Ownership and Reference Counting based Garbage Collection in the Actor World

Sylvan Clebsch Sebastian Blessing Juliana Franco Sophia Drossopoulou
Causality, and Imperial College London
{sylvan,sebastian,sophia}@causality.io j.vicente-franco@imperial.ac.uk

Abstract

We propose Pony-ORCA, a fully concurrent protocol for garbage collection in the actor paradigm. It allows actors to perform garbage collection concurrently with any number of other actors. It does not require any form of synchronization across actors except those introduced through the actor paradigm, i.e. message send and message receive.

Pony-ORCA is based on ideas from ownership and deferred, distributed, weighted reference counting. It adapts the messaging system of actors to keep the reference count consistent.

We introduce a formal model and describe the invariants on which it relies. We illustrate through an example and sketch how these invariants are maintained. We show some benchmarks and argue that the protocol can be implemented efficiently.

1. Introduction

The actor paradigm [4] was proposed in 1973 by Carl Hewitt [22]. An actor is a computational entity that, in response to a message it receives, can: 1) send a finite number of (asynchronous) messages to other actors; 2) create a finite number of new actors; and 3) designate the code to be executed for the next message it receives. The code executed upon receipt of a message is called a behaviour. The actor paradigm has been adopted in a functional setting, e.g. in Erlang [7], and in the object-oriented paradigm [5, 10, 14, 32].

Implicit garbage collection is crucial for convenience of programming, however automatic garbage collection often proves to be a performance bottleneck, e.g, [12] reports pauses of around 3 seconds.

In this paper, we propose Pony-ORCA, a garbage collection protocol for actor-based object oriented programming languages. We have implemented Pony-ORCA for the language Pony [6], but our protocol is applicable for any languages which fulfils the criteria we give later. Our protocol is based on ownership and deferred distributed weighted reference counting.

Ownership types [15, 30] were proposed with the remit to delimit groups of related objects into different areas of the heap. They were used for garbage collection under the requirement that there are no incoming references to these areas [31]. Such a scheme works well in the concurrent setting, but the no-incoming references requirement is often far too strong.

Reference counting garbage collection puts no requirement on the heap structure; instead, it tracks, for each object, a count of the number of references to it held by other objects [8]. This approach has been further developed to detect cycles and to deal with the distributed setting [19, 28]. However, the approach has poor locality and thus, in the concurrent setting, it requires synchronization across the various threads [25].

We employ the locality found in actors, and the implicit synchronisation afforded by the actor messaging system, to develop a fully concurrent garbage collection algorithm. Pony-ORCA allows the fully concurrent garbage collection of objects as well as actors. In particular:

- An actor may perform garbage collection concurrently with other actors while they are executing any kind of behaviour.
- An actor may decide whether to garbage collect an object solely based on its own local state, without consultation with, or inspecting the state of, any other actor.
- No synchronization between actors is required during garbage collection, other than potential message sends.
- An actor may garbage collect between its normal behaviours, i.e. it need not wait until its message queue is empty.
- Pony-ORCA can be applied to several programming languages, provided that they satisfy the following two requirements:
  - Actor behaviours are atomic.
  - Message delivery is causal—i.e. messages arrive before any messages they may have caused, if they have the same destination.

Our approach is based on *implicit* ownership, whereby an actor owns any object that it has allocated. Each actor is responsible for garbage collecting the objects it owns. The challenge is that an actor may have no path to an object it owns, while other actors may still have paths to that object, or the object may appear in messages in some other actor's message queue. It would be erroneous to garbage collect such an object.

In our approach, an actor maintains a reference count for all the objects it owns. The reference count is guaranteed to be non-zero for any object which is accessible from any message or any actor other than its owner. Thus, an actor can safely garbage collect any object which is *locally* unreachable, and whose reference count is zero.

There is the remaining challenge of ensuring that the reference count indeed is non-zero for objects which are accessible from actors other than their owner. This is maintained through a system whereby all the actors which may reach an object they do not own also maintain their (foreign) reference count, and whereby the sum of all foreign reference counts of an object corresponds to its owning actor's (local) reference count for that object. This requirement needs to be modified to take into account the messages in the various queues. Therefore, in Pony-ORCA, when an actor receives or sends references to objects, it may need to inform the owning actor through protocol-specific messages.

Our paper is organized as follows: in Section 2 we present related work on garbage collection protocols; in Section 3 we briefly introduce the Pony language; in Section 4 we define the runtime configuration and what is a well-formed configuration, in the Pony-ORCA protocol; in Section 5 we argue why objects can be safely collected and how the consistency of the runtime configuration is maintained; we discuss our results in Section 6; and we finish the paper with conclusions in Section 7.

2. Related work

Previously, actor-model languages and libraries have used five different approaches to garbage collection. The first is to combine manual termination of actors with a standard tracing garbage collector. These are primarily JVM based implementations, such as Scala [21], Akka [5], Kilim [32], AmbientTalk[33], and SALSA 2.0 [34]. The second is to combine manual termination with a largely per-actor tracing collector, using copying of messages to move data into actor heaps, that also has some global data. These are primarily BEAM based implementations, such as Erlang [7] and Elixir [20]. The third is to transform the actor graph into an object graph and use a tracing collector to collect actors as well as objects, as done in ActorFoundry [3]. The fourth is to use reference listing and snapshots to collect actors, as done in SALSA 1.0 [35]. The fifth is to use a message-based actor collection protocol that can collect actors and detect termination fully concurrently, without a stop-the-world step [17].

In this work, we extend message-based actor collection [17], applying it to passive object collection when objects are shared by reference amongst actors.

Our work draws heavily on existing distributed garbage collection algorithms, particularly on distributed reference counting [18, 19, 25, 27–29]. A key difference is that Pony-ORCA does not have reference count cycles amongst objects, since only actors hold reference counts for objects and those counts are independent of the number of paths to an object in an actor's reachable heap. Thus, we do not require cycle detection [8, 18, 19, 25, 27] or reference listing [29]. This approach also eliminates heap change related reference count changes, which in effect gives highly deferred reference counting [9].

Pony-ORCA is also influenced by the Emerald garbage collector [26], particularly in the design goals of comprehensiveness, concurrency, expediency, efficiency, and correctness. In addition, independently collected actor heaps are similar to Emerald's local collector, with our message protocol effectively replacing the global collector.

3. The language Pony and Causal Message Delivery

The language Pony supports actors (active objects), and objects (passive objects). Objects and actors have fields and synchronous methods; in addition, actors have asynchronous methods, called behaviour. Actors may receive asynchronous messages which contain any number of parameters. These may be addresses or literals (e.g. integers). The messages are stored in a queue. Whenever an actor is enabled, it removes the top message from its queue and executes the body of the corresponding behaviour. Actors and objects may call synchronous methods on objects and asynchronous behaviours on other actors. Pony also contains further features: traits, algebraic data types, generics etc., which, however, we will not be discussing here. A formal semantics of the Pony subset discussed here appears in [6].

