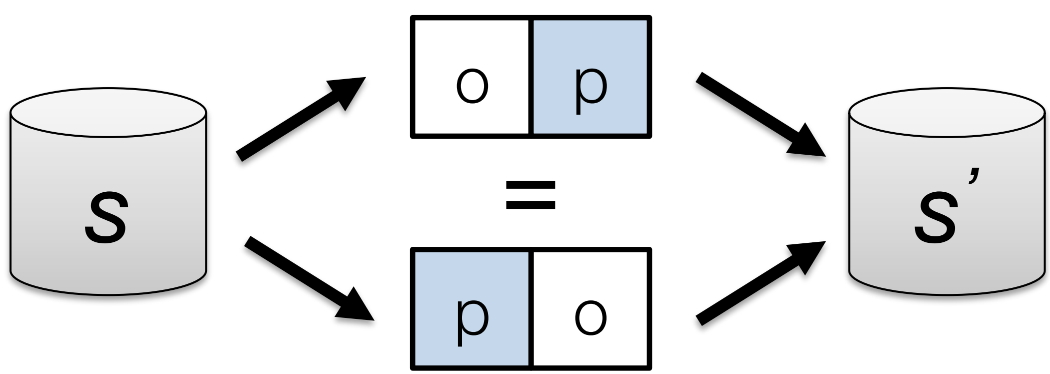
Why can ADD(x,1) execute correctly on split data in parallel?

- Does not return a value
- Commutative

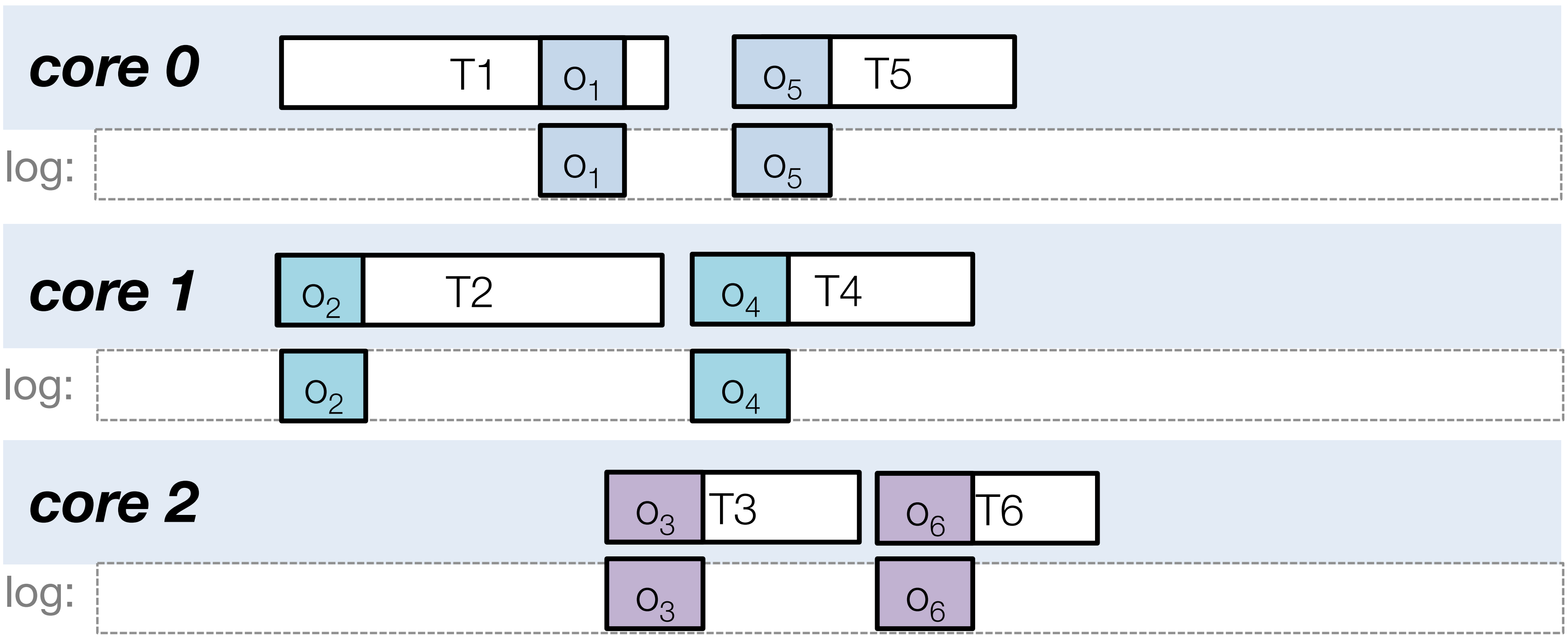
```
ADD(k,n) {
   v[k] = v[k] + n
}
```

Commutativity

Two operations *commute* if executed on the database *s* in either order, they produce the same state *s'* and the same return values.

