# Precisely Serializable Snapshot Isolation (PSSI)

ICDE '11

Stephen Revilak, Patrick O'Neil, and Elizabeth O'Neil
University of Massachusetts Boston

# Snapshot Isolation (SI)

- Introduced in 1995. Today, used in Oracle, Postgres, Microsoft SQL Server.
- Rather than locking, SI relies on multi-versioning. This gives SI two attractive properties:
  - Reads never wait for writes
  - Writes never wait for reads

# How SI Works

- Data is versioned (labeled with the writer's transaction id).
- Each transaction has two timestamps: start(Ti) and commit(Ti).
  - Ti reads the most recent versions committed prior to start(Ti) (the *snapshot*)
  - Ti's writes become visible to new transactions that begin after commit(Ti)
- First committer wins to prevent lost updates.

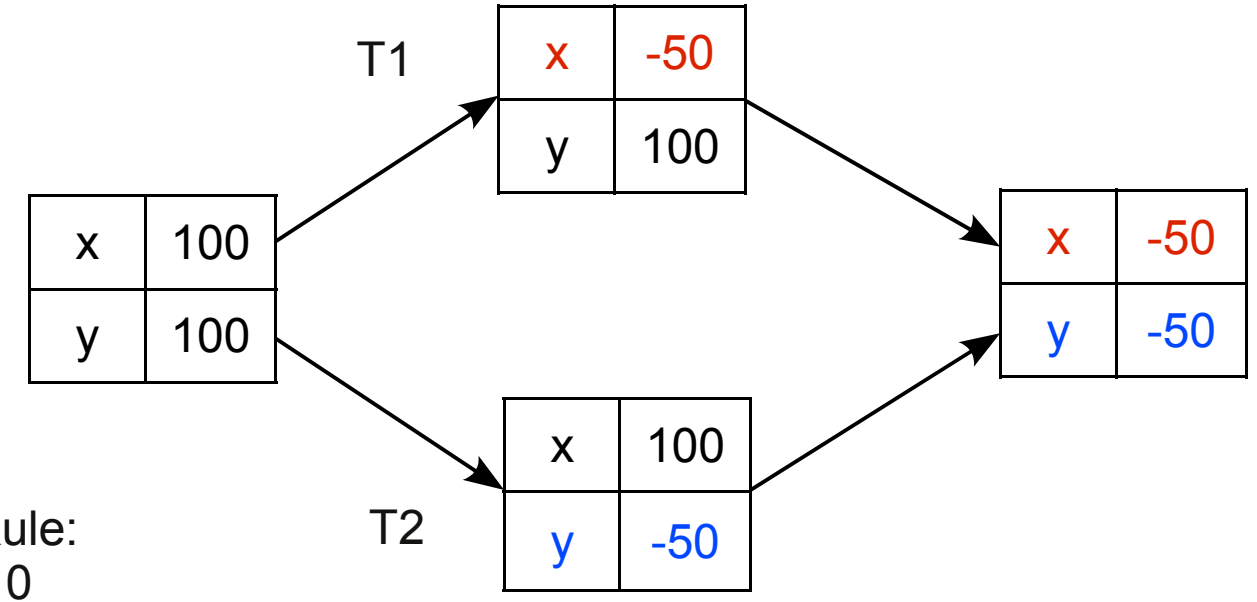
# First Updater Wins (FUW)

- Two concurrent transactions: Ti, Tj.
- Ti updates x. Update takes wi(x) lock.
- Tj tries to update x. Two outcomes:
  - If Ti already committed, then Tj aborts immediately (no lock wait).
  - If Ti is still active,Tj goes into lock wait.
    - If Ti commits, then Tj aborts.
    - If Ti aborts, then Tj takes wj(x) lock and continues.

# SI Anomalies - Write Skew

## Example of an SI Anomaly:

H1: r1(x0, 100),r1(y0, 100), r2(x0, 100), r2(y0, 100), w1(x1, -50), c1, w2(y2, -50), c2




Business Rule:
(x + y) ≥ 0

# Dependency Theory

- Write-write dependency: Ti-ww→Tj
  - Ti writes x, Tj writes successor version of x.
- Write-read dependency: Ti-wr→Tj
  - Ti writes x.  Tj reads what Ti wrote.
  - Data item or predicate read, conflicting with data item write.
- Read-write anti-dependency: Ti-rw→Tj
  - Ti reads x, Tj writes the successor version of x.
  - Ti, Tj may or may not be concurrent.

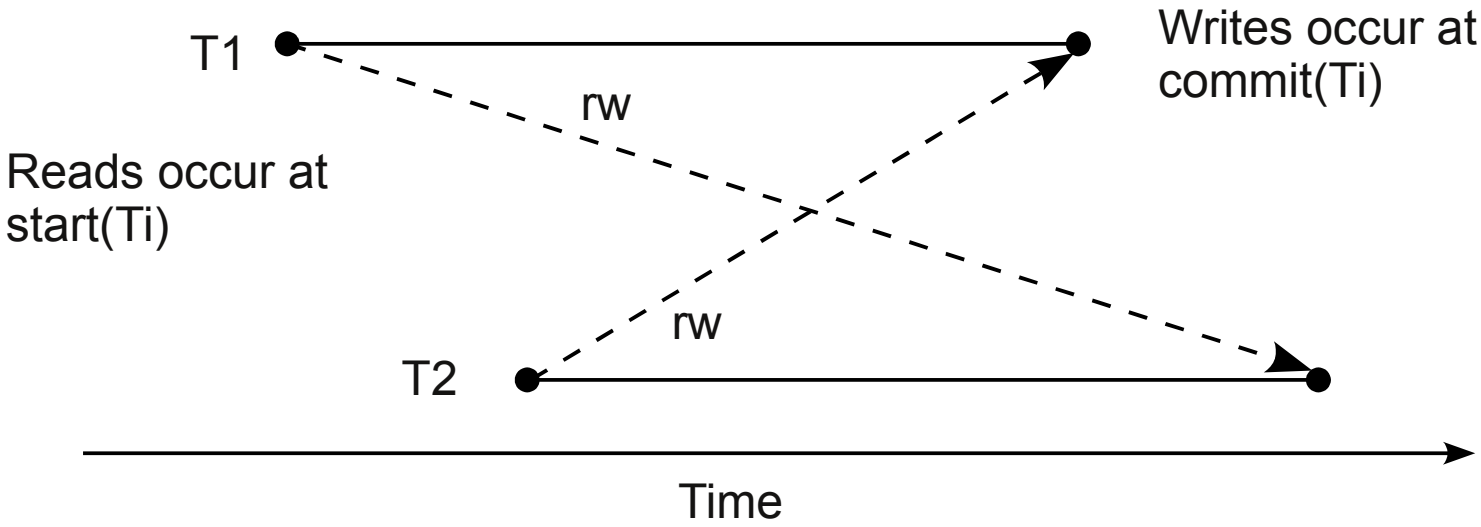
# Dependency Serialization Graph

- We can model dependencies with a directed graph, called a Dependency Serialization Graph (DSG).
  - Each node is a committed transaction. Each edge is a dependency
- DSG(H) is acyclic IFF history H is serializable.
- We can achieve serializability by preventing dependency cycles.

