Analyzing the Impact of System Architecture on the Scalability of OLTP Engines for High–Contention Workloads *by R. Appuswamy, A. Anadiotis, D. Porobic, M. Iman, A. Ailamaki*

Max Gilbert

m_gilbert13@cs.uni-kl.de

Lehrgebiet Informationssysteme

Technische Universität Kaiserslautern

July 16, 2018

Table of Contents

1. Introduction
2. DB Architectures
   - Shared Everything/Non–Partitioned
   - Data–Oriented Transaction Execution
   - Delegation
   - Partitioned Serial Execution
   - Summary
3. CC Algorithms
   - DL_DETECT
   - NO_WAIT
   - 2V_NO_WAIT
   - SILO
4. Performance Evaluation
   - Read–Only Workload
   - Update–Only Workload
   - Read–Only YCSB Workload
   - Update–Only YCSB Workload
   - Mixed YCSB Workload
   - Conclusion
   - Discussion

Section 1

Introduction

Requirements for a DBMS

- Reliability
  - ACID Transactions
  - high availability
  - etc.
- Functionality
  - simple to use programming model
  - simple to use API
  - etc.

Performance isn't everything, but without it, everything else is nothing.

- Performance
  - high transaction throughput
  - low latency
  - etc.

Some Implications of those Requirements

- work purely in–memory when the working set completely fits in main memory
- proper utilization of the computational resources is required
  - available CPU time (usually not the bottleneck)
  - available hardware contexts (simultaneous threads)
  - Cache Oblivious Algorithms (e.g. partitioning Hash–JOINs)
  - → Interleaved transaction execution to exploit abundant thread–level parallelism without violating the ACID properties!
  - → Interleaved operation execution to exploit intra–transaction parallelism!

→ physical & logical Synchronization

Some Implications of those Requirements

- work purely in–memory when the working set completely fits in main memory
- proper utilization of the computational resources is required
  - available CPU time (usually not the bottleneck)
  - available hardware contexts (simultaneous threads)
  - Cache Oblivious Algorithms (e.g. partitioning Hash–JOINs)

  → Interleaved transaction execution to exploit abundant thread–level parallelism without violating the ACID properties!

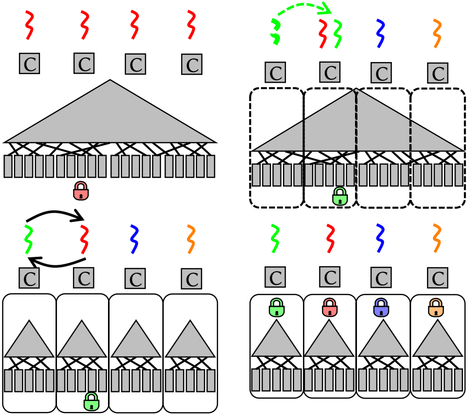
  → Interleaved operation execution to exploit intra–transaction parallelism!

→ physical & logical Synchronization

→ **Limits concurrency for high–contention workloads!**

Section 2

Database Architectures



Shared Every-thing/ Non–Partitioned

Data–Oriented Trans-action Execution (DORA)

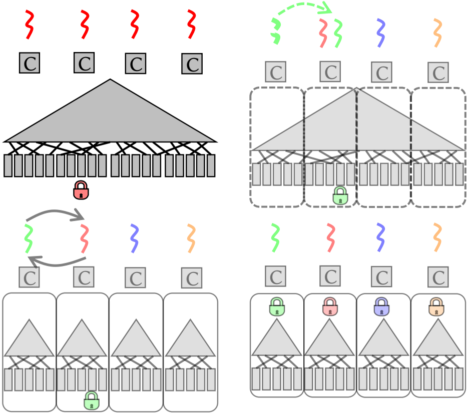
Delegation

Partitioned Serial Execution (PSE)

Subsection 1

Shared Everything/Non–Partitioned (SE/NP)

Shared Everything/Non–Partitioned



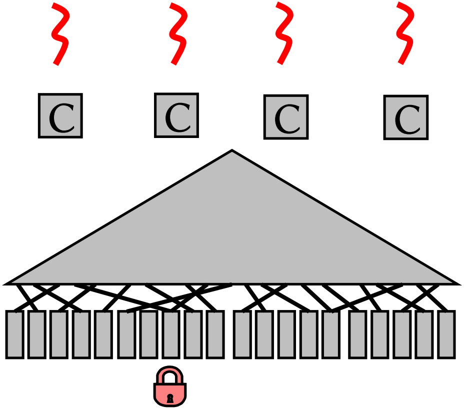
Data–Oriented Transaction Execution (DORA)

Delegation

Partitioned Serial Execution (PSE)

Subsection 1

Shared Everything/Non–Partitioned (SE/NP)



Properties of SE/NP

- metadata (incl. locks) are not partitioned

→ physical synchronization (latches, atomics) required

- data and indices are not partitioned

→ logical synchronization using a concurrency control protocol also required

- transactions completely executed by one thread
- thread–assignment depends only on load

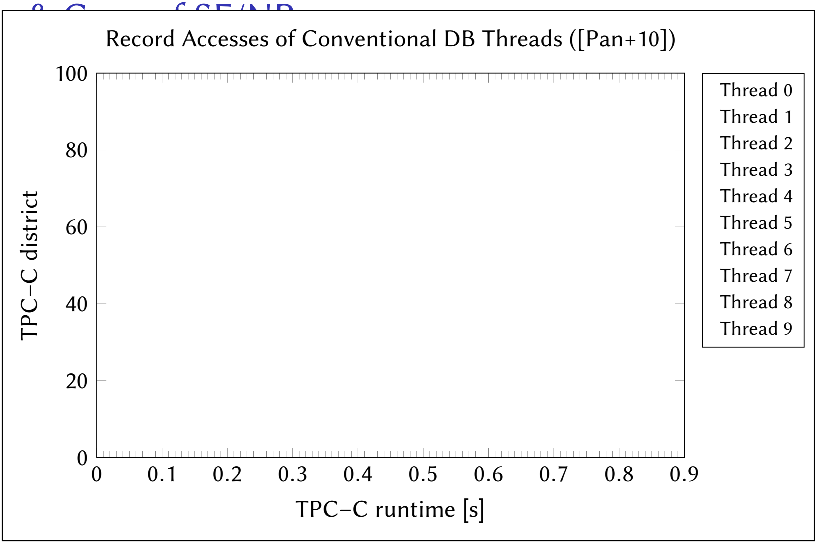
Pros & Cons of SE/NP

- + no partitioning required (e.g. manual selection of a strategy)
- + partitioning would be sensitive to the workload
- + changed workloads would require repartitioning to benefit from partitioning

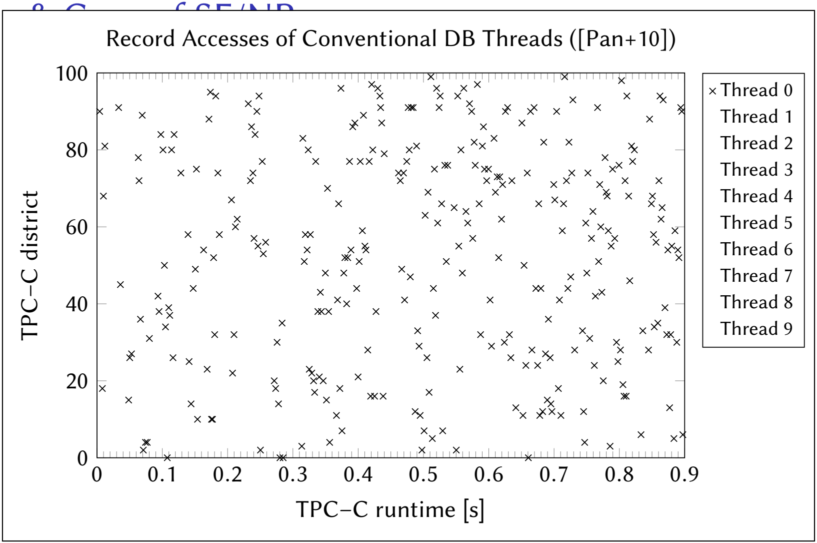
Pros & Cons of SE/NP

- + no partitioning required (e.g. manual selection of a strategy)
- + partitioning would be sensitive to the workload
- + changed workloads would require repartitioning to benefit from partitioning
- – each thread might access every record at arbitrary times
  - – each CPU cache may contain any part of the data
    → cache pollution
  - – each CPU may access any part of the data
    → data movement between NUMA regions
  - – each CPU may acquire any latch
    → data movement between NUMA regions
  - – each CPU may atomically write to any semaphore
    → hardware cache coherence overhead

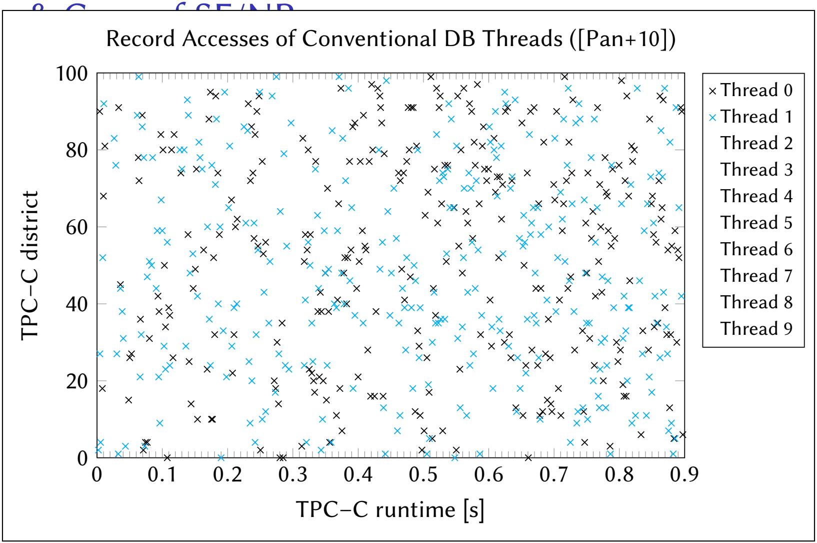
Pros & Cons of SE/NP



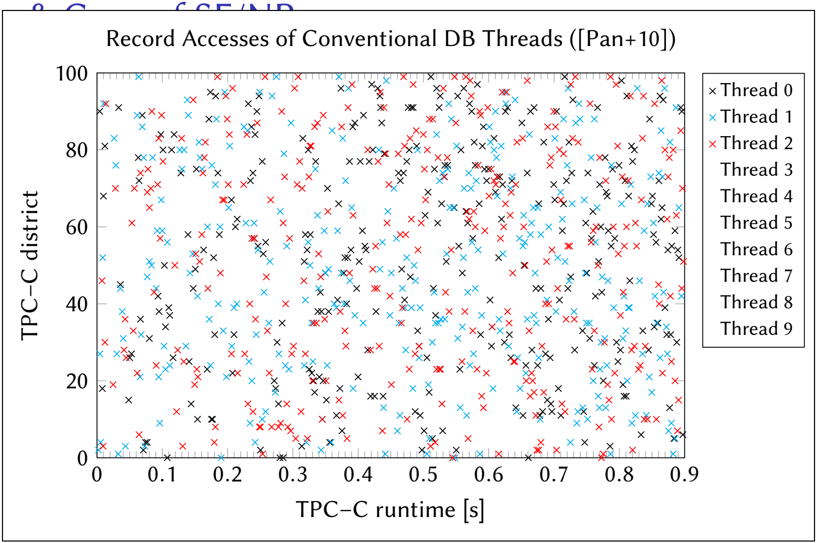
Pro


Record Accesses of Conventional DB Threads ([Pan+10])
TPC–C district
100
80
60
40
20
0
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
TPC–C runtime [s]
× Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 8
Thread 9

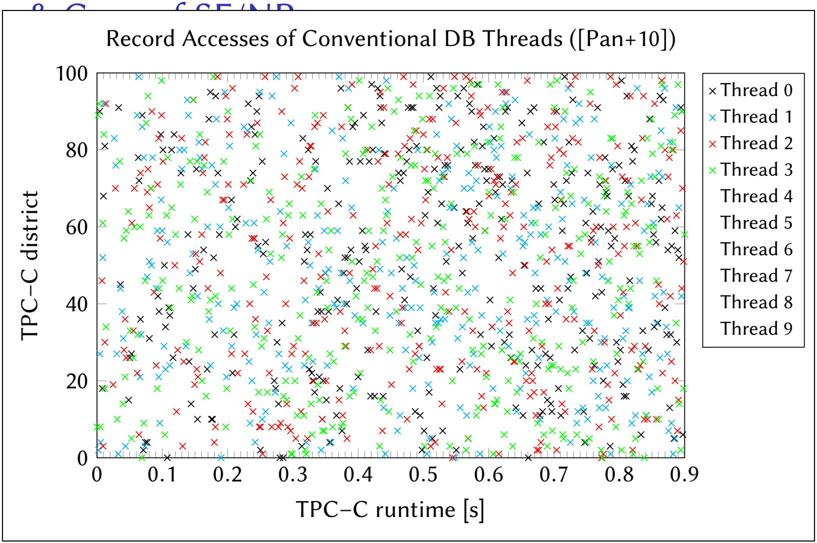
Pro


Record Accesses of Conventional DB Threads ([Pan+10])
TPC–C district
100
80
60
40
20
0
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
TPC–C runtime [s]
× Thread 0
× Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 8
Thread 9

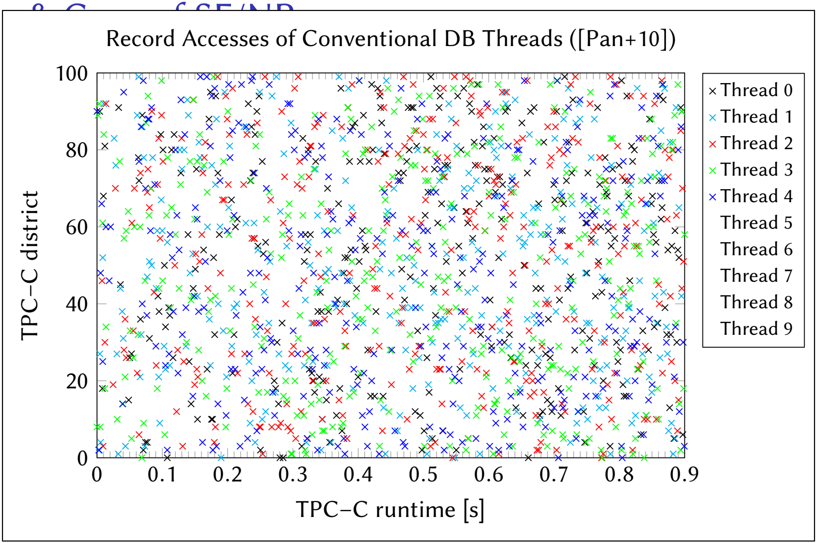
Pro


Record Accesses of Conventional DB Threads ([Pan+10])
TPC–C district
TPC–C runtime [s]
0
20
40
60
80
100
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
× Thread 0
× Thread 1
× Thread 2
Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 8
Thread 9

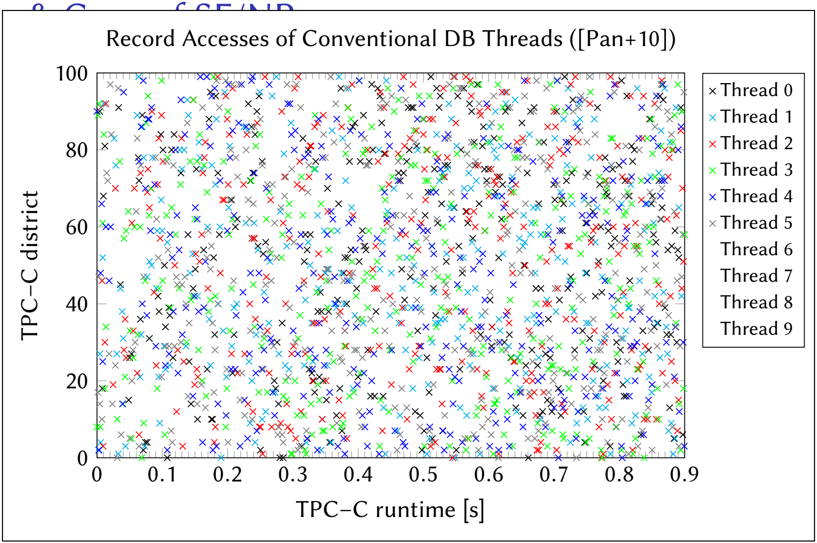
Pro


Record Accesses of Conventional DB Threads ([Pan+10])
100
80
60
40
20
0
TPC–C district
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
TPC–C runtime [s]
× Thread 0
× Thread 1
× Thread 2
× Thread 3
Thread 4
Thread 5
Thread 6
Thread 7
Thread 8
Thread 9

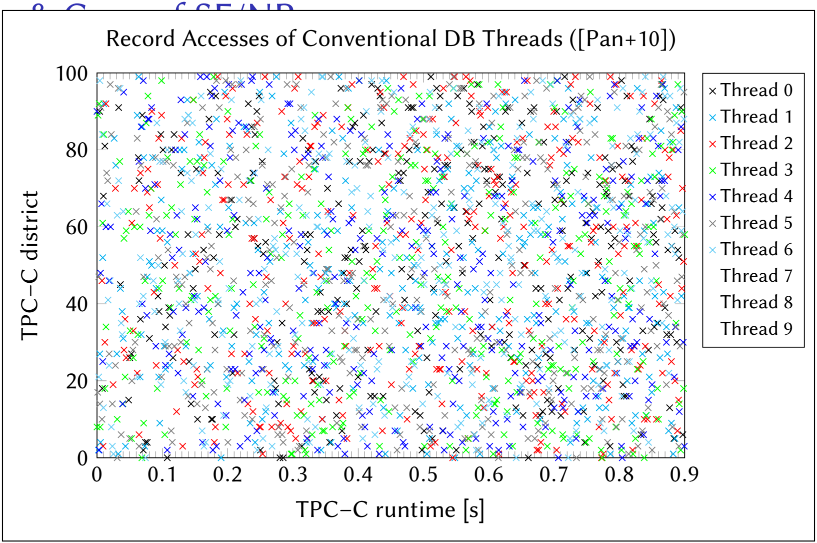
Pro


Record Accesses of Conventional DB Threads ([Pan+10])
TPC–C district
100
80
60
40
20
0
TPC–C runtime [s]
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
× Thread 0
× Thread 1
× Thread 2
× Thread 3
× Thread 4
Thread 5
Thread 6
Thread 7
Thread 8
Thread 9

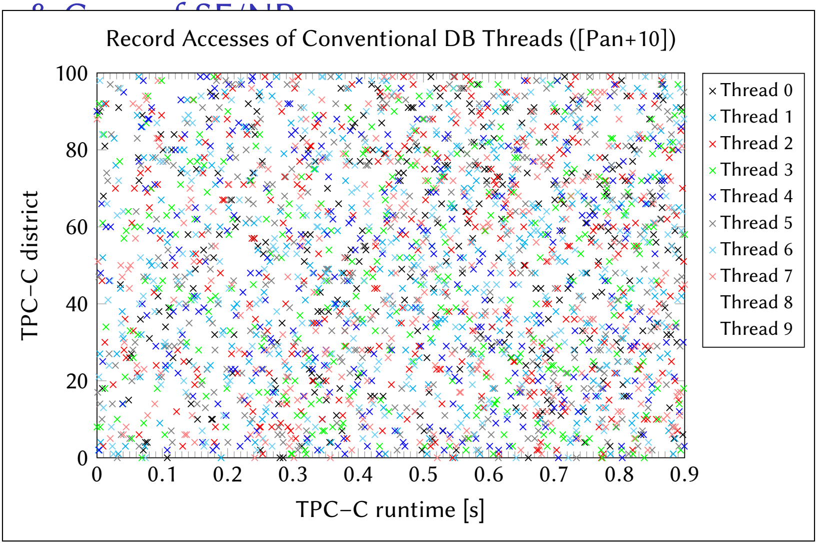
Pro


Record Accesses of Conventional DB Threads ([Pan+10])
TPC-C district
TPC-C runtime [s]
0
20
40
60
80
100
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
× Thread 0
× Thread 1
× Thread 2
× Thread 3
× Thread 4
× Thread 5
Thread 6
Thread 7
Thread 8
Thread 9

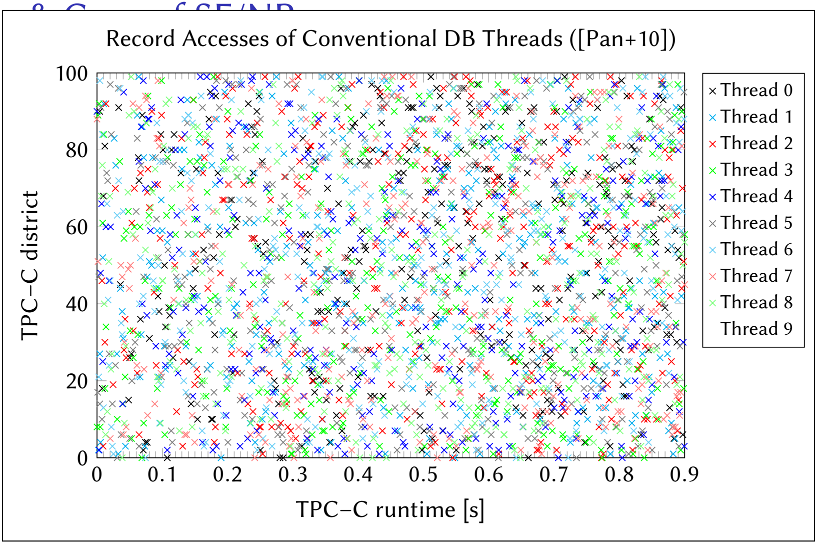
Pro


Record Accesses of Conventional DB Threads ([Pan+10])
TPC–C district
100
80
60
40
20
0
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
TPC–C runtime [s]
× Thread 0
× Thread 1
× Thread 2
× Thread 3
× Thread 4
× Thread 5
× Thread 6
Thread 7
Thread 8
Thread 9

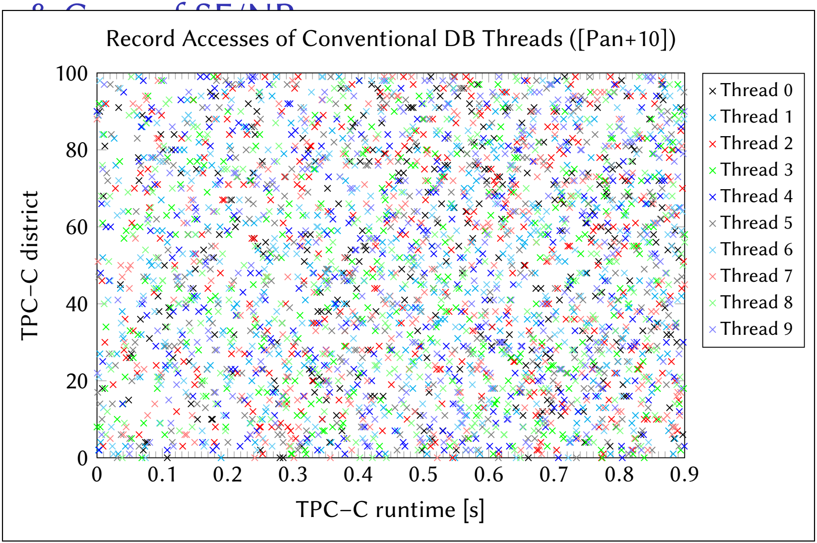
Pros & Cons of SE/NP


Record Accesses of Conventional DB Threads ([Pan+10])

