Linearizability via Order Extension Theorems

Ana Sokolova UNIVERSITY of SALZBURG

Dagstuhl, 25.5.2018

- Part I: Concurrent data structures
  correctness and performance

structure and power

via semantic relaxations

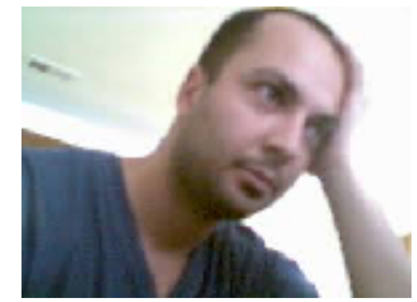
- Part II: Order extension results for
  verifying linearizability

Concurrent Data Structures
Correctness and Relaxations


Hannes Payer
Google


Tom Henzinger
IST AUSTRIA


Christoph Kirsch
UNIVERSITY of SALZBURG


Ali Sezgin UNIVERSITY OF CAMBRIDGE


Andreas Haas Google

Michael Lippautz

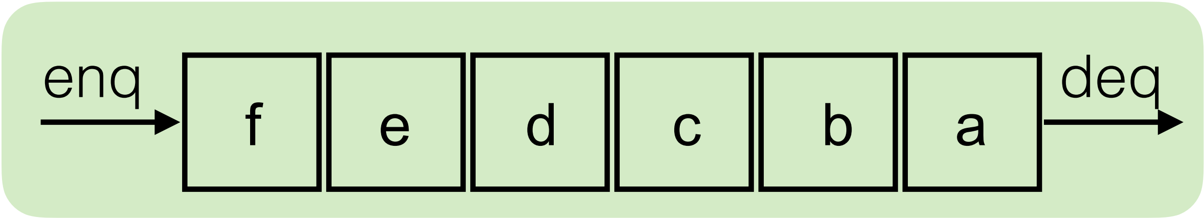

Andreas Holzer
Google


Helmut Veith TU WIEN

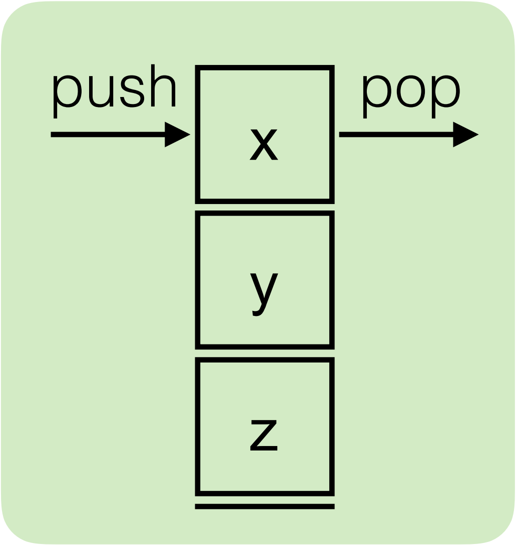
RiSE
Rigorous Systems Engineering

Data structures

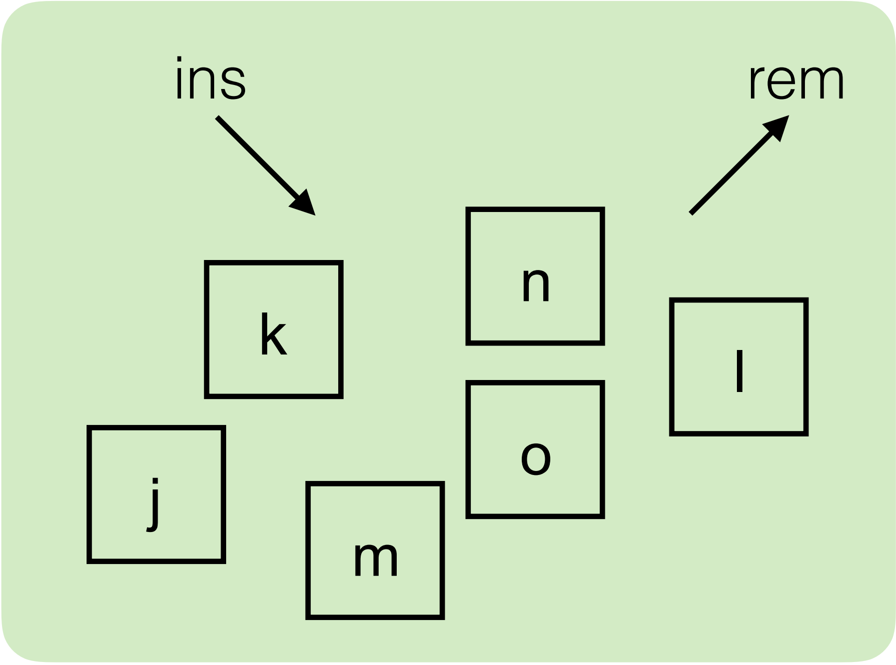