Causal message delivery requires that whenever a message, $\mathtt{msg}_1$, is a direct or indirect cause of another message, $\mathtt{msg}_2$, and the destination of the two messages is the same, $\mathtt{msg}_1$ will be delivered before $\mathtt{msg}_2$. A message `msg` is *a cause* of a further message $\mathtt{msg}'$, if a) an actor sends $\mathtt{msg}'$ after receiving `msg`, or b) an actor sends $\mathtt{msg}'$ after sending `msg`, or c) there exists an intermediate message $\mathtt{msg}''$ such that `msg` is a cause of $\mathtt{msg}''$, and $\mathtt{msg}''$ is a cause of $\mathtt{msg}'$. Therefore, the causal relationship is asymmetric, acyclic, and transitive.

For example, if actor `A` sends message $\mathtt{msg}_1$ to actor `B`, and then sends message $\mathtt{msg}_2$ to actor `C`, and actor `C` sends message $\mathtt{msg}_3$ to actor `B` after receiving $\mathtt{msg}_2$, then $\mathtt{msg}_1$ is a cause of $\mathtt{msg}_2$, and also $\mathtt{msg}_2$ is a cause of $\mathtt{msg}_3$, and by transitivity $\mathtt{msg}_1$ is a cause of $\mathtt{msg}_3$. Therefore, causal message delivery requires that actor `C` will receive $\mathtt{msg}_1$ before receiving $\mathtt{msg}_3$.

Causal message delivery is not required in the original formulation of the actor paradigm [22], where it mandates that message delivery is guaranteed, but need not be ordered. The motivation for this is to make the actor-model as general as possible. For the same reason, the original formulation does not require buffered queues. However, in [17] it is shown how causal messaging can be implemented efficiently in the concurrent setting, and in [11] we have implemented it in the distributed setting.

Crucially, causal message delivery has been of paramount importance in the development of the actor collection protocol [17], and of various distributed protocols developed in [11]. We plan to discuss efficient implementation of causal messaging in further work.

4. The Pony-ORCA Garbage Collection Protocol

Pony-ORCA is based on a reference counting scheme, whereby each actor keeps a reference count for the objects it owns. An actor can decide whether to garbage collect an object it owns solely on the basis of whether the object is reachable from the owning actor and whether the object's (local) reference count is zero. Therefore, the owner's reference count for the object must correspond to references held in other actors or in enqueued messages. For this reason, non-owning actors also hold (foreign) reference counts to objects. When an actor sends, receives, or drops a reference to an object it does not own, it sends protocol-specific messages to the owner. These protocol-specific messages result in the owner updating its (local) reference count.

For the approach to work, we require that the owner's reference count for an object is a true reflection of the global configuration, namely the owner's (local) reference count together with pending protocol-specific messages is the same as the sum of the (foreign) reference counts in the other actors together with pending language-level messages. In order to maintain this invariant, whenever objects are sent, received, or become unreachable, the reference counts will need to be modified and/or protocol-specific messages sent.

In this section we describe our protocol in more detail, and develop a formal model. In section 4.1 we show diagrammatically some actors, heaps and queues, and discuss which objects are globally unreachable. In section 4.2 we show Pony-ORCA specific data structures. In section 4.3 we define what it means for the owner's reference count to be a true reflection of the global configuration along with further necessary well-formedness conditions.

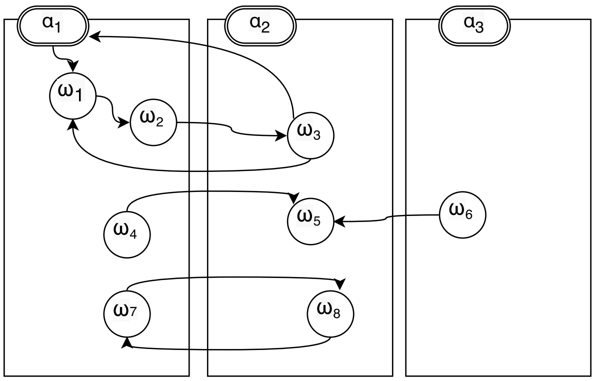

Figure 1. Ownership and References diagram.

4.1 Actors, queues, and unreachable objects

In Pony, actors can create new objects, send messages to other actors (with references to actors and objects, not necessarily allocated by the sending actor) and receive messages. As we said earlier, it is possible for an actor to own an object without having a reference to it, while other actors do have references to it—for instance, an actor may create an object, send it to other actors and then drop the reference to that actor. Given this, the protocol needs to ensure that an object, even though is not reachable from its owner, is not garbage collected if there exists another actor or a message in one of the actor's queues with a reference to it.

EXAMPLE 1. *Consider the Ownership and References diagram from Figure 1. We have actors α_1, α_2 and α_3 and show them in rounded double boxes. We have objects $\omega_1, \omega_2, \ldots, \omega_8$ and we show them in rounded single boxes. We show ownership through square boxes, e.g. α_1 owns ω_7. We show references through arrows, e.g. ω_3 references ω_1. Notice that object ω_3 is not reachable from α_2, its owner, but it is reachable from α_1.*

In Figure 2 we show an abstract representation of heaps, message queues and reference counts. We will discuss heaps and reference counts in the next section. We have message queues Qs_1 and Qs_2. In Qs_1 all the queues are empty. In Qs_2 the actor at α_1 has a Pony-level message $\texttt{APP}(\omega_6)$, the actor at α_2 has a message $\texttt{APP}(\omega_7, \omega_6, \omega_7)$ and the queue of the third actor is empty. Message identifiers are not used in Pony-ORCA; only addresses are considered.

If we consider the diagram from Figure 1 together with the queues from Qs_1 then the objects $\omega_4, \omega_5, \omega_6, \omega_7, \omega_8$ and actor α_3 are globally unreachable. However, if we consider the queues from Qs_2, then ω_4 is the only globally unreachable reference.

4.2 Runtime configuration modelling Pony-ORCA

We now model the data structures used in Pony-ORCA, as well as the Pony runtime entities relevant to the soundness of our protocol. In Pony-ORCA each actor contains a (local) reference count for any object it owns, as well as a (foreign) reference count for any other actor or object it does not own. We represent ownership through the mapping *Owner*, and we unify local and foreign reference counts to one mapping, RC.

We consider sets of addresses, $\mathbb{S}_{all}$, and distinguish between object addresses, $\mathbb{S}_{obj}$, and actor addresses, $\mathbb{S}_{act}$. Every actor or object has an owner, which is an actor. We require that the owner of an actor is the actor itself. We indicate addresses through ι, ι', etc, actor addresses through α, α', etc., and object addresses through ω, ω', etc.