Pros & Cons of SE/NP



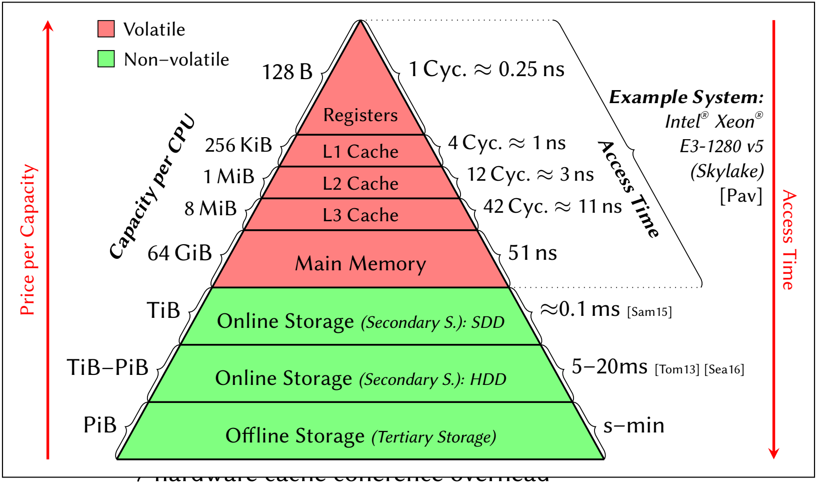
Pros & Cons of SE/NP



Pros & Cons of SE/NP

- + no partitioning required (e.g. manual selection of a strategy)
- + partitioning would be sensitive to the workload
- + changed workloads would require repartitioning to benefit from partitioning
- – each thread might access every record at arbitrary times
  - – each CPU cache may contain any part of the data
    → cache pollution
  - – each CPU may access any part of the data
    → data movement between NUMA regions
  - – each CPU may acquire any latch
    → data movement between NUMA regions
  - – each CPU may atomically write to any semaphore
    → hardware cache coherence overhead

Pros & Cons of SE/NP

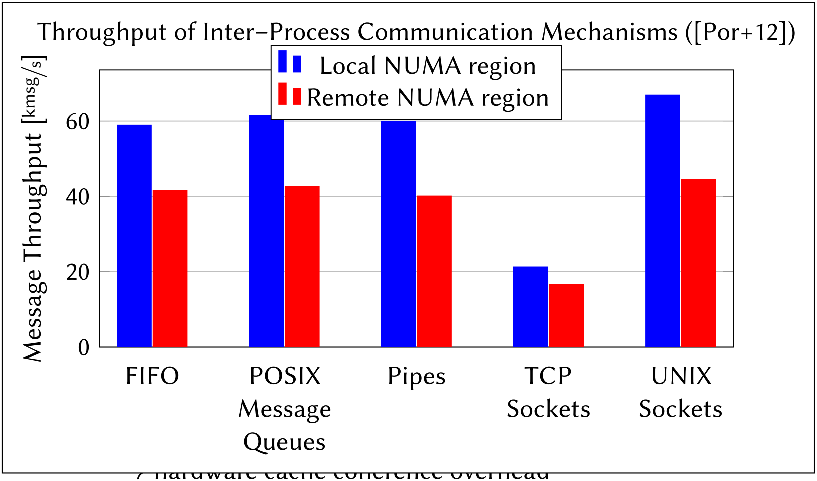


↗ hardware cache coherence overhead

Pros & Cons of SE/NP

- + no partitioning required (e.g. manual selection of a strategy)
- + partitioning would be sensitive to the workload
- + changed workloads would require repartitioning to benefit from partitioning
- – each thread might access every record at arbitrary times
  - – each CPU cache may contain any part of the data
    → cache pollution
  - – each CPU may access any part of the data
    → data movement between NUMA regions
  - – each CPU may acquire any latch
    → data movement between NUMA regions
  - – each CPU may atomically write to any semaphore
    → hardware cache coherence overhead

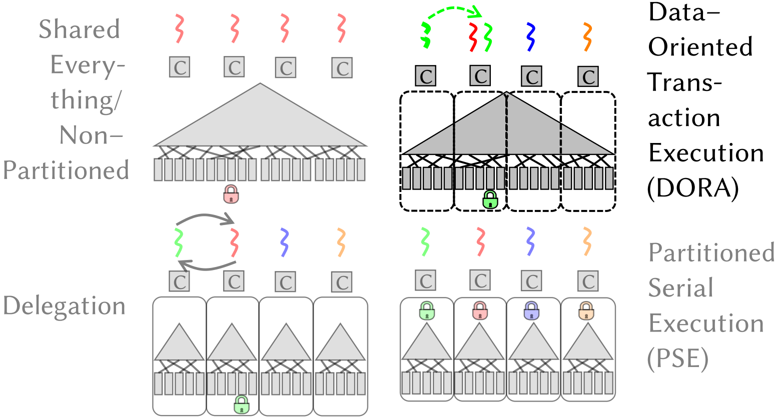
Pros & Cons of SE/NP



↗ hardware cache coherence overhead

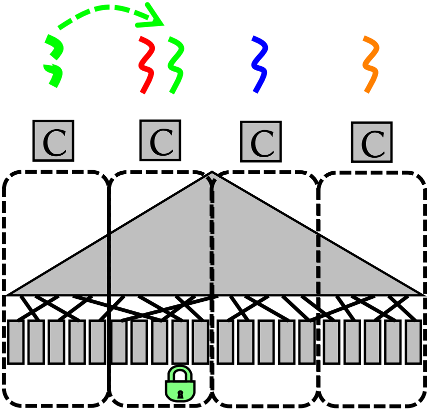
Subsection 2

Data–Oriented Transaction Execution (DORA)


Shared Every-thing/ Non–Partitioned
C
C
C
C
Data–Oriented Trans-action Execution (DORA)
C
C
C
C
Delegation
C
C
C
C
Partitioned Serial Execution (PSE)
C
C
C
C

Subsection 2

Data–Oriented Transaction Execution (DORA)



Properties of DORA

- metadata (incl. locks) are physically partitioned

→ no physical synchronization (latches, atomics) required

- data and indices are logically partitioned

→ logical synchronization using a concurrency control protocol only locally required

- threads are assigned to data
- transactions migrate to threads owning the accessed data

Interactive Example

Thread 0

Fibers:	*idle*
Locks:	
Partition:	0–1, …

Thread 1

Fibers:	*idle*
Locks:	
Partition:	2–3, …

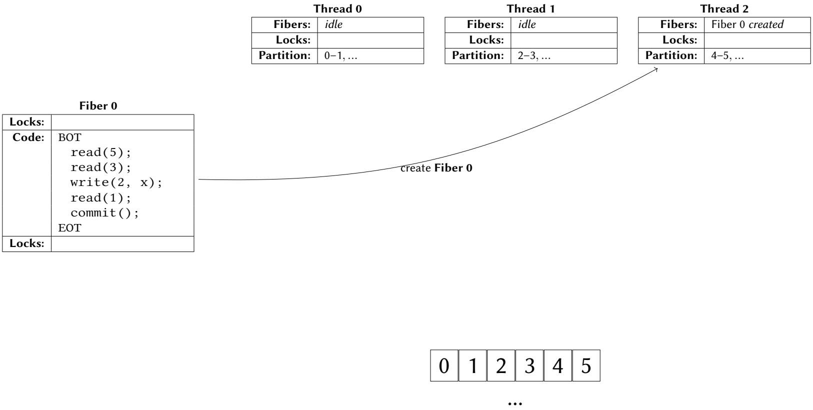
Thread 2

Fibers:	*idle*
Locks:	
Partition:	4–5, …

0	1	2	3	4	5

…

Interactive Example



Thread 0

Fibers:	*idle*
Locks:	
Partition:	0–1, …

Thread 1

Fibers:	*idle*
Locks:	
Partition:	2–3, …

Thread 2

Fibers:	Fiber 0 *created*
Locks:	
Partition:	4–5, …

Fiber 0

Locks:	

Code:	BOT
	read(5);
	read(3);
	write(2, x);
	read(1);
	commit();
	EOT

Locks:	

create **Fiber 0**

0	1	2	3	4	5

…

Interactive Example

Thread 0

Fibers:	*idle*
Locks:	
Partition:	0–1, …

Thread 1

Fibers:	*idle*
Locks:	
Partition:	2–3, …

Thread 2

Fibers:	Fiber 0 *waiting*
Locks:	
Partition:	4–5, …

Fiber 0

Locks:	

Code:	BOT
	`  read(5);`
	`  read(3);`
	`  write(2, x);`
	`  read(1);`
	`  commit();`
	EOT

Locks:	

0	1	2	3	4	5

…

Interactive Example

Thread 0

Fibers:	*idle*
Locks:	
Partition:	0–1, …

Thread 1

Fibers:	*idle*
Locks:	
Partition:	2–3, …

Thread 2

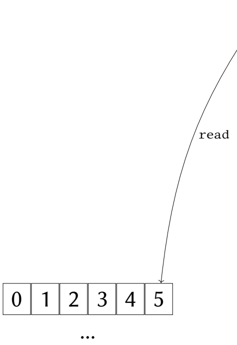
Fibers:	Fiber 0 *running*
Locks:	$(5, S)_0$
Partition:	4–5, …

Fiber 0

Locks:	$(5, S)$

Code:	`BOT`
	`  read(5); <-`
	`  read(3);`
	`  write(2, x);`
	`  read(1);`
	`  commit();`
	`EOT`

Locks:	



Interactive Example

Thread 0

Fibers:	*idle*
Locks:	
Partition:	0–1, …

Thread 1

Fibers:	*idle*
Locks:	
Partition:	2–3, …

Thread 2

Fibers:	Fiber 0 *suspended*
Locks:	$(5, S)_0$
Partition:	4–5, …

Fiber 0

Locks:	$(5, S)$

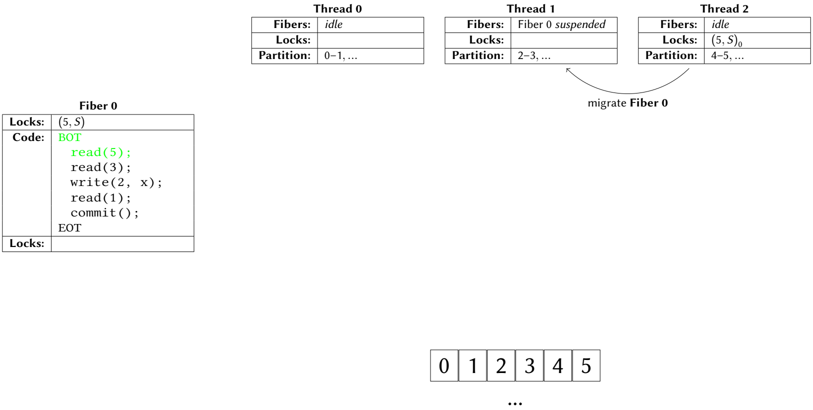
Code:	BOT
	read(5);
	read(3);
	write(2, x);
	read(1);
	commit();
	EOT

Locks:	

0	1	2	3	4	5

…

Interactive Example



Thread 0

Fibers:	*idle*
Locks:	
Partition:	0–1, …

Thread 1

Fibers:	Fiber 0 *suspended*
Locks:	
Partition:	2–3, …

Thread 2

Fibers:	*idle*
Locks:	$(5, S)_0$
Partition:	4–5, …

migrate **Fiber 0**

Fiber 0

Locks:	$(5, S)$

Code:	`BOT`
	`  read(5);`
	`  read(3);`
	`  write(2, x);`
	`  read(1);`
	`  commit();`
	`EOT`

Locks:	

0	1	2	3	4	5

…

Interactive Example

Thread 0

Fibers:	*idle*
Locks:	
Partition:	0–1, …

Thread 1

Fibers:	Fiber 0 *waiting*
Locks:	
Partition:	2–3, …

Thread 2

Fibers:	*idle*
Locks:	$(5, S)_0$
Partition:	4–5, …

Fiber 0

Locks:	$(5, S)$

Code:	BOT
	read(5);
	read(3);
	write(2, x);
	read(1);
	commit();
	EOT

Locks:	

0	1	2	3	4	5

…

Interactive Example

Thread 0

Fibers:	*idle*
Locks:	
Partition:	0–1, …

Thread 1

Fibers:	Fiber 0 *running*
Locks:	$(3, S)_0$
Partition:	2–3, …

Thread 2

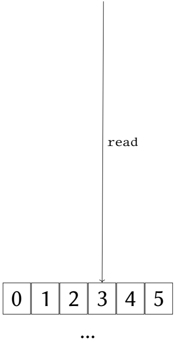
Fibers:	*idle*
Locks:	$(5, S)_0$
Partition:	4–5, …

Fiber 0

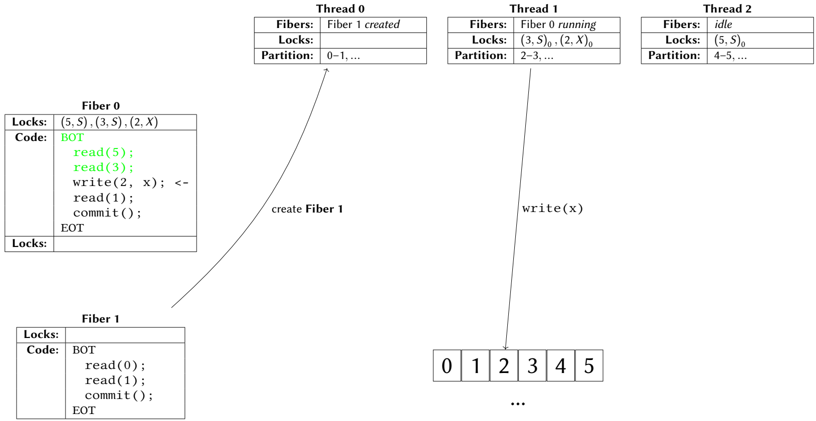
Locks:	$(5, S), (3, S)$

Code:	BOT
	  read(5);
	  read(3); <-
	  write(2, x);
	  read(1);
	  commit();
	EOT

Locks:	



Interactive Example



Thread 0

Fibers:	Fiber 1 *created*
Locks:	
Partition:	0–1, …

Thread 1

Fibers:	Fiber 0 *running*
Locks:	$(3,S)_0$, $(2,X)_0$
Partition:	2–3, …

Thread 2

Fibers:	*idle*
Locks:	$(5,S)_0$
Partition:	4–5, …

Fiber 0

Locks:	$(5,S)$, $(3,S)$, $(2,X)$

Code:	`BOT`
	`  read(5);`
	`  read(3);`
	`  write(2, x); <-`
	`  read(1);`
	`  commit();`
	`EOT`

Locks:	

Fiber 1

Locks:	

Code:	`BOT`
	`  read(0);`
	`  read(1);`
	`  commit();`
	`EOT`

Interactive Example

Thread 0

Fibers:	Fiber 1 *waiting*
Locks:	
Partition:	0–1, …

Thread 1

Fibers:	Fiber 0 *suspended*
Locks:	$(3, S)_0, (2, X)_0$
Partition:	2–3, …

Thread 2

Fibers:	*idle*
Locks:	$(5, S)_0$
Partition:	4–5, …

Fiber 0

Locks:	$(5, S), (3, S), (2, X)$

Code:	`BOT`
	`  read(5);`
	`  read(3);`
	`  write(2, x);`
	`  read(1);`
	`  commit();`
	`EOT`

Locks:	

Fiber 1

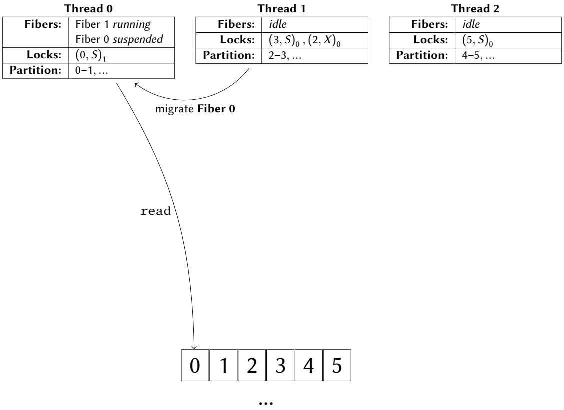
Locks:	

Code:	`BOT`
	`  read(0);`
	`  read(1);`
	`  commit();`
	`EOT`

0	1	2	3	4	5

…

Interactive Example



Thread 0

Fibers:	Fiber 1 *running* Fiber 0 *suspended*
Locks:	$(0, S)_1$
Partition:	0–1, …

Thread 1

Fibers:	*idle*
Locks:	$(3, S)_0, (2, X)_0$
Partition:	2–3, …

Thread 2

Fibers:	*idle*
Locks:	$(5, S)_0$
Partition:	4–5, …

Fiber 0

Locks:	$(5, S), (3, S), (2, X)$

Code:	`BOT`
	`  read(5);`
	`  read(3);`
	`  write(2, x);`
	`  read(1);`
	`  commit();`
	`EOT`

Locks:	

Fiber 1

Locks:	$(0, S)$

Code:	`BOT`
	`  read(0); <-`
	`  read(1);`
	`  commit();`
	`EOT`

Interactive Example

Thread 0

Fibers:	Fiber 1 *running* Fiber 0 *waiting*
Locks:	$(0, S)_1, (1, S)_1$
Partition:	0–1, ...

Thread 1

Fibers:	*idle*
Locks:	$(3, S)_0, (2, X)_0$
Partition:	2–3, ...

Thread 2

Fibers:	*idle*
Locks:	$(5, S)_0$
Partition:	4–5, ...

Fiber 0

Locks:	$(5, S), (3, S), (2, X)$

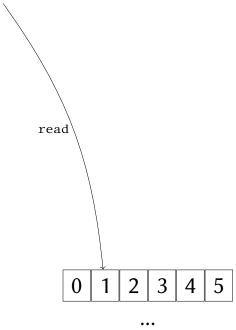
Code:	`BOT`
	`  read(5);`
	`  read(3);`
	`  write(2, x);`
	`  read(1);`
	`  commit();`
	`EOT`

Locks:	

Fiber 1

Locks:	$(0, S), (1, S)$

Code:	`BOT`
	`  read(0);`
	`  read(1); <-`
	`  commit();`
	`EOT`



Interactive Example

Thread 0

Fibers:	Fiber 1 *committing* Fiber 0 *waiting*
Locks:	
Partition:	0–1, …

Thread 1

Fibers:	*idle*
Locks:	$(3, S)_0, (2, X)_0$
Partition:	2–3, …

Thread 2

Fibers:	*idle*
Locks:	$(5, S)_0$
Partition:	4–5, …

Fiber 0

Locks:	$(5, S), (3, S), (2, X)$

Code:	BOT
	  read(5);
	  read(3);
	  write(2, x);
	  read(1);
	  commit();
	EOT

Locks:	

Fiber 1

Locks:	

Code:	BOT
	  read(0);
	  read(1);
	  commit(); <-
	EOT

0	1	2	3	4	5

…

Interactive Example

Thread 0

Fibers:	Fiber 1 *terminated* Fiber 0 *waiting*
Locks:	
Partition:	0–1, …

Thread 1

Fibers:	*idle*
Locks:	$(3, S)_0, (2, X)_0$
Partition:	2–3, …

Thread 2

Fibers:	*idle*
Locks:	$(5, S)_0$
Partition:	4–5, …

Fiber 0

Locks:	$(5, S), (3, S), (2, X)$

Code:	BOT
	read(5);
	read(3);
	write(2, x);
	read(1);
	commit();
	EOT

Locks:	

Fiber 1

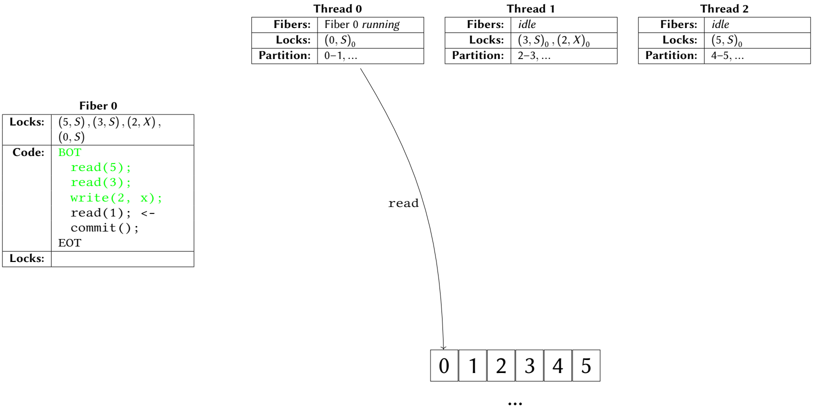
Locks:	

Code:	BOT
	read(0);
	read(1);
	commit();
	EOT

0	1	2	3	4	5

…

Interactive Example



Thread 0

Fibers:	Fiber 0 *running*
Locks:	$(0, S)_0$
Partition:	0–1, …

Thread 1

Fibers:	*idle*
Locks:	$(3, S)_0, (2, X)_0$
Partition:	2–3, …

Thread 2

Fibers:	*idle*
Locks:	$(5, S)_0$
Partition:	4–5, …

Fiber 0

Locks:	$(5, S), (3, S), (2, X),$ $(0, S)$

Code:	`BOT`
	`  read(5);`
	`  read(3);`
	`  write(2, x);`
	`  read(1); <-`
	`  commit();`
	`EOT`

Locks:	

Interactive Example

Thread 0

Fibers:	Fiber 0 *running*
Locks:	$(0, S)_0, (1, S)_0$
Partition:	0–1, …

Thread 1

Fibers:	*idle*
Locks:	$(3, S)_0, (2, X)_0$
Partition:	2–3, …

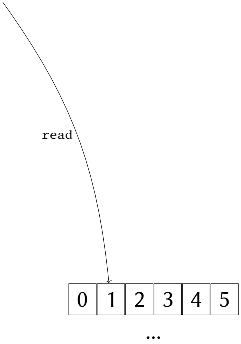
Thread 2

Fibers:	*idle*
Locks:	$(5, S)_0$
Partition:	4–5, …

Fiber 0

Locks:	$(5, S), (3, S), (2, X),$ $(0, S), (1, S)$
Code:	see below
Locks:	

```
BOT
  read(5);
  read(3);
  write(2, x);
  read(1);
  commit(); <-
EOT
```



Interactive Example

Thread 0

Fibers:	Fiber 0 *committing*
Locks:	
Partition:	0–1, …

Thread 1

Fibers:	*idle*
Locks:	$(3, S)_0$, $(2, X)_0$
Partition:	2–3, …

Thread 2

Fibers:	*idle*
Locks:	$(5, S)_0$
Partition:	4–5, …

Fiber 0

Locks:	$(5, S)$, $(3, S)$, $(2, X)$

Code:	`BOT`
	`  read(5);`
	`  read(3);`
	`  write(2, x);`
	`  read(1);`
	`  commit(); <-`
	`EOT`

Locks:	

0	1	2	3	4	5

…

Interactive Example

Thread 0

Fibers:	Fiber 0 *suspended*
Locks:	
Partition:	0–1, …

Thread 1

Fibers:	*idle*
Locks:	$(3, S)_0, (2, X)_0$
Partition:	2–3, …

Thread 2

Fibers:	*idle*
Locks:	$(5, S)_0$
Partition:	4–5, …

Fiber 0

Locks:	$(5, S), (3, S), (2, X)$

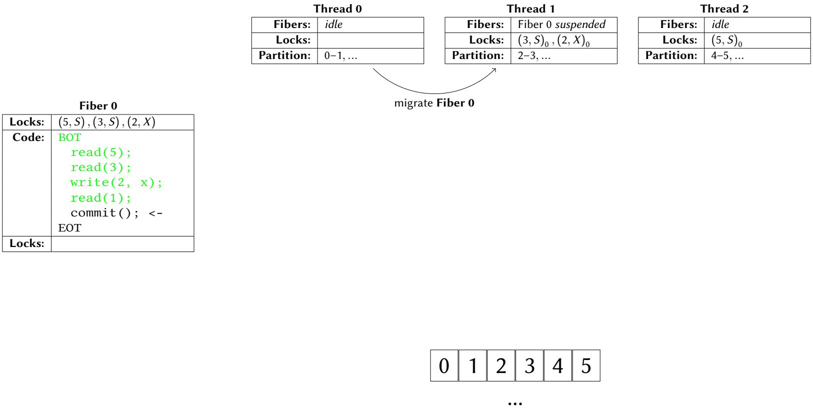
Code:	`BOT`
	`  read(5);`
	`  read(3);`
	`  write(2, x);`
	`  read(1);`
	`  commit(); <-`
	`EOT`

Locks:	

0	1	2	3	4	5

…

Interactive Example



Thread 0

Fibers:	*idle*
Locks:	
Partition:	0–1, ...

