Semantics for Concurrency

Ana Sokolova UNIVERSITY of SALZBURG

MOVEP, 18.7.2018

- Part I: Concurrent data structures
  correctness and performance

structure and power

via semantic relaxations

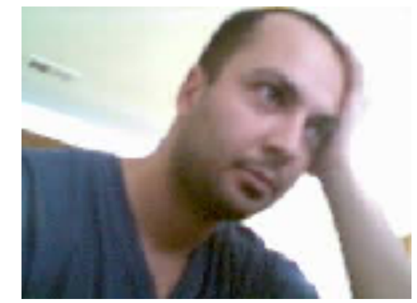
- Part II: Order extension results for
  verifying linearizability

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

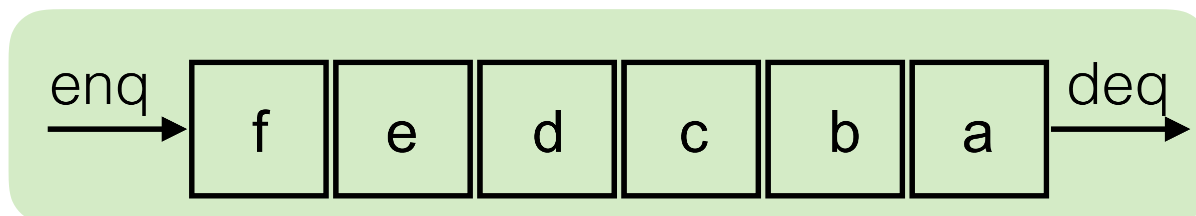
Andreas Holzer
Google

Helmut Veith TU WIEN

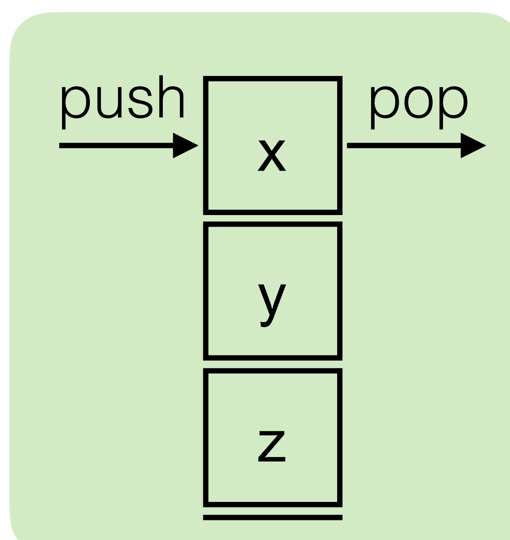
RiSE
Rigorous Systems Engineering

Data structures

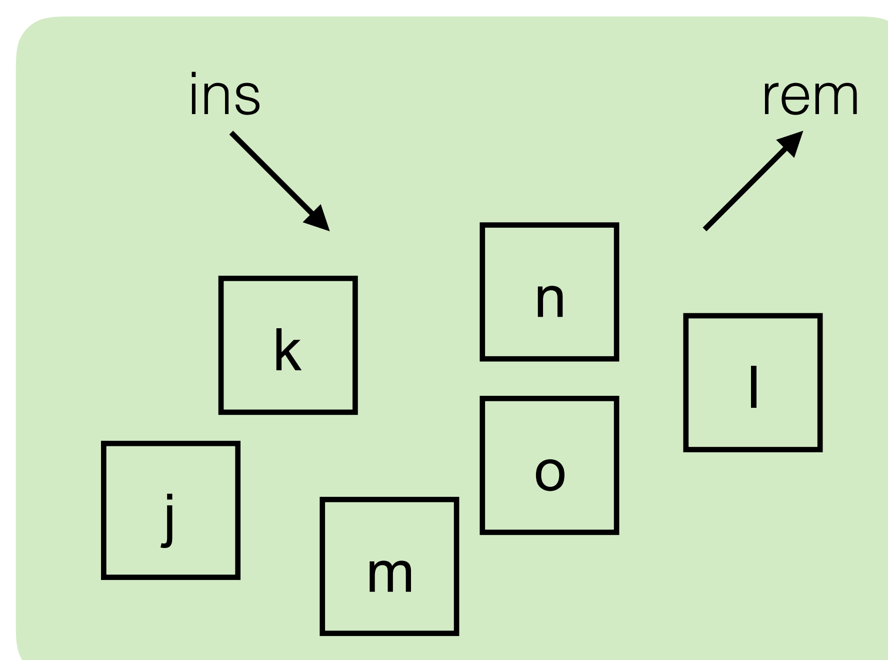
- Queue FIFO



- Stack LIFO

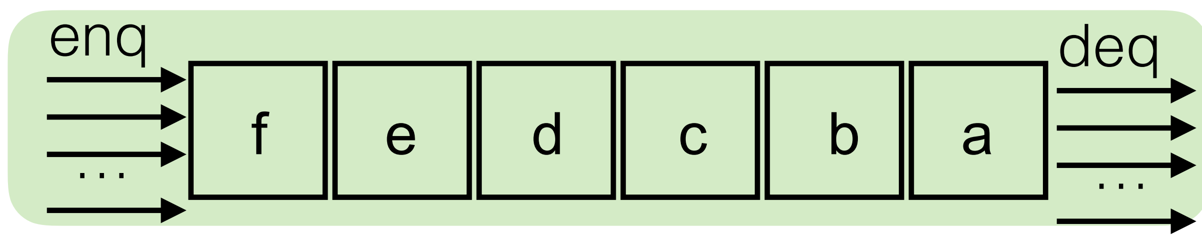


- Pool unordered

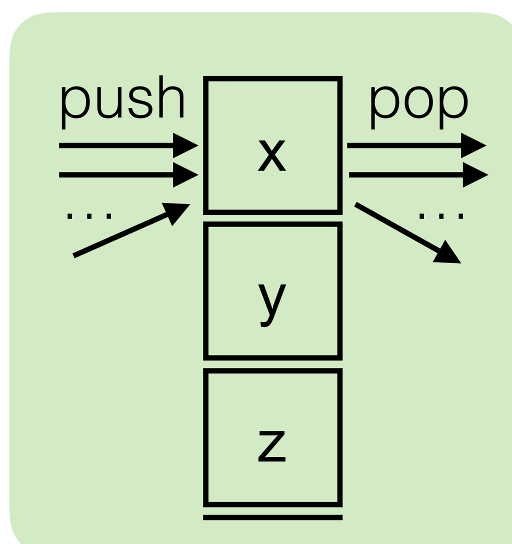


Concurrent data structures

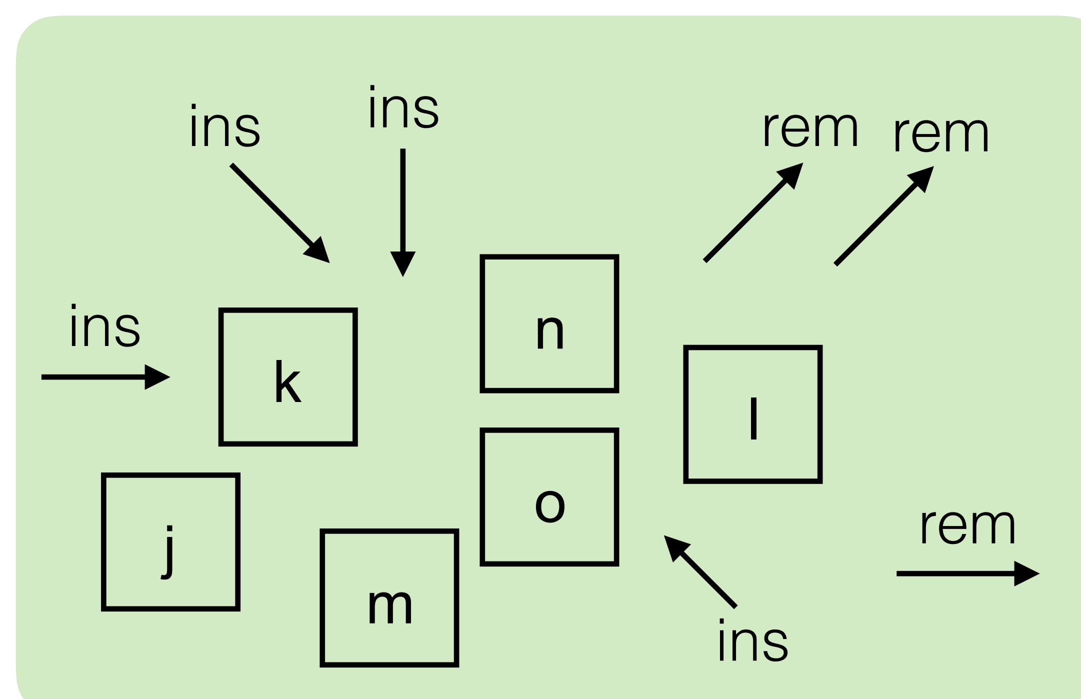