Heaps

H_1:

$$\begin{aligned} Heap_1(\alpha_1) &= \{\, \alpha_1, \alpha_2, \omega_1, \omega_2, \omega_3, \omega_4, \omega_5, \omega_7, \omega_8 \,\} \\ Heap_1(\alpha_2) &= \{\, \alpha_1, \alpha_2, \omega_1, \omega_2, \omega_3, \omega_5, \omega_7, \omega_8 \,\} \\ Heap_1(\alpha_3) &= \{\, \alpha_2, \alpha_3, \omega_5, \omega_6 \,\} \end{aligned}$$

H_2:

$$\begin{aligned} Heap_2(\alpha_1) &= \{\, \alpha_1, \alpha_2, \omega_1, \omega_2, \omega_3 \,\} \\ Heap_2(\alpha_2) &= \{\, \alpha_2 \,\} \end{aligned}$$

Queues

Qs_1:

$Qs_1(\alpha_1)$	$Qs_1(\alpha_2)$	$Qs_1(\alpha_3)$
$\varnothing$	$\varnothing$	$\varnothing$

Qs_2:

$Qs_2(\alpha_1)$	$Qs_2(\alpha_2)$	$Qs_2(\alpha_3)$
APP(ω_6)	APP($\omega_7, \omega_6, \omega_7$)	$\varnothing$

Counter Tables

CT_1:

	α_1	α_2	α_3
α_1	5	5	0
α_2	1	2	1
α_3	0	0	0
ω_1	50	50	
ω_2	3	3	
ω_3	10	10	0
ω_4			
ω_5	10	40	30
ω_6			
ω_7	10	10	
ω_8	10	10	

CT_2:

	α_1	α_2	α_3
α_1	**6**	5	
α_2	1	**4**	1
α_3			**2**
ω_1	50	50	
ω_2	3	3	
ω_3	10	10	0
ω_4			
ω_5	10	40	**28**
ω_6			**2**
ω_7	**12**	10	**1**
ω_8	10	**12**	**1**

Figure 2. Heaps, message queues and counter tables.

DEFINITION 1 (Addresses and Owners). *We require enumerable sets $\mathbb{S}_{all}$, $\mathbb{S}_{obj}$, and $\mathbb{S}_{act}$, and a function*

$$Owner \quad : \quad \mathbb{S}_{all} \to \mathbb{S}_{act}$$

such that

$$\begin{array}{lcl} Addr & = & \mathbb{S}_{obj} \uplus \mathbb{S}_{act} \\ \iota & \in & \mathbb{S}_{all} \\ \alpha & \in & \mathbb{S}_{act} \\ \omega & \in & \mathbb{S}_{obj} \end{array}$$

and

$$\forall \alpha \in \mathbb{S}_{act}. Owner(\alpha) = \alpha$$

A runtime configuration consists of a per-actor heap, a per-actor queue of messages, and a per-actor counter table. In order to argue soundness we need to model the heap. We do not need to distinguish the class of objects, or the contents of their fields. All we need to model is the set of addresses which are reachable from a given address. In the current paper we over-approximate this information, and our model only represents the set of addresses reachable from a given actor. In fact, we expect that a *valid heap for an actor contains a superset of the addresses reachable from an actor*, but we do not model reachability. We will give a finer grained model in further work.

The message queue is a sequence of messages, where the order matters. Messages are either Pony-level messages, or ORCA-specific messages. Our protocol is not concerned with the exact Pony-level messages sent, but it is concerned with the addresses these may contain. Pony-level messages are $\mathtt{APP}(\iota^\star)$. The ORCA-specific messages $\mathtt{DEC}(\iota, k)$ and $\mathtt{INC}(\iota, k)$ require the actor to change its reference count to ι accordingly. The counter table gives the reference count for a given address, reflecting references from other actors or messages in queues.

DEFINITION 2 (Runtime Configurations).

$$\begin{aligned} RunTimeCfg &= \mathbb{S}_{act} \to \ (\ \mathcal{P}(\mathbb{S}_{all}) \ \times Msg^\star \ \times (\mathbb{S}_{all} \to \mathbb{Z})) \\ Msg &::= \mathtt{APP}(\iota *) \ | \ \mathtt{INC}(\iota, k) \ | \mathtt{DEC}(\iota, k) \\ k &\in \mathbb{N} \\ \mathcal{K} &\in RunTimeCfg \\ q &\in Msg^\star \end{aligned}$$

In the remainder we use the following abbreviations

- $Heap_{\mathcal{K}}(\alpha) = \mathcal{K}(\alpha)\downarrow_1$
- $Queue_{\mathcal{K}}(\alpha) = \mathcal{K}(\alpha)\downarrow_2$
- $Message_{\mathcal{K}}(\alpha, j) = \mathcal{K}(\alpha)\downarrow_2 (j)$
- $RC_{\mathcal{K}}(\alpha, \iota) = \mathcal{K}(\alpha)\downarrow_3 (\iota)$

And we require the following well-formed conditions:

WF1. $RC_{\mathcal{K}}(\alpha, \iota) \geq 0$
WF2. $\omega \in Heap_{\mathcal{K}}(\alpha) \implies Owner(\omega) \in Heap_{\mathcal{K}}(\alpha)$

EXAMPLE 2. *If we consider the ownership and references diagram from Figure 1, then $Heap_1$ defined in Figure 2 is valid. Similarly, $Heap_2$ from Figure 2 is also valid. $Heap_1$ represents a possible heap before garbage collection, while $Heap_2$ represents a possible heap after all actors have performed all possible garbage collection steps. On the other hand, $Heap_3$ defined below*

$$\begin{array}{lcl} Heap_3(\alpha_1) & = & \{ \alpha_1, \alpha_2, \omega_1, \omega_2, \omega_3 \} \\ Heap_3(\alpha_2) & = & \{ \alpha_1, \alpha_2, \omega_1, \omega_2, \omega_3 \} \\ Heap_3(\alpha_3) & = & \{ \omega_6, \alpha_3 \} \end{array}$$

is invalid because even though ω_6 is in the heap of α_3, and ω_5 is reachable from ω_6, the heap of α_3 does not contain ω_5.

EXAMPLE 3. *In Figure 2 we show different reference count tables, CT_1 and CT_2. We thus have possible configurations:*

$$\begin{aligned} \mathcal{K}_1 &= (Heap_1, Qs_1, CT_1) \\ \mathcal{K}_2 &= (Heap_1, Qs_2, CT_1) \\ \mathcal{K}_3 &= (Heap_1, Qs_2, CT_2) \end{aligned}$$

4.3 Well-formed Configurations

In this section we define when a configuration is well-formed. Given that the actor only uses the values in its counter table to decide when to garbage collect an object, the counter for that specific object should be consistent with the reference count of other actors with references to it and the messages in all the queues of a configuration containing a reference to it. In other words, a configuration is well-formed, if it satisfies the conditions introduced in the previous section, an in addition:

WF3. If there is a message in some queue containing an address ι, then the local reference count of ι is greater than zero.

WF4. If an actor α can reach an address which does not own, then both the owner's (local) reference count and α's (foreign) reference count for that object are greater than zero.

WF5. The sum of the local reference count and the increment-decrement count for an address is always the same as the sum of the total foreign reference count and total application message count for that address.

WF6. For any prefix of any actor α's queue, if we add to the local reference count for ι the sum of weights of increment and decrement messages for ι, and we subtract the number of application messages that contain ι , the result is greater or equal to zero.

