Concurrent Data Structures: Semantics and Relaxations

Ana Sokolova

IMDEA, Madrid, 17.7.17
Background big picture
Background big picture

Computer Science
Background big picture

Computer Science

Theoretical Computer Science
Background big picture

Computer Science

Theoretical Computer Science

Concurrency
Background big picture
Background big picture

Computer Science

Theoretical Computer Science

Concurrency

Algebra and Coalgebra

Formal Methods
Background big picture
Background big picture

Computer Science

Theoretical Computer Science

Concurrency

Security

Algebra and Coalgebra

Formal Methods

Probabilistic Systems
Background big picture

Computer Science

Theoretical Computer Science

Concurrency

Security

Algebra and Coalgebra

Real-Time Systems

Formal Methods

Probabilistic Systems

Ana Sokolova

IMDEA 17-7-17
Background big picture

Computer Science

Theoretical Computer Science

Concurrency

Memory Management Systems

Security

Algebra and Coalgebra

Formal Methods

Probabilistic Systems

Real-Time Systems

Ana Sokolova

IMDEA 17-7-17
Background big picture

Computer Science

- Theoretical Computer Science
- Formal Methods
- Security
- Algebra and Coalgebra
- Probabilistic Systems
- Real-Time Systems
- Memory Management Systems
- Concurrency
- Data Structures
Background big picture

Computer Science

Data Structures

Theoretical Computer Science

Security

Algebra and Coalgebra

Formal Methods

Probabilistic Systems

Concurrency

Memory

Management

Systems

Real-Time Systems
Favourites

Computer Science

- Theoretical Computer Science
- Memory Management Systems
- Real-Time Systems
- Concurrency
- Formal Methods
- Algebra and Coalgebra
- Security
- Data Structures

- Probabilistic Systems

Ana Sokolova

IMDEA 17-7-17
Concurrent Data Structures: Correctness and Performance
Semantics of concurrent data structures
Semantics of concurrent data structures

e.g. pools, queues, stacks
Semantics of concurrent data structures

- t1: enq(2)  deq(1)
- t2: enq(1)  deq(2)

e.g. pools, queues, stacks
Semantics of concurrent data structures

- **Sequential specification** = set of legal sequences

- **Consistency condition** = e.g. linearizability / sequential consistency

```plaintext
t1:  enq(2)  deq(1)
t2:  enq(1)  deq(2)
```

e.g. pools, queues, stacks
Semantics of concurrent data structures

- **Sequential specification** = set of legal sequences
  
  e.g. queue legal sequence
  \[\text{enq}(1)\text{enq}(2)\text{deq}(1)\text{deq}(2)\]

- **Consistency condition** = e.g. linearizability / sequential consistency

- e.g. pools, queues, stacks
Semantics of concurrent data structures

- **Sequential specification** = set of legal sequences

  - e.g. pool, queues, stacks

  - 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

Linearizability  [Herlihy,Wing ’90]

Sequential Consistency  [Lamport’79]
Consistency conditions

- Linearizability [Herlihy, Wing ’90]
- Sequential Consistency [Lamport’79]

there exists a legal sequence that preserves precedence
Consistency conditions

Linearizability [Herlihy, Wing ’90]

Sequential Consistency [Lamport’79]

there exists a legal sequence that preserves precedence

```
t1: enq(2)  deq(1)
t2: enq(1)  deq(2)
```
Consistency conditions

Linearizability [Herlihy, Wing ’90]

Sequential Consistency [Lamport’79]

t1: enq(2) deq(1)
t2: enq(1) deq(2)

there exists a legal sequence that preserves precedence

Ana Sokolova
Consistency conditions

Linearizability  [Herlihy, Wing '90]

Sequential Consistency  [Lamport '79]

there exists a legal sequence that preserves precedence
Consistency conditions

Linearizability  [Herlihy,Wing ’90]

Sequential Consistency  [Lamport’79]

there exists a legal sequence that preserves precedence

there exists a legal sequence that preserves per-thread precedence (program order)
Consistency conditions

