Linearizability via Order Extension Theorems

Ana Sokolova
IRIF, 28.5.2018
• Part I: Concurrent data structures
  correctness and performance

• Part II: Order extension results for
  verifying linearizability
Data structures

- Queue FIFO

- Stack LIFO

- Pool unordered
Concurrent data structures

- **Queue FIFO**
  ![Queue FIFO diagram]

- **Stack LIFO**
  ![Stack LIFO diagram]

- **Pool unordered**
  ![Pool unordered diagram]
Semantics of concurrent data structures

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

  e.g. queues

  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]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1:</td>
<td>t2:</td>
</tr>
<tr>
<td>enq(2)²</td>
<td>1enq(1)</td>
</tr>
<tr>
<td>deq(1)³</td>
<td>deq(2)⁴</td>
</tr>
</tbody>
</table>

Sequential Consistency [Lamport’79]

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1:</td>
<td>t2:</td>
</tr>
<tr>
<td>1enq(1)</td>
<td>deq(2)⁴</td>
</tr>
<tr>
<td>deq(1)²</td>
<td>enq(2)³</td>
</tr>
</tbody>
</table>

Consistency is about extending partial orders to total orders.

There exists a legal sequence that preserves precedence order.

There exists a legal sequence that preserves per-thread precedence (program order).
Performance and scalability

[Graph showing the relationship between throughput and the number of threads/cores, with different lines representing varying performance levels.]

Throughput vs. # of threads / cores

Ana Sokolova

IRIF 28.5.18
Relaxations allow trading correctness for performance.

provide the potential for better-performing implementations.
Relaxing the Semantics

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

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

Local linearizability
Haas, Henzinger, Holzer, ..., S, Veith CONCUR16
Relaxing the sequential specification
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

correct in a relaxed stack ... 2-relaxed? 3-relaxed?

measure the error from correct behaviour
How can relaxing help?

Stack

- top
- c
- b
- a

thread 1
thread 2
... thread n

k-Relaxed stack

- top
- c
- b
- a

k

thread 1
thread 2
... thread n
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

$\Sigma$ - methods with arguments
The big picture

$S \subseteq \Sigma^*$

Sequential specification
Legal sequences

$S_k \subseteq \Sigma^*$

Relaxed sequential specification
Sequences at distance up to k from S

$\Sigma$ - methods with arguments
Relaxing the Consistency Condition

Local Linearizability (CONCUR16)
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-induced history), linearizable

(t2-induced history), linearizable

locally linearizable

(sequential) history not linearizable
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?

In general

Local Linearizability

Linearizability

Sequential Consistency
Where do we stand?

For queues (and most container-type data structures)

Local Linearizability

Linearizability

Sequential Consistency
Lead to scalable implementations

e.g. k-FIFO, k-Stack

Locally linearizable distributed implementation

k-out-of-order queue

Local inserts / global removes
As stated in Remark 1, observer methods are added to the thread-induced history. From other threads that write values read by the read-operation into histories, another possibility is to also include all write-operations considered. By doing so, the read-operation will end up in the thread-by-different threads?

There are different answers to this question. Usually, relaxations are applied in a way that gives a programming style freedom, but it enables new programming styles. Moreover, it shows the virtue of the simplicity of local linearizability's definition. A theoretical investigation of ExLL and efficient implementations of corresponding linearizable and relaxed implementations. On the theoretical side, we prove that local linearizability has desirable properties like compositionality.

Local linearizability utilizes the idea of decomposing a history. We believe that ExLL is a very promising research direction as it enables new programming styles. Moreover, it shows the virtue of the simplicity of local linearizability's definition. A theoretical investigation of ExLL and efficient implementations of corresponding linearizable and relaxed implementations. On the theoretical side, we prove that local linearizability has desirable properties like compositionality.

LL+D MS queue performs significantly better than MS queue.
Performance

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

![Graph showing performance of different queue types with varying number of threads]

- **LLD φ** performs significantly better than φ

---

**Figure 1:** Performance of different queue types on a 40-core (2 hyperthreads per core) machine with an increasing number of threads, measured in million operations per second (more is better).

- **Performance Metric:** Million operations per second (more is better).
- **X-axis:** Number of threads.
- **Y-axis:** Million operations per second.
There are different answers to this question. For the following discussion, we will make this more precise.