# SI-RW Diagrams

H1: r1(x0, 100),r1(y0, 100), r2(x0, 100), r2(y0, 100), w1(x1, -50), c1, w2(y2, -50), c2

Below is a SI-RW diagram for H1.




Note: cycle T1-rw→T2-rw→T1 didn't happen until *after* commit(T1).

# PSSI's Strategy

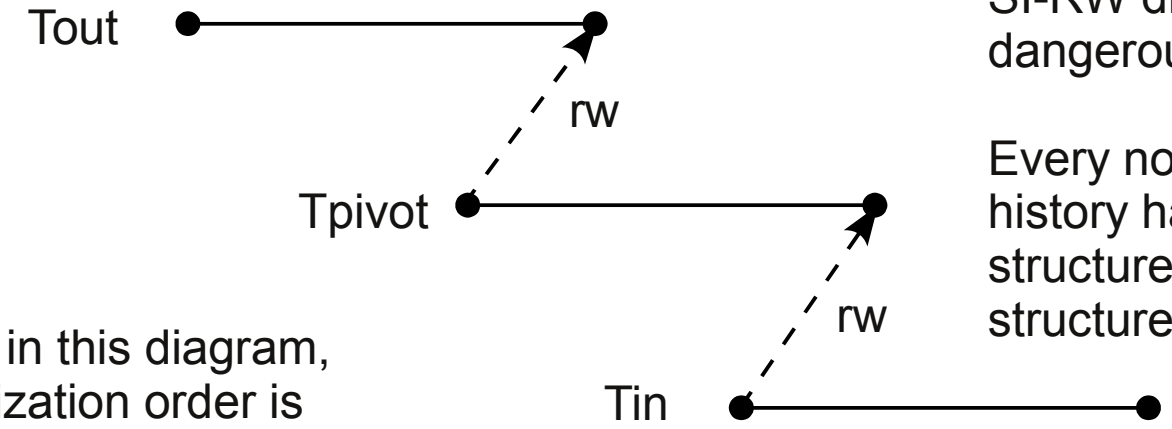
Ensure serializability by preventing cycles.

- Use a lock table to find dependencies.
- Use a cycle testing graph to find cycles.
- Use a variant of ARIES/IM to prevent phantom anomalies

We Implemented PSSI with MySQL 5.1.31's InnoDB Storage Engine.

# ESSI - An Alternate Approach

- Rather than preventing cycles, ESSI prevents *essential dangerous structures*.
- ESSI guarantees Serializability, but is more conservative than PSSI.




SI-RW diagram of a dangerous structure.

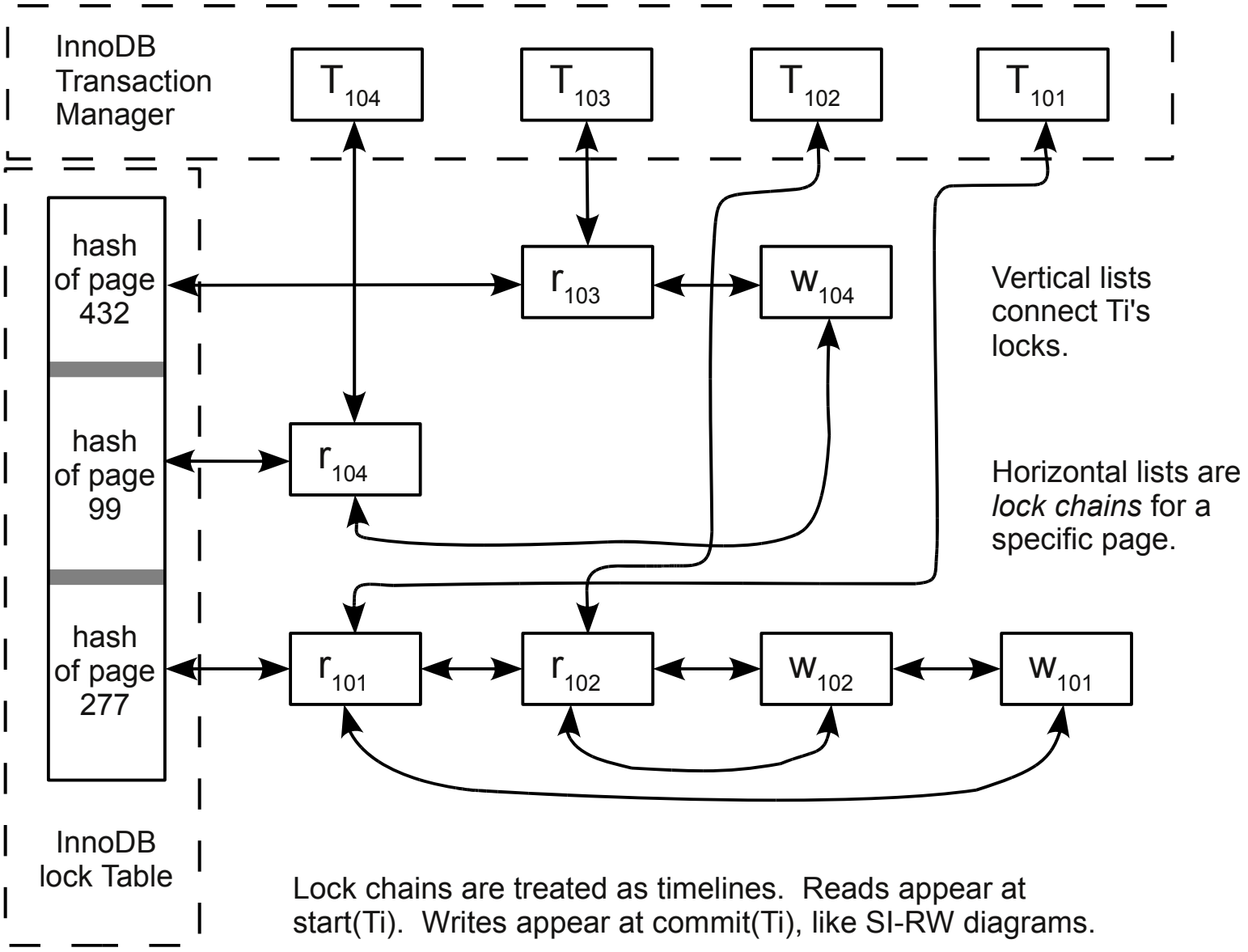
Every non-serializable SI history has a dangerous structure; but dangerous structure ≠ non-SR.