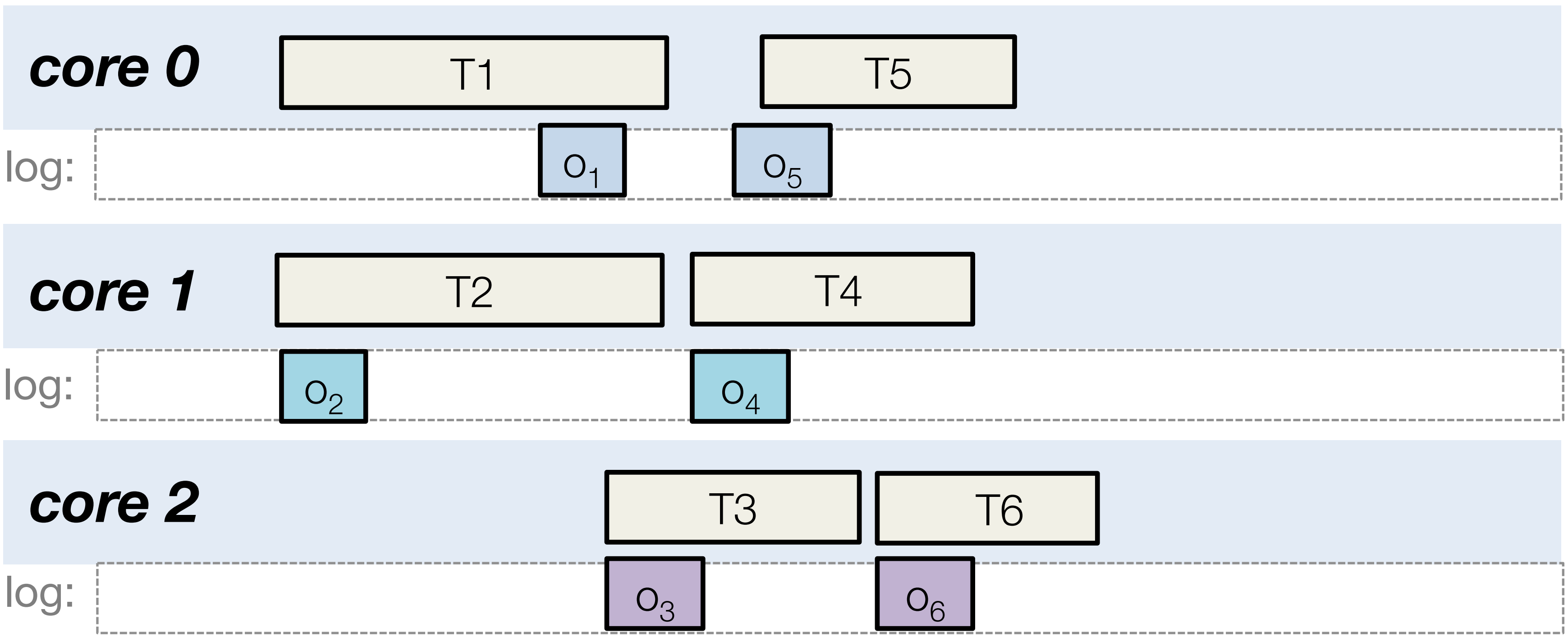


Hypothetical design: commutativity is sufficient



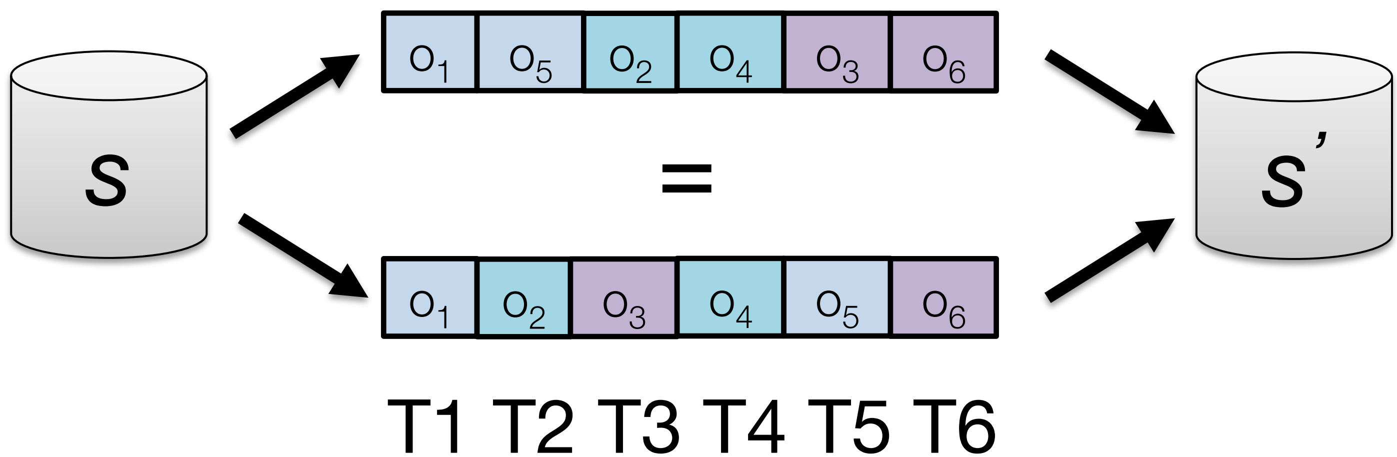
- Not-split operations in transactions execute
- Split operations are logged
- They have no return values and are on **different data**, so cannot affect transaction execution

Hypothetical design: apply logged operations later



- Logged operations are applied to database state *in a different order* than their containing transactions

Correct because split operations can be applied in any order



After applying the split operations
in *any order*,
same database state

Is commutativity enough?

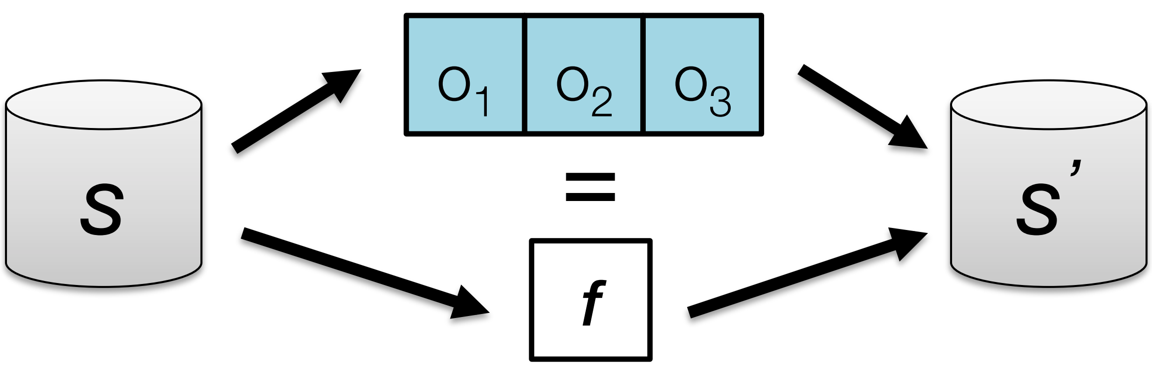
For correctness, yes.

For performance, no.

