Concurrent Data Structures

T1
Push(0)
Pop(1)

\ldots

Tn
Push(1)
Pop(0)
Empty(true)

Methods Implementation

Low Level Representation

Push
Pop
Empty
Abstract (Client) View

- Operations are considered to be **atomic**
- Thread executions are interleaved
- Executions satisfy sequential specifications

```
Push(1) Push(0) Pop(0) Pop(1) Empty(true)
```

[Diagram of operations and state transitions]
Abstract (Client) View

- Operations are considered to be atomic
- Thread executions are interleaved
- Executions satisfy sequential specifications

A “simple” implementation: **Coarse-grain Locking**

- Take a sequential implementation
- **Lock** at the beginning, **unlock** at the end of each method
Abstract (Client) View

- Operations are considered to be **atomic**
- Thread executions are interleaved
- Executions satisfy sequential specifications

```
Push(1) Push(0) Pop(0) Pop(1) Empty(true)
```

A “simple” implementation: **Coarse-grain Locking**

- Take a **sequential implementation**
- **Lock** at the beginning, **unlock** at the end of each method

- **Reference Implementation**: simple to understand
- **Low performances**
Efficient Concurrent Implementations

Allow **parallelism** between operations
Efficient Concurrent Implementations

Allow **parallelism** between operations

**Fine-grain locking** *(Lock-free algorithms)*

- Check interference and retry
- Use low-level synchronisation mechanisms (CAS)
void Push (int v) {
    node *n, *t
    node n = new node(v)
    do {
        node *t = Top
        n.next = t
    } while (not CAS(&Top, t, n))
}
void Push (int v) {
    node *n, *t
    node n = new node(v)
    do {
        node *t = Top
        n.next = t
    } while (not CAS(&Top, t, n))
}
void Push (int v) {
    node *n, *t
    node n = new node(v)
    do {
        node *t = Top
        n.next = t
    } while (not CAS(&Top, t, n))
}

Treiber’s Stack
void Push (int v) {
    node *n, *t
    node n = new node(v)
    do {
        node *t = Top
        n.next = t
    } while (not CAS(&Top, t, n))
}
Treiber’s Stack

```c
int Pop () {
    node *n, *t
    do {
        node *t = Top
        if (t==NULL) return Ø
        n = t.next
    } while (not CAS(&Top, t, n))
    int result = t.data
    free (t)
    return result
}
```
Treiber’s Stack

```c
int Pop () {
    node *n, *t
    do {
        node *t = Top
        if (t==NULL) return Ø
        n = t.next
    } while (not CAS(&Top, t, n))
    int result = t.data
    free (t)
    return result
}
```
Treiber’s Stack

```c
int Pop () {
    node *n, *t
    do {
        node *t = Top
        if (t==NULL) return Ø
        n = t.next
    } while (not CAS(&Top, t, n))
    int result = t.data
    free (t)
    return result
}
```
int Pop () {
    node *n, *t
    do {
        node *t = Top
        if (t==NULL) return Ø
        n = t.next
    } while (not CAS(&Top, t, n))
    int result = t.data
    free (t)
    return result
}
Treiber’s Stack

```c
int Pop () {
    node *n, *t
    do {
        node *t = Top
        if (t==NULL) return Ø
        n = t.next
    } while (not CAS(&Top, t, n))
    int result = t.data
    free (t)
    return result
}
```
Efficient Concurrent Implementations

Allow parallelism between operations

Fine-grain locking (Lock-free algorithms)

- Check interference and retry
- Use low-level synchronisation mechanisms (CAS)
Efficient Concurrent Implementations

Allow **parallelism** between operations

**Fine-grain locking** *(Lock-free algorithms)*
- Check interference and retry
- Use low-level synchronisation mechanisms (CAS)

**Consistency ??**
For every Client,

\[ \text{Exec (Client [ Impl ]) is included in Exec (Client [ Spec ]) } \]
Linearizability

[Herlihy, Wing, 1990]
For every execution:

- Push(0)
- Pop(1)

Find Linearisation Points:

- Push(1)
- Pop(0)
- Empty(true)

[Herlihy, Wing, 1990]
Linearizability [Herlihy, Wing, 1990]

For every execution:
- Push(0)
- Push(1)
- Pop(0)
- Pop(1)
- Empty(true)

Find Linearisation Points:
- Push(1)
- Pop(0)
- Empty(true)