Note: in this diagram, serialization order is opposite commit order.

# PSSI Lock Table (1)

- Purpose: to detect conflicts, and find dependencies between transactions.
  - Keeps track of all reads and writes
- Looks like a traditional lock manager (but behaves differently).
- Requires separate notions of *lock conflict* and *lock wait*.
- Assumes that all dependencies can be found via conflicts on data items.

# PSSI Lock Table (2)




Lock chains are treated as timelines. Reads appear at start(Ti). Writes appear at commit(Ti), like SI-RW diagrams.

# Cycle Testing Graph: CTG

- Each node is a transaction Ti.
- Each edge is a Ti→Tj dependency.
- CTG is a *suffix* of a complete history, containing only:
  - Committed transactions with *potential* to become part of a cycle ("zombies").
  - The currently committing transaction Tk.
- Tk commits if CTG ∪ Tk is acyclic.

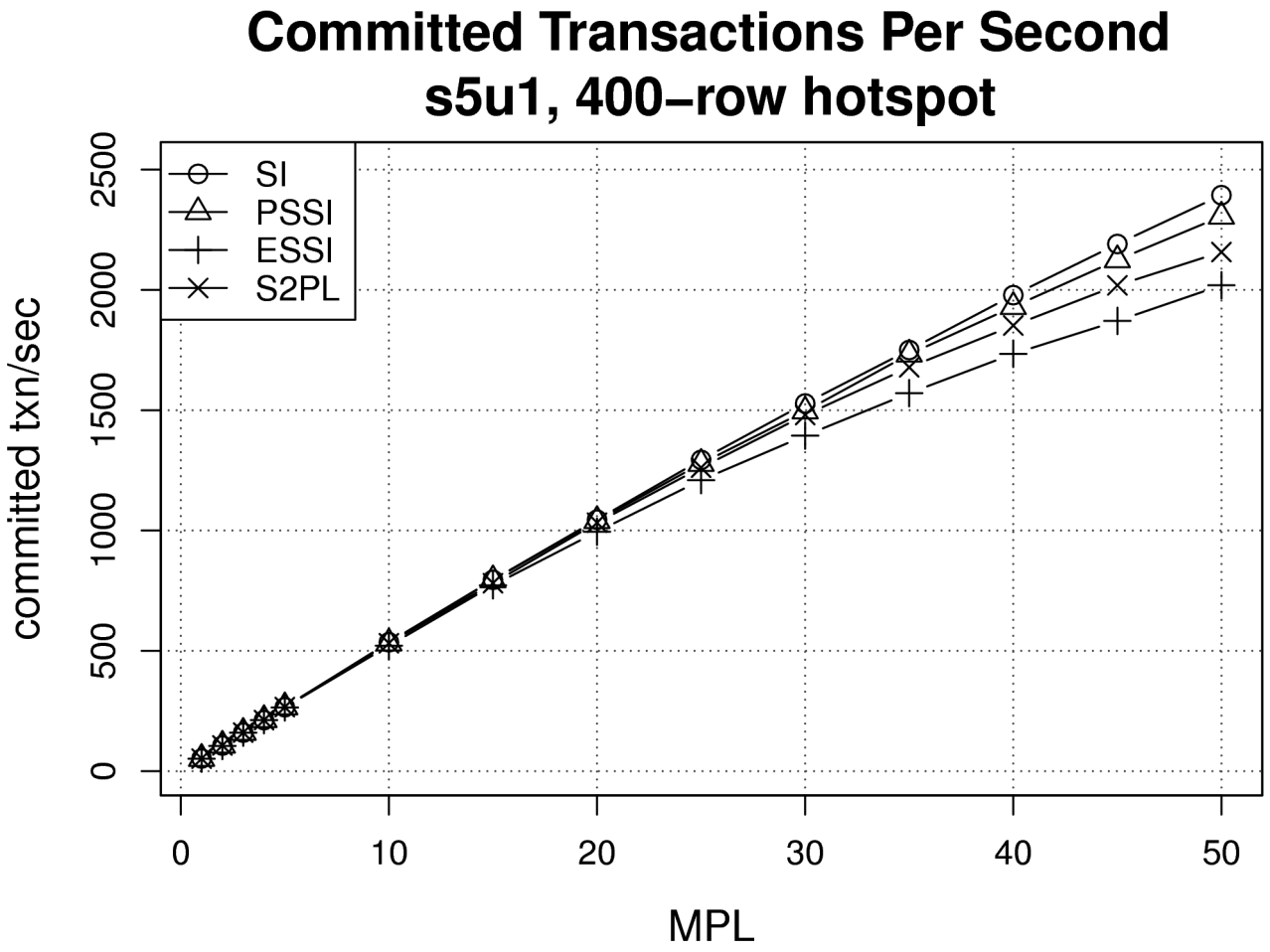
# CTG Pruning

- If Ti can't become part of a (future) cycle, then we'd like to *prune* Ti from the CTG.
- We can prune Ti if:
  - Ti has no in edges, and
  - Ti committed before the oldest active transaction started.
- Pruning happens each time a transaction commits or aborts.

