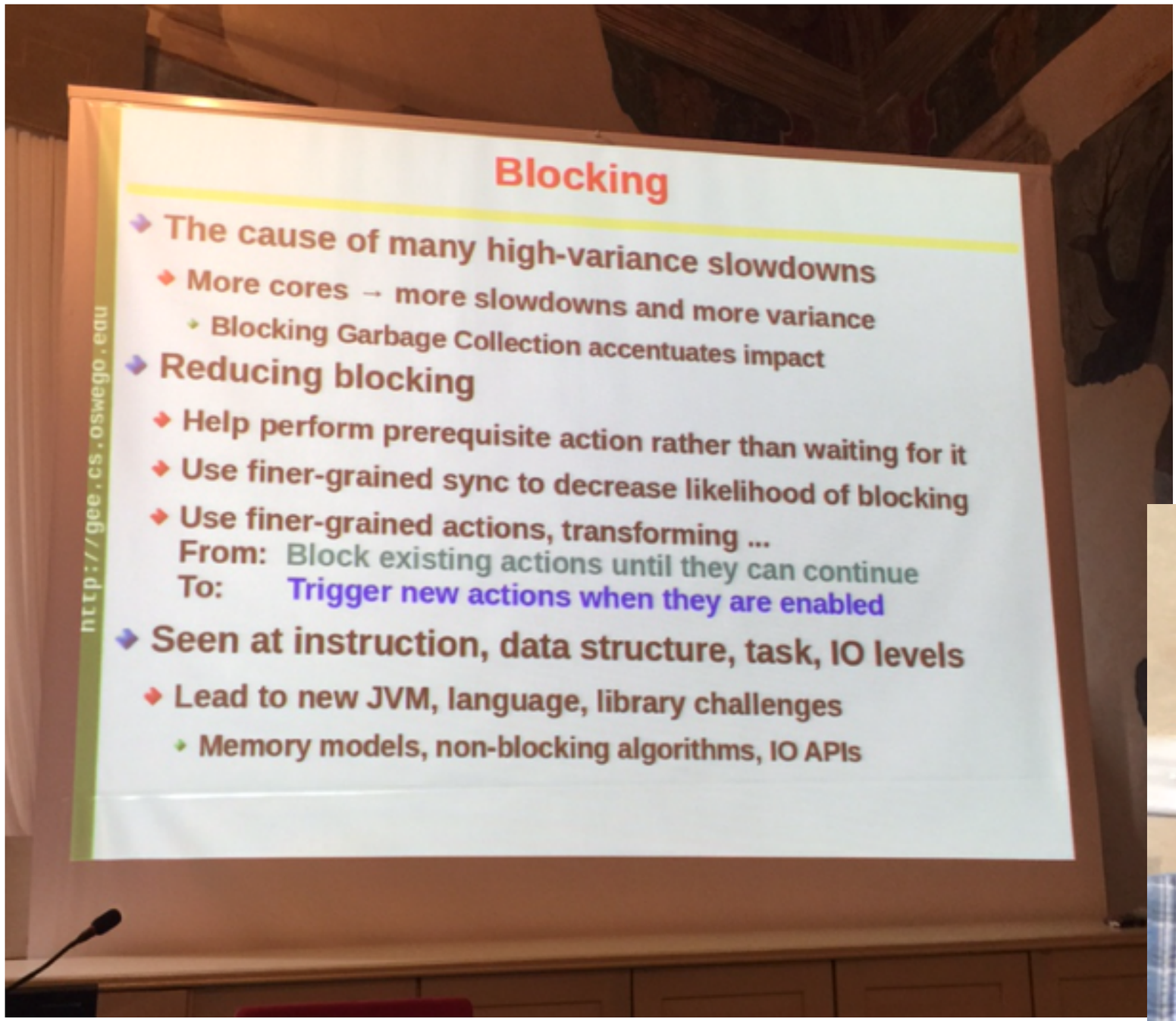
ORCA:
Ownership and
Reference Counting based
Garbage Collection in the
Actor World

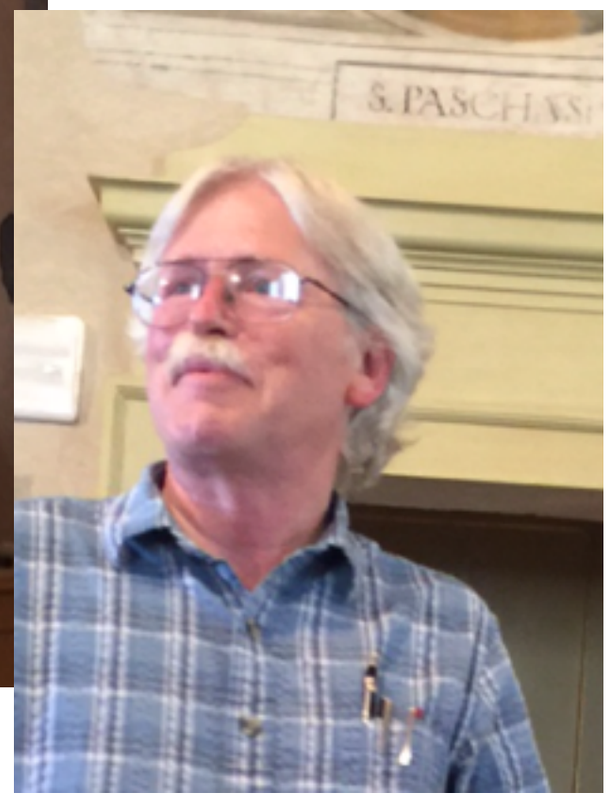
Sylvan Clebsch, Sebastian Blessing,
Juliana Franco, Sophia Drossopoulou

Motivation

Implicit garbage collection is crucial for convenience of programming, however automatic garbage collection often proves to be a performance bottleneck, e.g, [12] reports application pauses of around 3 seconds. This has been subject of improvement in other garbage collection protocols, such as the Pauseless GC Algorithm [18] and the Continuously Concurrent Compacting Collector (C4) [35] from Azul Systems, which allow a non-stop-the-world, concurrent and parallel collection by using read barriers. In concurrent GC protocols, it is important to ensure the absence of race conditions—for instance, a race condition during a marking phase can cause an object to be marked for collection when there still exist references to it. This is often solved using synchronization mechanisms, as for instance in some of the JVM collectors [28], which can cause program performance loss.

Motivation - 2





ORCA - idea

- The actor which created an object is its *owner*.
- Owning actor is responsible for GC-ing its objects.

