Linearizability via Order Extension Theorems

Ana Sokolova UNIVERSITY of SALZBURG

IRIF, 28.5.2018

- Part I: Concurrent data structures
  correctness and performance

structure and power

via semantic relaxations

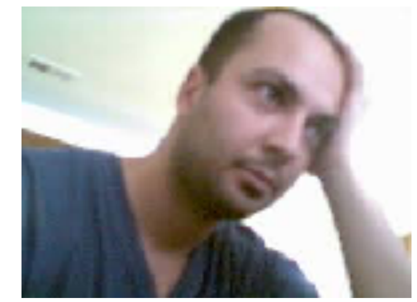
- Part II: Order extension results for
  verifying linearizability

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

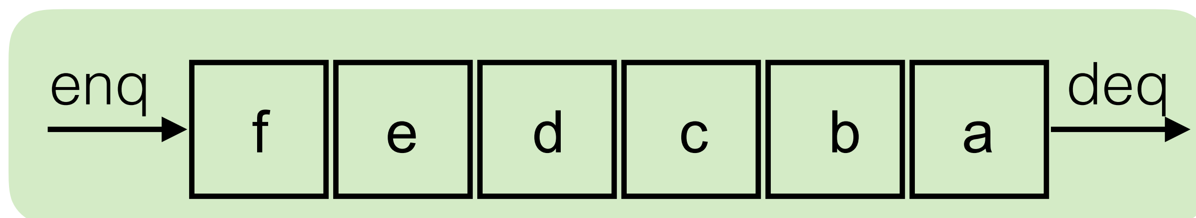
Andreas Holzer
Google

Helmut Veith TU WIEN

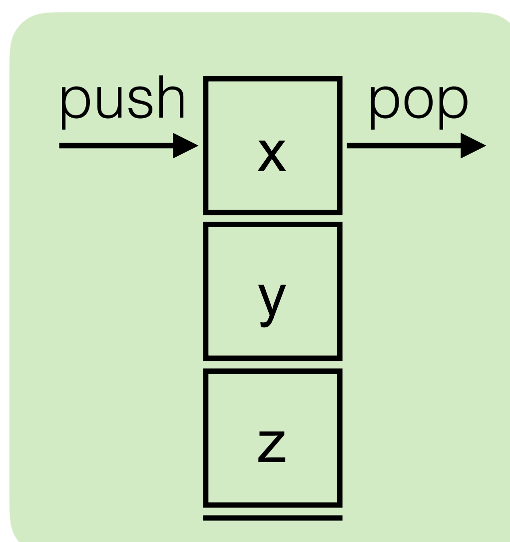
RiSE
Rigorous Systems Engineering

Data structures

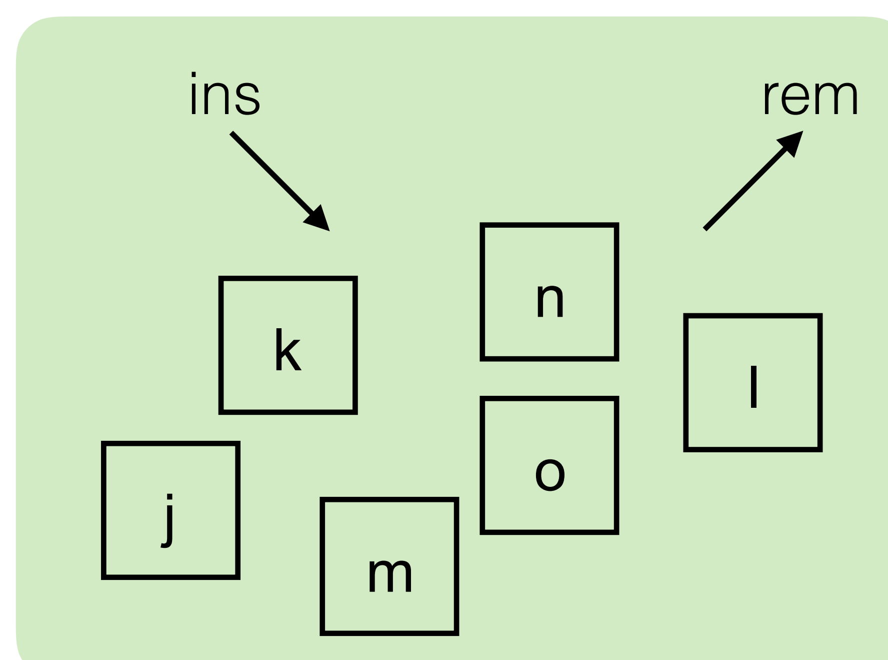
- Queue FIFO



- Stack LIFO

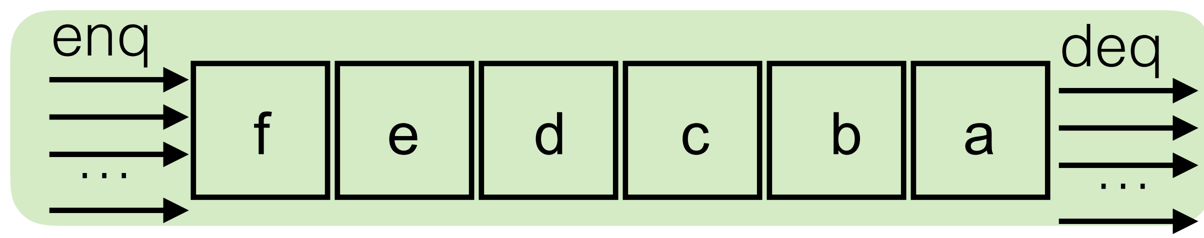


- Pool unordered

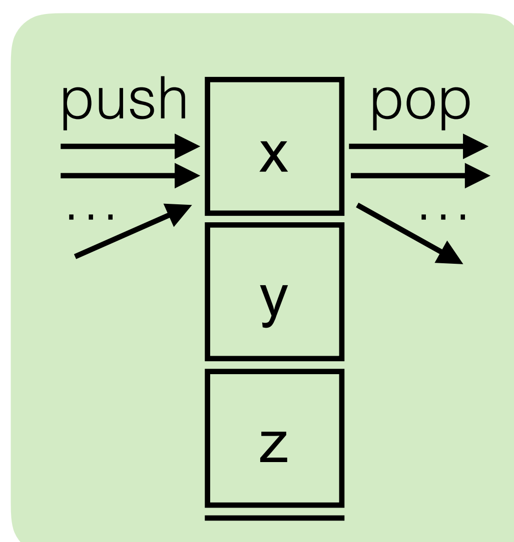


Concurrent data structures

