Concurrent Data Structures

Semantics and Quantitative Relaxations

Mike Dodds	University of York
Andreas Haas	University of Salzburg
Tom Henzinger	IST Austria
Andreas Holzer	TU Vienna
Christoph Kirsch	University of Salzburg
Michael Lippautz	University of Salzburg
Hannes Payer	University of Salzburg
Ali Sezgin	IST Austria
Ana Sokolova	University of Salzburg

FRIDA, VSL 23.7.2014

Semantics of sequential data structures

e.g. pools, queues, stacks

- Sequential specification - set of legal sequences

Stack - legal sequence

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

Semantics of concurrent data structures

Stack - legal sequence

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

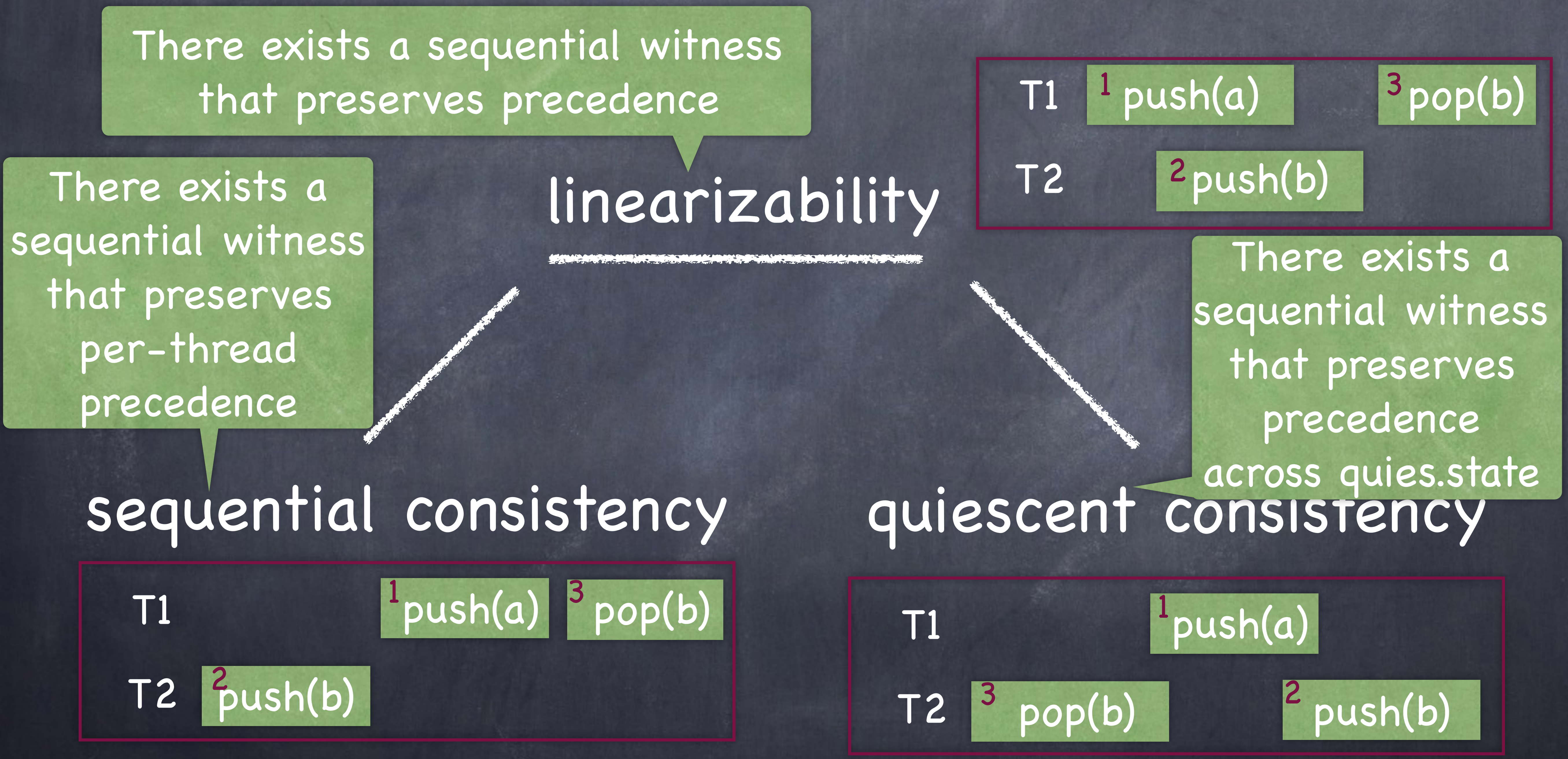
- Sequential specification - set of legal sequences
- Consistency condition - e.g. linearizability

linearizable
wrt seq.spec.

Stack - concurrent history

begin-push(a)begin-push(b) end-push(a) end-push(b)begin-pop(b)end-pop(b)

Consistency conditions



There exists a sequential witness that preserves precedence

linearizability

T1 ^{1}push(a) ^{3}pop(b)

T2 ^{2}push(b)

There exists a sequential witness per-thread that preserves precedence

sequential consistency

T1 ^{1}push(a) ^{3}pop(b)

T2 ^{2}push(b)