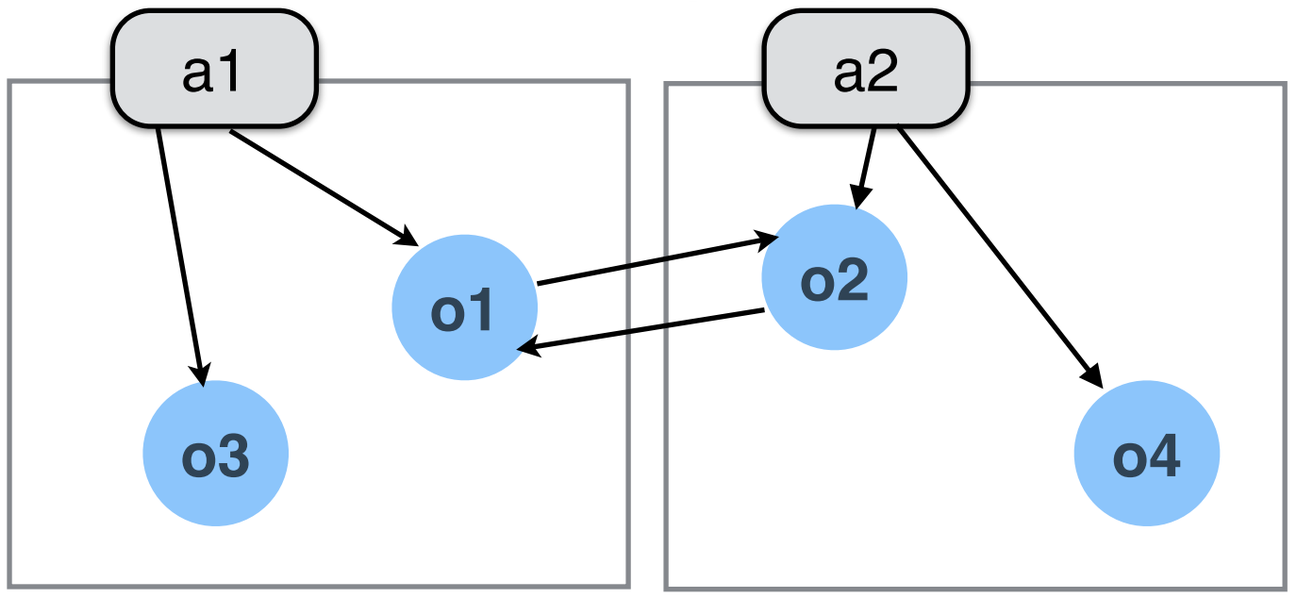
Challenges & Solutions

- An actor may have no path to an object it owns, while other actors have.
- Actor keeps a *Reference Count* for some objects; it represents the foreign references to that object held by the other actors.
- The number of foreign references to an owned object may change (actor's view of its owned objects may be inconsistent with world).
- ORCA-specific messages reconcile an actor's view of its object with the real world view.

Orca - configurations

Runtime configuration

Ownership diagram:



Queues:

APP(o1) :: INC(o1,1)	APP(o3)

Working Sets:

a1, a2, o1, o2, o3	a1, a2, o1, o2, o4

Reference Count Tables:

a1	a2	o1	o2	o3
12	5	1	3	1

a1	a2	o1	o2	o3
10	6	1	4	0

*Grey cells represent owned objects

Actor Object

An actor a has:

- references to objects and actors,
- queue of messages $\mathrm{Queue}(a)$,
- a working set $\mathrm{WS}(a)$, a superset of addresses reachable by actor a
- a ref-count table for *some* addresses from $\mathrm{WS}(a)$,

e.g. $\mathrm{RC}(a1,a2) = 5$

Note, no $\mathrm{RC}(a2,o4)$ entry

Each object is owned by an actor; each actor owns itself.

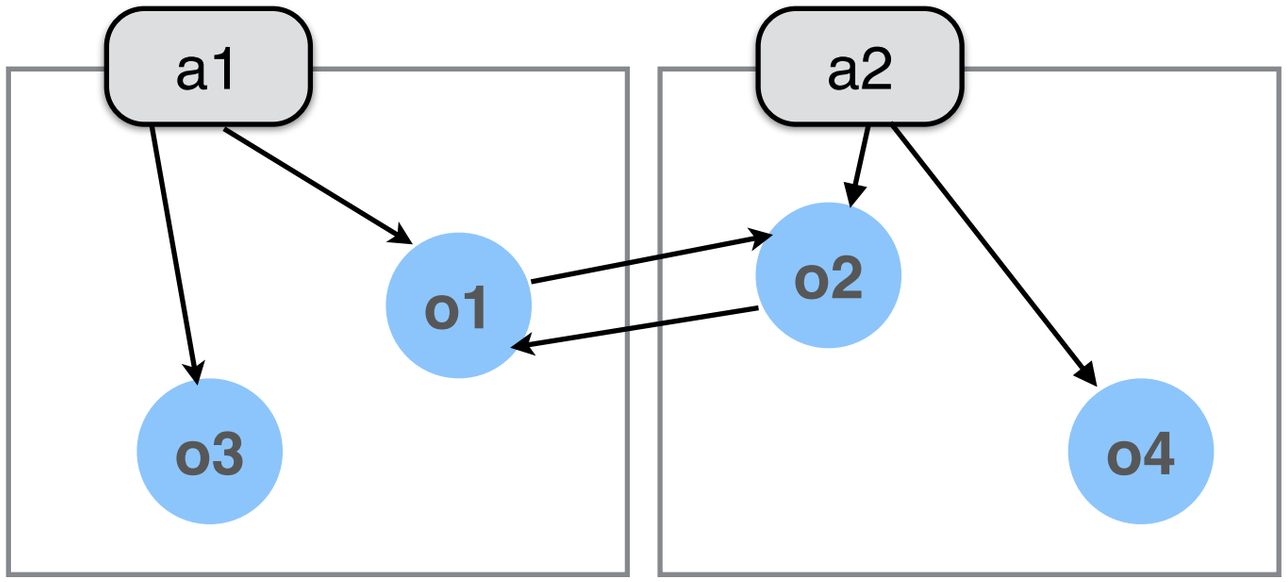
e.g. $\mathrm{Owner}(o1) = a1$

$\mathrm{Mssg}(a, j)$: j-th message in a's queue

e.g. $\mathrm{Mssg}(a1,2) = \mathrm{INC}(o1,1)$

Orca - derived properties

Derived Counts - 1



Queues:

APP(o1) :: INC(o1,1)	APP(o3)

Working Sets:

a1, a2, o1, o2, o3	a1, a2, o1, o2, o4

Reference Count Tables:

a1	a2	o1	o2	o3
12	5	1	3	1

a1	a2	o1	o2	o3
10	6	1	4	0

Local Reference Count:

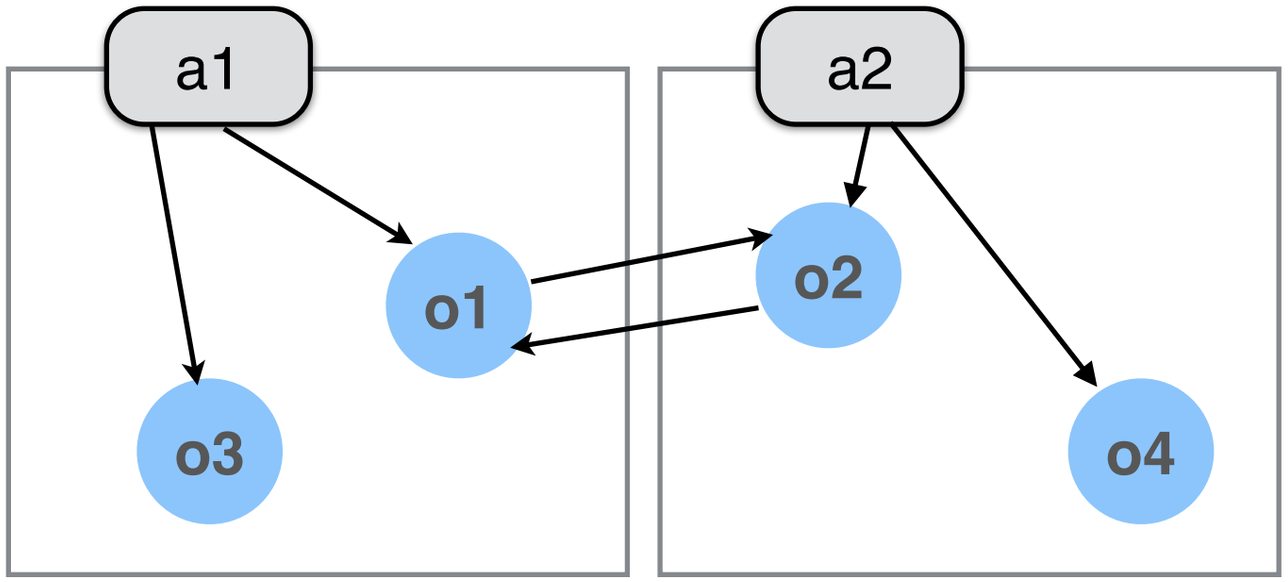
$LRC(\alpha) = RC(\alpha, Owner(\alpha))$

the entry for address α in the ref-cnt table of its owner.

