Quantitatively Relaxed Concurrent Data Structures

Thomas A. Henzinger	IST Austria
Christoph M. Kirsch	University of Salzburg
Hannes Payer	University of Salzburg
Ali Sezgin	IST Austria
Ana Sokolova	University of Salzburg

University of Leicester 14.12.2012

Semantics of concurrent data structures

- Sequential specification - set of legal sequences
- Correctness condition - linearizability

Semantics of concurrent data structures

Stack - legal sequence

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

- Sequential specification - set of legal sequences
- Correctness condition - linearizability

Semantics of concurrent data structures

Stack - legal sequence

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

- Sequential specification - set of legal sequences
- Correctness condition - linearizability

Stack - concurrent history

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

Semantics of concurrent data structures

Stack - legal sequence

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

- Sequential specification - set of legal sequences
- Correctness condition - 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)

Semantics of concurrent data structures

Stack - legal sequence

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

we relax this

- Sequential specification - set of legal sequences

linearizable wrt seq.spec.

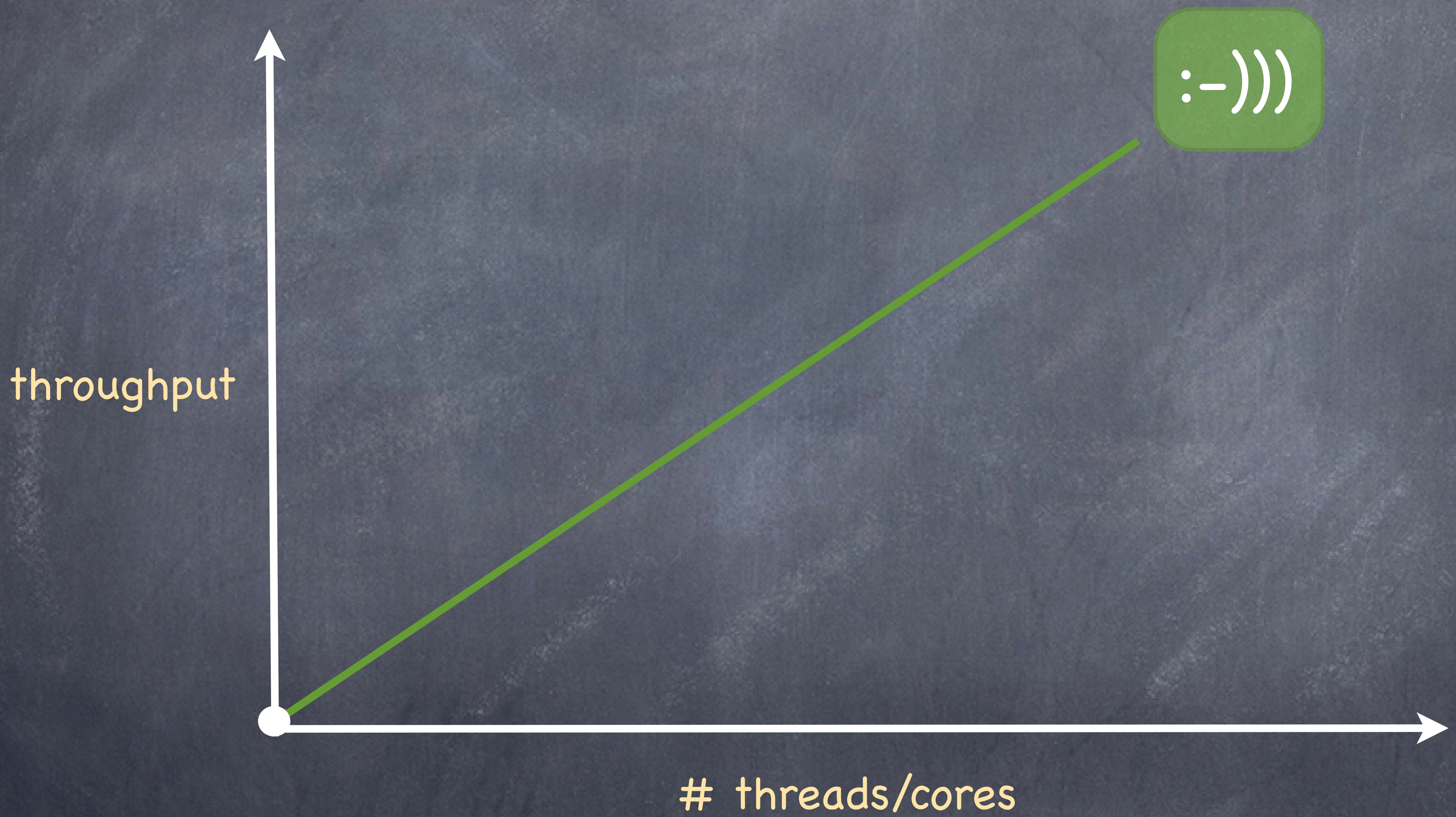
- Correctness condition - linearizability

Stack - concurrent history

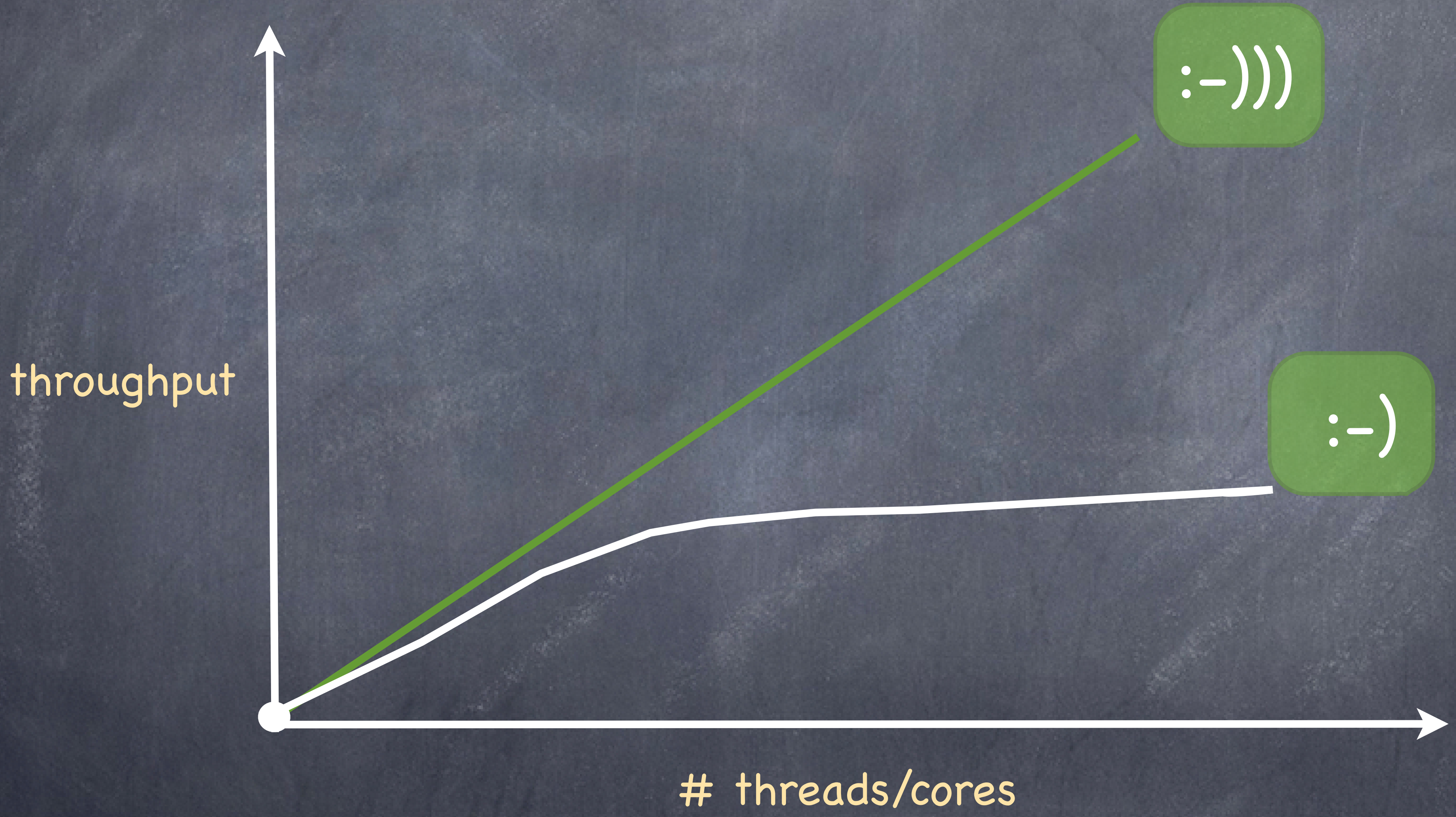
begin-push(a)begin-push(b) end-push(a) end-push(b)begin-pop(b)end-pop(b)