# SICycles: a Parametrized Workload

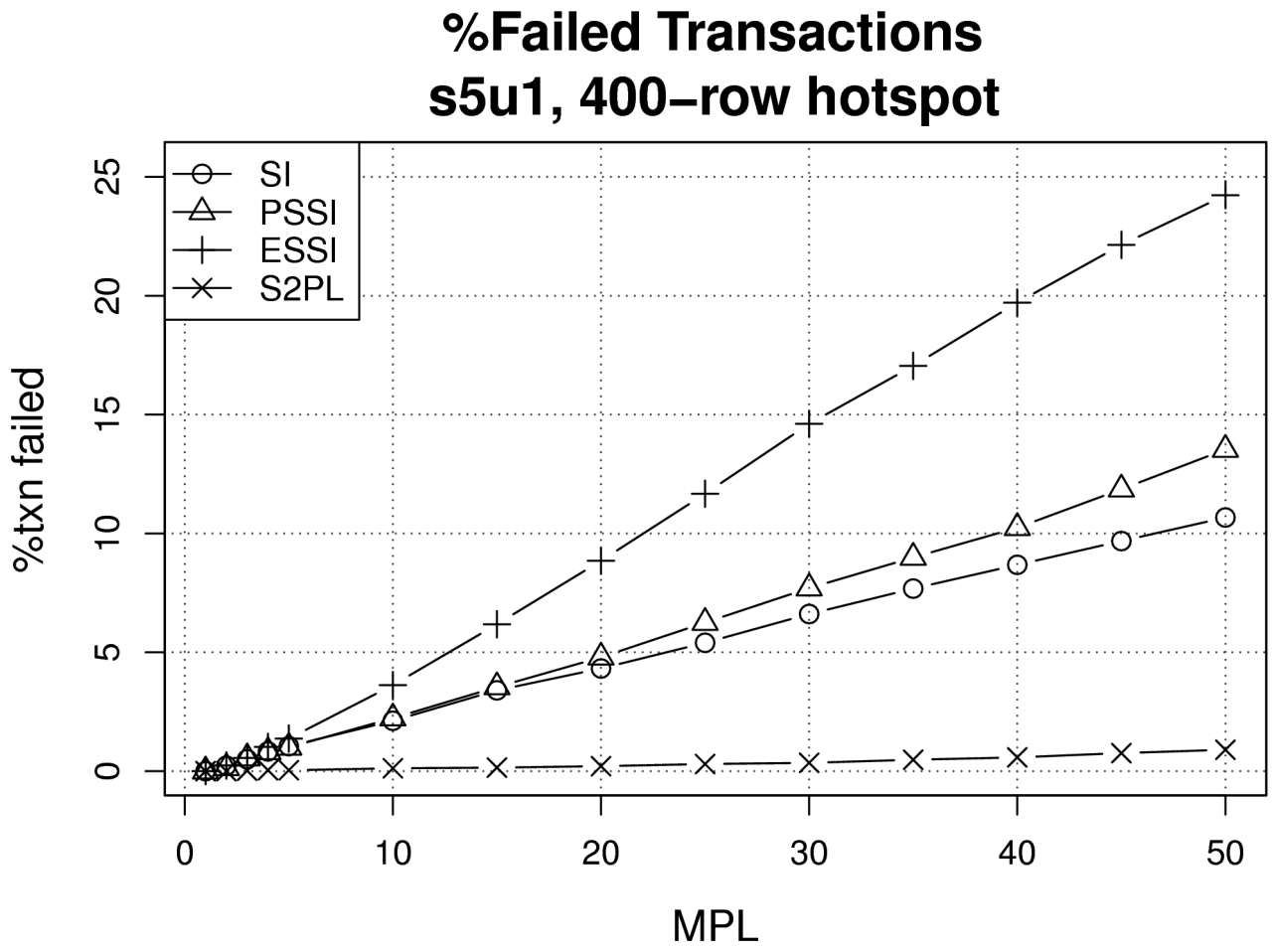
- 1MM rows, 100-bytes per row.
  - kseq, krandseq, kval
- sKuN. Each transaction runs:
  - K select statements (by krandseq),
  - N update statements (by krandseq).
  - 3ms +/- 50% delay between statements.
- Vary K, N, MPL, size of hotspot.

# s5u1 CTPS



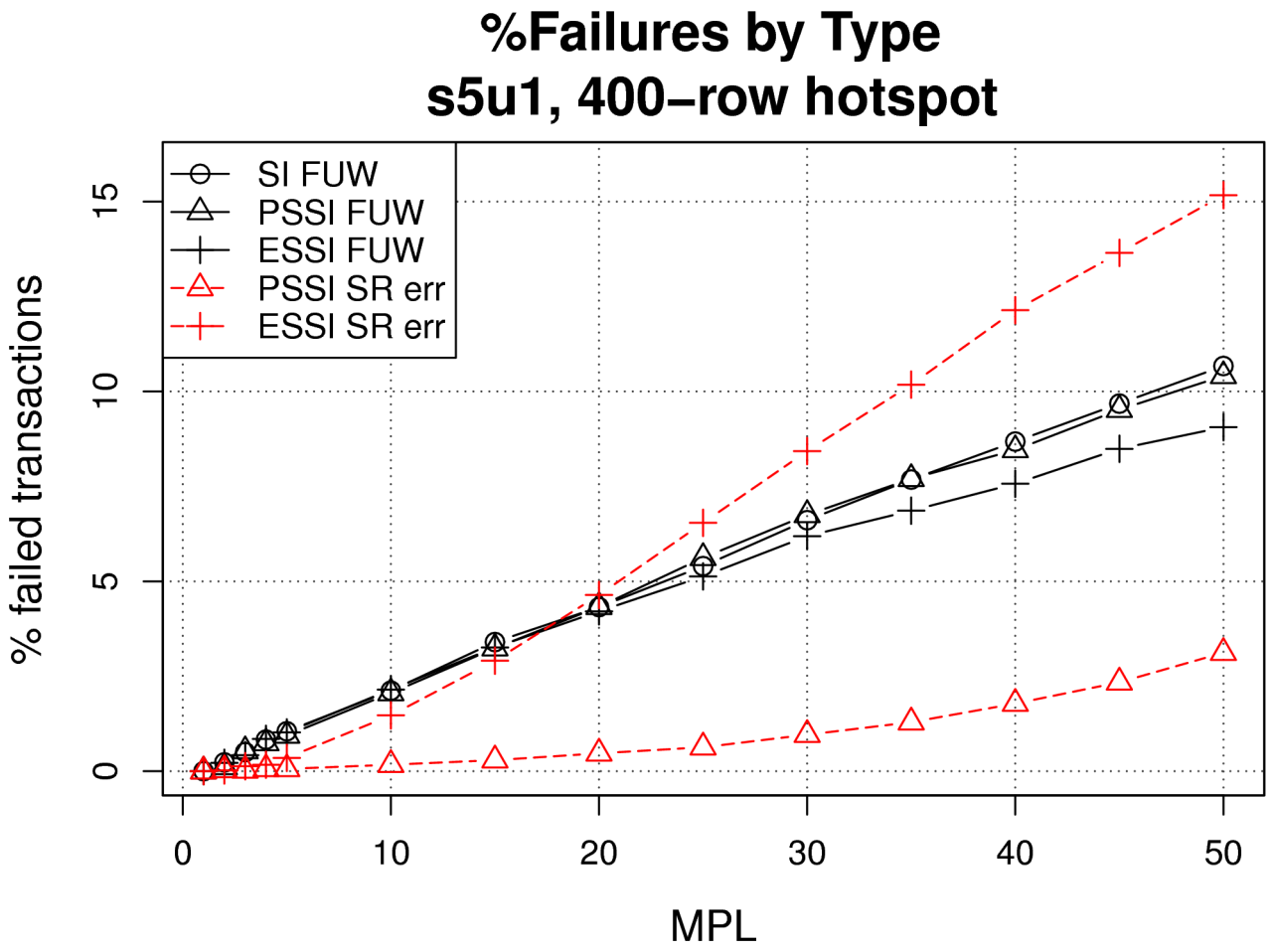

SI performs best, but it's making mistakes.
The other three isolation levels are serializable.