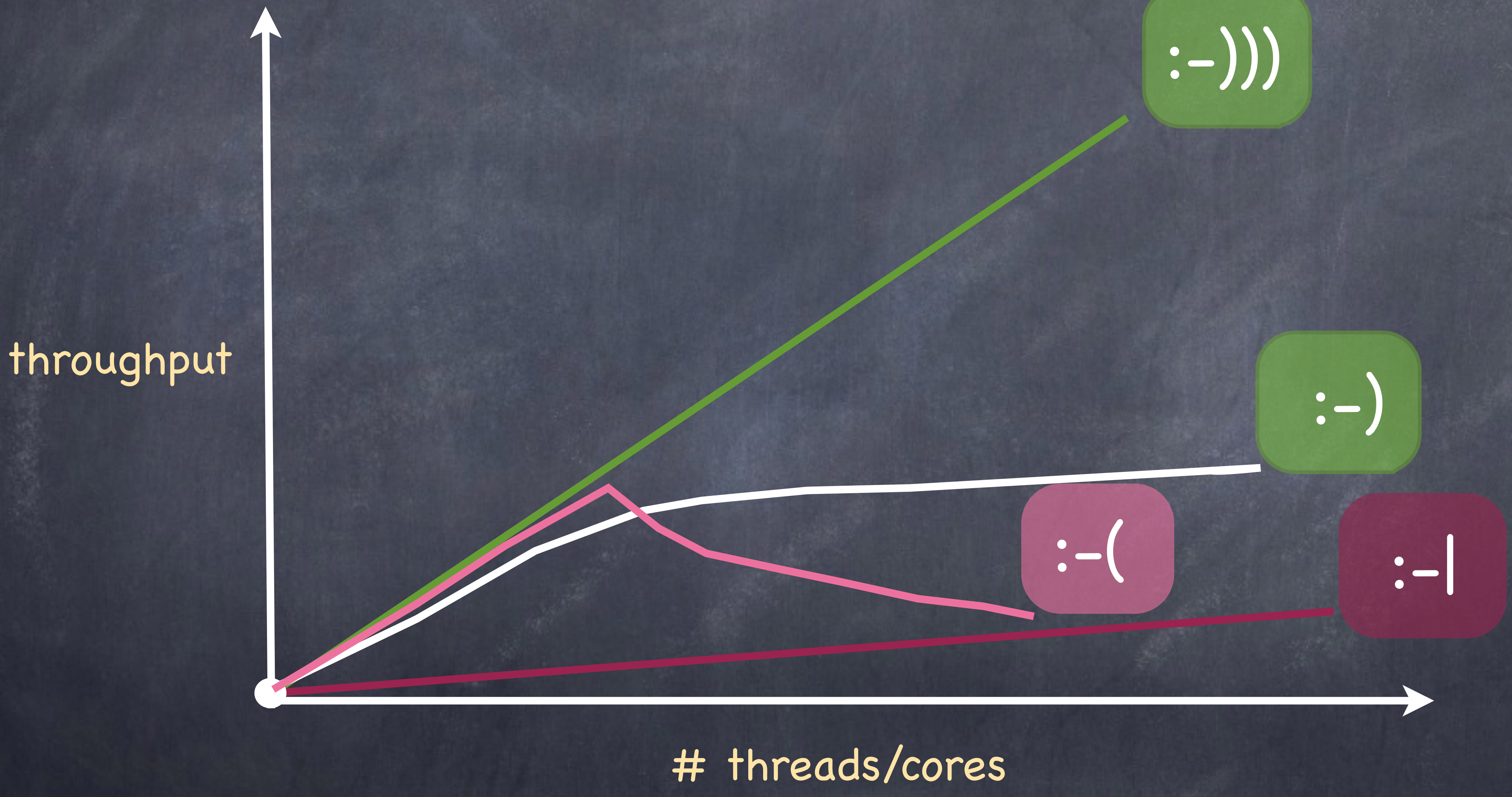
There exists a sequential witness that preserves precedence across quies.state

quiescent consistency

T1 ^{1}push(a)

T2 ^{3} pop(b) ^{2} push(b)

Performance and scalability



Relaxations allow

Stack - incorrect behavior

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

- Trading correctness for performance
- In a controlled way with quantitative bounds

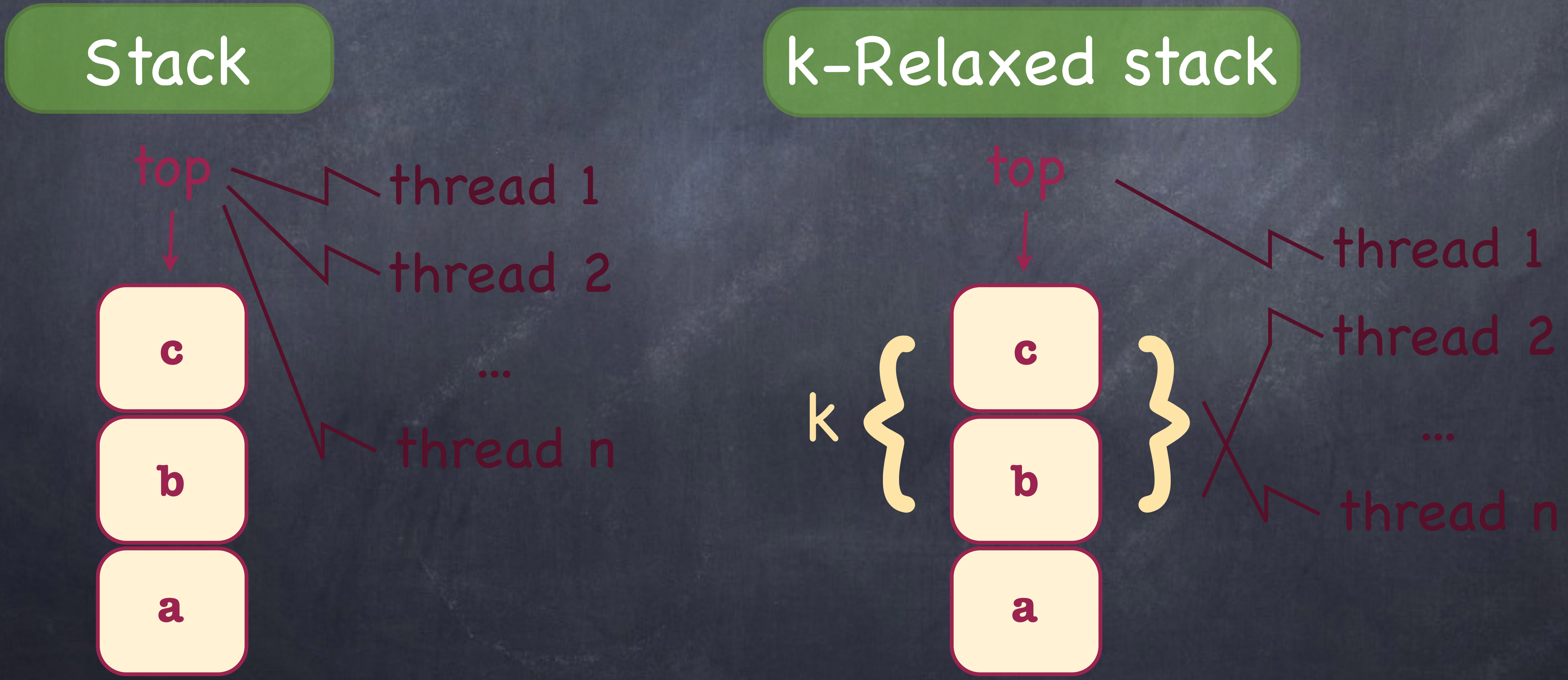
correct in a relaxed stack

... 2-relaxed? 3-relaxed?

measure the error from correct behavior

Why relax?