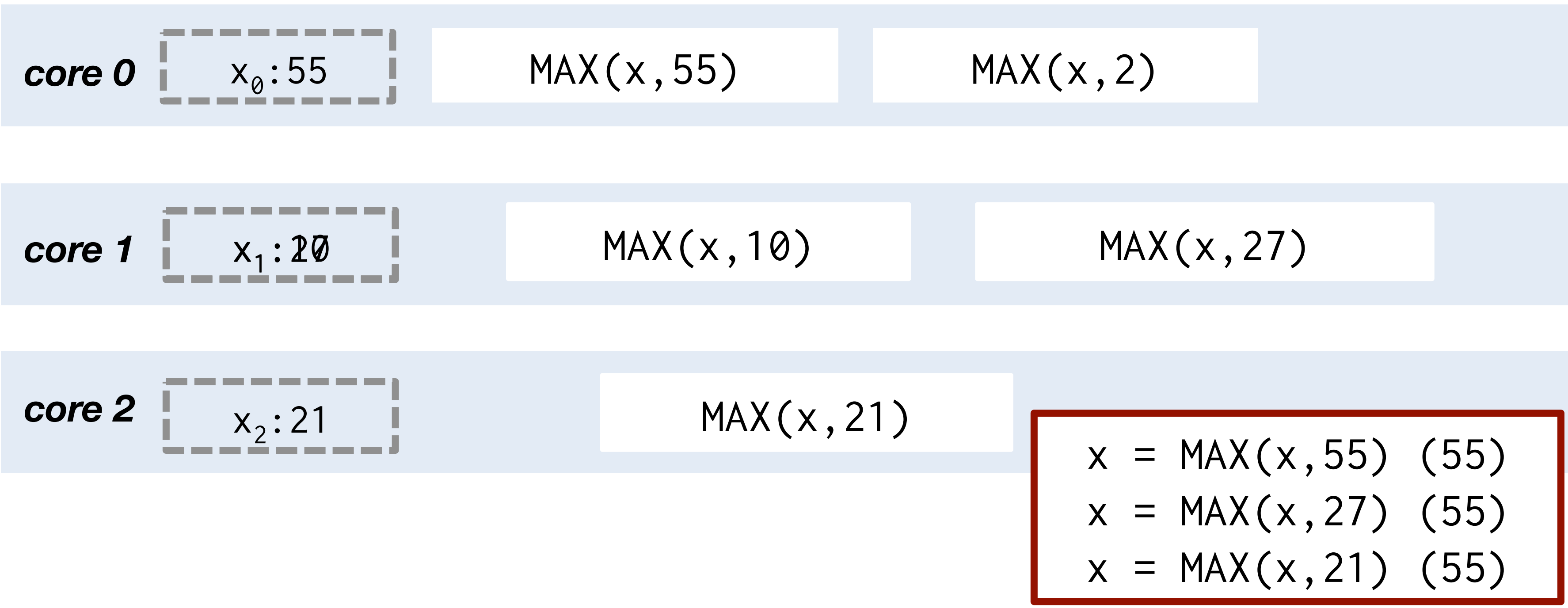
Which operations can be *summarized*?

Summarized operations

An set of operations can be *summarized* if for all sequences of operations in the set, there is a function *f* that produces the same result and runs in time order a single operation.



MAX can be summarized



- Each core keeps *one* piece of state
- 55 is an abbreviation of a function to apply later
- O(*#cores*) time to reconcile *x*

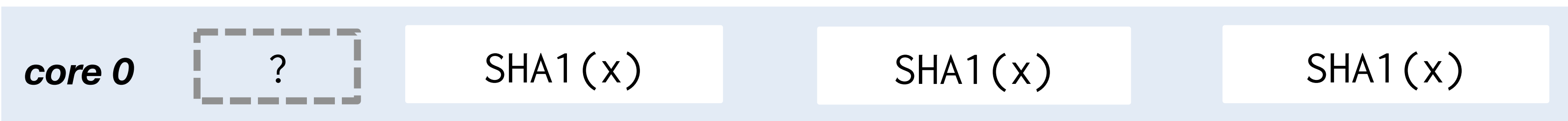
SHA1 cannot be summarized

```
SHA1(k) {
   v[k] = sha1(v[k])
}
```

`SHA1(x)`
commutes!

```
SHA1(SHA1(x)) = SHA1(SHA1(x))
```

SHA1 is commutative but we do not know how to summarize it



- Need to produce a function that produces the same value as SHA1 run *n* times on x, but has running time O(SHA1)
- No such function

Operation summary

Properties of operations that Doppel can split:

- Always commute
- Can be summarized
- Single key
- Have no return value

Runtime restriction:

- Only one type of operation per record per split phase

Example commutative and summarizable operations

Operations on numeric values which modify the existing value

```
void ADD(k,n)
void MAX(k,n)
void MULT(k,n)
```

With timestamps, last writer wins

Ordered PUT and insert to an ordered list

```
void OPUT(k,v,o)
void TOPK_INSERT(k,v,o)
```

Short indexes, top friends or follower lists

Outline

- Challenge 1: Phases
- Challenge 2: Operations
- Challenge 3: Detecting contention
- Performance evaluation
- Related work and discussion