Performance and scalability

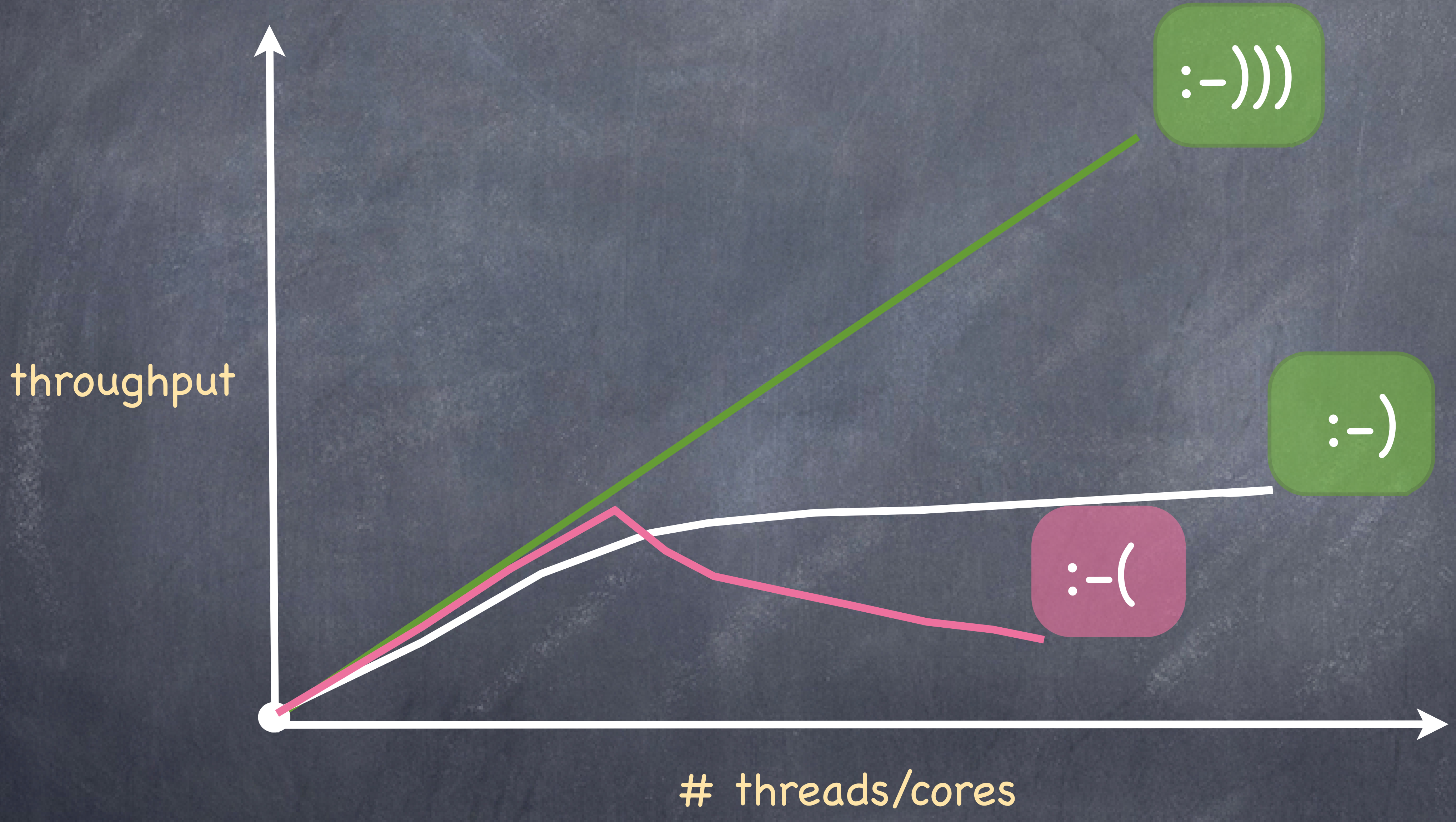
Performance and scalability



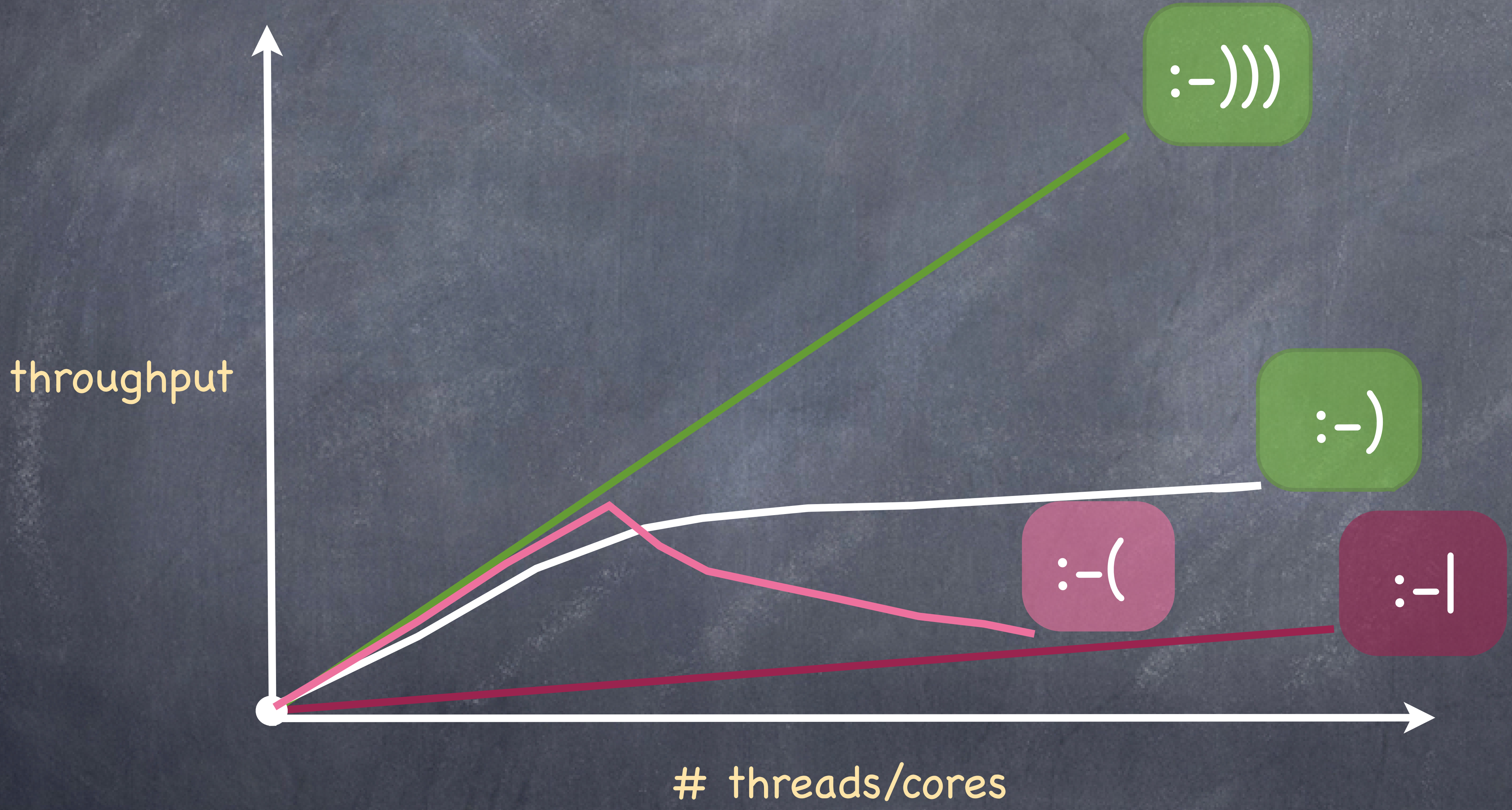
Performance and scalability



Performance and scalability



Performance and scalability



The goal

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

The goal

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

measure the error from correct behavior

The goal

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?

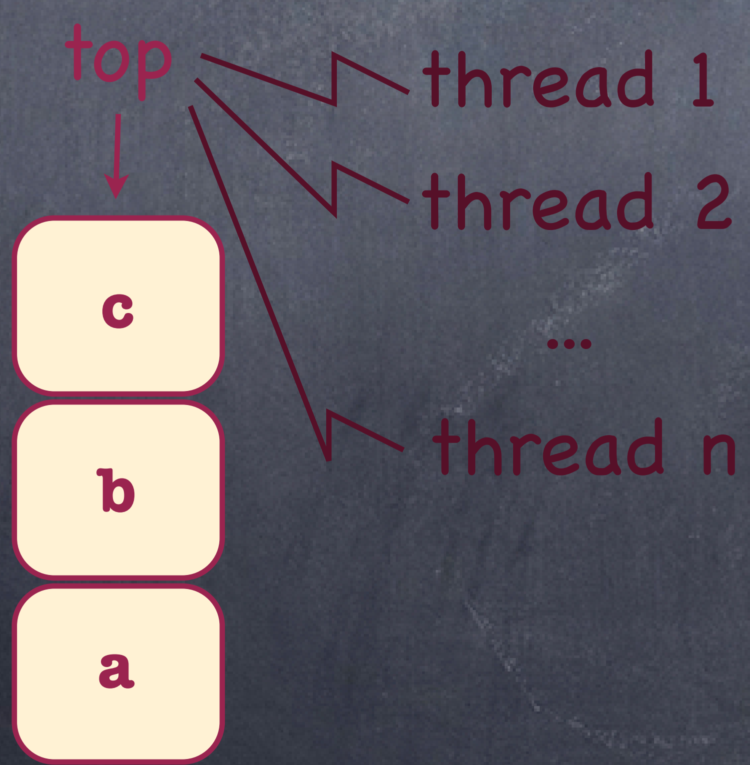
- It is theoretically interesting
- Provides potential for better performing concurrent implementations

...

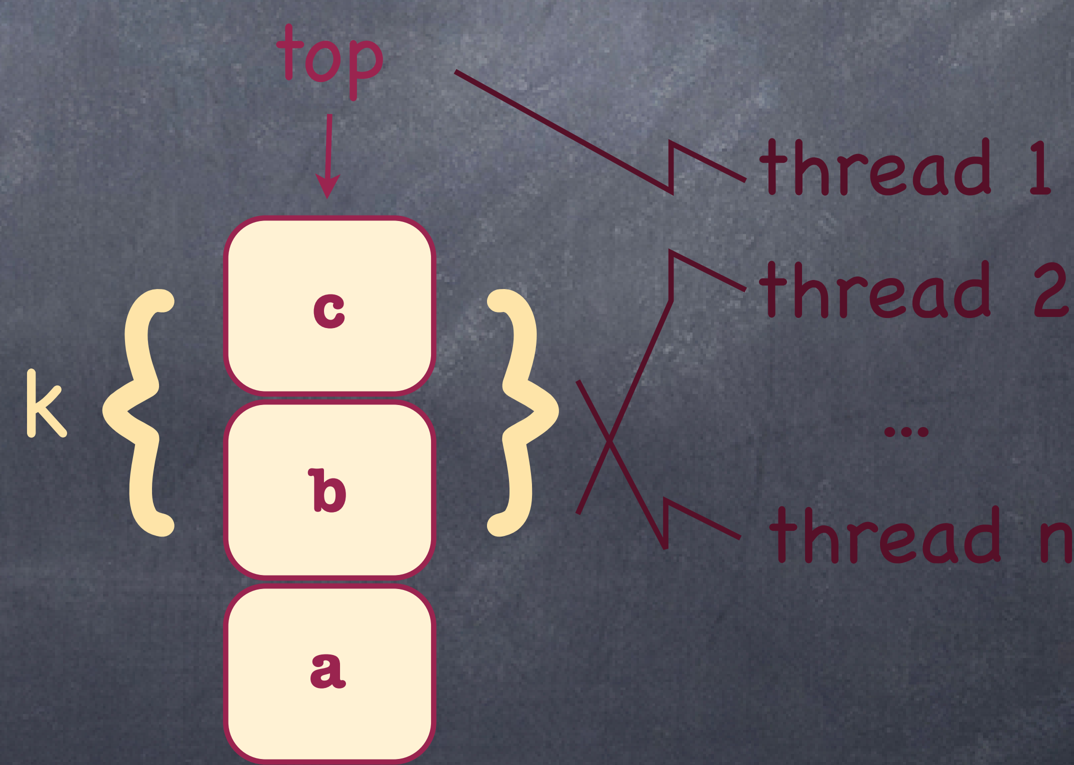
Why relax?