Given this, we define five derived counts. We define now the first four derived counts. For any address ι and a global configuration, $\mathcal{K}$, we define:

1. The *local reference count* of an address ι, $LRC_{\mathcal{K}}(\iota)$, gives the reference count for ι in the counter table of its owner.
2. The *foreign reference count* of an address, $FRC_{\mathcal{K}}(\iota)$, is the sum of the reference counts for ι from "outside", that is, from all actors different from the owner.
3. The *increment-decrement count* of an address, $IDC_{\mathcal{K}}(\iota)$, is the sum of weighted references to ι in the current in-flight `INC`, `DEC` messages in the queue of the owning actor.
4. The *application message count* of an address, $AMC_{\mathcal{K}}(\iota)$, is the number of Pony-level messages which contain ι, addresses owned by ι or addresses which are reachable from ι.

DEFINITION 3. *For a configuration $\mathcal{K}$, and address ι, we define the functions*

$$\begin{array}{rcl} FRC, \ LRC, \ AMC : RunTimeCfg \ \times \ \mathbb{S}_{all} & \to & \mathbb{N} \\ IDC : \ RunTimeCfg \ \times \ \mathbb{S}_{all} & \to & \mathbb{Z} \end{array}$$

as follows:

1. $LRC_{\mathcal{K}}(\iota) = RC_{\mathcal{K}}(Owner(\iota), \iota)$
2. $FRC_{\mathcal{K}}(\iota) = \sum_{\alpha \neq Owner(\iota)} RC_{\mathcal{K}}(\alpha, \iota)$
3. $IDC_{\mathcal{K}}(\iota) = \sum_j Weight(\iota, Message_{\mathcal{K}}(Owner(\iota), j))$
4. $AMC_{\mathcal{K}}(\iota) = \#\{ \ (\alpha, j) \mid Message_{\mathcal{K}}(\alpha, j) = \mathtt{APP}(args)$
   $\wedge \ \iota \in \mathtt{APP}(args) \ \}$

DEFINITION 4. *An address ι is contained in a Pony-level message if and only if it is one of the arguments, or is the owner of one of the arguments, or if it can be reached from one of the arguments,* *or is the owner of an object reachable from one of the arguments.*

$$\iota \in msg \Leftrightarrow msg = \mathtt{APP}(args) \wedge (\exists \iota', \iota''.\iota' \in \mathtt{APP}(args) \wedge$$
$$\iota'' \textit{ is reachable from } \iota' \wedge (\iota = \iota'' \vee \iota = Owner(\iota'')))$$

Note that every address is reachable from itself.

EXAMPLE 4. *In the heap from Figure 1, we have that*

- $\alpha_1, \alpha_2, \omega_1, \omega_2, \omega_3 \in \mathtt{APP}(\omega_3)$,
- $\alpha_2, \omega_5 \in \mathtt{APP}(\omega_5)$,
- $\alpha_2, \alpha_3, \omega_5, \omega_6 \in \mathtt{APP}(\omega_6)$,
- *and* $\alpha_1, \alpha_2, \omega_7, \omega_8 \in \mathtt{APP}(\omega_7)$.

DEFINITION 5. *The weight of a message, regarding an address ι, is given by the function*
$Weight : Addr \times Msg \to \mathbb{Z}$

$$Weight(\iota, msg) = \begin{cases} k & \textit{if } msg = \mathtt{INC}(\iota, k) \\ -k & \textit{if } msg = \mathtt{DEC}(\iota, k) \\ 0 & \textit{otherwise} \end{cases}$$

EXAMPLE 5. *Consider configuration $\mathcal{K}_1$ and in particular its object ω_1, then:*

$$FRC_{\mathcal{K}_1}(\omega_1) = 50 \qquad LRC_{\mathcal{K}_1}(\omega_1) = 50$$
$$IDC_{\mathcal{K}_1}(\omega_1) = 0 \qquad AMC_{\mathcal{K}_1}(\omega_1) = 0$$

On the other hand, in $\mathcal{K}_2$, and considering in particular their objects $\omega_5, \omega_6, \omega_7$ and ω_8, we have that:

$$AMC_{\mathcal{K}_2}(\omega_6) = AMC_{\mathcal{K}_2}(\omega_5) = 2$$
$$AMC_{\mathcal{K}_2}(\omega_7) = AMC_{\mathcal{K}_2}(\omega_8) = 1$$
$$AMC_{\mathcal{K}_2}(\alpha_1) = 1 \quad AMC_{\mathcal{K}_2}(\alpha_2) = 2 \quad AMC_{\mathcal{K}_2}(\alpha_3) = 2$$

and since we have no INC/DEC *messages,* $\forall \iota. IDC_{\mathcal{K}_2}(\iota) = 0$.

We now define the fifth derived count, which we call *pending changes count*, in short PCC. It sums the weights of INC and DEC messages for ι in some queue and subtracts the number of pending application messages containing ι in that queue. This last count summed with the local reference count of ι will be the new local reference count for ι in a configuration where all the messages of q have been consumed by α.

DEFINITION 6. *The* pending changes counter *of an address ι, in an actor α, $PCC_{\mathcal{K}}(\iota, \alpha, q)$, is defined as follows:*
$PCC : RunTimeCfg \times \mathbb{S}_{all} \times \mathbb{S}_{act} \times Msg^{\star} \to \mathbb{Z}$

$$PCC_{\mathcal{K}}(\iota, \alpha, q) = \sum_j Weight(\iota, q(j)) - \begin{cases} 0 & \textit{if } Owner(\iota) \neq \alpha \\ \#\{j \mid \iota \in q(j)\} & \textit{otherwise} \end{cases}$$

EXAMPLE 6. *Consider queues,* $q_1 = \mathtt{INC}(\omega_2, 1), q_2 = \mathtt{APP}(\omega_2)$, *then we have* $PCC(\omega_2, \alpha_1, q_1) = 1$ *and* $PCC(\omega_1, \alpha_1, q_1) = 0$. *Also,* $PCC(\omega_2, \alpha_1, q_2) = PCC(\omega_1, \alpha_1, q_2) = -1$. *Therefore,* $PCC(\omega_2, \alpha_1, q) = 0$ *but* $PCC(\omega_1, \alpha_1, q) = -1$.

We are now able to define when a configuration is well-formed.

DEFINITION 7. *A configuration $\mathcal{K}$ is well formed iff* $\forall \alpha, \iota. \forall q_1, q_2 \in Msg^{\star}$

WF3. $\iota \subseteq Message_{\mathcal{K}}(\alpha, j) \implies LRC_{\mathcal{K}}(\iota) > 0$
WF4. $\iota \in Heap_{\mathcal{K}}(\alpha) \wedge \alpha \neq Owner(\iota) \implies$
$\qquad RC_{\mathcal{K}}(\alpha, \iota) > 0 \wedge LRC_{\mathcal{K}}(\iota) > 0$
WF5. $LRC_{\mathcal{K}}(\iota) + IDC_{\mathcal{K}}(\iota) = FRC_{\mathcal{K}}(\iota) + AMC_{\mathcal{K}}(\iota)$
WF6. $q_1 :: q_2 = Queue_{\mathcal{K}}(\alpha) \implies$
$\qquad LRC_{\mathcal{K}}(\iota) + PCC_{\mathcal{K}}(\iota, \alpha, q_1) \geq 0$

DEFINITION 8. $\iota \subseteq msg \Leftrightarrow \iota \in msg \vee msg = \mathtt{INC}(\iota, \_) \vee msg = \mathtt{DEC}(\iota, \_)$

The condition **WF6** ensures that whenever consuming a message makes the actor decrease its count for some owned address (as we will see later), this count will not become negative.