# s5u1 Abort Rates



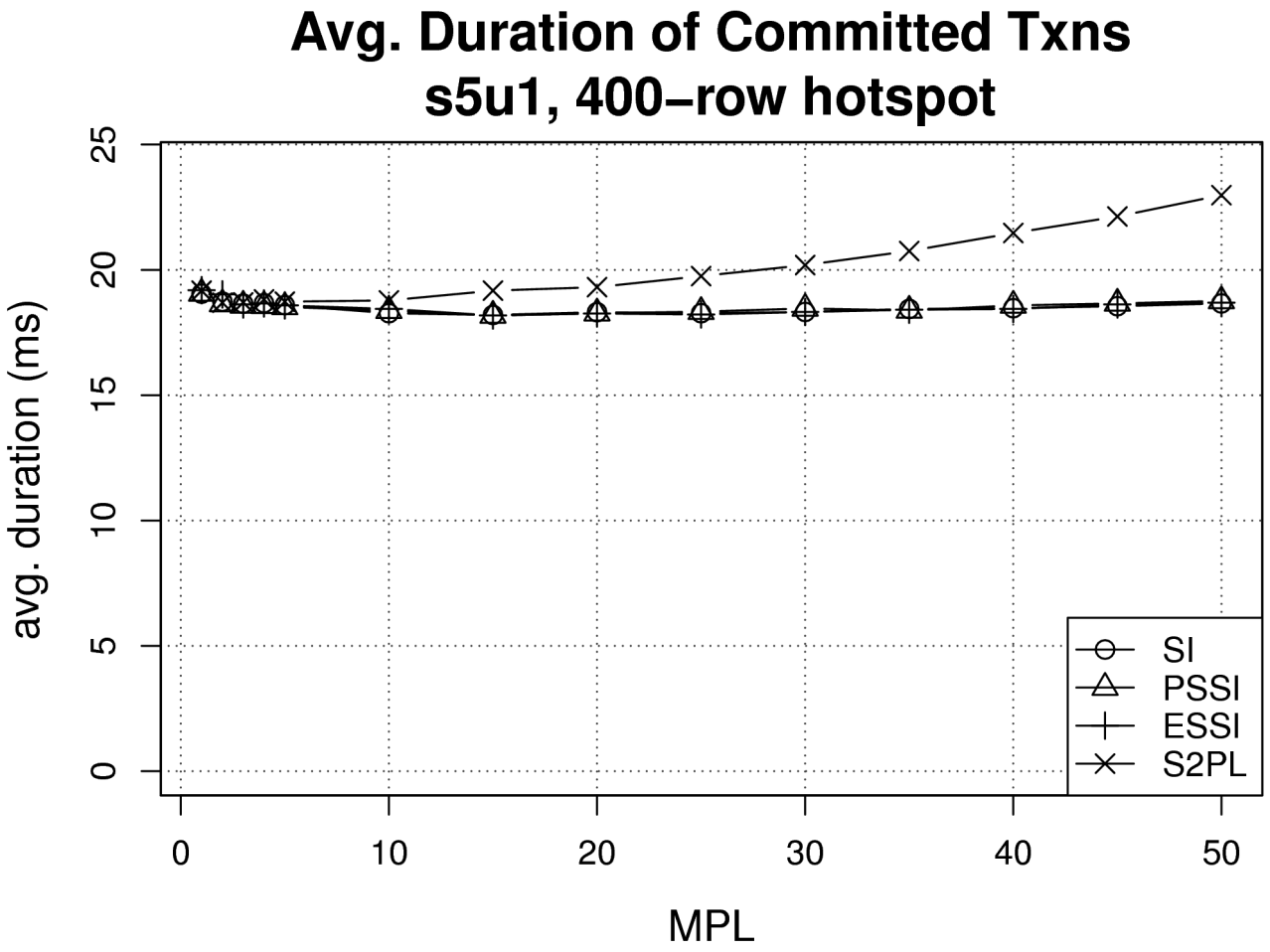

SI, PSSI, ESSI throughput is primarily determined by error rate.
S2PL proceeds with caution.

# s5u1 Abort Detail




Black=FUW errors. Red=Serialization Errors.

# s5u1 Durations




S2PL duration increases due to lock waits.
SI, PSSI, ESSI duration increases slightly (only ww conflicts wait)

# Summary

- Snapshot Isolation: good throughput, but doesn't provide serializability.
- PSSI: makes SI Serializable, with a minimum of false-positive aborts.
  - Lock Table to find dependencies.
  - Cycle Testing Graph to find cycles.
  - Pruning to keep the CTG small.

Thank You

# Extra slides

# Challenges in Making SI Serializable

- Ti might not become part of a cycle until after commit(Ti)
  - DBMS must retain information about transactions after they've committed
- In general, commit order does not match serialization order

  (We'll see examples of this shortly)

# InnoDB and Durability

- Controlled via `innodb_flush_log_at_trx_commit`.
- *flush=2*. Write logs at commit, flush logs asynchronously 1x/second.
  - High performance. Not very durable.
- *flush=1*. Write and flush logs at commit (group commit)
  - Lower performance. *Almost* durable.

# InnoDB and Group Commit

- Our paper states: "InnoDB does not offer group commit". MySQL 5.1.31's InnoDB offers group commit, but it's broken in some configurations.
- cf. MySQL bug #13669: *Group commit is broken in 5.0*.