- Queue FIFO



- Stack LIFO

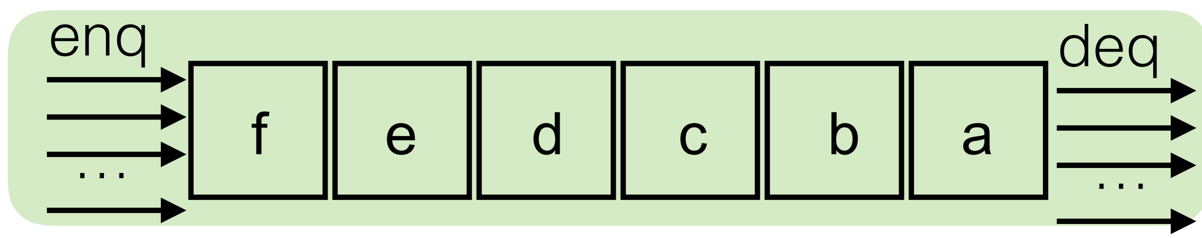


- Pool unordered

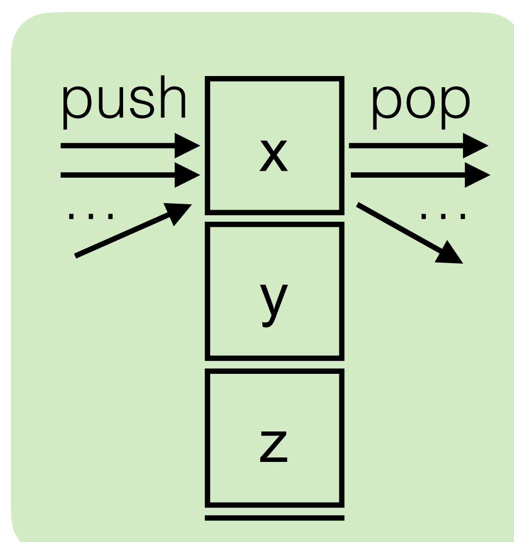


Concurrent data structures

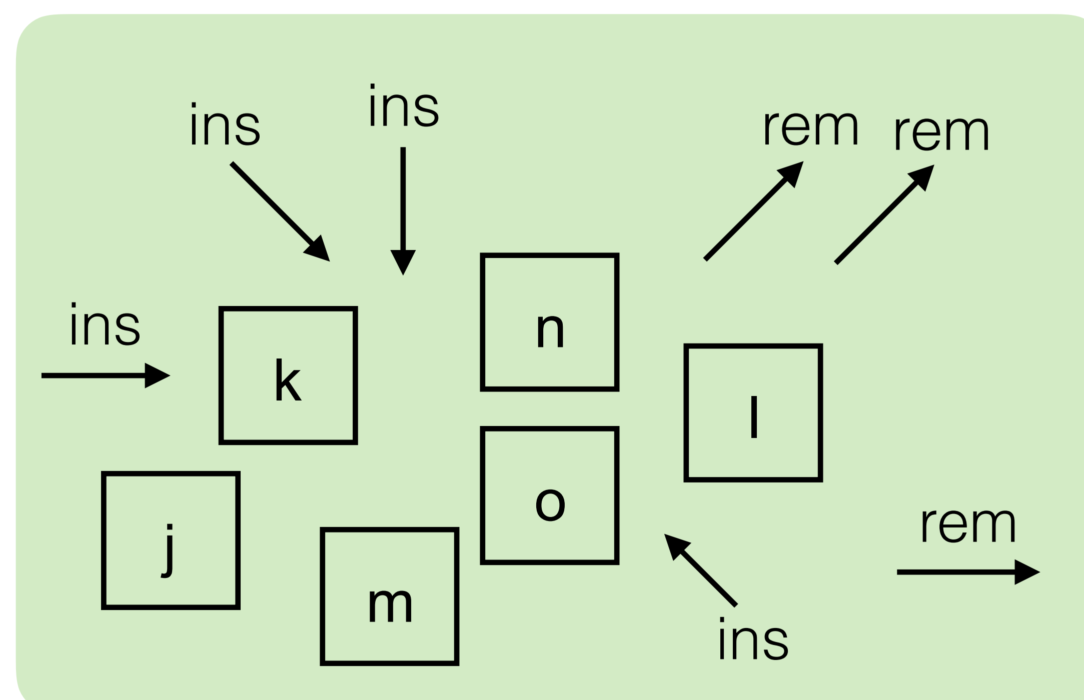
- Queue FIFO



- Stack LIFO



- Pool unordered



Semantics of concurrent data structures

t1:	enq(2)	deq(1)
t2:	enq(1)	deq(2)

e.g. queues

- Sequential specification = set of legal sequences

e.g. queue legal sequence
enq(1)enq(2)deq(1)deq(2)

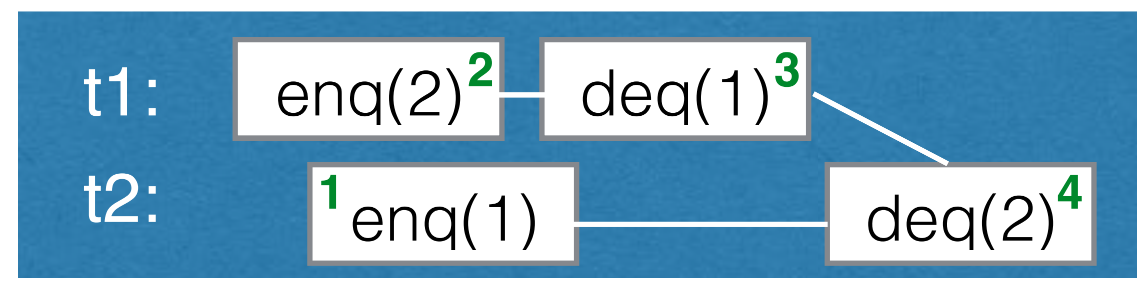
- Consistency condition = e.g. linearizability / sequential consistency

e.g. the concurrent history above is a linearizable queue concurrent history

Consistency conditions

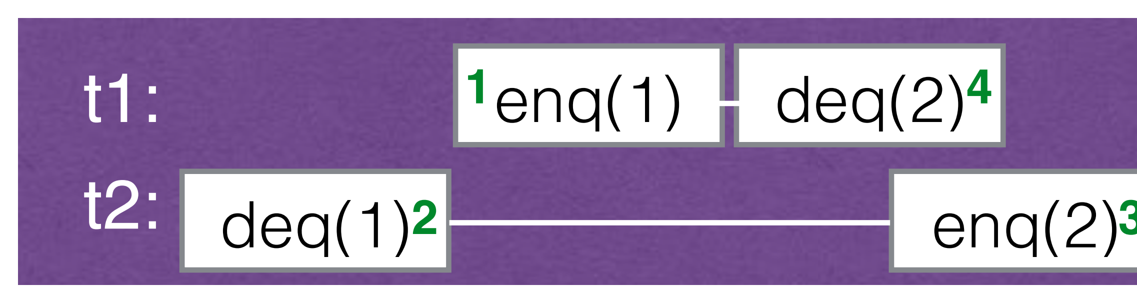
there exists a legal sequence that preserves precedence order

