Weak Isolation: Theory and Its Impact

Lecture at Seoul National University, March 14, 2012

Professor Alan Fekete (University of Sydney)

THE UNIVERSITY OF SYDNEY

University of Sydney Database Research Group

- Academics: Sanjay Chawla, Alan Fekete, Uwe Röhm
  - Postdocs, Visitors, Students
- Database management: internals and applications
- Focus on consistency and performance
- System-oriented approaches

Databases

- Data that is shared among several applications, can be stored and managed centrally in a complex software system with dedicated hardware and staff
  - Organizational benefits (accountability, economies of scale, etc)
- Database: a collection of shared data
- Database management system (DBMS): the complex software that controls access to the database

Database Research

- Study issues related to managing substantial amounts of data
- Storage, query processing, data mining, schema management, data integration
  - Hot topics in 2011: graph data, cloud data management, privacy/security, data analytics, impact of new memory technologies
- Combine approaches from infrastructure systems, programming languages, data structures, theory, AI, etc
- Large, unified and well-established research international community
  - 2012 is 38th VLDB conference, 32nd SIGMOD, 28th ICDE
- Great commercialization record

Transaction Processing

- A powerful model from business data processing
- Each real-world change is performed through a program which executes multiple database operations
  - Some ops modify the database contents, based on data already there and on program parameters
- Eg customer purchases goods
- ACID properties:
  - Atomic (all or nothing, despite failures)
  - Consistent (maintains data integrity)
  - Isolated (no problems from concurrency)
  - Durable (changes persist despite crashes)

Serializability (academic def^n)

- Used to define the correctness of an interleaved execution of several transactions (formalize “isolated”)
  - Same values read, same final values as in serial (batch) execution of the same transactions
- For every integrity condition C: if each txn acting alone preserves C, then a serializable execution will preserve C
  - That is: programmer makes sure txn does the right thing on its own, then platform makes sure no problems from concurrency
- Can be assessed by absence of cycles in a graph showing conflicts/dependencies
  - When different txns access the same items, and at least one txn modifies

But…. Vendor advice

- **Oracle DB**: "Database inconsistencies can result unless such application-level consistency checks are coded with this in mind, even when using serializable transactions"
- "PostgreSQL's Serializable mode does not guarantee serializable execution..." [before release 9.1]
- Why is this? Traditional lock-based correct concurrency control performs poorly in important situations, so platforms use different mechanisms that might perform more reliably, but sometimes do the wrong thing

Our research agenda

- Theory: what properties of application code allow certainty that data corruption will not arise from concurrency, for various system mechanisms
  - Provide assurance that all executions will be serializable
    - Running on platforms that don't provide this guarantee in general
- Impact 1: Guide DBAs (or application designers)
  - DBA can check if applications will run correctly, together
  - DBA can change things to get to this situation
  - Understand the performance implications of different ways to have this assurance
- Impact 2: Suggest new system mechanisms
  - Ensure correctness and also perform reasonably

Isolation Levels

- SQL standard offers several isolation levels
  - Each transaction can have level set separately
  - Problematic definitions, but in best practice done with variations in lock holding
- Serializable
  - (ought to be default, but not so in practice)
  - Traditionally done with Commit-duration locks on data and indices

    We'll call this "Two Phase Locking (2PL)"
- Repeatable Read
  - Commit-duration locks on data
- Read Committed
  - short duration read locks, commit-duration write locks
- Read Uncommitted
  - no read locks, commit-duration write locks

NB: note different usage of term; Here we talk about a single txn's concurrency control mechanism

Snapshot Isolation (SI)

- A multiversion concurrency control mechanism was described in SIGMOD '95 by H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, P. O'Neil
  - Does not guarantee serializable execution!
- Supplied by Oracle DB, and PostgreSQL (before rel 9.1), for "Isolation Level Serializable"
- Available in Microsoft SQL Server 2005 as "Isolation Level Snapshot"

Snapshot Isolation (SI)

- Read of an item may not give current value
- Instead, use old versions (kept with timestamps) to find value that had been most recently committed at the time the txn started
  - Exception: if the txn has modified the item, use the value it wrote itself
- The transaction sees a “snapshot” of the database, at an earlier time
  - Intuition: this should be consistent, if the database was consistent before

First committer wins (FCW)

- T will not be allowed to commit a modification to an item if any other transaction has committed a changed value for that item since T's start (snapshot)
- Similar to optimistic CC, but only write-sets are checked
- T must hold write locks on modified items at time of commit, to install them.
  - In practice, commit-duration write locks may be set when writes execute. These simplify detection of conflicting modifications when T tries to write the item, instead of waiting till T tries to commit.

Benefits of SI

- Reading is *never* blocked, and reads don't block writes
- Avoids common anomalies
  - No dirty read
  - No lost update
  - No inconsistent read
  - Set-based selects are repeatable (no phantoms)
- Matches common understanding of isolation: concurrent transactions are not aware of one another's changes

Is every execution serializable?

- For any set of txns, if they all run with Two Phase Locking, then every interleaved execution is serializable
- For some sets of txns, if they all run with SI, then every execution is serializable
  - Eg the txns making up TPC-C
- For some sets of txns, if they all run with SI, there can be non-serializable executions
  - Undeclared integrity constraints can be violated

Example

- Table Duties(Staff, Date, Status)
- Undeclared constraint: for every Date, there is at least 1 Staff with Status='Y'
- Transaction TakeBreak(S, D) running at SI

```
SELECT COUNT(*) INTO :tmp FROM Duties
 WHERE Date=:D AND Status='Y';
 IF tmp < 2 ROLLBACK;
 UPDATE Duties
     SET Status = 'N'
     WHERE Staff =:S AND Date =:D;
 COMMIT;
```

Example (continued)

- Possible execution, starting when two staff (S101, S103) are on duty for 2004-06-01
- *Concurrently* perform

TA: TakeBreak(S101, 2004-06-01)

TB: TakeBreak(S103, 2004-06-01)

  - Each succeeds, as each sees snapshot with 2 on duty
  - No problem committing, as they update different rows!
- End with no staff on duty for that date!
- RA(r1) RA(r3) RB(r1) RB(r3) WA(r1) CA WB(r3) CB
  - Non-serializable execution

S101	2004-06-01	‘Y’
S102	2004-06-01	‘N’
S103	2004-06-01	‘Y’
etc	etc	etc

Write Skew

- SI breaks serializability when txns modify different items in each other's read sets
  - Neither txn sees the other, but in a serial execution one would come later and so see the other's impact
- This is fairly rare in practice
- Eg the TPC-C benchmark always runs correctly under SI
  - whenever its txns conflict (eg read/write same data), there is also a ww-conflict: a shared item they both modify (like a total quantity) so SI will abort one of them

Interaction effects

- You can't think about one program, and say "this program can use SI"
- The problems have to do with the set of application programs, not with each one by itself
- Example where T1, T2, T3 can all be run under SI, but when T4 is present, we need to fix things in T1
- Non-serializable execution can involve read-only transactions, not just updaters

Overview

1. Review of databases, isolation levels and serializability
2. Theory to determine whether an application will have serializable executions when running at SI
3. Modifying applications
4. Fixing the DBMS
5. Replicated databases
6. Future work

Making Snapshot Isolation Serializable [ACM TODS, 2005]

Alan Fekete*, Dimitrios Liarokapis , Elizabeth O'Neil, Patrick O'Neil, Dennis Shasha**

*University of Sydney U. Massachusetts, Boston **NYU

Multiversion Serializability Theory

- From Y. Raz in RIDE’93
  - Suitable for multiversion histories
- WW-conflict from T1 to T2
  - T1 writes a version of x, T2 writes a later version of x
    - In our case, succession (version order) defined by commit times of writer txns
- WR-conflict from T1 to T2
  - T1 writes a version of x, T2 reads this version of x (or a later version of x)
- RW-conflict from T1 to T2 (Adya et al ICDE’00 called this “antidependency”)
  - T1 reads a version of x, T2 writes a later version of x
- Serializability tested by acyclic conflict graph

Interference Theory

- We produce the “static dependency graph”
  - Node for each application program
  - Draw directed edges each of which can be either
    - Non-vulnerable interference edge, or
    - Vulnerable interference edge
- Based on looking at program code, to see what sorts of conflict situations can arise
- More complicated with programs whose accesses are controlled by parameters
- A close superset of SDG can be calculated automatically in some cases

Edges in the SDG

- Non-vulnerable interference edge from T1 to T2
- Conflict, but it can't arise transactions can run concurrently
  - Eg "ww" conflict
    - Concurrent execution prevented by FCW
  - Or "wr" conflict
    - conflict won't happen in concurrent execution due to reading old version
- Eg
  - T1 = R1(x) R1(y) W1(x)
  - T2 = R2(x) R2(y) W2(x) W2(y)

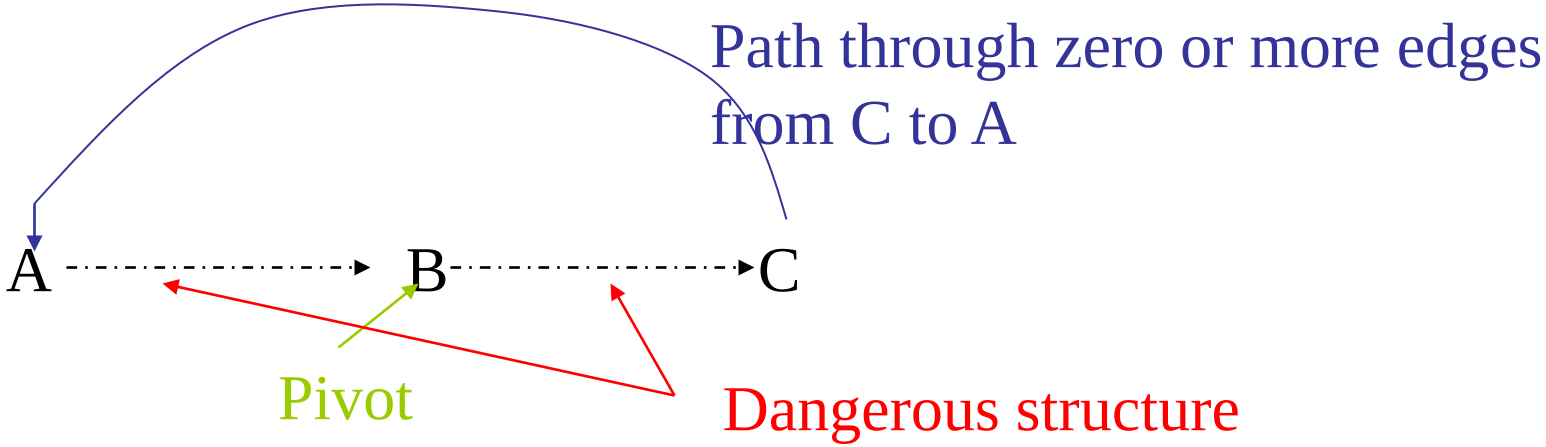
- Vulnerable interference edge from T1 to T2
- Conflict can occur when transactions run concurrently
  - Eg "rw without ww": rset(T1) intersects wset(T2), and wset(T1) disjoint from wset(T2)
- Eg
  - T1 = R1(x) R1(y) W1(x)
  - T2 = R2(x) R2(y) W2(y)
- Shown as dashed edge on diagram

Paired edges

- In SDG, an edge from X to Y implies an edge from Y to X
- But the type of edge is not necessarily the same
  - Both vulnerable, or
  - Both non-vulnerable, or
  - One vulnerable and one non-vulnerable

Dangerous Structures

- A *dangerous structure* is two edges linking three application programs, A, B, C such that
  - There are successive vulnerable edges (A,B) and (B,C)
  - (A, B, C) can be completed to a cycle in SDG
    - Call B a pivot
  - Special case: pair A, B with vulnerable edges in both directions



The main result

- Theorem: If the SDG does not contain a dangerous cycle, then every execution is serializable (with all transactions using SI for concurrency control)
  – Applies to TPC-C benchmark suite