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

Stack



k-Relaxed stack



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

Enough introduction

☺

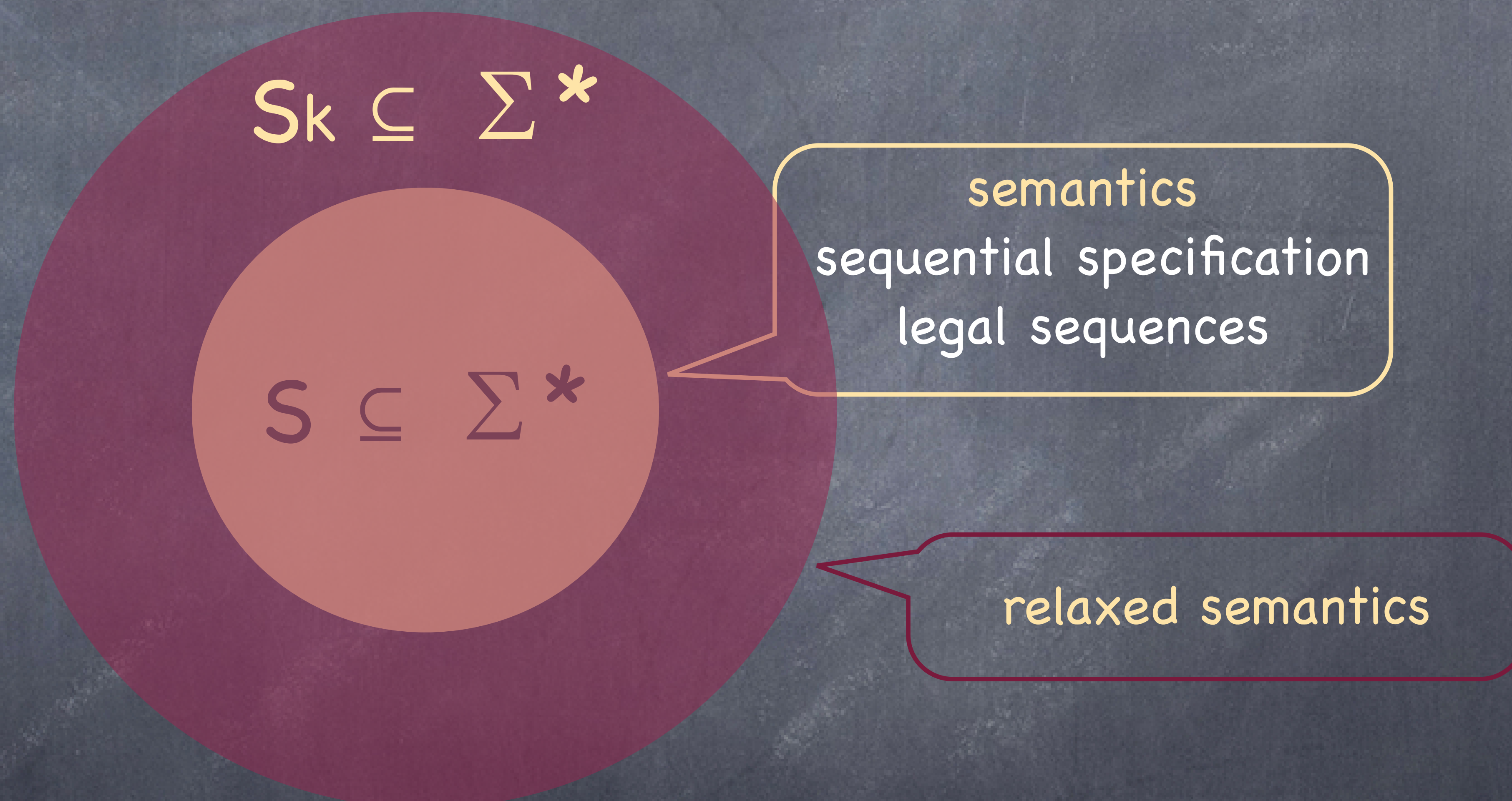
The big picture

$S \subseteq \Sigma^*$

semantics
sequential specification
legal sequences

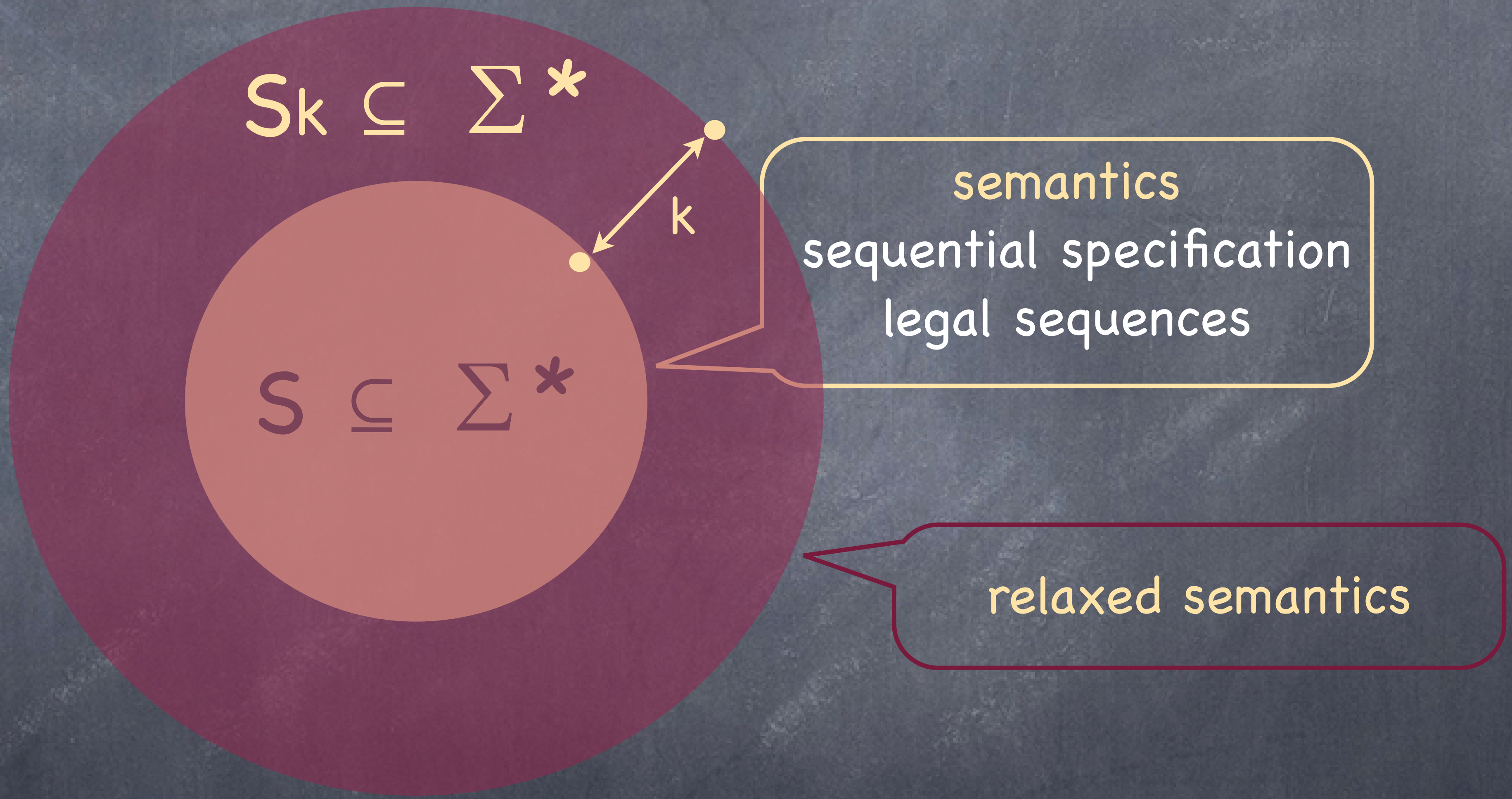
Σ - methods with arguments

The big picture



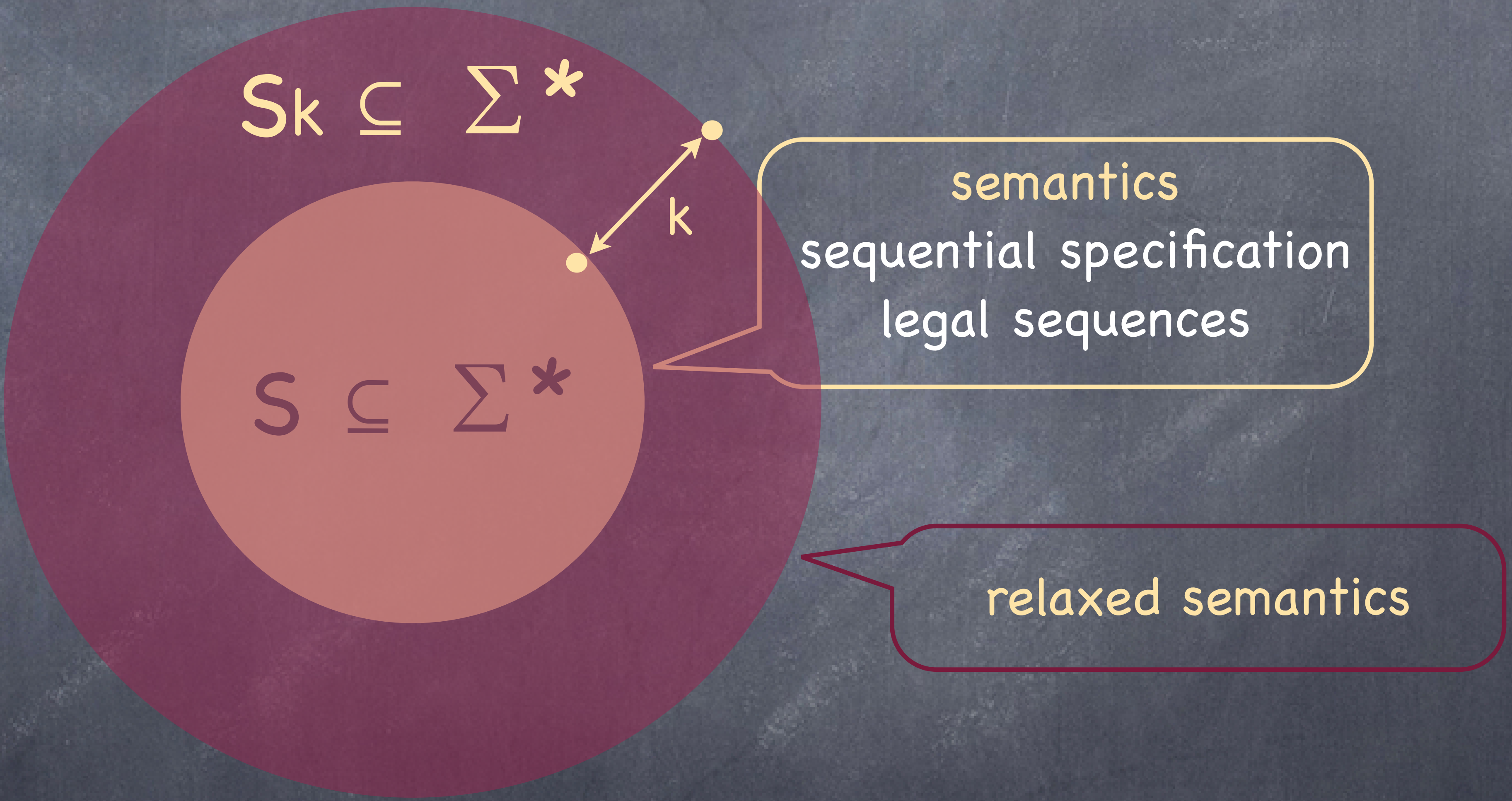
Σ - methods with arguments

The big picture



Σ - methods with arguments

The big picture



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 .....

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, ...

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?

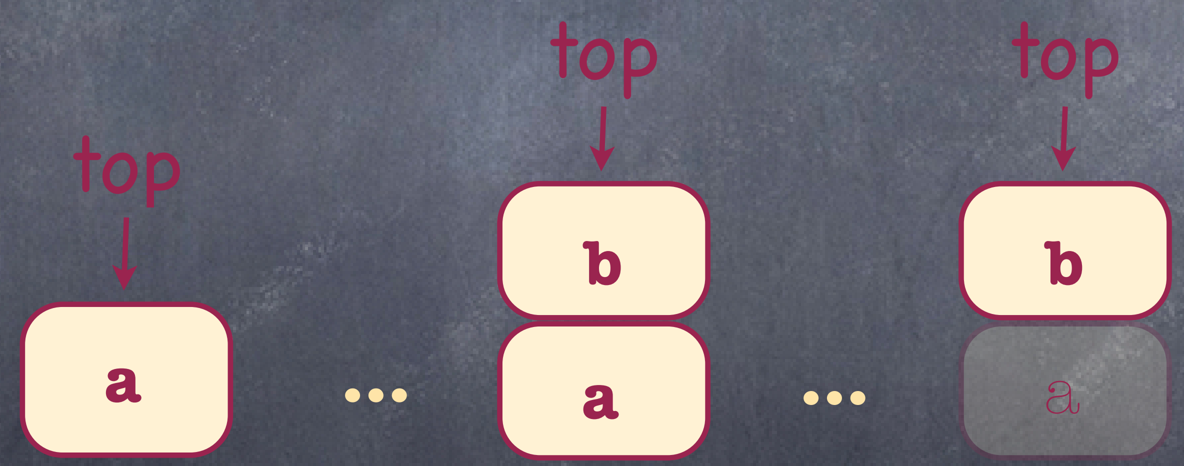
Syntactic distances do not help

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

Syntactic distances do not help

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

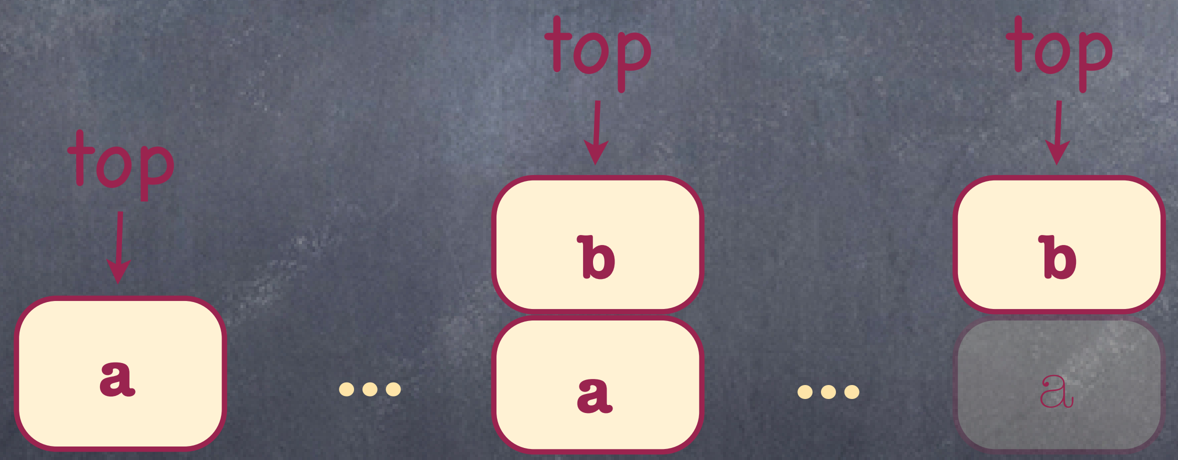
is a 1-out-of-order stack sequence



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)

Stack example

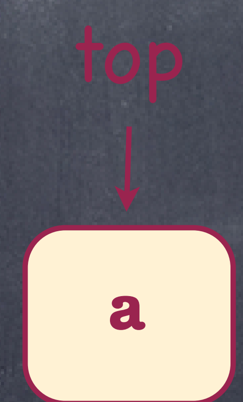
push(a)push(b)push(c)pop(a)pop(b)

state evolution

Stack example

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

state evolution



Stack example

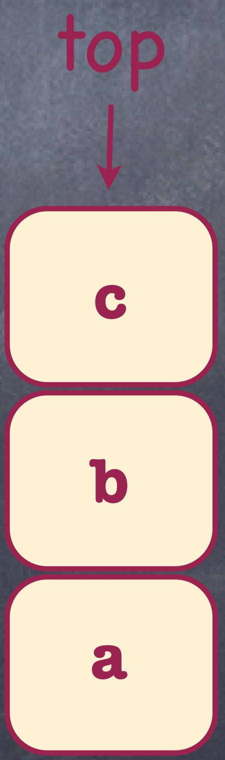
push(a)push(b)push(c)pop(a)pop(b)

state evolution

Stack example

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

state evolution

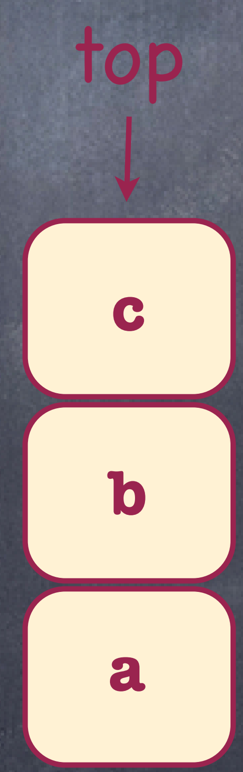


Stack example

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

state evolution

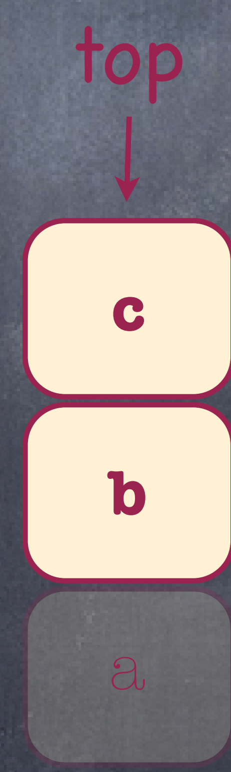
???