Challenge #3

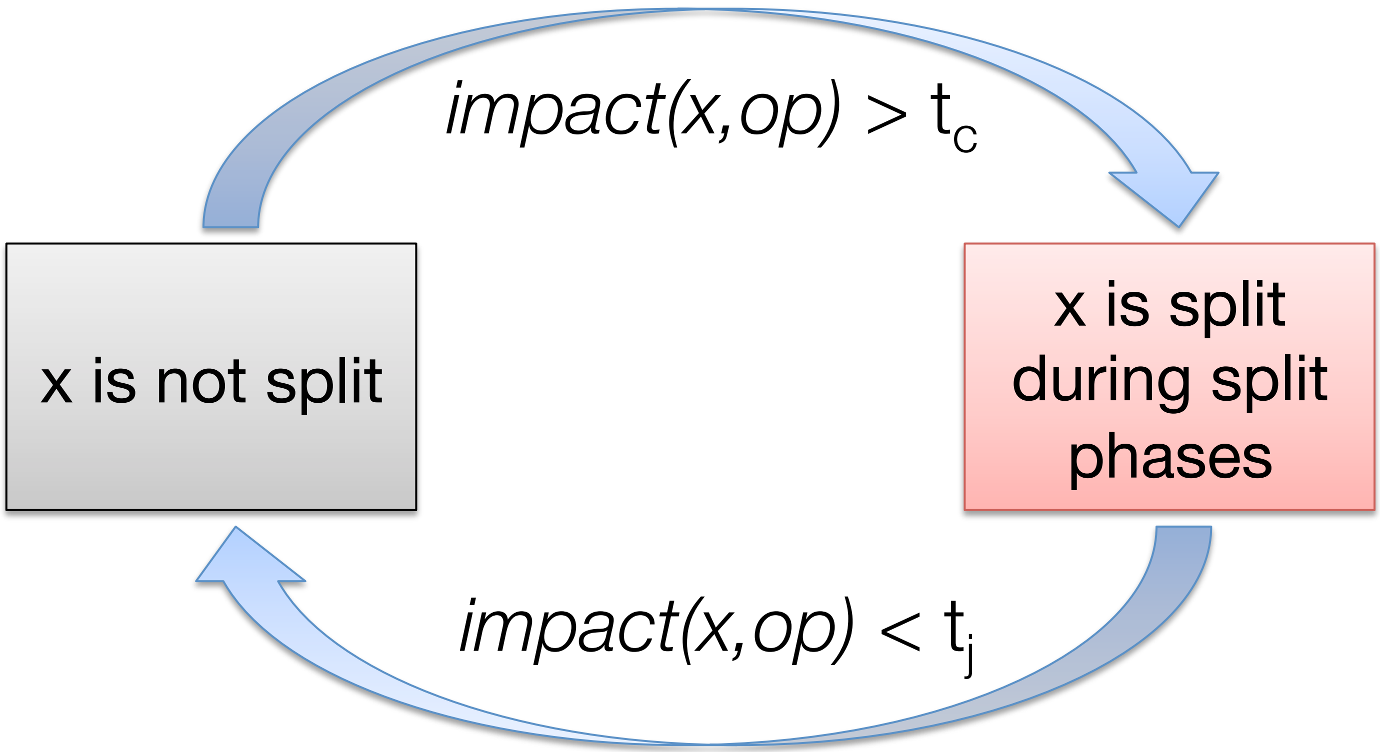
Dynamically adjust to changes in the workload:

- Which records are contended?
- What operations are happening on different records?

How to determine what to split?

- Developer annotates records
  - Difficult to determine
  - Popular data changes over time
- Automatically split data based on observed contention
  - Count records and operations which cause conflict
  - Split records *actually* causing serialization
  - Sample for low cost

Which records does Doppel split?



$$\boxed{impact(x,op) = \frac{conflicts_{op}(x)}{\sum other(x)}}$$

Implementation

- Doppel implemented as a multithreaded Go server; one worker thread per core
- Coordinator thread manages phase changes
- Transactions are procedures written in Go
- All data fits in memory; key/value interface with optionally typed values
- Doppel uses optimistic concurrency control

Outline

- Challenge 1: Phases
- Challenge 2: Operations
- Challenge 3: Detecting contention
- Performance evaluation
- Related work and discussion

Performance evaluation

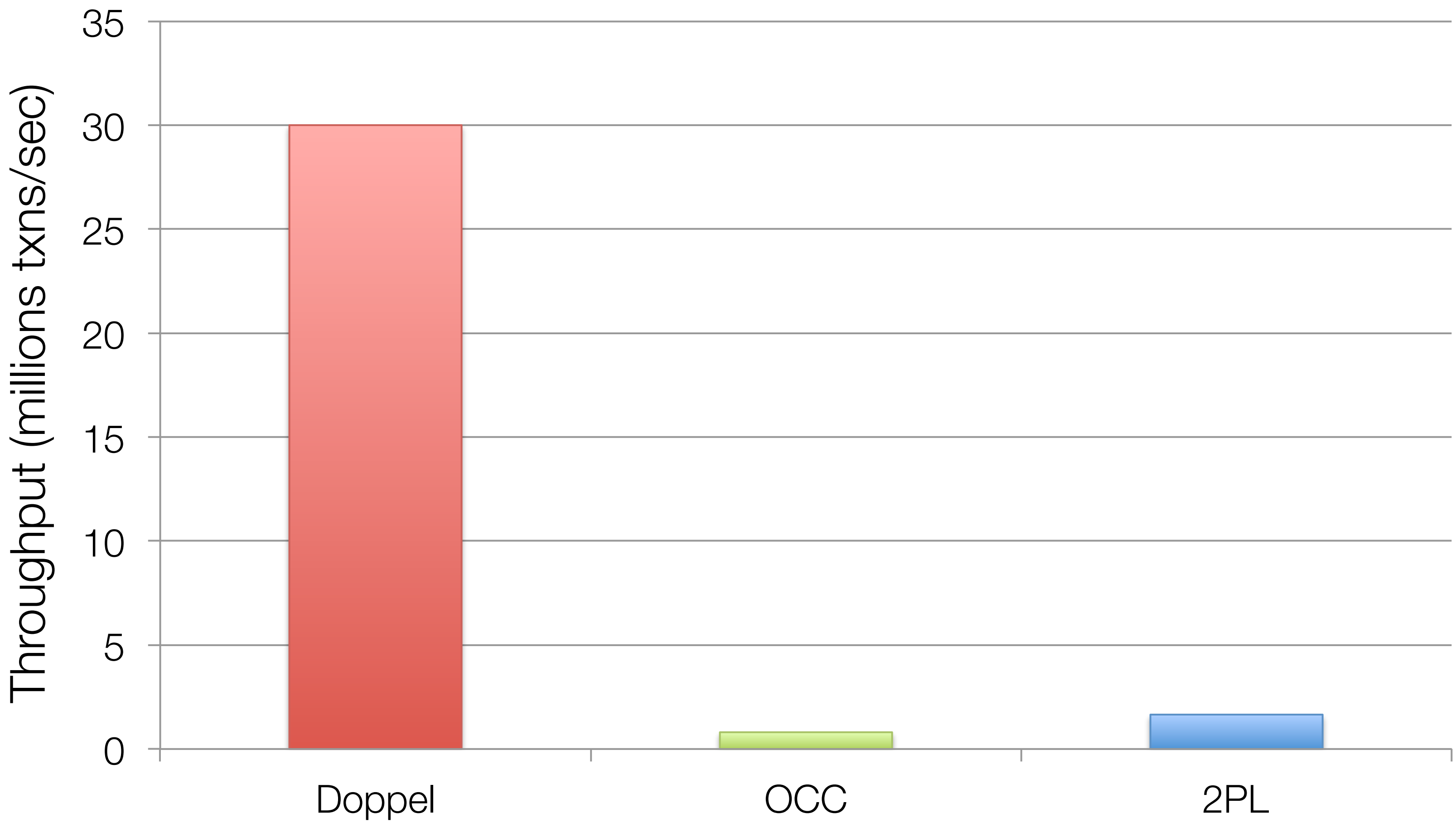
- Extreme contention
- A range of contention
- Changing workloads
- Workloads with a mix of reads and writes
- A complex application