LRC(a1) = 12

LRC(o2) = 4

Derived Counts - 2



Queues:

APP(o1) :: INC(o1,1)	APP(o3)

Working Sets:

a1, a2, o1, o2, o3	a1, a2, o1, o2, o4

Reference Count Tables:

a1	a2	o1	o2	o3
12	5	1	3	1

a1	a2	o1	o2	o3
10	6	1	4	0

FRC(a2) = 5

FRC(a1) = 10

Foreign Reference Count:

FRC(α)

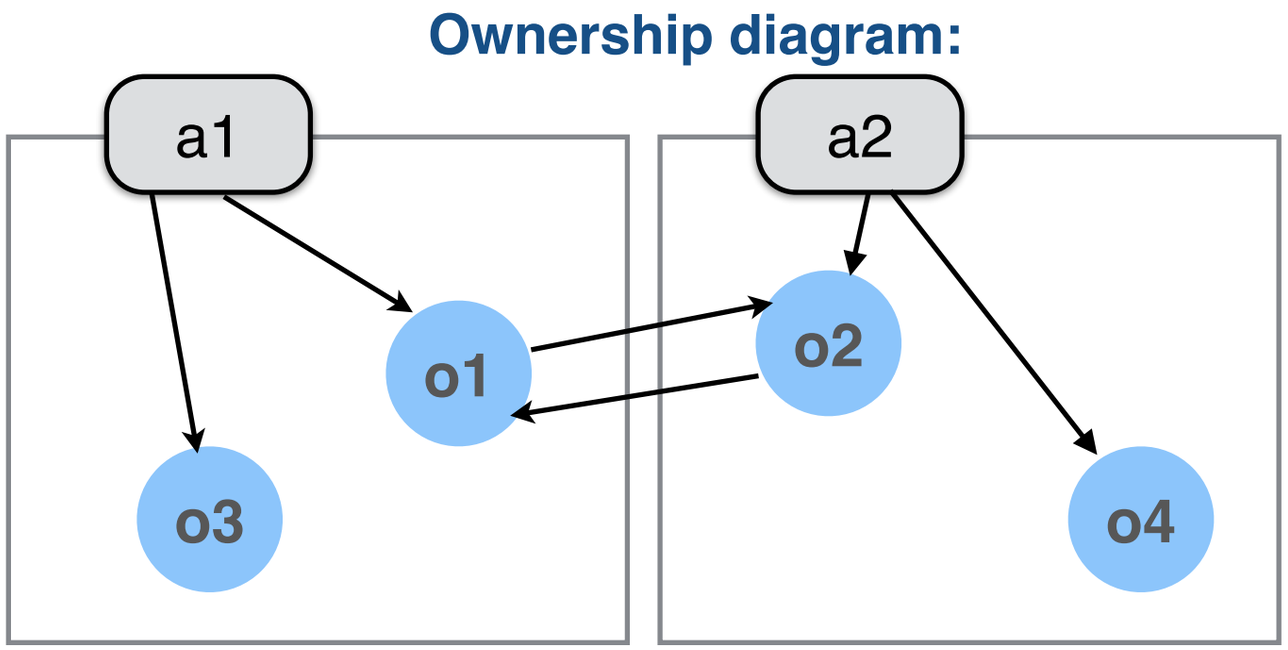
the sum of entries for address α in the ref-cnt table of all non-owning actors

Derived Counts - 3

Incr/decrement Count:

$\mathrm{IDC}(\alpha)$

sum of weighted references to address α in INC and DEC messages in α's owner queue.



Queues:

APP(o1) :: INC(o1,1)	APP(o3)

Working Sets:

a1, a2, o1, o2, o3	a1, a2, o1, o2, o4

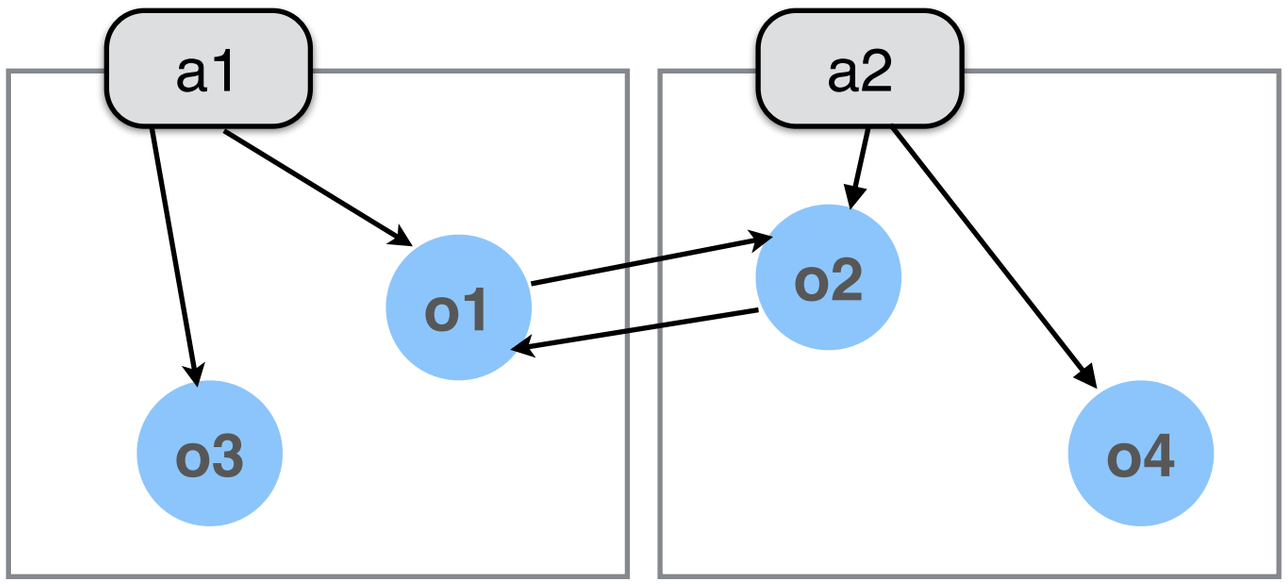
IDC(o1) = 1

Reference Count Tables:

a1	a2	o1	o2	o3
12	5	1	3	1

a1	a2	o1	o2	o3
10	6	1	4	0

Derived Counts - 4



Queues:

APP(o1) : INC(o1,1)

APP(o3)

Working Sets:

a1, a2, o1, o2, o3

a1, a2, o1, o2, o4

Reference Count Tables:

a1	a2	o1	o2	o3
12	5	1	3	1

a1	a2	o1	o2	o3
10	6	1	4	0

Application Message Count:

AMC(α)

number of APP messages in all queues which contain α, addresses owned by α or reachable from α.