Stack example

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

state evolution



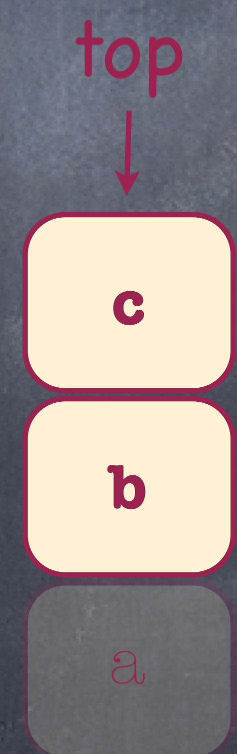
???

How much does this error cost?

Stack example

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

state evolution



Cost 2

Stack example

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

state evolution

top

c

b

a

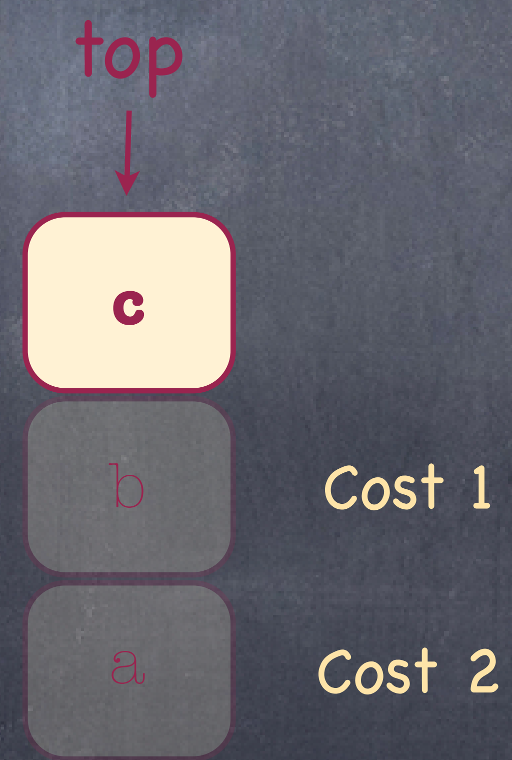
Cost 2

???

Stack example

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

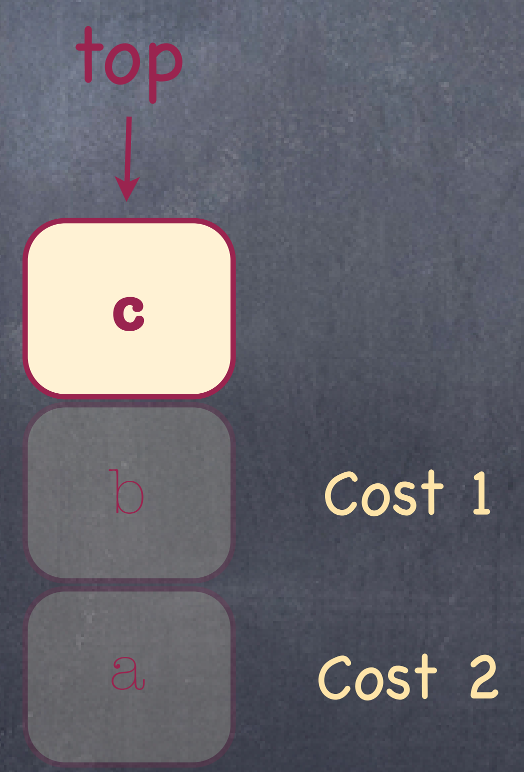
state evolution



Stack example

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

state evolution

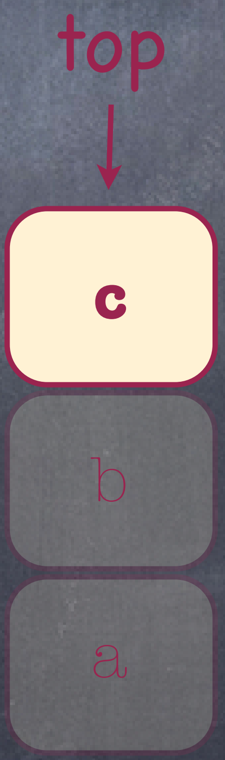


Total cost?

Stack example

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

state evolution



Cost 1

Cost 2

max = 2

sum = 3

Total cost?

Semantic distances need a notion of state

- States are equivalence classes of sequences in S
- Two sequences in S are equivalent if they have an indistinguishable future

Semantic distances need a notion of state

- States are equivalence classes of sequences in S
- Two sequences in S are equivalent if they have an indistinguishable future

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

Semantic distances need a notion of state

- States are equivalence classes of sequences in S

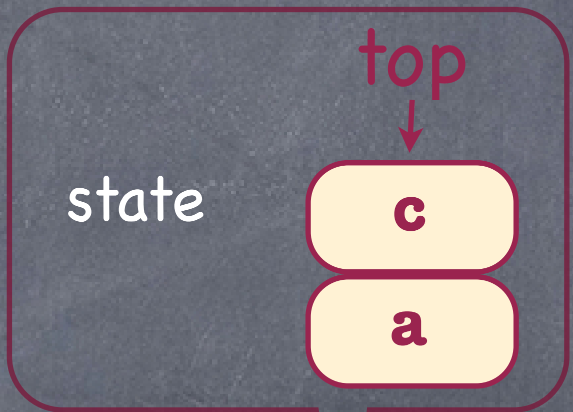
example: for stack

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

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

$$\mathbf{x} \equiv \mathbf{y} \quad \Leftrightarrow \quad \forall \mathbf{u} \in \Sigma^*. \ (\mathbf{xu} \in \mathbf{S} \Leftrightarrow \mathbf{yu} \in \mathbf{S})$$

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

$$\mathbf{x} \equiv \mathbf{y} \quad \Leftrightarrow \quad \forall \mathbf{u} \in \Sigma^*.\ (\mathbf{xu} \in \mathbf{S} \Leftrightarrow \mathbf{yu} \in \mathbf{S})$$

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

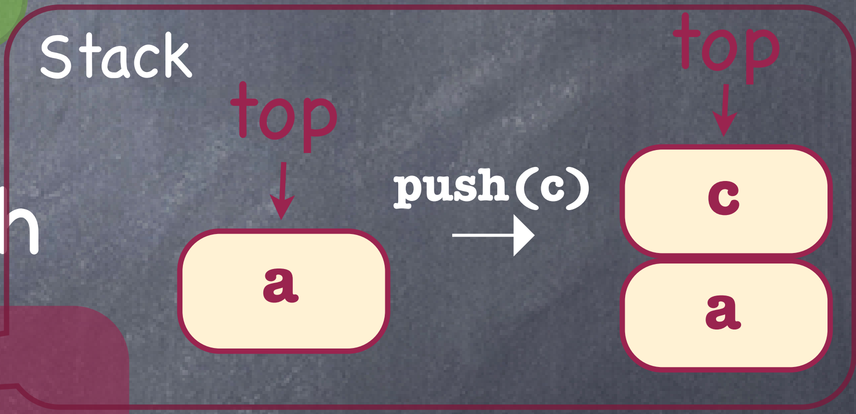
$$[s]_\equiv \xrightarrow{m} [sm]_\equiv \quad \Leftrightarrow \quad sm \in S$$

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

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

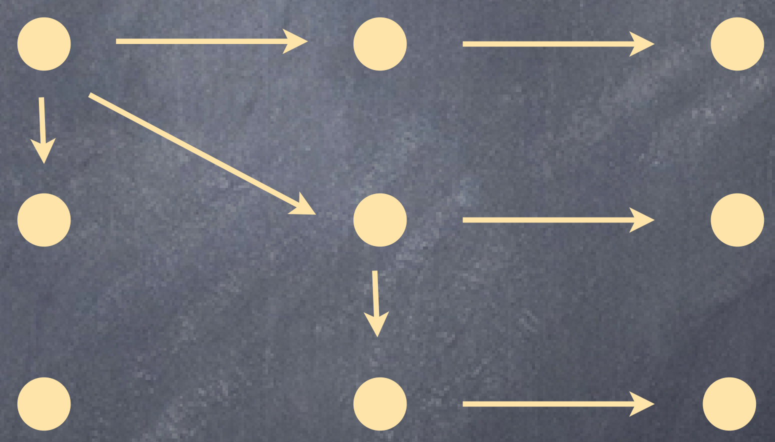
The framework

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

The framework

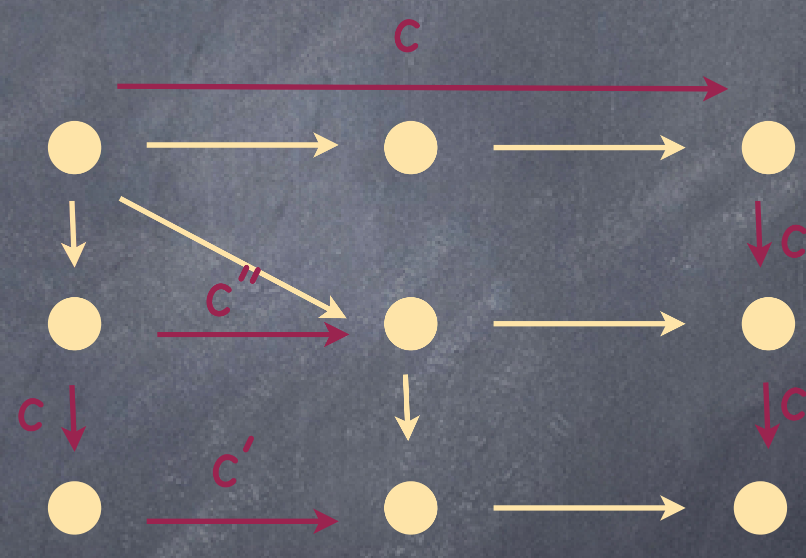
- Start from LTS(S)
- Add transitions with transition costs
- Fix a path cost function

Σ - singleton



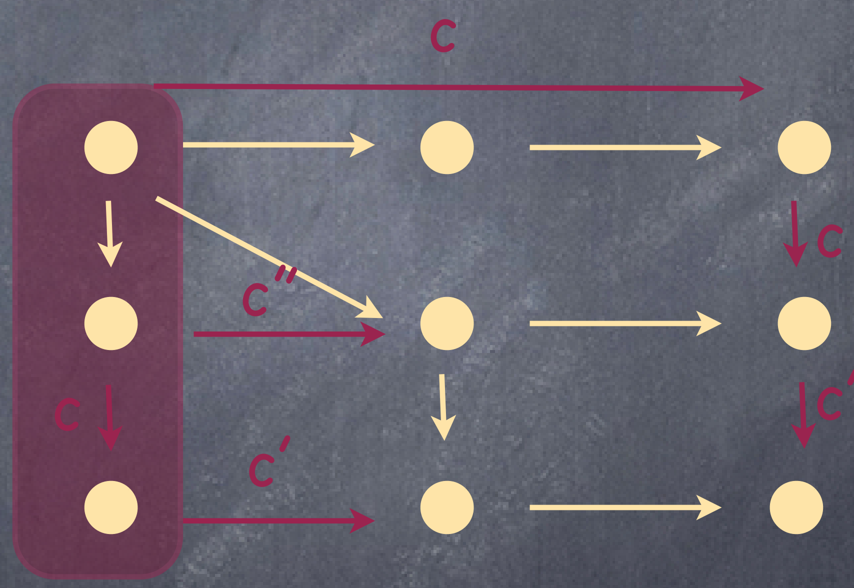
The framework

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



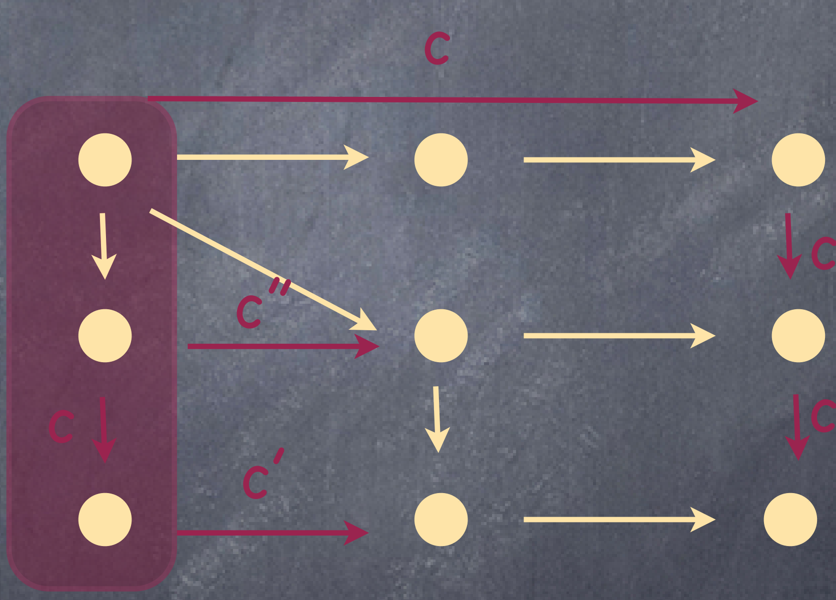
The framework

- 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

For the user

- Pick your favorite data structure S
- Add desired incorrect transitions and assign them transition costs
- Choose a path cost function

distance and relaxation follow

For the user

The framework clears the head,
direct concrete relaxations are also possible

- Pick your favorite data structure S
- Add desired incorrect transitions and assign them transition costs
- Choose a path cost function

