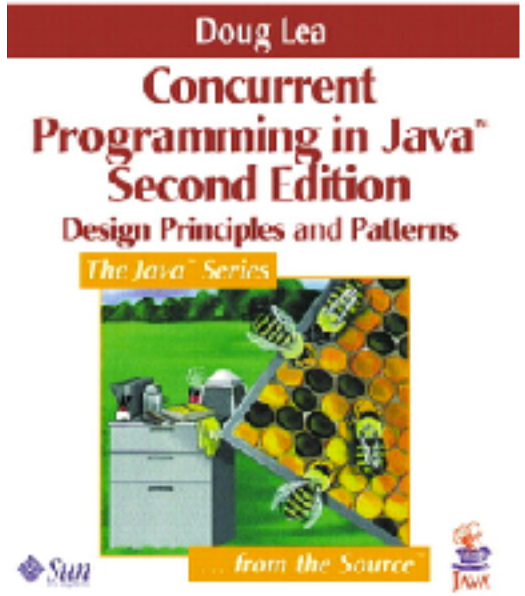
Composable lock-free programming for Multicore OCaml

KC Sivaramakrishnan

University of Cambridge

OCaml Labs



Microsoft .NET

JVM: `java.util.concurrent` **.Net:** `System.Concurrent.Collections`

Synchronization

Reentrant locks
Semaphores
R/W locks
Reentrant R/W locks
Condition vari
Coun

Data st

ay & list)
ronous
Priority, nonblocking
Priority, blocking
Deques
Sets
Maps (hash & skiplist)

Not Composable

stack.cmi

```
val push : ...
val pop  : ...
```

killer_app.ml

```
let v = pop(s1) in
push(s2,v)
```

```
val push     : ...
val pop      : ...
val pop_push : ...
```

```
(* atomically *)
let v = pop(s1) in
push(s2,v)
```

How to build *composable* & *scalable* lock-free libraries?

PLDI 2012

Reagents: Expressing and Composing Fine-grained Concurrency

Aaron Turon

Northeastern University

turon@ccs.neu.edu

Abstract

Efficient communication and synchronization is crucial for fine-grained parallelism. Libraries providing such features, while indispensable, are difficult to write, and often cannot be tailored or composed to meet the needs of specific users. We introduce *reagents*, a set of combinators for concisely expressing concurrency algorithms. Reagents scale as well as their hand-coded counterparts, while providing the composability existing libraries lack.

Categories and Subject Descriptors D.1.3 [*Programming techniques*]: Concurrent programming; D.3.3 [*Language constructs and features*]: Concurrent programming structures

General Terms Design, Algorithms, Languages, Performance

Such libraries are an enormous undertaking—and one that must be repeated for new platforms. They tend to be conservative, implementing only those data structures and primitives likely to fulfill common needs, and it is generally not possible to safely combine the facilities of the library. For example, JUC provides queues, sets and maps, but not stacks or bags. Its queues come in both blocking and nonblocking forms, while its sets and maps are nonblocking only. Although the queues provide atomic (thread-safe) dequeuing and sets provide atomic insertion, it is not possible to combine these into a single atomic operation that moves an element from a queue into a set.

In short, libraries for fine-grained concurrency are indispensable, but hard to write, hard to extend by composition, and hard to

Sequential >>> — Software transactional memory

Parallel <*> — Join Calculus

Selective <+> — Concurrent ML

still lock-free!

wait-free

Under contention, **each** thread makes progress

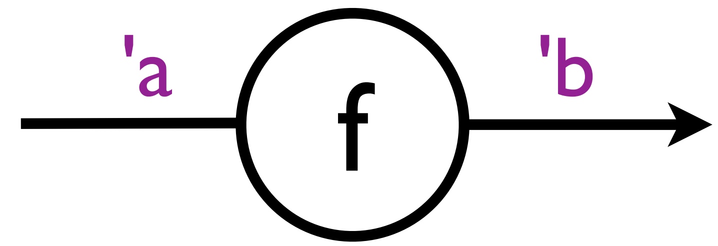
lock-free

Under contention, **at least 1** thread makes progress

obstruction-free

Single thread **in isolation** makes progress

Lambda abstraction:

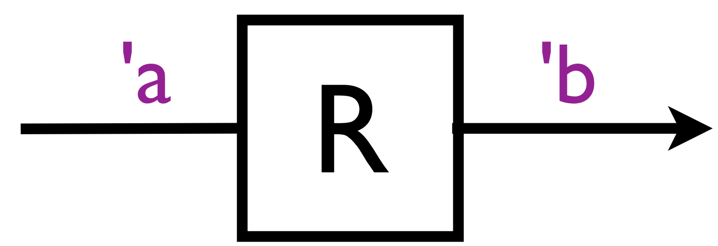


Value: `'a -> 'b`

Composition: `('a -> 'b) -> ('b -> 'c) -> 'a -> 'c`

Application: `('a -> 'b) -> 'a -> 'b`

Reagent abstraction:



Value: `('a,'b) t`

Composition: `val (>>>) : ('a,'b) t -> ('b,'c) t -> ('a,'c) t`

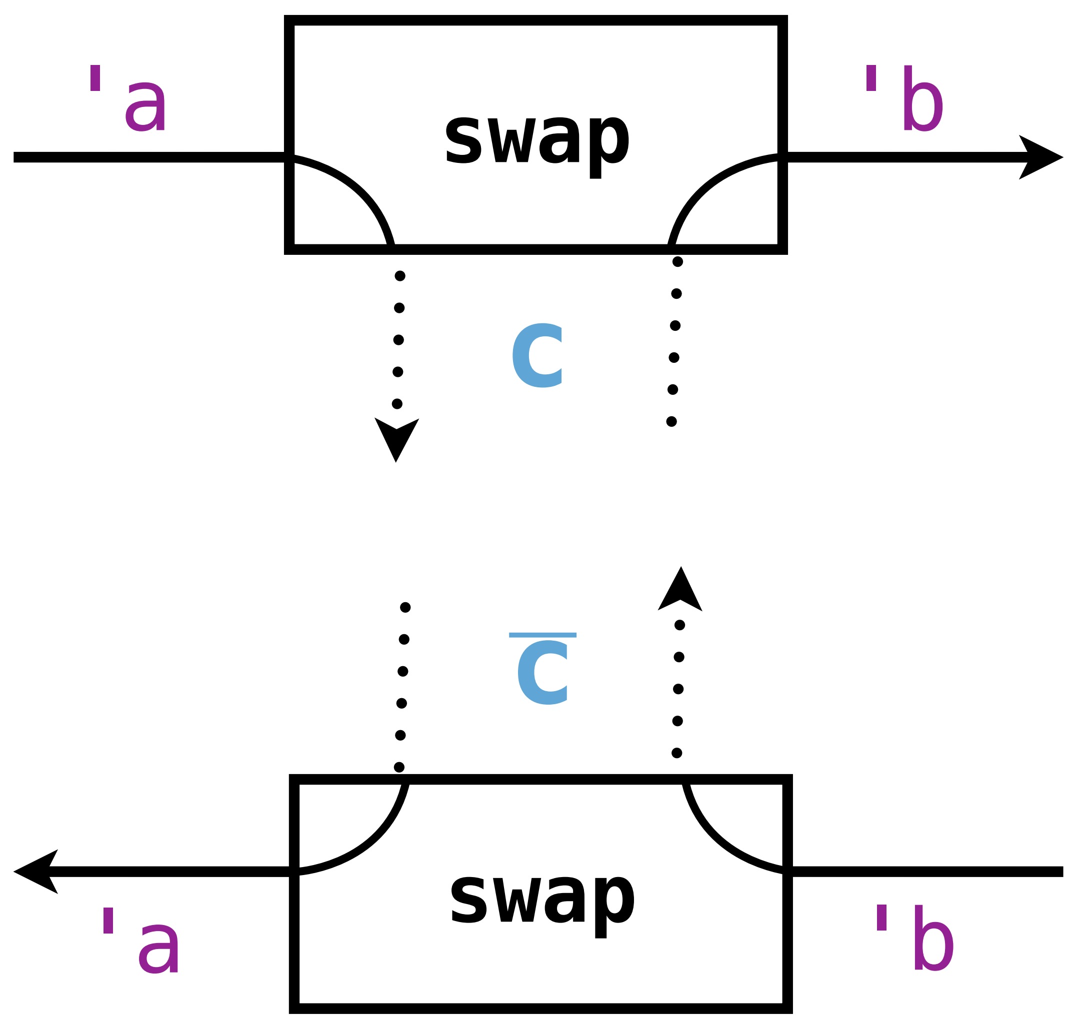
Application: `val run : ('a,'b) t -> 'a -> 'b`

Thread Interaction

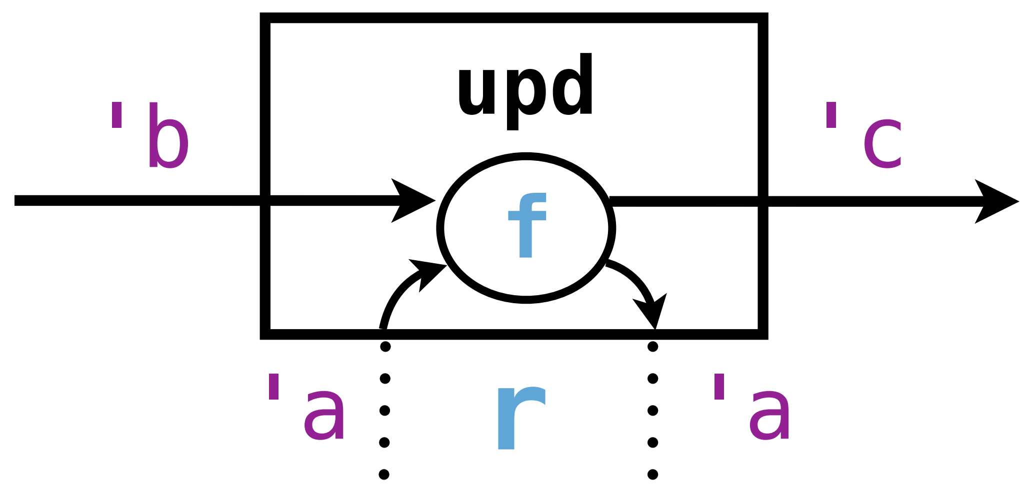
```
module type Reagents = sig
  type ('a,'b) t

  (* shared memory *)
  module Ref : Ref.S
    with type ('a,'b) reagent = ('a,'b) t

  (* communication channels *)
  module Channel : Channel.S
    with type ('a,'b) reagent = ('a,'b) t
  ...
end
```

```
module type Channel = sig
  type ('a,'b) endpoint
  type ('a,'b) reagent

  val mk_chan : unit -> ('a,'b) endpoint * ('b,'a) endpoint
  val swap    : ('a,'b) endpoint -> ('a,'b) reagent
end
```