**Linearizability**  [Herlihy,Wing ’90]

there exists a legal sequence that preserves precedence

```
t1:
enq(2)
deq(1)
tenq(1)
deq(2)
t2:
```

**Sequential Consistency**  [Lamport’79]

there exists a legal sequence that preserves per-thread precedence (program order)

```
t1:
enq(1)
deq(2)
t2:
deq(1)
enq(2)
```
Consistency conditions

Linearizability  [Herlihy,Wing ’90]

Sequential Consistency  [Lamport’79]

there exists a legal sequence that preserves precedence

there exists a legal sequence that preserves per-thread precedence (program order)
Consistency conditions

there exists a legal sequence that preserves precedence

Linearizability  [Herlihy,Wing ’90]

t1: enq(2)

t2: enq(1)

deq(1)

deq(2)

there exists a legal sequence that preserves per-thread precedence (program order)

Sequential Consistency  [Lamport’79]

t1: enq(1)

t2: deq(1)

deq(2)

enq(2)
Performance and scalability

throughput

# of threads / cores
Relaxations allow trading correctness for performance.
Relaxations allow trading correctness for performance provide the potential for better-performing implementations
Relaxing the semantics
Relaxing the semantics

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

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

Quantitative relaxations
POPL13
Relaxing the semantics

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

Quantitative relaxations POPL13

not “sequentially correct”
Relaxing the semantics

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

Quantitative relaxations
POPL13

Local linearizability
CONCUR16

not “sequentially correct”
Relaxing the semantics

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

Quantitative relaxations
POPL13

Local linearizability
CONCUR16

too weak

not "sequentially correct"
Relaxing the semantics

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

Local linearizability
CONCUR16

Quantitative relaxations
POPL13

not “sequentially correct”

for queues/stacks only (feel free to ask for more)

too weak
Relaxing the sequential specification
Relaxing the sequential specification

Quantitative relaxations (POPL13)
Goal

- trade correctness for performance
- in a controlled way with quantitative bounds

measure the error from correct behaviour
Goal

- trade correctness for performance
- in a controlled way with quantitative bounds

Stack - incorrect behavior

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

Stack

- top
- thread 1
- thread 2
- ... thread n

k-Relaxed stack

- top
- thread 1
- thread 2
- ... thread n

Ana Sokolova
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

\[ S \subseteq \Sigma^* \]

\( \Sigma \) - methods with arguments

sequential specification
legal sequences
The big picture

$\Sigma - \text{methods with arguments}$

$S \subseteq \Sigma^*$

$S_k \subseteq \Sigma^*$

sequential specification
legal sequences

relaxed sequential specification
sequences at distance up to $k$ from $S$
Syntactic distances do not help

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

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

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

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

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

• Two sequences in S are equivalent iff 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 iff 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 $S$
  
  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 iff 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

\[ [s]_\equiv \rightarrow [sm]_\equiv \iff sm \in S \]
Semantics goes operational

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

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

transition relation

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

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

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

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

- Start from \( \text{LTS}(S) \)

- Add transitions with transition costs

- Fix a path cost function
The relaxation framework

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

$\text{distance}$ - minimal cost on all paths labelled by the sequence
Generic out-of-order

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

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

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

Sequence of push's with no matching pop

• Possible path cost functions $\text{max}$, $\text{sum}$,...

also more advanced

Ana Sokolova
IMDEA 17-7-17
Relaxing the Consistency Condition
Relaxing the Consistency Condition
Local Linearizability
main idea
Local Linearizability
main idea

- **Partition** a history into a set of local histories
- **Require** linearizability per local history
Local Linearizability main idea

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

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

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

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

Local sequential consistency… is also possible
Local Linearizability
main idea

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

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

Local sequential consistency... is also possible

no global witness
Local Linearizability (queue) example

t1: enq(1) deq(2)

t2: enq(2) deq(1)
Local Linearizability (queue) example

(sequential) history not linearizable

<table>
<thead>
<tr>
<th>t1:</th>
<th>enq(1)</th>
<th>deq(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>t2:</td>
<td>enq(2)</td>
<td>deq(1)</td>
</tr>
</tbody>
</table>
Local Linearizability (queue) example

