

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



Section 1

Introduction

Requirements for a DBMS

Requirements for a DBMS

- ▶ Reliability

Requirements for a DBMS

- ▶ Reliability
- ▶ Functionality

Requirements for a DBMS

- ▶ Reliability
 - ▶ ACID Transactions
- ▶ Functionality

Requirements for a DBMS

- ▶ Reliability
 - ▶ ACID Transactions
 - ▶ high availability
- ▶ Functionality

Requirements for a DBMS

- ▶ Reliability
 - ▶ ACID Transactions
 - ▶ high availability
 - ▶ etc.
- ▶ Functionality

Requirements for a DBMS

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

Requirements for a DBMS

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

Requirements for a DBMS

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

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 performance everything is worth nothing.

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 performance everything is worth nothing.

- ▶ Performance

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 performance everything is worth nothing.

- ▶ Performance
 - ▶ high transaction throughput

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 performance everything is worth nothing.

- ▶ Performance
 - ▶ high transaction throughput
 - ▶ low latency

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 performance everything is worth nothing.

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

Some Implications of those Requirements

Some Implications of those Requirements

- ▶ work purely in-memory when the working set completely fits in main memory

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

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)

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)

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)

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!

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

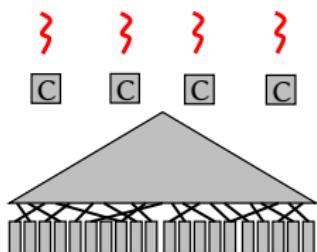
Some Implications of those Requirements

- ▶ 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!**

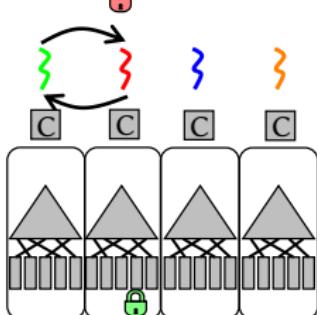
Section 2

Database Architectures

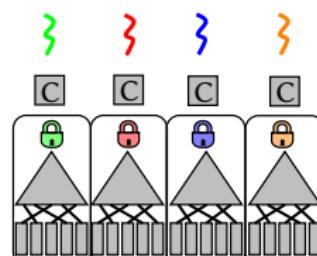
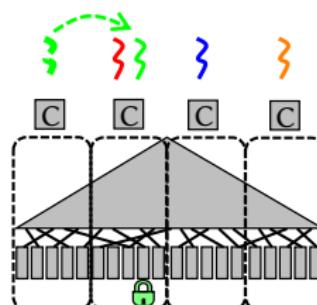
Shared
Every-
thing/
Non-
Partitioned



Delegation



Data-
Oriented
Trans-
action
Execution
(DORA)

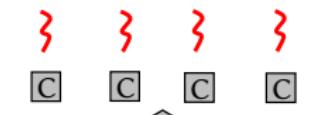


Partitioned
Serial
Execution
(PSE)

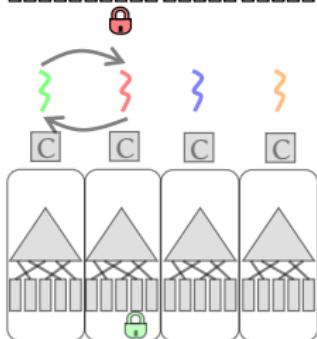
Subsection 1

Shared Everything/Non-Partitioned (SE/NP)

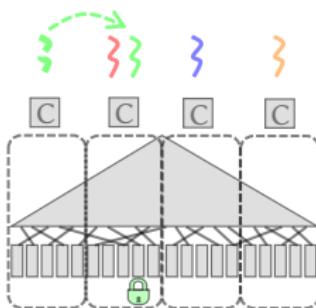
Shared
Every-
thing/
Non-
Partitioned



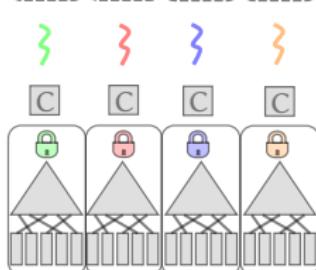
Delegation



Data-
Oriented
Trans-
action
Execu-
tion
(DORA)

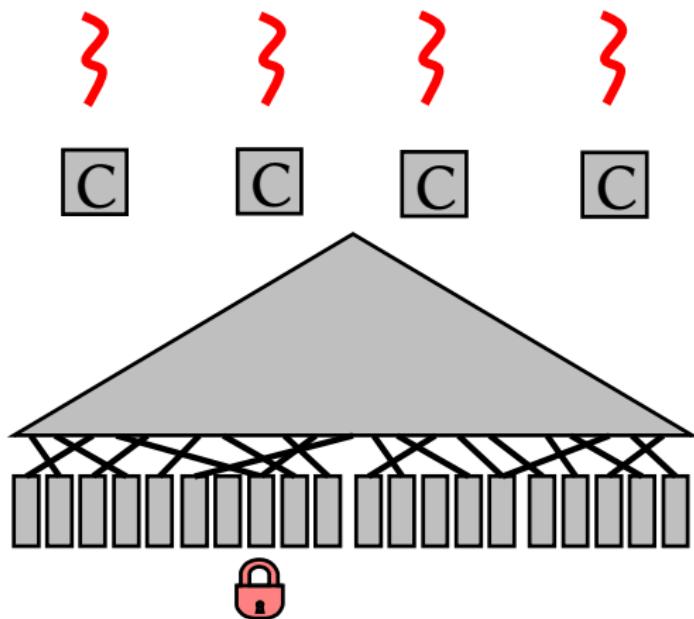


Parti-
tioned
Serial
Execu-
tion
(PSE)



Subsection 1

Shared Everything/Non-Partitioned (SE/NP)



Properties of SE/NP

Properties of SE/NP

- ▶ metadata (incl. locks) aren't partitioned

Properties of SE/NP

- ▶ metadata (incl. locks) aren't partitioned
- physical synchronization (latches, atomic instructions) required

Properties of SE/NP

- ▶ metadata (incl. locks) aren't partitioned
- physical synchronization (latches, atomic instructions) required
- ▶ data and indices aren't partitioned

Properties of SE/NP

- ▶ metadata (incl. locks) aren't partitioned
- physical synchronization (latches, atomic instructions) required
- ▶ data and indices aren't partitioned
- logical synchronization using a concurrency control protocol also required

Properties of SE/NP

- ▶ metadata (incl. locks) aren't partitioned
- physical synchronization (latches, atomic instructions) required
- ▶ data and indices aren't 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

Pros & Cons of SE/NP

- + no partitioning required (e.g. manual selection of a partitioning strategy)

Pros & Cons of SE/NP

- + no partitioning required (e.g. manual selection of a partitioning strategy)
- + partitioning would be sensitive to the workload

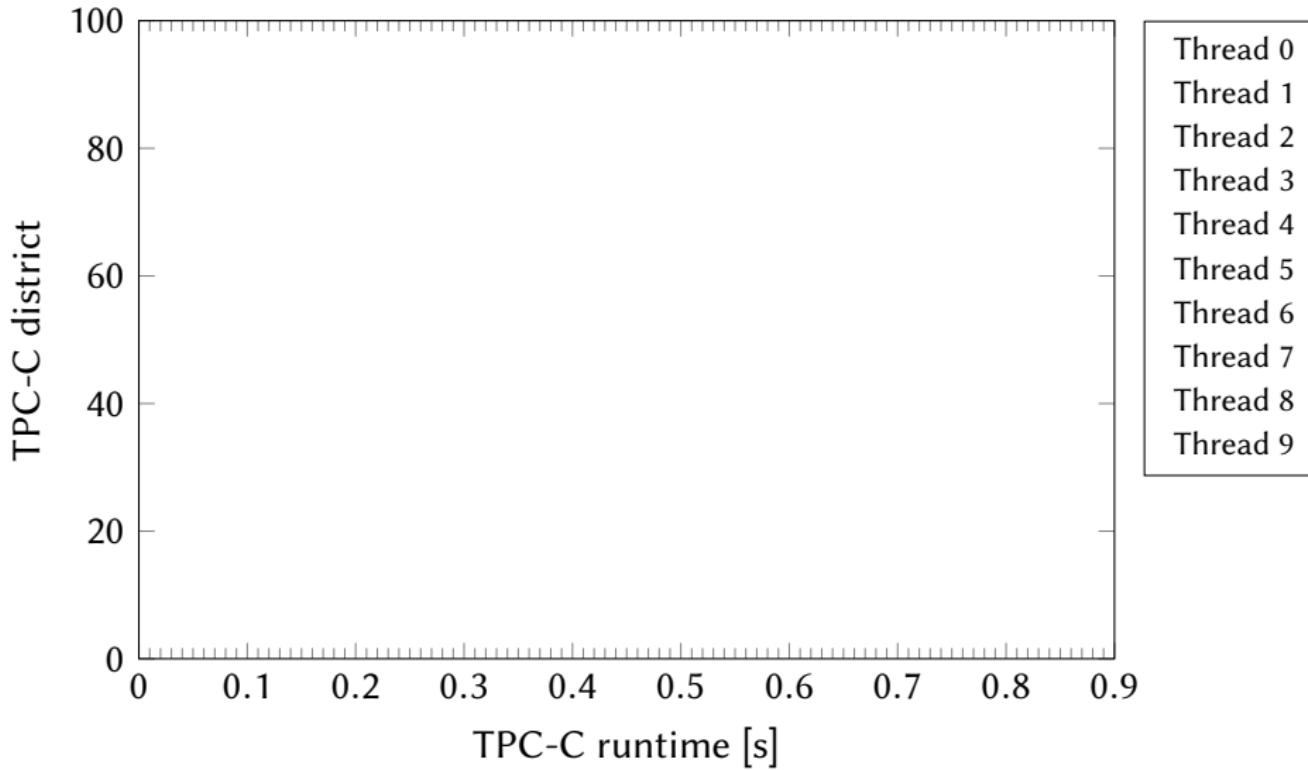
Pros & Cons of SE/NP

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

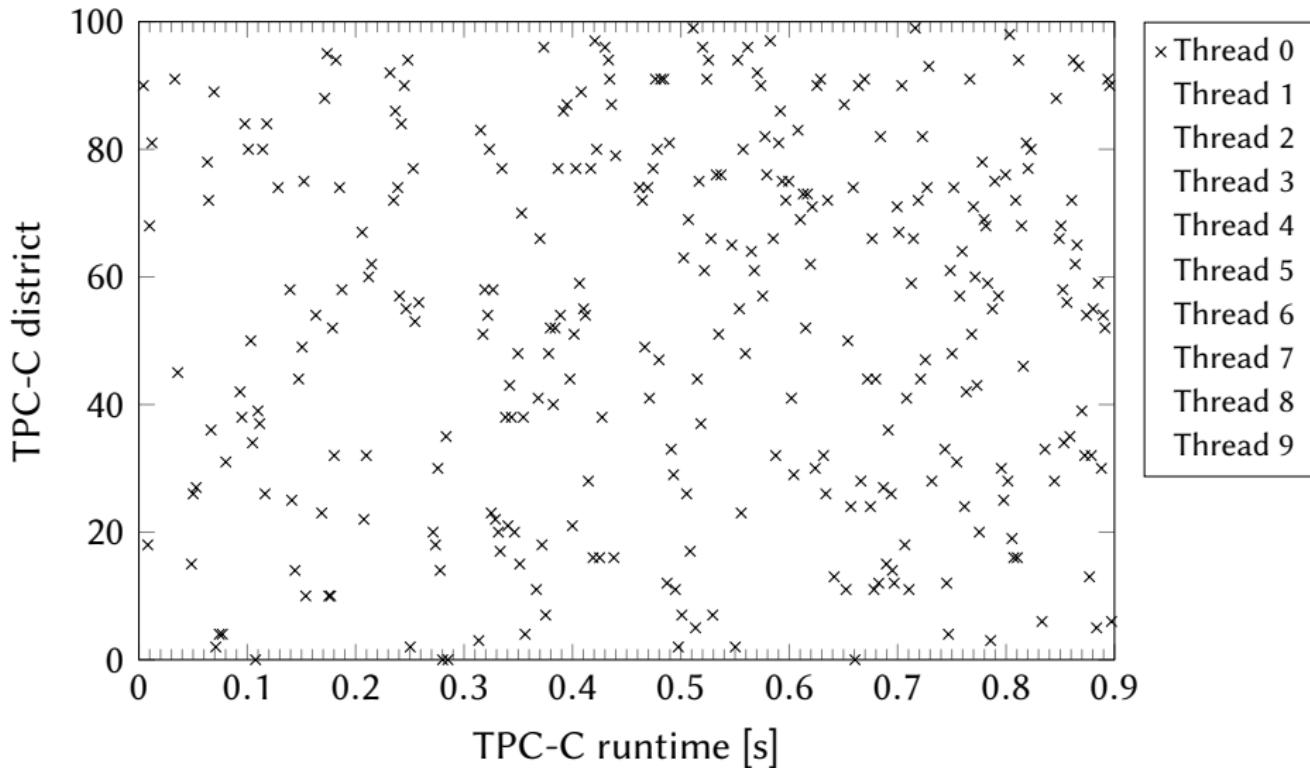
Pros & Cons of SE/NP

- + no partitioning required (e.g. manual selection of a partitioning strategy)
- + partitioning would be sensitive to the workload
- + changed workloads would require (probably expensive) repartitioning to benefit from partitioning
- each thread might access every record at arbitrary times

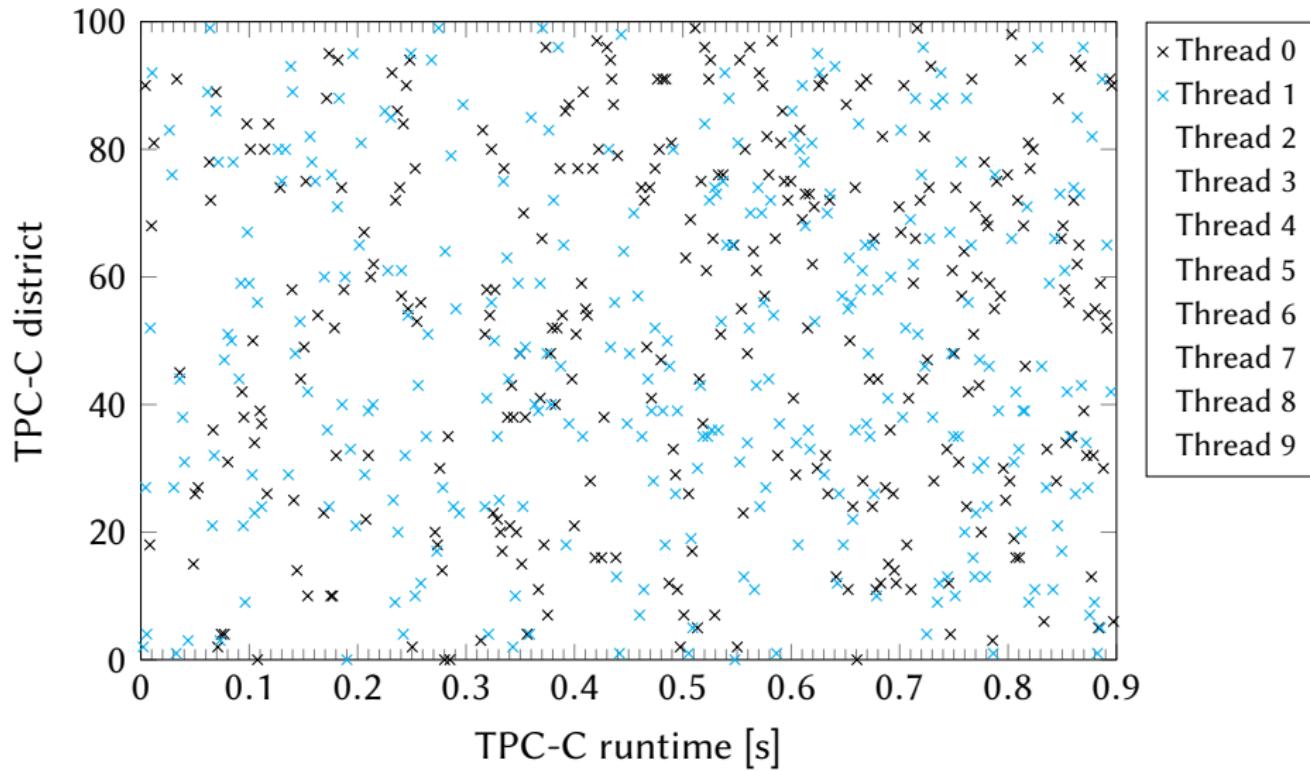
Record Accesses of Conventional DB Threads ([Pan+10])



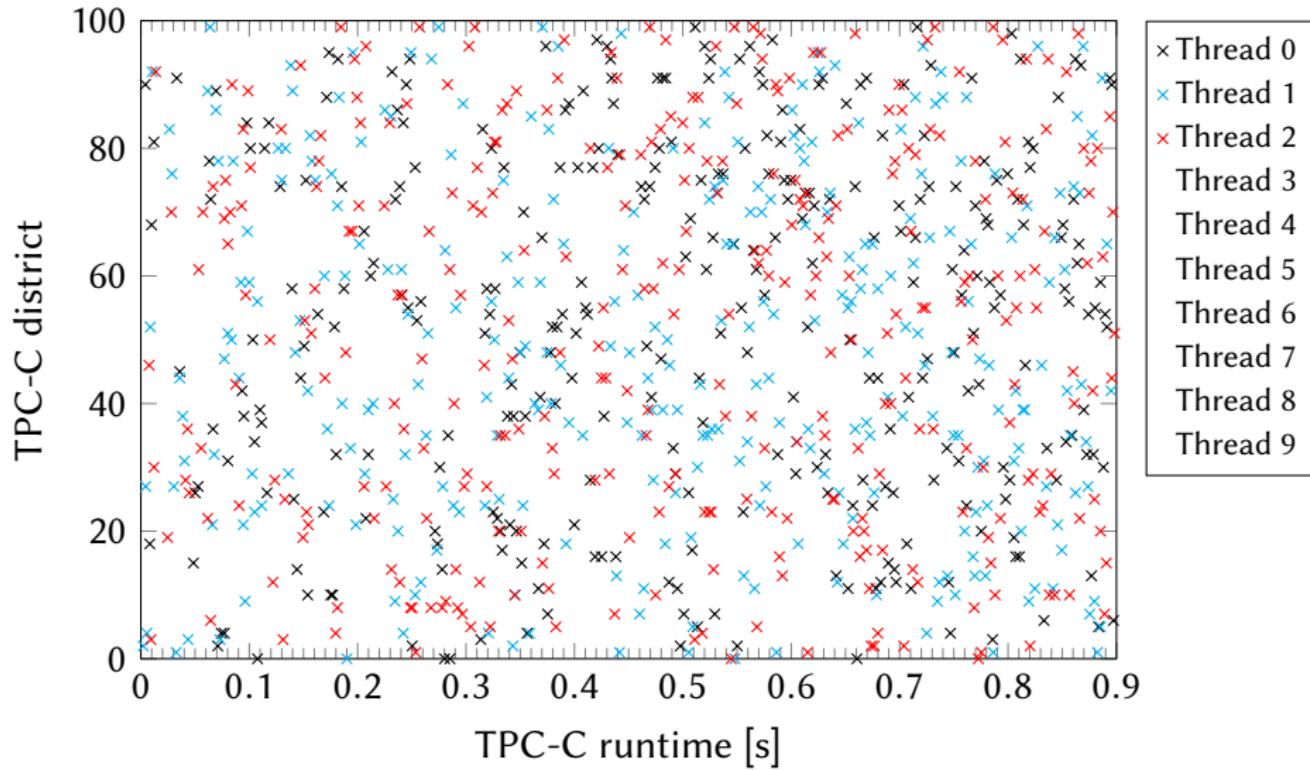
Record Accesses of Conventional DB Threads ([Pan+10])



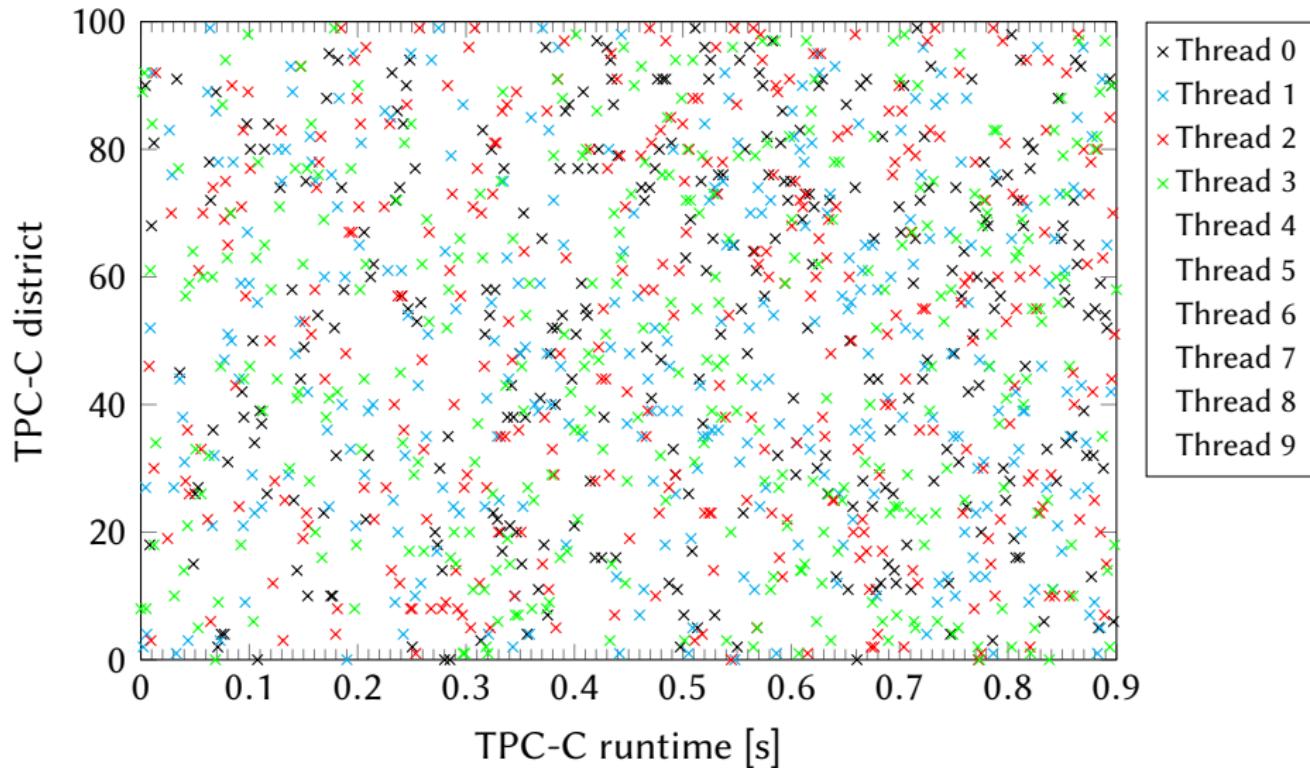
Record Accesses of Conventional DB Threads ([Pan+10])



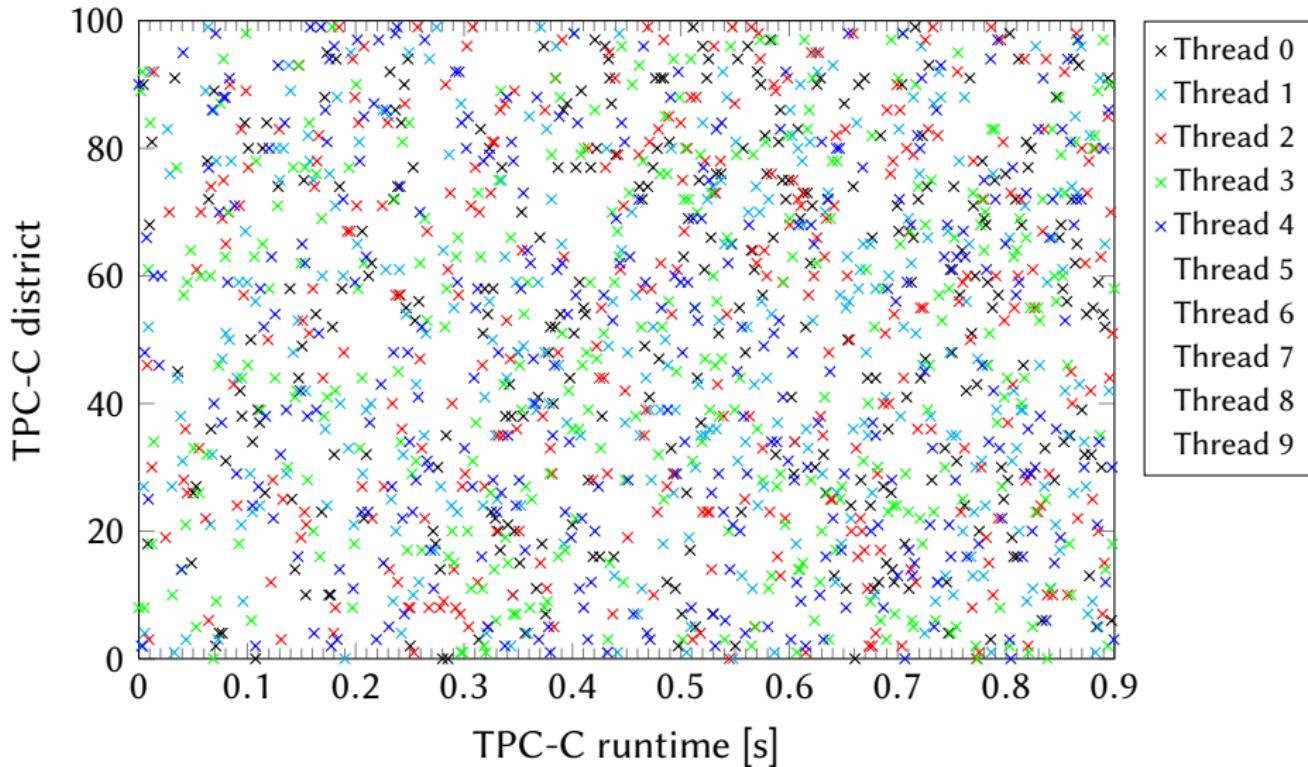
Record Accesses of Conventional DB Threads ([Pan+10])



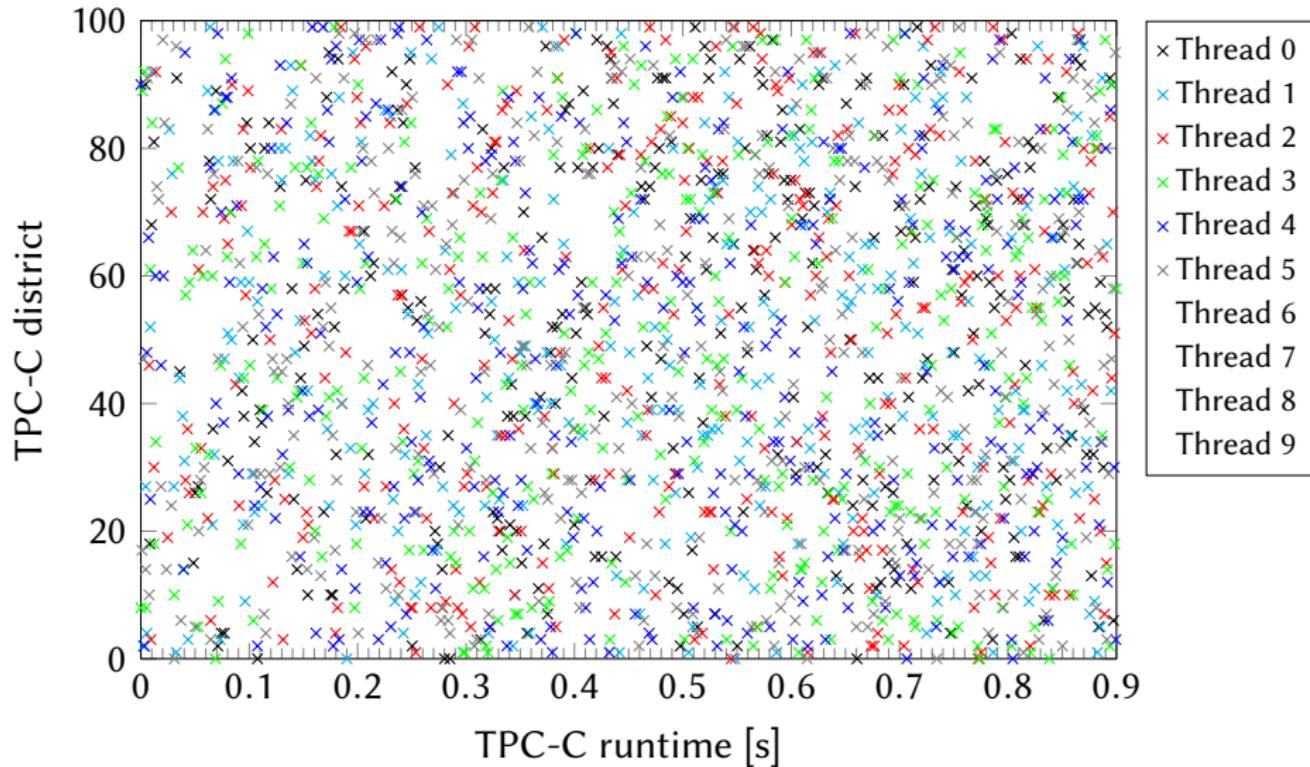
Record Accesses of Conventional DB Threads ([Pan+10])



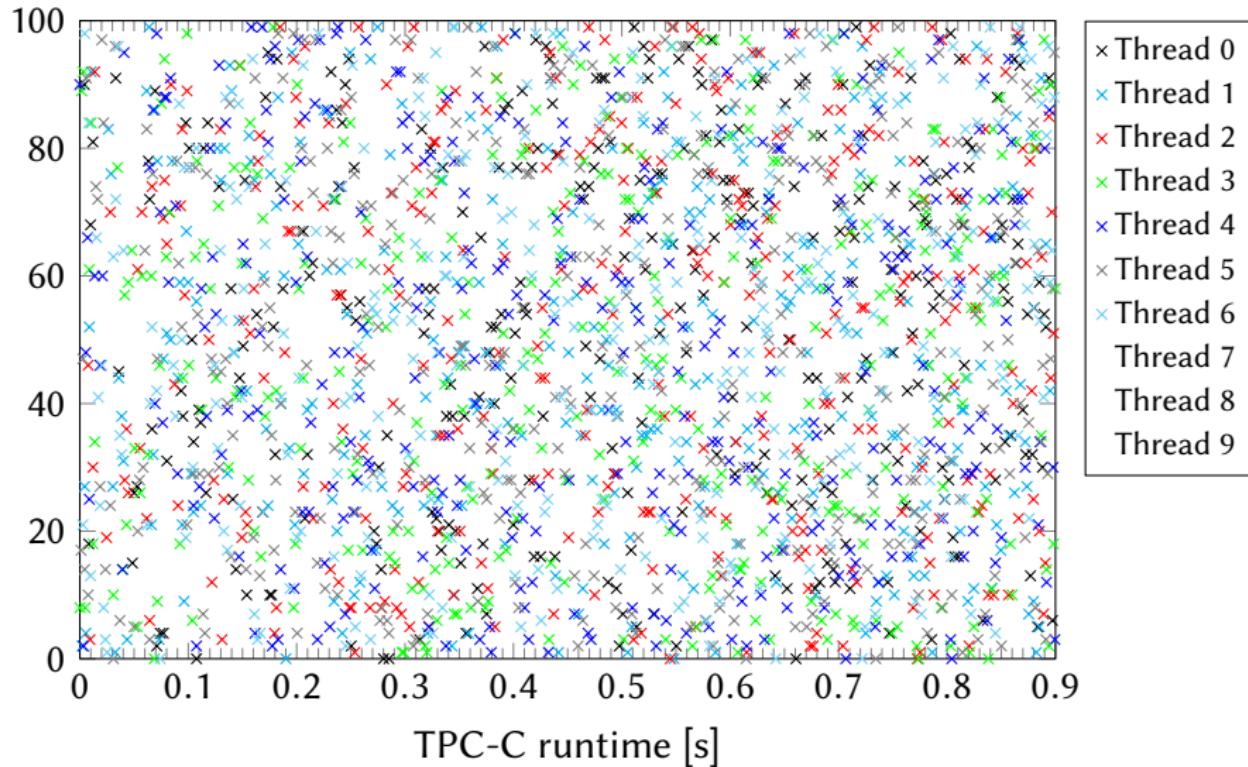
Record Accesses of Conventional DB Threads ([Pan+10])



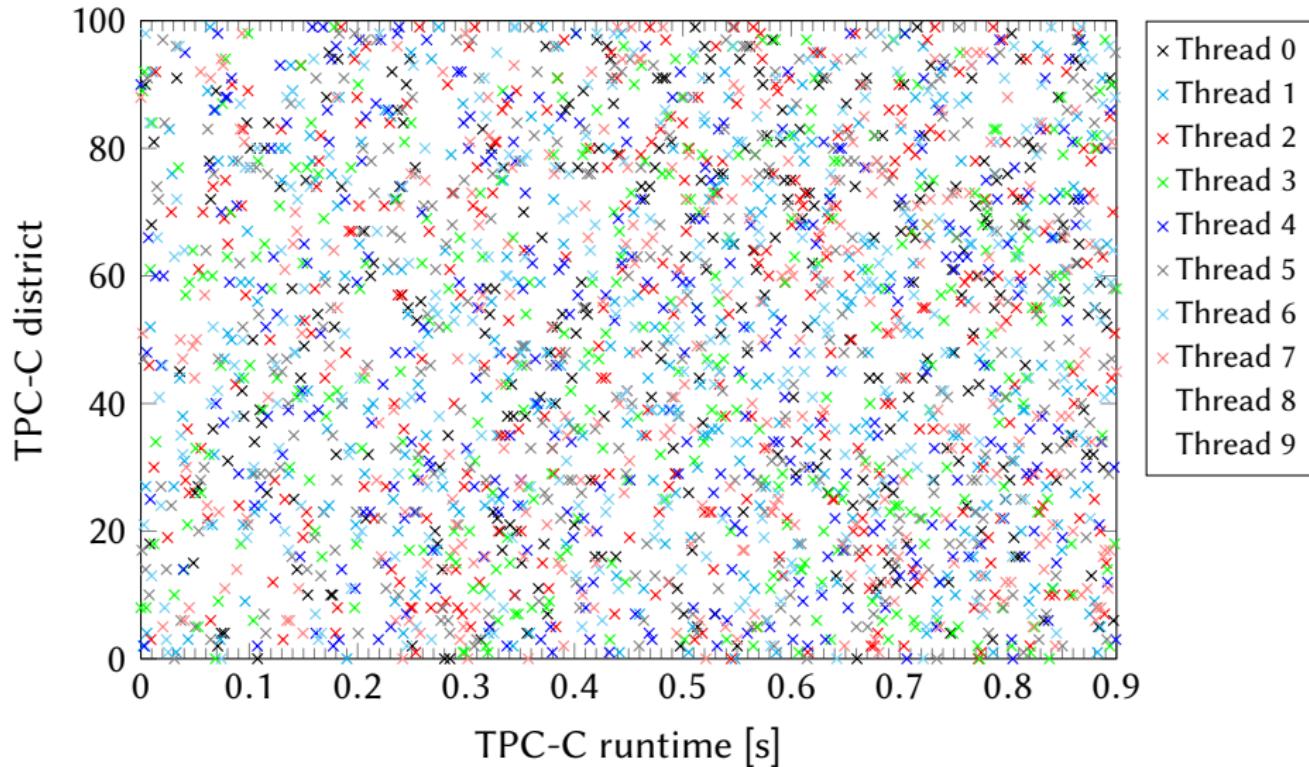
Record Accesses of Conventional DB Threads ([Pan+10])



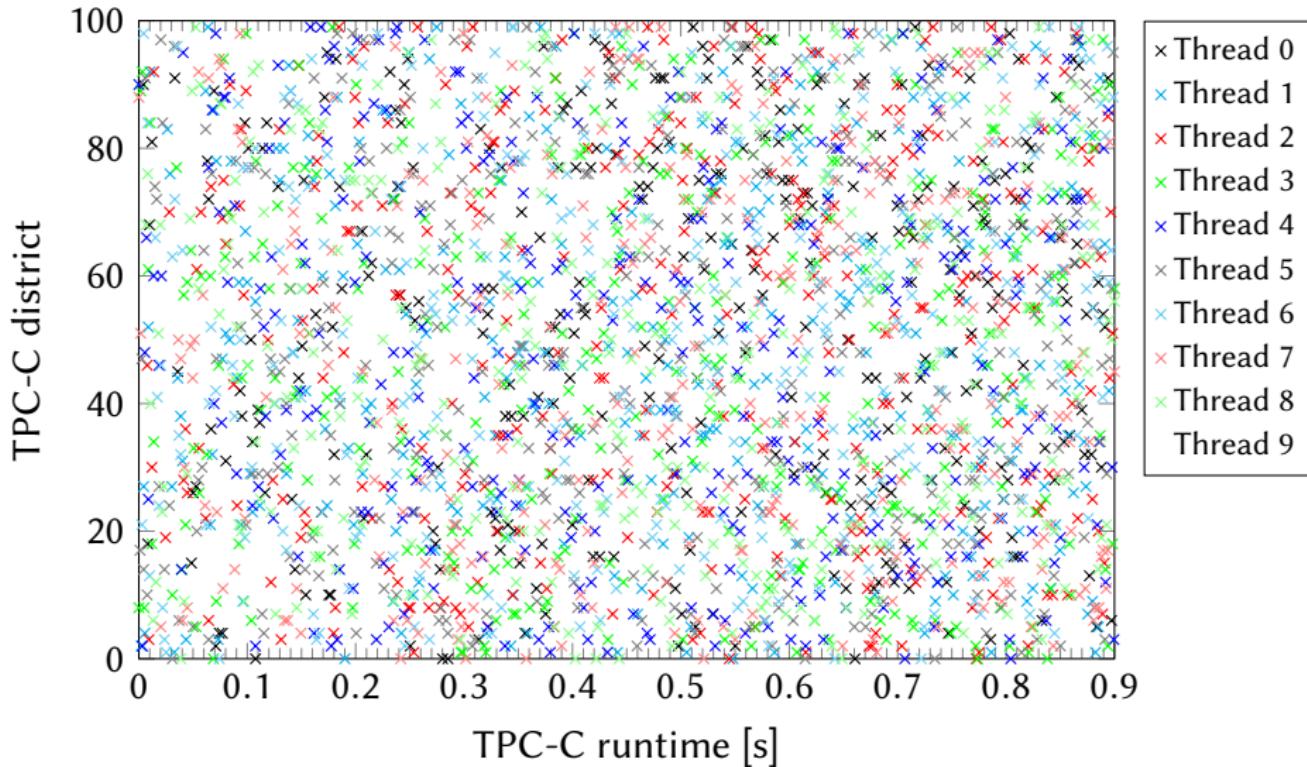
Record Accesses of Conventional DB Threads ([Pan+10])



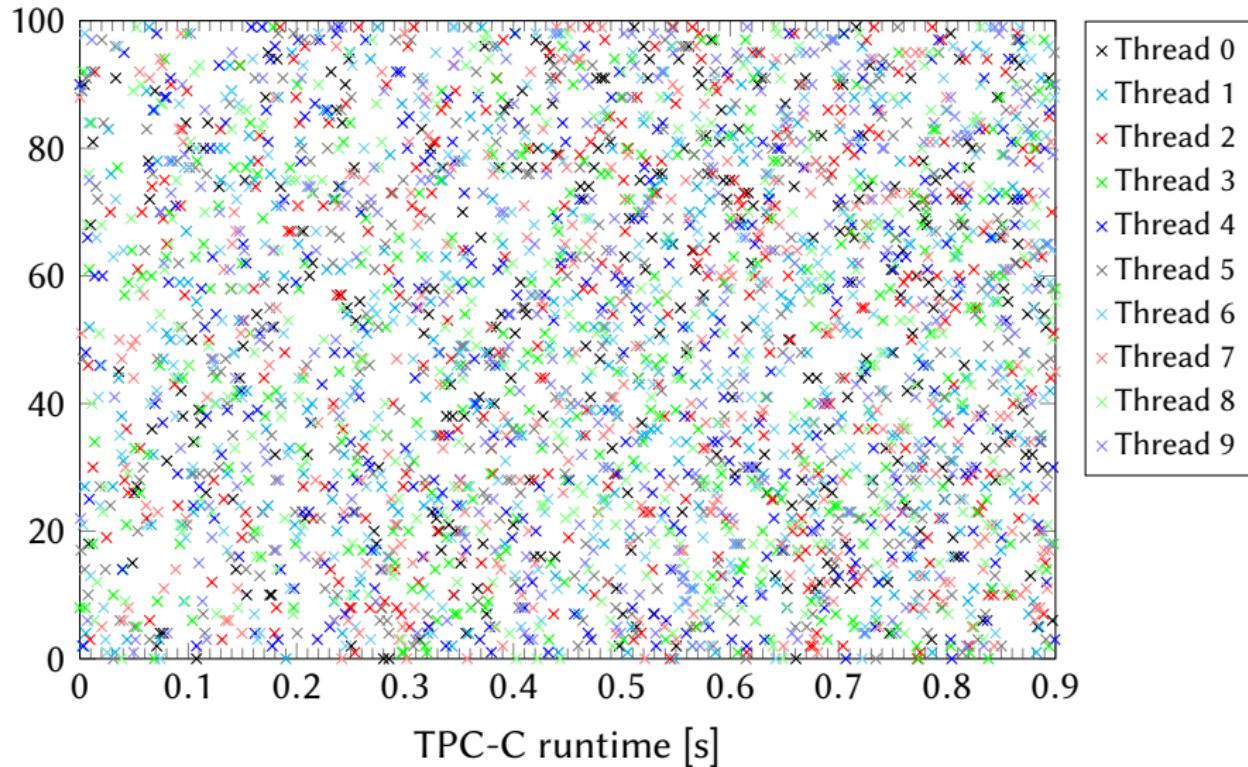
Record Accesses of Conventional DB Threads ([Pan+10])



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

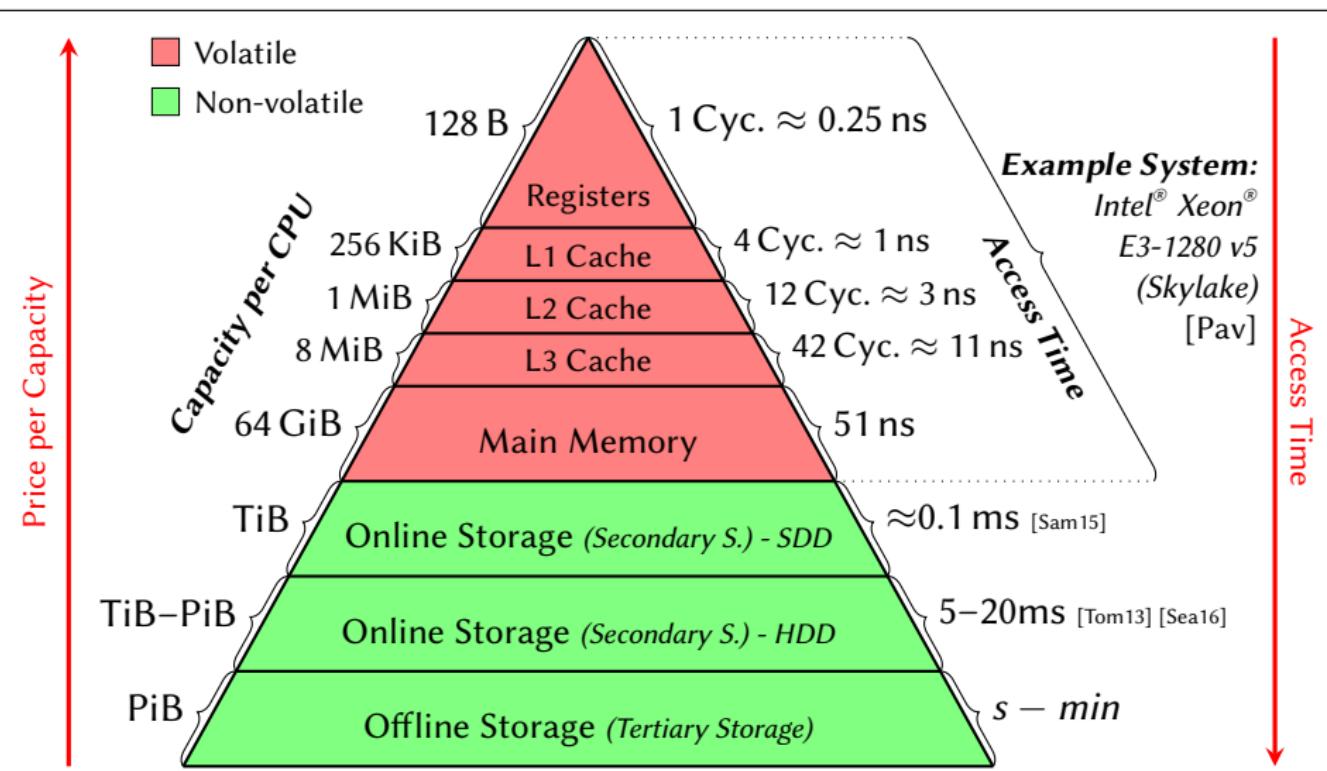


Record Accesses of Conventional DB Threads ([Pan+10])



Pros & Cons of SE/NP

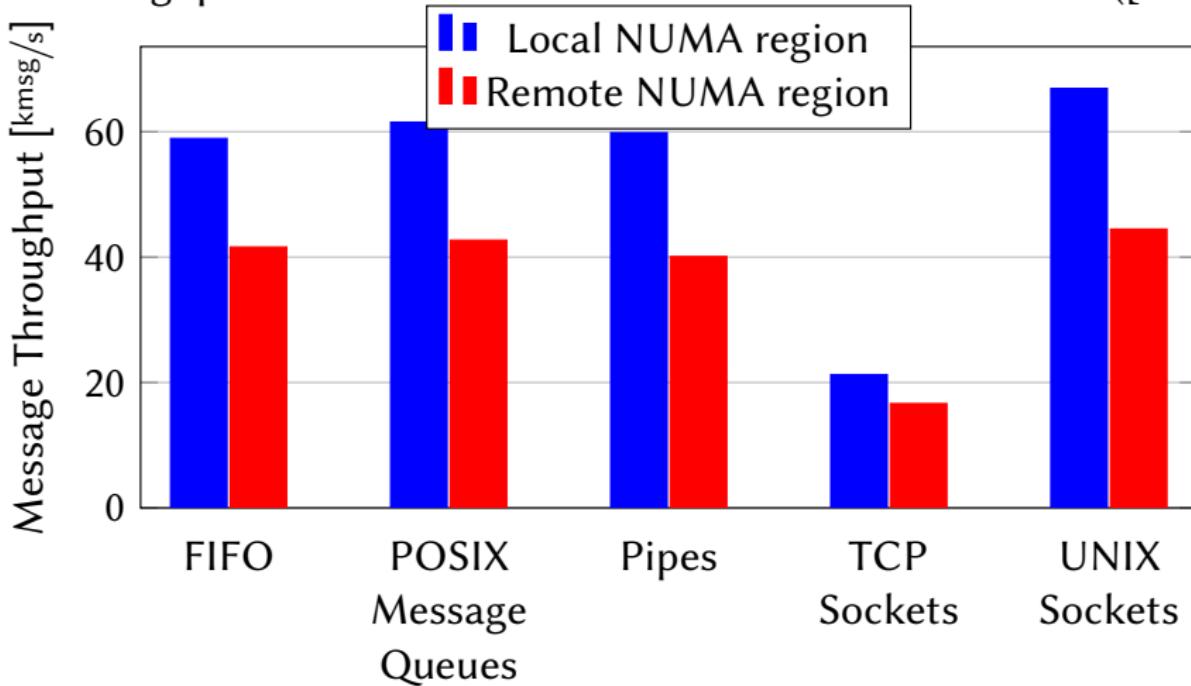
- + no partitioning required (e.g. manual selection of a partitioning strategy)
- + partitioning would be sensitive to the workload
- + changed workloads would require (probably expensive) repartitioning to benefit from partitioning
- each thread might access every record at arbitrary times
 - each CPU cache may contain any part of the data → cache pollution



Pros & Cons of SE/NP

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

Throughput of Inter-Process Communication Mechanisms ([Por+12])



Pros & Cons of SE/NP

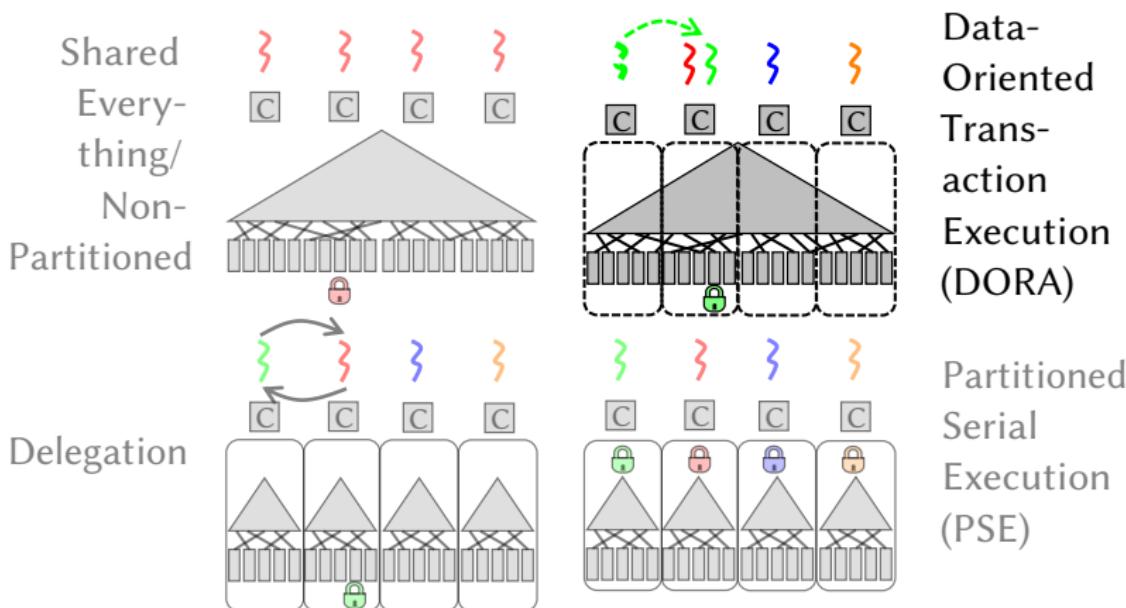
- + no partitioning required (e.g. manual selection of a partitioning strategy)
- + partitioning would be sensitive to the workload
- + changed workloads would require (probably expensive) repartitioning to benefit from partitioning
- each thread might access every record at arbitrary times
 - each CPU 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

Pros & Cons of SE/NP

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

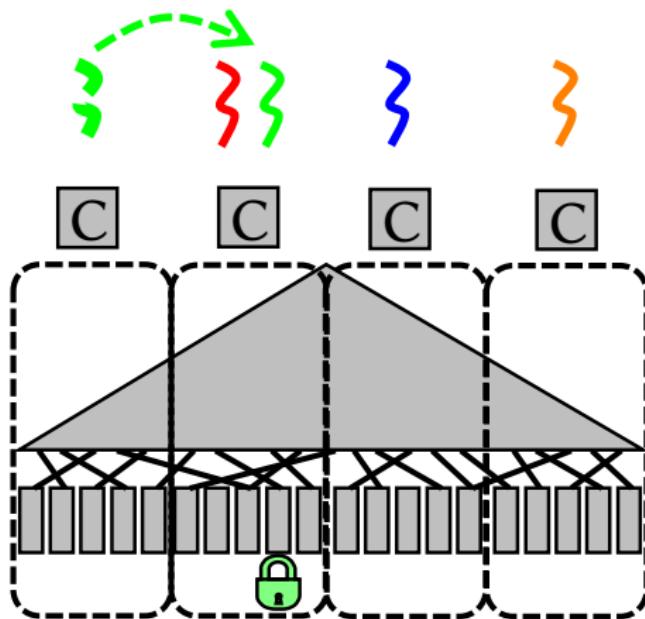
Subsection 2

Data-Oriented Transaction Execution (DORA)



Subsection 2

Data-Oriented Transaction Execution (DORA)



Properties of DORA

Properties of DORA

- ▶ metadata (incl. locks) are physically partitioned

Properties of DORA

- ▶ metadata (incl. locks) are physically partitioned
- no physical synchronization (latches, atomics) required

Properties of DORA

- ▶ metadata (incl. locks) are physically partitioned
- no physical synchronization (latches, atomics) required
- ▶ data and indices are logically partitioned

Properties of DORA

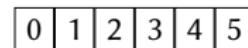
- ▶ metadata (incl. locks) are physically partitioned
- no physical synchronization (latches, atomics) required
- ▶ data and indices are logically partitioned
- logical synchronization using a concurrency control protocol
only locally required

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

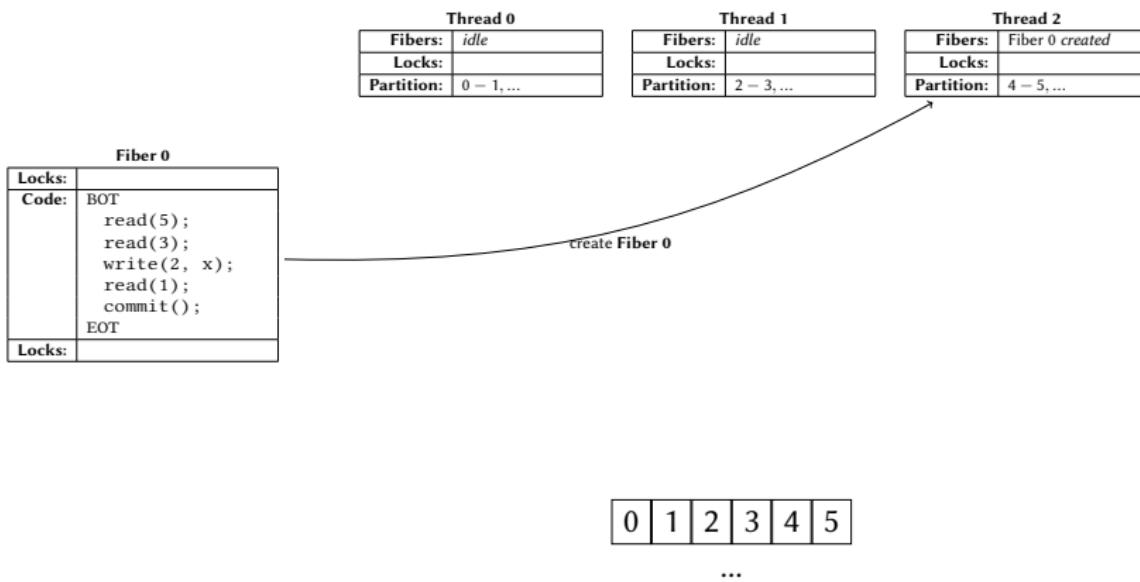
Example

Thread 0		Thread 1		Thread 2	
Fibers:	idle	Fibers:	idle	Fibers:	idle
Locks:		Locks:		Locks:	
Partition:	0 – 1, ...	Partition:	2 – 3, ...	Partition:	4 – 5, ...



