Saturn: a library of verified concurrent data structures for OCaml 5

Clément Allain (INRIA Paris)
Vesa Karvonen (Tarides)
Carine Morel (Tarides)

September 6, 2024

Plan

- ❒ Why Saturn?
- ❒ What is in Saturn?
- ❒ Optimizations and benchmarks
- ❒ Testing
- ❒ Verification

Why Saturn?

Github: ocaml-multicore/saturn

- A collection of concurrent-safe data structures for OCaml 5:
  - ◇ well-tested
  - ◇ benchmarked
  - ◇ optimized
  - ◇ verified

Why Saturn?

Github: ocaml-multicore/saturn

- A collection of concurrent-safe data structures for OCaml 5:
  - ◇ well-tested
  - ◇ benchmarked
  - ◇ optimized
  - ◇ verified
- Writting concurrent code is hard

Buggy queue with size

```
module Queue = Saturn.Queue
type 'a t = {size : int Atomic.t; queue : 'a Queue.t}

let create () =
    {size = Atomic.make 0; queue = Queue.create ()}

let push t msg =
  Atomic.incr t.size;
  Queue.push t.queue msg

let pop_opt t =
  match Queue.pop_opt t.queue with
  | Some elt ->  Atomic.decr t.size; Some elt
  | None -> Atomic.set t.size 0; None

let size t = Atomic.get t.size
```

Buggy queue with size

```
let test () =
 let queue = create () in
 let d1 = Domain.spawn (fun () -> push queue 1) in
 let d2 = Domain.spawn (fun () -> pop_opt queue |>
     ignore) in
 Domain.join d1;
 Domain.join d2;
 pop_opt queue |> ignore;
 size queue
```

Buggy queue with size

```
let test () =
 let queue = create () in
 let d1 = Domain.spawn (fun () -> push queue 1) in
 let d2 = Domain.spawn (fun () -> pop_opt queue |>
     ignore) in
 Domain.join d1;
 Domain.join d2;
 pop_opt queue |> ignore;
 size queue
```

In 10 to 20% of the tries, the test returns a size of -1.

Why Saturn?

Github: ocaml-multicore/saturn

- A collection of concurrent-safe data structures for OCaml 5:
  - ◇ well-tested
  - ◇ benchmarked
  - ◇ optimized
  - ◇ verified
- Writting concurrent code is hard
  - ◇ bugs that can be hard to reproduce and understand

Concurrent stack: Treiber stack

```
type 'a t = 'a list Atomic.t

let create () = Atomic.make []

let rec push q a =
  let old = Atomic.get q in
  if Atomic.compare_and_set q old (a :: old) then ()
  else push q a

let rec pop_opt q =
  let old = Atomic.get q in
  match old with
  | [] -> None
  | x :: xs ->
    if Atomic.compare_and_set q old xs then Some x
    else pop_opt q
```