AMC(o1) = 1

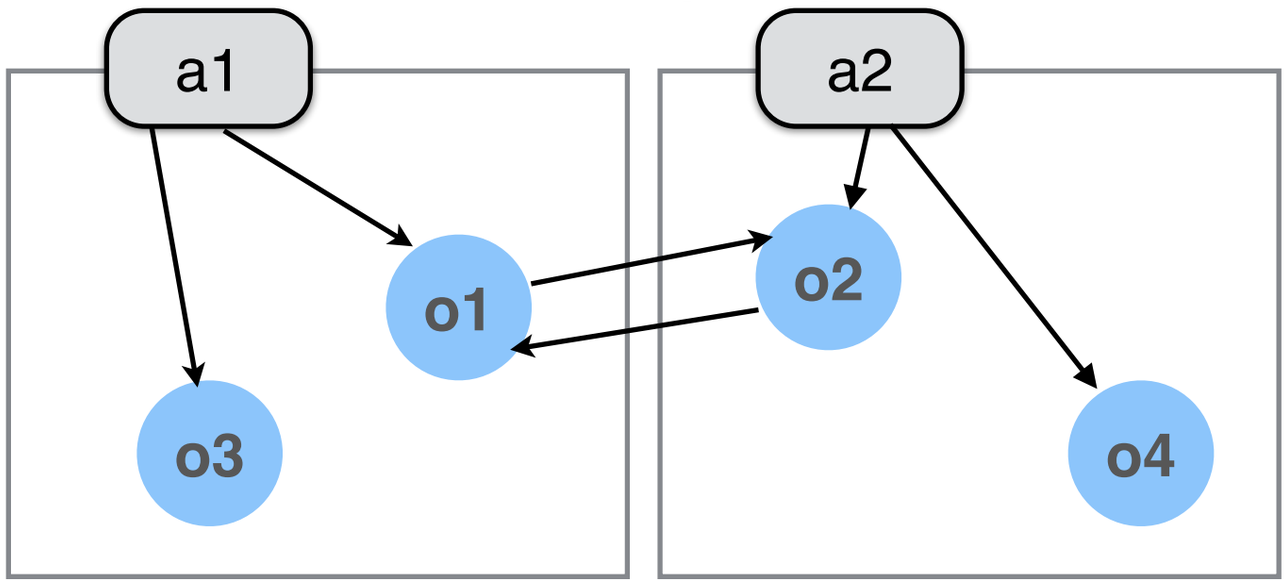
AMC(o2) = 1

AMC(a2) = 1

AMC(a1) = **2**

Derived Counts - 5

Ownership diagram:



Queues:

APP(o1) :: INC(o1,1)	APP(o3)

Working Sets:

a1, a2, o1, o2, o3	a1, a2, o1, o2, o4

Reference Count Tables:

a1	a2	o1	o2	o3
12	5	1	3	1

a1	a2	o1	o2	o3
10	6	1	4	0

*Grey cells represent owned objects

Pending Changes Count:

PCC(α,a,q)

sum of weights of INC & DEC messages in q minus number of APP messages containing address α, in actor a.

Take

q1=APP(o1)

q2=INC(o1,1)

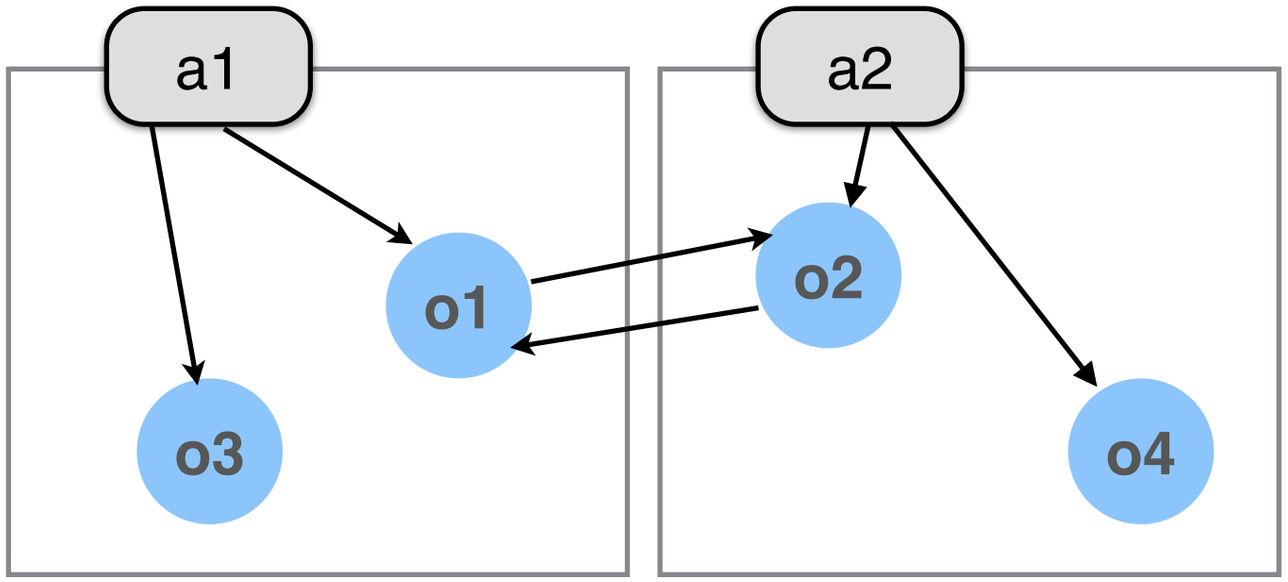
then

PCC(o1,a1,q1) = -1

PCC(o1,a1,q1:q2) = 0

Wellformed Configurations

Well-formedness



Queues:

APP(o1) :: INC(o1,1)	APP(o3)

Working Sets:

a1, a2, o1, o2, o3	a1, a2, o1, o2, o4

Reference Count Tables:

a1	a2	o1	o2	o3
12	5	1	3	1

a1	a2	o1	o2	o3
10	6	1	4	0

WF0: $LRC(\alpha)+IDC(\alpha) = FRC(\alpha)+AMC(\alpha)$

WF1: $RC(a,\alpha) \geq 0$

WF2: α reachable from a
$\Longrightarrow \alpha \in WS(a)$

WF3: α reachable from Msg(_,_)
$\Longrightarrow LRC(\alpha)>0$

WF4: $\alpha \in WS(a) \wedge a \neq Owner(\alpha)$
$\Longrightarrow RC(a,\alpha)>0 \wedge LRC(\alpha)>0$

Consequence:

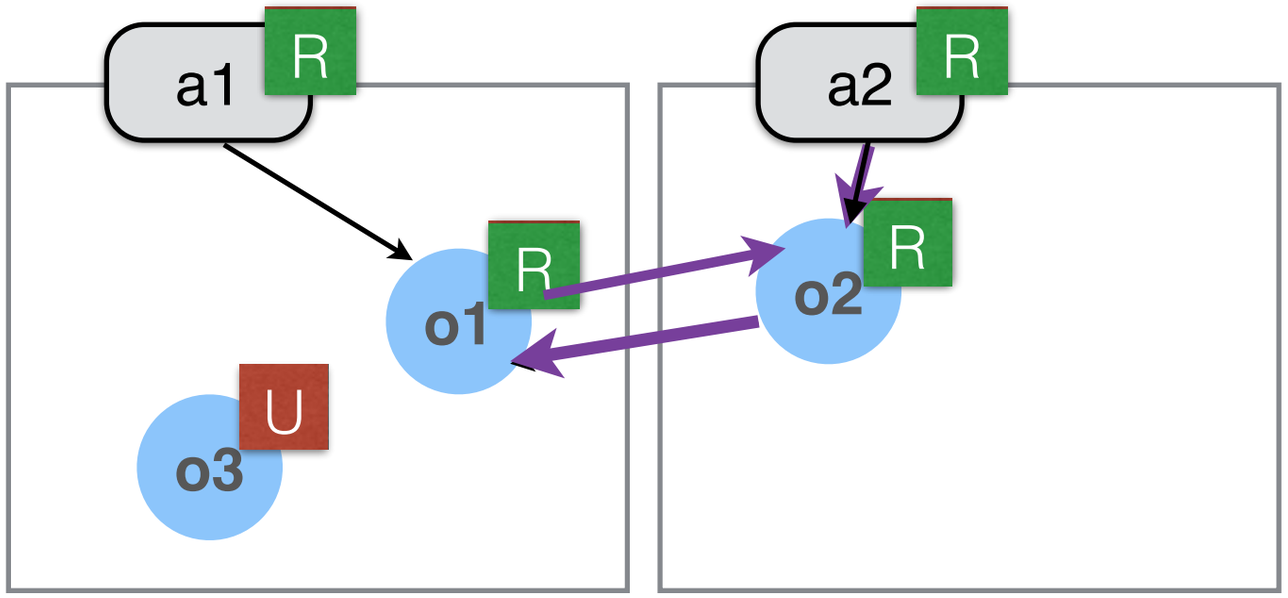
α unreachable from $Owner(\alpha) \wedge LRC(\alpha)=0$
$\Longrightarrow$ α globally unreachable

Garbage Collection

Garbage Collection
the recipe

1. Mark owned objects as **U**.
2. Mark unowned addresses with RC>0 as **U**.
3. Trace from the actor's fields, marking as **R**.
4. Mark owned objects with LRC>0 as **R**.
5. Collect **U**, owned objects
7. Send DEC messages for **U** unowned addresses; set their RC to 0.