- Queue FIFO



- Stack LIFO



- Pool unordered



Semantics of concurrent data structures

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

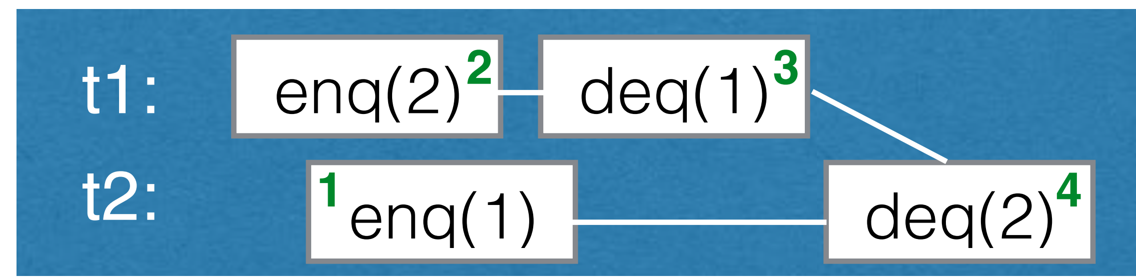
e.g. the concurrent history above is a linearizable queue concurrent history

Consistency conditions

A history is … wrt a sequential specification iff

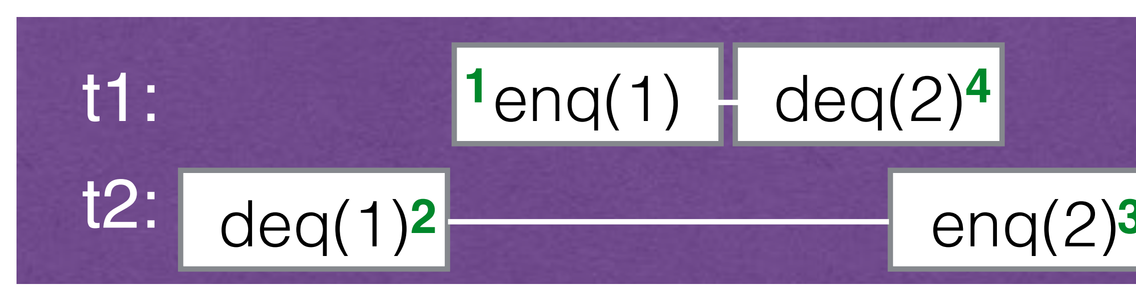
there exists a legal sequence that preserves precedence order

Linearizability [Herlihy,Wing '90]



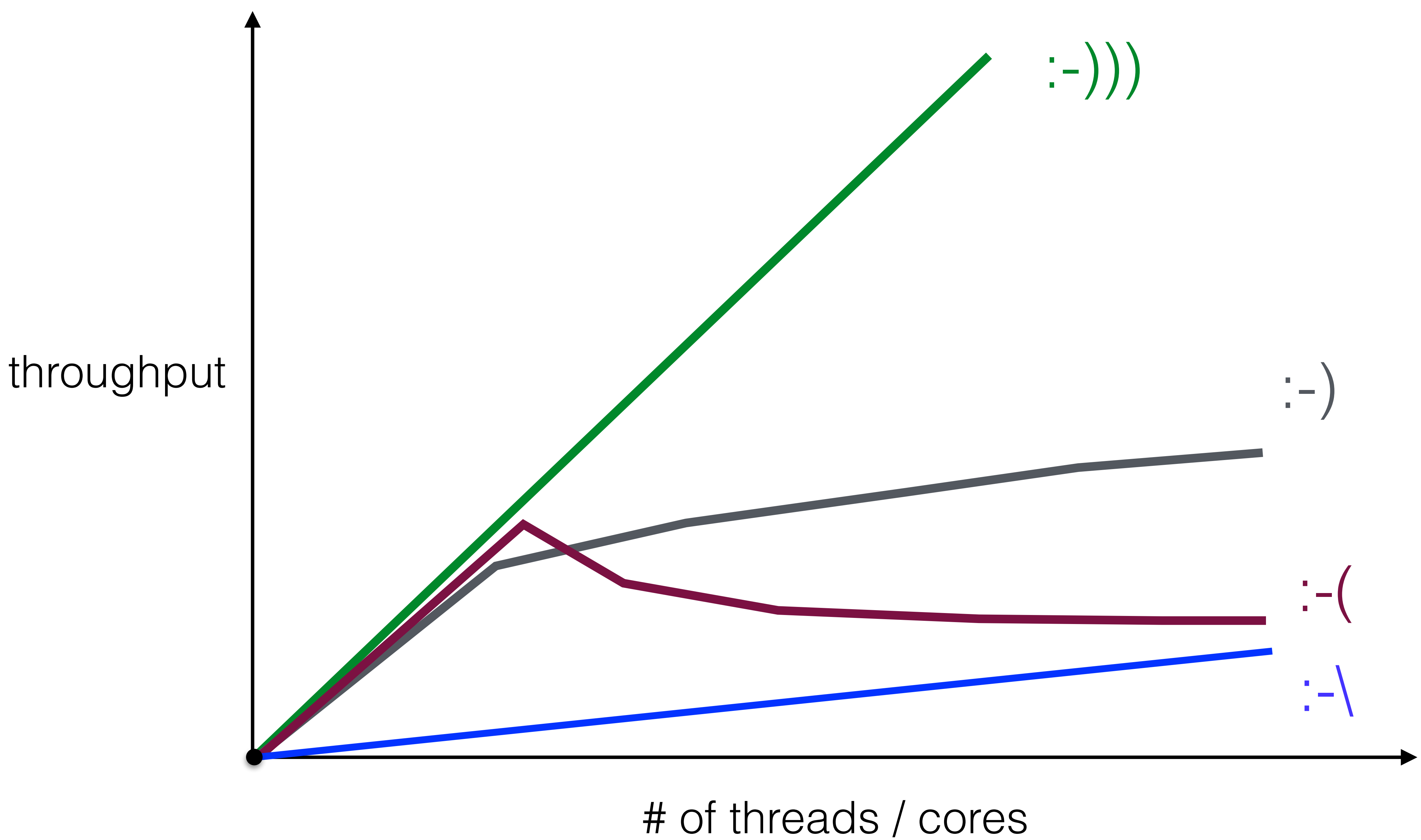
consistency is about extending partial orders to total orders

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

Goal

Stack - incorrect behavior

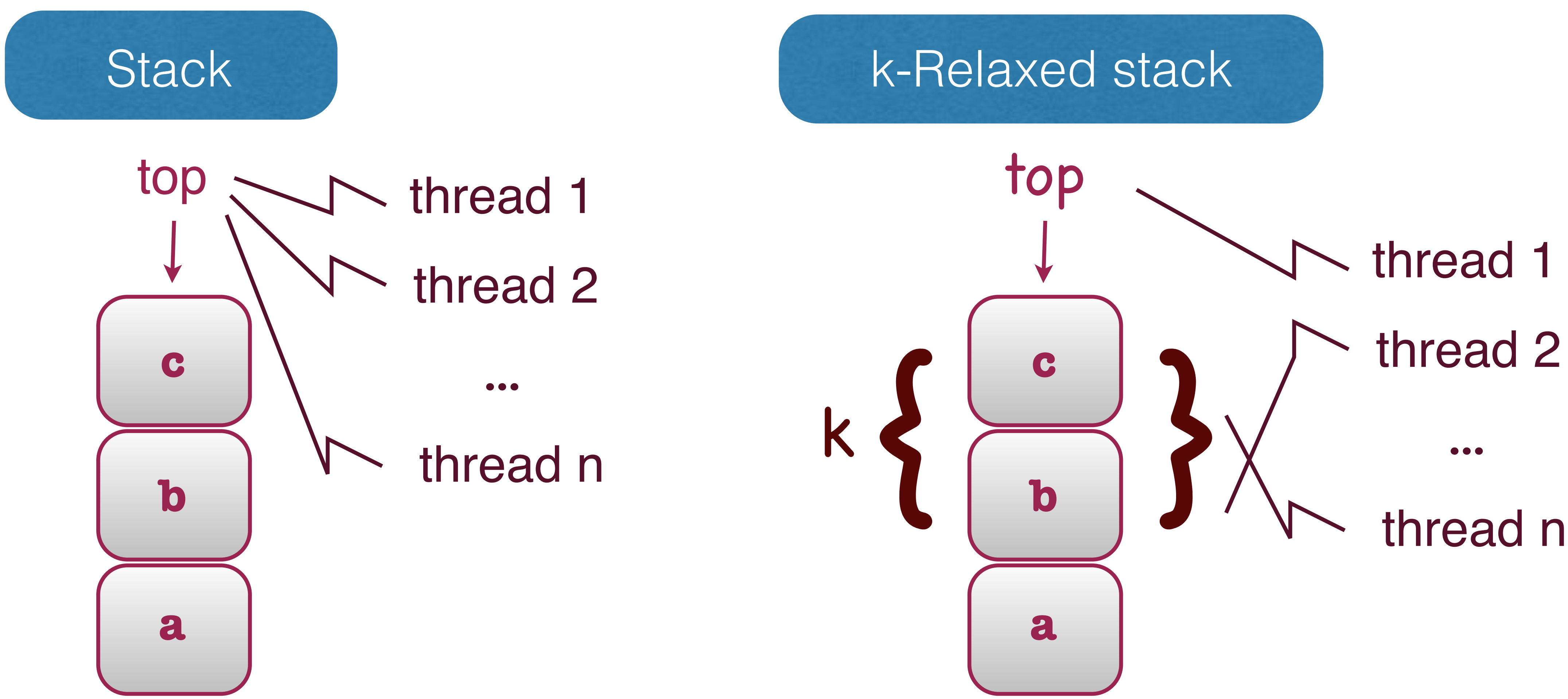
push(a)push(b)push(c)pop(a)pop(b)