EXAMPLE 7. *It is easy to check that $\mathcal{K}_1$ is well-formed. On the other hand, $\mathcal{K}_2$ is not a well-formed configuration as* ***WF3*** *and* ***WF5*** *do not hold.* ***WF3*** *does not hold because* $Queue_{\mathcal{K}_2}(\alpha_1) = \mathtt{APP}(\omega_6)$ *even though* $RC_{\mathcal{K}_2}(\alpha_3, \omega_6) = 0$. *With respect to* ***WF5***, *the AMC for the addresses $\alpha_1, \alpha_2, \alpha_3, \omega_5, \omega_6, \omega_7, \omega_8$ are no longer 0. For instance:*

$$AMC_{\mathcal{K}_2}(\omega_7) = AMC_{\mathcal{K}_2}(\alpha_1) = AMC_{\mathcal{K}_2}(\omega_8) = 1$$

A possible way to change $\mathcal{K}_2$ in order to make it well-formed could be by changing to $\mathcal{K}_3$, i.e, we replace CT_1 by CT_2.

5. Pony-ORCA: "killing" them safely

Pony-ORCA allows an actor to collect its own objects without checking information from other actors or from any queue. That is, an actor does not need to check any other heap, nor does it need to examine any queue, including its own, in order to determine whether an object is collectable or not. In this section we argue why it is safe to collect objects under these conditions, and what actions the protocol needs to perform in order to preserve well-formedness.

Pony-ORCA runs on top of Pony type system [6], which guarantees that there will be no race conditions even though it does not use any locking or synchronisation mechanism other than the messaging system.

In Section 5.1 we show why it is sound to collect any actor or object whose local reference count is 0. In Section 5.2 we describe what actions need to be taken when the configuration changes, i.e. upon message send, message receipt, or addresses becoming unreachable. We show a garbage collection scenario in Section 5.3. In Section 5.4 we discuss the role of causality. And in Section 5.5 we discuss the absence of race conditions in the system.

5.1 Application to Garbage collection

Here we will argue why it is sound to garbage collect an actor or object when its local reference count is 0. We will use the well-formedness conditions from sections 4.2 and 4.3 to show that under those circumstances the object or the actor is globally unreachable.

DEFINITION 9 (Globally unreachable objects and actors). *An address ι is* globally unreachable *if and only if it does not appear in any actor's heap (i.e. $\forall \alpha. \iota \notin Heap_{\mathcal{K}}(\alpha)$) and does not appear in any Pony level message (i.e. $\forall \alpha, j. \iota \notin Message_{\mathcal{K}}(\alpha, j)$).*

5.1.1 Collecting objects

DEFINITION 10. *An object ω is* collectable *by an actor α, iff*

C1. α *owns* ω, *i.e.,* $\alpha = Owner(\omega)$.
C2. α *has no path leading to* ω, *i.e.,* $\omega \notin Heap_{\mathcal{K}}(\alpha)$.
C3. α*'s (local) reference count for* ω *is 0, i.e.,* $RC_{\mathcal{K}}(\alpha, \omega) = 0$

These three requirements are local and therefore the actor can garbage collect without needing to consult other actors or examine any queues, including its own. We now argue that in any well-formed configuration a collectable object is globally unreachable.

1. From *C1, C3* and Definition 3 we know that the local reference count for ω is 0, i.e. $LRC_{\mathcal{K}}(\omega) = 0$.
2. From *C1, C3* and **WF3** we know that the counts for increment/decrement and application messages are 0, i.e., $IDC_{\mathcal{K}}(\omega) = 0$ and $AMC_{\mathcal{K}}(\omega) = 0$.
3. From 1, 2 and **WF5** we know that the foreign reference count of ω is 0, i.e. $FRC_{\mathcal{K}}(\omega) = 0$.
4. From *C2* we obtain that ω is not reachable from α.
5. From 3 and **WF4** we obtain that ω is not reachable from any further actor $\alpha' \neq \alpha$.
6. From 2, we obtain that ω is not in any in-flight message.

Therefore, ω is globally unreachable.

5.1.2 Collecting actors

DEFINITION 11. *An actor α is collectable, iff*

C1. Its local reference count for itself is 0, i.e., $RC_{\mathcal{K}}(\alpha, \alpha) = 0$.
C2. Its queue is empty, i.e., $Queue_{\mathcal{K}}(\alpha) = \emptyset$

The argument that a collectable actor is globally unreachable is similar to that for objects with the additional consideration of **WF2**. In addition, we can expand the protocol so as to collect cycles of actors with the local reference count greater than 0 [16].

5.2 Maintaining well-formedness

Language level computations, such as application message sends and receives, and dropping of references, affect the validity of the well-formedness conditions from section 4.3. Therefore, we need to take corrective actions. Here we outline what these actions are.

5.2.1 Sending a Pony level message

Consider that an actor α sends a message $\texttt{APP}(args)$ and ι such that $\iota \in \texttt{APP}(args)$. This action does not modify the heap, therefore **WF1**, **WF2** and **WF4** are preserved. However it does affect the values of $AMC(\iota)$, and thus affects validity of **WF5**. Therefore, for all $\iota \in \texttt{APP}(args)$, the actor α performs the following updates:

1. If $\alpha = Owner(\iota)$ then it will increase $RC(\alpha, \iota)$ by 1.
2. If $\alpha \neq Owner(\iota)$ then it will decrease $RC(\alpha, \iota)$ by 1.

If decreasing the counter would make ι 0 then, before sending the Pony message, an $\texttt{INC}(\iota, k+1)$ message is sent to $Owner(\iota)$ and $RC(\alpha, \iota)$ is set to k, for some $k > 0$. Such an increment message may be sent even if $RC > 1$, to allow future sends of ι without requiring additional increment messages.

These actions restore **WF5** and do not affect validity of **WF3** and **WF6**.

5.2.2 Receiving a Pony level message

The actions taken to preserve the well-formedness of a configuration are similar to those for sending a message containing ι, taking into consideration that when a message is received, it is removed from the queue. We consider an actor α that receives a message $\texttt{APP}(args)$. For all addresses ι such that $\iota \in \texttt{APP}(args)$ the receiving actor performs the following actions, which are essentially the opposite to those in Section 5.2.1:

1. If $\alpha = Owner(\iota)$ then it will decrease $RC(\alpha, \iota)$ by 1.
2. If $\alpha \neq Owner(\iota)$ then it will increase $RC(\alpha, \iota)$ by 1.

The address ι is added to the heap of α. Validity of **WF4** is trivially preserved in all four cases. Moreover, **WF6** from the previous configuration guarantees preservation of **WF1** in the new configuration.