Thread 1

Fibers:	Fiber 0 *suspended*
Locks:	$(3, S)_0, (2, X)_0$
Partition:	2–3, ...

Thread 2

Fibers:	*idle*
Locks:	$(5, S)_0$
Partition:	4–5, ...

migrate **Fiber 0**

Fiber 0

Locks:	$(5, S), (3, S), (2, X)$

Code:	BOT
	read(5);
	read(3);
	write(2, x);
	read(1);
	commit(); <-
	EOT

Locks:	

0	1	2	3	4	5

...

Interactive Example

Thread 0

Fibers:	*idle*
Locks:	
Partition:	0–1, …

Thread 1

Fibers:	Fiber 0 *waiting*
Locks:	$(3, S)_0, (2, X)_0$
Partition:	2–3, …

Thread 2

Fibers:	*idle*
Locks:	$(5, S)_0$
Partition:	4–5, …

Fiber 0

Locks:	$(5, S), (3, S), (2, X)$

Code:	`BOT`
	`  read(5);`
	`  read(3);`
	`  write(2, x);`
	`  read(1);`
	`  commit(); <-`
	`EOT`

Locks:	

0	1	2	3	4	5

…

Interactive Example

Thread 0

Fibers:	*idle*
Locks:	
Partition:	0–1, …

Thread 1

Fibers:	Fiber 0 *committing*
Locks:	
Partition:	2–3, …

Thread 2

Fibers:	*idle*
Locks:	$(5, S)_0$
Partition:	4–5, …

Fiber 0

Locks:	$(5, S)$

Code:	`BOT`
	`  read(5);`
	`  read(3);`
	`  write(2, x);`
	`  read(1);`
	`  commit(); <-`
	`EOT`

Locks:	

0	1	2	3	4	5

…

Interactive Example

Thread 0

Fibers:	*idle*
Locks:	
Partition:	0–1, ...

Thread 1

Fibers:	Fiber 0 *suspended*
Locks:	
Partition:	2–3, ...

Thread 2

Fibers:	*idle*
Locks:	$(5, S)_0$
Partition:	4–5, ...

Fiber 0

Locks:	$(5, S)$

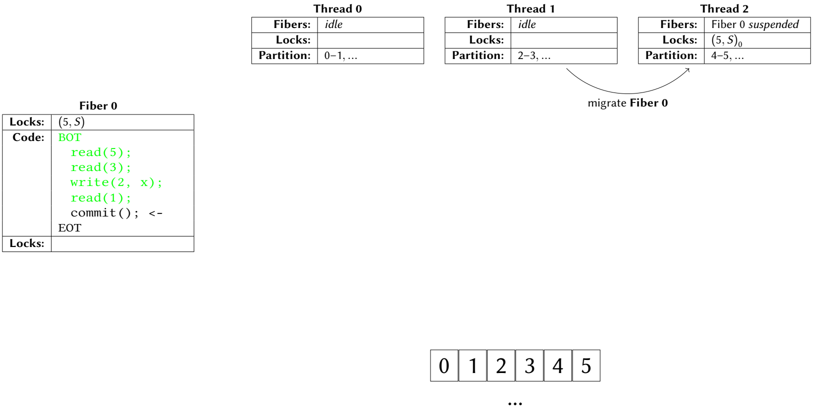
Code:	`BOT`
	`  read(5);`
	`  read(3);`
	`  write(2, x);`
	`  read(1);`
	`  commit(); <-`
	`EOT`

Locks:	

0	1	2	3	4	5

...

Interactive Example



Thread 0

Fibers:	*idle*
Locks:	
Partition:	0–1, …

Thread 1

Fibers:	*idle*
Locks:	
Partition:	2–3, …

Thread 2

Fibers:	Fiber 0 *suspended*
Locks:	$(5, S)_0$
Partition:	4–5, …

migrate **Fiber 0**

Fiber 0

Locks:	$(5, S)$

Code:	`BOT`
	`  read(5);`
	`  read(3);`
	`  write(2, x);`
	`  read(1);`
	`  commit(); <-`
	`EOT`

Locks:	

0	1	2	3	4	5

…

Interactive Example

Thread 0

Fibers:	*idle*
Locks:	
Partition:	0–1, …

Thread 1

Fibers:	*idle*
Locks:	
Partition:	2–3, …

Thread 2

Fibers:	Fiber 0 *waiting*
Locks:	$(5, S)_0$
Partition:	4–5, …

Fiber 0

Locks:	$(5, S)$

Code:	`BOT`
	`  read(5);`
	`  read(3);`
	`  write(2, x);`
	`  read(1);`
	`  commit(); <-`
	`EOT`

Locks:	

0	1	2	3	4	5

…

Interactive Example

Thread 0

Fibers:	*idle*
Locks:	
Partition:	0–1, …

Thread 1

Fibers:	*idle*
Locks:	
Partition:	2–3, …

Thread 2

Fibers:	Fiber 0 *committing*
Locks:	
Partition:	4–5, …

Fiber 0

Locks:	

Code:	BOT
	read(5);
	read(3);
	write(2, x);
	read(1);
	commit(); <-
	EOT

Locks:	

0	1	2	3	4	5

…

Interactive Example

Thread 0

Fibers:	*idle*
Locks:	
Partition:	0–1, …

Thread 1

Fibers:	*idle*
Locks:	
Partition:	2–3, …

Thread 2

Fibers:	Fiber 0 *terminated*
Locks:	
Partition:	4–5, …

Fiber 0

Locks:	

Code:	BOT
	  read(5);
	  read(3);
	  write(2, x);
	  read(1);
	  commit();
	EOT

Locks:	

0	1	2	3	4	5

…

Interactive Example

Thread 0

Fibers:	*idle*
Locks:	
Partition:	0–1, …

Thread 1

Fibers:	*idle*
Locks:	
Partition:	2–3, …

Thread 2

Fibers:	*idle*
Locks:	
Partition:	4–5, …

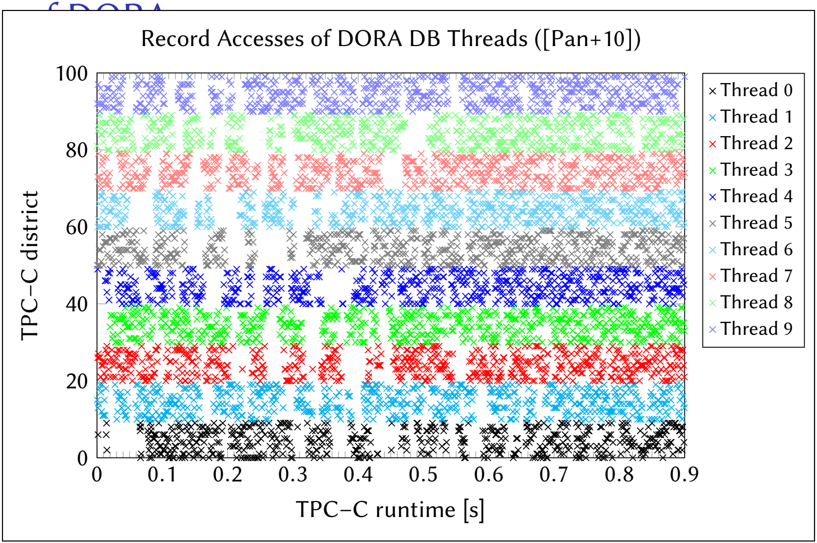
0	1	2	3	4	5

…

Pros of DORA

- + each thread accesses only the records of its partition
  - + each CPU cache may contain only data of its partition → lower cache pollution
  - + each CPU may access only data of its partitions → no data movement between NUMA regions (for single–CPU transactions)
  - → No physical synchronization required!
- + logical partitioning allows fast repartitioning when the workload changes
- + intra–transaction parallelism could be exploited for multi–site transactions

Pro


Record Accesses of DORA DB Threads ([Pan+10])
100
80
60
40
20
0
TPC–C district
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
TPC–C runtime [s]
× Thread 0
× Thread 1
× Thread 2
× Thread 3
× Thread 4
× Thread 5
× Thread 6
× Thread 7
× Thread 8
× Thread 9

Cons of DORA

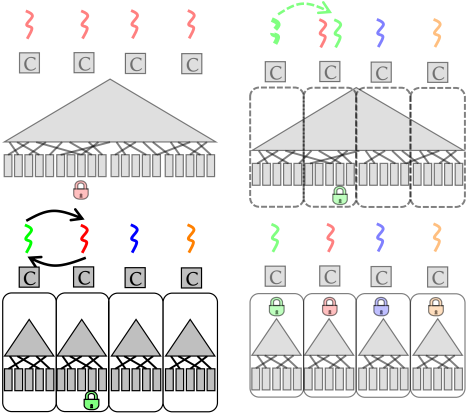
- partitioning required (e.g. manual selection of a partitioning strategy—*called routing rule*)
- partitioning is sensitive to the workload
- multi–site transactions require expensive fiber–migration (probably between NUMA regions)
- accessed partitions need to be calculated during query analysis for optimal performance
  → slower accesses with secondary index
- primary index is shared
  → centralized latching for inserts/deletes still required
  → some contention on the shared latch
- centralized deadlock detection still required (for DL_DETECT)

Subsection 3

Delegation

Shared Everything/ Non–Partitioned

Data–Oriented Transaction Execution (DORA)

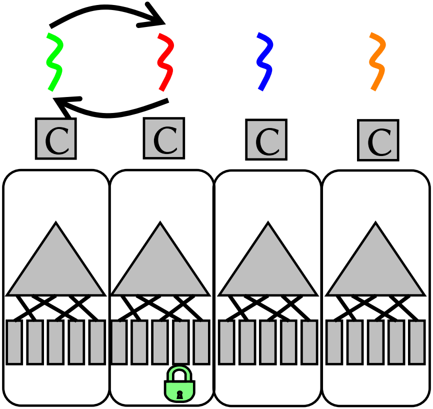


Delegation

Partitioned Serial Execution (PSE)

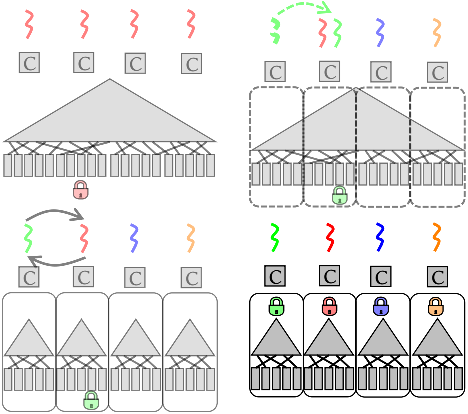
Subsection 3

Delegation



Subsection 4

Partitioned Serial Execution (PSE)



Shared Everything/ Non–Partitioned

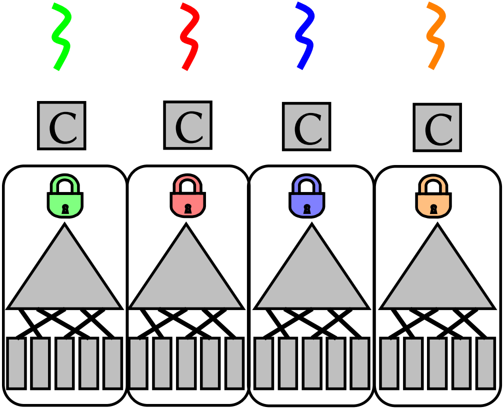
Data–Oriented Transaction Execution (DORA)

Delegation

Partitioned Serial Execution (PSE)

Subsection 4

Partitioned Serial Execution (PSE)



Summary

Architecture				
SE/NP				
PSE				
Delegation				
DORA				

Summary

Architecture	Process Management			
	Parallelism			
SE/NP	Shared Memory			
PSE	Shared Nothing			
Delegation	Message Passing			
DORA	Shared Memory			

Summary

Architecture	**Process Management**			
	Parallelism	**Thread Assignment**		
SE/NP	Shared Memory	thread–to–txn		
PSE	Shared Nothing	thread–to–txn		
Delegation	Message Passing	thread–to–txn		
DORA	Shared Memory	thread–to–data		

Summary

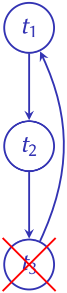
Architecture	Process Management		Transactional Storage Management	
	Parallelism	Thread Assignment	Logical Synchronization	
SE/NP	Shared Memory	thread–to–txn	CC Protocols	
PSE	Shared Nothing	thread–to–txn	Partition Lock	
Delegation	Message Passing	thread–to–txn	CC Protocols	
DORA	Shared Memory	thread–to–data	CC Protocols	

Summary

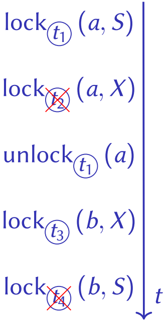
Architecture	Process Management		Transactional Storage Management	
	Parallelism	Thread Assignment	Logical Synchronization	Physical Synchronization
SE/NP	Shared Memory	thread–to–txn	CC Protocols	latch/atomics
PSE	Shared Nothing	thread–to–txn	Partition Lock	partition lock
Delegation	Message Passing	thread–to–txn	CC Protocols	Message Passing
DORA	Shared Memory	thread–to–data	CC Protocols	Transaction Migration

Section 3

Concurrency Control Algorithms



DL_DETECT
(2PL)



$\mathrm{lock}_{t_1}(a, S)$

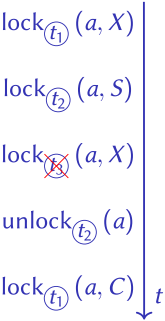
$\mathrm{lock}_{\cancel{t_2}}(a, X)$

$\mathrm{unlock}_{t_1}(a)$

$\mathrm{lock}_{t_3}(b, X)$

$\mathrm{lock}_{\cancel{t_4}}(b, S)$

NO_WAIT
(No–Waiting–2PL)



$\mathrm{lock}_{t_1}(a, X)$

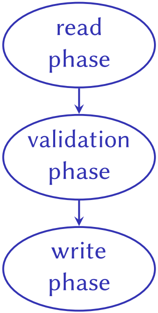
$\mathrm{lock}_{t_2}(a, S)$

$\mathrm{lock}_{\cancel{t_3}}(a, X)$

$\mathrm{unlock}_{t_2}(a)$

$\mathrm{lock}_{t_1}(a, C)$

2V_NO_WAIT
(Two–Version–2PL)



read phase

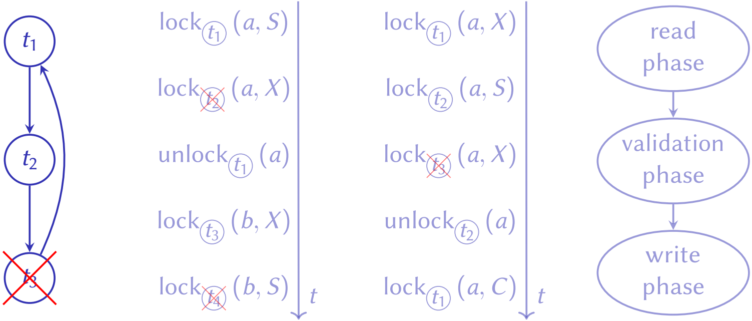
validation phase

write phase

SILO
(OCC)

Subsection 1

DL_DETECT (2PL)



$\text{lock}_{t_1}(a, S)$
$\text{lock}_{\cancel{t_2}}(a, X)$
$\text{unlock}_{t_1}(a)$
$\text{lock}_{t_3}(b, X)$
$\text{lock}_{\cancel{t_4}}(b, S)$
t

$\text{lock}_{t_1}(a, X)$
$\text{lock}_{t_2}(a, S)$
$\text{lock}_{\cancel{t_3}}(a, X)$
$\text{unlock}_{t_2}(a)$
$\text{lock}_{t_1}(a, C)$
t

read phase
validation phase
write phase

DL_DETECT (2PL)

NO_WAIT (No–Waiting–2PL)

2V_NO_WAIT (Two–Version–2PL)

SILO (OCC)

Properties of DL_DETECT (2PL)

- pessimistic concurrency control protocol
- transactions lock database objects (databases, tables, records, key ranges, etc.) before reading (shared mode S) or updating (exclusive mode X) them [Moh90]
- t_0 tries to acquire lock held by t_1 in compatible mode $\rightarrow$ t_0 can immediately acquire lock as well (starvation needs to be prevented)
- t_0 tries to acquire lock held by t_1 in incompatible mode $\rightarrow$ t_0 waits until t_1 releases lock
- deadlock detection using a repeatedly generated and analyzed wait–for graph

compatibility	shared mode	exclusive mode
shared mode	⊕	⊖
exclusive mode	⊖	⊖

Interactive Example

Transactions:

t_0 t_1 t_2

Locks:

Record 0		**Record 1**		**Record 2**		···
Current Mode:	NL	Current Mode:	NL	Current Mode:	NL	
Waiters:		Waiters:		Waiters:		
Data:	x_0	Data:	x_1	Data:	x_2	

Wait–for Graph:

Interactive Example

Transactions:

t_0	t_1	t_2
— BOT		

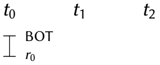
Locks:

Record 0		Record 1		Record 2		···
Current Mode:	NL	Current Mode:	NL	Current Mode:	NL	
Waiters:		Waiters:		Waiters:		
Data:	x_0	Data:	x_1	Data:	x_2	

Wait–for Graph:

Interactive Example

Transactions:



Locks:

Record 0		Record 1		Record 2		···
Current Mode:	S (1)	Current Mode:	NL	Current Mode:	NL	
Waiters:		Waiters:		Waiters:		
Data:	x_0	Data:	x_1	Data:	x_2	

Wait–for Graph:

Interactive Example

Transactions:

t_0 t_1 t_2

BOT
r_0

— BOT

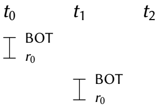
Locks:

Record 0		Record 1		Record 2		…
Current Mode:	S (1)	Current Mode:	NL	Current Mode:	NL	
Waiters:		Waiters:		Waiters:		
Data:	x_0	Data:	x_1	Data:	x_2	

Wait–for Graph:

Interactive Example

Transactions:



Locks:

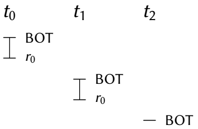
Record 0		Record 1		Record 2	
Current Mode:	S (2)	Current Mode:	NL	Current Mode:	NL
Waiters:		Waiters:		Waiters:	
Data:	x_0	Data:	x_1	Data:	x_2

···

Wait–for Graph:

Interactive Example

Transactions:



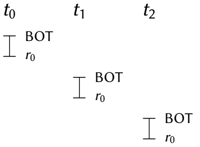
Locks:

Record 0		Record 1		Record 2		…
Current Mode:	S (2)	Current Mode:	NL	Current Mode:	NL	
Waiters:		Waiters:		Waiters:		
Data:	x_0	Data:	x_1	Data:	x_2	

Wait–for Graph:

Interactive Example

Transactions:



Locks:

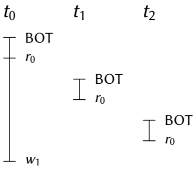
Record 0		Record 1		Record 2	
Current Mode:	S (3)	Current Mode:	NL	Current Mode:	NL
Waiters:		Waiters:		Waiters:	
Data:	x_0	Data:	x_1	Data:	x_2

···

Wait–for Graph:

Interactive Example

Transactions:

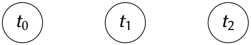


Locks:

Record 0		Record 1		Record 2	
Current Mode:	S (3)	Current Mode:	X (t_0)	Current Mode:	NL
Waiters:		Waiters:		Waiters:	
Data:	x_0	Data:	x_1	Data:	x_2

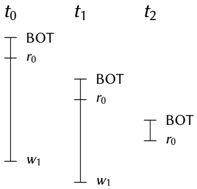
…

Wait–for Graph:



Interactive Example

Transactions:

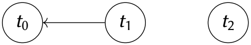


Locks:

Record 0		**Record 1**		**Record 2**	
Current Mode:	S (3)	Current Mode:	X (t_0)	Current Mode:	NL
Waiters:		Waiters:	(X, t_1)	Waiters:	
Data:	x_0	Data:	x_1'	Data:	x_2

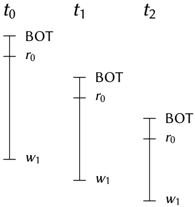
⋯

Wait–for Graph:



Interactive Example

Transactions:



Locks:

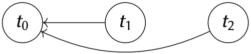
Record 0	
Current Mode:	S (3)
Waiters:	
Data:	x_0

