Towards Concurrent Stateful Stream Processing on Multicore Processors

Shuhao Zhang

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

National University of Singapore

Xtra Computing Group

lead by Prof. Bingsheng He

Data Stream Processing Systems

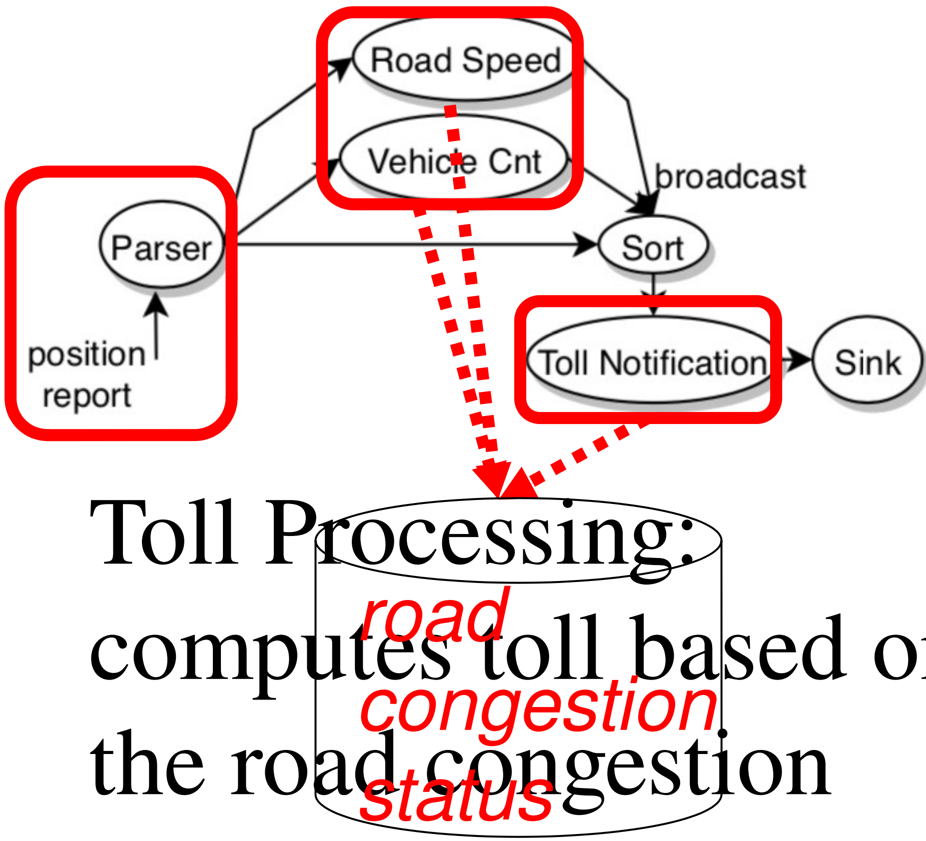
- [Available] Fast stream processing on large-scale multicore architectures
  - E.g., BriskStream[1], StreamBox[2], Grizzly[3]
- [Inadequate] Support of *consistent stateful stream processing*