Example: SmallBank Benchmark

- Traditional benchmarks (e.g. TPC-C) are already serializable under SI
- SmallBank benchmark: designed to have non-serializable executions under SI
  - three tables:
    Account, Saving, Checking
  - five transactions of a banking scenario:
    Balance, WriteCheck, DepositChecking, TransactionSaving, Amalgamate

SmallBank Dependencies

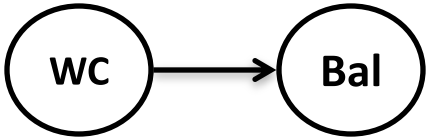
- **Read-Dependencies(WR):**

WriteCheck (N,V):

```
…
UPDATE Account
SET bal=bal-V
WHERE custid=x;
COMMIT
```

Balance(N):

```
...
SELECT bal
FROM Account
WHERE custid=x;
...
```



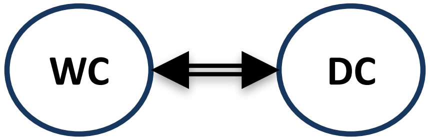
- **Write-Dependency(WW):**

WriteCheck (N,V):

```
...
UPDATE Account
SET bal=bal-v
WHERE custid=x;
...
```

DepositChecking (N,V):

```
...
UPDATE Account
SET bal=bal+V
WHERE custid=x;
...
```



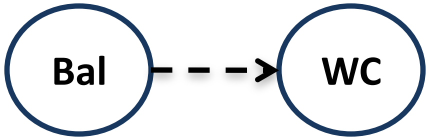
- **Anti-Dependencies(RW):**

Balance(N):

```
...
SELECT bal
FROM Account
WHERE custid=x;
…
```

Writecheck(N,V):

```
...
UPDATE Account
SET bal=bal-V
WHERE custid=x;
…
```



SDG of SmallBank

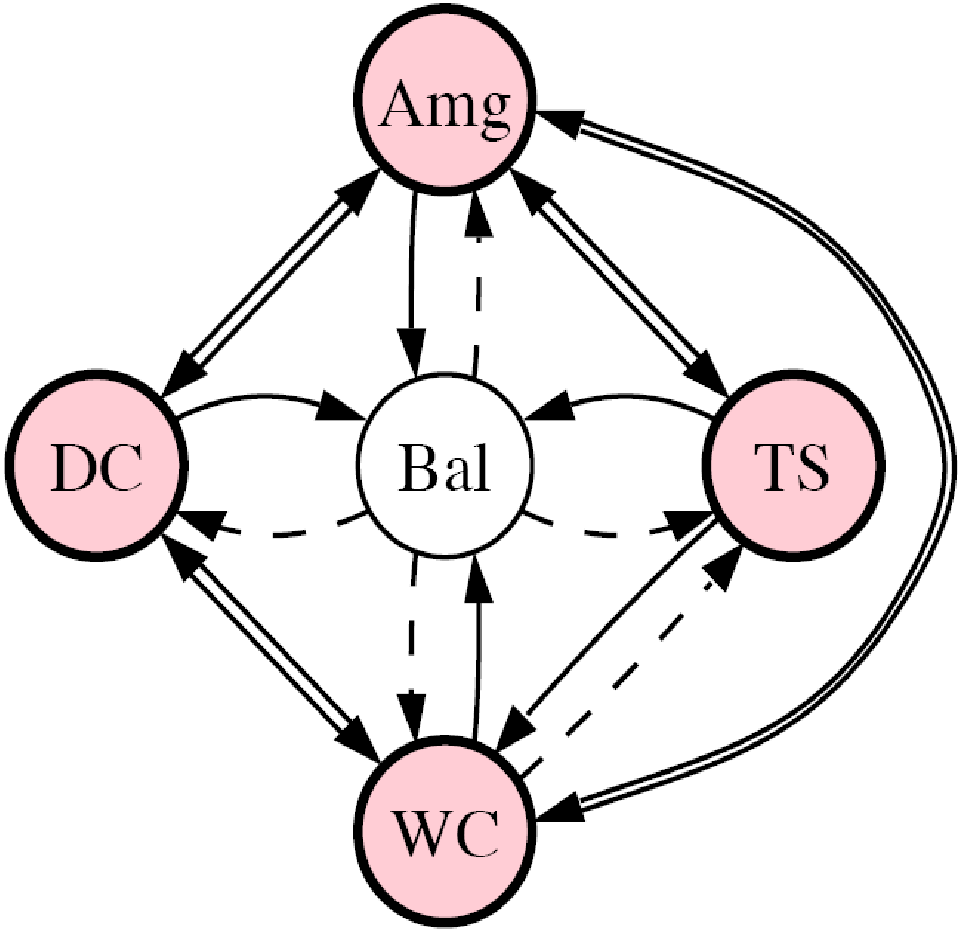
1-Balance (Bal)

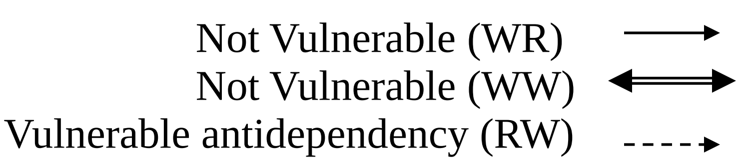
2-Amalgamate (Amg)

3-DepositChecking (DC)

4-TransactionSaving (TS)

5-WriteCheck (WC)





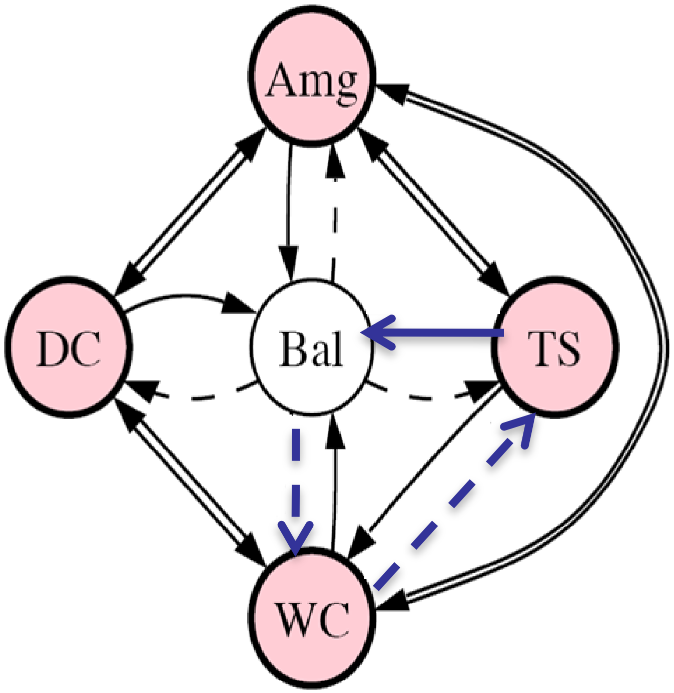
Analysis of SmallBank's SDG

What is the dangerous structure???

nodes A, B, and C:

- **anti-dependency A - - → B**
- **anti-dependency B - - → C**
- **path from C to A or A=C**

- **In this case, only dangerous structure is Bal - - → WC - - → TS**



Main theorem: Proof Sketch I (Find crucial feature in CSG)

- In any cycle in CSG, there exists
  - TA to TB have rw-dependency, and are concurrent
  - TB to TC have rw-dependency, and are concurrent
- Here TC is earliest committer among the cycle
- Case analysis relating types of dependency edge to ordering between start/commit times

Main theorem: Proof Sketch II (Relate CSG and SDG)

- If TA to TB is in CSG, then TA to TB is in SDG
- If edge in CSG has rw-dependency and transactions are concurrent, then edge in SDG is vulnerable

Main theorem: Proof Sketch III

- Assume existence of non-serializable execution
- So exists cycle in CSG
- So has special structure
  - TA to TB to TC, each being (rw and concurrent)
- So cycle in SDG with consecutive vulnerable edges
  - dangerous structure
- Contradiction, if SDG has no dangerous structure

Overview

1. Review of databases, isolation levels and serializability
2. Theory to determine whether an application will have serializable executions when running at SI
3. Modifying applications
4. Fixing the DBMS
5. Replicated databases
6. Future work

A Robust Technique to Ensure Serializable Executions with Snapshot Isolation DBMS[ICDE'09]

Mohammad Alomari, Alan Fekete, Uwe Röhm

University of Sydney

Modifying application code

- DBA modifies one or more of the programs that make up the mix
- Modifications should not alter the observed semantics of any program
- Modified set of programs should have all executions serializable
  – So modified SDG has no dangerous structure

Decisions

- Decide **WHERE**: choose a set of edges containing at least one from each a dangerous structure
  - Finding a minimal set is NP-Hard
  - One easy choice: choose ALL vulnerable edges
- Decide **HOW**: introduce ww conflict on chosen edges
  - Without changing program semantics
  - Materialize or Promotion or External Locking
- Outcome: modified application mix has SDG where each chosen edge is not vulnerable
  - Modified application SDG has no dangerous structure

Approach 1: Materialize the Conflict

Both programs in the chosen edge get an extra update to a new table that is not used elsewhere in the application

•target row parameterized so FCW conflict happens exactly when txns have rw-dependency

Writecheck(N,V):

```
.....
SELECT Custid INTO X
FROM Account
WHERE Name=N;

UPDATE Conflict
SET value=value+1
WHERE custid=X;

UPDATE checking
SET bal=bal-V
WHERE custid=X;
.....
```

Conflict Table

CustomerID	Value
500345	50.00

First Committer Wins Rule

RW-dependency

Balance(N):

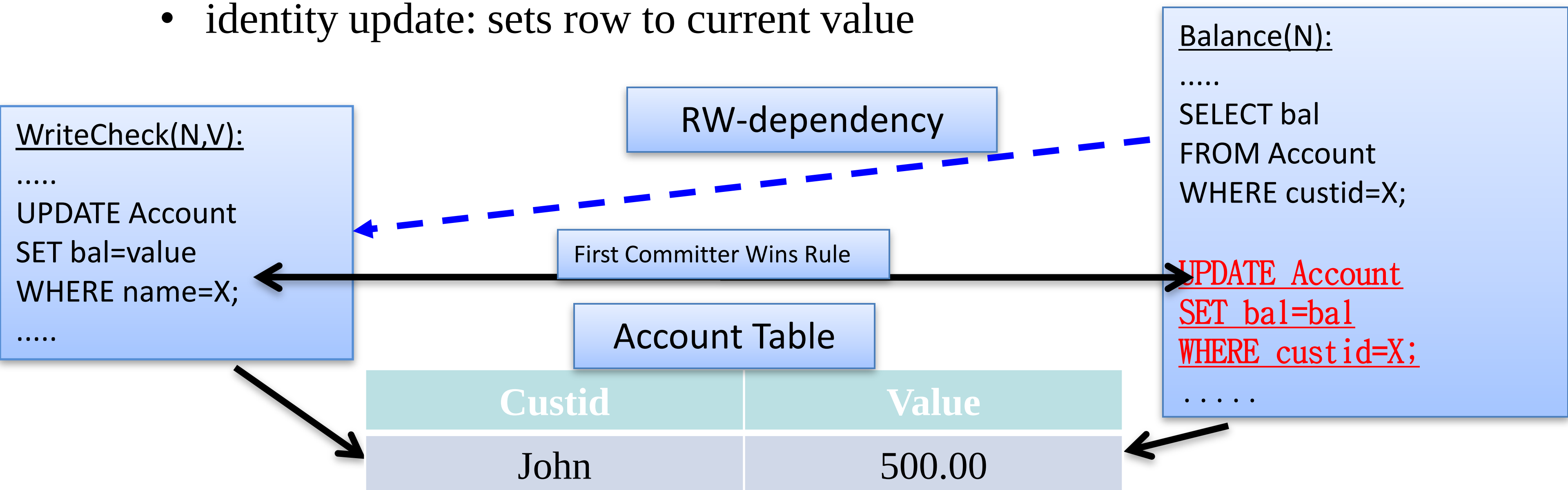
```
.....
SELECT Custid INTO X
FROM Account
WHERE Name=N;

UPDATE Conflict
SET value=value+1
WHERE custid=X;

SELECT bal
FROM Checking
WHERE custid=X;
```

Proposed in Fekete et al, TODS 2005

Approach 2: Promote a Read

Source program of chosen edge gets an extra update to the row which is in rw-dependency

- identity update: sets row to current value



Custid	Value
John	500.00

Proposed in Fekete et al, TODS 2005

In Oracle, can use SELECT FOR UPDATE to get the FCW check as if this actually did a write

Doesn’t work this way in other platforms like MS SQL Server

Approach 3:External Lock (ELM)

Each transaction in the chosen edge is surrounded by explicitly lock/unlock on a suitable set of parameters

Lock(N)

```
Writecheck(N,V):
.....
SELECT Custid INTO X
FROM Account
WHERE Name=N;

UPDATE checking
SET bal=bal-V
WHERE custid=X;
......
```

Commit

Release(N)

Account Table

CustomerID	Value
500345	50.00

Lock (N)

```
Balance(N):
.....
SELECT Custid INTO X
FROM Account
WHERE Name=N;

SELECT bal
FROM Checking
WHERE custid=X;
...
```

Commit

Release(N)

Why ELM is different from 2PL?