Record 1	
Current Mode:	X (t_0)
Waiters:	(X, t_1) (X, t_2)
Data:	x_1'

Record 2	
Current Mode:	NL
Waiters:	
Data:	x_2

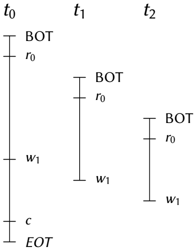
…

Wait–for Graph:



Interactive Example

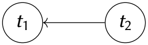
Transactions:



Locks:

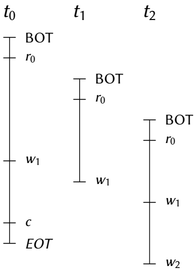
Record 0		**Record 1**		**Record 2**		⋯
Current Mode:	S (2)	Current Mode:	X (t_1)	Current Mode:	NL	
Waiters:		Waiters:	(X, t_2)	Waiters:		
Data:	x_0	Data:	x_1'	Data:	x_2	

Wait–for Graph:



Interactive Example

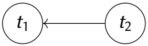
Transactions:



Locks:

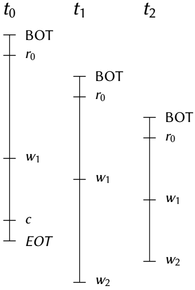
Record 0		**Record 1**		**Record 2**		…
Current Mode:	S (2)	Current Mode:	X (t_1)	Current Mode:	X (t_2)	
Waiters:		Waiters:	(X, t_2)	Waiters:		
Data:	x_0	Data:	x_1''	Data:	x_2	

Wait–for Graph:



Interactive Example

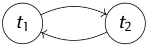
Transactions:



Locks:

Record 0		**Record 1**		**Record 2**		…
Current Mode:	S (2)	Current Mode:	X (t_1)	Current Mode:	X (t_2)	
Waiters:		Waiters:	(X, t_2)	Waiters:	(X, t_1)	
Data:	x_0	Data:	x_1''	Data:	x_2'	

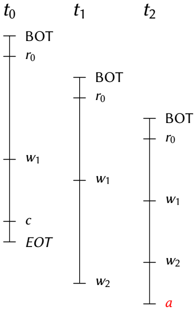
Wait–for Graph:



Cycle → Deadlock → Rollback a blocked Transaction

Interactive Example

Transactions:



Locks:

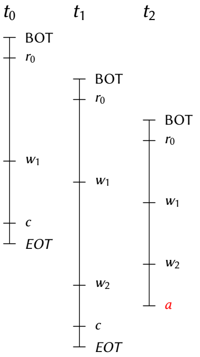
Record 0		**Record 1**		**Record 2**	
Current Mode:	S (1)	Current Mode:	X (t_1)	Current Mode:	X (t_1)
Waiters:		Waiters:		Waiters:	
Data:	x_0	Data:	x_1''	Data:	x_2

⋯

Wait–for Graph:

Interactive Example

Transactions:



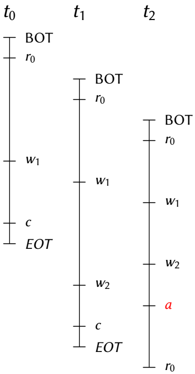
Locks:

Record 0		**Record 1**		**Record 2**		…
Current Mode:	NL	Current Mode:	NL	Current Mode:	NL	
Waiters:		Waiters:		Waiters:		
Data:	x_0	Data:	x_1''	Data:	x_2''	

Wait–for Graph:

Interactive Example

Transactions:



t_0: BOT, r_0, w_1, c, EOT

t_1: BOT, r_0, w_1, w_2, c, EOT

t_2: BOT, r_0, w_1, w_2, a, r_0

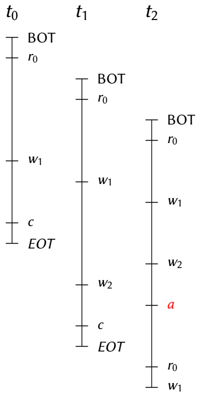
Locks:

Record 0		Record 1		Record 2		⋯
Current Mode:	S (1)	Current Mode:	NL	Current Mode:	NL	
Waiters:		Waiters:		Waiters:		
Data:	x_0	Data:	x_1''	Data:	x_2''	

Wait–for Graph:

Interactive Example

Transactions:



Locks:

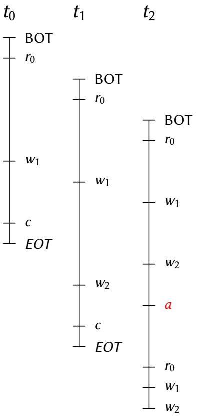
Record 0		**Record 1**		**Record 2**	
Current Mode:	S (1)	Current Mode:	X (t_2)	Current Mode:	NL
Waiters:		Waiters:		Waiters:	
Data:	x_0	Data:	x_1''	Data:	x_2''

…

Wait–for Graph:

Interactive Example

Transactions:



Locks:

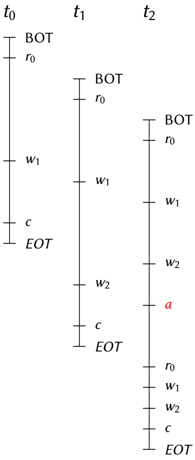
Record 0		Record 1		Record 2	
Current Mode:	S (1)	Current Mode:	X (t_2)	Current Mode:	X (t_2)
Waiters:		Waiters:		Waiters:	
Data:	x_0	Data:	x_1'''	Data:	x_2''

…

Wait–for Graph:

Interactive Example

Transactions:



Locks:

Record 0		**Record 1**		**Record 2**		···
Current Mode:	NL	Current Mode:	NL	Current Mode:	NL	
Waiters:		Waiters:		Waiters:		
Data:	x_0	Data:	x_1'''	Data:	x_2'	

Wait–for Graph:

Pros & Cons of DL_DETECT (2PL)

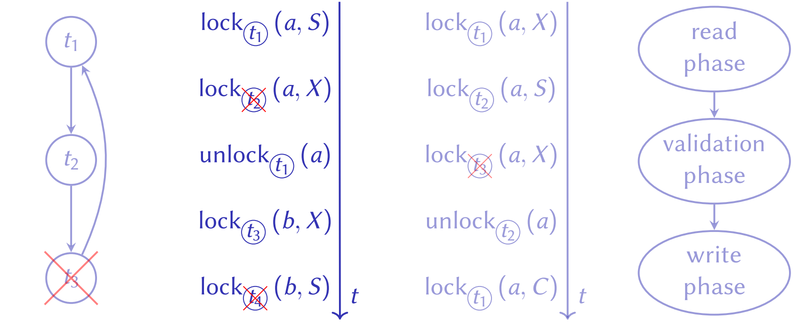
+ aborts only after deadlocks

Pros & Cons of DL_DETECT (2PL)

- + aborts only after deadlocks
- – deadlocks are possible
- – locks prevent concurrency too often (e.g. blind writes)
- – calculation and analysis of wait–for graph expensive
  → done offline → transactions deadlocked for a while
- – aborts happen
  → work done before needs to be repeated
- – queue of waiters requires latching
  → limits scalability
- – even writes need to acquire latches and wait

Subsection 2

NO_WAIT (No–Waiting–2PL)



lock$_{t_1}$ (a, S)
lock$_{\cancel{t_2}}$ (a, X)
unlock$_{t_1}$ (a)
lock$_{t_3}$ (b, X)
lock$_{\cancel{t_4}}$ (b, S)
t

lock$_{t_1}$ (a, X)
lock$_{t_2}$ (a, S)
lock$_{\cancel{t_3}}$ (a, X)
unlock$_{t_2}$ (a)
lock$_{t_1}$ (a, C)
t

DL_DETECT (2PL)

NO_WAIT (No–Waiting–2PL)

2V_NO_WAIT (Two–Version–2PL)

SILO (OCC)

Properties of NO_WAIT (No–Waiting–2PL)

- pessimistic concurrency control protocol
- transactions lock database objects (databases, tables, records, key ranges, etc.) before reading (shared mode *S*) or updating (exclusive mode *X*) them [Moh90]
- t_0 tries to acquire lock held by t_1 in compatible mode
  $\longrightarrow$ t_0 can immediately acquire lock as well (starvation needs to be prevented)
- t_0 tries to acquire lock held by t_1 in incompatible mode
  $\longrightarrow$ t_0 aborts

compatibility	shared mode	exclusive mode
shared mode	⊕	⊖
exclusive mode	⊖	⊖

Interactive Example

Transactions:

t_0 t_1 t_2

Locks:

Record 0		**Record 1**		**Record 2**		···
Current Mode:	0	Current Mode:	0	Current Mode:	0	
Data:	x_0	Data:	x_1	Data:	x_2	

Interactive Example

Transactions:

t_0 t_1 t_2

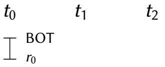
— BOT

Locks:

Record 0		**Record 1**		**Record 2**		···
Current Mode:	0	Current Mode:	0	Current Mode:	0	
Data:	x_0	Data:	x_1	Data:	x_2	

Interactive Example

Transactions:



t_0 t_1 t_2

BOT
r_0

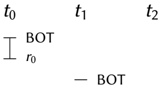
Locks:

Record 0		**Record 1**		**Record 2**	
Current Mode:	2	Current Mode:	0	Current Mode:	0
Data:	x_0	Data:	x_1	Data:	x_2

…

Interactive Example

Transactions:

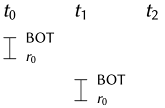


Locks:

Record 0		Record 1		Record 2		···
Current Mode:	2	Current Mode:	0	Current Mode:	0	
Data:	x_0	Data:	x_1	Data:	x_2	

Interactive Example

Transactions:



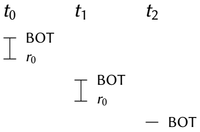
Locks:

Record 0		Record 1		Record 2	
Current Mode:	4	Current Mode:	0	Current Mode:	0
Data:	x_0	Data:	x_1	Data:	x_2

···

Interactive Example

Transactions:

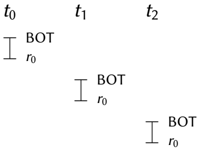


Locks:

Record 0		**Record 1**		**Record 2**		···
Current Mode:	4	Current Mode:	0	Current Mode:	0	
Data:	x_0	Data:	x_1	Data:	x_2	

Interactive Example

Transactions:

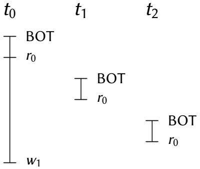


Locks:

Record 0		**Record 1**		**Record 2**		⋯
Current Mode:	6	Current Mode:	0	Current Mode:	0	
Data:	x_0	Data:	x_1	Data:	x_2	

Interactive Example

Transactions:

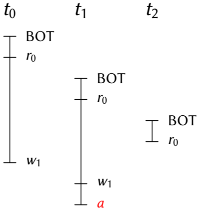


Locks:

Record 0		**Record 1**		**Record 2**		…
Current Mode:	6	Current Mode:	1	Current Mode:	0	
Data:	x_0	Data:	x_1	Data:	x_2	

Interactive Example

Transactions:



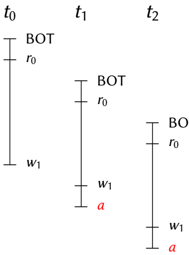
Locks:

Record 0		**Record 1**		**Record 2**	
Current Mode:	4	Current Mode:	1	Current Mode:	0
Data:	x_0	Data:	x_1'	Data:	x_2

···

Interactive Example

Transactions:



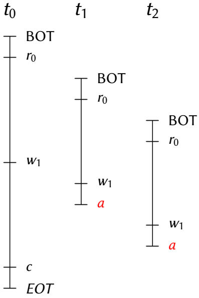
Locks:

Record 0		Record 1		Record 2	
Current Mode:	2	Current Mode:	1	Current Mode:	0
Data:	x_0	Data:	x_1'	Data:	x_2

⋯

Interactive Example

Transactions:

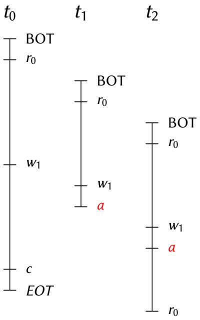


Locks:

Record 0		**Record 1**		**Record 2**		⋯
Current Mode:	0	Current Mode:	0	Current Mode:	0	
Data:	x_0	Data:	x_1'	Data:	x_2	

Interactive Example

Transactions:



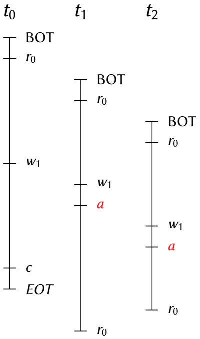
Locks:

Record 0		Record 1		Record 2	
Current Mode:	2	Current Mode:	0	Current Mode:	0
Data:	x_0	Data:	x'_1	Data:	x_2

⋯

Interactive Example

Transactions:



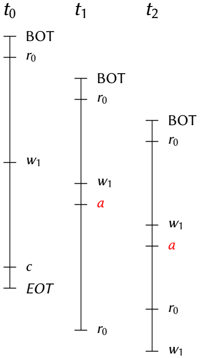
Locks:

Record 0		Record 1		Record 2	
Current Mode:	4	Current Mode:	0	Current Mode:	0
Data:	x_0	Data:	x_1'	Data:	x_2

⋯

Interactive Example

Transactions:

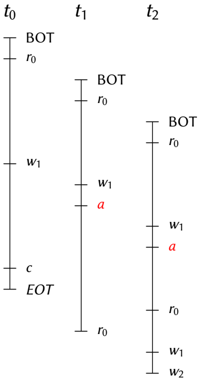


Locks:

Record 0		**Record 1**		**Record 2**		⋯
Current Mode:	4	Current Mode:	1	Current Mode:	0	
Data:	x_0	Data:	x_1'	Data:	x_2	

Interactive Example

Transactions:

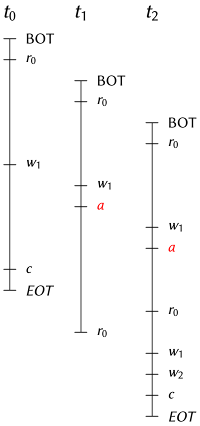


Locks:

Record 0		**Record 1**		**Record 2**		…
Current Mode:	4	Current Mode:	1	Current Mode:	1	
Data:	x_0	Data:	x_1''	Data:	x_2	

Interactive Example

Transactions:



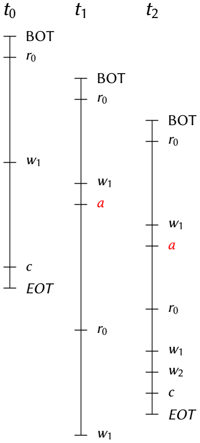
Locks:

Record 0		Record 1		Record 2	
Current Mode:	2	Current Mode:	0	Current Mode:	0
Data:	x_0	Data:	x_1''	Data:	x_2'

···

Interactive Example

Transactions:

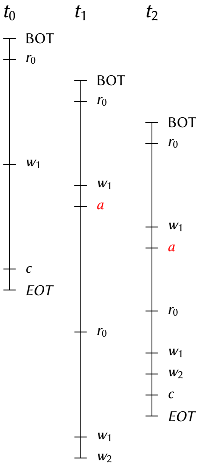


Locks:

Record 0		**Record 1**		**Record 2**		…
Current Mode:	2	Current Mode:	1	Current Mode:	0	
Data:	x_0	Data:	x_1''	Data:	x_2'	

Interactive Example

Transactions:



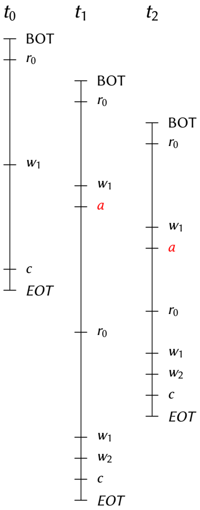
Locks:

Record 0		Record 1		Record 2	
Current Mode:	2	Current Mode:	1	Current Mode:	1
Data:	x_0	Data:	x_1'''	Data:	x_2'

⋯

Interactive Example

Transactions:



Locks:

Record 0		**Record 1**		**Record 2**		…
Current Mode:	0	Current Mode:	0	Current Mode:	0	
Data:	x_0	Data:	x_1'''	Data:	x_2''	

Pros & Cons of NO_WAIT (No–Waiting–2PL)

+ deadlocks are impossible
+ locks can be implemented using a semaphore and atomics
  → scales better than latches
+ no need to expensively calculate and analysis a wait–for graph

Pros & Cons of NO_WAIT (No–Waiting–2PL)

+ deadlocks are impossible

+ locks can be implemented using a semaphore and atomics
→ scales better than latches

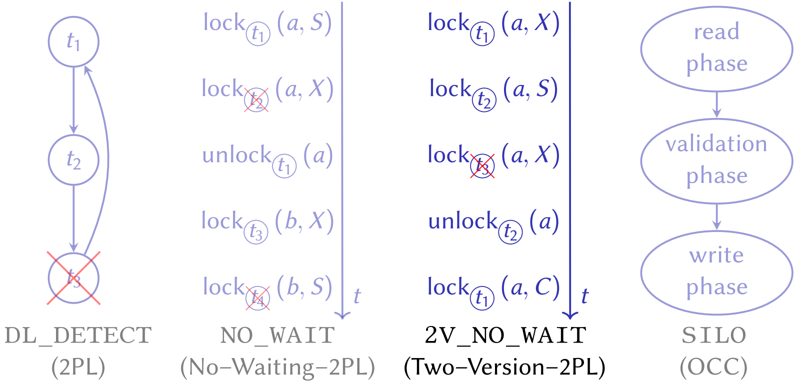
+ no need to expensively calculate and analysis a wait–for graph

– many lock conflicts for update–intensive high–contention workloads
→ many aborts → work done before needs to be repeated

– locks prevent concurrency too often (e.g. blind writes)

Subsection 3

2V_NO_WAIT (Two–Version–2PL)



$\text{lock}_{t_1}(a, S)$
$\text{lock}_{\cancel{t_2}}(a, X)$
$\text{unlock}_{t_1}(a)$
$\text{lock}_{t_3}(b, X)$
$\text{lock}_{\cancel{t_4}}(b, S)$
t

$\text{lock}_{t_1}(a, X)$
$\text{lock}_{t_2}(a, S)$
$\text{lock}_{\cancel{t_3}}(a, X)$
$\text{unlock}_{t_2}(a)$
$\text{lock}_{t_1}(a, C)$
t

read phase

validation phase

write phase

DL_DETECT (2PL)

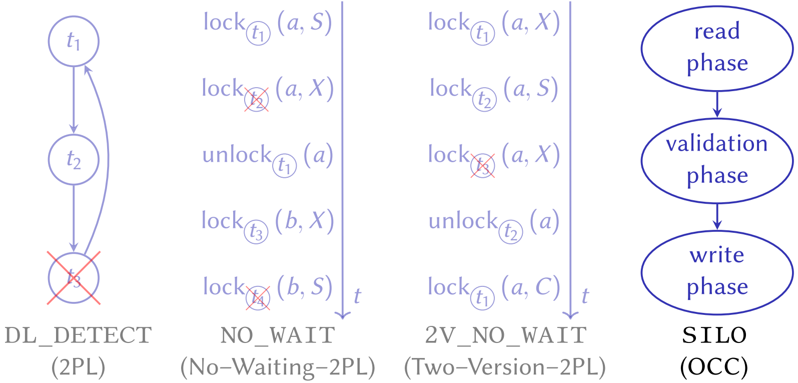
NO_WAIT (No–Waiting–2PL)

2V_NO_WAIT (Two–Version–2PL)

SILO (OCC)

Subsection 4

SILO (OCC)



t_1 t_2 t_3

DL_DETECT
(2PL)

$\text{lock}_{t_1}(a, S)$
$\text{lock}_{t_2}(a, X)$
$\text{unlock}_{t_1}(a)$
$\text{lock}_{t_3}(b, X)$
$\text{lock}_{t_4}(b, S)$
t

NO_WAIT
(No–Waiting–2PL)

$\text{lock}_{t_1}(a, X)$
$\text{lock}_{t_2}(a, S)$
$\text{lock}_{t_3}(a, X)$
$\text{unlock}_{t_2}(a)$
$\text{lock}_{t_1}(a, C)$
t

2V_NO_WAIT
(Two–Version–2PL)

read phase
validation phase
write phase

SILO
(OCC)

Section 4

Performance Evaluation

	SE/NP	DORA	Delegation	PSE
DL_DETECT	⊕	⊕	⊕	⊙
NO_WAIT	⊕	⊕	⊕	
2V_NO_WAIT	⊕	⊕	⊕	
SILO	⊕	⊖	⊕	

Evaluation Set–Up

- 4x Intel Xeon E7-8890 v3 NUMA machine (72 cores @ 2.5 GHz)
- 32 kB L1I cache and 32 kB L1D cache per core
- 256 kB L2 cache per core
- 45 MB L3 cache per CPU
- 512 GB DDR4 RAM
- hyperThreading not used
- threads pinned to physical cores
- sockets filled sequentially with threads

Benchmarks

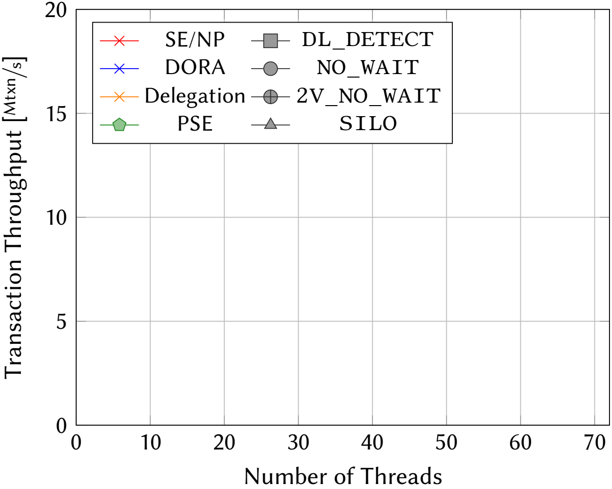
Microbenchmark

- 13 GB database
- Hot Set: 16 records *distributed to 16 partitions*
- Cold Set: 100 000 000 − 16 records
- Txn: 2 accesses to Hot Set & 8 accesses to *(thread–local)* Cold Set

Yahoo! Cloud Serving Benchmark (YCSB)

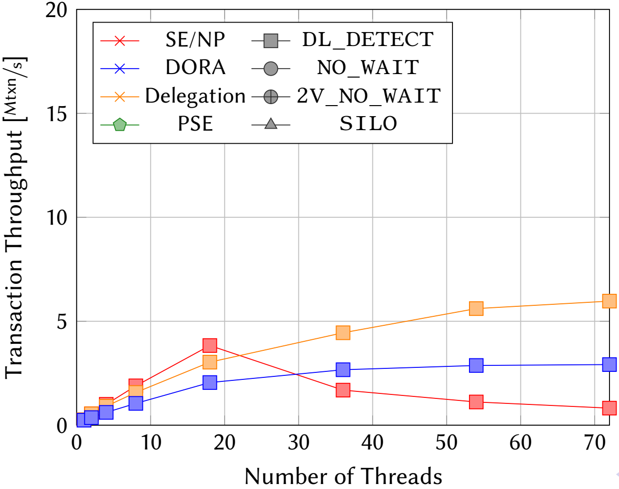
- 20 GB database
- 20 000 000 records
- Txn: reads/updates 16 records following Zipfian distribution according to parameter Θ