Garbage Collection



Queues:

APP(o1) :: INC(o1,1)	DEC(o3, 1)	APP(o1)

Working Sets:

a1, a2, o1, o2, o3	a1, a2, o1, o2

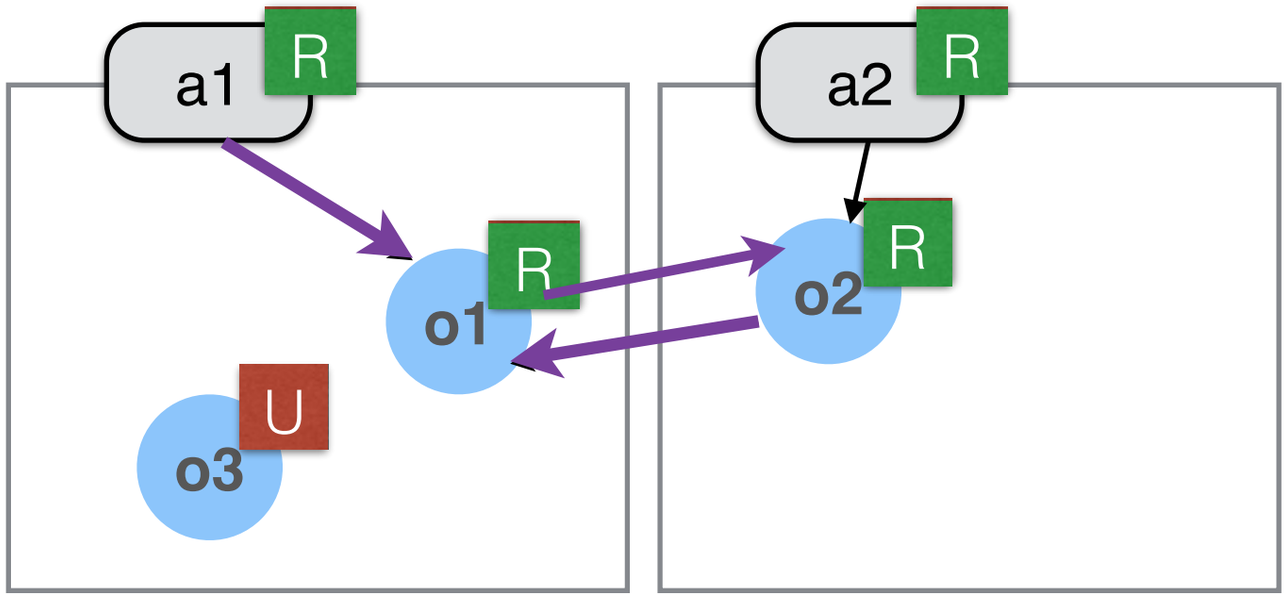
Reference Count Tables:

a1	a2	o1	o2	o3
13	5	2	2	1

a1	a2	o1	o2	o3
11	6	1	4	0

a2 is doing Garbage Collection

1. Mark owned objects as **U**.
   a2, o2
2. Mark unowned addresses with RC>0 as **U**.
   a1, o1, o3
3. Trace from the actor's fields, marking as **R**.
   o2, a2, o1, a1
4. Mark owned objects with LRC>0 as **R**.
   a2, o2
5. Collect **U**, owned objects
6. Send DEC messages for **U** unowned addresses; set their RC to 0.
   o3



Queues:

The queue is empty	APP(o1)

Working Sets:

a1, a2, o1, o2	a1, a2, o1, o2

Reference Count Tables:

a1	a2	o1	o2	o3
13	5	2	2	0

a1	a2	o1	o2	o3
11	6	1	4	0

Collecting Objects:
the recipe

a1 is doing Garbage Collection

1. Mark owned objects as **U**.
   a1, o1, o3
2. Mark unowned addresses with RC>0 as **U**.
   a2, o2
3. Trace from the actor's fields, marking as **R**.
   o1, o2, a2
4. Mark owned objects with LRC>0 as **R**.
   a1, o1
5. Collect **U**, owned objects
   o3
6. Send DEC messages for **U** unowned addresses; set their RC to 0.

Maintaining Well-formedness

WF0: LRC(α) + IDC(α) = FRC(α)+AMC(α)

WF1: RC(a,α) ≥ 0

WF2: α reachable from a $\implies$ $\alpha \in$ WS(a)

WF3: α reachable from Msg(_,_) $\implies$ LRC(α)>0

WF4: $\alpha \in$ WS(a) $\wedge$ a≠Owner(α) $\implies$ RC(a,α)>0 $\wedge$ LRC(α)>0

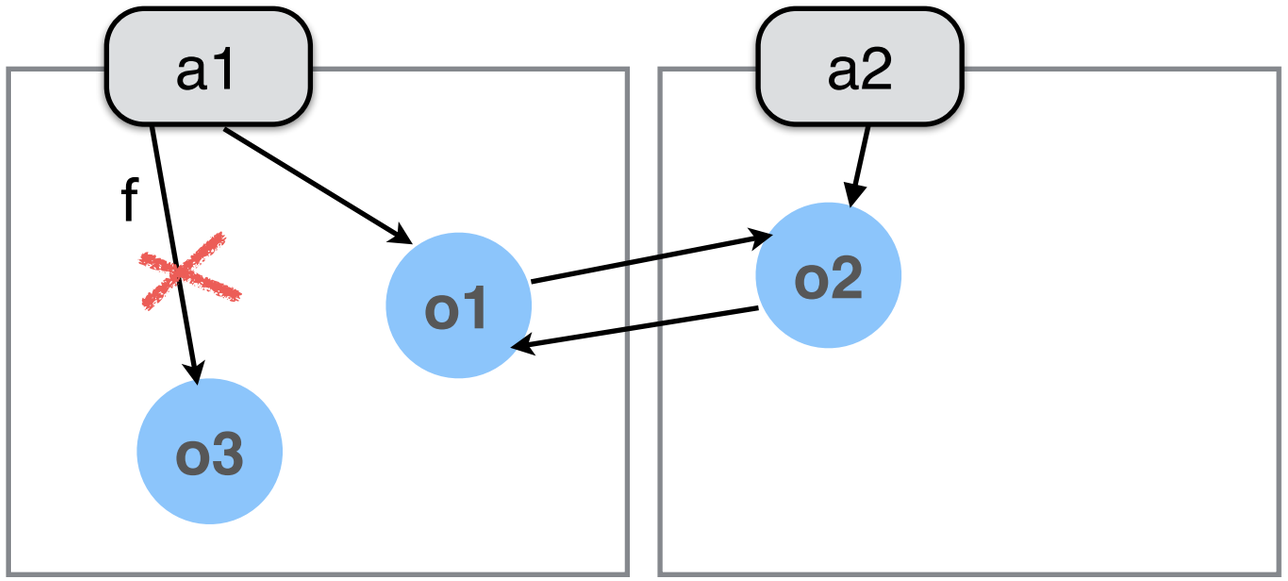
WF5: q::_ = Queue(a) $\implies$ LRC(α)+PCC(α,a,q)>0

Which actor actions affect WF1-5?

