Towards Concurrent Stateful Stream Processing on Multicore Processors

Shuhao Zhang

co-author: Yingjun Wu (AWS), Fengzhang (Renmin University of China), Bingsheng He

Xtra Computing Group

lead by Prof. Bingsheng He

NUS National University of Singapore

Data Stream Processing Systems

1. *BriskStream: Scaling Data Stream Processing on Shared-Memory Multicore Architectures, SIGMOD'19*
2. *StreamBox: Modern Stream Processing on a Multicore Machine, ATC'19*
3. *Grizzly: Efficient Stream Processing Through Adaptive Query Compilation, SIGMOD'20*

NUS National University of Singapore

Data Stream Processing Systems

- [Available] Fast stream processing on large-scale multicore architectures
  - E.g., BriskStream[1], StreamBox[2], Grizzly[3]
- [Inadequate] Support of *consistent stateful stream processing*

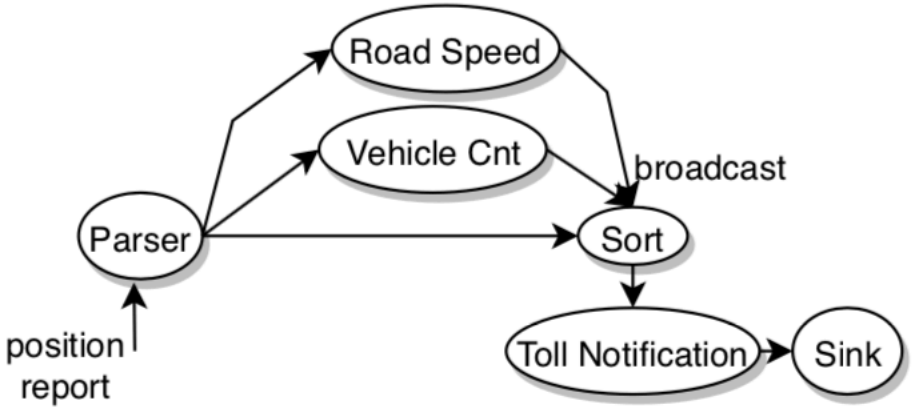
1. *BriskStream: Scaling Data Stream Processing on Shared-Memory Multicore Architectures, SIGMOD'19*
2. *StreamBox: Modern Stream Processing on a Multicore Machine, ATC'19*
3. *Grizzly: Efficient Stream Processing Through Adaptive Query Compilation, SIGMOD'20*

NUS National University of Singapore

Outline

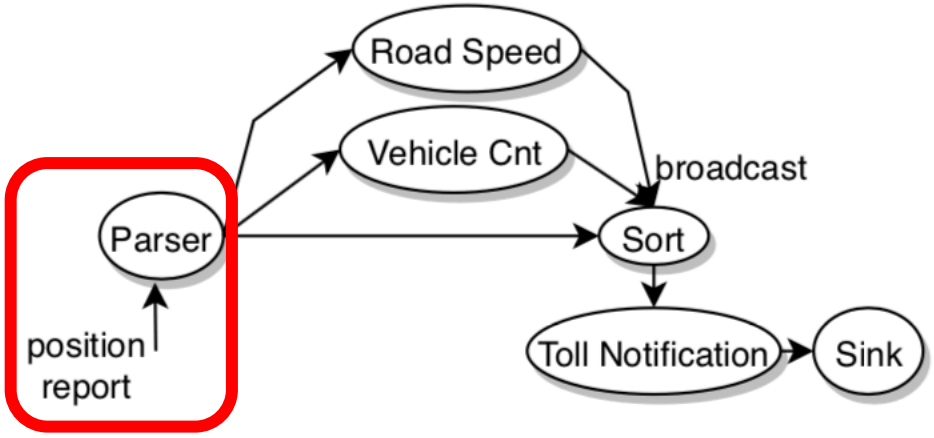
- **Motivation**
- **State-of-the-art**
- **Our Approach**
- **Experimental Results**

Motivation Example



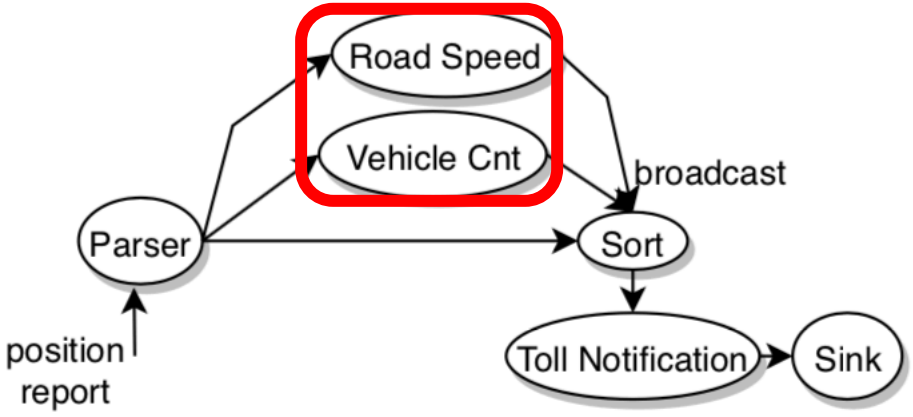
Toll Processing: computes toll based on the road congestion status

Motivation Example



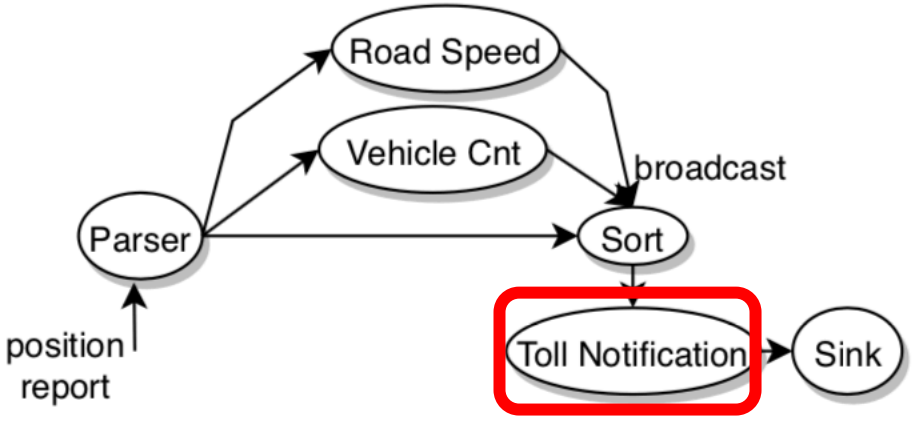
Toll Processing: computes toll based on the road congestion status

Motivation Example



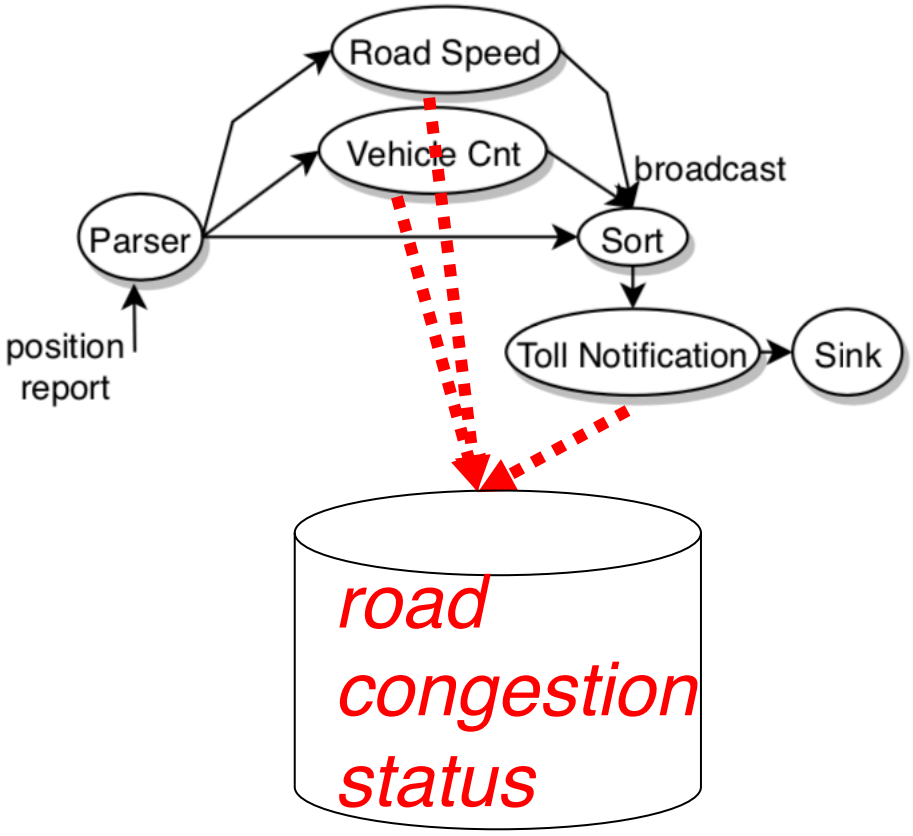
Toll Processing: computes toll based on the road congestion status

Motivation Example



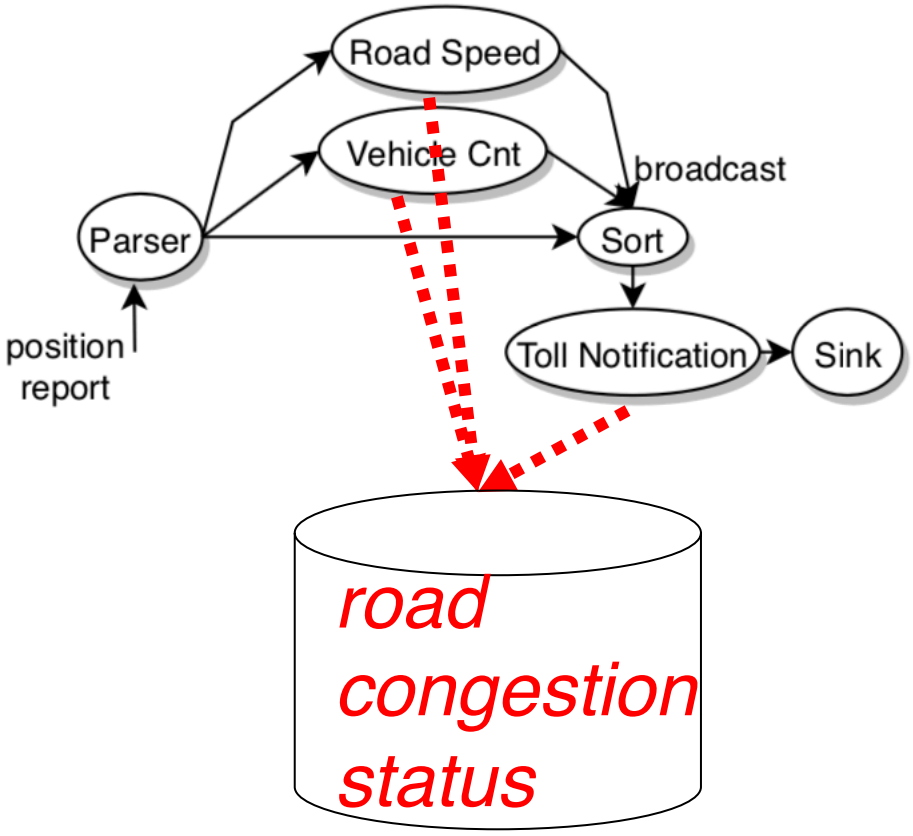
Toll Processing: computes toll based on the road congestion status

Motivation Example



- *Road congestion status is shared among different streaming operators*
- *Its consistency needs to be preserved*

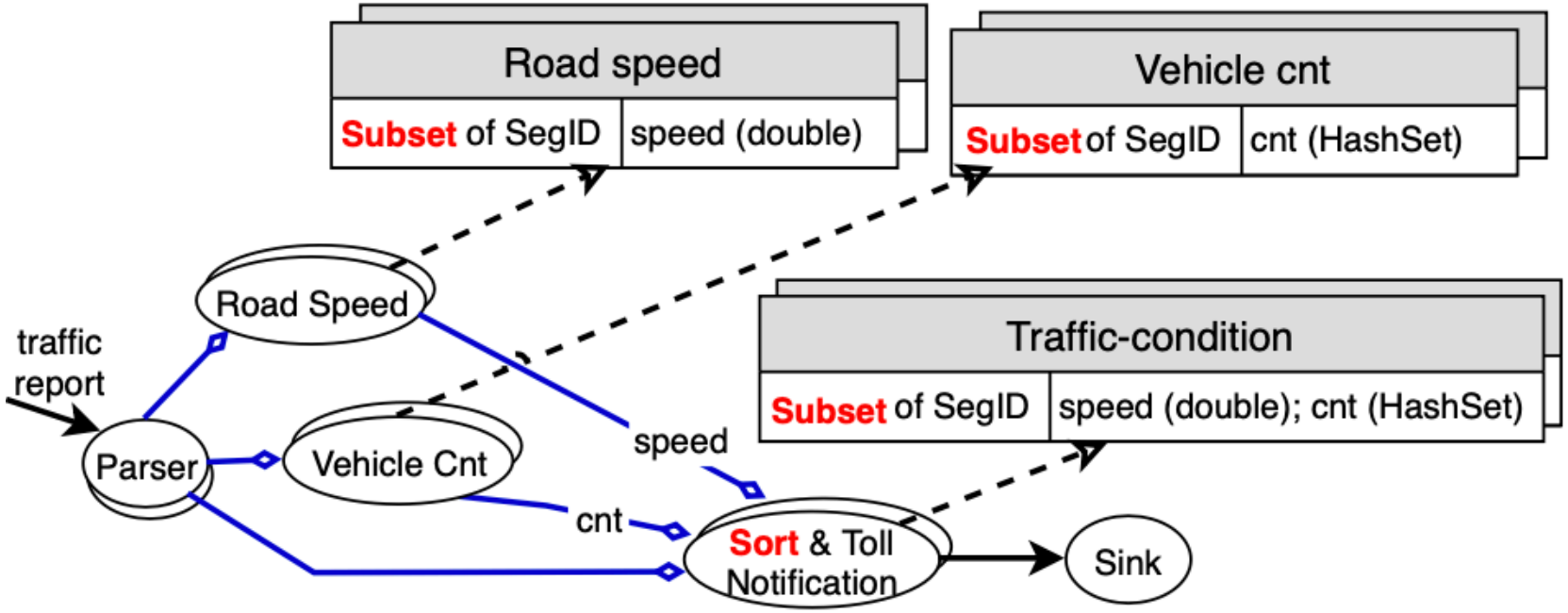
Motivation Example



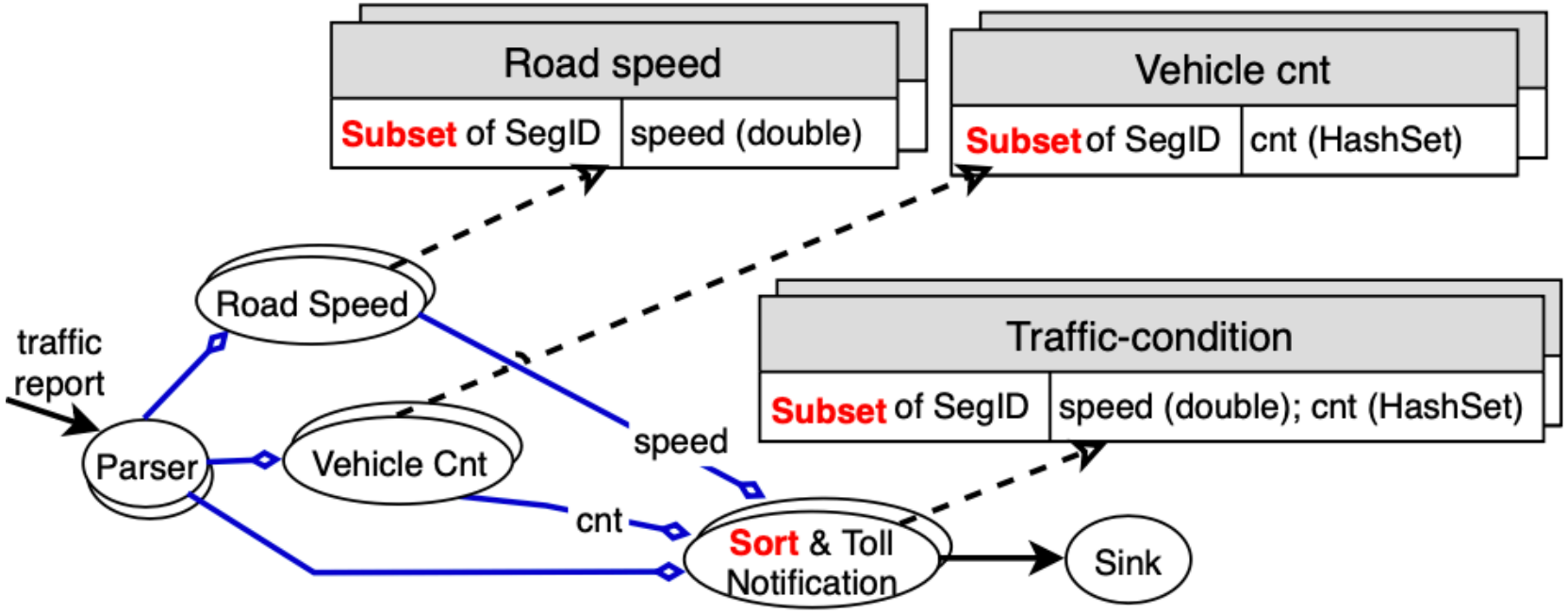
- *Road congestion status is shared among different streaming operators*
- *Its consistency needs to be preserved*

-- consistent stateful stream processing

The Current Common System Design



The Current Common System Design

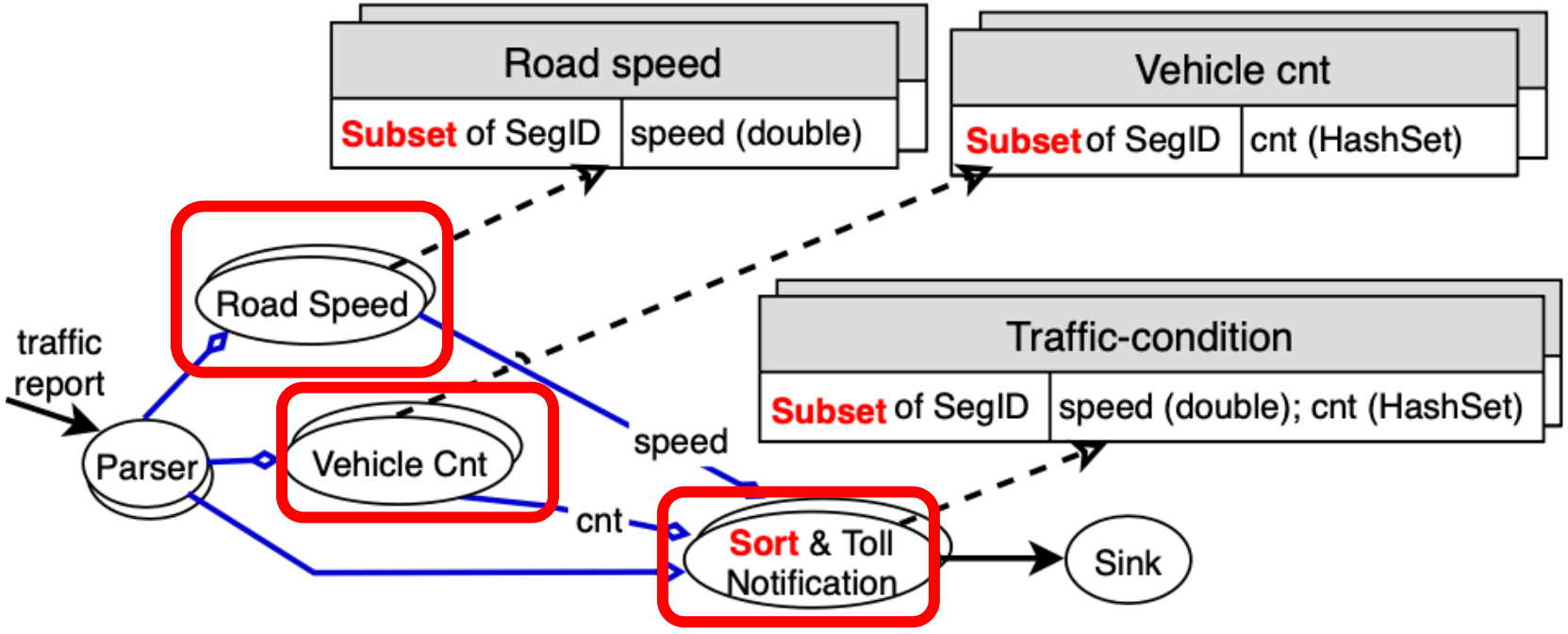


Common designs:

a) Pipelined processing with message passing

b) On-demand data parallelism

The Current Common System Design



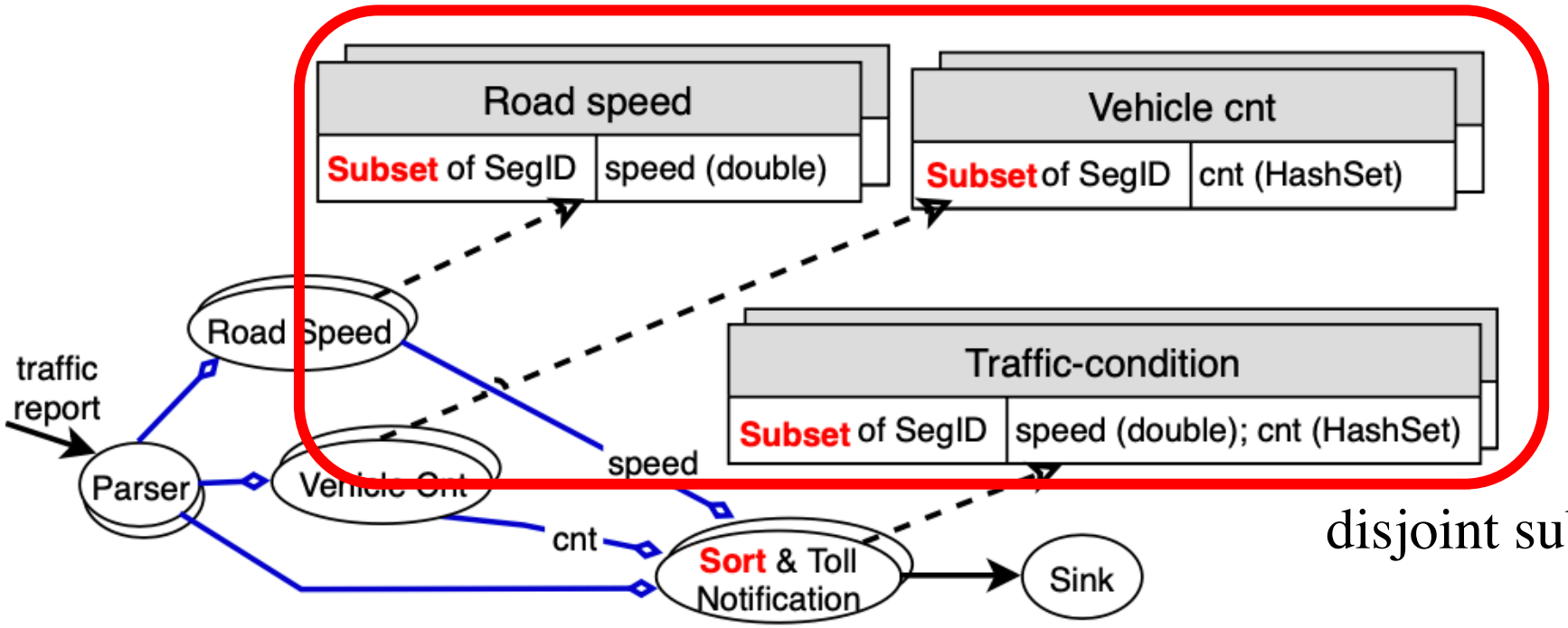
operator parallelism

Common designs:

a) Pipelined processing with message passing

b) On-demand data parallelism

The Current Common System Design



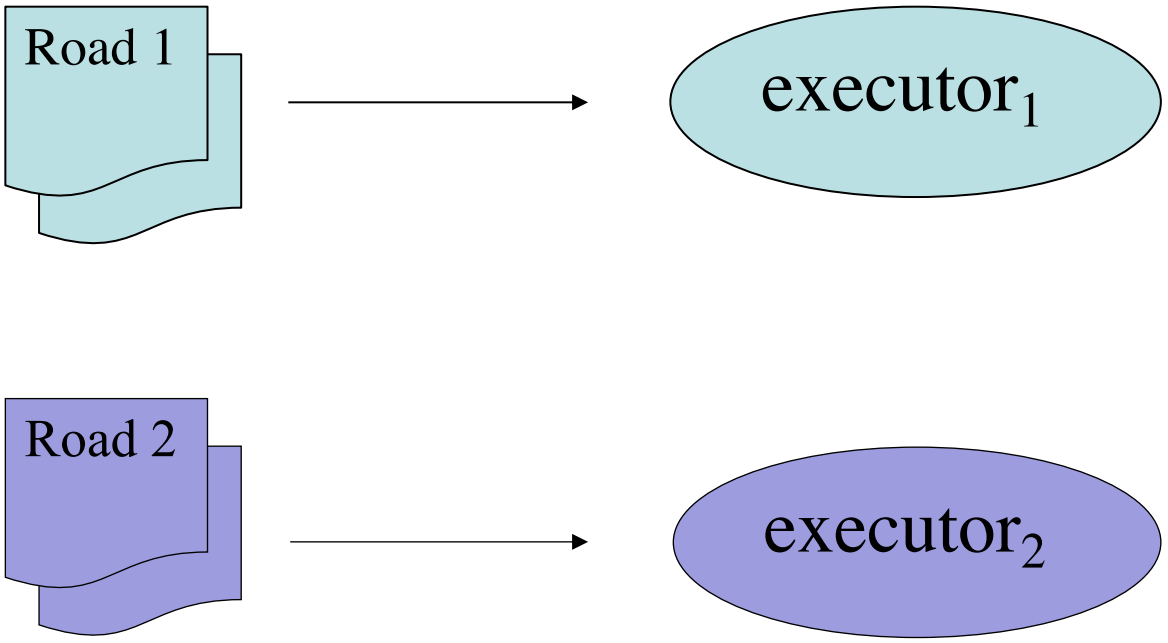
disjoint subset

Common designs:

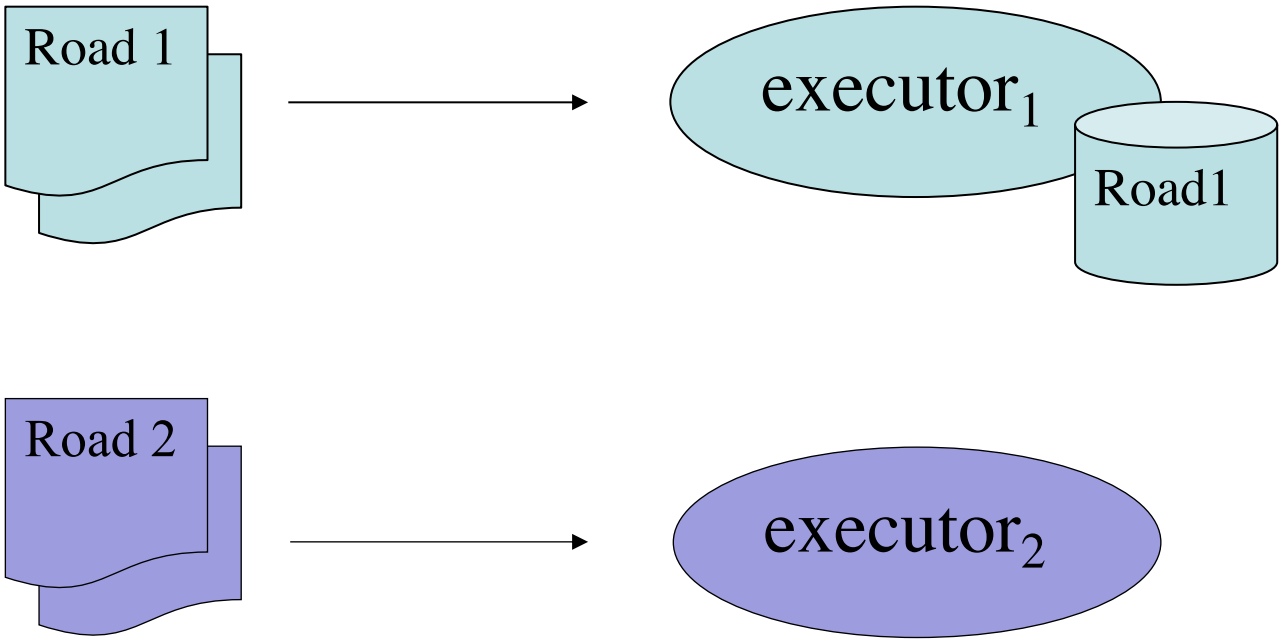
a) Pipelined processing with message passing

b) On-demand data parallelism

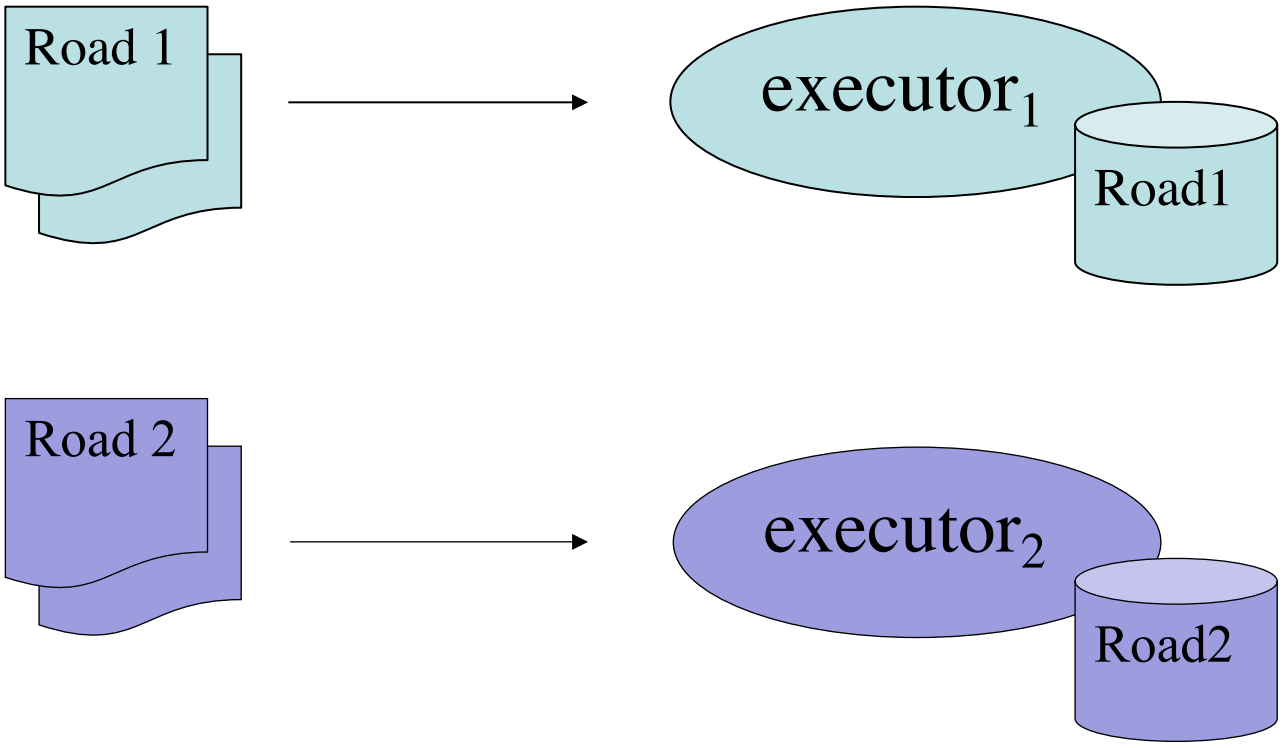
Key-based Stream Partition



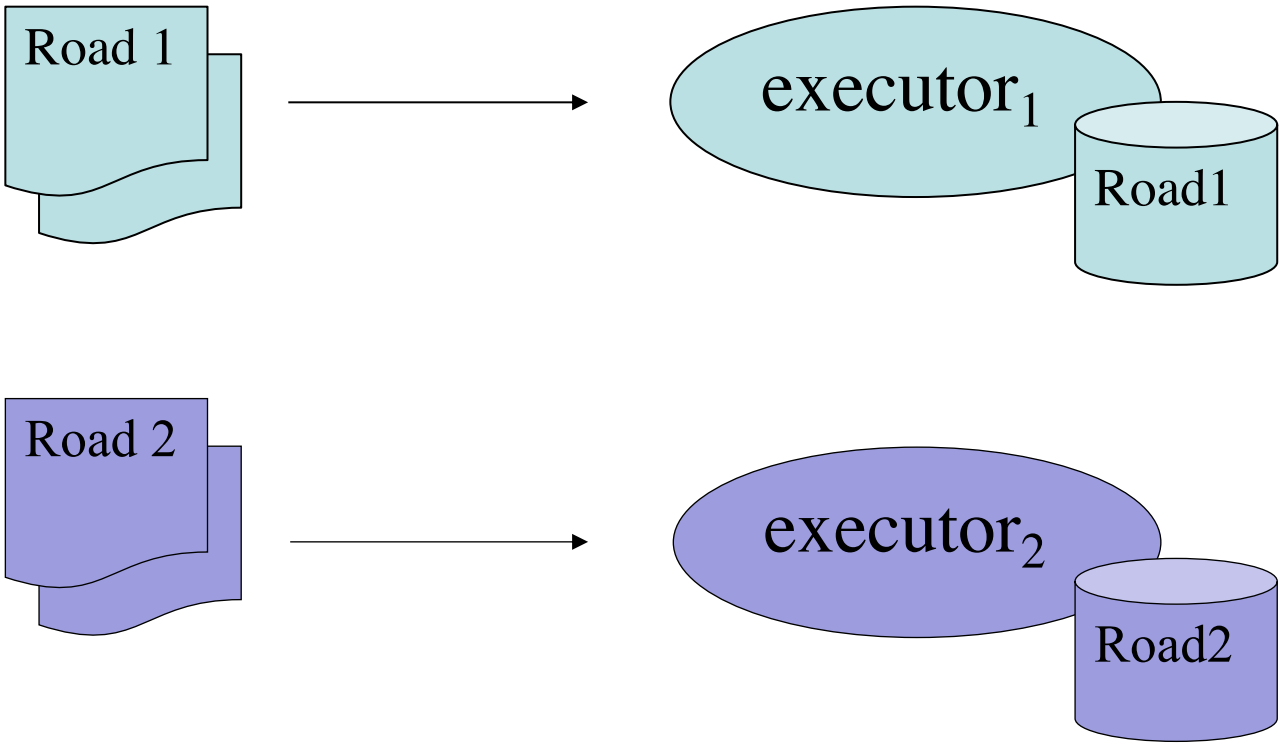
Key-based Stream Partition



Key-based Stream Partition

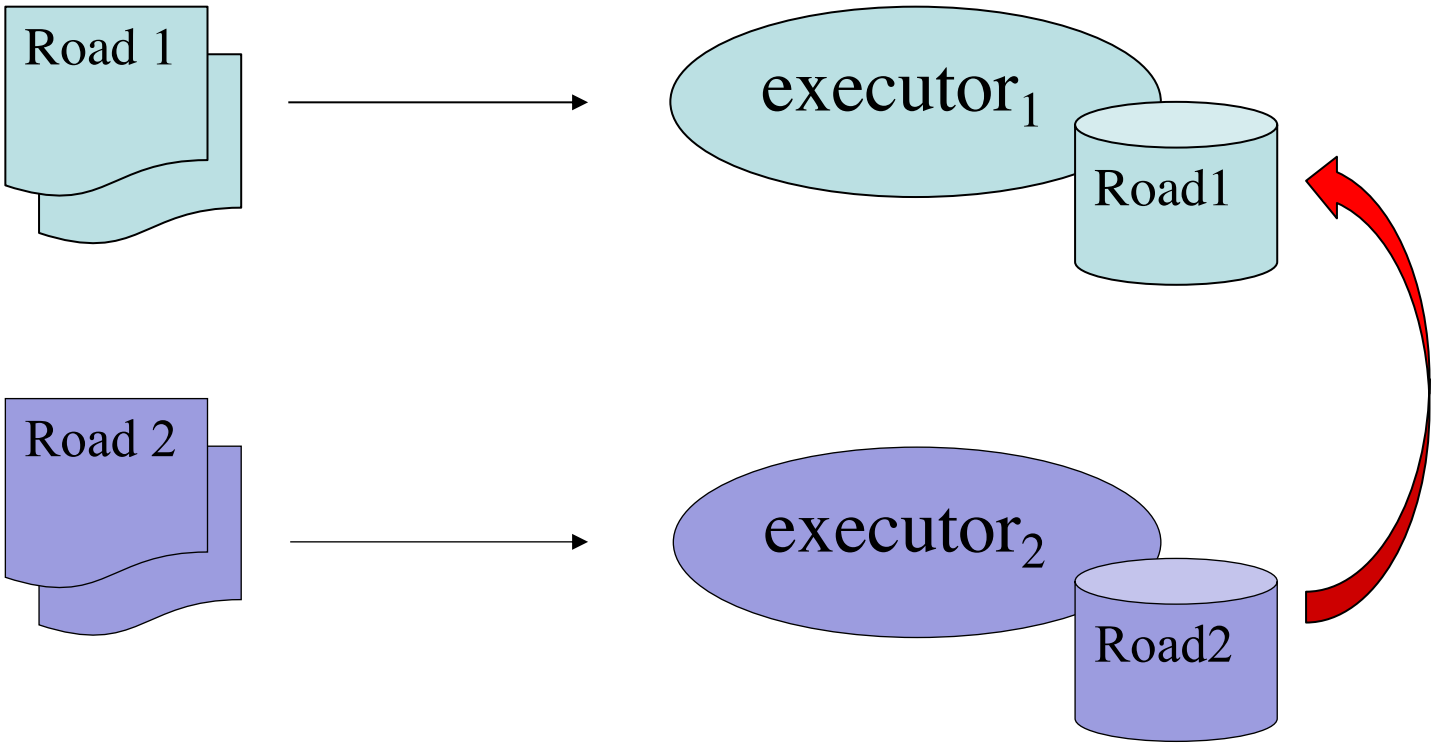


Key-based Stream Partition



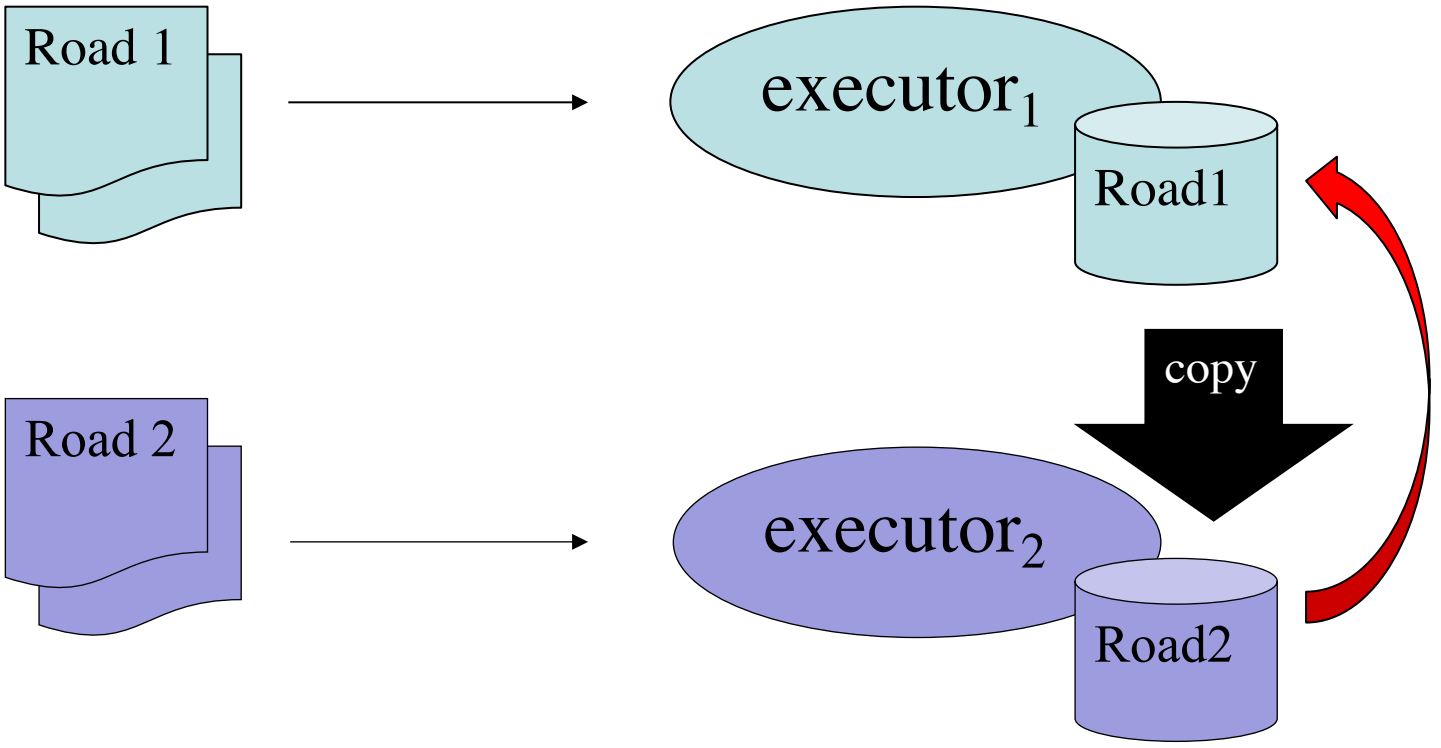
No Conflict!☺

Key-based Stream Partition



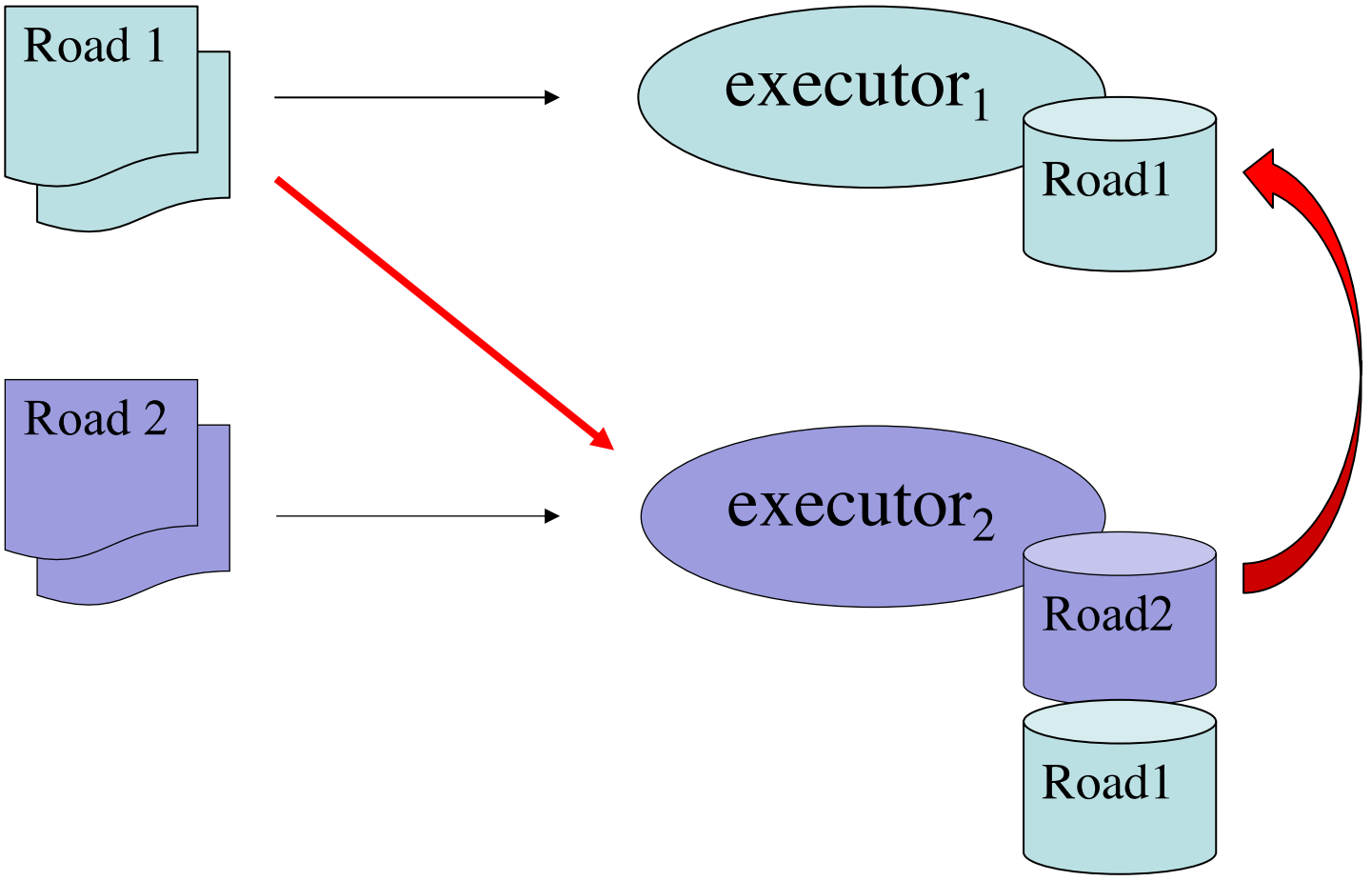
Pay extra penalty if road1 is empty!

Key-based Stream Partition



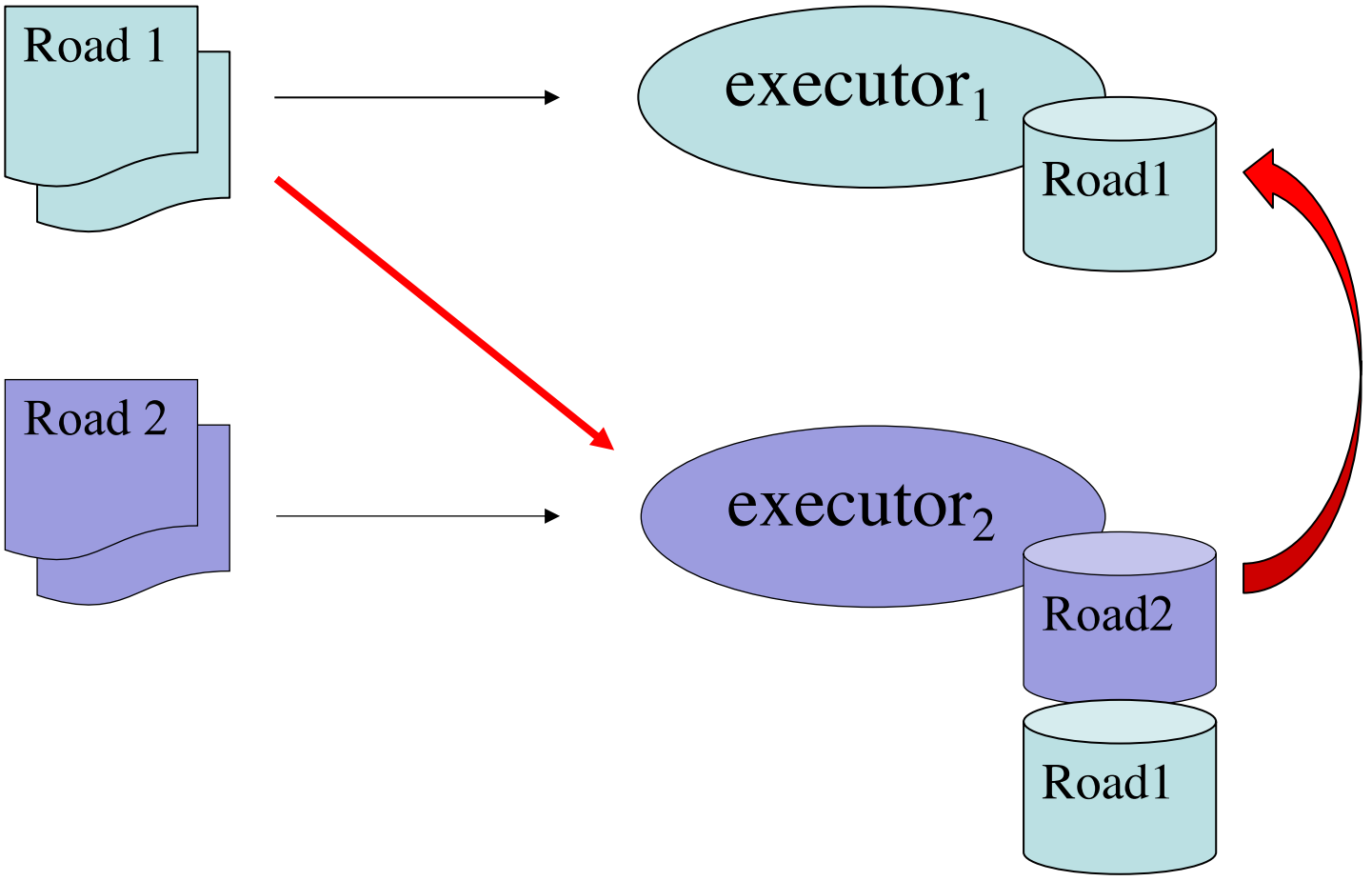
Pay extra penalty if road1 is empty!

Key-based Stream Partition



Pay extra penalty if road1 is empty!

