Quantitatively Relaxed Concurrent Data Structures

Thomas A. Henzinger
Christoph M. Kirsch
Hannes Payer
Ali Sezgin
Ana Sokolova

IST Austria
University of Salzburg
University of Salzburg
IST Austria
University of Salzburg

University of Tokyo 30.10.2012
Semantics of concurrent data structures

- Sequential specification - set of legal sequences
- Correctness condition - linearizability
Semantics of concurrent data structures

Stack - legal sequence
\[ \text{push(a)push(b)pop(b)} \]

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)

Ana Sokolova University of Salzburg

University of Tokyo 30.10.2012
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

Sequential specification - set of legal sequences

Correctness condition - linearizability

Stack - legal sequence

\[ \text{push}(a) \text{push}(b) \text{pop}(b) \]

Stack - concurrent history

\[ \text{begin-push}(a) \text{begin-push}(b) \text{end-push}(a) \text{end-push}(b) \text{begin-pop}(b) \text{end-pop}(b) \]
Performance and scalability

# threads/cores

throughput

Ana Sokolova  University of Salzburg

University of Tokyo  30.10.2012
Performance and scalability

throughput

# threads/cores

Ana Sokolova  University of Salzburg

University of Tokyo  30.10.2012
Performance and scalability

Throughput vs. # threads/cores

Ana Sokolova University of Salzburg

University of Tokyo 30.10.2012
Performance and scalability

Throughput vs. # threads/cores
Performance and scalability

throughput

# threads/cores

Ana Sokolova  University of Salzburg

University of Tokyo  30.10.2012
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

Trading correctness for performance

In a controlled way with quantitative bounds

Stack - incorrect behavior

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

Correct in a relaxed stack

... 2-relaxed? 3-relaxed?

Measure the error from correct behavior
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

\[ \text{push}(a) \text{push}(b) \text{push}(c) \text{pop}(a) \text{pop}(b) \]

state evolution

?

Ana Sokolova  University of Salzburg

University of Tokyo  30.10.2012
Stack example

\[\text{push}(a) \text{push}(b) \text{push}(c) \text{pop}(a) \text{pop}(b)\]

state evolution

How much does this error cost?
Stack example

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

state evolution

c
b

Cost 2
Stack example

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

state evolution

???

c
b
Cost 2
Stack example

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

state evolution

top

c  Cost 1
b  Cost 2
a
Stack example

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

state evolution

c

Total cost?

Cost 1

Cost 2

Ana Sokolova University of Salzburg

University of Tokyo 30.10.2012
Stack example

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

state evolution

Total cost?

c

Cost 1

Cost 2

max = 2

sum = 3
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
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
Enough introduction
The big picture

$S \subseteq \Sigma^*$

$\Sigma$ - methods with arguments

semantics
sequential specification
legal sequences

Ana Sokolova University of Salzburg
University of Tokyo 30.10.2012
The big picture

\( S_k \subseteq \Sigma^* \)

\( S \subseteq \Sigma^* \)

semantics

sequential specification

legal sequences

relaxed semantics

\( \Sigma \) - methods with arguments

Ana Sokolova  University of Salzburg

University of Tokyo  30.10.2012
The big picture

\[ S_k \subseteq \Sigma^* \]

\[ S \subseteq \Sigma^* \]

- methods with arguments

semantics
sequential specification
legal sequences

relaxed semantics

Ana Sokolova University of Salzburg

University of Tokyo 30.10.2012
The big picture

\[ S_k \subseteq \Sigma^* \]

\[ S \subseteq \Sigma^* \]

semantics
sequential specification
legal sequences

relaxed semantics
leads to relaxed linearizability

\( \Sigma \) - methods with arguments

Ana Sokolova University of Salzburg

University of Tokyo 30.10.2012
Theoretical challenge

There are natural concrete relaxations...

Stack

Each **pop** pops one of the k-youngest elements
Each **push** pushes .....
Theoretical challenge

There are natural concrete relaxations...

Stack

Each **pop** pops one of the k-youngest elements
Each **push** pushes .....  

k-out-of-order relaxation
Theoretical challenge

There are natural concrete relaxations...

Stack

Each **pop** pops one of the k-youngest elements
Each **push** pushes ..... 

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

k-out-of-order relaxation
Theoretical challenge

There are natural concrete relaxations...