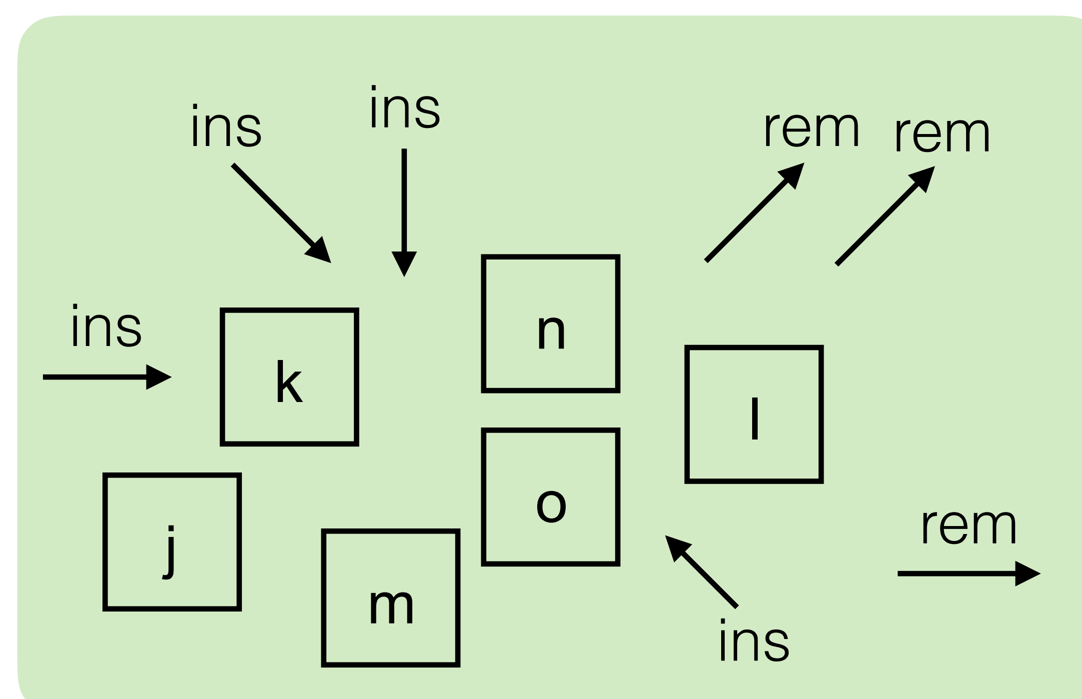
- Queue FIFO



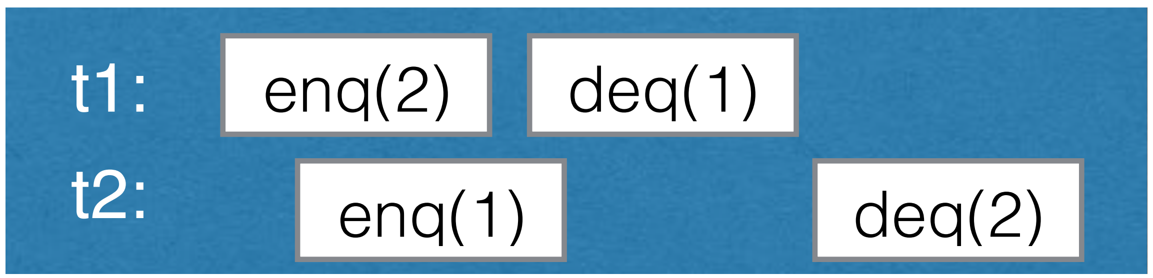
- Stack LIFO



- Pool unordered



Semantics of concurrent data structures



e.g. queues

- Sequential specification = set of legal sequences

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

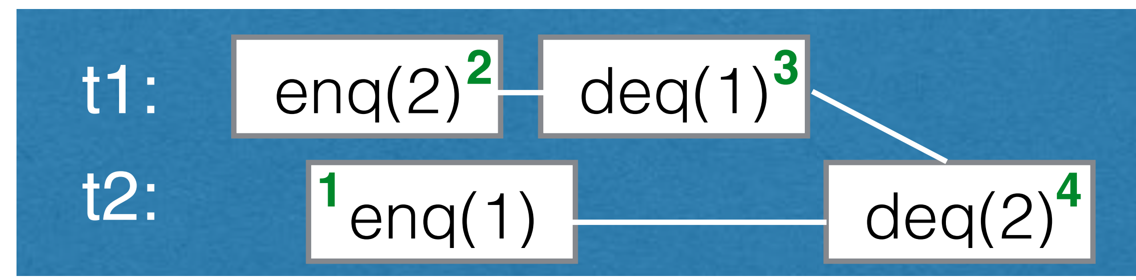
- Consistency condition = e.g. linearizability / sequential consistency

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

Consistency conditions

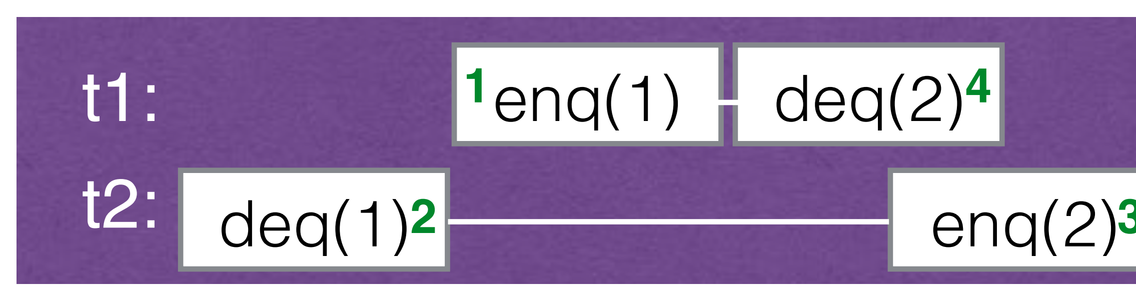
there exists a legal sequence that preserves precedence order

Linearizability [Herlihy,Wing '90]



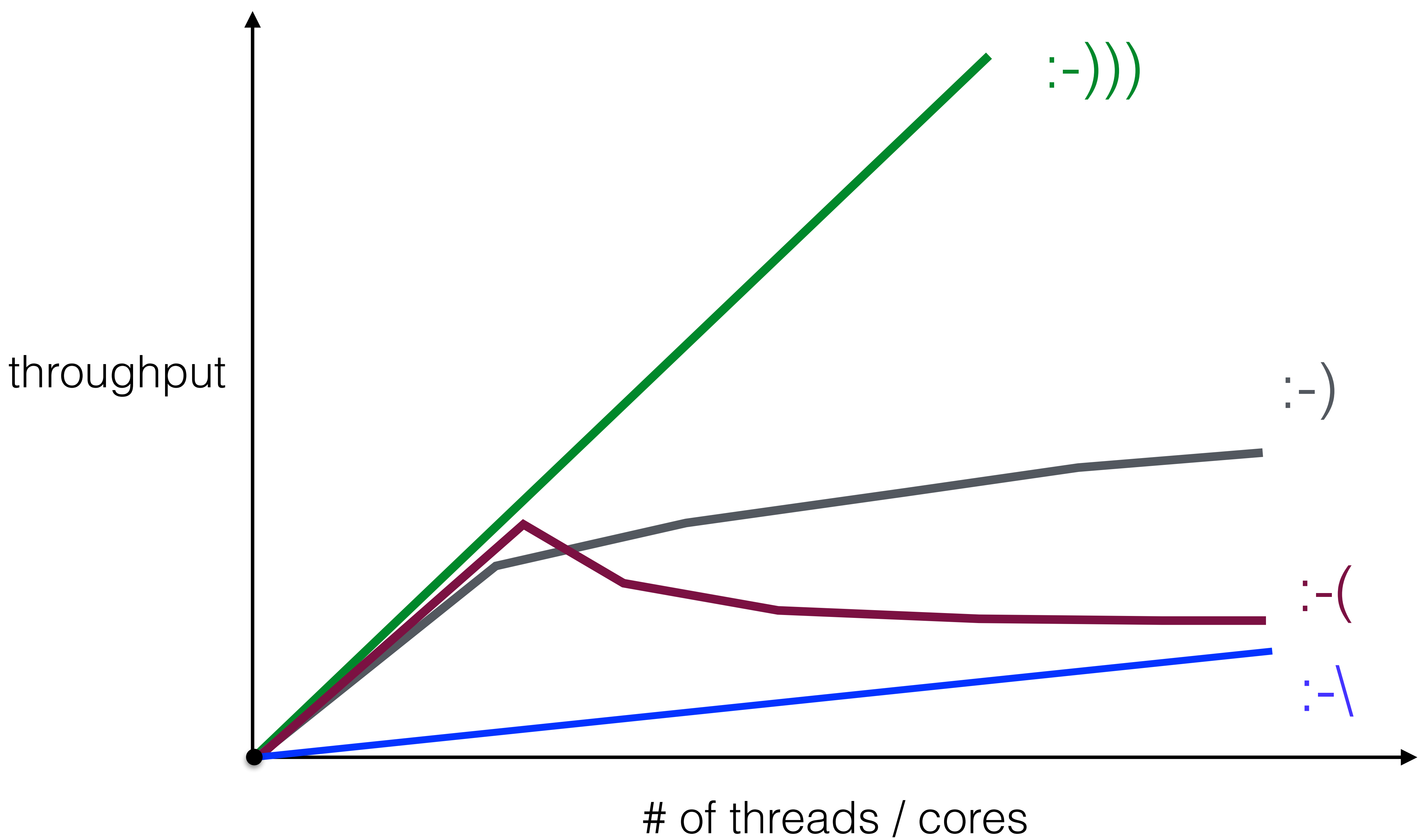
consistency is about extending partial orders to total orders