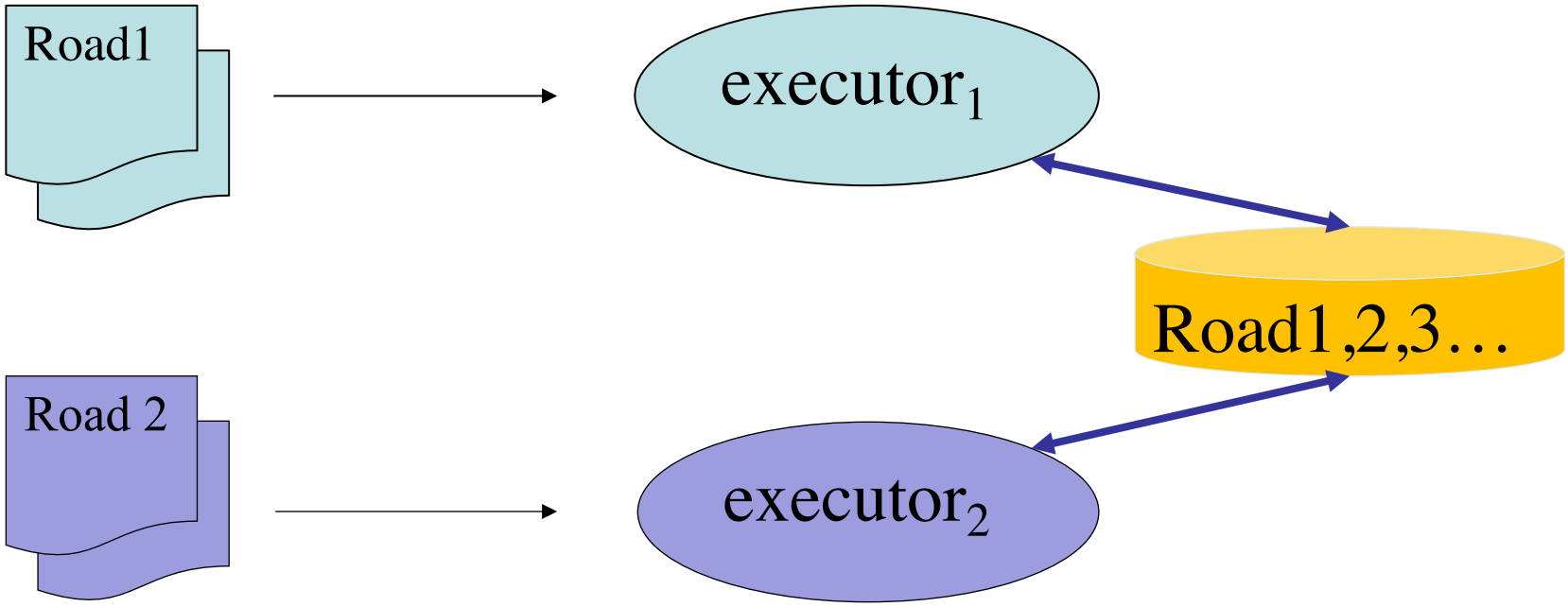
Key-based Stream Partition



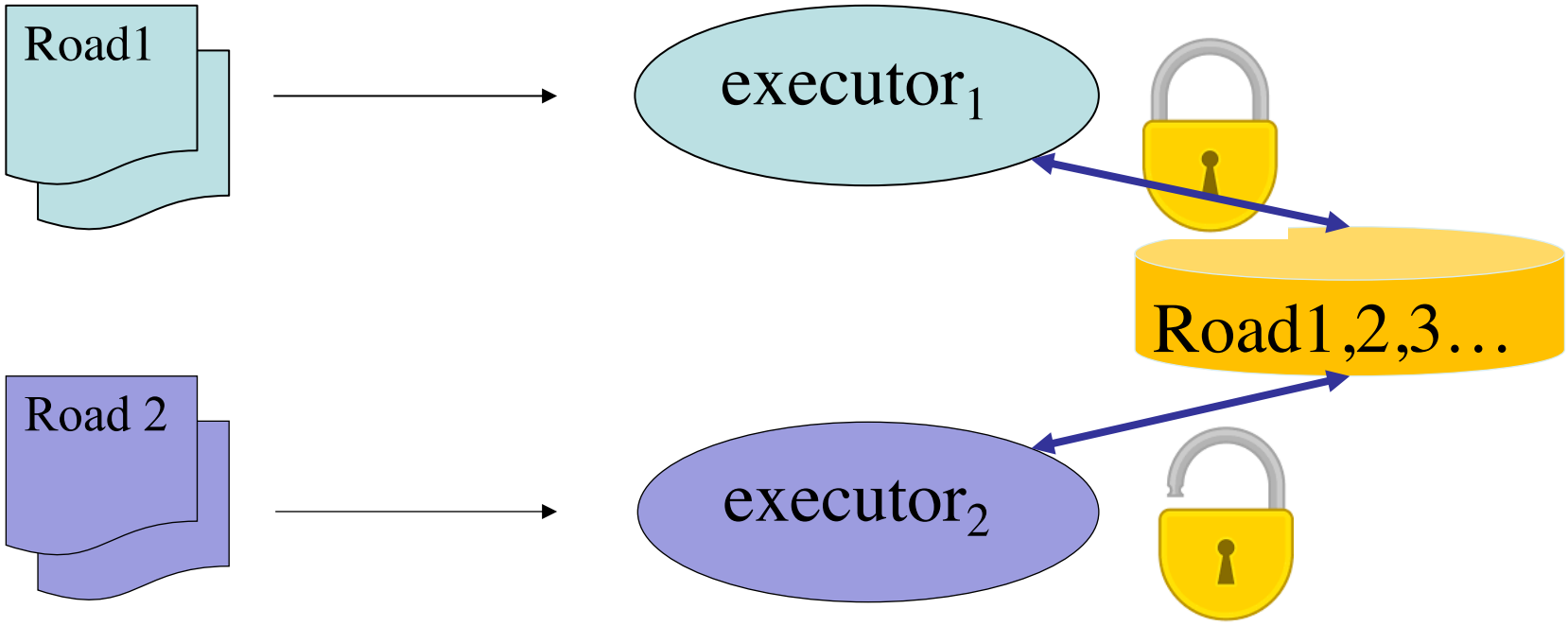
Pay extra penalty if road1 is empty!

Can not efficiently handle general case

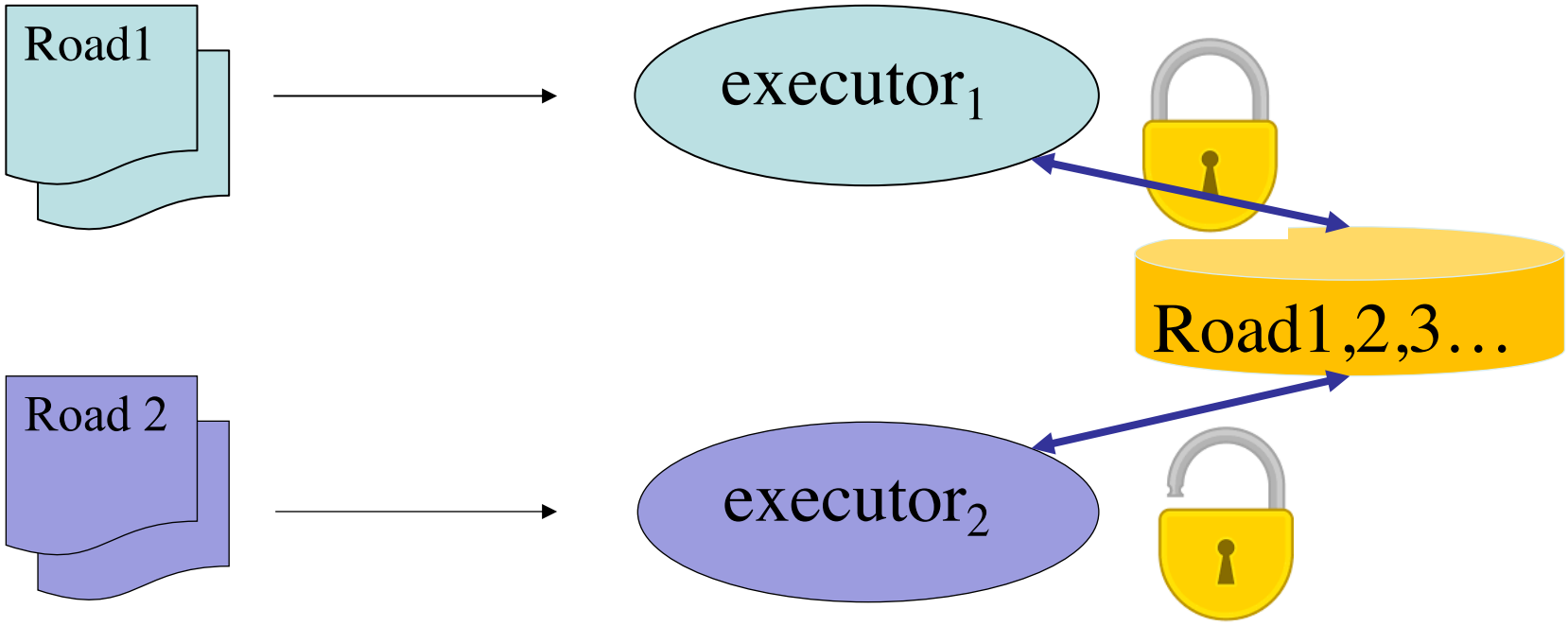
Lock-based State Sharing



Lock-based State Sharing



Lock-based State Sharing



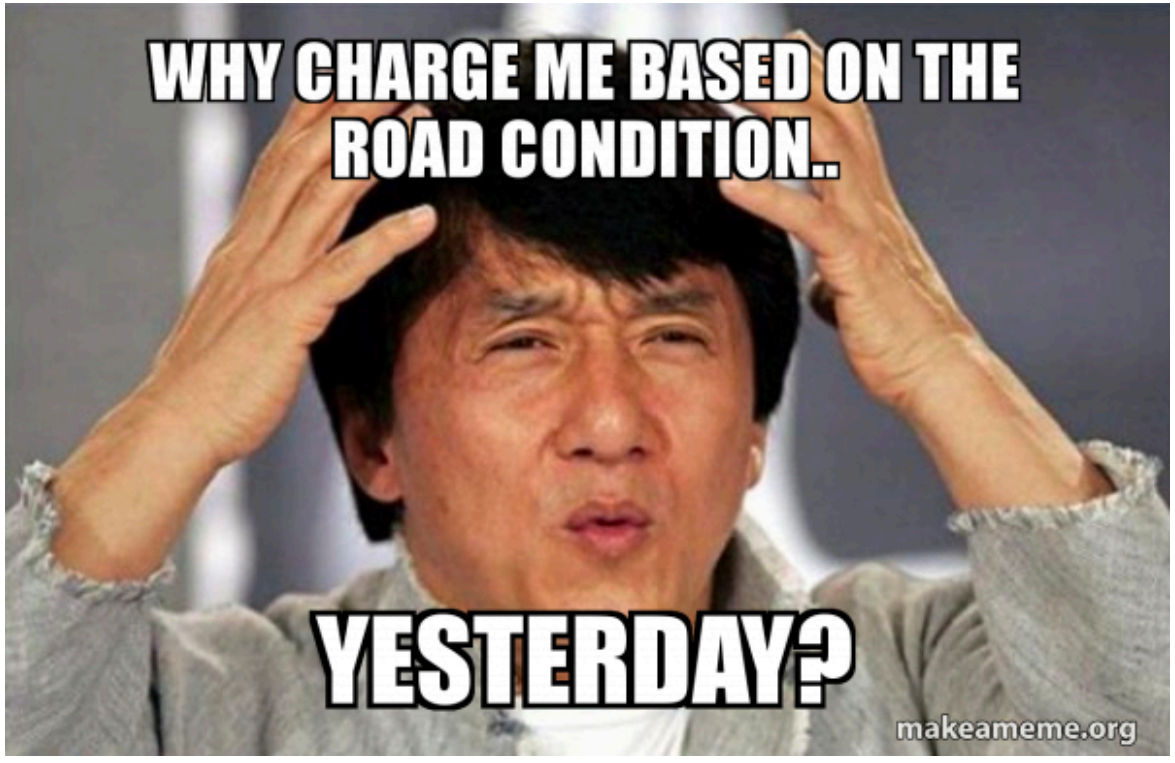
Lead to poor performance

Access Ordering

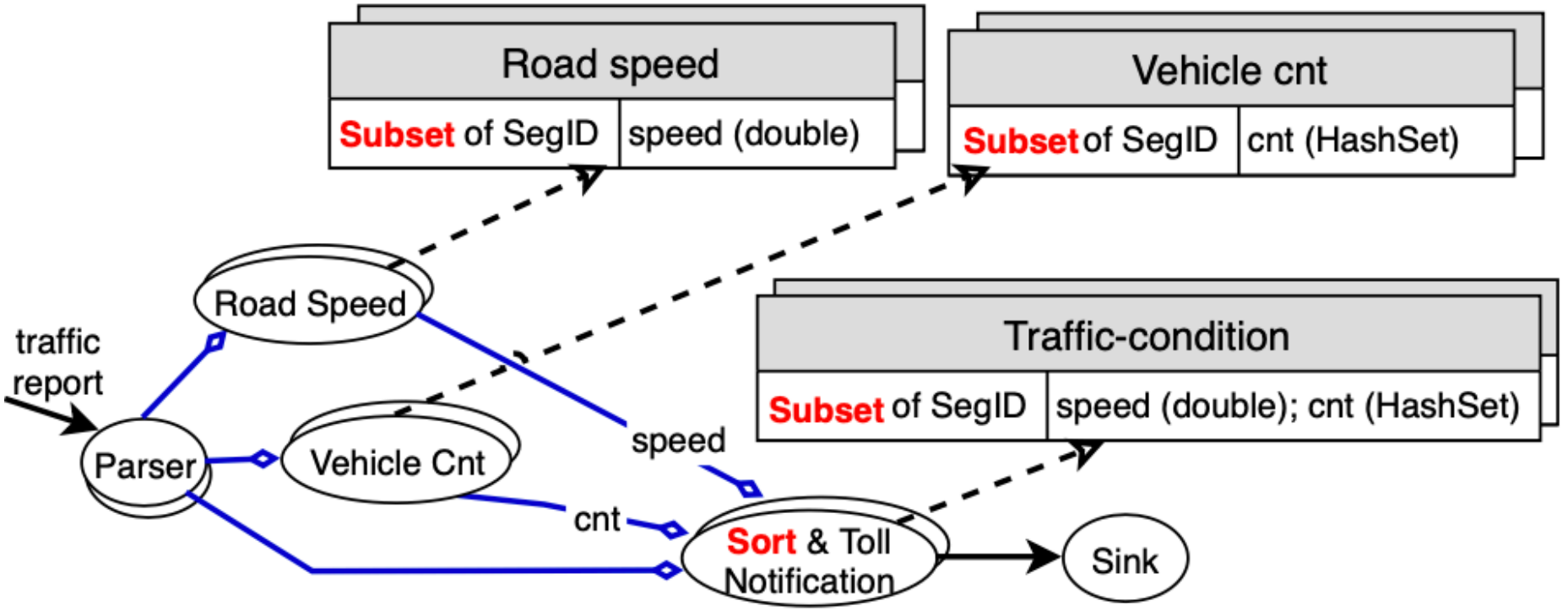
- **A toll should based on exactly current road condition, otherwise..**

Access Ordering

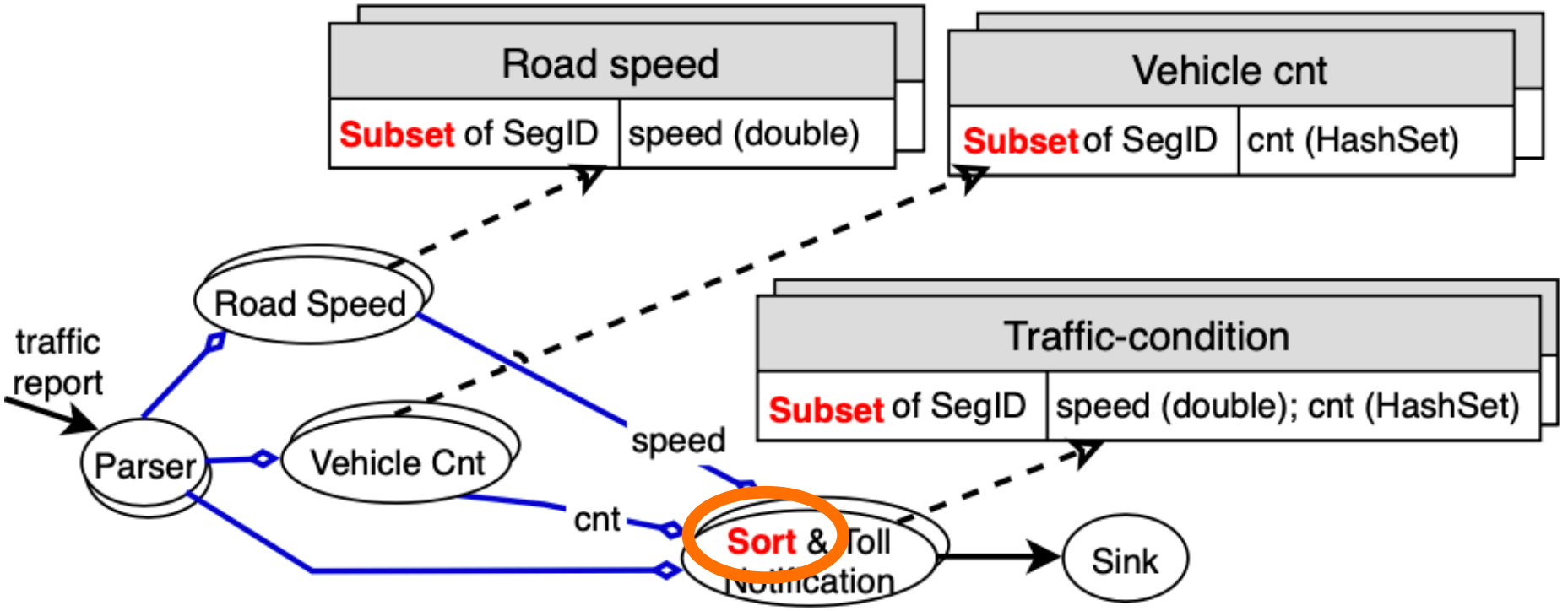
- **A toll should based on exactly current road condition, otherwise..**



Access Ordering



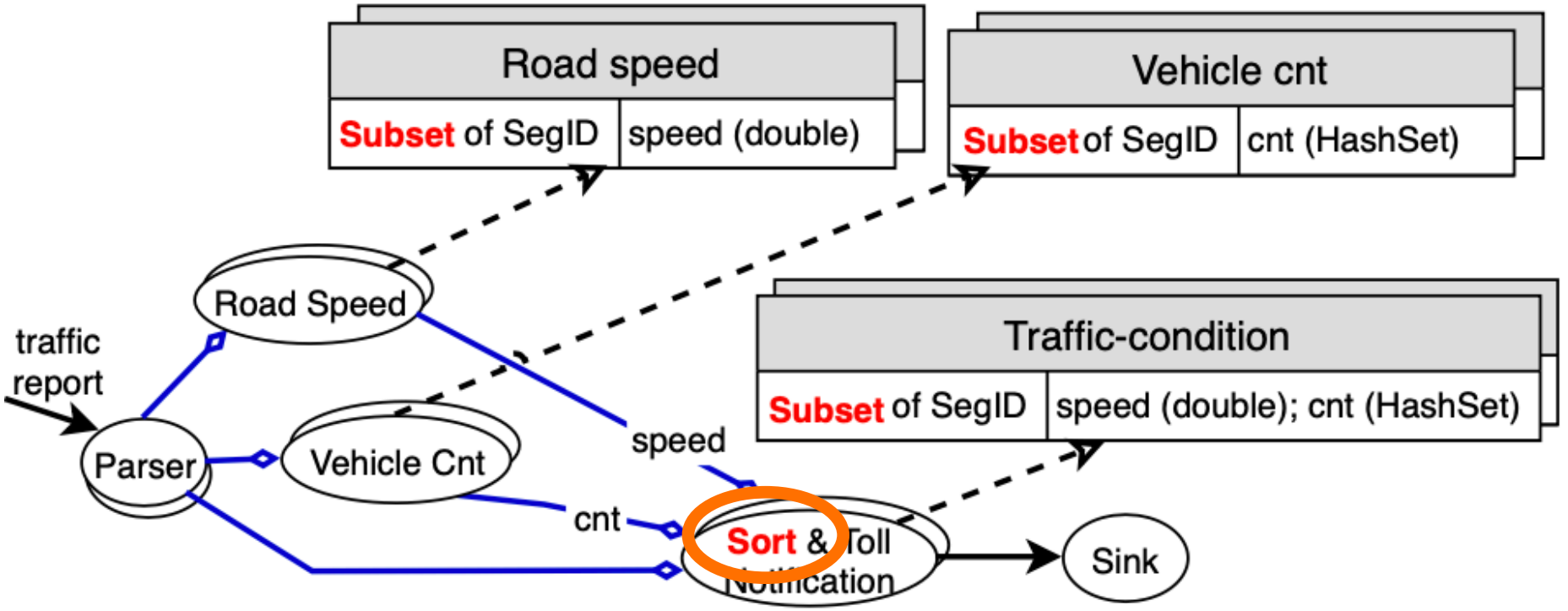
Access Ordering



Common workaround:

- ***Buffer and sort***

Access Ordering



Common workaround:

- ***Buffer and sort***

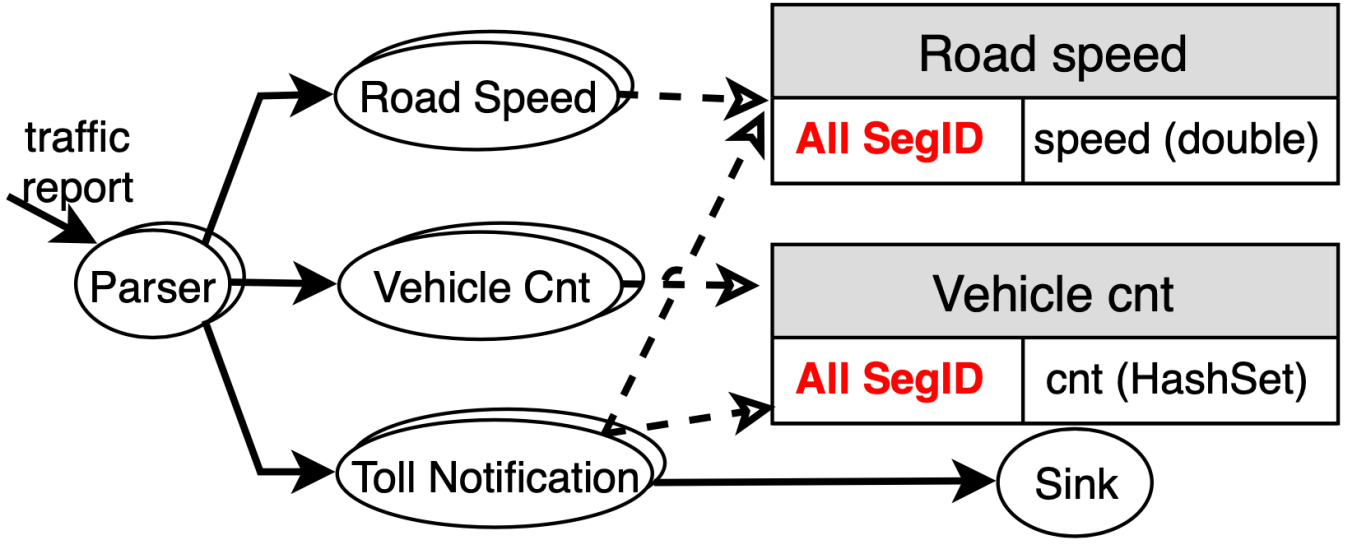
Lead to poor performance

Outline

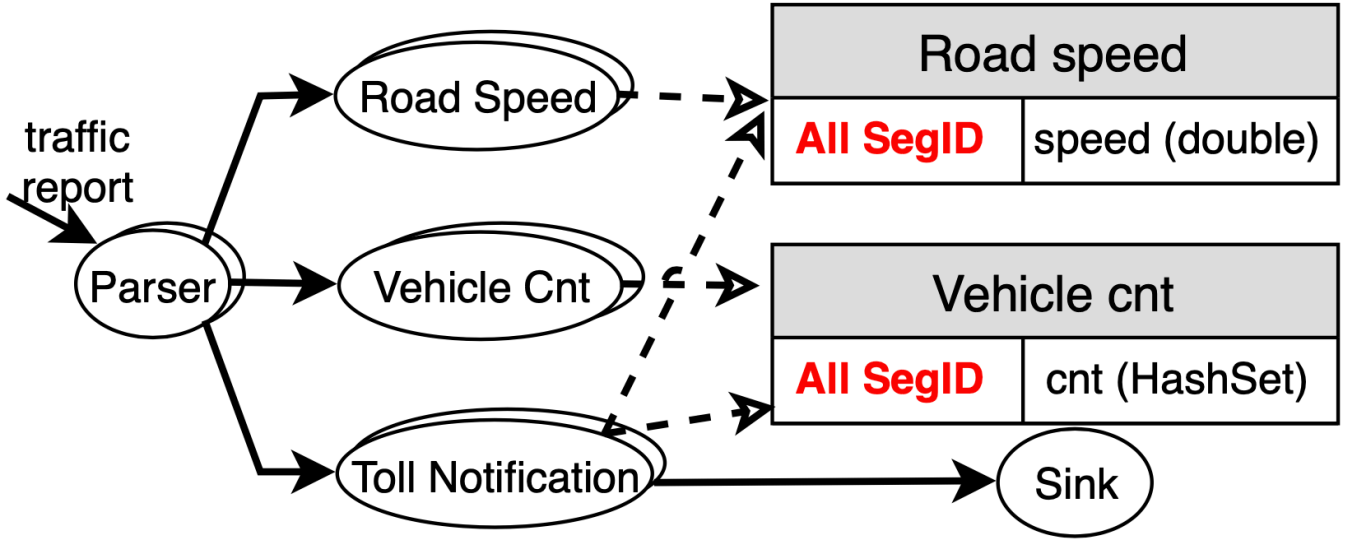
- **Motivation**
- **State-of-the-art**
- **Our Approach**
- **Experimental Results**

NUS National University of Singapore

Consistent Stateful Stream Processing



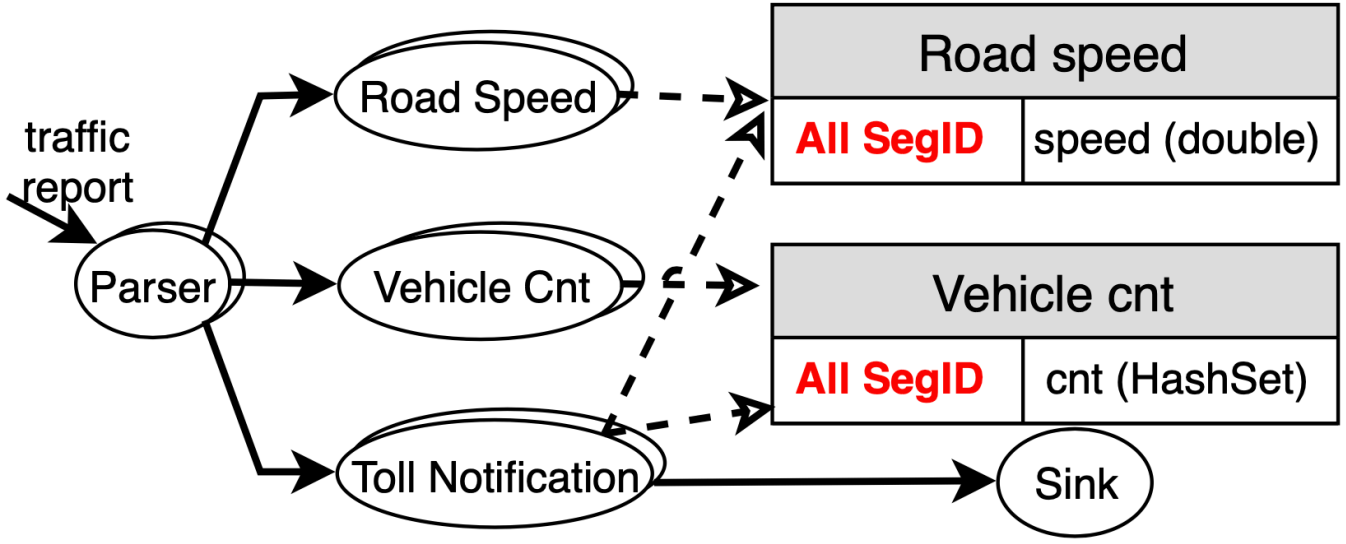
Consistent Stateful Stream Processing



Employing transactional schematics.