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*

Outline

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

Motivation Example

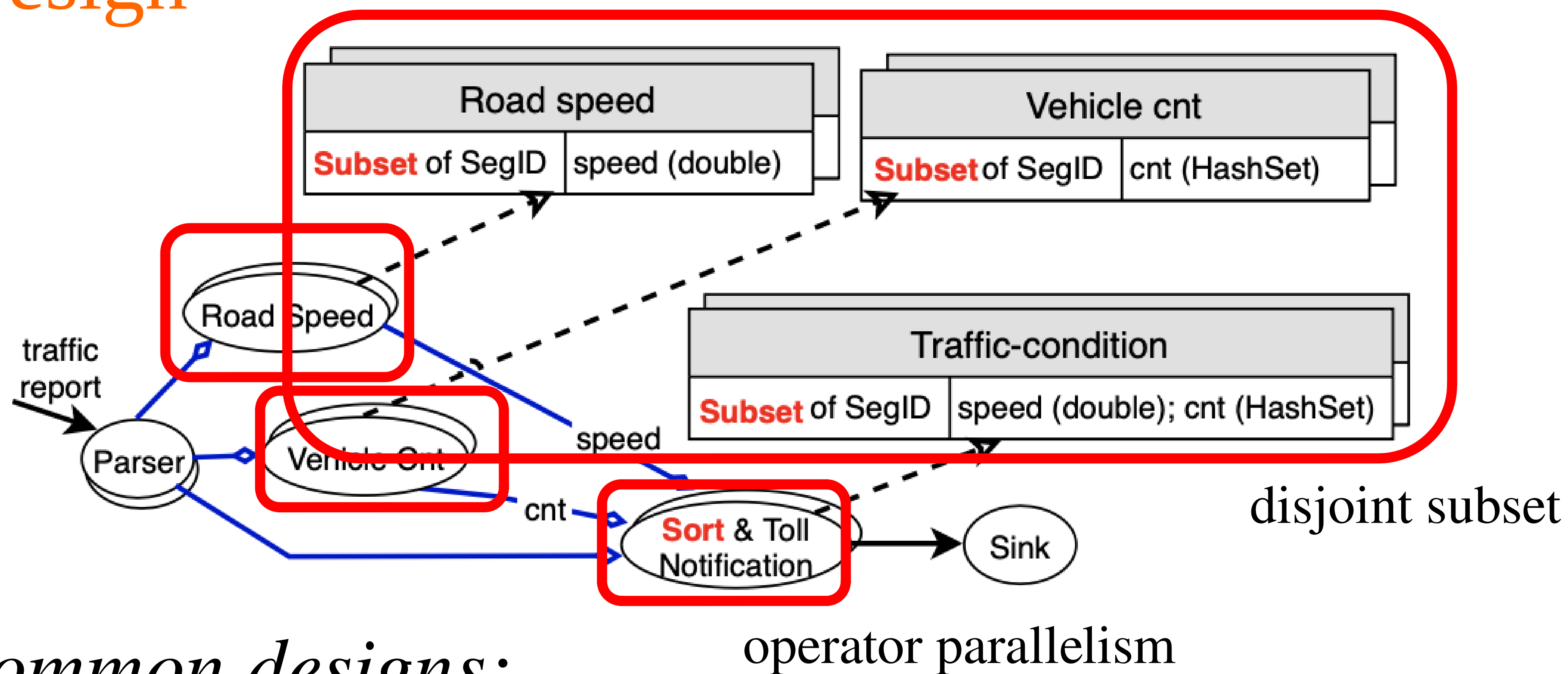


Toll Processing: computes toll based on the road congestion status

