Semantics of Concurrent Data Structures

Ana Sokolova
UNIVERSITY of SALZBURG

IFIP WG 1.3 meeting at Royal Holloway, 5.7.2018

- Part I: Concurrent data structures
  correctness and performance

structure and power

via semantic relaxations

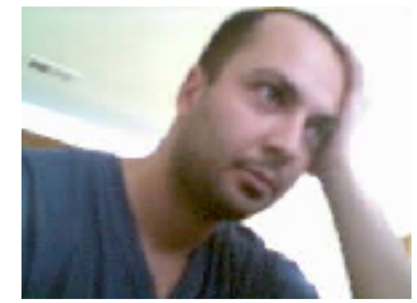
- Part II: Order extension results for
  verifying linearizability

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

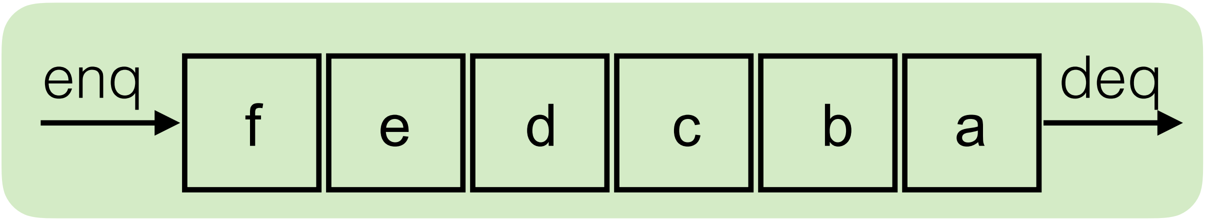
Andreas Holzer
Google

Helmut Veith TU WIEN

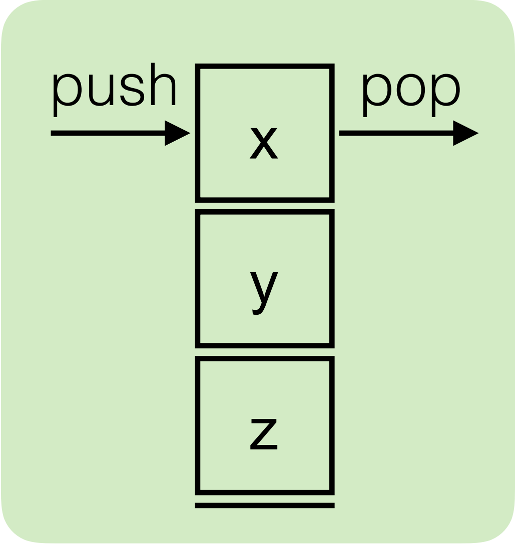
RiSE