Linearizability [Herlihy,Wing '90]



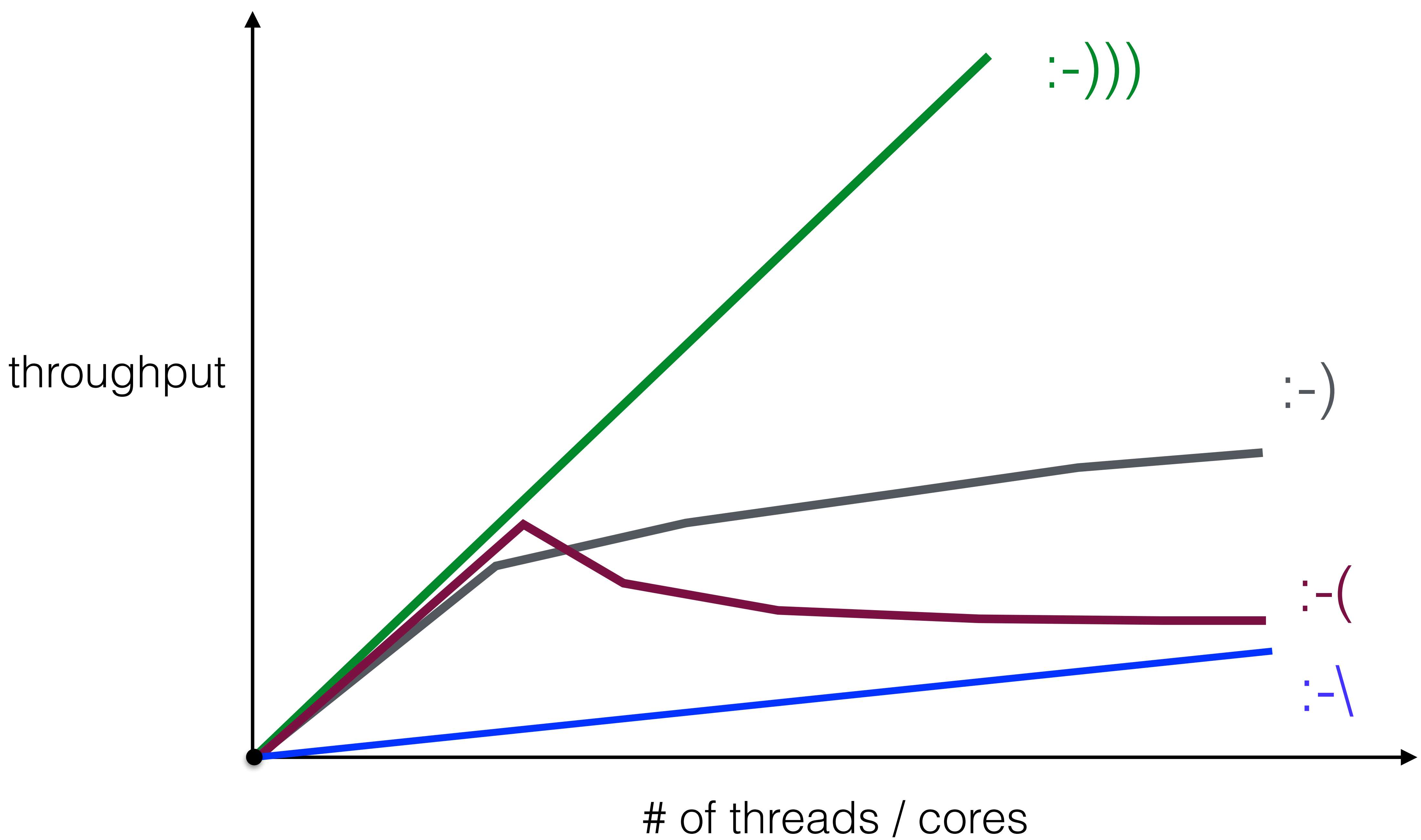
consistency is about extending partial orders to total orders

Sequential Consistency [Lamport'79]



there exists a legal sequence that preserves per-thread precedence (program order)

Performance and scalability



Relaxations allow trading

correctness
for
performance

provide the potential for better-performing implementations

Relaxing the Semantics

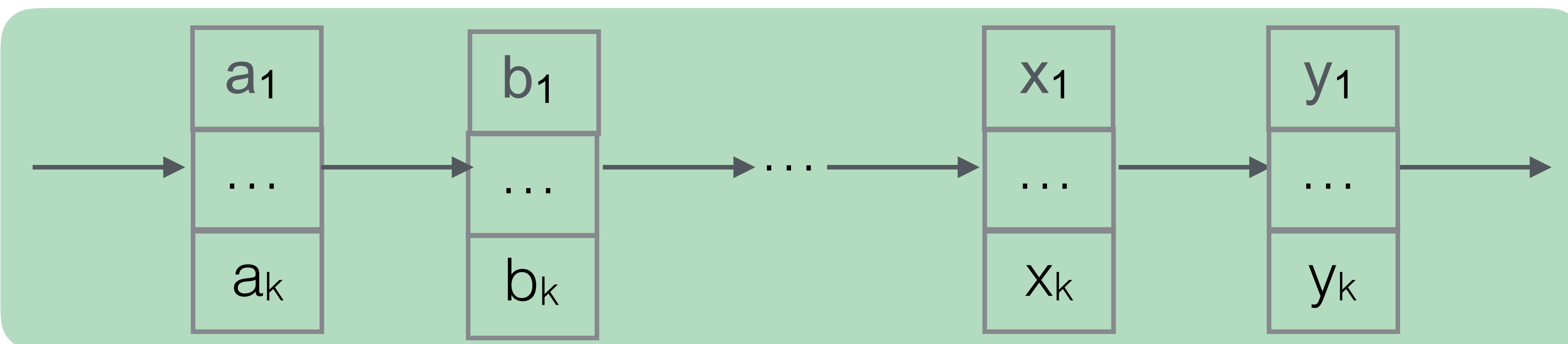
Quantitative relaxations
Henzinger, Kirsch, Payer, Sezgin,S. POPL13