Match a valid sequence:
- Push(1) → Push(0) → Pop(0) → Pop(1) → Empty(true)
Linearizability

For every execution:

Find Linearisation Points

Match a valid sequence

Push(0)  Pop(1)
Push(1)  Pop(0)
Push(1)  Push(0)  Pop(0)
Pop(1)  Empty(true)

Push(0)  Pop(0)
Pop(1)  Empty(true)

History (Return *Before Call*)

[Herlihy, Wing, 1990]
Linearizability [Herlihy, Wing, 1990]

For every execution:
- Push(0)
- Push(1)
- Pop(0)
- Pop(1)
- Empty(true)

Find Linearisation Points
- Push(1)
- Pop(0)
- Push(0)
- Pop(1)
- Empty(true)

Find a valid sequence
- Push(1)
- Push(0)
- Pop(0)
- Pop(1)
- Empty(true)

Match a valid sequence

For every history:
- Push(0)
- Push(1)
- Pop(0)
- Push(1)
- Empty(true)

Find a valid compatible total order
Histories

History of a library execution $e$:

$$H(e) = (O, \text{label}, <)$$

where

- $O = \text{Operations}(e)$
- $\text{label}: O \rightarrow M \times V \times V$
- $<$ is a partial order s.t.

$O_1 < O_2$ iff $\text{Return}(O_1)$ is before $\text{Call}(O_2)$ in $e$

```
c(push,1) r(push,tt) c(pop,-) c(pop,-) r(pop,1) c(push,2) r(push,tt) r(pop,2)
```

Graph:
- push(1) -> pop(1) -> push(2)
- push(1) -> pop(2)
Linearizability as a History Inclusion

Consider an **abstract data structure**, let **S** be its **sequential specification**, and let **Ls** be a **sequential implementation** of **S**, i.e., **Ls satisfies S**

**Lc** reference concurrent implementation = **Ls** + **lock/unlock at beginning/end of each method**
Linearizability as a History Inclusion

Consider an abstract data structure, let \( S \) be its sequential specification, and let \( L_S \) be a sequential implementation of \( S \), i.e., \( L_S \) satisfies \( S \)

\[
L_c \text{ reference concurrent implementation} = L_S + \text{lock/unlock at beginning/end of each method}
\]

Lemma:
\( H(L_S) \) is the set histories that are linearised to a sequence in \( S \)

Thm: \( L \) is linearisable wrt \( S \) iff \( H(L) \) is included in \( H(L_S) \)
History Inclusion vs OR vs Linearizability

History Inclusion vs OR

Thm: $L_1$ refines $L_2$ iff $H(L_1)$ is included in $H(L_2)$
History Inclusion vs OR vs Linearizability

History Inclusion vs OR

Thm: $L_1$ refines $L_2$ iff $H(L_1)$ is included in $H(L_2)$

- ($\Rightarrow$) Given $h$ in $H(L_1)$, construct a client $P_h$ that imposes all the happen-before constraints of $h$.
- ($\Leftarrow$) Clients cannot distinguish executions with the same history.
History Inclusion vs OR vs Linearizability

History Inclusion vs OR

**Thm:** $L_1$ refines $L_2$ iff $H(L_1)$ is included in $H(L_2)$

- $(\Rightarrow)$ Given $h$ in $H(L_1)$, construct a client $P_h$ that imposes all the happen-before constraints of $h$.
- $(\Leftarrow)$ Clients cannot distinguish executions with the same history.
  (clients and libraries do not share variables)

OR vs Linearizability

**Coro:** $L$ is linearisable w.r.t. $S$ iff $L$ refines $L_S$

since: $L$ is linearisable w.r.t. $S$ iff $H(L)$ is included in $H(L_S)$
Verifying Linearizability?
Verifying Linearizability?

Reduction to State Reachability Checking?

- Reuse existing tools for Invariance/Reachability checking
- Complexity, Decidability
General Approach:

Given a library $L$ and a specification $S$, define a monitor $M$ (including designated bad states) such that $L$ is linearisable with respect to $S$ if and only if $L \times M$ does not reach a bad state.

Verifying Linearizability?

Reduction to State Reachability Checking?

- Reuse existing tools for Invariance/Reachability checking
- Complexity, Decidability
Verifying Linearizability?

Reduction to State Reachability Checking?