c: ('a,'b) endpoint



```
module type Ref = sig
  type 'a ref
  val ref : 'a -> 'a ref
  val upd : 'a ref -> f:(‘a -> 'b -> ('a * ‘c) option)
              -> ('b, 'c) Reagent.t
end
```



- Hides the complexity:
  - Compare-and-swap (and associated backoff mechanisms)
  - Wait and notify mechanism

```ocaml
module Treiber_stack = struct
  type 'a t = 'a list Ref.ref

  let create () = Ref.ref []

 (* val push : 'a t -> ('a, unit) Reagent.t *)
  let push s = Ref.upd s (fun xs x -> Some (x::xs,()))

 (* val pop : 'a t -> (unit, 'a) Reagent.t *)
  let pop s = Ref.upd s (fun l () ->
    match l with
    | [] -> None (* block *)
    | x::xs -> Some (xs,x))
end
```

- Not much complex than a sequential stack implementation
- No mention of CAS, back off, retry, etc.
- No mention of threads, wait, notify, etc.

Combinators

```
(* Sequential composition *)
val (>>>) : ('a,'b) t -> ('b,'c) t -> ('a,'c) t

(* Disjunction (left-biased) *)
val (<+>) : ('a,'b) t -> ('a,'b) t -> ('a,'b) t

(* Conjunction *)
val (<*>) : ('a,'b) t -> ('a,'c) t -> ('a, 'b * 'c) t
```

Composability

Transfer elements atomically

```
Treiber_stack.pop s1 >>> Treiber_stack.push s2
```