Read–Only Microbenchmark



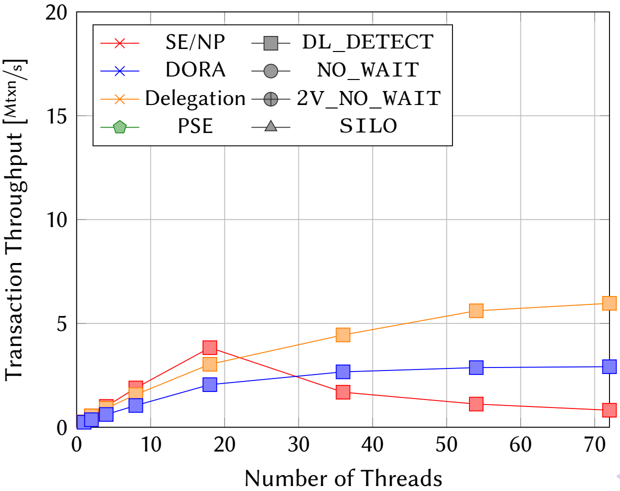
Observations

Read–Only Microbenchmark



Observations

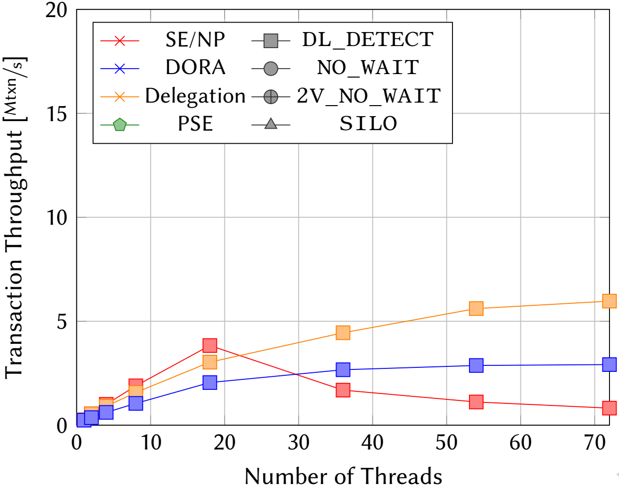
Read–Only Microbenchmark



Observations

- ✕/✕ suffer from remote data access overhead

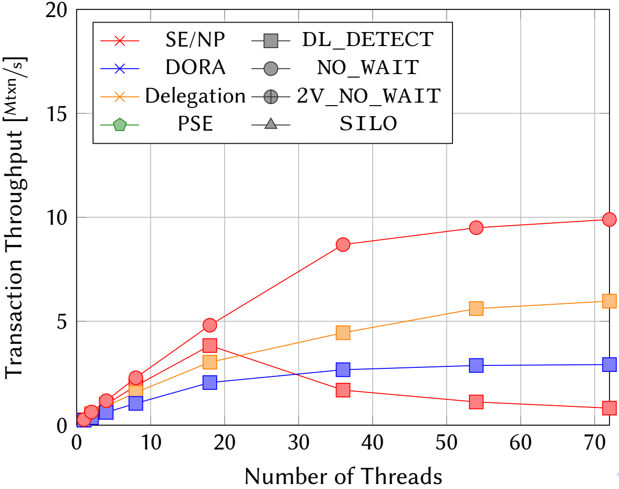
Read–Only Microbenchmark



Observations

- ×/× suffer from remote data access overhead
- × suffers from latch contention on locks

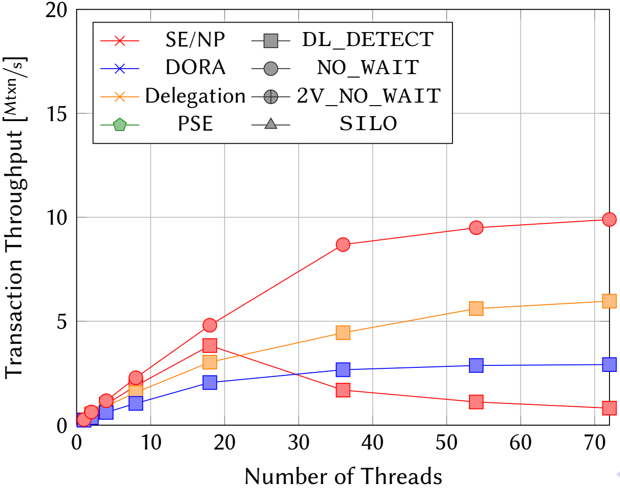
Read–Only Microbenchmark



Observations

- ×/× suffer from remote data access overhead
- × suffers from latch contention on locks

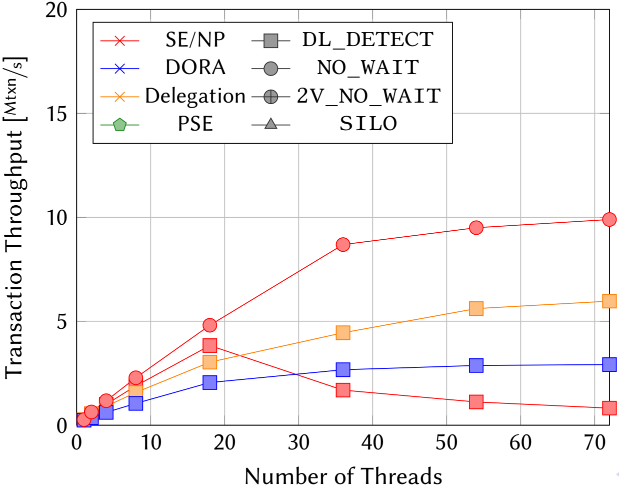
Read–Only Microbenchmark



Observations

- ×/× suffer from remote data access overhead
- × suffers from latch contention on locks
- atomics of ● outperform latches of ■

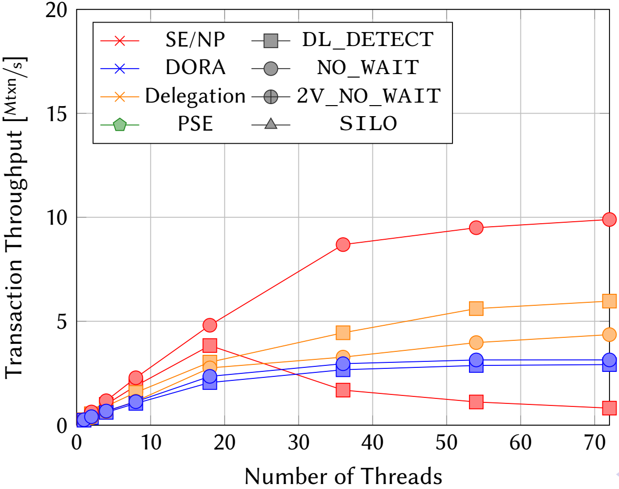
Read–Only Microbenchmark



Observations

- ×/× suffer from remote data access overhead
- × suffers from latch contention on locks
- atomics of ● outperform latches of ■
- scaling of ● limited by hardware cache coherence mechanism

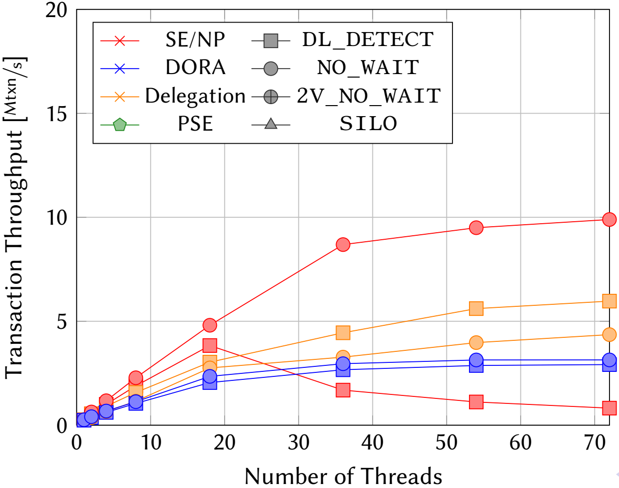
Read–Only Microbenchmark



Observations

- ×/× suffer from remote data access overhead
- × suffers from latch contention on locks
- atomics of ● outperform latches of ■
- scaling of ● limited by hardware cache coherence mechanism

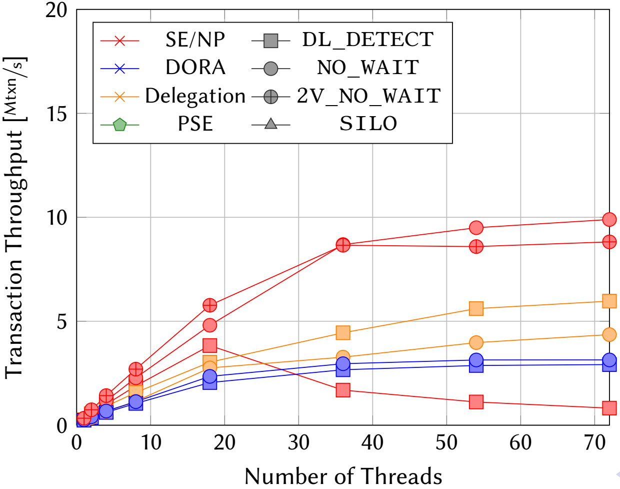
Read–Only Microbenchmark



Observations

- × suffers from latch contention on locks
- atomics of ● outperform latches of ■
- scaling of ● limited by hardware cache coherence mechanism
- ×/× suffer more from remote data accesses than × suffers from cache coherence

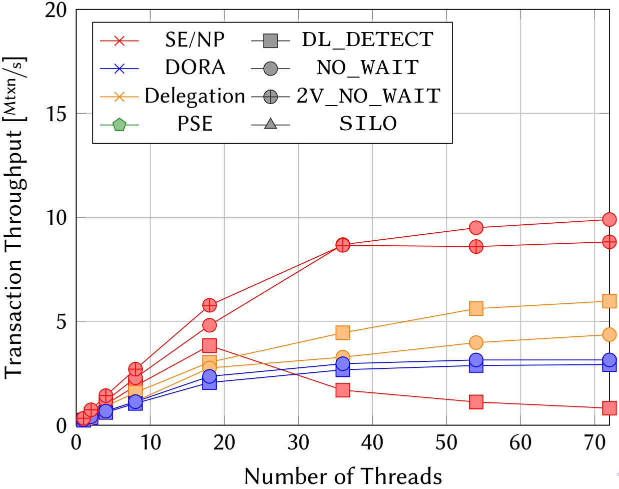
Read–Only Microbenchmark



Observations

- × suffers from latch contention on locks
- atomics of ● outperform latches of ■
- scaling of ● limited by hardware cache coherence mechanism
- ×/× suffer more from remote data accesses than × suffers from cache coherence

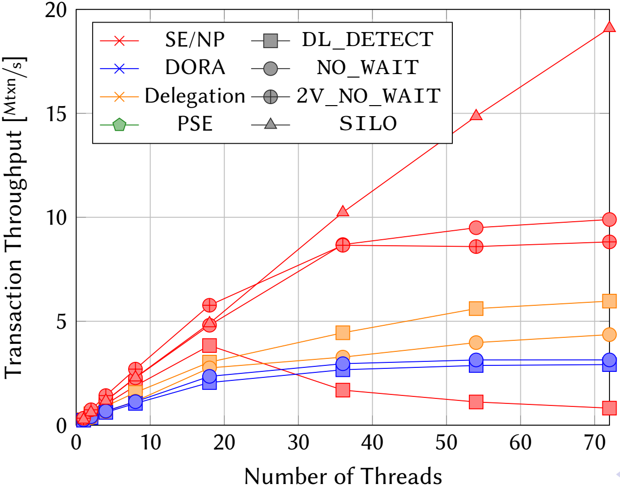
Read–Only Microbenchmark



Observations

- atomics of ● outperform latches of ■
- scaling of ● limited by hardware cache coherence mechanism
- ×/× suffer more from remote data accesses than × suffers from cache coherence
- ⊕ and ● perform identical for read–only

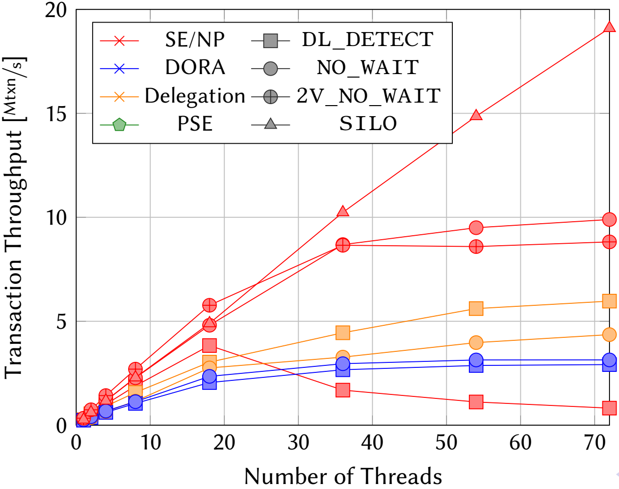
Read–Only Microbenchmark



Observations

- atomics of ● outperform latches of ■
- scaling of ● limited by hardware cache coherence mechanism
- ×/× suffer more from remote data accesses than × suffers from cache coherence
- ⊕ and ● perform identical for read–only

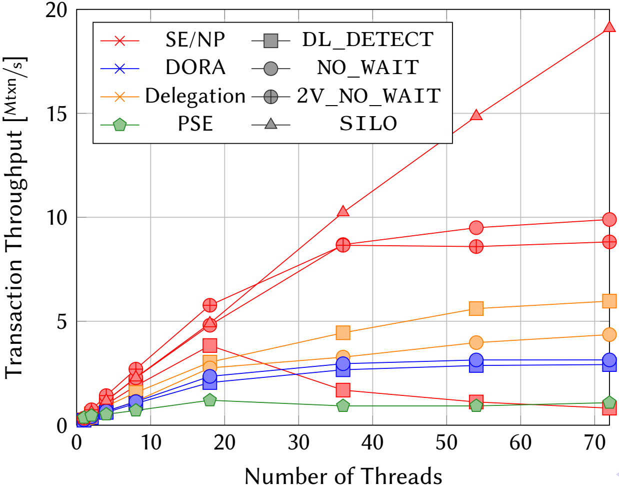
Read–Only Microbenchmark



Observations

- scaling of ● limited by hardware cache coherence mechanism
- ×/× suffer more from remote data accesses than × suffers from cache coherence
- ⊕ and ● perform identical for read–only
- ▲ behaves identical for × and × for read–only

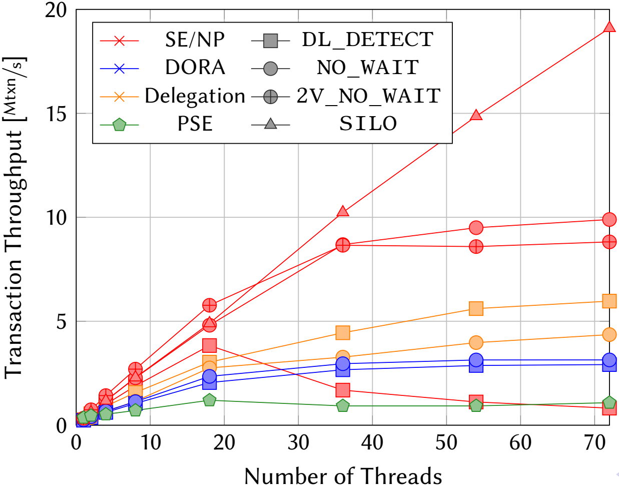
Read–Only Microbenchmark



Observations

- scaling of ● limited by hardware cache coherence mechanism
- ×/× suffer more from remote data accesses than × suffers from cache coherence
- ⊕ and ● perform identical for read–only
- ▲ behaves identical for × and × for read–only

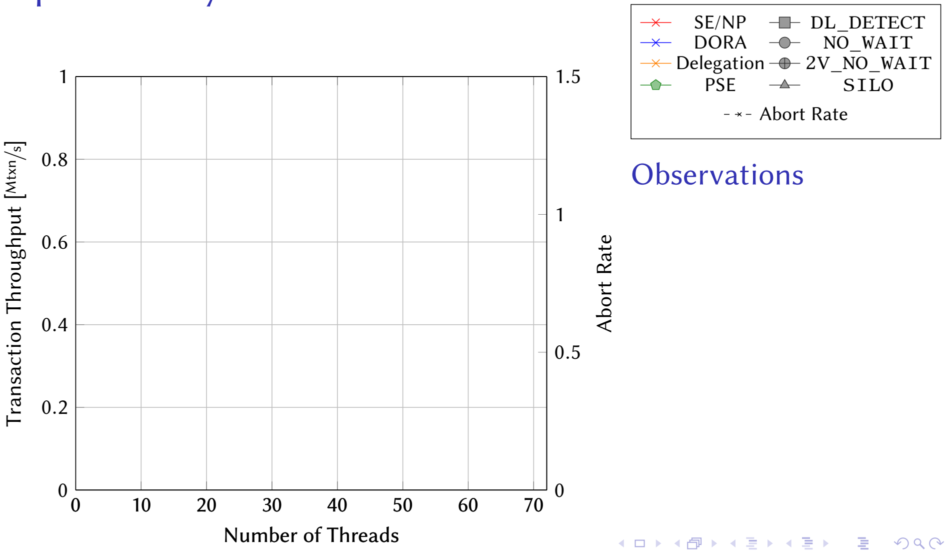
Read–Only Microbenchmark



Observations

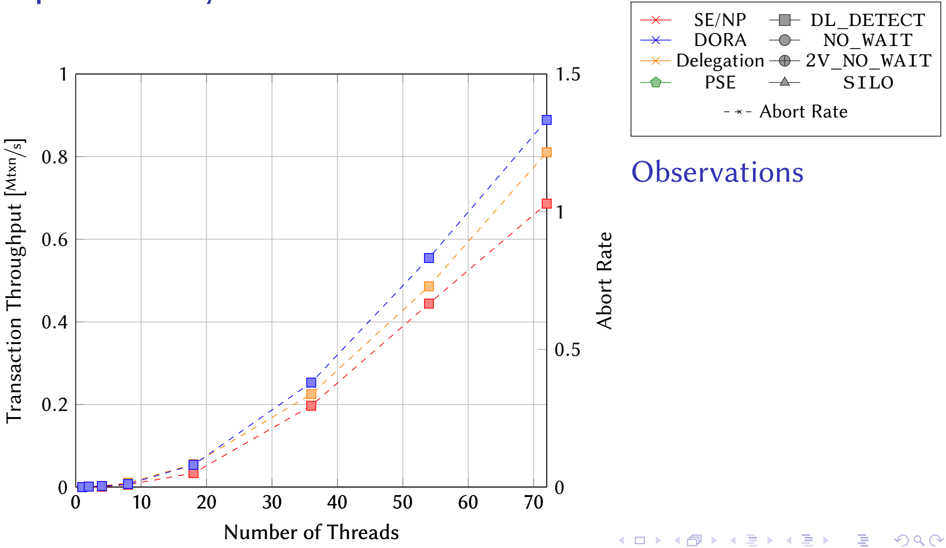
- ×/× suffer more from remote data accesses than × suffers from cache coherence
- ⊕ and ● perform identical for read–only
- ▲ behaves identical for × and × for read–only
- coarse–grained partition locking of ⬟ does not scale due to multi–site workload

Update–Only Microbenchmark



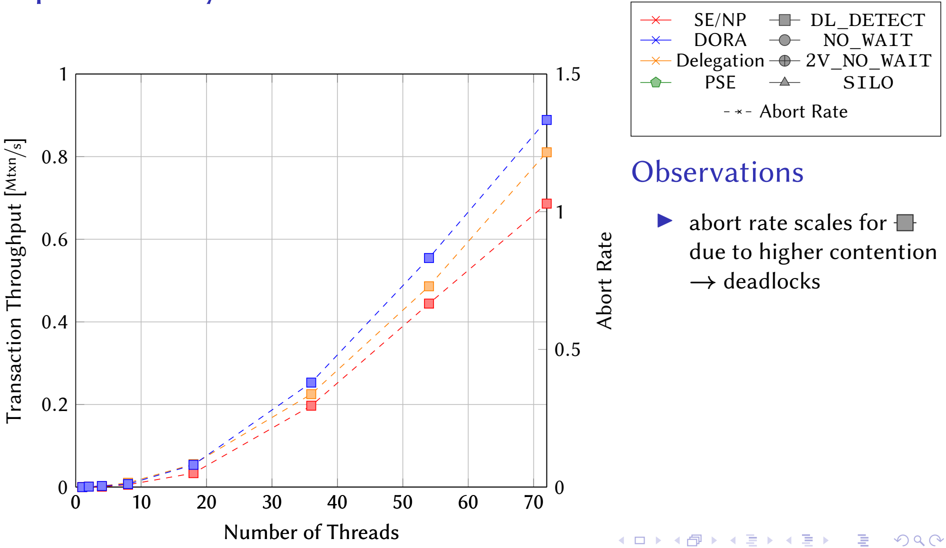
Observations

Update–Only Microbenchmark



Observations

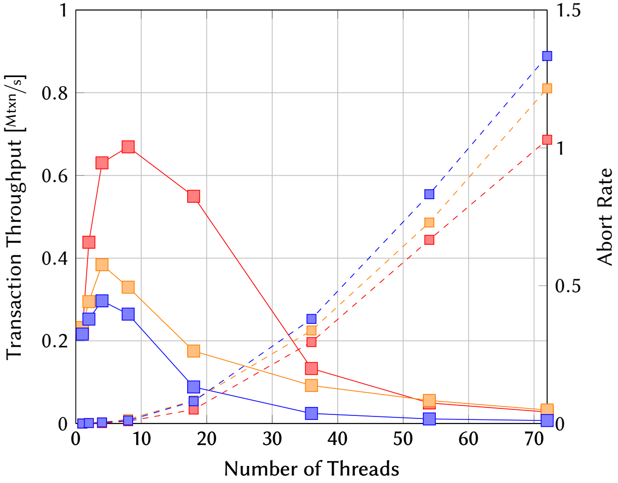
Update–Only Microbenchmark



Observations

- abort rate scales for ■ due to higher contention → deadlocks

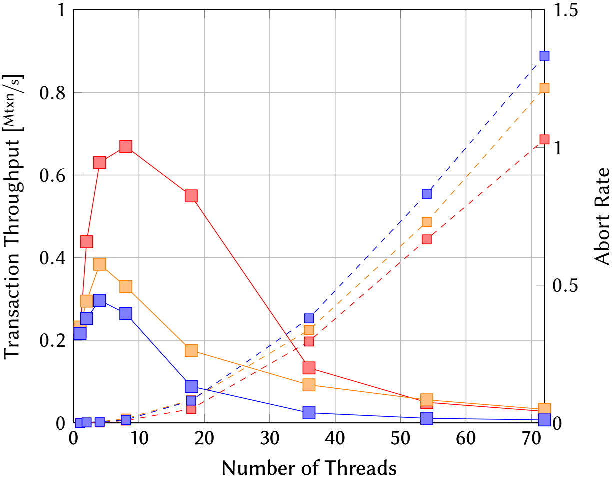
Update–Only Microbenchmark



Observations

- abort rate scales for ■ due to higher contention → deadlocks

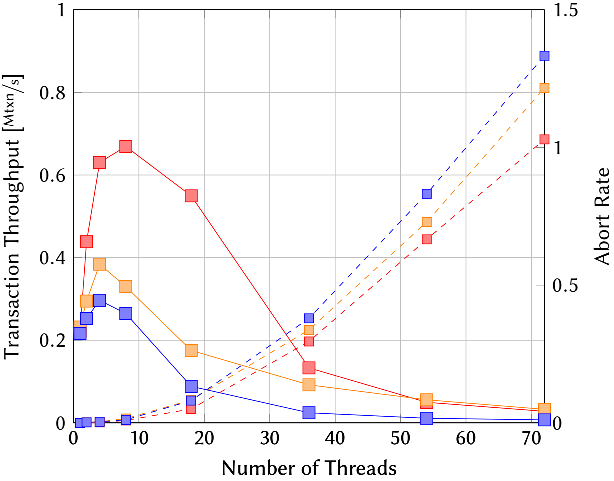
Update–Only Microbenchmark



Observations

- abort rate scales for ■ due to higher contention → deadlocks
- [Mtxn/s] suffers from aborts and lock thrashing

