Orca: GC and Type System Co-Design for Actor Languages

SYLVAN CLEBSCH, Microsoft Research Cambridge, United Kingdom
JULIANA FRANCO, Imperial College London, United Kingdom
SOPHIA DROSSOPOULOU, Imperial College London, United Kingdom
ALBERT MINGKUN YANG, Uppsala University, Sweden
TOBIAS WRIGSTAD, Uppsala University, Sweden
JAN VITEK, Northeastern University, United States of America

Orca is a concurrent and parallel garbage collector for actor programs, which does not require any stop-the-world steps, or synchronisation mechanisms, and which has been designed to support zero-copy message passing and sharing of mutable data. Orca is part of the runtime of the actor-based language Pony. Pony's runtime was co-designed with the Pony language. This co-design allowed us to exploit certain language properties in order to optimise performance of garbage collection. Namely, Orca relies on the absence of race conditions in order to avoid read/write barriers, and it leverages actor message passing for synchronisation among actors. This paper describes Pony, its type system, and the Orca garbage collection algorithm. An evaluation of the performance of Orca suggests that it is fast and scalable for idiomatic workloads.

CCS Concepts: •**Software and its engineering** → **Garbage collection; Concurrent programming languages;** *Runtime environments;* •**Theory of computation** → *Type structures;*

Additional Key Words and Phrases: actors, messages

ACM Reference format:
Sylvan Clebsch, Juliana Franco, Sophia Drossopoulou, Albert Mingkun Yang, Tobias Wrigstad, and Jan Vitek. 2017. Orca: GC and Type System Co-Design for Actor Languages. *PACM Progr. Lang.* 1, OOPSLA, Article 72 (January 2017), 28 pages.
DOI: 10.1145/3133896

1 INTRODUCTION

Pony is an object-oriented programming language designed from the ground up to support low-latency, highly concurrent applications written in the actor model of computation (Hewitt et al. 1973). The impetus for a new language comes from the authors' experience with the requirements of financial applications, namely a need for *i*) scalable concurrency, from tens to thousands of concurrent components; *ii*) performance approaching that of low-level languages; and *iii*) ease of development and rapid prototyping. Alternatives such as Erlang and Java were considered but performance was felt to be inadequate for the former, and pauses due to garbage collection were a stumbling block for adoption of the latter.

This paper introduces Orca, Pony's concurrent garbage collection algorithm. Orca stands for **O**wnership and **R**eference **C**ounting-based Garbage Collection in the **A**ctor World. It was co-designed with the language's type system to allow actors to share mutable objects and to reclaim memory without any form of synchronisation between actors. Orca's core design principle is to allow each individual actor to collect the objects it allocated without having to wait on, or synchronise with, other actors running in parallel. The approach has its roots in Henriksson's (1998) work on real-time memory management where a collector thread was scheduled during slack time, *i.e.* the portions of a system's schedule during which no high-priority task is running. In actor systems, there is a different notion of slack time: once an actor is done processing a message, it will idle until the next message comes along. Due to the asynchronous nature of actor computation it may be possible for the collector to process multiple actors in parallel without impacting overall application throughput. Garbage collection becomes part of each actor's behaviour and can be properly accounted and scheduled by the scheduler.

In a purely functional actor language such as Erlang (Armstrong 2007), it would be trivial to implement a collector such as ours. When data structures are immutable (*i.e.* cannot be changed), an implementation can simply copy all data exchanged in messages between actors. This ensures that each actor is the root of a disjoint partition of the system's heap. Isolated partitions can be garbage collected in parallel without need for synchronisation. One of the early decisions in the design of Pony was to support mutable data structures and, for efficiency reasons, to implement zero-copy message passing. Mutability introduce challenges in a highly concurrent system. The Pony type system enforces a key property: it ensures that programs are data race-free. Thus, while actors can exchange mutable objects in messages, and these objects are not copied, the type system makes sure that at most one actor at a time is able to *update* any given object. This allows the garbage collector to inspect objects without synchronisation or barriers.

To reclaim an object shared with other actors, the creating actor must be informed when those actors have dropped all references to the shared object. Orca tracks dependencies by deferred reference counts. The meaning of a count larger than one is that at least one, but possibly more, actors other than the object's creator may hold a reference to that object (or has a yet-to-be-processed message containing the object in its mailbox). Orca piggy-backs reference updates on actor message passing, and messages are traced by the collector. This tracing comes at a run-time cost, but does not require synchronisation due to Pony's type system. Also, because reference counts model actors' interest in an object, as opposed to actual reference topologies, cycles are not an issue in Orca, unlike traditional reference counting (*c.f.* Section 5.6). Figure 1 illustrates the fine-grained interleaving of Orca and actor operations running on a four-core system.

This paper describes the implementation of Orca and presents the features of Pony that are needed by the collector. While our presentation is Pony-centric, we believe that Orca could be used in other concurrent languages. In fact, one experiment that is underway is to reuse the Pony run-time system and in particular Orca to implement the Encore programming language (Brandauer et al. 2015), which uses a different type system (Castegren and Wrigstad. 2016) and shared-memory features like futures. Moreover, while the implementation presented here is limited to a single node, we have designed Orca so that it can be extended to a distributed setting.

This paper makes the following contributions:

(1) We present a runtime — language co-design that shows how actor isolation can be leveraged for concurrent garbage collection without deep copying of messages and how race-free tracing on message send/receives can replace write barriers. (Language is discussed in Section 3. GC in Section 4.)
(2) We describe Orca in terms of C-like pseudocode, give the intuitions for the design, sketch invariants which underpin Orca's soundness, and show how these invariants can be used to reason about optimisations to the protocol. (Section 5.)

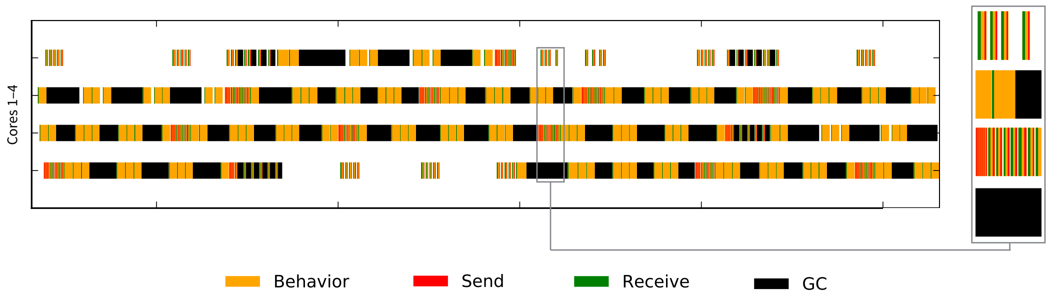


Fig. 1. CPU usage during part of a small Pony program. To the right is a blow-up of a small window. Y-axis: the different core IDs. X-axis the application's timeline (from 5002×10^5 to 5010×10^5 CPU cycles). The diagram demonstrates that while a core may be garbage collecting, other cores may also be garbage collecting, or executing behaviours, or tracing upon send/receipt. (Behaviour=mutator)

(3) We evaluate our implementation on a number of small benchmarks including both small idiomatic programs and synthetic benchmarks aimed at exploring the scalability limits of Orca. (Section 6.)