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

Relaxing the sequential specification

Quantitative relaxations (POPL13)

Goal

Stack - incorrect behavior

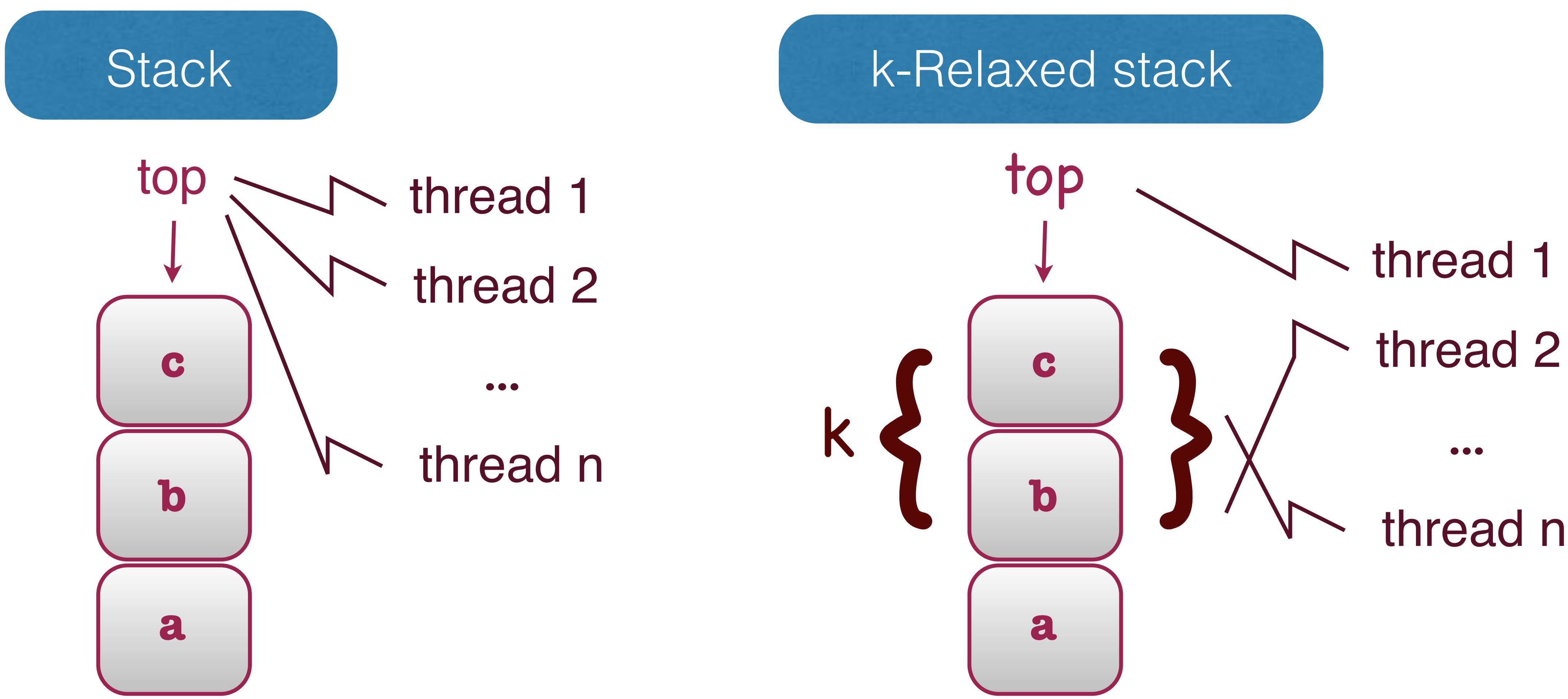
push(a)push(b)push(c)pop(a)pop(b)

- trade correctness for performance
- in a controlled way with quantitative bounds

measure the error from correct behaviour

correct in a relaxed stack
... 2-relaxed? 3-relaxed?

How can relaxing help?



