Modular Verification of Linearizability with Non-Fixed Linearization Points

Hongjin Liang and Xinyu Feng
Univ. of Science and Technology of China

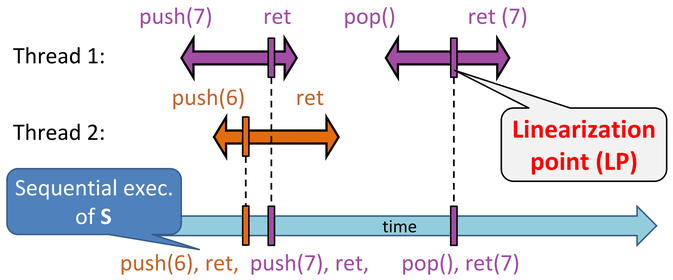
http://kyhcs.ustcsz.edu.cn/relconcur/lin
PLDI 2013, Jun 19, 3:15-4:30pm, Session A

How to specify and prove the correctness of concurrent objects (libraries)?

Linearizability

- Standard correctness criterion for concurrent objects **O**
- **O** $\leq_{lin}$ **S**: All concurrent executions of **O** are "equivalent" to some sequential executions of abstract object **S**

Concurrent exec. of **O**: push(7), push(6), ret, ret, pop(), ret(7)



Challenges in Verification

1. Non-Fixed LPs

- Helping mechanism
  - LP is in other threads' code
  - Lose thread-modularity ?
- Future-dependent (FD) LPs
  - LP is at prior access, but only if later validation succeeds
  - Refer to unpredictable future behaviors ?

2. No program logic with soundness w.r.t. linearizability

Our Contributions

- A program logic for linearizability
  - Support non-fixed LPs
- A light instrumentation mechanism to help verification
  - Try-commit clause as an alternative to prophecy variables
- Logic ensures contextual refinement → linearizability
  - A new forward-backward simulation as meta-theory
- Verified 12 well-known algorithms
  - Some are used in `java.util.concurrent` (JUC)

Our Approach to Verifying O $\leq_{lin}$ S

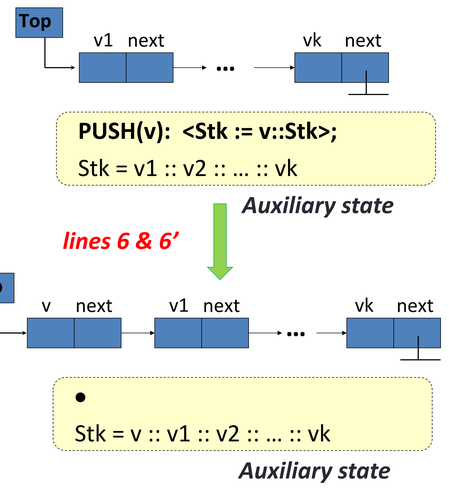
- **C** : instrument **O** with auxiliary commands (ACs) at **LP**s
  - ACs manipulate auxiliary states (**S** and abstract states)
  - Execute **S** simultaneously with **O**'s LP step
- Reason about **C** using our program logic
  - Extend an existing logic (e.g. Rely-Guarantee) with inference rules for ACs
  - Ensure **O**'s LP is the single step with the same effect as **S**

Linself for Fixed LPs

Treiber Stack:

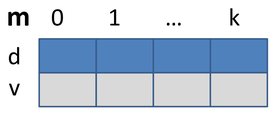
push(v) $\leq_{lin}$ PUSH(v) ?

```
push(int v):
- { list(Top, Stk) ∗ (cid → PUSH(v)) }
1  local b:=false, x, t;
2  x := new Node(v);
3  while (!b) {
4    t := Top;
5    x.next := t;
6    < b := cas(&Top, t, x);
6'     if (b) linself; >
7  }
- { list(Top, Stk) ∗ (cid → •) }
```



Try-Commit for Future-Dependent LPs

Pair Snapshot




```
write(int i, d):
1  <m[i].d := d; m[i].v++;
1'   linself; >
```

```
readPair(int i, j):
2  local s:=false, a, b, v, w;
3  while (!s) {
4    <a := m[i].d;  v := m[i].v;>
5    <b := m[j].d;  w := m[j].v; trylinself; >
6    <if (v = m[i].v)  { s:= true; commit( cid → • ); } >
7  }
8  return (a, b);
```

Line 5 is LP only if line 6 succeeds

- **Speculate (trylin)** at potential LP, keep both result and original abstract code & abstract states ➔ **speculation set**
- **Commit** to correct branch at later validation and discard others

Pending Thread Pool for Helping

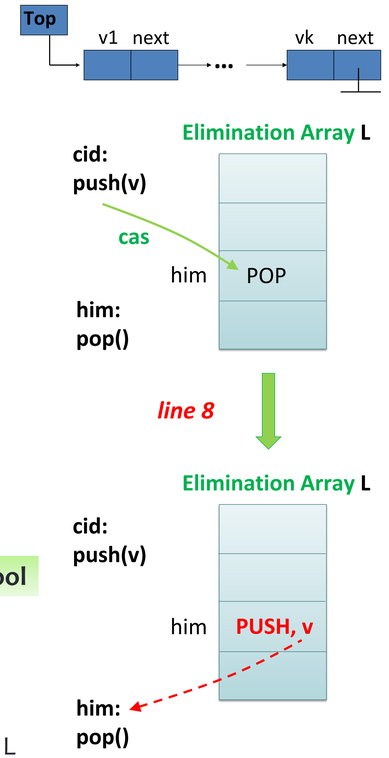
HSY Elimination-Backoff Stack

(A push and a pop cancel out each other)

```
push(int v):
1  local p, q, him, b;
2  p := new ThrdDesc(PUSH, v);
3  while (!b) {
4    ...
5    him := rand();   q := L[him];
6    if (q != null && q.op = POP) {
7      ...
8      < b := cas(&L[him], q, p);
8'       if (b) {lin(cid); lin(him);} >
9    }
10   ...
11 }
```

Auxiliary State: Pending Thread Pool

- U = {him → POP, …}
- lin(t) executes & updates U(t)
  - U' = {him → •, …}
- Abstraction of elimination array L
- Still thread-modular!



Objects	Helping	FD LPs	Java Pkg (JUC)	Herlihy-Shavit Book
Treiber stack				✓
HSY stack	✓			✓
MS two-lock queue				✓
MS lock-free queue		✓	✓	✓
DGLM queue		✓		
Lock-coupling list				✓
Optimistic list				✓
Heller et al lazy list	✓	✓		✓
HM lock-free list	✓	✓	✓	✓
Pair snapshot		✓		
CCAS	✓	✓		
RDCSS	✓	✓		

Verified Algorithms Using Our Logic