(sequential) history not linearizable
Local Linearizability (queue) example

(t1-induced history, linearizable)

(sequential) history not linearizable
Local Linearizability (queue) example

- **t1:**
  - `enq(1)`
  - `deq(2)`

- **t2:**
  - `enq(2)`
  - `deq(1)`

(sequential) history not linearizable

- t1-induced history, linearizable
Local Linearizability (queue) example

(t1-induced history, linearizable)
(t2-induced history, linearizable)

(sequential) history not linearizable
Local Linearizability (queue) example

- **t1:**
  - enq(1)
  - deq(2)

- **t2:**
  - enq(2)
  - deq(1)

- (sequential) history
  - not linearizable

- t2-induced history, linearizable
- t1-induced history, linearizable

- locally linearizable

Ana Sokolova
IMDEA 17-7-17
Local Linearizability (queue) definition
Local Linearizability (queue) definition

Queue signature $\Sigma = \{\text{enq}(x) \mid x \in V\} \cup \{\text{deq}(x) \mid x \in V\} \cup \{\text{deq}(\text{empty})\}$
Local Linearizability
(queue) definition

Queue signature \( \Sigma = \{ \text{enq}(x) \mid x \in V \} \cup \{ \text{deq}(x) \mid x \in V \} \cup \{ \text{deq}(\text{empty}) \} \)

For a history \( h \) with a thread \( T \), we put

\[
I_T = \{ \text{enq}(x)^T \in h \mid x \in V \}
\]

\[
O_T = \{ \text{deq}(x)^T \in h \mid \text{enq}(x)^T \in I_T \} \cup \{ \text{deq}(\text{empty}) \}
\]
Local Linearizability (queue) definition

Queue signature $\Sigma = \{\text{enq}(x) | x \in V\} \cup \{\text{deq}(x) | x \in V\} \cup \{\text{deq}(\text{empty})\}$

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

$$\mathcal{I}_T = \{\text{enq}(x)^T \in h | x \in V\}$$

$$\mathcal{O}_T = \{\text{deq}(x)^{T'} \in h | \text{enq}(x)^T \in \mathcal{I}_T\} \cup \{\text{deq}(\text{empty})\}$$
Local Linearizability (queue) definition

Queue signature $\Sigma = \{\text{enq}(x) \mid x \in V\} \cup \{\text{deq}(x) \mid x \in V\} \cup \{\text{deq}(\text{empty})\}$

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

$\text{I}_T = \{\text{enq}(x)^T \in h \mid x \in V\}$

$\text{O}_T = \{\text{deq}(x)^{T'} \in h \mid \text{enq}(x)^T \in \text{I}_T\} \cup \{\text{deq}(\text{empty})\}$

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
Local Linearizability (queue) definition

Queue signature \( \Sigma = \{\text{enq}(x) \mid x \in V\} \cup \{\text{deq}(x) \mid x \in V\} \cup \{\text{deq}(\text{empty})\} \)

For a history \( h \) with a thread \( T \), we put

\[
I_T = \{\text{enq}(x)^T \in h \mid x \in V\}
\]
\[
O_T = \{\text{deq}(x)^{T'} \in h \mid \text{enq}(x)^T \in I_T\} \cup \{\text{deq}(\text{empty})\}
\]

\( h \) is locally linearizable iff every thread-induced history \( h_T = h \mid (I_T \cup O_T) \)

is linearizable.
Where do we stand?
Where do we stand?

In general

Local Linearizability

Linearizability

Sequential Consistency

Ana Sokolova

IMDEA 17-7-17
Where do we stand?