Our evaluation has the limitations one would expect from a new language, namely few benchmarks. Commercial users of Pony, in the financial sector, are not able to share their code. We are left with synthetic benchmarks we implemented ourselves. Their value is limited and they likely do not cover the full range of interesting behaviours. Nevertheless, they are consistent with our experience and the experience of our customers. To validate our claims of performance and responsiveness, we compare with a version of Pony that does not perform garbage collection and with commercial collectors for Erlang and Java.

Reproducing our results requires a parallel machine (Orca scales up to 64 cores), used in exclusive mode, and with installations of the three languages and the various versions of the GC. In Section 6, we provide links to code that allows the interested reader to build on our work.

2 BACKGROUND

The actor paradigm was first introduced in the 70's in (Hewitt et al. 1973). It models concurrent entities with spawnable actors which execute behaviours (methods) in response to messages from other actors. The increasing levels of parallelism available in modern hardware has rejuvenated interest in this model of computation. Some languages, such as Erlang (Armstrong 2007) and Salsa (Wang 2013), are designed to support actors directly, but this is not strictly necessary. Several successful actor frameworks are implemented as a libraries, for example Akka (2017), ActorFoundry (2017) and ProActive (Caromel and Henrio 2004), which are widely used libraries for Scala and Java. Pony is a language designed from the ground up to support actors. Features of the language, such as its type system, were crafted with an eye towards helping the run-time system — including the garbage collector — improve throughput and reduce pause times. To this end, it leverages the isolation arising naturally in actor systems, important to maintain the single thread of control abstraction (Agha 1986).

Improving responsiveness of concurrent applications is long-standing goal of garbage collection research. Algorithms such as Azul's Pauseless GC (Click et al. 2005) and C4 (Tene et al. 2011) target Java enterprise systems with hundreds to thousands of threads. As Java threads share mutable state, memory barriers are often added to stores to protect the invariants of a collector in the presence of threads operating in parallel. Real-time collectors such as Schism (Pizlo et al. 2010)

```
actor Ring
  var next: (Ring | None)
  new create() ⇒next = None
  new create_set(n: Ring) ⇒next = n
  be set(n: Ring) ⇒next = n
  be pass(i: U32) ⇒
    if i > 0 then
      try (next as Ring).pass(i−1) end
    end
```

```
actor Main
  new create(env: Env) ⇒
    let hd = Ring
    var tl = hd
    for k in Range[U32](0, 8) do
      tl = Ring.create_set(tl)
    end
    hd.set(tl)
    hd.pass(16)
```

Fig. 2. Actor ring.

and Metronome (Bacon et al. 2003) manage to further reduce pause times, but at a cost in terms of performance — up to 40% slowdowns can be expected. As Orca, thanks to its co-design with the Pony type system, does not require barriers on access to individual memory locations, it is reasonable to expect better throughput. In a way, Orca can be viewed as having barriers on message sends which are less frequent than stores. Various designs for segregated heaps have been explored in the literature. Domani et al. (2002) introduced a collector that segregates between thread-local objects and shared objects. Write barriers are used to distinguish between shared and unshared objects, and shared objects are collected in a full GC phase which introduces significant global pauses. Orca does not require full GC as all objects belong to a single actor and are collected by that actor. Pizlo et al. (2007) introduced hierarchical real-time collection in. The idea is to segregate the heap into heaplets which can be collected by different collectors. To deal with references across heaplets, a global collection phase is required. Write barriers are used to record cross-heaplet references in a global data structure. Orca avoids the need for global collection thanks to its reference counting scheme. Spring et al. (2007) proposed Reflexes as an abstraction for real-time concurrent computing. Like actors in Pony, Reflexes are isolated, single-threaded computations communicating by message passing. The Reflex type system ensures that mutable messages can be communicated without synchronisation or copy. Unlike Pony, Reflexes were not garbage collected but relied on a constrained form of region allocation. Furthermore, only a limited set of data types were allowed in messages. Auerbach et al. (2008) extended Reflexes with per-task garbage collection. Orca gives programmers more flexibility as arbitrary objects can be communicated in messages.

Erlang is an actor-based language with its own dedicated virtual machine called BEAM (Armstrong 2007). For memory management purposes, BEAM performs a deep copy of messages, *i.e.* the transitive closure of objects reachable from the message is copied. This ensures that actor states are isolated. Binary objects (byte-oriented data) are treated specially, as copying them would be too costly, so they are reference counted. Binary objects may not contain pointers and so cannot create cycles, thus obviating the need for a cycle collector. Orca does not copy messages, but it does trace them, both on send and receive, to *track* actor–object dependencies.

A related problem for actor-based languages is the collection of actors. An actor must be kept alive as long as any other actor has a reference to it, the actor is executing, or it has a non-empty message queue. In Actor Foundry, the actor graph is turned into an object graph so that a tracing collector may reclaim actors as well as objects (Vardhan and Agha 2002). SALSA (2013) uses snapshots, reference listing, and trace-based global heaps to collect both local and distributed actors. Passive objects are collected using the underlying JVM's trace-based collector. Pony uses MAC (Clebsch and Drossopoulou 2013) to collect actors wheras Orca (and hence this paper) is only concerned with collection of objects.

3 PONY: ACTORS, OBJECTS & CAPABILITIES

Actors in Pony are single-threaded stateful constructs with a first-in-first-out message queue. Figure 2 illustrates how to create a ring of actors exchanging a decreasing numeric value. It shows two actor declarations. The Main actor creates eight Ring actors and connects them together. A Ring has two behaviours: when it receives message set, it updates its next field to refer to the message's argument; when it receives message pass, it sends a message to the next actor in the ring.

Actors take messages from the front of their message queue and execute the behaviour associated with that message. As part of executing a behaviour, an actor can create a new actor, change its state, or send messages to other actors – placed at the end of their respective message queues. Message delivery preserves causality: thus if an actor A sends a message msg1 to B followed by msg2 to C, then msg2 is causally dependent on msg1. If actor C reacts to msg2 by sending msg3 to B, causal delivery requires that msg1 be processed before msg3. Section 3.1 and Section 5.4 discusses *how* and *why* of causality in more detail.