- Sequential specification = set of legal sequences
- Consistency condition = e.g. linearizability / sequential consistency

Local linearizability
Haas, Henzinger, Holzer,…, S, Veith CONCUR16

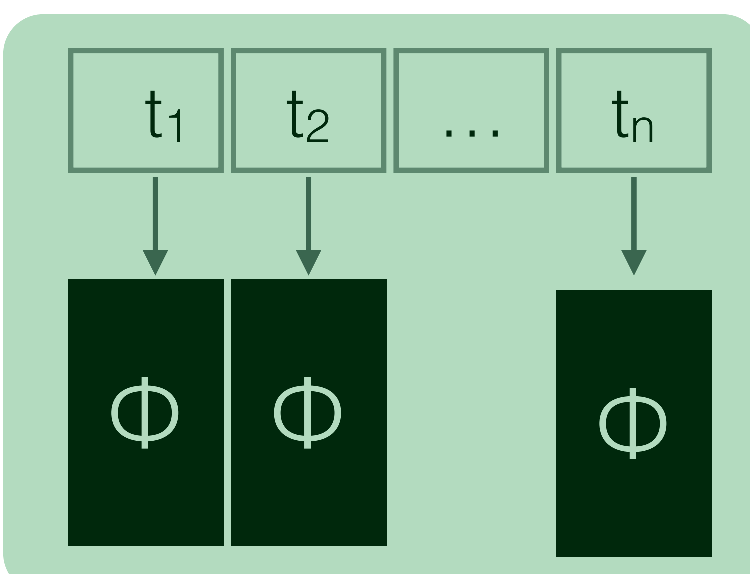
Lead to scalable implementations

e.g. k-FIFO, k-Stack



k-out-of-order queue

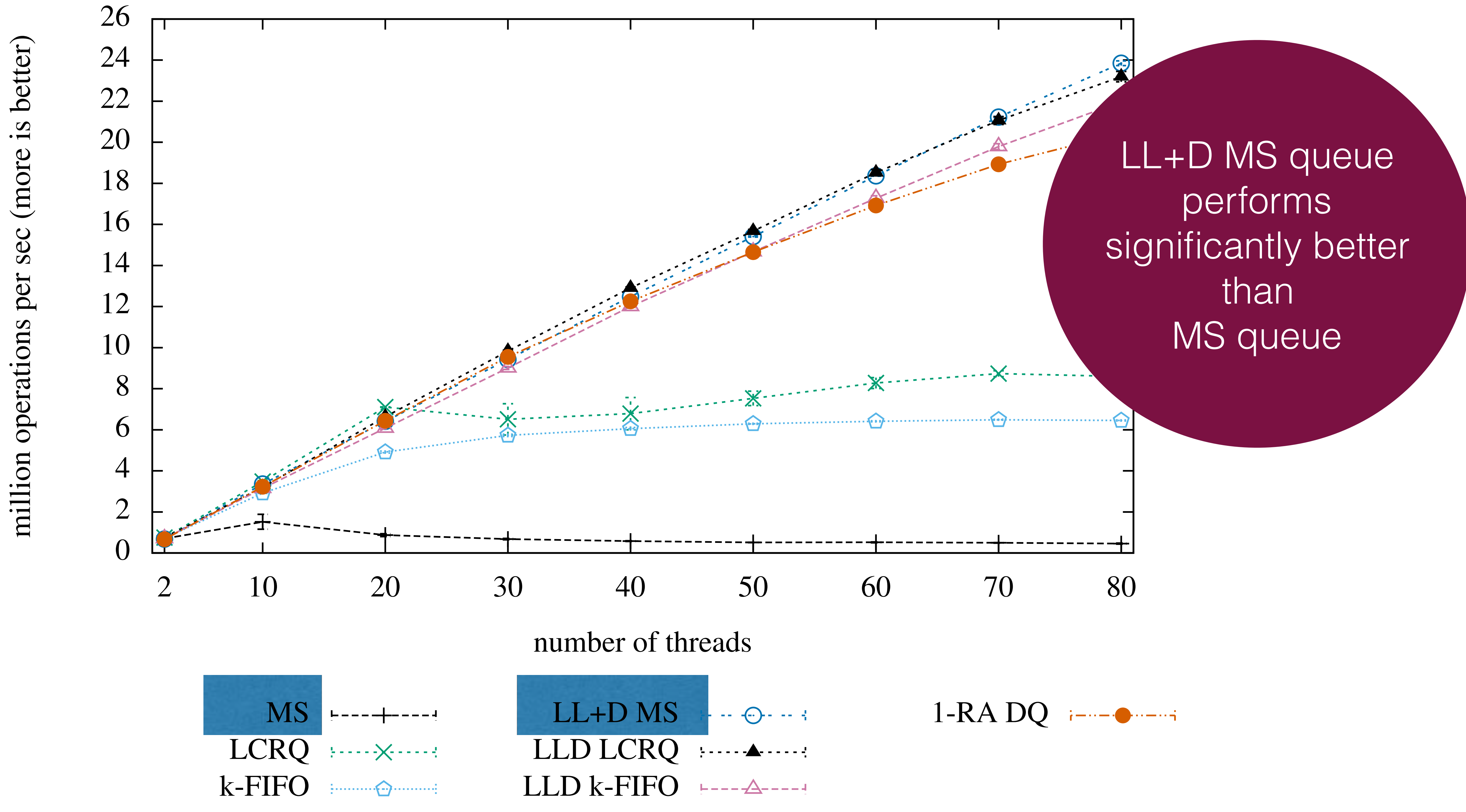
locally linearizable distributed implementation