Rigorous Systems Engineering

Data structures

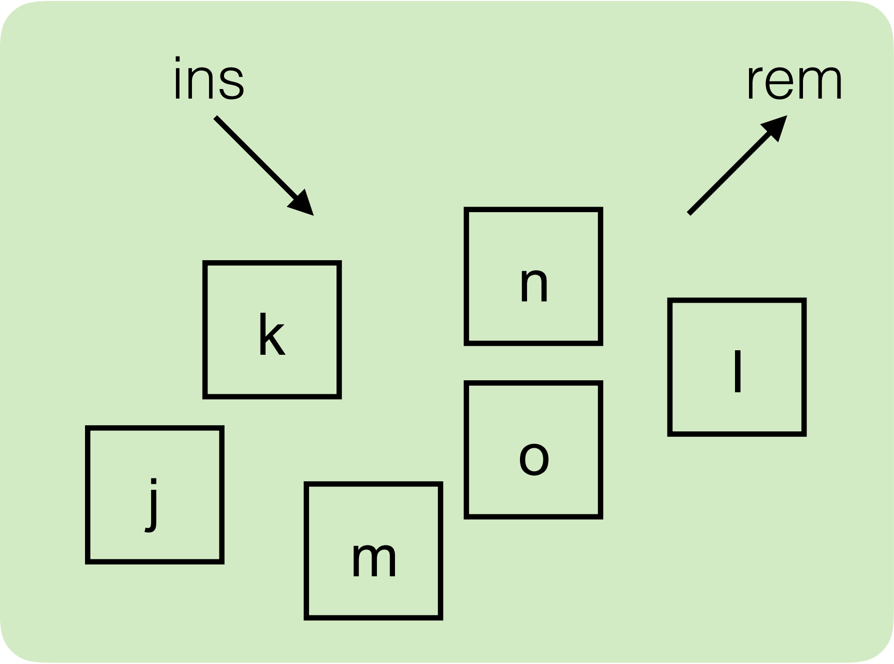
- Queue FIFO



- Stack LIFO

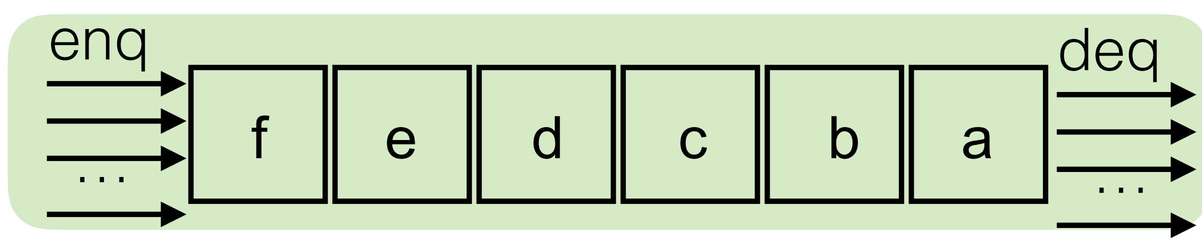


- Pool unordered

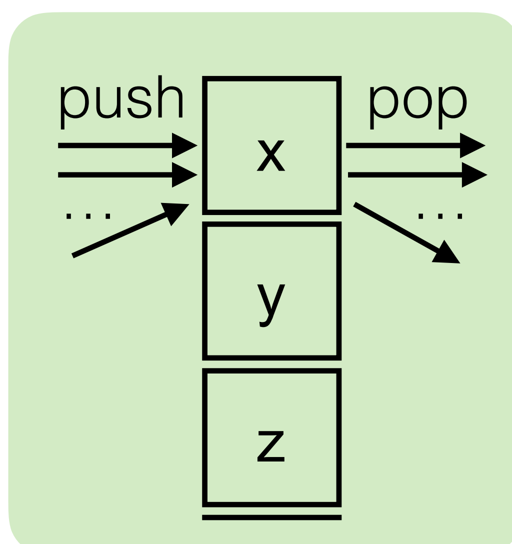


Concurrent data structures

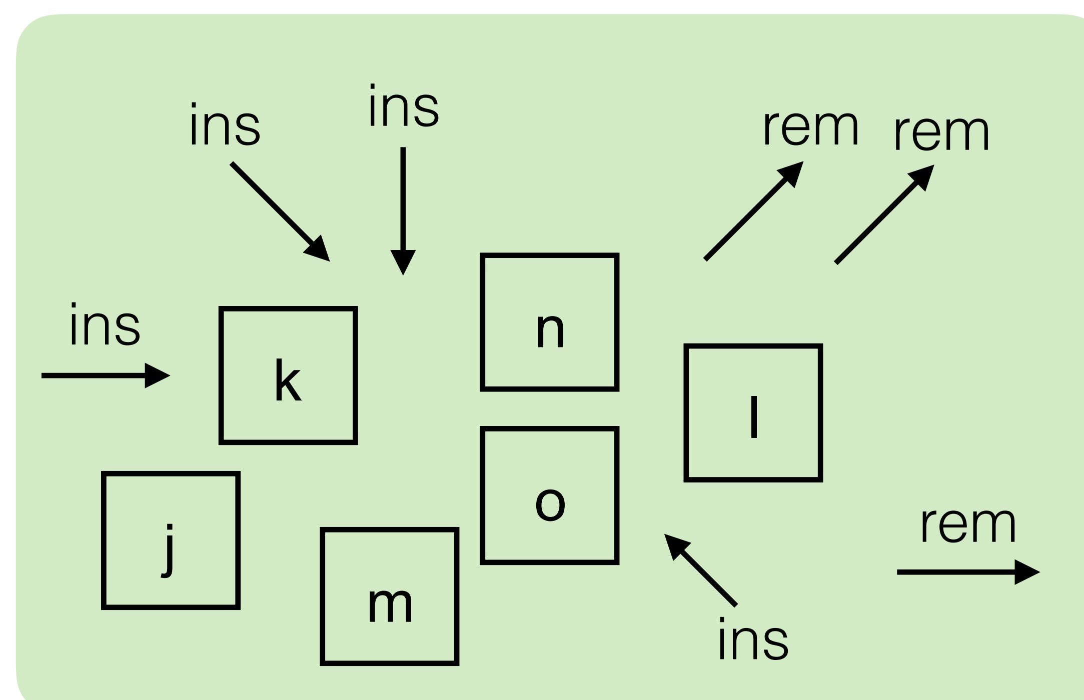
- Queue FIFO



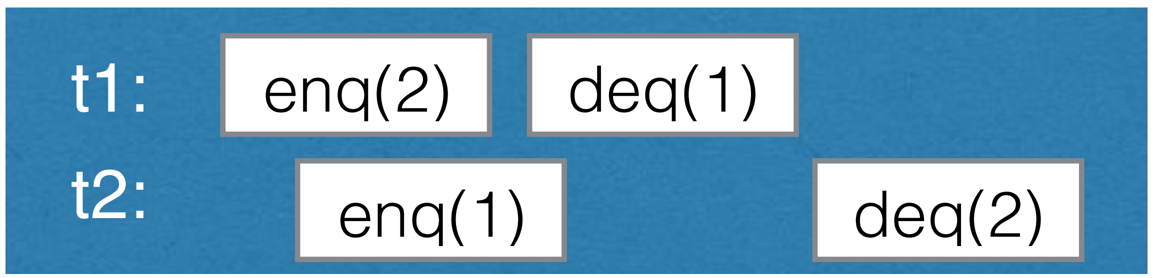
- Stack LIFO



- 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

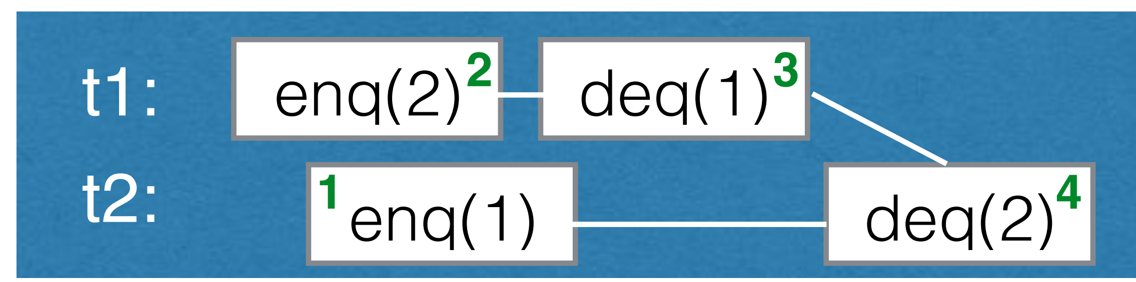
e.g. the concurrent history above is a linearizable queue concurrent history