- **Reuse existing tools** for Invariance/Reachability checking
- **Complexity, Decidability**

**General Approach:**

Given a **library** \( L \) and a **specification** \( S \), define a **monitor** \( M \) (+ designated **bad states**) s.t.

\[ L \text{ is linearisable wrt } S \iff L \times M \text{ does not reach a bad state} \]

**Issue:**

- **The computational power** of \( M \)?
- **Size** of \( M \)?
- Ideally, finite-state, polynomial size, but …
A Monitor for Linearizability

Given a specification: a state machine

Memory of the monitor
  • Set of all possible linearizations
  • A linearization is represented as a pair:
    • state of the specification
    • set of pending (not yet linearised) methods

Actions of the monitor
  • Observe calls and returns: call —> pending —> return
  • Guess linearisation points for pending methods in each linearisation (store expected return values by the spec.)
  • Checks that returns indeed match the specification
  • Fail if all linearizations violate the specification
Checking Linearizability: Complexity

Given a specification: a state machine

Memory

- Set of all possible linearizations
- A linearization is a pair:
  - state of the specification
  - set of pending (not yet linearised) methods

Fixed number of threads

- => EXPSPACE algorithm (see also [Alur, McMillan, Peled 96])
Checking Linearizability: Complexity

Given a specification: a state machine

Memory

• Set of all possible linearizations
• A linearization is a pair:
  • state of the specification
  • set of pending (not yet linearised) methods

Fixed number of threads

• => EXPSPACE algorithm (see also [Alur, McMillan, Peled 96])
• EXPSPACE-hard problem [Hamza 2015]
• Contrasts with State Reachability: PSPACE-complete
Checking Linearizability: Complexity

Given a specification: a state machine

Memory
- Set of all possible linearizations
- A linearization is a pair:
  - state of the specification
  - set of pending (not yet linearised) methods

Unbounded number of threads
- => Unbounded memory
Checking Linearizability: Complexity

Given a specification: a state machine

Memory

- Set of all possible linearizations
- A linearization is a pair:
  - state of the specification
  - set of pending (not yet linearised) methods

Unbounded number of threads

- => Unbounded memory
- Undecidable problem [B., Emmi, Enea, Hamza 2013]
- Contrasts with State Reachability: EXPSPACE-complete
Checking Linearizability: Undecidability

- Reduction of the reachability problem in 2-counter machines
- Given a Machine $M$, build a library $L_M$ and a specification $S_M$
  There is a computation of $M$ reaching a state $q_f$
  iff
  $L_M$ is not linearisable w.r.t. $S_M$
Checking Linearizability: Undecidability

• Reduction of the reachability problem in 2-counter machines
• Given a Machine $M$, build a library $L_M$ and a specification $S_M$
  There is a computation of $M$ reaching a state $q_f$
  iff
  $$L_M \text{ is not linearisable w.r.t. } S_M$$

• $M$ has methods $\text{Inc}(i)$, $\text{Dec}(i)$, $\text{Zero}(i)$, and $m(q)$
• Encoding of a counter: a multi-set of parallel Inc’s and Dec’s
• $S_M$ corresponds to “non acceptable” computations
  • $\text{Zero}(i)$ occurs when $\#\text{Inc}(i) \neq \#\text{Dec}(i)$
  • State $q_f$ is not reached (do not contain $m(q_f)$
Checking Linearizability: Undecidability

- Reduction of the reachability problem in 2-counter machines.
- Given a Machine $M$, build a library $L_M$ and a specification $S_M$.
  There is a computation of $M$ reaching a state $q_f$ iff
  $$L_M \text{ is not linearisable w.r.t. } S_M$$

- $M$ has methods $\text{Inc}(i)$, $\text{Dec}(i)$, $\text{Zero}(i)$, and $m(q)$.
- Encoding of a counter: a multi-set of parallel $\text{Inc}$’s and $\text{Dec}$’s.
- $S_M$ corresponds to “non acceptable” computations:
  - $\text{Zero}(i)$ occurs when $\#\text{Inc}(i) \neq \#\text{Dec}(i)$
  - State $q_f$ is not reached (do not contain $m(q_f)$).