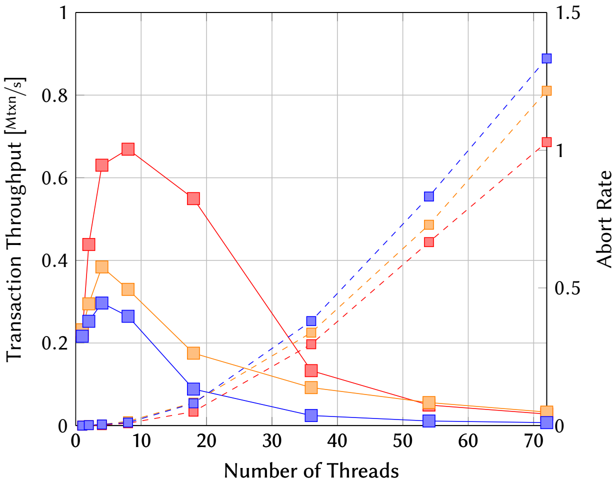
Update–Only Microbenchmark

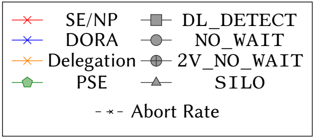


Observations

- abort rate scales for ■ due to higher contention → deadlocks
- $[^{\text{Mtxn}}/_{\text{s}}]$ suffers from aborts and lock thrashing
- ×/× suffer more from remote data access overhead

Update–Only Microbenchmark

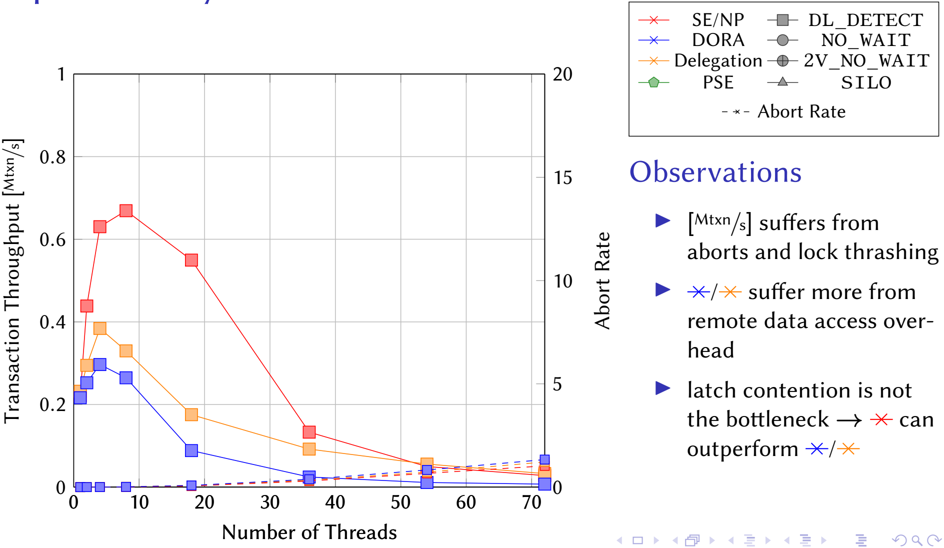




Observations

- [Mtxn/s] suffers from aborts and lock thrashing
- ×/× suffer more from remote data access overhead
- latch contention is not the bottleneck → × can outperform ×/×

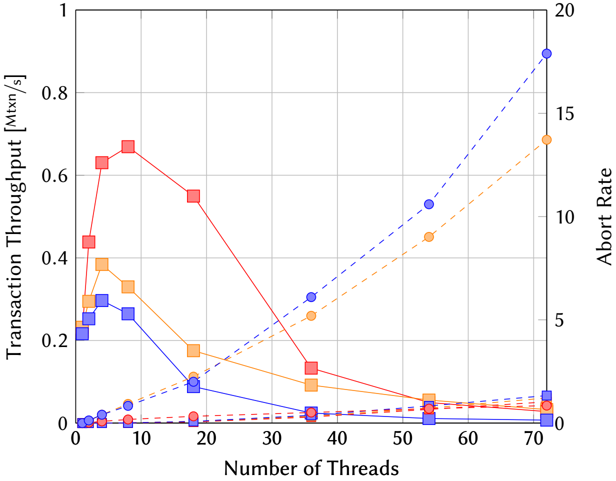
Update–Only Microbenchmark



Observations

- [Mtxn/s] suffers from aborts and lock thrashing
- ×/× suffer more from remote data access overhead
- latch contention is not the bottleneck → × can outperform ×/×

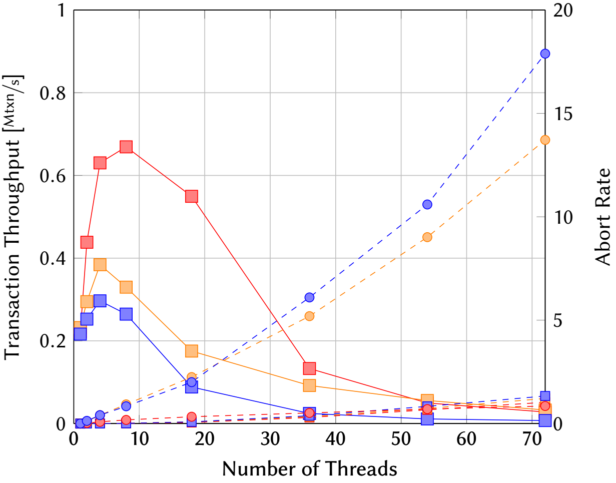
Update–Only Microbenchmark



Observations

- $[^{Mtxn}/_s]$ suffers from aborts and lock thrashing
- ×/× suffer more from remote data access overhead
- latch contention is not the bottleneck → × can outperform ×/×

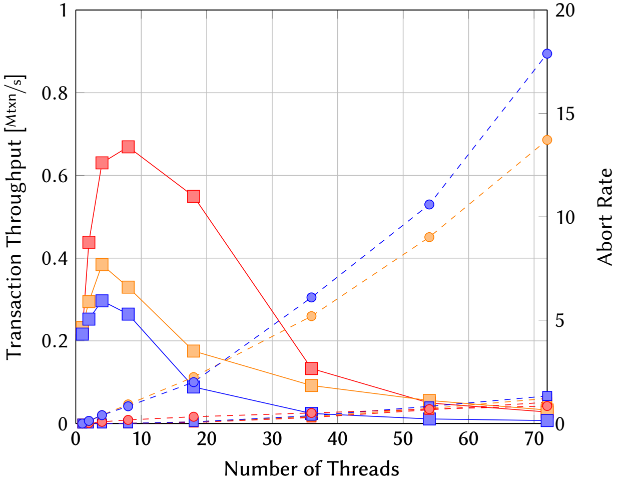
Update–Only Microbenchmark



Observations

- ×/× suffer more from remote data access overhead
- latch contention is not the bottleneck → × can outperform ×/×
- lock thrashing does not cause many aborts for ● with × for few threads

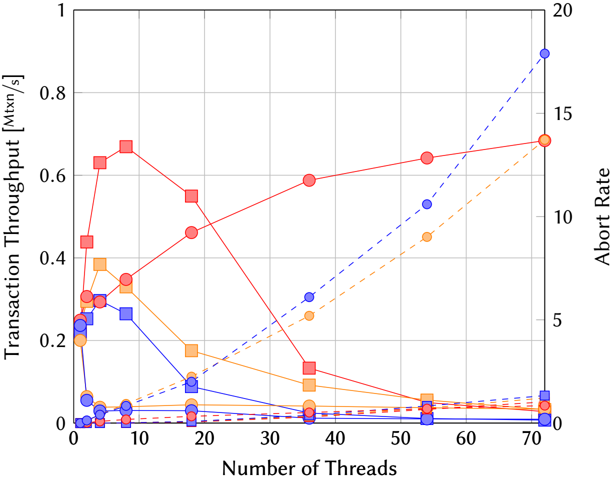
Update–Only Microbenchmark



Observations

- lock thrashing does not cause many aborts for —●— with × for few threads
- lock thrashing caused by long commit latencies caused by overloaded (hot) partitions causes many aborts for ×/×

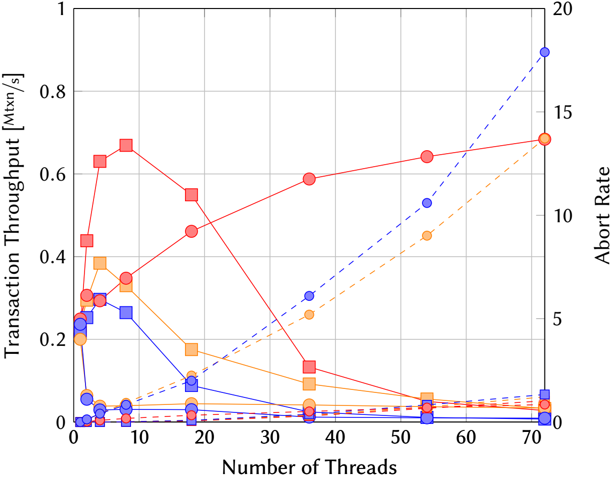
Update–Only Microbenchmark



Observations

- lock thrashing does not cause many aborts for ● with × for few threads
- lock thrashing caused by long commit latencies caused by overloaded (hot) partitions causes many aborts for ×/×

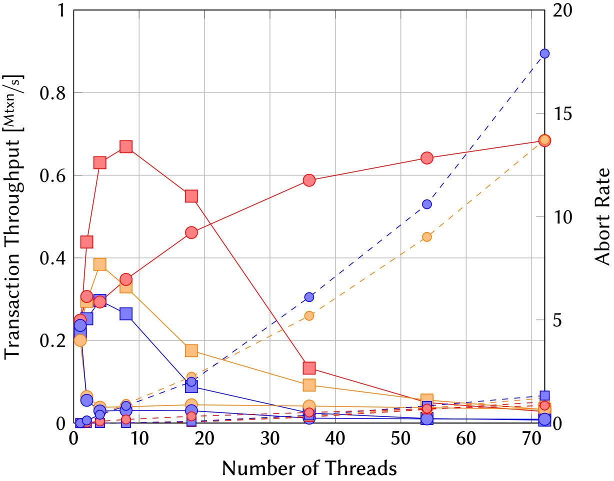
Update–Only Microbenchmark



Observations

- lock thrashing does not cause many aborts for ● with × for few threads
- lock thrashing caused by long commit latencies caused by overloaded (hot) partitions causes many aborts for ×/×
- the aborts are the major bottleneck for ●

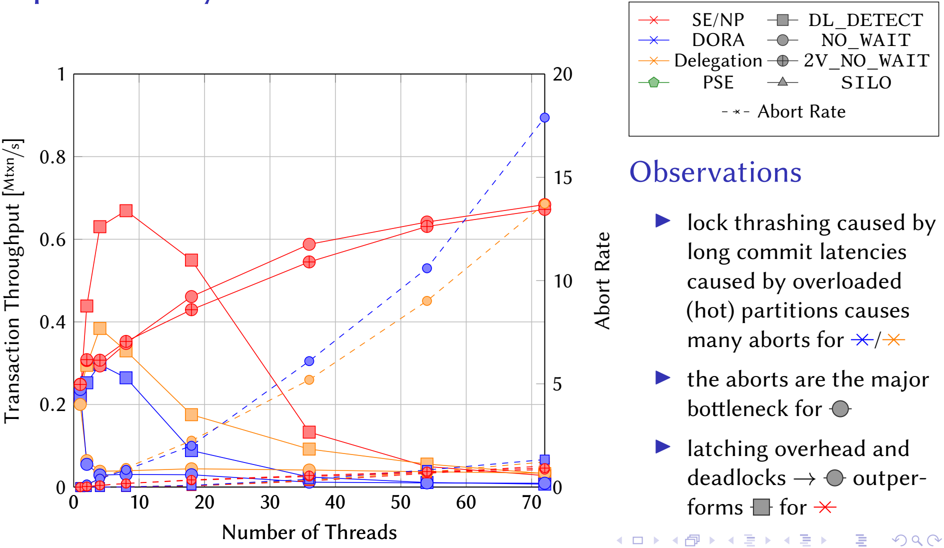
Update–Only Microbenchmark



Observations

- lock thrashing caused by long commit latencies caused by overloaded (hot) partitions causes many aborts for ✕/✕
- the aborts are the major bottleneck for -●-
- latching overhead and deadlocks → -●- outperforms -■- for ✕

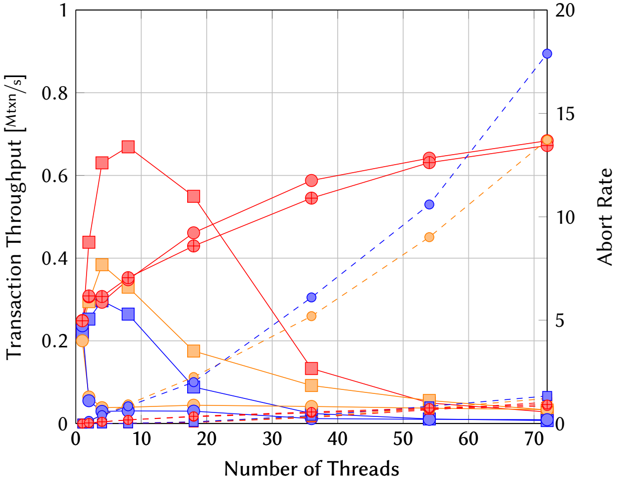
Update–Only Microbenchmark



Observations

- lock thrashing caused by long commit latencies caused by overloaded (hot) partitions causes many aborts for ×/×
- the aborts are the major bottleneck for ●
- latching overhead and deadlocks → ● outperforms ■ for ×

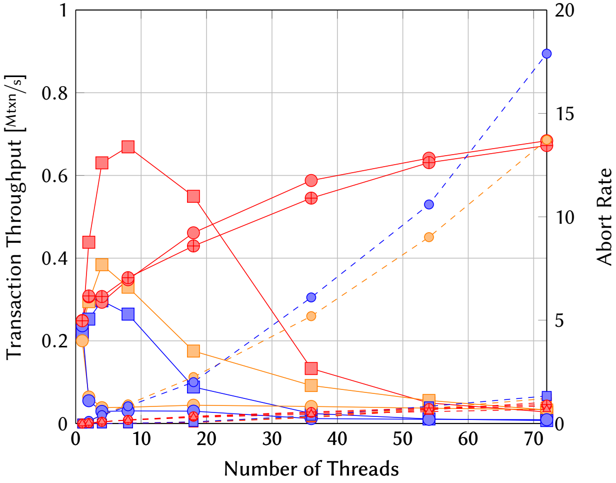
Update–Only Microbenchmark



Observations

- the aborts are the major bottleneck for ●
- latching overhead and deadlocks → ● outperforms ■ for ×
- for update–only ● and ⊕ behave identical

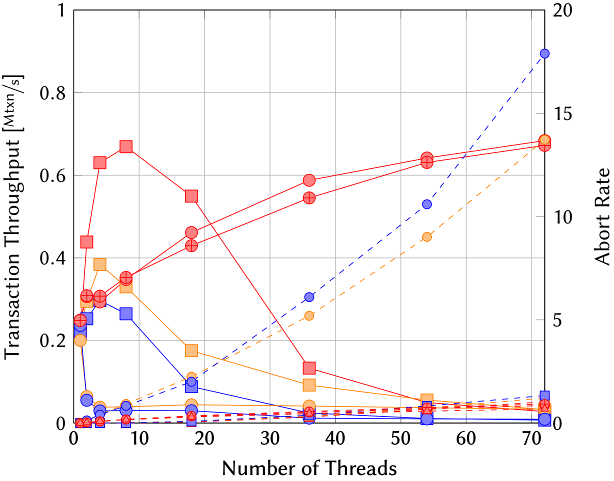
Update–Only Microbenchmark



Observations

- the aborts are the major bottleneck for ●
- latching overhead and deadlocks → ● outperforms ■ for ×
- for update–only ● and ⊕ behave identical

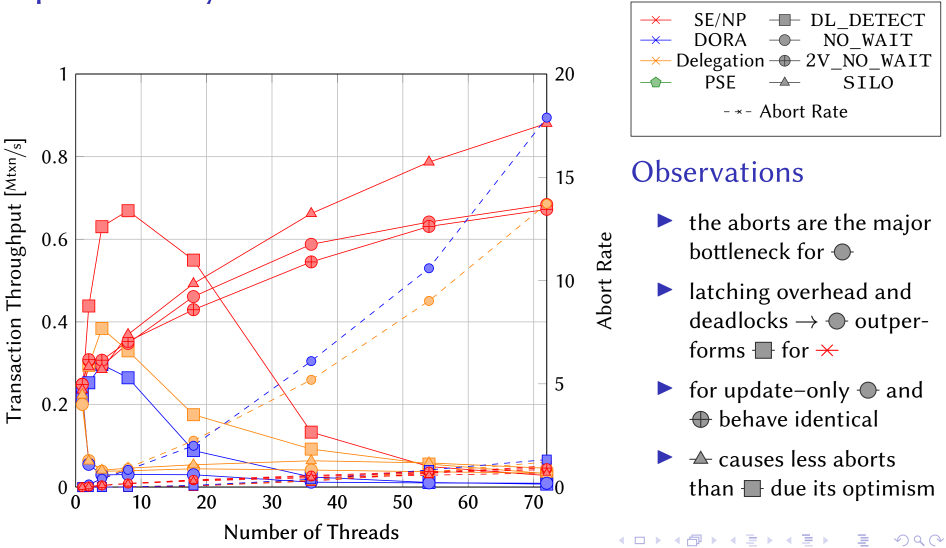
Update–Only Microbenchmark



Observations

- the aborts are the major bottleneck for ●
- latching overhead and deadlocks → ● outperforms ■ for ×
- for update–only ● and ⊕ behave identical
- ▲ causes less aborts than ■ due its optimism

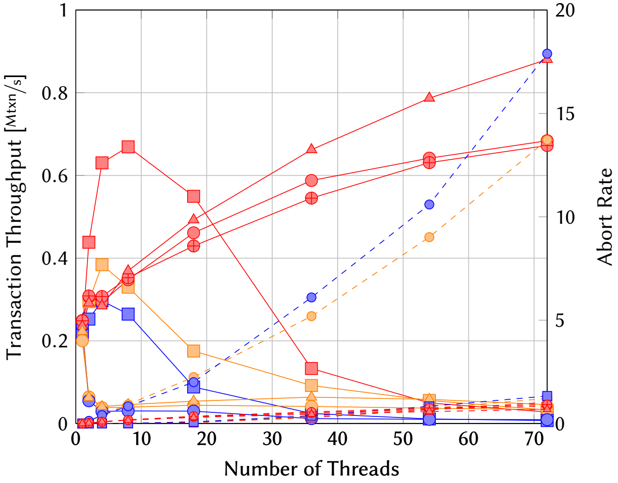
Update–Only Microbenchmark



Observations

- the aborts are the major bottleneck for ○
- latching overhead and deadlocks → ○ outperforms □ for ×
- for update–only ○ and ⊕ behave identical
- △ causes less aborts than □ due its optimism

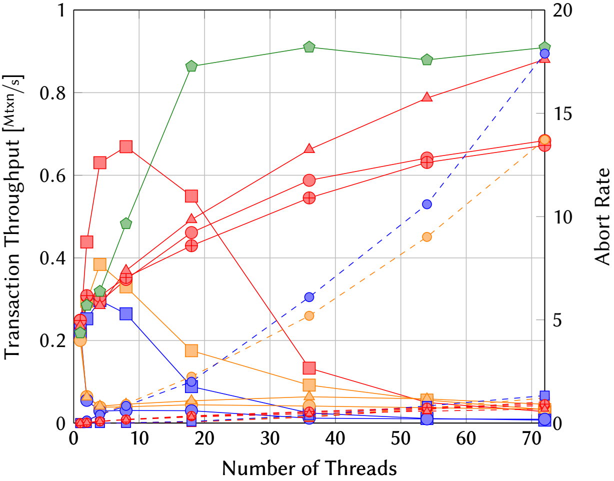
Update–Only Microbenchmark



Observations

- for update–only ● and ⊕ behave identical
- ▲ causes less aborts than ■ due its optimism
- long commit latencies of × cause high update contention and therefore many aborts (low $[^{Mtxn}/_{s}]$) for ▲