Consistency conditions

A history is … wrt a sequential specification iff

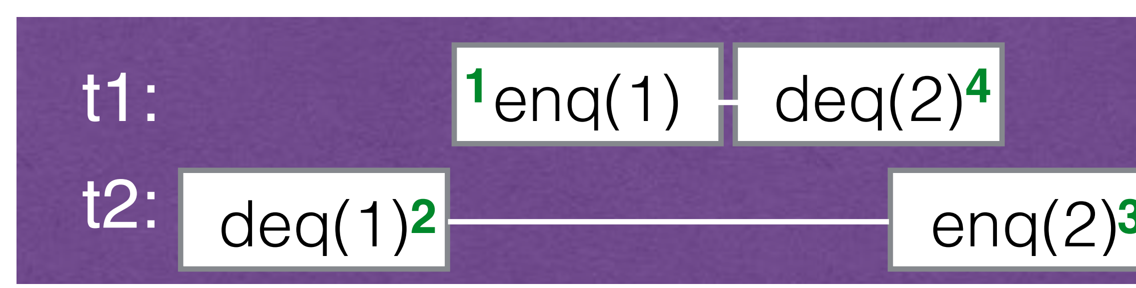
there exists a legal sequence that preserves precedence order

Linearizability [Herlihy,Wing '90]



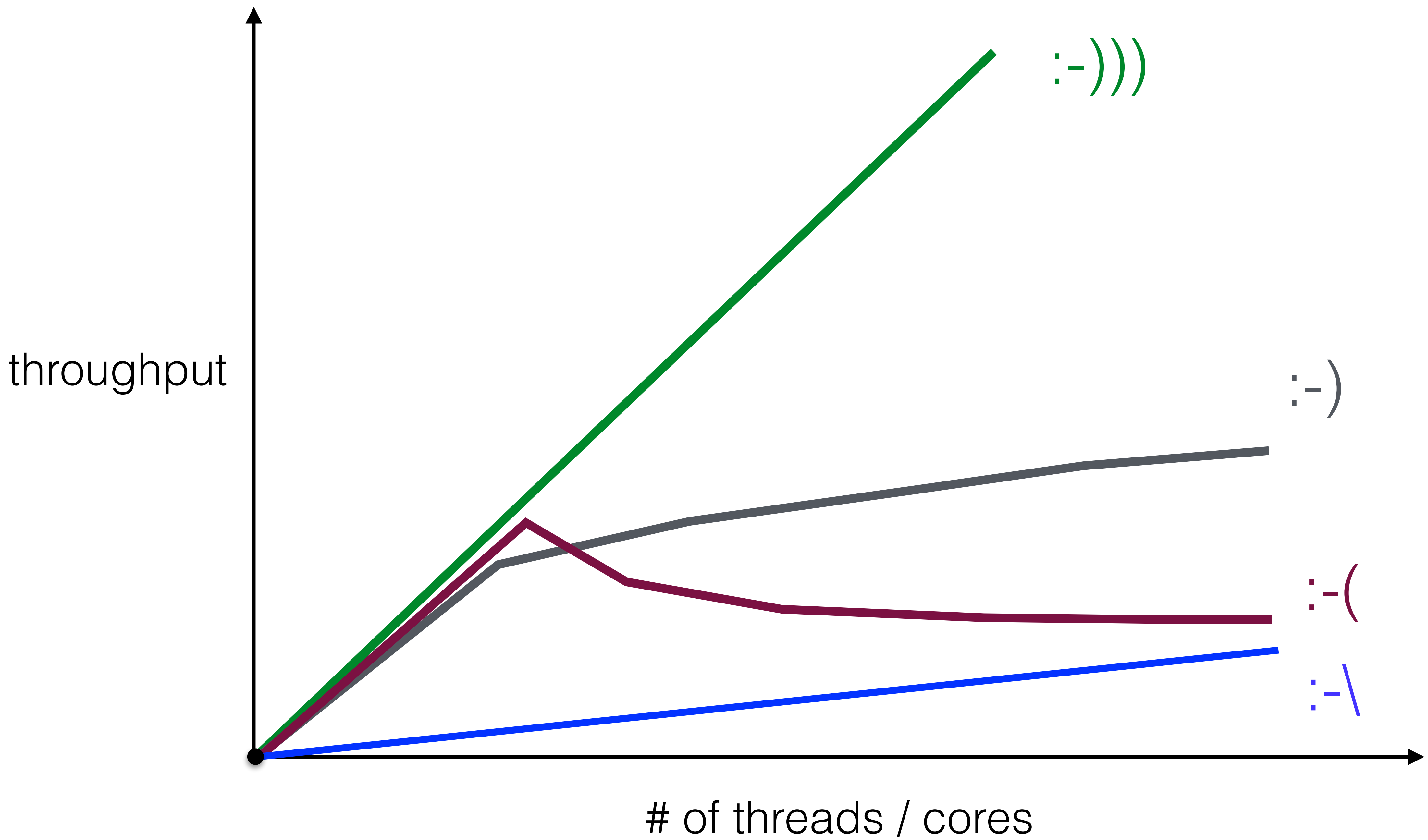
consistency is about extending partial orders to total orders