- $S_M$ can be a regular language (particular order $\text{Inc}$’s and $\text{Dec}$’s).
- Checking linearizability $\Rightarrow$ consider all orders.
Checking Linearizability: Common Approaches

Enumerate executions and linearizations (bug finding)

e.g. Line-up [Burckhardt et al. 2010]
Checking Linearizability: Common Approaches

Enumerate executions and linearizations (bug finding)

  e.g. *Line-up* [Burckhardt et al. 2010]

**Fixed linearization points** in the code (verif. correctness)

  e.g., [Vafeiadis, CAV’10], [B., Emmi, Enea, Hamza 2013],
  [Abdulla et al., TACAS 2013]
void Push (int v) {
    node *n, *t
    node n = new node(v)
    do {
        node *t = Top
        n.next = t
    } while (not CAS(&Top, t, n))
}

int Pop () {
    node *n, *t
    do {
        node *t = Top
        if (t==NULL) return Ø
        n = t.next
    } while (not CAS(&Top, t, n))
    int result = t.data
    free (t)
    return result
}
Checking Linearizability: Fixed Linearisation Points
[B., Emmi, Enea, Hamza 2013]

• No need to guess linearisation points
• => Monitor keeps track of only one linearisation
• Linearisation = state of the spec. + set of pending op.
• Linearisation point => Move the state of the spec. + record the expected return value
• Return => check the value is conform to the spec.
Checking Linearizability: Fixed Linearisation Points

[B., Emmi, Enea, Hamza 2013]

- No need to guess linearisation points
- => Monitor keeps track of only one linearisation
- Linearisation = state of the spec. + set of pending op.
- Linearisation point => Move the state of the spec. + record the expected return value
- Return => check the value is conform to the spec.

- Fixed number of FS threads, FS spec. : PSPACE-complete
- Unbounded number of FS threads, FS spec. :
  - Count the number of pending methods of each type
  - => State reachability in VASS (Petri Nets): EXPSPACE-complete
Checking Linearizability: Common Approaches

Enumerate executions and linearizations (bug finding)

  e.g. *Line-up* [Burckhardt et al. 2010]

**Fixed linearization points** in the code (verif. correctness)

  e.g., [Vafeiadis, CAV’10], [B., Emmi, Enea, Hamza 2013],
  [Abdulla et al., TACAS 2013]
Checking Linearizability: Common Approaches

Enumerate executions and linearizations (bug finding)

  e.g. *Line-up* [Burckhardt et al. 2010]

**Scalability issues!**

Fixed linearization points in the code (verif. correctness)

  e.g., [Vafeiadis, CAV’10], [B., Emmi, Enea, Hamza 2013],
  [Abdulla et al., TACAS 2013]
Checking Linearizability: Common Approaches

Enumerate executions and linearizations (bug finding)
  e.g. Line-up [Burckhardt et al. 2010]

**Scalability issues!**

Fixed linearization points in the code (verif. correctness)
  e.g., [Vafeiadis, CAV’10], [B., Emmi, Enea, Hamza 2013],
  [Abdulla et al., TACAS 2013]

Fixing linearisation points in not always possible!
  e.g.,
  Helping mechanisms based stacks/queues
  Time-stamping based stack [Dodds, Haas, Kirsch, 2015]
Implementing a Set

- Operations: Add, Remove, Contains
- Representation: Sorted linked list

```java
boolean Contains (int x)
    current := Head ;
    while current.val < x
        current := current.next ;
    return current.val = x
```
Implementing a Set

- Operations: Add, Remove, Contains
- Representation: Sorted linked list

```java
boolean Contains (int x) {
    current := Head;
    while current.val < x {
        current := current.next;
    }
    return current.val = x;
}
```

Example:
Contains(5)
Implementing a Set

- Operations: Add, Remove, Contains
- Representation: Sorted linked list

```java
boolean Contains (int: x)
  current := Head ;
  while current.val < x
    current := current.next ;
  return current.val = x
```

Diagram:
- Head
- 2
- 7
- current

Contains(5)

```java
current := current.next
```
Implementing a Set

- Operations: Add, Remove, Contains
- Representation: Sorted linked list

```java
boolean Contains (int: x) {
    current := Head;
    while current.val < x {
        current := current.next;
    }
    return current.val = x
}
```

Contains(5)

```
current := current.next
```

return(false)
Implementing a Set

• Operations: Add, Remove, Contains
• Representation: Sorted linked list

boolean Contains (int: x)
current := Head ;
while current.val < x
    current := current.next ;
return current.val = x

returns (false)
Implementing a Set

- Operations: Add, Remove, Contains
- Representation: Sorted linked list

```java
boolean Contains (int: x)
    current := Head ;
    while current.val < x
        current := current.next ;
    return current.val = x
```

```
Add(5)
Contains(5)
```

```
“commit”
return(false)
```

current := current.next
Implementing a Set

- Operations: Add, Remove, Contains
- Representation: Sorted linked list

```java
boolean Contains (int x)
    current := Head ;
    while current.val < x
        current := current.next ;
    return current.val = x
```