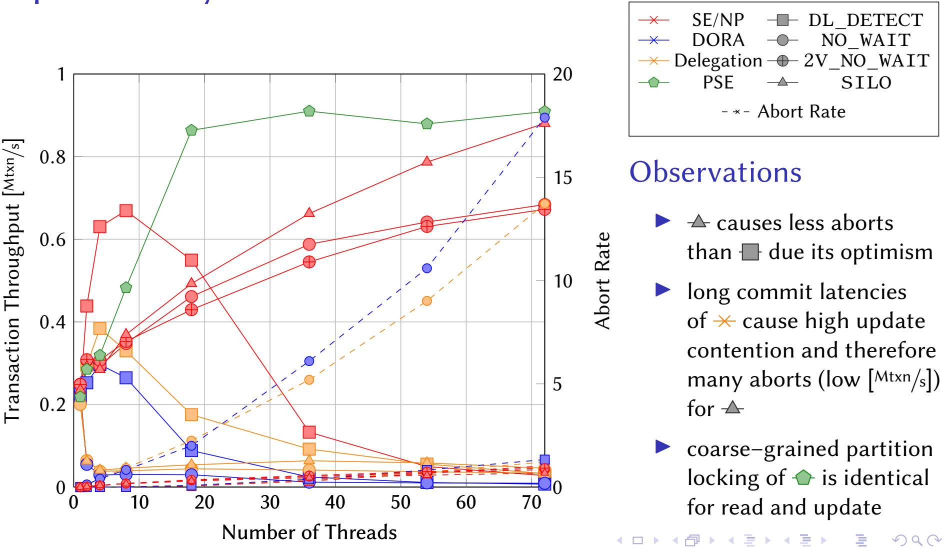
Update–Only Microbenchmark



Observations

- for update–only ● and ⊕ behave identical
- ▲ causes less aborts than ■ due its optimism
- long commit latencies of × cause high update contention and therefore many aborts (low [Mtxn/s]) for ▲

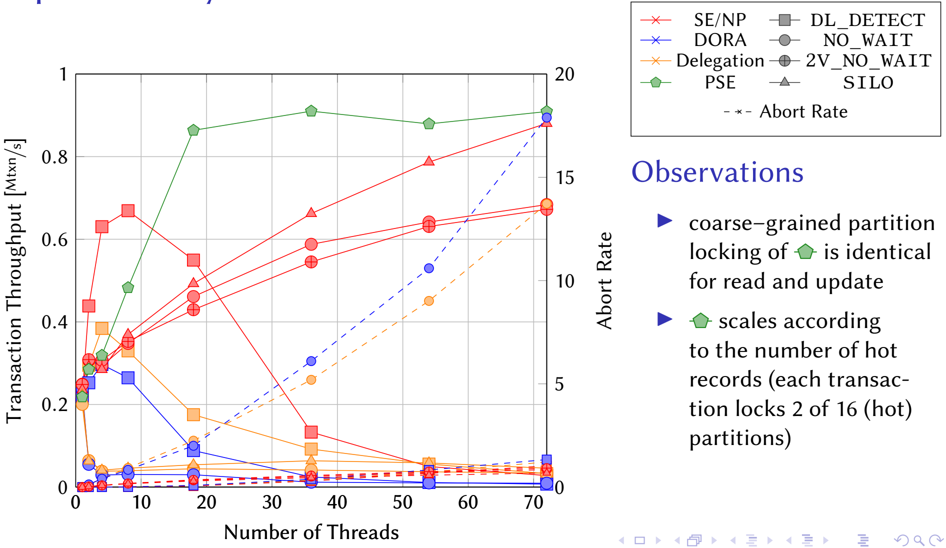
Update–Only Microbenchmark



Observations

- ▲ causes less aborts than ■ due its optimism
- long commit latencies of × cause high update contention and therefore many aborts (low $[^{\text{Mtxn}}/_{\text{s}}]$) for ▲
- coarse–grained partition locking of ⬟ is identical for read and update

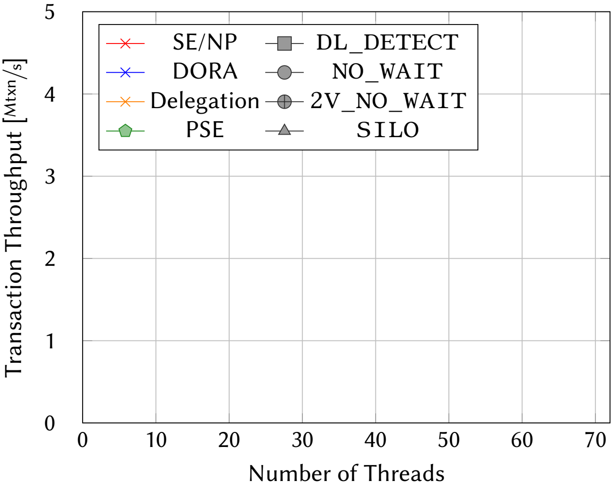
Update–Only Microbenchmark



Observations

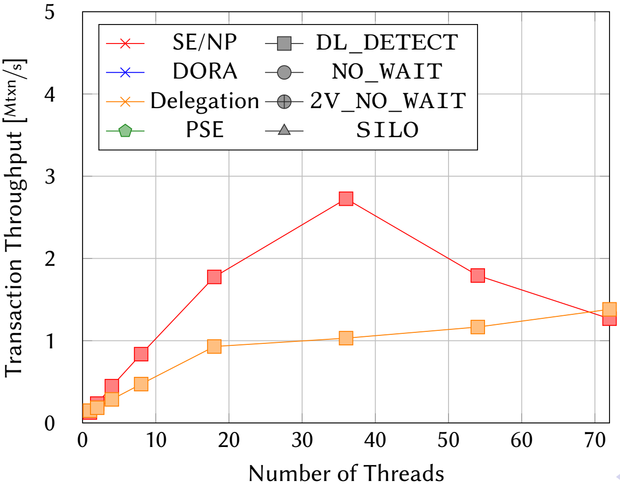
- coarse–grained partition locking of ⬠ is identical for read and update
- ⬠ scales according to the number of hot records (each transaction locks 2 of 16 (hot) partitions)

Read–Only YCSB ($\Theta = 0.8$)



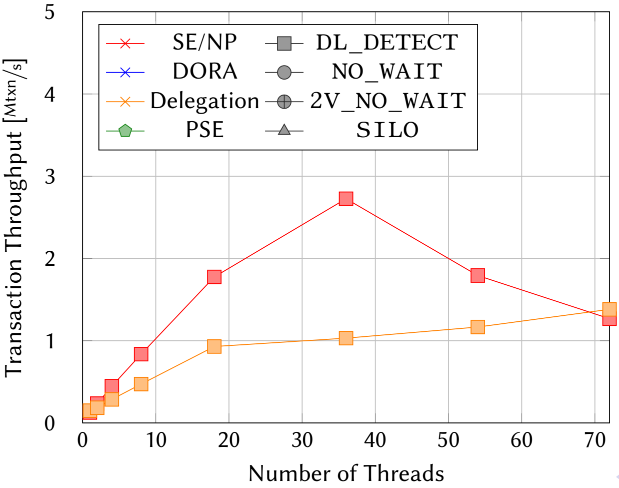
Observations

Read–Only YCSB ($\Theta = 0.8$)



Observations

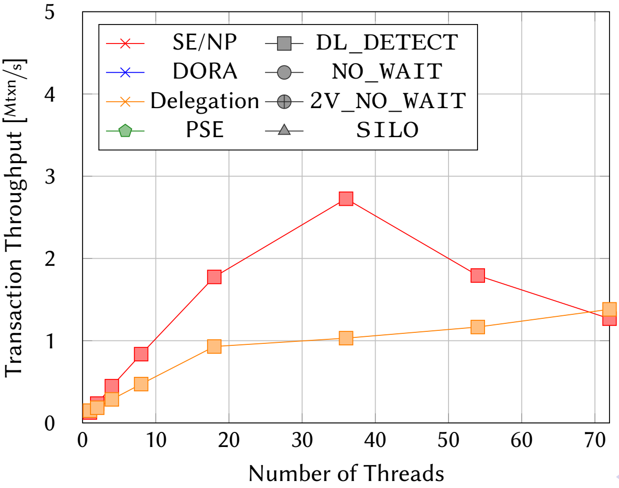
Read–Only YCSB ($\Theta = 0.8$)



Observations

- × scales well with ▪ until the latch contention becomes a bottleneck

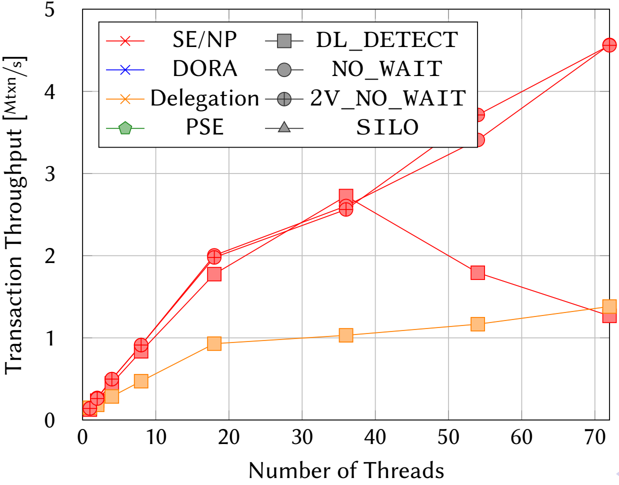
Read–Only YCSB ($\Theta = 0.8$)



Observations

- ▶ × scales well with ■ until the latch contention becomes a bottleneck
- ▶ × (and ×) does not scale well due to partition–unfriendly Zipfian access distribution

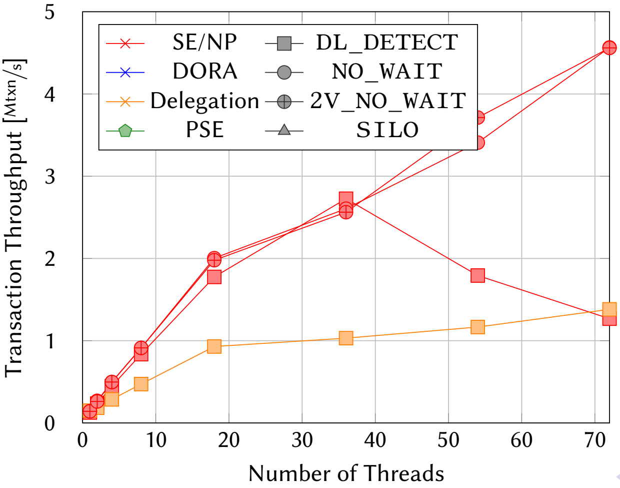
Read–Only YCSB ($\Theta = 0.8$)



Observations

- × scales well with ■ until the latch contention becomes a bottleneck
- × (and ×) does not scale well due to partition–unfriendly Zipfian access distribution

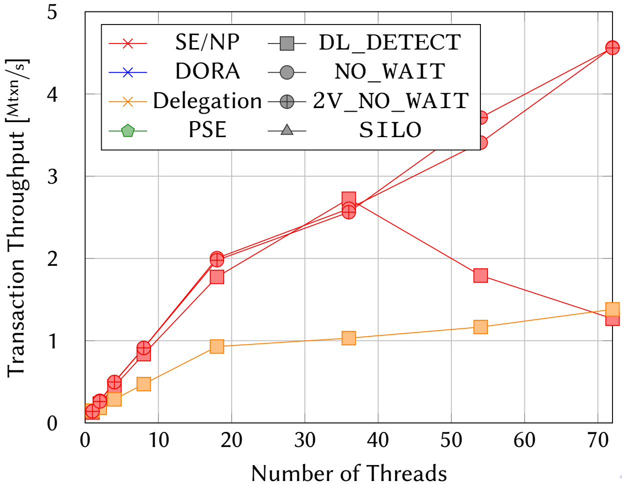
Read–Only YCSB ($\Theta = 0.8$)



Observations

- × scales well with ■ until the latch contention becomes a bottleneck
- × (and ×) does not scale well due to partition–unfriendly Zipfian access distribution
- atomics of ● scale better than latches of ■

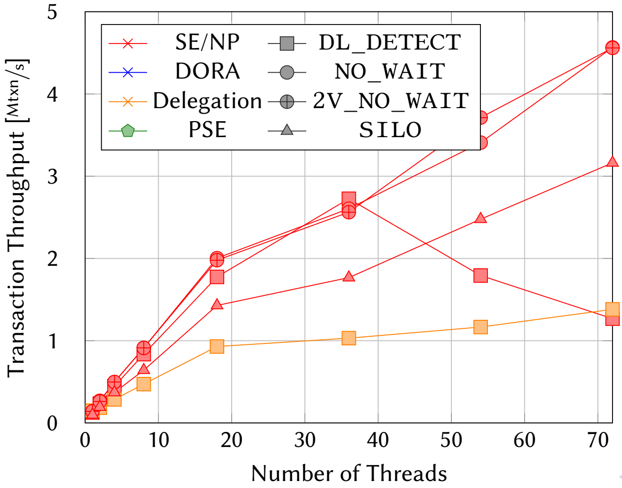
Read–Only YCSB ($\Theta = 0.8$)



Observations

- × scales well with ■ until the latch contention becomes a bottleneck
- × (and ×) does not scale well due to partition–unfriendly Zipfian access distribution
- atomics of ● scale better than latches of ■
- ⊕ and ● perform identical for read–only

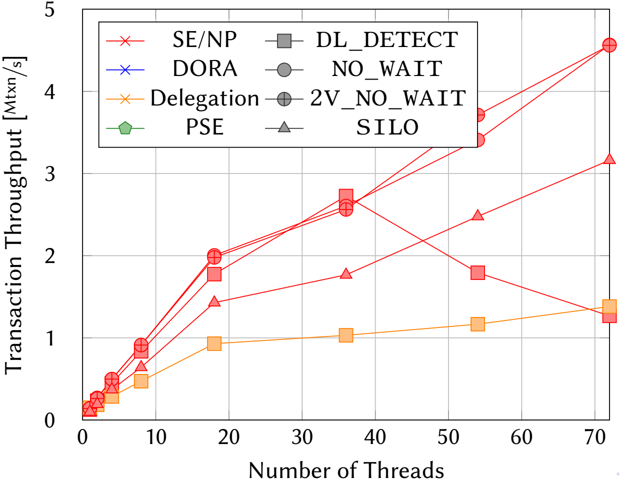
Read–Only YCSB ($\Theta = 0.8$)



Observations

- × scales well with ■ until the latch contention becomes a bottleneck
- × (and ×) does not scale well due to partition–unfriendly Zipfian access distribution
- atomics of ● scale better than latches of ■
- ⊕ and ● perform identical for read–only

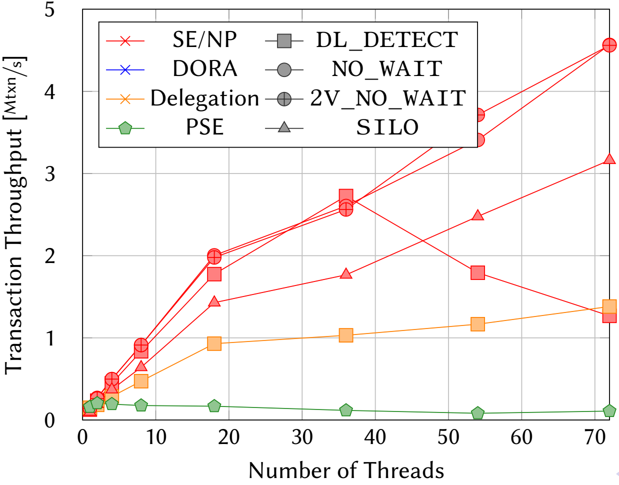
Read–Only YCSB ($\Theta = 0.8$)



Observations

- × (and ×) does not scale well due to partition–unfriendly Zipfian access distribution
- atomics of ● scale better than latches of ■
- ⊕ and ● perform identical for read–only
- △ lags behind ⊕ due to the overhead of copying read (large) records for validation

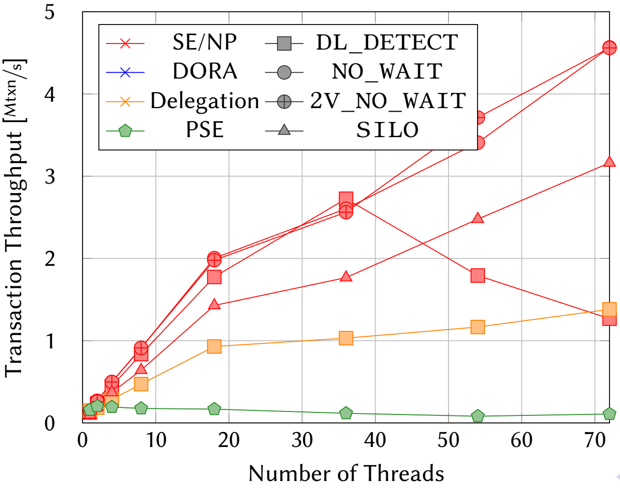
Read–Only YCSB ($\Theta = 0.8$)



Observations

- × (and ×) does not scale well due to partition–unfriendly Zipfian access distribution
- atomics of ● scale better than latches of ■
- ⊕ and ● perform identical for read–only
- ▲ lags behind ⊕ due to the overhead of copying read (large) records for validation

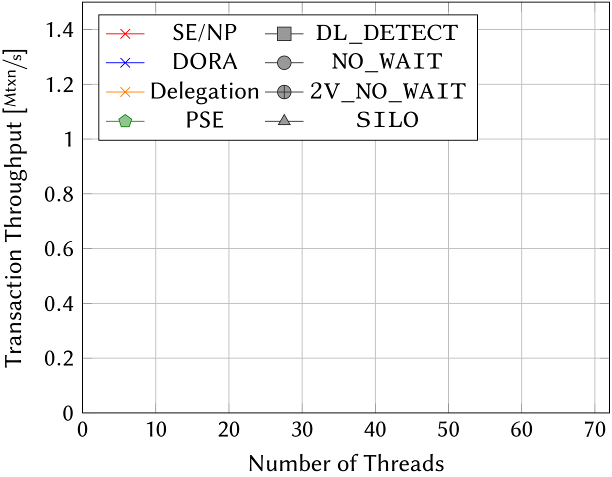
Read–Only YCSB ($\Theta = 0.8$)



Observations

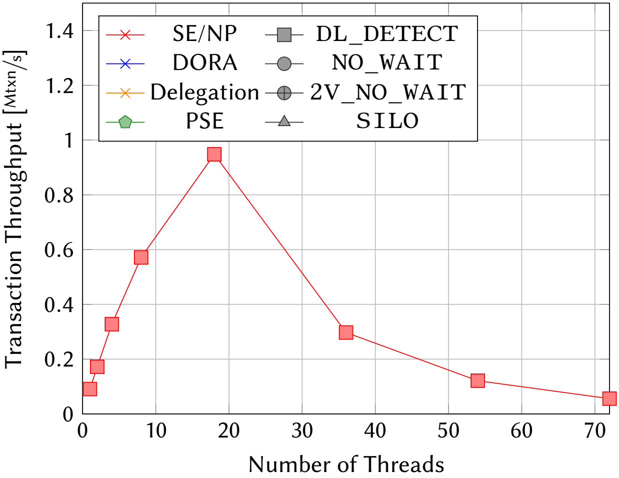
- atomics of ● scale better than latches of ■
- ⊕ and ● perform identical for read–only
- ▲ lags behind ⊕ due to the overhead of copying read (large) records for validation
- coarse–grained partition locking of ⬟ is identical for read and update

Update–Only YCSB ($\Theta = 0.8$)



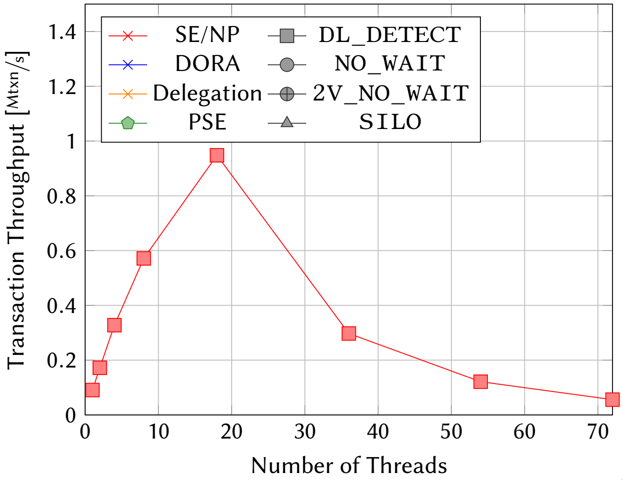
Observations

Update–Only YCSB ($\Theta = 0.8$)



Observations

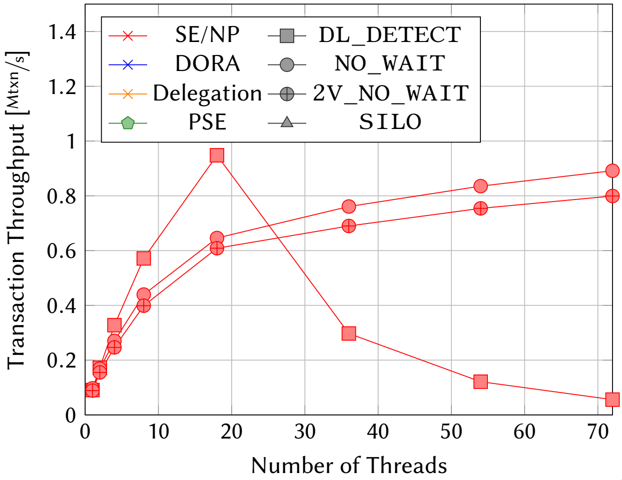
Update–Only YCSB ($\Theta = 0.8$)



Observations

- ■ suffers from deadlocks for many threads

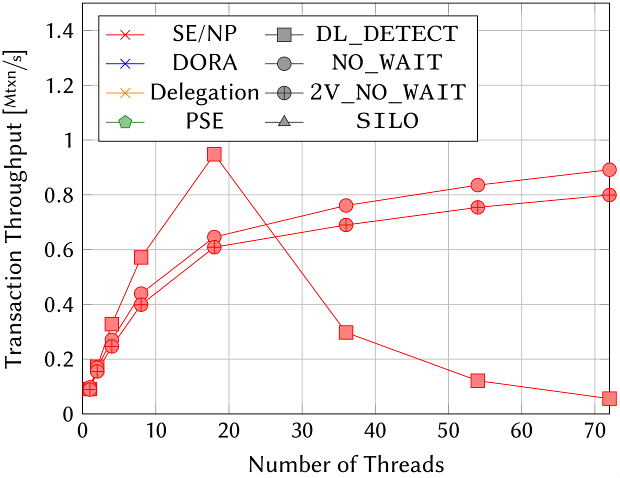
Update–Only YCSB ($\Theta = 0.8$)



Observations

- ■ suffers from deadlocks for many threads

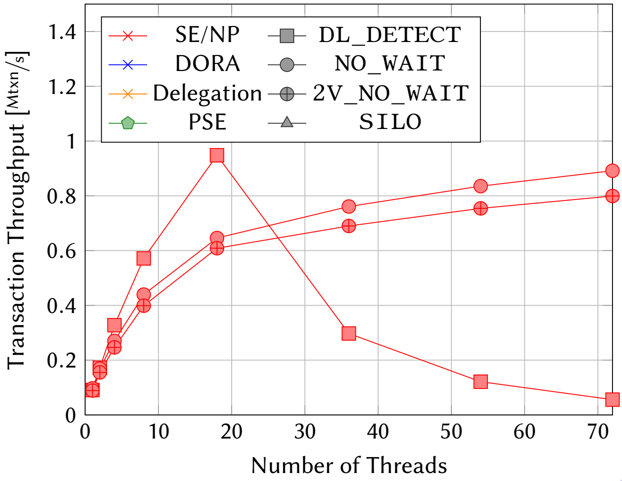
Update–Only YCSB ($\Theta = 0.8$)