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

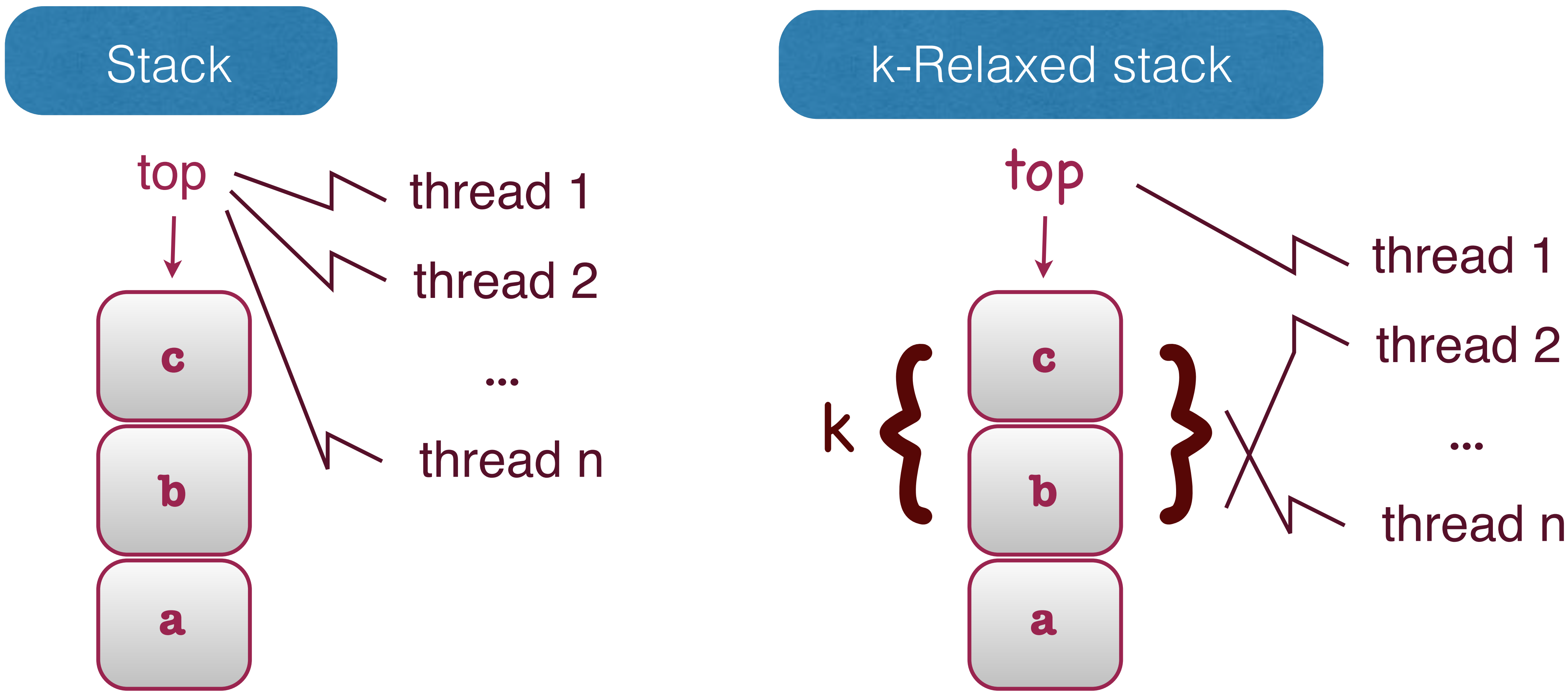
push(a)push(b)push(c)pop(a)pop(b)

- 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
- Generic examples
- Concrete relaxation examples
- Efficient concurrent implementations