Add(5)  "commit"

Contains(5)  return(false)

Linearize before Commit of Add(5)
Implementing a Set

- Operations: Add, Remove, Contains
- Representation: Sorted linked list

boolean Contains (int: x)
current := Head;
while current.val < x
  current := current.next;
return current.val = x

Contains(7)
Implementing a Set

- Operations: Add, Remove, Contains
- Representation: Sorted linked list

```java
boolean Contains (int: x)
    current := Head;
    while current.val < x
        current := current.next;
    return current.val = x
```

Contains(7)

current := current.next
Implementing a Set

- Operations: Add, Remove, Contains
- Representation: Sorted linked list

```java
boolean Contains (int: x)
    current := Head ;
    while current.val < x
        current := current.next ;
    return current.val = x
```

Contains(7)

```
current := current.next
return(true)
```
Implementing a Set

- Operations: Add, Remove, Contains
- Representation: Sorted linked list

```java
boolean Contains (int: x)
    current := Head ;
    while current.val < x
        current := current.next ;
    return current.val = x
```

Contains(7)
```
current := current.next
```

return(true)
Implementing a Set

- Operations: Add, Remove, Contains
- Representation: Sorted linked list

boolean Contains (int: x)
current := Head ;
while current.val < x
current := current.next ;
return current.val = x
Implementing a Set

- Operations: Add, Remove, Contains
- Representation: Sorted linked list

```java
def contains(int x):
    current = Head
    while current.val < x:
        current = current.next
    return current.val == x
```

Remove(7)
```
current = current.next
```

Contains(7)
```
current = current.next
```

```
return true
```

Linearize before Commit of Remove(7)
Fixed Linearisation Points + Read-Only Methods

• No need to guess linearisation points
• => Monitor keeps track of only **one** linearisation
• Linearisation = state of the spec. + set of pending op.
• Linearisation point => Move the state of the spec. + record the expected return value
• + Linearize all read-only methods returning false Before
• + Linearize all read-only methods returning true After
• Return => check the value is conform to the spec.
Summary

• Correctness of a concurrent library = Linearizability
Summary

- Correctness of a concurrent library = Linearizability
- Checking linearizability is a **complex** problem

<table>
<thead>
<tr>
<th></th>
<th>Linearizability</th>
<th>Fixed Lin. Points Linearizability + Read-Only</th>
<th>State Reachability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed Nb Threads</td>
<td>EXPSPACE-C (1)</td>
<td>PSPACE-C (2)</td>
<td>PSPACE-C</td>
</tr>
<tr>
<td>Unbounded Nb of Threads</td>
<td>Undecidable (2)</td>
<td>EXPSPACE-C (2)</td>
<td>EXPSPACE-C</td>
</tr>
</tbody>
</table>

(2) B., Emmi, Enea, Hamza, 2013
## Summary

- Correctness of a concurrent library = Linearizability
- Checking linearizability is a complex problem

<table>
<thead>
<tr>
<th></th>
<th>Linearizability</th>
<th>Fixed Lin. Points</th>
<th>State Reachability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed Nb Threads</td>
<td>EXPSPACE-C (1)</td>
<td>PSPACE-C (2)</td>
<td>PSPACE-C</td>
</tr>
<tr>
<td>Unbounded Nb of Threads</td>
<td>Undecidable (2)</td>
<td>EXPSPACE-C (2)</td>
<td>EXPSPACE-C</td>
</tr>
</tbody>
</table>

(2) B., Emmi, Enea, Hamza, 2013

- **Tractable** reductions to state reachability?
- Avoid reasoning about linearisation points?
Tractable Linearizability Checking?
Tractable Linearizability Checking?

Special classes of implementations
- Special policies of linearization
- => Stronger correctness criteria than linearizability
- => Sound verification approach for linearizability
Tractable Linearizability Checking?

Special classes of implementations
- Special policies of linearization
- => Stronger correctness criteria than linearizability
- => Sound verification approach for linearizability

Special classes of specifications (abstract structures)
Common structures: stacks, queues, registers, …
Tractable Linearizability Checking?