- It is interesting
- Provides potential for better performing concurrent implementations



Relaxations of concurrent data structures

Quantitative relaxations
Henzinger, Kirsch, Payer, Sezgin, S.
POPL 2013

- Sequential specification - set of legal sequences
- Consistency condition - e.g. linearizability

(Quantitative) relaxations
Dodds, Sezgin, S.
work in progress

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

$S \subseteq \Sigma^*$

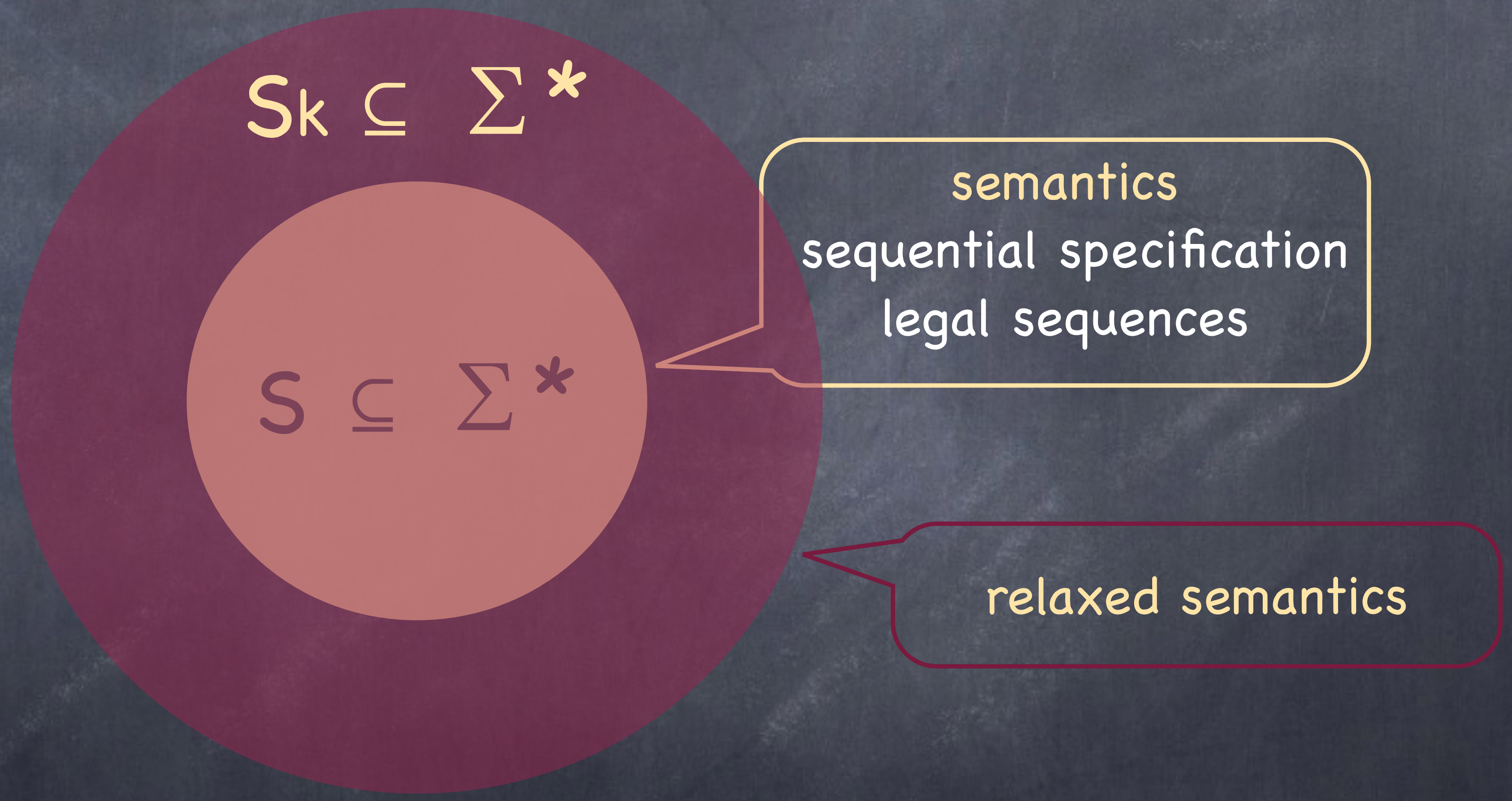
semantics

sequential specification

legal sequences

Σ - methods with arguments

The big picture



$S_k \subseteq \Sigma^*$

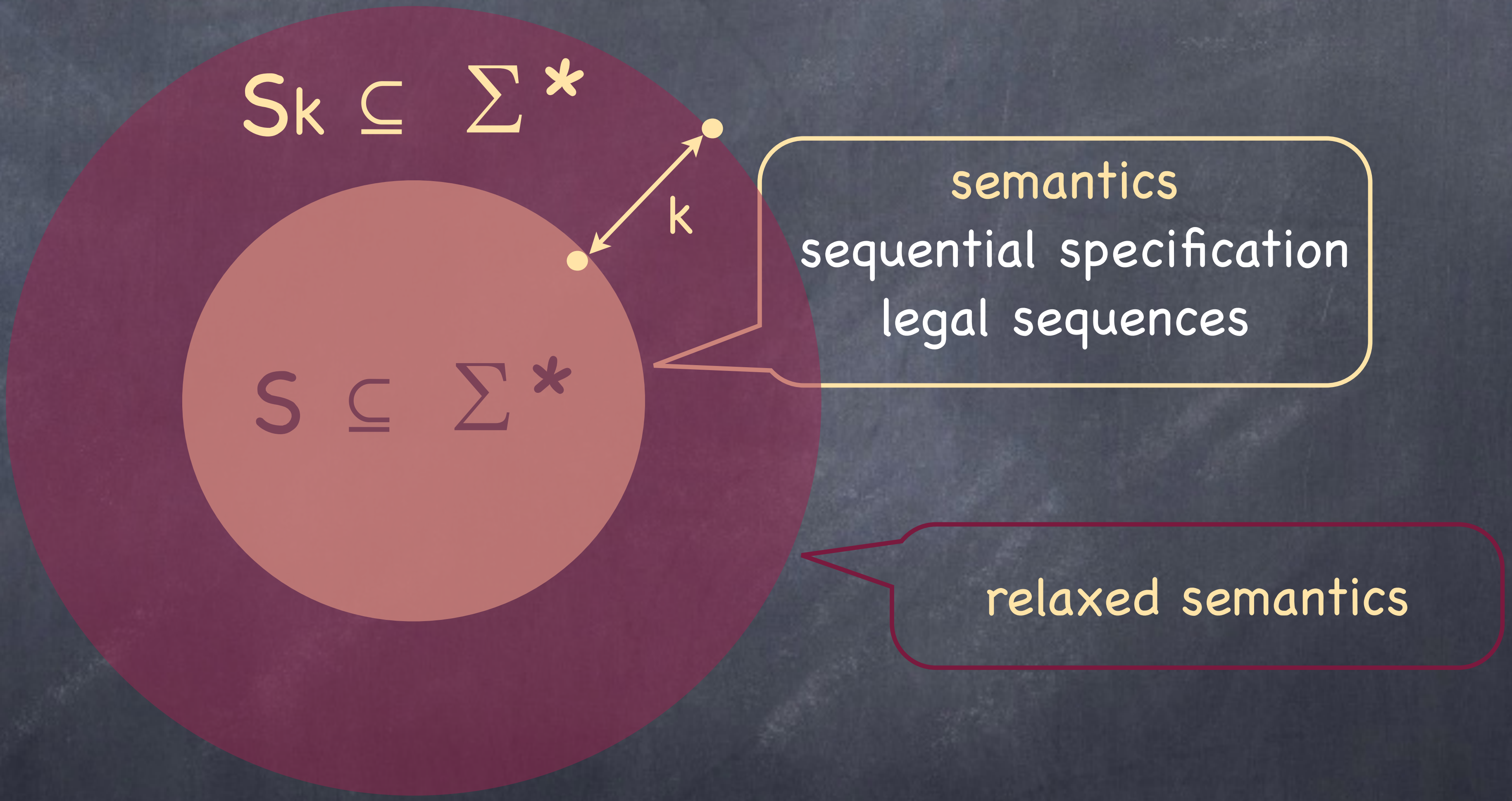
$S \subseteq \Sigma^*$

semantics
sequential specification
legal sequences