local inserts / global removes

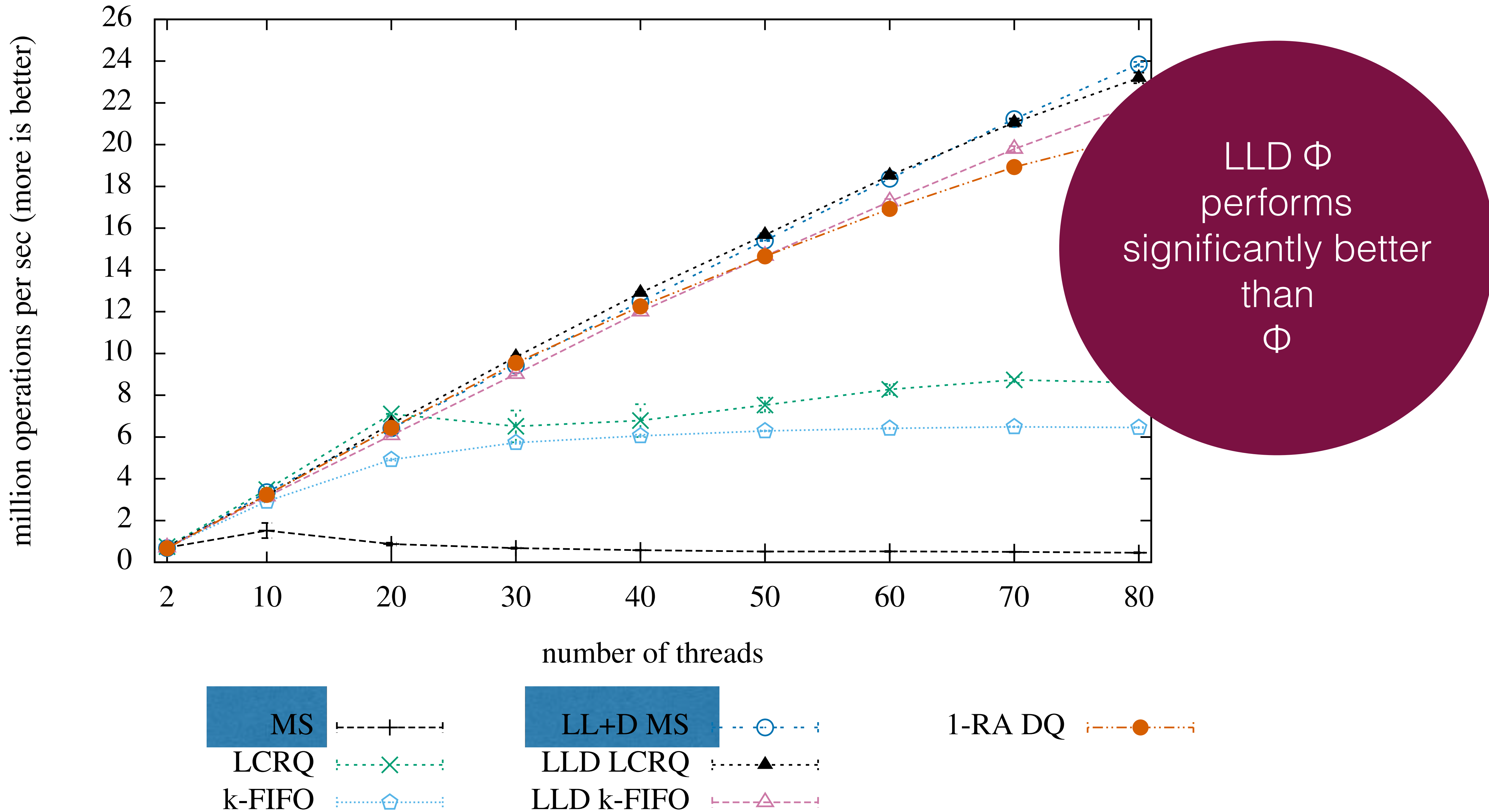
LLD Φ
LL+D Φ

Performance



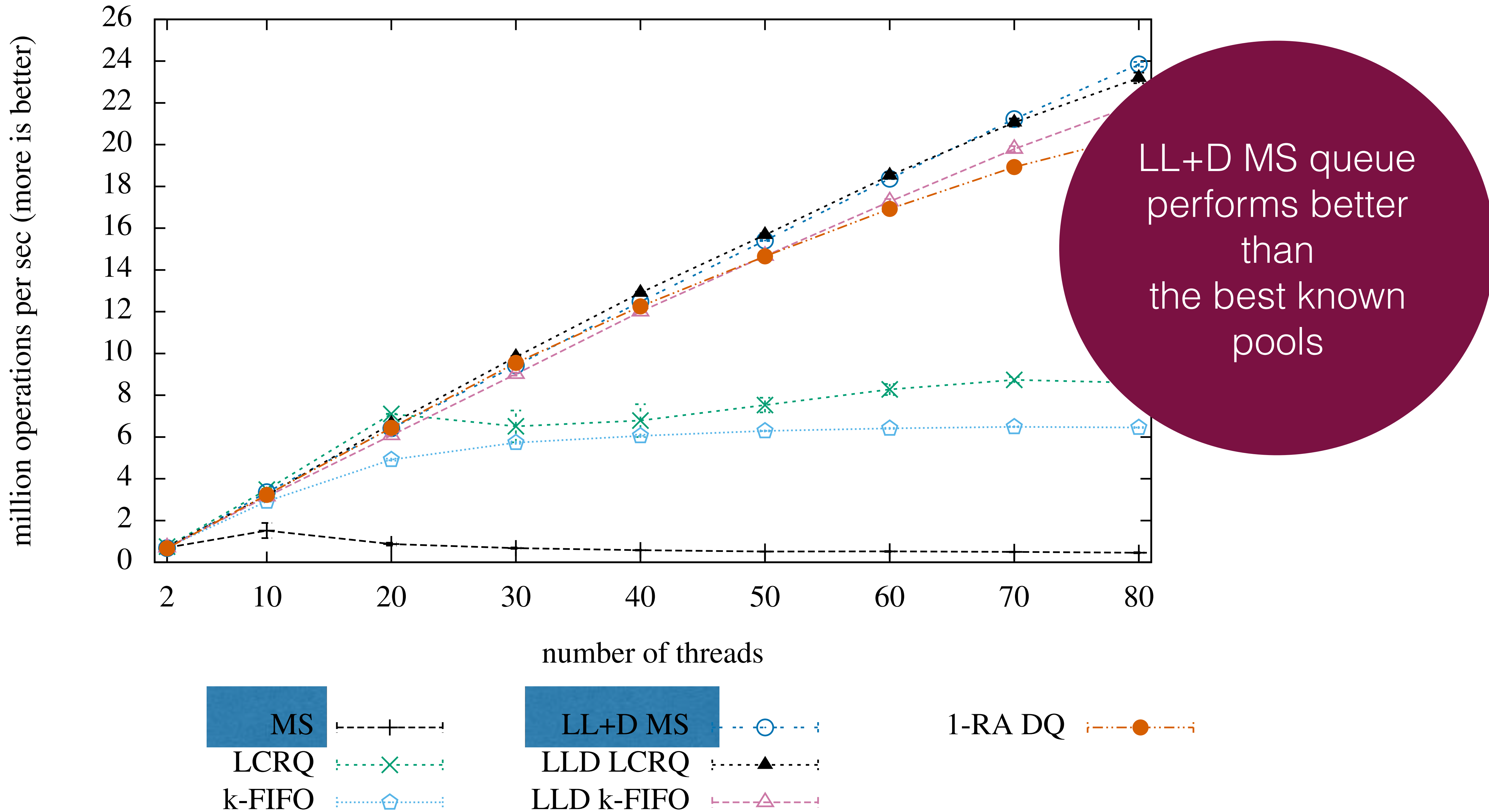
(a) Queues, LL queues, and "queue-like" pools

Performance



(a) Queues, LL queues, and "queue-like" pools

Performance



(a) Queues, LL queues, and "queue-like" pools

Linearizability via Order Extension Theorems

joint work with

Harald Woracek

foundational results
for
verifying linearizability

Inspiration

As well as
Reducing Linearizability to
State Reachability
[Bouajjani, Emmi, Enea, Hamza]
ICALP15 + …

Queue sequential specification (axiomatic)

s is a legal queue sequence

iff

1. **s** is a legal pool sequence, and
2. $enq(x) <_{\mathbf{s}} enq(y) \;\wedge\; deq(y) \in \mathbf{s} \;\Rightarrow\; deq(x) \in \mathbf{s} \;\wedge\; deq(x) <_{\mathbf{s}} deq(y)$

Queue linearizability (axiomatic)