**Special classes of implementations**
- Special policies of linearization
- => Stronger correctness criteria than linearizability
- => Sound verification approach for linearizability

**Special classes of specifications (abstract structures)**

Common structures: stacks, queues, registers, ...

**Special classes of behaviours**
- Suitable bounding concepts
- Parametrised under-approximation schemas (bugs detection)
- Good coverage, scalability?
Tractable Linearizability Checking?

Special classes of implementations
• Special policies of linearization
• => Stronger correctness criteria than linearizability
• => Sound verification approach for linearizability

Special classes of specifications (abstract structures)
Common structures: stacks, queues, registers, …

Special classes of behaviours
• Suitable bounding concepts
• Parametrised under-approximation schemas (bugs detection)
• Good coverage, scalability?
Focusing on a Class of Concurrent Objects

[B, Emmi, Enea, Hamza, ICALP’15]

• Consider a class of specifications including: stack, queue, register, mutex.

• Characterizing the set of concurrent violations: A finite number of “bad patterns” (ordered sets of operations that should not be embedded in any correct execution)

• Defining finite-state automata recognising the set executions that include one of the “bad patterns” (using a data independence assumption)

• Linear reduction of linearizability checking to state reachability problem (using these automata as monitors.)

• Decidability for an unbounded number of FS threads.
Specifying queues and stacks

Queue

• \( u . v : Q \ & \ u : ENQ^* \rightarrow \text{Enq}(x) . u . \text{Deq}(x) . v : Q \)

• \( u . v : Q \ & \ \text{no unmatched Enq in } u \rightarrow u . \text{Emp} . v : Q \)

Stack

• \( u . v : S \ & \ \text{no unmatched Push in } u \rightarrow \text{Push}(x) . u . \text{Pop}(x) . v : S \)

• \( u . v : S \ & \ \text{no unmatched Push in } u \rightarrow u . \text{Emp} . v : S \)
Order Violation

FIFO violations:

\[ \text{ret(Deq(1)) < call(Enq(1))} \]
Order Violation

FIFO violations:

```
ret(Deq(1)) < call(Enq(1))
```

```
ret(Enq(1)) < call(Enq(2))  \&  ret(Deq(2)) < call(Deq(1))
```
Order Violation

FIFO violations:

\[ \text{ret(Deq}(1)) < \text{call(Enq}(1)) \]

\[ \text{ret(Enq}(1)) < \text{call(Enq}(2)) \quad \& \quad \text{ret(Deq}(2)) < \text{call(Deq}(1)) \]

- Regular Language over Call and Return events
- Only 3 different data values are needed
Empty Violation

Push$_1$

EMP

Pop$_1$
Empty Violation

\[\text{Push}_1\]

\[\text{Emp}\]

\[\text{Pop}_1\]

\[\text{Push}_1\]

\[\text{Push}_1\]

\[\text{Push}_1\]

\[\text{Push}_1\]

\[\text{Push}_1\]

\[\text{Pop}_1\]

\[\text{Pop}_1\]

\[\text{Pop}_1\]

\[\text{Pop}_1\]
Order Violation cont. (stack)

\[
\begin{align*}
PUSH_2 & \quad \text{Push}_1 \quad \text{Push}_1 \\
& \quad \text{Call} \quad \text{Push}(1) \\
& \quad \text{Call} \quad \text{Push}(2) \\
& \quad \text{Ret} \quad \text{Push}(2) \\
& \quad \text{Ret} \quad \text{Push}(1) \\
Pop_2 & \quad \text{Pop}_1 \quad \text{Pop}_1 \\
& \quad \text{Call} \quad \text{Pop}(1) \\
& \quad \text{Call} \quad \text{Pop}(2) \\
& \quad \text{Ret} \quad \text{Pop}(2) \\
& \quad \text{Ret} \quad \text{Push}(1)
\end{align*}
\]
Automaton for Empty Violation

Recognized by:
Automaton for Push-Pop Order Violation

Recognized by:

\[ q_i \xrightarrow{\Sigma_3} q_0 \xrightarrow{\text{call } Push(1)} q_1 \xrightarrow{\text{call } Pop(2)} q_2 \xrightarrow{\text{call } Pop(1)} q_3 \xrightarrow{\text{ret } Push(1)} q_4 \]
Linearizability to State Reachability

Thm:

For each $S$ in \{Stack, Queue, Mutex, Register\}, there is an automaton $A(S)$ s.t.
for every data independent concurrent implementation $L$,