relaxed semantics

Σ - methods with arguments

The big picture



semantics
sequential specification
legal sequences

relaxed semantics

Σ - methods with arguments

distance?

Challenge

There are natural concrete relaxations...

Stack

Each **pop** pops one of the (k+1)-youngest elements

Each **push** pushes .....

k-out-of-order relaxation

Challenge

There are natural concrete relaxations...

Stack

Each **pop** pops one of the (k+1)-youngest elements

Each **push** pushes .....

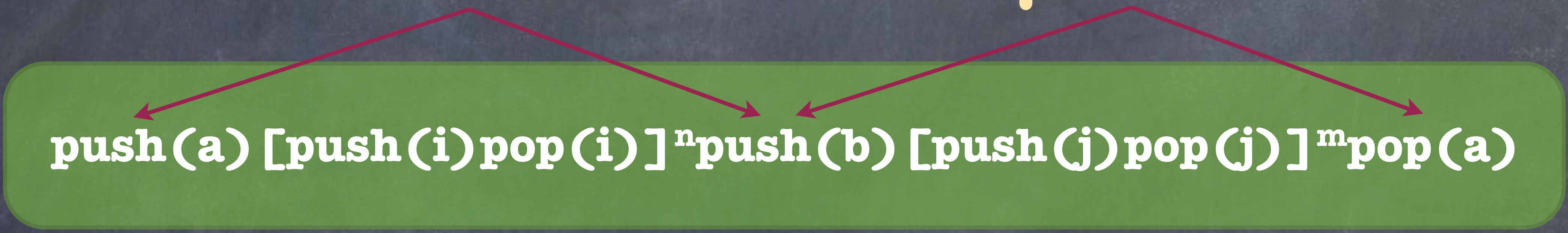
k-out-of-order relaxation

makes sense also for queues, priority queues, ...

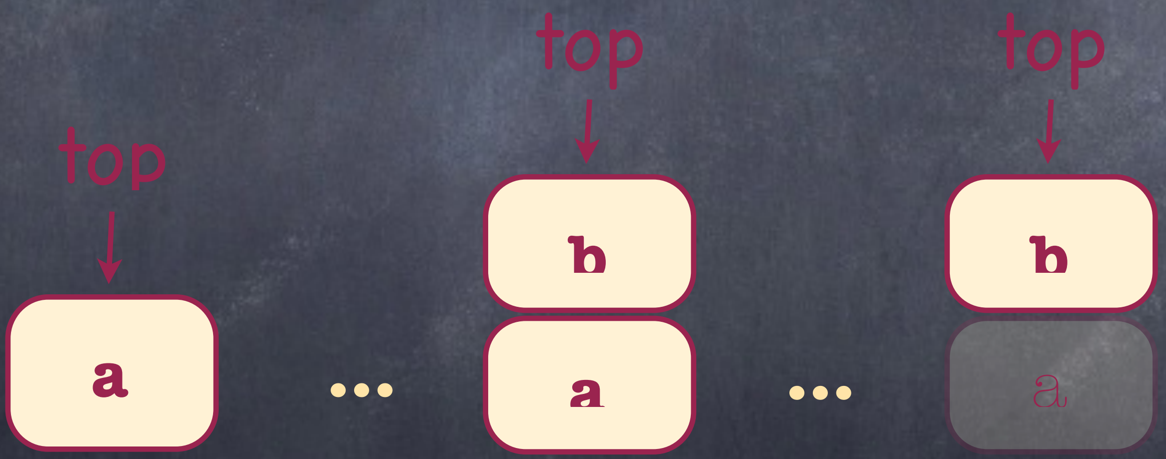
How is it reflected by a distance between sequences?

one distance for all?