for semantic relaxations

out-of-order / stuttering

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

of relaxation instances

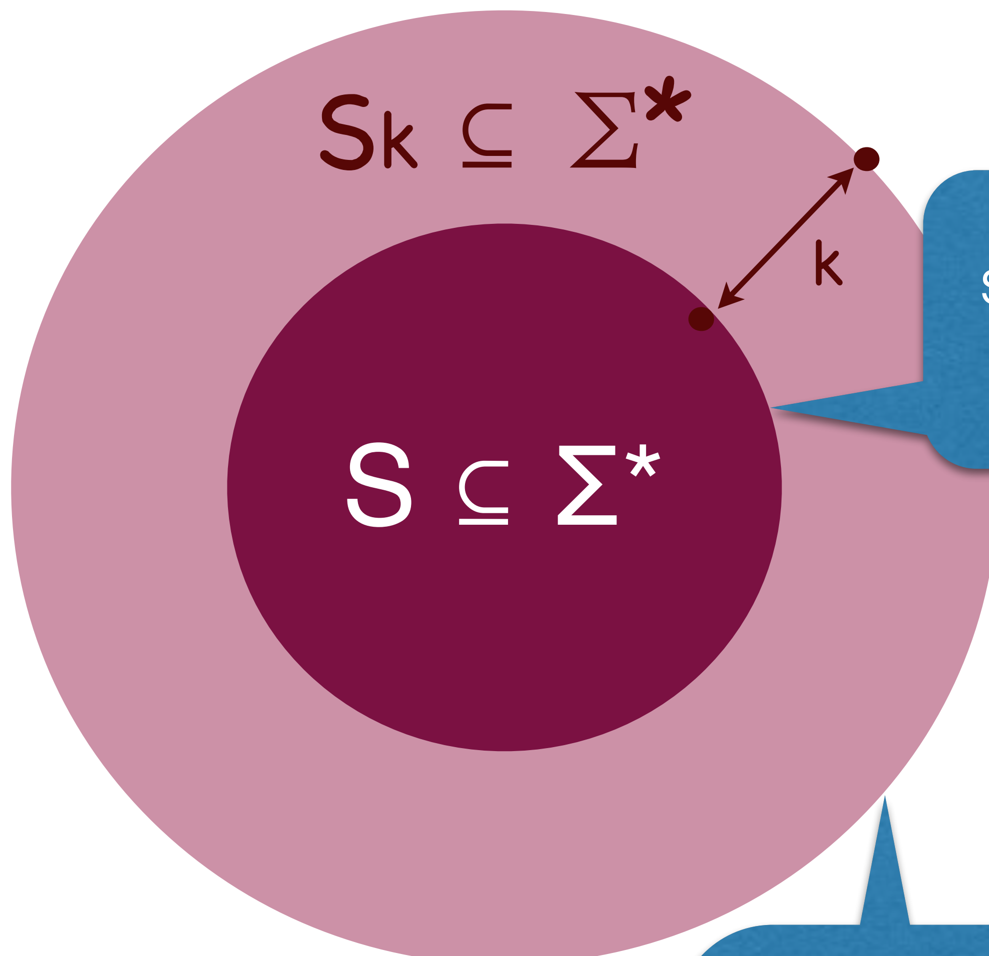
The big picture

$S \subseteq \Sigma^*$

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

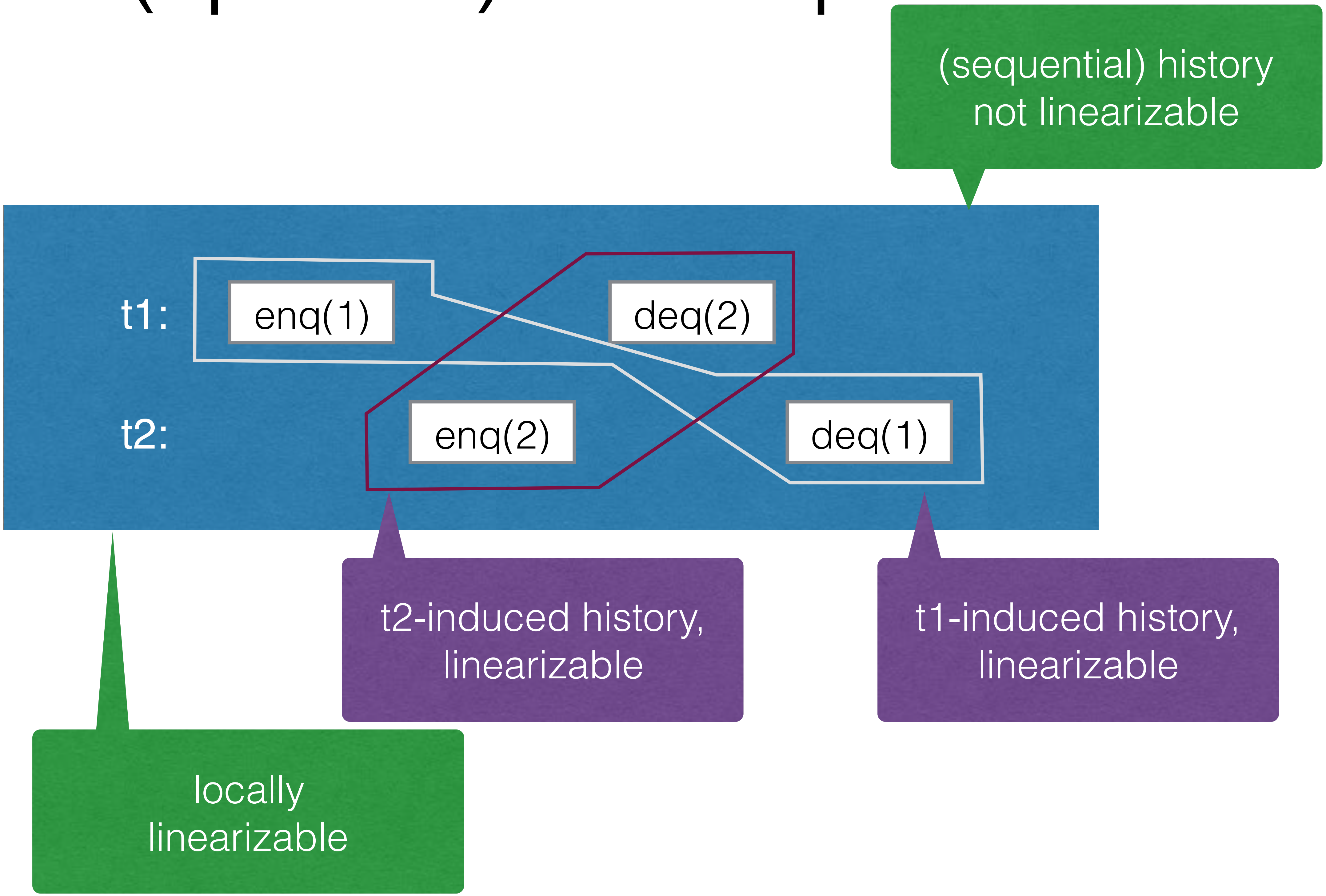
Already present in some shared-memory consistency conditions
(not in our form of choice)

- 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

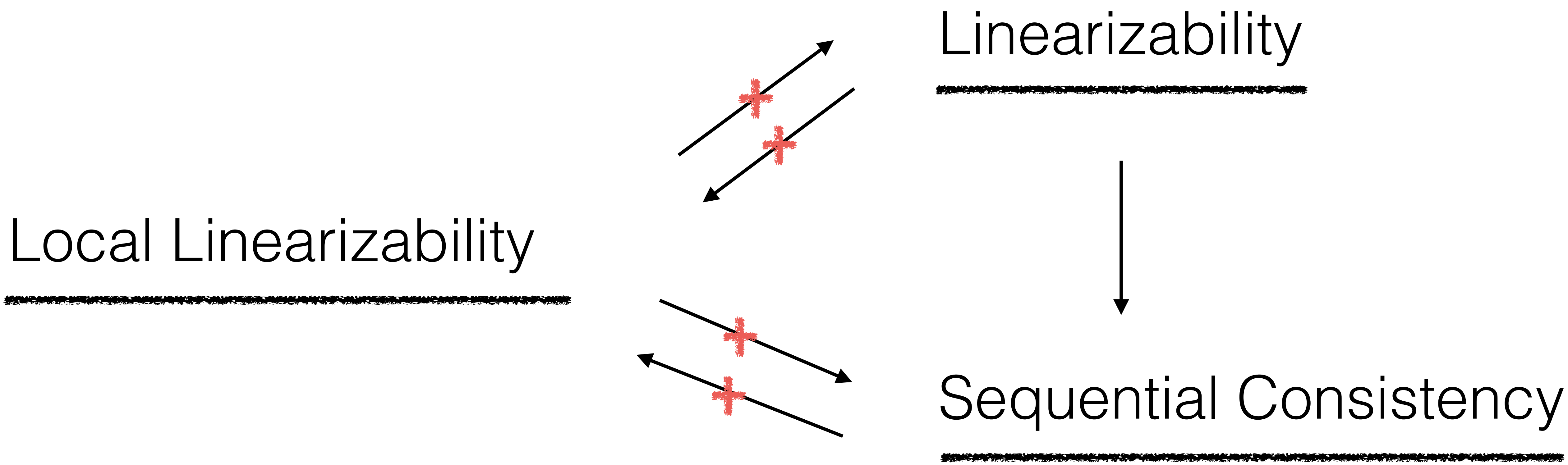
h is locally linearizable iff every thread-induced history

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

is linearizable.