- 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

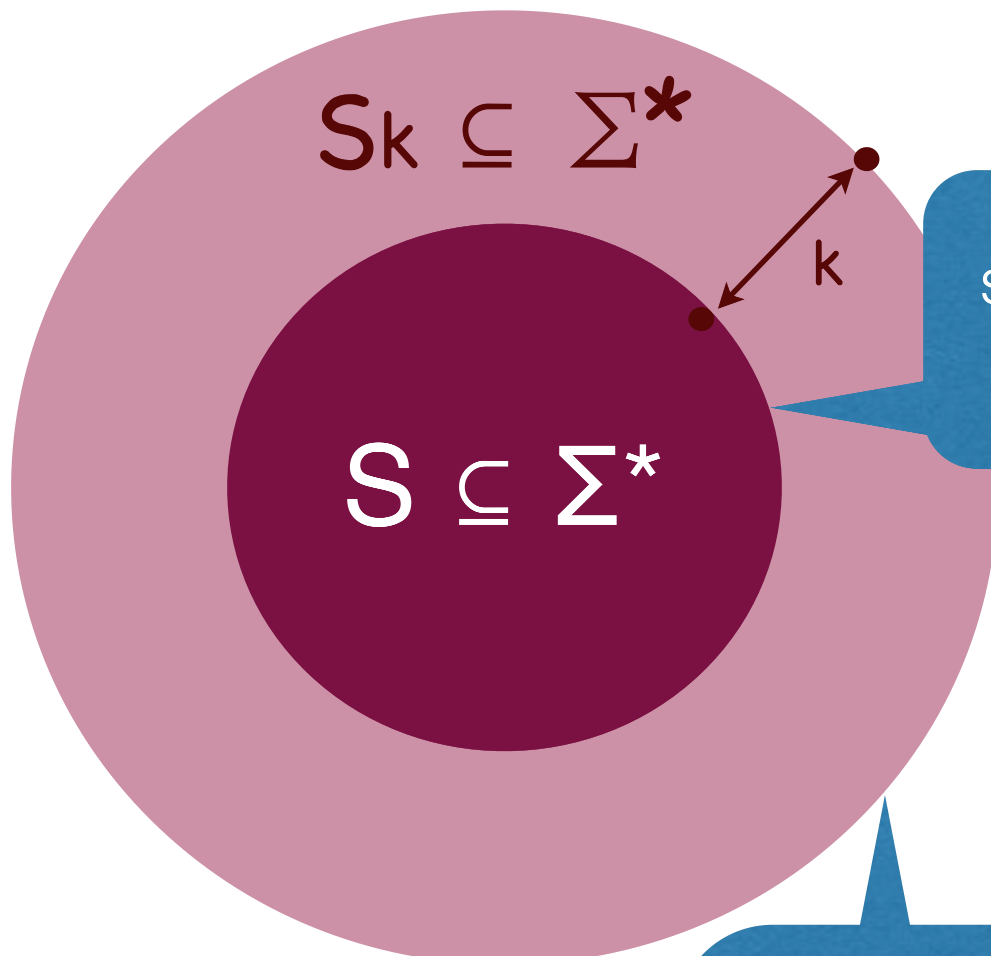
$S \subseteq \Sigma^*$

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 Semantics

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

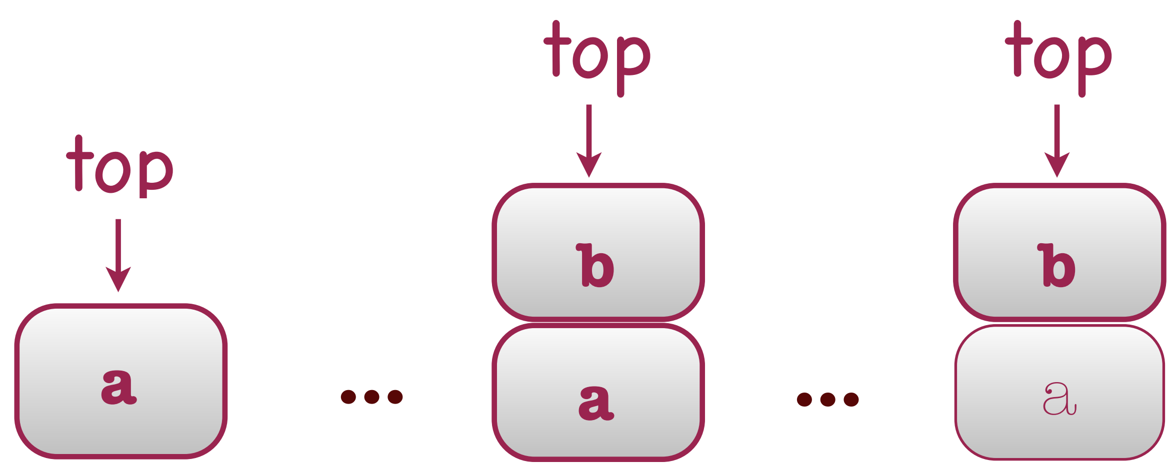
- Sequential specification = set of legal sequences
- Consistency condition = e.g. linearizability / sequential consistency

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

Syntactic distances do not help

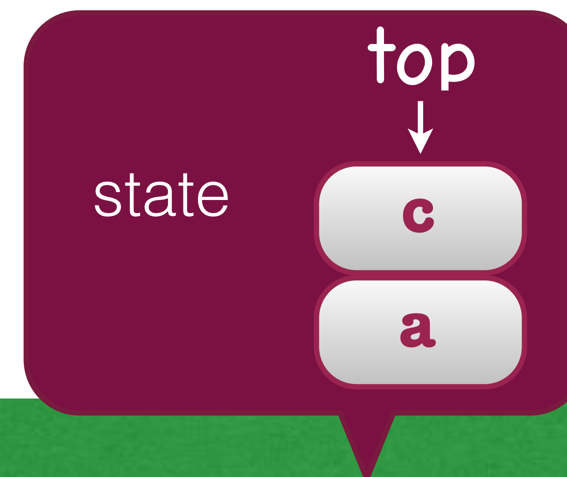
$$\mathrm{push}(a)[\mathrm{push}(i)\mathrm{pop}(i)]^n\mathrm{push}(b)[\mathrm{push}(j)\mathrm{pop}(j)]^m\mathrm{pop}(a)$$

is a 1-out-of-order stack sequence



its permutation distance is min(2n,2m)

Semantic distances need a notion of state



- States are equivalence classes of sequences in S

example: for stack

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

- Two sequences in S are equivalent iff they have an indistinguishable future

$$x \equiv y \quad \Leftrightarrow \quad \forall u \in \Sigma^*.\,(xu \in S \;\Leftrightarrow\; yu \in S)$$