...

Example



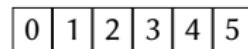
Example

Thread 0		
Fibers:	<i>idle</i>	
Locks:		
Partition:	0 – 1, ...	

Thread 1		
Fibers:	<i>idle</i>	
Locks:		
Partition:	2 – 3, ...	

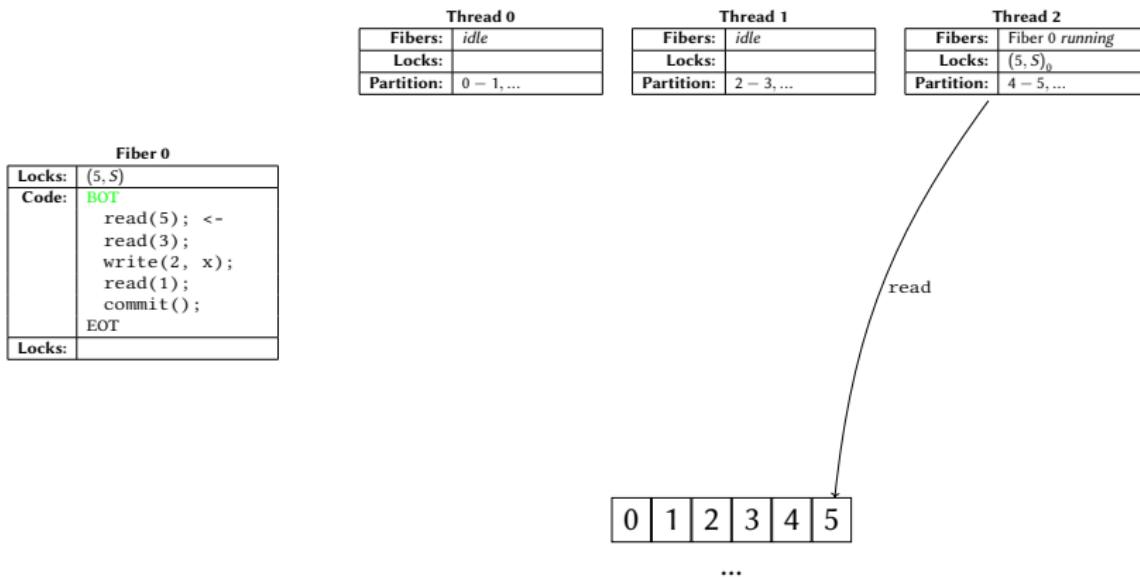
Thread 2		
Fibers:	Fiber 0 <i>waiting</i>	
Locks:		
Partition:	4 – 5, ...	

Fiber 0	
Locks:	
Code:	BOT read(5); read(3); write(2, x); read(1); commit(); EOT
Locks:	



...

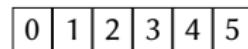
Example



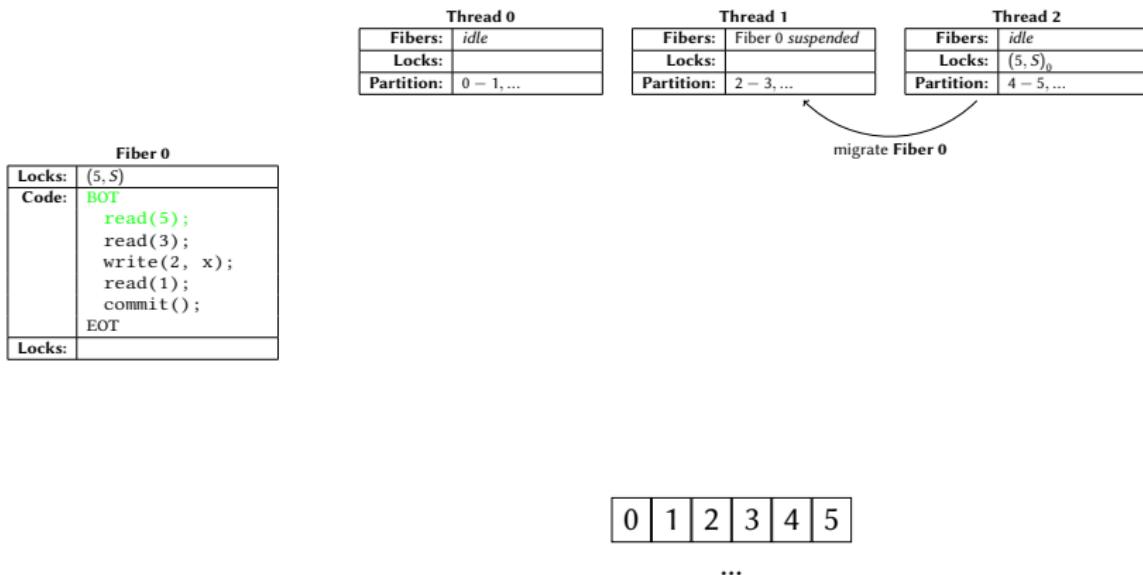
Example

Thread 0		Thread 1		Thread 2	
Fibers:	idle	Fibers:	idle	Fibers:	Fiber 0 suspended
Locks:		Locks:		Locks:	$(5, S)_0$
Partition:	0 – 1, ...	Partition:	2 – 3, ...	Partition:	4 – 5, ...

Fiber 0	
Locks:	$(5, S)$
Code:	<pre>BOT read(5); read(3); write(2, x); read(1); commit(); EOT</pre>
Locks:	



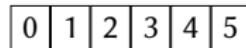
Example



Example

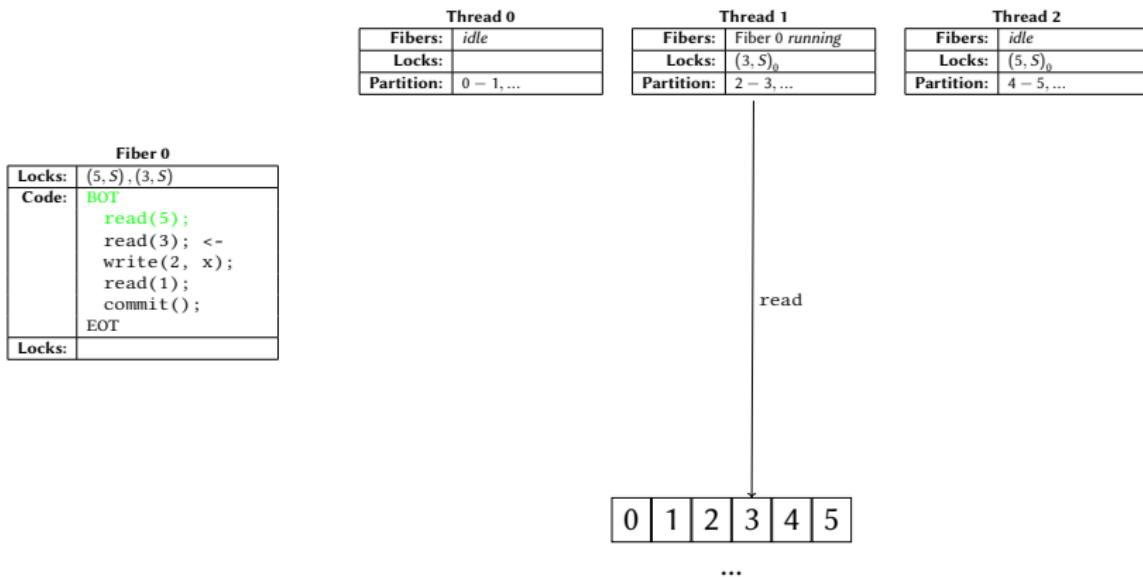
Thread 0			Thread 1			Thread 2		
Fibers:	idle	Fibers:	Fiber 0 waiting	Fibers:	idle	Fibers:	idle	
Locks:		Locks:		Locks:	$(5, S)_0$	Locks:		
Partition:	0 – 1, ...	Partition:	2 – 3, ...	Partition:	4 – 5, ...	Partition:	6 – 7, ...	

Fiber 0	
Locks:	$(5, S)$
Code:	<pre> BOT read(5); read(3); write(2, x); read(1); commit(); EOT </pre>
Locks:	

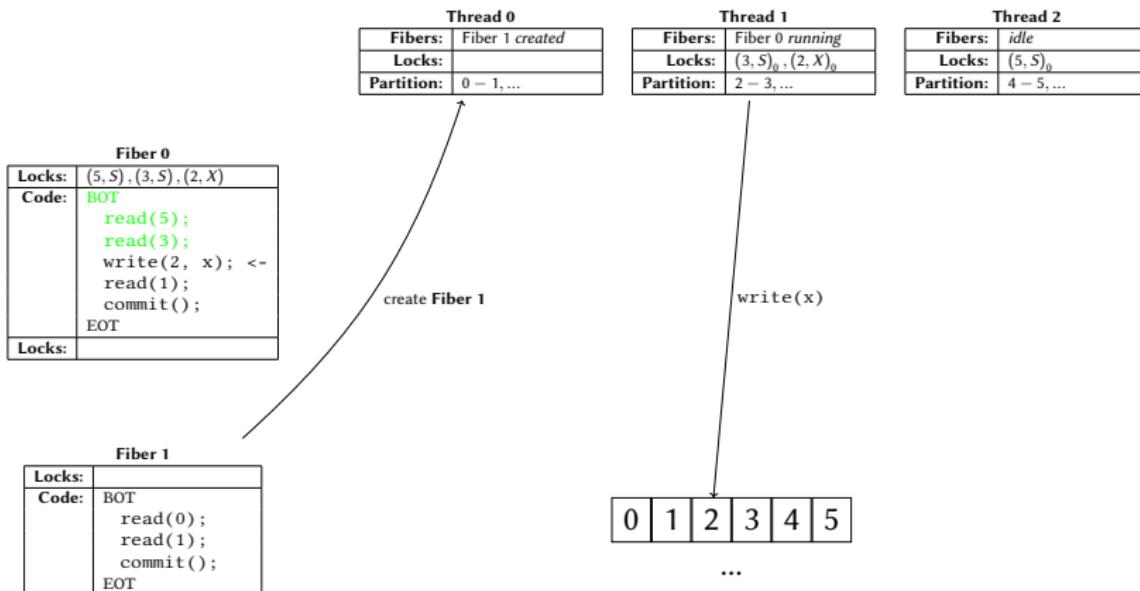


...

Example



Example



Example

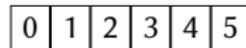
Thread 0			Thread 1			Thread 2		
Fibers:	Fiber 1 waiting	Fibers:	Fiber 0 suspended	Fibers:	idle	Locks:		Locks:
Locks:		Locks:	$(3, S)_0, (2, X)_0$	Locks:	$(5, S)_0$	Partition:		Partition:
Partition: 0 – 1, ...		Partition: 2 – 3, ...		Partition: 4 – 5, ...				

Fiber 0

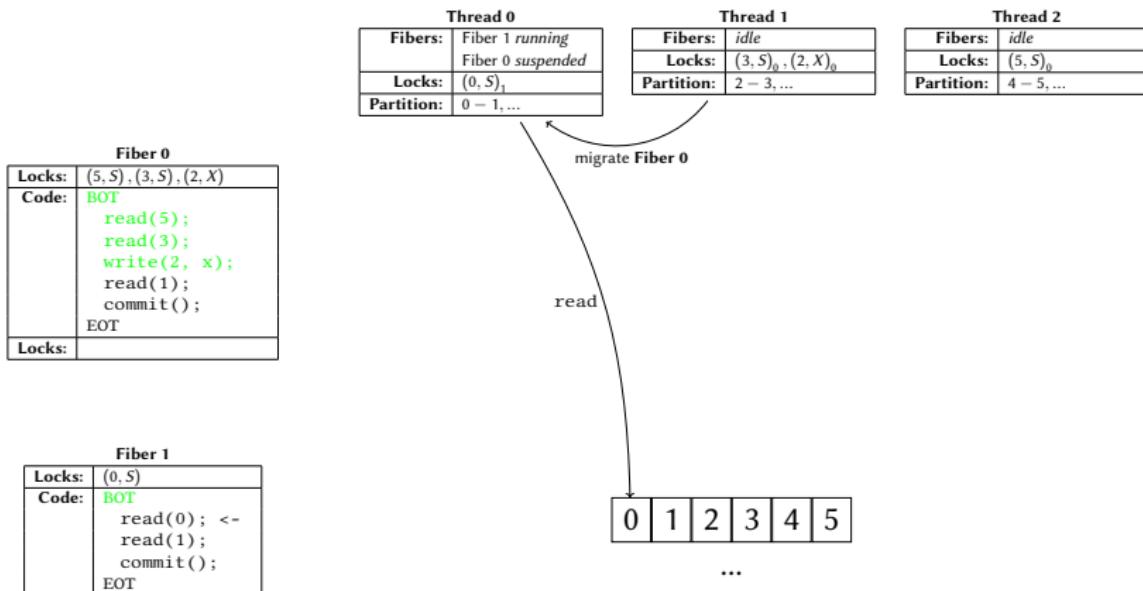
Locks:	$(5, S), (3, S), (2, X)$
Code:	BOT read(5); read(3); write(2, x); read(1); commit(); EOT
Locks:	

Fiber 1

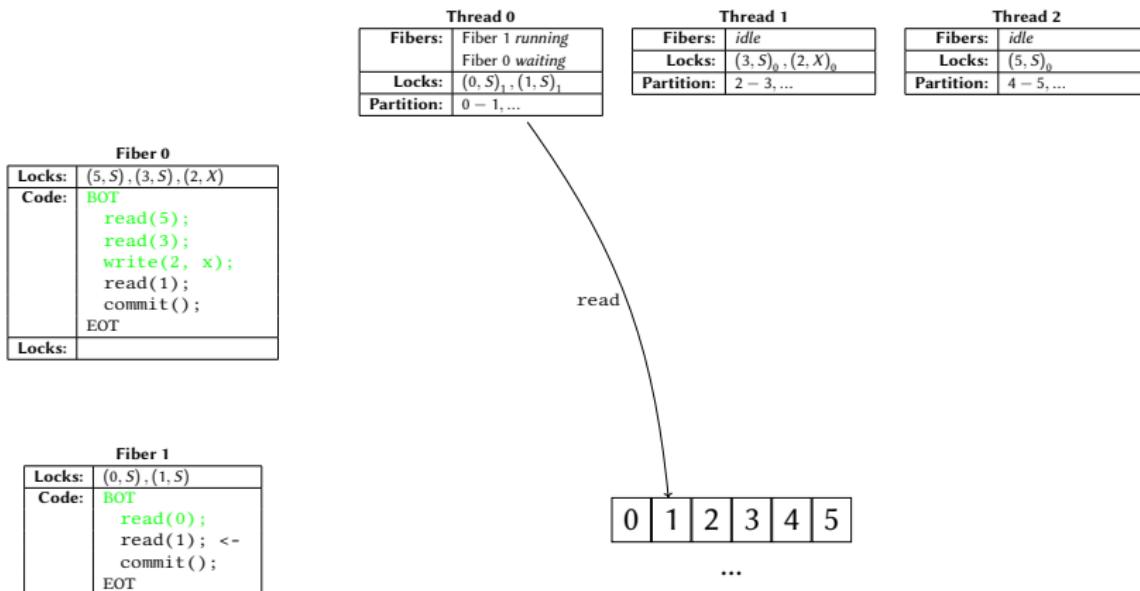
Locks:	
Code:	BOT read(0); read(1); commit(); EOT



Example



Example

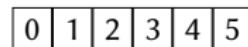


Example

Thread 0		Thread 1		Thread 2	
Fibers:	Fiber 1 committing Fiber 0 waiting	Fibers:	idle	Fibers:	idle
Locks:		Locks:	$(3, S)_0, (2, X)_0$	Locks:	$(5, S)_0$
Partition:	0 – 1, ...	Partition:	2 – 3, ...	Partition:	4 – 5, ...

Fiber 0	
Locks:	$(5, S), (3, S), (2, X)$
Code:	<pre>BOT read(5); read(3); write(2, x); read(1); commit(); EOT</pre>
Locks:	

Fiber 1	
Locks:	
Code:	<pre>BOT read(0); read(1); commit(); <- EOT</pre>

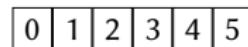


Example

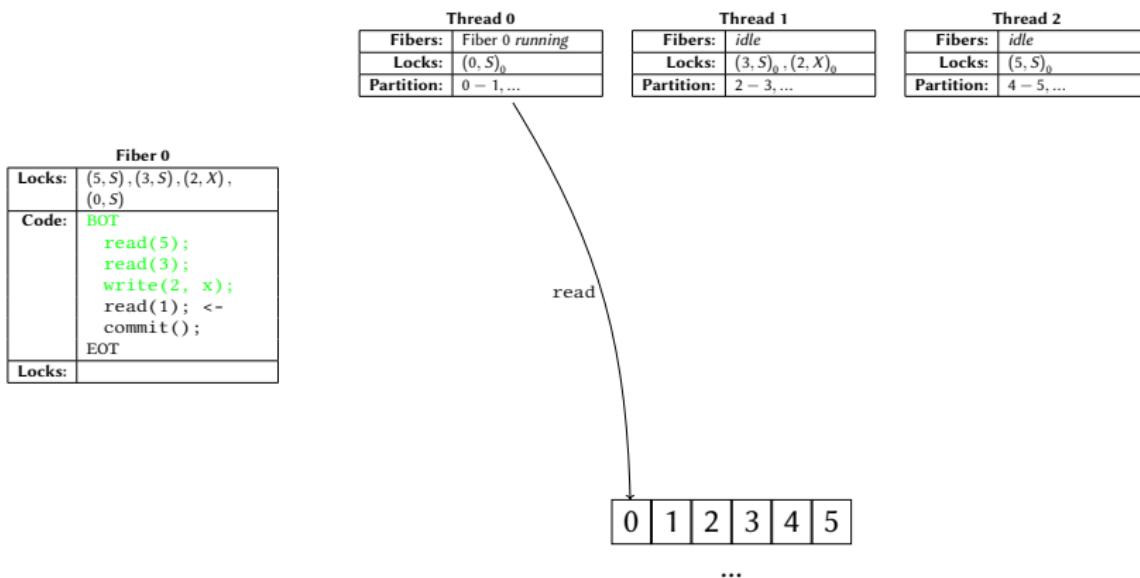
Thread 0		Thread 1		Thread 2	
Fibers:	Fiber 1 terminated Fiber 0 waiting	Fibers:	idle	Fibers:	idle
Locks:		Locks:	$(3, S)_0, (2, X)_0$	Locks:	$(5, S)_0$
Partition:	0 – 1, ...	Partition:	2 – 3, ...	Partition:	4 – 5, ...

Fiber 0	
Locks:	$(5, S), (3, S), (2, X)$
Code:	<pre> BOT read(5); read(3); write(2, x); read(1); commit(); EOT </pre>
Locks:	

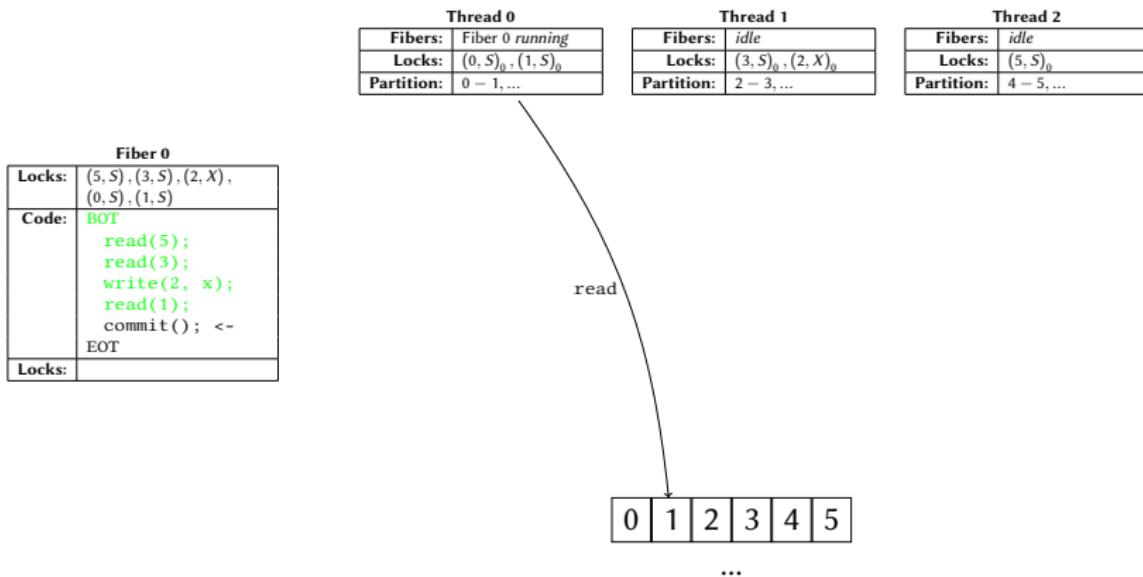
Fiber 1	
Locks:	
Code:	<pre> BOT read(0); read(1); commit(); EOT </pre>
Locks:	



Example



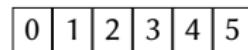
Example



Example

Thread 0			Thread 1			Thread 2		
Fibers:	Fiber 0 committing		Fibers:	idle		Fibers:	idle	
Locks:			Locks:	$(3, S)_0, (2, X)_0$		Locks:		
Partition:	0 – 1, ...		Partition:	2 – 3, ...		Partition:	4 – 5, ...	

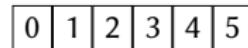
Fiber 0	
Locks:	$(5, S), (3, S), (2, X)$
Code:	<pre>BOT read(5); read(3); write(2, x); read(1); commit(); <- EOT</pre>
Locks:	



Example

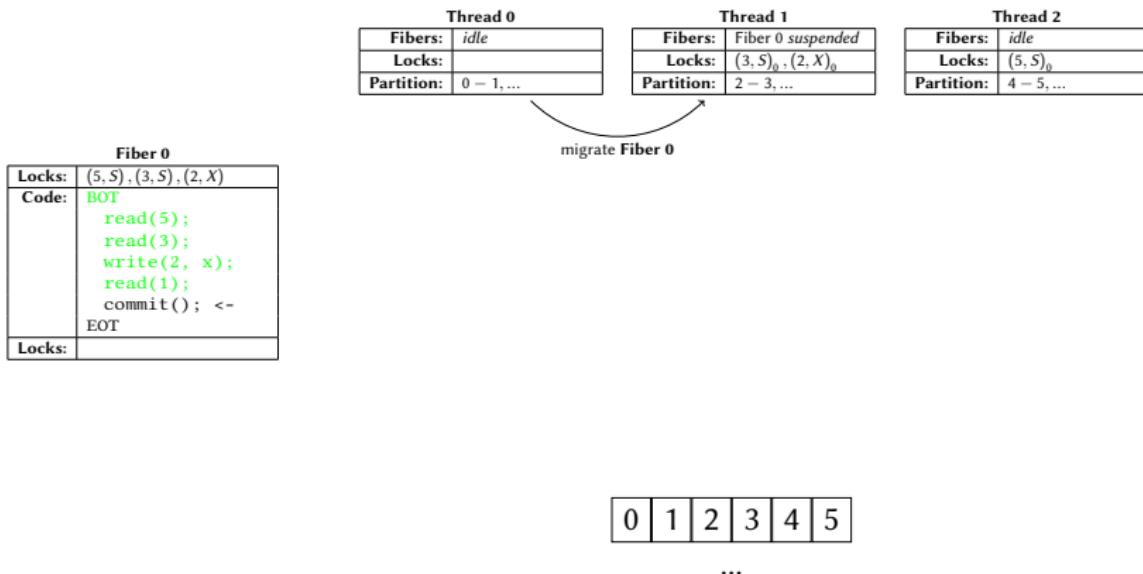
Thread 0			Thread 1			Thread 2		
Fibers:	Fiber 0 suspended		Fibers:	idle		Fibers:	idle	
Locks:			Locks:	$(3, S)_0, (2, X)_0$		Locks:		
Partition:	0 – 1, ...		Partition:	2 – 3, ...		Partition:	4 – 5, ...	

Fiber 0	
Locks:	$(5, S), (3, S), (2, X)$
Code:	<pre>BOT read(5); read(3); write(2, x); read(1); commit(); <- EOT</pre>
Locks:	



...

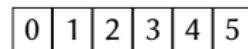
Example



Example

Thread 0			Thread 1			Thread 2		
Fibers:	idle	Fibers:	Fiber 0 waiting	Fibers:	idle	Fibers:	idle	
Locks:		Locks:	$(3, S)_0, (2, X)_0$	Locks:		Locks:	$(5, S)_0$	
Partition:	0 – 1, ...	Partition:	2 – 3, ...	Partition:	4 – 5, ...	Partition:	6 – 7, ...	

Fiber 0	
Locks:	$(5, S), (3, S), (2, X)$
Code:	<pre>BOT read(5); read(3); write(2, x); read(1); commit(); <- EOT</pre>
Locks:	

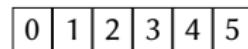


...

Example

Thread 0			Thread 1			Thread 2		
Fibers:	idle	Fibers:	Fiber 0 committing	Fibers:	idle	Fibers:	idle	
Locks:		Locks:		Locks:	$(5, S)_0$	Locks:		
Partition:	0 – 1, ...		Partition:	2 – 3, ...		Partition:	4 – 5, ...	

Fiber 0	
Locks:	$(5, S)$
Code:	<pre> BOT read(5); read(3); write(2, x); read(1); commit(); <- EOT </pre>
Locks:	

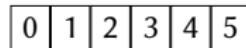


...

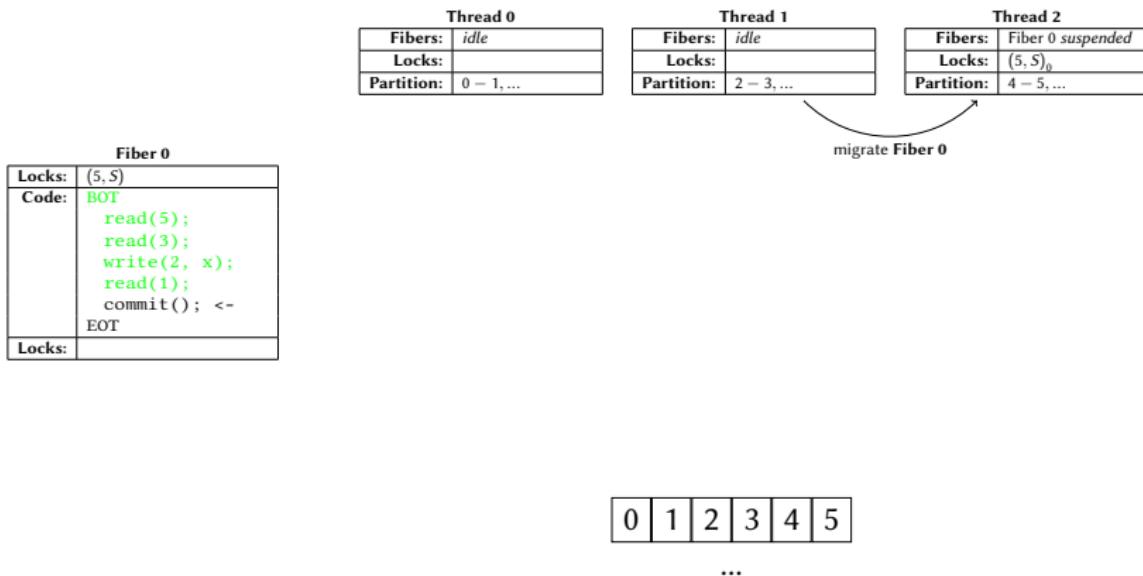
Example

Thread 0			Thread 1			Thread 2		
Fibers:	idle	Fibers:	Fiber 0 suspended	Fibers:	idle	Fibers:	idle	
Locks:		Locks:		Locks:	$(5, S)_0$	Locks:		
Partition:	0 – 1, ...		Partition:	2 – 3, ...		Partition:	4 – 5, ...	

Fiber 0	
Locks:	$(5, S)$
Code:	<pre>BOT read(5); read(3); write(2, x); read(1); commit(); <- EOT</pre>
Locks:	



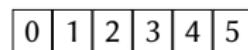
Example



Example

Thread 0			Thread 1			Thread 2		
Fibers:	<i>idle</i>		Fibers:	<i>idle</i>		Fibers:	<i>Fiber 0 waiting</i>	
Locks:			Locks:			Locks:	$(5, S)_0$	
Partition:	0 – 1, ...		Partition:	2 – 3, ...		Partition:	4 – 5, ...	

Fiber 0	
Locks:	$(5, S)$
Code:	<pre>BOT read(5); read(3); write(2, x); read(1); commit(); <- EOT</pre>
Locks:	

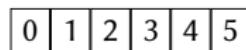


...

Example

Thread 0		Thread 1		Thread 2	
Fibers:	idle	Fibers:	idle	Fibers:	Fiber 0 committing
Locks:		Locks:		Locks:	
Partition:	0 – 1, ...	Partition:	2 – 3, ...	Partition:	4 – 5, ...

Fiber 0	
Locks:	
Code:	<pre>BOT read(5); read(3); write(2, x); read(1); commit(); <- EOT</pre>
Locks:	



...

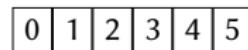
Example

Thread 0		
Fibers:	<i>idle</i>	
Locks:		
Partition:	0 – 1, ...	

Thread 1		
Fibers:	<i>idle</i>	
Locks:		
Partition:	2 – 3, ...	

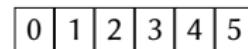
Thread 2		
Fibers:	<i>Fiber 0 terminated</i>	
Locks:		
Partition:	4 – 5, ...	

Fiber 0	
Locks:	
Code:	<code>BOT read(5); read(3); write(2, x); read(1); commit(); EOT</code>
Locks:	



Example

Thread 0		Thread 1		Thread 2	
Fibers:	idle	Fibers:	idle	Fibers:	idle
Locks:		Locks:		Locks:	
Partition:	0 – 1, ...	Partition:	2 – 3, ...	Partition:	4 – 5, ...



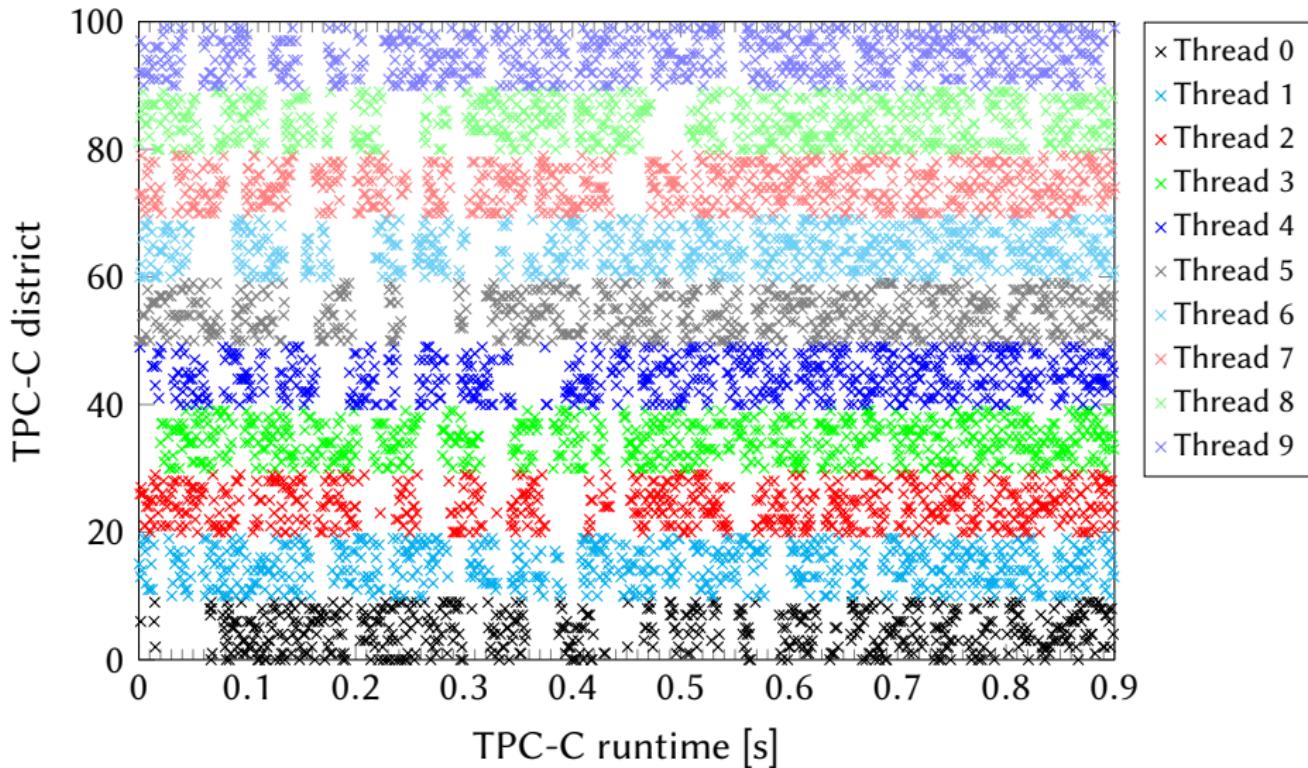
...

Pros & Cons of DORA I

Pros & Cons of DORA I

- + each thread accesses only the records of its partition

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



Pros & Cons of DORA I

- + each thread accesses only the records of its partition
 - + each CPU cache may contain only data of its partition → lower cache pollution

Pros & Cons of DORA I

- + 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)

Pros & Cons of DORA I

- + 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!

Pros & Cons of DORA I

- + 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

Pros & Cons of DORA I

- + 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
- + multi-site transactions could exploit intra-transaction parallelism

Pros & Cons of DORA I

- + 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
- + multi-site transactions could exploit intra-transaction parallelism
- partitioning required (e.g. manual selection of a partitioning strategy—*called routing rule*)

Pros & Cons of DORA II

Pros & Cons of DORA II

- partitioning is sensitive to the workload

Pros & Cons of DORA II

- partitioning is sensitive to the workload
- multi-site transactions require expensive fiber-migration (probably between NUMA regions)

Pros & Cons of DORA II

- 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

Pros & Cons of DORA II

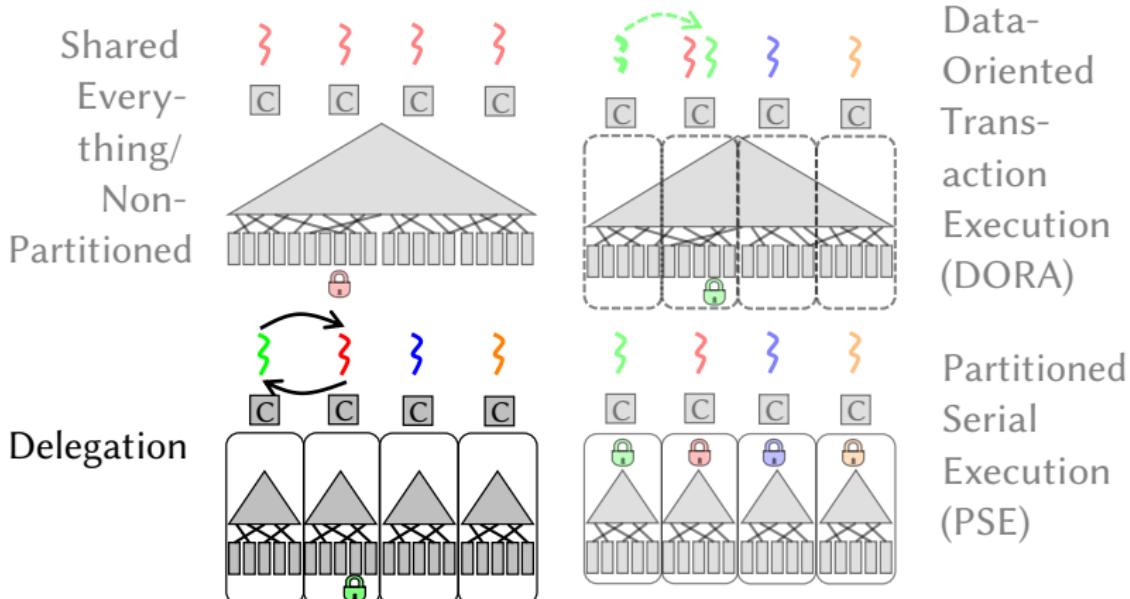
- partitioning is sensitive to the workload
- multi-site transactions require expensive fiber-migration (probably between NUMA regions)
- accessed partitions need to be calculated during query analysis for optimal performance → slower accesses with secondary index
- primary index is shared → centralized latching for inserts/deletes still required → some contention on the shared latch

Pros & Cons of DORA II

- 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)

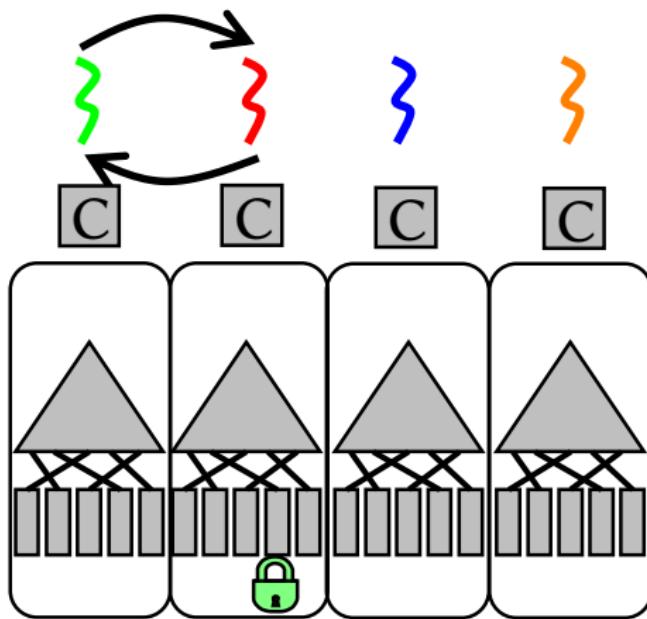
Subsection 3

Delegation



Subsection 3

Delegation



Properties of Delegation

Properties of Delegation

- ▶ metadata (incl. locks) are physically partitioned

Properties of Delegation

- ▶ metadata (incl. locks) are physically partitioned
- no physical synchronization (latches, atomics) required

Properties of Delegation

- ▶ metadata (incl. locks) are physically partitioned
- no physical synchronization (latches, atomics) required
- ▶ data and indices are physically partitioned

Properties of Delegation

- ▶ metadata (incl. locks) are physically partitioned
- no physical synchronization (latches, atomics) required
- ▶ data and indices are physically partitioned
- logical synchronization using a concurrency control protocol
only locally required

Properties of Delegation

- ▶ metadata (incl. locks) are physically partitioned
- no physical synchronization (latches, atomics) required
- ▶ data and indices are physically partitioned
- logical synchronization using a concurrency control protocol only locally required
- ▶ transactions completely executed by one thread (ideally the one owning the majority of the records)

Properties of Delegation

- ▶ metadata (incl. locks) are physically partitioned
- no physical synchronization (latches, atomics) required
- ▶ data and indices are physically partitioned
- logical synchronization using a concurrency control protocol only locally required
- ▶ transactions completely executed by one thread (ideally the one owning the majority of the records)
- ▶ thread accesses remote records by passing messages to the threads owning them

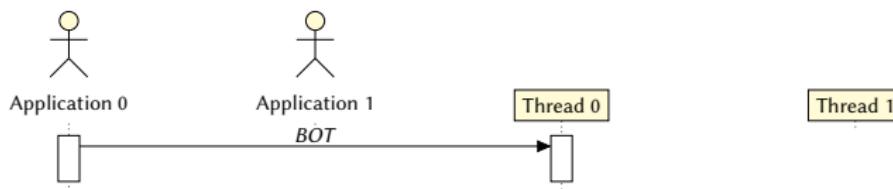
Properties of Delegation

- ▶ metadata (incl. locks) are physically partitioned
- no physical synchronization (latches, atomics) required
- ▶ data and indices are physically partitioned
- logical synchronization using a concurrency control protocol only locally required
- ▶ transactions completely executed by one thread (ideally the one owning the majority of the records)
- ▶ thread accesses remote records by passing messages to the threads owning them
 - ▶ message passing implemented using shared variables

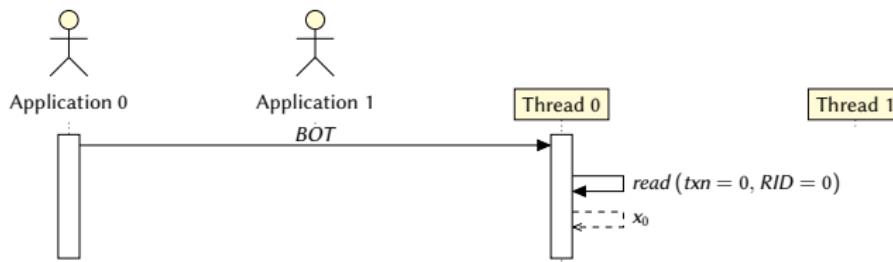
Properties of Delegation

- ▶ metadata (incl. locks) are physically partitioned
- no physical synchronization (latches, atomics) required
- ▶ data and indices are physically partitioned
- logical synchronization using a concurrency control protocol only locally required
- ▶ transactions completely executed by one thread (ideally the one owning the majority of the records)
- ▶ thread accesses remote records by passing messages to the threads owning them
 - ▶ message passing implemented using shared variables
 - ▶ remote records are passed as pointers