What we have

- Framework

for semantic relaxations

- Generic examples

out-of-order / stuttering

- Concrete relaxation examples

stacks, queues, priority queues,.. / CAS, shared counter

- Efficient concurrent implementations

of relaxation instances

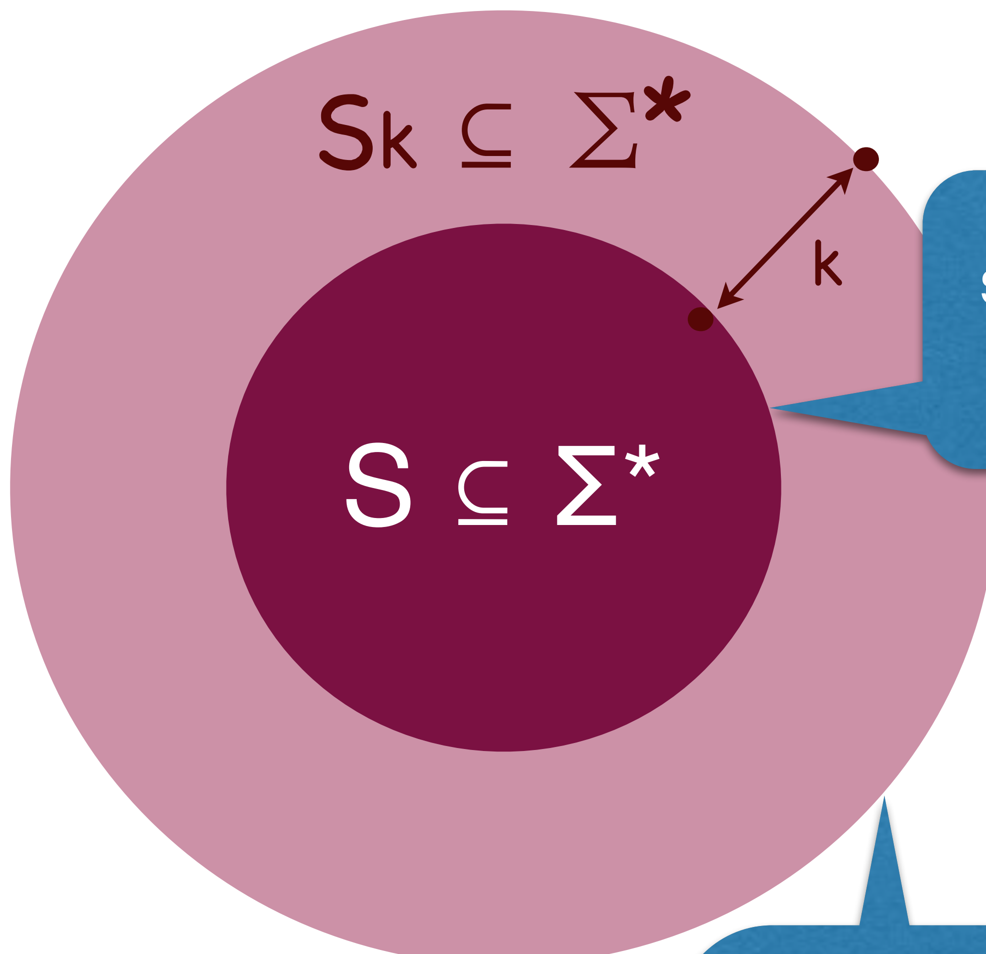
The big picture

$S \subseteq \Sigma^*$

sequential specification
legal sequences

Σ - methods with arguments

The big picture



sequential specification
legal sequences

relaxed sequential specification
sequences at distance up to k from S

Σ - methods with arguments

Relaxing the Consistency Condition

Local Linearizability (CONCUR16)

Local Linearizability main idea

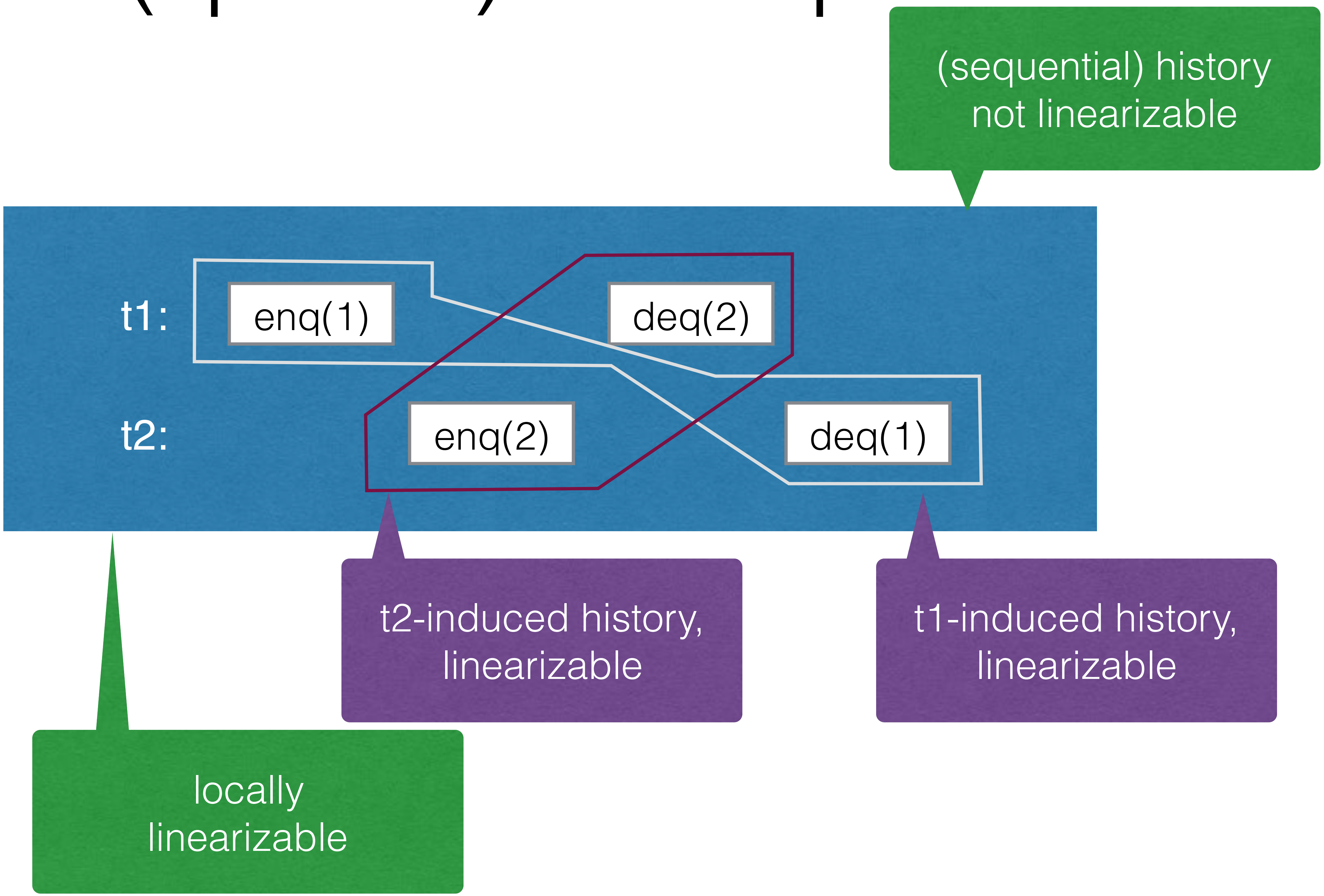
Already present in some shared-memory consistency conditions
(not in our form of choice)

- Partition a history into a set of local histories
- Require linearizability per local history

no global witness

Local sequential consistency… is also possible

Local Linearizability (queue) example



Local Linearizability (queue) definition

Queue signature $\Sigma = \{enq(x) \mid x \in V\} \cup \{deq(x) \mid x \in V\} \cup \{deq(empty)\}$

For a history **h** with a thread T, we put

$$I_T = \{enq(x)^T \in \mathbf{h} \mid x \in V\}$$

$$O_T = \{deq(x)^{T'} \in \mathbf{h} \mid enq(x)^T \in I_T\} \cup \{deq(empty)\}$$

in-methods of thread T are enqueues performed by thread T

out-methods of thread T are dequeues (performed by any thread) corresponding to enqueues that are in-methods