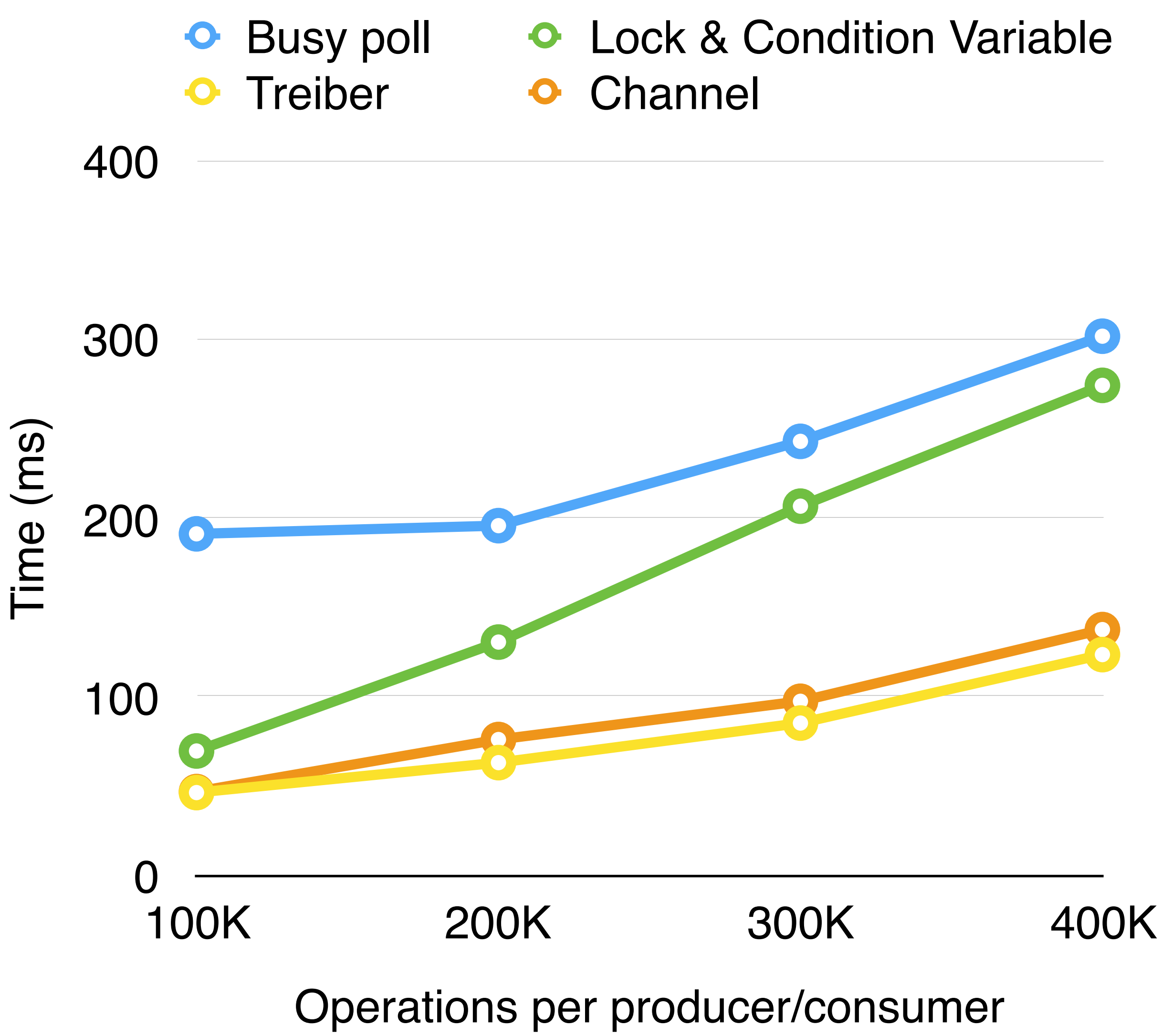
Consume elements atomically

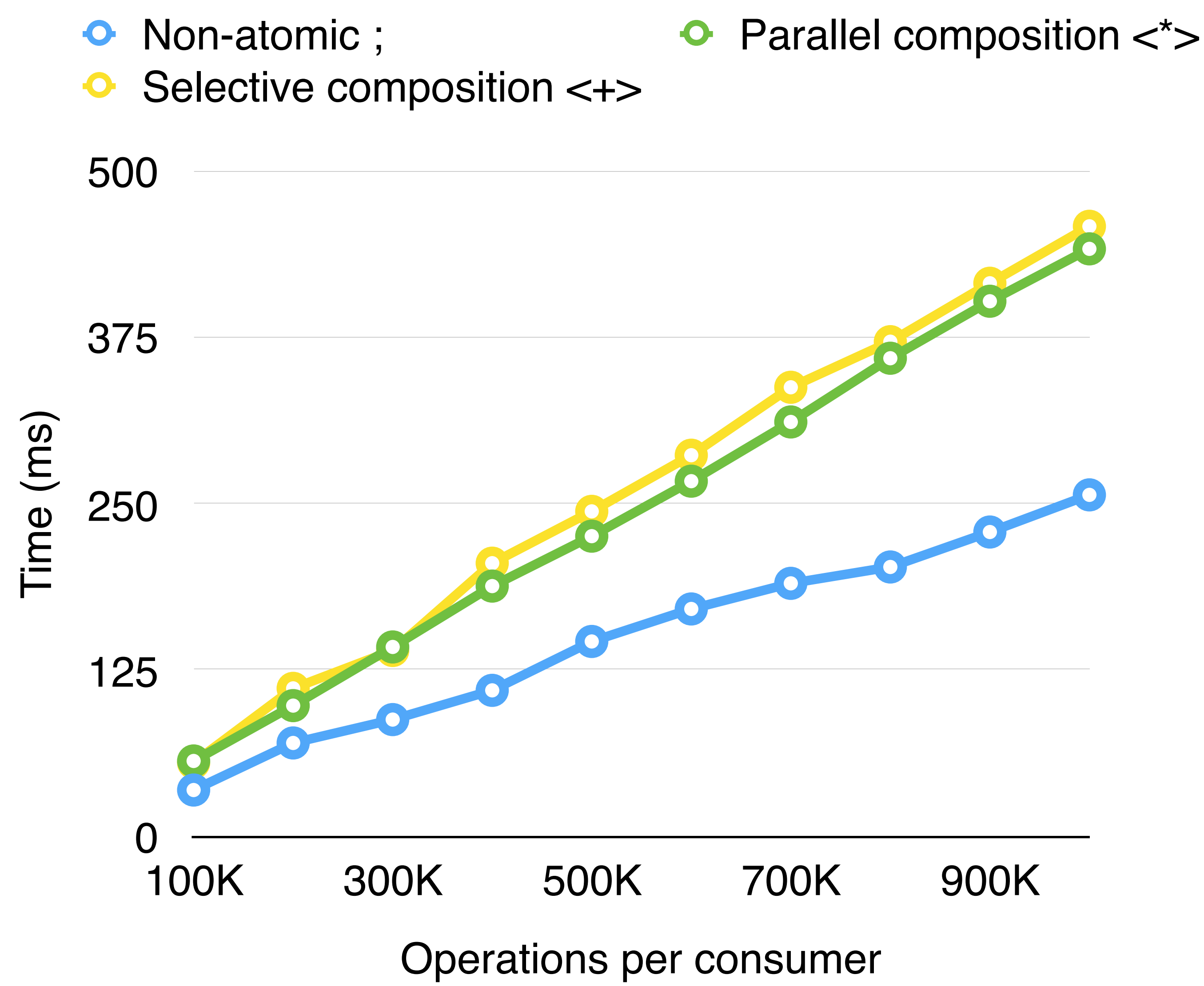
```
Treiber_stack.pop s1 <*> Treiber_stack.pop s2
```