- actor sends a Pony message — WF0, WF3, WF5
- actor receives a Pony message — WF0, WF2, WF3
- actor receives an ORCA message (INC or DEC) — WF2, WF0, WF5
- actor garbage collects
- heap topology changes — WF0, WF1, WF2, WF3, WF4, WF5
- call/execute synchronous method

No Pony's types

Maintaining wellformedness



Queues:

APP(o1) :: INC(o1,1)	APP(o1)

Working Sets:

a1, a2, o1, o2, o3	a1, a2, o1, o2, o3

Reference Count Tables:

a1	a2	o1	o2	o3
13	5	2	2	1

a1	a2	o1	o2	o3
11	6	1	4	1

Action: heap mutation

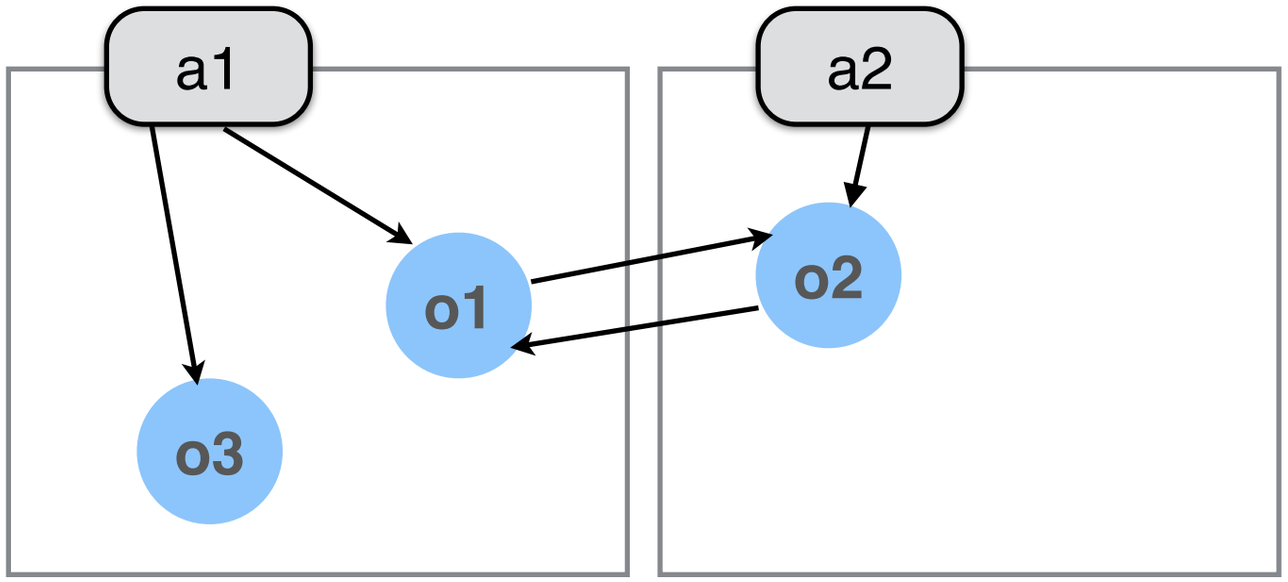
a1.f = null

Which WF invariant is affected?

None!

However o3 is in the working set of a1… at least until the next Garbage Collection cycle!

Maintaining wellformedness



Queues:

APP(o1) :: INC(o1,1)	APP(o3) **:: APP(o1)**

Working Sets:

a1, a2, o1, o2, o3	a1, a2, o1, o2

Reference Count Tables:

a1	a2	o1	o2	o3
12+1	5-**1**	1+1	3-1	1

a1	a2	o1	o2	o3
10	6	1	4	0

Action: actor sends message

a1 sends to a2 APP(o1)

WFO:
LRC(o1) + IDC(o1) = FRC(o1) + AMC(o1)

↓

+1

How can a1 preserve WF0?

RC(a1,o1) += 1

RC(a1,a1) += 1

RC(a1,o2) -= 1

RC(a1,a2) -= 1

Why Causality?

Causality required because …

WF0: LRC(α) + IDC(α) = FRC(α)+AMC(α)

WF1: RC(a,α) ≥ 0

WF2: α reachable from a $\implies$ $\alpha\in$WS(a)

WF3: α reachable from Msg(_,_) $\implies$ LRC(α)>0

WF4: $\alpha\in$WS(a) ∧ a≠Owner(α) $\implies$ RC(a,α)>0 ∧ LRC(α)>0

WF5: q::_ = Queue(a) $\implies$ LRC(α)+PCC(α,a,q)>0

Race Conditions?

Race Conditions?

Source language level

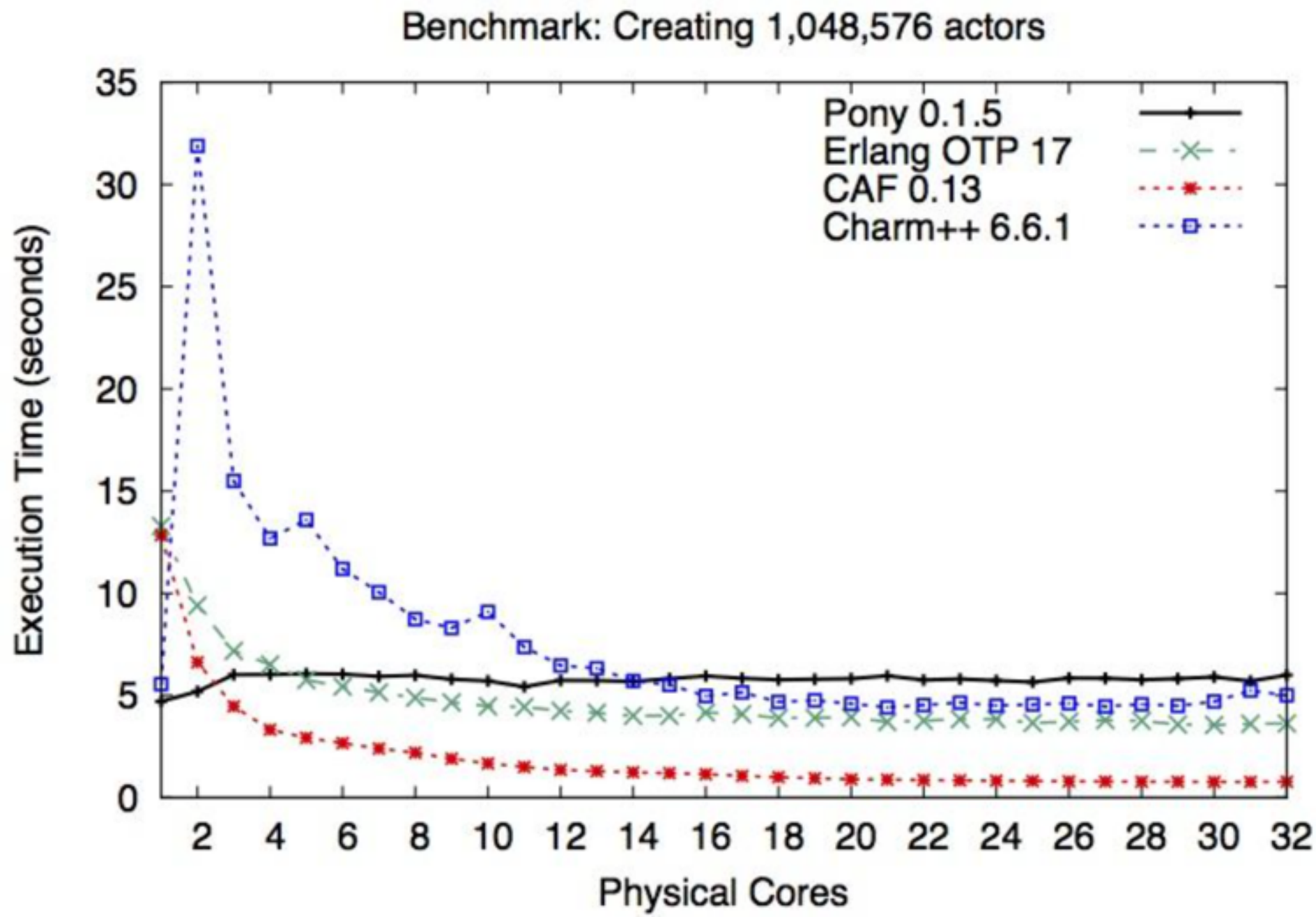
- Can an actor read an object mutated by another actor?
- Can an actor mutate an object mutated by another actor?