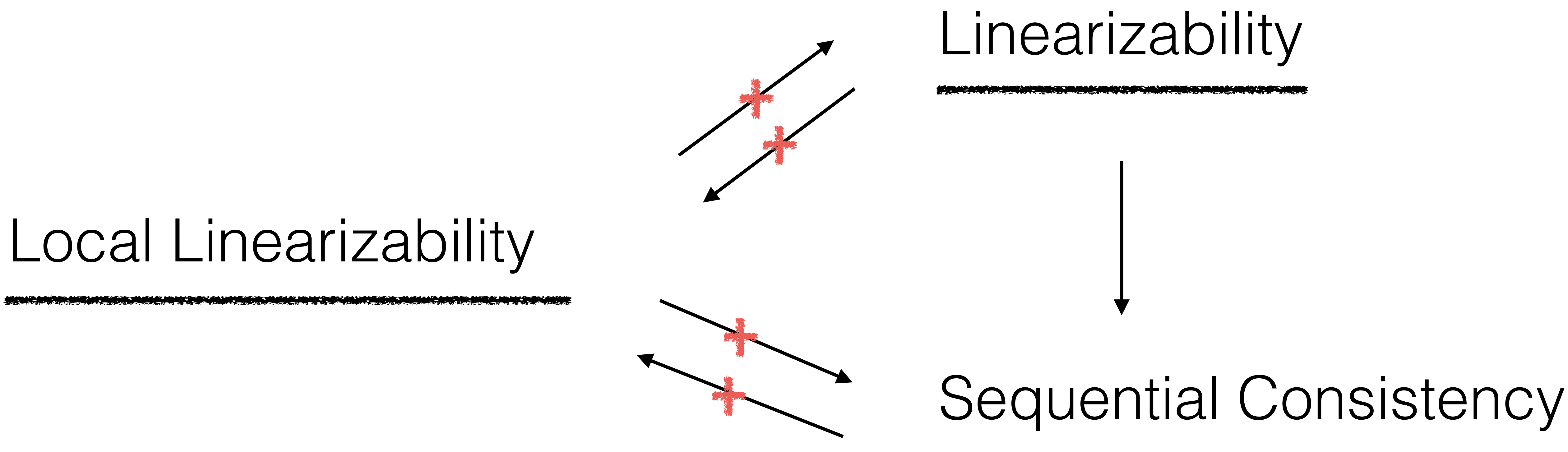
h is locally linearizable iff every thread-induced history

$$\mathbf{h}_T = \mathbf{h} \mid (I_T \cup O_T)$$

is linearizable.

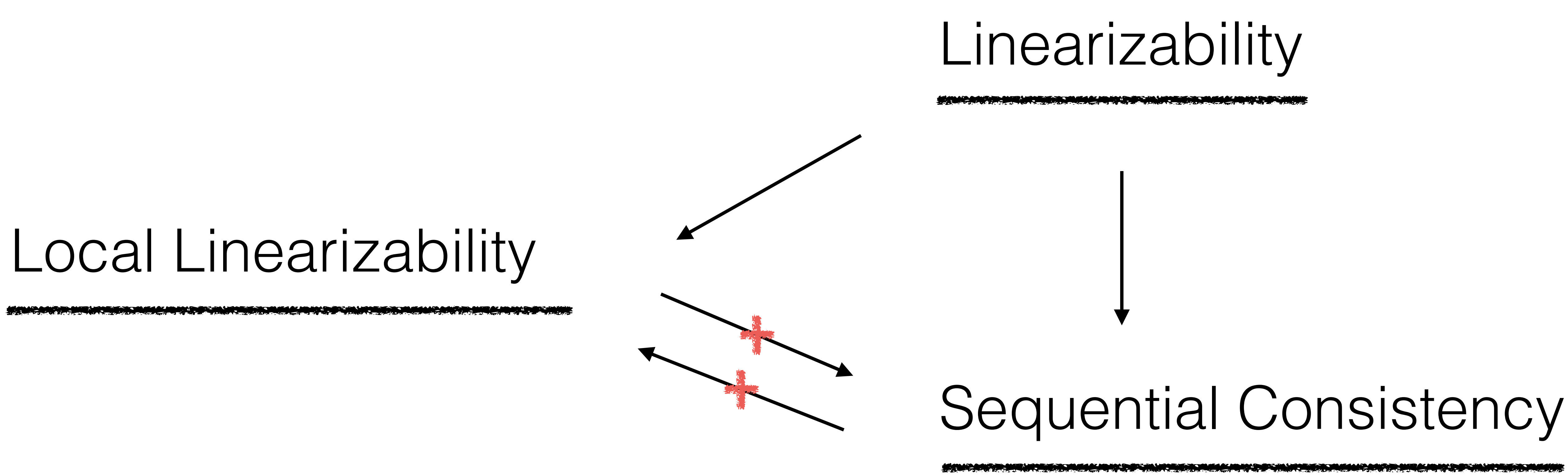
Where do we stand?

In general



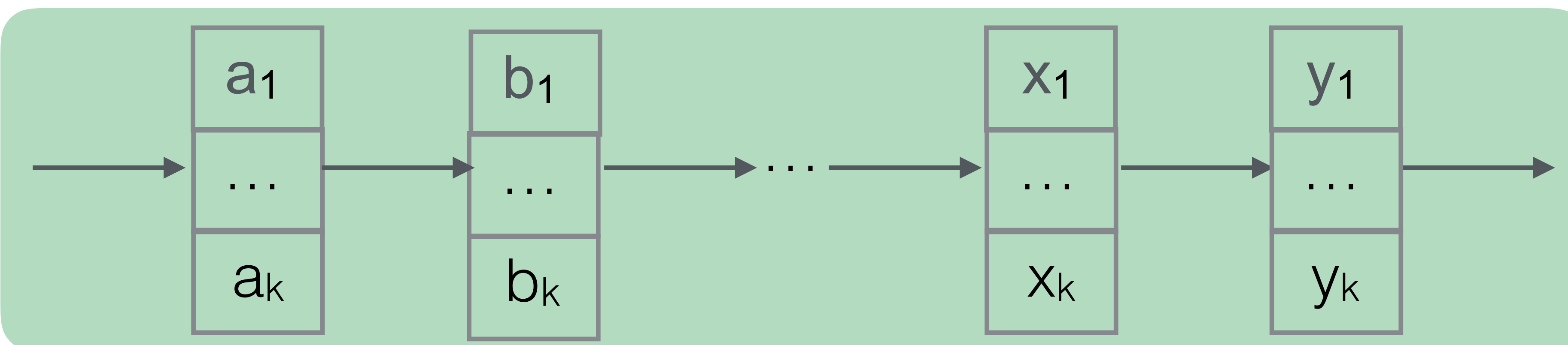
Where do we stand?

For queues (and most container-type data structures)