Observations

- ■ suffers from deadlocks for many threads
- lock thrashing (aborts for ●) is not a bottleneck due to lower contention

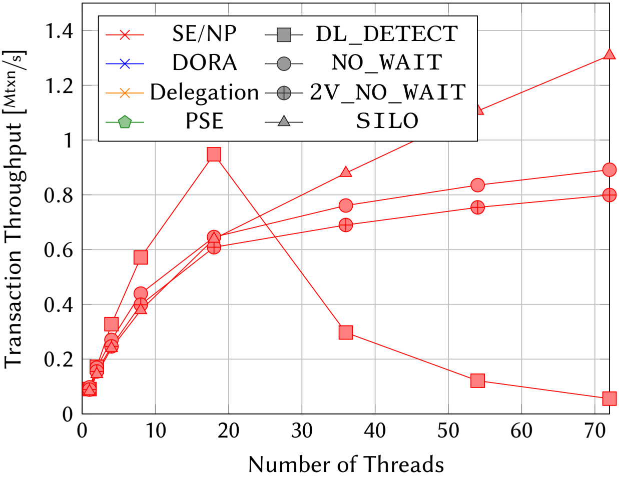
Update–Only YCSB ($\Theta = 0.8$)



Observations

- ■ suffers from deadlocks for many threads
- lock thrashing (aborts for ●) is not a bottleneck due to lower contention
- ⊕ and ● perform identical for update–only

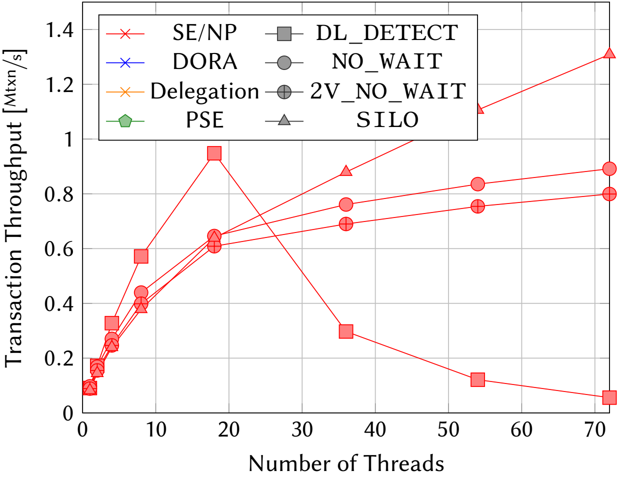
Update–Only YCSB ($\Theta = 0.8$)



Observations

- ■ suffers from deadlocks for many threads
- lock thrashing (aborts for ●) is not a bottleneck due to lower contention
- ⊕ and ● perform identical for update–only

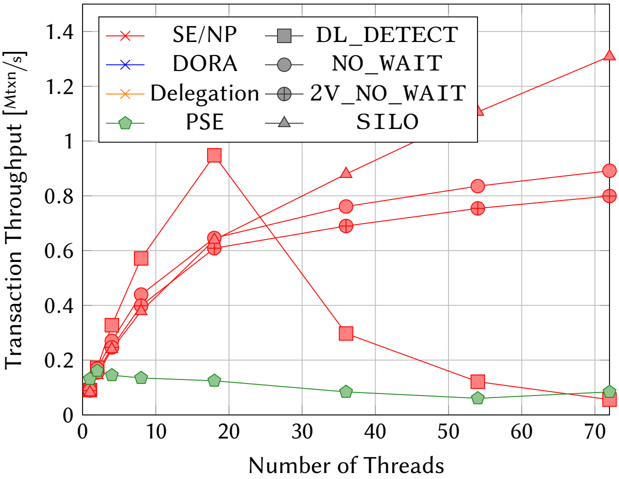
Update–Only YCSB ($\Theta = 0.8$)



Observations

- ■ suffers from deadlocks for many threads
- lock thrashing (aborts for ●) is not a bottleneck due to lower contention
- ⊕ and ● perform identical for update–only
- ▲ causes less aborts than ● due its optimism → higher [Mtxn/s]

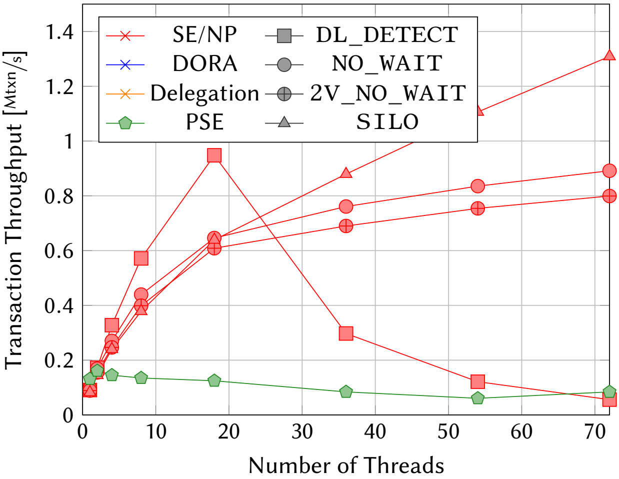
Update–Only YCSB ($\Theta = 0.8$)



Observations

- ■ suffers from deadlocks for many threads
- lock thrashing (aborts for ●) is not a bottleneck due to lower contention
- ⊕ and ● perform identical for update–only
- ▲ causes less aborts than ● due its optimism → higher [Mtxn/s]

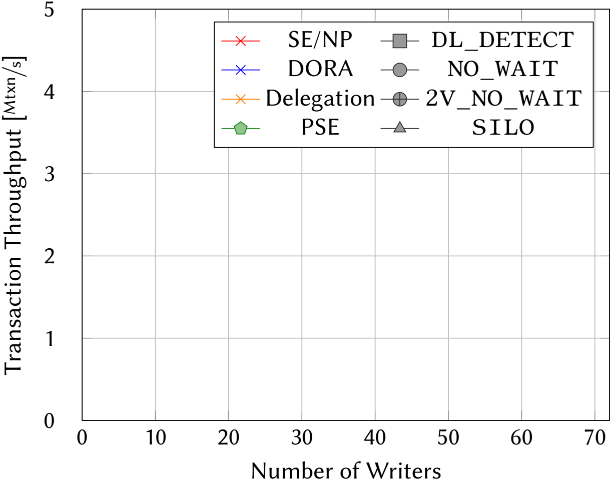
Update–Only YCSB ($\Theta = 0.8$)



Observations

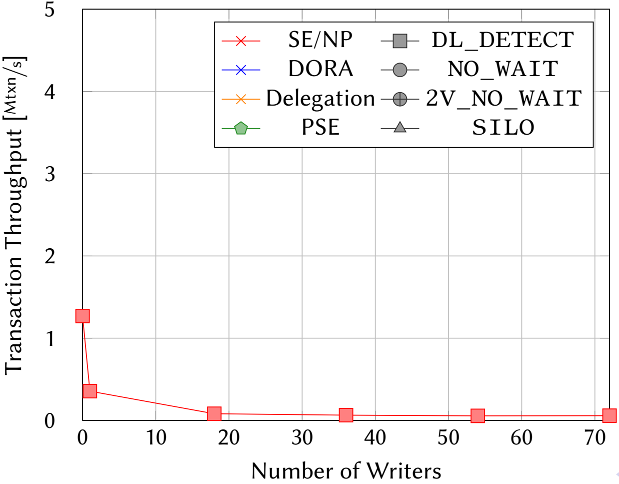
- lock thrashing (aborts for ●) is not a bottleneck due to lower contention
- ⊕ and ● perform identical for update–only
- ▲ causes less aborts than ● due its optimism → higher [Mtxn/s]
- ⬟ (and ✕/✕) does not scale well due to partition–unfriendly Zipfian access distribution

Mixed YCSB ($\Theta = 0.8$, 72 Threads)



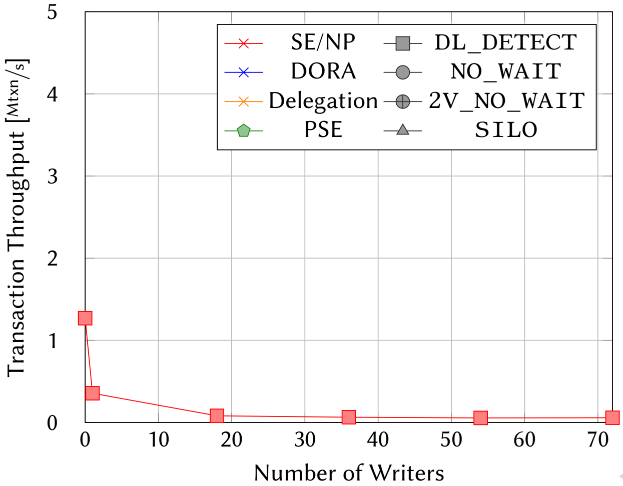
Observations

Mixed YCSB ($\Theta = 0.8$, 72 Threads)



Observations

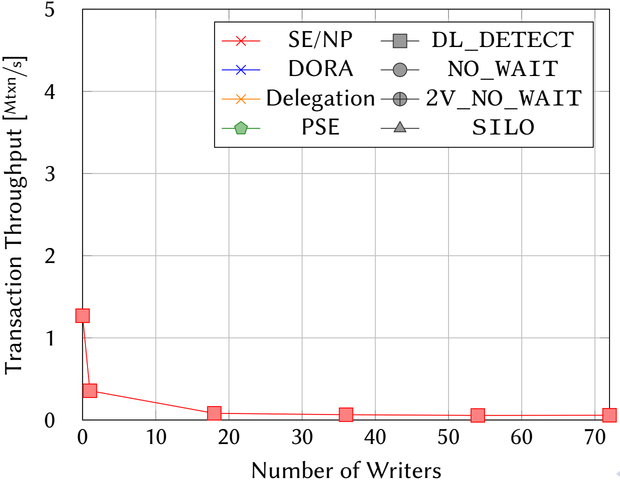
Mixed YCSB ($\Theta = 0.8$, 72 Threads)



Observations

- ■ suffers from latch contention for 72 reading threads

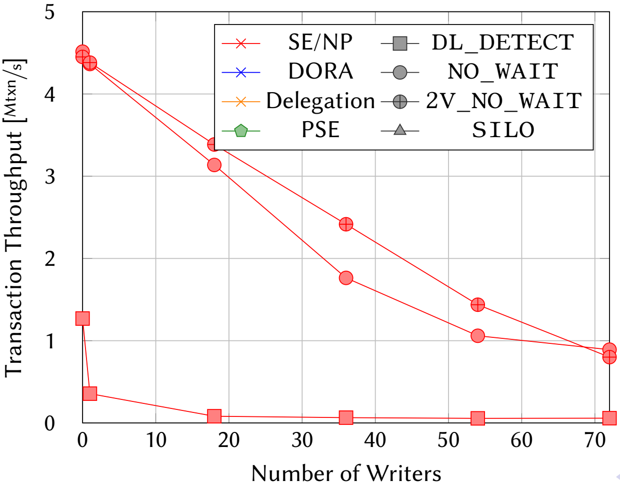
Mixed YCSB ($\Theta = 0.8$, 72 Threads)



Observations

- ■ suffers from latch contention for 72 reading threads
- ■ suffers from deadlocks for writing threads

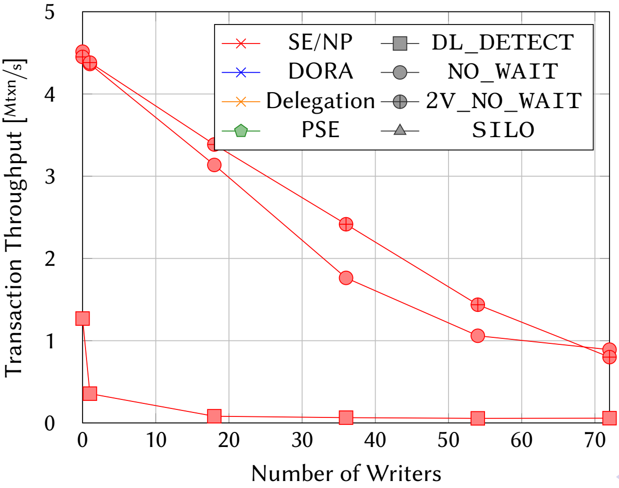
Mixed YCSB ($\Theta = 0.8$, 72 Threads)



Observations

- ■ suffers from latch contention for 72 reading threads
- ■ suffers from deadlocks for writing threads

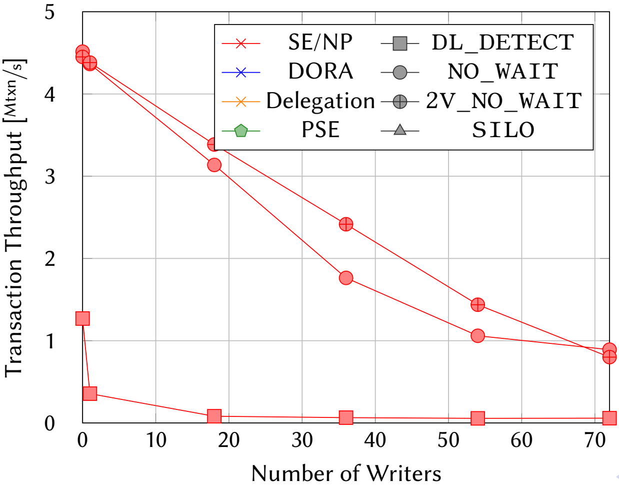
Mixed YCSB ($\Theta = 0.8$, 72 Threads)



Observations

- ■ suffers from latch contention for 72 reading threads
- ■ suffers from deadlocks for writing threads
- atomics of ● scale better than latches of ■

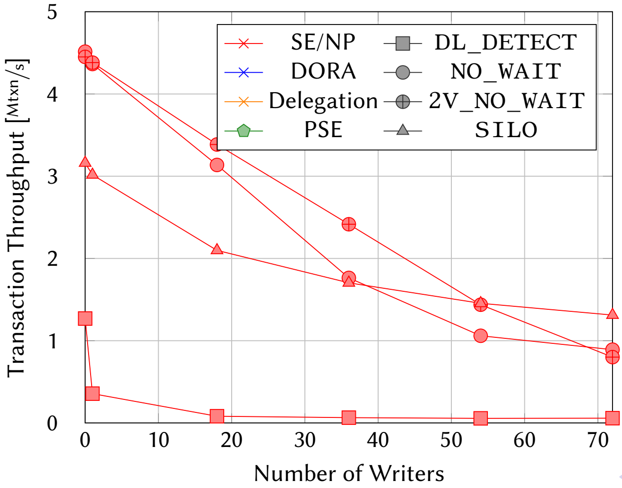
Mixed YCSB ($\Theta = 0.8$, 72 Threads)



Observations

- ■ suffers from latch contention for 72 reading threads
- ■ suffers from deadlocks for writing threads
- atomics of ● scale better than latches of ■
- multi–versioning of ⊕ improves concurrency for mixed workloads

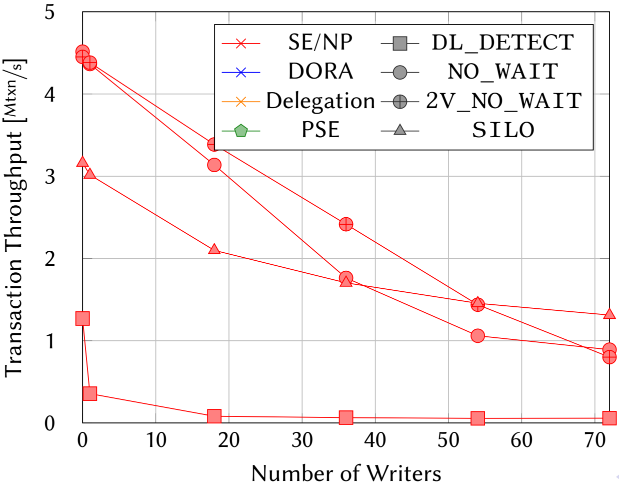
Mixed YCSB ($\Theta = 0.8$, 72 Threads)



Observations

- ■ suffers from latch contention for 72 reading threads
- ■ suffers from deadlocks for writing threads
- atomics of ● scale better than latches of ■
- multi–versioning of ⊕ improves concurrency for mixed workloads

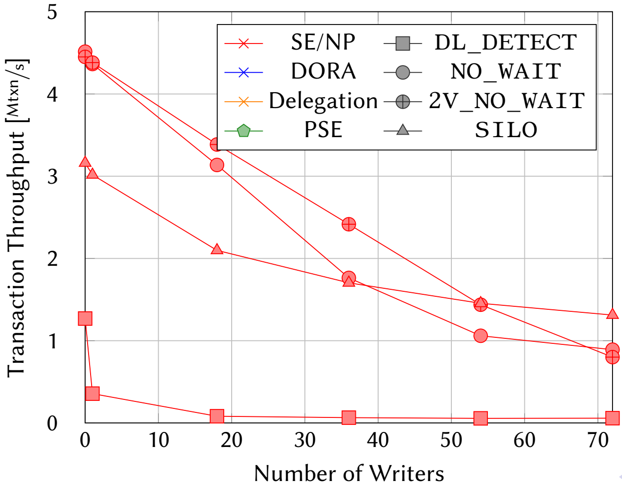
Mixed YCSB ($\Theta = 0.8$, 72 Threads)



Observations

- ■ suffers from deadlocks for writing threads
- atomics of ● scale better than latches of ■
- multi–versioning of ⊕ improves concurrency for mixed workloads
- ▲ lags behind ⊕ due to the overhead of copying read (large) records for validation

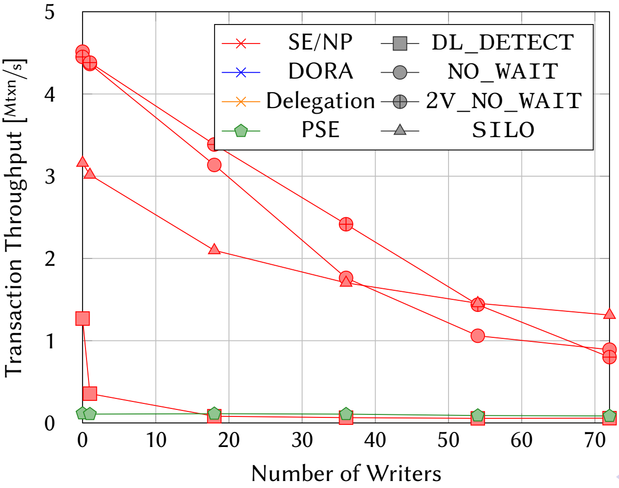
Mixed YCSB (Θ = 0.8, 72 Threads)



Observations

- atomics of ● scale better than latches of ■
- multi–versioning of ⊕ improves concurrency for mixed workloads
- ▲ lags behind ⊕ due to the overhead of copying read (large) records for validation
- ▲ causes less aborts than ⊕ due its optimism for many writers

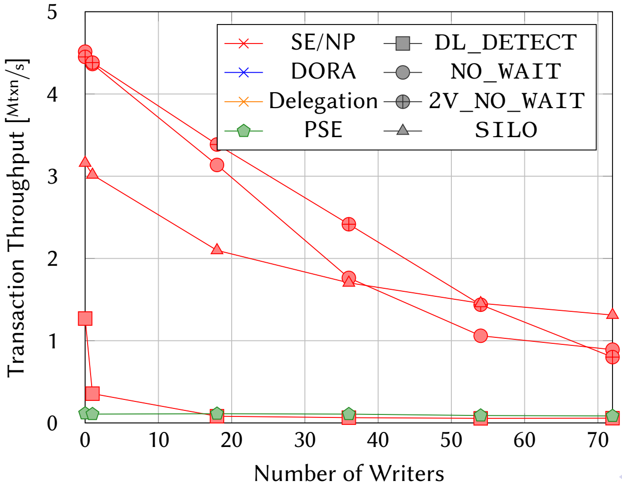
Mixed YCSB ($\Theta = 0.8$, 72 Threads)



Observations

- atomics of ● scale better than latches of ■
- multi–versioning of ⊕ improves concurrency for mixed workloads
- ▲ lags behind ⊕ due to the overhead of copying read (large) records for validation
- ▲ causes less aborts than ⊕ due its optimism for many writers

Mixed YCSB ($\Theta = 0.8$, 72 Threads)



Observations

- △ lags behind ⊕ due to the overhead of copying read (large) records for validation
- △ causes less aborts than ⊕ due its optimism for many writers
- ⬠ (and ✕/✕) does not scale well due to partition–unfriendly Zipfian access distribution

Conclusion I

- optimistic concurrency control scales better than pessimistic CC for most workloads
- optimistic CC suffers from large record sizes
- atomic operations scale better than latches
- partitioning makes latches scalable
- 2PL does not scale for mixed workloads
- partitioning DB architectures perform bad under partition–unfriendly workloads
- partitioning DB architectures perform bad under multi–sited transactions

Conclusion II

- the transaction throughput decreases by an order of magnitude for update–only instead of read–only workloads (PSE is insensitive to writes)
  → PSE scales best for update–intensive workloads
- PSE does not scale for read–intensive high–contention workloads with small hot sets

→ None of the architectures or CC protocols outperform the others for any workload!

→ Every architecture and CC protocol performs very bad for some specific workload!

Discussion of the Performance Evaluation

- read–only and update–only workload are not appropriate to evaluate concurrency control algorithms
- partition–unfriendly workloads are not appropriate to evaluate database architectures that use partitioning
- neither the microbenchmark nor YCSB are OLTP benchmarks

→ The authors did not properly analyze the combination of database architecture and concurrency control algorithm for OLTP workloads!

References I

- *Enterprise-Festplatten: 36 High-Performance-Festplatten im Vergleichstest.* Oct. 2, 2013. URL: http://www.tomshardware.de/enterprise-hdd-sshd,testberichte-241390-6.html (visited on Feb. 8, 2017).
- C. Mohan. “ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes”. Aug. 1990.
- Ippokratis Pandis et al. “Data-Oriented Transaction Execution”. Sept. 2010.

References II

- Igor Pavlov. *Intel Skylake.* URL: http://www.7-cpu.com/cpu/Skylake.html (visited on Jan. 19, 2017).
- Danica Porobic et al. “OLTP on Hardware Islands”. July 2012.
- *Seagates Speicherriese ist schnell und sehr sparsam.* Aug. 16, 2016. URL: https://www.computerbase.de/2016-08/seagate-enterprise-capacity-3.5-hdd-10tb-test/3/#diagramm-zugriffszeiten-lesen-h2benchw-316 (visited on Feb. 8, 2017).
- “Why SSDs Are Awesome - An SSD Primer”. Aug. 2015.

Any Questions?