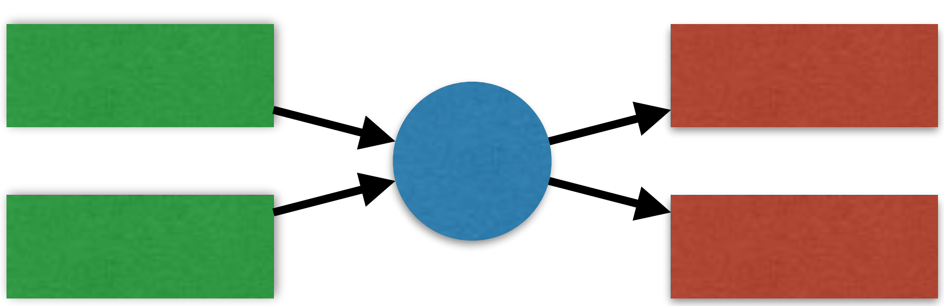
Consume elements from either

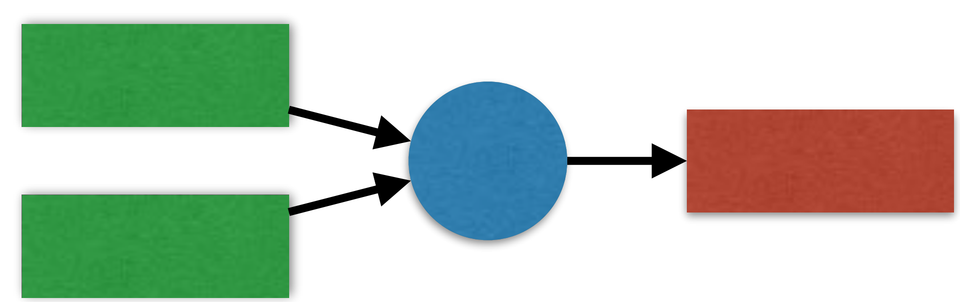
```
Treiber_stack.pop s1 <+> Treiber_stack.pop s2
```