Pony's object model is familiar from languages such as C#, Java, and Scala — namely a statically typed, class-based system with both structural (interfaces) and nominal (traits) subtyping. Figure 3 illustrates how to define classes with a combination of mutable and immutable state. Class D has no fields or functions, instances of this class are created by invoking the default (empty) constructor, *e.g.* d = D. Class E has two fields. Fields that may be uninitialised are given union types with None; here field g has type (E iso | None).

```
class D

class E
  var f: D val
  var g: (E iso | None)
  new create(v: D val) ⇒
    f = v
    g = None
  fun ref update(v: E iso) ⇒
    g = consume v
```

Fig. 3. Classes.

We now describe how Pony's type system — capabilities are attached to types — (Clebsch et al. 2015; Steed 2016) uses capabilities to constrain behaviours. We distinguish between sendable and non-sendable capabilities. Sendable capabilities are used for objects that may be exchanged in messages. The val capability denotes the ability to read fields of an object, which is immutable (*i.e.* cannot change[1]). The tag capability denotes an opaque reference, one that cannot be read or written to (only the object's identity can be used). The iso capability denotes the ability to write fields of an object, and other actors may neither read nor write to fields of that object. Non-sendable capabilities, *i.e.* reference (ref), transition (trn), and box (box), may not be used in messages. The ref capability is for objects that can be read from and written to, as well as aliased internally to the current actor.

```
1  var v: D val = recover val D end
2  var v1: D val = v // ✓
3  var v2: D iso = v // ✗
4
5  var i: D iso = recover iso D end
6  var i1: D tag = i // ✓
7  var i2: D iso = i // ✗
8  var i3: D val = i // ✗
9
10 var i4: D iso = consume i // ✓
```

Fig. 4. Aliasing constraints.

Capabilities also limit aliasing. An object reachable through a val reference can only be aliased as val and tag and box; an object reachable through a tag capability can be aliased without constraint; and an object reachable through an iso reference can only be aliased by tag references. These aliasing constraints ensure at compile time that Pony programs are data race free — a fact leveraged by Orca as we shall soon see.

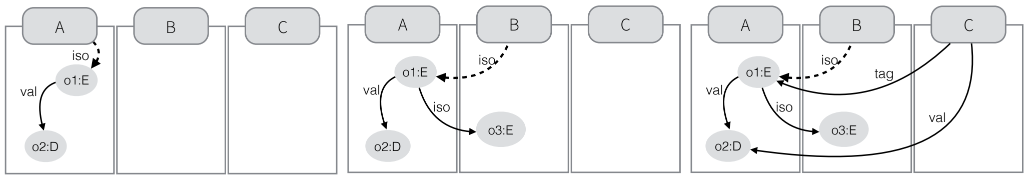
Figure 4 illustrates some of the constraints enforced by capabilities. Line 1 creates v, an instance of D, and gives a val capability to it. Upon creation, objects have ref capability by default. Thus, when creating objects with other capability, one needs to recover the newly created object to val or iso capability. It is thus allowed to create further val aliases (Line 2). However

[1]This is a stronger property than *read-only*, where immutability only applies through certain references or to certain agents.

```
1  actor A
2    be m(b: B, c: C) ⇒
3      let e: E iso = recover iso
4        E(recover val D end)
5      end
6      c.m(e, e.f)
7      b.m(consume e)
```

```
8  actor B
9    be m(e: E iso) ⇒
10     var d: D iso = recover iso
            E end
11     e.update(consume d)
12     ... // more code
```

```
13 actor C
14   var _e: E tag
15   var _d: D val
16   be m(e: E tag, d: D val) ⇒
17       ...
```



(a) Snapshot 1. (b) Snapshot 2. (c) Snapshot 3.

Fig. 5. Actors and their $\boxed{\text{heaps}}$. Dashed arrows are stack references. Continuous arrows are heap references.

attempting to create an iso alias is not allowed (Line 3). Line 5 creates another instance of D, this time with an iso reference which allows the current actor to read and write fields of the object (if it has any). Only tag aliases to this object are allowed (Lines 6–8). Line 10 illustrates how to transfer an iso capability by moving the reference from the variable i to the variable i_4 using the consume operator. A variable whose contents has been consumed cannot be read before it has been re-assigned.

In addition to transferring capabilities between variables, the type system also allows converting a capability into another. For example, mutable objects can become immutable (but not the other way around) during their lifetimes.

Figure 5 combines the features we have discussed in a single example. Namely, three actors A, B and C create objects (o_1, o_2, ...) and share them via message passing. When A receives message m, and executes Lines 3–5, it creates objects o_2 of class D, and o_1 of class E, and holds an iso capability to o_1 (variable e) — this is shown in Snapshot 1. Then, in Line 6, it sends m to C, containing a tag reference to o_1 (e), and a val reference to o_2 (e.f). In Line 7, it sends m to B, containing an iso reference to o_1, after consuming its own reference e. The order in which B and C process their messages from A is non-deterministic. Assuming that B is scheduled first, receiving the message first (lines 10–11), B can update o_1, because it holds an iso reference to it. This is shown in Snapshot 2. Assume that while B is executing the code from Line 12 onwards, C is scheduled. This is shown in Snapshot 3. We now see that objects o_1 and o_2 are accessible from both B and C. Note that this cannot introduce data races: B can read and modify o_1 while C cannot read nor modify it, and both B and C can read but not modify o_2.

3.1 Leveraging the Co-Design

Co-designing a language together with its runtime allows enforcing a number of properties desirable from the point of view of its implementation. The use of capabilities segregates objects into non-sendable and potentially-shared objects. This naturally leads to a programming style that favours local objects, as shared objects require additional attention. Non-sendable objects tend to be more numerous than sendable objects (Wrigstad et al. 2009). Trivially, such objects are never subject to data races.

The preponderance of non-sendable objects means that it is sensible to design a collector where allocation and reclamation happens locally inside a single actor, in parallel with all other actors, no matter what they are doing. Such local garbage collection gives a more intuitive cost model of memory management than a single shared heap: each actor only has to account for garbage collection of the object it has created in its execution costs. This means for example that judicious allocation elsewhere in a system cannot slow down the current actor by forcing it to participate in garbage collection.

Objects referenced by sendable capabilities may be shared by multiple actors, but if they are mutable, Pony's type system ensures that, at any given point in time, at most one actor is allowed to modify them. Thus Pony programs are data race-free. When an actor is idle, the only objects accessible to it are in its fields or in messages in the actor's queue (transitively).

Scheduling collection when an actor is idle avoids having to consider roots on the stack and ensures that behaviours need not pause for memory management. Moreover, because shared objects are data race-free, it is possible to implement a non-blocking collector — as no other actor may be mutating an object while the object is being traced. In terms of synchronisation operations, the only memory barrier present in Pony is when messages are enqueued, which causes manipulation of reference counts (which are local to the current object, and therefore trivially atomic without need for synchronisation). This is sufficient to ensure visibility of writes for shared objects. Also, the type system ensures that all fields are initialised, so there is no need to zero out pages.