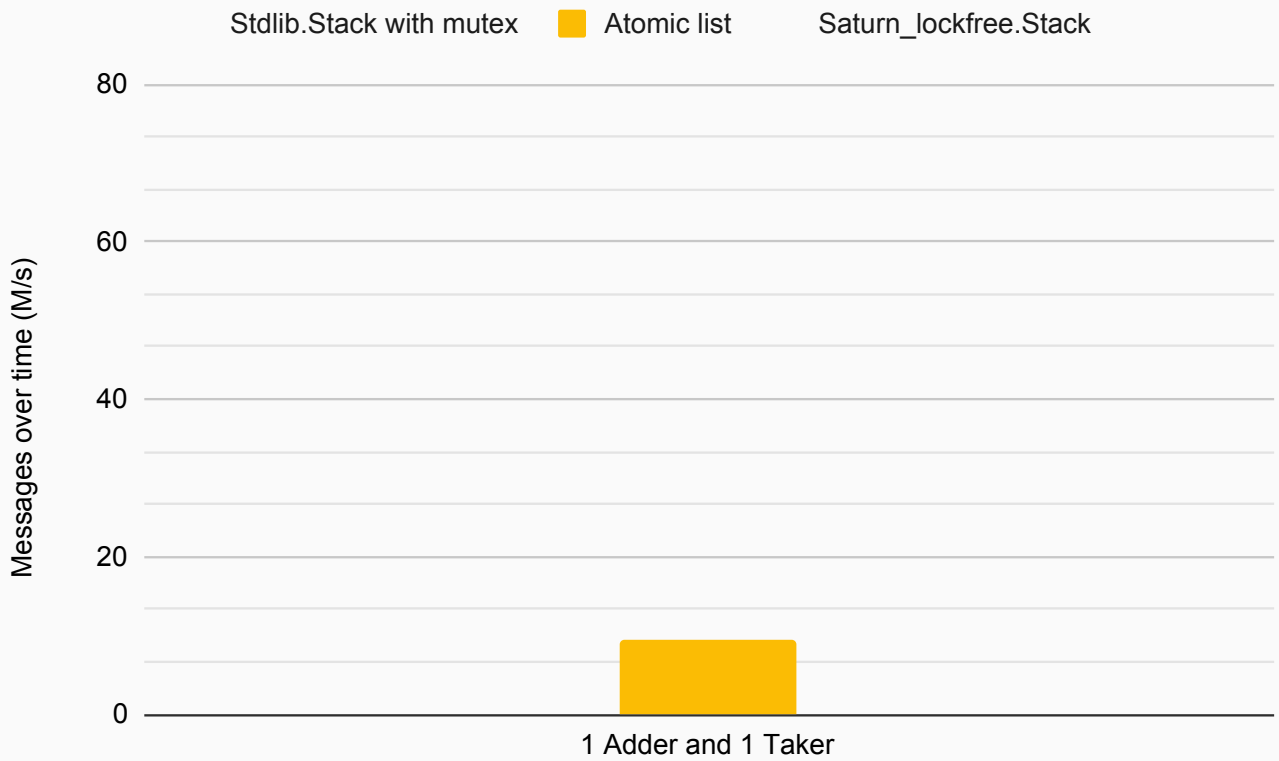
Concurrent stack: Treiber stack

```
type 'a t = 'a list Atomic.t

let create

let rec push
  let old =
  if Atomic
  else push

let rec pop_
  let old =
  match old
  | [] -> N
  | x :: xs ->
    if Atomic.compare_and_set q old xs then Some x
    else pop_opt q
```