$L$ is linearisable wrt $S$ iff $L$ intersected with $A(S)$ is empty

Same complexity as state reachability
Under-approximate Analysis

[B, Emmi, Enea, Hamza, POPL’15]

• Bounded information about computations
• Useful for efficient bug detection

• Bounding concept for detecting linearizability violations?
• Should offer good coverage, and scalability

• Interval-length bounded analysis
• Based on characterising linearizability as history inclusion
• Monitor uses counters
• Allows for symbolic encodings
• Efficient static and dynamic analysis
Linearizability as a History Inclusion (Recall)

Consider an **abstract data structure**, let **S** be its **sequential specification**, and let **Ls** be a **sequential implementation** of **S**, i.e., **Ls satisfies S**

**Lc** reference concurrent implementation = **Ls** + lock/unlock at beginning/end of each method

**Lemma:**

**H(Lc)** is the set histories that are linearised to a sequence in **S**

**Thm:** **L** is linearisable wrt **S** iff **H(L)** is included in **H(Lc)**
Abstracting Histories

Weakening relation

\[ h_1 \leq h_2 \quad (h_1 \text{ is weaker than } h_2) \]

iff

\[ h_1 \text{ has less constraints than } h_2 \]
Abstracting Histories

Weakening relation

\[ h_1 \leq h_2 \quad (h_1 \text{ is weaker than } h_2) \]
iff
\[ h_1 \text{ has less constraints than } h_2 \]

Lemma:

\[ (h_1 \leq h_2 \text{ and } h_2 \text{ is in } H(L)) \implies h_1 \text{ is in } H(L) \]
Approximation Schema

Weakening function $A_k$, for any given $k \geq 0$, s.t.

- $A_k(h) \leq h$
- $A_0(h) \leq A_1(h) \leq A_2(h) \leq \ldots \leq h$
- There is a $k$ s.t. $h = A_k(h)$
Approximation Schema

Weakening function $A_k$, for any given $k \geq 0$, s.t.

- $A_k(h) \leq h$
- $A_0(h) \leq A_1(h) \leq A_2(h) \leq \ldots \leq h$
- There is a $k$ s.t. $h = A_k(h)$

Approximate History Inclusion Checking, for fixed $k \geq 0$

- Given a library $L$ and a specification $S$
- Check: Is there an $h$ in $H(L)$ s.t. $A_k(h)$ is not in $H(S)$?
- $A_k(h)$ is not in $H(S)$ => $h$ is not in $H(S)$ — Violation!
Histories are Interval Orders

Interval Orders = partial order \((O, <)\) such that

\((o_1 < o_1' \text{ and } o_2 < o_2')\) implies \((o_1 < o_2' \text{ or } o_2 < o_1')\)

Prop: For every execution \(e\), \(H(e)\) is an interval order
Notion of Length

Let \( h = (O, <) \) be an Interval Order (history in our case)

- Past of an operation: \( \text{past}(o) = \{o' : o' < o\} \)
- Lemma [Rabinovitch’78]:
  The set \( \{\text{past}(o) : o \in O\} \) is linearly ordered
- The \textit{length} of the order = number of pasts - 1
Canonical Representation of Interval Orders

- Mapping \( I : O \rightarrow [n]^2 \) where \( n = \text{length}(h) \) [Greenough ’76]
- \( I(o) = [i, j] \), with \( i, j \leq n \), such that
  \[
  i = |\{\text{past}(o') : o' < o\}| \quad \text{and} \quad j = |\{\text{past}(o') : \text{not } (o < o')\}| - 1
  \]
Let $A_k$ maps each $h$ to some $h' \leq h$ of length $k$

$\Rightarrow$ Keep precise the information about the $k$ last intervals
Counting Representation of Interval Orders

Count the number of occurrences of each operation type in each interval

- \( h = (O, <) \) an IO with canonical representation \( I: O \rightarrow [k]^2 \)
- Associate a **counter** with each operation type and interval
- \( \prod(h) \) is the Parikh image of \( h \)
- It represents the multi-set \( \{ [\text{label}(o), I(o)] : o \text{ in } O \} \)

**Prop:** \( H_k(e) \) is in \( H_k(L) \) iff \( \prod(H_k(e)) \) is in \( \prod(H_k(L)) \)
Reduction to Reachability with Counters