Source language and GC level

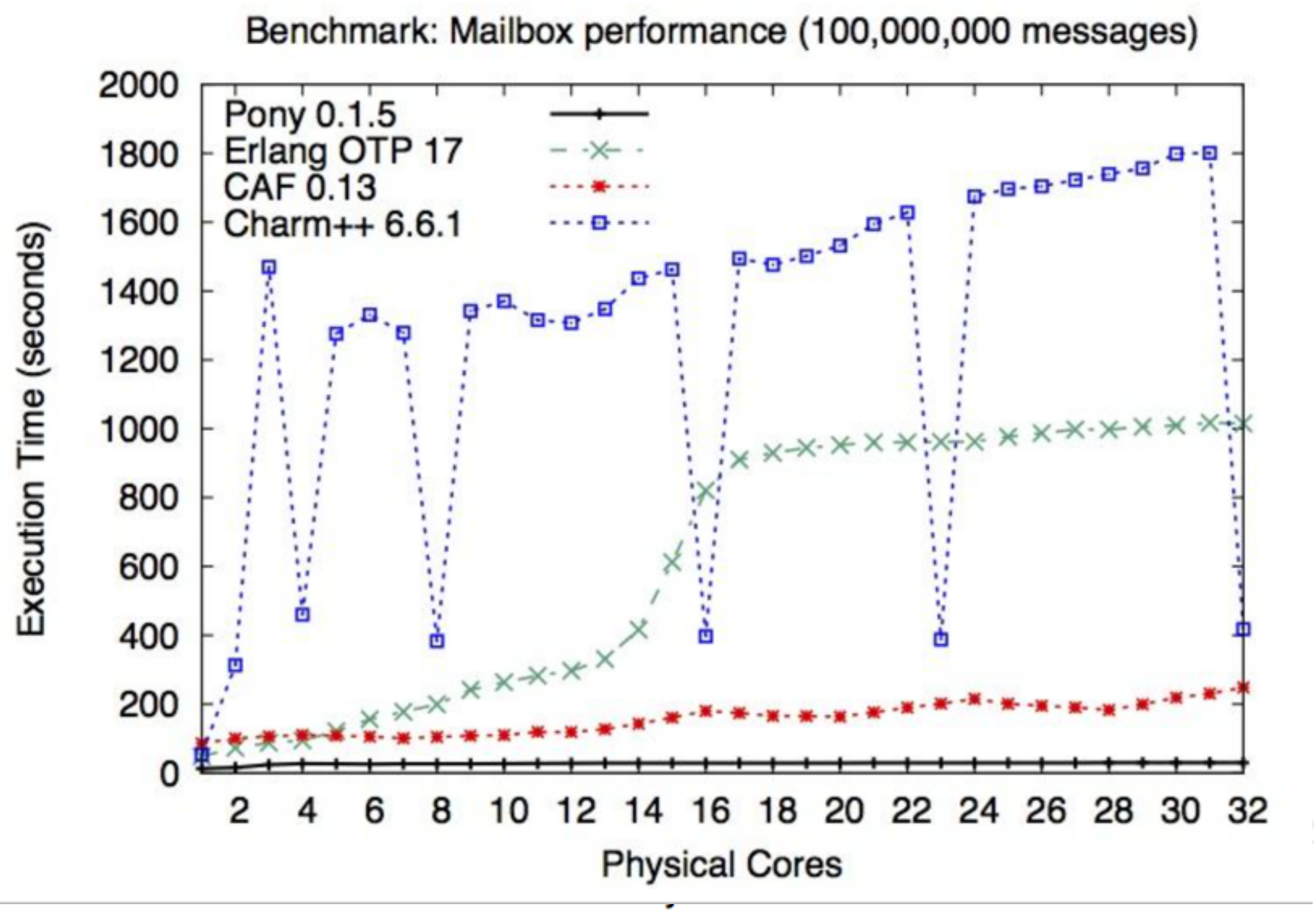
- Can an actor trace an object mutated by another actor?
- Can an actor read an object garbage collected by another actor?
- Can an actor trace an object garbage collected by another actor?

How efficient is ORCA?