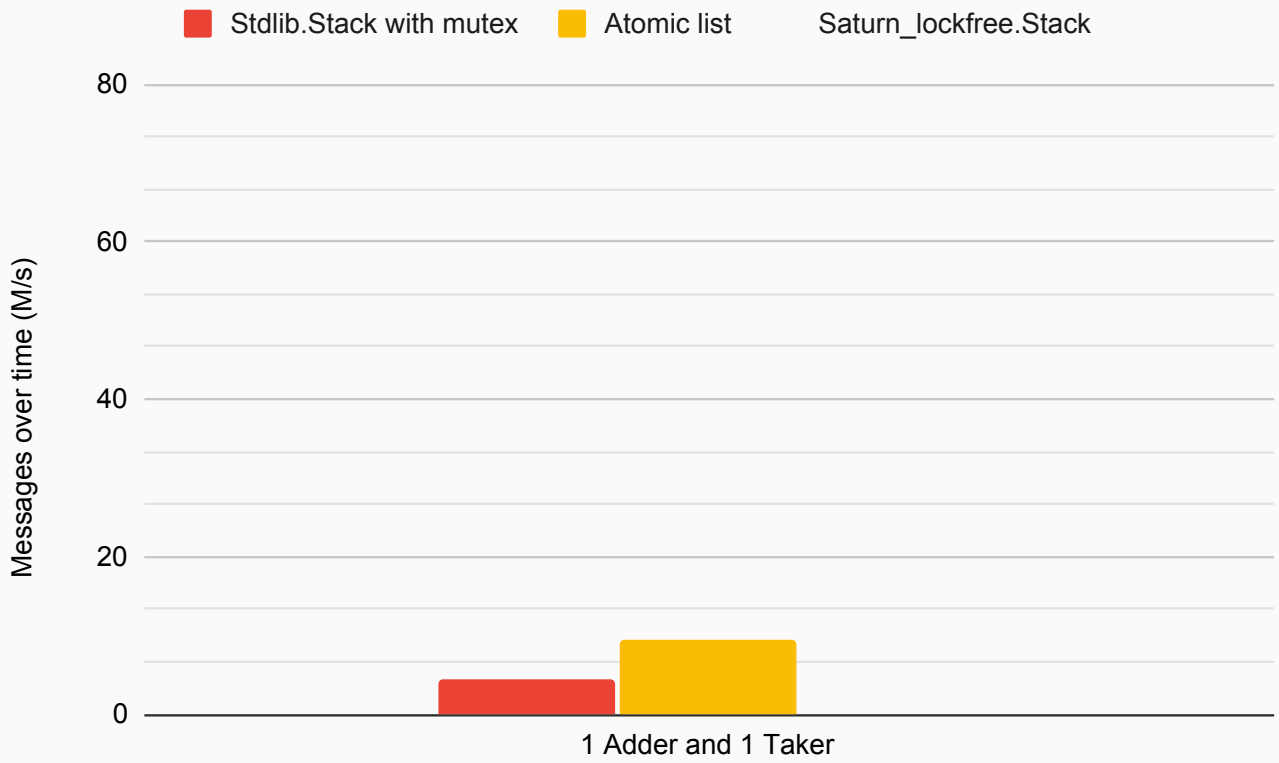
Concurrent stack: Treiber stack

```
type 'a t = 'a list Atomic.t

let create (

let rec push
  let old =
  if Atomic.
  else push

let rec pop_
  let old =
  match old
  | [] -> No
  | x :: xs ->
    if Atomic.compare_and_set q old xs then Some x
    else pop_opt q
```



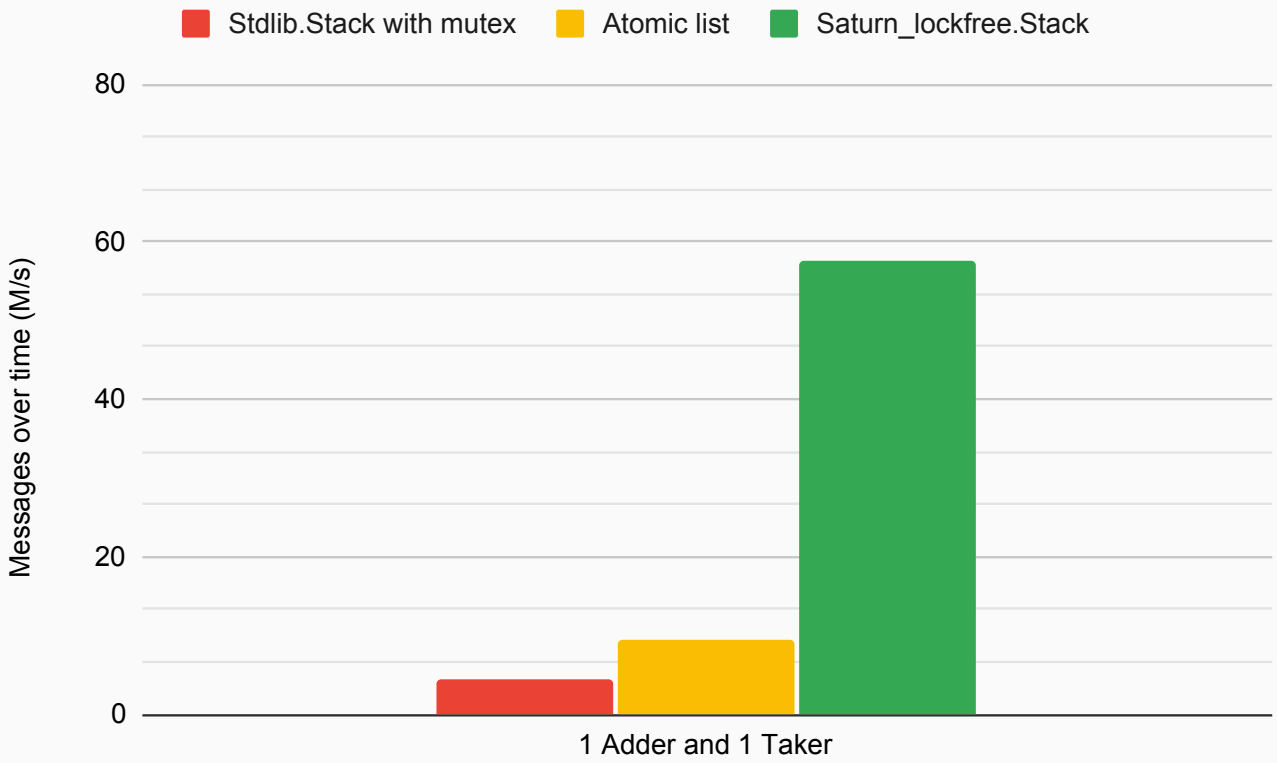
Concurrent stack: Treiber stack

```
type 'a t = 'a list Atomic.t

let create ()

let rec push
  let old = A
  if Atomic.c
  else push c

let rec pop_c
  let old = A
  match old w
  | [] -> Nor
  | x :: xs ->
    if Atomic.compare_and_set q old xs then Some x
    else pop_opt q
```