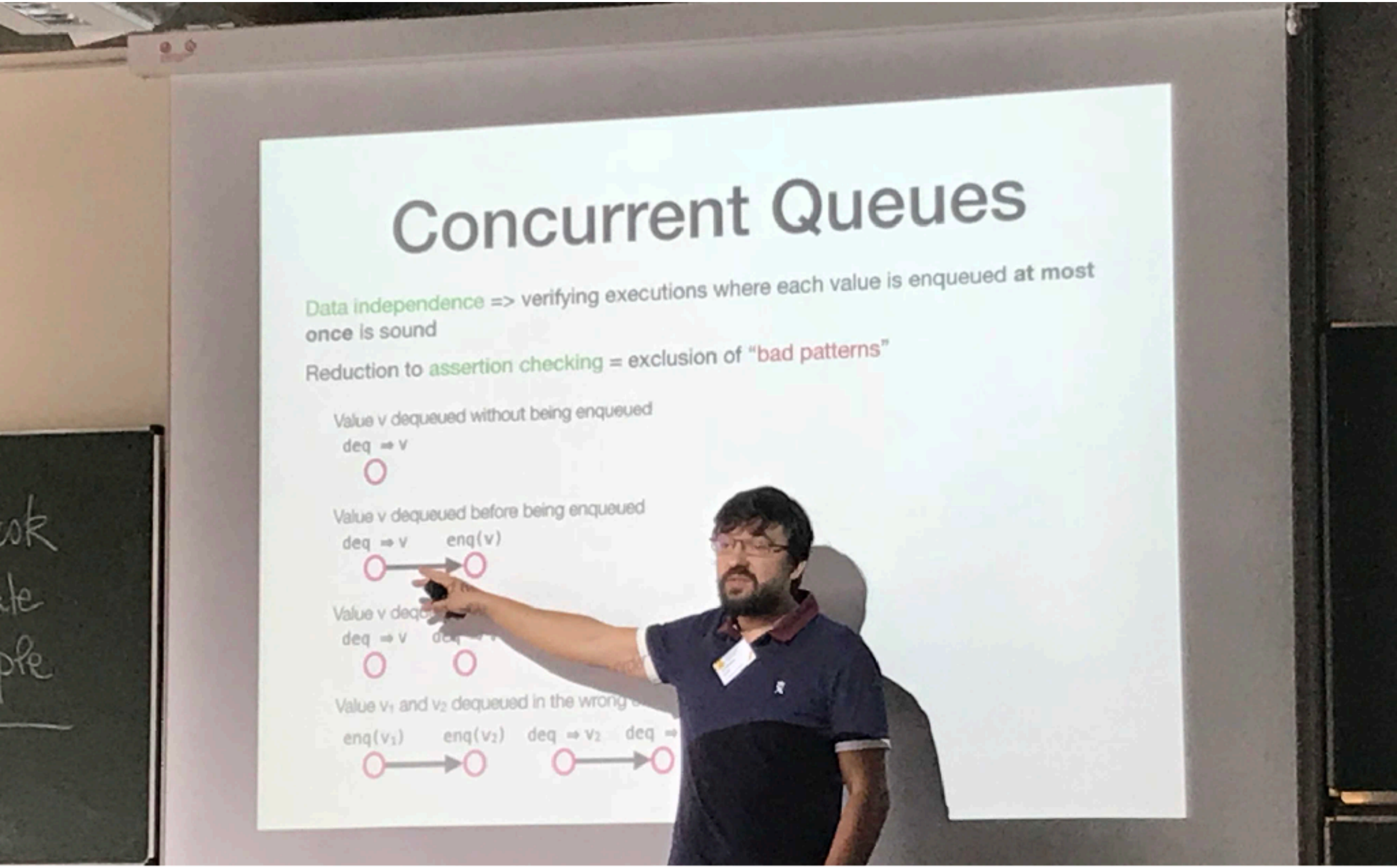
Henzinger, Sezgin, Vafeiadis CONCUR13

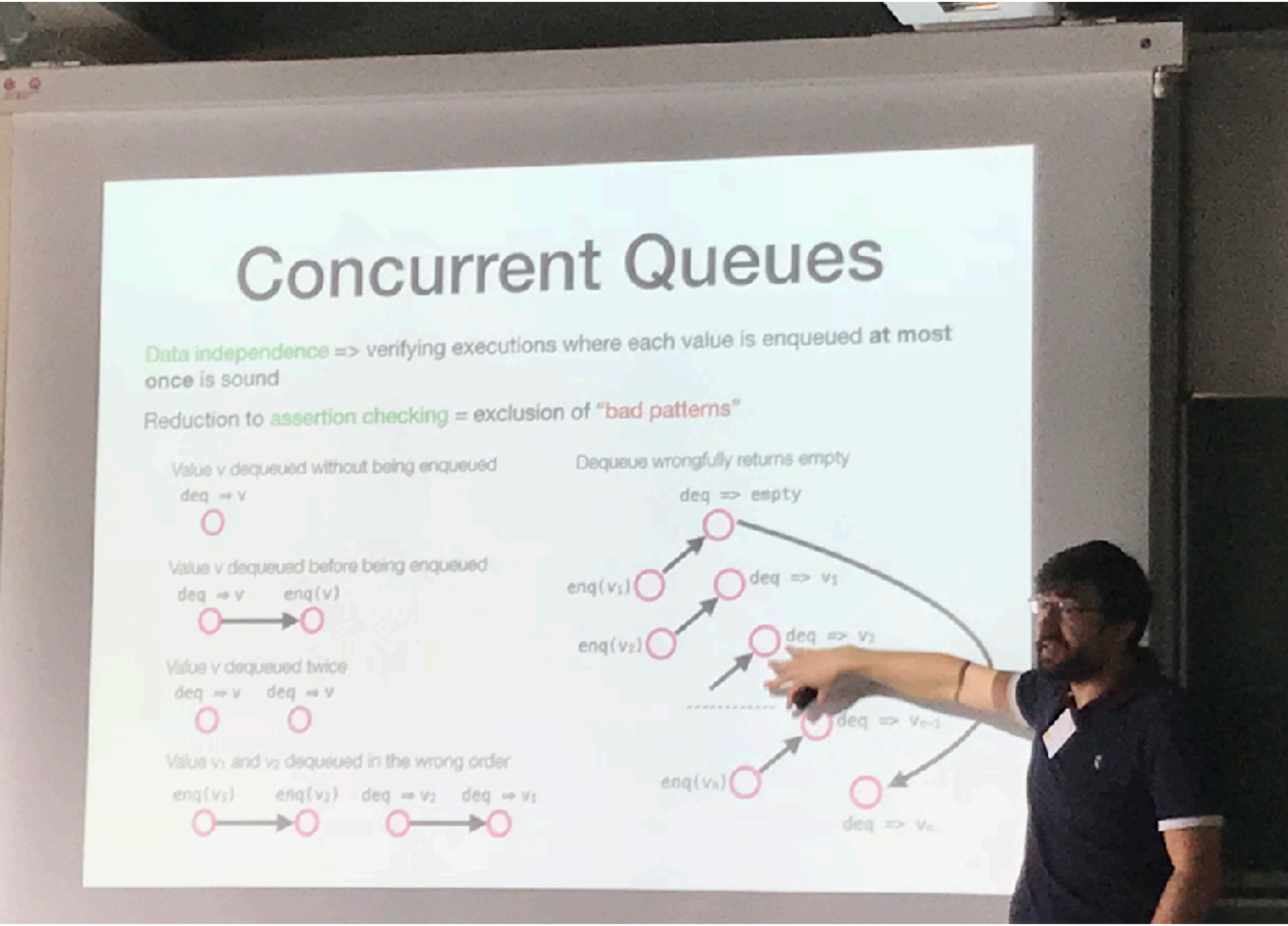
h is queue linearizable

iff

1. **h** is pool linearizable, and
2. $enq(x) <_{\mathbf{h}} enq(y) \;\wedge\; deq(y) \in \mathbf{h} \;\Rightarrow\; deq(x) \in \mathbf{h} \;\wedge\; deq(y) \not<_{\mathbf{h}} deq(x)$

precedence order


Concurrent Queues
Data independence => verifying executions where each value is enqueued at most once is sound
Reduction to assertion checking = exclusion of "bad patterns"
Value v dequeued without being enqueued
deq ⇒ v
Value v dequeued before being enqueued
deq ⇒ v
enq(v)
deq ⇒ v
enq(v1)
enq(v2)
deq ⇒ v2