Semantics goes operational

$S \subseteq \Sigma^*$ is the sequential specification

states

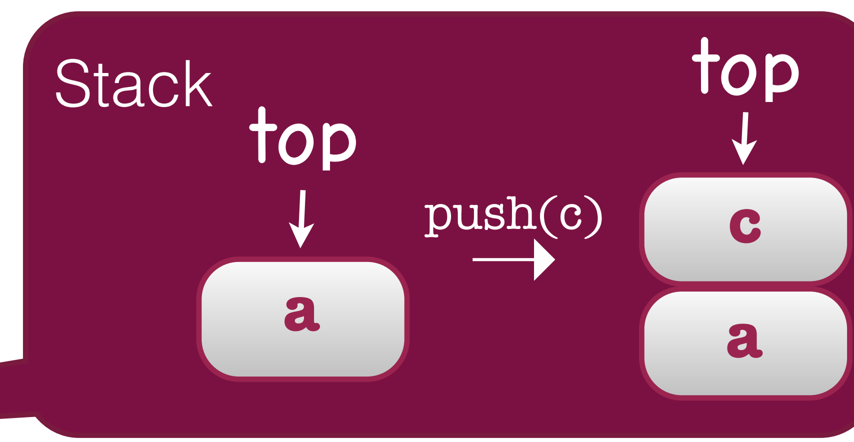
labels

initial state

$LTS(S) = (S/_\equiv, \Sigma, \rightarrow, [\varepsilon]_\equiv)$ with

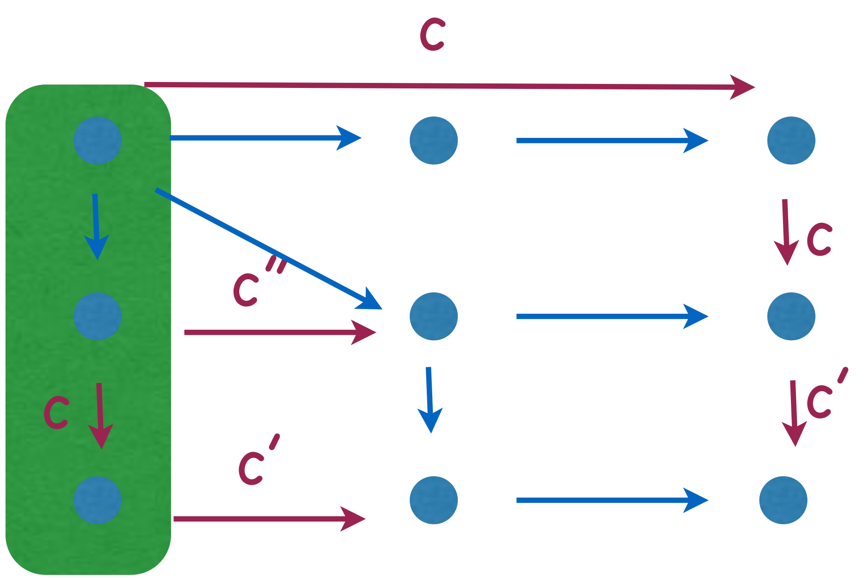
transition relation

$[s]_\equiv \xrightarrow{m} [sm]_\equiv \quad \Leftrightarrow \quad sm \in S$



The relaxation framework

- Start from LTS(S)
- Add transitions with transition costs
- Fix a path cost function



distance = minimal cost on all paths labelled by the sequence

Generic out-of-order

$\mathtt{segment\_cost}(\, q \xrightarrow{m} q' \,) \;=\; |\mathbf{v}|$

transition cost

Where **v** is a sequence of minimal length s.t.

removing **v** enables a transition

or

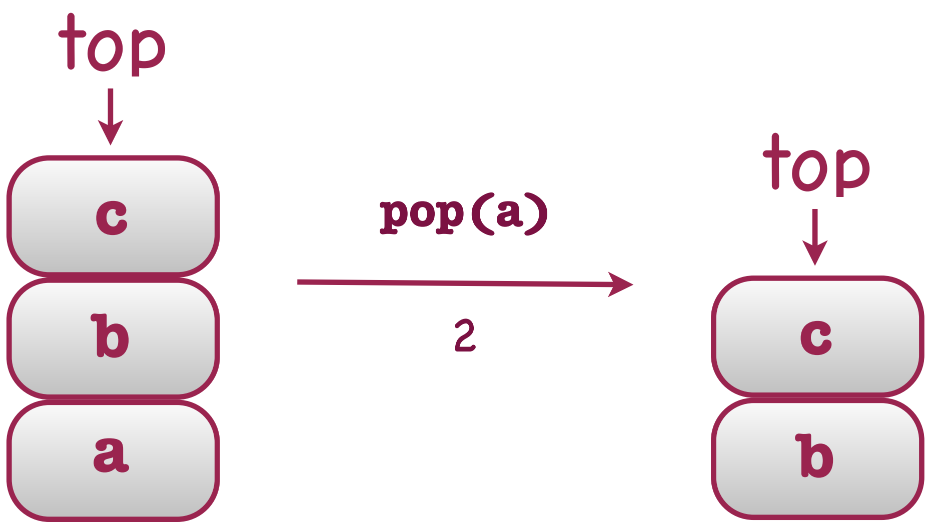
inserting **v** enables a transition

goes with different path costs

Out-of-order stack

Sequence of **push**'s with no matching **pop**

- Canonical representative of a state
- Add incorrect transitions with segment-costs



- Possible path cost functions max, sum,...

also more advanced

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)

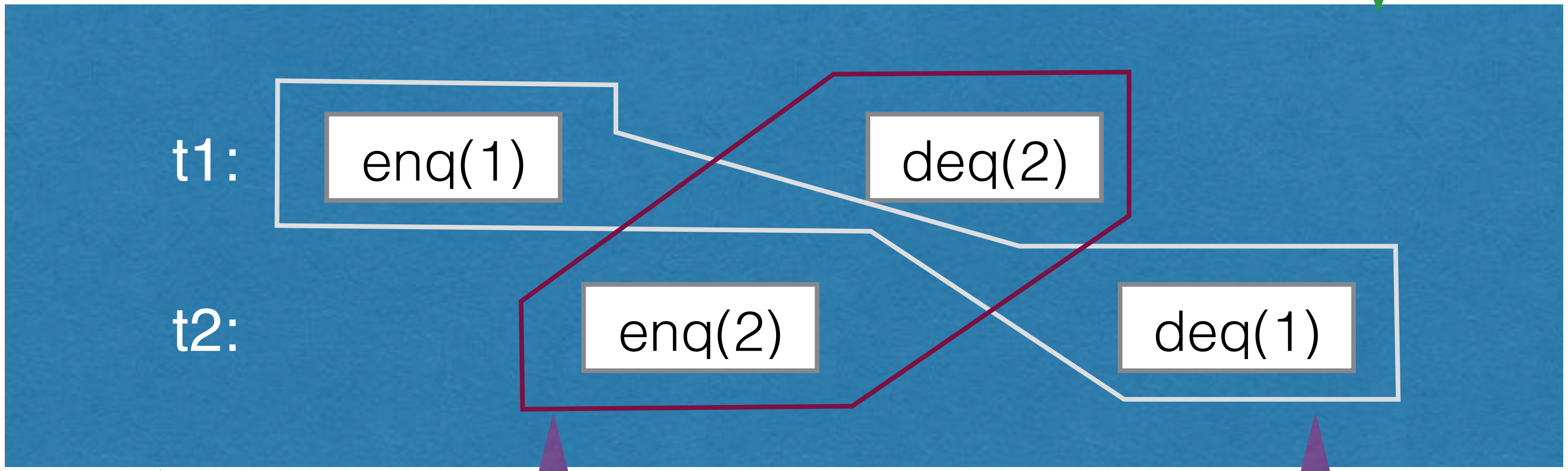
- Partition a history into a set of local histories
- Require linearizability per local history

no global witness

Local sequential consistency… is also possible

Local Linearizability (queue) example

(sequential) history not linearizable



t2-induced history, linearizable

t1-induced history, linearizable

locally linearizable

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

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

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

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

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.

Local Linearizability for Container-Type DS

Signature $\Sigma = Ins \cup Rem \cup SOb \cup DOb$

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

$$I_T = \{m^T \in \mathbf{h} \mid m \in Ins\}$$

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

$$O_T = \{m(a) \in \mathbf{h} \cap Rem \mid i(a)^T \in I_T\} \cup \{m(e) \mid e \in Emp\} \cup \{m(a) \in \mathbf{h} \cap DOb \mid i(a)^T \in I_T\}$$

out-methods of thread T are removes and data-observations (performed by any thread) …-methods

h is locally linearizable iff every thread-induced history

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

is linearizable.