One possibility is to ignore returned values that are written by other threads that write values read by the read-operation into histories. Another possibility is to also include all write-operations of programs that are (in)sensitive to notions of (relaxed) semantics. However, there is no further synchronization between those threads. An insert-operation happens or not. ExLL enables the introduction of fine-grained data structure synchronization at the program level. For example, insert-operations can be extended by thread identifiers to specify global parameterization, we define two more sets. First, given a thread index \( t \), we define the set of insert-operations that are local to thread \( t \), yielding an intuitive and verifiable consistency condition.

For ease of presentation with respect to history including values). Then, given a set of programs that are (in)sensitive to notions of (relaxed) semantics.

Figure 8: Performance and scalability of producer-consumer micro-benchmarks: million operations per sec (more is better).

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

(b) Stacks, LL stacks, and “stack-like” pools.
Linearizability via Order Extension Theorems

foundational results for verifying linearizability

joint work with

Harald Woracek
Inspiration

Queue sequential specification (axiomatic)

\( s \) is a legal queue sequence

iff

1. \( s \) is a legal pool sequence, and
2. \( \text{enq}(x) <_s \text{enq}(y) \land \text{deq}(y) \in s \implies \text{deq}(x) \in s \land \text{deq}(x) <_s \text{deq}(y) \)

Queue linearizability (axiomatic)

\( h \) is queue linearizable

iff

1. \( h \) is pool linearizable, and
2. \( \text{enq}(x) <_h \text{enq}(y) \land \text{deq}(y) \in h \implies \text{deq}(x) \in h \land \text{deq}(y) \prec_h \text{deq}(x) \)

As well as

Reducing Linearizability to State Reachability

[Bouajjani, Emmi, Enea, Hamza]

ICALP15 + …

precedence order

Ana Sokolova

IRIF 28.5.18
**Concurrent Queues**

Data independence $\Rightarrow$ 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
  - $\text{deq} \Rightarrow v$

- Value $v$ dequeued before being enqueued
  - $\text{deq} \Rightarrow v$
  - $\text{enq}(v)$

- Value $v$ dequeued twice
  - $\text{deq} \Rightarrow v$
  - $\text{deq} \Rightarrow v$

- Value $v_1$ and $v_2$ dequeued in the wrong order
  - $\text{enq}(v_1)$
  - $\text{enq}(v_2)$
  - $\text{deq} \Rightarrow v_2$
  - $\text{deq} \Rightarrow v_1$
  - $\text{deq} \Rightarrow v_{n-2}$
  - $\text{deq} \Rightarrow v_n$
Linearizability verification

Data structure
- signature $\Sigma$ - set of method calls including data values
- sequential specification $S \subseteq \Sigma^*$, prefix closed

Sequential specification via violations
Extract a set of violations $V$, relations on $\Sigma$, such that $s \in S$ iff $s$ has no violations.

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.

we build $CV$ iteratively from $V$

identify sequences with total orders

$\mathcal{P}(s) \cap V = \emptyset$

legal sequence

concurrent history
Pool without empty removals

**Pool sequential specification (axiomatic)**

\( s \) is a legal pool (without empty removals) sequence iff

1. \( \text{rem}(x) \in s \Rightarrow \text{ins}(x) \in s \land \text{ins}(x) <_s \text{rem}(x) \)

**Pool linearizability (axiomatic)**

\( h \) is pool (without empty removals) linearizable iff

1. \( \text{rem}(x) \in h \Rightarrow \text{ins}(x) \in h \land \text{rem}(x) \prec_h \text{ins}(x) \)

\( V \) violations
\( \text{rem}(x) < s \text{ins}(x) \)

\( CV \) violations
\( = V \) violations
Queue without empty removals

Queue sequential specification (axiomatic)

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

1. \( \text{deq}(x) \in s \implies \text{enq}(x) \in s \land \text{enq}(x) <_s \text{deq}(x) \)

2. \( \text{enq}(x) <_s \text{enq}(y) \land \text{deq}(y) \in s \implies \text{deq}(x) \in s \land \text{deq}(x) <_s \text{deq}(y) \)

Queue linearizability (axiomatic)

\( h \) is queue (without empty removals) linearizable iff

1. \( \text{rem}(x) \in h \implies \text{ins}(x) \in h \land \text{rem}(x) \not<_h \text{ins}(x) \)