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

-- consistent stateful stream processing

The Current Common System Design

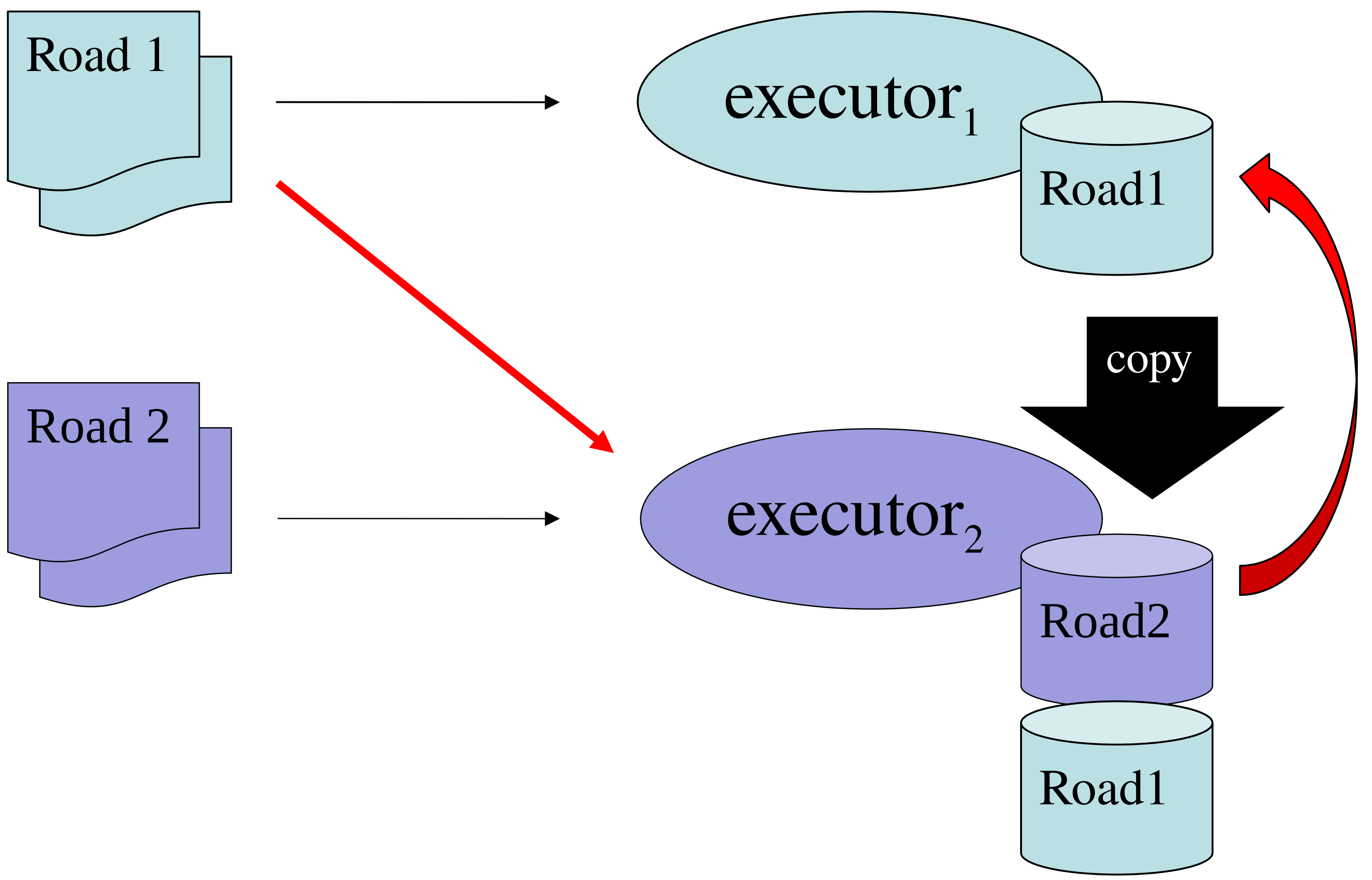


Common designs:

a) Pipelined processing with message passing

b) On-demand data parallelism

Key-based Stream Partition

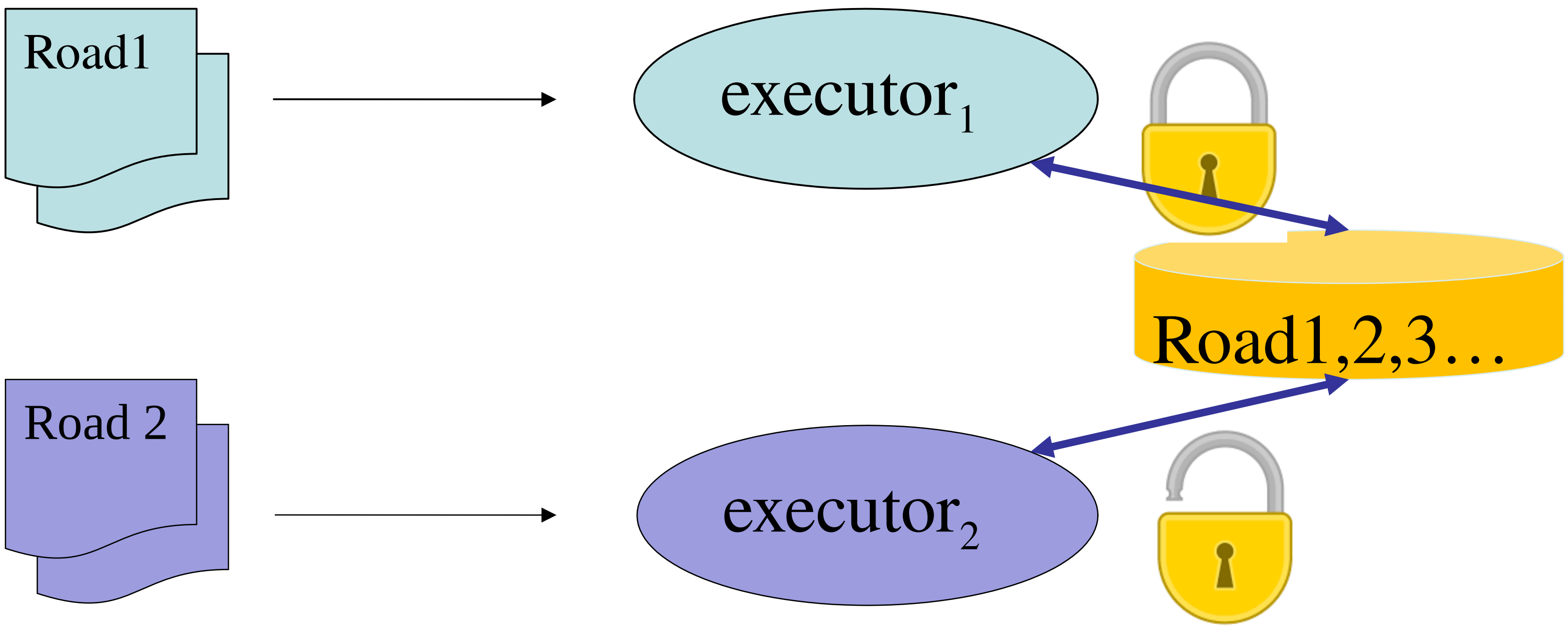


No Conflict!^^

Pay extra penalty if road1 is empty!