Experimental setup

- All experiments run on an 80 core Intel server running 64 bit Linux 3.12 with 256GB of RAM
- All data fits in memory; don't measure RPC or disk
- All graphs measure throughput in transactions/sec

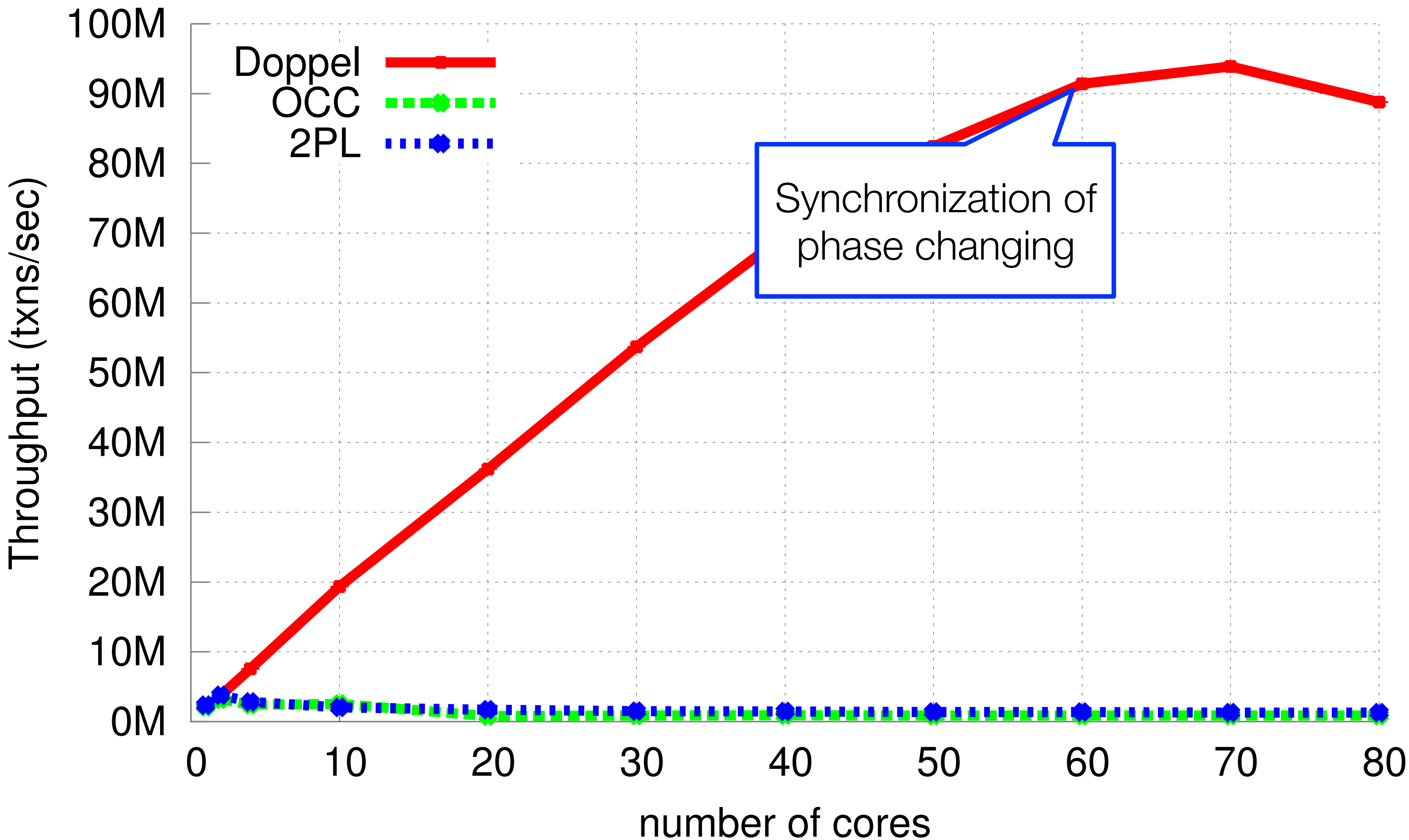
How much does Doppel improve throughput on contentious write-only workloads?