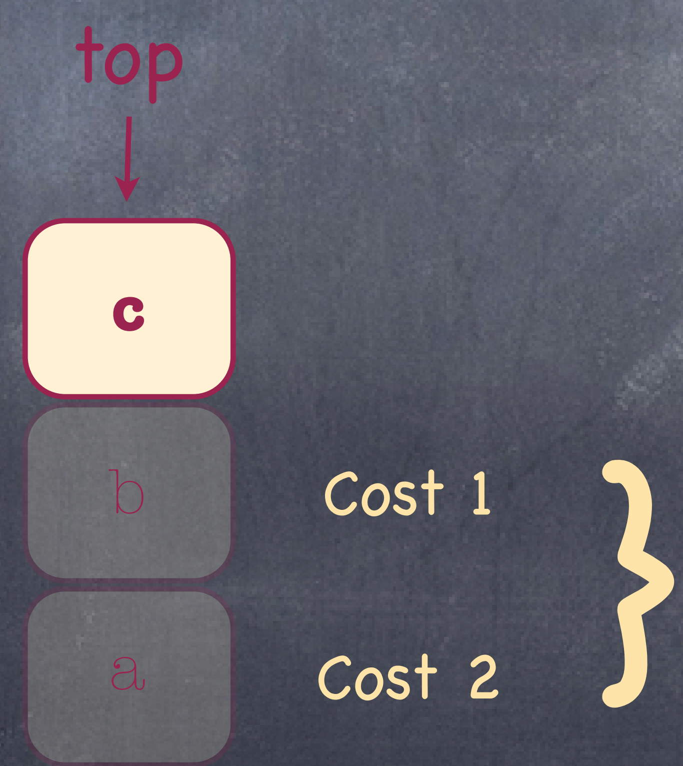
distance and relaxation follow

Stack example

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

state evolution

Total cost



max = 2
sum = 3

Stack example

- Canonical representative of a state
- Add incorrect transitions with costs
- Possible path cost functions max, sum,...

Stack example

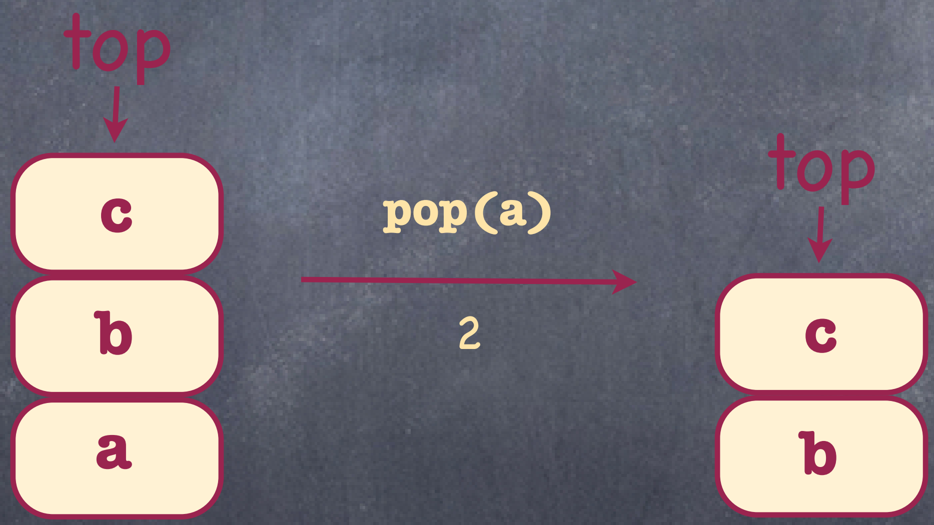
Sequence of **push**'s with no matching **pop**

- Canonical representative of a state
- Add incorrect transitions with costs
- Possible path cost functions max, sum,...

Stack example

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

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



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

It's more general...

Generic out-of-order

segment_cost($q \xrightarrow{m} q'$) $= |\mathbf{v}|$

transition cost

where $\mathbf{v}$ is a sequence of minimal length s.t.

(1) $[\mathbf{uvw}]_{\equiv} = q$, $\mathbf{uvw}$ is minimal, $\mathbf{uw}$ is minimal

removing $\mathbf{v}$ enables a transition

(1.2) $[\mathbf{uw}]_{\equiv} \xrightarrow{m} [\mathbf{uw'}]_{\equiv}$, $[\mathbf{uvw'}]_{\equiv} = q'$

(2) $[\mathbf{uw}]_{\equiv} = q$, $\mathbf{uw}$ is minimal, $\mathbf{uvw}$ is minimal

inserting $\mathbf{v}$ enables a transition

goes with different path costs

Generic out-of-order

$\text{segment\_cost}(q \xrightarrow{m} q') = |\mathbf{v}|$ transition cost

where $\mathbf{v}$ is a sequence of minimal length s.t.

(1) $[\mathbf{uvw}]_\equiv = q$, $\mathbf{uvw}$ is minimal, $\mathbf{uw}$ is minimal

(1.1) $[\mathbf{uw}]_\equiv \xrightarrow{m} [\mathbf{u'w}]_\equiv$, $[\mathbf{u'vw}]_\equiv = q'$

(1.2) $[\mathbf{uw}]_\equiv \xrightarrow{m} [\mathbf{uw'}]_\equiv$, $[\mathbf{uvw'}]_\equiv = q'$

(2) $[\mathbf{uw}]_\equiv = q$, $\mathbf{uw}$ is minimal, $\mathbf{uvw}$ is minimal

(1.1) $[\mathbf{uvw}]_\equiv \xrightarrow{m} [\mathbf{u'vw}]_\equiv$, $[\mathbf{u'w}]_\equiv = q'$

(1.2) $[\mathbf{uvw}]_\equiv \xrightarrow{m} [\mathbf{uvw'}]_\equiv$, $[\mathbf{uw'}]_\equiv = q'$

Generic out-of-order

$\text{segment\_cost}(q \xrightarrow{m} q') = |\mathbf{v}|$ transition cost

where $\mathbf{v}$ is a sequence of minimal length s.t.

(1) $[\mathbf{uvw}]_\equiv = q$, $\mathbf{uvw}$ is minimal, $\mathbf{uw}$ is minimal

removing $\mathbf{v}$ enables a transition

(1.2) $[\mathbf{uw}]_\equiv \xrightarrow{m} [\mathbf{uw'}]_\equiv$, $[\mathbf{uvw'}]_\equiv = q'$

(2) $[\mathbf{uw}]_\equiv = q$, $\mathbf{uw}$ is minimal, $\mathbf{uvw}$ is minimal

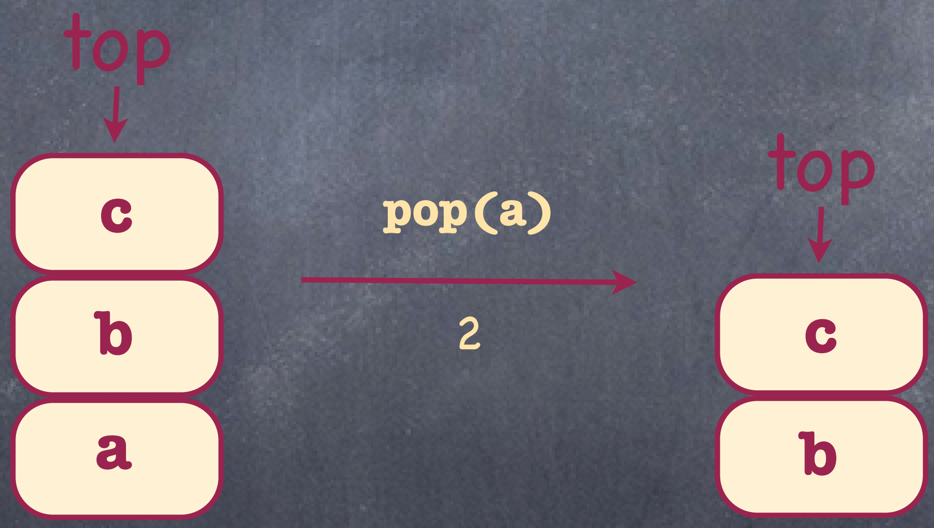
inserting $\mathbf{v}$ enables a transition

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 ``shrinking window''
restricted out-of-order

Out-of-order queue

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

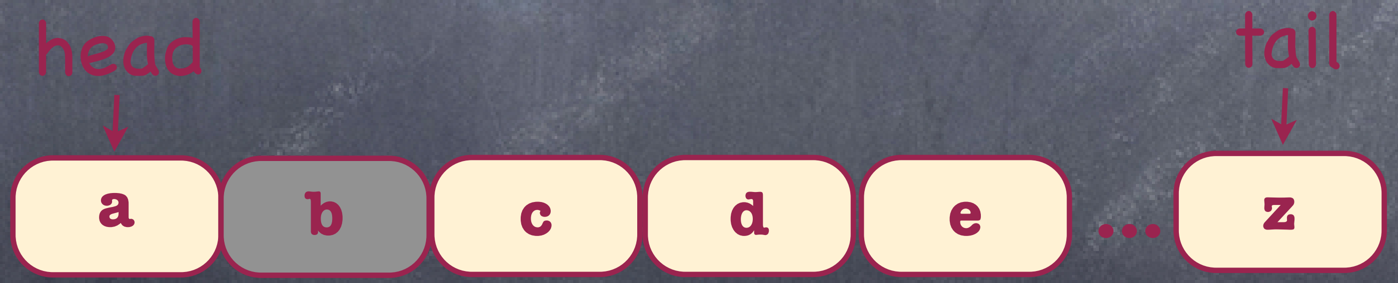
- Canonical representative of a state
- Add incorrect transitions with segment-costs

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

also "shrinking window" restricted out-of-order

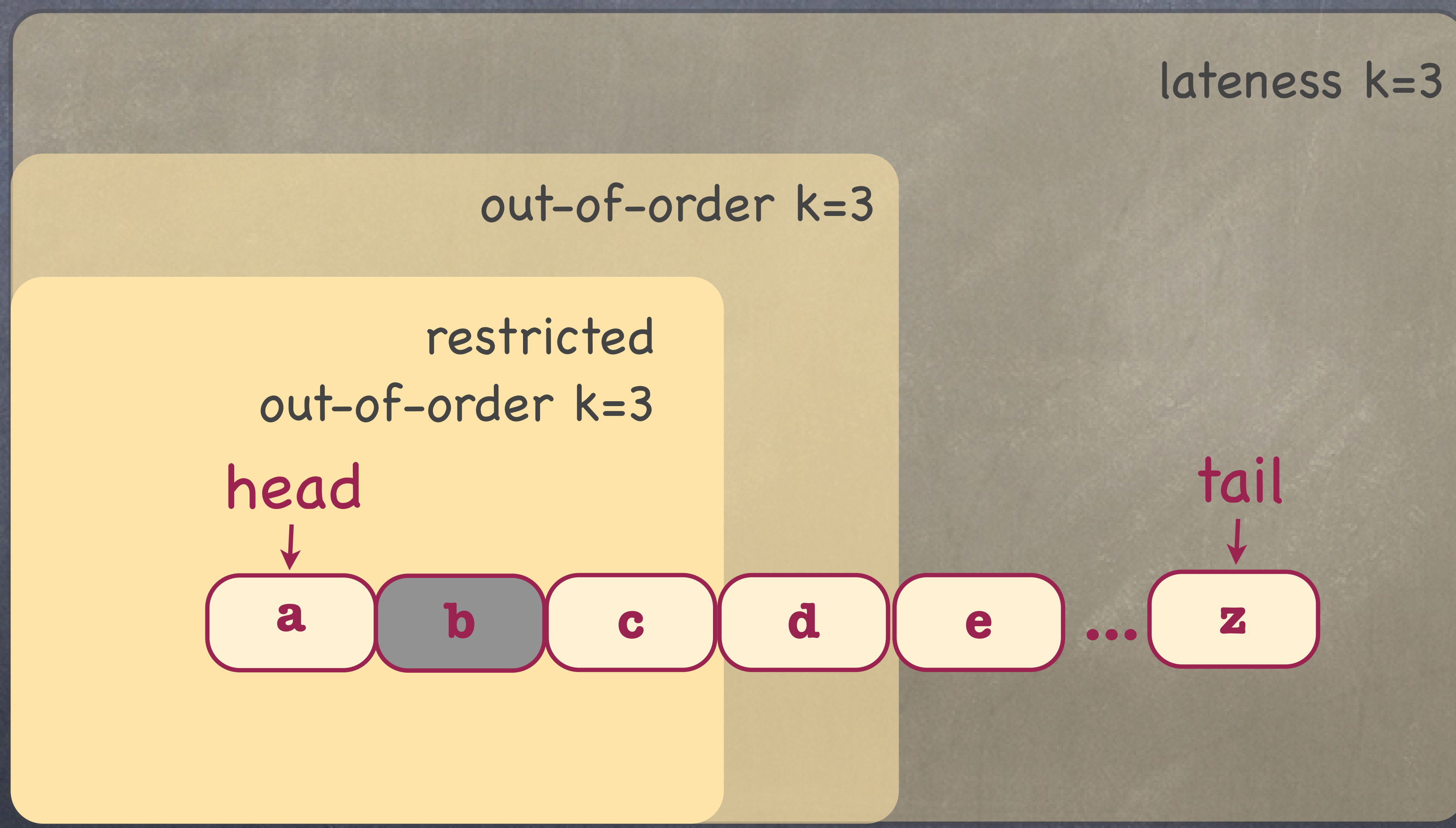
Out-of-order variants

Queue



Out-of-order variants

Queue



How about implementations? Performance?