- State transaction (txn_{ts}):
  1. txn_{t1} {*Update Road1 Cnt, update Roadx…*} @ 10:00;
  2. txn_{t2} {Read Road1 Cnt,…} @ 10:05
- Correct schedule: $txn_{t1} \prec txn_{t2} \prec \ldots txn_{tn}$

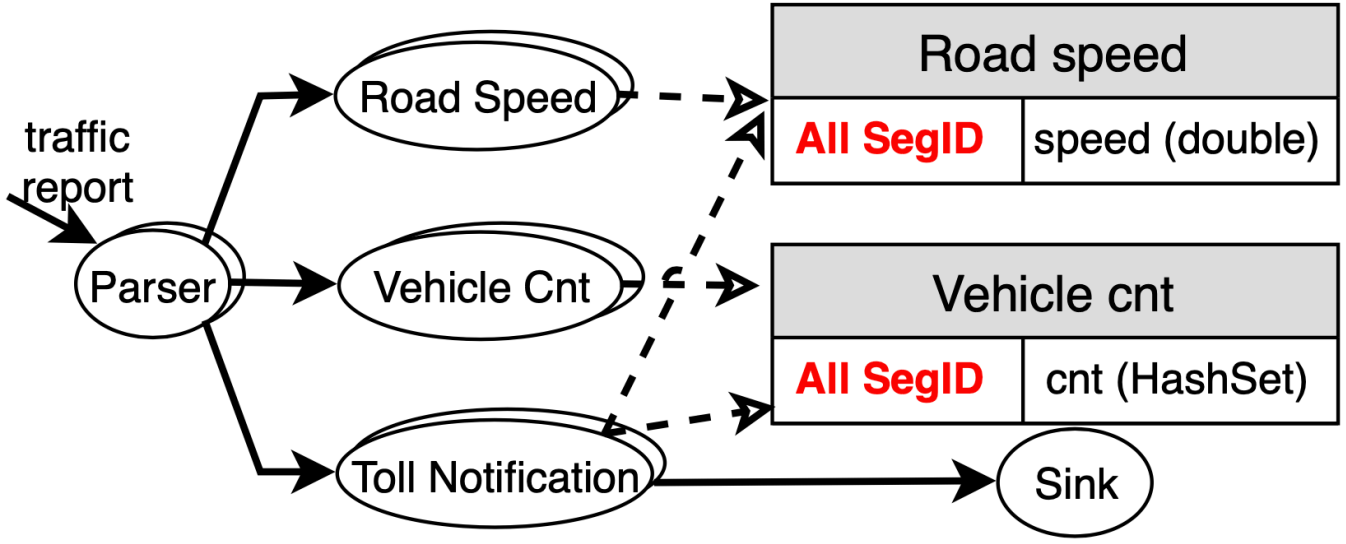
Consistent Stateful Stream Processing



Employing transactional schematics.

- State transaction (txn_{ts}):
  1. txn_{t1} {*Update Road1 Cnt, update Roadx…*} @ 10:00;
  2. txn_{t2} {Read Road1 Cnt,…} @ 10:05
- Correct schedule: $txn_{t1} \prec txn_{t2} \prec \ldots txn_{tn}$

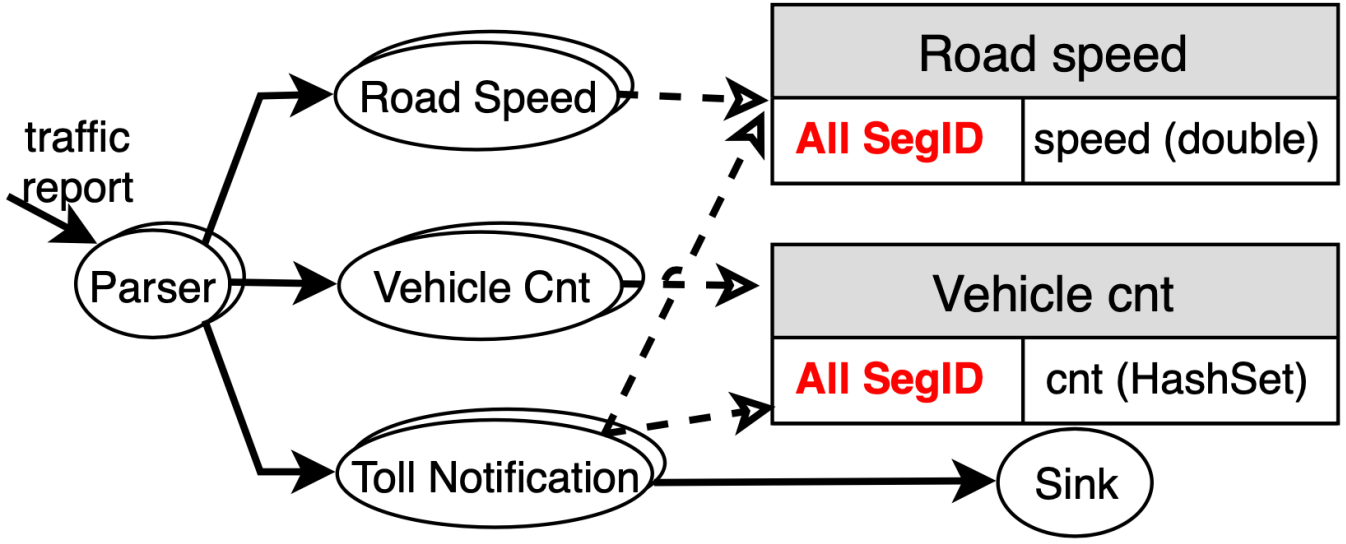
Consistent Stateful Stream Processing



Employing transactional schematics.

- State transaction (txn_{ts}):
  1. txn_{t1} {*Update Road1 Cnt, update Roadx…*} @ 10:00;
  2. txn_{t2} {Read Road1 Cnt,…} @ 10:05
- Correct schedule: $txn_{t1} \prec txn_{t2} \prec \ldots txn_{tn}$

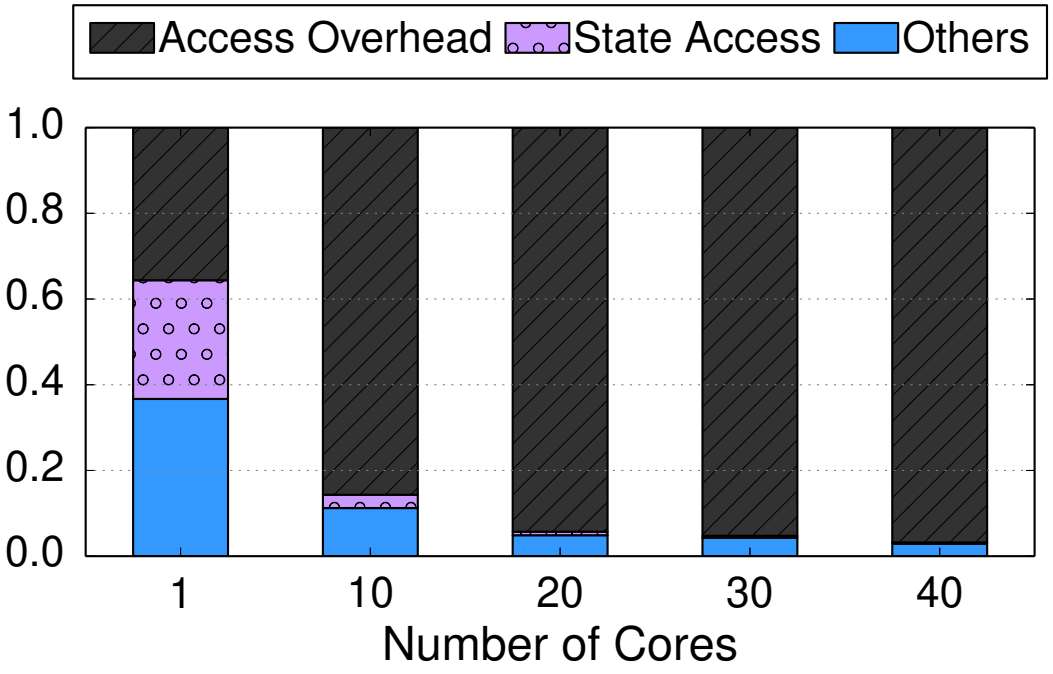
Consistent Stateful Stream Processing



Employing transactional schematics.

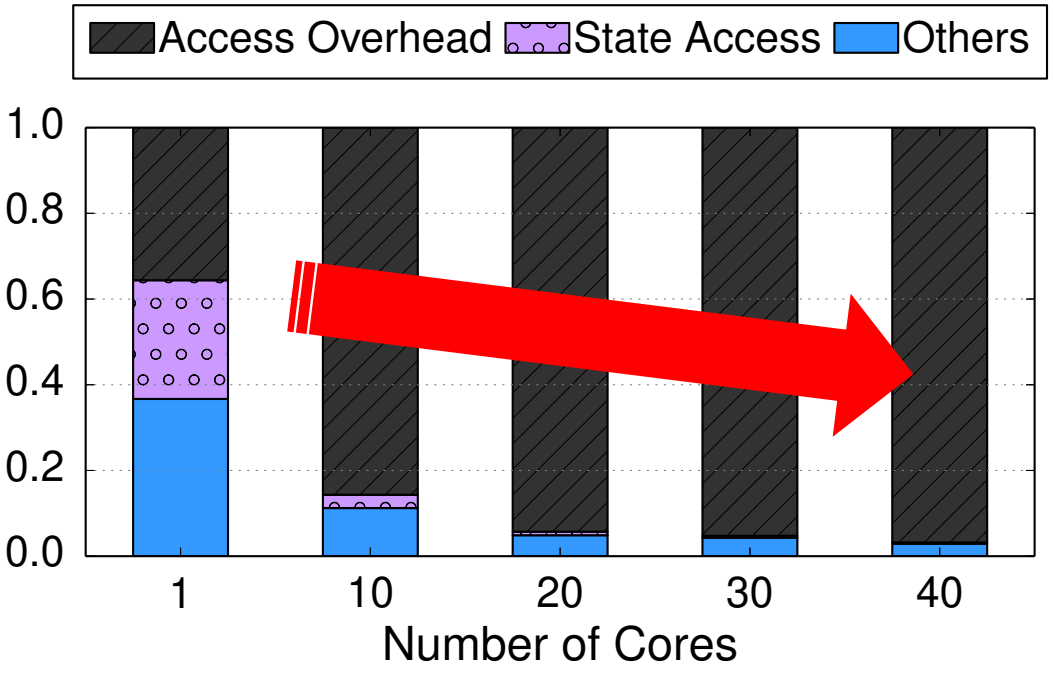
- State transaction (txn_{ts}):
  1. txn_{t1} {*Update Road1 Cnt, update Roadx…*} @ 10:00;
  2. txn_{t2} {Read Road1 Cnt,…} @ 10:05
- Correct schedule: $txn_{t1} \prec txn_{t2} \prec \ldots txn_{tn}$

Existing Solutions Revisited



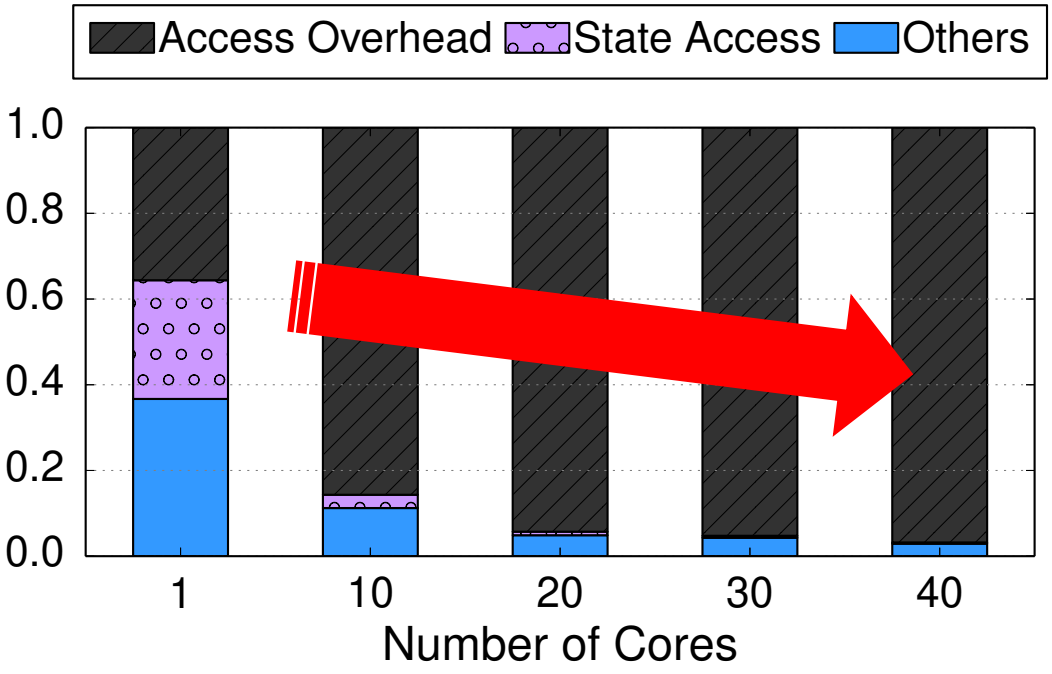
[1] S-Store: Streaming Meets Transaction Processing, VLDB'15

Existing Solutions Revisited



[1] S-Store: Streaming Meets Transaction Processing, VLDB'15

Existing Solutions Revisited

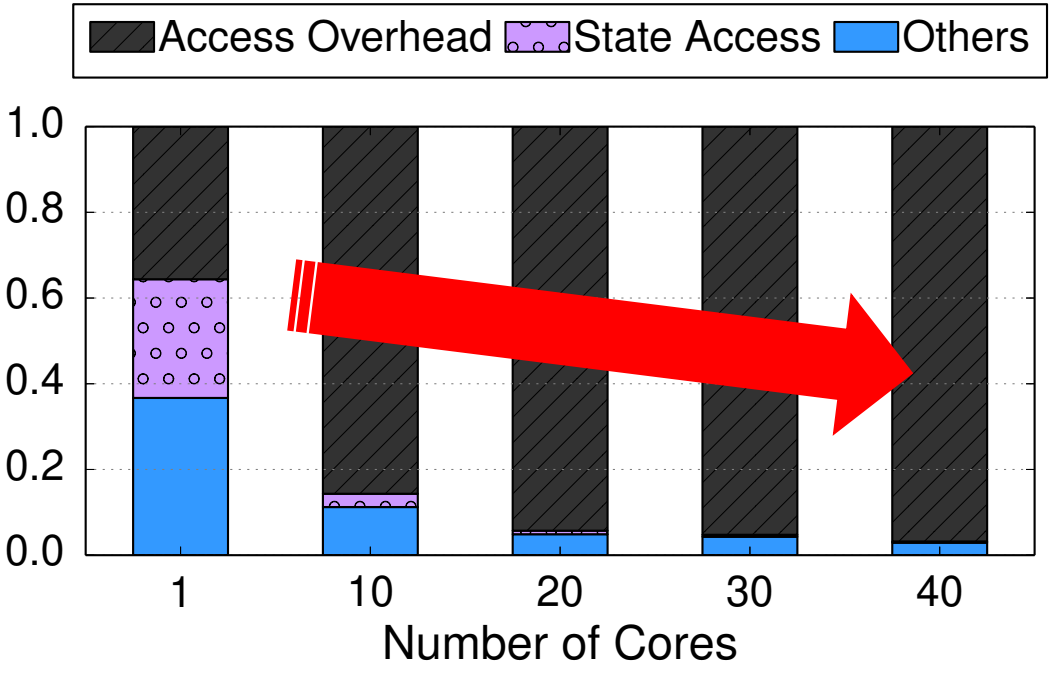


What if we use existing design[1]?

- Sever lock contention

[1] S-Store: Streaming Meets Transaction Processing, VLDB'15

Existing Solutions Revisited



What if we use existing design[1]?

- Sever lock contention

A new solution for scaling concurrent state access is needed!

[1] S-Store: Streaming Meets Transaction Processing, VLDB'15

NUS National University of Singapore

Outline

- **Motivation**
- **State-of-the-art**
- **Our Approach:**
  - TStream (An extension of BriskStream[2])
- **Experimental Results**

[2] BriskStream: Scaling Data Stream Processing on Shared-Memory Multicore Architectures, SIGMOD'19

Design Overview

Design Overview

- **Design 1) Dual-Mode Scheduling**
  - Exposing Parallelism
- **Design 2) Dynamic Restructuring Execution**
  - Exploiting Parallelism.

NUS National University of Singapore

Design Overview

- **Design 1) Dual-Mode Scheduling**
  - Exposing Parallelism
- **Design 2) Dynamic Restructuring Execution**
  - Exploiting Parallelism.

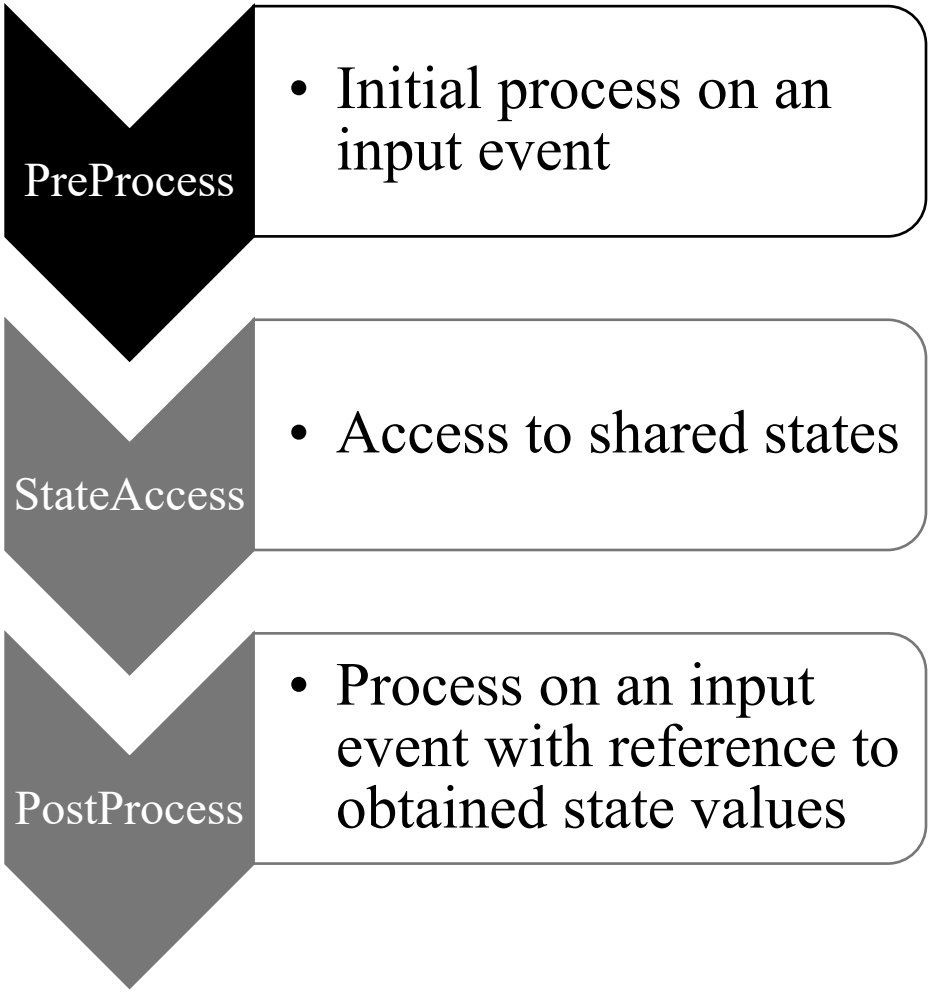
NUS National University of Singapore

Design Overview

- **Design 1) Dual-Mode Scheduling**
  - Exposing Parallelism
- **Design 2) Dynamic Restructuring Execution**
  - Exploiting Parallelism.

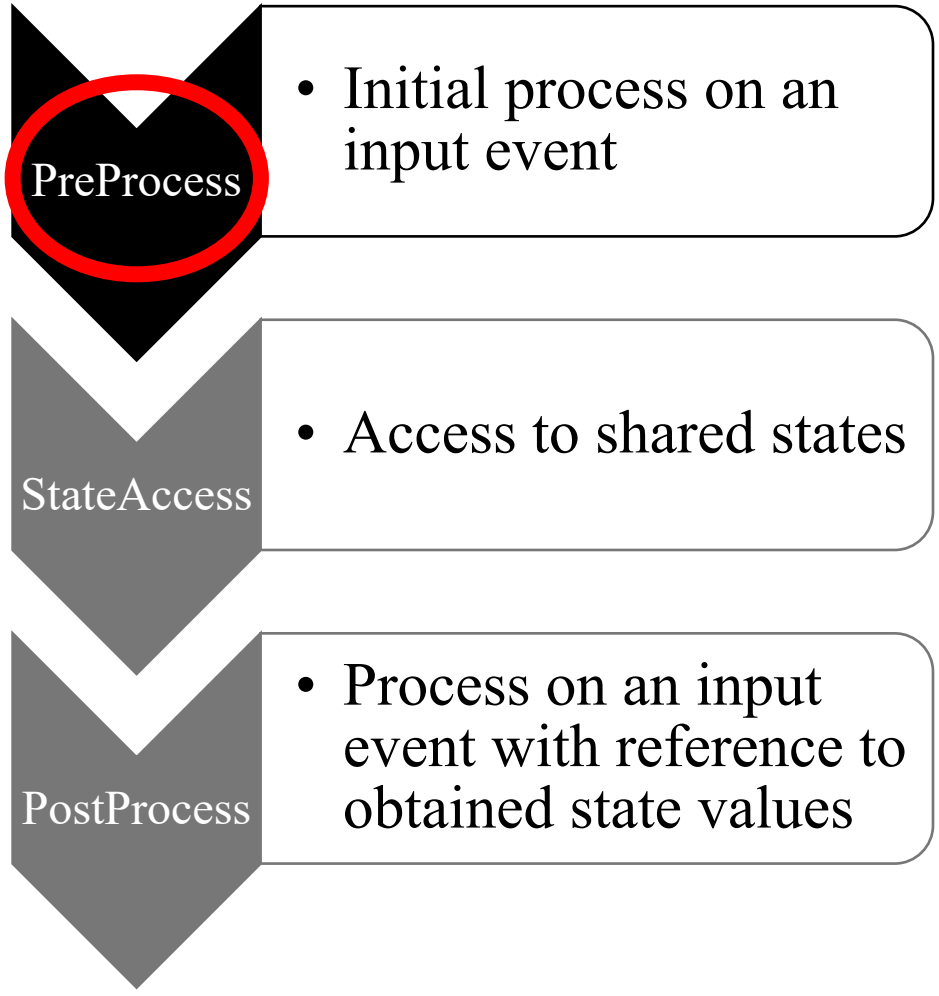
Up to 4.8 times higher throughput with similar or even lower processing latency!