$H_k(L)$ subset of $H_k(S)$

iff

$\prod(H_k(L))$ subset of $\prod(H_k(S))$

• Consider **k-bounded-length abstract histories**

• Track histories of $L$ using a **finite number of counters**

• Use an **arithmetic-based representation of** $\prod(H_k(S))$

• $\prod(H_k(S))$ can be either computed, or given manually

• Check that $\prod(H_k(S))$ is an invariant
Experimental Results: Coverage

Comparison of violations covered with $k \leq 4$

- Data point: Counts in logarithmic scale over all executions (up to 5 preemptions) on Scal’s nonblocking bounded-reordering queue with $\leq 4$ enqueue and $\leq 4$ dequeue
- x-axis: increasing number of executions (1023-2359292)
- White: total number of unique histories over a given set of executions
- Black: violations detected by traditional linearizability checker (e.g., Line-up)
Experimental Results: Runtime Monitoring

Comparison of runtime overhead between Linearization-based monitoring and Operation counting

- Data point: runtime on logarithmic scale, normalised on unmonitored execution time
- Scal’s nonblocking Michael-Scott queue, 10 enqueue and 10 dequeue operations.
- x-axis is ordered by increasing number of operations
### Experimental Results: Static Analysis

<table>
<thead>
<tr>
<th>Library</th>
<th>Bug</th>
<th>P</th>
<th>k</th>
<th>m</th>
<th>n</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Michael-Scott Queue</td>
<td>B1 (head)</td>
<td>2x2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>24.76s</td>
</tr>
<tr>
<td>Michael-Scott Queue</td>
<td>B1 (tail)</td>
<td>3x1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>45.44s</td>
</tr>
<tr>
<td>Treiber Stack</td>
<td>B2</td>
<td>3x4</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>52.59s</td>
</tr>
<tr>
<td>Treiber Stack</td>
<td>B3 (push)</td>
<td>2x2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>24.46s</td>
</tr>
<tr>
<td>Treiber Stack</td>
<td>B3 (pop)</td>
<td>2x2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>15.16s</td>
</tr>
<tr>
<td>Elimination Stack</td>
<td>B4</td>
<td>4x1</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>317.79s</td>
</tr>
<tr>
<td>Elimination Stack</td>
<td>B5</td>
<td>3x1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>222.04s</td>
</tr>
<tr>
<td>Elimination Stack</td>
<td>B2</td>
<td>3x4</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>434.84s</td>
</tr>
<tr>
<td>Lock-coupling Set</td>
<td>B6</td>
<td>1x2</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>11.27s</td>
</tr>
<tr>
<td>LFDS Queue</td>
<td>B7</td>
<td>2x2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>77.00s</td>
</tr>
</tbody>
</table>

- Static detection of injected refinement violations with CSeq & CBMC.
- Program Pij with i and j invocations to the push and pop methods, explore n-round round-robin schedules with m loop iterations unrolled, with monitor for Ak.
- Bugs: (B1) non-atomic lock, (B2) ABA bug, (B3) non-atomic CAS operation, (B4) misplaced brace, (B5) forgotten assignment, (B6) misplaced
Conclusion

- **Linearizability** checking is **hard/undecidable** in general.
- But **tractable reductions to state reachability** are **possible**.
- Consider **relevant** classes of **concurrent objects**:
  - Covers **common structures** such as **stacks and queues**
  - **Finite-state monitor**: **Linear reduction to state reachability**
  - Decidability for **unbounded number of threads**

- Consider **relevant types executions**:
  - Bounding principle based on an abstraction of histories
  - **Monitor**: **Counter machine**
  - Use **symbolic techniques** => Static and dynamic analysis
  - **Good coverage, scalable monitoring**
Some future work

- Extend the first approach to other structures, e.g., sets.
- Specification language+systematic construction of monitors.
- Combine our approach with providing linearisation policies
  [Abdulla et al., TACAS’13, SAS’16]
Some future work

• Extend the first approach to other structures, e.g., sets.
• Specification language+systematic construction of monitors.
• Combine our approach with providing linearisation policies
  [Abdulla et al., TACAS’13, SAS’16]
• Extend it to distributed (replicated) data structures
  Weaker consistency notions are needed:
  Eventual consistency, causal consistency, etc.
  • Eventual consistency —> Reachability, Model-checking
    [B., Enea, Hamza, POPL’14]
  • Causal consistency ?
    [Recent work for the Read-Write memory/Key-value store]