Stack

Each \textbf{pop} pops one of the k-youngest elements
Each \textbf{push} pushes ..... 

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

k-out-of-order relaxation

How is it reflected by a distance between sequences?

one distance for all?
Syntactic distances do not help

\[ \text{push}(a) \left[ \text{push}(i) \text{pop}(i) \right]^n \text{push}(b) \left[ \text{push}(j) \text{pop}(j) \right]^m \text{pop}(a) \]
Syntactic distances do not help

\[
push(a) \ [push(i)pop(i)]^npush(b) \ [push(j)pop(j)]^mpop(a)
\]

is a 1-out-of-order stack sequence
Syntactic distances do not help

\[
\text{push}(a) \left[ \text{push}(i)\text{pop}(i) \right]^n \text{push}(b) \left[ \text{push}(j)\text{pop}(j) \right]^m \text{pop}(a)
\]

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 $S$

- 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

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

$x \equiv y \iff \forall u \in \Sigma^*. (x u \in S \iff y u \in S)$

example: for stack

$\text{push}(a)\text{push}(b)\text{pop}(b)\text{push}(c) \equiv \text{push}(a)\text{push}(c)$
Semantic distances need a notion of state

- States are equivalence classes of sequences in $S$

  - Example: for stack
    
    $\text{push}(a)\text{push}(b)\text{pop}(b)\text{push}(c) \equiv \text{push}(a)\text{push}(c)$

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

\[ x \equiv y \iff \forall u \in \Sigma^*. (xu \in S \iff yu \in S) \]
Semantics goes operational

\[ S \subseteq \Sigma^* \] is the sequential specification

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

- states
- labels
- initial state

transition relation

\[ [s]_\equiv \xrightarrow{m} [sm]_\equiv \iff \text{sm} \in S \]
Semantics goes operational

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

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

transition relation

$[s]_{\equiv} \xrightarrow{m} [sm]_{\equiv} \iff sm \in S$
The framework

- Start from LTS(S)
- Add transitions with transition costs
- Fix a path cost function
Start from $\text{LTS}(S)$

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
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

Ana Sokolova  University of Salzburg
University of Tokyo  30.10.2012
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

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

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

distance and relaxation follow
Stack example

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

state evolution

c

Total cost

max = 2
sum = 3
Stack example

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

- Possible path cost functions $\text{max, sum, ...}$
Stack example

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

- Possible path cost functions \textit{max}, \textit{sum},...
Stack example

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

Possible path cost functions $\text{max, sum, }...$
It’s more general...
Generic out-of-order

\[
\text{segment\_cost}(q \xrightarrow{m} q') = |v|
\]

where \(v\) is a sequence of minimal length s.t.

\[(1)\]
\[
[uvw] = q, uvw \text{ is minimal, } uw \text{ is minimal}
\]
\[(1.1)\]
\[
[uvw] \equiv \rightarrow [u'w] \equiv , [uvw'] \equiv = q'
\]
\[(1.2)\]
\[
[uvw] \equiv \rightarrow [uvw'] \equiv , [uvw'] \equiv = q'
\]

\[(2)\]
\[
[uw] = q, uw \text{ is minimal, } uvw \text{ is minimal}
\]
\[(1.1)\]
\[
[uvw] \equiv \rightarrow [u'vw] \equiv , [uvw'] \equiv = q'
\]
\[(1.2)\]
\[
[uvw] \equiv \rightarrow [uvw'] \equiv , [uvw'] \equiv = q'
\]

transition cost

removing \(v\) enables a transition

inserting \(v\) enables a transition

goes with different path costs
Generic out-of-order

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

where \( v \) is a sequence of minimal length s.t.