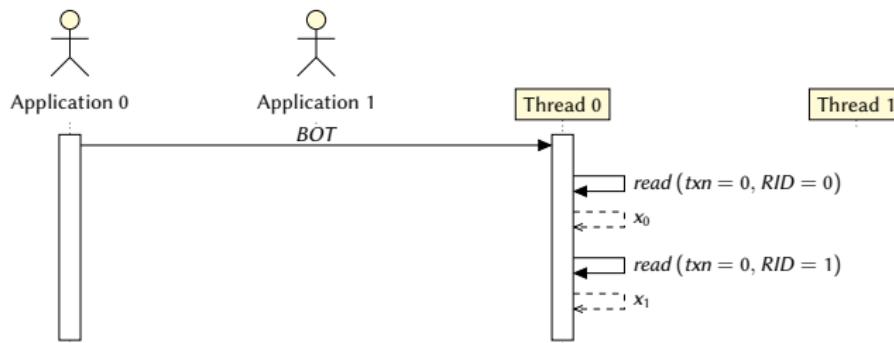
Example



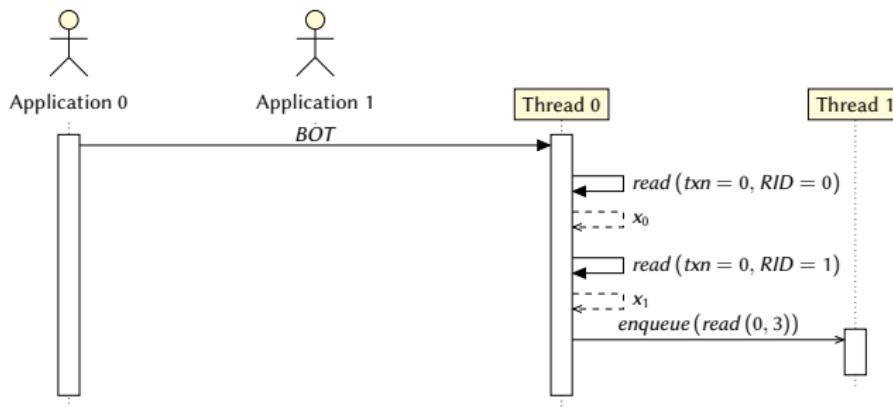
Example



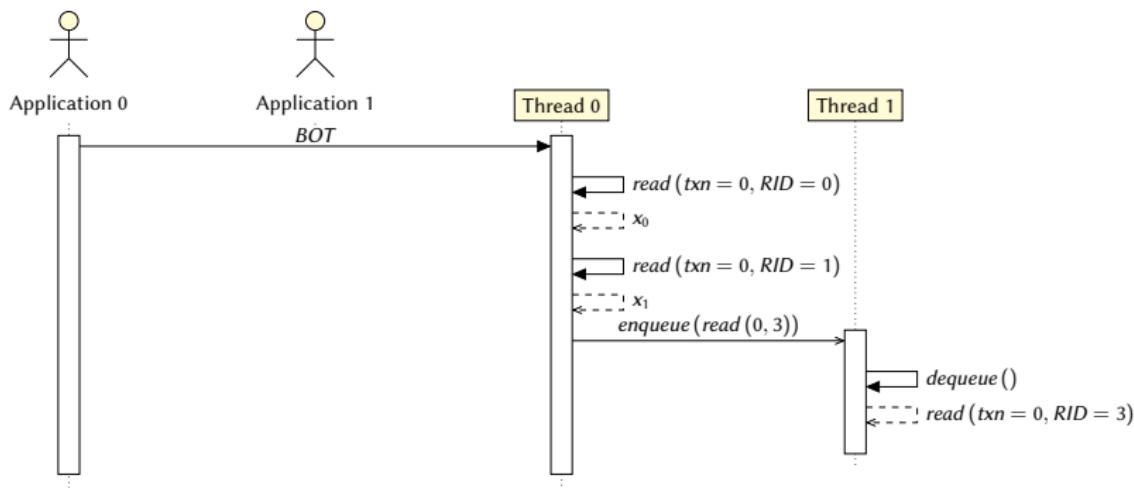
Example



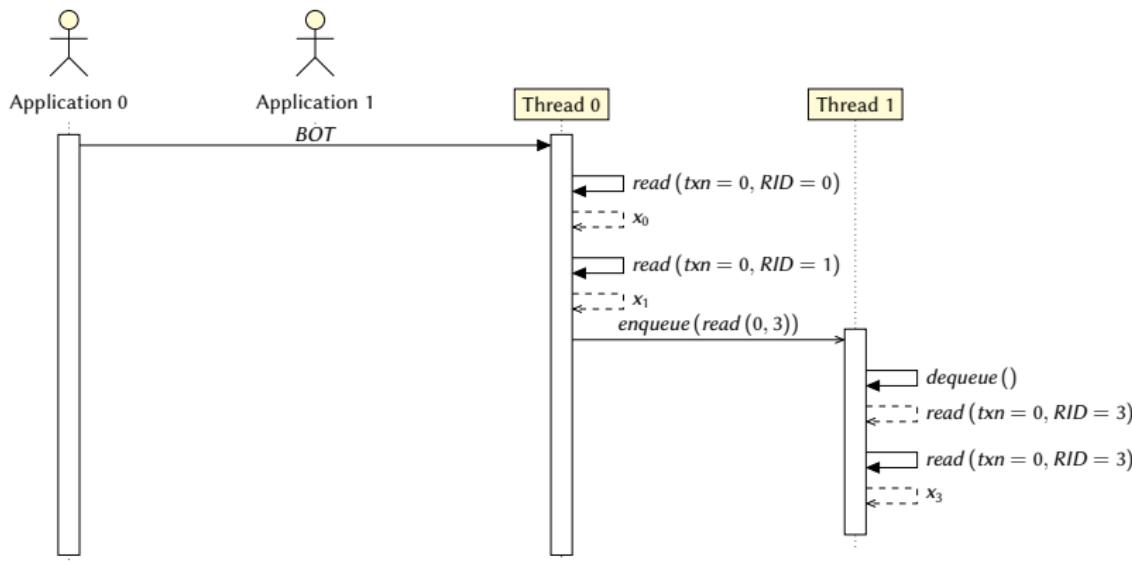
Example



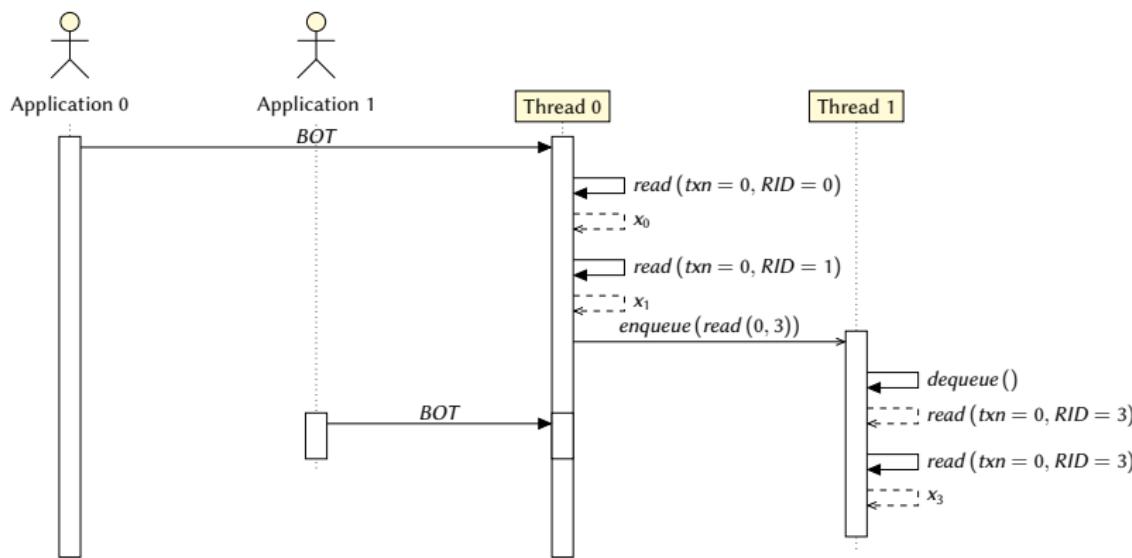
Example



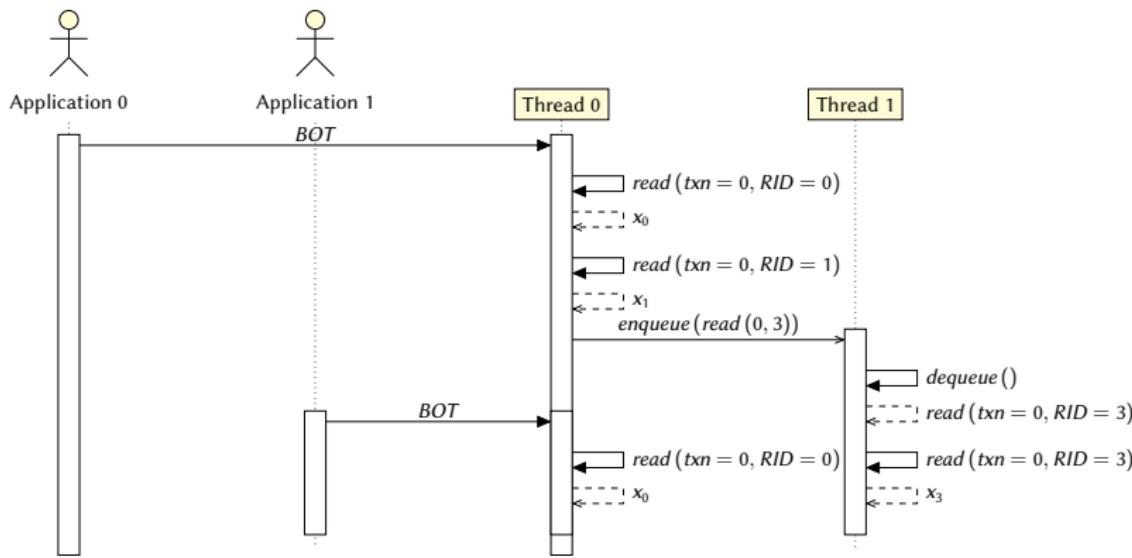
Example



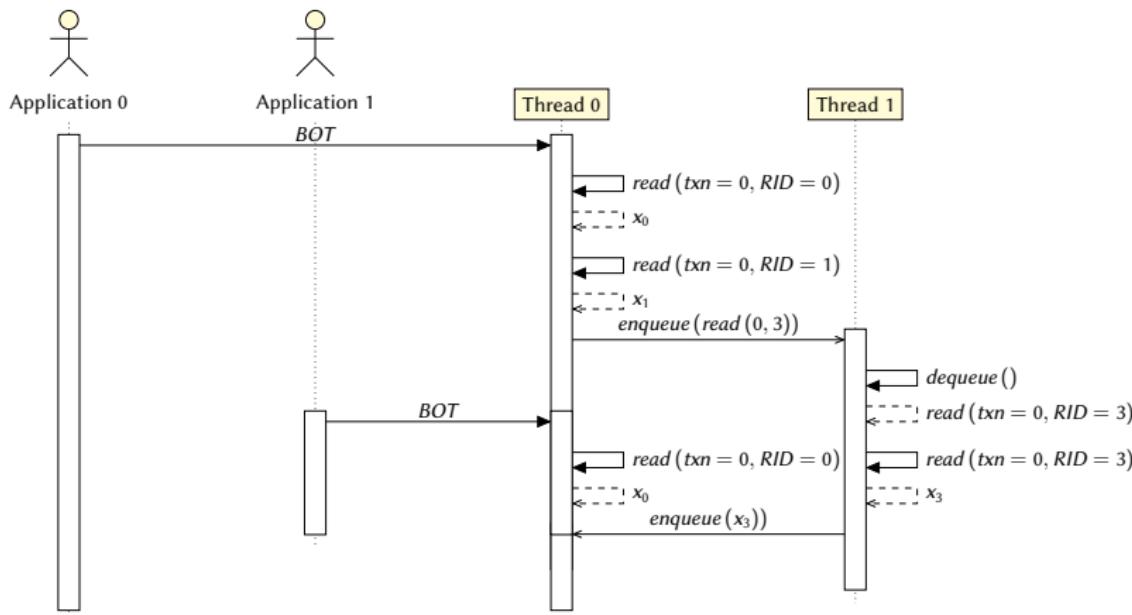
Example



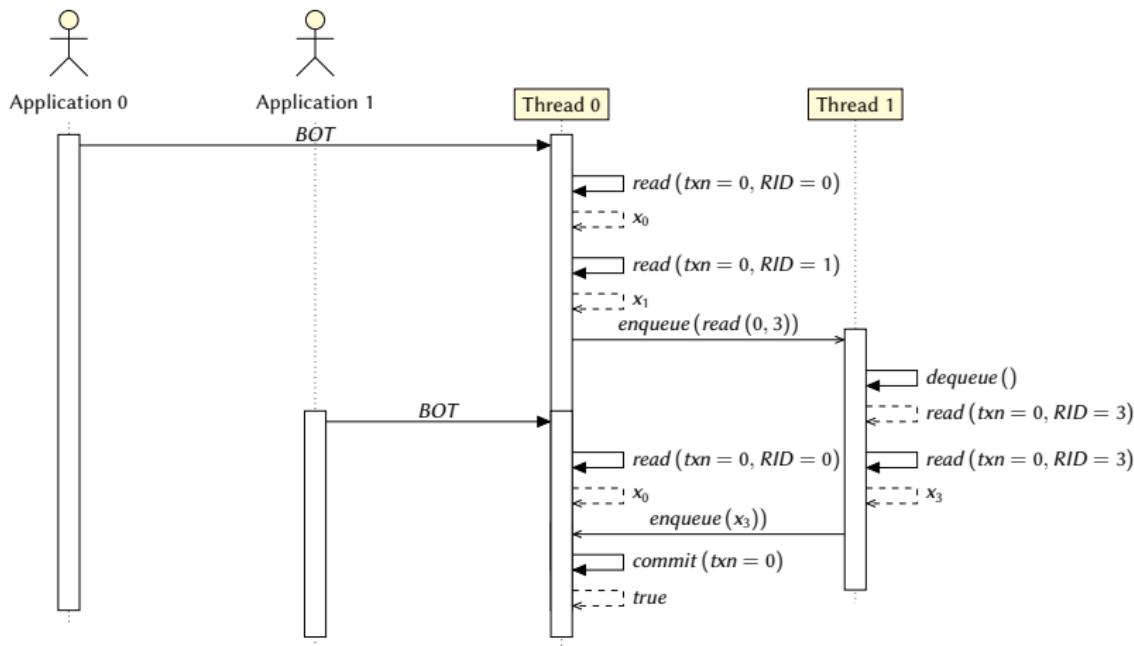
Example



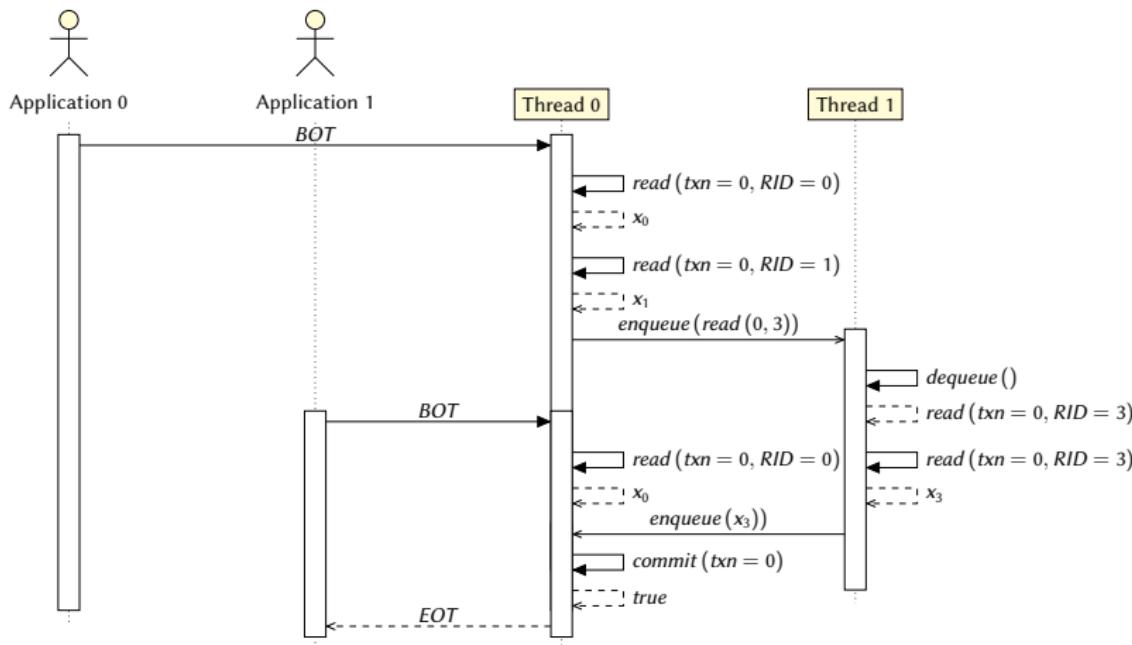
Example



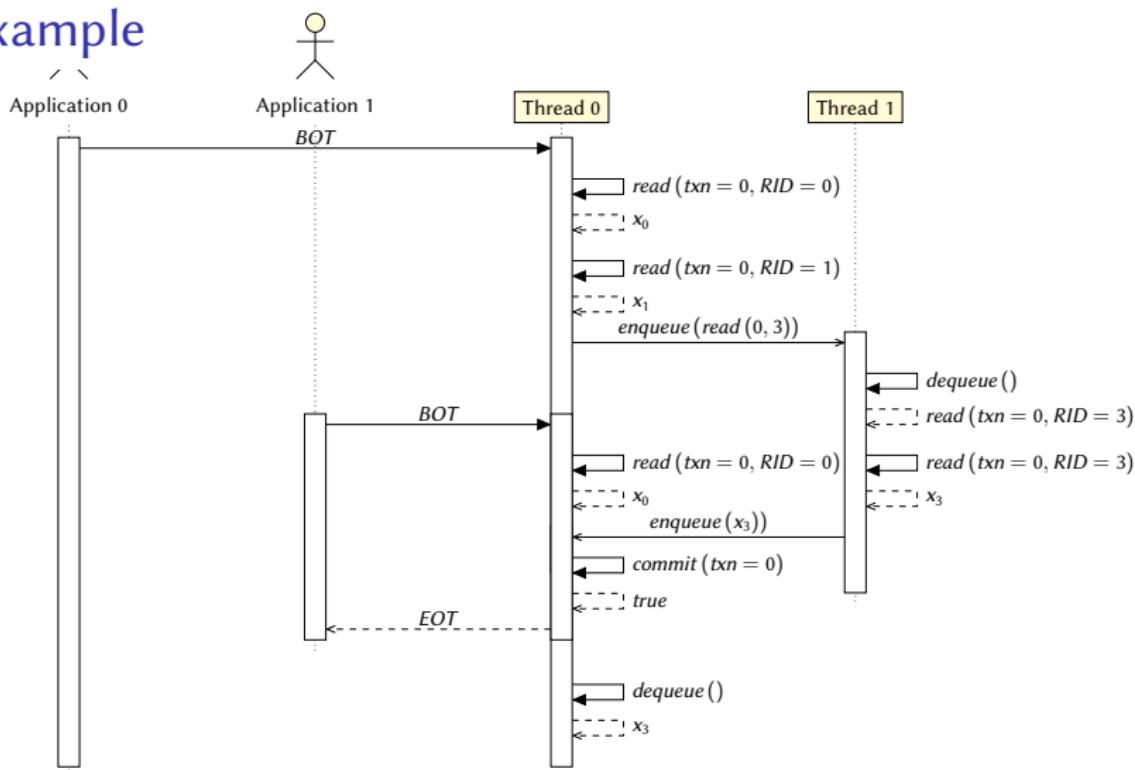
Example



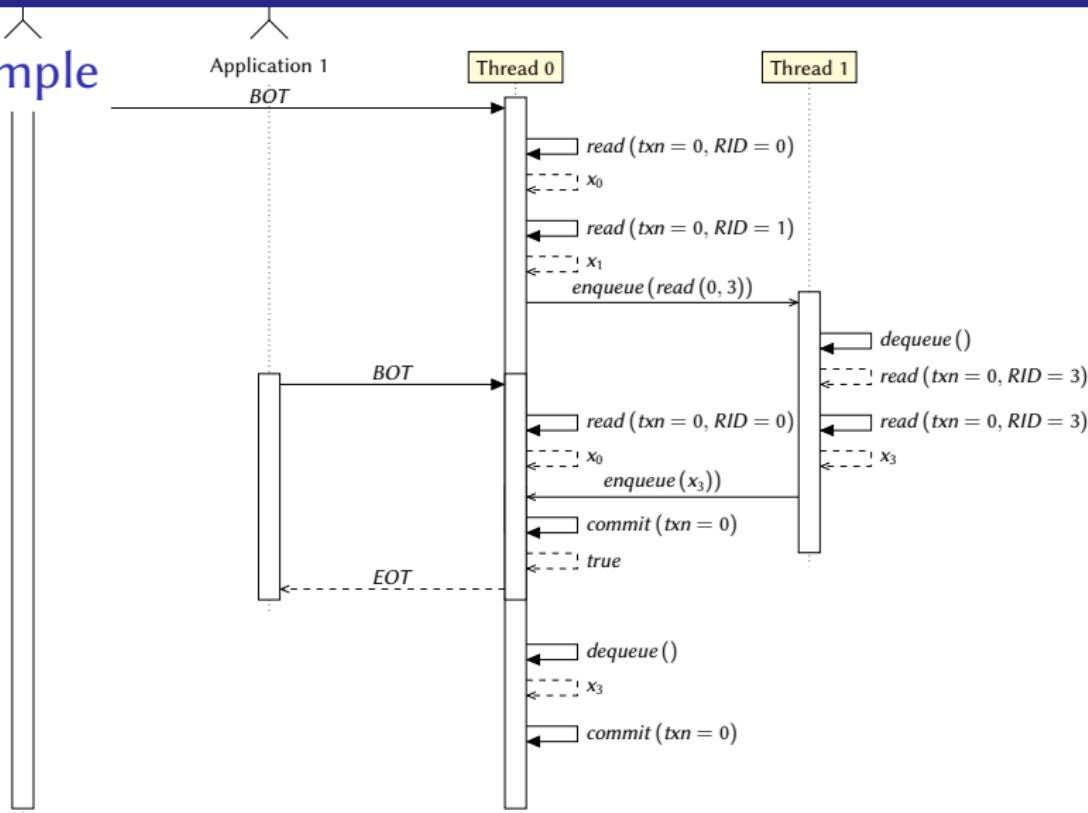
Example



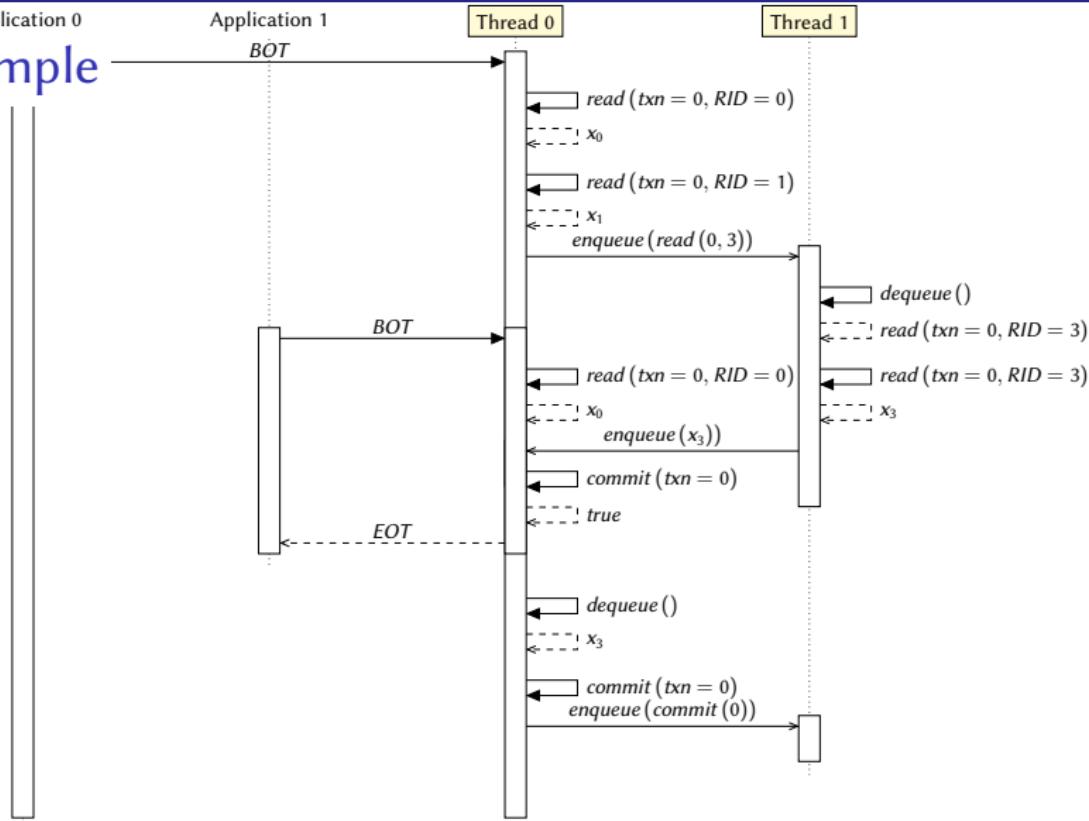
Example



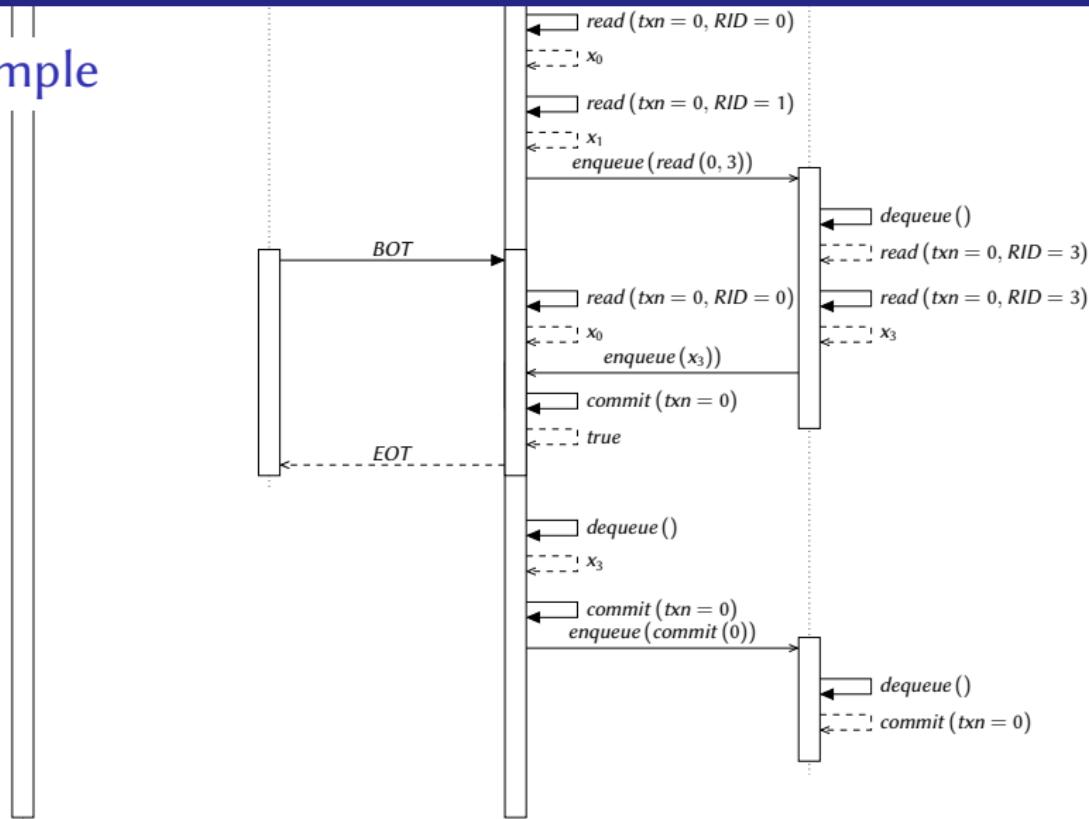
Example



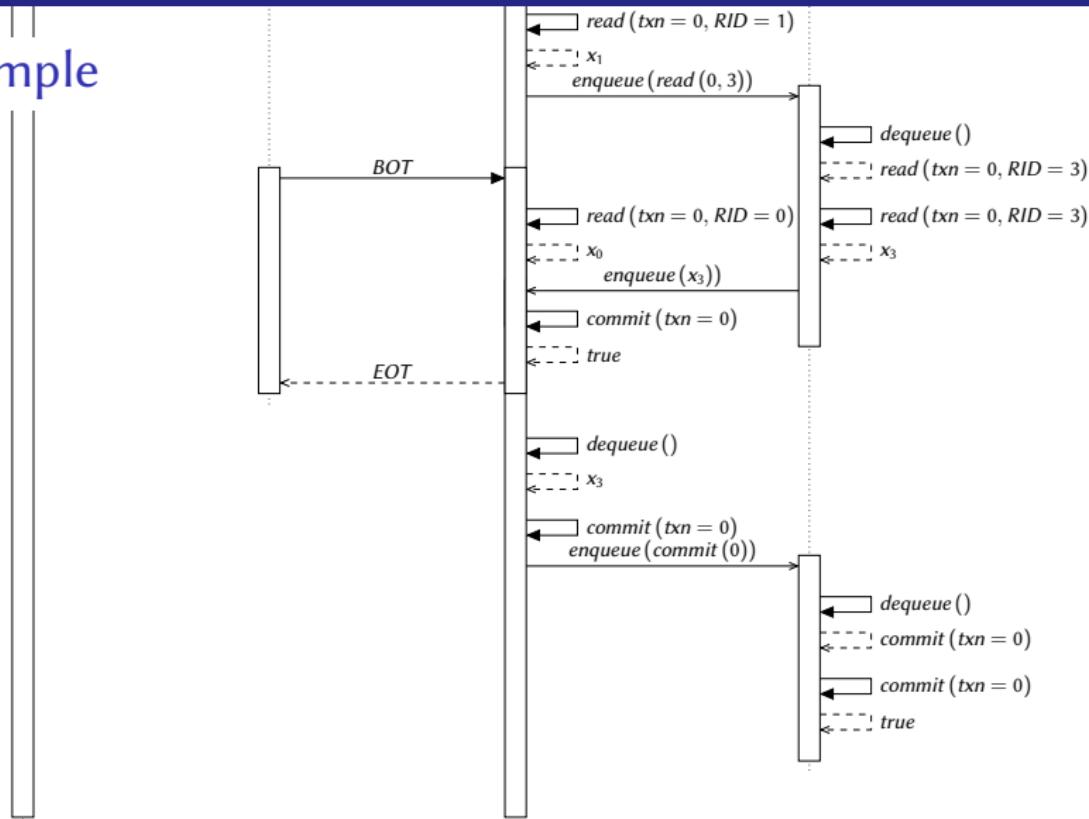
Example



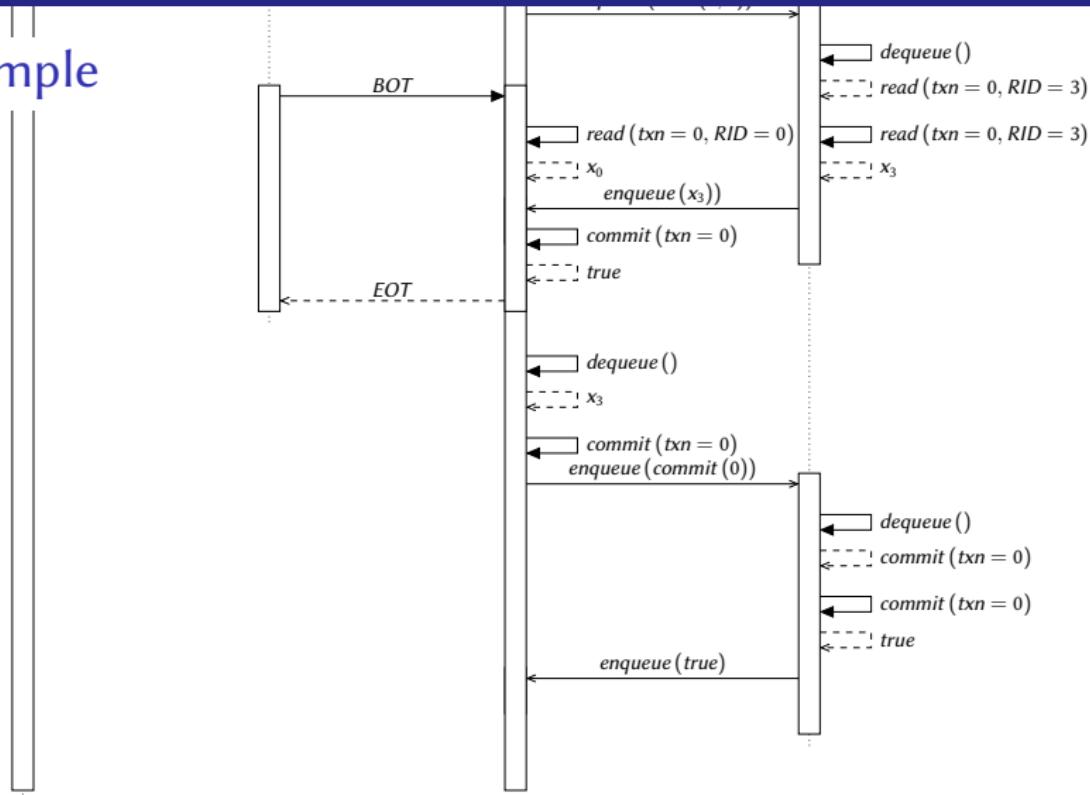
Example



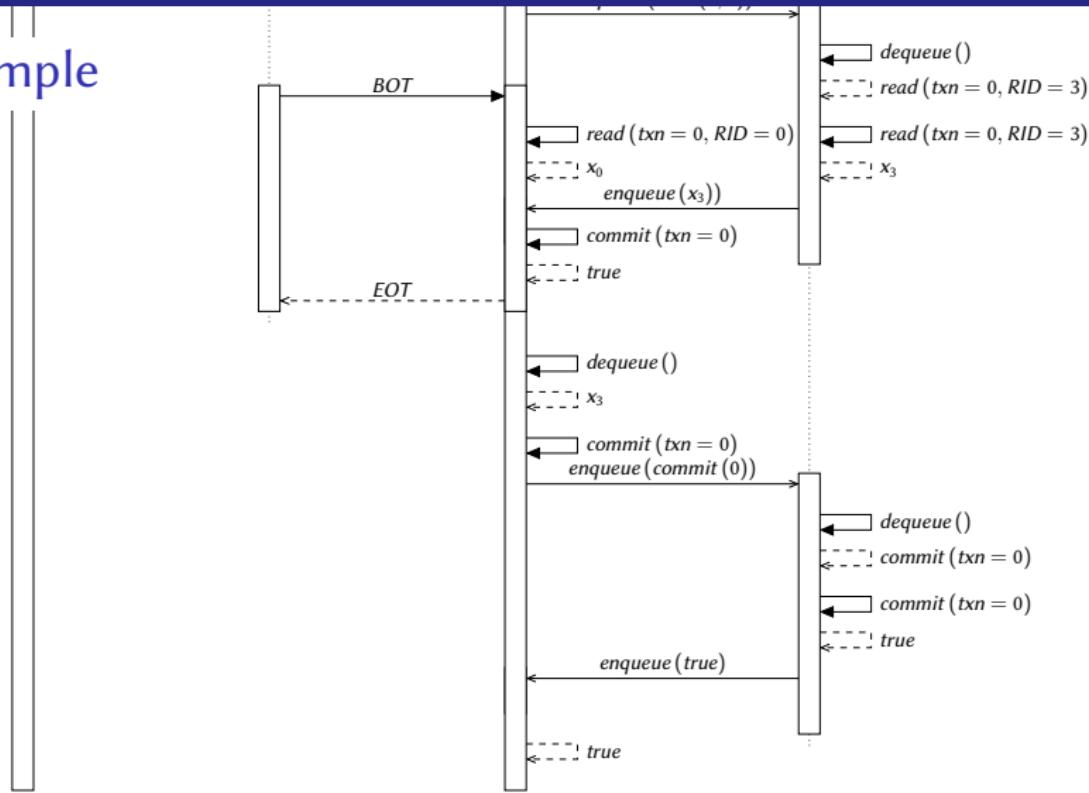
Example



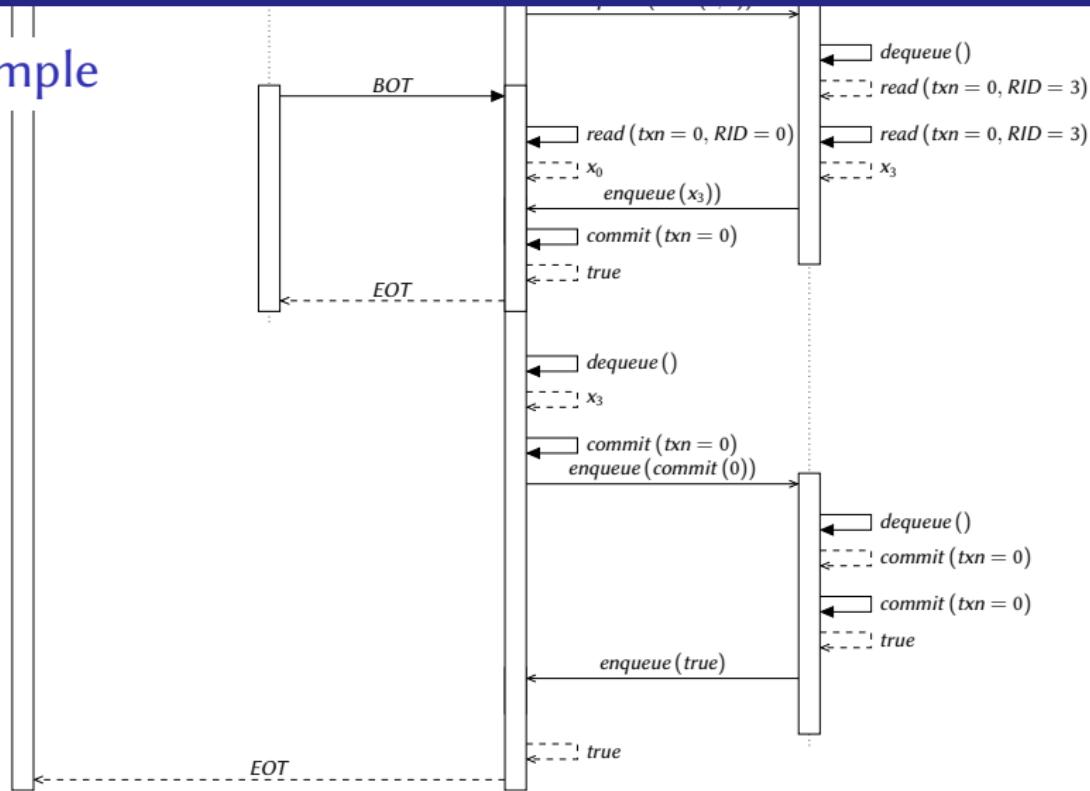
Example



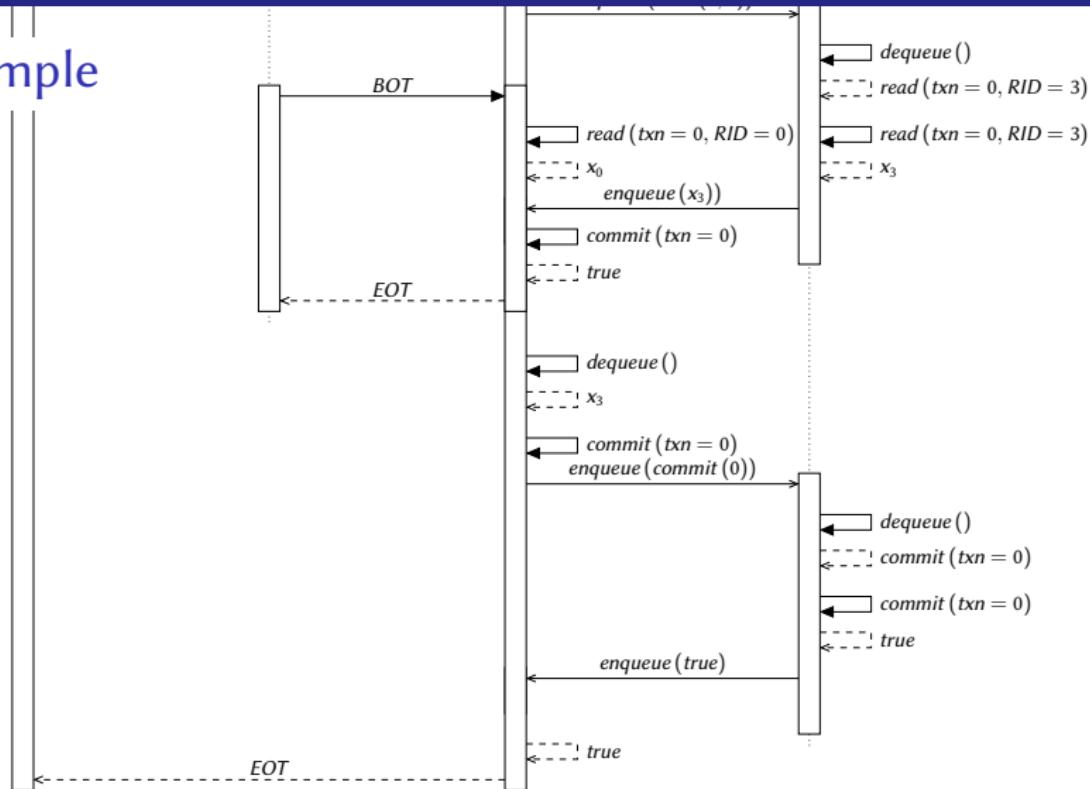
Example



Example



Example



Pros & Cons of Delegation

Pros & Cons of Delegation

- + only multi-site transactions cause a thread to access unowned records

Pros & Cons of Delegation

- + only multi-site transactions cause a thread to access unowned records
 - + each CPU cache usually only contains data of its partition → lower cache pollution

Pros & Cons of Delegation

- + only multi-site transactions cause a thread to access unowned records
 - + each CPU cache usually only contains data of its partition → lower cache pollution
 - + each CPU usually only accesses data of its partitions → fewer data movement between NUMA regions

Pros & Cons of Delegation

- + only multi-site transactions cause a thread to access unowned records
 - + each CPU cache usually only contains data of its partition → lower cache pollution
 - + each CPU usually only accesses data of its partitions → fewer data movement between NUMA regions
 - No physical synchronization required!

Pros & Cons of Delegation

- + only multi-site transactions cause a thread to access unowned records
 - + each CPU cache usually only contains data of its partition → lower cache pollution
 - + each CPU usually only accesses data of its partitions → fewer data movement between NUMA regions
 - No physical synchronization required!
- + message-passing required for multi-site transactions imposes only a low overhead

Pros & Cons of Delegation

- + only multi-site transactions cause a thread to access unowned records
 - + each CPU cache usually only contains data of its partition → lower cache pollution
 - + each CPU usually only accesses data of its partitions → fewer data movement between NUMA regions
 - No physical synchronization required!
- + message-passing required for multi-site transactions imposes only a low overhead
- partitioning required (e.g. manual selection of a partitioning strategy—*called routing rule*)

Pros & Cons of Delegation

- + only multi-site transactions cause a thread to access unowned records
 - + each CPU cache usually only contains data of its partition → lower cache pollution
 - + each CPU usually only accesses data of its partitions → fewer data movement between NUMA regions
 - No physical synchronization required!
- + message-passing required for multi-site transactions imposes only a low overhead
- partitioning required (e.g. manual selection of a partitioning strategy—*called routing rule*)
- partitioning is sensitive to the workload

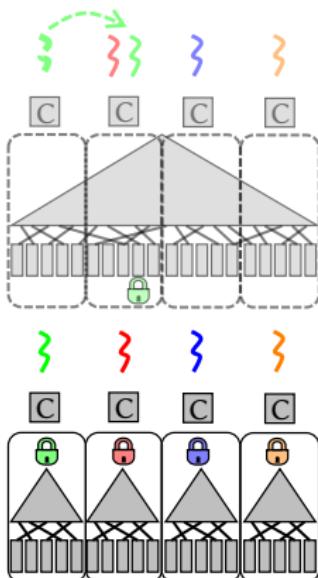
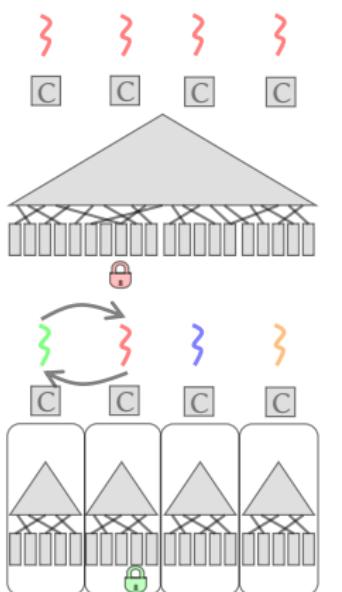
Pros & Cons of Delegation

- + only multi-site transactions cause a thread to access unowned records
 - + each CPU cache usually only contains data of its partition → lower cache pollution
 - + each CPU usually only accesses data of its partitions → fewer data movement between NUMA regions
 - No physical synchronization required!
- + message-passing required for multi-site transactions imposes only a low overhead
- partitioning required (e.g. manual selection of a partitioning strategy—*called routing rule*)
- partitioning is sensitive to the workload
- physical partitioning requires expensive repartitioning when the workload changes

Subsection 4

Partitioned Serial Execution (PSE)

Shared Everything/
Non-Partitioned
Delegation

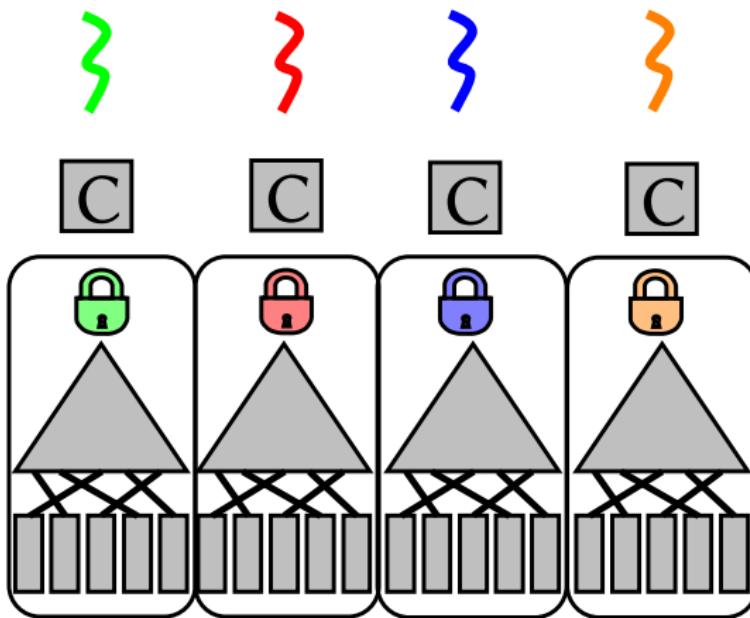


Data-
Oriented
Trans-
action
Execution
(DORA)

Partitioned
Serial
Execution
(PSE)

Subsection 4

Partitioned Serial Execution (PSE)



Properties of PSE

Properties of PSE

- ▶ data and associated metadata are physically partitioned → Shared-Nothing

Properties of PSE

- ▶ data and associated metadata are physically partitioned → Shared-Nothing
- ▶ partition-level locking using latches is used (only exclusive mode)

Properties of PSE

- ▶ data and associated metadata are physically partitioned → Shared-Nothing
- ▶ partition-level locking using latches is used (only exclusive mode)
- No concurrency control algorithm required!