5.2.3 Receiving an ORCA specific message

When an actor receives a message $\texttt{INC}(\iota, k)$ or $\texttt{DEC}(\iota, k)$ then, by construction, it is the owner of ι. Therefore,

1. When α receives $\texttt{INC}(\iota, k)$, it increments $RC(\alpha, \iota)$ by k.
2. When α receives $\texttt{DEC}(\iota, k)$, it decrements $RC(\alpha, \iota)$ by k.

Condition **WF6** guarantees that **WF1** and **WF6** are preserved. The other conditions are not affected.

5.2.4 Tracing and collecting

Part of the protocol is the tracing mechanism that actors use to determine whether its objects are reachable or not. We now show the tracing algorithm step-by-step:

1. All owned objects are marked unreachable.
2. All unowned objects with a foreign reference count greater than zero are marked as unreachable.
3. Tracing occurs from the actor's fields only, marking objects reachable, whether they are owned or not.
4. All owned objects with the local reference count greater than zero are marked as reachable.
5. Owned objects that are locally unreachable and have $LRC = 0$ are collected.
6. Decrement messages are sent for unowned objects that are unreachable, and their FRC is set to 0.

Soundness of step 5 has been discussed in Section 5.1. Step 6 trivially preserves **WF1** and **WF2**. Moreover, when the actor sends the decrement message, $\texttt{DEC}(\iota, k)$, where ι is the address to be collected and k is the former reference count of the actor ι, both $FRC(\iota)$ and $LRC(\iota)$ will decrease by k and **WF5** is preserved.

5.3 A Garbage Collection Scenario

We will elucidate how the garbage collector works by means of a scenario where α_3 runs garbage collection on $Heap_1$, counter table CT_1. Then α_2 runs garbage collection, followed by α_1, which again is followed by α_2. In the end, we will have collected all the globally inaccessible actors or objects, and we will be left with a heap as in $Heap_2$. The garbage collector considers an object or actor as *locally reachable* from some actor α if there exists a path to that object or actor from α. We explain this now in more detail:
1st Step: actor α_3 performs garbage collection on heap $Heap_1$ and counter table CT_1. The references α_2, α_3, ω_5, ω_6 are locally unreachable. Since the actor owns ω_6 and since ω_6's RC entry is 0, it will collect ω_6. It cannot collect α_2 nor ω_5, but it will send to α_2 a message to decrement its RC entry for α_2 by 1, and its RC entry for ω_5 by 30. As a result we will now have $RC(\alpha_3, \alpha_2)$=0, and $RC(\alpha_2, \alpha_2)$=1, and $RC(\alpha_2, \omega_5)$=10. Then, because $RC(\alpha_3, \alpha_3)$=0, actor α_3 is collected, and its heap discarded.
2nd Step: actor α_2 performs garbage collection. The references α_1, ω_1, ω_2, ω_3, ω_5, ω_7 and ω_8 are locally unreachable. Of these addresses, it owns ω_3, ω_5, and ω_8. The RC entry for all three objects is not 0; therefore they will not be collected, and the heap stays unmodified. The actor will set its RC entry for α_1, ω_1, ω_2, and ω_7 to 0, and will send to α_1 a message to decrement its RC entry for $\alpha_1, \omega_1, \omega_2$, and ω_7 by 5, 50, 3, and 10. When α_1 consumes this message, the RC-table will look as follows:

CT_3 :

	α_1	α_2
α_1	0	0
α_2	1	1
ω_3	10	10
ω_5	10	10
ω_8	10	10

3rd Step: actor α_1 performs garbage collection. The references ω_4, ω_5, ω_7 and ω_8 are locally inaccessible. Of these addresses, the actor owns ω_4 and ω_7, and as their RC entry is 0, both objects will be collected. We will now have $Heap(\alpha_1) = \{\ \alpha_1, \alpha_2, \omega_1, \omega_2, \omega_3\ \}$. The actor will also send to α_2 messages to decrement its RC entry for ω_5 and ω_8 by 10 and 10. When this message is consumed, we will have $RC(\alpha_1, \omega_5) = 0 = RC(\alpha_1, \omega_8)$, and also $RC(\alpha_2, \omega_5) = 0 = RC(\alpha_2, \omega_8)$.
4th Step: actor α_2 performs garbage collection. As the references ω_5, ω_8 are locally inaccessible, and as their RC has value 0, the actor will collect ω_5, ω_8. We now have $Heap(\alpha_2) = \{\ \alpha_2\ \}$.

Provided that the actor α_1 is not blocked, any further garbage collection steps will make no difference, unless, of course, the heaps or queues were to change. If the queue of α_1 becomes empty then it can be collected. Our heap will have the contents as in $Heap_2$. The contents of the RC-table is as in CT_4, shown below.
CT_4 :

	α_1	α_2
α_1	0	0
α_2	1	1
ω_3	10	10

5.4 Causality

Causality is required in Pony-ORCA in order to maintain **WF4** and **WF6**. Consider namely a situation where an actor α whose RC entry for an object ω is 1, sends to α', the owner of ω, a message containing ω, then ω becomes locally inaccessible in α and then α performs garbage collection. In this case, α will send to α' a message $\mathtt{INC}(\omega, k)$ for some value for $k > 0$, followed by $\mathtt{APP}(\omega)$, followed by $\mathtt{DEC}(\omega, k-1)$.

Causality considers that $\mathtt{INC}(\omega, k)$ is a cause of $\mathtt{APP}(\omega)$, and that $\mathtt{APP}(\omega)$ is a cause of $\mathtt{DEC}(\omega, k-1)$, and therefore guarantees that they will arrive at α' in that order. Therefore, if the value of $RC(\alpha, \omega)$ was k', then upon consumption of these steps it would become $k' + k$, then $k' + k - 1$, and k', and thus stay positive.

However, if we allowed the messages to overtake each other, and if we allowed the delivery to be $\mathtt{APP}(\omega)$, $\mathtt{DEC}(\omega, k-1)$, $\mathtt{INC}(\omega, k)$ then the values would be $k' - 1$ (thus possibly breaking **WF4**), $k' - k + 1$ (thus possibly breaking **WF6**) and k'.

5.5 Absence of race conditions in Pony-ORCA

To ensure the absence of race conditions we need to be certain that tracing and garbage collection in one actor cannot interfere with tracing, garbage collection, or normal behaviour in another actor. We guarantee this through the type system of the underlying language [6], which ensures that whenever an actor has a readable path to an object, no other actor can write to it. Therefore, by creating a different tracing function for each class according to the read capabilities of each of the fields in that class, we ensure that tracing does not interfere with any other actor's activity. And since garbage collection only removes globally unreachable actors or objects, we also ensure that garbage collection does not interfere with any other actor's activity.