Generalizations of Local Linearizability

Signature Σ

For a history **h** with n threads, choose

$In_{\mathbf{h}}(i)$

$Out_{\mathbf{h}}(i)$

in-methods of thread i, methods that go in $\mathbf{h}_i$

by increasing the in-methods, LL gradually moves to linearizability

out-methods of thread i, dependent methods on the methods in $In_{\mathbf{h}}(i)$ (performed by any thread)

h is locally linearizable iff every thread-induced history

$$\mathbf{h}_i = \mathbf{h} \mid (In_{\mathbf{h}}(i) \cup Out_{\mathbf{h}}(i))$$

is linearizable.

Where do we stand?

In general

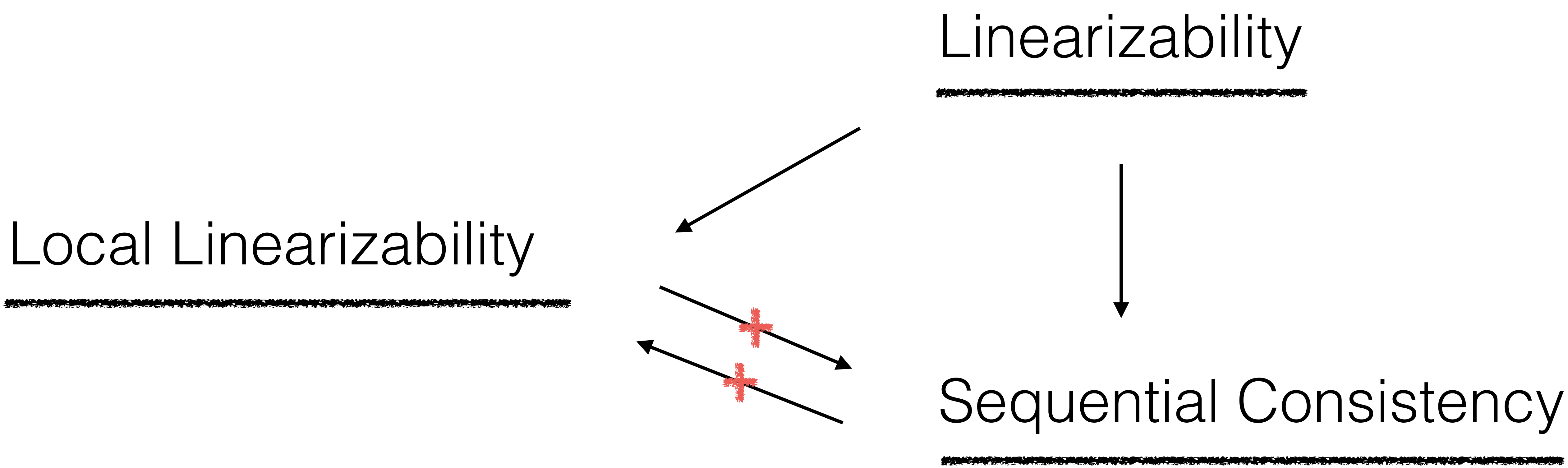
Linearizability

Local Linearizability

Sequential Consistency

Where do we stand?

For queues (and most container-type data structures)



Properties

Local linearizability is compositional

like linearizability
unlike sequential consistency

h (over multiple objects) is locally linearizable
iff
each per-object subhistory of **h** is locally linearizable

Local linearizability is modular / "decompositional"

uses decomposition into smaller histories, by definition

↓

may allow for modular verification

Verification (queue)

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

Verification (queue)

Queue sequential specification (axiomatic)

s is a legal queue sequence

iff

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

Queue local linearizability (axiomatic)