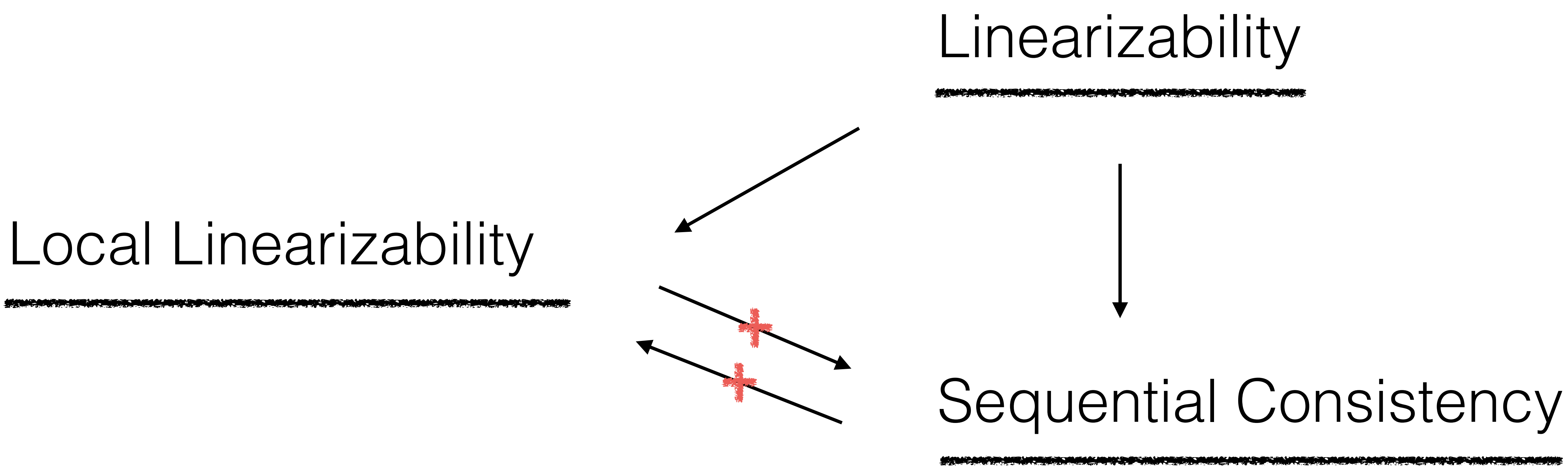
Where do we stand?

In general



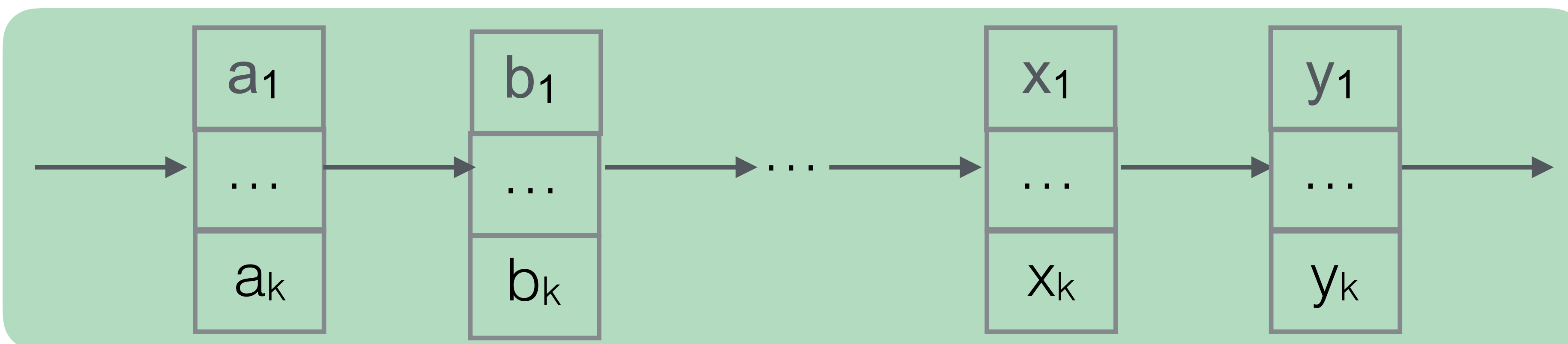
Where do we stand?

For queues (and most container-type data structures)