6. Discussions

The message overhead of our approach is low. Reference counts (RC) are not tied to heap references but rather to messages, which means no Garbage Collection (GC) messages are required for heap mutation. When receiving a message, no GC messages are generated because the receiving actor can simply decrease its RC for objects it owns and increase its RC for other actors and objects. When sending a message, GC increment messages are only generated when the sending actor has an RC of 1 for a sent actor or object it does not own; otherwise, the sending actor simply increases its RC for objects it owns and decreases its RC for other actors and objects. In addition, when sending does require a GC increment message, the cost is amortised by creating a large RC, allowing many future sends of an object without additional GC increment messages. This is coupled with the ability to send GC increment messages that refer to many objects, which means that a message send that requires GC increment messages generates at most one GC increment message for any given actor, which also reduces message overhead.

Because the disappearance of a reference from an actor's reachable heap is detected only during GC, GC decrement messages are generated lazily. These decrement messages are also combined, as for GC increment messages, such that at most one GC decrement message is generated for any given actor. This reduces the GC message bounds from $O(unreachable)$ to $O(Owner(unreachable))$.

Causal message delivery allows us to never require an acknowledgement for a GC message. This means that there is no GC message related latency due to requiring a round trip.

Our approach only runs a GC pass on an actor when that actor is not executing a behaviour. As a result, GC only occurs when there is no stack. This means there is no requirement for a stack map or a stack crawler.

We do not require any form of read or write barrier [35]. This is achieved by combining a data-race free type system with handling all RC changes when sending and receiving messages.

While this does mean data structures are traced when they are sent and received, it eliminates the need to treat objects with a local reference count greater than as GC roots. Instead, such object are simply marked as in-use without being traced. This is a key element of the system: it allows such objects to be garbage collected safely by the owning actor even when some other actor is in the process of mutating them. This allows heaps to be garbage collected entirely independently, as mutation in another actor does not affect GC. We have not modelled this in our current paper, as we do not model the tracing of structures, but will describe this in more detail in further work.

The combination of having no stack when garbage collecting and garbage collecting each actor's heap independently means safepoints [24] are not required. Any actor can GC its own heap without waiting for a safepoint to be reached.

Independent actor heap collection functions as both a concurrent GC mechanism (actors can GC concurrently) and an incremental GC mechanism (actors can GC separately). This allows optimisations to both the tracing algorithm and the memory allocator.

We use a mark-and-don't-sweep collector [23] that keeps mark bits, rather than pointers, in the heap data structure. By moving the mark bit out of the object, object contents are never written to during GC, and unreachable objects are never traced. This minimises cache pollution, eliminates page misses on unreachable objects, and reduces the trace phase to $O(reachable)$ rather than $O(reachable + unreachable)$.

This approach also results in a memory allocator that works like a bump allocator [23]. No best/first-fit search is required, and free list maintenance is handled by page (or group of pages) rather than by object. Memory allocation cost is amortised to the cost of a single find-first-set bit operation.

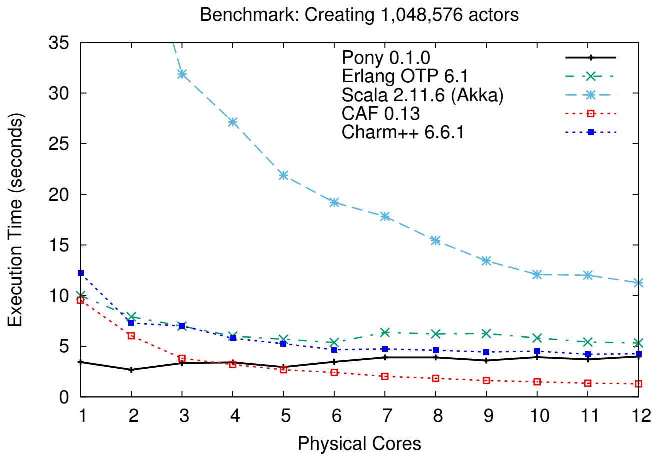

Figure 3. Actor creation performance

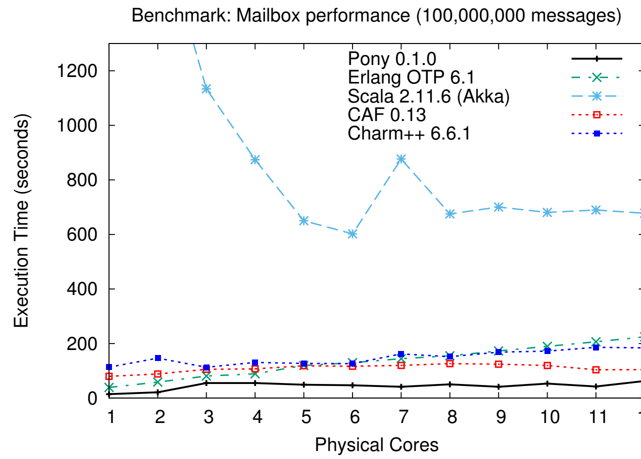

Figure 4. Mailbox performance

These optimisations are made possible because independent actor heap collection guarantees mutation doesn't affect either tracing or allocation semantics.

7. Conclusions and Further Work

Conclusion. Pony is a concurrent and distributed object-oriented, actor-based programming language which supports (passive) objects. One of the features of Pony is its message-based garbage collection mechanism. This allows the collection of dead actors, as presented in [17]. We have presented the garbage collection for passive objects in Pony. We formally define what constitutes a runtime configuration, with respect to the data structures that the garbage collector maintains, and in particular what constitutes a well-formed configuration which allows the deallocation of passive objects. Moreover, we informally describe how to keep a runtime configuration consistent when certain operations are executed.

This protocol was implemented in the Pony compiler which was benchmarked against other actor-model languages with the CAF [13] benchmark suite [1] and against MPI with HPC Challenge LINPACK GUPS [2]. We now report the results of these experiments taken from [6]. Benchmarking was done on a 12-core 2.3 GHz Opteron 6338P with 64 GB of memory across 2 NUMA nodes. The results shown are the average of 100 runs.

The first benchmark, shown in Figure 3 shows the performance of creating large numbers of actors. In particular, the performance of garbage collecting the actors [17] and passive objects. It shows better results then the other existing systems (except CAF which is not garbage collected).

In Figure 4, we can see that the performance of a highly contended mailbox, where additional cores tend to degrade performance. The results of the third experience, illustrated in 5 shows performance of a mixed case, where a heavy message load is combined with brute force factorisation of large integers. The last figure 6 shows a benchmark that is not tailored for actors: we take the GUPS benchmark from high-performance computing, which tests random access memory subsystem performance, and demonstrate that Pony's implementation is significantly faster than the highly optimised MPI implementation.

Further work. We plan to give a full formal model, including a model of the heap and reachability, and proof of soundness. We want to investigate in how far we can weaken the requirements for causality, and applications to further language features such as futures or promises.