Concurrent Queues
Data independence => verifying executions where each value is enqueued at most once is sound
Reduction to assertion checking = exclusion of "bad patterns"
Value v dequeued without being enqueued
deq ⇒ v
Value v dequeued before being enqueued
deq ⇒ v
enq(v)
Value v dequeued twice
deq ⇒ v
deq ⇒ v
Value v1 and v2 dequeued in the wrong order
enq(v1)
enq(v2)
deq ⇒ v2
deq ⇒ v1
Dequeue wrongfully returns empty
deq => empty
enq(v1)
deq => v1
enq(v2)
deq => v2
deq => vn-1
enq(vn)
deq => vn

Linearizability verification

Data structure

- signature Σ - set of method calls including data values
- sequential specification $S \subseteq \Sigma^*$, prefix closed

identify sequences with total orders

Sequential specification via violations

Extract a set of violations V, relations on Σ, such that **s** $\in$ S iff **s** has no violations

it is easy to find a large CV,
but difficult to find a small representative

$\mathcal{P}(\mathbf{s}) \cap V = \varnothing$

Linearizability verification

Find a set of violations CV such that: every interval order with no CV violations extends to a total order with no V violations.

concurrent history

we build CV iteratively from V

legal sequence

Pool without empty removals

Pool sequential specification (axiomatic)

s is a legal pool (without empty removals) sequence
iff
1. $\text{rem}(x) \in \mathbf{s} \quad \Rightarrow \quad \text{ins}(x) \in \mathbf{s} \;\wedge\; \text{ins}(x) <_{\mathbf{s}} \text{rem}(x)$

V violations
$\text{rem}(x) <_{\mathbf{s}} \text{ins}(x)$

Pool linearizability (axiomatic)

h is pool (without empty removals) linearizable
iff
1. $\text{rem}(x) \in \mathbf{h} \quad \Rightarrow \quad \text{ins}(x) \in \mathbf{h} \;\wedge\; \text{rem}(x) \nless_{\mathbf{h}} \text{ins}(x)$

CV violations
= V violations

Queue without empty removals

Queue sequential specification (axiomatic)

s is a legal queue (without empty removals) sequence
iff

1. $deq(x) \in \mathbf{s} \Rightarrow enq(x) \in \mathbf{s} \wedge enq(x) <_{\mathbf{s}} deq(x)$
2. $enq(x) <_{\mathbf{s}} enq(y) \wedge deq(y) \in \mathbf{s} \Rightarrow deq(x) \in \mathbf{s} \wedge deq(x) <_{\mathbf{s}} deq(y)$

V violations
$deq(x) <_{\mathbf{s}} enq(x)$
and
$enq(x) <_{\mathbf{s}} enq(y) \wedge$
$deq(y) <_{\mathbf{s}} deq(x)$

Queue linearizability (axiomatic)

h is queue (without empty removals) linearizable
iff

1. $rem(x) \in \mathbf{h} \Rightarrow ins(x) \in \mathbf{h} \wedge rem(x) \not<_{\mathbf{h}} ins(x)$
2. $enq(x) <_{\mathbf{h}} enq(y) \wedge deq(y) \in \mathbf{h} \Rightarrow deq(x) \in \mathbf{h} \wedge deq(y) \not<_{\mathbf{h}} deq(x)$

CV violations
= V violations

Pool

infinite inductive violations

Pool sequential specification (axiomatic)

s is a legal pool (with empty removals) sequence

iff

1. $\text{rem}(x) \in \mathbf{s} \quad \Rightarrow \quad \text{ins}(x) \in \mathbf{s} \;\wedge\; \text{ins}(x) <_{\mathbf{s}} \text{rem}(x)$
2. $\text{rem}(\perp) <_{\mathbf{s}} \text{rem}(x) \Rightarrow \text{rem}(\perp) <_{\mathbf{s}} \text{ins}(x) \;\wedge\; \text{ins}(x) <_{\mathbf{s}} \text{rem}(\perp) \Rightarrow \text{rem}(x) <_{\mathbf{s}} \text{rem}(\perp)$

V violations
$\text{rem}(x) <_{\mathbf{s}} \text{ins}(x)$
and
$\text{ins}(x) <_{\mathbf{s}} \text{rem}(\perp) <_{\mathbf{s}} \text{rem}(x)$

Pool linearizability (axiomatic)

h is pool (with empty removals) linearizable

iff

1. $\text{rem}(x) \in \mathbf{h} \quad \Rightarrow \quad \text{ins}(x) \in \mathbf{h} \;\wedge\; \text{rem}(x) \not<_{\mathbf{h}} \text{ins}(x)$
2. … …

infinitely many CV violations
$\text{ins}(x_1) <_{\mathbf{h}} \text{rem}(\perp) \wedge \text{ins}(x_2) <_{\mathbf{h}} \text{rem}(x_1) \wedge \ldots \wedge \text{ins}(x_{n+1}) <_{\mathbf{h}} \text{rem}(x_n) \wedge \text{rem}(\perp) <_{\mathbf{h}} \text{rem}(x_{n+1})$

It works for

- Pool without empty removals
- Queue without empty removals
- Priority queue without empty removals
- Pool
- Queue
- Priority que

Thank You !

But not yet for Stack:
infinite CV violations
without clear
inductive structure

Exploring the space of
data structures
as well as new ideas
for problematic cases