Short-term history

- SCAL queues [KPRS'11]
- Quasi linearizability theory and implementations [AKY'10]
- Some straightforward implementations [HKPSS'12]
- Efficient lock-free segment queue [KLP'12]

(almost) all implement restricted out-of-order

Short-term history

distributed, one k-queue

- SCAL queues [KPRS'11]
- Quasi linearizability theory and implementations [AKY'10]
- Some straightforward implementations [HKPSS'12]
- Efficient lock-free segment queue [KLP'12]

(almost) all implement restricted out-of-order

Short-term history

distributed, one k-queue

syntactic, does not work for stacks

- SCAL queues [KPRS'11]
- Quasi linearizability theory and implementations [AKY'10]
- Some straightforward implementations [HKPSS'12]
- Efficient lock-free segment queue [KLP'12]

(almost) all implement restricted out-of-order

Short-term history

distributed, one k-queue

- SCAL queues [KPRS'11]

syntactic, does not work for stacks

- Quasi linearizability theory and implementations ~~[AKY'10]~~

not too well performing

- Some straightforward implementations [HKPSS'12]
- Efficient lock-free segment queue [KLP'12]

(almost) all implement restricted out-of-order

Short-term history

distributed, one k-queue

syntactic, does not work for stacks

not too well performing

not too well performing

- SCAL queues [KPRS'11]
- Quasi linearizability theory and implementations [AKY'10]
- Some straightforward implementations [HKPSS'12]
- Efficient lock-free segment queue [KLP'12]

(almost) all implement restricted out-of-order

Short-term history

- SCAL queues [KPRS'11]
- Quasi linearizability theory and implementations [AKY'10]
- Some straightforward implementations [HKPSS'12]
- Efficient lock-free segment queue [KLP'12]

distributed, one k-queue

syntactic, does not work for stacks

not too well performing

not too well performing

performs very well

(almost) all implement restricted out-of-order

Lessons learned

Lessons learned

The way from sequential specification to concurrent implementation is hard

Lessons learned

The way from sequential specification to concurrent implementation is hard

Being relaxed not necessarily means better performance

Lessons learned

The way from sequential specification to concurrent implementation is hard

Being relaxed not necessarily means better performance

Well-performing implementations of relaxed specifications do exist!

Lessons learned

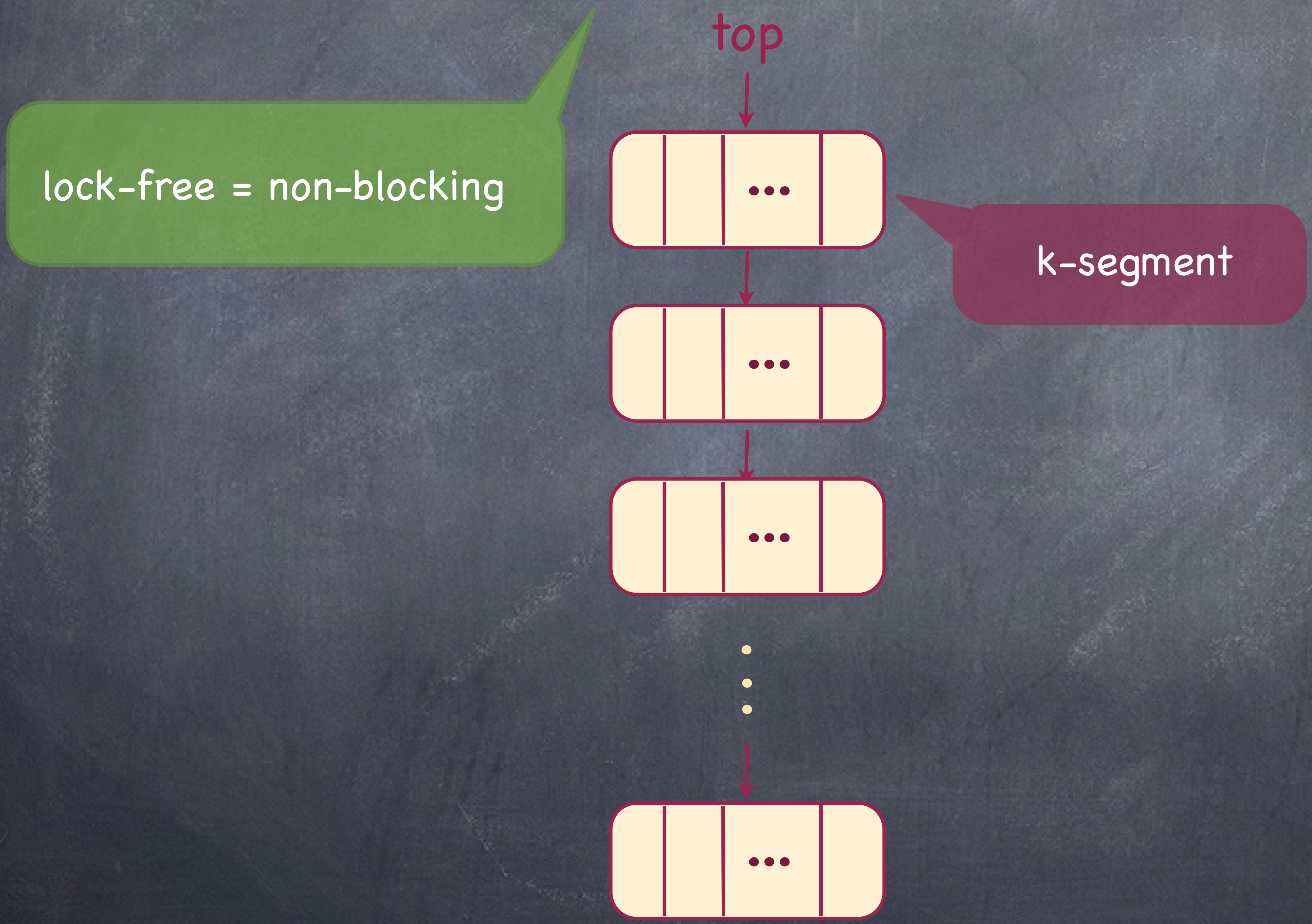
The way from sequential specification to concurrent implementation is hard

Being relaxed not necessarily means better performance

Well-performing implementations of relaxed specifications do exist!

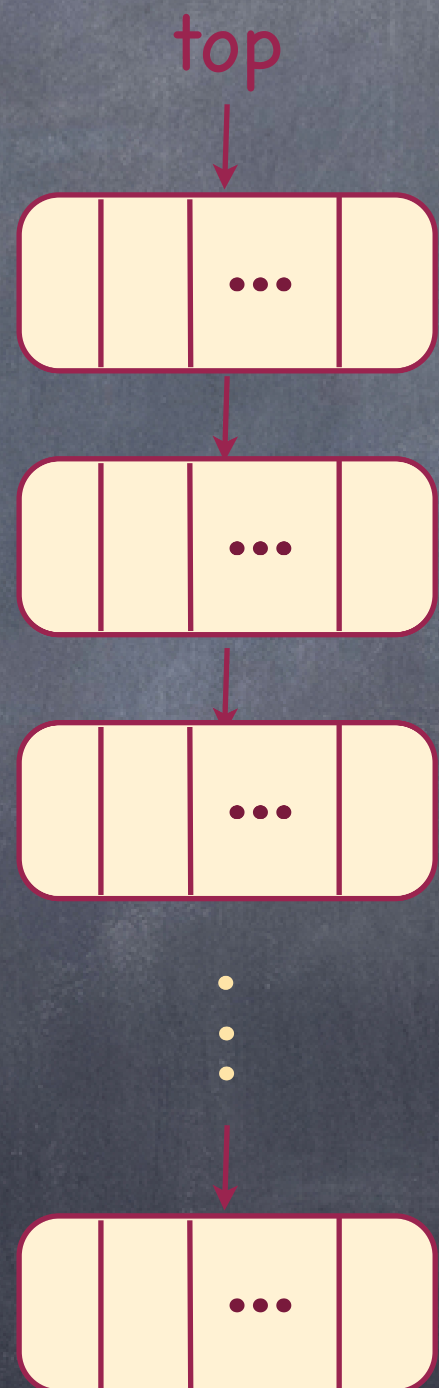
Let's see them!