  (Fixed in the current InnoDB, but not the version we used for development)

# Testing Environment

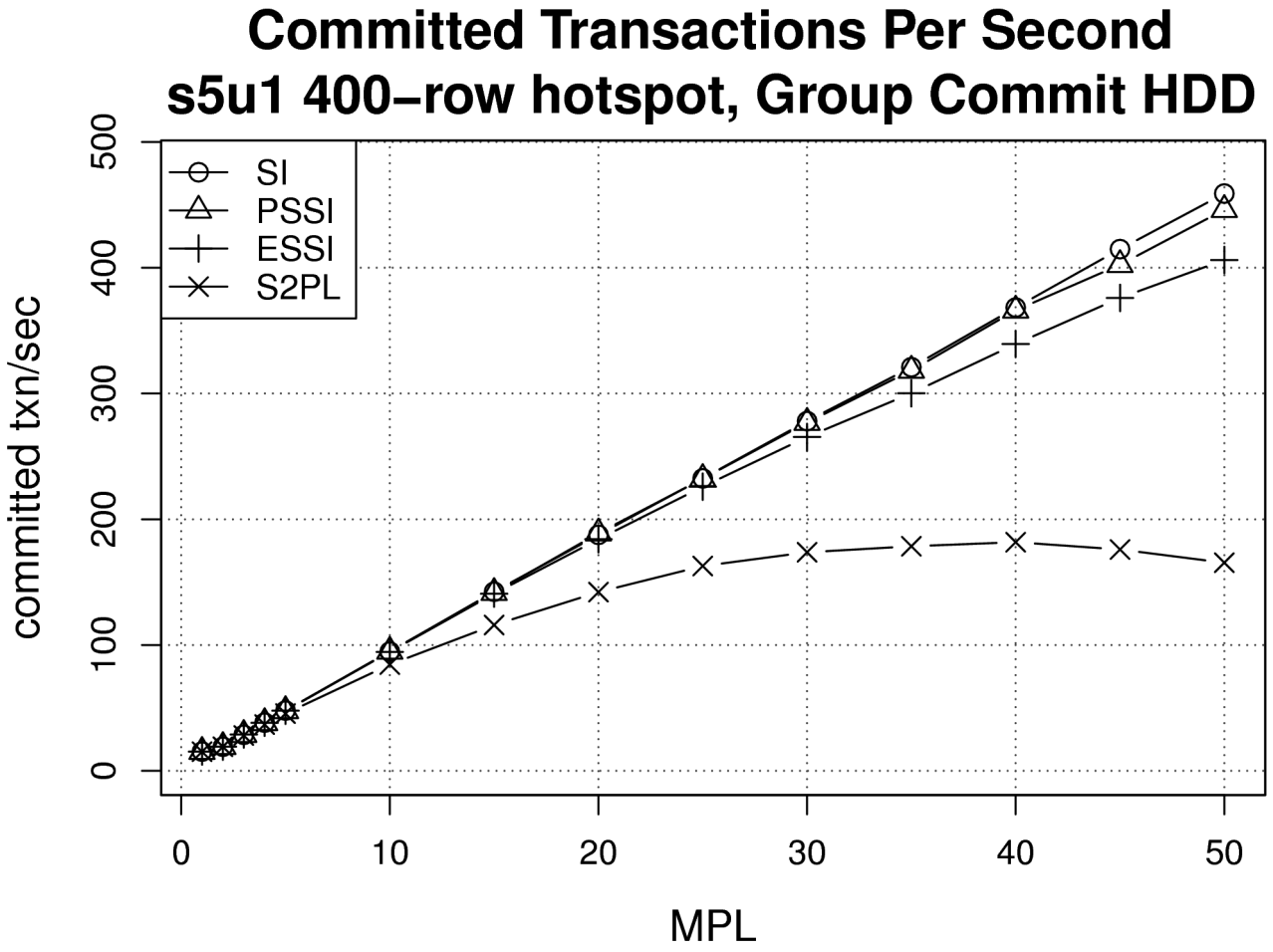
Two Linux machines

- Quad Core server.  2.67 GHz, 4GB ram, 64-bit OS.
- Dual Core client.  2.53 GHz, 2GB ram, 32-bit OS.
- Java benchmarking program.  Plain JDBC + MySQL Connector/J

# PSSI Commit Sequence

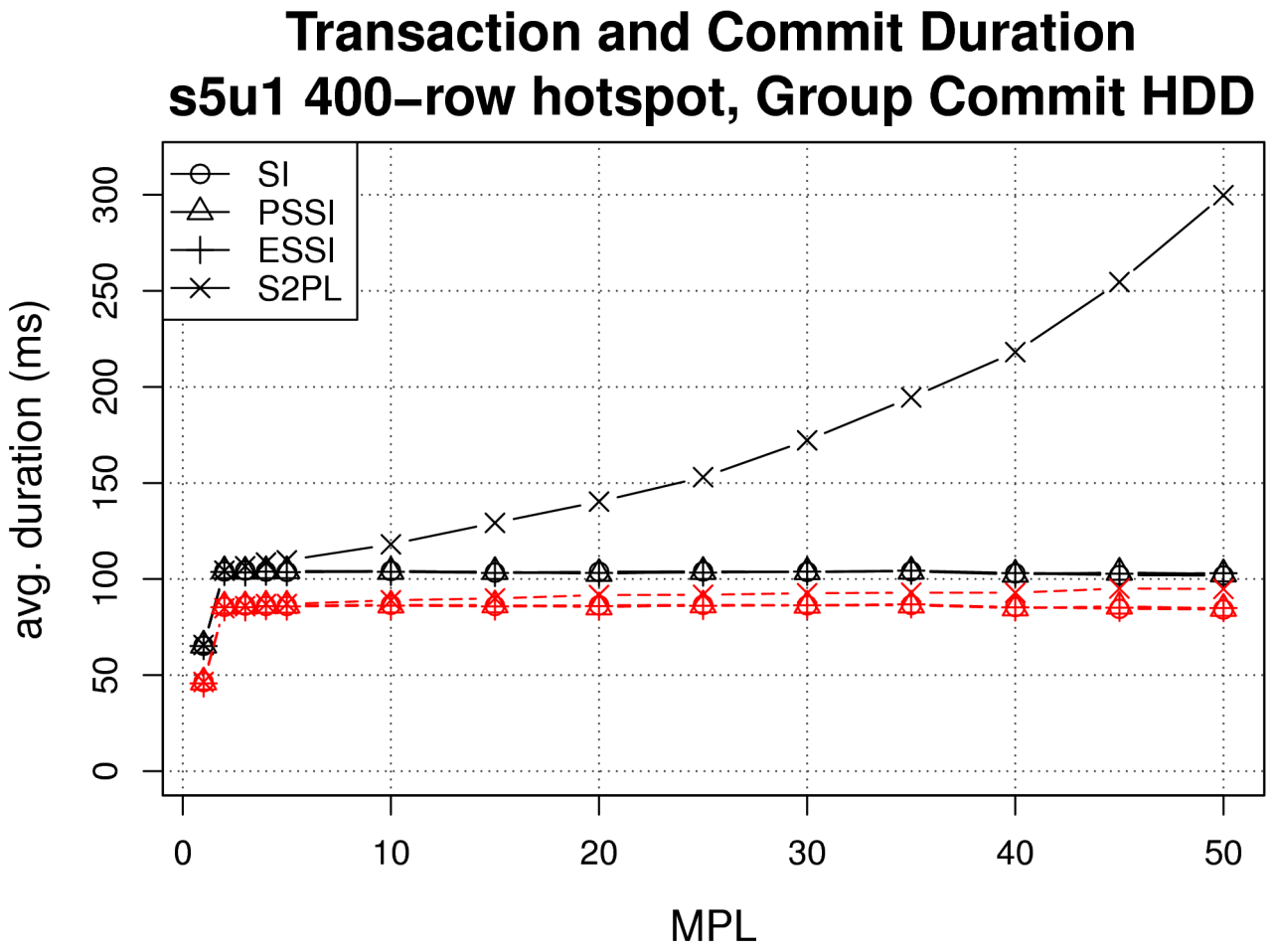
- Find dependencies that commit(Ti) would cause.
  - Add Ti + dependencies to CTG.
- Perform a cycle test.
  - If cycle, abort. Otherwise, commit continues.
- Rest of commit (write logs, release write locks, handle FUW, etc).
- Prune the CTG.