- Transactions that are not involved in chosen edges do not set locks at all
- There are only exclusive locks, no shared locks
- Even if a transaction touches many objects, it may need to lock only one or a few string values
- All locking is done at the start of the transaction, before any database activity has occurred
- It can be implemented without risk of deadlock

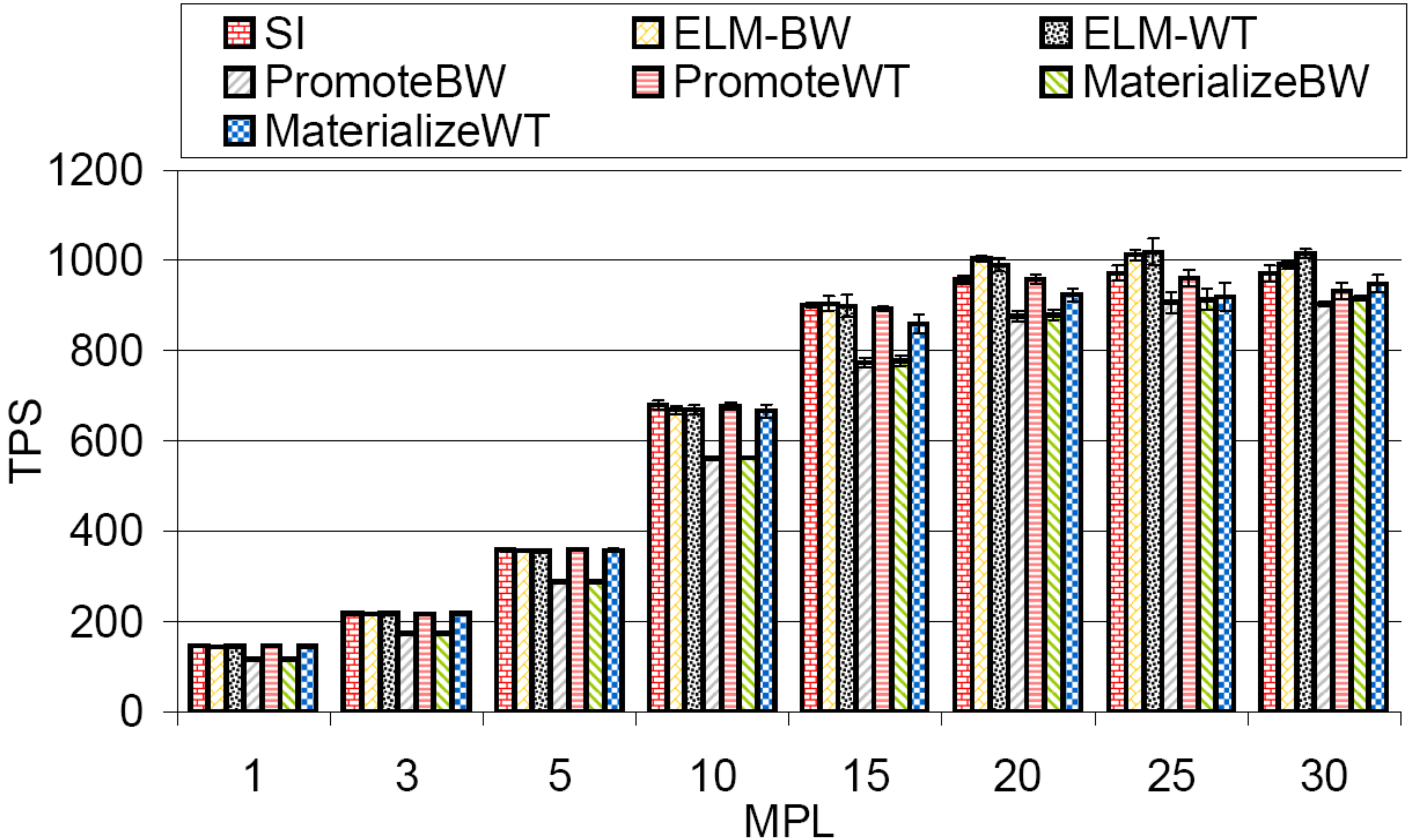
Performance impact

- Does modification impact much on performance?
- For SmallBank, DBA could
  - Choose a minimal edge set which is just Bal - - → WT (call this choice BW)
  - Choose a minimal edge set WT - - → TS (call this choice WT)
  - Choose ALL vulnerable edges
- Each can be done by Materialize or Promotion or ELM
- This gives at least 9 options for DBA to modify application; which gives best performance?
- We take performance of SI as “target” (but we try to get this level of performance as well as serializability)

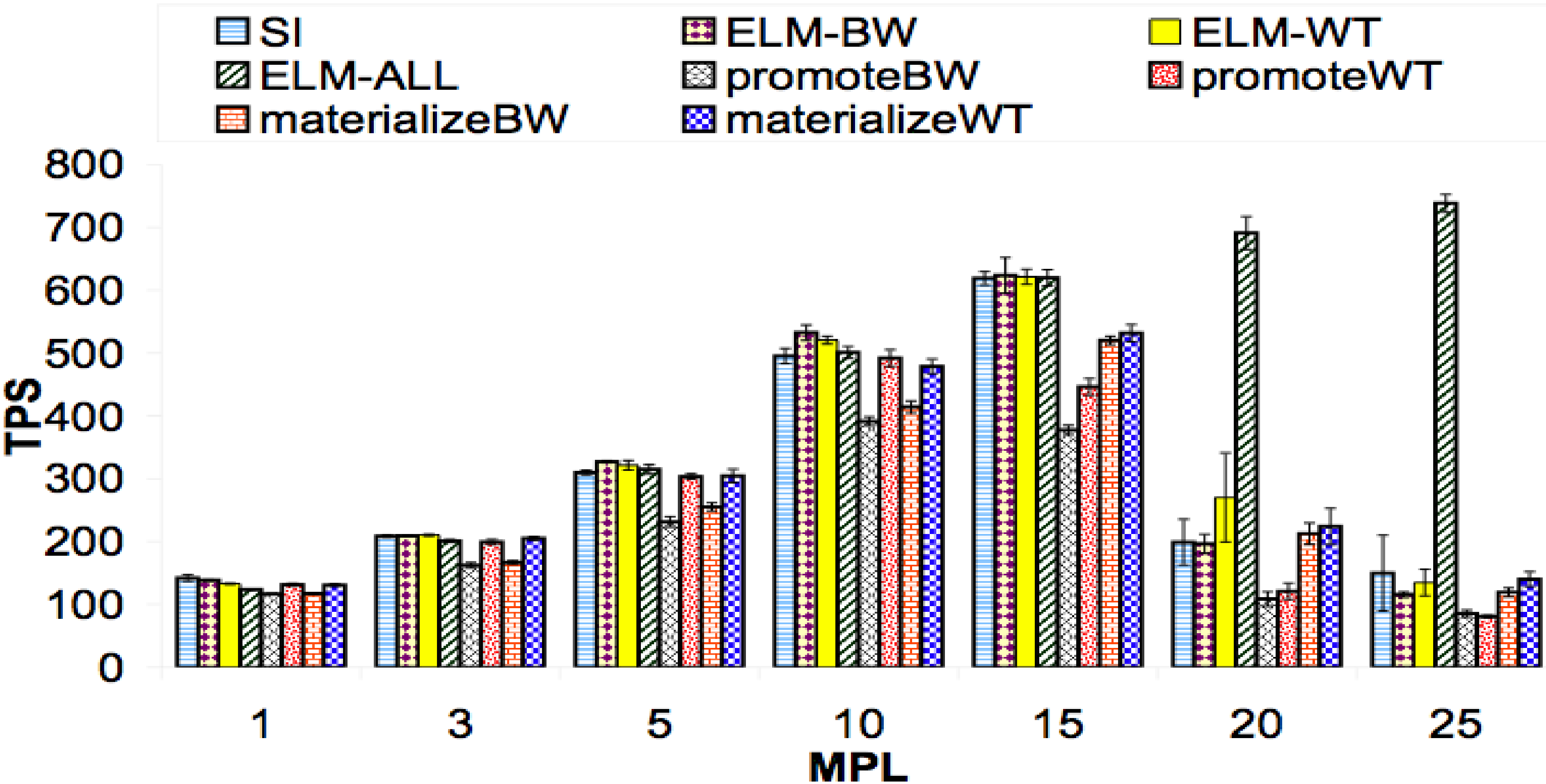
Experiment Setup

- Evaluating techniques on PostgreSQL 8.2 and a commercial platform offering SI
- Multi-threaded client executing SmallBank transactions using stored procedures
  - Each thread chooses one transaction type randomly
  - a ramp-up period 30 second followed by one minute measurement interval
- **Parameters:**
  Choice of SDG edges on which to introduce conflict, technique to introduce conflict, low & high contention scenarios (controlled by size of hotspot getting 90% of accesses)

PostgreSQL-Low contention



Commercial Platform - Low contention



Modifying applications: Lessons

- Choice of edge set really matters with promote or Materialize
- Some choices can suffer substantial loss of performance compared to SI
  - It is not wise to place write operations in a previously read-only txn
- ELM gets good performance for all the various edge sets
  - ELM can even get better performance than SI under contention, because locks on an edge also lead to blocking on self-loops of SDG, where ww-conflicts lead to frequent aborts with SI

Allocating Isolation Levels to Transactions [PODS’05]

Alan Fekete

University of Sydney

Mixing isolation levels

- Theory usually assumes one cc mechanism for the dbms
- But in fact different txns can use different mechanisms
- Either declaratively, by setting “isolation level”
- Or programatically, by explicit LOCK TABLE and UNLOCK TABLE statements