Syntactic distances do not help

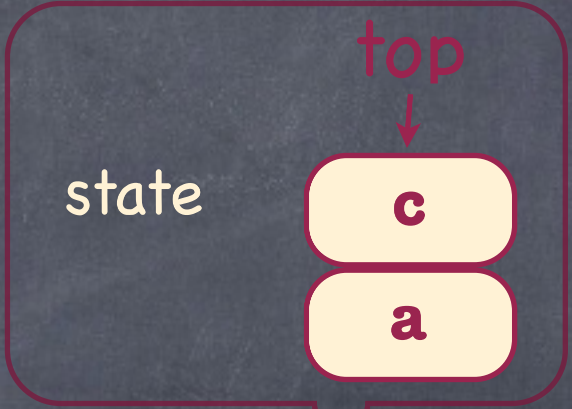

push(a)[push(i)pop(i)]^{n}push(b)[push(j)pop(j)]^{m}pop(a)

is a 1-out-of-order stack sequence



its permutation distance is min(n,m)

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 if 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

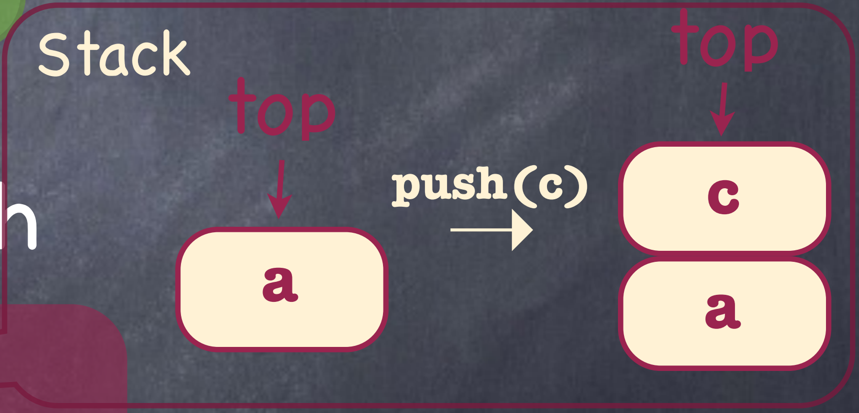
- $S \subseteq \Sigma^*$ is the sequential specification

states

labels

initial state

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



transition relation

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

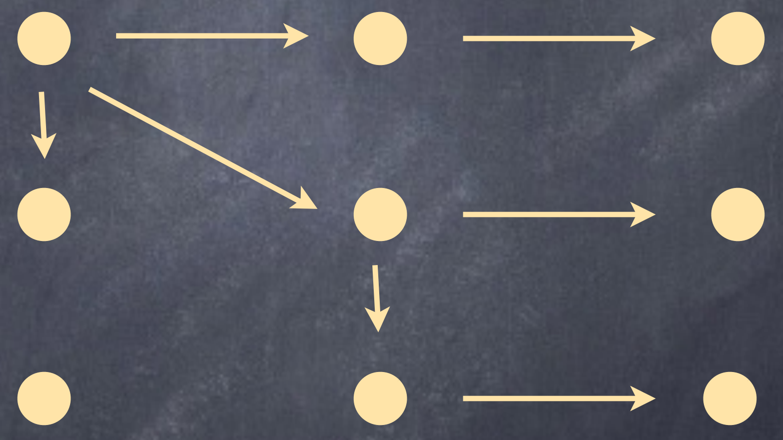
The framework

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

The framework

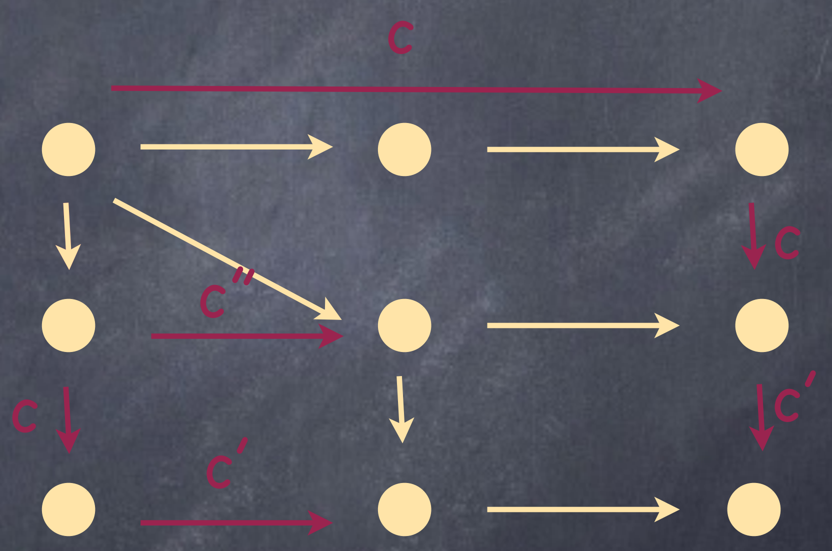
- Start from LTS(S)
- Add transitions with transition costs
- Fix a path cost function

