Semantics of Concurrent Data Structures

Ana Sokolova

AVM, 25.9.2018
Concurrent data structures

**correctness and performance**

structure and power via semantic relaxations

* New results enabling verifying linearizability
Concurrent Data Structures
Correctness and Relaxations

Hannes Payer
Google

Tom Henzinger
IST Austria

Christoph Kirsch
University of Salzburg

Ali Sezgin
University of Cambridge

Andreas Haas
Google

Michael Lippautz

Andreas Holzer
Google

Helmut Veith
TU Wien

RiSE
Rigorous Systems Engineering
Data structures

- **Queue FIFO**
  - enq
  - f e d c b a
  - deq

- **Stack LIFO**
  - push
  - x y z
  - pop

- **Pool unordered**
  - ins
  - k n l
  - rem
  - j m o
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

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

  e.g. queue legal sequence
  enq(1)enq(2)deq(1)deq(2)

  e.g. the concurrent history above is a linearizable queue concurrent history

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

  e.g. queues
Consistency conditions

Linearizability [Herlihy, Wing ’90]

Sequential Consistency [Lamport’79]

A history is ... wrt a sequential specification iff

there exists a legal sequence that preserves precedence order

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

consistency is about extending partial orders to total orders

there exists a legal sequence that preserves precedence order
Performance and scalability

throughput

# of threads / cores

:-)))

:-)

:-(

:-\
Relaxations allow trading correctness for performance.
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
- thread 1
- thread 2
- thread n

k-Relaxed stack

- top
- thread 1
- thread 2
- thread n

Ana Sokolova
AVM 25.9.18
We have got

- 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

\[ \Sigma \text{- methods with arguments} \]

Ana Sokolova
The big picture

$S \subseteq \Sigma^*$

$Sk \subseteq \Sigma^*$

sequential specification
legal sequences

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

(sequential) history not linearizable

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

locally 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
Performance

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

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

Figure 8: Performance and scalability of producer-consumer micro-benchmarks. We perform a range of experiments with increasing number of threads and compare different implementations of containers. The chart shows the number of operations per second (more is better) for various data structures. The chart includes MS, LCRQ, k-FIFO, LL+D MS, LLD LCRQ, LLD k-FIFO, and 1-RA DQ. It can be observed that LLD ϕ performs significantly better than ϕ.

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

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

LL+D MS queue performs better than the best known pools
Scal

High-Performance Multicore-Scalable Computing

We study the design, implementation, performance, and scalability of concurrent objects on multicore systems by analyzing the apparent trade-off between adherence to concurrent data structure semantics and scalability.

scal.cs.uni-salzburg.at

Thank You!

Ana Sokolova

AVM 25.9.18
We study the design, implementation, performance, and scalability of concurrent objects on multicore systems by analyzing the apparent trade-off between adherence to concurrent data structure semantics and scalability.

Scal.cs.uni-salzburg.at

High-Performance Multicore-Scalable Computing

Thank You!

Ana Sokolova

AVM 25.9.18
Concurrent Data Structures
Correctness and Performance

Thank You!
Linearizability via Order Extension Theorems

joint work with

Harald Woracek

foundational results for verifying linearizability
Queue sequential specification (axiomatic)

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

Queue linearizability (axiomatic)

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

As well as
Reducing Linearizability to State Reachability
[Bouajjani, Emmi, Enea, Hamza]
ICALP15 + …

precedence order
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
$\text{enq}(v_1)$  $\text{enq}(v_2)$  $\text{deq} \Rightarrow v_2$  $\text{deq} \Rightarrow v_1$

Dequeue wrongfully returns empty
$\text{deq} \Rightarrow \text{empty}$

$\text{enq}(v_1)$  $\text{enq}(v_2)$  $\text{deq} \Rightarrow v_1$  $\text{deq} \Rightarrow v_2$

$\text{enq}(v_n)$  $\text{deq} \Rightarrow v_n$
Problems (stack)

Stack sequential specification (axiomatic)

- $s$ is a legal stack sequence
  iff
1. $s$ is a legal pool sequence, and
2. $\text{push}(x) \prec_s \text{push}(y) \prec_s \text{pop}(x) \Rightarrow \text{pop}(y) \in s \land \text{pop}(y) \prec_s \text{pop}(x)$

Stack linearizability (axiomatic)

- $h$ is stack linearizable
  iff
1. $h$ is pool linearizable, and
2. $\text{push}(x) \prec_h \text{push}(y) \prec_h \text{pop}(x) \Rightarrow \text{pop}(y) \in h \land \text{pop}(x) \prec_h \text{pop}(y)$
Problems (stack)

Stack sequential specification (axiomatic)

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

Stack linearizability (axiomatic)

\( h \) is stack linearizable
iff
1. \( h \) is pool linearizable, and
2. \( \text{push}(x) <_h \text{push}(y) <_h \text{pop}(x) \Rightarrow \text{pop}(y) \in h \land \text{pop}(x) <_h \text{pop}(y) \)
Problems (stack)

Stack linearizability (axiomatic)

\( h \) is stack linearizable iff

1. \( h \) is pool linearizable, and
2. \( \text{push}(x)<_h \text{push}(y)<_h \text{pop}(x) \Rightarrow \text{pop}(y) \in h \land \text{pop}(x) \not\leq_h \text{pop}(y) \)

not stack linearizable

Ana Sokolova

AVM 25.9.18
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

it is easy to find a large \( CV \), but difficult to find a small representative

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 \( V \)

legal sequence

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

concurrent history

Ana Sokolova

AVM 25.9.18
It works for

• Pool without empty removals
• Queue without empty removals
• Priority queue without empty removals

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!