Finally, because message delivery is causal, the same messaging infrastructure that delivers application-level messages can be used to deliver reference increments and decrements. Here, causality is important because processing increments and decrements for the same object out of order may lead to premature deallocation.

To see how causality arises naturally in Pony, consider a scenario where three actors with empty message queues, A, B and C are executing, possibly in parallel, the statements in the table (the rows of each column is in program order). Inside each actor, sends and receives are not reordered. Furthermore, mailboxes are FIFO ordered, and send(T, msg) is a synchronous operation that returns only after msg has successfully been appended to the message queue of the target T. The order of the sends in A thus guarantee that msg1(...) will end up in B's mailbox before msg2(...) ends up in C's mailbox. Since C's sending of msg3 to B is triggered by the receipt of msg2(...) (a causal dependence), regardless of when in time B is executed, recv(x) in B will have x = msg1(...) and *not* x = msg3(...).

in A	in B	in C
send(B, msg1(...))	recv(x)	recv(z)
send(C, msg2(...))	recv(y)	send(B, msg3(...))

4 ORCA: A NON-BLOCKING CONCURRENT COLLECTOR

Having introduced the Pony language, we are finally ready to discuss its garbage collector. Orca, like Pony itself is written in C. In the following discussion, we show pseudo-code simplified for explanatory purposes. Interested readers are referred to the open source Pony repository for full source code of the collector. We skip over aspects of the system that are not directly relevant to Orca. The object model is simple: objects are structures with a header field containing a pointer to a type, and a sequence of fields accessible at fixed offsets. Primitive values are unboxed machine representations. Programs are compiled ahead-of-time.

Orca is a non-moving, concurrent, multi-threaded collector with no atomic operations. The collector has no read/write barriers. Each actor is tasked with reclaiming the objects that it has allocated. This is implemented as a combination of mark and sweep for objects that are not shared with other actors and a variant of reference counting for shared objects. Reference counts are coarse-grained and represent the total interest in an object from the actors and messages that reference it. This is an abstract number, which allows additional optimisations (*e.g.* further sharing an object without notifying its owner). Reference counts are incremented or decremented by the runtime system upon message send or receipt. Moreover, explicit requests for increments or decrements may be sent to the owner. Increment messages (INCs) communicate additional use of an object in the system. Decrements (DECs) communicate a decrease in external use of an object.

4.1 Fundamentals and Correctness

We now discuss the fundamental properties which guarantee that Orca will never collect locally reachable or visible objects. We hope, as Orca diverges from mainstream collectors, that this will make the design and its rationale more intuitive. We call an object *visible*, if it is reachable from a foreign actor or from a message. An object's *owner* is the actor that created it. An actor is *foreign* to an object if it is not its owner. An object is *protected* at some actor, if the actor's reference count for this object is greater than 0, meaning that *the actor will make sure the object is not reclaimed.* Orca relies on the type system and the reference counts to reflect and respect object visibility as well actors' interest in an object (**I**=invariant):

- **I1** At any point, if an actor may write to an object, then no other actor can read from or write to this object's fields. Thus, ORCA can avoid write barriers and tracing needs no synchronisation.
- **I2** Immutability is persistent (*i.e.* an immutable object will never be seen as mutable) and deep (*i.e.* no object accessible from an immutable object is seen an mutable).
- **I3** Any live object is protected at its owner.
- **I4** Any object reachable from a foreign actor is protected at this actor.
- **I5** The owner's reference count for an object is *consistent* with the state of the system.

The first and second invariant are enforced by the Pony type system, while the rest are Orca's responsibility. The notion of consistency in **I5** intuitively means that the owner of an object must have a view of the number of outstanding remote references that agrees with that of the other entities in the system. For any given object, LRC is the owner's reference count, OMC is the sum of all *INC* and *DEC* messages which increment and decrement reference counts, FRC is the sum of the reference counts in all other actors, and AMC is the number of application messages from which the object is reachable. Consistency means that LRC + OMC = FRC + AMC. Additionally, Orca assumes that finalisers are "safe" in the sense that running a finaliser cannot revive an object (Pony provies statically guaranteed safe finalisers).

4.2 Preliminaries: Key Data Structures

The reference counts discussed earlier are held within the structures describing actors. We define these, as well as some more Pony runtime data structures in Figure 6.

A Context is a thread-local data structure used to keep information about the execution context of the current thread. The field curr is a reference to the currently executing actor, traceobj and traceact are two function pointers used for garbage collection, the function they refer to depends on the phase of the collector. The gc field is a stack of object references and associated tracing function used by the collector during marking. Finally, acquire is a map of actor reference count data structures which is used to record foreign objects discovered during marking.

An Actor has a queue of messages mq, a local heap hp, and three fields dedicated to garbage collection. The current epoch is an unsigned integer held in mark. Fields orcs and arcs are hashmaps

```
struct Context {
    Actor curr
    Trace traceobj
    Trace traceact
    Stack gc
    ARCmap acquire
}
```

```
struct Actor {
    Messages mq
    Heap hp
    uint mark
    ORCmap orcs
    ARCmap arcs
}
```

```
struct Heap {
    Chunk free[S]
    Chunk full[S]
    Chunk large
    uint used
    uint ngc
}
```

```
struct Chunk {
    Actor owner
    char[] mem
    uint sz
    uint slots
    uint shallow
    uint finalize
    Chunk next
}
```

```
struct ORC {
    Any tgt
    uint rc
    uint mark
    bool immut
}
```

```
struct ARC {
    Actor actor
    uint rc
    uint mark
    ORCmap map
}
```

Fig. 6. Data structures.

used to record reference counts for local objects shared with other actors through message sends, and foreign objects shared with the current actor through message receipts.

A Heap is an actor-local data structure that contains the set of Chunks which hold objects allocated by that actor. Each chunk holds up to 64 objects and are segregated into S+1 size classes. Small objects are allocated in one of the S size classes. Large objects are allocated into their own chunks. An actor's Heap thus consists of an array (one per size class) of chunks with available slots (free), an array of fully occupied chunks (full), as well as a chunk for large objects (large). The heap also keeps track of the total amount of live memory (used) and the threshold used to determine when the next GC cycle should run (ngc). Note that this is determined *per actor*.

A Chunk is a block of memory (mem) associated to an actor (owner). Each chunk holds a number of equal sized slots (sz). A bitmap (slots) indicates which slots are occupied and which slots are available. This bitmap is also used during marking. The shallow field is a bitmap used during GC to indicate which objects should be traced recursively. The finalise bit map indicates which objects have finalizers. Chunks are arranged as linked lists (next).

