Semantics of Concurrent Data Structures

Ana Sokolova

IFIP WG 1.3 meeting at Royal Holloway, 5.7.2018
• Part I: Concurrent data structures correctness and performance

structure and power

via semantic relaxations

• Part II: Order extension results for verifying linearizability
Concurrent Data Structures
Correctness and Relaxations
Data structures

- Queue FIFO

- Stack LIFO

- Pool unordered
Concurrent data structures

- Queue FIFO
- Stack LIFO
- Pool unordered
Semantics of concurrent data structures

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

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

- *Example:*
  
  ```
  t1: enq(2) deq(1)
  t2: enq(1) deq(2)
  ```

- *Example queue legal sequence:* enq(1)enq(2)deq(1)deq(2)

- *Example concurrent history:* e.g. the concurrent history above is a linearizable queue concurrent history
Consistency conditions

Linearizability [Herlihy,Wing ’90]

Sequential Consistency [Lamport’79]

There exists a legal sequence that preserves precedence order

Consistency is about extending partial orders to total orders

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

A history is ... wrt a sequential specification iff

[IFIP WG 1.3 - RHUL 5.7.18]
Performance and scalability

throughput

# of threads / cores

:-)))

:-)

:-(

:-\
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

Quantitative relaxations (POPL13)
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

IFIP WG 1.3 - RHUL 5.7.18
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

$\Sigma - \text{methods with arguments}$
The big picture

$S_k \subseteq \Sigma^*$

sequential specification
legal sequences

relaxed sequential specification
sequences at distance up to $k$ from $S$
Relaxing the Consistency Condition
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*

$\text{t1:}\ enq(1)\ deq(2)\ enq(2)\ deq(1)$

$\text{t2:}\ enq(2)$

*(sequential) history not linearizable*

$t1$-induced history, linearizable

$t2$-induced history, linearizable

locally 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

LLD $\Phi$

LL+D $\Phi$
For ease of presentation, we will define a set \( \text{Obs} \) with respect to history including values. Then, given a set \( \text{Obs} \) of observations in history, we can impose synchronization on all observers, i.e., add the observer methods.

As stated in Remark 1, observer methods are added to the memory consistency conditions. On the practical side, we provide a theoretically control the parallelism of the application in a very fine grained manner, then again, there is no control over when a relaxation eventuates.

Local linearizability utilizes the idea of decomposing a history into a sequence of insert-operations and a sequence of read-operations. How- ever, there is no further synchronization between those threads induced histories of all threads that wrote a value that was returned. In such a case, the situation gets more complex:

What if these values were written by different threads?

One possibility is to ignore returned values that are written by different threads.

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

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

One hyperthread runs several threads. In a machine (2 hyperthreads per core) running crobenchmarks with an increasing number of threads on a 40-core machine, LL+D MS queue performs significantly better than MS queue.
As stated in Remark 1, observer methods are added to the memory consistency conditions. On the practical side, we provide a generic implementation scheme that turns a linearizable data structure into a locally linearizable one, resulting in implementations with superior performance and scalability compared to linearizable and relaxed implementations. On the theoretical side, we prove that priority performance and scalability compared to linearizable and relaxed implementations. Local linearizability utilizes the idea of decomposing a history as follows: for a thread-induced history, another possibility is to also include all write-operations except the appearance of this operation in all the thread-induced histories. Another possibility is to ignore returned values that are written by other threads that write values read by the read-operation into histories. However, there is no further synchronization between those threads. ExLL adds an additional set $P_{\text{Sync}}$ and $I_{\text{Sync}}$ directly corresponds to the first set in the definition of $\text{LocalIns}$: given three sets $P, I, q_{\text{Obs}}$, we define the set $\text{Obs} = \{ p, q_{\text{Obs}} \}$. Given these three sets, we can define an extension of standard local linearizability corresponds to $h_{\text{Local}}, p_h, I_p, q_{\text{Loc}}, q_{\text{P}}$. For the following discussion, we will make this more significant better than $\Phi$. Figure 8: Performance and scalability of producer-consumer microbenchmarks with an increasing number of threads on a 40-core (2 hyperthreads per core) machine. (b) Stacks, LL stacks, and “stack-like” pools.
The set \( \text{Obs} \) is defined explicitly by defining a set \( \text{Rems} \) as stated in Remark 1, observer methods are added to the histories. Another possibility is to also include all write-operations induced histories of all threads that wrote a value that was returned.

Figure 8: Performance and scalability of producer-consumer microbenchmarks.

One possibility is to ignore returned values that are written by other threads.

What if these values were written to a corre-
correspondence function that gives a pro-
cal linearizability (further implications of decomposition to correct-
directions for future work: (1) A detailed study of extensions of lo-
generic implementation scheme that turns a linearizable data struc-
tions of linearizability are performed in a way that gives a pro-
relaxed implementations. On the theoretical side, we prove that
into a set of thread-induced histories and requiring consistency
into a locally linearizable one, resulting in im-
programming only very little control over the degree of semantic re-
operations that appear in
local linearizability which we call in the following ExLL:

\[ \text{LocalIns} \]

\[ \text{Sync} \]

Another example are quiescent consistent
pools.

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

LL+D MS queue performs better than the best known pools.
Linearizability via Order Extension Theorems

joint work with

foundational results for verifying linearizability

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. $\text{enq}(x) <_h \text{enq}(y) \land \text{deq}(y) \in h \implies \text{deq}(x) \in h \land \text{deq}(y) \leq_h \text{deq}(x)$
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 $V$

it is easy to find a large $CV$, but difficult to find a small representative

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

legal sequence

identify sequences with total orders

concurrent history
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
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!