Why Saturn?

- OCaml 5 : multicore programming
- Writting concurrent code is hard
  - ◇ bugs that can be hard to reproduce and understand
  - ◇ progress properties: deadlock, starvation etc..
- Writting efficient concurrent code is even harder
  - ◇ different behaviors on different CPU architectures
  - ◇ space to benchmarks is way larger
  - ◇ ...

What is in Saturn?

- A collection of concurrent-safe data structures for OCaml 5

What is in Saturn?

- A collection of concurrent-safe data structures for OCaml 5
  - ◇ Queues
    - ○ multi-producer, multi-consumer
    - ○ single-producer, single-consumer
    - ○ single-producer, multi-consumer
    - ○ bounded, blocking (opened PRs)

What is in Saturn?

- A collection of concurrent-safe data structures for OCaml 5
  - ◇ Queues
    - ○ multi-producer, multi-consumer
    - ○ single-producer, single-consumer
    - ○ single-producer, multi-consumer
    - ○ bounded, blocking (opened PRs)
  - ◇ Work-stealing deque
  - ◇ Stacks
  - ◇ Hashtable (opened PR)
  - ◇ Skiplist: a sorted linked list with $o(\log(n))$ operations
  - ◇ Bag

What is in Saturn?

- A collection of concurrent-safe data structures for OCaml 5
  - ◇ Queues
    - ○ multi-producer, multi-consumer
    - ○ single-producer, single-consumer
    - ○ single-producer, multi-consumer
    - ○ bounded, blocking (opened PRs)
  - ◇ Work-stealing deque
  - ◇ Stacks
  - ◇ Hashtable (opened PR)
  - ◇ Skiplist: a sorted linked list with $o(\log(n))$ operations
  - ◇ Bag
- Most available data structures are lock-free

What is in Saturn?

- A collection of concurrent-safe data structures for OCaml 5
  - ◇ Queues
    - ○ multi-producer, multi-consumer
    - ○ single-producer, single-consumer
    - ○ single-producer, multi-consumer
    - ○ bounded, blocking (opened PRs)
  - ◇ Work-stealing deque
  - ◇ Stacks
  - ◇ Hashtable (opened PR)
  - ◇ Skiplist: a sorted linked list with $o(\log(n))$ operations
  - ◇ Bag
- Most available data structures are lock-free
  - → Two libraries: Saturn and Saturn_lockfree

Testing in Saturn

What should (and can) be tested ?