An object reference count, ORC, is a data structure allocated to keep track of shared objects. Each ORC refers to an object (tgt), keeps a reference count (rc) which is an upper bound on the number of references to that object; a mark field used during GC and a field to indicate if the object should be treated as immutable. Note that reference counts do not directly reflect the number of references to an object or the topology of object graphs. Instead, they are an upper bound on the number of entities (other actors or messages) which have references to the target object. This entails that cycles do not prevent collection[2] and that update of reference counts can be deferred.

Figure 7 illustrates a configuration with two actors. Object o_1 is local to actor a_1. It is referenced locally by object o_0 and externally from object o_2 which is local to actor a_2. Since actor a_1 has shared o_1, it has an ORC for that object (in Actor.orcs). Actor a_2 has received o_1 in a message, so it has a reference to actor a_1 (in Actor.arcs) and that data structure has an ORC for a_1.

[2]Cycles of actors are handled separately from Orca, see Clebsch and Drossopoulou (2013).

The value of the reference count for o_1 is 1, because object o_2 points to it. The value of reference count in a_1 is also 1, because one other entity has access to the object o_1. Note that the local reference from o_0 to o_1 is not counted. This diagram also shows that even though local objects may point to foreign objects (here o_2 is local to, and o_1 is foreign to, a_2), the associated book-keeping information (*i.e.* Chunk, ORCMap and ARCMap) are contained within the actor, and thus can be manipulated by the actor without race conditions.

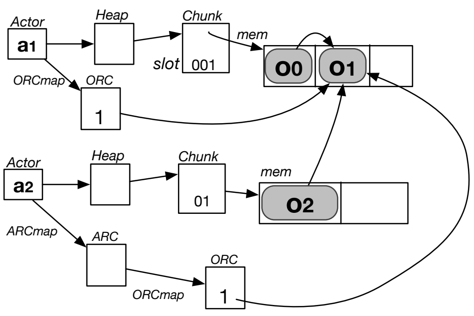


Fig. 7. Reference counts.

Object reference counts are only created for objects that have been sent in a message. When ORC.rc drops to zero for an object, its ORC can be deallocated. Local tracing can now safely determine whether the object is live or garbage. The owner of an object is notified that other actors have either acquired a reference to an object, or dropped all their reference through *INC* and *DEC* messages. Thus, line 5 in Figure 9 may process *e.g.* a decrease in a reference count for an object to the point where its ORC.rc drops to zero. If the next turn in the event loop is a garbage collect, this object will be freed if no local references remain.

Two additional data structures play a role in GC: The first is *scheduler threads* each of which keeps a queue of actors that have pending work. Whenever a thread is idle, it pops an actor from this queue and it schedules the actor's work, passing its context to the actor. If a thread has no work, it may steal an actor from another thread's queue. The second data structure is a lock-free multiple-producer, single-consumer FIFO *message queue*, one for each actor, from which it can obtain messages. There are two kinds of messages in Pony, *application messages* sent by other actors and *system messages* sent by the run time system (*e.g.* reference count increments and decrements). Message queues are the only data structure requiring synchronisation: push and pop operations are atomic.

4.3 Allocation and De-Allocation

Orca has several allocation functions, Figure 8 shows the allocation function for small objects that require finalisation. This function does not require synchronisation in the fast path. The slow path is hit when there are no chunks of the requested size class with free slots. If this occurs, a new chunk is allocated. Because this operation takes "global memory" and makes it local to an actor, it requires synchronisation (and may end up request more memory from the operating system). The allocation function finds the first free slot, and sets the corresponding bit in Chunk.slot and Chunk.finalize. If the chunk is full, it is moved from the free list to the full list. Reclamation happens implicitly at the end of a collector cycle, the Chunk.slot field is written to during marking, any slot that has not been marked is available for reuse. Chunks that have free slots are moved from the Heap.full list to Heap.free.

```
alloc_small_fin(Actor a, Heap h, uint
        szclass) {
    Chunk c = heap.free[szclass]
    if (c == NULL) c = allocate(szclass)
    uint bit = ctz(c.slots)
    c.slots &= ~(1 << bit)
    c.finalize |= (1 << bit)
    if (slots == 0) {
        h.free[szclass] = c.next
        c.next = h.full[szclass]
        h.full[szclass] = c
    }
    h.used += sz_in_bytes(szclass)
    return c.mem + (bit << MINBITS)
}
```

Fig. 8. Pseudo-code for allocation.

4.4 Garbage Collection & Collection Cycles

Orca allows multiple threads to perform collection in parallel without synchronisation. Each thread's work is summarised by the pseudo-code of Figure 9. Given an actor a, the scheduler takes a snapshot of a's message queue (Line 3) and then alternates between handling messages and potentially performing a collection cycle. For any given actor, each collection cycle is identified by its epoch (Actor.mark), used during marking. Epochs are incremented at the end of cycle (Line 11); no action is needed to prevent overflows as epochs are only compared for equality. Epochs are actor-local and thus need no synchronisation.

A garbage collection cycle is kicked off (Line 6) if the memory allocated by the actor (Heap.used) is larger than the gradually increasing threshold (Heap.ngc). Selecting small values for the threshold will result in more frequent cycles. There is no global limit on allocated memory as this would entail synchronisation. There is nothing that prevents triggering garbage collection during execution of a behaviour, but the implementation of Orca in Pony currently only runs between behaviours to avoid stack scanning. We have not had any reports that suggests a need to implement inter-behaviour GC from Pony users.

```
1  run(Context ctx, Actor a) {
2    Message msg, end
3    end = atomic_load(a.mq.end)
4    while ((msg = pop(a.mq))) {
5      handle(ctx, a, msg)
6      if (needgc(a.heap)) {
7        roots(ctx, a)
8        markimmut(ctx)   // Fig 10 left
9        traverse(ctx)    // Fig 10 center
10       sweeporcs(a.orcs) // Fig 10 right
11       a.mark++
12       finalize(a.heap)
13       free(a.heap)
14     }
15     if (msg == end) break
16   }
17 }
```

Fig. 9. Pseudo-code for the scheduler's run.

The initial value for Heap.ngc is 2^N bytes, for some (command-line) configurable N, which is 14 by default. Upon each garbage collection cycle Heap.ngc is set to M times its current value, for some (command-line) configurable M, which is 2 by default.

The other steps of a collection cycle are as follows. The roots function (Line 7) pushes all the fields of the current actor on the stack (Context.gc). These are the only roots in Pony. The markimmut function goes over the local immutable objects which have been shared, marks them as reachable, and recurses into their substructures using a trace function obtained in a standard fashion from the object header via the type() function. Function traverse recursively marks objects on the Context.gc stack.