Can not efficiently handle general case

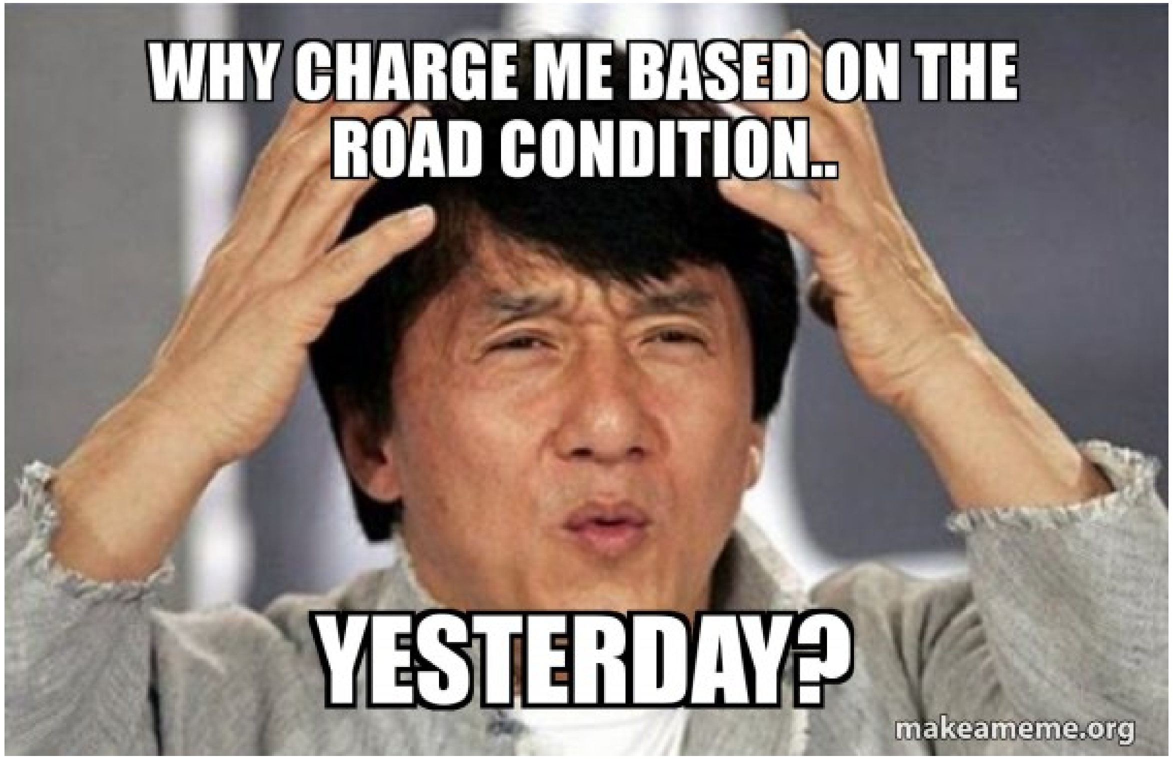
Lock-based State Sharing



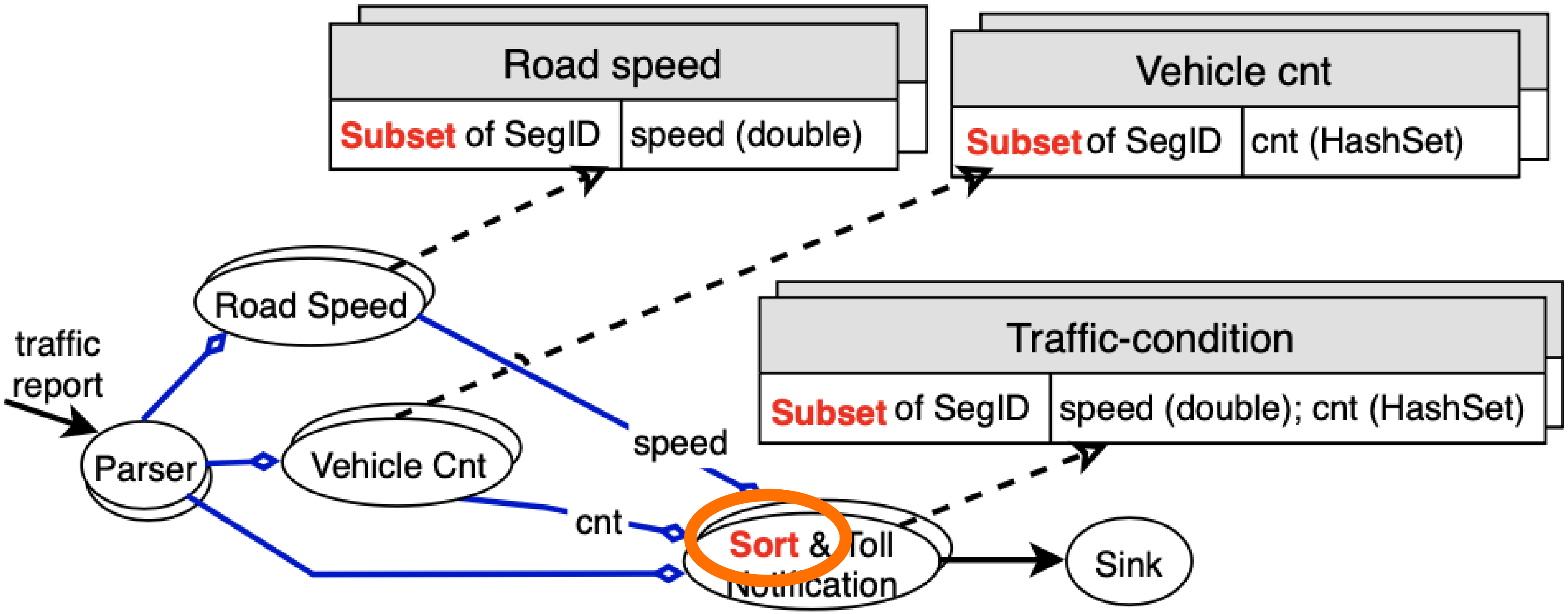
Lead to poor performance

Access Ordering

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



Access Ordering



Common workaround:

- ***Buffer and sort***

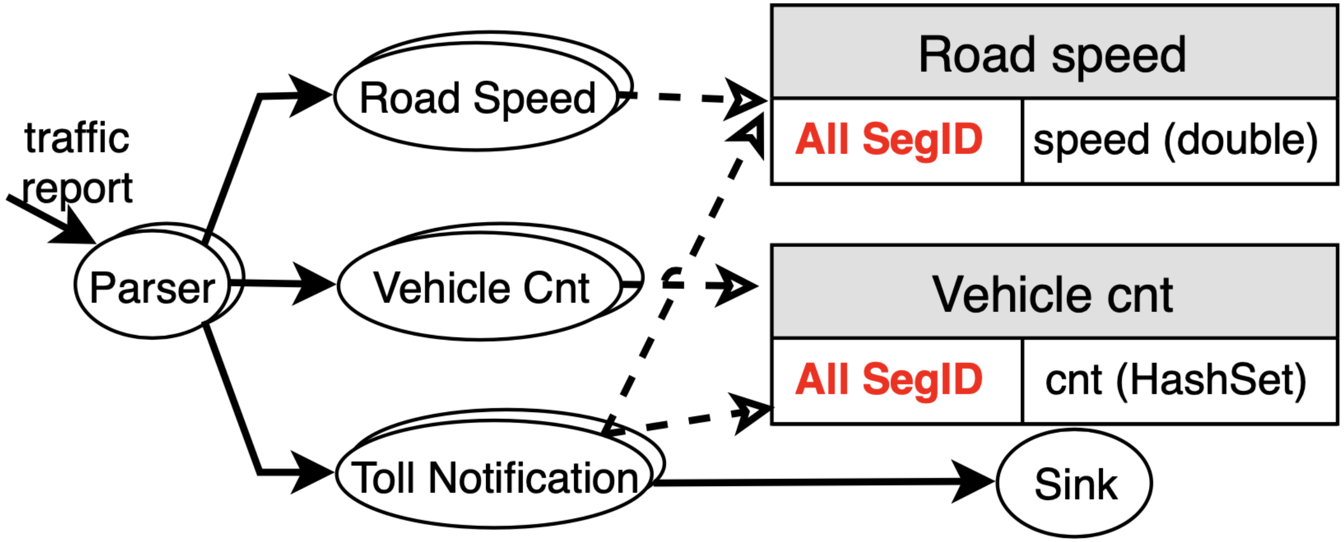
Lead to poor performance

NUS National University of Singapore

Outline

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

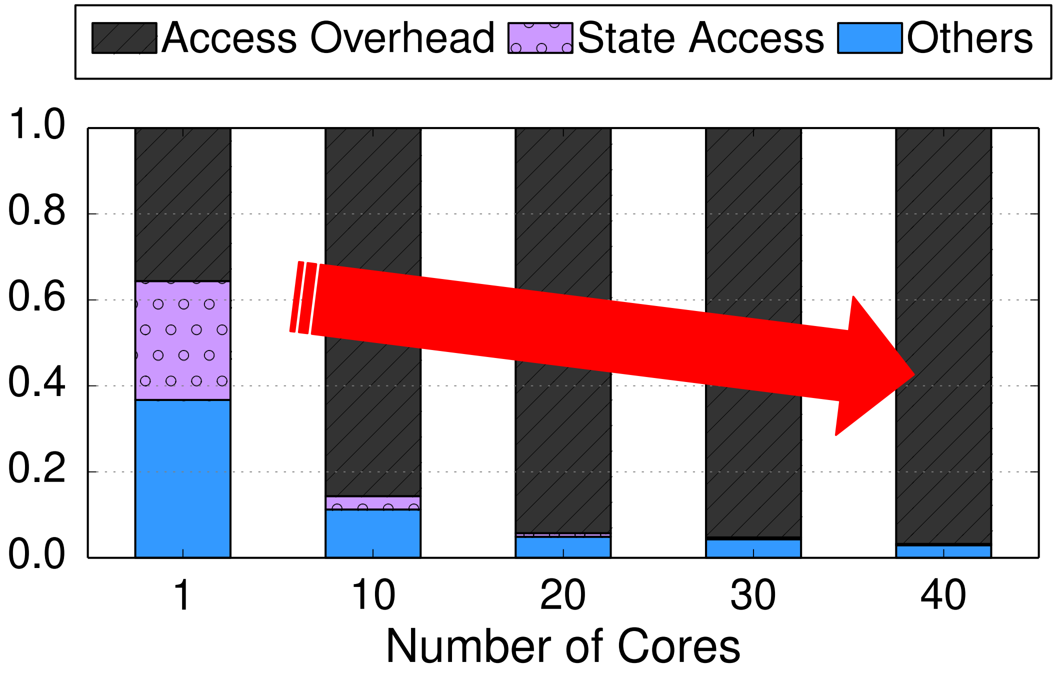
Consistent Stateful Stream Processing



Employing <u>transactional schematics</u>.

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

Existing Solutions Revisited



What if we use existing design[1]?

- Severe lock contention

A new solution for scaling concurrent state access is needed!

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

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 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

PreProcess
- Initial process on an input event