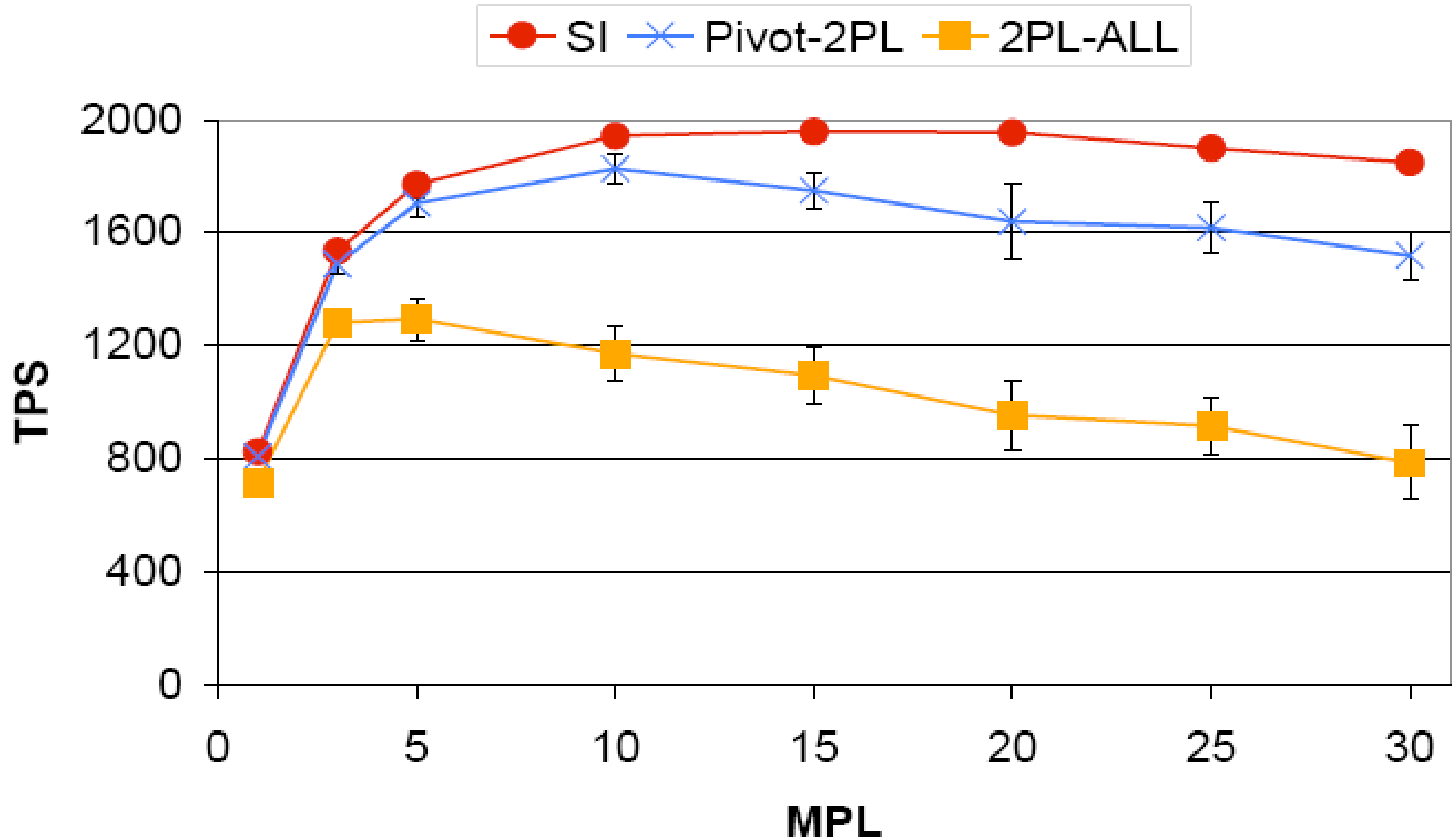
Alternative: allocate isolation levels

- Can we ensure serializable execution without modifying application code?
  - Just set isolation level for each transaction appropriately
  - In configuration, or at session establishment
- Potential advantage: don't need to modify application source

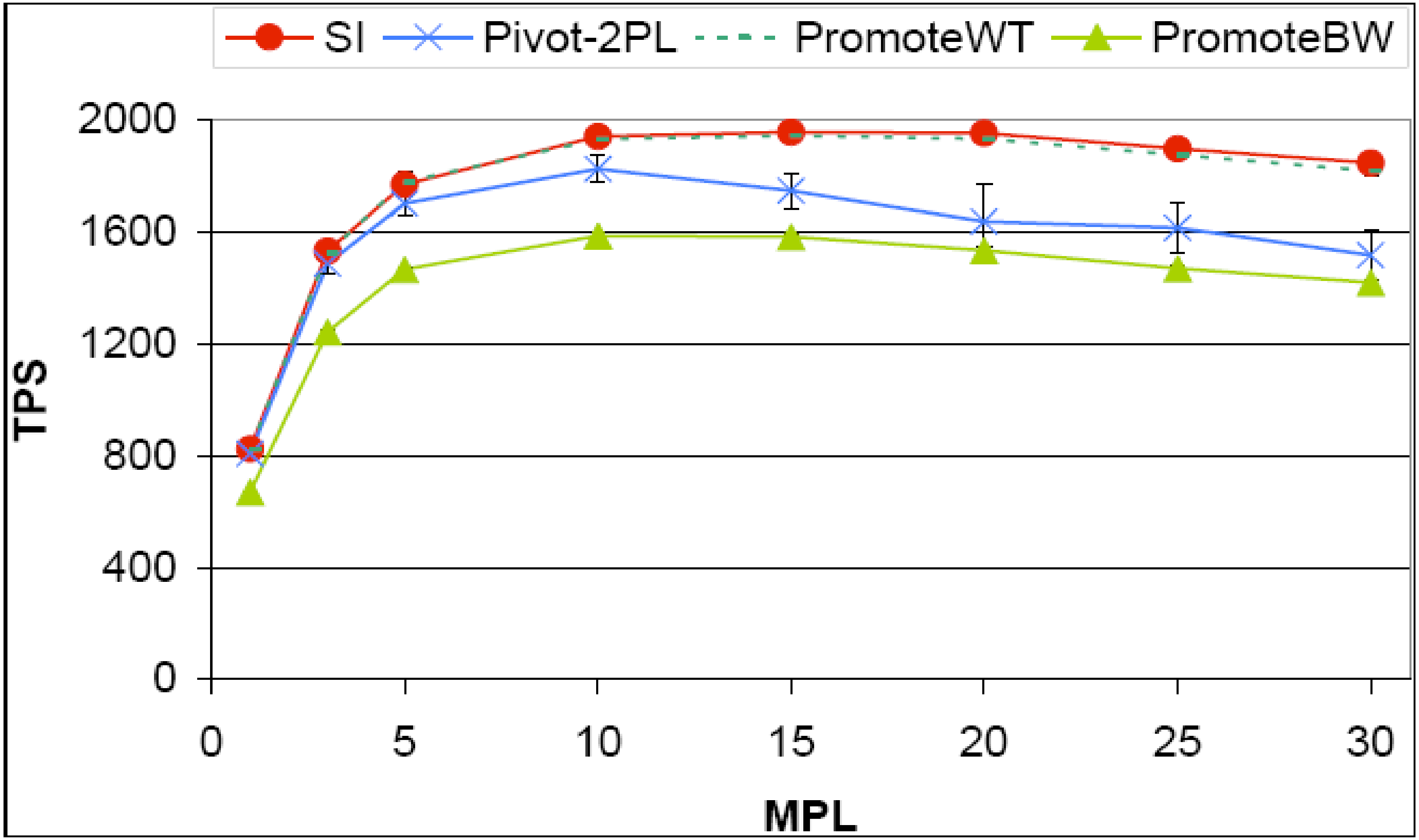
Extension of theory

- Allocate some transactions to use 2PL and others to use SI
  - Eg on MS SQLServer 2005
- Theorem: If every pivot uses 2PL, then every execution is serializable (with other transactions using either 2PL or SI for concurrency control)
  - Minimal set of transactions to run with 2PL is the set of pivots (call this approach Pivot2PL)
  - Of course, using 2PL for ALL transactions guarantees serializable execution; this is a maximal set

Mixing Isolation Levels; Low Contention



Compare to application modification



Allocating Isolation Levels: Lessons

- Can lose quite a bit of SI's performance
- Generally, it would be better for the DBA to get the information needed and make a wise choice of how to modify application code
  - If they have permissions etc to do so

Overview

1. Review of databases, isolation levels and serializability
2. Theory to determine whether an application will have serializable executions when running at SI
3. Modifying applications
4. Fixing the DBMS
5. Replicated databases
6. Future work

Serializable Isolation for Snapshot Databases [Sigmod’08 “Best paper”, then ACM TODS 2009]

Michael Cahill, Alan Fekete, Uwe Röhm

University of Sydney

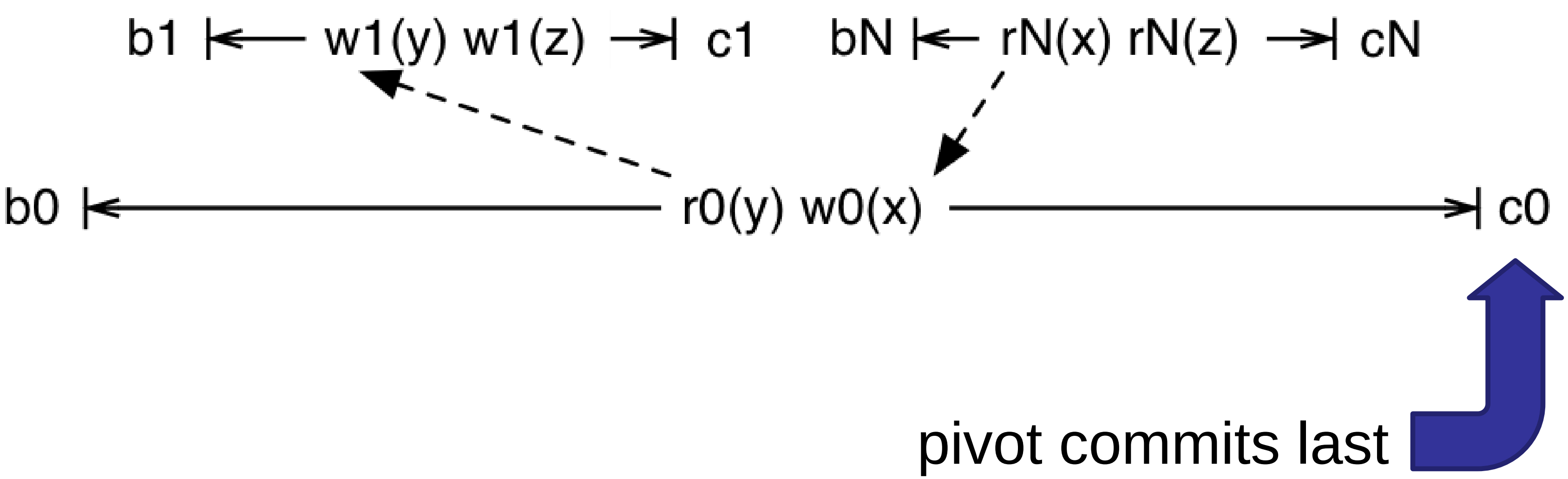
Serializable SI

- If we can alter the DBMS, we could provide a new algorithm for serializable isolation
  - Online, dynamic
  - Modifications to standard Snapshot Isolation
- To do so:
  - Keep versions, read from snapshot, FCW (like SI)
  - Detect read-write conflicts at runtime
  - Abort transactions with consecutive rw-edges
    - Much less often than traditional optimistic CC
    - Don’t do full cycle detection

Challenges

- During runtime, rw-pairs can interleave arbitrarily
- Have to consider begin and commit timestamps:
  - which snapshot is a transaction reading?
  - can conflict with committed transactions
- Want to use existing engines as much as possible
- Low runtime overhead
- But minimize unnecessary aborts

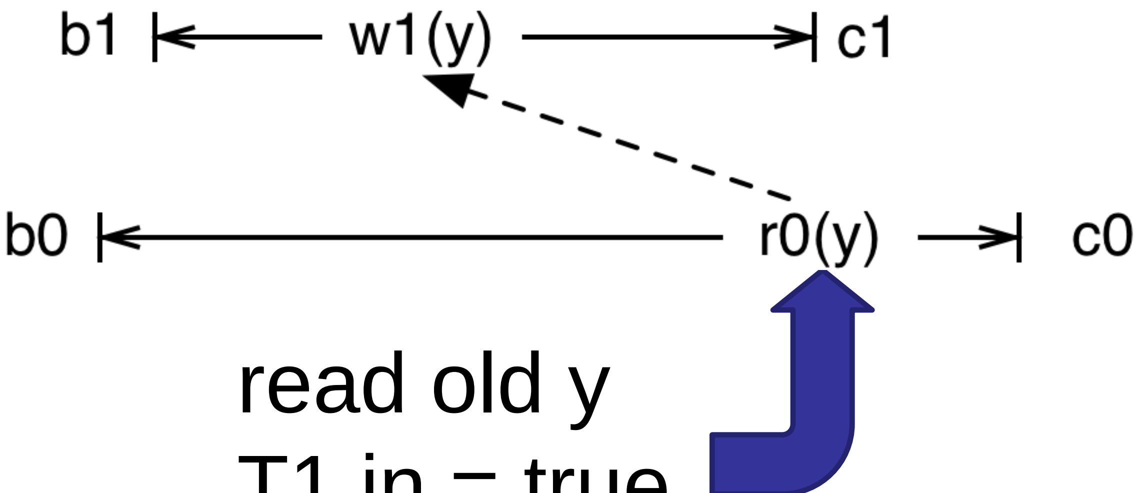
SI anomalies: a simple case



Algorithm in a nutshell

- Add two flags to each transaction (*in* & *out*)
- Set T0.out if *rw*-conflict T0 → T1
- Set T0.in if *rw*-conflict TN → T0
- Abort T0 (the pivot) if both T0.in and T0.out are set
  - If T0 has already committed, abort the conflicting transaction