Design1) Dual-Mode Scheduling

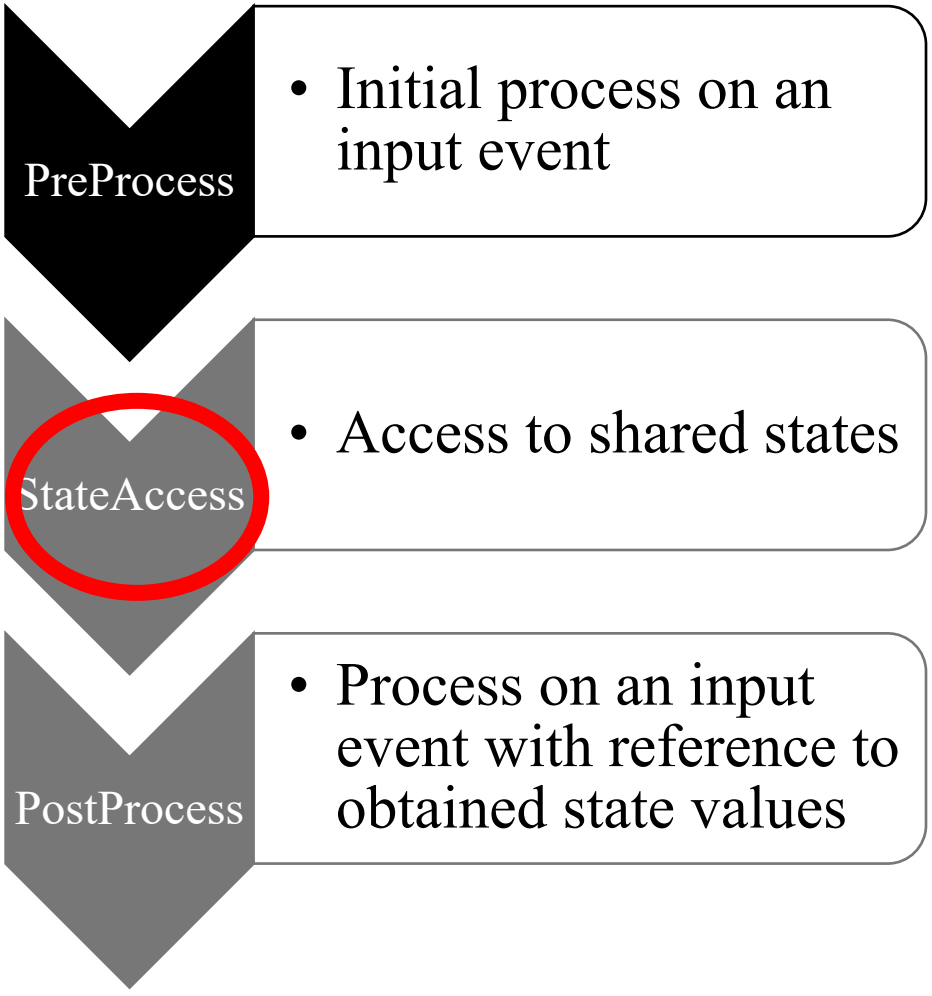


NUS National University of Singapore

Design1) Dual-Mode Scheduling

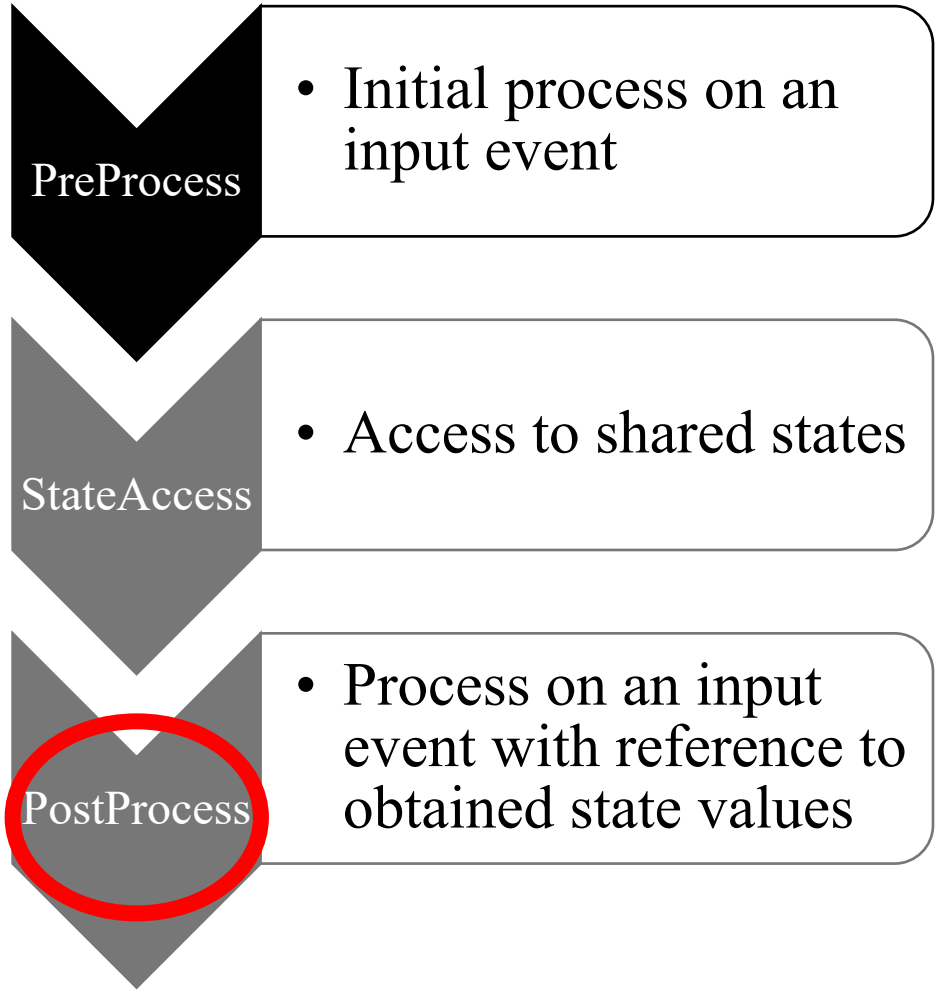


Design1) Dual-Mode Scheduling



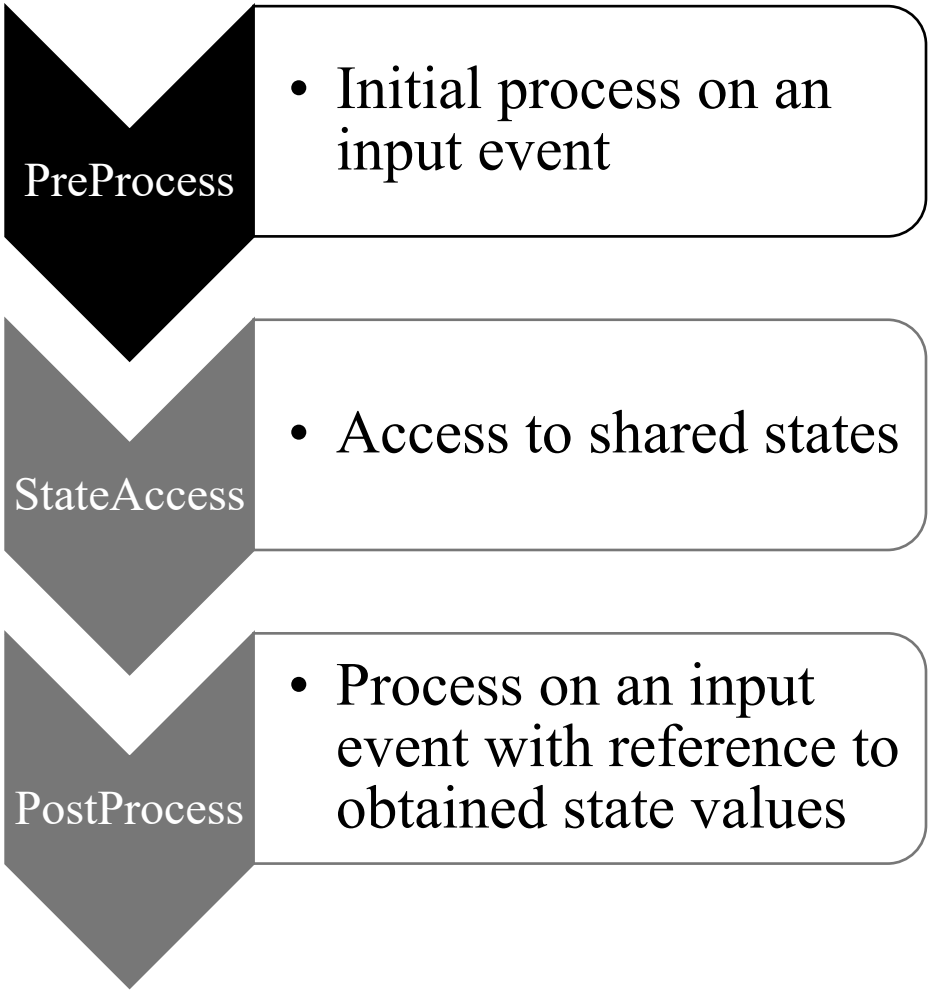
NUS National University of Singapore

Design1) Dual-Mode Scheduling

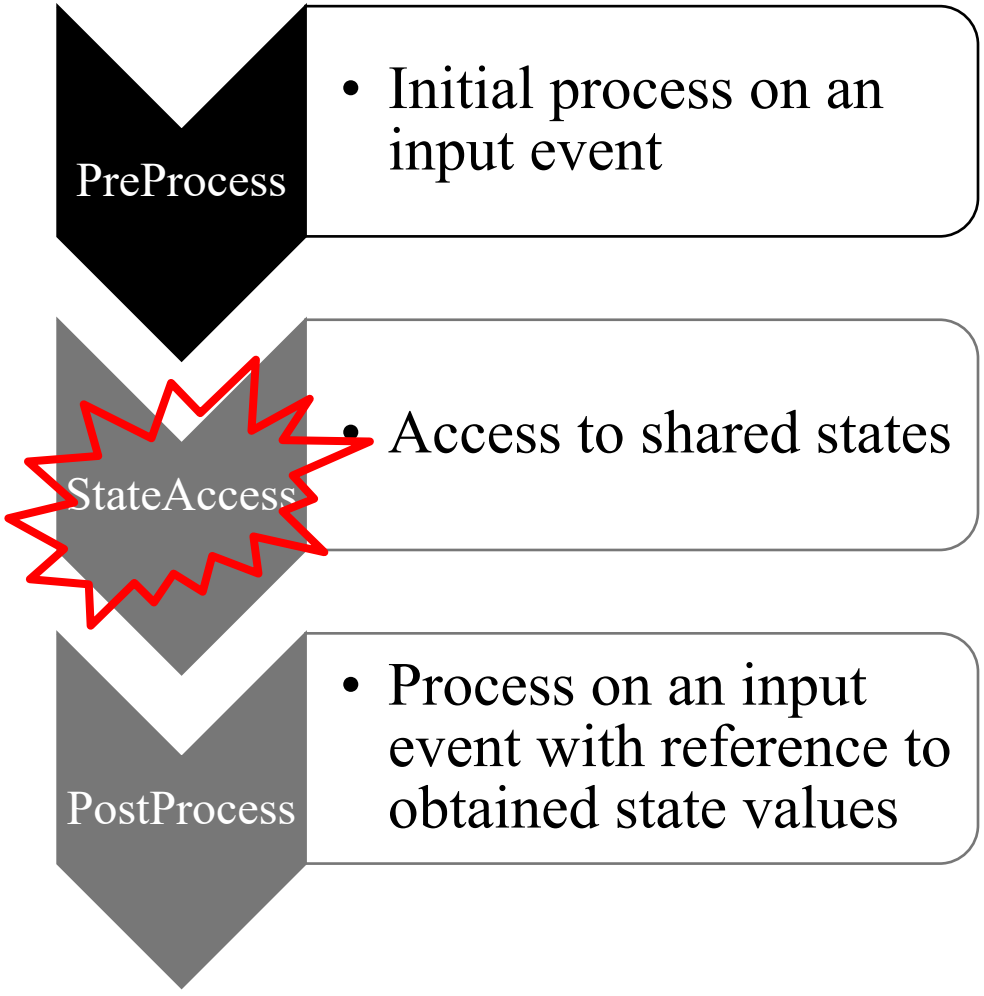


NUS National University of Singapore

Design1) Dual-Mode Scheduling

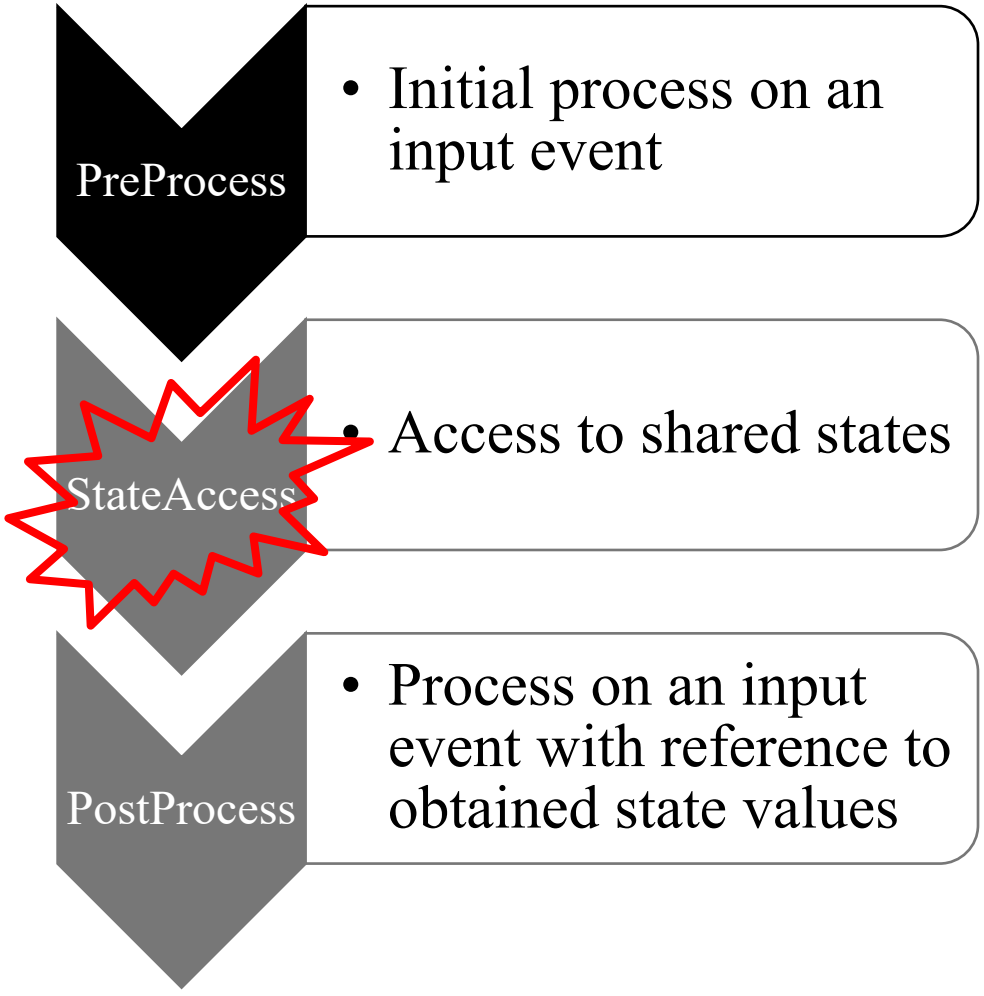


Design1) Dual-Mode Scheduling



- State access is often the bottleneck.
- It *unnecessarily* blocks all subsequent processes.

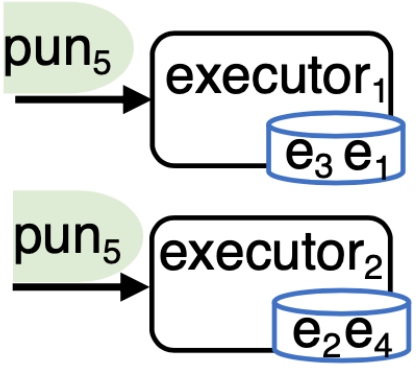
Design1) Dual-Mode Scheduling



- State access is often the bottleneck.
- It *unnecessarily* blocks all subsequent processes.

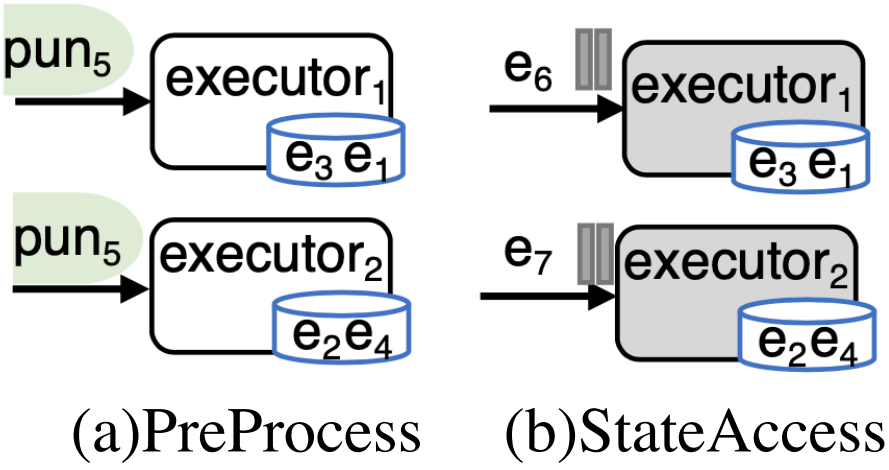
Let's postpone it.

Design 1) Dual-Mode Scheduling



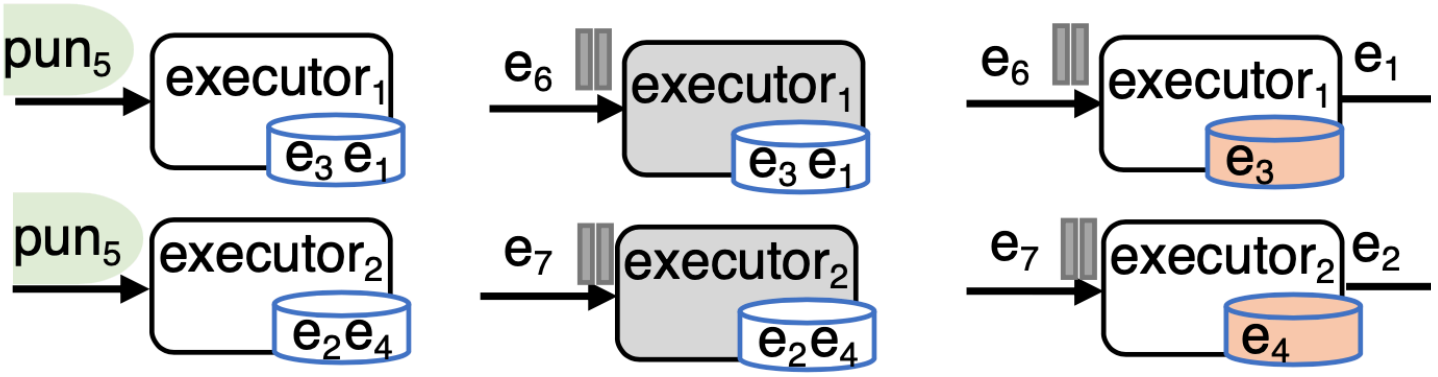
(a)PreProcess

Design 1) Dual-Mode Scheduling



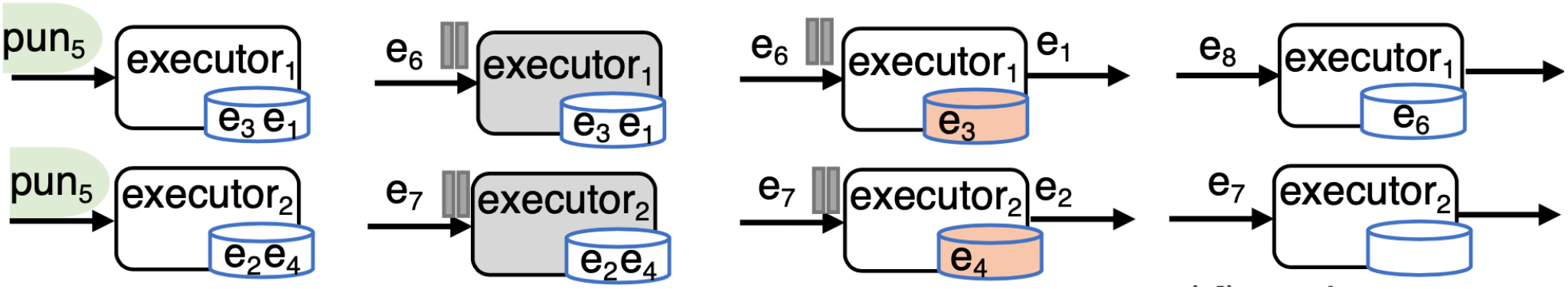
(a)PreProcess (b)StateAccess

Design 1) Dual-Mode Scheduling



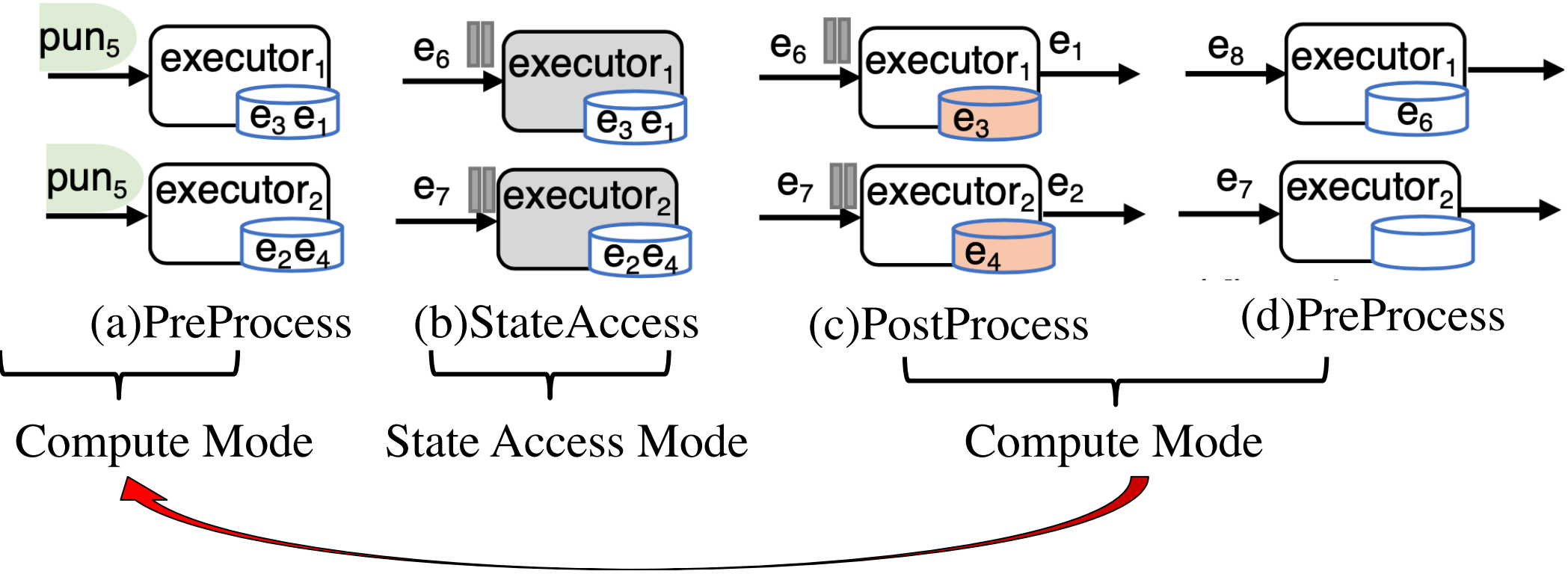
(a)PreProcess (b)StateAccess (c)PostProcess

Design 1) Dual-Mode Scheduling

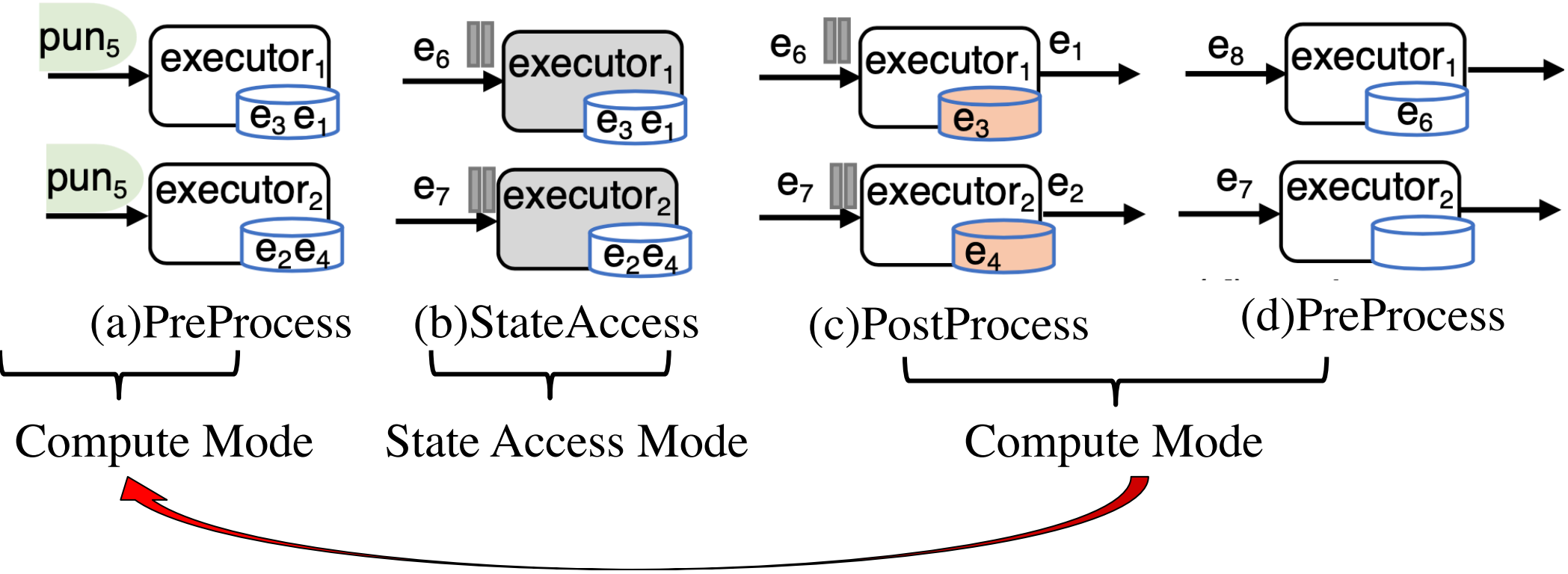


(a)PreProcess (b)StateAccess (c)PostProcess (d)PreProcess

Design 1) Dual-Mode Scheduling



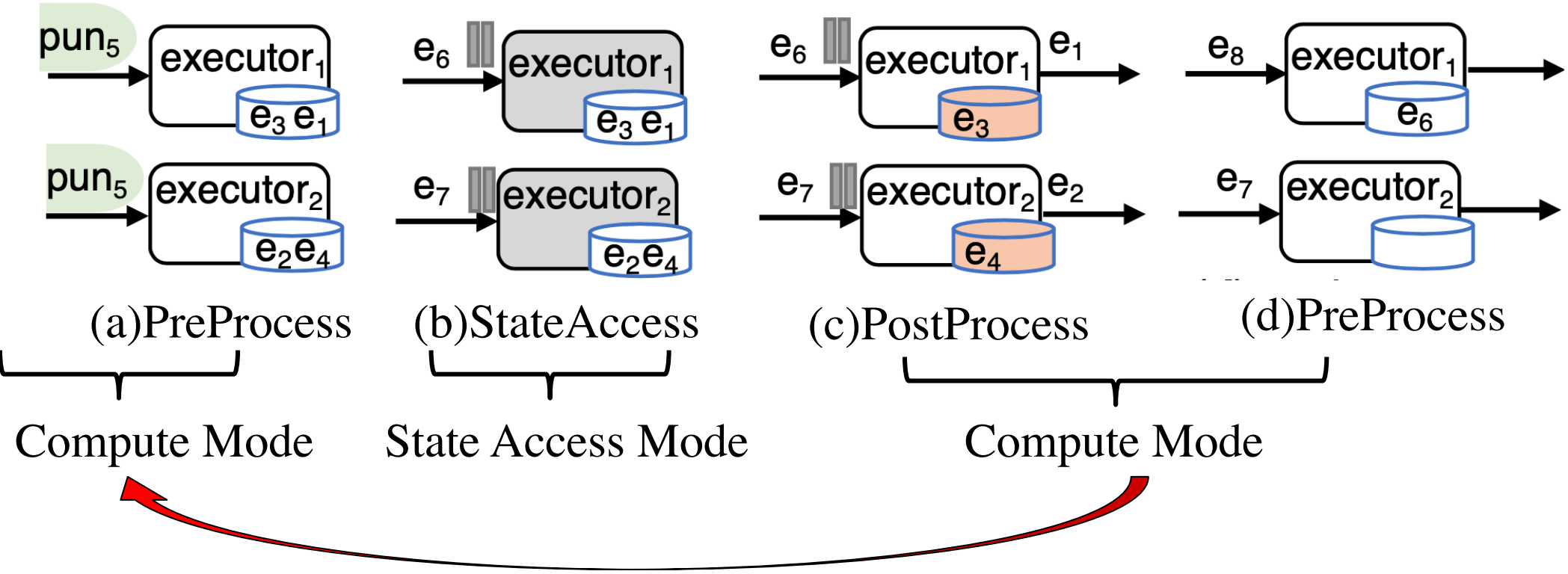
Design 1) Dual-Mode Scheduling



Pros: Enlarging inter-event parallelism opportunities.