Restricted-out-of-order k-Stack



Restricted-out-of-order k-Stack

lock-free = non-blocking

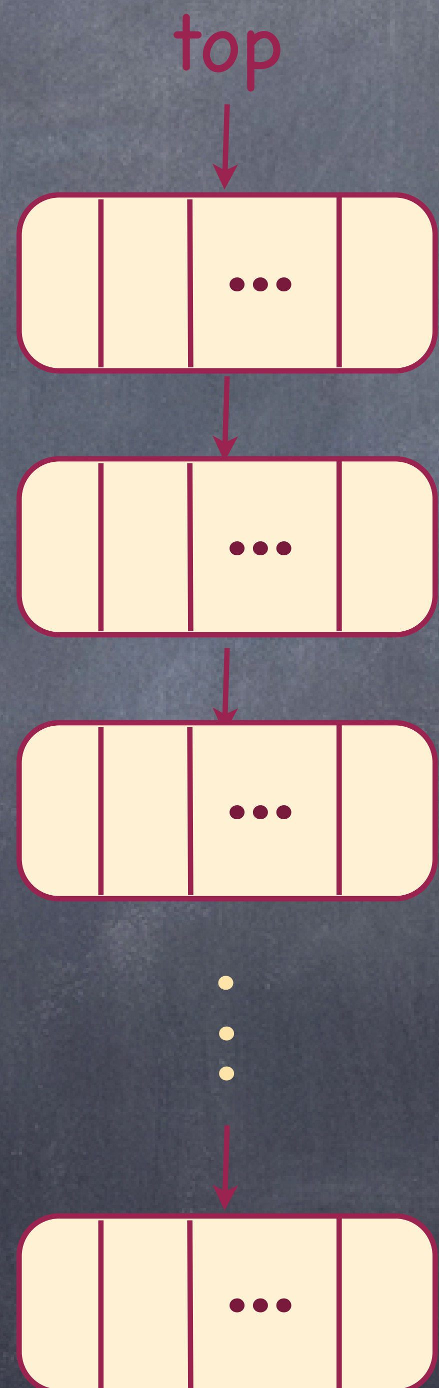


k-segment

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

Restricted-out-of-order k-Stack

lock-free = non-blocking

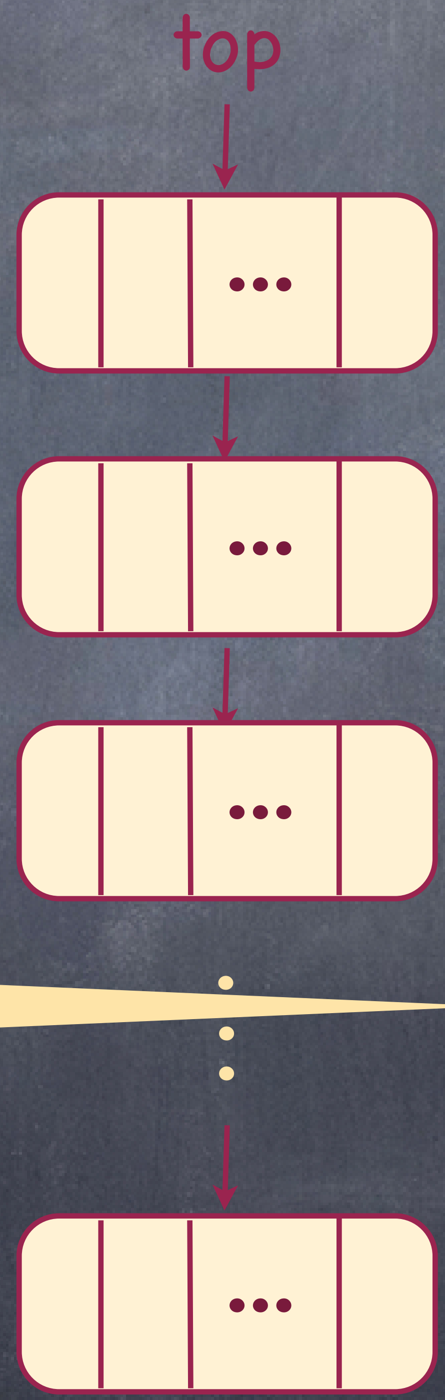


k-segment

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)
```

Restricted-out-of-order k-Stack



lock-free = non-blocking

k-segment

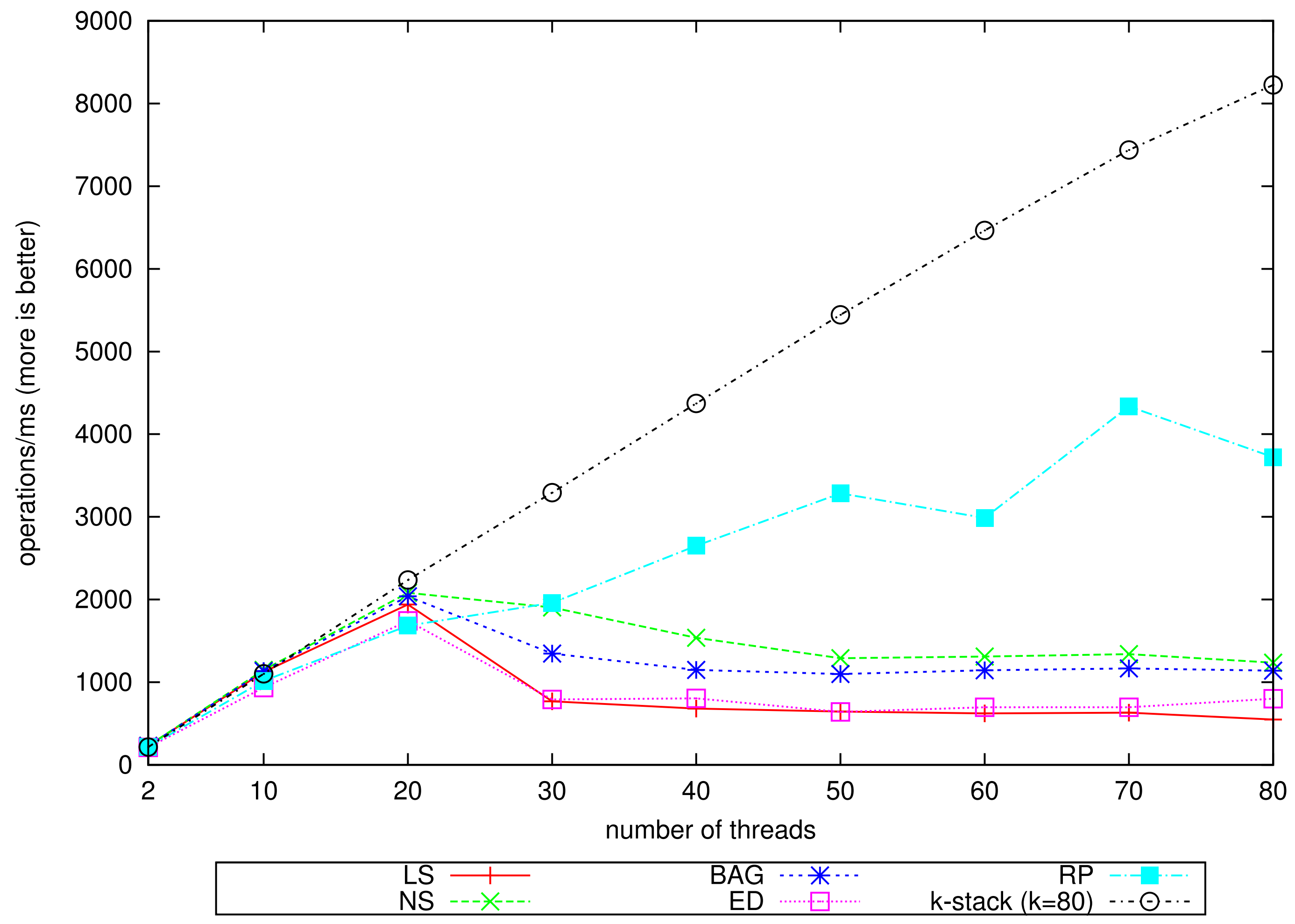
add/remove segment

CAS - based

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

Stack

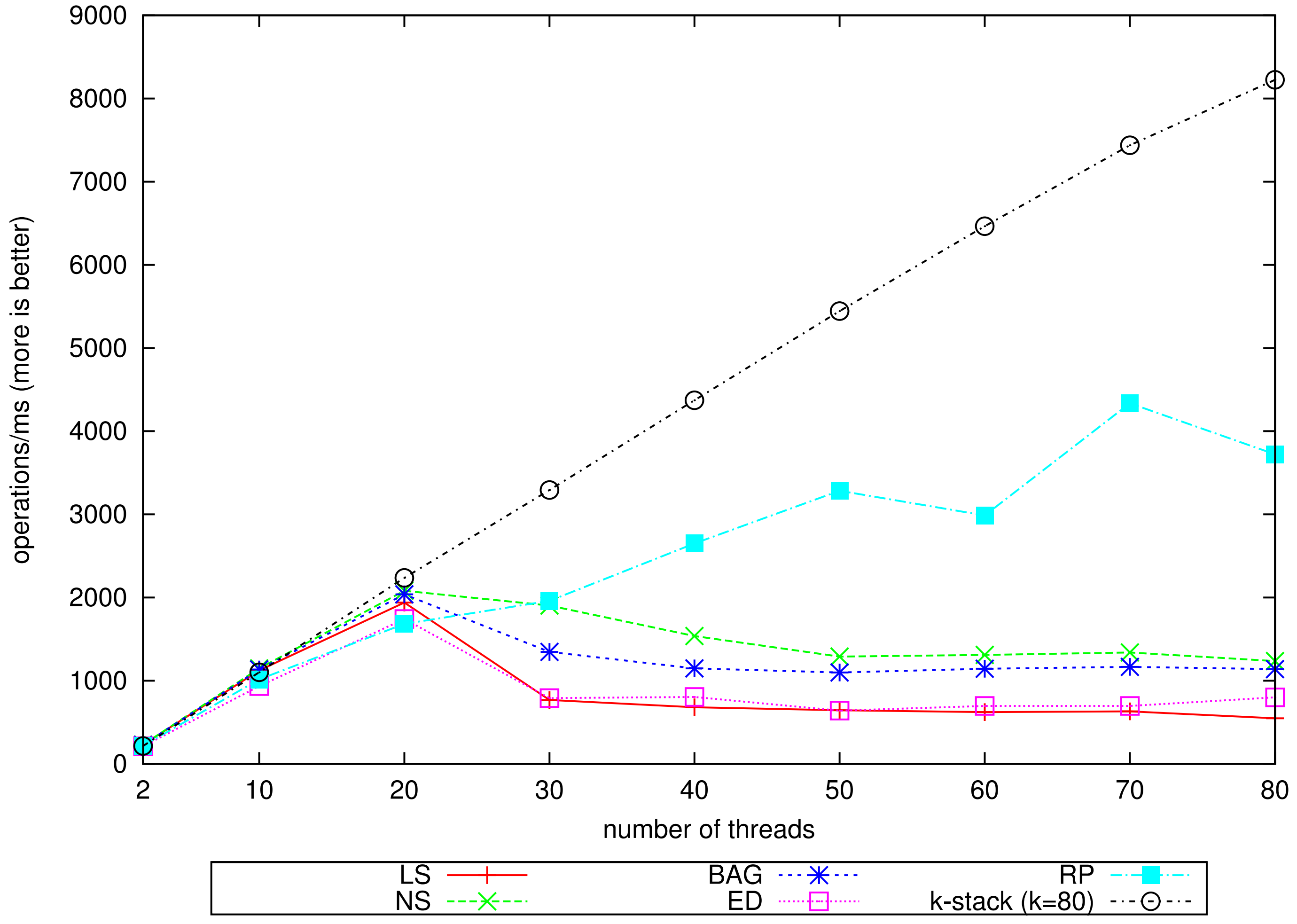
Scalability comparison



Stack

Scalability comparison

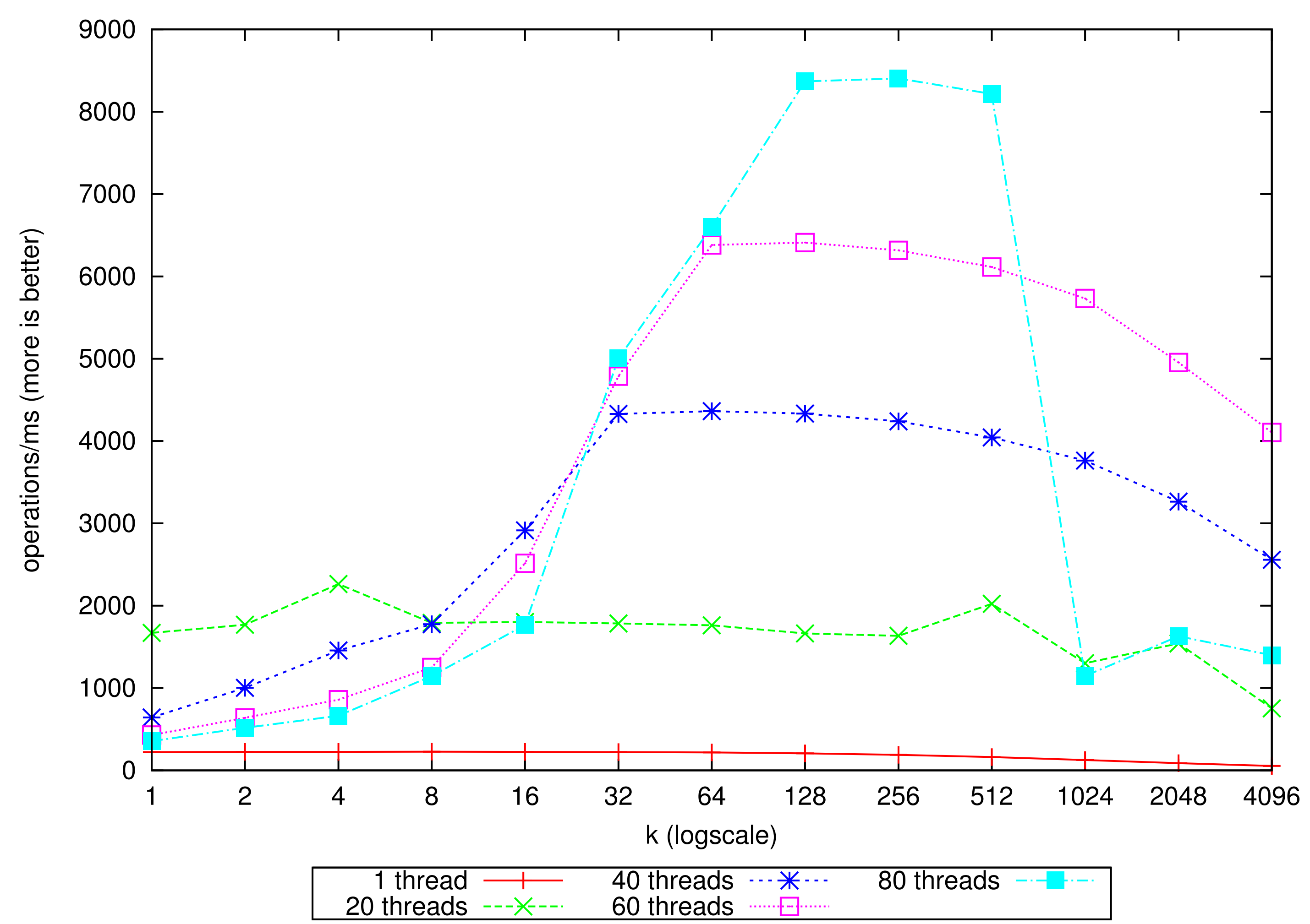
"80"-core machine



k-Stack

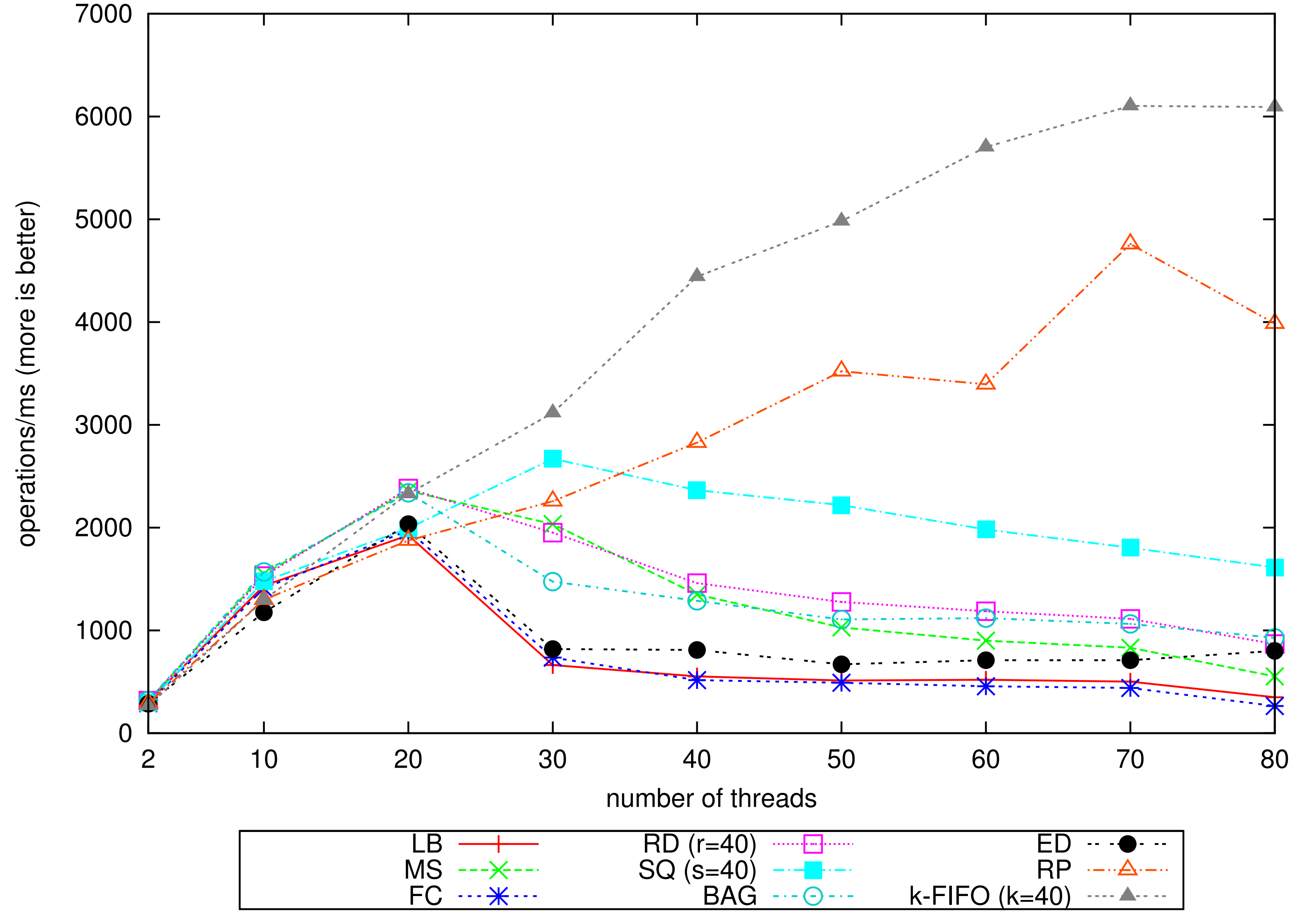
The more relaxed, the better

lock-free segment stack



Queue

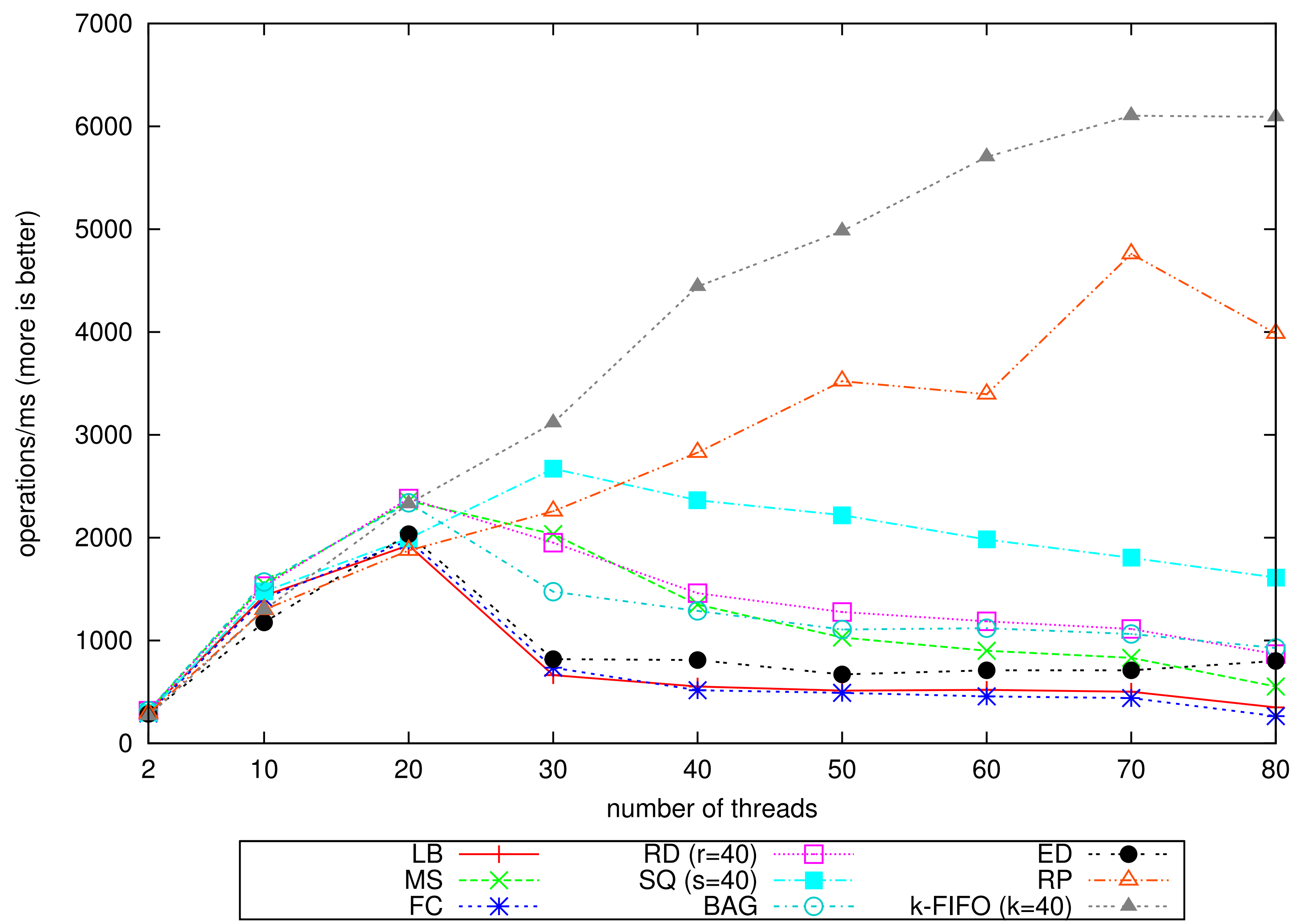
Scalability comparison



Queue

Scalability comparison

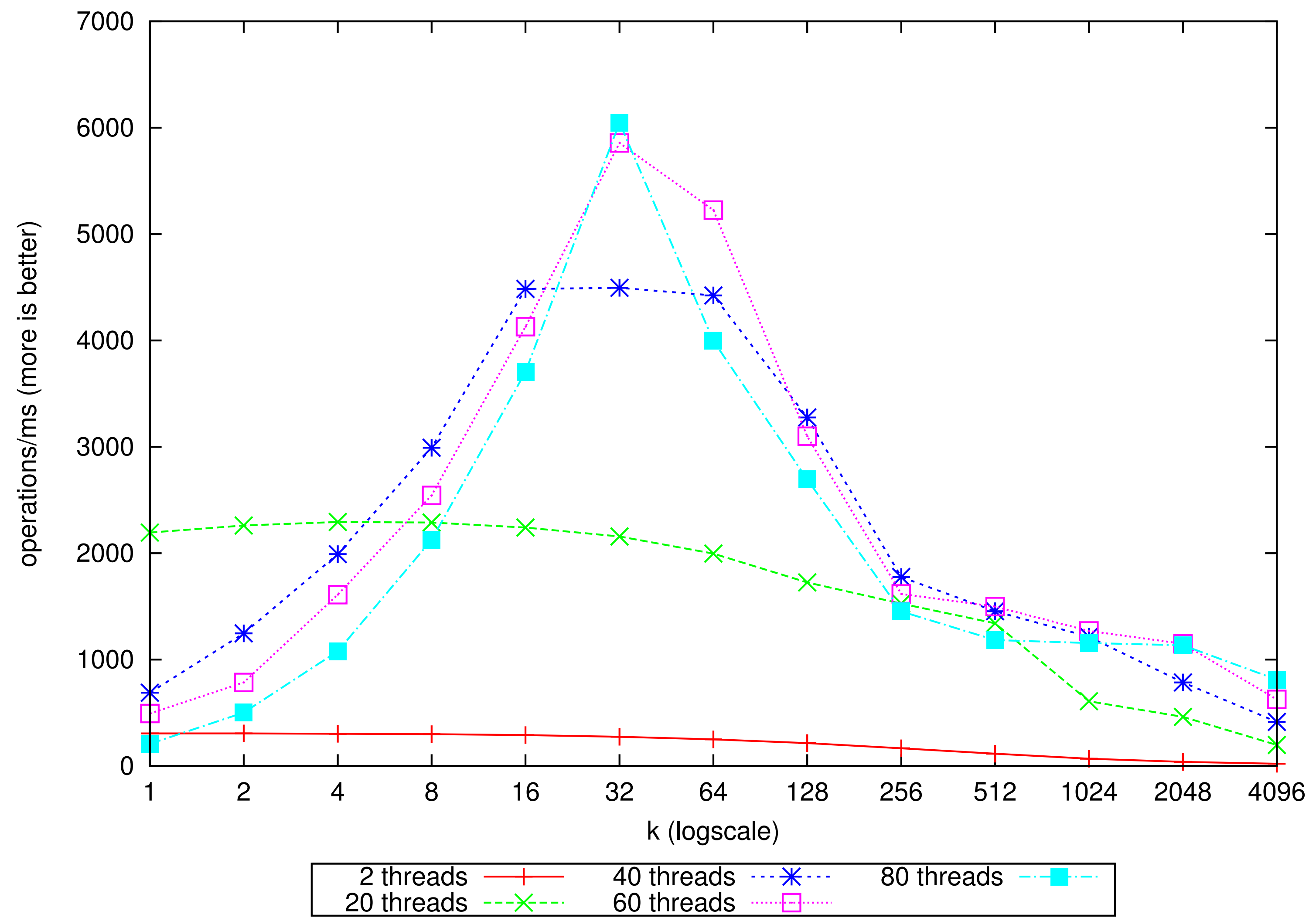
"80"-core machine



k-Queue

The more relaxed, the better

lock-free segment queue



Conclusions

Contributions

Framework for quantitative relaxations
generic relaxations, concrete examples,
efficient implementations exist

Conclusions

all kinds of

Contributions

Framework for quantitative relaxations
generic relaxations, concrete examples,
efficient implementations exist

Conclusions

all kinds of

Contributions

Framework for quantitative relaxations
generic relaxations, concrete examples,
efficient implementations exist

Difficult open problem

How to get from theory to practice?

Conclusions

all kinds of

Contributions

Framework for quantitative relaxations
generic relaxations, concrete examples,
efficient implementations exis

Difficult open problem

THANK YOU

How to get from theory to practice?

For the future

- Study applicability
- Learn from efficient implementations

For the future

- Study applicability
- Learn from efficient implementations

which applications tolerate relaxation ?

maybe there is nothing to tolerate!

For the future

- Study applicability
  - which applications tolerate relaxation ?
  - maybe there is nothing to tolerate!
- Learn from efficient implementations
  - towards synthesis
  - lock-free universal construction ?

For the future

- Study applicability
  - which applications tolerate relaxation ?
  - maybe there is nothing to tolerate!
- Learn from efficient implementations
  - towards synthesis
  - lock-free universal construction ?

THANK YOU