Properties of PSE

- ▶ data and associated metadata are physically partitioned → Shared-Nothing
- ▶ partition-level locking using latches is used (only exclusive mode)
- No concurrency control algorithm required!
- ▶ single-site transactions run serially on the partition's thread (partition needs to be locked)

Properties of PSE

- ▶ data and associated metadata are physically partitioned → Shared-Nothing
- ▶ partition-level locking using latches is used (only exclusive mode)
- No concurrency control algorithm required!
- ▶ single-site transactions run serially on the partition's thread (partition needs to be locked)
- ▶ multi-site transactions require upfront lock of all relevant partitions/coordination between threads

Pros & Cons of PSE I

Pros & Cons of PSE I

- + only multi-site transactions cause a thread to access unowned records

Pros & Cons of PSE I

- + only multi-site transactions cause a thread to access unowned records
 - + each CPU cache usually only contains data of its partition → lower cache pollution
 - + partition-lock is usually in the CPU cache → synchronization imposes minor overhead

Pros & Cons of PSE I

- + only multi-site transactions cause a thread to access unowned records
 - + each CPU cache usually only contains data of its partition → lower cache pollution
 - + partition-lock is usually in the CPU cache → synchronization imposes minor overhead
 - + each CPU usually only accesses data of its partitions → fewer data movement between NUMA regions

Pros & Cons of PSE I

- + only multi-site transactions cause a thread to access unowned records
 - + each CPU cache usually only contains data of its partition → lower cache pollution
 - + partition-lock is usually in the CPU cache → synchronization imposes minor overhead
 - + each CPU usually only accesses data of its partitions → fewer data movement between NUMA regions
 - no physical synchronization beyond the partition-lock required

Pros & Cons of PSE I

- + only multi-site transactions cause a thread to access unowned records
 - + each CPU cache usually only contains data of its partition → lower cache pollution
 - + partition-lock is usually in the CPU cache → synchronization imposes minor overhead
 - + each CPU usually only accesses data of its partitions → fewer data movement between NUMA regions
 - no physical synchronization beyond the partition-lock required
- scales linearly for single-site transactions

Pros & Cons of PSE I

- + only multi-site transactions cause a thread to access unowned records
 - + each CPU cache usually only contains data of its partition → lower cache pollution
 - + partition-lock is usually in the CPU cache → synchronization imposes minor overhead
 - + each CPU usually only accesses data of its partitions → fewer data movement between NUMA regions
 - no physical synchronization beyond the partition-lock required
- scales linearly for single-site transactions
- multi-site transactions require the locking of all relevant partitions → decreases concurrency drastically

Pros & Cons of PSE I

- + only multi-site transactions cause a thread to access unowned records
 - + each CPU cache usually only contains data of its partition → lower cache pollution
 - + partition-lock is usually in the CPU cache → synchronization imposes minor overhead
 - + each CPU usually only accesses data of its partitions → fewer data movement between NUMA regions
 - no physical synchronization beyond the partition-lock required
- scales linearly for single-site transactions
- multi-site transactions require the locking of all relevant partitions → decreases concurrency drastically
- partitioning required (e.g. manual selection of a partitioning strategy—*called routing rule*)

Pros & Cons of PSE II

Pros & Cons of PSE II

- partitioning is sensitive to the workload

Pros & Cons of PSE II

- partitioning is sensitive to the workload
- physical partitioning requires expensive repartitioning when the workload changes

Pros & Cons of PSE II

- partitioning is sensitive to the workload
- physical partitioning requires expensive repartitioning when the workload changes
- coordination of multi-site transactions required

Summary

Architect- ture				
SE/NP				
PSE				
Dele- gation				
DORA				

Summary

Architect- ure	Process Management		
	Paral- lelism		
SE/NP	Shared Memory		
PSE	Shared Nothing		
Dele- gation	Message Passing		
DORA	Shared Memory		

Summary

Architect- ure	Process Management			
	Paral- lelism	Thread Assignment		
SE/NP	Shared Memory	thread-to-txn		
PSE	Shared Nothing	thread-to-txn		
Dele- gation	Message Passing	thread-to-txn		
DORA	Shared Memory	thread-to-data		

Summary

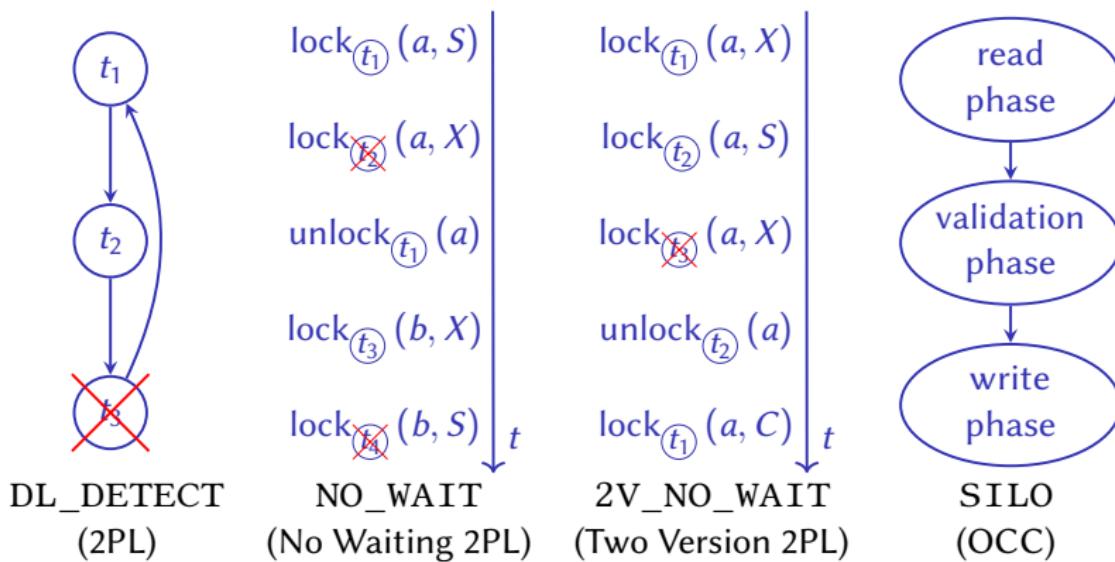
Architect- ture	Process Management		Transactional Storage Management	
	Paral- lelism	Thread Assignment	Logical Synchroniza- tion	
SE/NP	Shared Memory	thread-to-txn	CC Proto- cols	
PSE	Shared Nothing	thread-to-txn	Partition Lock	
Dele- gation	Message Passing	thread-to-txn	CC Proto- cols	
DORA	Shared Memory	thread-to-data	CC Proto- cols	

Summary

Architect- ture	Process Management		Transactional Storage Management	
	Paral- lelism	Thread Assignment	Logical Synchron- ization	Physical Synchron- ization
SE/NP	Shared Memory	thread-to-txn	CC Proto- cols	latch/- atomics
PSE	Shared Nothing	thread-to-txn	Partition Lock	partition lock
Dele- gation	Message Passing	thread-to-txn	CC Proto- cols	Message Passing
DORA	Shared Memory	thread-to-data	CC Proto- cols	Transaction Migration

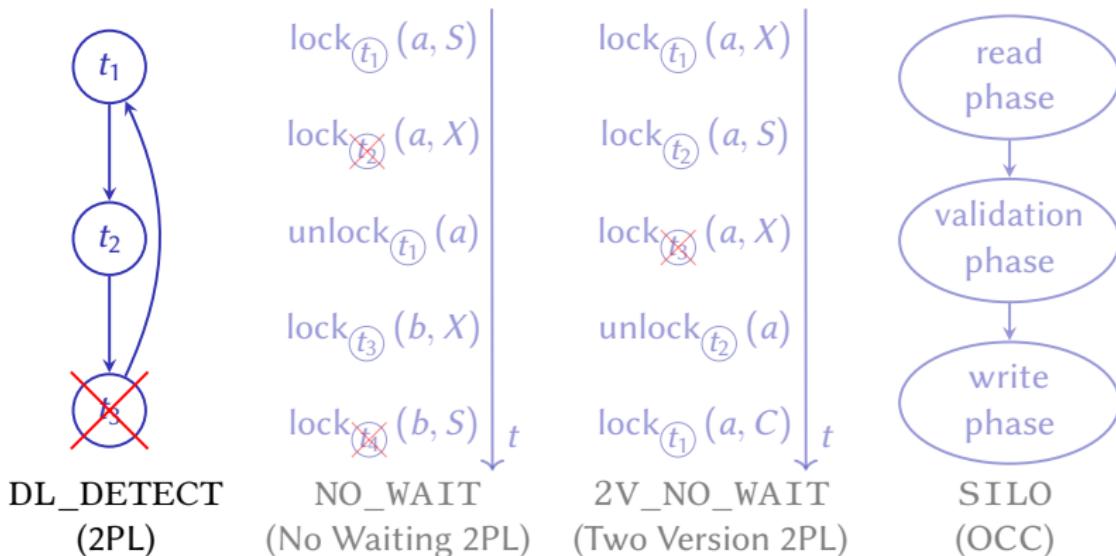
Section 3

Concurrency Control Algorithms



Subsection 1

DL_DETECT (2PL)



Properties of DL_DETECT (2PL) I

Properties of DL_DETECT (2PL) I

- ▶ pessimistic concurrency control protocol

Properties of DL_DETECT (2PL) I

- ▶ pessimistic concurrency control protocol
- ▶ transactions lock database objects (databases, tables, records, key ranges, etc.) before reading (shared mode S)

Properties of DL_DETECT (2PL) I

- ▶ 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

Properties of DL_DETECT (2PL) I

- ▶ 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
- ▶ if transaction t_0 tries to acquire a lock held by t_1 in a compatible mode → t_0 can immediately acquire the lock as well (starvation needs to be prevented)

Properties of DL_DETECT (2PL) I

- ▶ 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
- ▶ if transaction t_0 tries to acquire a lock held by t_1 in a compatible mode → t_0 can immediately acquire the lock as well (starvation needs to be prevented)
- ▶ if transaction t_0 tries to acquire a lock held by t_1 in an incompatible mode → t_0 waits until t_1 releases the lock

Properties of DL_DETECT (2PL) I

- ▶ 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
- ▶ if transaction t_0 tries to acquire a lock held by t_1 in a compatible mode $\rightarrow t_0$ can immediately acquire the lock as well (starvation needs to be prevented)
- ▶ if transaction t_0 tries to acquire a lock held by t_1 in an incompatible mode $\rightarrow t_0$ waits until t_1 releases the lock

compatibility	shared mode	exclusive mode
shared mode		
exclusive mode		

Properties of DL_DETECT (2PL) II

Properties of DL_DETECT (2PL) II

- ▶ deadlock detection using a repeatedly generated and analyzed wait-for graph

Properties of DL_DETECT (2PL) II

- ▶ deadlock detection using a repeatedly generated and analyzed wait-for graph
- ▶ intention lock modes (*IS*, *IX*, *SIX*) required for hierarchical locking (database \leftarrow table \leftarrow key range \leftarrow record)

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:

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:



Example

Transactions:

$t_0 \quad t_1 \quad t_2$
| BOT
 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:



Example

Transactions:

t_0	t_1	t_2
\sqcup r_0		
	— BOT	

Locks:

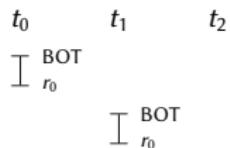
Record 0		Record 1		Record 2		...
Current Mode:	S (1)	Current Mode:	NL	Current Mode:	NL	
Waiters:		Waiters:		Waiters:		
Data:	x_0	Data:	x_1	Data:	x_2	

Wait-for Graph:



Example

Transactions:



Locks:

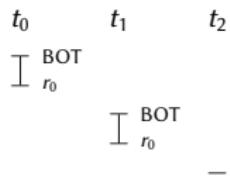
Record 0		Record 1		Record 2		...
Current Mode:	S (2)	Current Mode:	NL	Current Mode:	NL	
Waiters:		Waiters:		Waiters:		
Data:	x_0	Data:	x_1	Data:	x_2	

Wait-for Graph:



Example

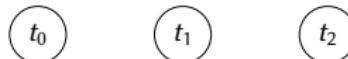
Transactions:



Locks:

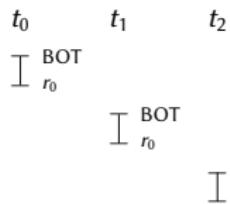
Record 0		Record 1		Record 2		...
Current Mode:	S (2)	Current Mode:	NL	Current Mode:	NL	
Waiters:		Waiters:		Waiters:		
Data:	x_0	Data:	x_1	Data:	x_2	

Wait-for Graph:



Example

Transactions:



Locks:

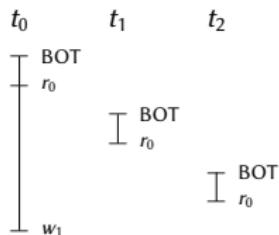
Record 0		Record 1		Record 2		...
Current Mode:	S (3)	Current Mode:	NL	Current Mode:	NL	
Waiters:		Waiters:		Waiters:		
Data:	x_0	Data:	x_1	Data:	x_2	

Wait-for Graph:



Example

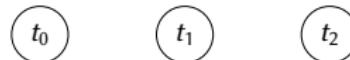
Transactions:



Locks:

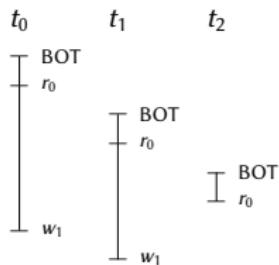
Record 0		Record 1		Record 2		...
Current Mode:	S (3)	Current Mode:	X (t_0) <th>Current Mode:</th> <td>NL</td> <th></th>	Current Mode:	NL	
Waiters:		Waiters:		Waiters:		
Data:	x_0	Data:	x_1	Data:	x_2	

Wait-for Graph:



Example

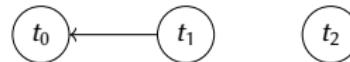
Transactions:



Locks:

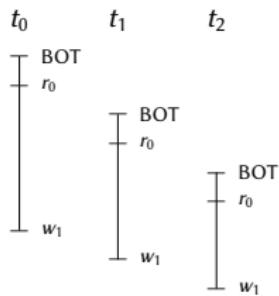
Record 0		Record 1		Record 2		...
Current Mode:	S (3)	Current Mode:	X (t_0)	Current Mode:	NL	
Waiters:		Waiters:	(X, t_1) <th>Waiters:</th> <td></td> <th></th>	Waiters:		
Data:	x_0	Data:	x'_1 <th>Data:</th> <td>x_2</td> <th></th>	Data:	x_2	

Wait-for Graph:



Example

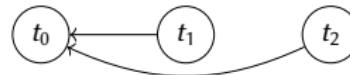
Transactions:



Locks:

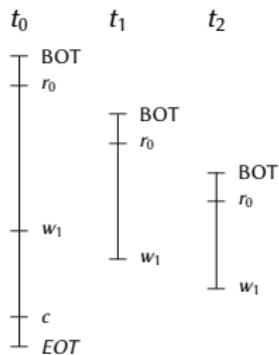
Record 0		Record 1		Record 2	...
Current Mode:	S (3)	Current Mode:	X (t_0)	Current Mode:	NL
Waiters:		Waiters:	(X, t_1) (X, t_2)	Waiters:	
Data:	x_0	Data:	x'_1 <th>Data:</th> <td>x_2</td>	Data:	x_2

Wait-for Graph:



Example

Transactions:



Locks:

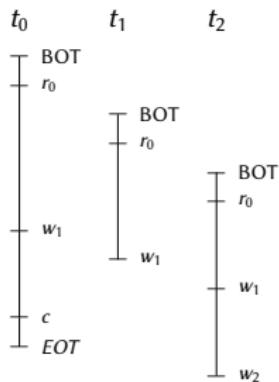
Record 0		Record 1		Record 2		...
Current Mode:	S (2)	Current Mode:	X (t_1)	Current Mode:	NL	
Waiters:		Waiters:	(X, t_2)	Waiters:		
Data:	x_0	Data:	x'_1	Data:	x_2	

Wait-for Graph:



Example

Transactions:



Locks:

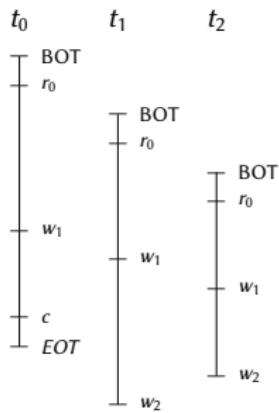
Record 0		Record 1		Record 2		...
Current Mode:	S (2)	Current Mode:	X (t_1)	Current Mode:	X (t_2) <th></th>	
Waiters:		Waiters:	(X, t_2) <th>Waiters:</th> <td></td> <th></th>	Waiters:		
Data:	x_0	Data:	x_1''	Data:	x_2	

Wait-for Graph:



Example

Transactions:



Locks:

Record 0		Record 1		Record 2		...
Current Mode:	S (2)	Current Mode:	X (t_1)	Current Mode:	X (t_2) <th></th>	
Waiters:		Waiters:	(X, t_2)	Waiters:	(X, t_1) <th></th>	
Data:	x_0	Data:	x_1''	Data:	x_2'	

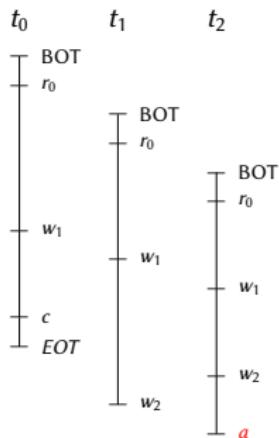
Wait-for Graph:



Cycle → Deadlock → Rollback one Transaction

Example

Transactions:



Locks:

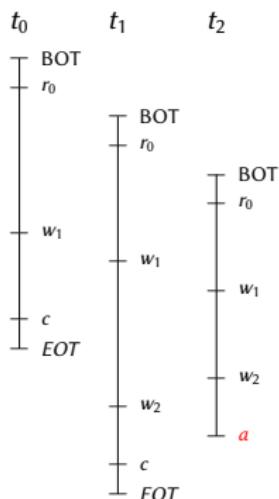
Record 0		Record 1		Record 2		...
Current Mode:	S (1)	Current Mode:	X (t_1)	Current Mode:	X (t_1) <th></th>	
Waiters:		Waiters:		Waiters:	<th></th>	
Data:	x_0	Data:	x''_1	Data:	x_2 <th></th>	

Wait-for Graph:



Example

Transactions:



Locks:

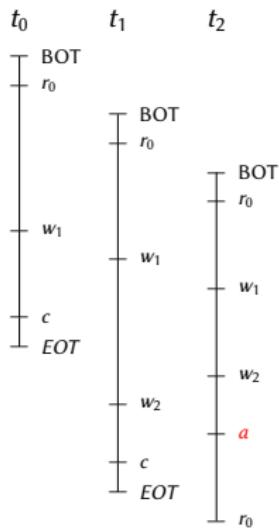
Record 0		Record 1		Record 2		...
Current Mode:	NL	Current Mode:	NL	Current Mode:	NL	
Waiters:		Waiters:		Waiters:		
Data:	x_0	Data:	x_1''	Data:	x_2''	

Wait-for Graph:



Example

Transactions:



Locks:

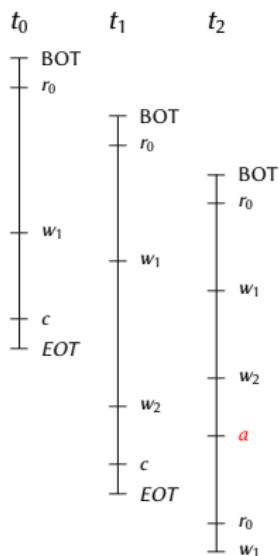
Record 0		Record 1		Record 2		...
Current Mode:	S (1)	Current Mode:	NL	Current Mode:	NL	
Waiters:		Waiters:		Waiters:		
Data:	x_0	Data:	x_1''	Data:	x_2''	

Wait-for Graph:



Example

Transactions:



Locks:

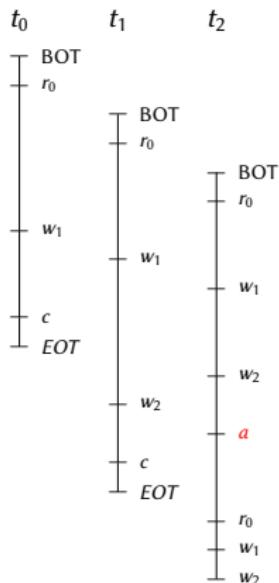
Record 0		Record 1		Record 2		...
Current Mode:	S (1)	Current Mode:	X (t_2)	Current Mode:	NL	
Waiters:		Waiters:		Waiters:		
Data:	x_0	Data:	x_1''	Data:	x_2''	

Wait-for Graph:



Example

Transactions:



Locks:

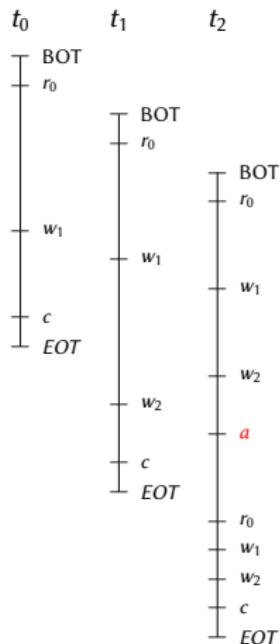
Record 0		Record 1		Record 2		...
Current Mode:	S (1)	Current Mode:	X (t_2)	Current Mode:	X (t_2) <th></th>	
Waiters:		Waiters:		Waiters:	<th></th>	
Data:	x_0	Data:	x_1'''	Data:	x_2''	

Wait-for Graph:



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)

Pros & Cons of DL_DETECT (2PL)

- + aborts only after deadlocks

Pros & Cons of DL_DETECT (2PL)

- + aborts only after deadlocks
- deadlocks are possible

Pros & Cons of DL_DETECT (2PL)

- + aborts only after deadlocks
- deadlocks are possible
- locks prevent concurrency too often (e.g. blind writes)

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

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

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

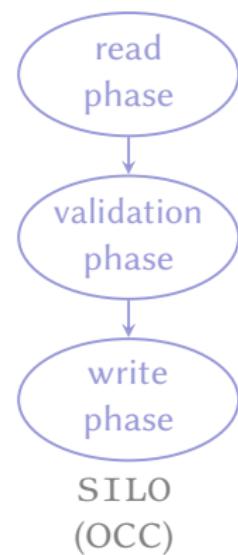
Pros & Cons of DL_DETECT (2PL)

- + 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)

<p>DL_DETECT (2PL)</p>	$\text{lock}_{t_1}(a, S)$ $\text{lock}_{\cancel{t_3}}(a, X)$ $\text{unlock}_{t_1}(a)$ $\text{lock}_{t_3}(b, X)$ $\text{lock}_{\cancel{t_3}}(b, S)$	$\text{lock}_{t_1}(a, X)$ $\text{lock}_{t_2}(a, S)$ $\text{lock}_{\cancel{t_3}}(a, X)$ $\text{unlock}_{t_2}(a)$ $\text{lock}_{t_1}(a, C)$	t
NO_WAIT (No Waiting 2PL)		2V_NO_WAIT (Two Version 2PL)	



Properties of NO_WAIT (No Waiting 2PL) I

Properties of NO_WAIT (No Waiting 2PL) I

- ▶ pessimistic concurrency control protocol

Properties of NO_WAIT (No Waiting 2PL) I

- ▶ pessimistic concurrency control protocol
- ▶ transactions lock database objects (databases, tables, records, key ranges, etc.) before reading (shared mode S)

Properties of NO_WAIT (No Waiting 2PL) I

- ▶ 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

Properties of NO_WAIT (No Waiting 2PL) I

- ▶ 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
- ▶ if transaction t_0 tries to acquire a lock held by t_1 in a compatible mode → t_0 can immediately acquire the lock as well (starvation automatically prevented)

Properties of NO_WAIT (No Waiting 2PL) I

- ▶ 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
- ▶ if transaction t_0 tries to acquire a lock held by t_1 in a compatible mode → t_0 can immediately acquire the lock as well (starvation automatically prevented)
- ▶ if transaction t_0 tries to acquire a lock held by t_1 in a incompatible mode → t_0 aborts

Properties of NO_WAIT (No Waiting 2PL) I

- ▶ 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
- ▶ if transaction t_0 tries to acquire a lock held by t_1 in a compatible mode $\rightarrow t_0$ can immediately acquire the lock as well (starvation automatically prevented)
- ▶ if transaction t_0 tries to acquire a lock held by t_1 in a incompatible mode $\rightarrow t_0$ aborts

compatibility	shared mode	exclusive mode
shared mode		
exclusive mode		

Properties of NO_WAIT (No Waiting 2PL) II

Properties of NO_WAIT (No Waiting 2PL) II

- ▶ intention lock modes (IS , IX , SIX) required for hierarchical locking (database \leftarrow table \leftarrow key range \leftarrow record)

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

Example

Transactions:

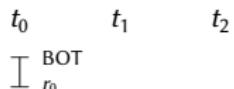
$t_0 \quad t_1 \quad t_2$
— BOT

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	

Example

Transactions:

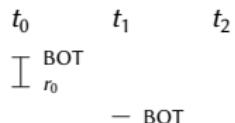


Locks:

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

Example

Transactions:

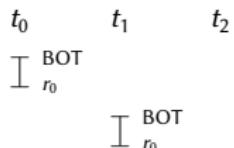


Locks:

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

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	

Example

Transactions:

$t_0 \quad t_1 \quad t_2$

$\begin{array}{|c|} \hline \text{BOT} \\ \hline r_0 \\ \hline \end{array}$

$\begin{array}{|c|} \hline \text{BOT} \\ \hline r_0 \\ \hline \end{array}$

— BOT

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	

Example

Transactions:

$t_0 \quad t_1 \quad t_2$

$\begin{array}{|c|} \hline \text{BOT} \\ \hline r_0 \\ \hline \end{array}$

$\begin{array}{|c|} \hline \text{BOT} \\ \hline r_0 \\ \hline \end{array}$

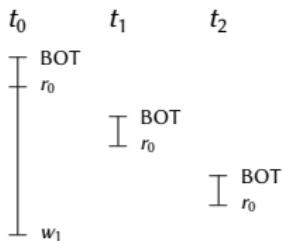
$\begin{array}{|c|} \hline \text{BOT} \\ \hline r_0 \\ \hline \end{array}$

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	

Example

Transactions:

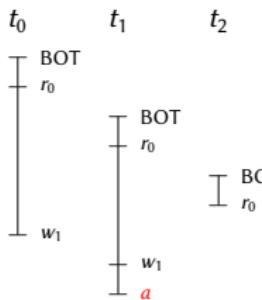


Locks:

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

Example

Transactions:

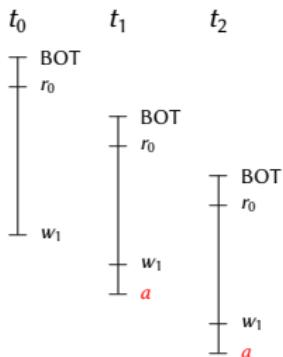


Locks:

Record 0		Record 1		Record 2		...
Current Mode:	4	Current Mode:	1	Current Mode:	0	
Data:	x_0	Data:	x'_1	Data:	x_2	

Example

Transactions:

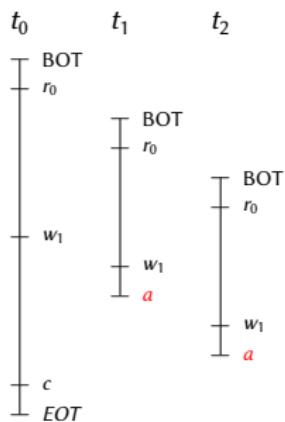


Locks:

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

Example

Transactions:

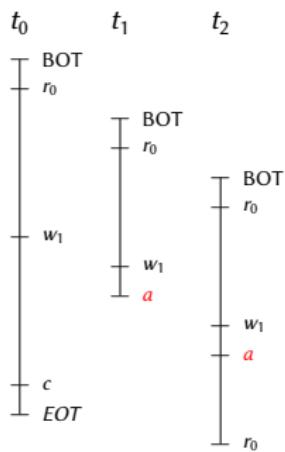


Locks:

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

Example

Transactions:

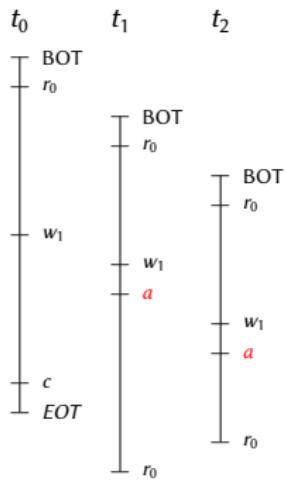


Locks:

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

Example

Transactions:

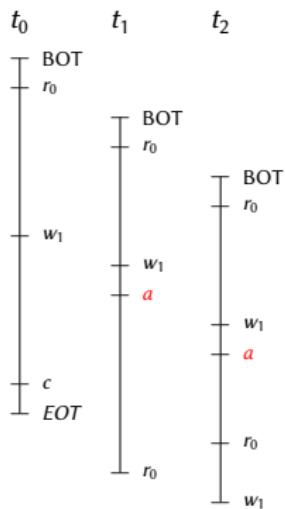


Locks:

Record 0		Record 1		Record 2		...
Current Mode:	4	Current Mode:	0	Current Mode:	0	
Data:	x_0	Data:	x'_1	Data:	x_2	

Example

Transactions:

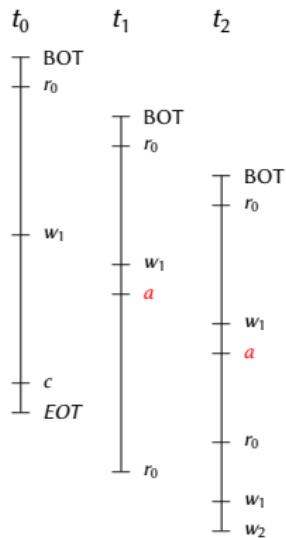


Locks:

Record 0		Record 1		Record 2		...
Current Mode:	4	Current Mode:	1	Current Mode:	0	
Data:	x_0	Data:	x'_1	Data:	x_2	

Example

Transactions:

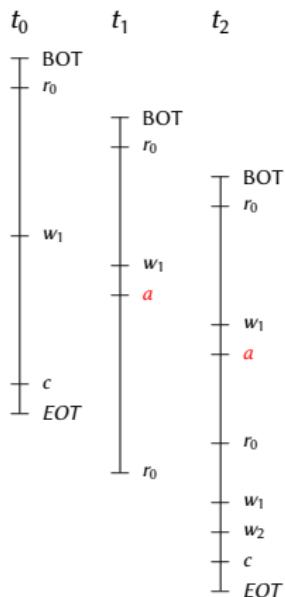


Locks:

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

Example

Transactions:

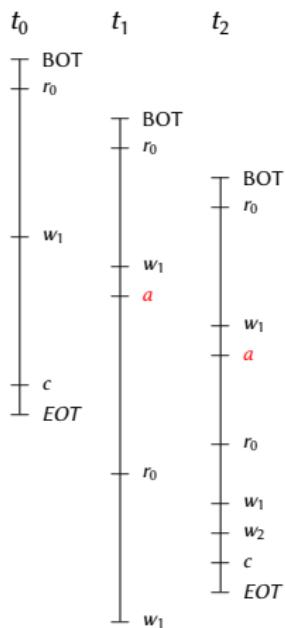


Locks:

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

Example

Transactions:

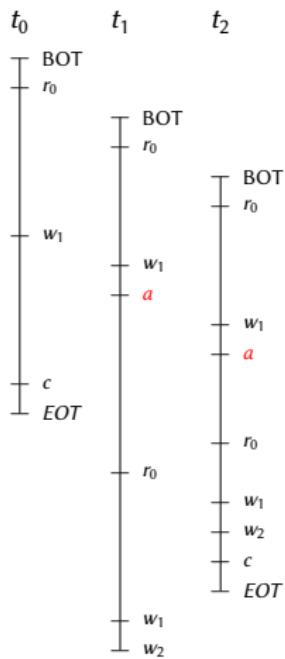


Locks:

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

Example

Transactions:

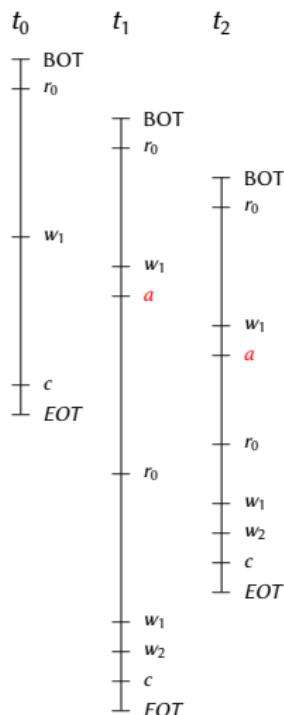


Locks:

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

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)

Pros & Cons of NO_WAIT (No Waiting 2PL)

- + deadlocks are impossible

Pros & Cons of NO_WAIT (No Waiting 2PL)

- + deadlocks are impossible
- + locks can be implemented using a semaphore updated with atomic instructions → scales better than latches

Pros & Cons of NO_WAIT (No Waiting 2PL)

- + deadlocks are impossible
- + locks can be implemented using a semaphore updated with atomic instructions → scales better than latches
- + no need to expensively calculate and analysis a wait-for graph

Pros & Cons of NO_WAIT (No Waiting 2PL)

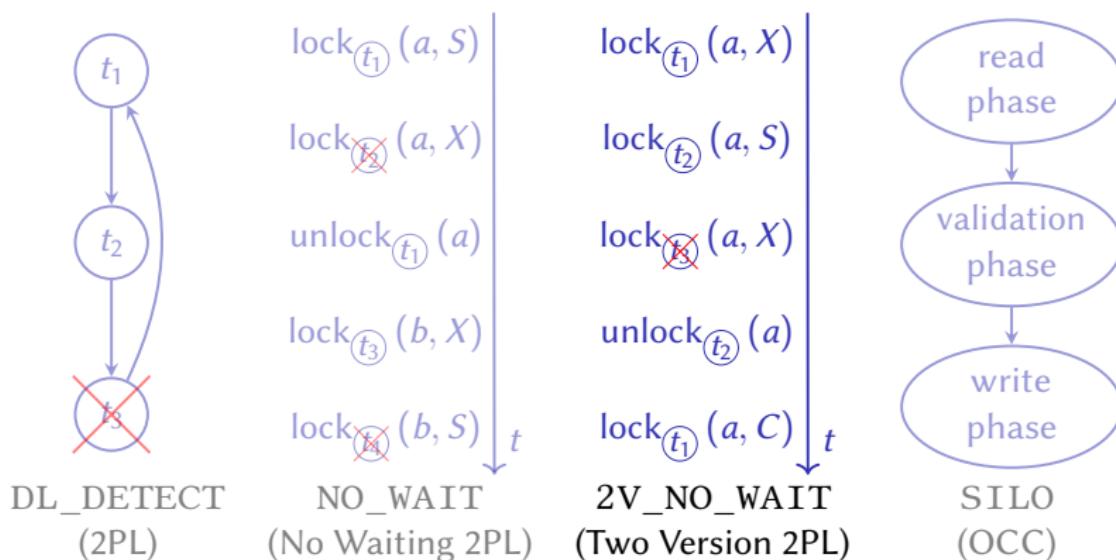
- + deadlocks are impossible
- + locks can be implemented using a semaphore updated with atomic instructions → scales better than latches
- + no need to expensively calculate and analysis a wait-for graph
- update-intensive high contention workloads result in many lock conflicts → many aborts → work done before needs to be repeated

Pros & Cons of NO_WAIT (No Waiting 2PL)

- + deadlocks are impossible
- + locks can be implemented using a semaphore updated with atomic instructions → scales better than latches
- + no need to expensively calculate and analysis a wait-for graph
- update-intensive high contention workloads result in many lock conflicts → 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)



Properties of 2V_NO_WAIT (Two Version 2PL) I

Properties of 2V_NO_WAIT (Two Version 2PL) I

- ▶ multiversion pessimistic concurrency control protocol

Properties of 2V_NO_WAIT (Two Version 2PL) I

- ▶ multiversion pessimistic concurrency control protocol
- ▶ 3 phases: read, certify, write/commit

Properties of 2V_NO_WAIT (Two Version 2PL) I

- ▶ multiversion pessimistic concurrency control protocol
- ▶ 3 phases: read, certify, write/commit
- ▶ transactions lock records before reading (shared mode S)

Properties of 2V_NO_WAIT (Two Version 2PL) I

- ▶ multiversion pessimistic concurrency control protocol
- ▶ 3 phases: read, certify, write/commit
- ▶ transactions lock records before reading (shared mode S), privately updating (exclusive mode X)

Properties of 2V_NO_WAIT (Two Version 2PL) I

- ▶ multiversion pessimistic concurrency control protocol
- ▶ 3 phases: read, certify, write/commit
- ▶ transactions lock records before reading (shared mode S), privately updating (exclusive mode X) or certifying/globally updating (certify mode C) them

Properties of 2V_NO_WAIT (Two Version 2PL) I

- ▶ multiversion pessimistic concurrency control protocol
- ▶ 3 phases: read, certify, write/commit
- ▶ transactions lock records before reading (shared mode S), privately updating (exclusive mode X) or certifying/globally updating (certify mode C) them
- ▶ updates happen first on a private copy → committed copy can still be read by other transactions

Properties of 2V_NO_WAIT (Two Version 2PL) I

- ▶ multiversion pessimistic concurrency control protocol
- ▶ 3 phases: read, certify, write/commit
- ▶ transactions lock records before reading (shared mode S), privately updating (exclusive mode X) or certifying/globally updating (certify mode C) them
- ▶ updates happen first on a private copy → committed copy can still be read by other transactions
- ▶ before committing writes (replace original record with private copy) the absence of relevant conflicts needs to be certified (certification step)

Properties of 2V_NO_WAIT (Two Version 2PL) II

Properties of 2V_NO_WAIT (Two Version 2PL) II

- ▶ if transaction t_0 tries to acquire a lock held by t_1 in a compatible mode → t_0 can immediately acquire the lock as well (starvation automatically prevented)

Properties of 2V_NO_WAIT (Two Version 2PL) II

- ▶ if transaction t_0 tries to acquire a lock held by t_1 in a compatible mode → t_0 can immediately acquire the lock as well (starvation automatically prevented)
- ▶ if transaction t_0 tries to acquire a lock held by t_1 in a incompatible mode → t_0 aborts

Properties of 2V_NO_WAIT (Two Version 2PL) II

- ▶ if transaction t_0 tries to acquire a lock held by t_1 in a compatible mode → t_0 can immediately acquire the lock as well (starvation automatically prevented)
- ▶ if transaction t_0 tries to acquire a lock held by t_1 in a incompatible mode → t_0 aborts

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

Protocol I

Protocol I

read r_i

Protocol I

read r_i

C acquired r_i gets certified or committed by another transaction

Protocol I

read r_i

- C acquired r_i gets certified or committed by another transaction
- ▶ abort this transaction

Protocol I

read r_i

C acquired r_i gets certified or committed by another transaction
▶ abort this transaction

C not acquired other threads might read r_i or privately update r_i

Protocol I

read r_i

C acquired r_i gets certified or committed by another transaction

- ▶ abort this transaction

C not acquired other threads might read r_i or privately update r_i

- ▶ acquire r_i 's lock in S mode

Protocol I

read r_i

- C acquired r_i gets certified or committed by another transaction
 - ▶ abort this transaction

- C not acquired other threads might read r_i or privately update r_i
 - ▶ acquire r_i 's lock in S mode
 - ▶ read global (committed) value of r_i

Protocol I

read r_i

- C acquired r_i gets certified or committed by another transaction
- ▶ abort this transaction

- C not acquired other threads might read r_i or privately update r_i
- ▶ acquire r_i 's lock in S mode
 - ▶ read global (committed) value of r_i

update r_i

Protocol I

read r_i

- C acquired r_i gets certified or committed by another transaction
- ▶ abort this transaction

- C not acquired other threads might read r_i or privately update r_i
- ▶ acquire r_i 's lock in S mode
 - ▶ read global (committed) value of r_i

update r_i

- W acquired r_i gets (privately or globally) updated by another transaction → there are already two versions

Protocol I

read r_i

- C acquired r_i gets certified or committed by another transaction
- ▶ abort this transaction

- C not acquired other threads might read r_i or privately update r_i
- ▶ acquire r_i 's lock in S mode
 - ▶ read global (committed) value of r_i

update r_i

- W acquired r_i gets (privately or globally) updated by another transaction → there are already two versions
- ▶ abort this transaction

Protocol I

read r_i

- C acquired r_i gets certified or committed by another transaction
- ▶ abort this transaction

- C not acquired other threads might read r_i or privately update r_i
- ▶ acquire r_i 's lock in S mode
 - ▶ read global (committed) value of r_i

update r_i

- W acquired r_i gets (privately or globally) updated by another transaction → there are already two versions
- ▶ abort this transaction

- W not acquired other threads at most read r_i

Protocol I

read r_i

- C acquired r_i gets certified or committed by another transaction
- ▶ abort this transaction

- C not acquired other threads might read r_i or privately update r_i
- ▶ acquire r_i 's lock in S mode
 - ▶ read global (committed) value of r_i

update r_i

- W acquired r_i gets (privately or globally) updated by another transaction → there are already two versions
- ▶ abort this transaction

- W not acquired other threads at most read r_i
- ▶ acquire r_i 's lock in X mode

Protocol I

read r_i

- C acquired r_i gets certified or committed by another transaction
- ▶ abort this transaction

- C not acquired other threads might read r_i or privately update r_i
- ▶ acquire r_i 's lock in S mode
 - ▶ read global (committed) value of r_i

update r_i

- W acquired r_i gets (privately or globally) updated by another transaction → there are already two versions
- ▶ abort this transaction

- W not acquired other threads at most read r_i
- ▶ acquire r_i 's lock in X mode
 - ▶ create local copy of global (committed) value of r_i

Protocol I

read r_i

- C acquired r_i gets certified or committed by another transaction
- ▶ abort this transaction

- C not acquired other threads might read r_i or privately update r_i
- ▶ acquire r_i 's lock in S mode
 - ▶ read global (committed) value of r_i

update r_i

- W acquired r_i gets (privately or globally) updated by another transaction → there are already two versions
- ▶ abort this transaction

- W not acquired other threads at most read r_i
- ▶ acquire r_i 's lock in X mode
 - ▶ create local copy of global (committed) value of r_i
 - ▶ update local copy of r_i

Protocol II

Protocol II

certify r_i (only if r_i was updated)

Protocol II

certify r_i (only if r_i was updated)

S acquired r_i 's global (committed) value gets read by other transaction → globally updating r_i would cause non-repeatable reads

Protocol II

certify r_i (only if r_i was updated)

- S acquired r_i 's global (committed) value gets read by other transaction \rightarrow globally updating r_i would cause non-repeatable reads
 - ▶ **abort** this transaction

Protocol II

certify r_i (only if r_i was updated)

S acquired r_i 's global (committed) value gets read by other transaction →
globally updating r_i would cause non-repeatable reads
▶ **abort** this transaction

S not acquired other threads at most read r_i

Protocol II

certify r_i (only if r_i was updated)

S acquired r_i 's global (committed) value gets read by other transaction → globally updating r_i would cause non-repeatable reads

- ▶ abort this transaction

S not acquired other threads at most read r_i

- ▶ acquire r_i 's lock in C mode

Protocol II

certify r_i (only if r_i was updated)

S acquired r_i 's global (committed) value gets read by other transaction → globally updating r_i would cause non-repeatable reads

- ▶ **abort** this transaction

S not acquired other threads at most read r_i

- ▶ acquire r_i 's lock in C mode

commit

Protocol II

certify r_i (only if r_i was updated)

S acquired r_i 's global (committed) value gets read by other transaction → globally updating r_i would cause non-repeatable reads

- ▶ abort this transaction

S not acquired other threads at most read r_i

- ▶ acquire r_i 's lock in C mode

commit

- ▶ (only if r_i was updated) replace global r_i with updated local version

Protocol II

certify r_i (only if r_i was updated)

S acquired r_i 's global (committed) value gets read by other transaction → globally updating r_i would cause non-repeatable reads

- ▶ abort this transaction

S not acquired other threads at most read r_i

- ▶ acquire r_i 's lock in C mode

commit

- ▶ (*only if r_i was updated*) replace global r_i with updated local version
- ▶ release the locks held on r_i

Pros & Cons 2V_NO_WAIT (Two Version 2PL)

Pros & Cons 2V_NO_WAIT (Two Version 2PL)

- + deadlocks are impossible

Pros & Cons 2V_NO_WAIT (Two Version 2PL)

- + deadlocks are impossible
- + transactions can concurrently read while a transaction updates a record

Pros & Cons 2V_NO_WAIT (Two Version 2PL)

- + deadlocks are impossible
- + transactions can concurrently read while a transaction updates a record
- + locks can be implemented using a semaphores and a flag updated with atomic instructions → scales better than latches

Pros & Cons 2V_NO_WAIT (Two Version 2PL)

- + deadlocks are impossible
- + transactions can concurrently read while a transaction updates a record
- + locks can be implemented using a semaphores and a flag updated with atomic instructions → scales better than latches
- + no need to expensively calculate and analysis a wait-for graph

Pros & Cons 2V_NO_WAIT (Two Version 2PL)

- + deadlocks are impossible
- + transactions can concurrently read while a transaction updates a record
- + locks can be implemented using a semaphores and a flag updated with atomic instructions → scales better than latches
- + no need to expensively calculate and analysis a wait-for graph
- update-intensive high contention workloads result in many lock conflicts → many aborts → work done before needs to be repeated

Pros & Cons 2V_NO_WAIT (Two Version 2PL)

- + deadlocks are impossible
- + transactions can concurrently read while a transaction updates a record
- + locks can be implemented using a semaphores and a flag updated with atomic instructions → scales better than latches
- + no need to expensively calculate and analysis a wait-for graph
- update-intensive high contention workloads result in many lock conflicts → many aborts → work done before needs to be repeated
- locks still prevent concurrency too often (e.g. blind writes)

Pros & Cons 2V_NO_WAIT (Two Version 2PL)

- + deadlocks are impossible
- + transactions can concurrently read while a transaction updates a record
- + locks can be implemented using a semaphores and a flag updated with atomic instructions → scales better than latches
- + no need to expensively calculate and analysis a wait-for graph
- update-intensive high contention workloads result in many lock conflicts → many aborts → work done before needs to be repeated
- locks still prevent concurrency too often (e.g. blind writes)
- additional steps for updates required:

Pros & Cons 2V_NO_WAIT (Two Version 2PL)

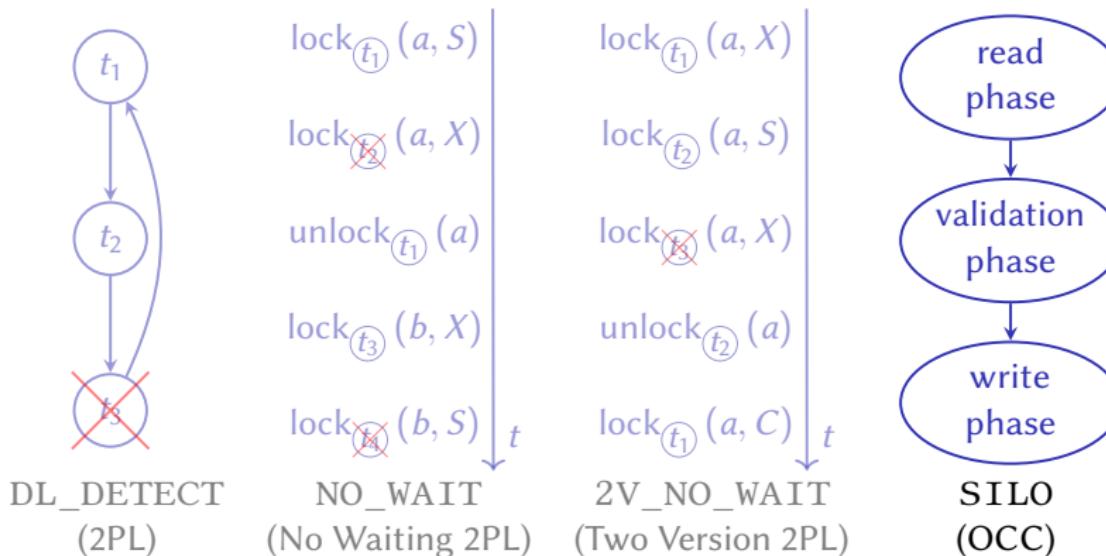
- + deadlocks are impossible
- + transactions can concurrently read while a transaction updates a record
- + locks can be implemented using a semaphores and a flag updated with atomic instructions → scales better than latches
- + no need to expensively calculate and analysis a wait-for graph
- update-intensive high contention workloads result in many lock conflicts → many aborts → work done before needs to be repeated
- locks still prevent concurrency too often (e.g. blind writes)
- additional steps for updates required:
 - ▶ create private copy of updated record (expensive but scalable)

Pros & Cons 2V_NO_WAIT (Two Version 2PL)

- + deadlocks are impossible
- + transactions can concurrently read while a transaction updates a record
- + locks can be implemented using a semaphores and a flag updated with atomic instructions → scales better than latches
- + no need to expensively calculate and analysis a wait-for graph
- update-intensive high contention workloads result in many lock conflicts → many aborts → work done before needs to be repeated
- locks still prevent concurrency too often (e.g. blind writes)
- additional steps for updates required:
 - ▶ create private copy of updated record (expensive but scalable)
 - ▶ certify update (cheap)

Subsection 4

SILO (OCC)



Properties of SILO (OCC) I

Properties of SILO (OCC) I

- ▶ optimistic concurrency control protocol

Properties of SILO (OCC) I

- ▶ optimistic concurrency control protocol
- ▶ 3 phases: read, validate, write/commit
- ▶ each record contains the transaction ID (global ordered number based on epochs) from the last update

Properties of SILO (OCC) I

- ▶ optimistic concurrency control protocol
- ▶ 3 phases: read, validate, write/commit
- ▶ each record contains the transaction ID (global ordered number based on epochs) from the last update
- ▶ transactions perform reads and local writes during the read phase without acquiring locks

Properties of SILO (OCC) I

- ▶ optimistic concurrency control protocol
- ▶ 3 phases: read, validate, write/commit
- ▶ each record contains the transaction ID (global ordered number based on epochs) from the last update
- ▶ transactions perform reads and local writes during the read phase without acquiring locks
- ▶ read and write sets (records read and written from the transaction) are recorded locally

Properties of SILO (OCC) I

- ▶ optimistic concurrency control protocol
- ▶ 3 phases: read, validate, write/commit
- ▶ each record contains the transaction ID (global ordered number based on epochs) from the last update
- ▶ transactions perform reads and local writes during the read phase without acquiring locks
- ▶ read and write sets (records read and written from the transaction) are recorded locally
- ▶ read and write sets are used to validate the absence of relevant conflicts

Properties of SILO (OCC) I

- ▶ optimistic concurrency control protocol
- ▶ 3 phases: read, validate, write/commit
- ▶ each record contains the transaction ID (global ordered number based on epochs) from the last update
- ▶ transactions perform reads and local writes during the read phase without acquiring locks
- ▶ read and write sets (records read and written from the transaction) are recorded locally
- ▶ read and write sets are used to validate the absence of relevant conflicts
- ▶ commit requires three phases: locking of updated records, verification of read set (based on TIDs), execute global writes

Properties of SILO (OCC) II

Properties of SILO (OCC) II

- ▶ deletes invalid records using updates → garbage collection required

Properties of SILO (OCC) II

- ▶ deletes invalidate records using updates → garbage collection required
- ▶ records for inserts are created before the validation to provide locks

Pros & Cons SILO (OCC)

Pros & Cons SILO (OCC)

- + deadlocks are impossible (locks acquired only in last phase → global order can be used)

Pros & Cons SILO (OCC)

- + deadlocks are impossible (locks acquired only in last phase → global order can be used)
- + transactions can concurrently read and write

Pros & Cons SILO (OCC)

- + deadlocks are impossible (locks acquired only in last phase → global order can be used)
- + transactions can concurrently read and write
- + only actual conflicts cause aborts (optimism)

Pros & Cons SILO (OCC)

- + deadlocks are impossible (locks acquired only in last phase → global order can be used)
- + transactions can concurrently read and write
- + only actual conflicts cause aborts (optimism)
- update-intensive high contention workloads result in many invalid reads → many aborts → work done before needs to be repeated

Pros & Cons SILO (OCC)

- + deadlocks are impossible (locks acquired only in last phase → global order can be used)
- + transactions can concurrently read and write
- + only actual conflicts cause aborts (optimism)
- update-intensive high contention workloads result in many invalid reads → many aborts → work done before needs to be repeated
- globally sorted transaction IDs need to be generated (epochs make that cheap)

Pros & Cons SILO (OCC)

- + deadlocks are impossible (locks acquired only in last phase → global order can be used)
- + transactions can concurrently read and write
- + only actual conflicts cause aborts (optimism)
- update-intensive high contention workloads result in many invalid reads → many aborts → work done before needs to be repeated
- globally sorted transaction IDs need to be generated (epochs make that cheap)
- additional steps for updating transactions required:

Pros & Cons SILO (OCC)

- + deadlocks are impossible (locks acquired only in last phase → global order can be used)
- + transactions can concurrently read and write
- + only actual conflicts cause aborts (optimism)
- update-intensive high contention workloads result in many invalid reads → many aborts → work done before needs to be repeated
- globally sorted transaction IDs need to be generated (epochs make that cheap)
- additional steps for updating transactions required:
 - ▶ record write and read sets locally (expensive but scalable)

Pros & Cons SILO (OCC)

- + deadlocks are impossible (locks acquired only in last phase → global order can be used)
- + transactions can concurrently read and write
- + only actual conflicts cause aborts (optimism)
- update-intensive high contention workloads result in many invalid reads → many aborts → work done before needs to be repeated
- globally sorted transaction IDs need to be generated (epochs make that cheap)
- additional steps for updating transactions required:
 - ▶ record write and read sets locally (expensive but scalable)
 - ▶ validate reads