A reference p that is not found in this tracing is in precisely one category below:

```
markimmut(Context ctx) {
  foreach (ORC o in ctx.curr.orcs)
    if (o.immut && (o.rc > 0)) {
      mark(o.tgt)
      Trace fn = type(o.tgt).trace
      fn(ctx, o.tgt)
    }
}
```

```
traverse(Context ctx) {
  foreach (pair in ctx.gc) {
    Trace fn = pair[1]
    Any p = pair[2]
    fn(ctx, p)
  }
}
```

```
sweeporcs(ORCmap orcs) {
  foreach (ORC o in orcs)
    if (o.rc > 0) {
      Chunk c = chunk(o.tgt)
      setbit(c.shallow, p, c.sz,
             c.mem)
    } else {
      delete_index(map, i)
      free(o)
    }
}
```

Fig. 10. Pseudo-code for the auxiliary functions. pair is a (trace function, object) entry from the GC stack.

1: Local and visible If p is a local visible object (*i.e.* it has an entry in LRC with a refcount > 0), then p must be kept alive. Note that p may not actually be in use because of unprocessed decrements in the message queue. This eventual consistency might delay collection of some garbage, but not leak.

2: Local and invisible If p is a local and invisible object (*i.e.* whose refcount is 0), we can safely delete p, after executing its finalizer (if any).

3: Foreign If p is foreign, we send a decrement to its owner that corresponds to the entry for p in FRC (the foreign reference count table) and subsequently delete our FRC entry for p.

Invariant **I3** (live objects protected at its owner) guarantees that no visible object will be collected, and the actions in the second and third case preserve **I5** (reference count consistency). For objects to be collected, we care about the object's owner, and the mode in which the object is referenced. If the mode is tag, we do not recurse through the object.

The sweeporcs function visits all the ORCs of local objects and either sets them to be shallow (if there is an outstanding reference count) or (if the reference count is zero) deletes the corresponding entry in the ORCmap.

The finalize function runs the finalizers of objects that have been found unreachable. The free function finds all chunks in the actor's heap that have no live objects in them and returns them to the free list on the global heap, causing the actor's heap to shrink.

4.5 Send, Receive and Trace

4.5.1 Send. Figure 11 shows the pseudocode for tracing objects on message sends. When sending, we are increasing AMC for the object (and eventually FRC upon receipt). For reference count consistency (maintaining **I5**), we increase LRC for the object sent when sending a local object. In the case of sending a remote object, we cannot directly access the owner's LRC (that would introduce synchronisation overhead) we instead decrease the sender's FRC for that object — a simple non-atomic decrement. Remember, reference count consistency is LRC + OMC = FRC + AMC — which clearly shows why decreasing FRC and increasing AMC accordingly as the result of an object being passed around does not need to modify the object's owner's LRC. However, if the sender's FRC is too small to be decreased (we cannot decrease it to zero), we *inflate* its reference count with some constant value GCINC and send a corresponding *acquire* message to the object's owner to inform it of the inflated reference count (this increases ORC and the owner's LRC eventually on receipt).

We now walk through this in more detail following the pseudo code in Figure 11, but omitting the parts highlighted — these represent optimisations which will be discussed later. The sendobject function derives the owner of the object being sent — p — from its location in memory. If the object we are sending is owned by the current (sending) actor, delegate to send_local, otherwise delegate to send_remote. The parameter view tracks the static view of the reference passed, *e.g.* if it is a tag (OPAQUE) or val (IMM), or mutable (MUT) capability.

On Line 8, the send_local function gets a handle to the ORC entry for the object p being sent, from the local reference counts (LRC), which involves possibly creating it (**I2**). The field a.mark holds the current epoch, and if we have already marked p in the current epoch, there is no more work to be done (Line 9). Otherwise, we update the reference count for the object in the ORC entry and mark it with the current epoch. Opaque (tag) values are not traced (Line 12) further.

The send_remote function is more involved. Lines 26 and 27 are isomorphic with send_local. The conditional on Line 29 is true when p is shared immutably and discussed in the next section.

Lines 34–37 deal with the the case when we cannot simply decrease the local FRC for reference count consistency (as this would break **I4**). Line 36 inflates the current FRC and line 38 adds the

```
1  sendobject(Context ctx, Any p, Type t, int view) {
2    Actor a = chunk(p).owner
3    if (a == ctx.curr) send_local(ctx, a, p, t, view)
4    else send_remote(ctx, a, p, t, view)
5  }
6
7  send_local(Context ctx, Actor a, Any p, Type t, int
         view) {
8    ORC obj = getorput(&gc.local, p, a.mark)
9    if (obj.mark == a.mark) return
10   obj.rc++
11   obj.mark = a.mark
12   if (view == OPAQUE) return
13   if (view == IMM) obj.immut = true
14   if (!obj.immut) push(ctx.gc, (p, t.trace))
15 }
16
17 acquire(Context ctx, Actor actor, Any p, bool immut) {
18   ARC aref = getorput(ctx.acquire, actor, 0)
19   ORC o = getorput(aref, p, 0)
20   o.rc += GCINC
21   o.immut = immut
22 }

24 send_remote(Context ctx, Actor a, Any p
       , Type t, int view) {
25   Actor this = ctx.curr
26   ORC obj = getorput(this.ORCmap, p,
         this.mark)
27   if (obj.mark == this.mark) return
28   obj.mark = this.mark
29   if (view == IMM && !obj.immut &&
         obj.rc > 0) {
30     obj.rc += (GCINC - 1)
31     obj.immut = true
32     acquire(ctx, a, p, true)
33     mutability = MUT
34   } else if (obj.rc <= 1) {
35     if (view == IMM) obj.immut = true
36     obj.rc += (GCINC − 1)
37     acquire(ctx, a, p, obj.immut)
38   } else {
39     obj.rc--
40   }
41   if (view ==
         MUT) push(ctx.gc, (p, t.trace))
42 }
```

Fig. 11. ORCA logic for tracing argument objects on message sends. Section 5 describes optimisations.

object and its immutability into a collection which will be used at the end of the call to notify the object's owner that we have inflated our FRC — this will allow the owner to inflate its LRC accordingly, thus preserving **I5**. If we are sending an object as immutable, Line 35 records this in the ORC metadata, otherwise we push the object and its tracing function on the stack (Line 14).

Lines 38–39 deal with the case when it is possible to decrease the local FRC for reference count consistency.

Finally, line 41 pushes the object onto the GC stack so that its *contents* are also traced using the statically generated trace function t.trace for the type t. This function is generated by the Pony compiler and leverages statically available capability information — for example whether it is immutable or not.

4.5.2 Receive. In the interest of saving space, we refrain from discussing tracing on message receive at the same level of detail as we did for sending. The code for receiving is simpler than, and otherwise mostly isomorphic to, the code for sending, *e.g.*, tracing does not recurse into opaque or immutable structures, with one addition and one difference, which we discuss below.