Lead to scalable implementations

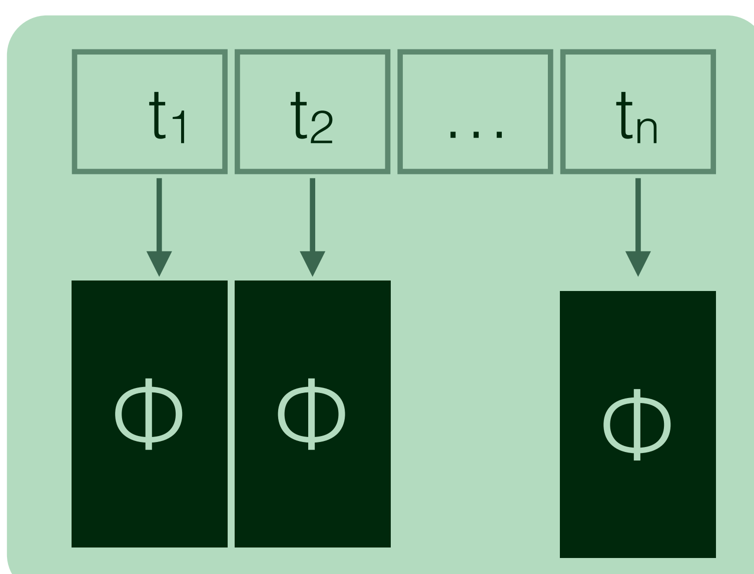
e.g. k-FIFO, k-Stack



k-out-of-order queue

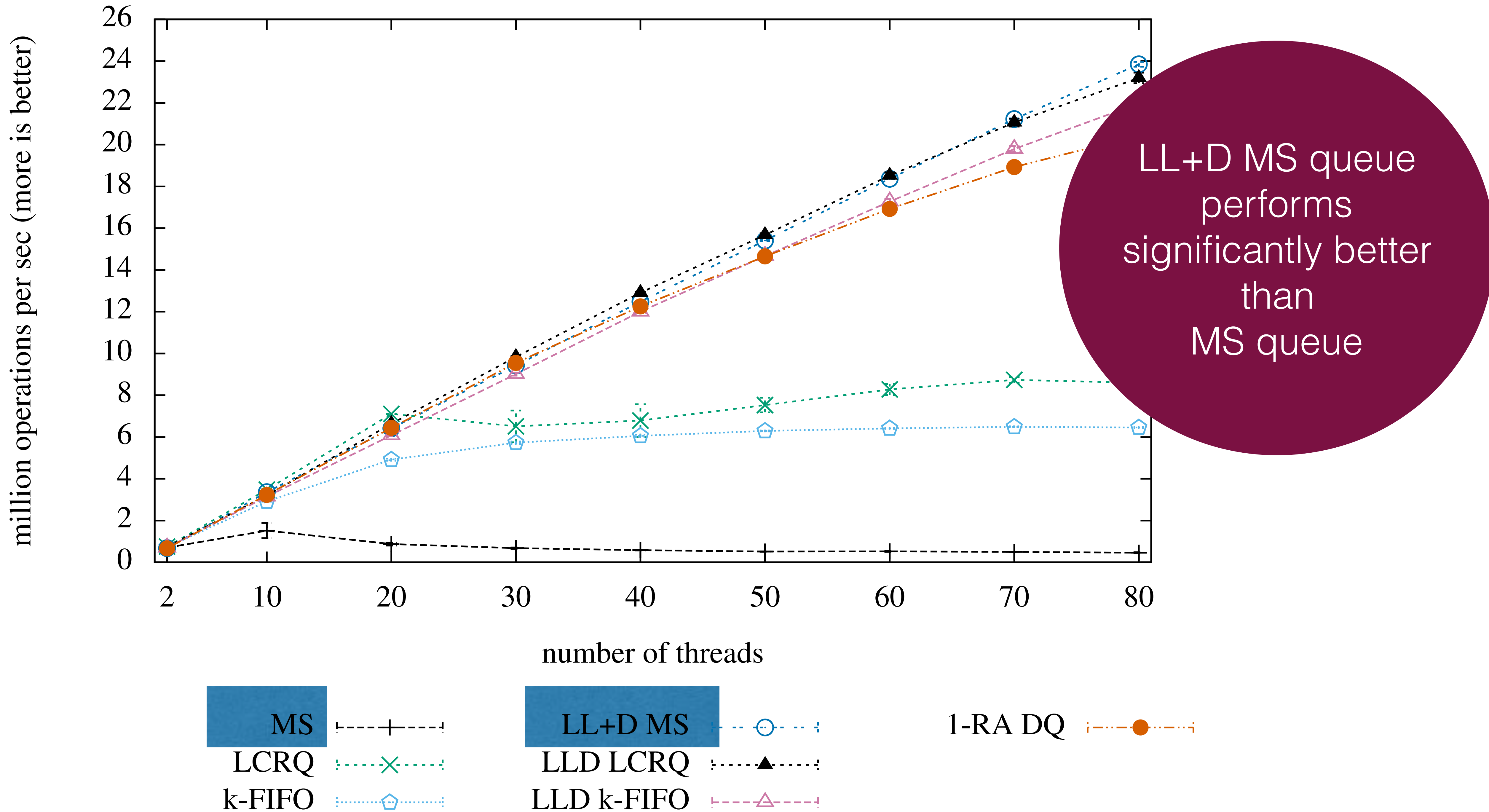
locally linearizable distributed implementation