Σ - singleton



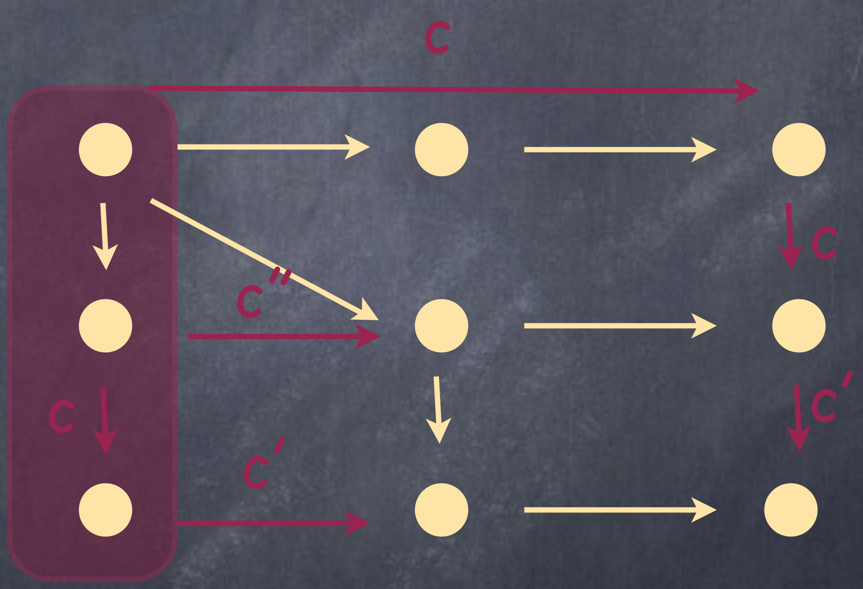
The framework

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



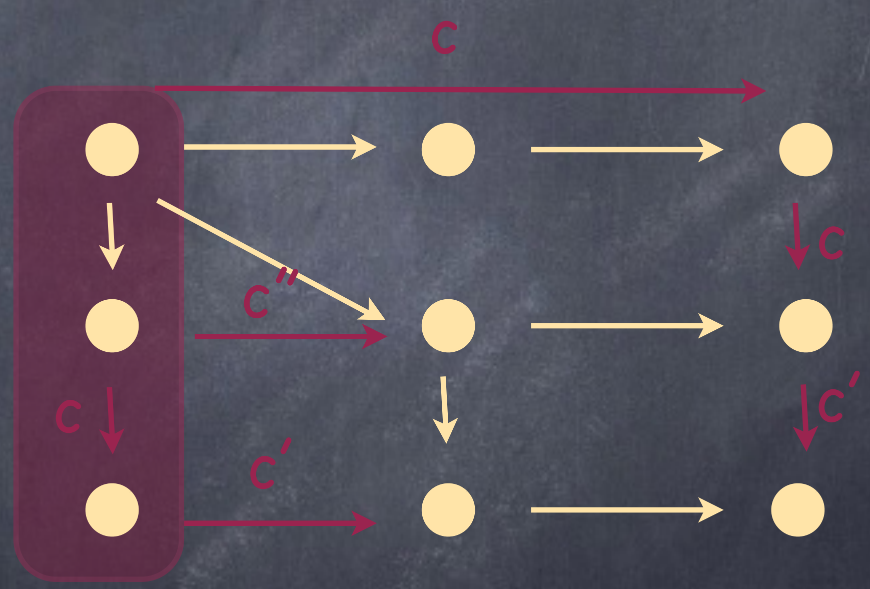
The framework

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



The framework

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

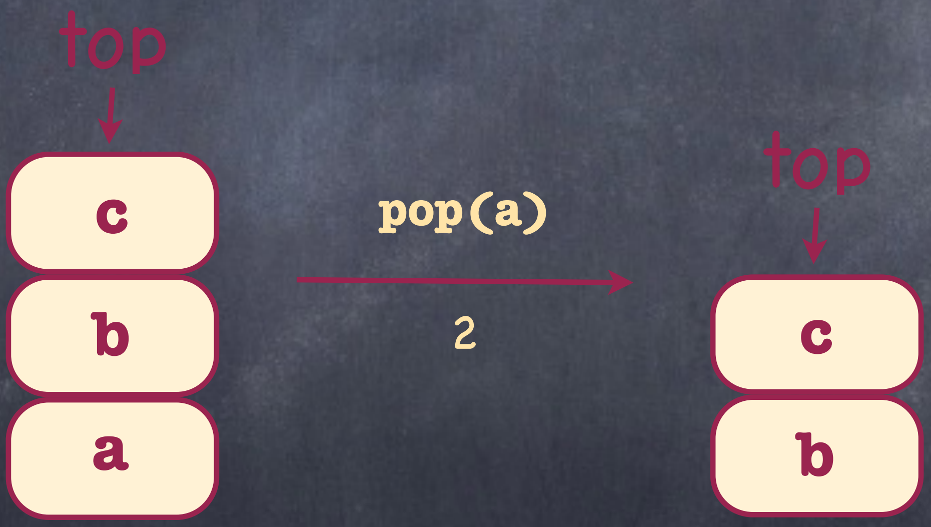


distance - minimal cost on all paths labelled by the sequence

Out-of-order stack

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

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



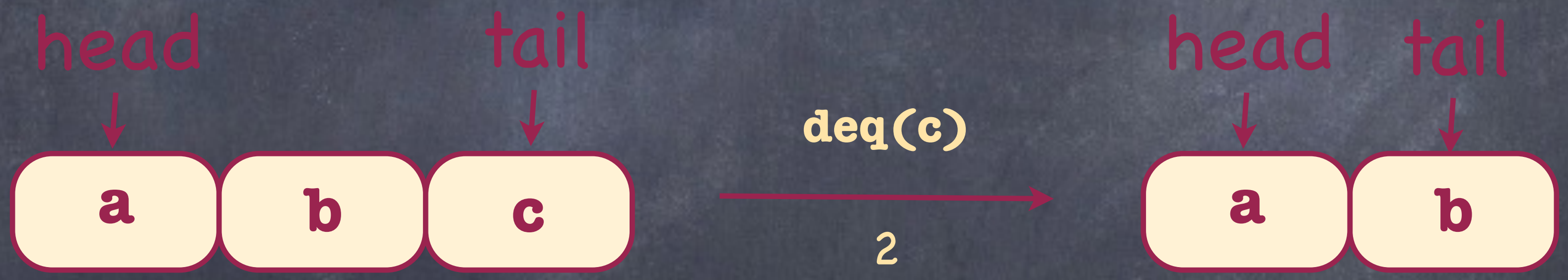
- Possible path cost functions max, sum,...