Addition: On first receipt of an object, the actor will increase its apparent used memory to provoke garbage collection. This is necessary as an actor who only handles remote objects would otherwise never trigger garbage collection, and thus never send decrements for objects it has dropped. Because the tracing of val objects has been optimised away, receipt of a val object increases the apparent used memory with a constant which is currently 1024 bytes. This is a heuristic based on the small number of Pony programs in existence.

Difference: Upon receiving an object owned by itself, the actor's reference count for the object will *decrease*. This is natural, since reference counts model the references from non-owning actors and messages on the wire.

```
1  class Obj
2    let f: Obj2 ref
3
4  actor Act
5    let f1: Obj iso
6    let f2: Obj trn
7    let f3: Obj ref
8    let f4: Obj box
9    let f5: Obj val
10   let f6: Obj tag
```

```
12 Obj_trace(Context ctx, Act obj) {
13   trace(ctx, obj.f, Obj2_trace, MUT)
14 }
15
16 Act_trace(Context ctx, Obj obj) {
17   trace(ctx, obj.f1, Obj_trace, MUT)
18   trace(ctx, obj.f2, Obj_trace, MUT)
19   trace(ctx, obj.f3, Obj_trace, MUT)
20   trace(ctx, obj.f4, Obj_trace, MUT)
21   trace(ctx, obj.f5, Obj_trace, IMM)
22   trace(ctx, obj.f6, Obj_trace, TAG)
23 }
```

```
25 trace(Context ctx, Any obj, Trace
       fn, int view) {
26   if (local(obj)) {
27     mark(obj)
28     if (view != TAG) {
29       fn(ctx, obj) // recurse
30     }
31   } else {
32     mark_remote_obj(ctx, obj,
           view)
33   }
34 }
```

Fig. 12. Synthesised trace functions for an actor and an object.

4.5.3 Synthesising Trace Functions. In order to trace objects, the Pony compiler generates a trace function for each concrete type describing how to reach the fields of a *readable* instance of that type. How to trace an object of a given type depends on the types (and capabilities) of its fields: a field of primitive type does not require any action; a field of a type annotated with tag points to an unreadable object which is marked as reachable but cannot be considered a root, since the actor cannot read its fields; otherwise, the field points to a readable object which is marked as reachable and, recursively, considered a root, meaning its contents are traced.

Figure 12 shows a Pony object and a Pony actor to the left, the pseudo code for their synthesised trace functions in the middle, and the generic trace function from the run time to the right. Note the close correspondence between the class Obj and Obj_trace and the actor Act and Act_trace.

Lines 21–22 show how the val and tag capabilities are carried through in compilation of the trace function. Line 28 shows how we are not recursing through ţ objects, even when they are local.Objects that are not local to an actor are handled by the mark_remote_obj function (called on Line 32), which is shown in Figure 13.

```
1  mark_remote_obj(Context ctx, Any obj, int
       view){
2    if (marked(ctx, obj)) return
3    mark(owner(obj))
4    mark(obj)
5    if ((view == IMM) && !imm(ctx, obj) &&
6        (rc(ctx, obj) > 0)) {
7      rcinc(ctx, obj, AMOUNT)
8      setimm(ctx, obj)
9      acquire(ctx, owner(obj), obj, IMM)
10     view = MUT
11   else if (rc(ctx, obj) == 0) {
12     if (view == IMM) setimm(ctx, obj) {
13       rcinc(ctx, obj, AMOUNT)
14       acquire(ctx, owner(obj), obj, imm(ctx,
             obj))
15     }
16   }
17   if (view == IMM) recurse(obj)
18 }
```

Fig. 13. Mark remote. Optimisations highlighted.

In mark_remote_obj, we mark the ORC of each reachable object with the current epoch (Line 4). Because actor lifetimes are lower bounded by objects on their local heaps, we also mark the object's owner (Line 3). If we hit on an object that is already marked, we stop. The rest is discussed in section 5.3.

5 OPTIMISATIONS, CORRECTNESS AND CAUSALITY

Immutability is deep and persistent. This allows immutable objects to be handled more efficiently than mutable objects. If o_1 is an immutable object and o_3 is reachable from o_1 (*e.g.*, o_1 stores a reference to o_2 in a field and o_2 stores a reference to o_3), it is easy to see that o_1's lifetime upper bounds the lifetime of o_3. This gives rise to the idea that an immutable data structure need only be traced by the owner of the immutable objects it contains.

Thus, a positive reference count for the root of an immutable data structure suffices to keep alive all objects reachable from it which are *owned by the same actor*. This reduces both the number of reference count manipulations and the amount of tracing during send, receive and garbage collection.

Figure 14 depicts an immutable aggregate (val) rooted in o_1 with subobjects o_2 and o_3 and o_4 s.t. o_1.owner = o_2.owner and o_1.owner ≠ o_3.owner=o_4.owner. This data structure can be protected by only two positive reference counts, one for o_1 and one for o_3. The reason why a positive reference count for o_1 will not protect o_3 is because o_3's owner may be unaware of the relation between the two objects. Hence, the aggregate has two roots.

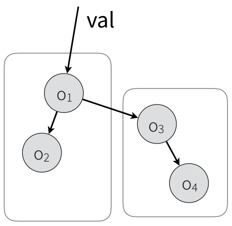


Fig. 14. An immutable aggregate with two roots at o_1 and o_3.

5.1 Fundamentals and Correctness — Revisited

The optimisation for val objects can be explained by redefining what is means for an object to be *protected*, and redefining *AMC*, and thus implicitly weakening invariants **I3**–**I5**. Namely, an object o' is *protected by an object o*, if either they are the same object, or o is immutable, and o reaches o', and they have the same owner. An object o is *protected at actor a*, if there exists an object o' such that a has a reference count strictly greater than 0 for a', and o' protects o. We also reinterpret $AMC(o)$ as the number of application messages which can access o without going through an immutable object.

With these new definitions, **I3**–**I5** read the same as before, but their meaning is now weaker. For the example from Figure 14, we have that o_1 protects o_2 but not o_3, and o_3 protects o_4, and o_4 is protected at a, where $a{=}owner(o_3)$, as long as a's reference count for o_4 or for o_3 is > 0.

In the implementation, whether an object is known to be immutable is shown by the immut flag. Lines 13, 31 and 35 keep track of the fact that the object has been reached as immutable. As a consequence, future sends of p will not recurse deeper into the object and garbage collection will not need to trace inside foreign immutable objects. Instead, the objects will be kept alive by Orca because they are protected by an object with an LRC > 0.