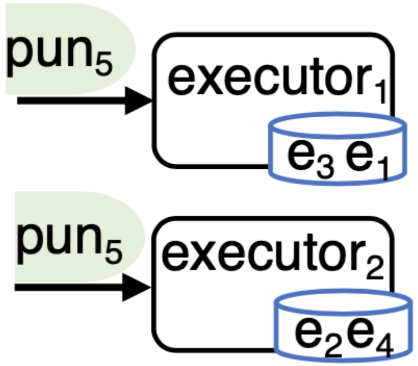
StateAccess
- Access to shared states

PostProcess
- Process on an input event with reference to obtained state values

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

Let's postpone it.

Design 1) Dual-Mode Scheduling



(a)PreProcess

Compute Mode State Access Mode Compute Mode

Pros: Enlarging inter-event parallelism opportunities.

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

Design 2) Dynamic Restructuring Execution

For each state transaction:

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

Road 1 Road 2

Outline

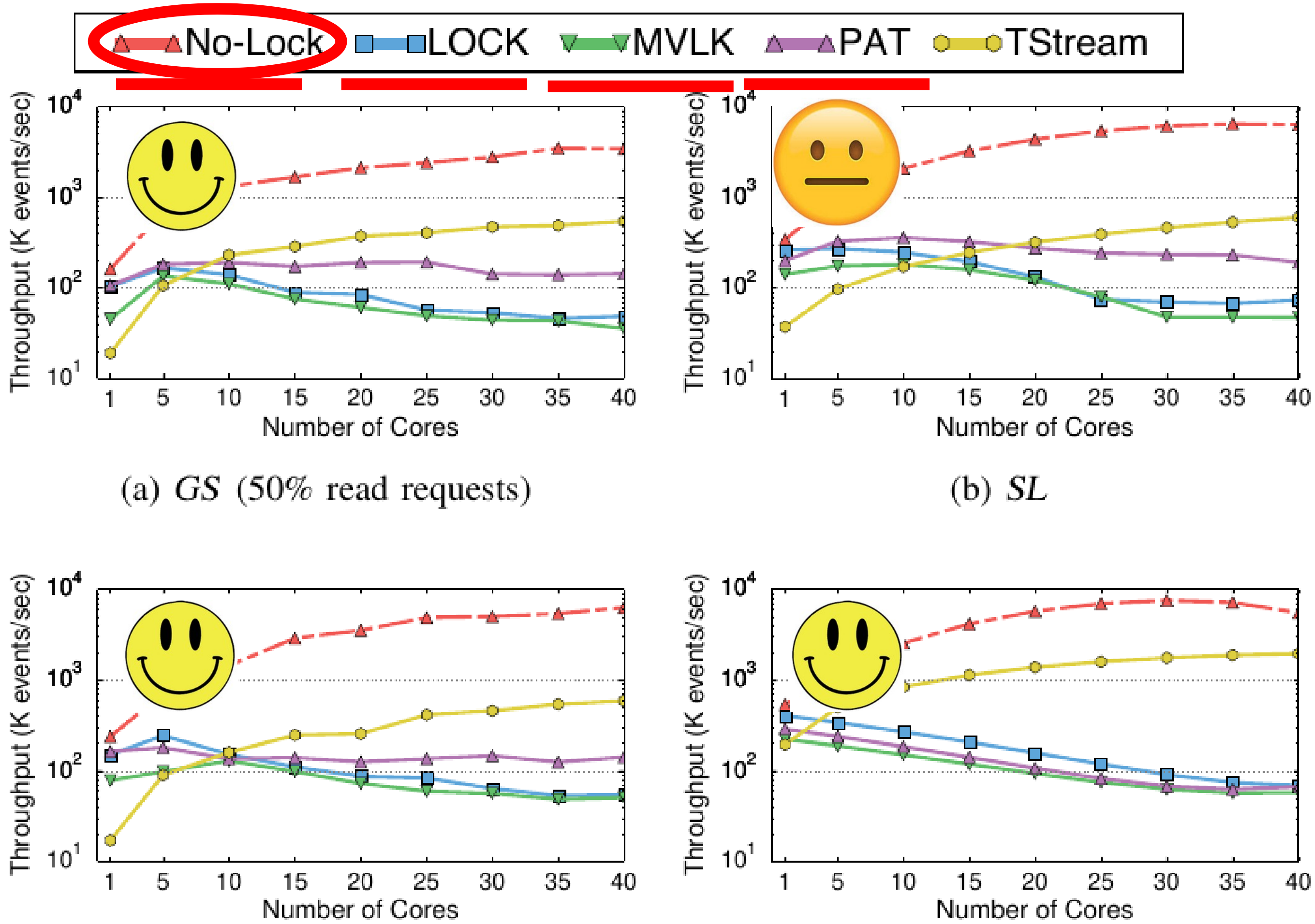
- Motivation
- State-of-the-art
- Our Approach
- **Experimental Results**