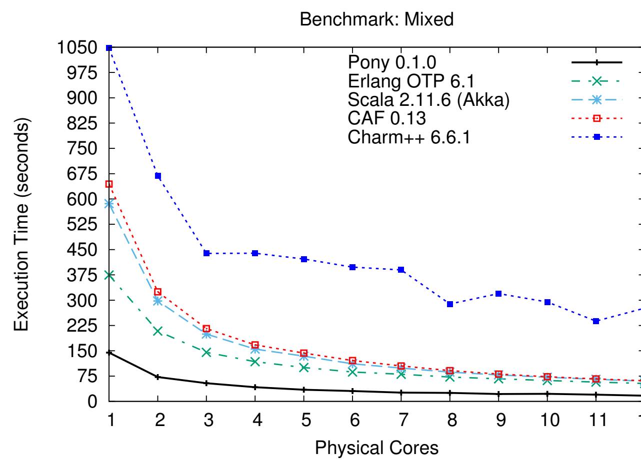

Figure 5. Mixed case performance

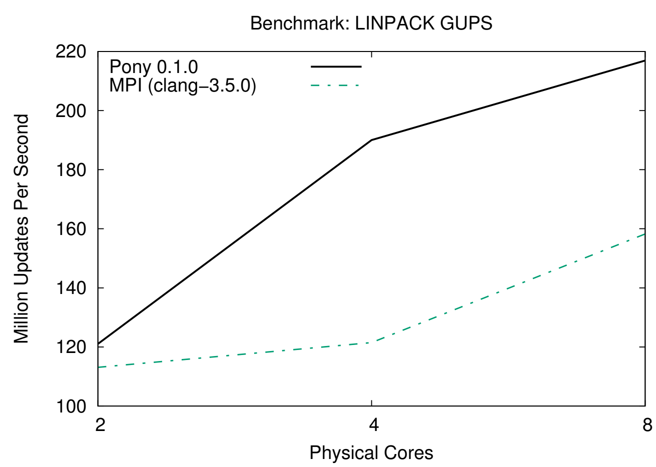

Figure 6. LINPACK GUPS performance

Acknowledgements We thank Alexander Summers, Tobias Wrigstad, Susan Eisenbach and Tim Wood for extensive feedback. This work was funded mainly by Causality Ltd, and also by the EU project UpScale FP7-612985.

References

[1] https://github.com/actor-framework/benchmarks/. URL `https://github.com/actor-framework/benchmarks/`.

[2] http://icl.cs.utk.edu/hpcc/.

[3] ActorFoundry. Actorfoundry. http://osl.cs.illinois.edu/software/actor-foundry/.

[4] G. Agha and C. Hewitt. Concurrent programming using actors. In *Object-oriented concurrent programming*, pages 37–53. MIT Press, 1987.

[5] akka. Akka - scala actor library. http://akka.io/.

[6] A. anonymous as under submission. Deny capabilities for safe, fast actors. URL `http://www.doc.ic.ac.uk/~scd/fast-cheap.pdf`.

[7] J. Armstrong. A history of erlang. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 6–1. ACM, 2007.

[8] D. F. Bacon and V. Rajan. Concurrent cycle collection in reference counted systems. In *ECOOP 2001—Object-Oriented Programming*, pages 207–235. Springer, 2001.

[9] H. G. Baker. Minimizing reference count updating with deferred and anchored pointers for functional data structures. *ACM Sigplan Notices*, 29(9):38–43, 1994.

[10] A. P. Black, N. C. Hutchinson, E. Jul, and H. M. Levy. Object structure in the emerald system. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86), Portland, Oregon, Proceedings.*, pages 78–86, 1986. . URL `http://doi.acm.org/10.1145/28697.28706`.

[11] S. Blessing. A string of ponies: Transparent distributed programming with actors. Master's thesis, Imperial College London, 2013.

[12] M. Carpen-Amarie, P. Marlier, P. Felber, and G. Thomas. A performance study of java garbage collectors on multicore architectures. In *PMAM '15*. ACM, 2015.

[13] D. Charousset, T. C. Schmidt, R. Hiesgen, and M. Wählisch. Native actors: a scalable software platform for distributed, heterogeneous environments. In *Proceedings of the 2013 workshop on Programming based on actors, agents, and decentralized control*, pages 87–96. ACM, 2013.

[14] D. Clarke, T. Wrigstad, J. Östlund, and E. Johnsen. Minimal ownership for active objects. *Programming Languages and Systems*, pages 139–154, 2008.

[15] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA '98*, pages 48–64. ACM, 1998. ISBN 1-58113-005-8. . URL `http://doi.acm.org/10.1145/286936.286947`.

[16] S. Clebsch and S. Drossopoulou. Fully concurrent garbage collection of actors on many-core machines. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages and applications*, pages 553–570. ACM, 2013.

[17] S. Clebsch and S. Drossopoulou. Fully concurrent garbage collection of actors on many-core machines. *SIGPLAN Not.*, 48(10):553–570, Oct. 2013.

[18] A. de Araújo Formiga and R. D. Lins. A new architecture for concurrent lazy cyclic reference counting on multi-processor systems. *J. UCS*, 13(6):817–829, 2007.

[19] F. Dehne and R. D. Lins. Distributed cyclic reference counting. In *Parallel and Distributed Computing Theory and Practice*, pages 95–100. Springer, 1994.

[20] Elixir. Elixir. http://elixir-lang.org/.

[21] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2): 202–220, 2009.

[22] C. Hewitt, P. Bishop, and R. Steiger. Artificial intelligence a universal modular actor formalism for artificial intelligence, 1973.

[23] R. Jones and R. D. Lins. Garbage collection: algorithms for automatic dynamic memory management. 1996.

[24] R. Jones, A. Hosking, and E. Moss. *The garbage collection handbook: the art of automatic memory management*. Chapman & Hall/CRC, 2011.

[25] R. E. Jones and R. D. Lins. Cyclic weighted reference counting without delay. In *PARLE '93*, pages 712–715. Springer-Verlag, 1993.

[26] N. C. Juul and E. Jul. Comprehensive and robust garbage collection in a distributed system. In *IWMM 92*, pages 103–115, 1992.

[27] R. D. Lins. Lazy cyclic reference counting. *J. UCS*, 9(8):813–828, 2003.

[28] L. Moreau and J. Duprat. A construction of distributed reference counting. *Acta Informatica*, 37(8):563–595, 2001.

[29] L. Moreau, P. Dickman, and R. Jones. Birrell's distributed reference listing revisited. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(6):1344–1395, 2005.

[30] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *ECOOP'98*, pages 158–185. Springer, 1998.

[31] F. Pizlo, J. M. Fox, D. Holmes, and J. Vitek. Real-time java scoped memory: Design patterns and semantics. In *(ISORC 2004)*, pages 101–110, 2004.

[32] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for java. In *ECOOP 2008–Object-Oriented Programming*, pages 104–128. Springer, 2008.

[33] T. Van Cutsem. Ambient references: Object designation in mobile ad hoc networks. *Programming Technology Lab, Faculty of Sciences, Vrije Universiteit Brussel*, 2008.

[34] C. Varela and G. Agha. Programming dynamically reconfigurable open systems with salsa. *ACM SIGPLAN Notices*, 36(12):20–34, 2001.

[35] W.-J. Wang. Conservative snapshot-based actor garbage collection for distributed mobile actor systems. *Telecommunication Systems*, 52(2): 647–660, 2013. ISSN 1018-4864.