Doppel executes conflicting workloads in parallel



20 cores, 1M 16 byte keys, transaction: ADD(x,1) all on same key

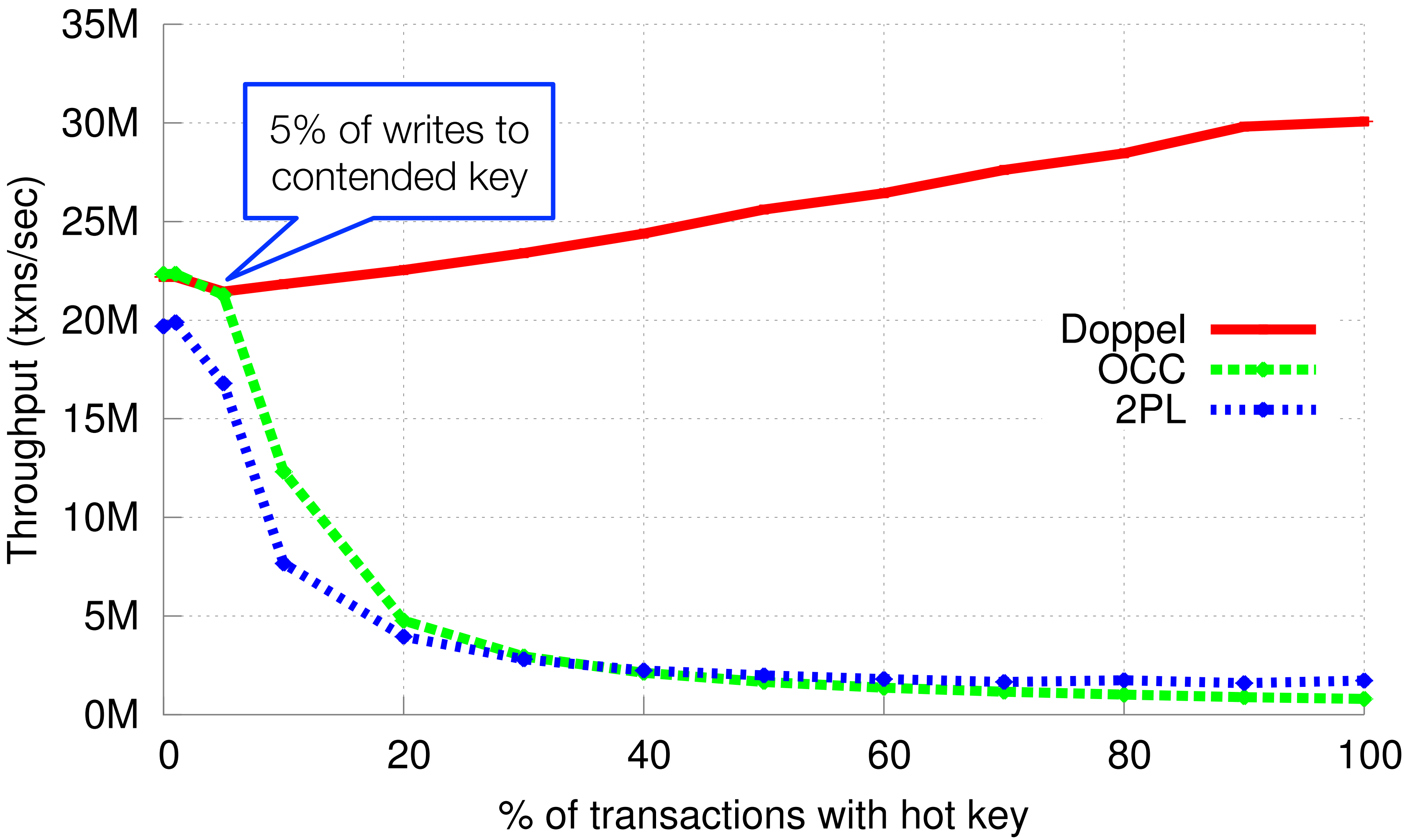
Contentious workloads scale well



1M 16 byte keys, transaction: ADD(x,1) all writing same key

How much contention is required for Doppel's techniques to help?

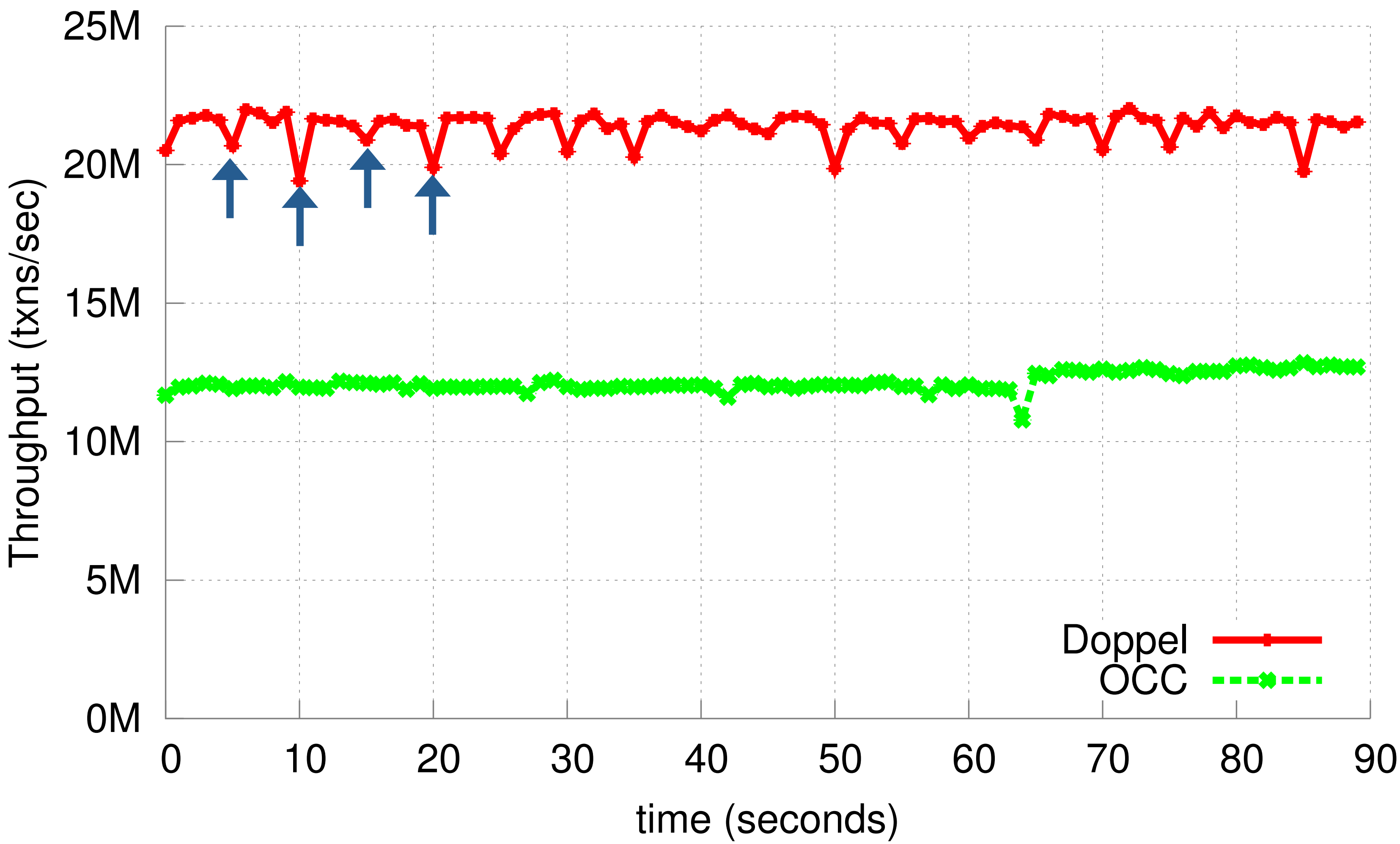
Doppel outperforms 2PL and OCC even with low contention