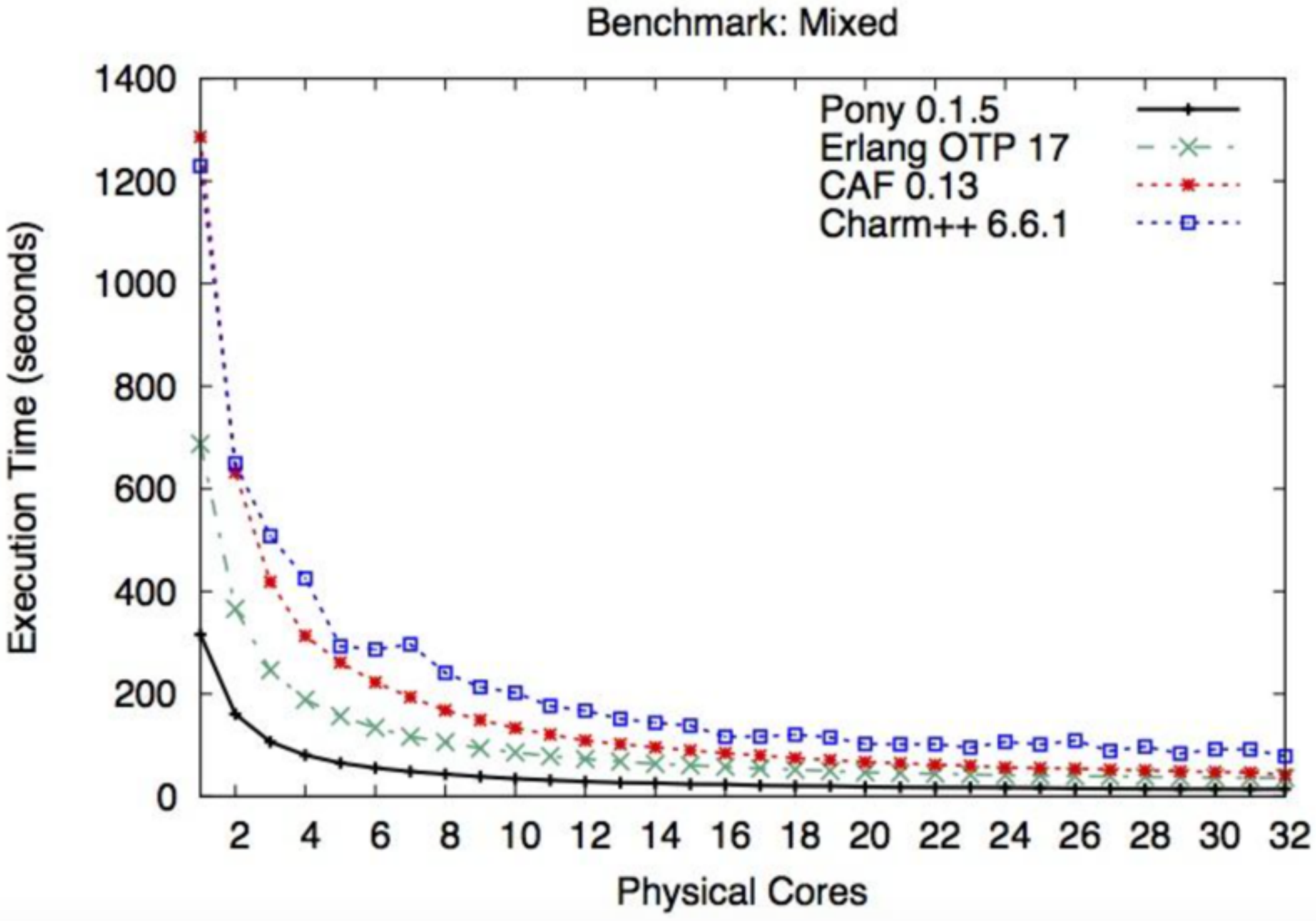
Benchmark - creating 1M actors



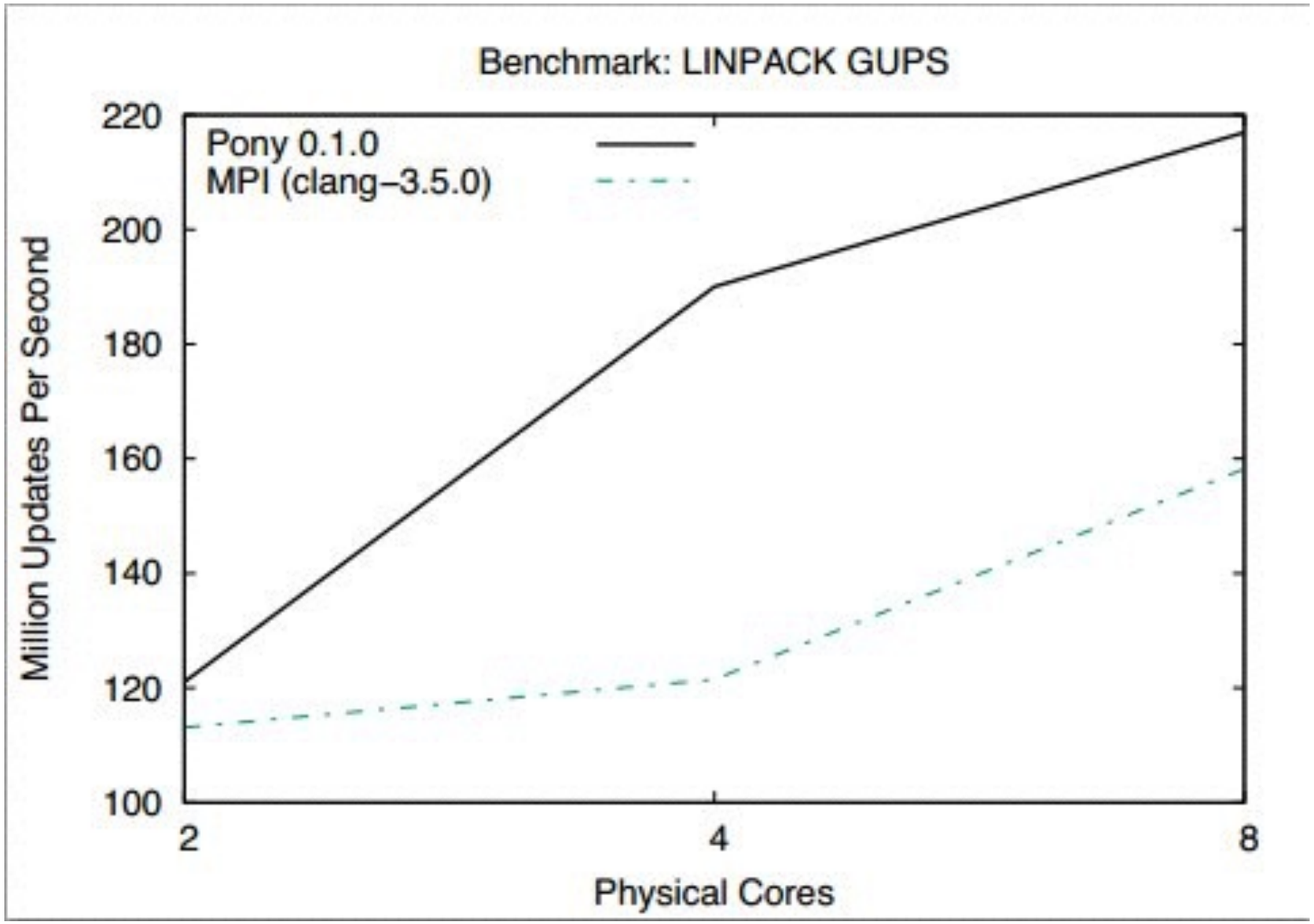
Benchmark - mailbox 100,000,000 msgs



Benchmark - mixed



Benchmarks - GUPS linpack



Summary

- Fully concurrent (other actors may run while an actor is GCing)
- No synchronisation mechanism needed, no stop-the world, no barrier

- Owning actor responsible for GC-ing its objects
- Application of the actor paradigm into the design of the protocol

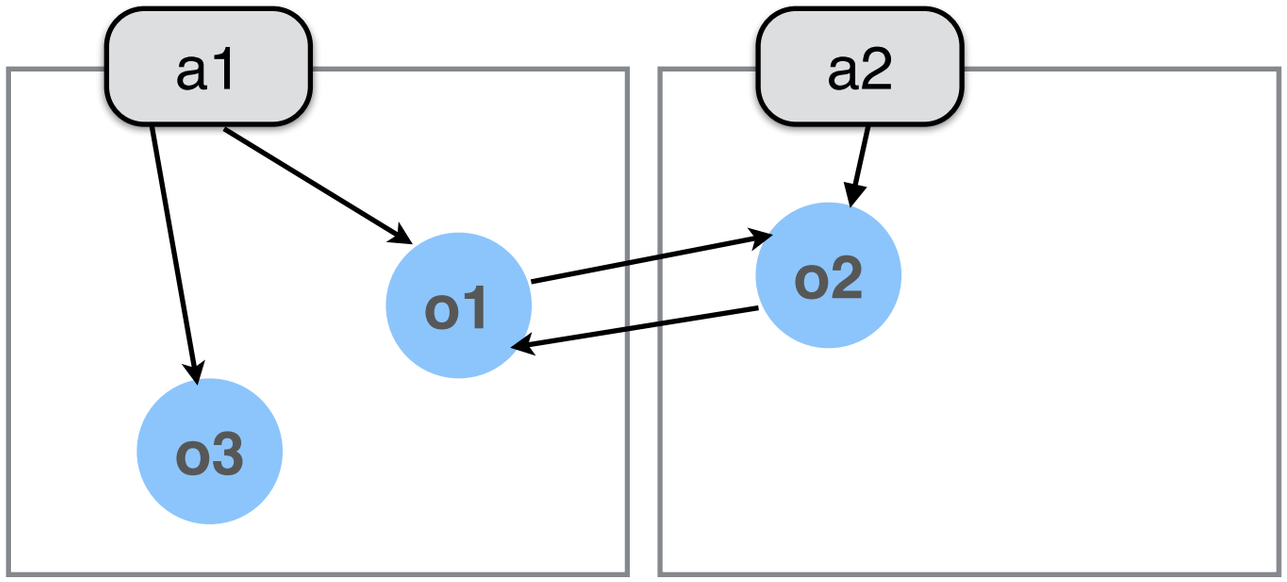
- Further Work:
  - complete the formal model
  - weaken the WF-conditions and Causality requirement
  - **HOMEWORK**: apply to Encore

more about it: OOPSLA'13, ICOOOLPS'15

More about Pony at http://www.ponylang.org/

Thank you!

Ownership diagram:



Queues:

APP(o1) :: INC(o1,1)	~~APP(o3)~~ :: APP(o1)

Working Sets:

a1, a2, o1, o2, o3	a1, a2, o1, o2 **o3**

Reference Count Tables:

a1	a2	o1	o2	o3
13	5	2	2	1

a1	a2	o1	o2	o3
10+1	6	1	4	0+1

*Grey cells represent owned objects

Invariants affected here
3,4 & 5

Maintaining wellformedness

Action: actor receives message

a2 receives APP(o3)

LRC(o1) + IDC(o1) = FRC(o1) + AMC(o1)

↓

-1

How can a2 preserve the invariant?

it must increment its RC to o3 and a1 by 1

Also, o3 is now in the working set of a2.