2. \( \text{enq}(x) <_h \text{enq}(y) \land \text{deq}(y) \in h \implies \text{deq}(x) \in h \land \text{deq}(y) \not<_h \text{deq}(x) \)

V violations
\( \text{deq}(x) <_s \text{enq}(x) \) and
\( \text{enq}(x) <_s \text{enq}(y) \land \text{deq}(y) <_s \text{deq}(x) \)

CV violations
\( = V \) violations

Ana Sokolova

IRIF 28.5.18
Pool sequential specification (axiomatic)

\[ s \text{ is a legal pool (with empty removals) sequence } \]
\[ \text{iff} \]
\[ 1. \ \text{rem}(x) \in s \Rightarrow \text{ins}(x) \in s \land \text{ins}(x) <_s \text{rem}(x) \]
\[ 2. \ \text{rem}(\bot) <_s \text{rem}(x) \Rightarrow \text{rem}(\bot) <_s \text{ins}(x) \land \text{ins}(x) <_s \text{rem}(\bot) \Rightarrow \text{rem}(x) <_s \text{rem}(\bot) \]

Pool linearizability (axiomatic)

\[ h \text{ is pool (with empty removals) linearizable } \]
\[ \text{iff} \]
\[ 1. \ \text{rem}(x) \in h \Rightarrow \text{ins}(x) \in h \land \text{rem}(x) \not<_h \text{ins}(x) \]
\[ 2. \ \ldots \ \ldots \]

V violations
\[ \text{rem}(x) <_s \text{ins}(x) \]
and
\[ \text{ins}(x) <_s \text{rem}(\bot) <_s \text{rem}(x) \]

infinitely many CV violations
\[ \text{ins}(x_1) <_h \text{rem}(\bot) \land \text{ins}(x_2) <_h \text{rem}(x_1) \land \ldots \land \text{ins}(x_{n+1}) <_h \text{rem}(x_n) \land \text{rem}(\bot) <_h \text{rem}(x_{n+1}) \]
It works for

- Pool without empty removals
- Queue without empty removals
- Priority queue without empty removals
- Pool
- Queue
- Priority queue

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

\[
\begin{align*}
\text{PO}[\mathcal{V}] &= \{R \in \text{PO} \mid \mathcal{P}(R) \cap \mathcal{V} = \emptyset\} \\
\text{IO}[\mathcal{V}] &= \{R \in \text{IO} \mid \mathcal{P}(R) \cap \mathcal{V} = \emptyset\} \\
\text{TO}[\mathcal{V}] &= \{R \in \text{TO} \mid \mathcal{P}(R) \cap \mathcal{V} = \emptyset\}
\end{align*}
\]

\forall (a, b), (c, d) \in R. (a, d) \in R \lor (c, b) \in R
The problem

Given a set of violations $\mathcal{V}$, find a “small” set of violations $\mathcal{V}'$ such that

$$\forall R \in IO[\mathcal{V}]. \exists \overline{R} \in TO[\mathcal{V}]. \overline{R} \supseteq R$$

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 $\mathcal{V}' = \mathcal{V}$.

Theorem (singleton violations)

this solves the case of pool without empty removals
The closures

\[ \text{Clos}_O(\mathcal{V}) = \bigcap_{S \in O[\mathcal{V}]} \mathcal{P}(S)^c \]

O-closure of a set of violations

Proposition

\[ \forall R \in IO[\mathcal{V}]. \exists \overline{R} \in TO[\mathcal{V}]. \overline{R} \supseteq R \]

iff

\[ \text{Clos}_TO(\mathcal{V}) = \text{Clos}_IO(\mathcal{V}') \]

monotone, extensive, idempotent
How does it work?

**Theorem**

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

1. $\forall N, M \in \mathcal{V} \cdot \forall (a_1, a_2) \in N. |\{(b_1, b_2) \in M \mid a_2 = b_1\}| \leq 1$

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

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
\[
\{(a, y) \mid (a, x) \in N_2\} \\
\cup \{(x, b) \mid (y, b) \in N_2\} \\
\cup \{(a, b) \in N_2 \mid b \neq x \land a \neq y\} \\
\cup N_1 \backslash \{(x, y)\}
\]

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

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

Thank You!