Detection: write before read

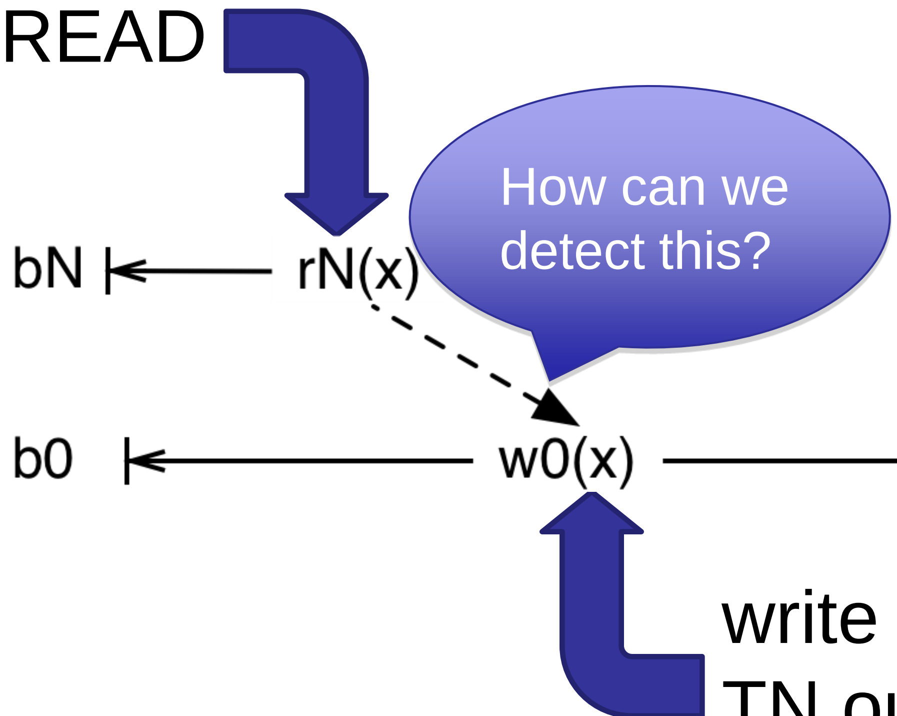


read old y

T1.in = true

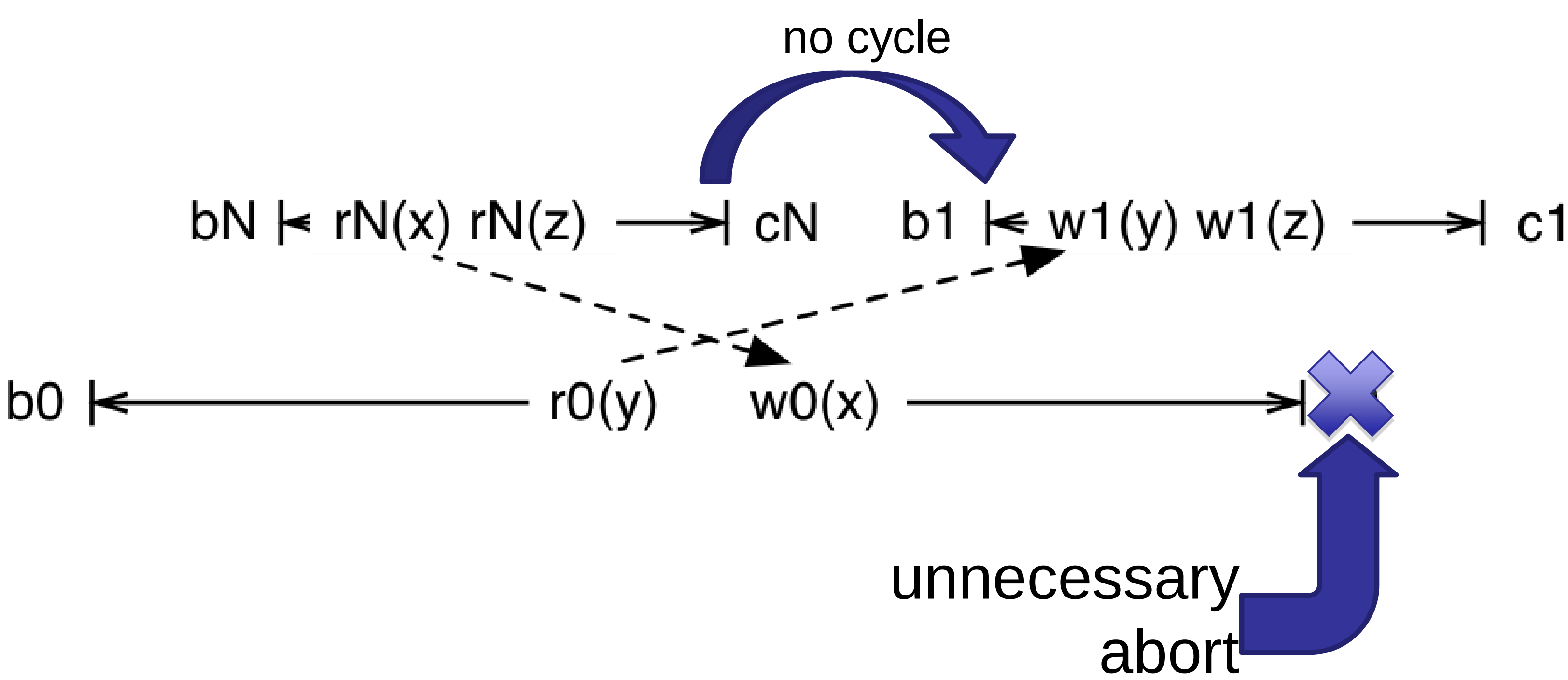
T0.out = true

Detection: read before write



SIREAD mode lock doesn't block anything
Just for record keeping
Kept even after transaction commits

Main Disadvantage: False positives



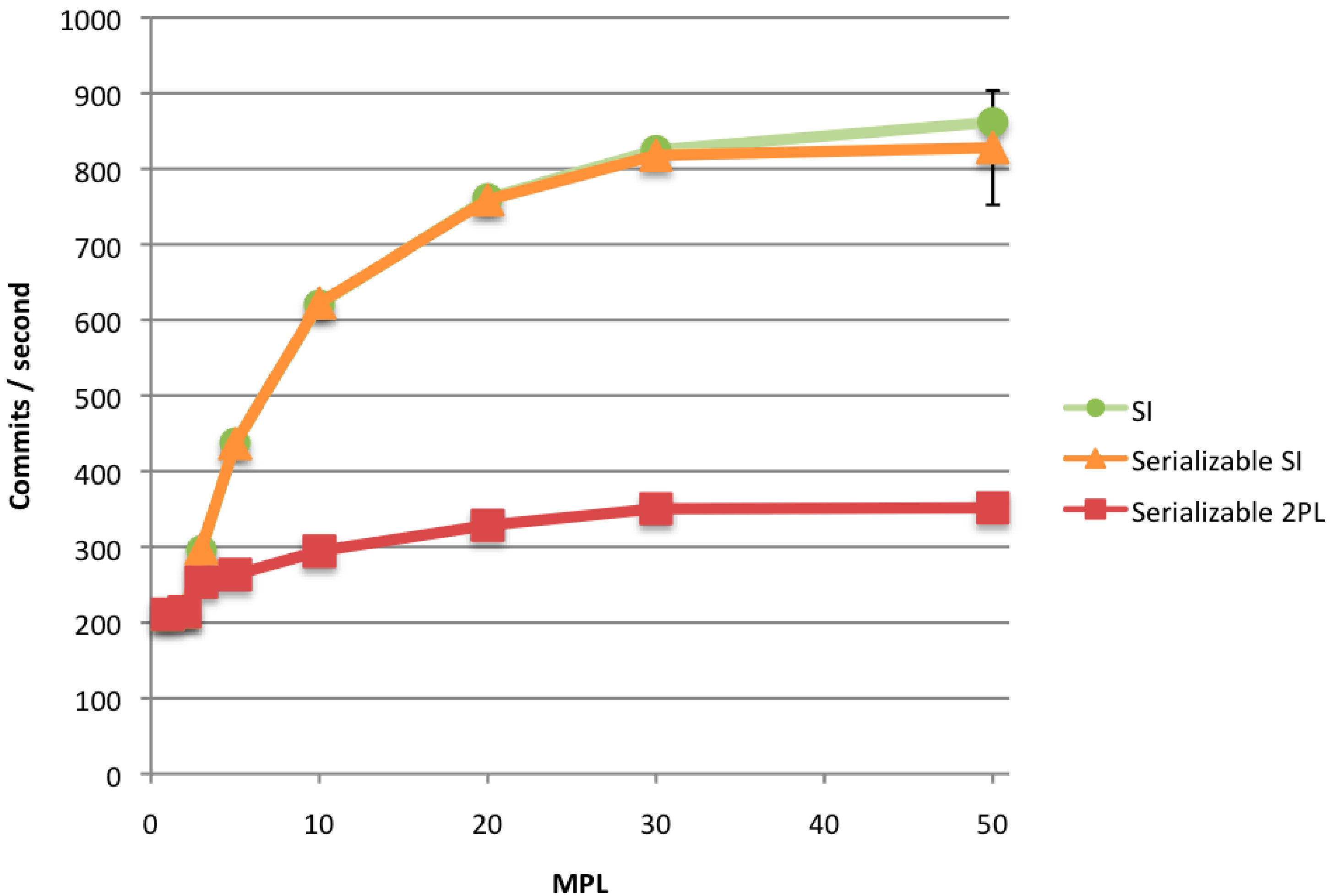
Prototype in Oracle InnoDB

- Implemented in Oracle InnoDB plugin 1.0.1
  - Most popular transactional backend for MySQL
  - Already includes multiversion concurrency control
- Added:
  - True Snapshot Isolation with first-committer-wins (InnoDB's "repeatable read" isolation has non-standard semantics)
  - Serializable SI, including phantom detection (uses InnoDBs next-key locking)
- Added 230 lines of code to 130K lines in InnoDB
  - Most changes related to transaction lifecycle management