h is queue locally linearizable

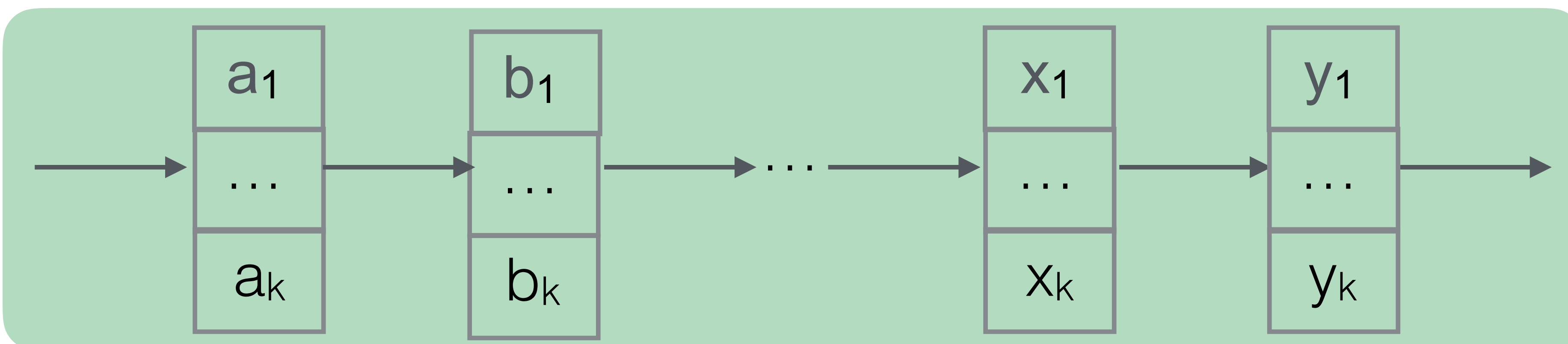
iff

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

thread-local precedence order

Relaxations lead to scalable implementations

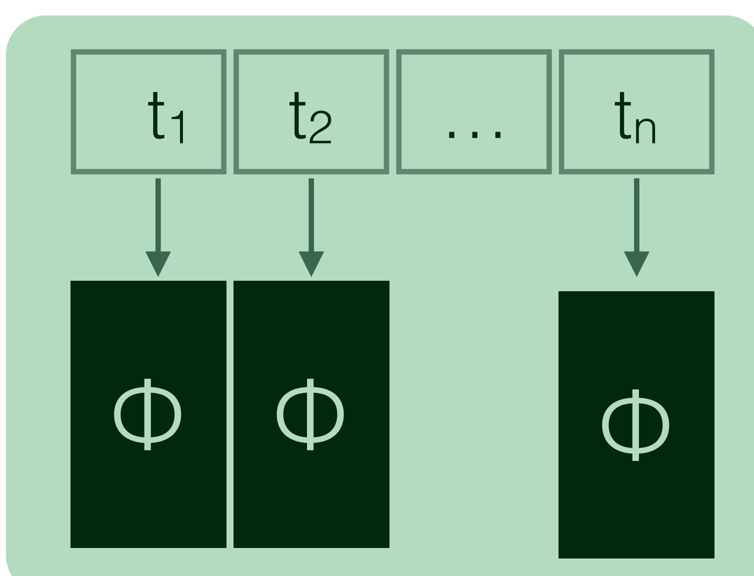
e.g. k-FIFO, k-Stack



k-out-of-order queue

locally linearizable distributed implementation

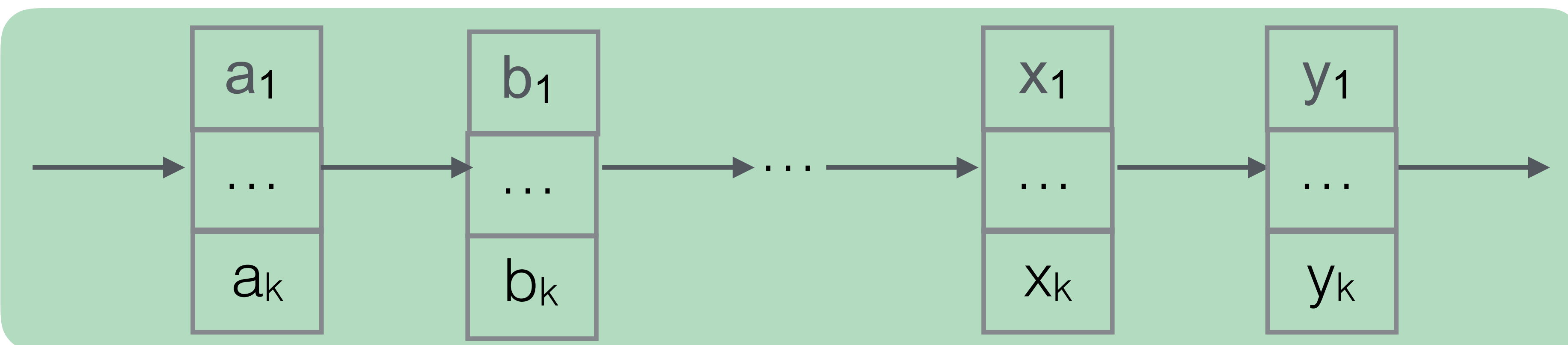
LLD Φ
LL+D Φ



local inserts / global removes

Relaxations lead to scalable implementations

e.g. k-FIFO, k-Stack



k-out-of-order queue

CAS-based algorithm…

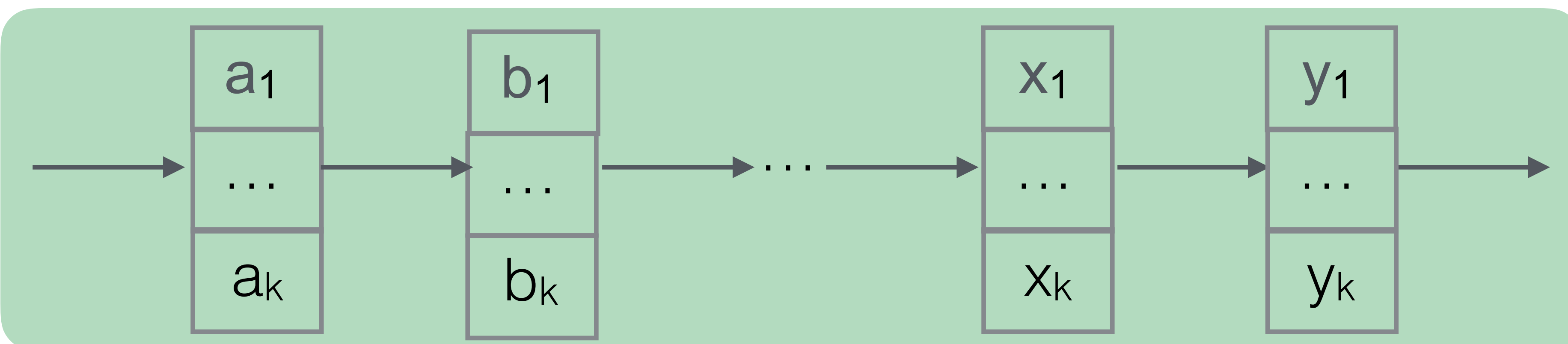
CAS - based

add/remove segment

```
loop:
  read consistent state
  try to add/remove an item (*)
  if successful:
    return
  else:
    try to repair the stack
    goto loop (retry)
```

Relaxations lead to scalable implementations

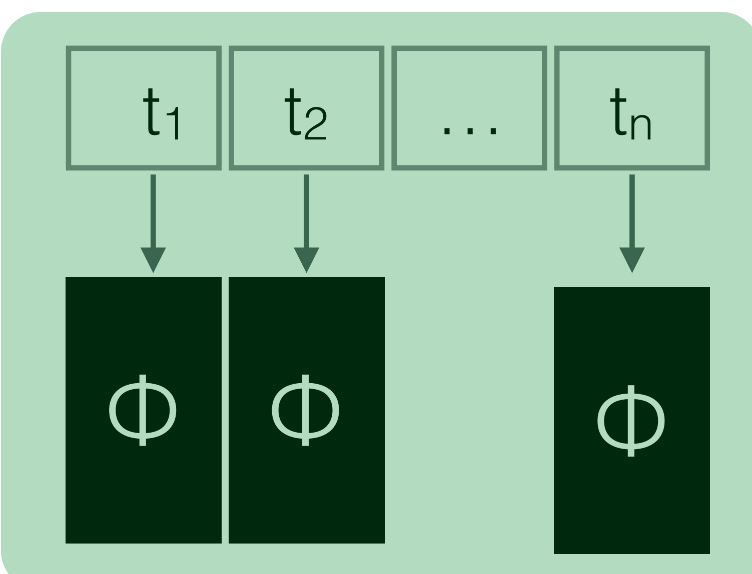
e.g. k-FIFO, k-Stack



k-out-of-order queue

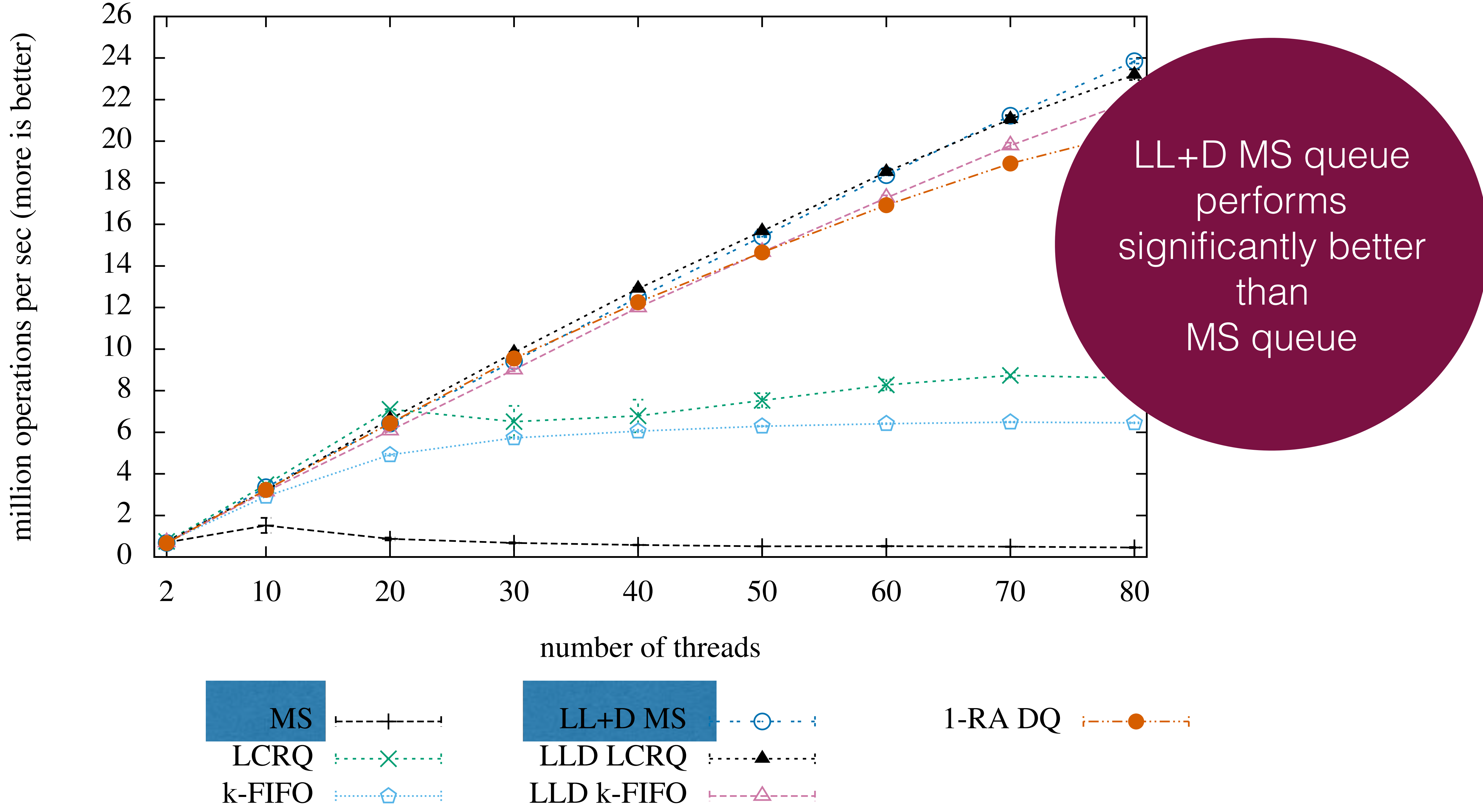
locally linearizable distributed implementation