Section 4

Performance Evaluation

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

Evaluation Set-Up

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

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

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

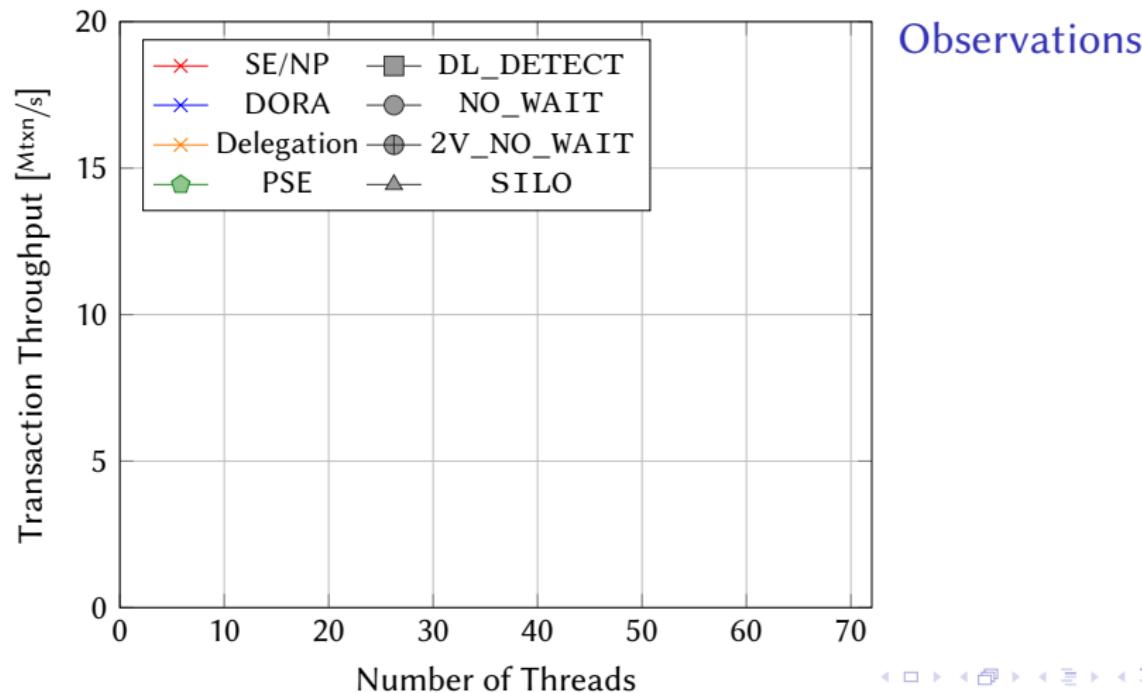
Microbenchmark

- ▶ 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

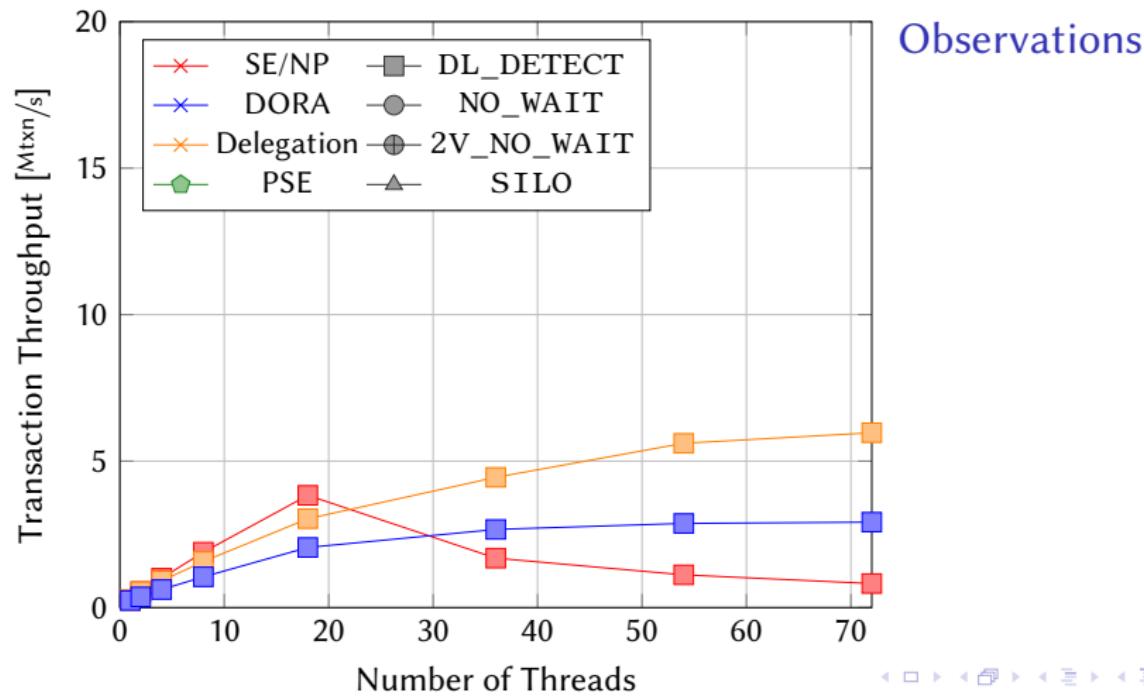
Yahoo! Cloud Serving Benchmark (YCSB)

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

Read-Only Microbenchmark

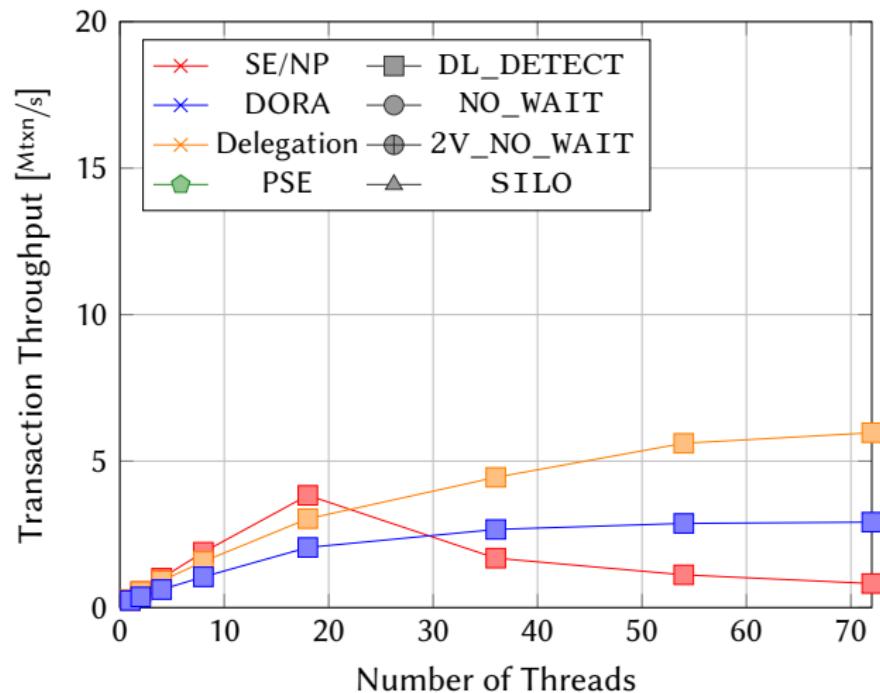


Read-Only Microbenchmark



Observations

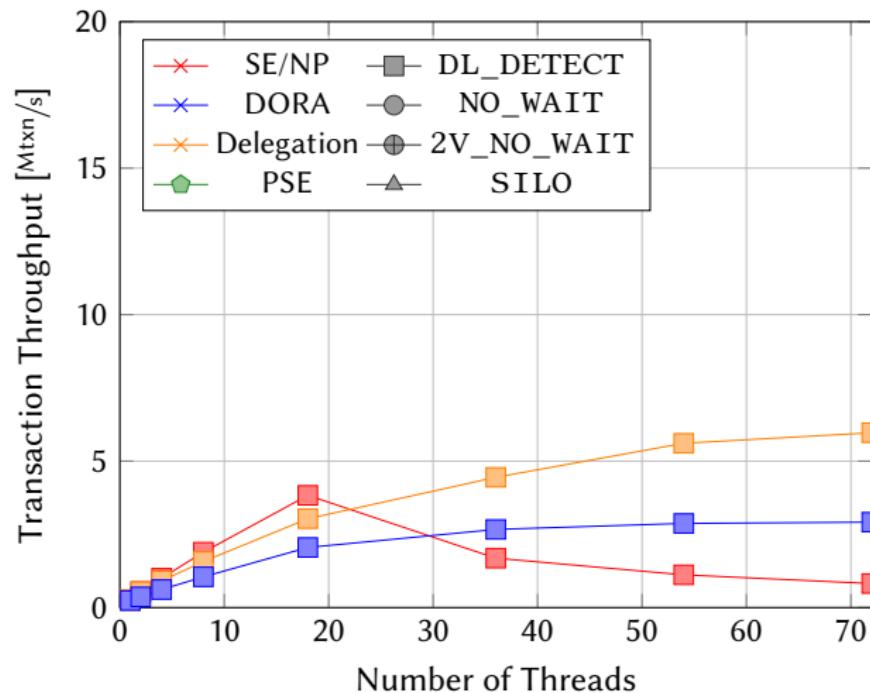
Read-Only Microbenchmark



Observations

- $\text{*/\textcolor{orange}{*}}$ suffer from remote data access overhead

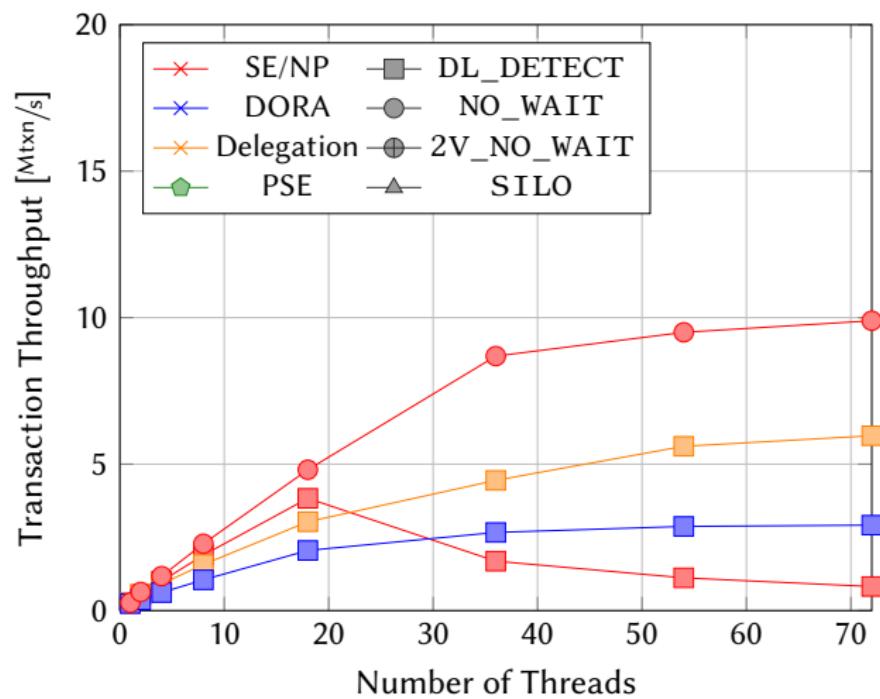
Read-Only Microbenchmark



Observations

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

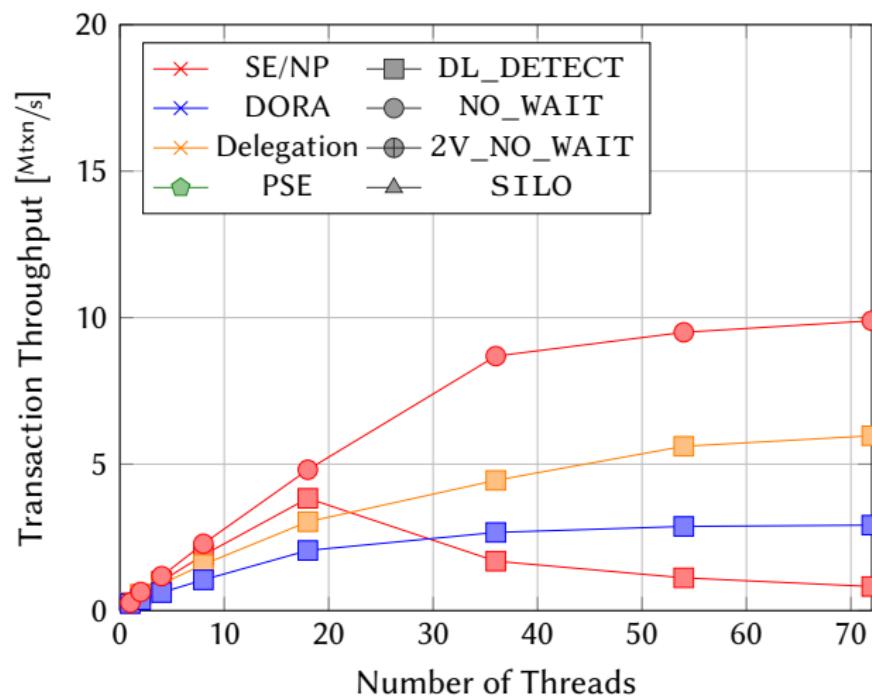
Read-Only Microbenchmark



Observations

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

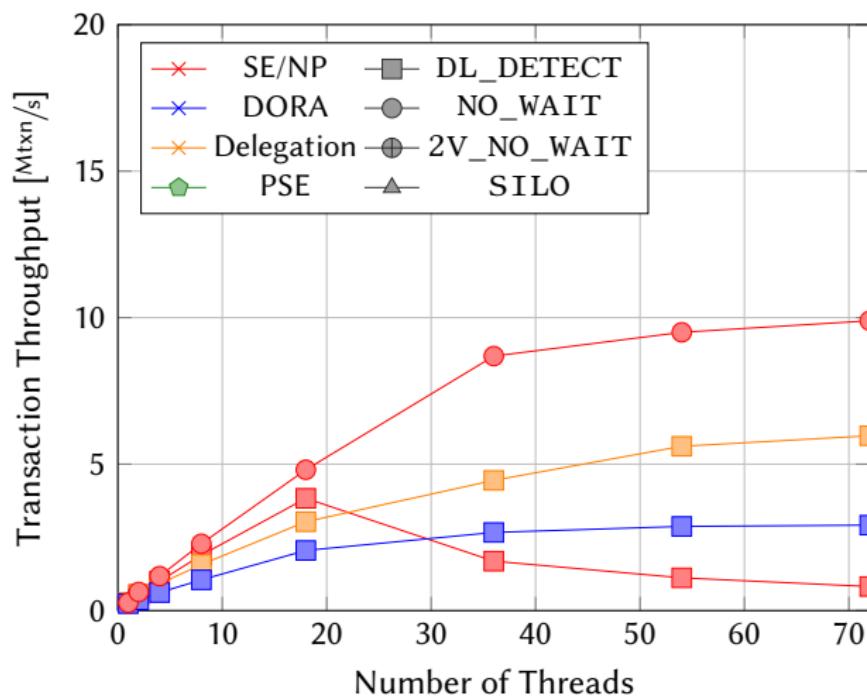
Read-Only Microbenchmark



Observations

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

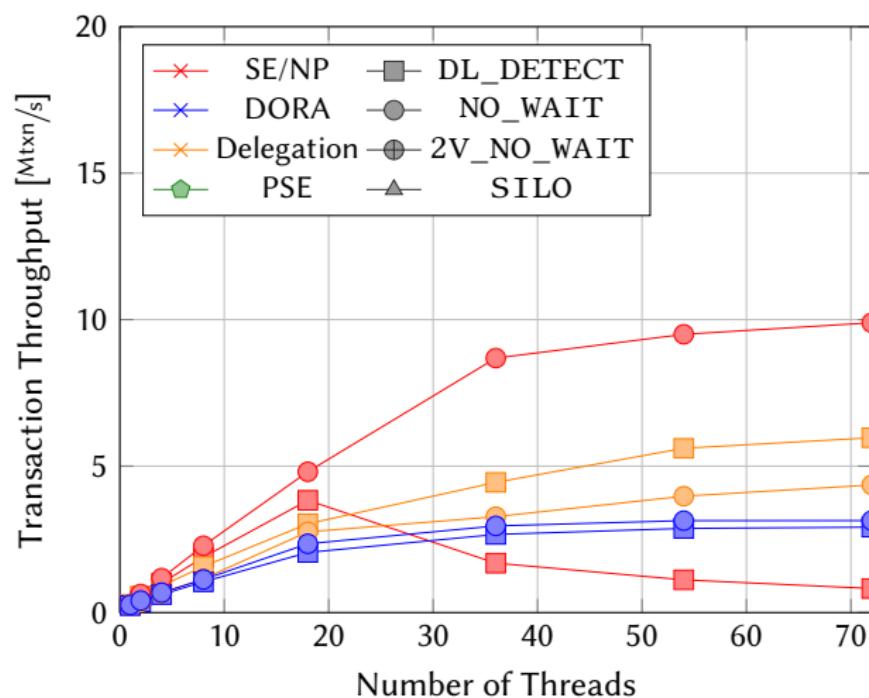
Read-Only Microbenchmark



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

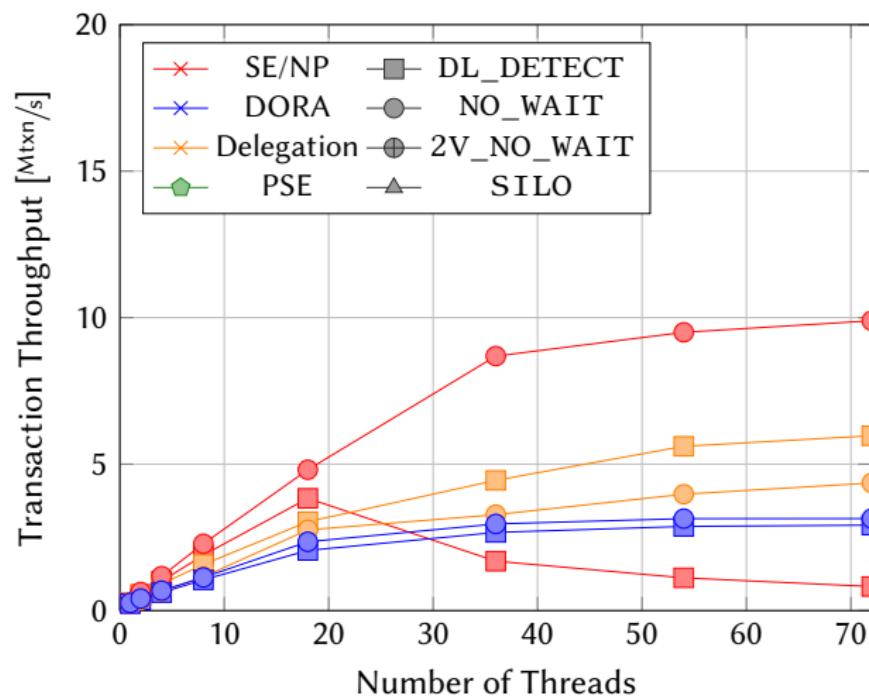
Read-Only Microbenchmark



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

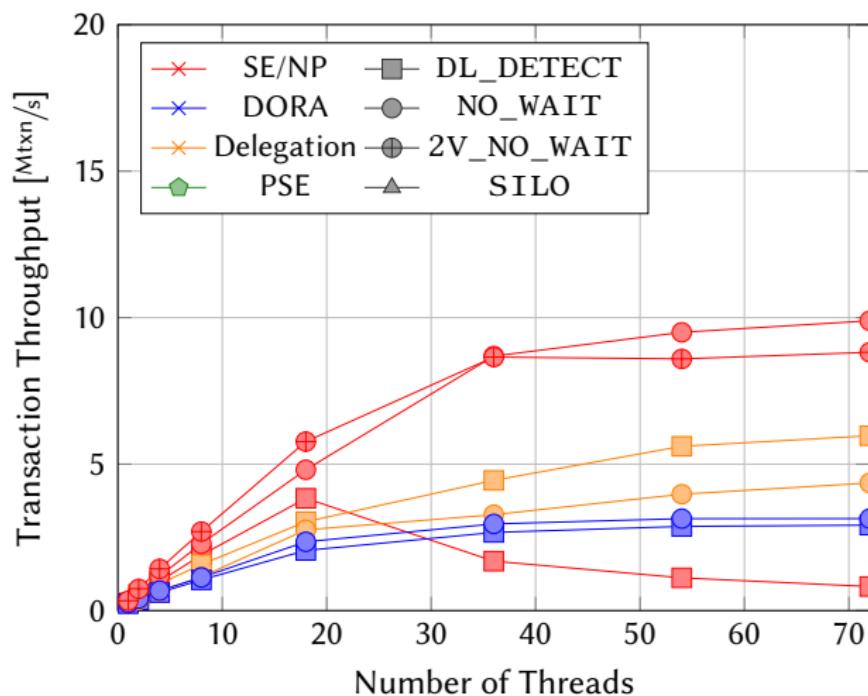
Read-Only Microbenchmark



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

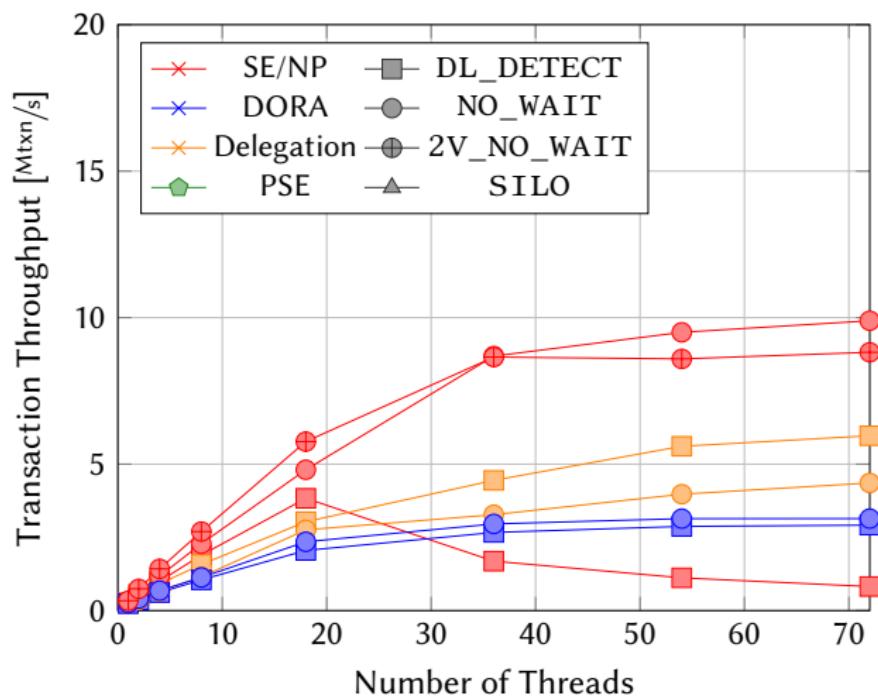
Read-Only Microbenchmark



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

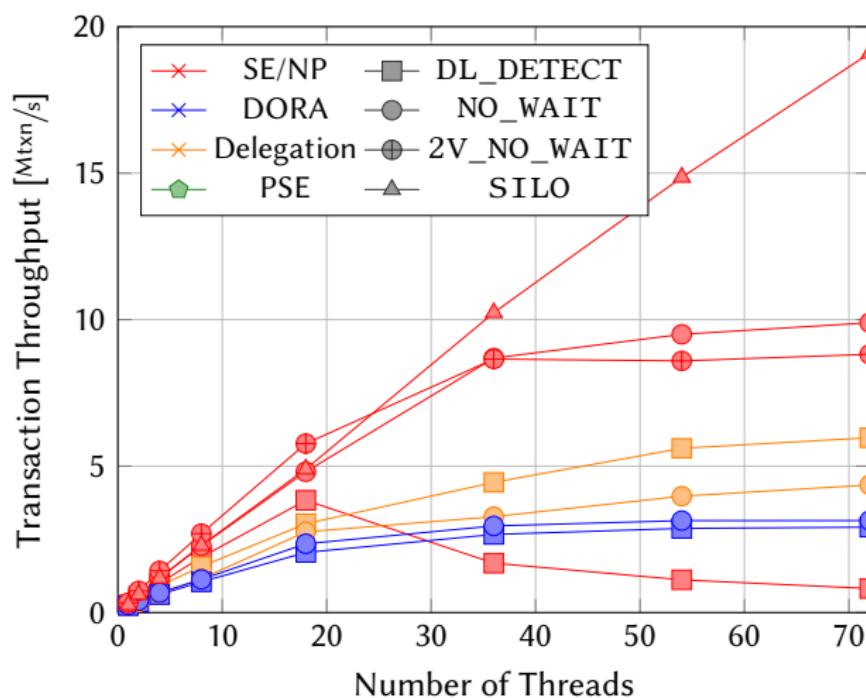
Read-Only Microbenchmark



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

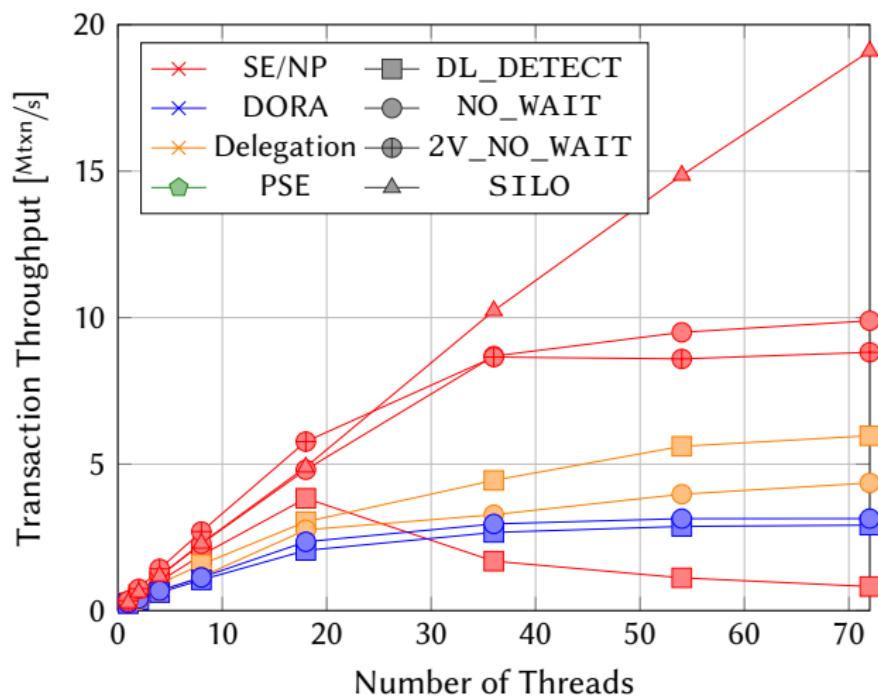
Read-Only Microbenchmark



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

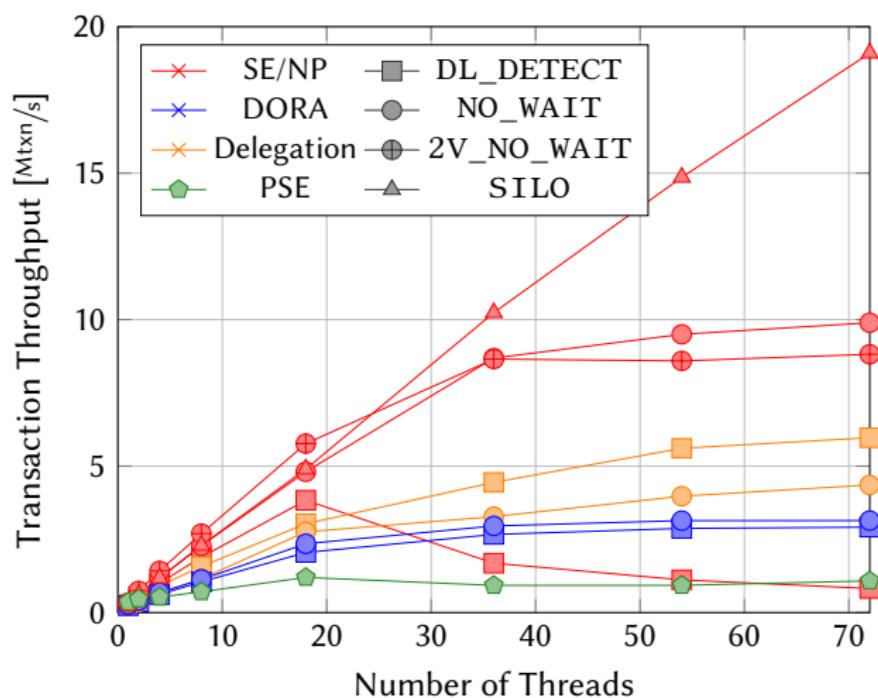
Read-Only Microbenchmark



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

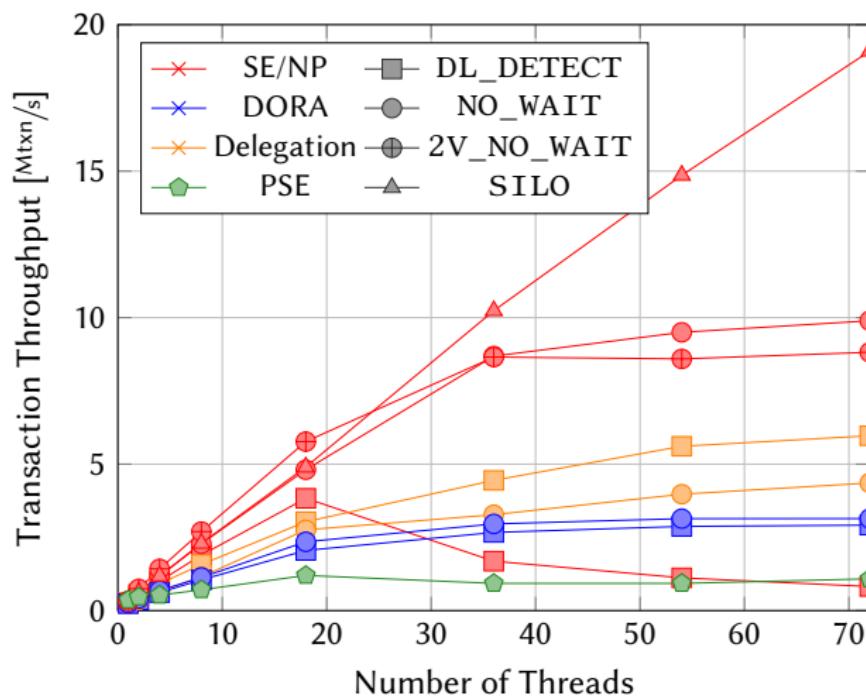
Read-Only Microbenchmark



Observations

- scaling of \bullet limited by hardware cache coherence mechanism
- \ast / \times suffer more from remote data accesses than \ast suffers from cache coherence
- \odot and \bullet perform identical for read-only
- \triangle behaves identical for \ast and \times for read-only

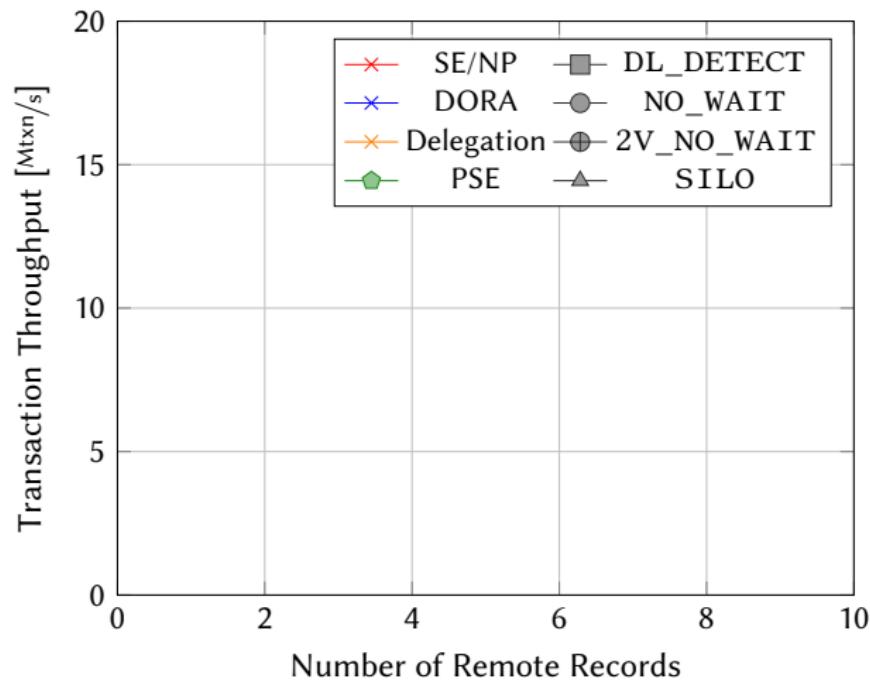
Read-Only Microbenchmark



Observations

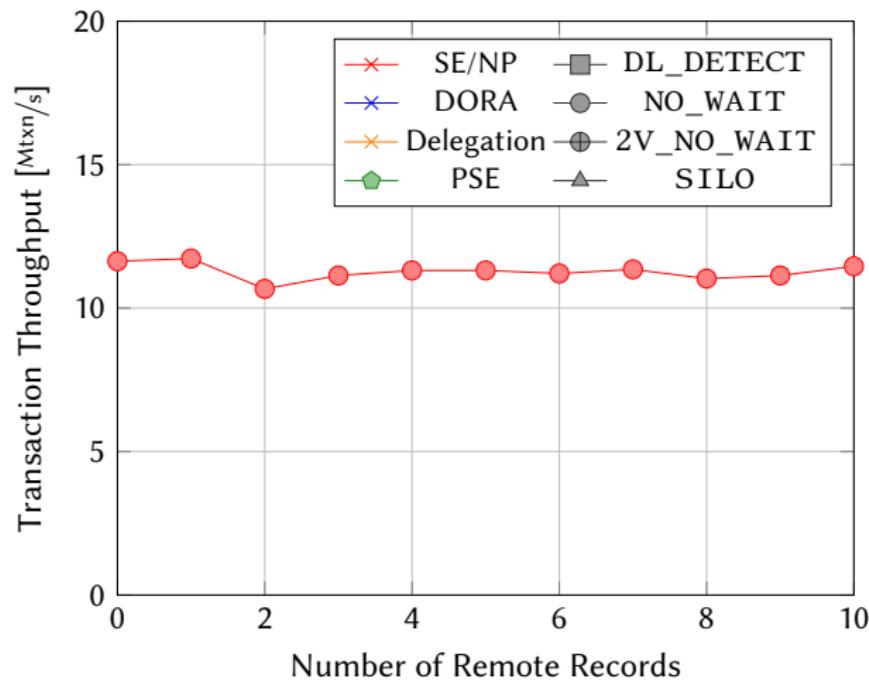
- ***/*** suffer more from remote data accesses than ***** suffers from cache coherence
- **○** and **●** perform identical for read-only
- **▲** behaves identical for ***** and **○** for read-only
- coarse-grained partition locking of **◆** doesn't scale due to multi-site workload

Multi-Site Read-Only Microbenchmark



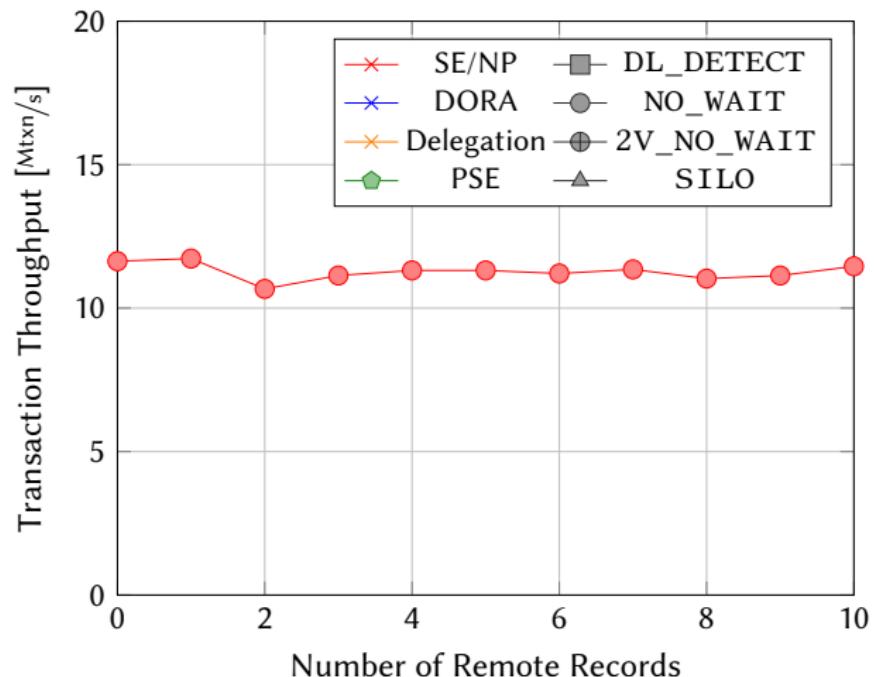
Observations

Multi-Site Read-Only Microbenchmark



Observations

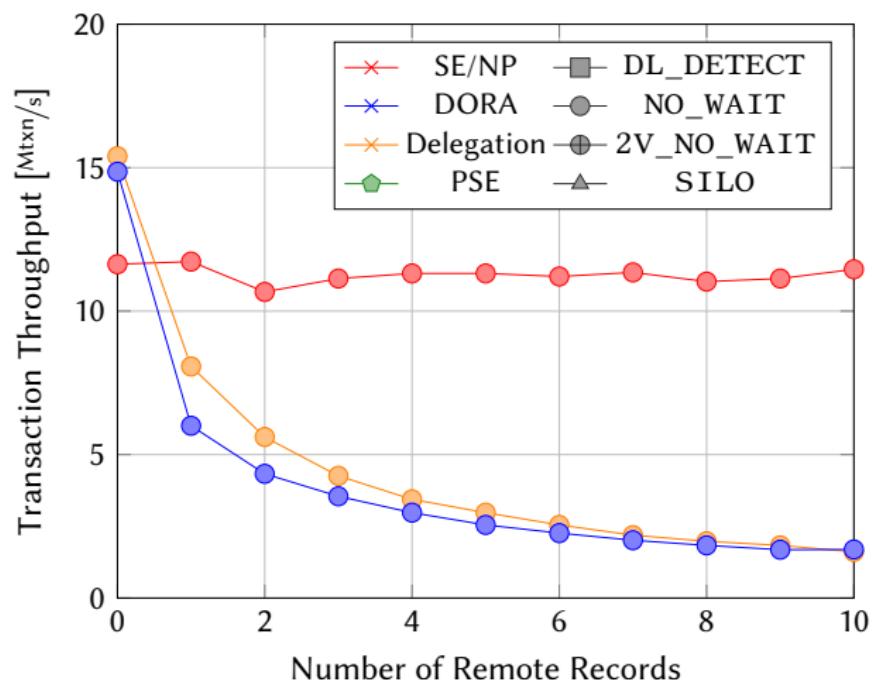
Multi-Site Read-Only Microbenchmark



Observations

- ▶ SE/NP $\text{\textcolor{red}{*}}$ doesn't know remote records

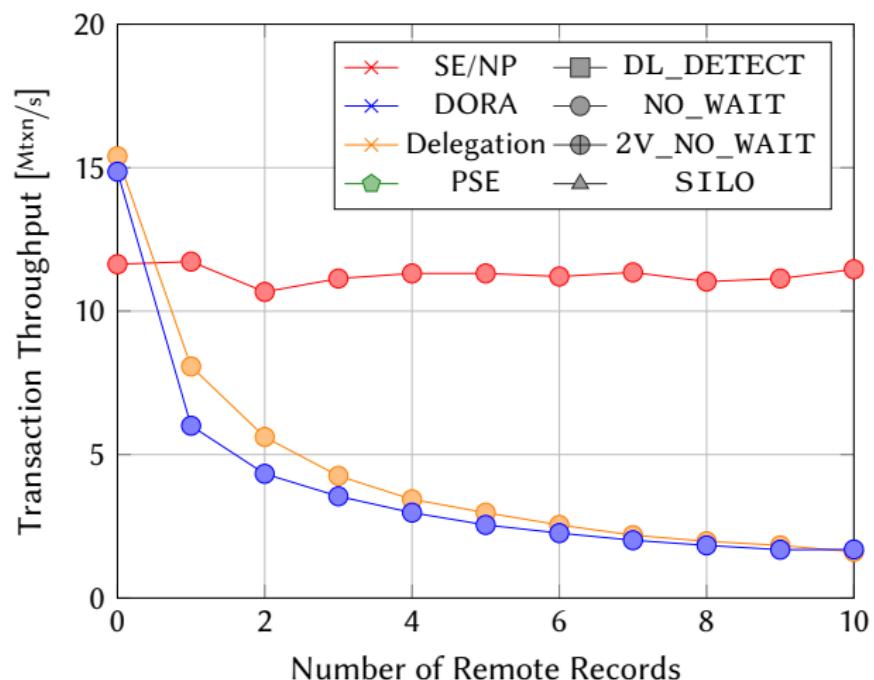
Multi-Site Read-Only Microbenchmark



Observations

- SE/NP doesn't know remote records

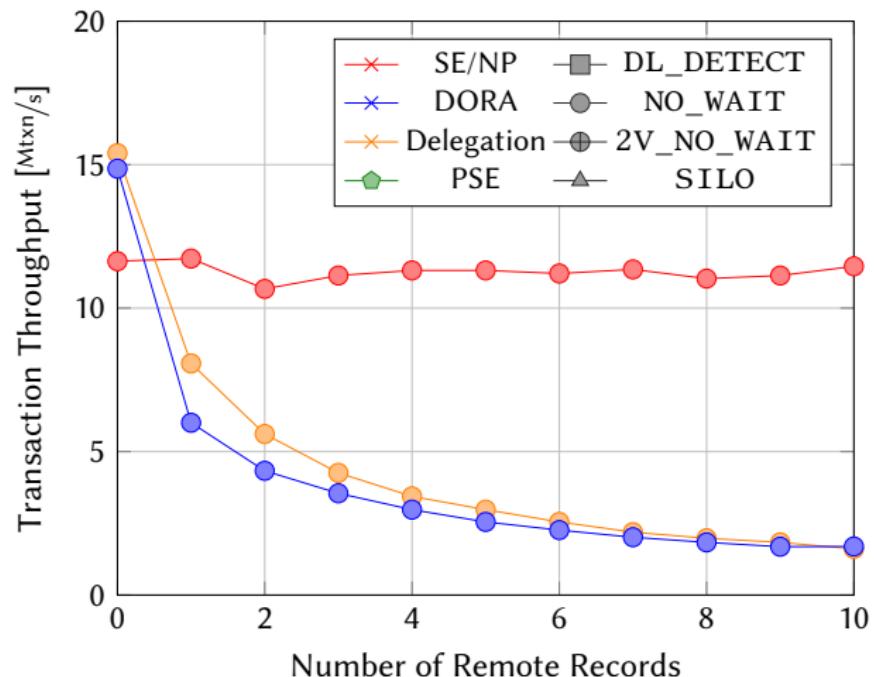
Multi-Site Read-Only Microbenchmark



Observations

- SE/NP doesn't know remote records
- DORA/* outperform SE/NP for 0 remote records due to lower cache coherence activity

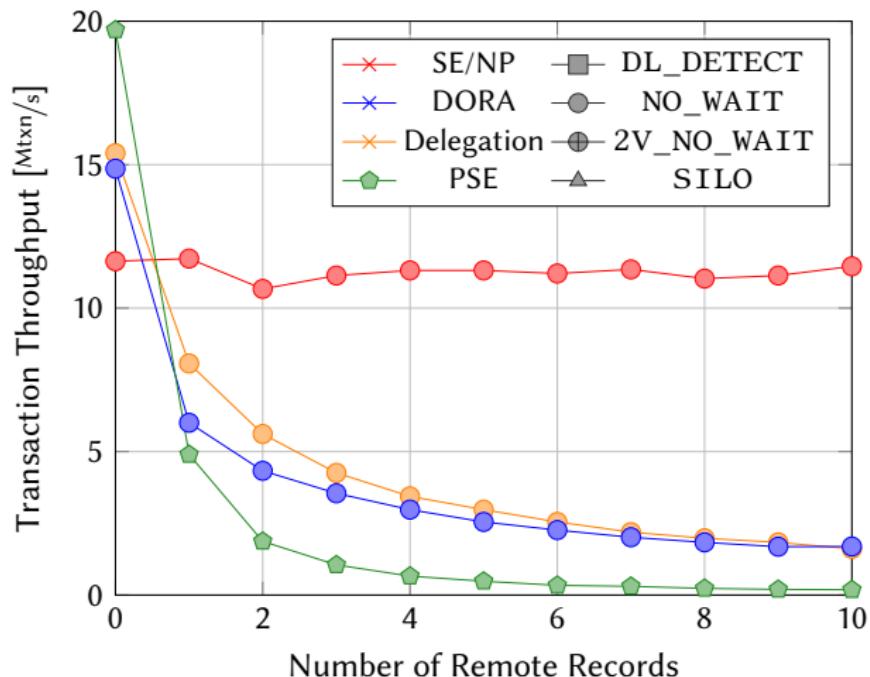
Multi-Site Read-Only Microbenchmark



Observations

- SE/NP doesn't know remote records
- $\text{DORA}/\text{Delegation}$ outperform SE/NP for 0 remote records due to lower cache coherence activity
- $\text{DORA}/\text{Delegation}$ suffer from remote data access overhead for > 0 remote records

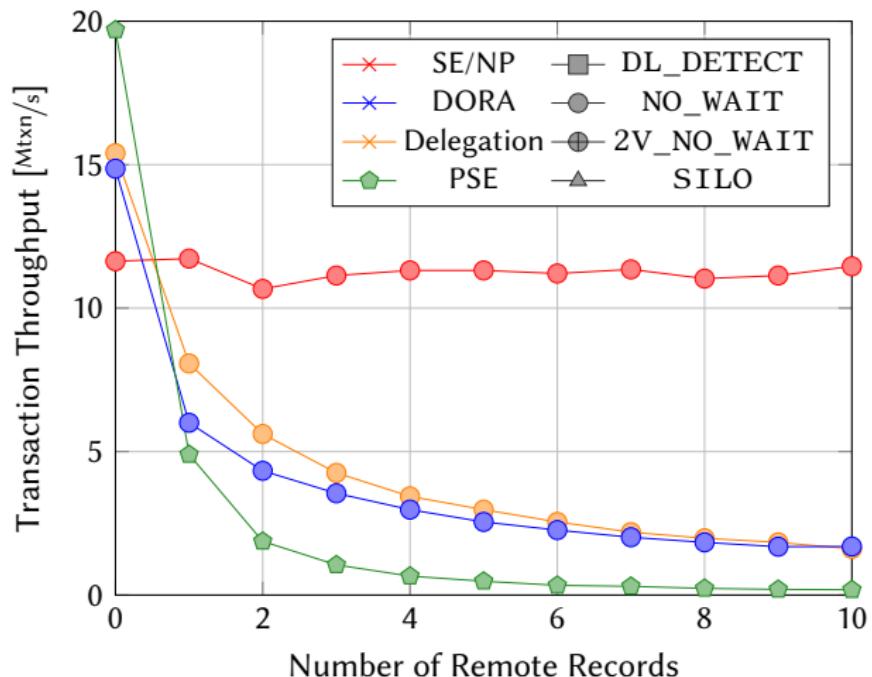
Multi-Site Read-Only Microbenchmark



Observations

- ▶ doesn't know remote records
- ▶ / outperform for 0 remote records due to lower cache coherence activity
- ▶ / suffer from remote data access overhead for > 0 remote records

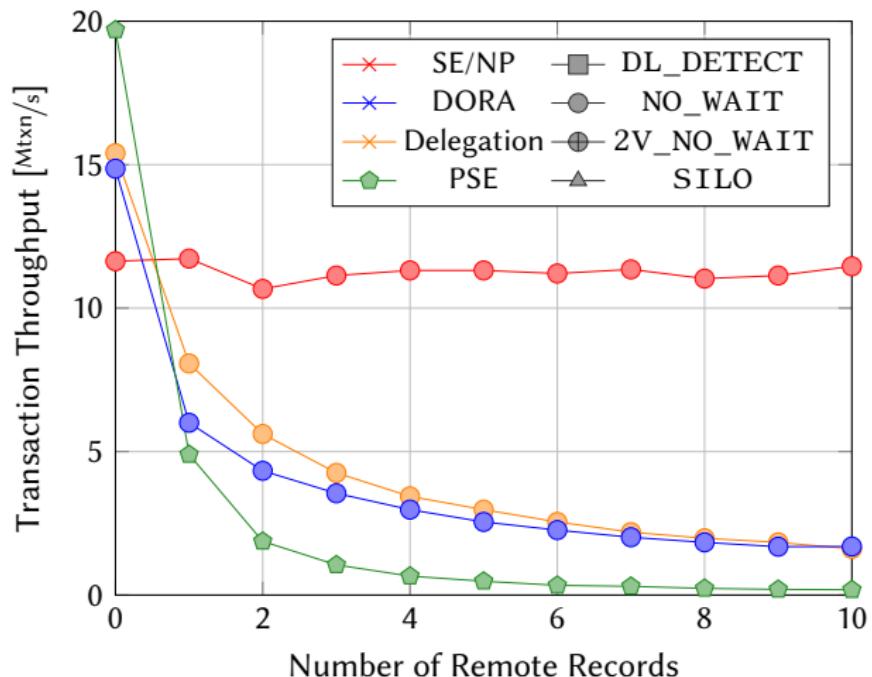
Multi-Site Read-Only Microbenchmark



Observations

- ▶ / outperform for 0 remote records due to lower cache coherence activity
- ▶ / suffer from remote data access overhead for > 0 remote records
- ▶ coarse-grained partition locking of imposes nearly no overhead for suitable partitioning