Performance

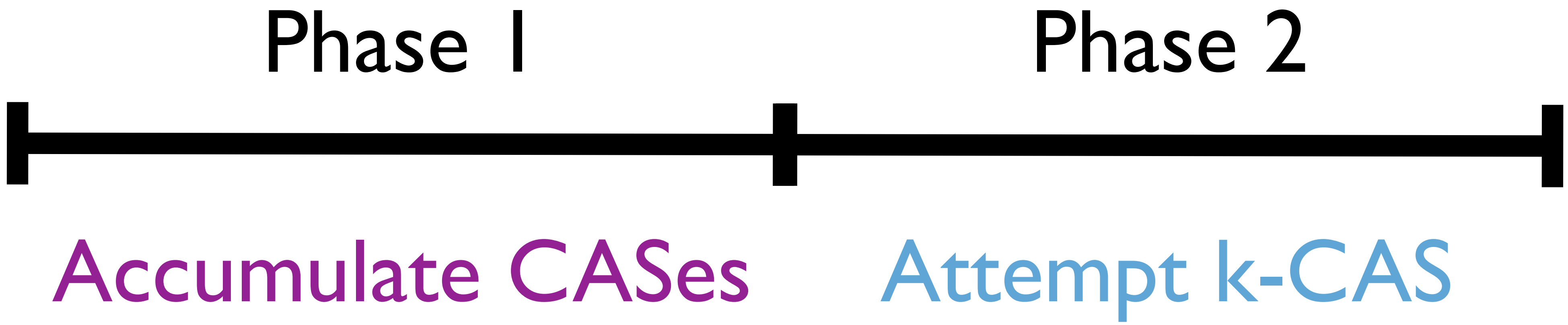




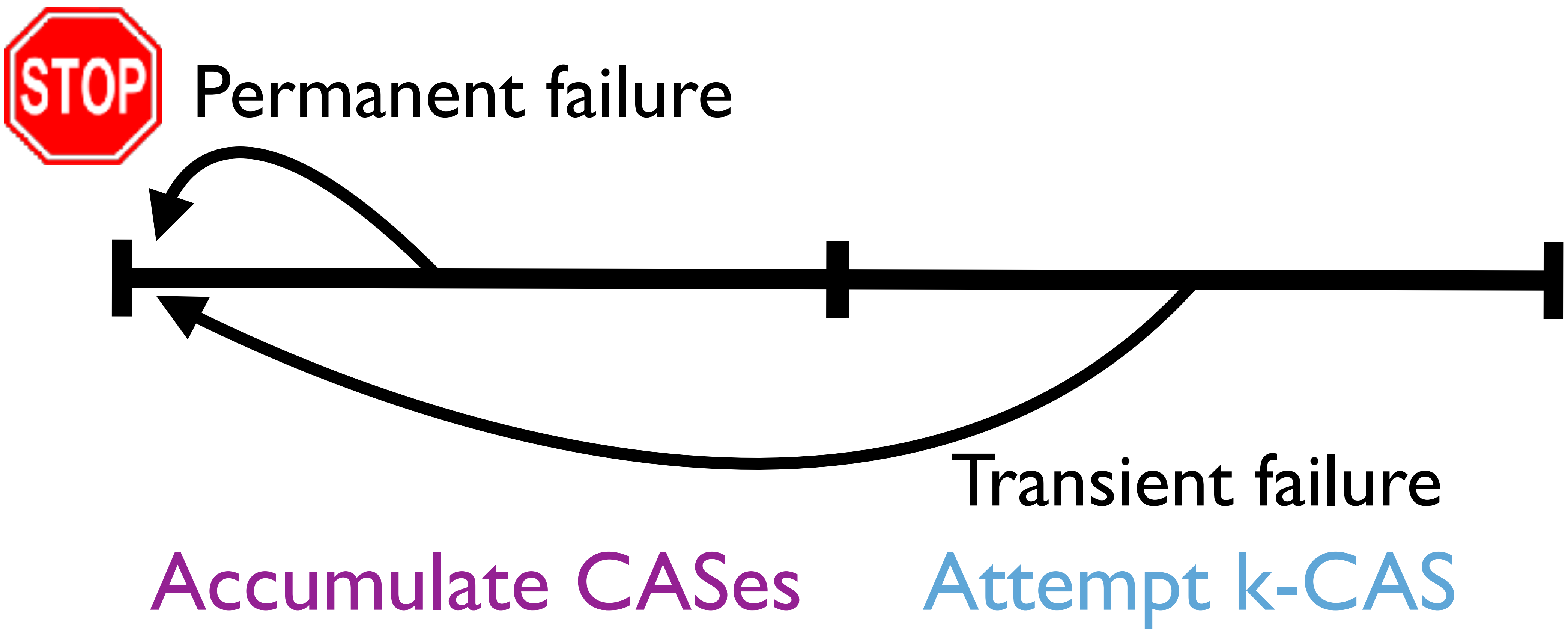




Implementation



Implementation



- WIP: HTM to perform k-CAS
  - HTM backend ~40% faster on low contention micro benchmarks
  - HTM (with STM fallback) does no worse than STM under medium to high contention

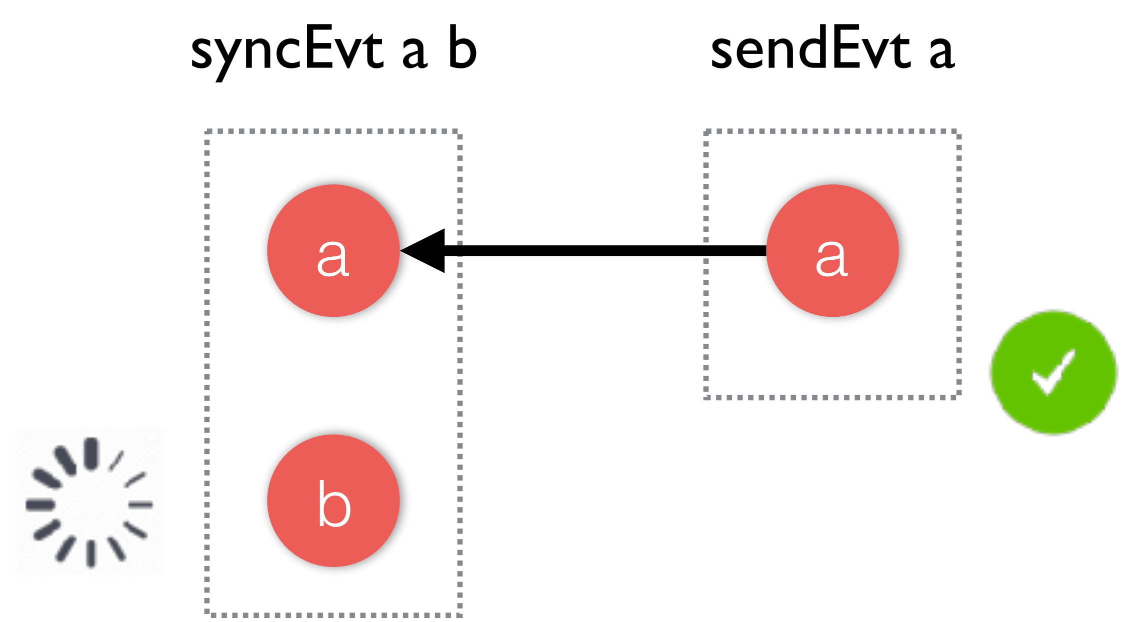
Comparison to STM

- STM is both more and less expressive
    - Reagents = STM + Synchronous communication
    - No RMW guarantee in Reagents
- Reagents geared towards performance
    - Reagents are lock-free. Most STM implementations are not.
    - Reagents map nicely to hardware transactions