Cons: Need to handle ``on-the-fly'' events.

Design 1) Dual-Mode Scheduling



Pros: Enlarging inter-event parallelism opportunities.

Cons: Need to handle ``on-the-fly'' events.

Design 2) Dynamic Restructuring Execution

Design 2) Dynamic Restructuring Execution

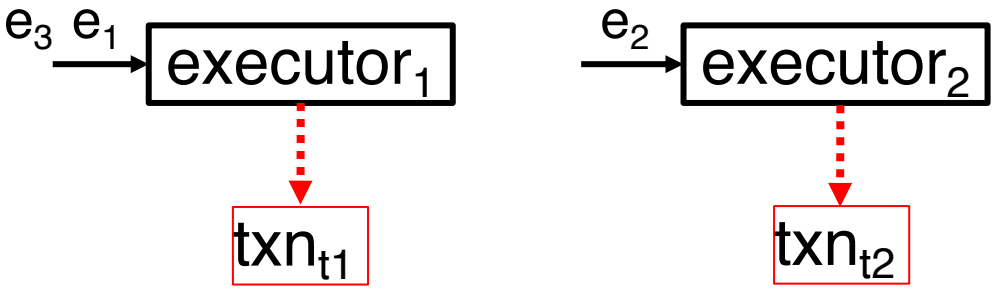
For each state transaction:

Dynamically decompose and regroup operations to ***avoid*** conflict before actual parallel evaluation is applied.

Design 2) Dynamic Restructuring Execution

For each state transaction:

Dynamically decompose and regroup operations to ***avoid*** conflict before actual parallel evaluation is applied.

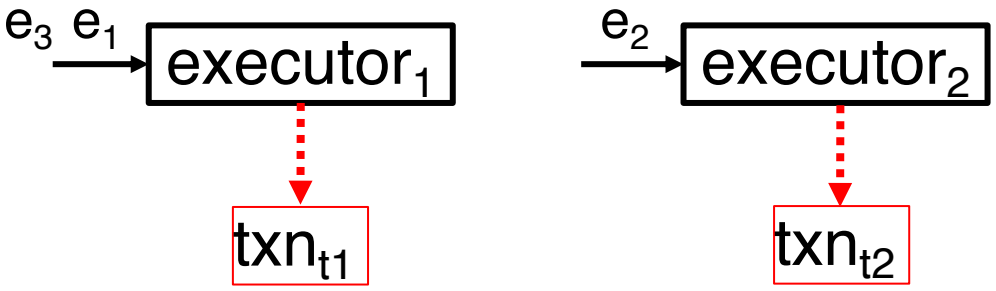


txn_{t1}: $\{O_1, O_2\}$
txn_2: $\{O_3\}$

Design 2) Dynamic Restructuring Execution

For each state transaction:

Dynamically decompose and regroup operations to ***avoid*** conflict before actual parallel evaluation is applied.



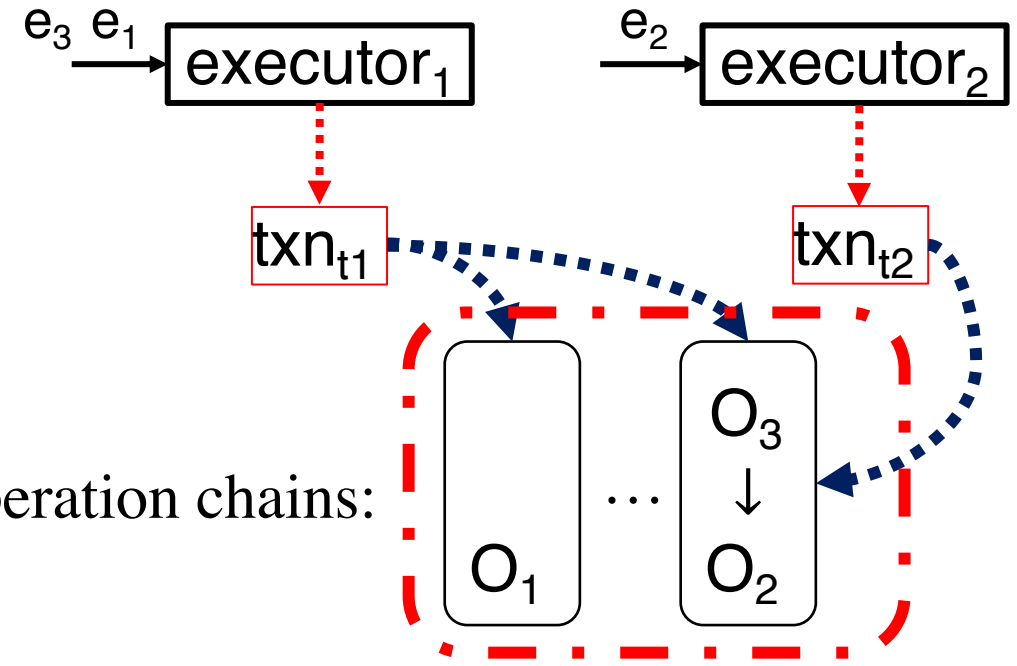
txn_{t1}: $\{O_1, O_2\}$
txn_2: $\{O_3\}$

O_1 : Read of Road 1
O_2 : Update to Road 2
O_3 : Read of Road 2

Design 2) Dynamic Restructuring Execution

For each state transaction:

Dynamically decompose and regroup operations to ***avoid*** conflict before actual parallel evaluation is applied.



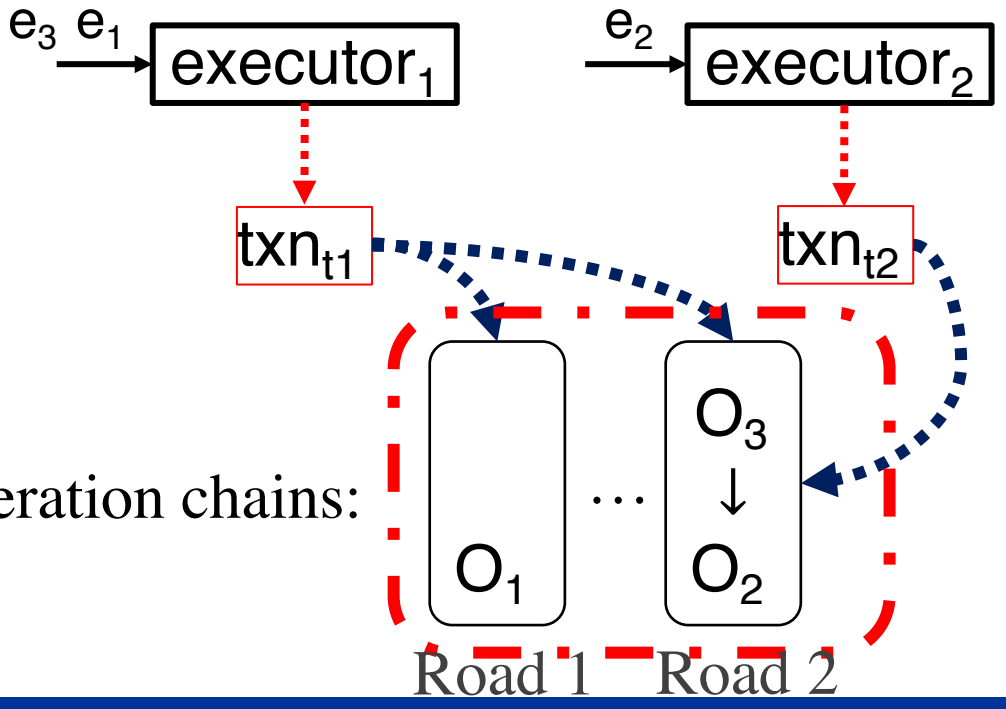
txn_{t1}: $\{O_1, O_2\}$
txn_2: $\{O_3\}$

O_1 : Read of Road 1
O_2 : Update to Road 2
O_3 : Read of Road 2

Design 2) Dynamic Restructuring Execution

For each state transaction:

Dynamically decompose and regroup operations to ***avoid*** conflict before actual parallel evaluation is applied.



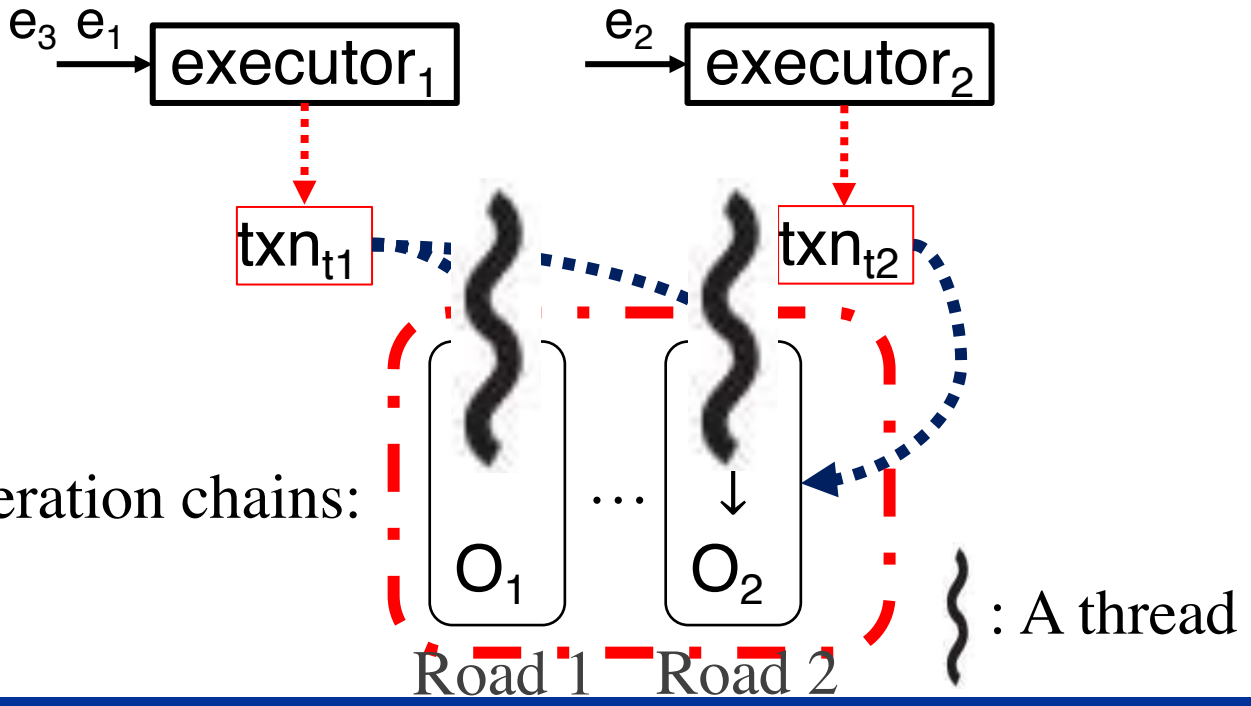
txn_{t1}: {O_1, O_2}
txn_2: {O_3}

O_1 : Read of Road 1
O_2 : Update to Road 2
O_3 : Read of Road 2

Design 2) Dynamic Restructuring Execution

For each state transaction:

Dynamically decompose and regroup operations to ***avoid*** conflict before actual parallel evaluation is applied.



txn_{t1}: {O_1, O_2}
txn_2: {O_3}

O_1 : Read of Road 1
O_2 : Update to Road 2
O_3 : Read of Road 2

Outline

- **Motivation**
- **State-of-the-art**
- **Our Approach**
- **Experimental Results**

Benchmark Workloads

Benchmark Workloads

- **Four applications:**
  - Grep Sum (GS); Streaming Ledger (SL); Online Bidding (OB); Toll Processing (TP).
- **Diverse characteristics:**
  - Varying compute/state access ratio
  - Varying state transaction types: read-only, write-only
  - Varying data dependencies
  - Varying multi-partition transactions

NUS National University of Singapore

Benchmark Workloads

- **Four applications:**
  - Grep Sum (GS); Streaming Ledger (SL); Online Bidding (OB); Toll Processing (TP).
- **Diverse characteristics:**
  - Varying compute/state access ratio
  - Varying state transaction types: read-only, write-only
  - Varying data dependencies
  - Varying multi-partition transactions

Benchmark Workloads

- **Four applications:**
  - Grep Sum (GS); Streaming Ledger (SL); Online Bidding (OB); Toll Processing (TP).
- **Diverse characteristics:**
  - Varying compute/state access ratio
  - Varying state transaction types: read-only, write-only
  - Varying data dependencies
  - Varying multi-partition transactions

Benchmark Workloads

- **Four applications:**
  - Grep Sum (GS); Streaming Ledger (SL); Online Bidding (OB); Toll Processing (TP).
- **Diverse characteristics:**
  - Varying compute/state access ratio
  - Varying state transaction types: read-only, write-only
  - Varying data dependencies
  - Varying multi-partition transactions

NUS National University of Singapore

Benchmark Workloads

- **Four applications:**
  - Grep Sum (GS); Streaming Ledger (SL); Online Bidding (OB); Toll Processing (TP).
- **Diverse characteristics:**
  - Varying compute/state access ratio
  - Varying state transaction types: read-only, write-only
  - Varying data dependencies
  - Varying multi-partition transactions

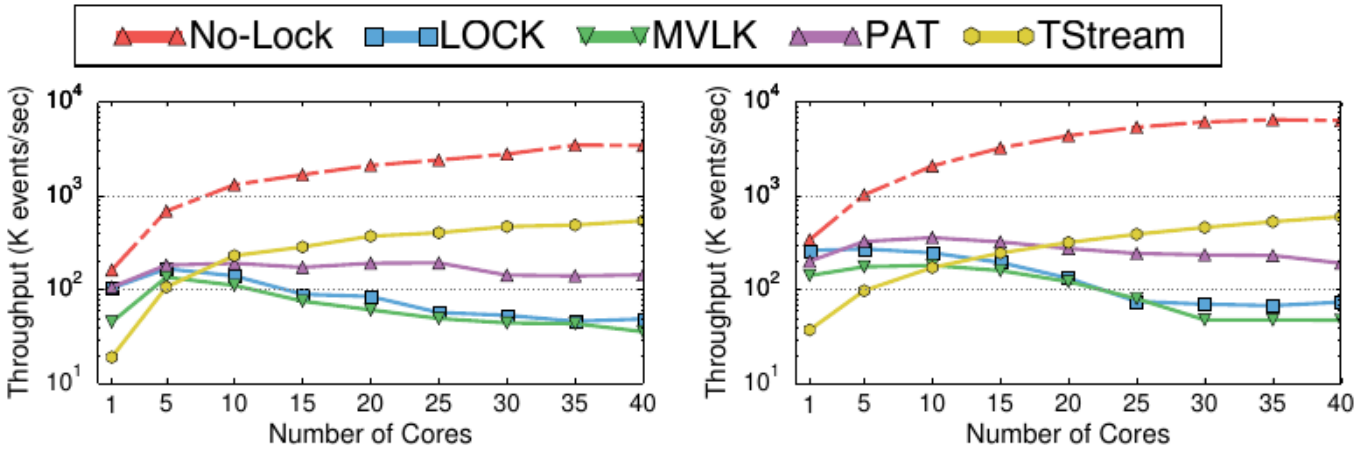
NUS National University of Singapore

Benchmark Workloads

- **Four applications:**
  - Grep Sum (GS); Streaming Ledger (SL); Online Bidding (OB); Toll Processing (TP).
- **Diverse characteristics:**
  - Varying compute/state access ratio
  - Varying state transaction types: read-only, write-only
  - Varying data dependencies
  - Varying multi-partition transactions

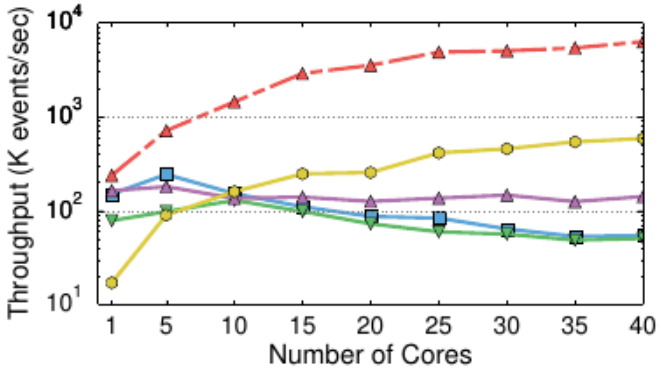
Experiments conducted on a 4-socket Intel Xeon E7- 4820 server with 128 GB DRAM.

Overall Performance Comparison

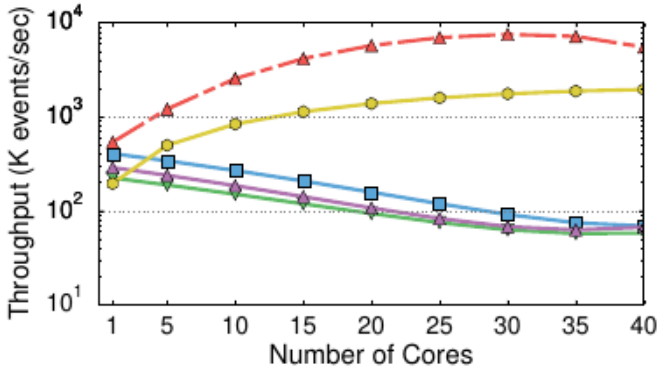


(a) *GS* (50% read requests)

(b) *SL*

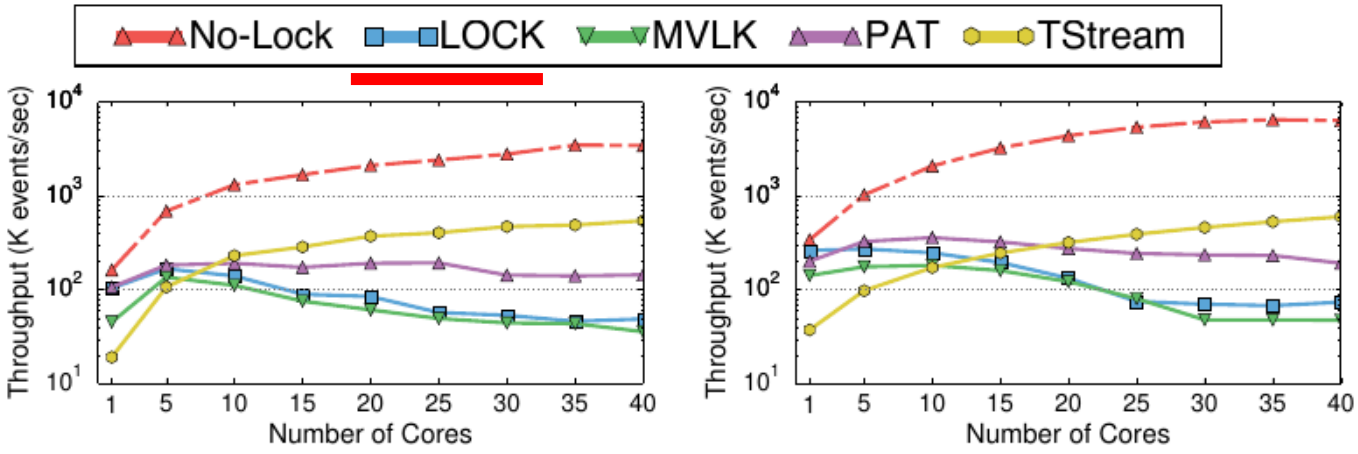


(c) *OB*



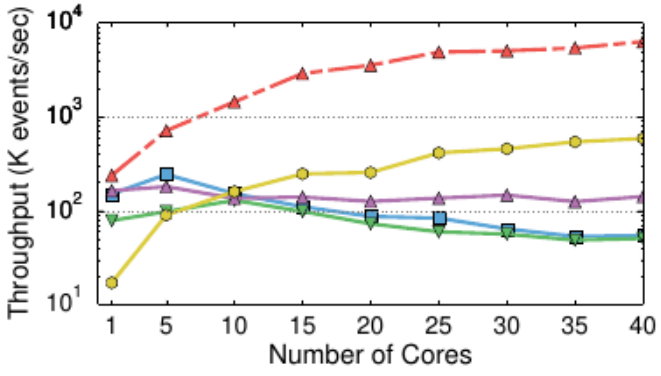
(d) *TP*

Overall Performance Comparison

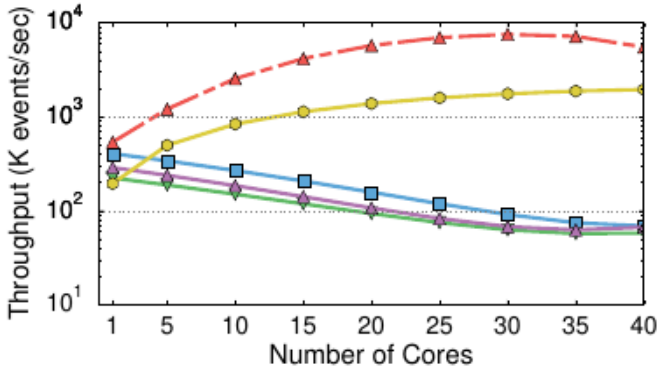


(a) *GS* (50% read requests)

(b) *SL*

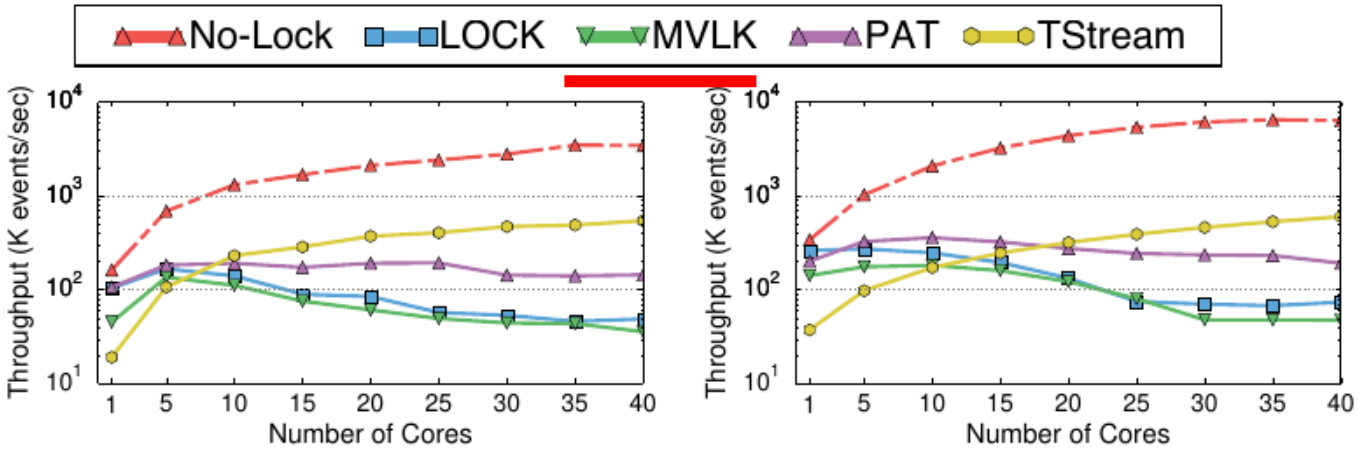


(c) *OB*



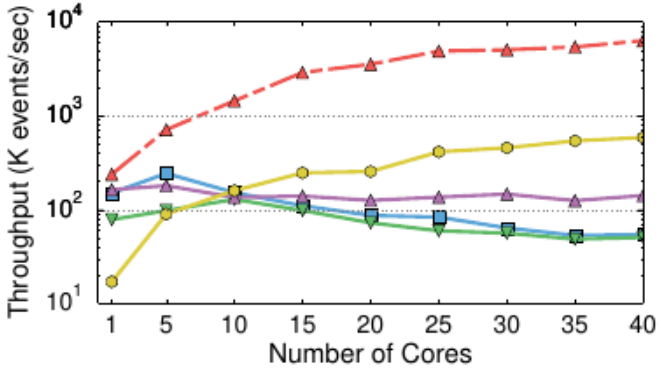
(d) *TP*

Overall Performance Comparison

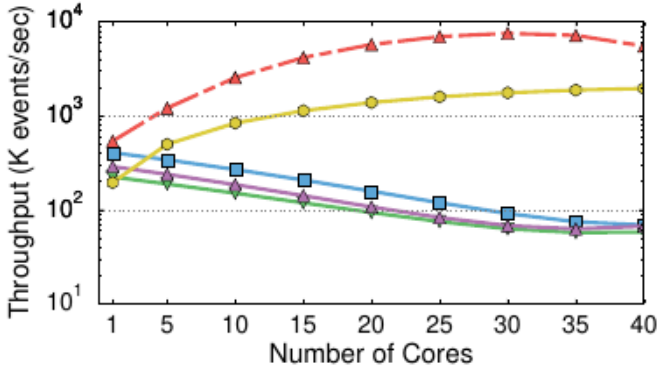


(a) *GS* (50% read requests)