LLD Φ
LL+D Φ



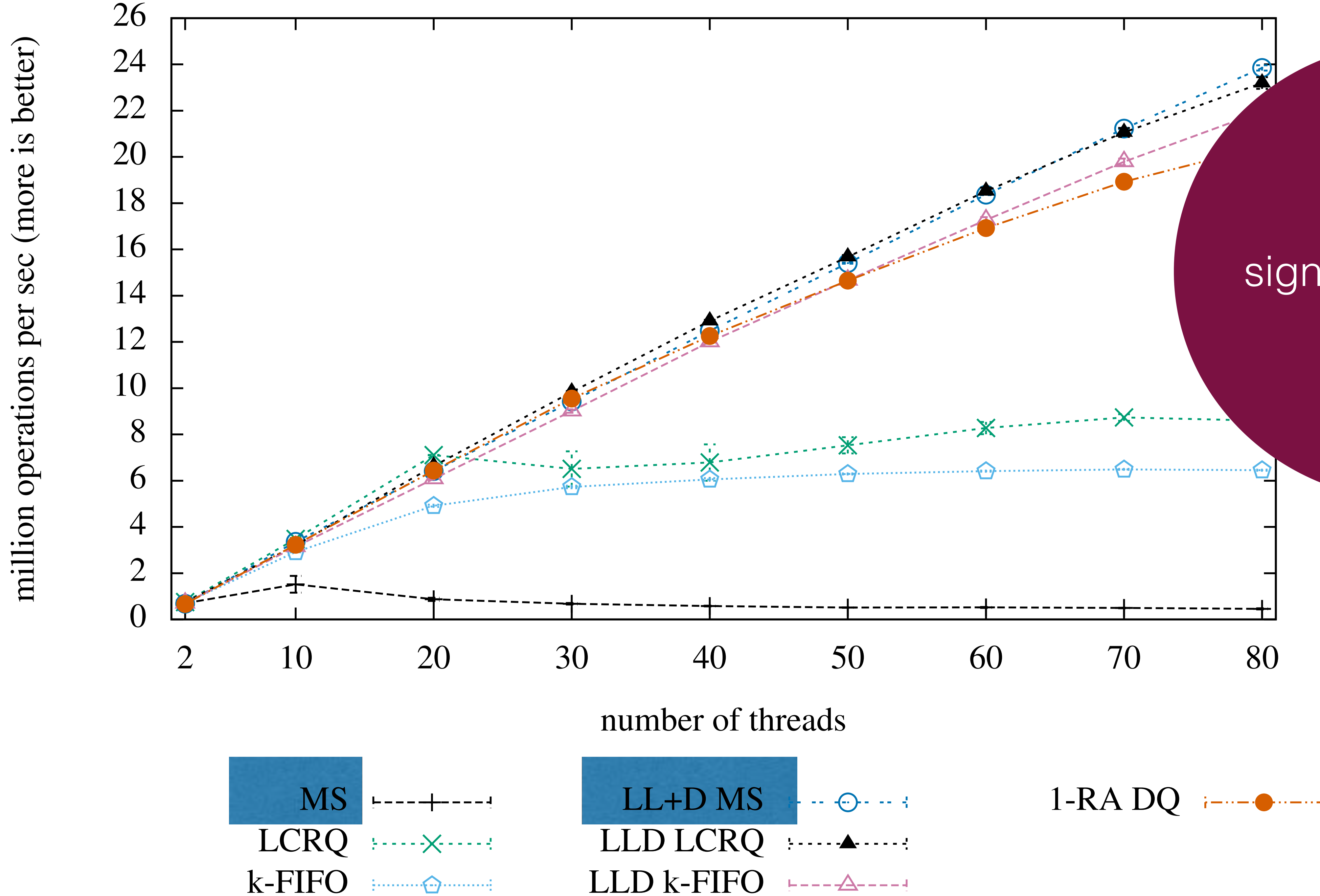
local inserts / global removes

Performance



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

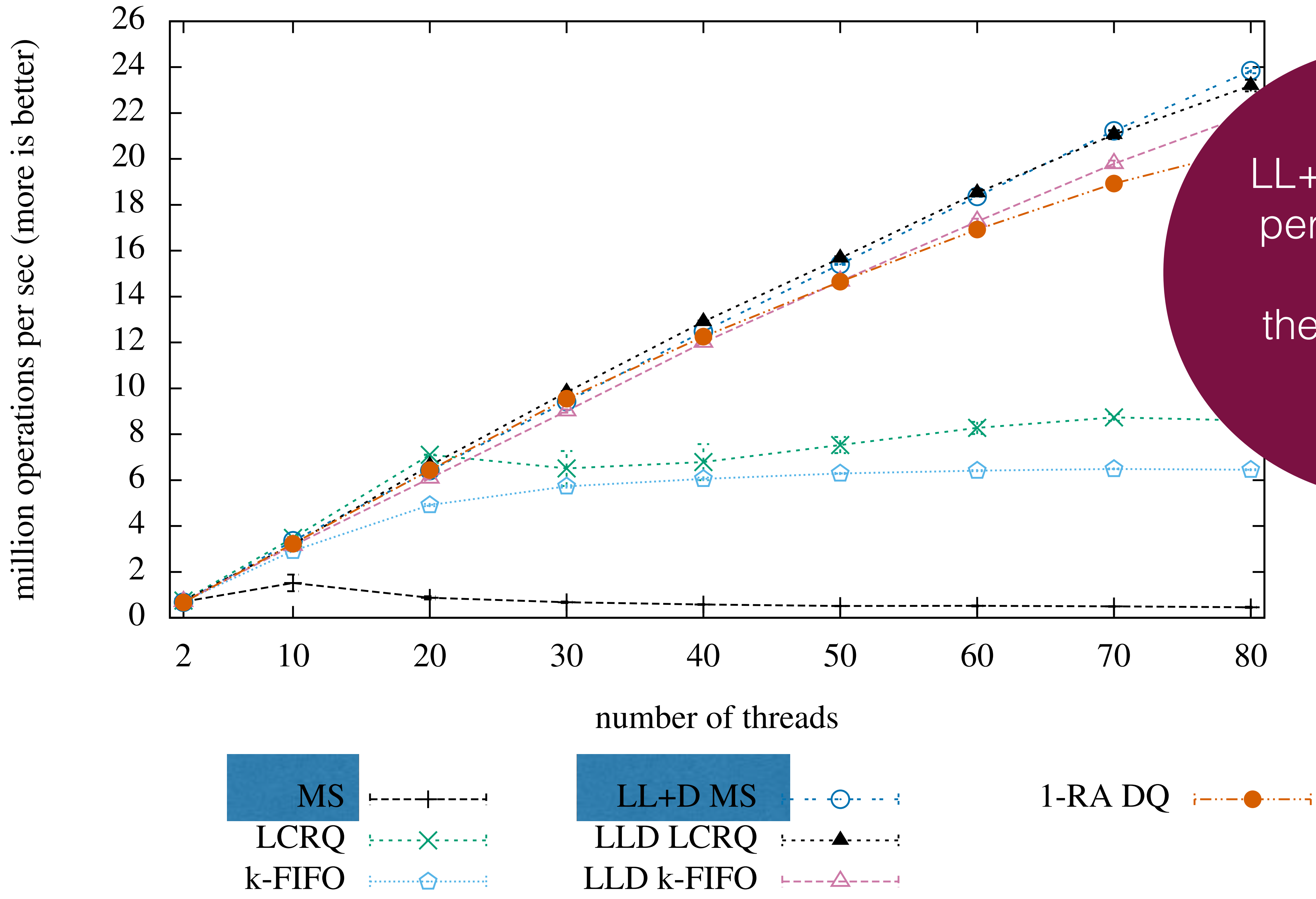
Performance



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

LLD Φ performs significantly better than Φ

Performance



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

LL+D MS queue performs better than the best known 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. $\text{enq}(x) <_{\mathbf{s}} \text{enq}(y) \ \wedge \ \text{deq}(y) \in \mathbf{s} \quad \Rightarrow \quad \text{deq}(x) \in \mathbf{s} \ \wedge \ \text{deq}(x) <_{\mathbf{s}} \text{deq}(y)$

Queue linearizability (axiomatic)