also more advanced

Out-of-order queue

Sequence of **enq**'s with no matching **deq**

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



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

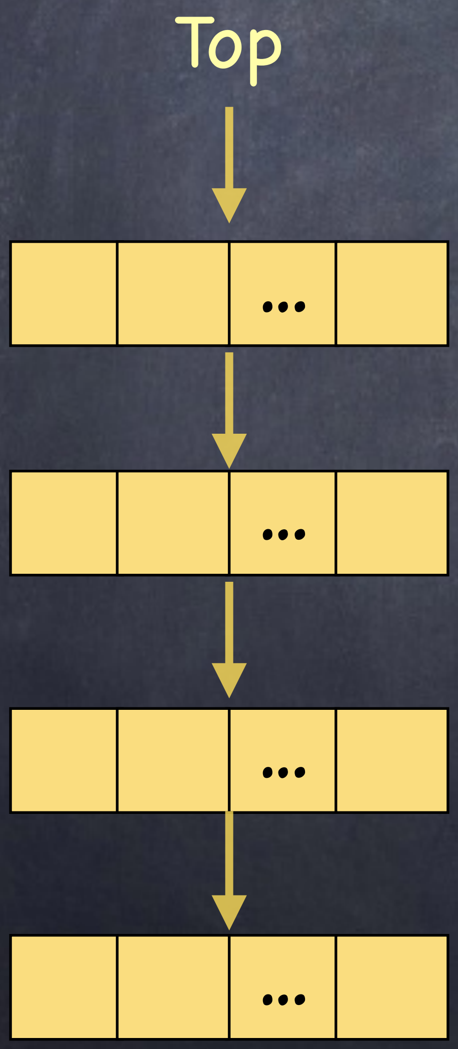
also more advanced

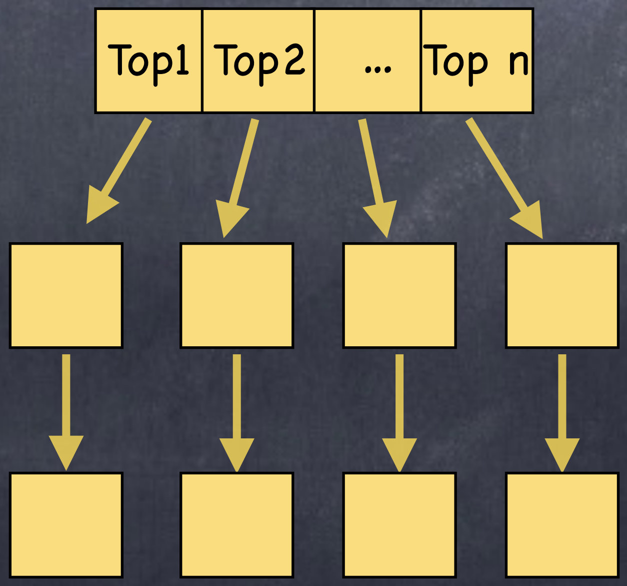
Implementations and Performance

Relaxed implementations

k-Stack
Henzinger, Kirsch, Payer, Sezgin, S.
POPL 2013

Distributed queues / stacks
Haas, Henzinger, Kirsch, Lippautz, Payer, Sezgin, S.
CF 2013