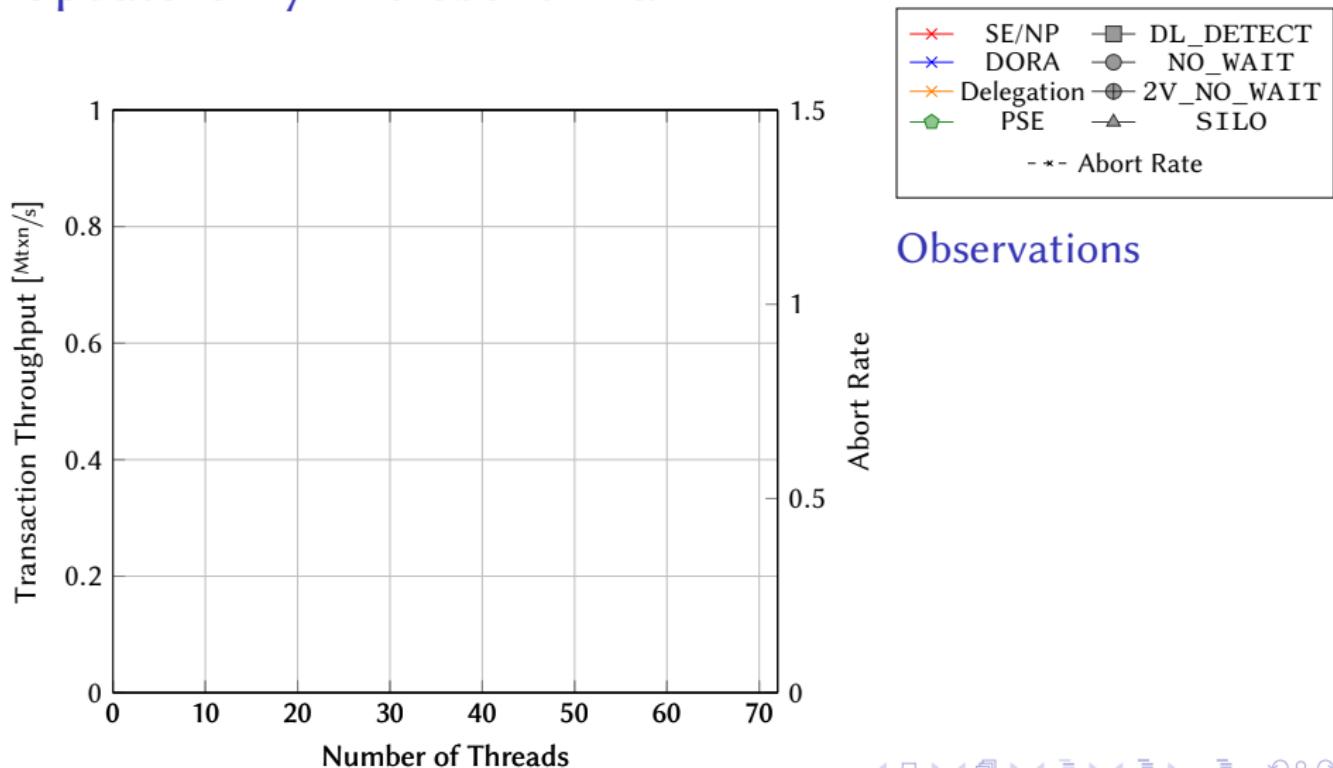
Multi-Site Read-Only Microbenchmark



Observations

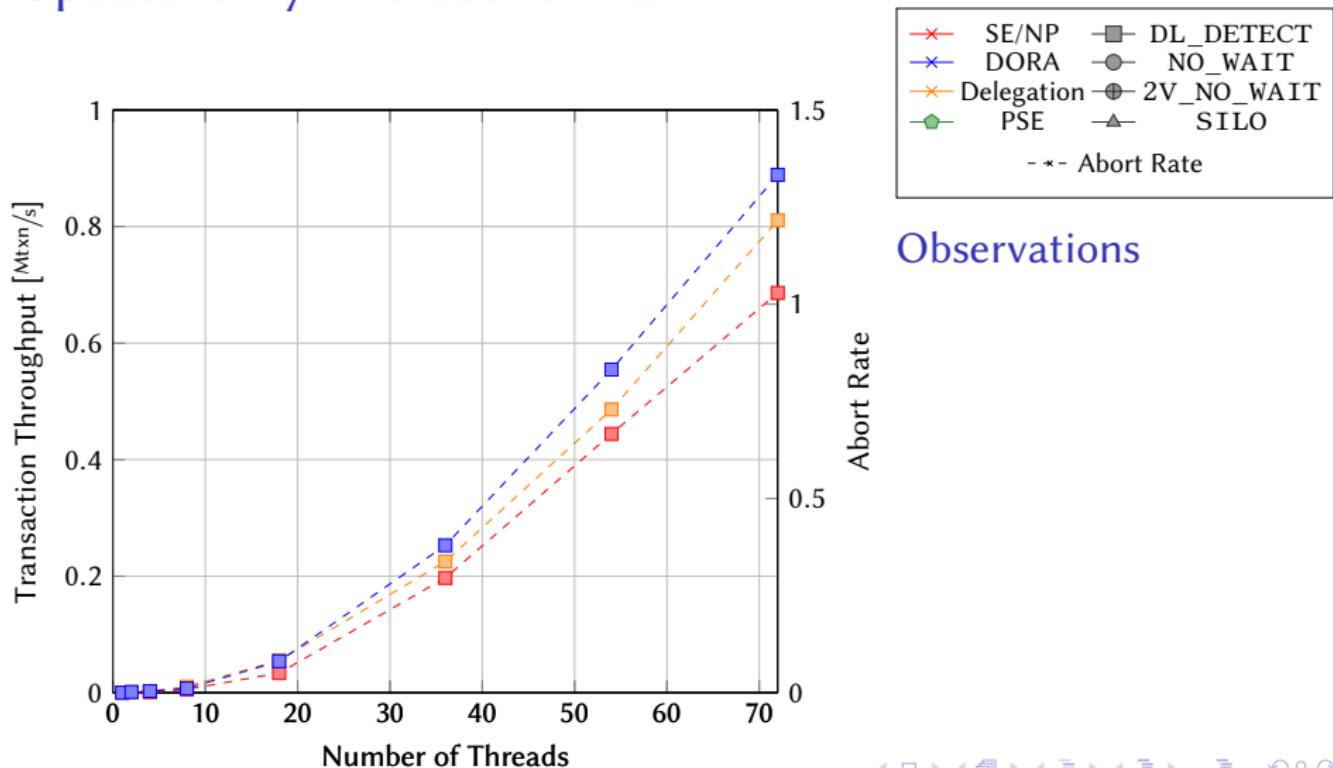
- ▶ suffer from remote data access overhead for > 0 remote records
- ▶ coarse-grained partition locking of imposes nearly no overhead for suitable partitioning
- ▶ coarse-grained partition locking of limits the concurrency drastically for > 0 remote records

Update-Only Microbenchmark



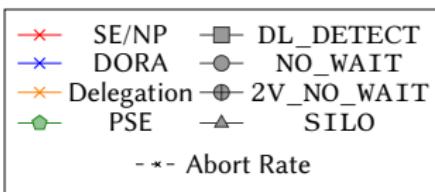
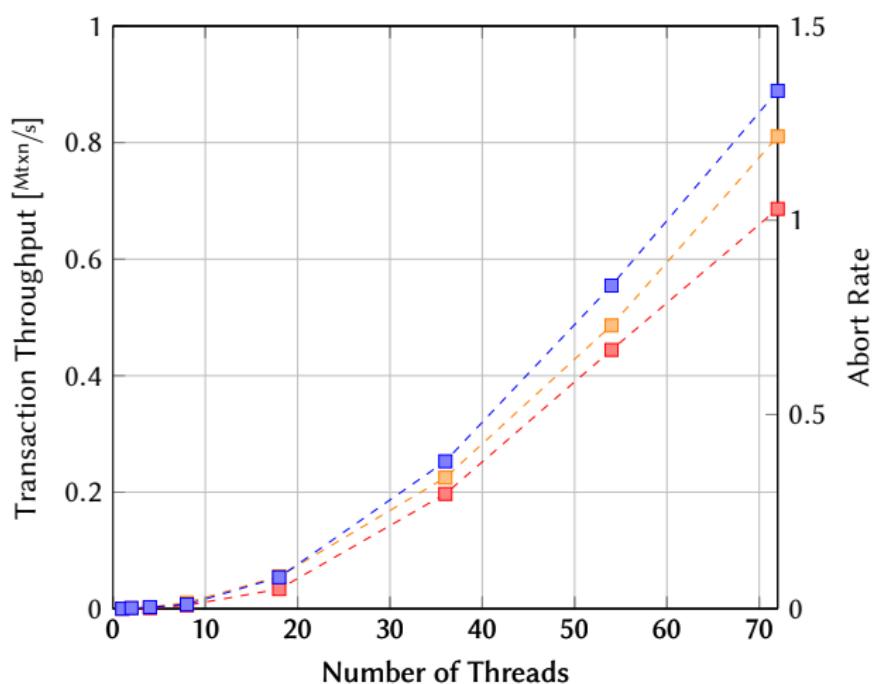
Observations

Update-Only Microbenchmark



Observations

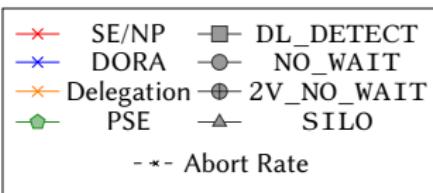
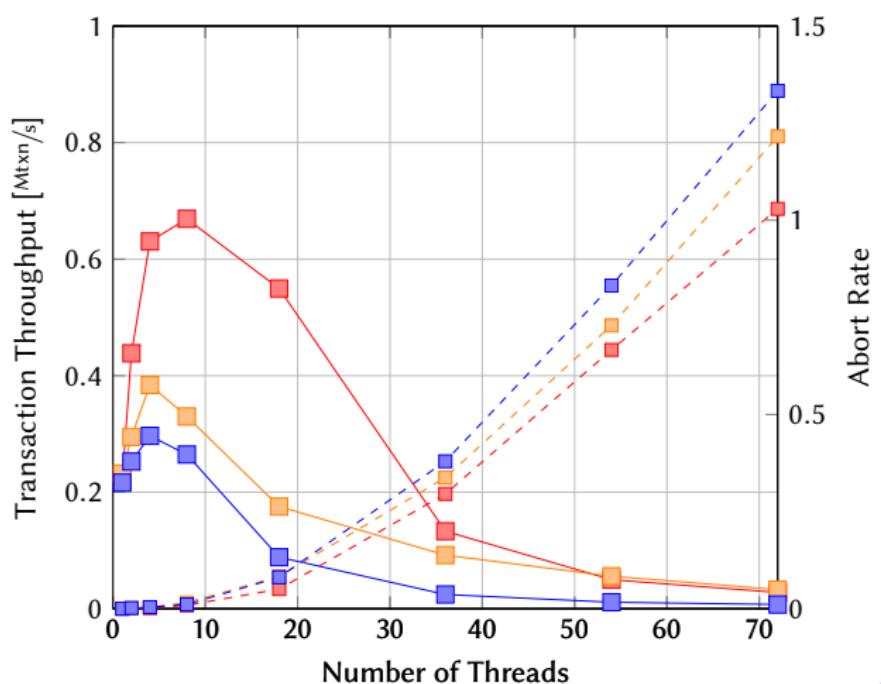
Update-Only Microbenchmark



Observations

- abort rate scales for \square due to higher contention
→ deadlocks

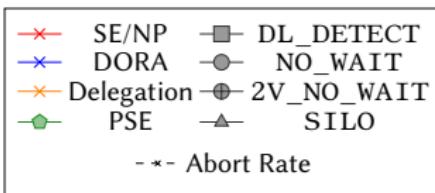
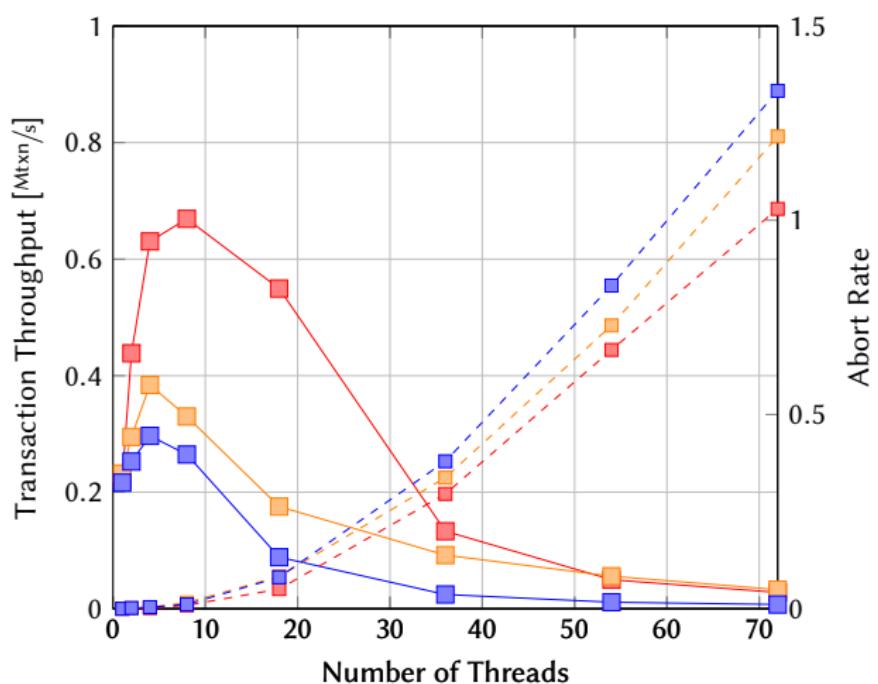
Update-Only Microbenchmark



Observations

- abort rate scales for **DL_DETECT** due to higher contention
→ deadlocks

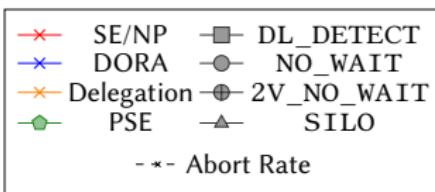
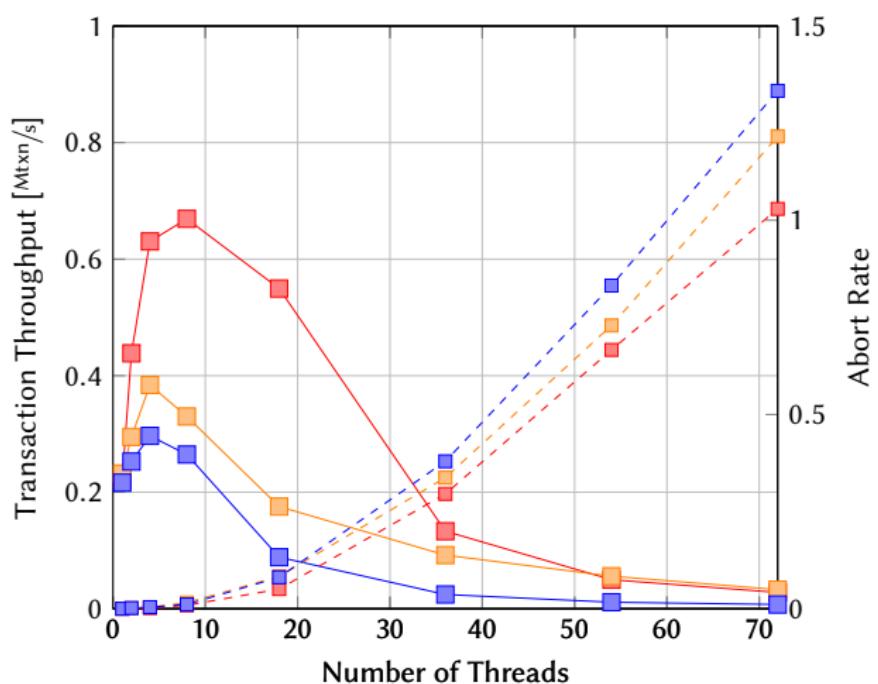
Update-Only Microbenchmark



Observations

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

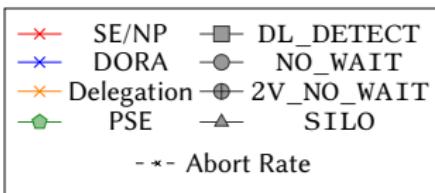
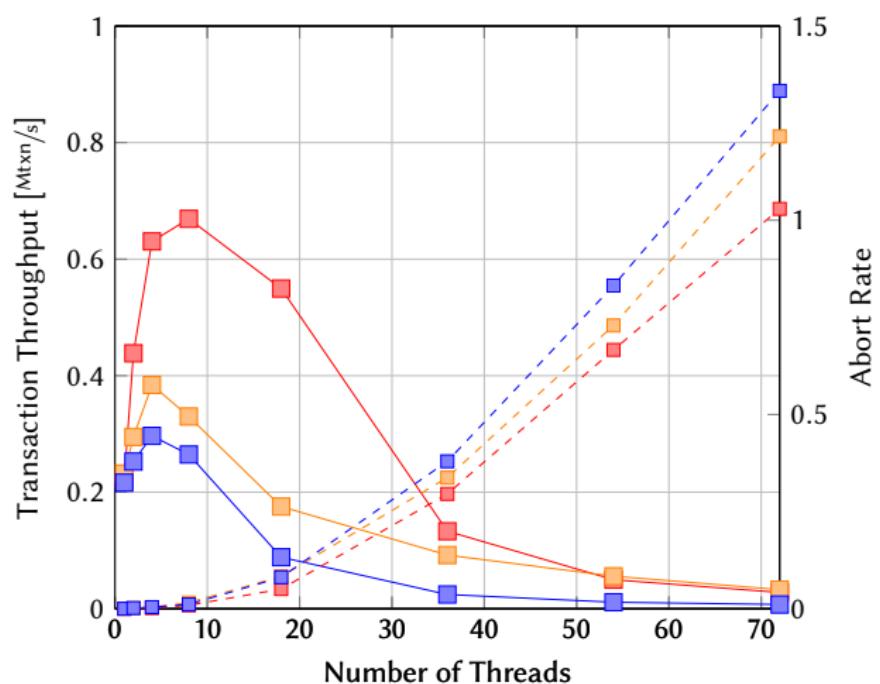
Update-Only Microbenchmark



Observations

- abort rate scales for **DL_DETECT** due to higher contention → deadlocks
- **[Mtxn/s]** suffers from aborts and lock thrashing
- **DORA**/**Delegation** suffer more from remote data access overhead

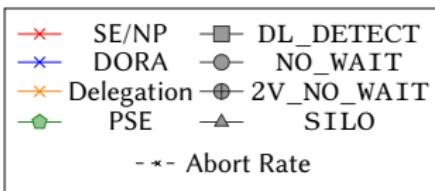
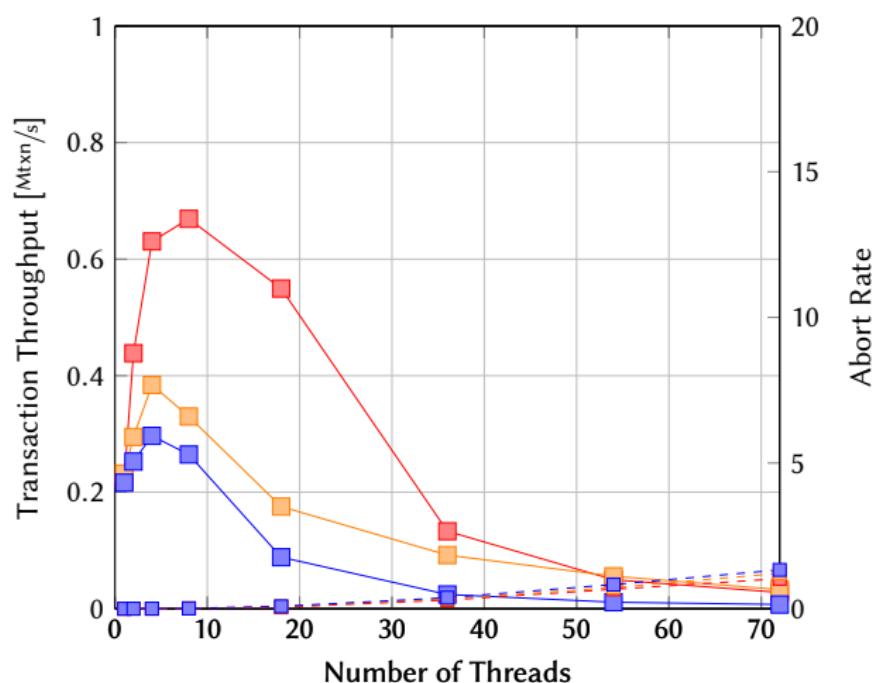
Update-Only Microbenchmark



Observations

- $[Mtxn/s]$ suffers from aborts and lock thrashing
- \times/\diamond suffer more from remote data access overhead
- latch contention isn't the bottleneck → \ast can outperform \times/\diamond

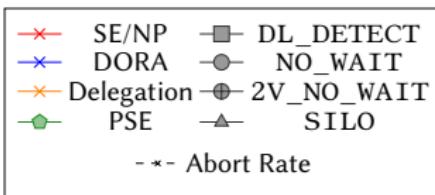
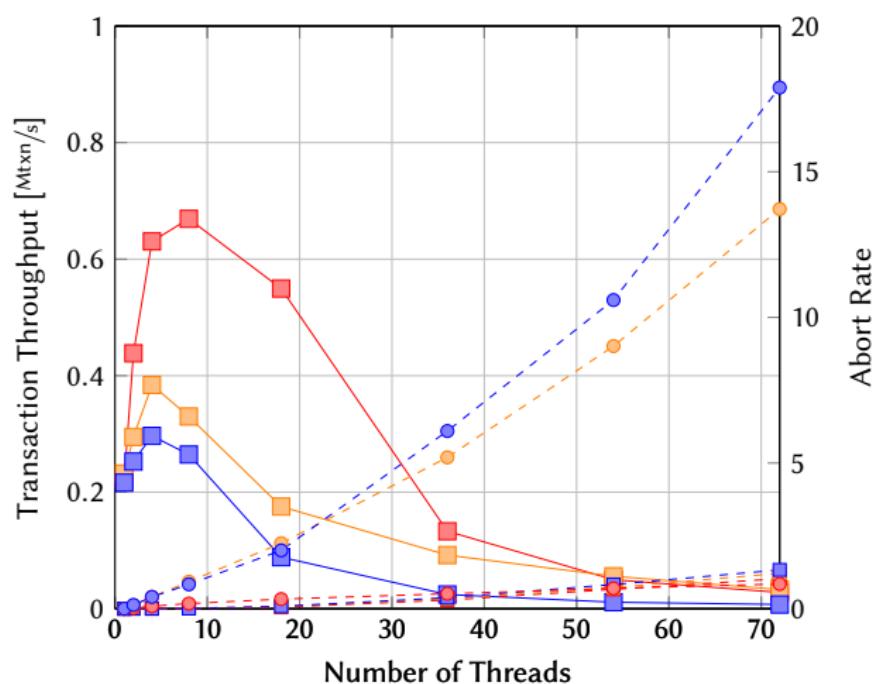
Update-Only Microbenchmark



Observations

- $[Mtxn/s]$ suffers from aborts and lock thrashing
- \times/\ast suffer more from remote data access overhead
- latch contention isn't the bottleneck → \ast can outperform \times/\ast

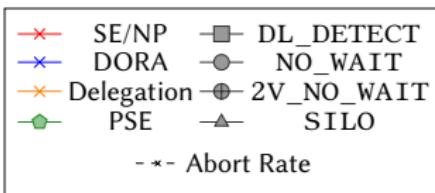
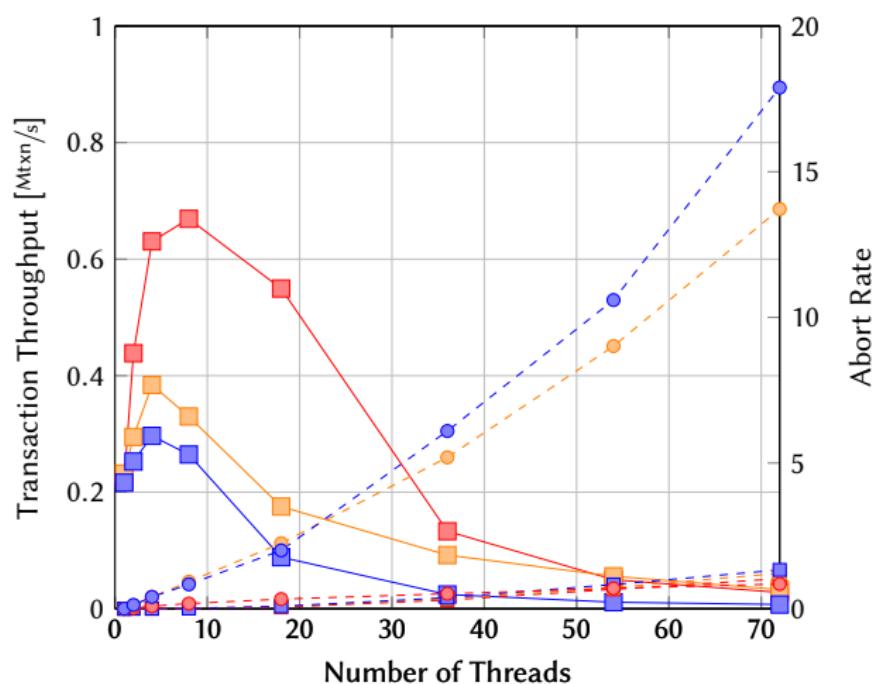
Update-Only Microbenchmark



Observations

- ▶ [Mtxn/s] suffers from aborts and lock thrashing
- ▶ ✕/✖ suffer more from remote data access overhead
- ▶ latch contention isn't the bottleneck → SE/NP can outperform ✕/✖

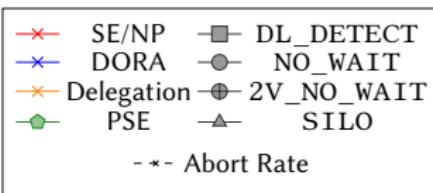
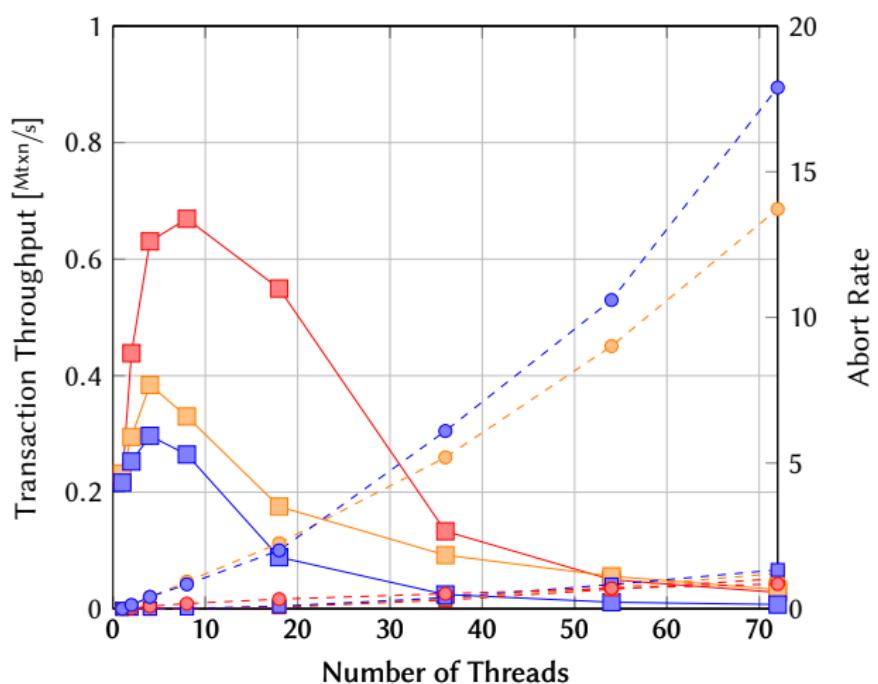
Update-Only Microbenchmark



Observations

- ▶ `*`/`*` suffer more from remote data access overhead
- ▶ latch contention isn't the bottleneck → `*` can outperform `*`/`*`
- ▶ lock thrashing doesn't cause many aborts for `●` with `*` for few threads

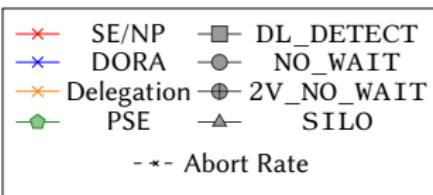
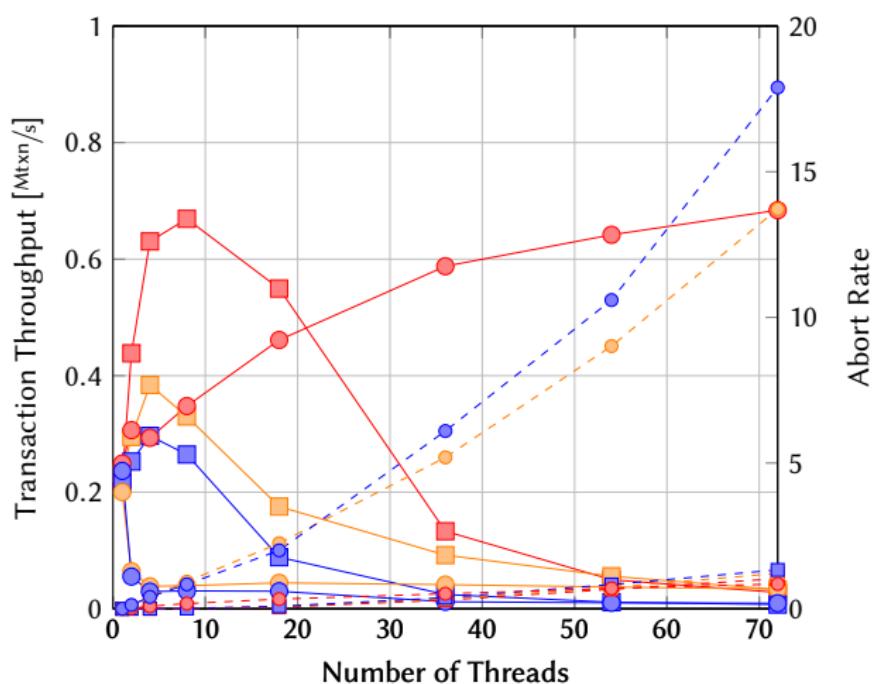
Update-Only Microbenchmark



Observations

- lock thrashing doesn't cause many aborts for with for few threads
- lock thrashing caused by long commit latencies caused by overloaded (hot) partitions causes many aborts for /

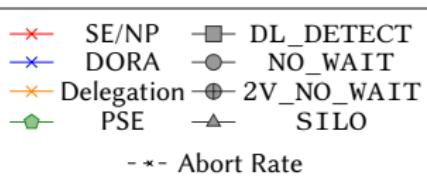
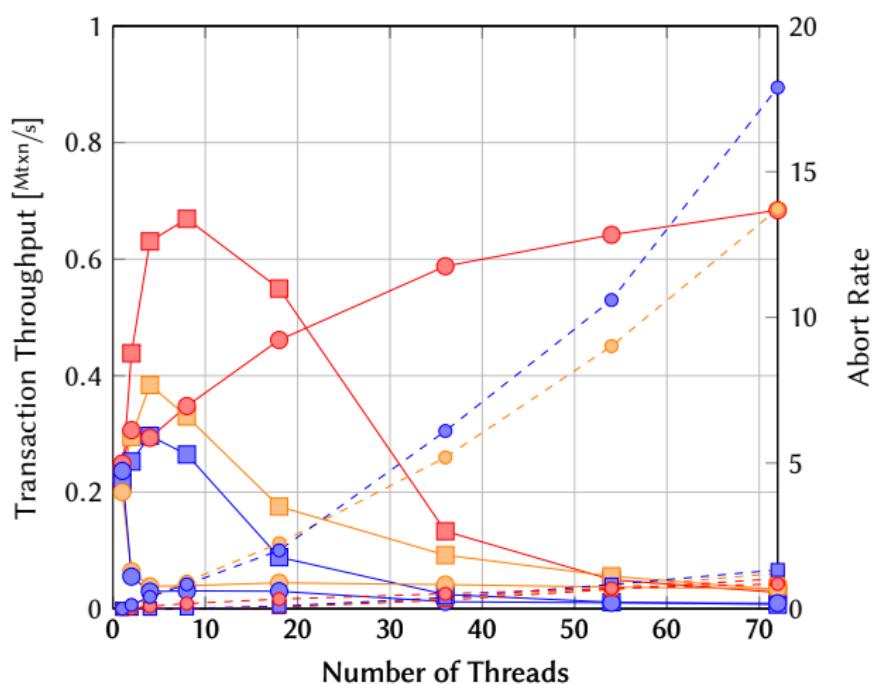
Update-Only Microbenchmark



Observations

- lock thrashing doesn't cause many aborts for **NO_WAIT** with **SE/NP** for few threads
- lock thrashing caused by long commit latencies caused by overloaded (hot) partitions causes many aborts for **DORA**/**Delegation**

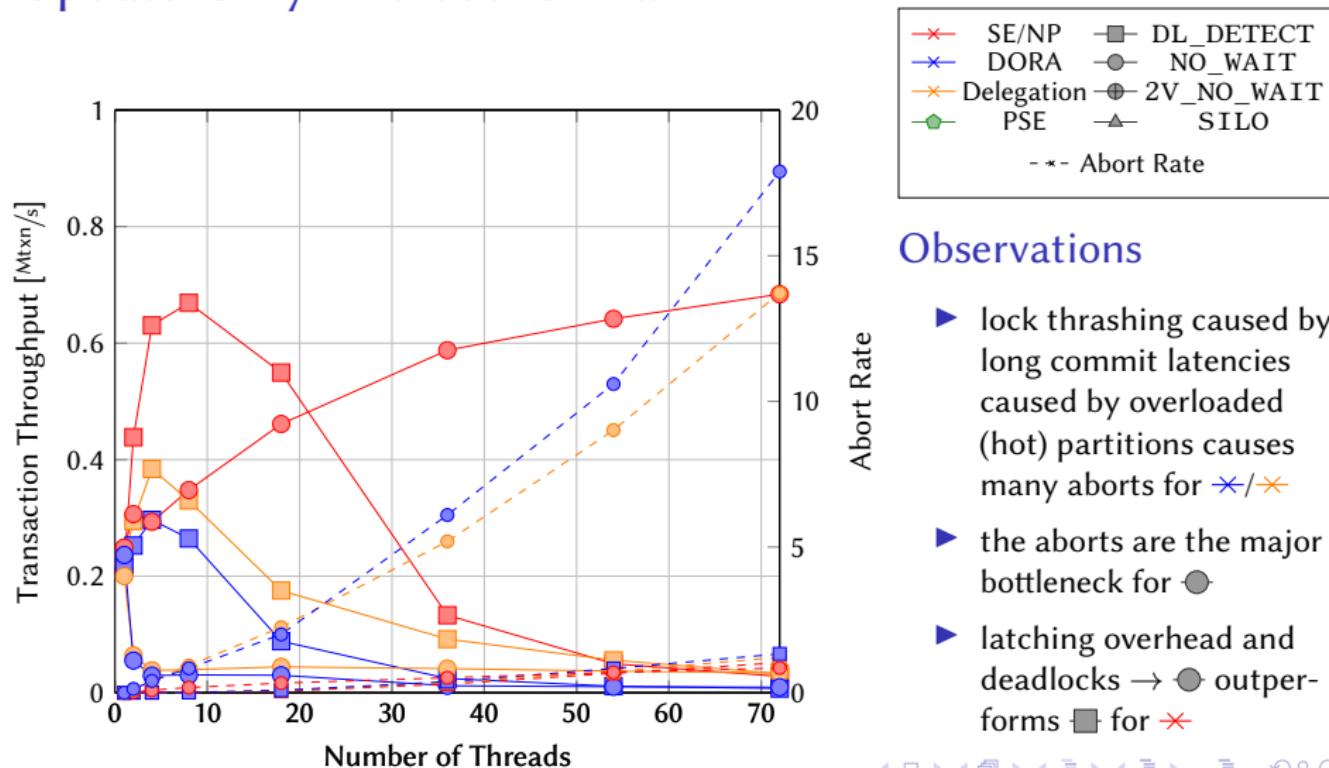
Update-Only Microbenchmark



Observations

- lock thrashing doesn't cause many aborts for \bullet with \ast for few threads
- lock thrashing caused by long commit latencies caused by overloaded (hot) partitions causes many aborts for \ast / \times
- the aborts are the major bottleneck for \bullet

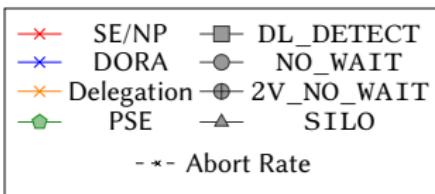
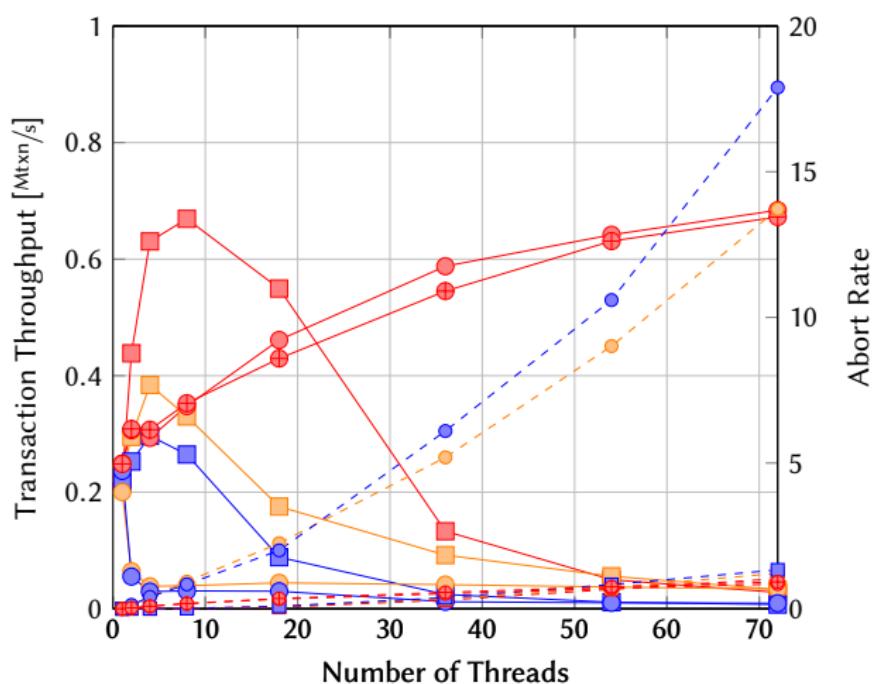
Update-Only Microbenchmark



Observations

- lock thrashing caused by long commit latencies caused by overloaded (hot) partitions causes many aborts for **DORA**/**SE/NP**
- the aborts are the major bottleneck for **NO_WAIT**
- latching overhead and deadlocks → **NO_WAIT** outperforms **DL_DETECT** for **SE/NP**

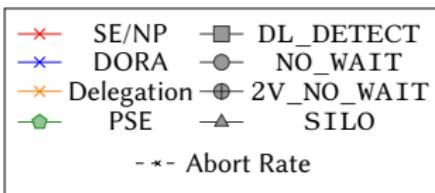
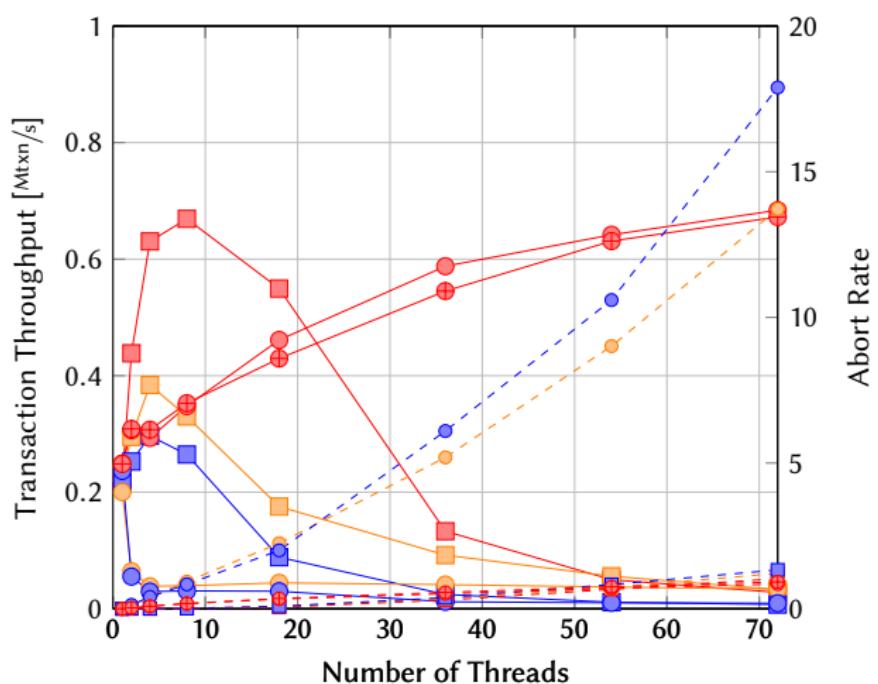
Update-Only Microbenchmark



Observations

- lock thrashing caused by long commit latencies caused by overloaded (hot) partitions causes many aborts for \times/\ast
- the aborts are the major bottleneck for \bullet
- latching overhead and deadlocks $\rightarrow \bullet$ outperforms \blacksquare for \ast

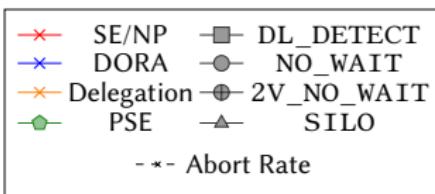
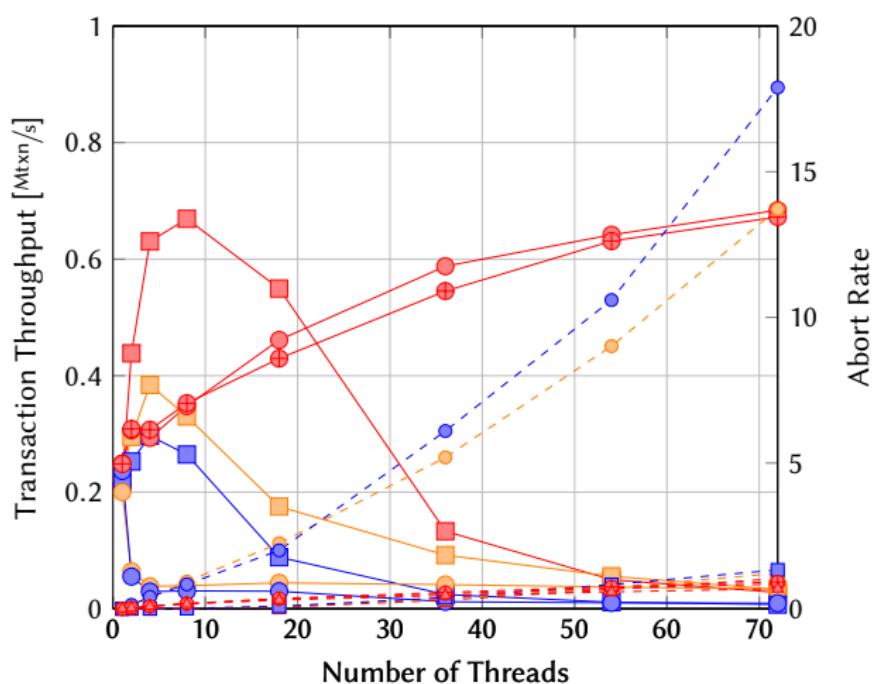
Update-Only Microbenchmark



Observations

- the aborts are the major bottleneck for \bullet
- latching overhead and deadlocks $\rightarrow \bullet$ outperforms \blacksquare for \times
- for update-only \bullet and \circ behave identical

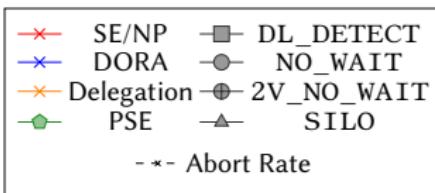
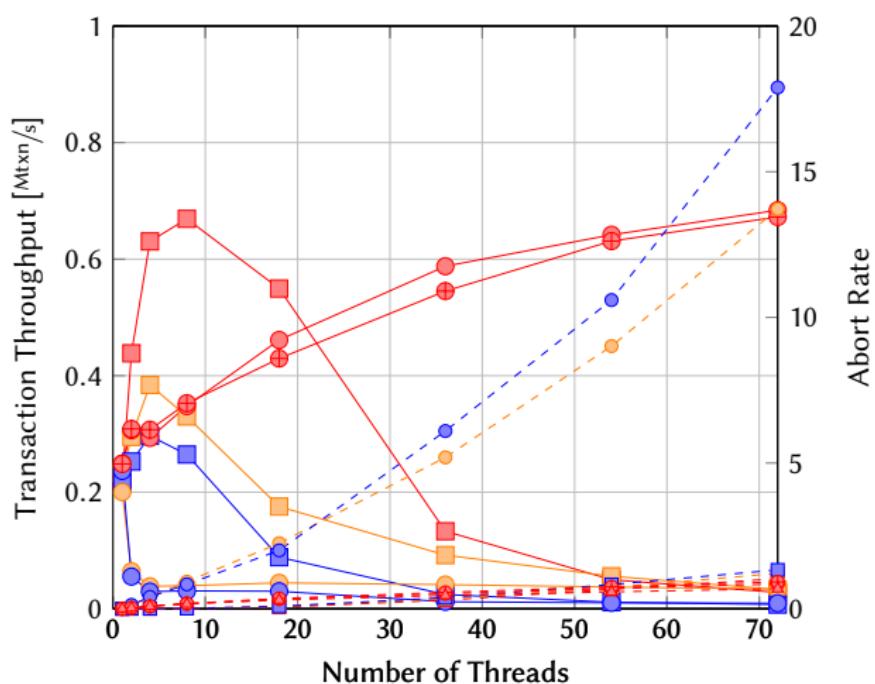
Update-Only Microbenchmark



Observations

- the aborts are the major bottleneck for \bullet
- latching overhead and deadlocks $\rightarrow \bullet$ outperforms \blacksquare for \times
- for update-only \bullet and \circ behave identical

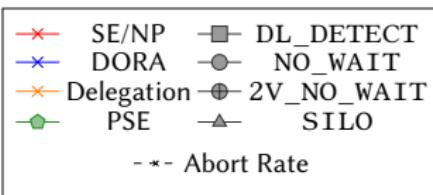
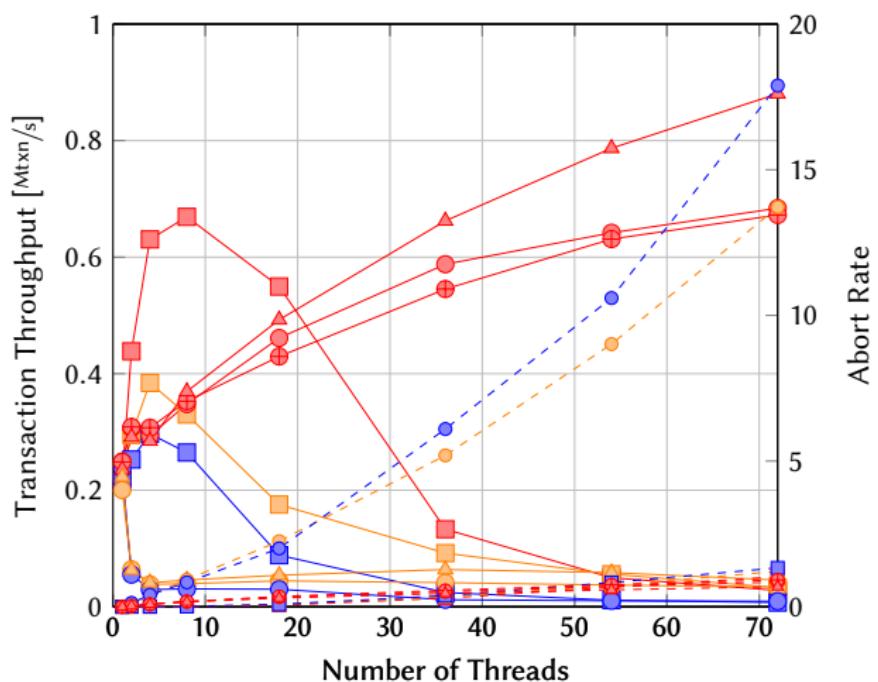
Update-Only Microbenchmark



Observations

- the aborts are the major bottleneck for \bullet
- latching overhead and deadlocks $\rightarrow \bullet$ outperforms \blacksquare for \times
- for update-only \bullet and \circ behave identical
- \blacktriangle causes less aborts than \blacksquare due its optimism

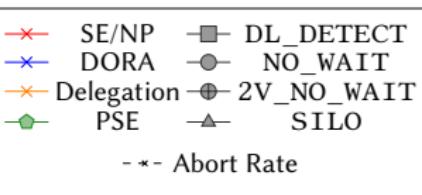
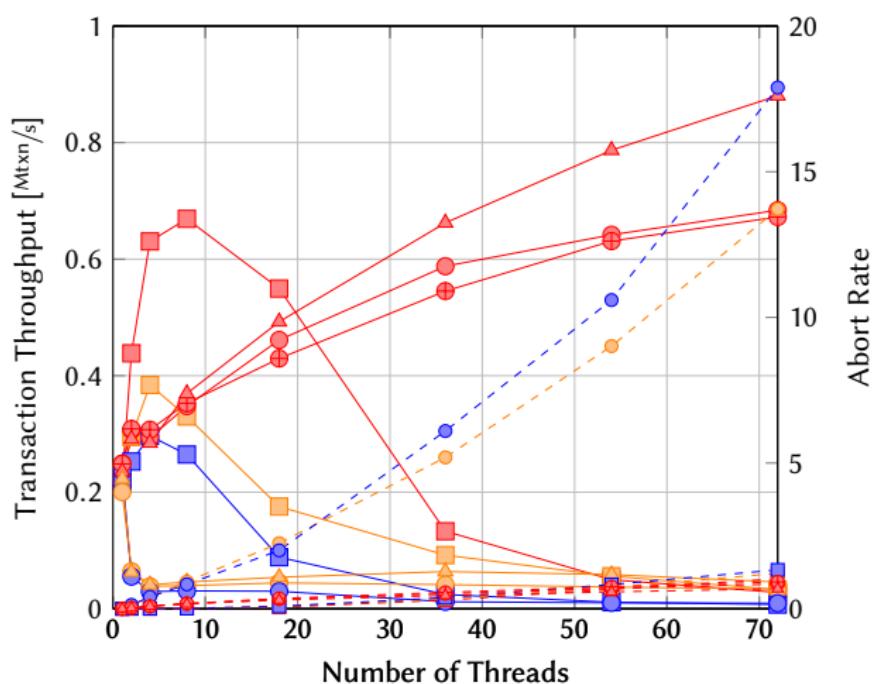
Update-Only Microbenchmark