- Correctness
- Linearizability
- Progress (i.e. lock freedom)

How ?

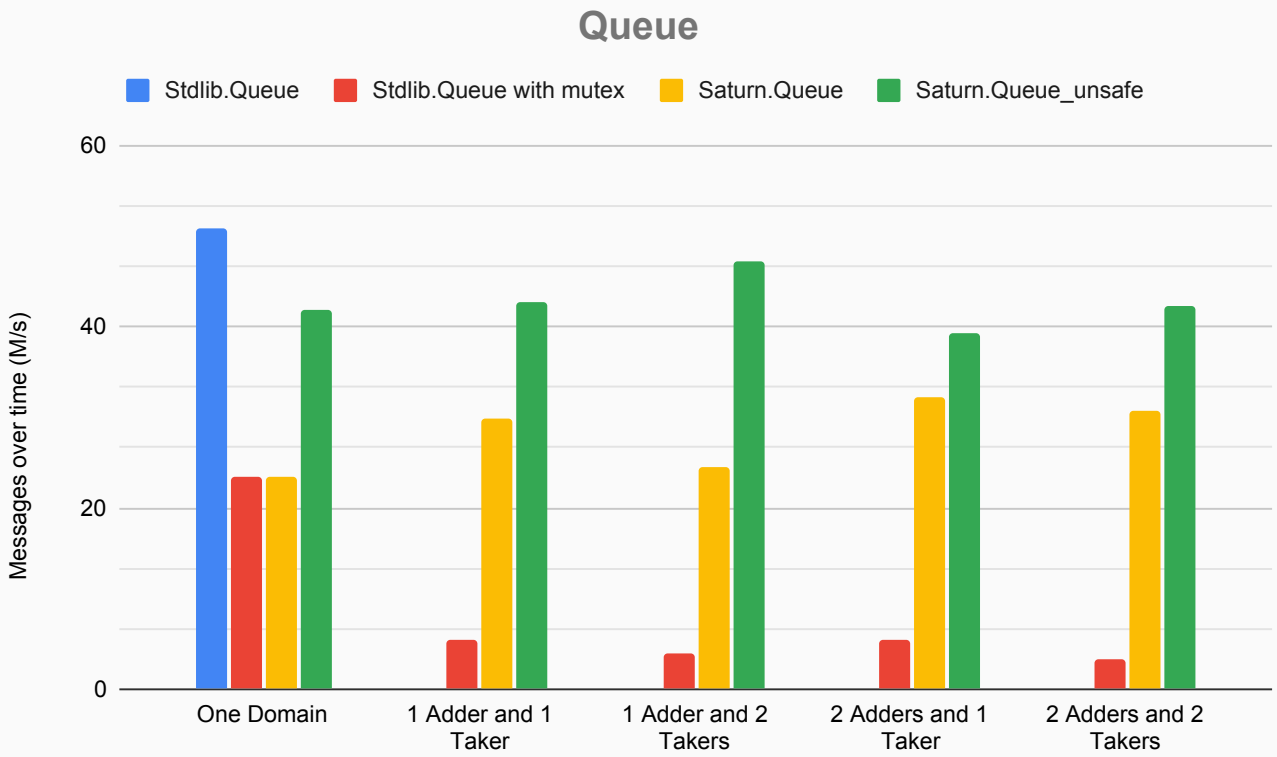
- Lin / STM (ocaml-multicore/multicoretest)
  - Correctness
  - Linearizability
- Dscheck (ocaml-multicore/dscheck)
  - Correctness
  - Lock-freedom