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*

(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

- [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

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

① Ordering-lock-based approach (LOCK).

② Multi-versioning-based approach (MVLK).

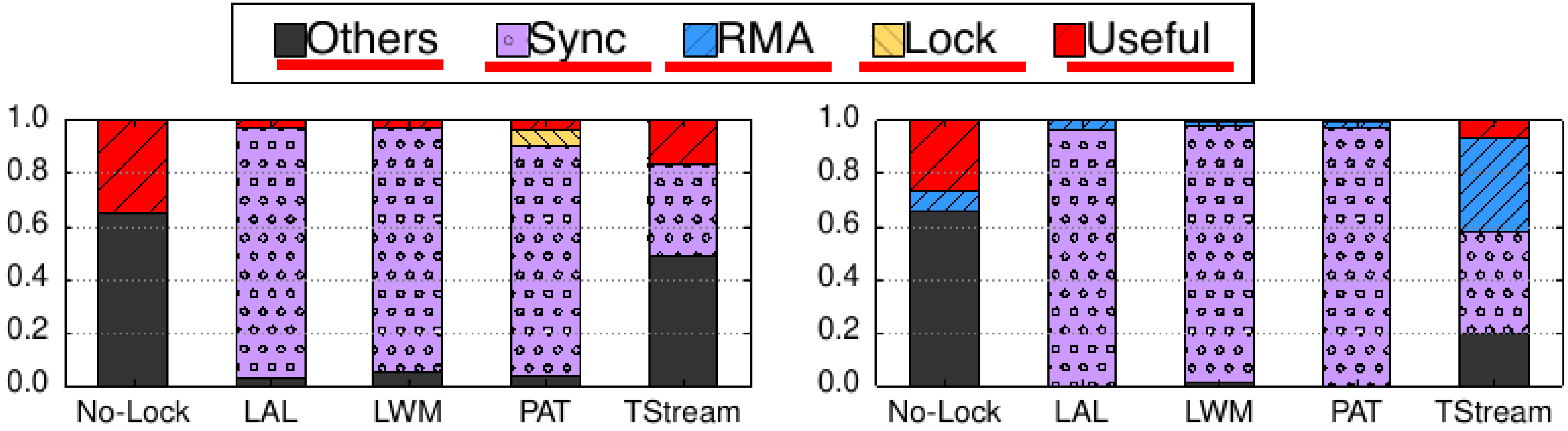
③ Static-partition-based approach (PAT).

- 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

(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.

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