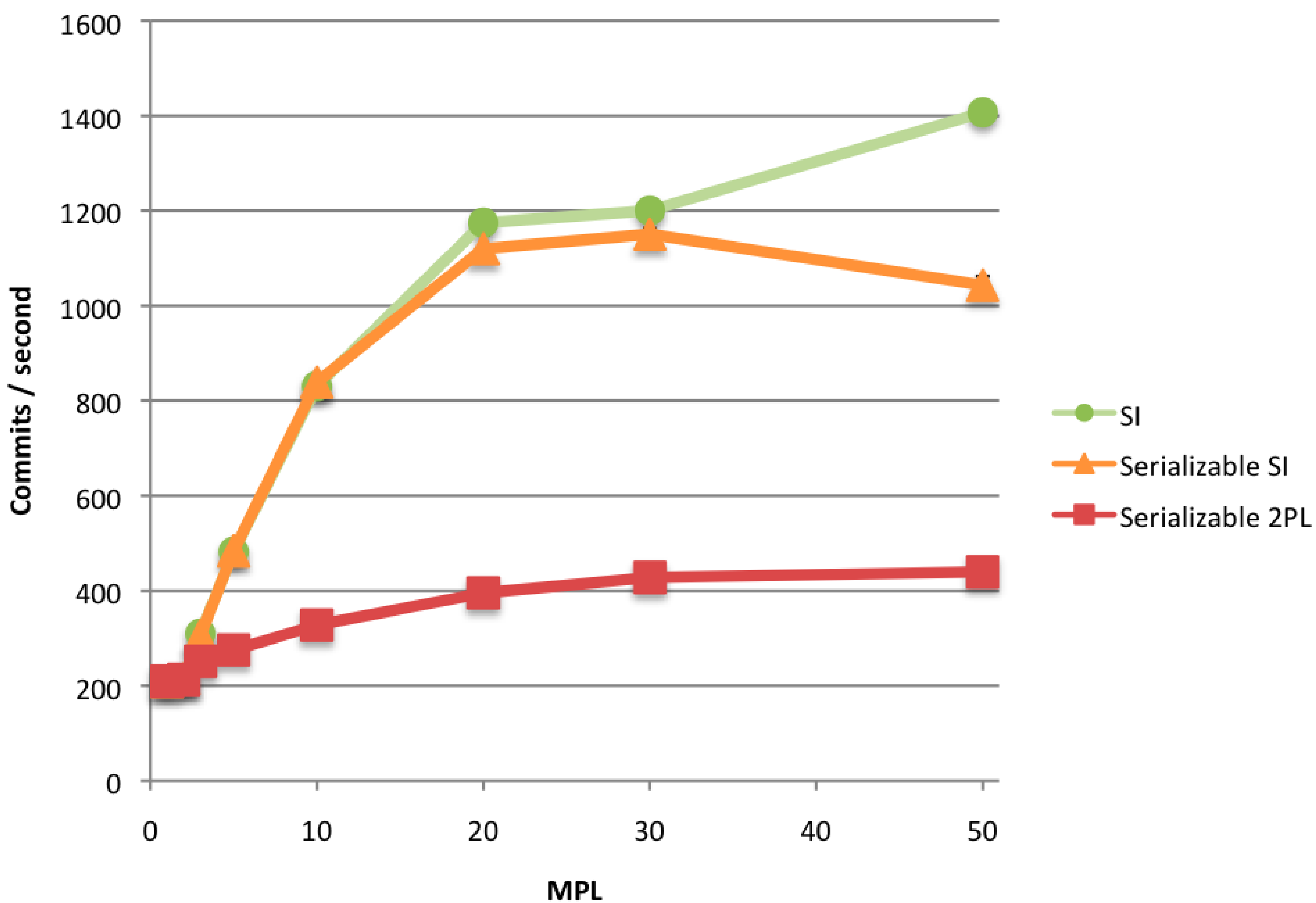
Experimental scenarios

- sibench – synthetic microbenchmark
  - conflict between sequential scan and updating a row
  - table size determines write-write conflict probability and CPU time required for scan
- TPC-C++ - modified TPC-C to introduce an SI anomaly
  - added a “credit check” transaction type to the mix
  - measured throughput under a variety of conditions
  - most not sensitive to choice of isolation level, but we found a mix favoring “stock level” transactions that demonstrates the tradeoff