# Group Commit HDD CTPS




Nearly 80% lower throughput (!) than flushing 1x/sec.
Note: InnoDB modified so S2PL releases locks *after* flush.

# Group Commit HDD Durations

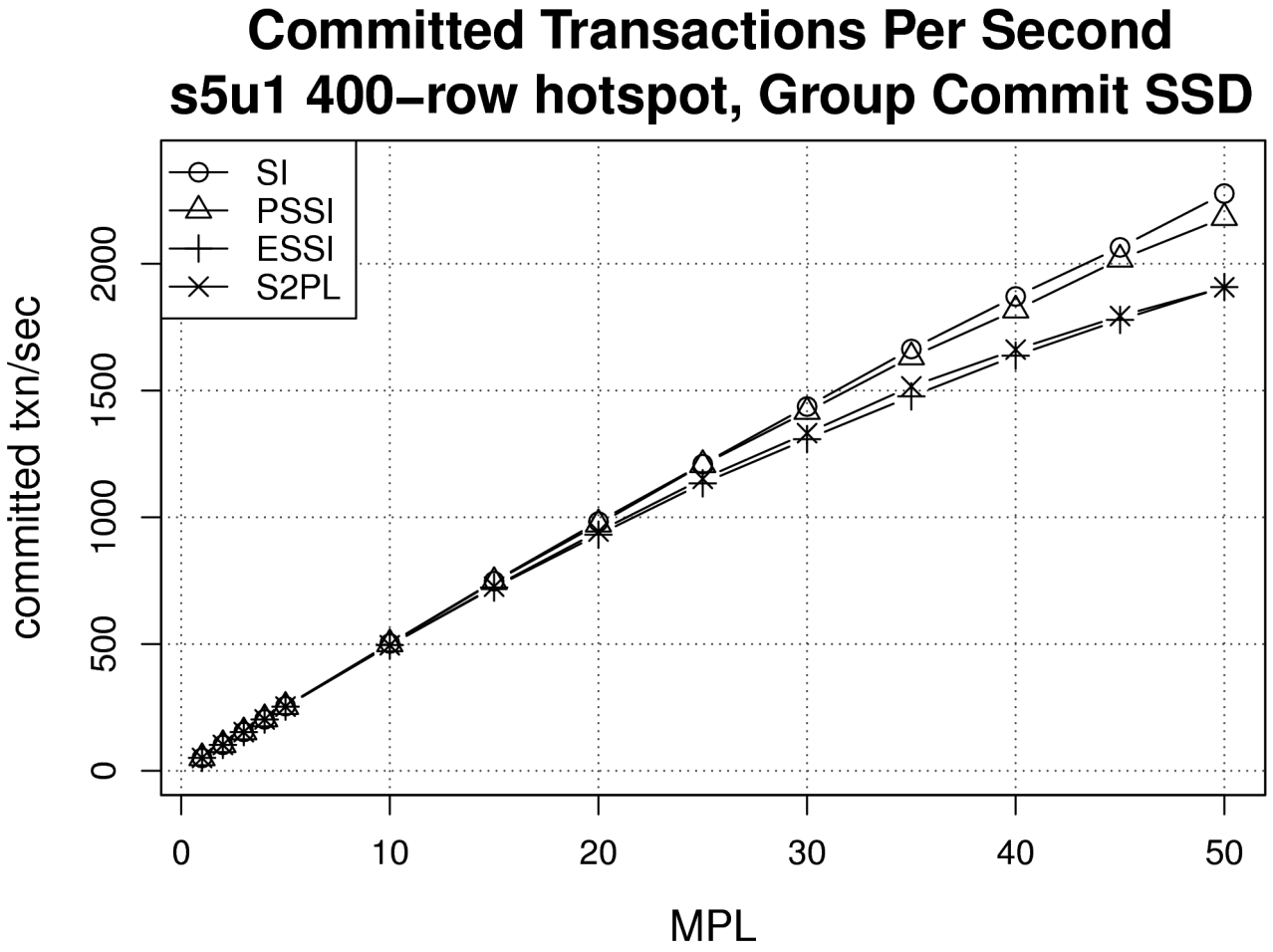



Black=tx duration

Red=commit duration

SI, PSSI, ESSI duration now > 100ms; much longer than flush=2's 18ms

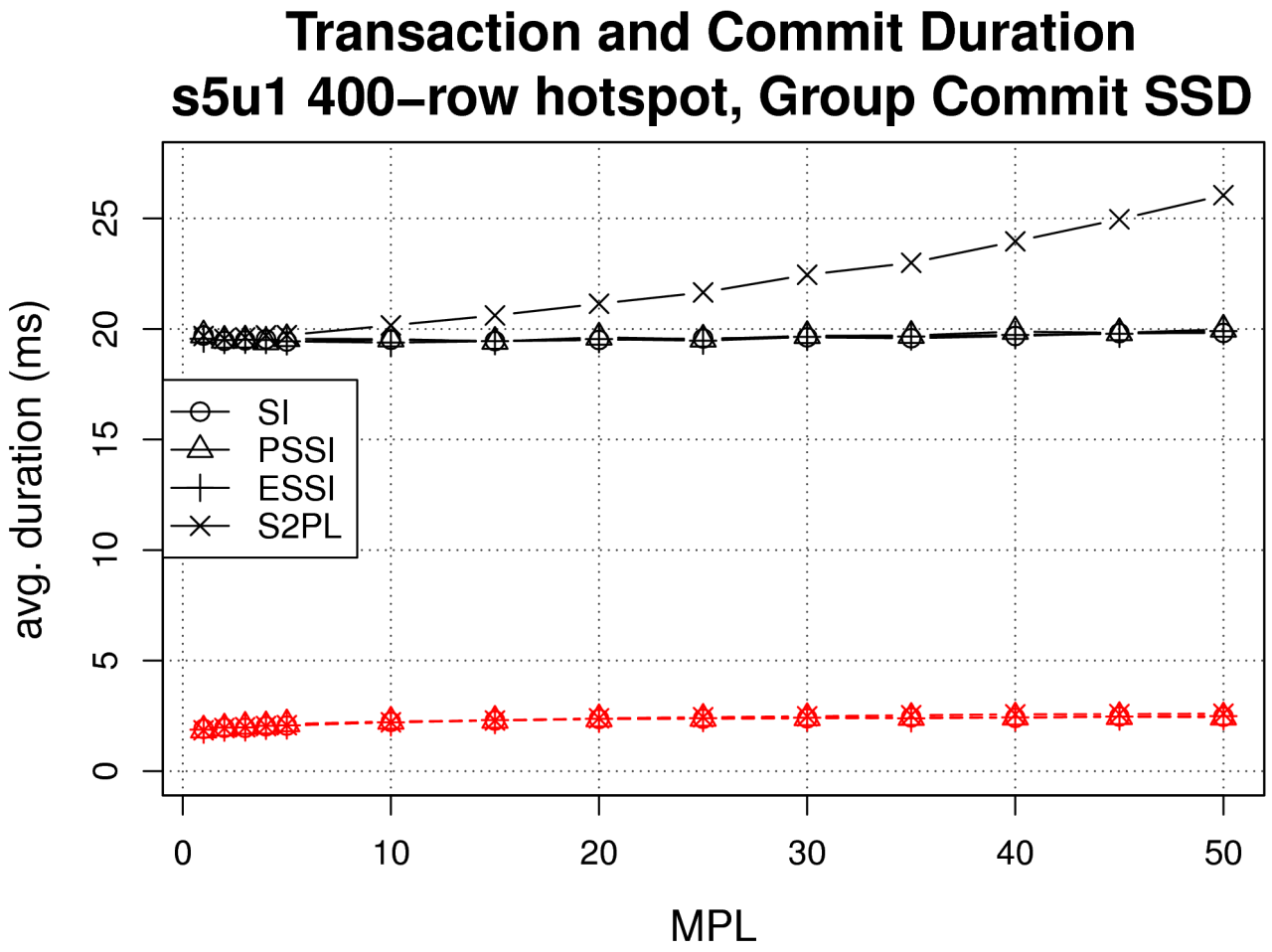