(b) *SL*

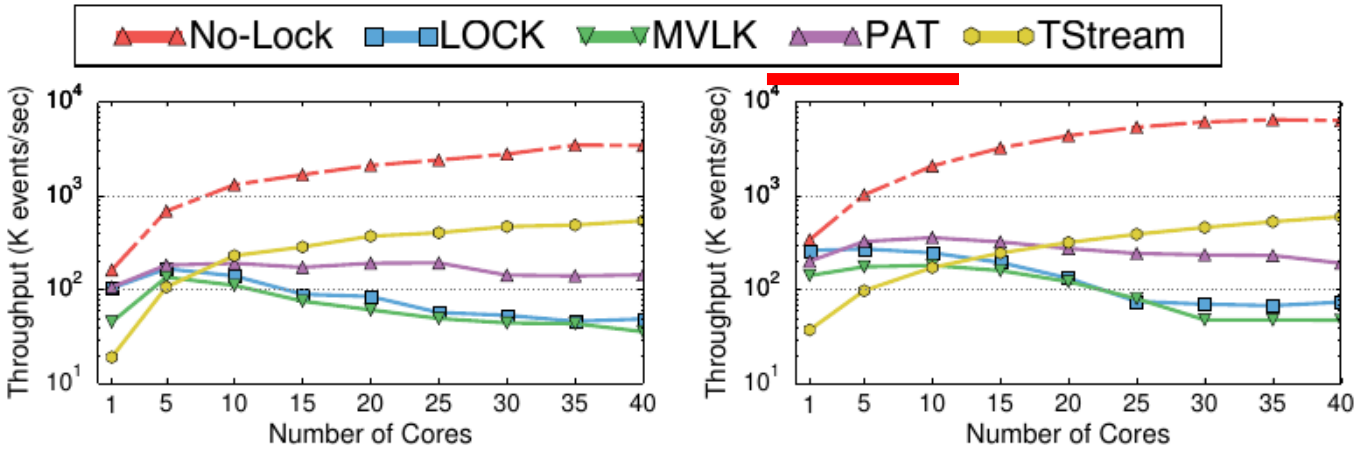


(c) *OB*



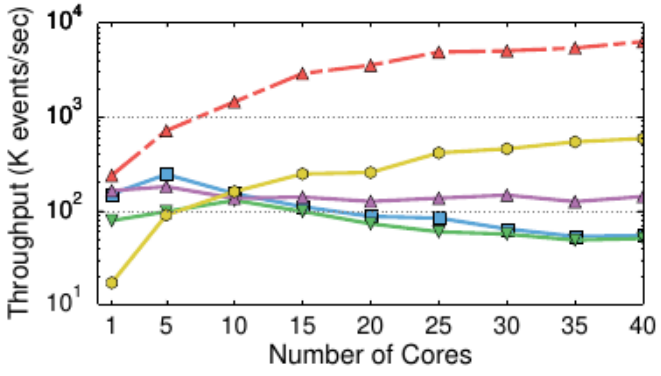
(d) *TP*

Overall Performance Comparison

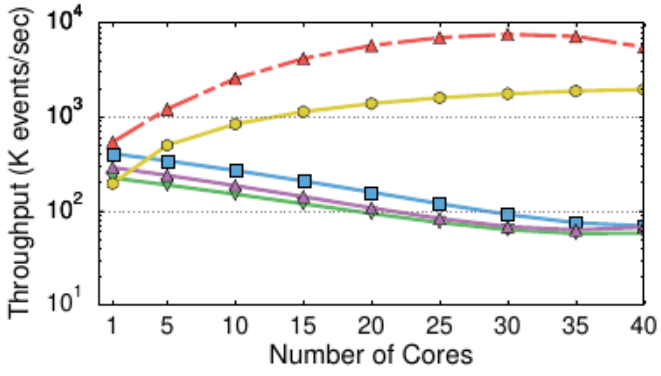


(a) *GS* (50% read requests)

(b) *SL*

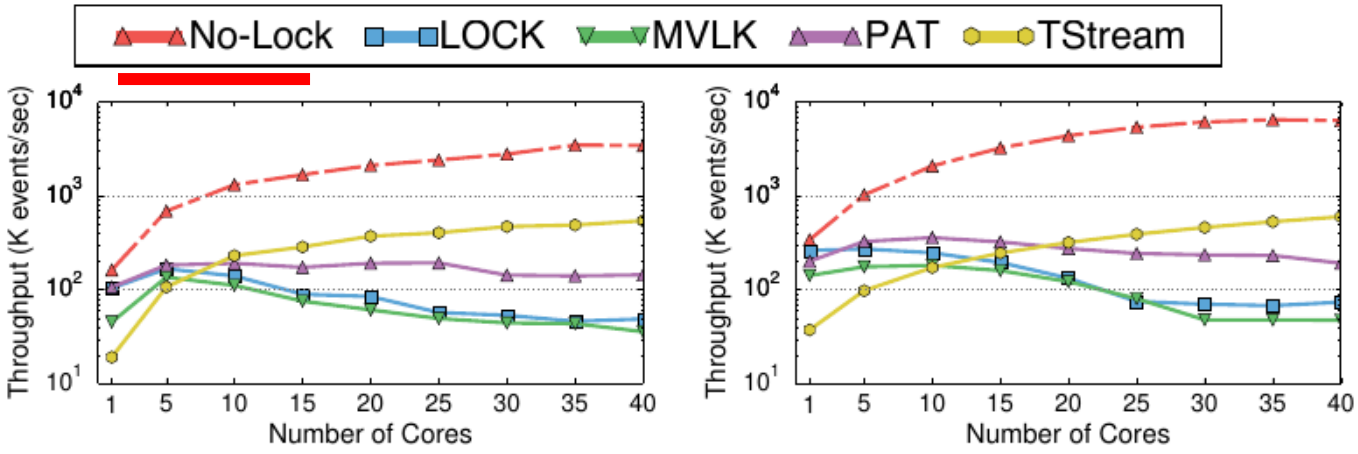


(c) *OB*



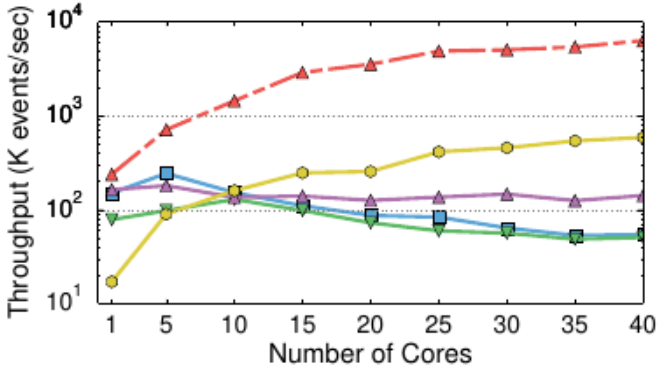
(d) *TP*

Overall Performance Comparison

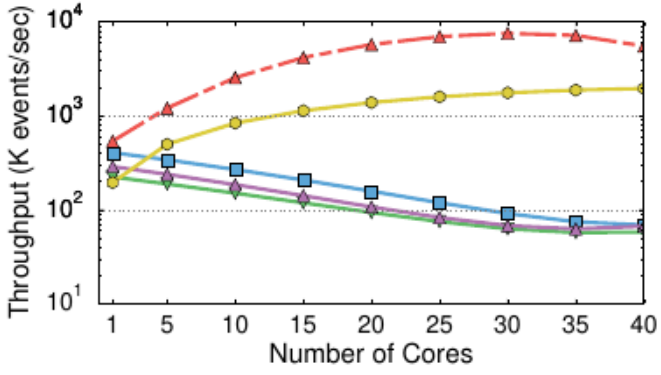


(a) *GS* (50% read requests)

(b) *SL*

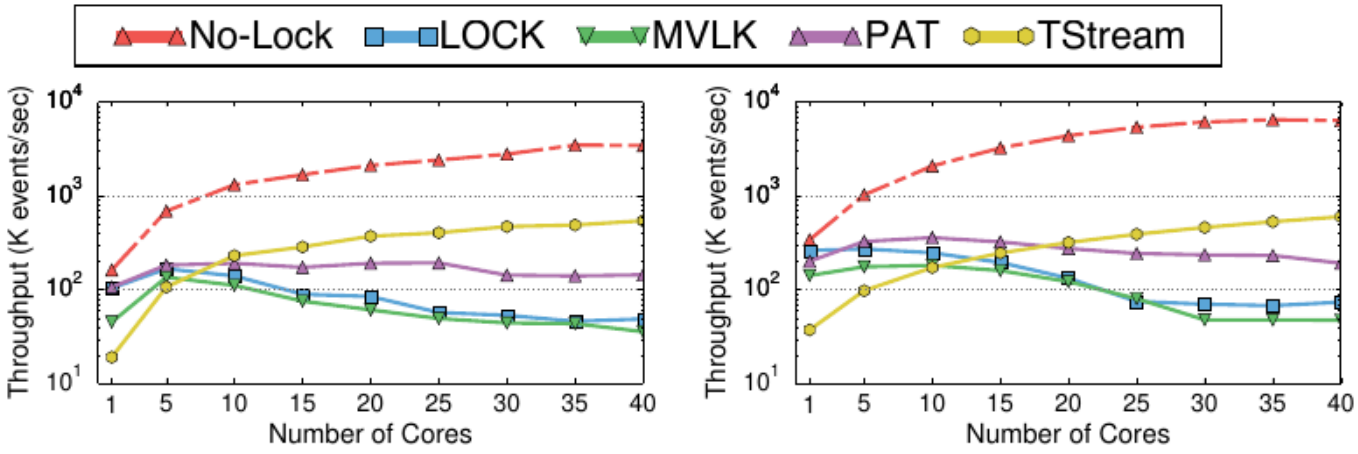


(c) *OB*



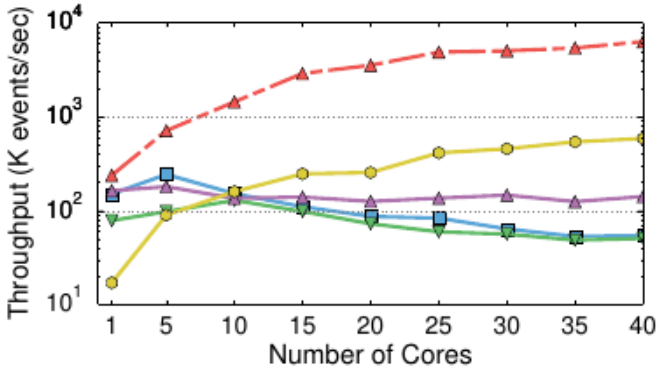
(d) *TP*

Overall Performance Comparison

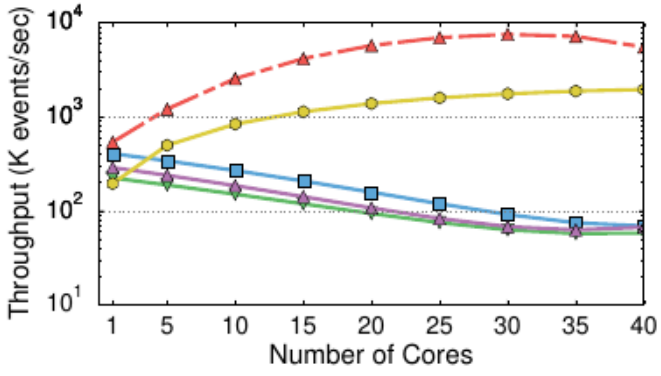


(a) *GS* (50% read requests)

(b) *SL*

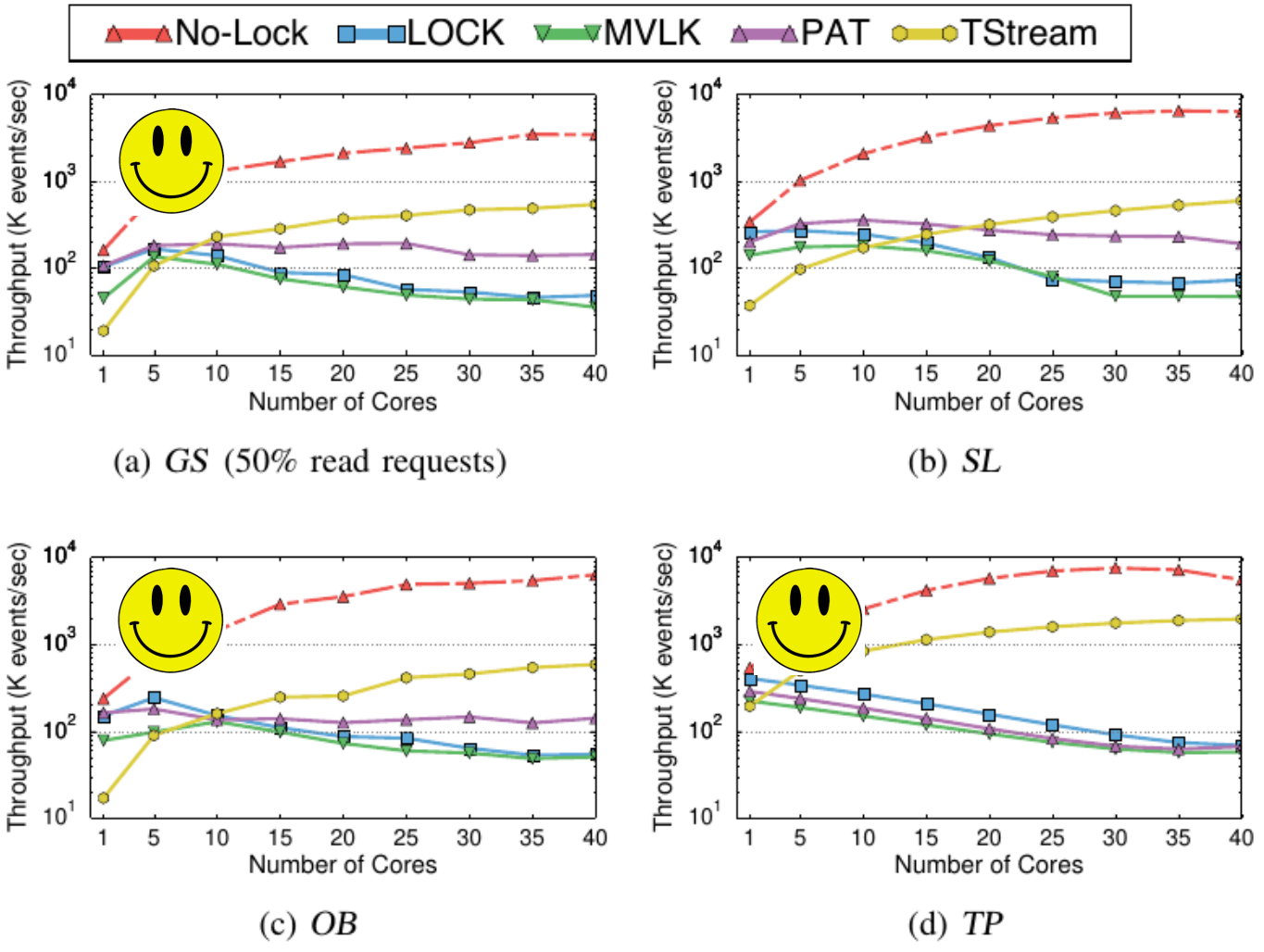


(c) *OB*



(d) *TP*

Overall Performance Comparison

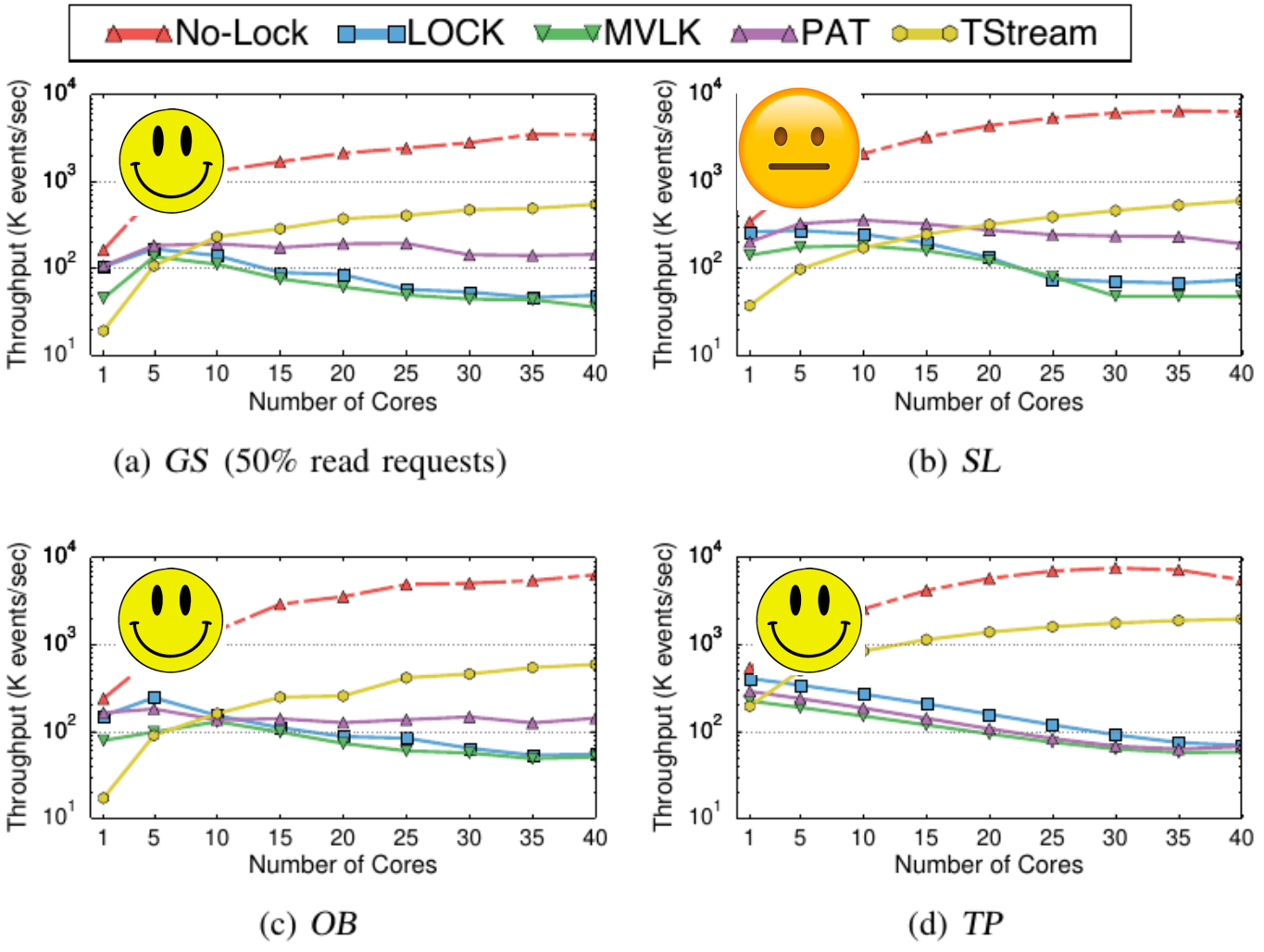


(i) In general, TStream performs well at large core counts.

(ii) TStream performs slightly better when data dependency is heavily presented (SL).

(iii) Large room for further improvement.

Overall Performance Comparison

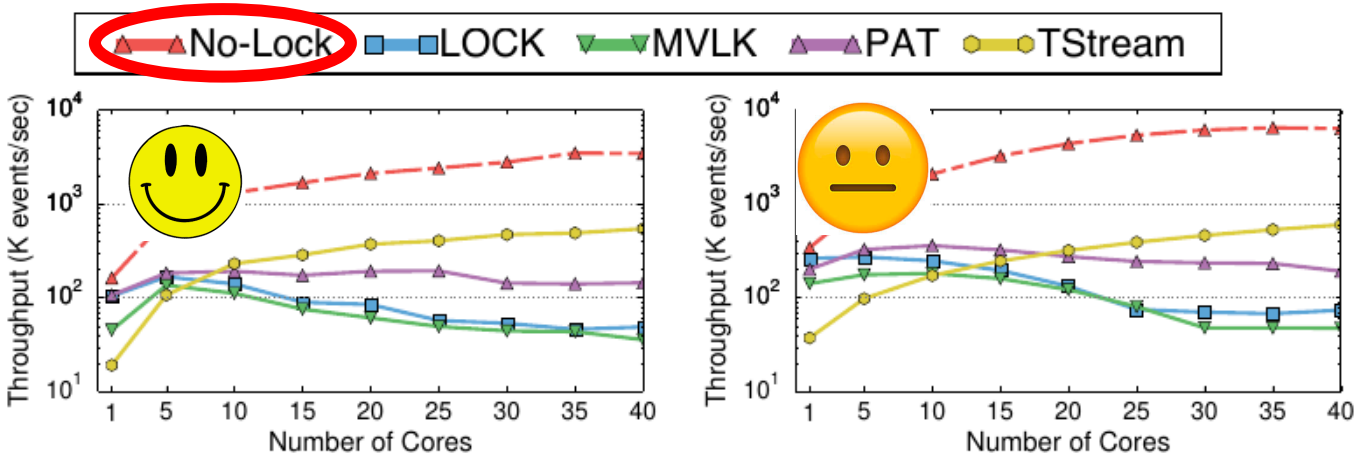


(i) In general, TStream performs well at large core counts.

(ii) TStream performs slightly better when data dependency is heavily presented (SL).

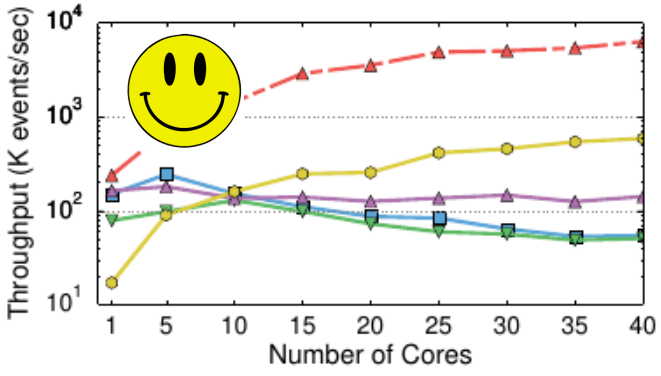
(iii) Large room for further improvement.

Overall Performance Comparison

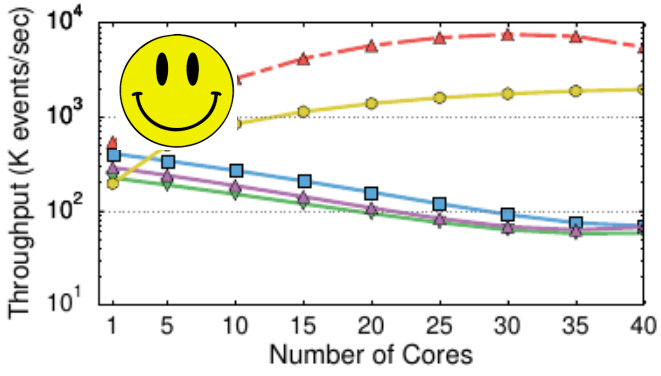


(a) *GS* (50% read requests)

(b) *SL*



(c) *OB*



(d) *TP*

(i) In general, TStream performs well at large core counts.

(ii) TStream performs slightly better when data dependency is heavily presented (SL).

(iii) Large room for further improvement.

Recap

Recap

- [Key Designs] 1) dual-mode scheduling, 2) transaction restructuring

- [Results] TStream performs constantly better than prior solutions under varying workloads.

NUS National University of Singapore

Recap

- [Key Designs] 1) dual-mode scheduling, 2) transaction restructuring

- [Results] TStream performs constantly better than prior solutions under varying workloads.

https://github.com/Xtra-Computing/briskstream/tree/TStream

NUS National University of Singapore

Current Work

Next Generation IoT Data Management Platform

(lead by Prof. Volker Markl)

visit us @

www.nebula.stream

Twitter@nebulastream

NUS National University of Singapore

Current Work

Next Generation IoT Data Management Platform

(lead by Prof. Volker Markl)

visit us @

www.nebula.stream

Twitter@nebulastream

Thank you!

shuhao.zhang@tu-berlin.de

Competitor Schemes

Competitor Schemes

① Ordering-lock-based approach (LOCK).