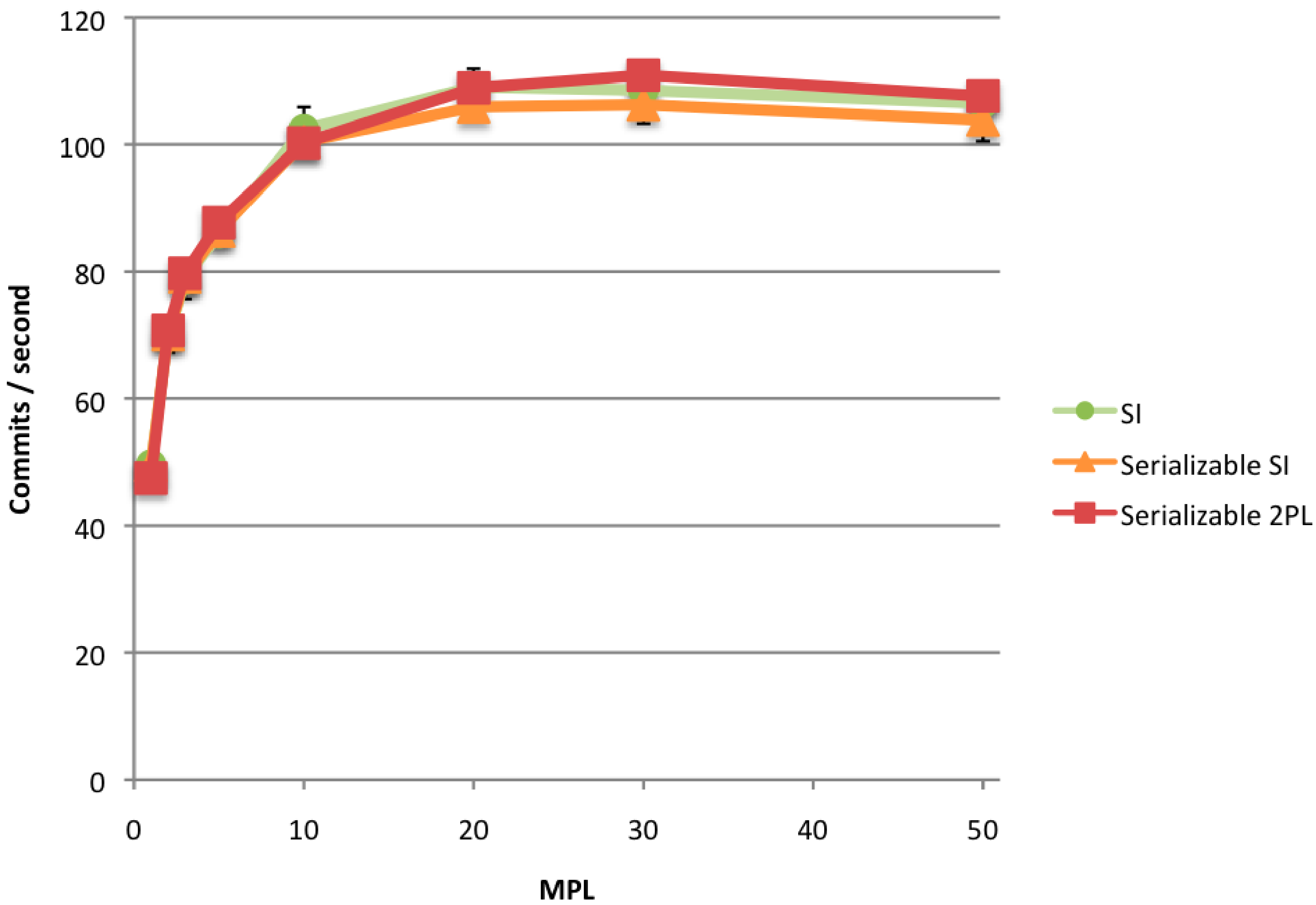
sibench: 10 reads per write



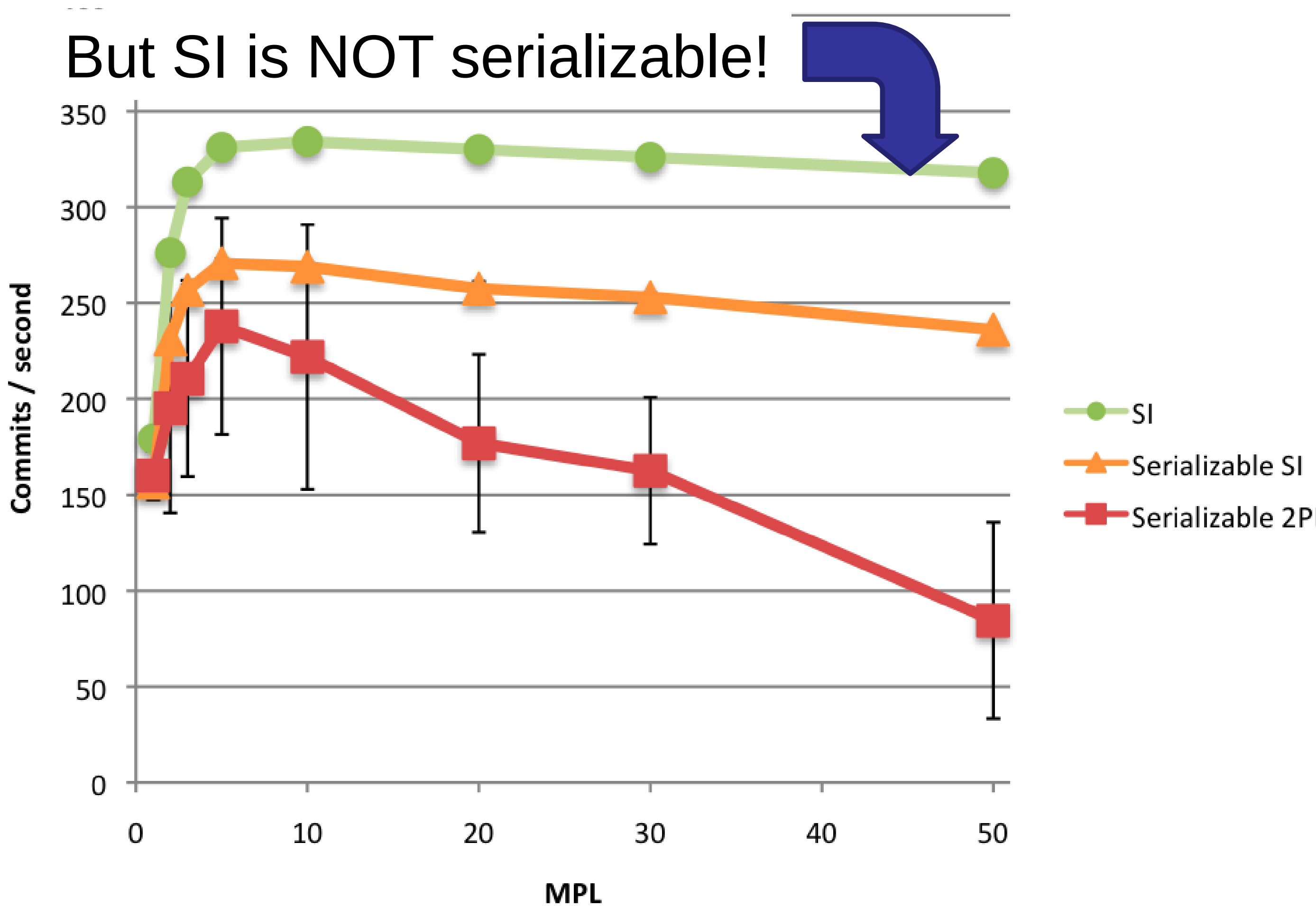
sibench: 100 reads per write



TPC-C++: 10 warehouses



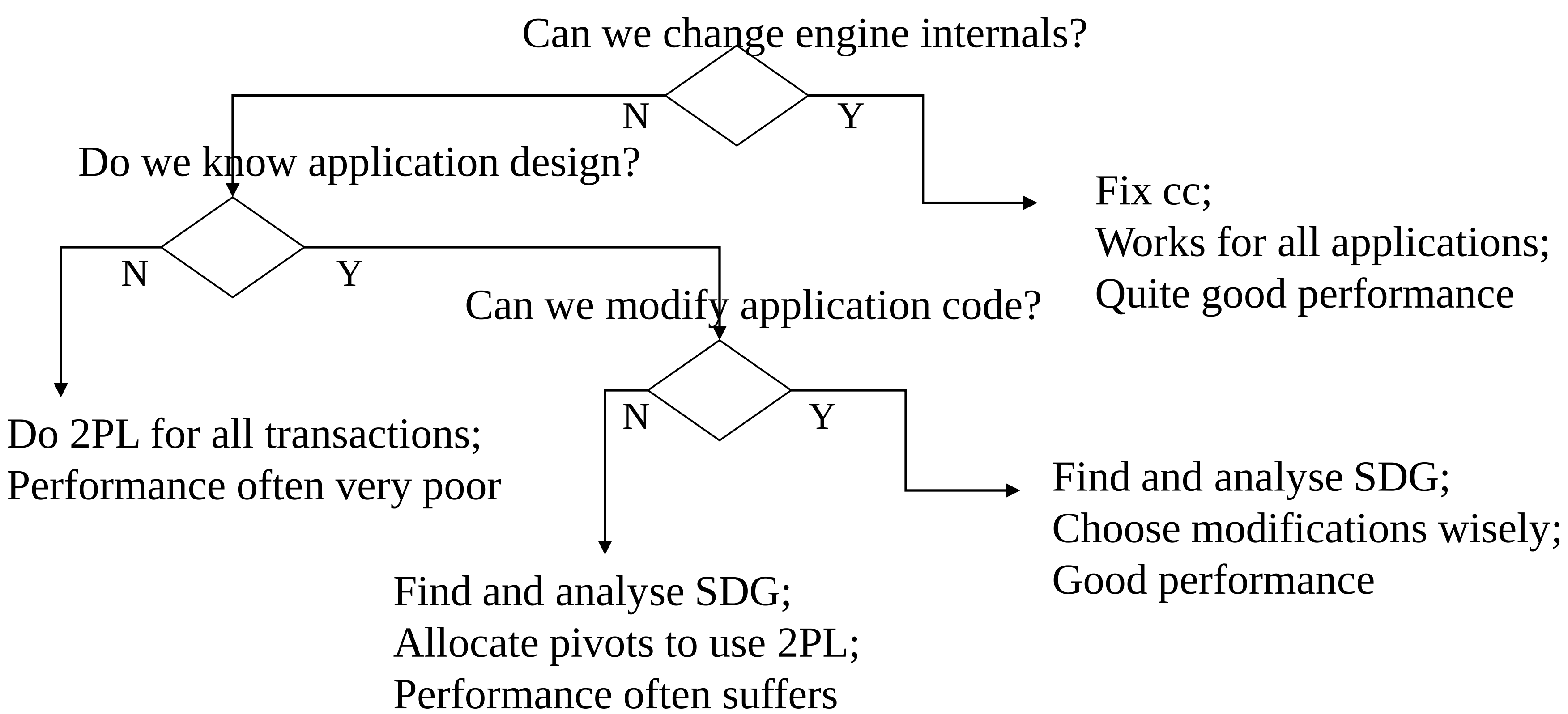
TPC-C++: special "stock level" mix



Serializable SI: Lessons

- New algorithm for serializable isolation
  - Online, dynamic, and general solution
  - Modification to standard Snapshot Isolation
  - Keeps the features that make SI attractive: Readers don't block writers, much better scalability than S2PL
- In most cases, performance is comparable with SI
- Never worse than locking serializable isolation
- Feasible to add to an RDBMS using Snapshot Isolation (such as Oracle) with modest changes
  - PostgreSQL release 9.1 has done this – Isolation Level Serializable now executes serializably!

Summary


Can we change engine internals?
N
Y
Fix cc;
Works for all applications;
Quite good performance
Do we know application design?
N
Y
Do 2PL for all transactions;
Performance often very poor
Can we modify application code?
N
Y
Find and analyse SDG;
Allocate pivots to use 2PL;
Performance often suffers
Find and analyse SDG;
Choose modifications wisely;
Good performance

Overview

1. Review of databases, isolation levels and serializability
2. Theory to determine whether an application will have serializable executions when running at SI
3. Modifying applications
4. Fixing the DBMS
5. Replicated databases
6. Future work

Serializable Snapshot Isolation for Replicated Databases in High-Update Scenarios [VLDB'11]

Hyungsoo Jung Hyuck Han* Alan Fekete Uwe Röhm

University of Sydney

***Seoul National University**

Our Approach

- Update anywhere-anytime-anyway transactional replication
- **1-copy SR** over SI replicas
- New **theorem** (extension of [TODS2005], with extra properties to reduce false positive aborts)
- System design and **prototype** implementation
  - Detect read-write conflicts at **commit time**.
  - Abort transactions with *a certain pattern of consecutive* rw*-edges*
  - Retrieving complete rw-dependency information without propagating entire readsets.

Previous Work for 1-copy SR over SI

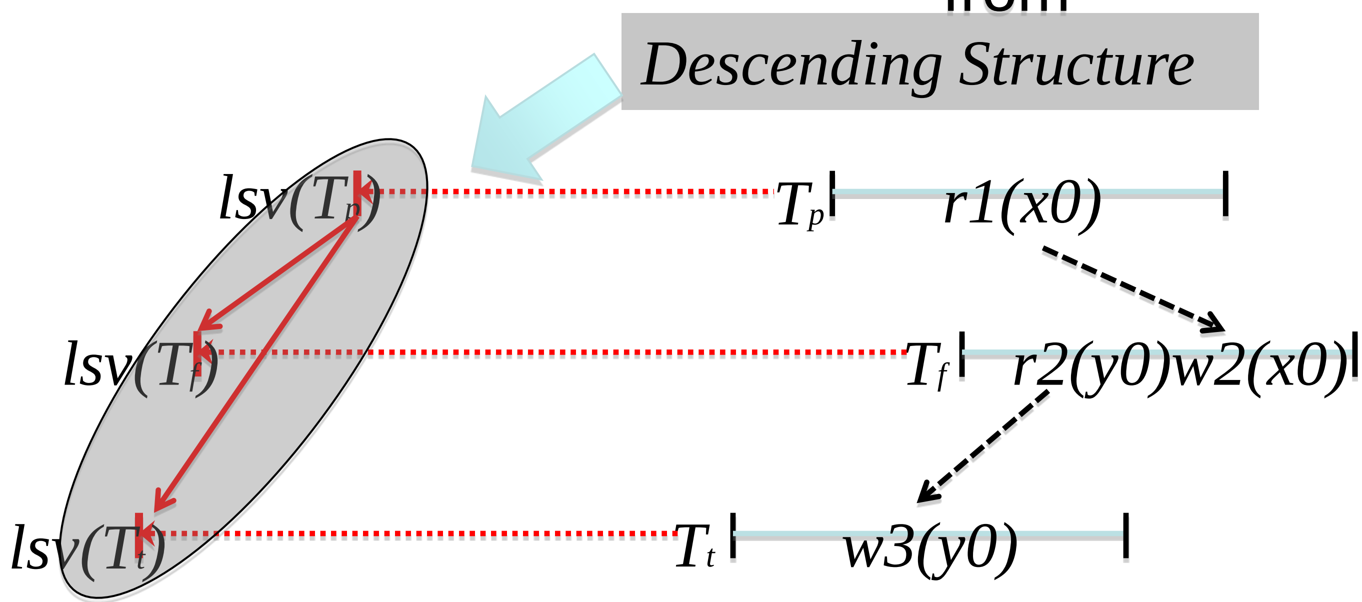
[Bornea et al., ICDE2011]

	Bornea et al.	This Work
Architecture	Middleware	Kernel
Readset Extraction	SQL parsing	Kernel interception
Certification	*ww*-conflict 1 *rw*-edge	*ww*-conflict 2 *rw*-edges
Optimized for	Read mostly	Update heavy

Descending Structure

- There are three transactions T_p, T_f *and* T_t with the following relationships:
  1. $T_p \xrightarrow{rw} T_f$ *and* $T_f \xrightarrow{rw} T_t$
  2. $lsv(T_f) \preceq lsv(T_p)$ && $lsv(T_t) \preceq lsv(T_p)$

lsv is a number we keep for each transaction: largest timestamp a transaction reads from



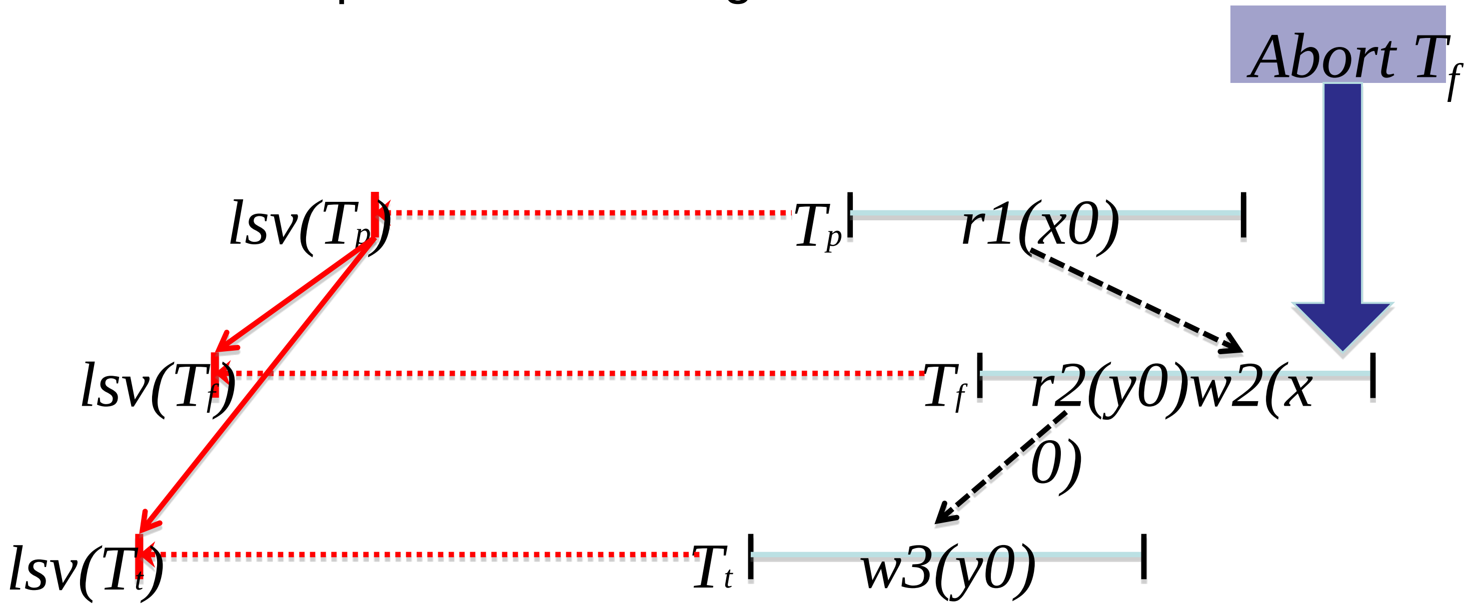
Main Theorem for 1-copy SR

- **Central Theorem**: *Let h be a history over a set of transactions obeying the following conditions*
  - *1-copy SI*
  - ***No descending structure***

  Then h is 1-copy serializable.

Concurrency Control Algorithm

- Replicated Serializable Snapshot Isolation (RSSI)
  - *ww*-conflicts are handled by 1-copy SI.
  - When certification detects a "*descending structure*", we abort whichever completes last among the three transactions.