# Group Commit SSD CTPS






With SSD, throughput almost as good as flushing 1x/sec

# Group Commit SSD Durations

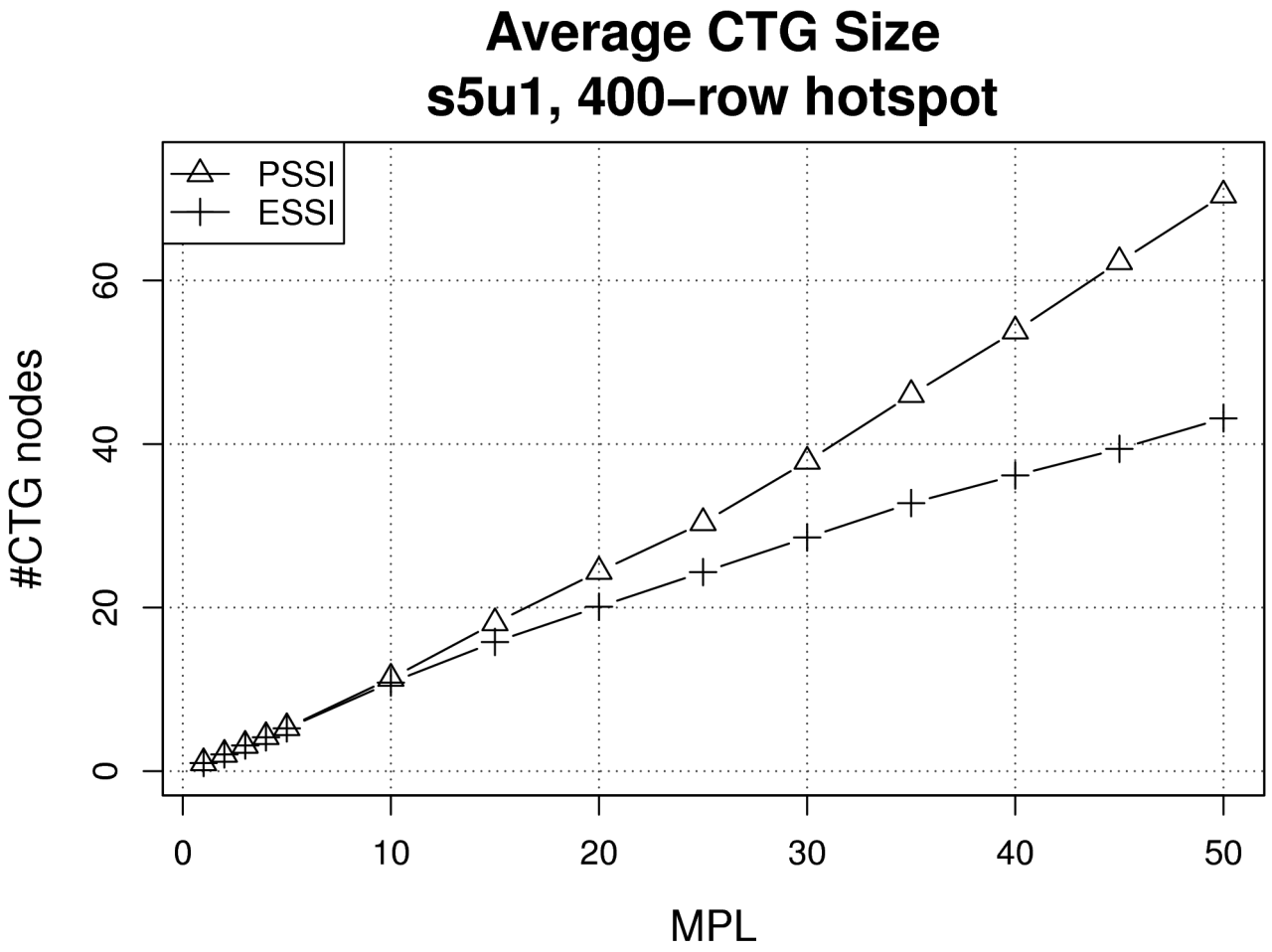



Black=tx duration

Red=commit duration

Transaction duration ~19ms, close to original ~18ms

# Avg. CTG Size


Average CTG Size
s5u1, 400–row hotspot
PSSI
ESSI
#CTG nodes
0
20
40
60
0
10
20
30
40
50
MPL

# Cycle Length Distribution