Comparison to CML

- Reagents more expressive than CML — atomicity

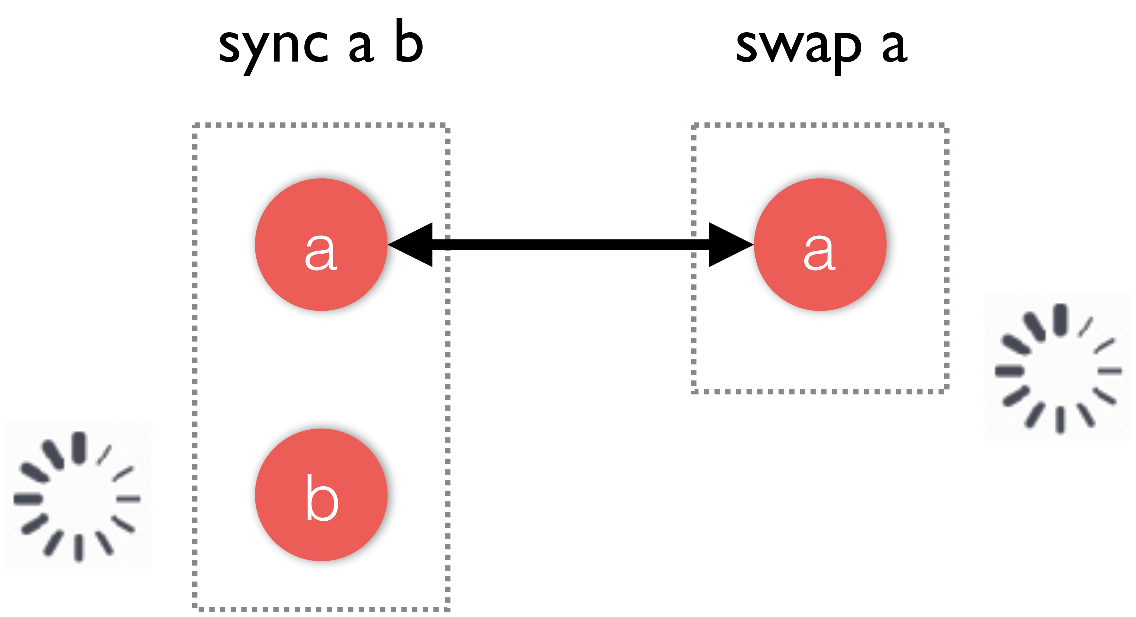
```
let syncEvt a b =
  choose [ wrap (recvEvt a, fun () -> sync (recvEvt b)),
           wrap (recvEvt b, fun () -> sync (recvEvt a)) ]
```



Comparison to CML

- Reagents more expressive than CML — atomicity

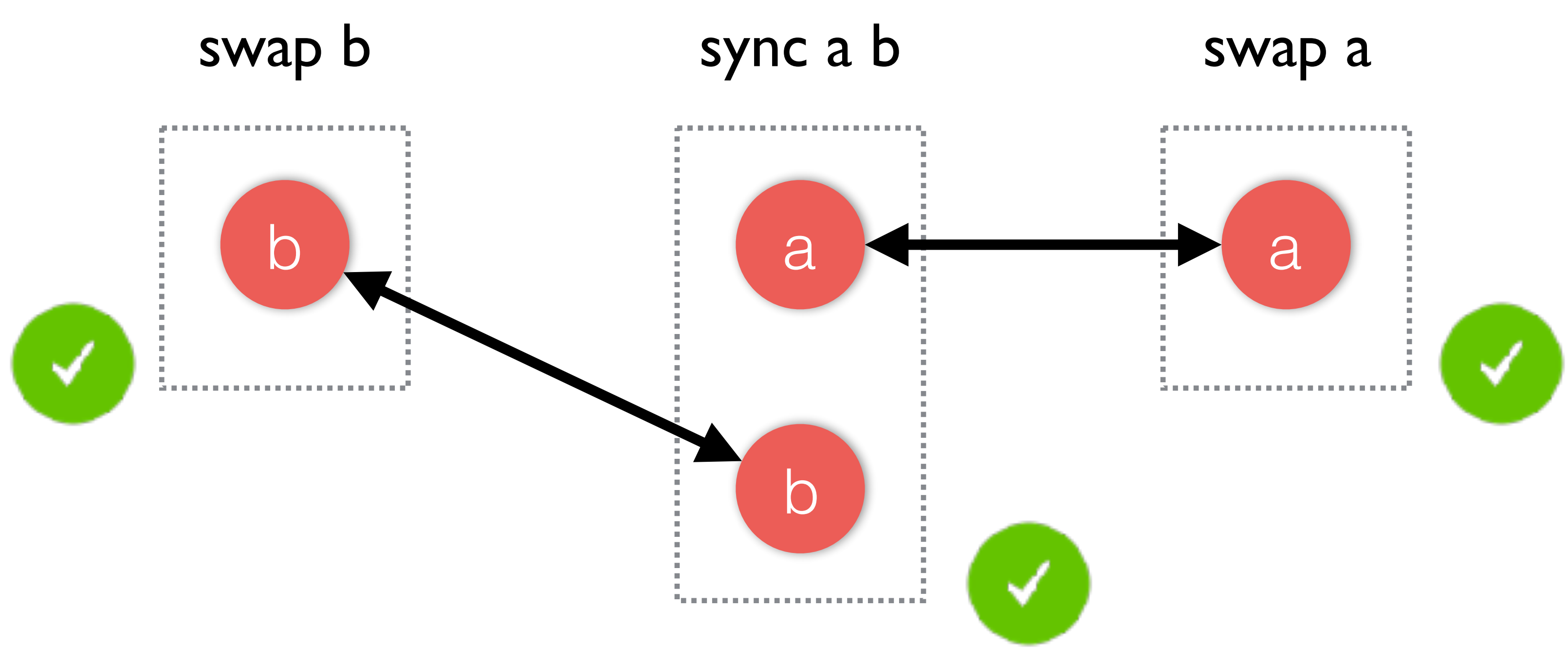
```
let sync a b = (swap a >>> swap b) <+> (swap b >>> swap a)
```