Optimizations

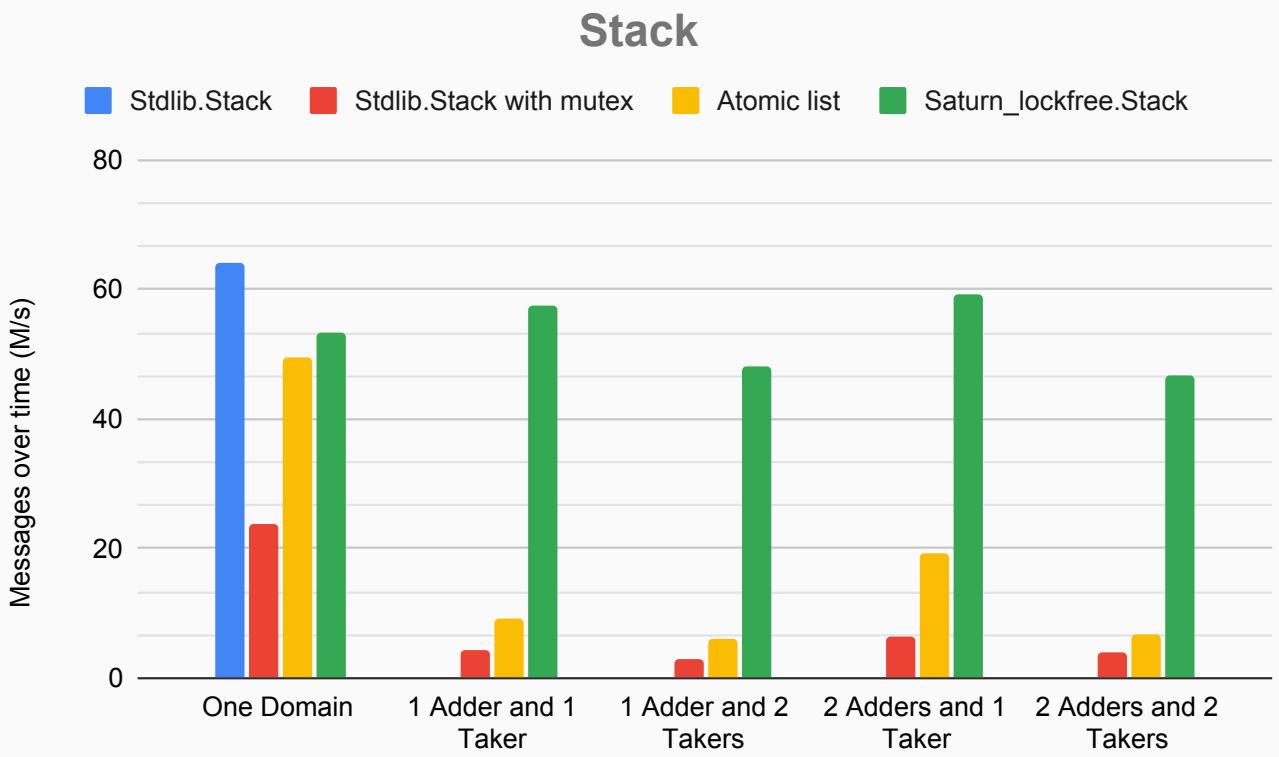
Micro-optimizations

- On algorithms, to prevent false sharing, indirections etc..
- Experimental: some use **Obj.magic** (e.g. Saturn.Queue_unsafe)