LLD Φ
LL+D Φ



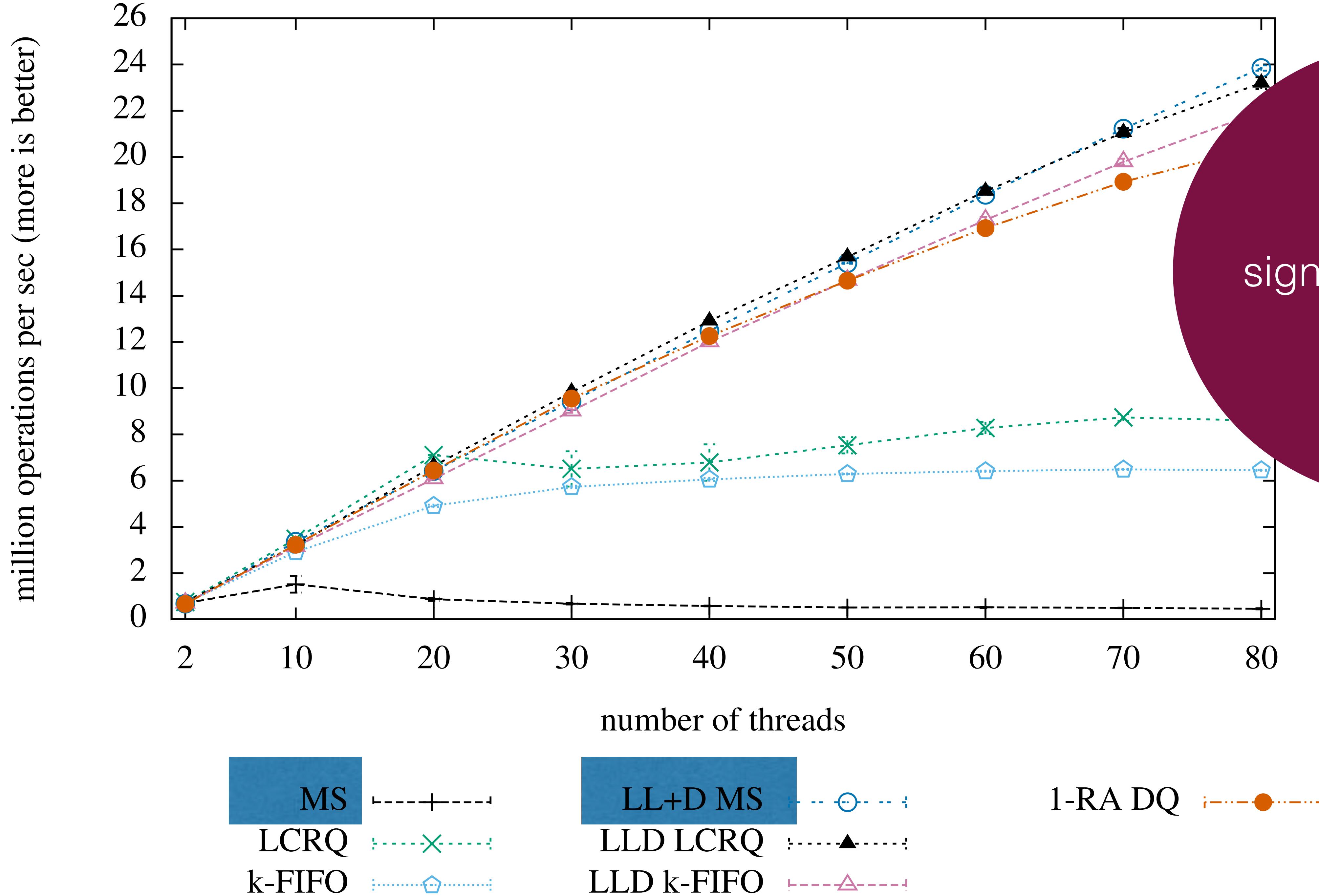
local inserts / global removes

Performance



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

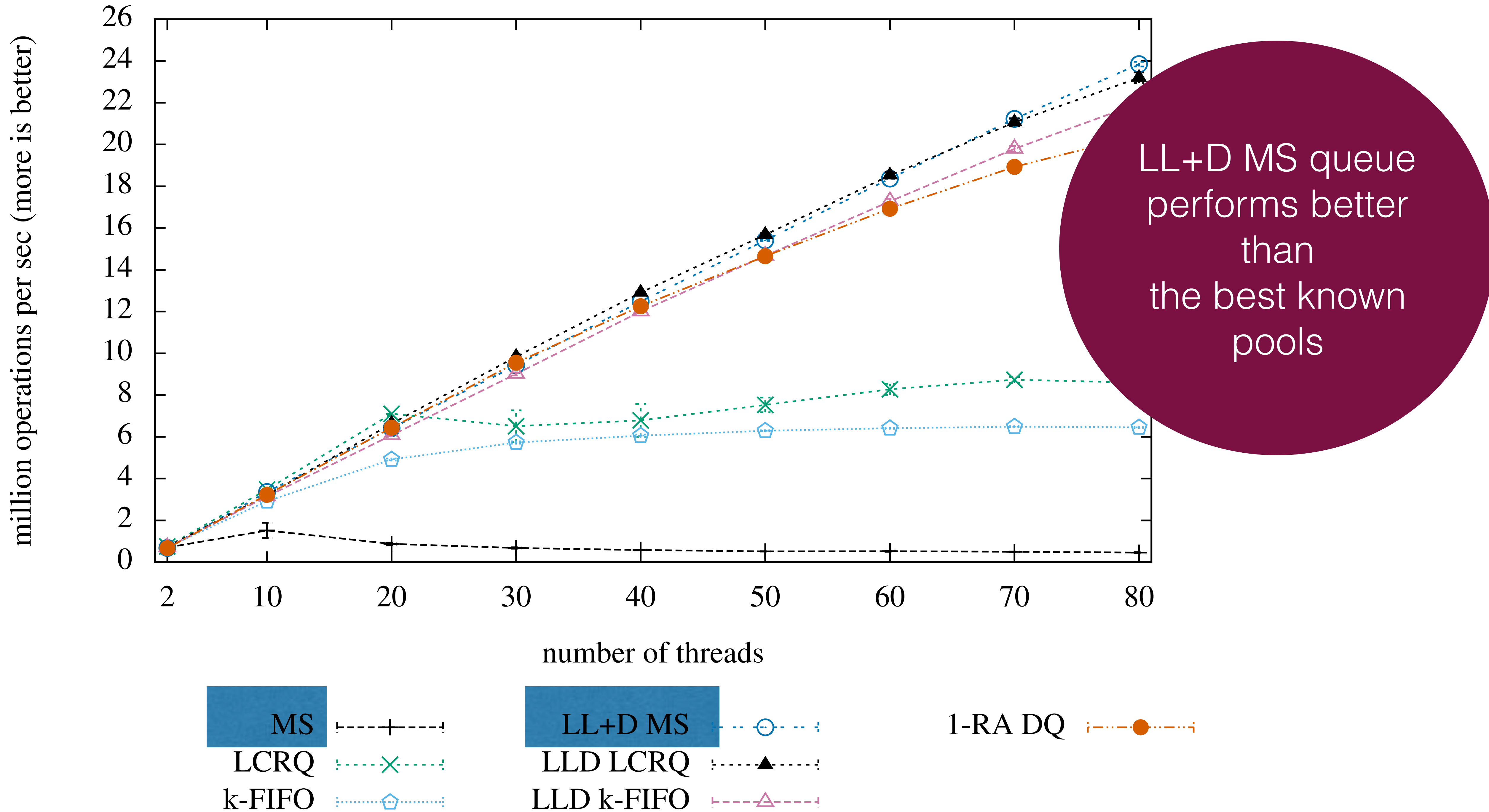
Performance



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

LLD Φ performs significantly better than Φ

Performance



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

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

Queue linearizability (axiomatic)

Henzinger, Sezgin, Vafeiadis CONCUR13

h is queue linearizable

iff

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

precedence order

Problems (stack)

Stack sequential specification (axiomatic)

s is a legal stack sequence

iff

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

Stack linearizability (axiomatic)

h is stack linearizable

iff

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

???

Problems (stack)

Stack sequential specification (axiomatic)

s is a legal stack sequence

iff

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

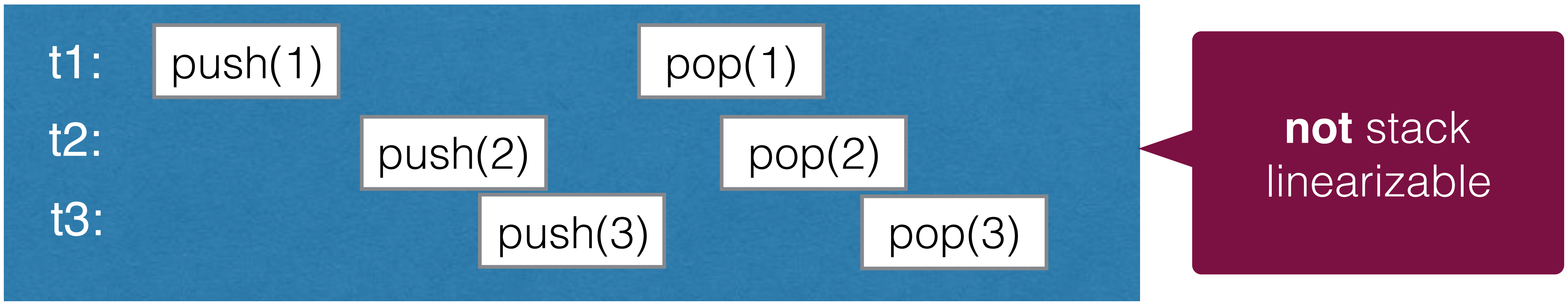
Stack linearizability (axiomatic)

~~**h** is stack linearizable~~

~~iff~~

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

Problems (stack)



Stack linearizability (axiomatic)

h is stack linearizable

iff

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

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.