1. \([uvw] = q\), \(uvw\) is minimal, \(uw\) is minimal
   
   1.1. \([uw] \xrightarrow{m} [u'w]\), \([u'vw] = q'\)
   
   1.2. \([uw] \xrightarrow{m} [uw']\), \([uvw'] = q'\)

2. \([uw] = q\), \(uw\) is minimal, \(uvw\) is minimal
   
   1.1. \([uvw] \xrightarrow{m} [u'vw]\), \([u'w] = q'\)
   
   1.2. \([uvw] \xrightarrow{m} [uvw']\), \([uw'] = q'\)
Generic out-of-order

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

where \( v \) is a sequence of minimal length s.t.

\begin{align*}
(1) & \quad [uvw] = q, \text{ }uvw\text{ is minimal, }uw\text{ is minimal} \\
& \quad \text{removing } v \text{ enables a transition (1.1)} \\
& \quad [uvw] \xrightarrow{m} [u'vw], [u'vw'] = q' \\
(1.2) & \quad [uw] \xrightarrow{m} [uw'], [uvw'] = q' \\
(2) & \quad [uw] = q, \text{ }uw\text{ is minimal, }uvw\text{ is minimal} \\
& \quad \text{inserting } v \text{ enables a transition (1.1)} \\
& \quad [uvw] \xrightarrow{m} [uvw'], [uvw'] = q' \\
& \quad [uw'] = q' \\
& \quad \text{goes with different path costs}
\end{align*}
Out-of-order stack

- Canonical representative of a state
- Add incorrect transitions with segment-costa
- Possible path cost functions max, sum,...
Out-of-order stack

- Canonical representative of a state
- Add incorrect transitions with segment-costs
- Possible path cost functions $\text{max, sum, ...}$

Also "shrinking window" restricted out-of-order
Out-of-order queue

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

- Possible path cost functions $\text{max}, \text{sum},...$
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,...
Out-of-order queue

- Canonical representative of a state
- Add incorrect transitions with segment-costs
- Possible path cost functions $\text{max, sum, ...}$
Out-of-order queue

- Sequence of \texttt{enq}'s with no matching \texttt{deq}

- Canonical representative of a state

- Add incorrect transitions with segment-costs

- Possible path cost functions \texttt{max}, \texttt{sum}, ...

Also "shrinking window" restricted out-of-order
Out-of-order variants

Queue

head

a b c d e ...

z
tail
Out-of-order variants

Queue

out-of-order k=3
restricted
out-of-order k=3

head

a b c d e ... z
tail

lateness k=3
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

- 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]

Notes:
- Distributed, one k-queue
- Syntactic, does not work for stacks
- (almost) all implement restricted out-of-order
Short-term history

- **SCAL queues** [KPRS’11] (distributed, one k-queue)
- **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

Syntactic, does not work for stacks

Not too well performing
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]

Ana Sokolova, University of Salzburg

not too well performing

University of Tokyo 30.10.2012
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]

Ana Sokolova  University of Salzburg
University of Tokyo  30.10.2012
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

lock-free = non-blocking

k-segment
Restricted-out-of-order k-Stack

lock-free = non-blocking

k-segment

1: loop:
2:   read consistent state
3:   try to add/remove an item (*)
4:   if successful:
5:     return
6:   else:
7:     try to repair the stack
8:     goto loop (retry)
Restricted-out-of-order k-Stack

lock-free = non-blocking

k-segment

add/remove segment

1: loop:
2:   read consistent state
3:   try to add/remove an item (*)
4:   if successful:
5:     return
6:   else:
7:     try to repair the stack
8:     goto loop (retry)
Restricted-out-of-order k-Stack

1: loop:
2:   read consistent state
3:   try to add/remove an item (*)
4:   if successful:
5:     return
6:   else:
7:     try to repair the stack
8:     goto loop (retry)

lock-free = non-blocking

CAS-based

k-segment

add/remove segment

Ana Sokolova University of Salzburg

University of Tokyo 30.10.2012
Stack

Scalability comparison

![Graph showing scalability comparison](image)

Ana Sokolova  University of Salzburg

University of Tokyo  30.10.2012
Stack

Scalability comparison

"80"-core machine
k-Stack

The more relaxed, the better

lock-free segment stack
Queue

Scalability comparison

![Graph showing scalability comparison](image)

Ana Sokolova  University of Salzburg

University of Tokyo  30.10.2012
Queue

Scalability comparison

“80”-core machine

Operations/ms (more is better)

Number of threads

- LB
- MS
- FC
- RD (r=40)
- SQ (s=40)
- ED
- RP
- BAG
- k-FIFO (k=40)
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

Contributions

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

all kinds of
Conclusions

Contributions

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

Difficult open problem

How to get from theory to practice?
Conclusions

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

Difficult open problem
How to get from theory to practice?

THANK YOU
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
- Learn from efficient implementations

THANK YOU