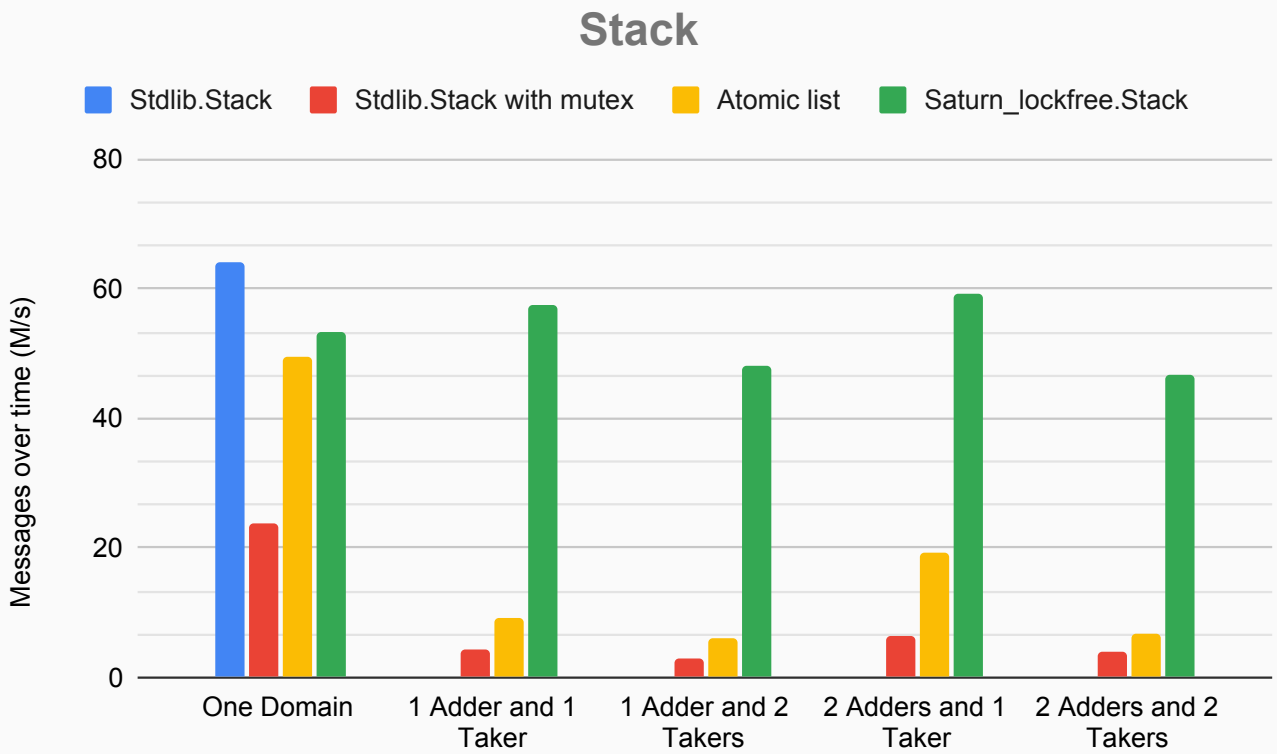
Is it any good ?



Is it any good ?



Is it any good ?



https://github.com/lyrm/saturn-benchmarks/

Formal verification

Testing a concurrent algorithm is hard due to the number of potential interleavings.

Formal verification is here to help us!

From OCaml to Coq

OCaml

Coq

Proving linearizability

$$\text{linearizability} \quad \underset{\text{in SC}}{\simeq} \quad \frac{\frac{\frac{\frac{\{\,\text{stack-inv}\ t\,\}}{\langle\, vs.\,\text{stack-model}\ t\ vs\,\rangle}}{\texttt{stack\_push}\ t\ v}}{\langle\,\text{stack-model}\ t\ (v :: vs)\,\rangle}}{\{\ ()\,.\,\text{True}\,\}}$$

Theorems for free from separation logic specifications

Birkedal, Dinsdale-Young, Guéneau, Jaber, Svendsen & Tzevelekos

Relaxed memory model (future work)

$$\frac{\{\,\text{stack-inv } t\,\}}{\dfrac{\langle\, v_0, \ldots, v_n, \mathcal{V}_0, \ldots, \mathcal{V}_n.\, \text{stack-model } t\ ((v_0, \mathcal{V}_0), \ldots (v_n, \mathcal{V}_n))\,\rangle}{\dfrac{\texttt{stack\_push}\ t\ v, \mathcal{V}}{\dfrac{\langle\, \text{stack-model } t\ ((v, \mathcal{V}), (v_0, \mathcal{V}_0), \ldots (v_n, \mathcal{V}_n))\,\rangle}{\{\ ()\ .\ \text{True}\,\}}}}}$$

Cosmo: a concurrent separation logic for multicore OCaml

Mével, Jourdan & Pottier

Writing concurrent protocols in Iris

Thank you for your attention!