Comparison to CML

- Reagents more expressive than CML — atomicity

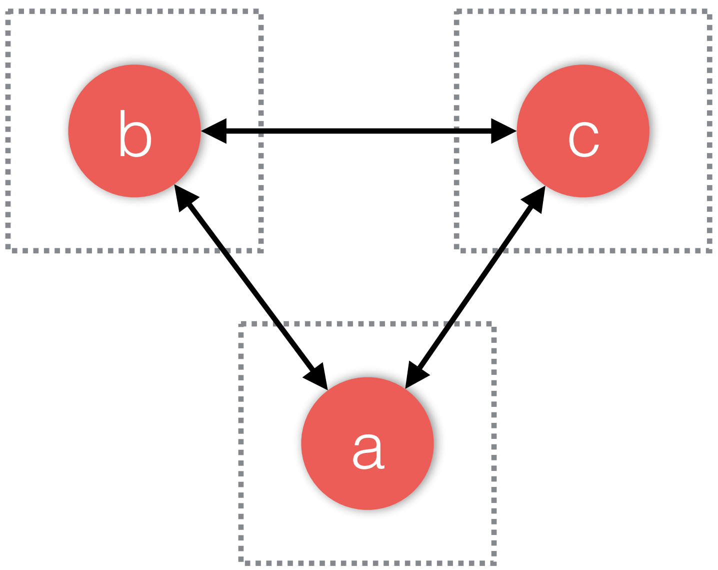
```
let sync a b = (swap a >>> swap b) <+> (swap b >>> swap a)
```



Compassion to TE

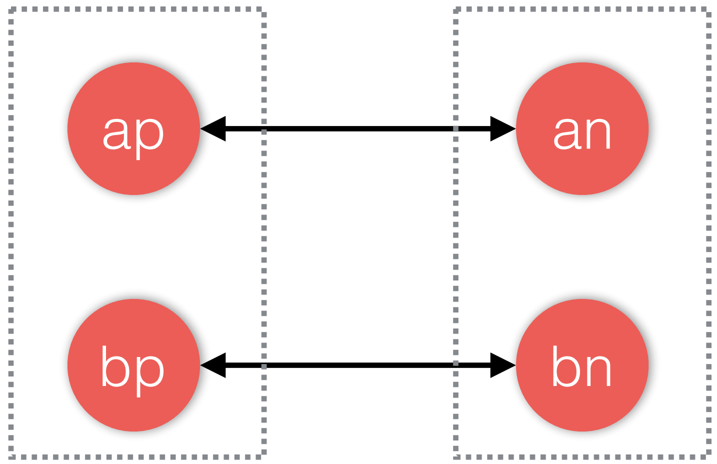
- Weaker than transactional events — 3-way rendezvous not possible

```
let mk_tw_chan () =
  let ab,ba = mk_chan () in
  let bc,cb = mk_chan () in
  let ac,ca = mk_chan () in
  (ab,ac), (ba,bc), (ca,cb)

let main () =
  let sw1, sw2, sw3 = mk_tw_chan () in
  let tw_swap (c1, c2) () =
    run (swap c1 <*> swap c2) ()
  in
  fork (tw_swap sw1); (* a *)
  fork (tw_swap sw2); (* b *)
  tw_swap sw3 ()      (* c *)
```



Also..

```
let (ap,an) = mk_chan () in
let (bp,bn) = mk_chan () in
fork (run (swap ap >>> swap bp));
run (swap an >>> swap bn) ()
```



- Axiomatic model
  - Events ∈ {CAS} ∪ {swaps}
  - Bi-directional communication edges between swaps
  - Unidirectional edges between CASes
- **Safety:** Any schedule that has cycle between txns that involves 1+ communication edge cannot be satisfied
- **Progress:** If there exists such a schedule without cycles, reagents will find it.

Reagent Libraries

Synchronization

- Locks
- Reentrant locks
- Semaphores
- R/W locks
- Reentrant R/W locks
- Condition variables
- Countdown latches
- Cyclic barriers
- Phasers
- Exchangers

Data structures

- Queues
  - Nonblocking
  - Blocking (array & list)
  - Synchronous
  - Priority, nonblocking
  - Priority, blocking
- Stacks
  - Treiber
  - Elimination backoff
- Counters
- Deques
- Sets
- Maps (hash & skiplist)

https://github.com/ocamllabs/reagents

Questions

- Multicore OCaml: github.com/ocamllabs/ocaml-multicore
- OCaml Labs: ocamllabs.io