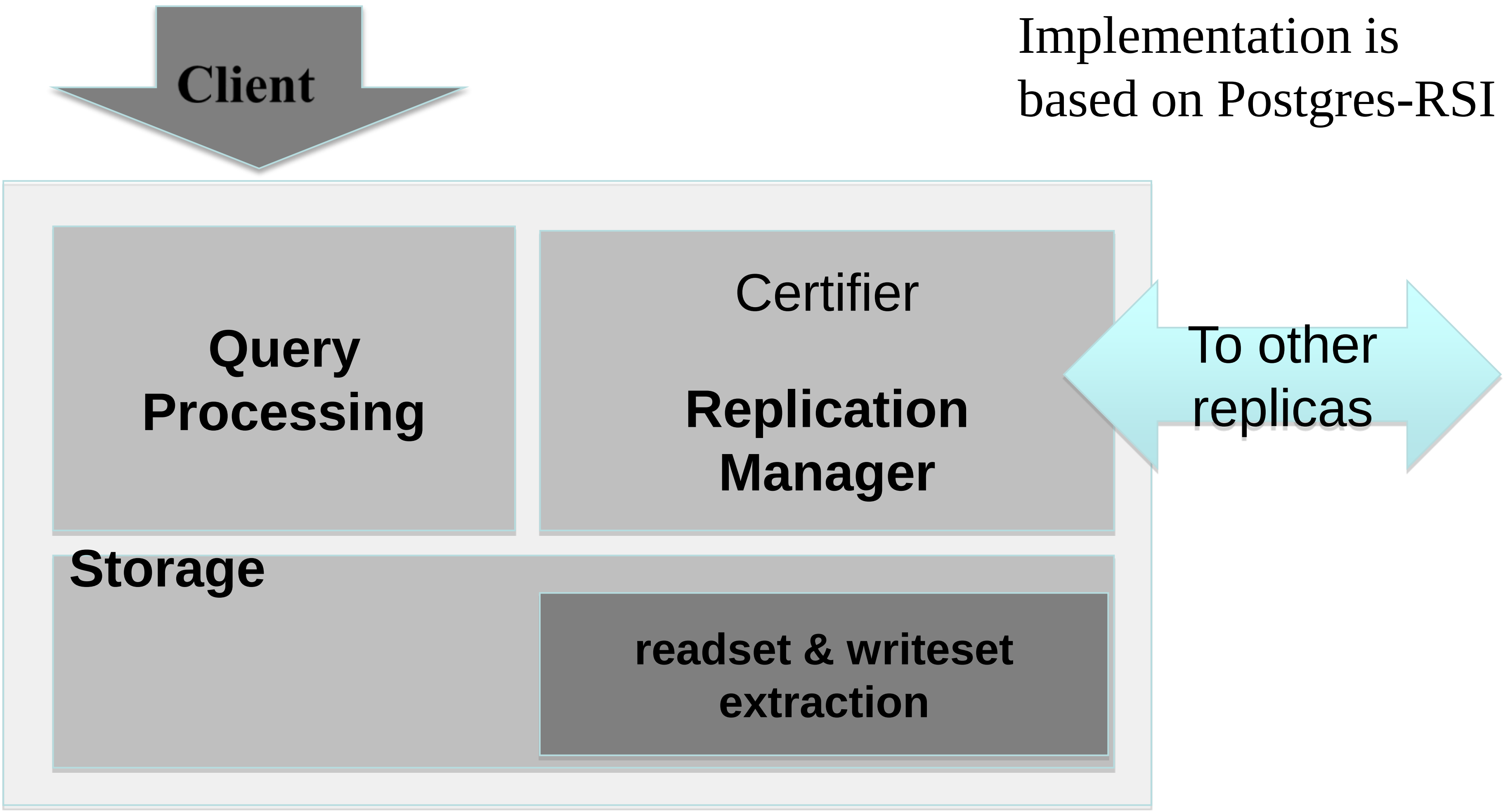


Technical Challenges

- The management of readset information and *lsv*-timestamps is pivotal to certification.
- We developed a global dependency checking protocol (GDCP) on top of LCR broadcast protocol [Guerraoui et al., ACM TOCS2010].
  - GDCP mainly performs two tasks at the same time:
    - Total order generation using existing LCR protocol.
    - Exchanging *rw*-dependency information without sending the entire readset.

In Each Participating Node

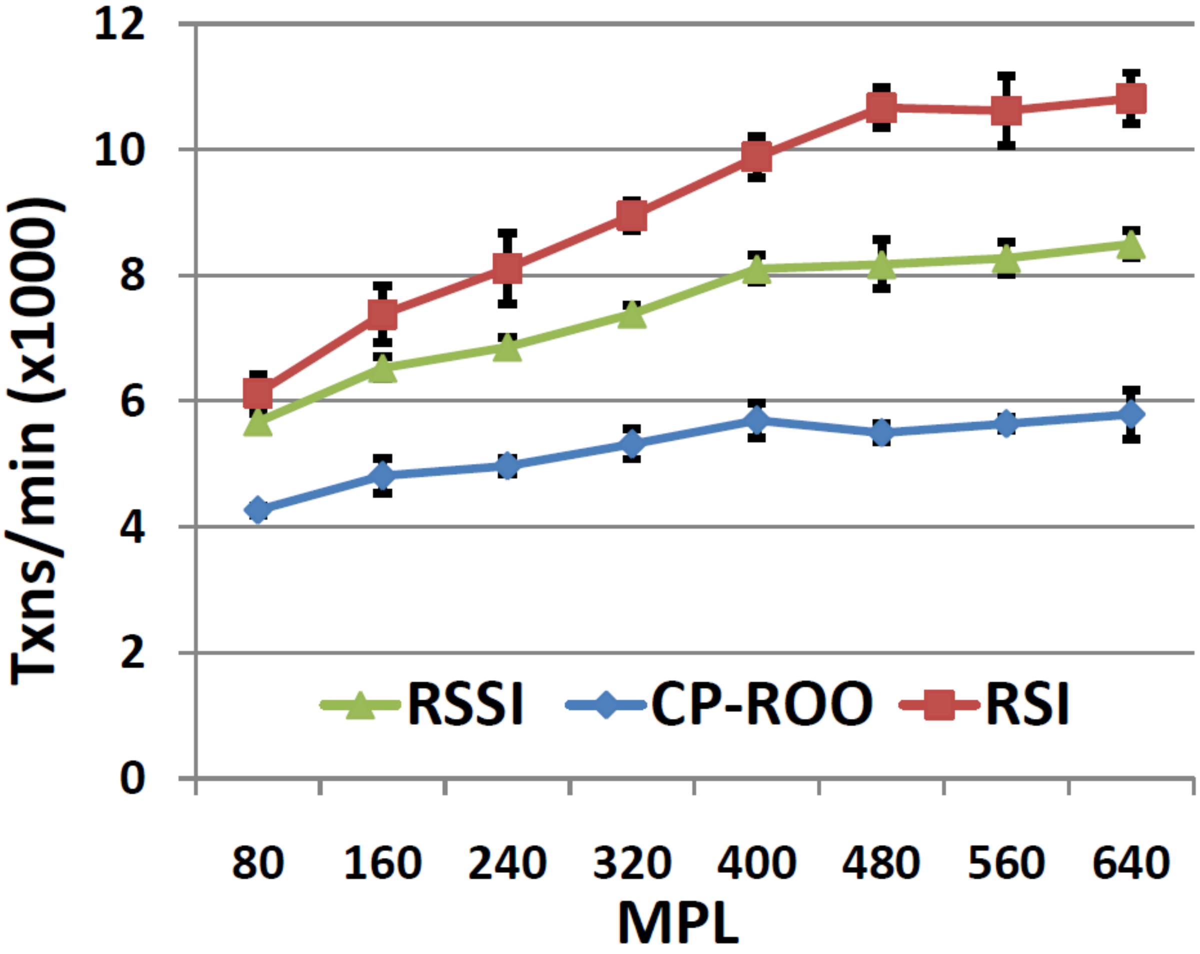
Implementation is based on Postgres-RSI



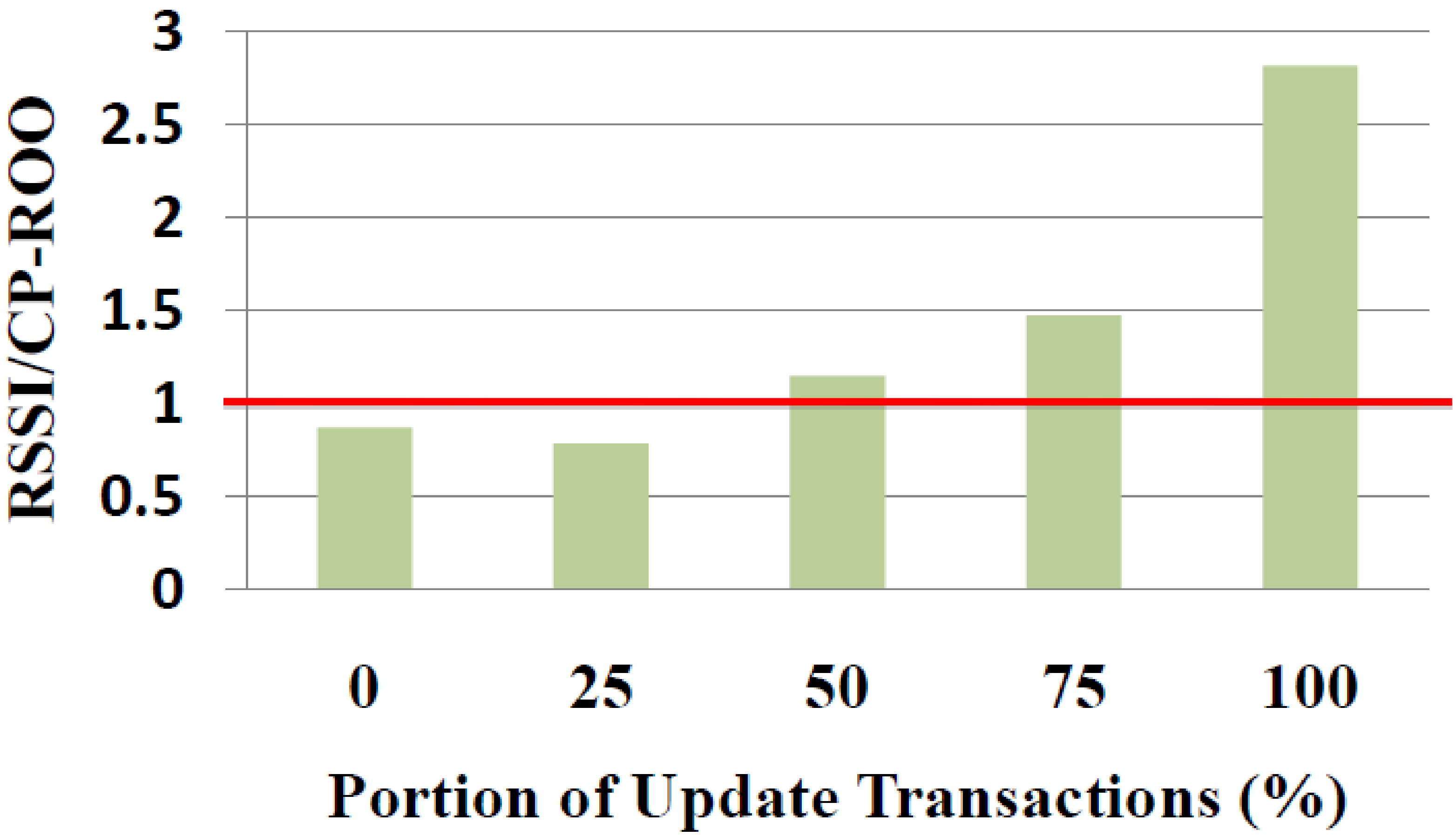
Experimental Setup

- Comparing
  - RSSI (Postgres-RSSI) : our proposal (1SR)
  - CP-ROO – conflict-management of Bornea et al. with our architecture (1SR)
  - RSI : certification algorithm of Lin et al. with our architecture
    - 1-SI, but not 1SR !!
- Synthetic micro-benchmark
  - Update transactions read from a table, update records in a different table.
  - Read-only transactions read from a table.
- TPC-C++ [Cahill et al.,TODS2009]
  - No evident difference in performance between the three algorithms (details in the paper)

Micro-benchmark, 75%Updates: Throughput (8 Replicas)



Micro-benchmark: Performance Spectrum (8 Replicas, MPL=640)



Overview

1. Review of databases, isolation levels and serializability
2. Theory to determine whether an application will have serializable executions when running at SI
3. Modifying applications
4. Fixing the DBMS
5. Replicated databases
6. Future work

Future Research Directions

- Read Committed
  - Actually, two different algorithms (one lock-based, one multiversion)
- Eventual Consistency
  - Common in Cloud data management platforms
  - Actually many quite different sets of properties [see Wada et al, CIDR'11]
- Performance Models
  - How to predict performance properties from key parameters such as transaction weight

Conclusion

- Theory: identify conditions on application program conflict patterns, for which all executions are serializable when run on a particular concurrency control mechanism
- Impact 1: Guide application developer to produce code that has these patterns
  - How to modify existing code, to produce these patterns
  - What impact on performance
- Impact 2: Propose new concurrency control mechanisms, that have similar performance to the original ones, but guarantee correctness