Lead to scalable implementations

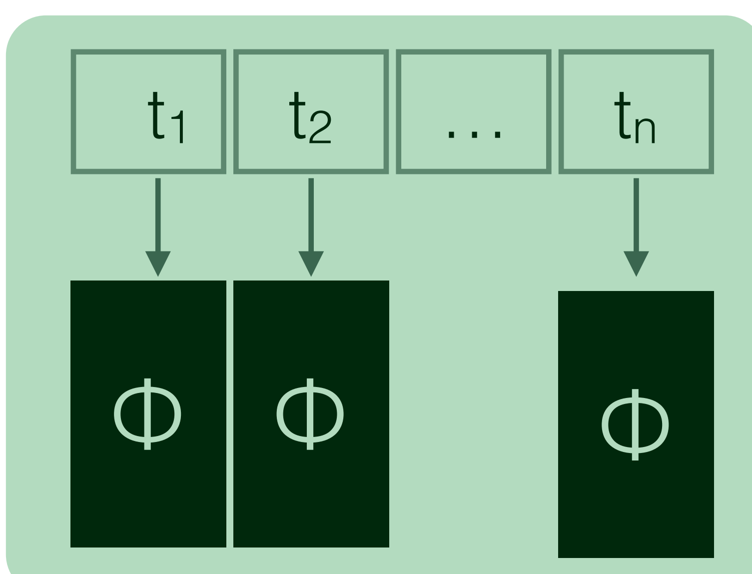
e.g. k-FIFO, k-Stack



k-out-of-order queue

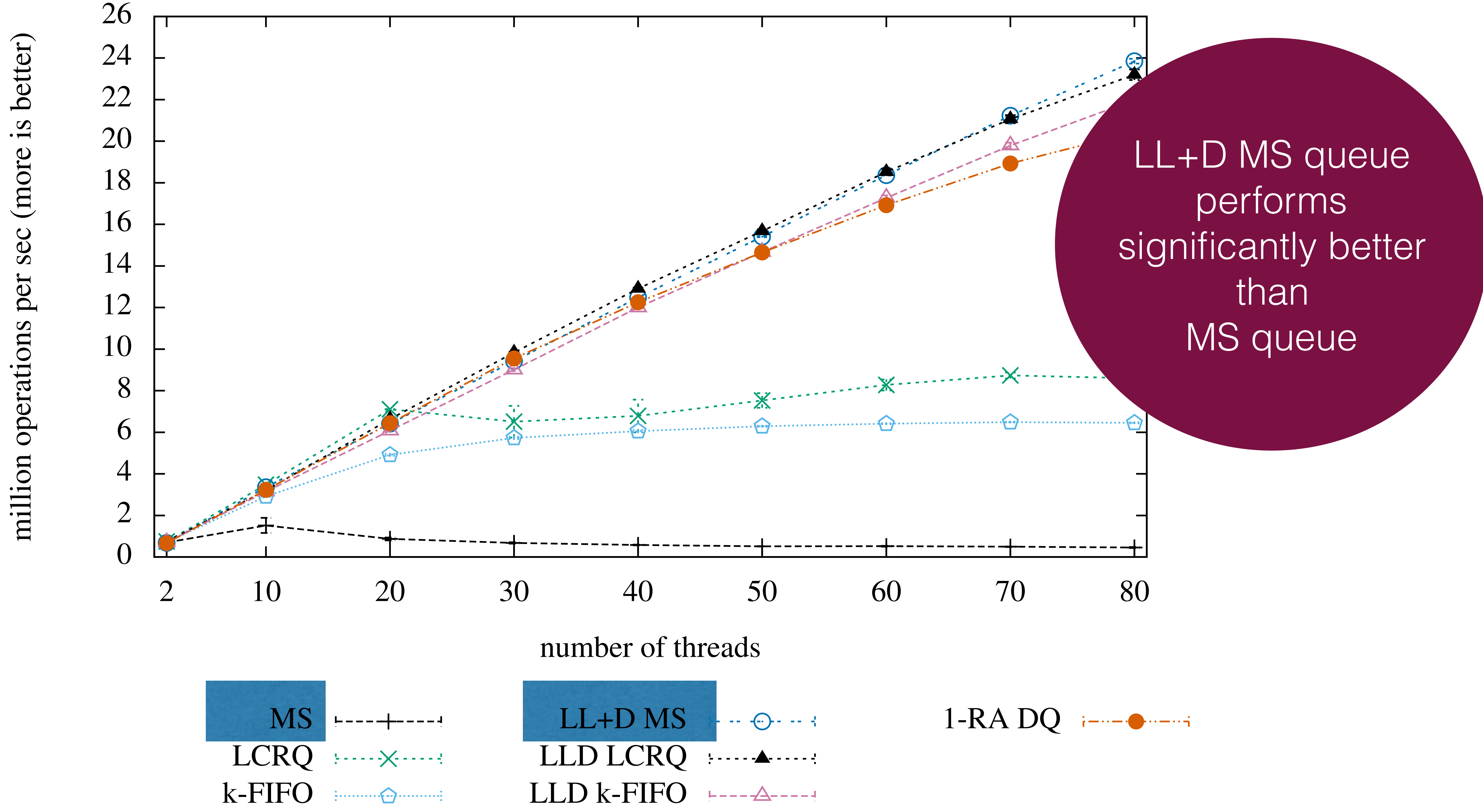
locally linearizable distributed implementation