② Multi-versioning-based approach (MVLK).

③ Static-partition-based approach (PAT).

- S-Store

④ No Consistency Guarantee (No-Lock).

- Remove locks from LOCK scheme.
- Represent the performance upper bound.

Competitor Schemes

① Ordering-lock-based approach (LOCK).

② Multi-versioning-based approach (MVLK).

③ Static-partition-based approach (PAT).

- S-Store

④ No Consistency Guarantee (No-Lock).

- Remove locks from LOCK scheme.
- Represent the performance upper bound.

Competitor Schemes

① Ordering-lock-based approach (LOCK).

② Multi-versioning-based approach (MVLK).

③ Static-partition-based approach (PAT).

- S-Store

④ No Consistency Guarantee (No-Lock).

- Remove locks from LOCK scheme.
- Represent the performance upper bound.

Competitor Schemes

① Ordering-lock-based approach (LOCK).

② Multi-versioning-based approach (MVLK).

③ Static-partition-based approach (PAT).

- S-Store

④ No Consistency Guarantee (No-Lock).

- Remove locks from LOCK scheme.
- Represent the performance upper bound.

Competitor Schemes

① Ordering-lock-based approach (LOCK).

② Multi-versioning-based approach (MVLK).

③ Static-partition-based approach (PAT).

- S-Store

④ No Consistency Guarantee (No-Lock).

- Remove locks from LOCK scheme.
- Represent the performance upper bound.

Competitor Schemes

① Ordering-lock-based approach (LOCK).

② Multi-versioning-based approach (MVLK).

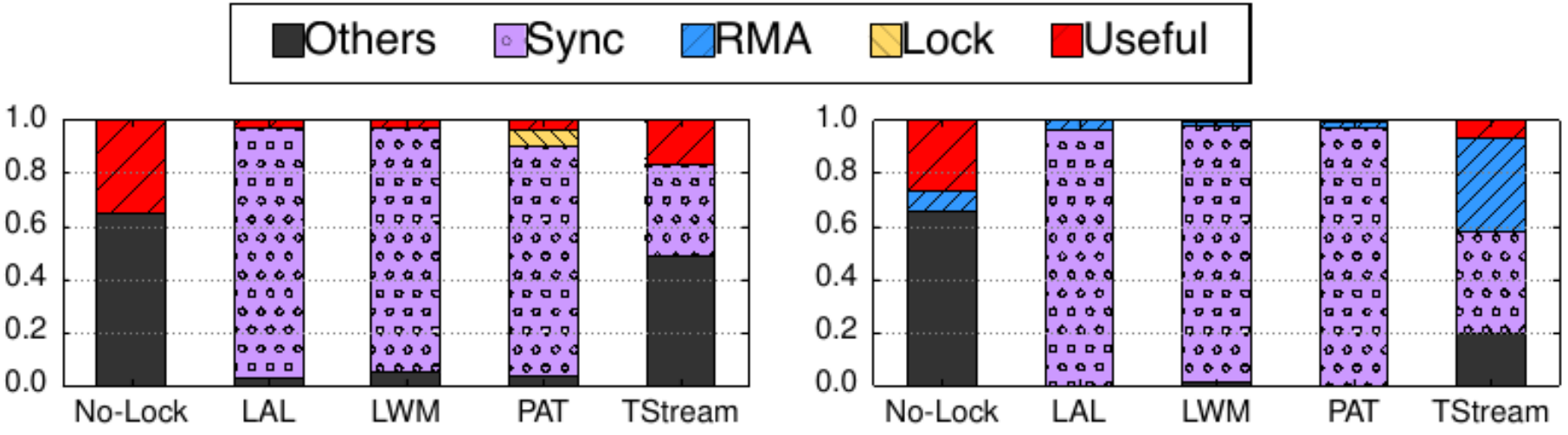
③ Static-partition-based approach (PAT).

- S-Store

④ No Consistency Guarantee (No-Lock).

- Remove locks from LOCK scheme.
- Represent the performance upper bound.

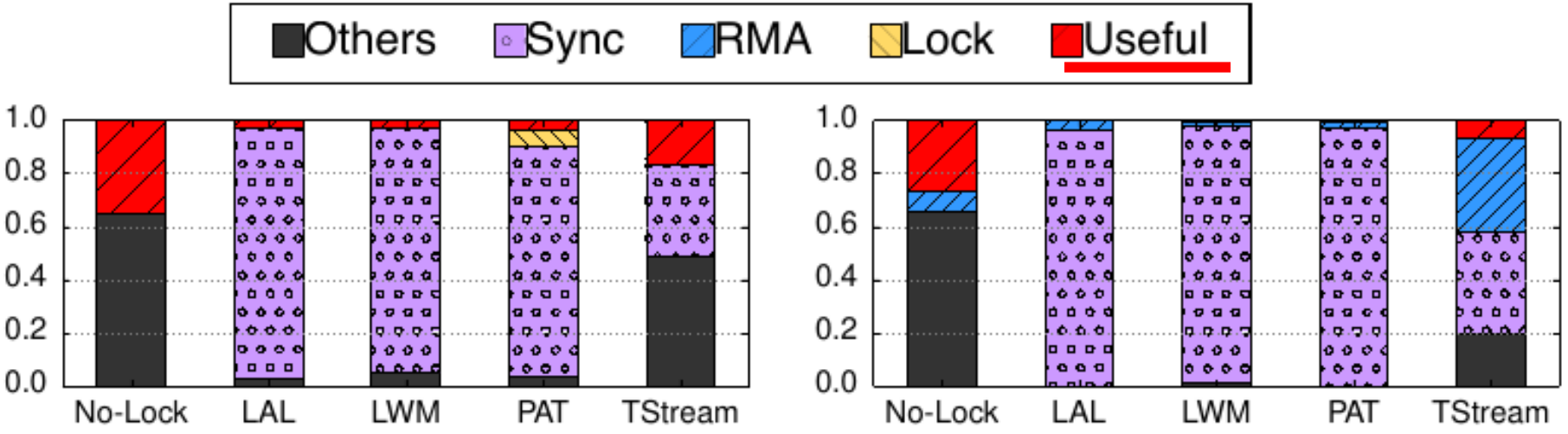
Runtime Breakdown



(a) Single socket

(b) Four sockets

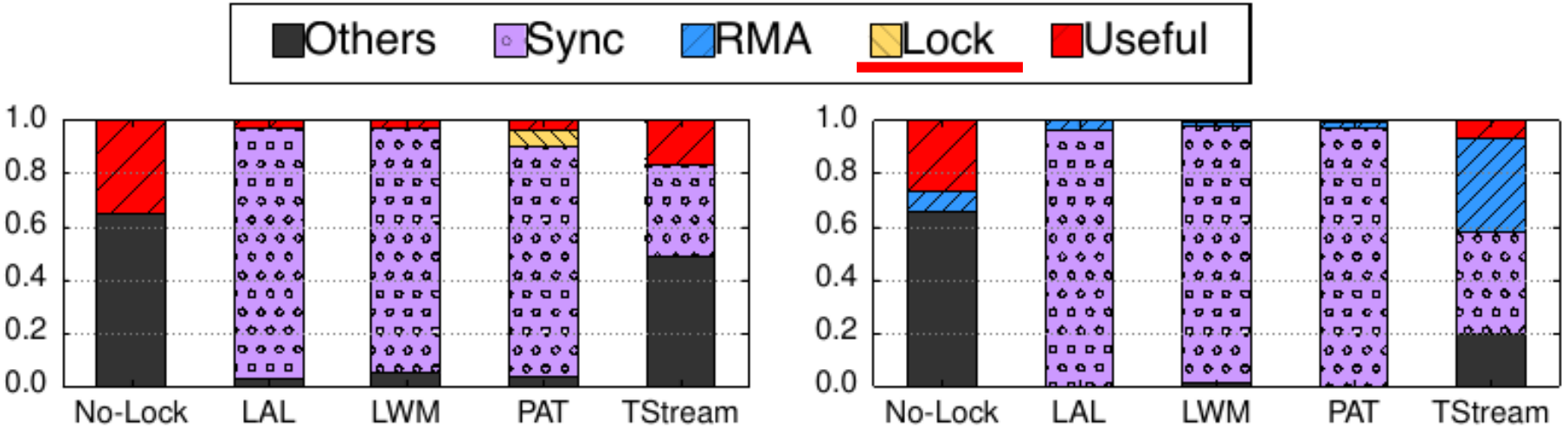
Runtime Breakdown



(a) Single socket

(b) Four sockets

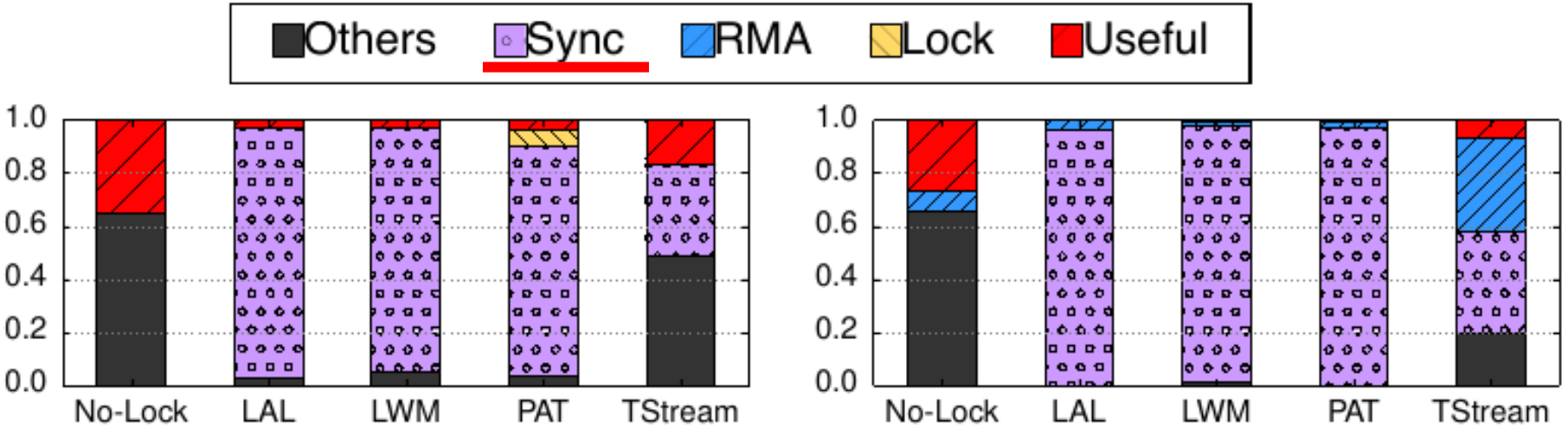
Runtime Breakdown



(a) Single socket

(b) Four sockets

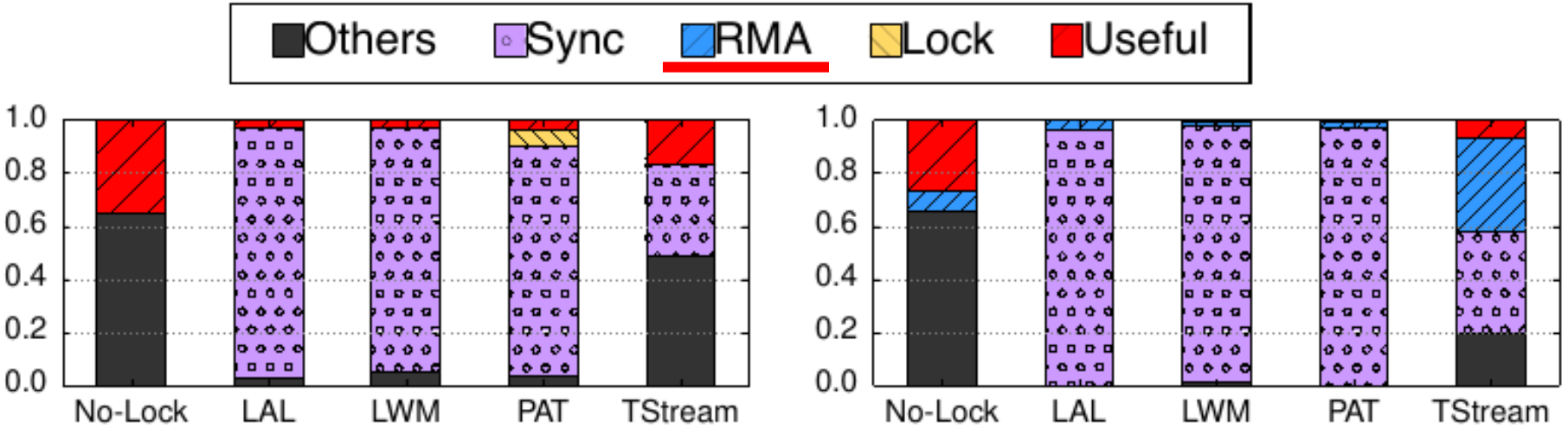
Runtime Breakdown



(a) Single socket

(b) Four sockets

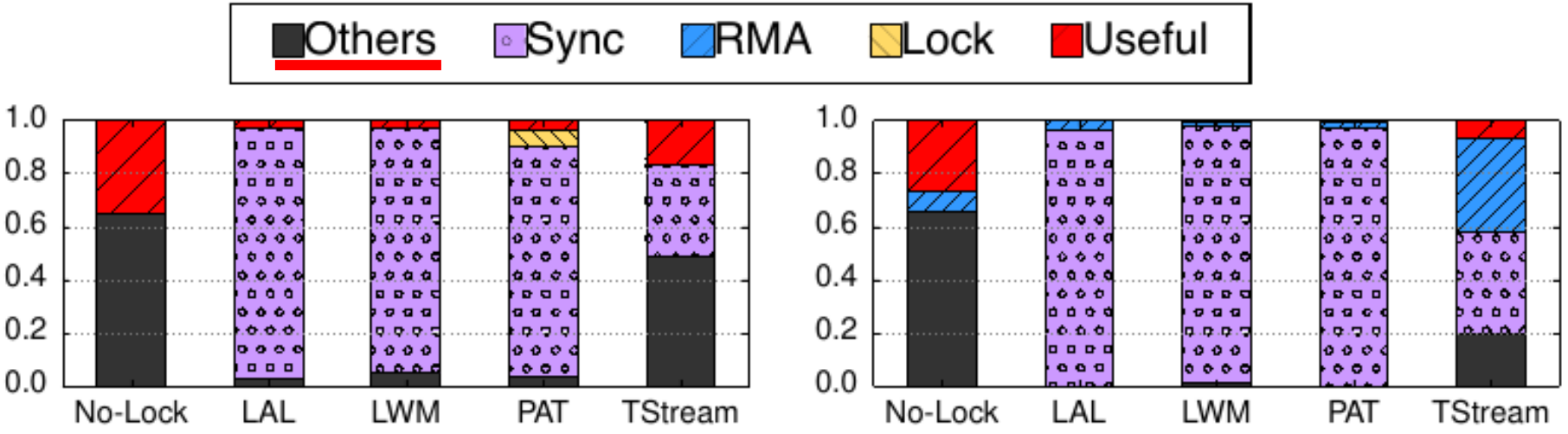
Runtime Breakdown



(a) Single socket

(b) Four sockets

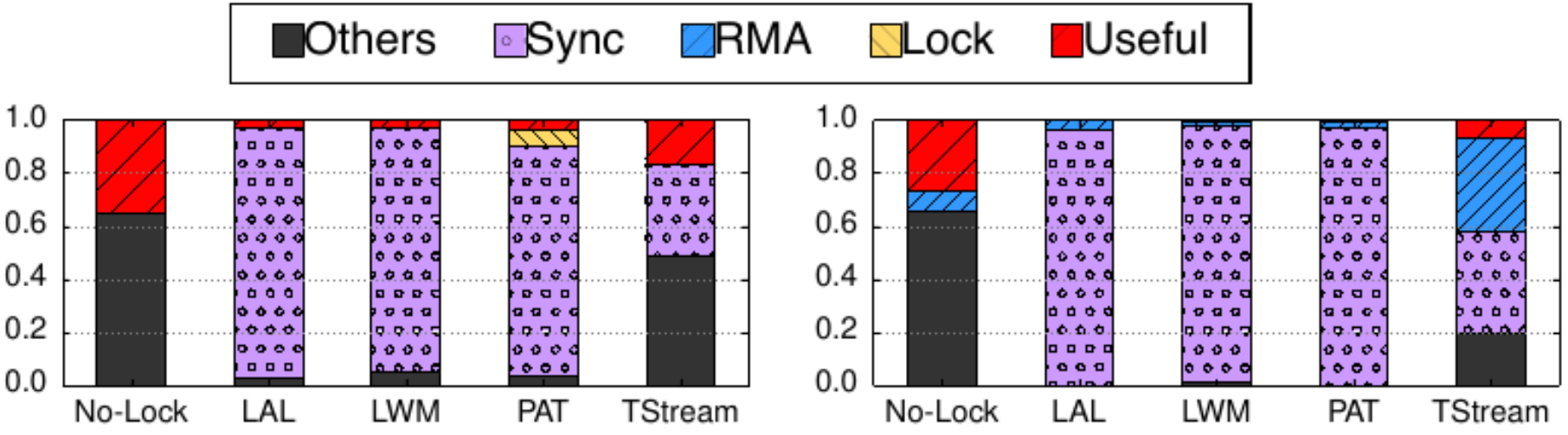
Runtime Breakdown



(a) Single socket

(b) Four sockets

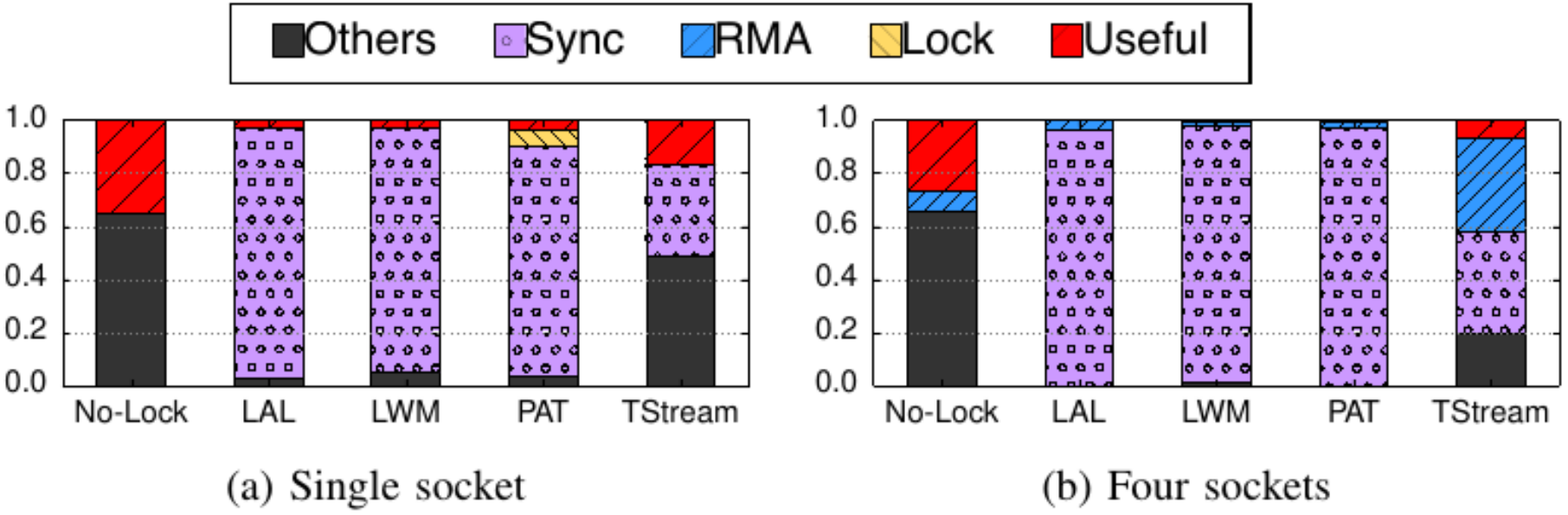
Runtime Breakdown



(a) Single socket

(b) Four sockets

Runtime Breakdown



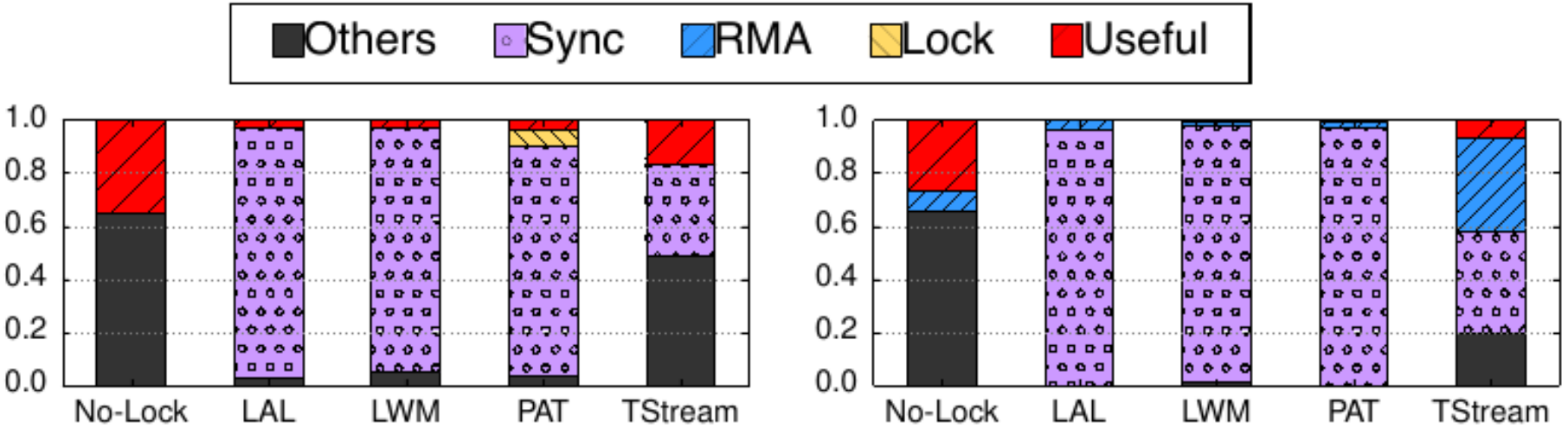
(a) No-Lock scheme spends more than 60% in others mainly due to index look up.

(b) Synchronization overhead dominates in all prior solutions regardless of NUMA effect.

(c) TStream involves high RMA overhead.

NUMA-aware optimization detailed in our paper.

Runtime Breakdown



(a) Single socket (b) Four sockets

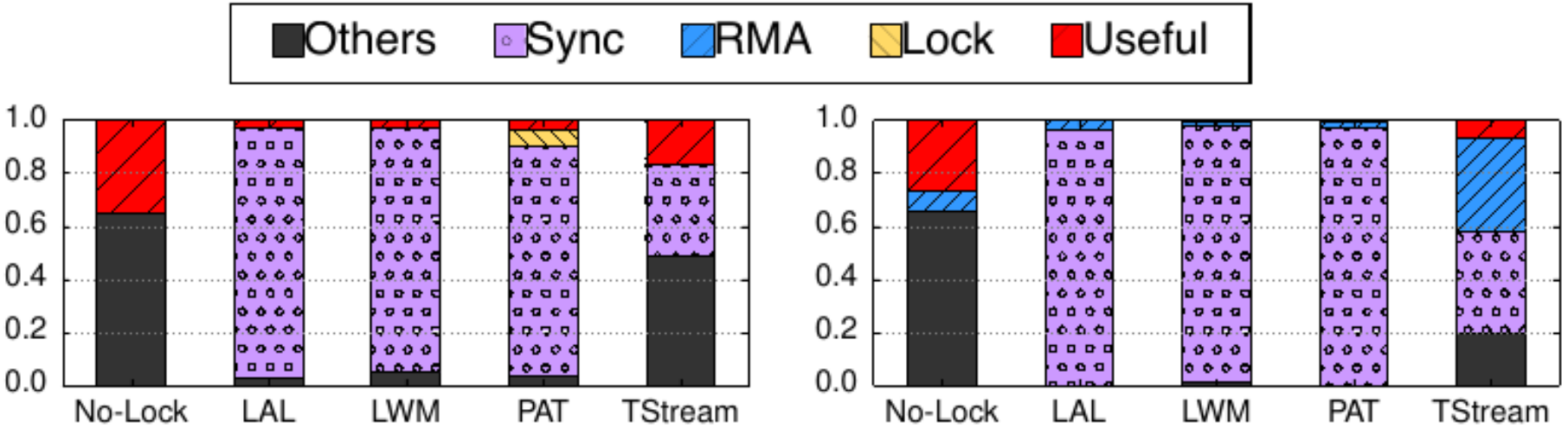
(a) No-Lock scheme spends more than 60% in others mainly due to index look up.

(b) Synchronization overhead dominates in all prior solutions regardless of NUMA effect.

(c) TStream involves high RMA overhead.

NUMA-aware optimization detailed in our paper.

Runtime Breakdown



(a) No-Lock scheme spends more than 60% in others mainly due to index look up.

(b) Synchronization overhead dominates in all prior solutions regardless of NUMA effect.

(c) TStream involves high RMA overhead.

NUMA-aware optimization detailed in our paper.

NUS National University of Singapore

No free lunch

- **Currently, TStream has its drawback on efficiently handling transaction abort.**
  - E.g., one that leaves the average speed of a road to become negative.

NUS National University of Singapore

No free lunch

- **Currently, TStream has its drawback on efficiently handling transaction abort.**
  - E.g., one that leaves the average speed of a road to become negative.

Stay tuned for future work