For queues (and most container-type data structures)

Local Linearizability

Linearizability

Sequential Consistency
Properties
Properties

Local linearizability is compositional
Properties

Local linearizability is compositional

like linearizability
unlike sequential consistency
Local linearizability is compositional

\( h \) (over multiple objects) is locally linearizable
iff
each per-object subhistory of \( h \) is locally linearizable
Local linearizability is compositional:

- $h$ (over multiple objects) is locally linearizable if and only if each per-object subhistory of $h$ is locally linearizable.

Local linearizability is modular / “decompositional” like linearizability

unlike sequential consistency
Properties

Local linearizability is compositional

\( h \) (over multiple objects) is locally linearizable
\[ \text{iff} \]
each per-object subhistory of \( h \) is locally linearizable

Local linearizability is modular / “decompositional”

like linearizability
unlike sequential consistency

uses decomposition into smaller histories, by definition
Local linearizability is compositional

\( h \) (over multiple objects) is locally linearizable

iff

each per-object subhistory of \( h \) is locally linearizable

Local linearizability is modular / “decompositional”

like linearizability

unlike sequential consistency

uses decomposition into smaller histories, by definition

may allow for modular verification
Generic Implementations
Generic Implementations

Your favorite linearizable data structure implementation
Generic Implementations

Your favorite linearizable data structure implementation
Generic Implementations

Your favorite linearizable data structure implementation turns into a locally linearizable implementation by: \( \Phi \)
Generic Implementations

Your favorite linearizable data structure implementation

turns into a locally linearizable implementation by:

\[ t_1, t_2, \ldots, t_n \]
Generic Implementations

Your favorite linearizable data structure implementation turns into a locally linearizable implementation by:

segment of possibly dynamic size (n)
Generic Implementations

Your favorite linearizable data structure implementation

turns into a locally linearizable implementation by:

segment of possibly dynamic size (n)

local inserts / global (randomly distributed) removes
Generic Implementations

Your favorite linearizable data structure implementation

turns into a locally linearizable implementation by:

LLD Φ (locally linearizable)

local inserts / global (randomly distributed) removes

segment of possibly dynamic size (n)
Generic Implementations

Your favorite linearizable data structure implementation

turns into a locally linearizable implementation by:

- LLD $\Phi$ (locally linearizable)
- LL+D $\Phi$ (also pool linearizable)

segment of possibly dynamic size (n)

local inserts / global (randomly distributed) removes
The set with respect to explicit by defining a set. For ease of presentation, we consider. By doing so, the read-operation will end up in the thread-induced history of all threads that wrote a value that was returned. However, there is no further synchronization between those threads.

What if these values were written by different threads? The situation gets more complex:

- For the following discussion, we will make this more generic implementation scheme that turns a linearizable data structure synchronization at the program level. For example, queues which can relax their semantics if operations are performed in a way that gives a programmer control over when a relaxation is performed in thread.

Local linearizability utilizes the idea of decomposing a history directly corresponds to the first set in the definition of observations in history. For the following discussion, we will make this more generic implementation scheme that turns a linearizable data structure synchronization at the program level. For example, queues which can relax their semantics if operations are performed in a way that gives a programmer control over when a relaxation is performed in thread.

Local linearizability has desirable properties like compositionality and linearizability (further implications of decomposition to correct linearization). How does this affect performance and scalability? There are at least two improvements of performance and scalability. There are at least two improvements of performance and scalability. There are at least two improvements of performance and scalability. There are at least two improvements of performance and scalability.

There are different answers to this question. For the following discussion, we will make this more generic implementation scheme that turns a linearizable data structure synchronization at the program level. For example, queues which can relax their semantics if operations are performed in a way that gives a programmer control over when a relaxation is performed in thread.

8. Conclusions

Investigation of ExLL and efficient implementations of correspond-
Performance

- LL+D MS queue performs significantly better than MS queue

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

![Graph showing performance and scalability of producer-consumer microbenchmarks with an increasing number of threads on a 40-core machine.](image)

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

LLD $\Phi$ performs significantly better than $\Phi$
Performance

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

LL+D MS queue performs better than the best known pools
Thank You!

and many thanks to my dear coauthors