LLD Φ
LL+D Φ



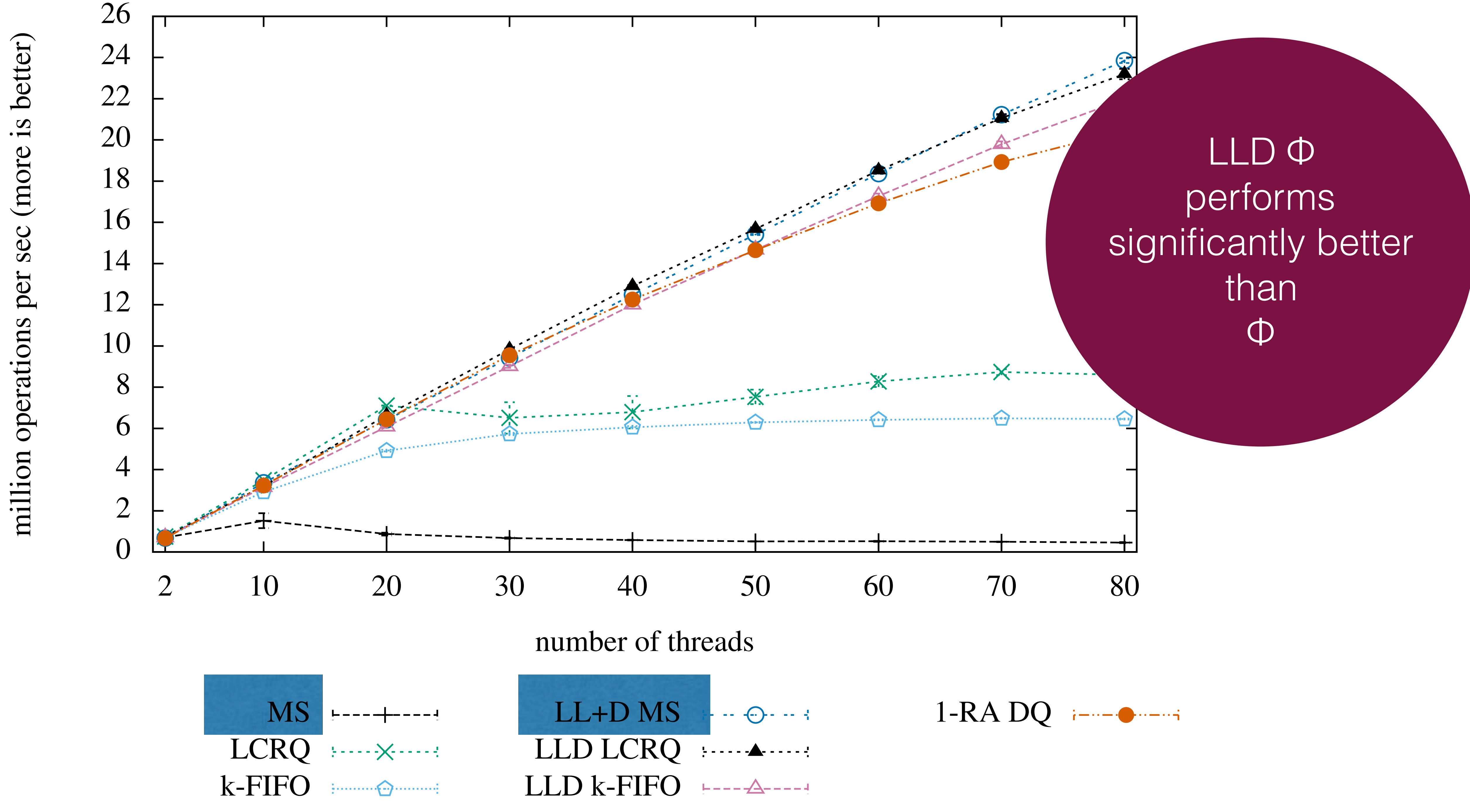
local inserts / global removes

Performance



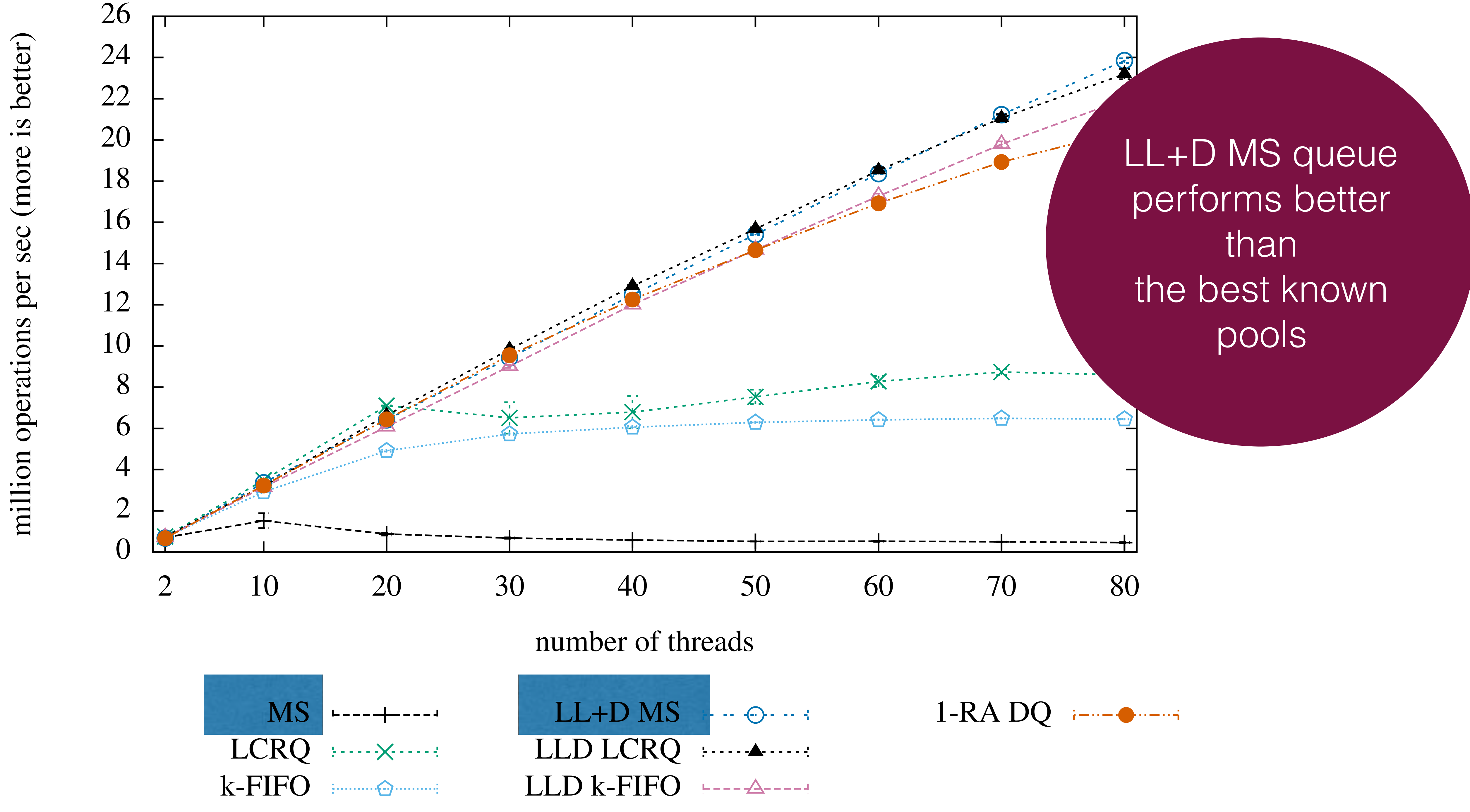
(a) Queues, LL queues, and "queue-like" pools

Performance



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

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

Queue linearizability (axiomatic)

Henzinger, Sezgin, Vafeiadis CONCUR13

h is queue linearizable

iff

1. **h** is pool linearizable, and
2. $enq(x) <_h enq(y) \wedge deq(y) \in h \Rightarrow deq(x) \in h \wedge deq(y) \not<_h deq(x)$

precedence order

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

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.

we build CV iteratively from V

legal sequence

concurrent history

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

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