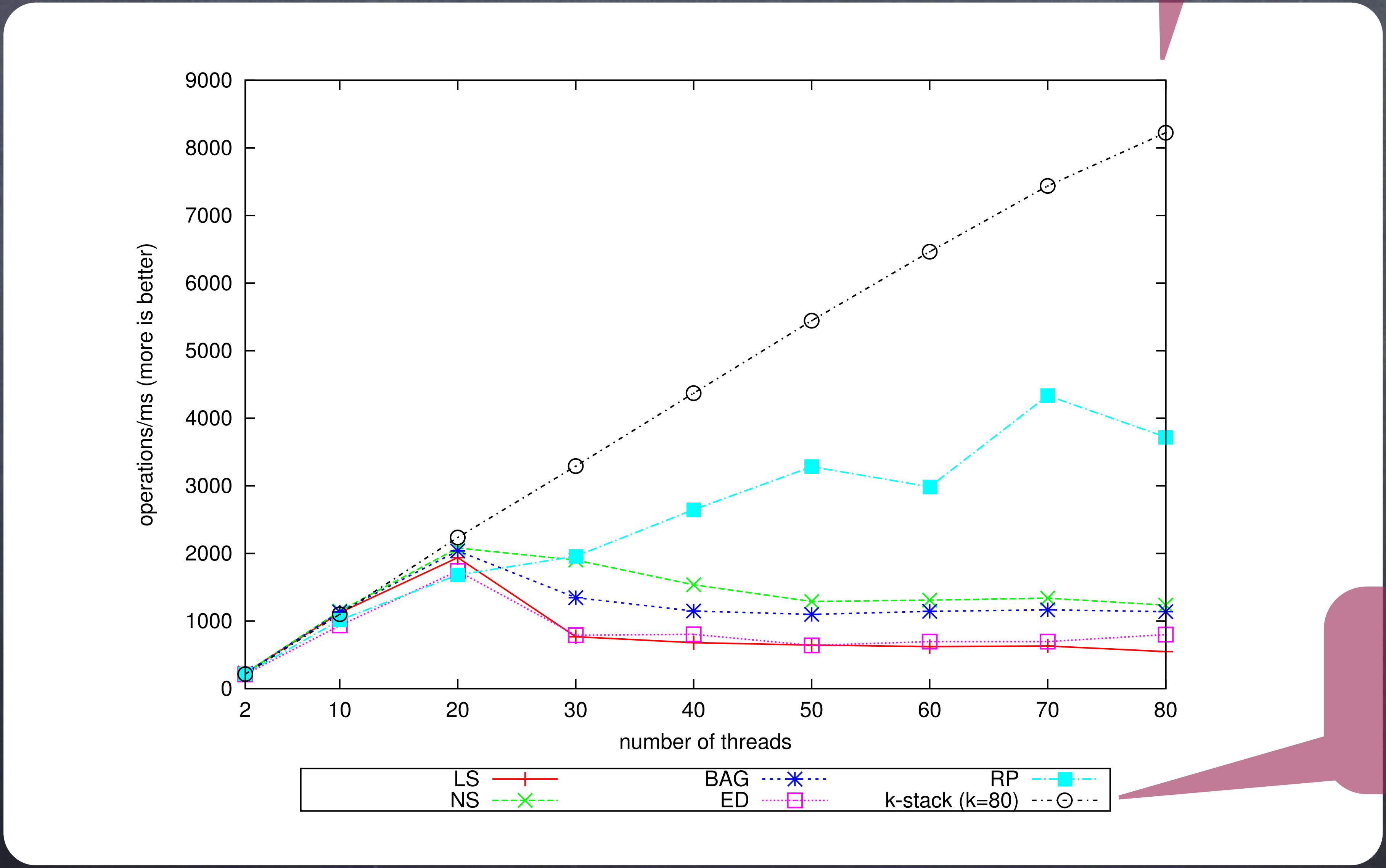




k-Stack

Performance and Scalability comparison

"80"-core machine

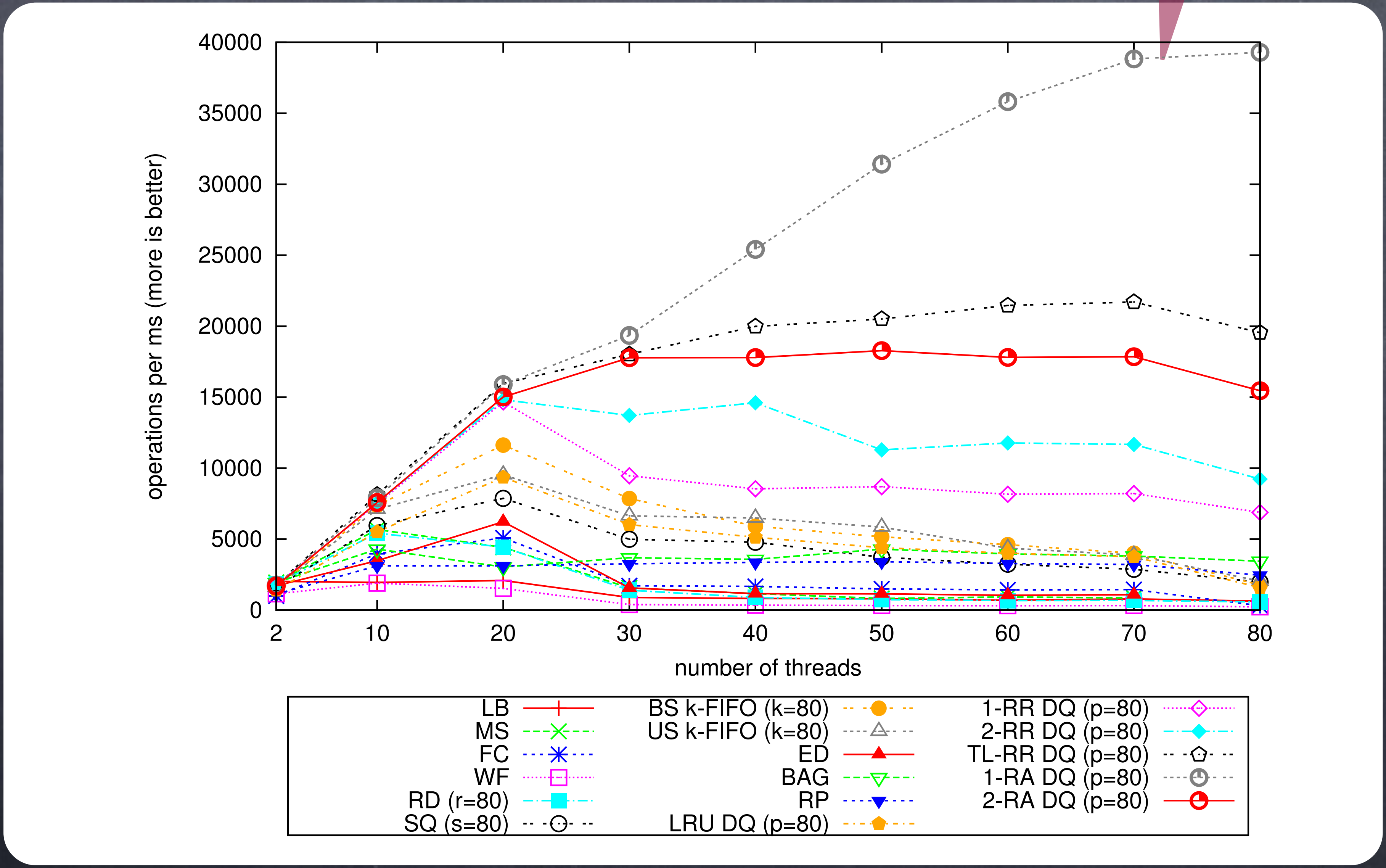


lock-free segment stack

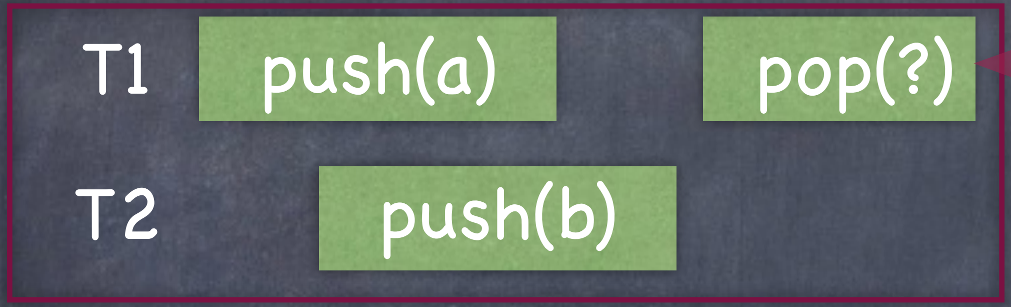
Distributed queues

Performance and Scalability comparison

“80”-core machine

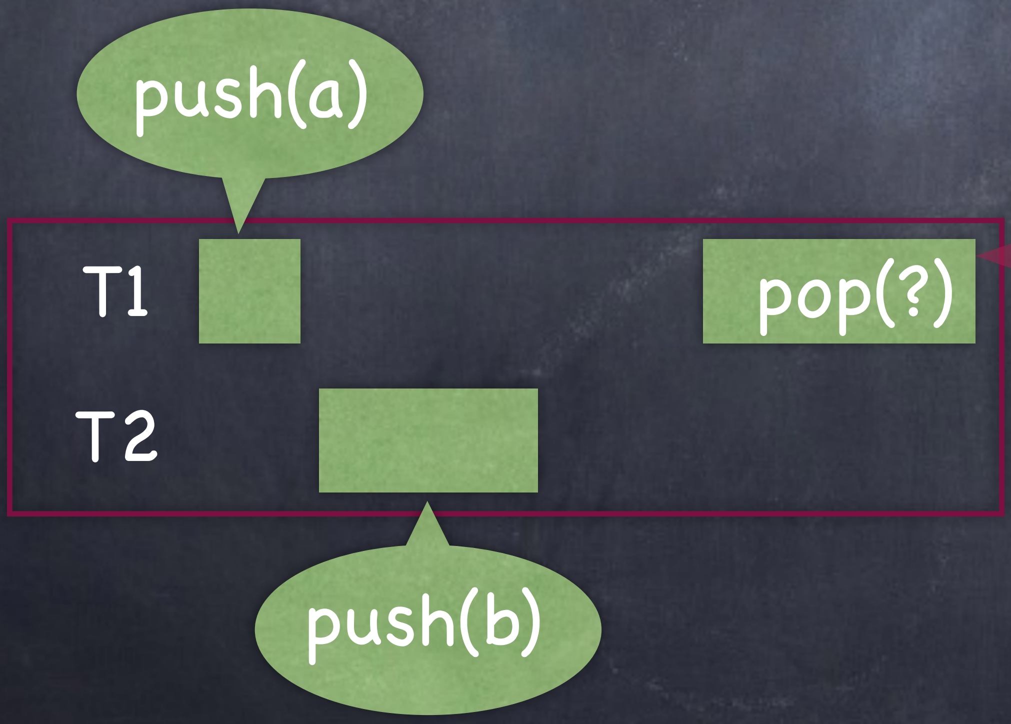


Bad performance also relaxes semantics



may return a or b

The slower the implementation, the more nondeterminism



must return a

Semantics vs. performace comparison (Con^{2}Colic testing)
Haas, Henzinger, Holzer, Kirsch, ... S.
work in progress