Observations

- the aborts are the major bottleneck for \bullet
- latching overhead and deadlocks $\rightarrow \bullet$ outperforms \blacksquare for \times
- for update-only \bullet and \circ behave identical
- \blacktriangle causes less aborts than \blacksquare due its optimism

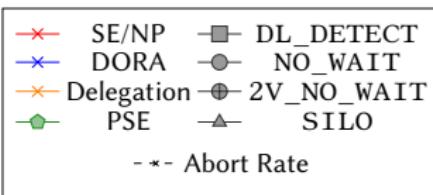
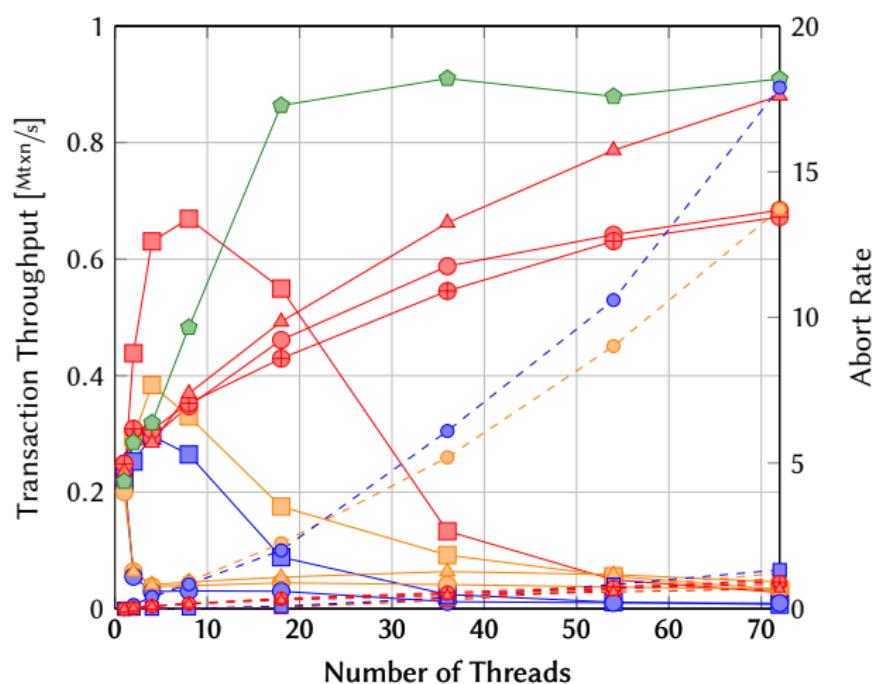
Update-Only Microbenchmark



Observations

- for update-only \bullet and \circ behave identical
- \blacktriangle causes less aborts than \blacksquare due its optimism
- long commit latencies of \times cause high update contention and therefore many aborts (low $[Mtxn/s]$) for \blacktriangle

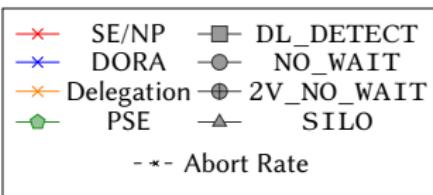
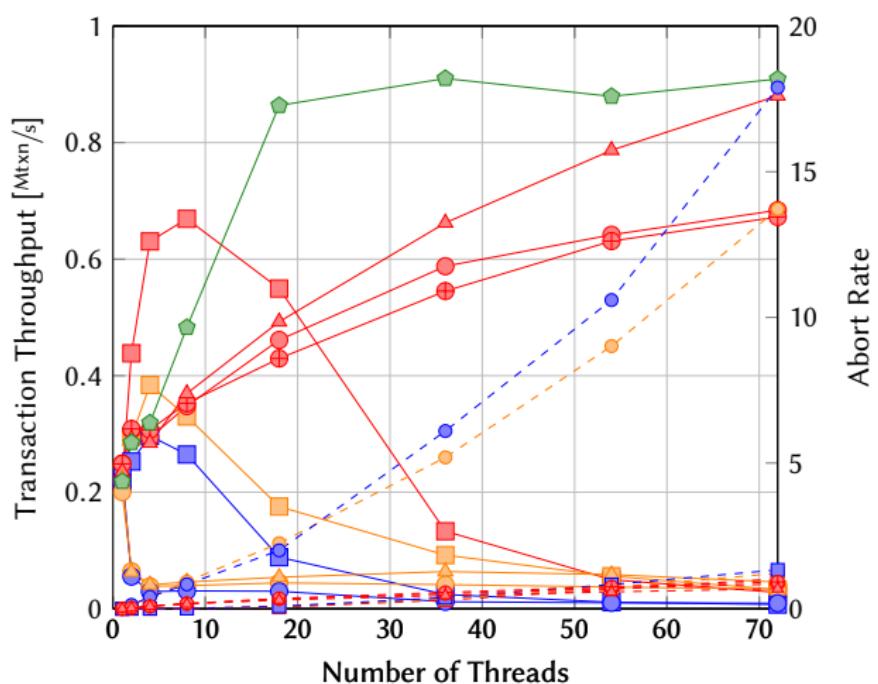
Update-Only Microbenchmark



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 ▲

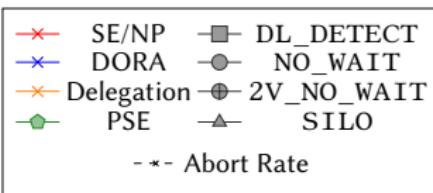
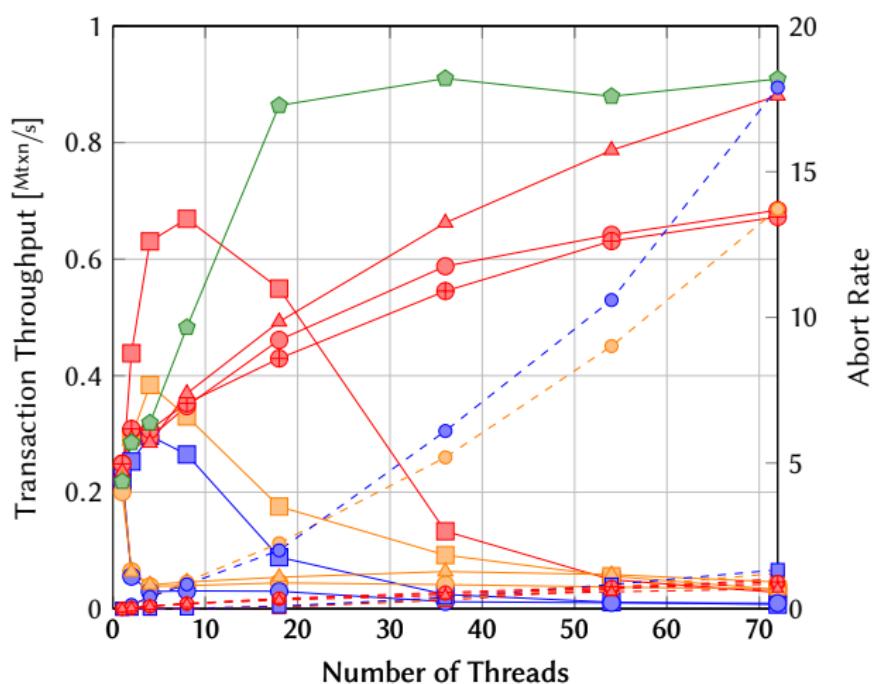
Update-Only Microbenchmark



Observations

- ▲ causes less aborts than ■ due to its optimism
- long commit latencies of * cause high update contention and therefore many aborts (low [Mtxn/s] for ▲)
- coarse-grained partition locking of ♦ is identical for read and update

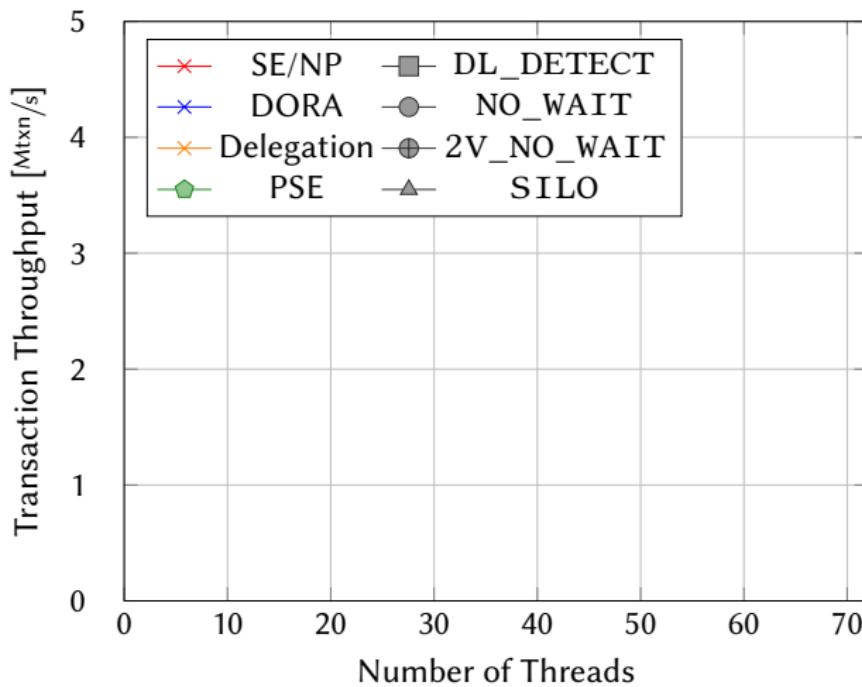
Update-Only Microbenchmark



Observations

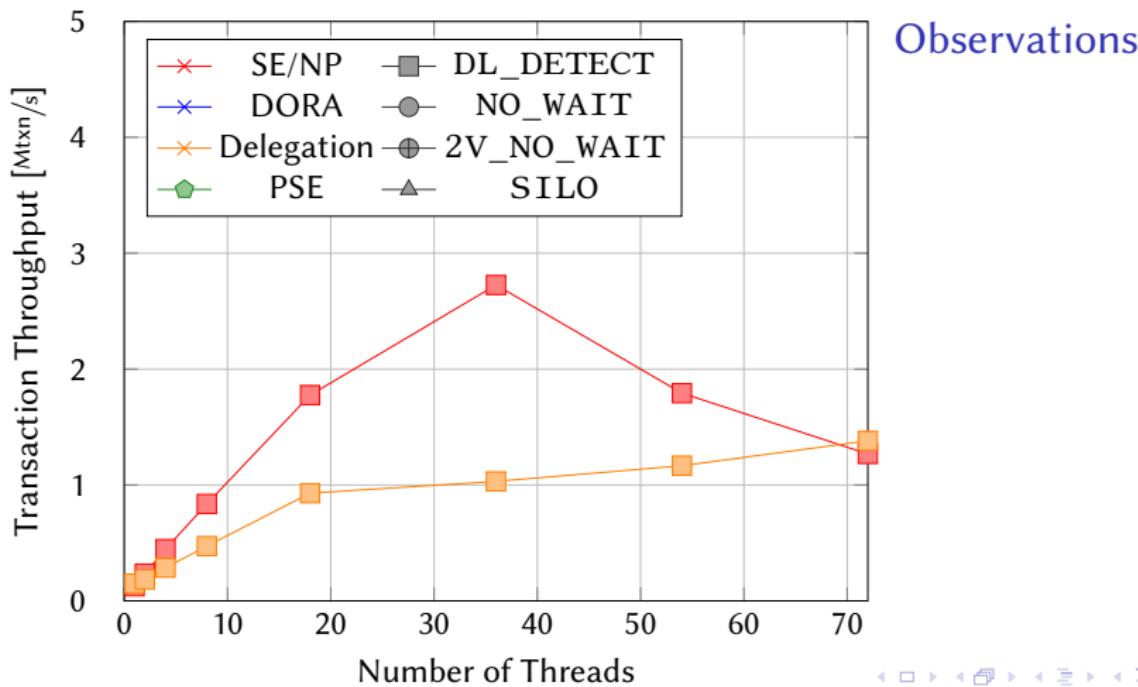
- coarse-grained partition locking of PSE is identical for read and update
- PSE scales according to the number of hot records (each transaction locks 2 of 16 (hot) partitions)

Read-Only YCSB ($\Theta = 0.8$)

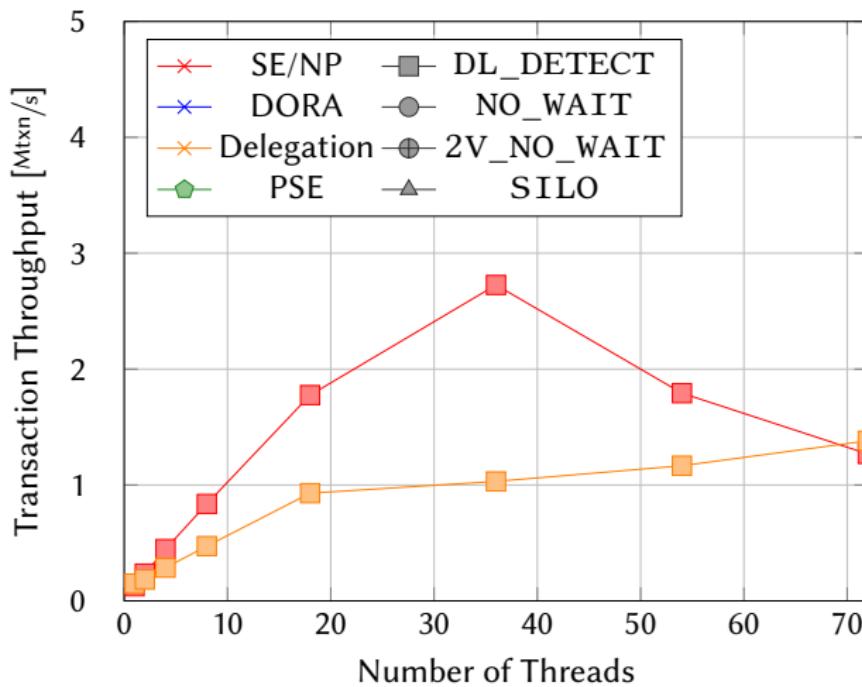


Observations

Read-Only YCSB ($\Theta = 0.8$)



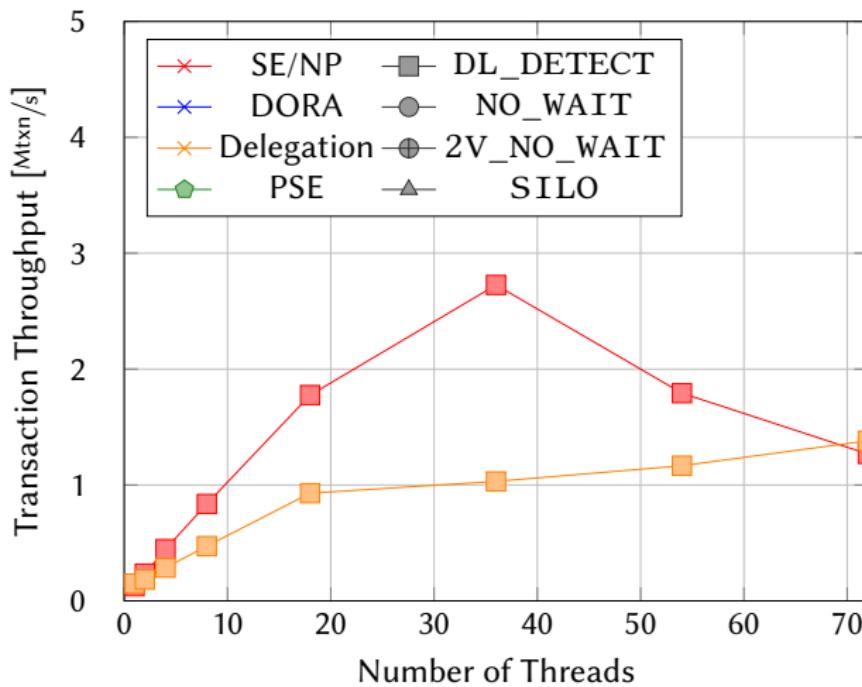
Read-Only YCSB ($\Theta = 0.8$)



Observations

- SE/NP scales well with DL_DETECT until the latch contention becomes a bottleneck

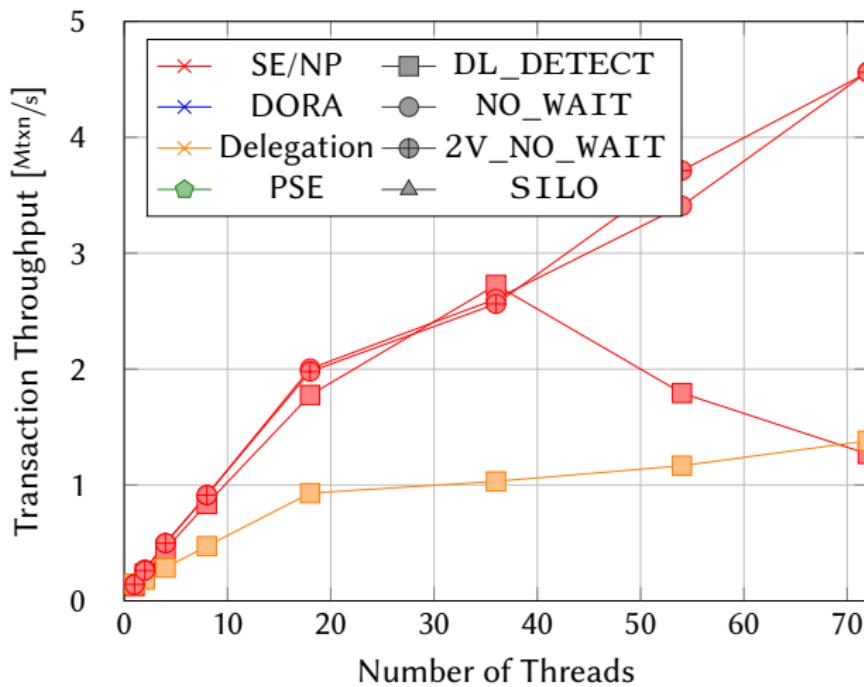
Read-Only YCSB ($\Theta = 0.8$)



Observations

- SE/NP scales well with DL_DETECT until the latch contention becomes a bottleneck
- Delegation (and DORA) doesn't scale well due to partition-unfriendly zipfian access distribution

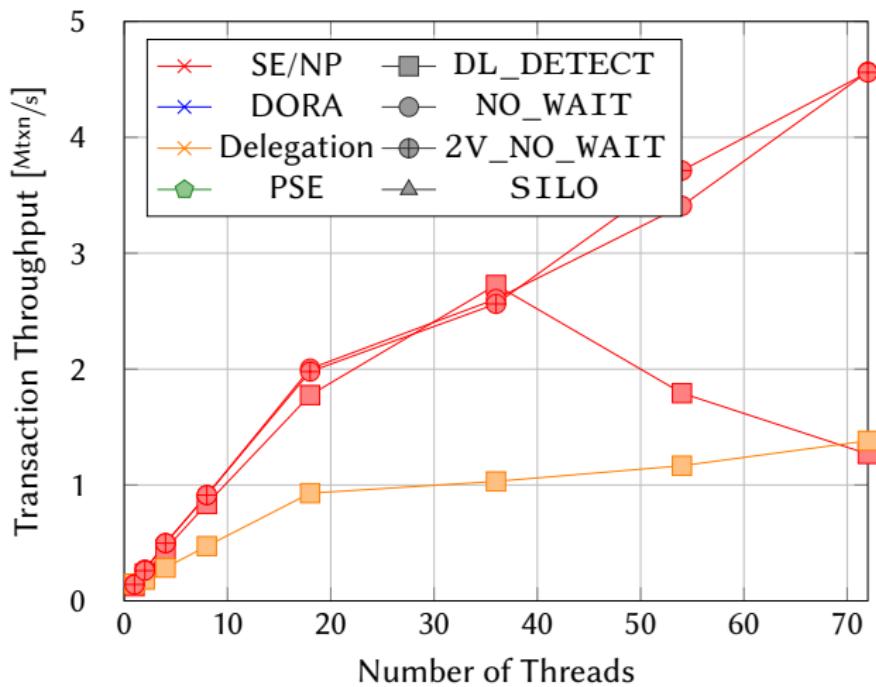
Read-Only YCSB ($\Theta = 0.8$)



Observations

- \ast scales well with \blacksquare until the latch contention becomes a bottleneck
- $\textcolor{orange}{\diamond}$ (and \times) doesn't scale well due to partition-unfriendly zipfian access distribution

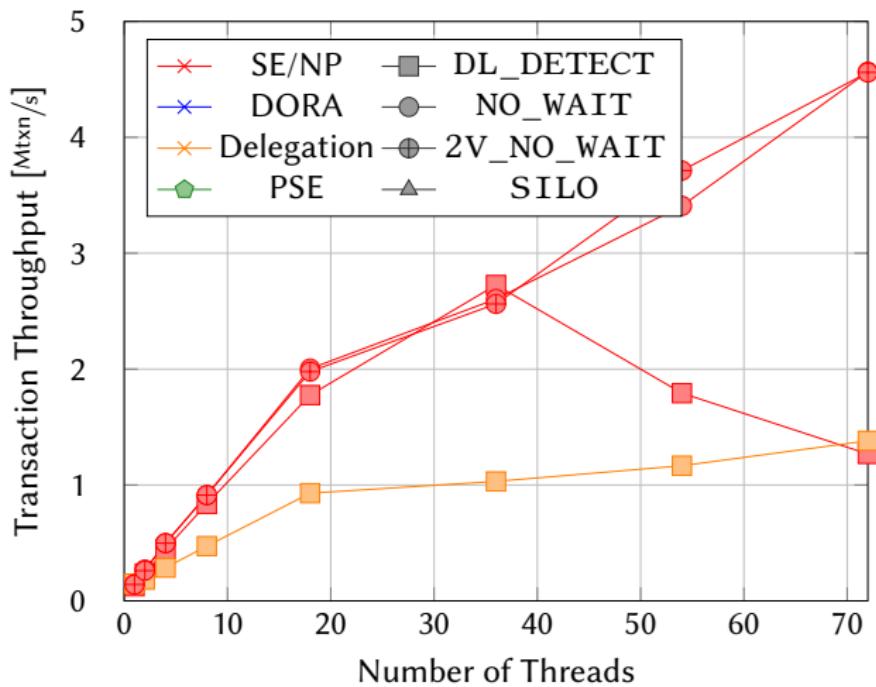
Read-Only YCSB ($\Theta = 0.8$)



Observations

- \ast scales well with \blacksquare until the latch contention becomes a bottleneck
- \ast (and \times) doesn't scale well due to partition-unfriendly zipfian access distribution
- atomics of \bullet scale better than latches of \blacksquare

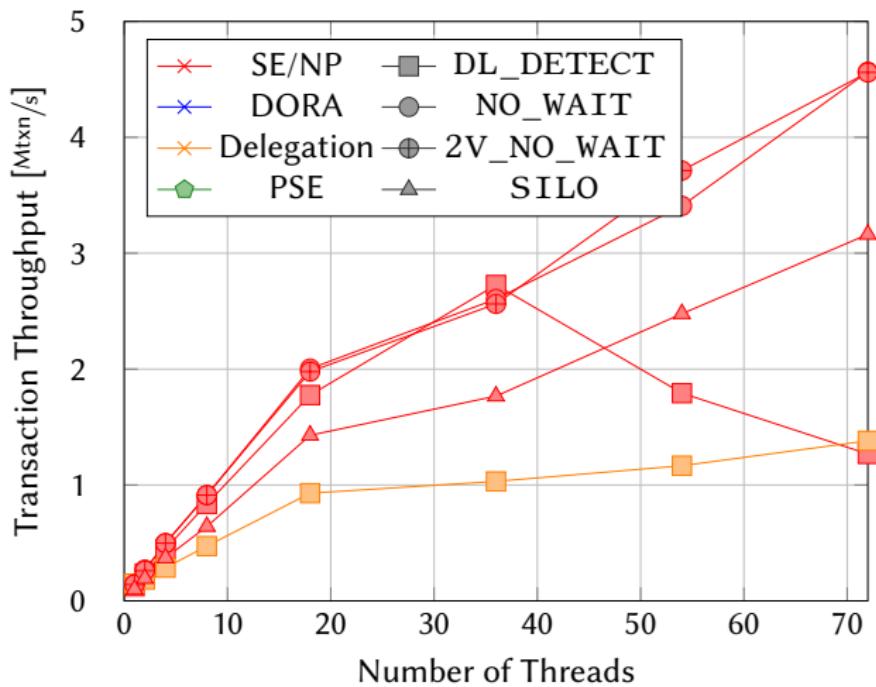
Read-Only YCSB ($\Theta = 0.8$)



Observations

- SE/NP scales well with $\Theta = 0.8$ until the latch contention becomes a bottleneck
- Delegation (and DORA) doesn't scale well due to partition-unfriendly zipfian access distribution
- atomics of NO_WAIT scale better than latches of DL_DETECT
- 2V_NO_WAIT and NO_WAIT perform identical for read-only

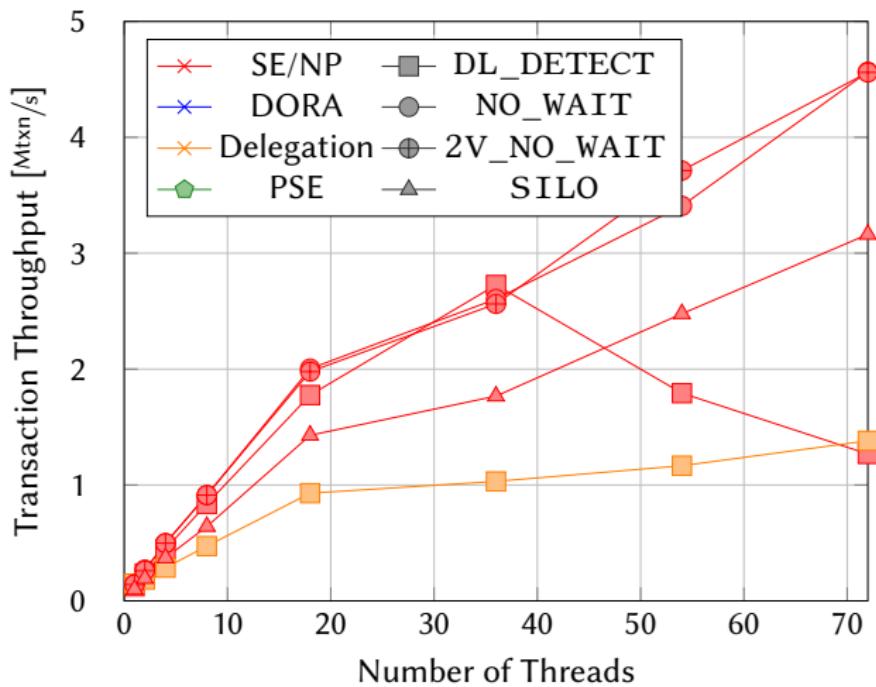
Read-Only YCSB ($\Theta = 0.8$)



Observations

- SE/NP scales well with $\Theta = 0.8$ until the latch contention becomes a bottleneck
- Delegation (and DORA) doesn't scale well due to partition-unfriendly zipfian access distribution
- atomics of NO_WAIT scale better than latches of DL_DETECT
- 2V_NO_WAIT and NO_WAIT perform identical for read-only

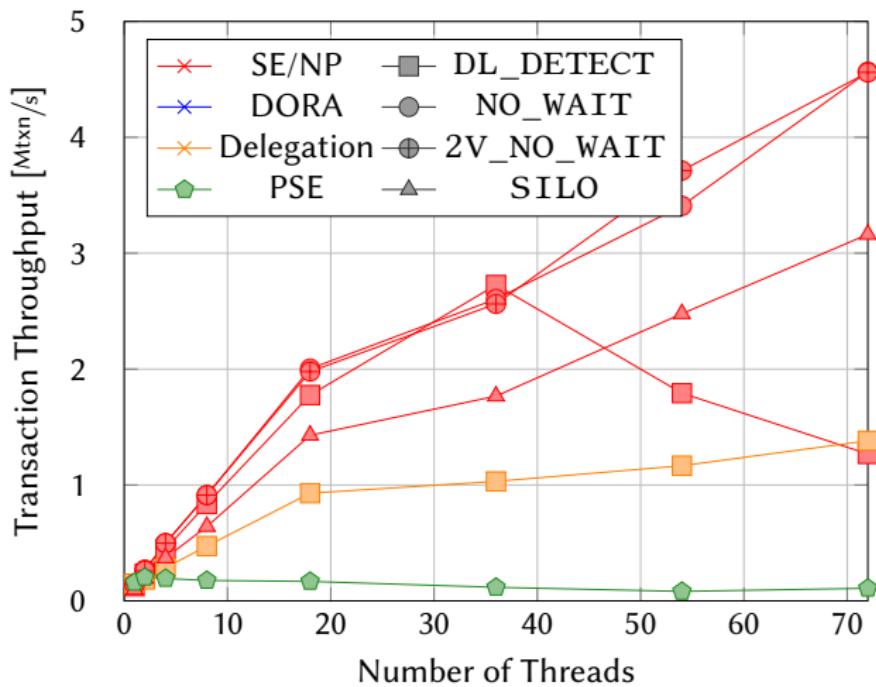
Read-Only YCSB ($\Theta = 0.8$)



Observations

- (and) doesn't 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

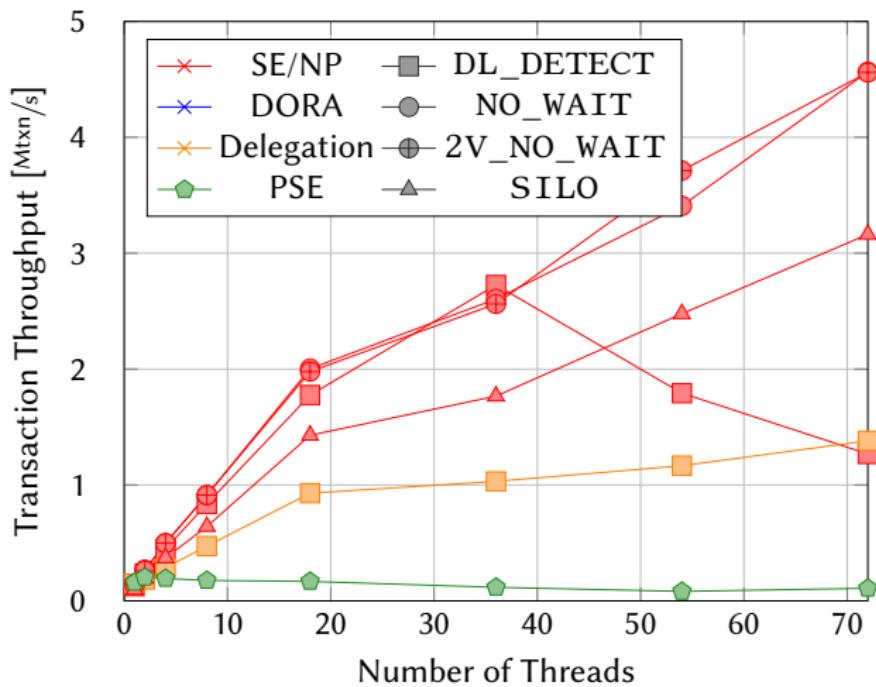
Read-Only YCSB ($\Theta = 0.8$)



Observations

- (and) doesn't 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

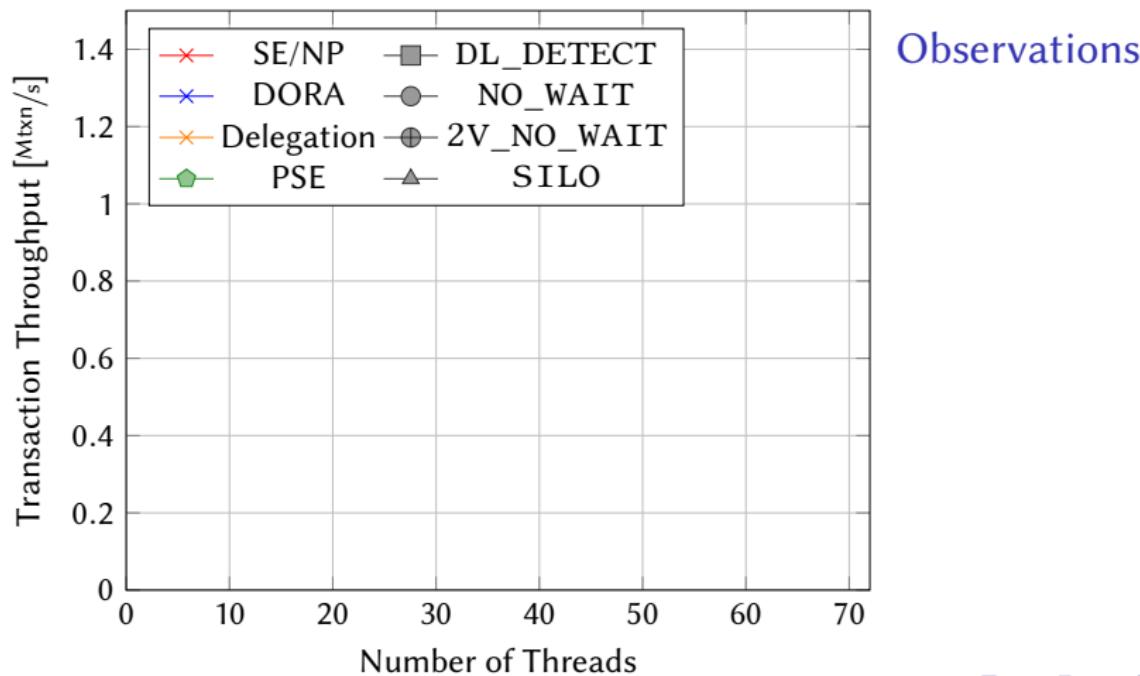
Read-Only YCSB ($\Theta = 0.8$)



Observations

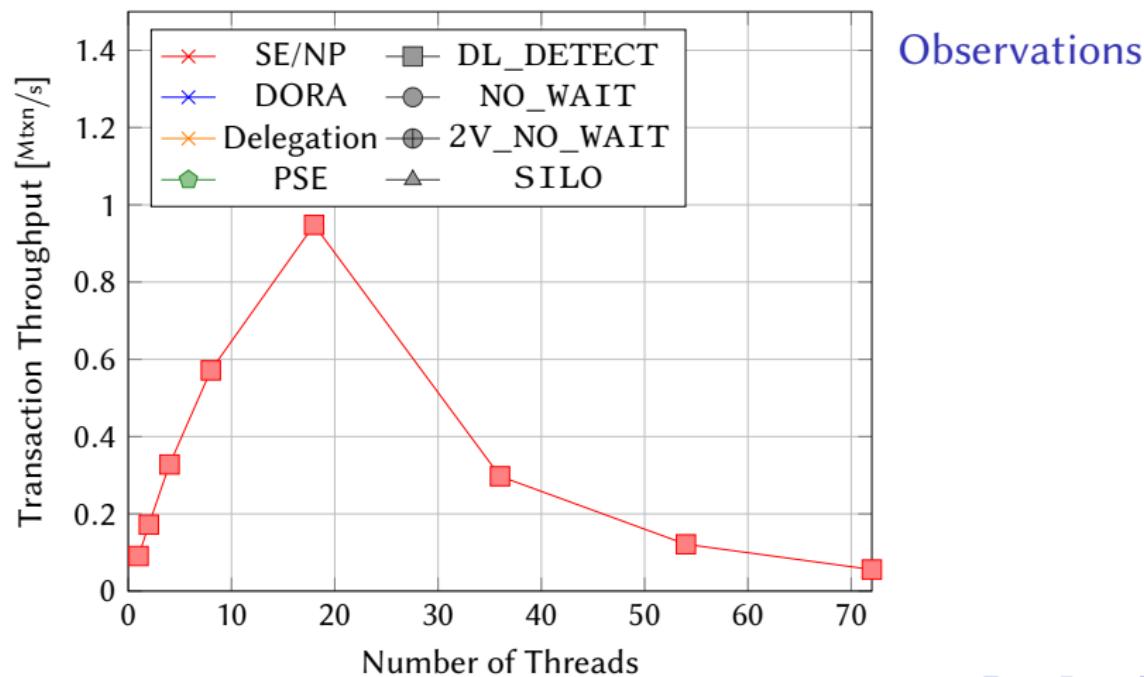
- ▶ atomics of scale better than latches of
- ▶ and perform identical for read-only
- ▶ lags behind due to the overhead of copying read (large) records for validation
- ▶ coarse-grained partition locking of is identical for read and update

Update-Only YCSB ($\Theta = 0.8$)

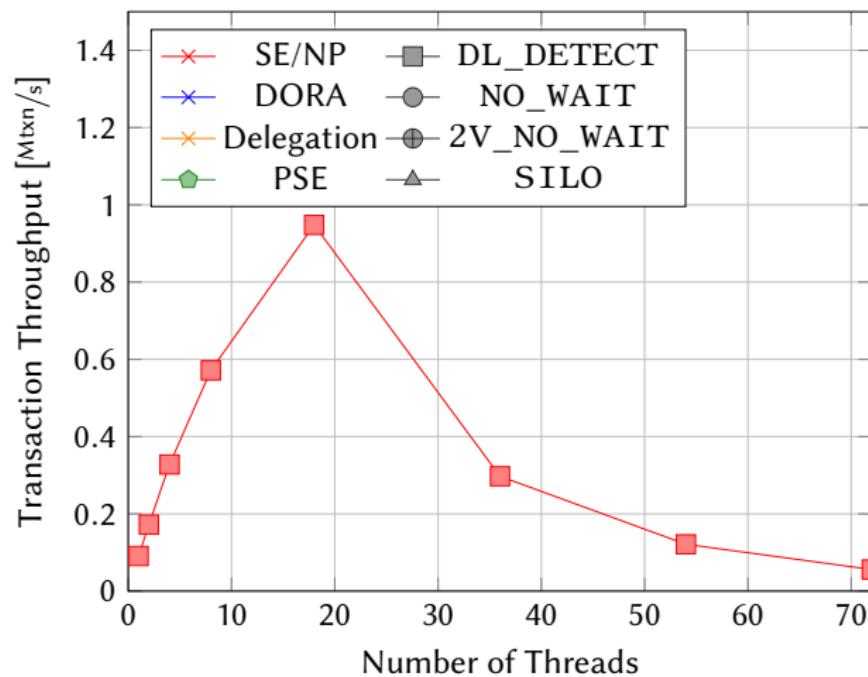


Observations

Update-Only YCSB ($\Theta = 0.8$)



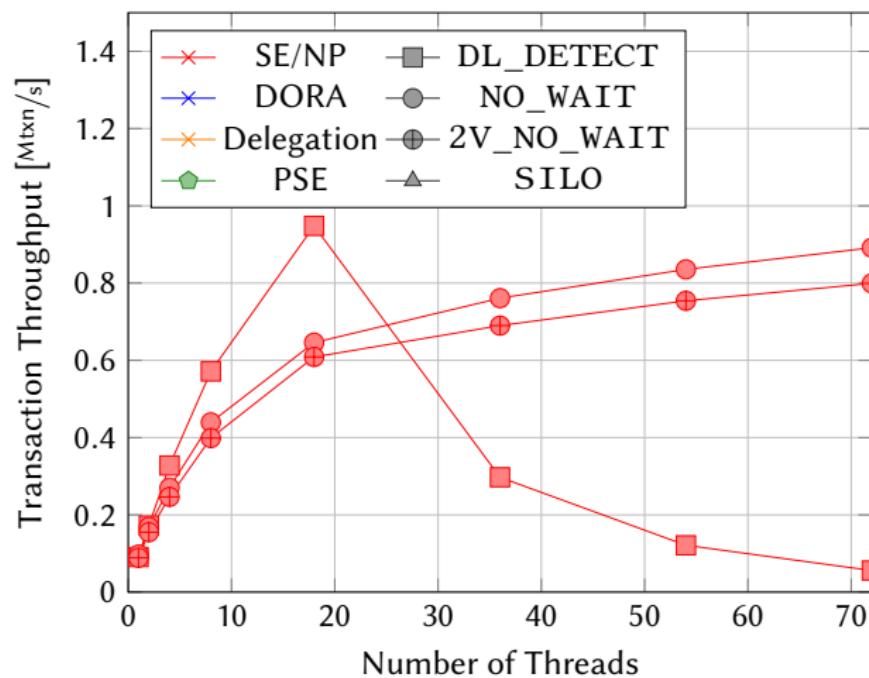
Update-Only YCSB ($\Theta = 0.8$)



Observations

- █ suffers from deadlocks for many threads

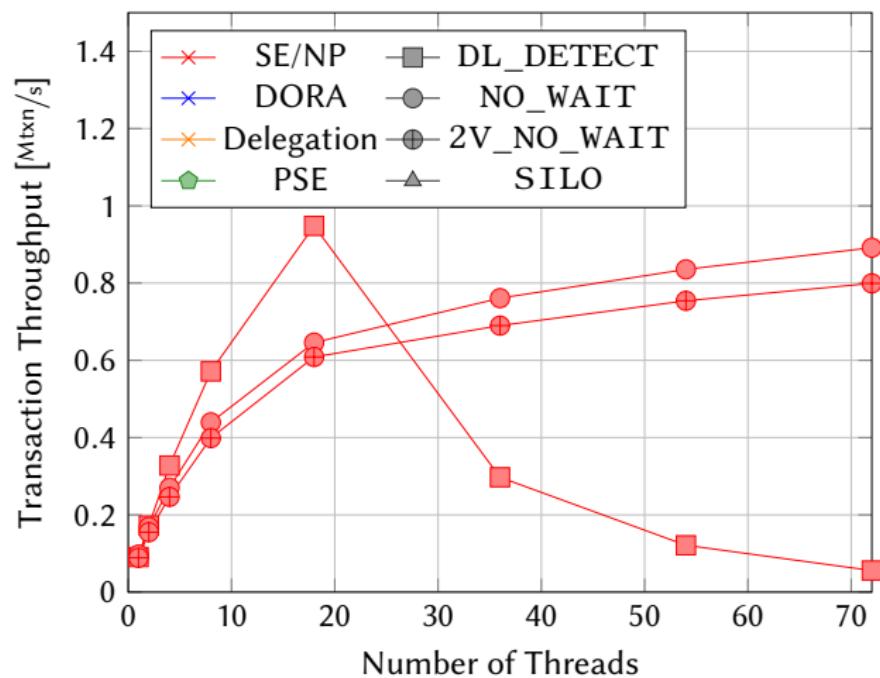
Update-Only YCSB ($\Theta = 0.8$)



Observations

- suffers from deadlocks for many threads

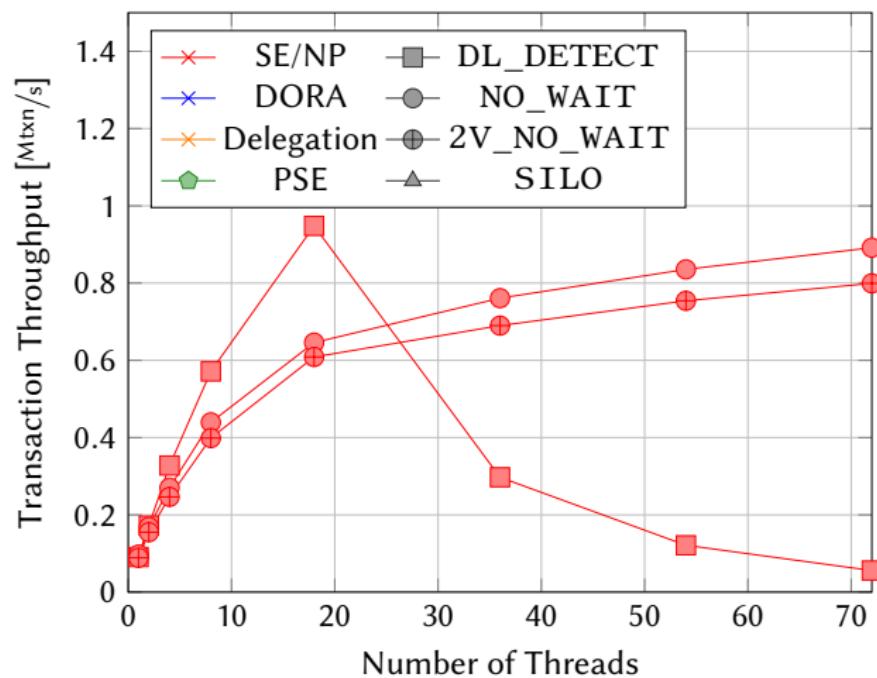
Update-Only YCSB ($\Theta = 0.8$)



Observations

- suffers from deadlocks for many threads
- lock thrashing (aborts for) isn't a bottleneck due to lower contention

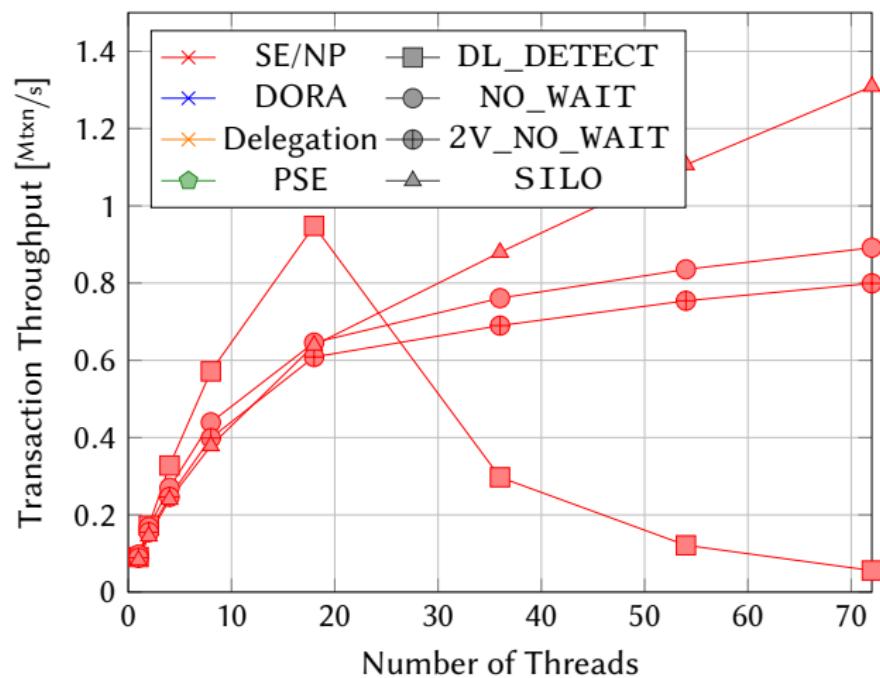
Update-Only YCSB ($\Theta = 0.8$)



Observations

- `DL_DETECT` suffers from deadlocks for many threads
- lock thrashing (aborts for `NO_WAIT`) isn't a bottleneck due to lower contention
- `2V_NO_WAIT` and `NO_WAIT` perform identical for update-only

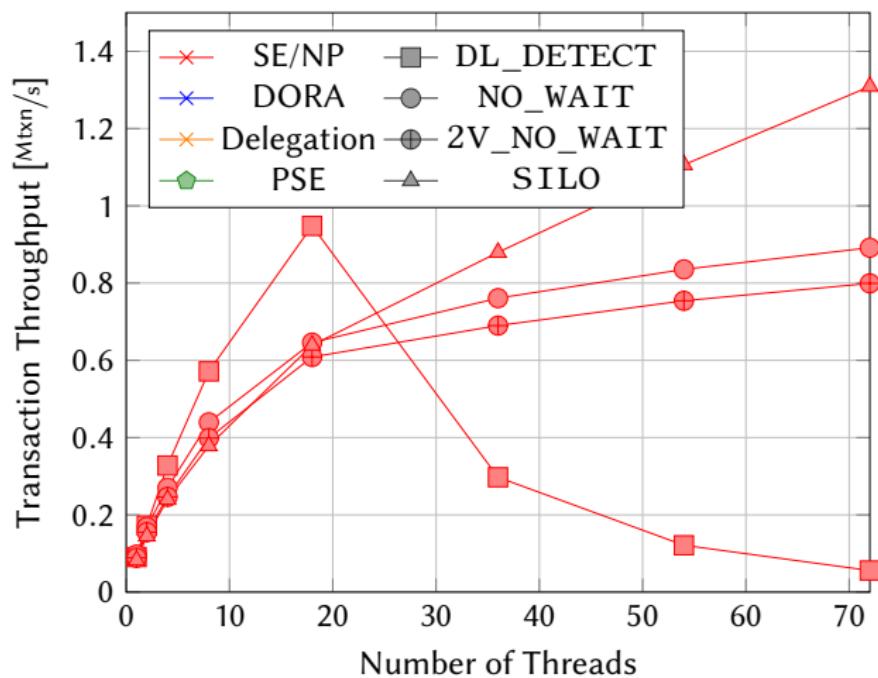
Update-Only YCSB ($\Theta = 0.8$)