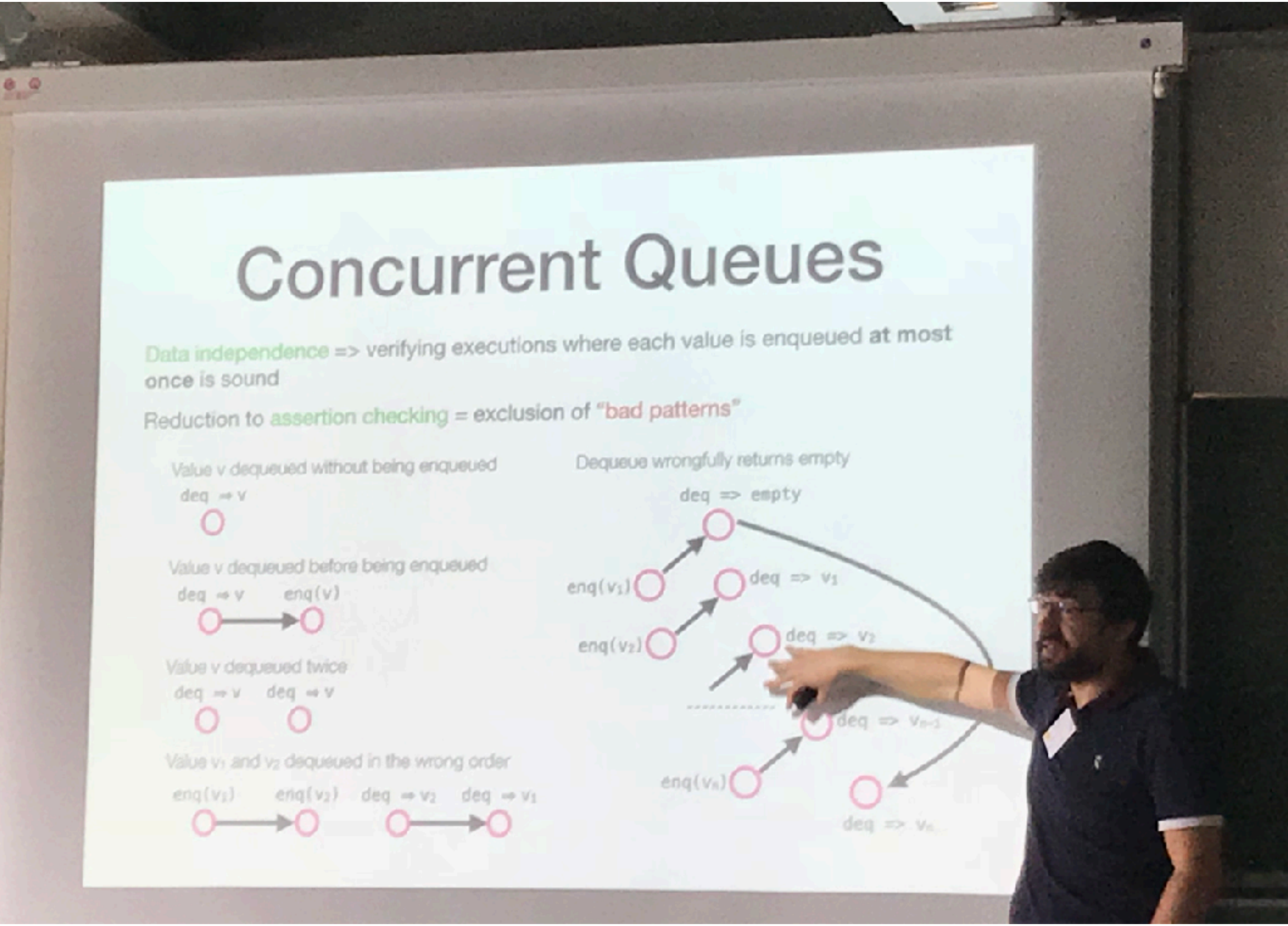
Henzinger, Sezgin, Vafeiadis CONCUR13

h is queue linearizable

iff

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

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)
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 $\mathbf{s} \in S$ iff $\mathbf{s}$ has no violations

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

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

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

legal sequence

we build CV iteratively from V

Pool without empty removals

Pool sequential specification (axiomatic)

s is a legal pool (without empty removals) sequence
iff
1. rem(x) ∈ **s** ⇒ ins(x) ∈ **s** ∧ ins(x) $<_{\mathbf{s}}$ rem(x)

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

Pool linearizability (axiomatic)

h is pool (without empty removals) linearizable
iff
1. rem(x) ∈ **h** ⇒ ins(x) ∈ **h** ∧ rem(x) $\nless_{\mathbf{h}}$ ins(x)

CV violations
= V violations

Queue without empty removals

V violations
deq(x) $<_s$ enq(x)
and
enq(x) $<_s$ enq(y) $\wedge$
deq(y) $<_s$ deq(x)

Queue sequential specification (axiomatic)

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

1. deq(x) $\in$ **s** $\Rightarrow$ enq(x) $\in$ **s** $\wedge$ enq(x) $<_s$ deq(x)
2. enq(x) $<_s$ enq(y) $\wedge$ deq(y) $\in$ **s** $\Rightarrow$ deq(x) $\in$ **s** $\wedge$ deq(x) $<_s$ deq(y)

Queue linearizability (axiomatic)

h is queue (without empty removals) linearizable
iff

1. rem(x) $\in$ **h** $\Rightarrow$ ins(x) $\in$ **h** $\wedge$ rem(x) $\not<_h$ ins(x)
2. enq(x) $<_h$ enq(y) $\wedge$ deq(y) $\in$ **h** $\Rightarrow$ deq(x) $\in$ **h** $\wedge$ deq(y) $\not<_h$ deq(x)

CV violations
= V violations

Pool

infinite inductive violations

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

Pool sequential specification (axiomatic)

s is a legal pool (with empty removals) sequence
iff

1. rem(x) $\in$ **s** $\Rightarrow$ ins(x) $\in$ **s** $\wedge$ ins(x) $<_{\mathbf{s}}$ rem(x)
2. rem(⊥) $<_{\mathbf{s}}$ rem(x) $\Rightarrow$ rem(⊥) $<_{\mathbf{s}}$ ins(x) $\wedge$ ins(x) $<_{\mathbf{s}}$ rem(⊥) $\Rightarrow$ rem(x) $<_{\mathbf{s}}$ rem(⊥)

Pool linearizability (axiomatic)

h is pool (with empty removals) linearizable
iff

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

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

It works for

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

infinite inductive violations

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

How does it work?

The basics

$$\mathrm{PO}[\mathcal{V}] = \{R \in \mathrm{PO} \mid \mathcal{P}(R) \cap \mathcal{V} = \emptyset\}$$
$$\mathrm{IO}[\mathcal{V}] = \{R \in \mathrm{IO} \mid \mathcal{P}(R) \cap \mathcal{V} = \emptyset\}$$
$$\mathrm{TO}[\mathcal{V}] = \{R \in \mathrm{TO} \mid \mathcal{P}(R) \cap \mathcal{V} = \emptyset\}$$

partial orders

interval orders

total orders

$$\forall (a,b),(c,d) \in R.\ (a,d) \in R \vee (c,b) \in R$$

The problem

Given a set of violations $\mathcal{V}$, find a "small" set of violations $\mathcal{V}'$ such that

$$\forall R \in \mathrm{IO}[\mathcal{V}'].\ \exists \overline{R} \in \mathrm{TO}[\mathcal{V}].\ \overline{R} \supseteq R$$

this solves the case of pool without empty removals

Theorem (singleton violations)

Let $\mathcal{V}$ consist only of singletons, and let $V = \bigcup \mathcal{V}$.

If V is transitive and not a cycle, then the problem is solved with $\mathcal{V}' = \mathcal{V}$.

The closures

$$\mathrm{Clos}_{\mathrm{O}}(\mathcal{V}) = \bigcap_{S \in \mathrm{O}[\mathcal{V}]} \mathcal{P}(S)^c$$

O-closure of a set of violations

monotone, extensive, idempotent

Proposition

$$\forall R \in \mathrm{IO}[\mathcal{V}']. \ \exists \overline{R} \in \mathrm{TO}[\mathcal{V}]. \ \overline{R} \supseteq R$$

iff

$$\mathrm{Clos}_{\mathrm{TO}}(\mathcal{V}) = \mathrm{Clos}_{\mathrm{IO}}(\mathcal{V}')$$

How does it work ?

Theorem

Let $\mathcal{V}$ consist only of finite sets and assume

(1) ★

(2) $\forall N, M \in \mathcal{V}.\ \forall (a_1, a_2) \in N.\ |\{(b_1, b_2) \in M \mid a_2 = b_1\}| \leqslant 1$

then the problem is solved

if we manage to construct such a set of violations, we are done

we provide an algorithm that produces a set of violations such that ★ holds

if we are lucky, (2) holds too

The algorithm

Take two violations $N_1, N_2 \in \mathcal{V}$ and an element $x \in X$ and produce a new violation

$$\{(a,b) \mid (a,x) \in N_1, (x,b) \in N_2\}$$
$$\cup \{(a,b) \in N_1 \mid b \neq x\}$$
$$\cup \{(a,b) \in N_2 \mid a \neq x\}$$

Take two violations $N_1, N_2 \in \mathcal{V}$ and a pair $(x,y) \in X \times X$ and produce a new violation

$$\{(a,y) \mid (a,x) \in N_2\}$$
$$\cup \{(x,b) \mid (y,b) \in N_2\}$$
$$\cup \{(a,b) \in N_2 \mid b \neq x \wedge a \neq y\}$$
$$\cup N_1 \backslash \{(x,y)\}$$

until no new violations are produced

It works for

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

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