Local Linearizability

Ana Sokolova

joint work with:

Andreas Haas
Andreas Holzer
Michael Lippautz
Ali Sezgin

Tom Henzinger
Christoph Kirsch
Hannes Payer
Helmut Veith
Concurrent Data Structures
Correctness and Performance
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. the concurrent history above is a linearizable queue concurrent history

### Example

- \(t_1:\) enq(2) deq(1)
- \(t_2:\) enq(1) deq(2)

\[\text{e.g. pools, queues, stacks}\]
Consistency conditions

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

There exists a sequential witness that preserves precedence.

- **t1:** enq(2)\(^2\) deq(1)\(^3\)
- **t2:** 1\(^{\text{enq}(1)}\) deq(2)\(^4\)

**Sequential Consistency** [Lamport '79]

There exists a sequential witness that preserves per-thread precedence (program order).

- **t1:** 1\(^{\text{enq}(1)}\) deq(2)\(^4\)
- **t2:** deq(1)\(^2\) enq(2)\(^3\)
Performance and scalability
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

Not “sequentially correct”

Local linearizability in this talk

for queues only (feel free to ask for more)

too weak
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)`

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

(sequential) history not linearizable

Locally linearizable

RiSE/PUMA Workshop 2015
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 \( n \) threads, we put

\[
\begin{align*}
\text{In}_h(i) &= \{\text{enq}(x)^i \in h \mid x \in V\} \\
\text{Out}_h(i) &= \{\text{deq}(x)^j \in h \mid \text{enq}(x)^i \in \text{In}_h(i)\} \cup \{\text{deq}(\text{empty})\}
\end{align*}
\]

- \( h \) is locally linearizable iff every thread-induced history \( h_i = h \mid (\text{In}_h(i) \cup \text{Out}_h(i)) \) is linearizable.
Generalizations of Local Linearizability

Signature $\Sigma$

For a history $h$ with $n$ threads, choose

- $\text{In}_h(i)$: in-methods of thread $i$,
- $\text{Out}_h(i)$: methods that go out $h_i$

by increasing the in-methods, LL gradually moves to linearizability.

$h$ is locally linearizable iff every thread-induced history

$$h_i = h \mid (\text{In}_h(i) \cup \text{Out}_h(i))$$

is linearizable.
Where do we stand?

- Linearizability
- Local Linearizability
- Sequential Consistency

In general
Where do we stand?

For queues (and all pool-like data structures)

Local Linearizability

Linearizability

Sequential Consistency
Where do we stand?

Local Linearizability & Pool-seq.cons.

- Linearizability
- Sequential Consistency

C: For queues
Properties

Local linearizability is compositional

\[ h \text{ (over multiple objects) is locally linearizable} \]

\[ \iff \]

\[ \text{each per-object subhistory of } h \text{ is locally linearizable} \]

Local linearizability is modular / “decompositional”

\[ \text{uses decomposition into smaller histories, by definition} \]

may allow for modular verification

like linearizability

unlike sequential consistency
Verification (queue)

Queue sequential specification (axiomatic)

\[ s \text{ is a legal queue sequence} \iff \]
1. \( s \) is a legal pool sequence, and
2. \( \text{enq}(x) < s \text{ enq}(y) \land \text{deq}(y) \in s \Rightarrow \text{deq}(x) \in s \land \text{deq}(x) < s \text{ deq}(y) \)

Queue linearizability (axiomatic)

\[ h \text{ is queue linearizable} \iff \]
1. \( h \) is pool linearizable, and
2. \( \text{enq}(x) < h \text{ enq}(y) \land \text{deq}(y) \in h \Rightarrow \text{deq}(x) \in h \land \text{deq}(y) < h \text{ deq}(x) \)

precedence order
Verification (queue)

Queue sequential specification (axiomatic)

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

Queue local linearizability (axiomatic)

\( h \) is queue locally linearizable
iff
1. \( h \) is pool locally linearizable, and
2. \( \text{enq}(x) <_h \text{enq}(y) \land \text{deq}(y) \in h \Rightarrow \text{deq}(x) \in h \land \text{deq}(y) <_h \text{deq}(x) \)

thread-local precedence order
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 dynamic size (n)

local inserts / global (randomly distributed) removes
Performance

![Graph showing performance comparison between different types of queues and pools. The graph plots million operations per second (more is better) against the number of threads. LL+D MS queue performs significantly better than MS queue.]

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Number of Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>MS LCRQ</td>
<td>2, 10, 20, 30, 40, 50, 60, 70, 80</td>
</tr>
<tr>
<td>LL+D MS LCRQ</td>
<td>2, 10, 20, 30, 40, 50, 60, 70, 80</td>
</tr>
<tr>
<td>LLD LCRQ</td>
<td>2, 10, 20, 30, 40, 50, 60, 70, 80</td>
</tr>
<tr>
<td>LLD k-FIFO</td>
<td>2, 10, 20, 30, 40, 50, 60, 70, 80</td>
</tr>
<tr>
<td>1-RA DQ</td>
<td>2, 10, 20, 30, 40, 50, 60, 70, 80</td>
</tr>
</tbody>
</table>
Performance

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

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

![Graph showing performance of different data structures](image)

Thank You!

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

- **MS LCRQ**
- **LL+D MS LLD LCRQ**
- **1-RA DQ LLD k-FIFO**

### LLD MS queue performs better than the best known pools

- **LLD MS queue**
- **LLD LCRQ**
- **LLD k-FIFO**

---

**Ana Sokolova**

**University of Salzburg**

**RiSE/PUMA Workshop 2015**