20 cores, 1M 16 byte keys, transaction: ADD(x,1) on different keys

Can Doppel detect and respond to changing workloads over time?

Doppel adapts to changing popular data



20 cores, 1M 16 byte keys, transaction: ADD(x,1) 10% on same key

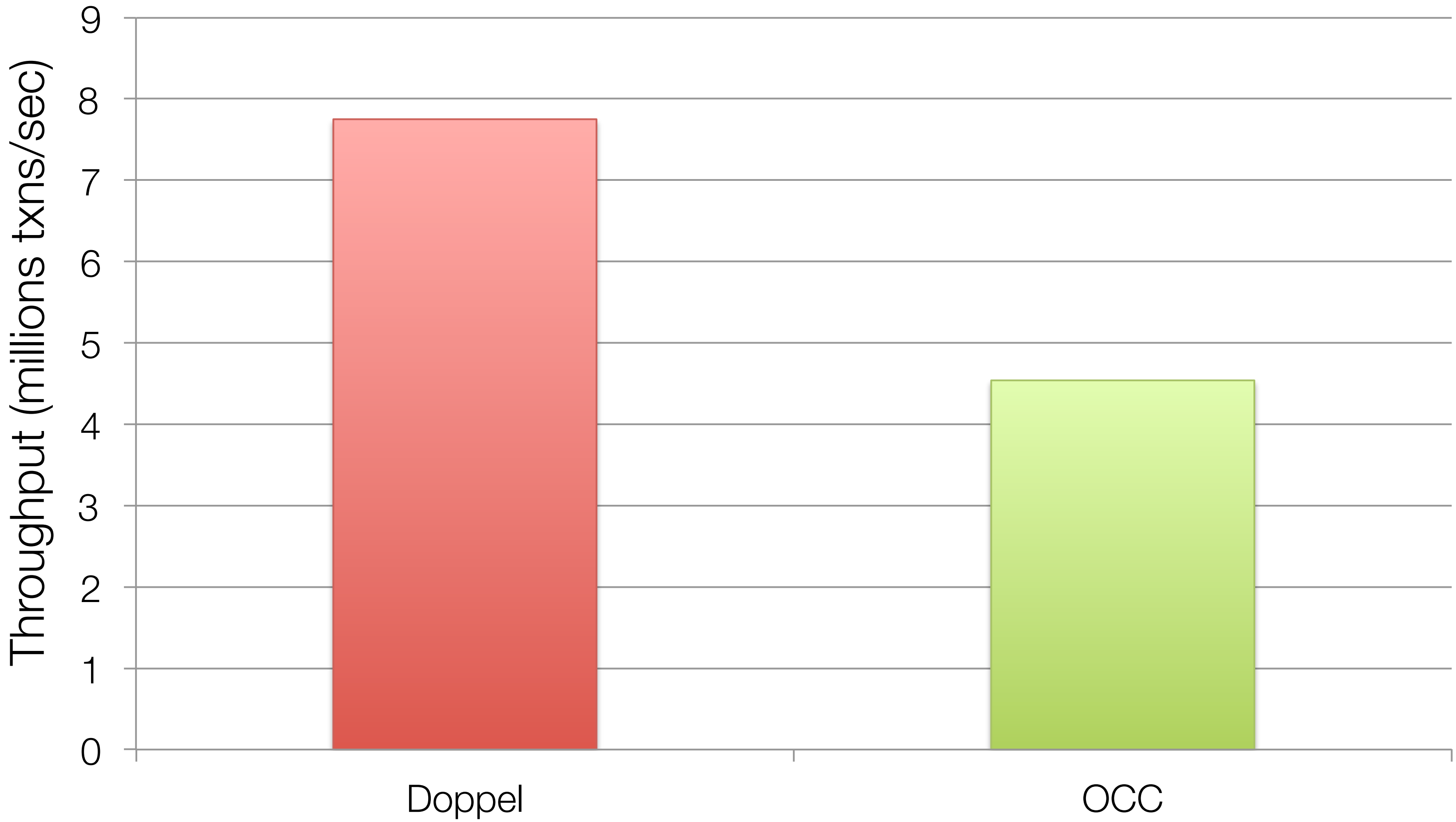
How much benefit can Doppel get with many stashed transactions?

Read/Write benchmark

- Users liking pages on a social network
- 2 tables: users, pages
- Two transactions:
  - ADD 1 to a page's like count, PUT user like of page
  - GET a page's like count, GET user's last like
- 1M users, 1M pages, Zipfian distribution of page popularity