Observations

- `DL_DETECT` suffers from deadlocks for many threads
- lock thrashing (aborts for `NO_WAIT`) isn't a bottleneck due to lower contention
- `2V_NO_WAIT` and `NO_WAIT` perform identical for update-only

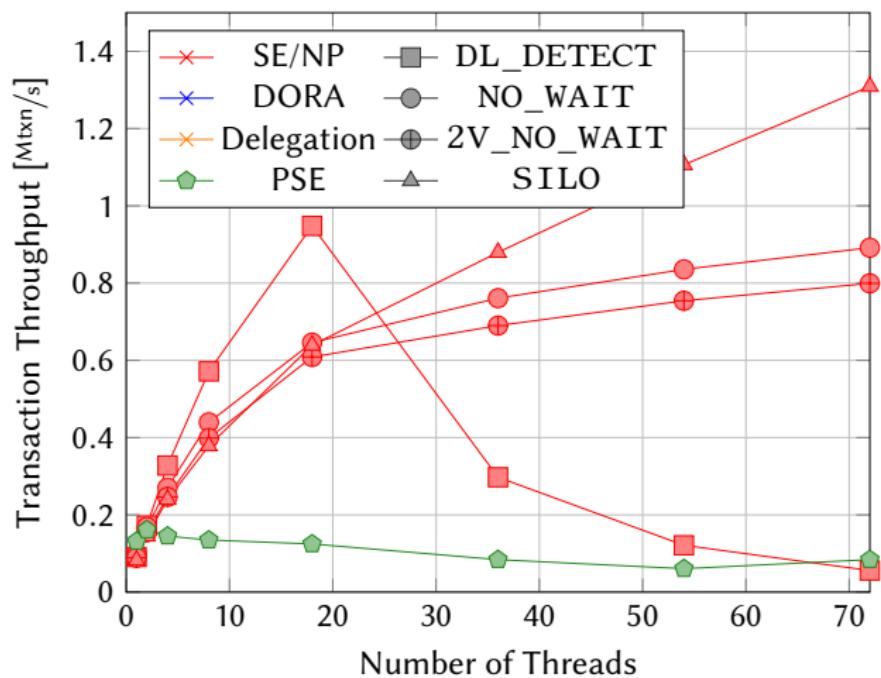
Update-Only YCSB ($\Theta = 0.8$)



Observations

- suffers from deadlocks for many threads
- lock thrashing (aborts for) isn't a bottleneck due to lower contention
- and perform identical for update-only
- causes less aborts than due to its optimism
→ higher [$Mtxn/s$]

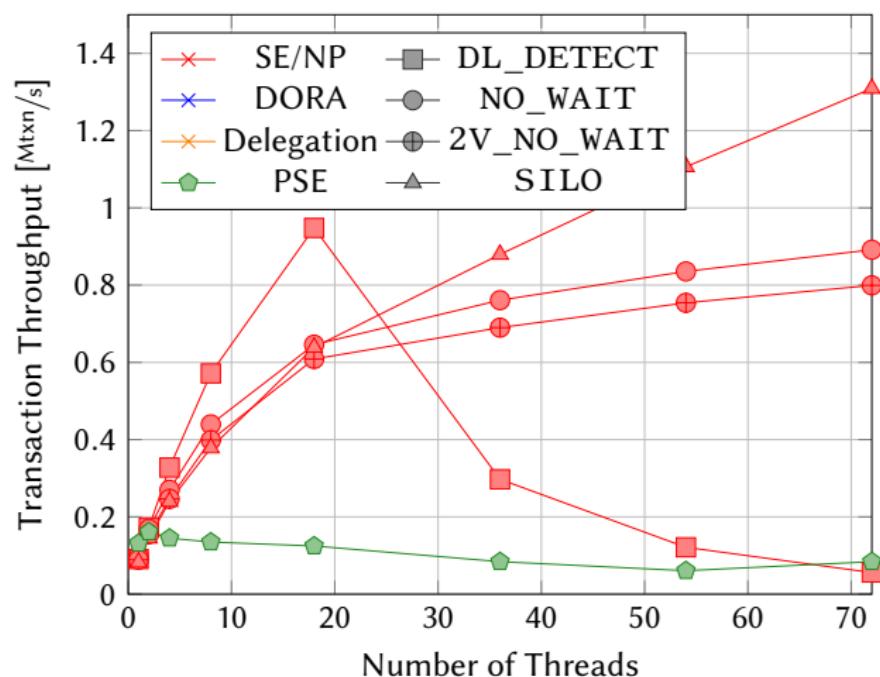
Update-Only YCSB ($\Theta = 0.8$)



Observations

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

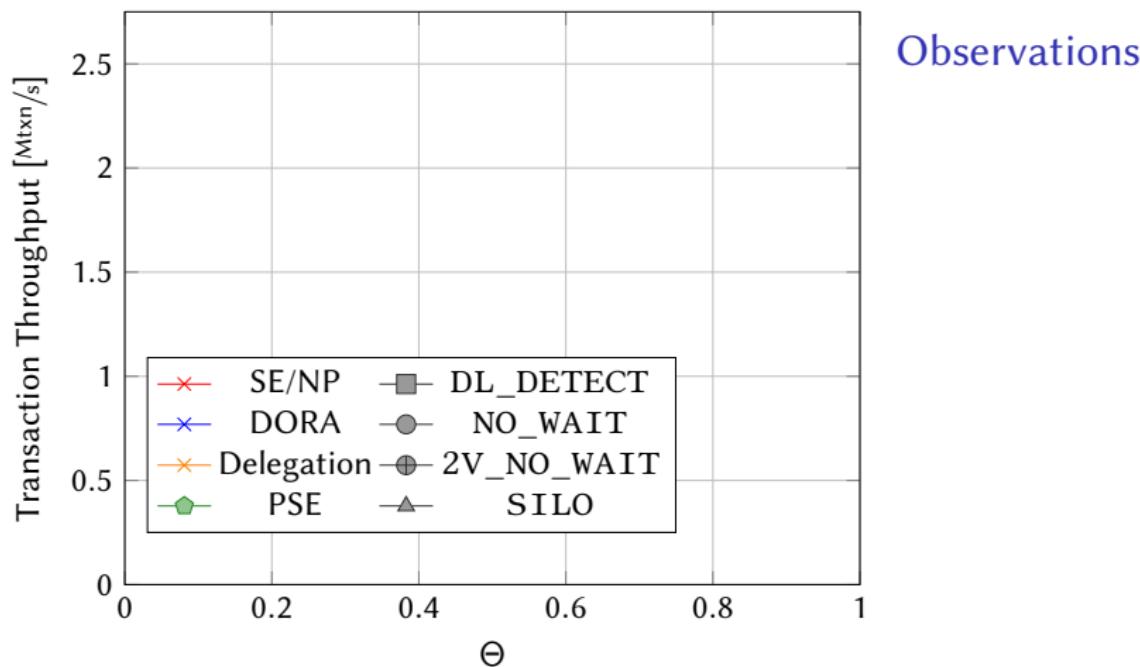
Update-Only YCSB ($\Theta = 0.8$)



Observations

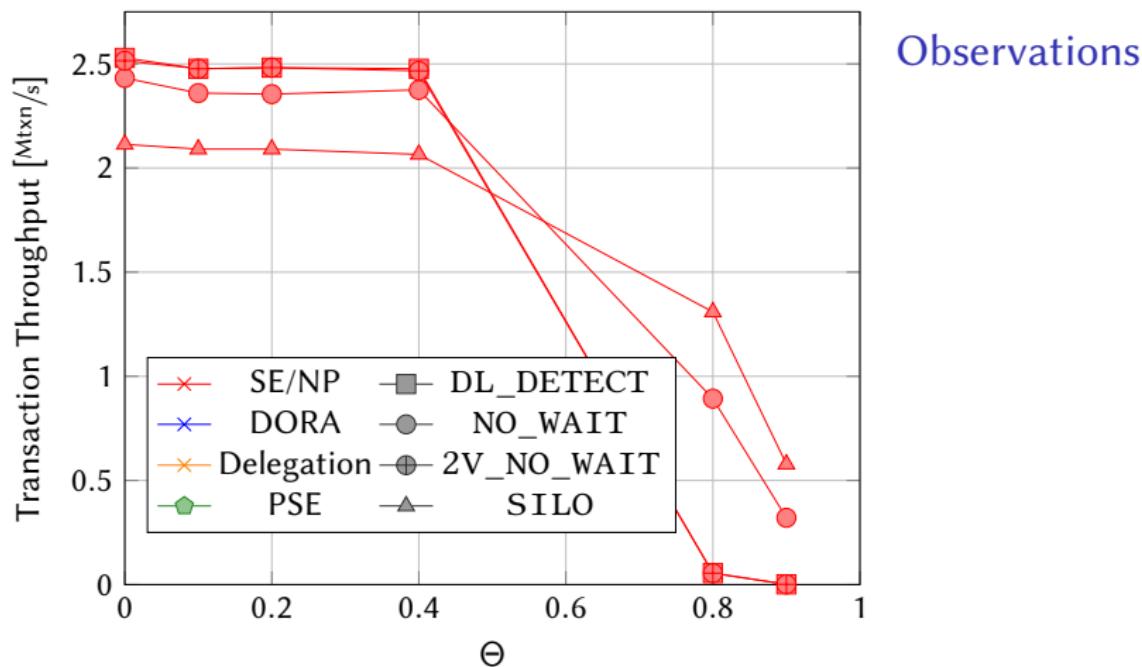
- lock thrashing (aborts for \bullet) isn't a bottleneck due to lower contention
- \circlearrowleft and \circlearrowright perform identical for update-only
- \triangle causes less aborts than \bullet due its optimism
→ higher $[Mtxn/s]$
- \diamond (and \times/\times) doesn't scale well due to partition-unfriendly zipfian access distribution

Update-Only YCSB (72 Threads)



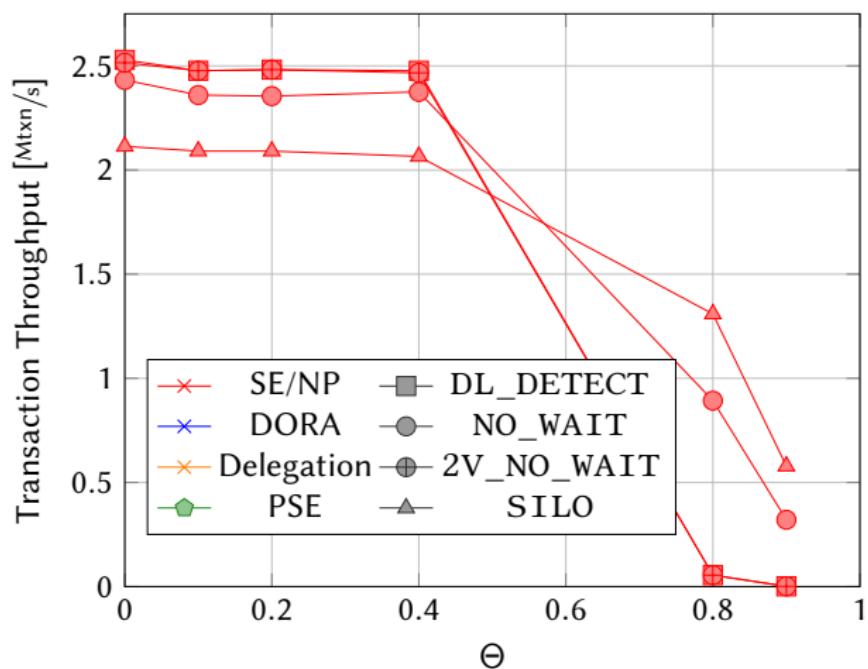
Observations

Update-Only YCSB (72 Threads)



Observations

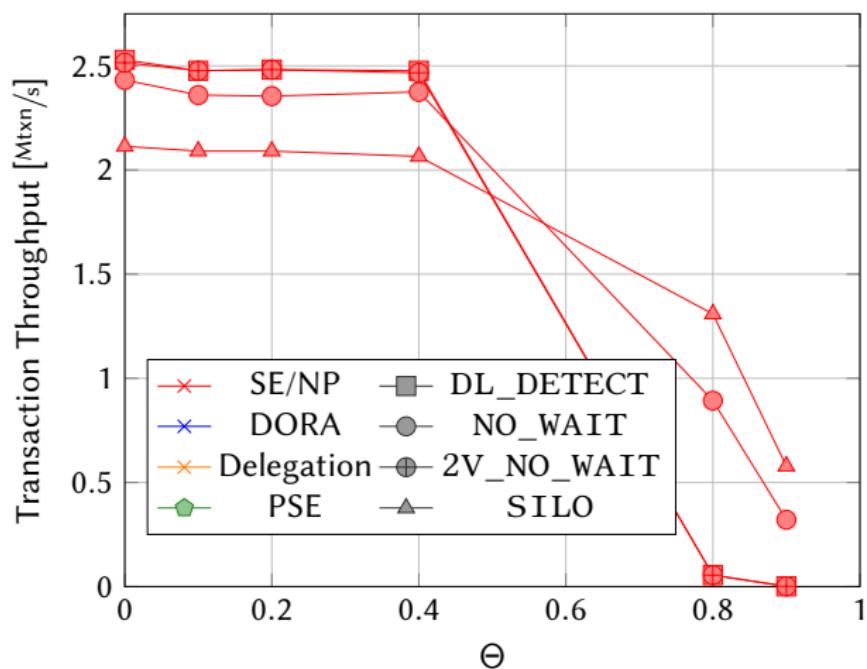
Update-Only YCSB (72 Threads)



Observations

- ▶ for $\Theta \leq 0.4$ the contention is very low → high concurrency possible

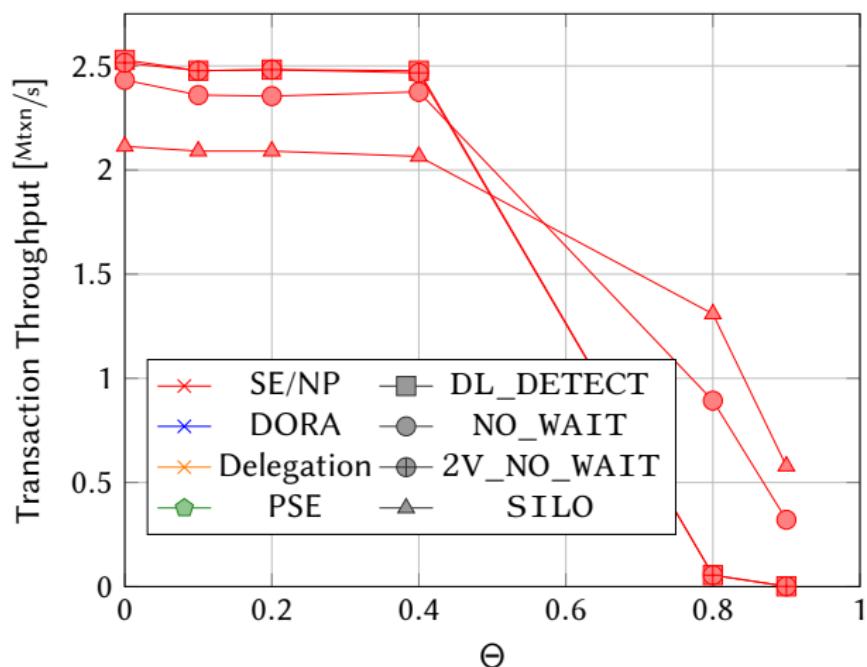
Update-Only YCSB (72 Threads)



Observations

- ▶ for $\Theta \leq 0.4$ the contention is very low → high concurrency possible
- ▶ copying records imposes an overhead to \bullet/Δ

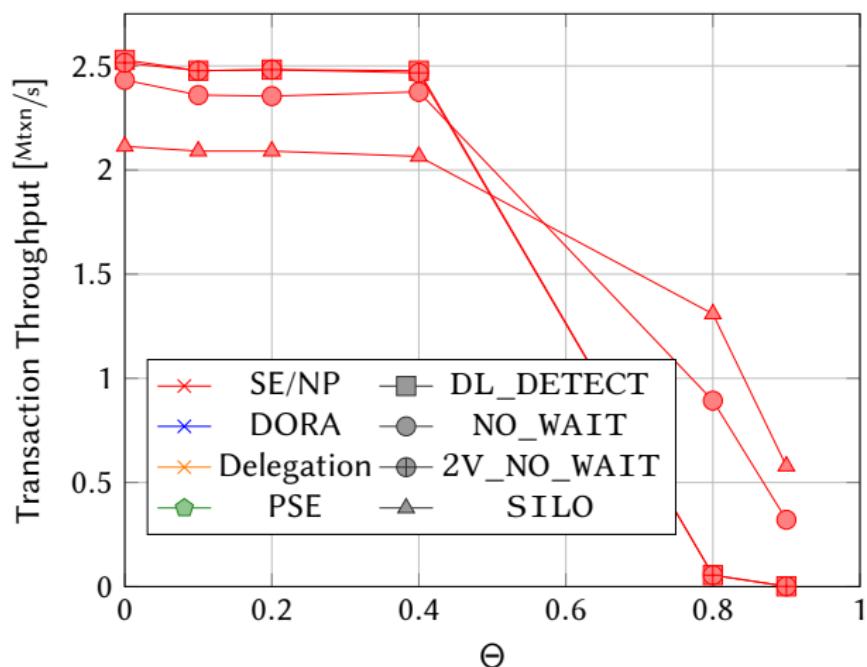
Update-Only YCSB (72 Threads)



Observations

- ▶ for $\Theta \leq 0.4$ the contention is very low → high concurrency possible
- ▶ copying records imposes an overhead to \bullet/Δ
- ▶ atomics of \bullet scale better than latches of \blacksquare

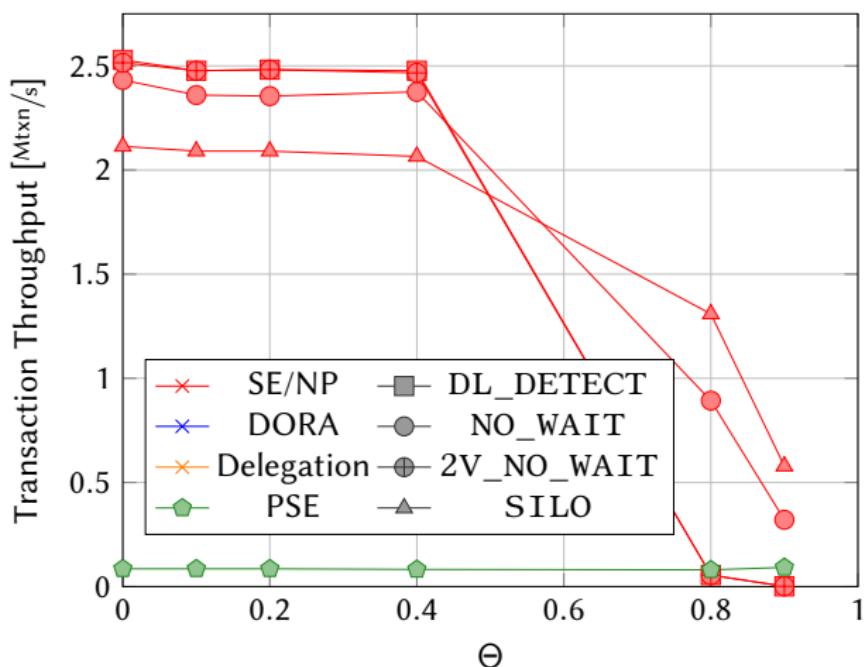
Update-Only YCSB (72 Threads)



Observations

- ▶ for $\Theta \leq 0.4$ the contention is very low → high concurrency possible
- ▶ copying records imposes an overhead to \bullet/Δ
- ▶ atomics of \bullet scale better than latches of \blacksquare
- ▶ Δ causes less aborts than \bullet due to its optimism → higher $[Mtxn/s]$

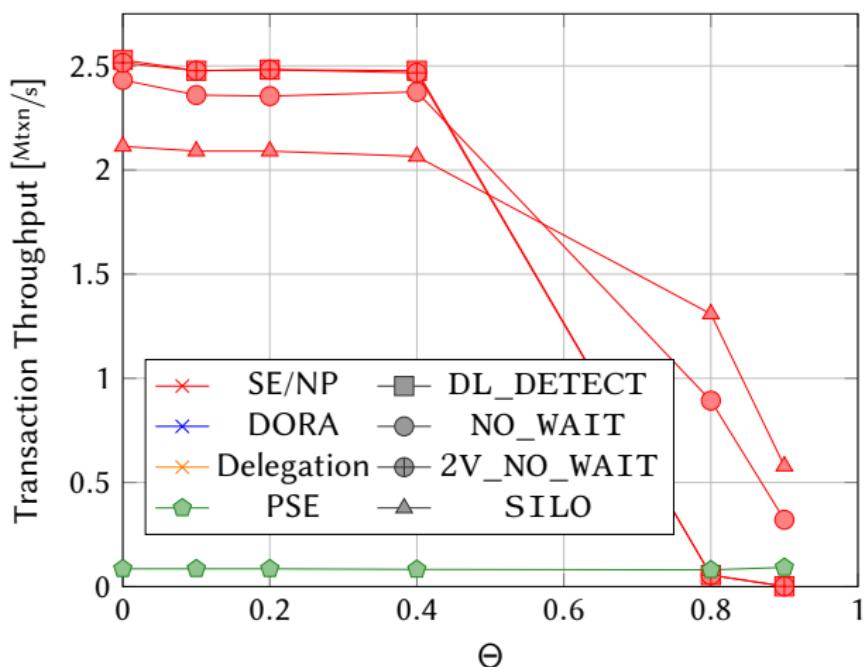
Update-Only YCSB (72 Threads)



Observations

- ▶ for $\Theta \leq 0.4$ the contention is very low → high concurrency possible
- ▶ copying records imposes an overhead to \bullet/Δ
- ▶ atomics of \bullet scale better than latches of \blacksquare
- ▶ Δ causes less aborts than \bullet due to its optimism → higher $[Mtxn/s]$

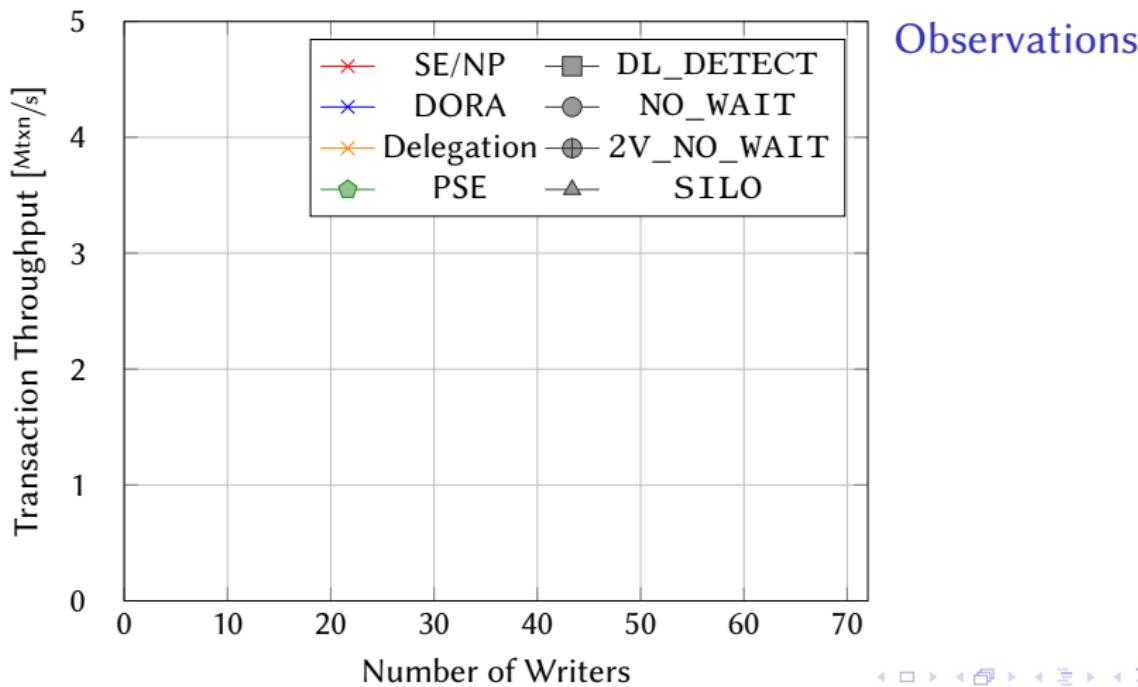
Update-Only YCSB (72 Threads)



Observations

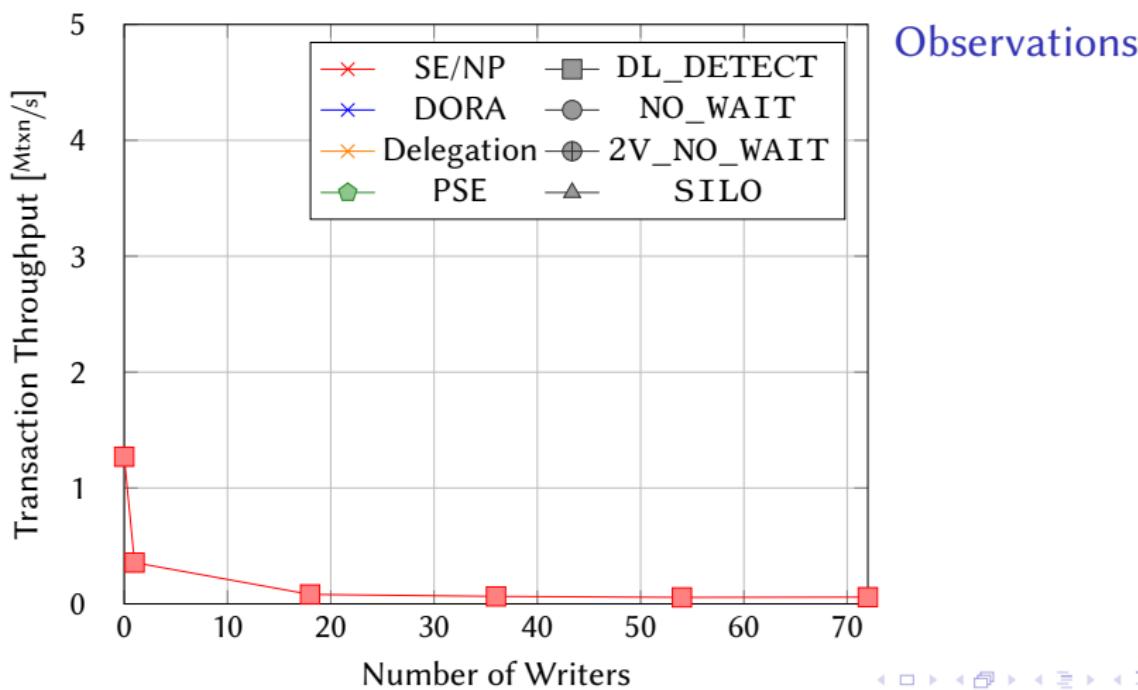
- ▶ copying records imposes an overhead to \bullet/\blacktriangle
- ▶ atomics of \bullet scale better than latches of \blacksquare
- ▶ \blacktriangle causes less aborts than \bullet due its optimism \rightarrow higher $[Mtxn/s]$
- ▶ \blacklozenge doesn't scale well due to partition-unfriendly zipfian access distribution

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



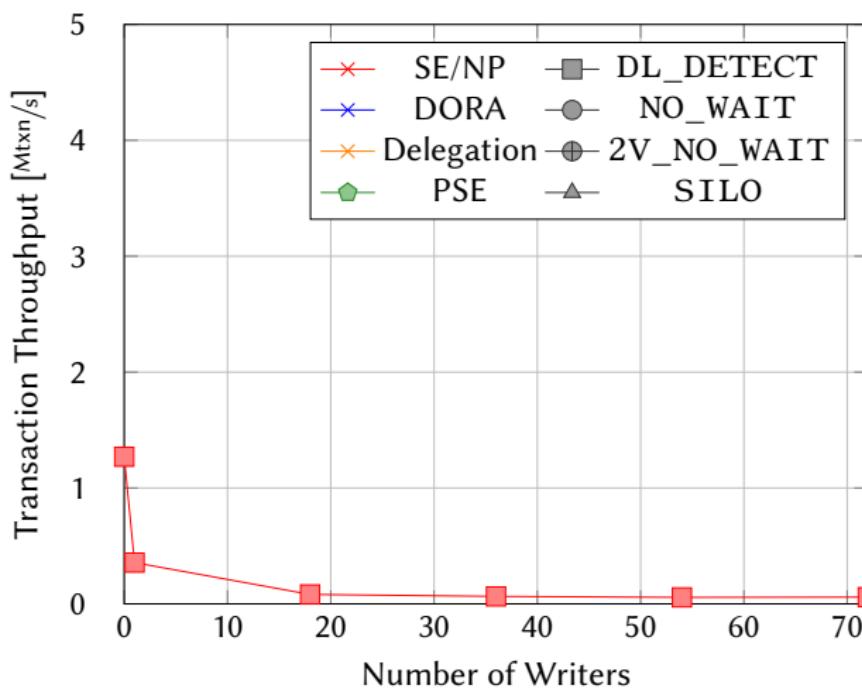
Observations

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



Observations

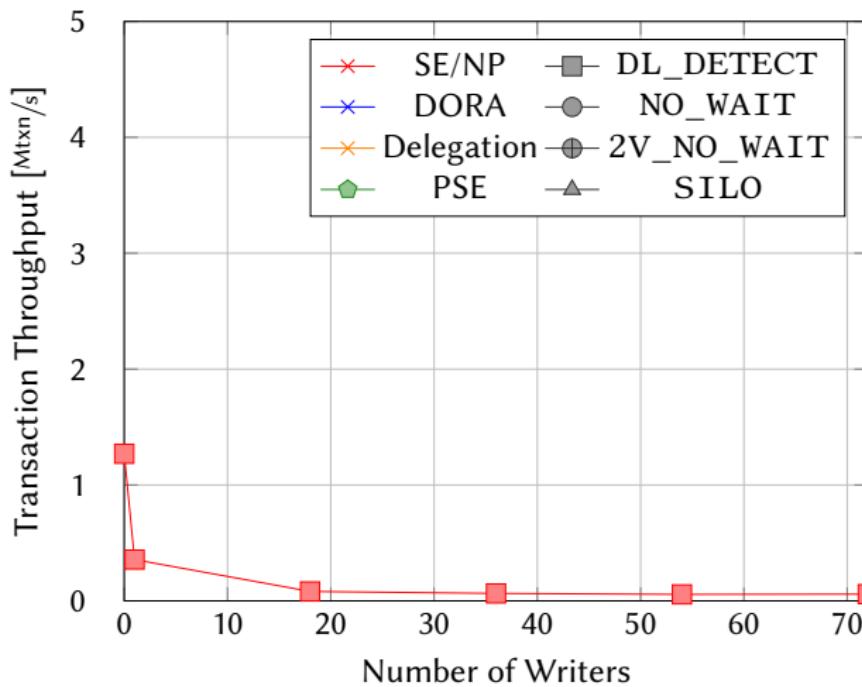
Mixed YCSB ($\Theta = 0.8$, 72 Threads)



Observations

- suffers from latch contention for 72 reading threads

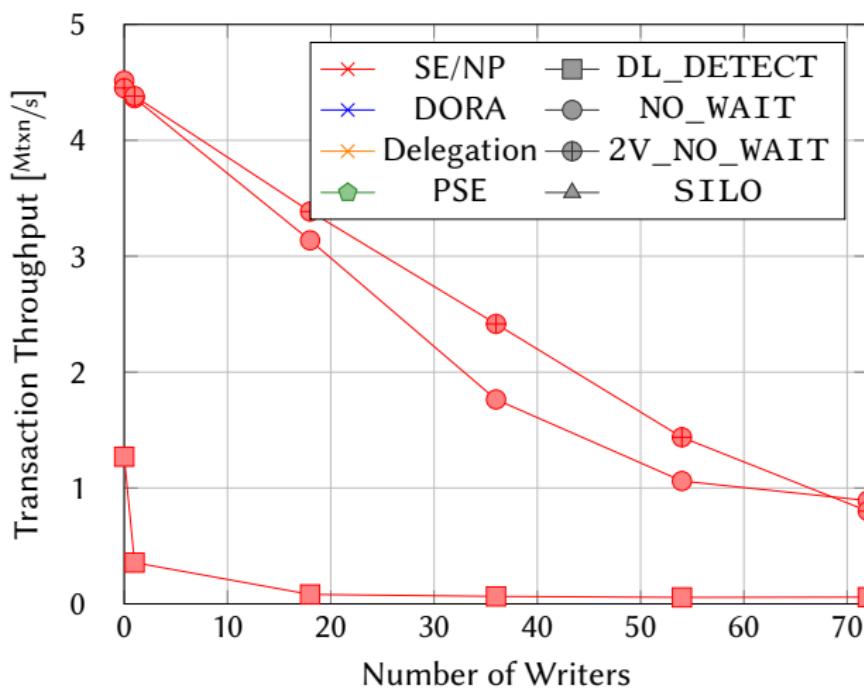
Mixed YCSB ($\Theta = 0.8$, 72 Threads)



Observations

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

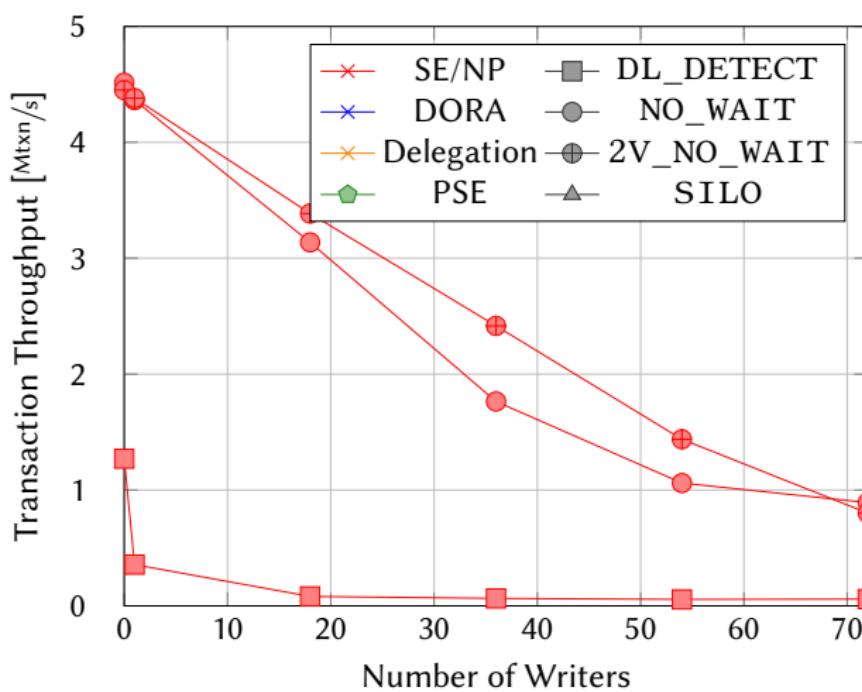
Mixed YCSB ($\Theta = 0.8$, 72 Threads)



Observations

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

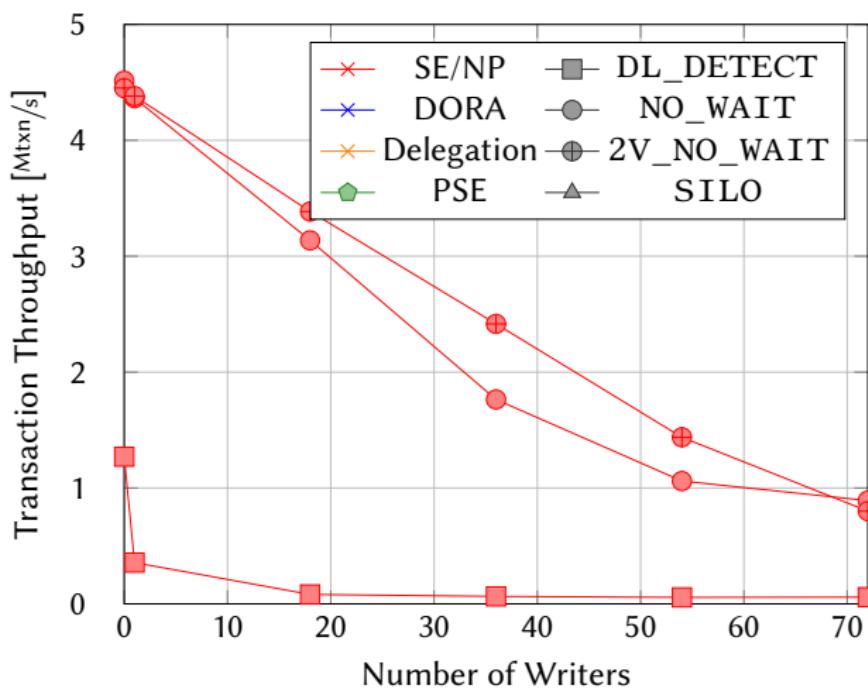
Mixed YCSB ($\Theta = 0.8$, 72 Threads)



Observations

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

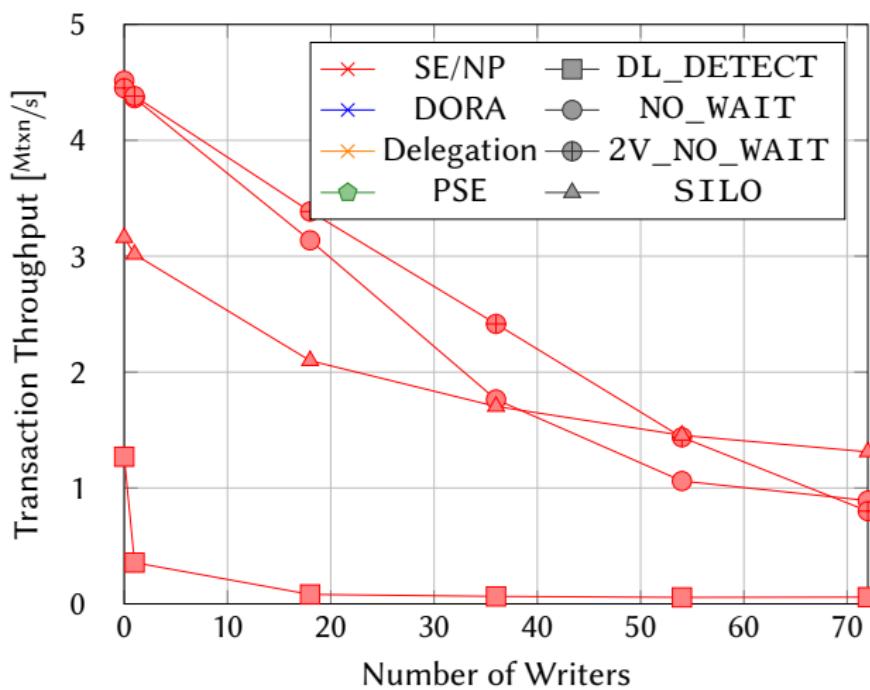
Mixed YCSB ($\Theta = 0.8$, 72 Threads)



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

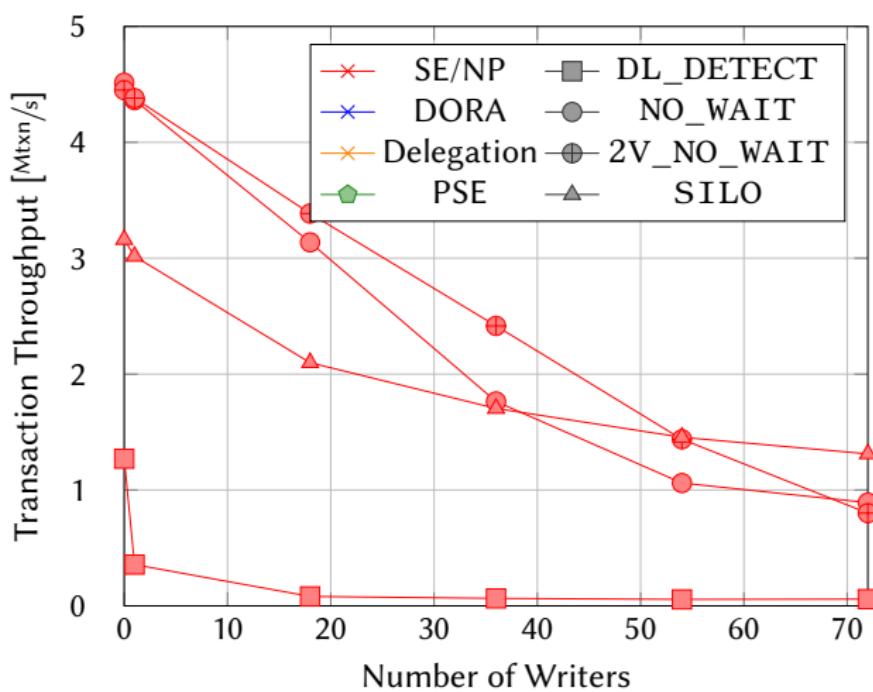
Mixed YCSB ($\Theta = 0.8$, 72 Threads)



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

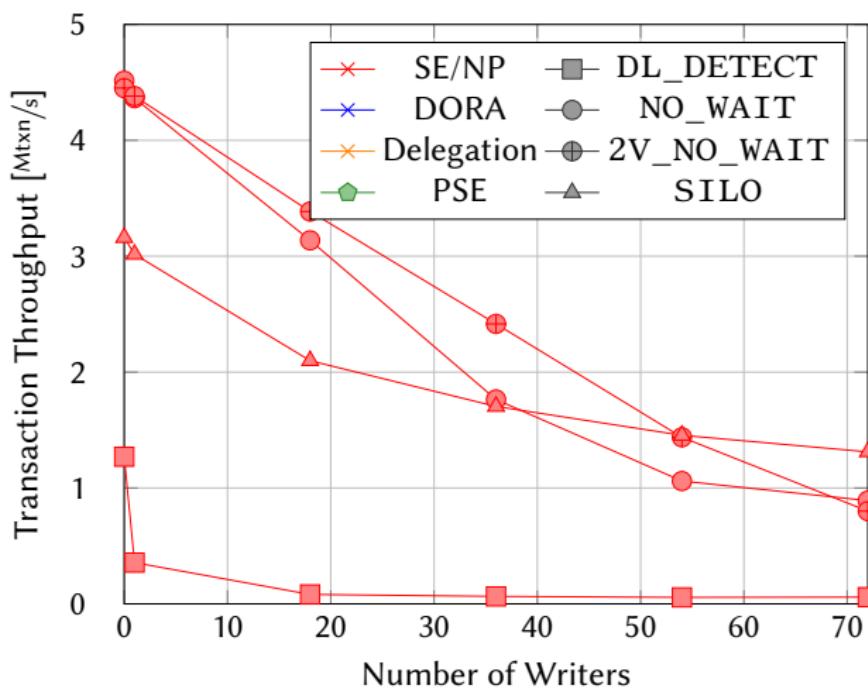
Mixed YCSB ($\Theta = 0.8$, 72 Threads)



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

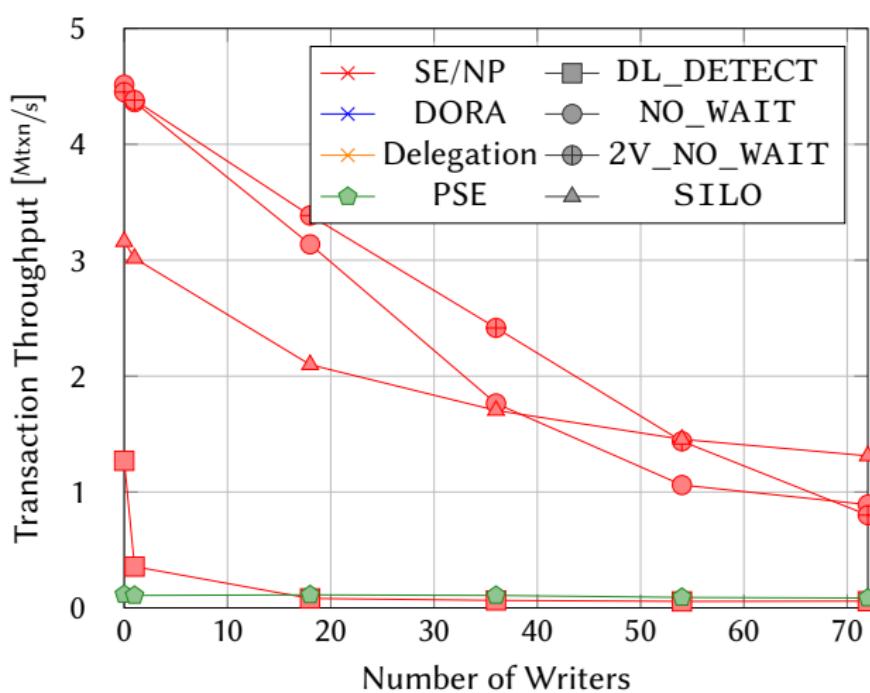
Mixed YCSB ($\Theta = 0.8$, 72 Threads)



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 to its optimism for many writers

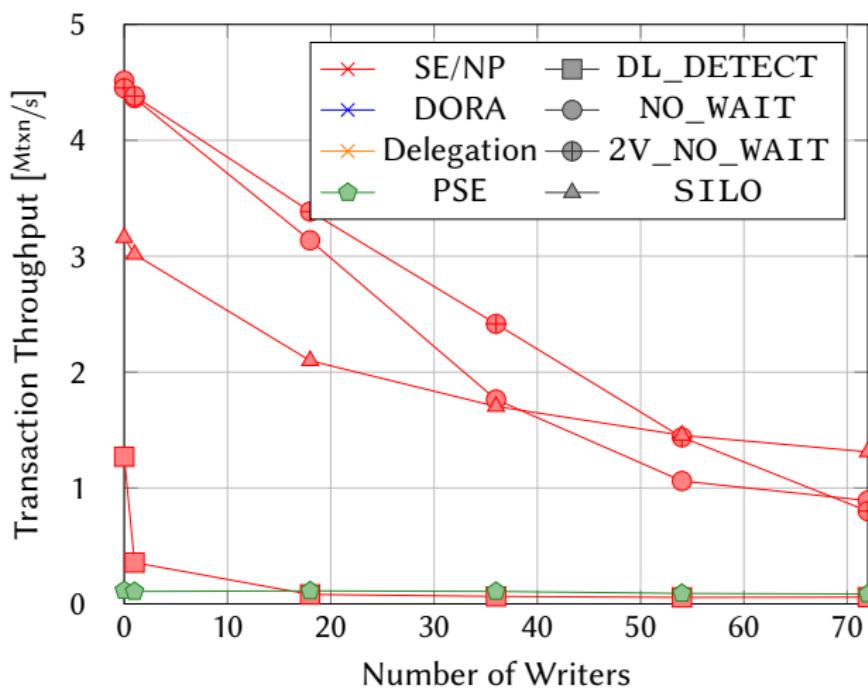
Mixed YCSB ($\Theta = 0.8$, 72 Threads)



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 to 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 to its optimism for many writers
- ◆ (and ✕/✖) doesn't scale well due to partition-unfriendly zipfian access distribution

Conclusion I

Conclusion I

- ▶ optimistic CC scales better than pessimistic CC for most workloads

Conclusion I

- ▶ optimistic CC scales better than pessimistic CC for most workloads
- ▶ optimistic CC suffers from large record sizes

Conclusion I

- ▶ optimistic CC scales better than pessimistic CC for most workloads
- ▶ optimistic CC suffers from large record sizes
- ▶ atomic operations scale better than latches

Conclusion I

- ▶ optimistic CC scales better than pessimistic CC for most workloads
- ▶ optimistic CC suffers from large record sizes
- ▶ atomic operations scale better than latches
- ▶ partitioning make latches scalable

Conclusion I

- ▶ optimistic CC scales better than pessimistic CC for most workloads
- ▶ optimistic CC suffers from large record sizes
- ▶ atomic operations scale better than latches
- ▶ partitioning make latches scalable
- ▶ 2PL doesn't scale for mixed workloads

Conclusion I

- ▶ optimistic CC scales better than pessimistic CC for most workloads
- ▶ optimistic CC suffers from large record sizes
- ▶ atomic operations scale better than latches
- ▶ partitioning make latches scalable
- ▶ 2PL doesn't scale for mixed workloads
- ▶ partitioning DB architectures perform bad under partition-unfriendly workloads

Conclusion I

- ▶ optimistic CC scales better than pessimistic CC for most workloads
- ▶ optimistic CC suffers from large record sizes
- ▶ atomic operations scale better than latches
- ▶ partitioning make latches scalable
- ▶ 2PL doesn't scale for mixed workloads
- ▶ partitioning DB architectures perform bad under partition-unfriendly workloads
- ▶ partitioning DB architectures perform bad under multi-sited transactions

Conclusion II

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

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 doesn't scale for read-intensive high contention workloads wth small hot sets

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 doesn't scale for read-intensive high contention workloads wth small hot sets
- None of the architectures or CC protocols outperform the others for any workload!

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 doesn't scale for read-intensive high contention workloads wth 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

Discussion of the Performance Evaluation

- ▶ read-only and update-only workload aren't appropriate to evaluate concurrency control algorithms

Discussion of the Performance Evaluation

- ▶ read-only and update-only workload aren't appropriate to evaluate concurrency control algorithms
- ▶ partition-unfriendly workloads aren't appropriate to evaluate database architectures that use partitioning

Discussion of the Performance Evaluation

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

Discussion of the Performance Evaluation

- ▶ read-only and update-only workload aren't appropriate to evaluate concurrency control algorithms
- ▶ partition-unfriendly workloads aren't appropriate to evaluate database architectures that use partitioning
- ▶ neither the microbenchmark nor YCSB are OLTP benchmarks
- The authors didn't 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).



Ippokratis Pandis et al. “Data-Oriented Transaction Execution”. Sept. 2010.



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.

References II



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?