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

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

Pool linearizability (axiomatic)

h is pool (without empty removals) linearizable
iff
1. $rem(x) \in \mathbf{h} \Rightarrow ins(x) \in \mathbf{h} \wedge rem(x) \not<_{\mathbf{h}} ins(x)$

CV violations
= V violations

Queue without empty removals

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

Queue sequential specification (axiomatic)

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

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

Queue linearizability (axiomatic)

h is queue (without empty removals) linearizable
iff

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

CV violations
= V violations

Pool

infinite inductive violations

V violations
rem(x) $<_s$ ins(x)
and
ins(x) $<_s$ rem(⊥) $<_s$ rem(x)

Pool sequential specification (axiomatic)

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

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

Pool linearizability (axiomatic)

h is pool (with 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. … …

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

infinite inductive violations

Queue

V violations
rem(x) $<_s$ ins(x)
and
ins(x) $<_s$ rem(⊥) $<_s$ rem(x)
and
enq(x) $<_s$ enq(y) ∧
deq(y) $<_s$ deq(x)

Queue sequential specification (axiomatic)

s is a legal queue (with empty removals) sequence

iff

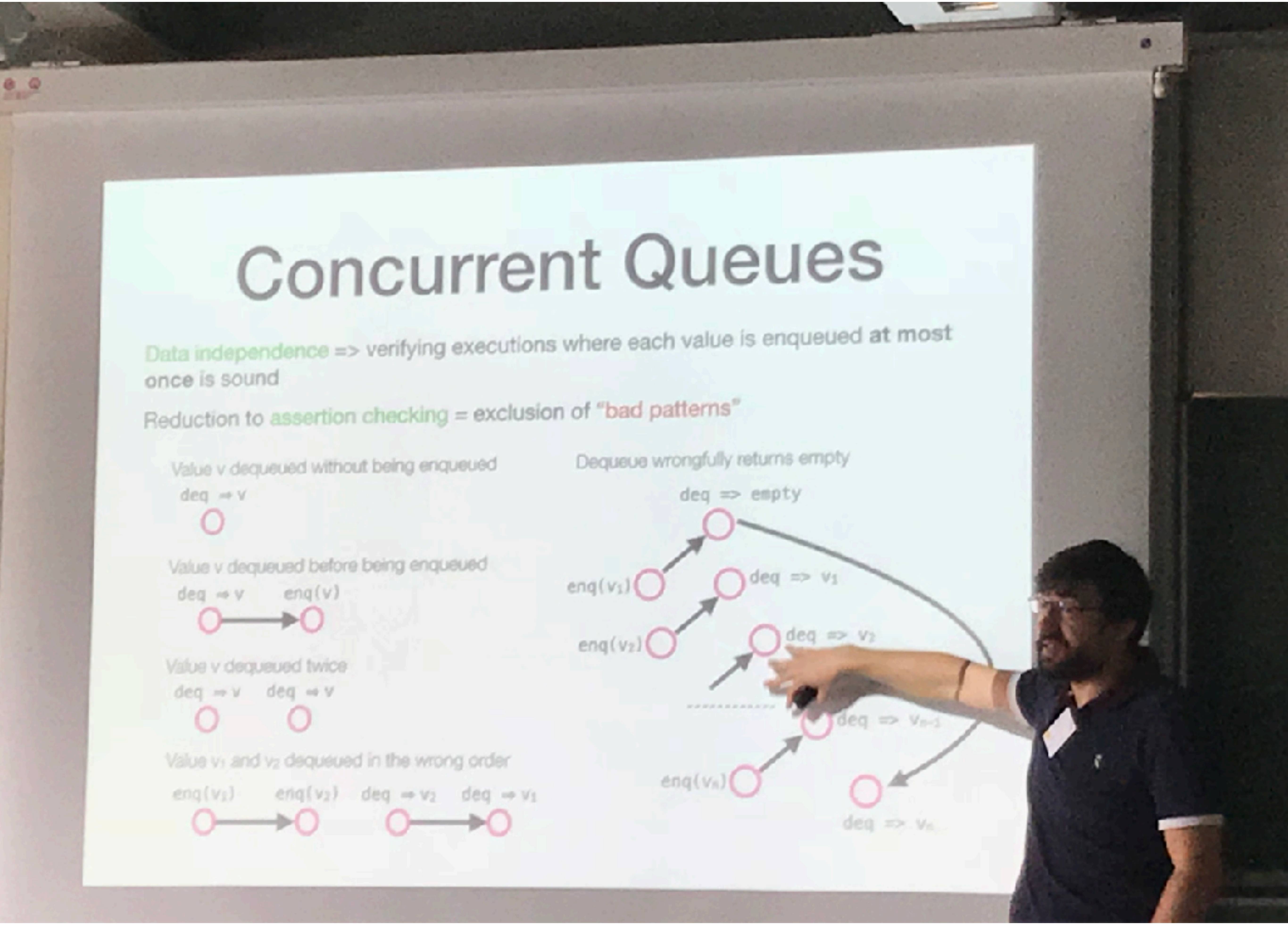
1. deq(x) ∈ **s** ⇒ enq(x) ∈ **s** ∧ enq(x) $<_s$ deq(x)
2. deq(⊥) $<_s$ deq(x) ⇒ deq(⊥) $<_s$ enq(x) ∧ enq(x) $<_s$ deq(⊥) ⇒ deq(x) $<_s$ deq(⊥)
3. enq(x) $<_s$ enq(y) ∧ deq(y) ∈ **s** ⇒ deq(x) ∈ **s** ∧ deq(x) $<_s$ deq(y)

Queue linearizability (axiomatic)

h is queue (with empty removals) linearizable

iff

1. deq(x) ∈ **h** ⇒ enq(x) ∈ **h** ∧ deq(x) $\not<_h$ enq(x)
2. infinitely many CV violations
enq(x_1) $<_h$ deq(⊥) ∧ enq(x_2) $<_h$ deq(x_1) ∧ … ∧ enq(x_{n+1}) $<_h$ deq(x_n) ∧ deq(⊥) $<_h$ deq(x_{n+1})
3. enq(x) $<_h$ enq(y) ∧ deq(y) ∈ **h** ⇒ deq(x) ∈ **h** ∧ deq(y) $\not<_h$ deq(x)


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

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

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

The axioms

Proposition PO

$\mathcal{V}$ is PO-closed iff

$$(\mathrm{C1}) \quad \mathcal{D} \subseteq \mathcal{V}$$

$$(\mathrm{C2}) \quad \forall N \in \mathcal{V}.\ \forall M.\ (N \subseteq \mathrm{tr}(M) \Rightarrow M \in \mathcal{V})$$

Proposition IO

$\mathcal{V}$ is IO-closed iff (C1) and (C2) and

$$(C3) \quad \forall M \in \mathbb{X} \backslash \mathcal{C}.\ \forall a, b, c, d \in X.\ a \neq d \wedge c \neq b \Rightarrow$$
$$[(a, b) \in M \wedge (c, d) \in M \wedge M \cup \{(a, d)\} \in \mathcal{V} \wedge M \cup \{(c, b)\} \in \mathcal{V} \Rightarrow M \in \mathcal{V}]$$

Proposition TO

$\mathcal{V}$ is TO-closed iff (C1) and (C2) and

$$(\mathrm{C4}) \quad \forall N \in \mathcal{V}, N \cap \Delta \neq \emptyset.\ \forall M \in \mathbb{X}.\ \forall a \in N \backslash M.\ \exists a_1, a_2 \in X.$$
$$a = (a_1, a_2) \wedge M \cup \{(a_2, a_1)\} \in \mathcal{V} \Rightarrow M \in \mathcal{V}$$

How does it work ?

Theorem

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

(1) $\bigstar$

(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 $\bigstar$ 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 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