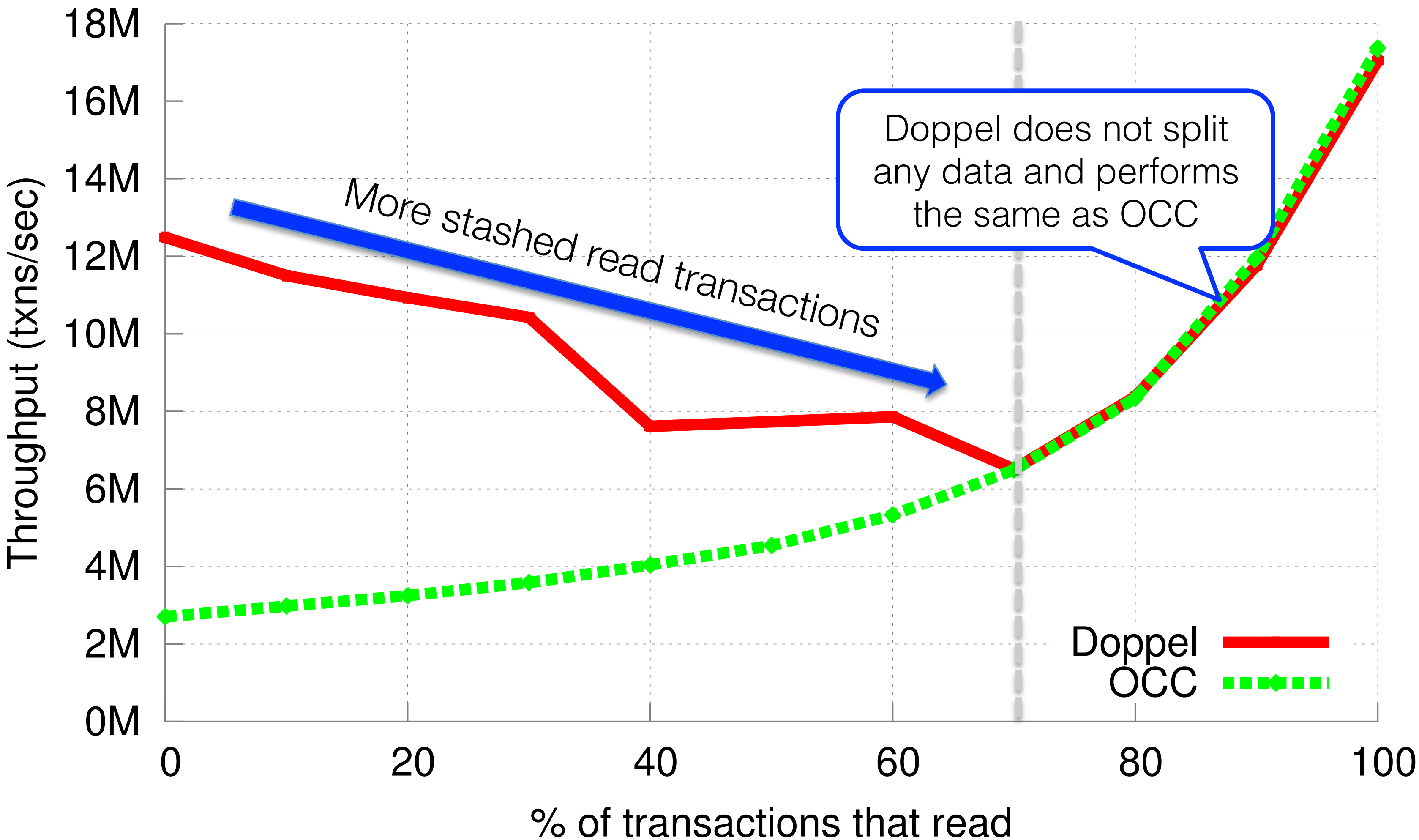
Doppel splits the popular page counts
But those counts are also read most often

Benefits even when there are reads and writes to the same popular keys



20 cores, transactions: 50% read, 50% write

Doppel outperforms OCC for a wide range of read/write mixes



20 cores, transactions: RW benchmark

Does Doppel improve throughput for a realistic application: RUBiS?

RUBiS

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

- Two workloads:
  - Roughly uniform distribution of bids
  - Skewed distribution of bids; a few auctions are very popular

RUBiS StoreBid transaction

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

The contended data is only operated on by splittable operations.

Inserting new bids is not likely to conflict

Doppel improves throughput for the RUBiS benchmark

80 cores, 1M users 33K auctions, RUBiS bidding mix. 50% bids on top auction

Outline

- Challenge 1: Phases
- Challenge 2: Operations
- Challenge 3: Detecting contention
- Performance evaluation
- Related work and discussion

Related work

- Shared memory DBs
  - Silo, Hekaton, ShoreMT
- Partitioned DBs
  - DORA, PLP, Hstore
- Choosing partitions
  - Schism, Estore, Horticulture
- Transactional memory
  - Scheduling [Kim 2010, Attiya 2012]

Doppel runs conflicting transactions in parallel

Related work

- Commutativity
  - Abstract Datatypes [Weihl 1988]
  - CRDTs [Shapiro 2011]
  - RedBlue consistency [Li 2
  - Walter [Sovran 2011]
- Scalable operating systems
  - Clustered objects in Tornado [Parsons 1995]
  - OpLog [Boyd-Wickizier 2013]
  - Scalable commutativity rule [Clements 2013]

Doppel combines these ideas in a transactional database

Future Work

- Generalizing to distributed transactions
- More data representations
- Larger class of operations which commute
- Durability and recovery

Conclusion

Multi-core phase reconciliation:

- Achieves parallel performance when transactions conflict by combining split data and concurrency control
- Performs well on uniform workloads while improving performance significantly on skewed workloads.

CSAIL

Thanks

Robert, Eddie, and Barbara

Co-authors and colleagues

PDOS and former PMG

Academic and industry communities

Family and friends

Brian Allen, Neelam Narula, Arun Narula, Megan Narula, Adrienne Winans, Austin Clements, Yandong Mao, Adam Marcus, Alex Pesterev, Alex Yip, Max Krohn, Cody Cutler, Frank Wang, Xi Wang, Ramesh Chandra, Emily Stark, Priya Gupta, James Cowling, Dan Ports, Irene Zhang, Jean Yang, Grace Woo, Szymon Jakubczak, Omar Khan, Sharon Perl, Brad Chen, Ben Swanson, Ted Benson, Eugene Wu, Evan Jones, Vijay Pandurangan, Keith Winstein, Jonathan Perry, Stephen Tu, Vijay Boyapati, Ines Sombra, Tom Santero, Chris Meiklejohn, John Wards, Gergely Hodicska, Zeeshan Lakhani, Bryan Kate, Michael Kester, Aaron Elmore, Grant Schoenebeck, Matei Zaharia, Sam Madden, Mike Stonebraker, Frans Kaashoek, Nickolai Zeldovich

Phase length and read latency