5.2 Send — Revisited

The val optimisation is implemented in Figure 11. Hitting an object that is marked as val, tracing does not recurse into the object structure (Line 14). Something similar happens in send_remote, but here we must cater to two more cases: 1) a iso turned into a val, and 2) dropping parts of an immutable structure giving it new roots *e.g.*, obtaining a direct reference to o_2 and dropping o_1.

For send_local, both cases are handled in Line 13 and 14 which marks an IMM object thusly, and does not recurse. For send_remote, Case 1 is caught by the conditional on Line 29. We update the ORC to reflect the immutability of p, and send a message to the owner updating its immutability status accordingly (Line 33). At the same time, we inflate our own reference count (−1 to account for the sent reference) for the object (Line 31) to allow sharing it cheaply again.

Case 2 is caught by the conditional on Line 34. In this case, the reference count for p will be zero, unless there is a direct reference to p elsewhere in the same actor. Since we come here from a field or variable whose type is val, mutable will be IMM and we update the ORC entry accordingly. This is required because we may not have seen the object before if it was shared as part of an immutable structure which therefore was not traced (this applies also to send_local). For the same reason, this status change must be communicated to the owner (Line 37).

Without the optimisation, Line 41's guard changes to != OPAQUE.

5.3 Marking — Revisited

The code in Fig. 13 implements the immutability optimisations for marking. The guard on Line 5 captures the case when we see an object as immutable, but the object's immutability flag is not yet set. In this case, we set our local immutability flag, and construct a transfer message that informs the owner of the change. Last, we set view = MUT in order to avoid recursing on Line 17. The guard on Line 11 captures the case when tracing finds an object for which there is no *FRC* entry. This is true when new references *into* immutable structures are created.

Finally, mark_remote_obj only recurses through an object when the view is immutable. This is necessary to identify all protectors in an immutable structure.

5.4 The Importance of Causality for Reference Counting

Causality is crucial for the safety of Orca — in particular in what concerns the delivery of reference count increment and decrement messages. Consider the following scenario with actors A, B and C. The object o is owned by A, and its reference count in A and B is 1. B sends o to C, triggering a notification to o's owner A. In the table below, the rows of each column are in program order and the message queues are empty.

in A	in B	in C
recv(x)	send(A, INC(o, 256))	recv(z)
recv(y)	send(C, msg1(o))	send(A, DEC(o,1))

The causal relationship between B's send(C, msg1(o)) and C's send(A, DEC(o, 1)), guarantees that the message INC(o,256) will be be stored in A's mailbox before DEC(o,1). If messages were not delivered in causal order, the DEC message could overtake the INC message, causing A to erroneously collect o even though B still has references to it. Similar ordering problems occur in manual reference counting (Hillegass 2011).

5.5 The Need to Refine I5

In sections 4.5.1 and 4.5.2 we argued that the actions upon sending and receiving objects preserve **I5**. But this is a simplification: Upon closer inspection, we can see that the invariant is restored only at the end each procedure (*e.g.* on line 10 in Fig. 11, by incrementing the RC we break the invariant temporarily, and only restore it when we send the message and thus implicitly also incrementing the AMC).

Moroever, if we consider that actors execute concurrently with other actors, we realize that invariant **I5**, as stated so far, need not hold even at the end of a procedure. For example if an actor α_1 competes sending a message m_1 containing object o, and at the same time another actor α_2 starts sending another message containing the same object o, then, even though at the point where α_1 puts m_1 on the message queue and $AMC(o)$ is incremented and thus counterbalances the increment or increment of α_1's RC for o, actor α_2 has modified its RC for o, and so **I5** does not hold.

Therefore, **I5** is a useful simplification. It would only hold at the start and end of the procedures, and only if the procedures were executed atomically. In a companion paper in preparation, we refine the definition of **I5**, so that it holds at each point of execution.

5.6 Object Cycles Require No Special Treatment

As we already said in the introduction, because reference counts are upper bounds on the number of actors which have a stake in an object, rather than the number of paths leading to the object, the treatment of object cycles requires no special treatment: As the simplest case, assume a cycle of objects all owned by the same actor. When these objects become globally unreachable, then their *LRC* will become 0, and the owning actor will collect them. In the general case, assume a cycle of globally unreachable objects owned by n different actors. When one of these actors performs GC, it will trace and find that objects from that cycle are no longer reachable from it and will send to the owner of each foreign object from the cycle the appropriate decrement message. When all n actors have completed one GC cycle, all the objects from the unreachable cycle will have a *LRC*=0, and will then be collectable by their owner.

6 EXPERIMENTAL EVALUATION

In this Section we evaluate these design choices by studying the performance of our Orca implementation in terms of responsiveness, scalability, message overhead and footprint. We compared the execution of "equivalent" actor programs written in Pony, Erlang and Akka. For Akka we used C4 (Tene et al. 2011), G1GC (Detlefs et al. 2004), Concurrent Mark–Sweep, and Parallel OldGen. We found that for our particular benchmarks C4 performed better than the other JVM collectors, and therefore we only report C4 for Akka in the paper, but show results for the other collectors in Appendix A. Moreover, we compare execution of Pony programs with and without garbage collection.

Given that Pony is still a young language with a small set of users (compared to those of Akka and Erlang), the set of programs available is still quite small. We have therefore used a small set of micro-benchmarks for our tests. While these micro-benchmarks are not enough for a comprehensive evaluation of a garbage collector, they show that Orca can be efficiently implemented in an actor language.

Reproducibility

We obtained our results by running on a parallel machine (64 cores), used in exclusive mode, and with installations of the three languages and the various versions of the GC. We were unable to bundle installations of all these and so have been unable to provide a successful artifact. Nevertheless, we provide here links to the source code of Pony implementation used for this evaluation, as well as to the sources for the Pony, Akka and Erlang benchmark programs, and some of the scripts to run them. These can be found at https://github.com/jupvfranco/ponyc. The latest version of Pony is: https://github.com/ponylang/ponyc.

6.1 How We Evaluate Orca

We use small benchmarks either designed by ourselves or taken from well-known benchmark suites (Imam and Sarkar 2014; team 2017; The Computer Language Benchmarks Game 2017).

Our choice of benchmarks was guided by the following principles: they should be actor-based, send many messages, and have high memory pressure. Moreover, they should cover other characteristics which affect the performance of our protocol: different message sizes, different object lifetimes, different number of actors, *c.f.* Figure 15.

We used existing Akka and Erlang sources as much as possible. When porting these programs to Pony, we tried to stay faithful to the original programs. Nevertheless, differences in available libraries necessitated some adaptations. Naturally these comparisons are rough, and also include non-GC factors. We use these benchmarks:

Category	Property	trees	trees′	rings	heavyRing	mailbox	serv.Sim.
Program	Different lifetimes	✓	✓	✗	✗	✗	✗
	Large allocation	✓	✓	✗	✓	✗	✓
	Many Messages	✓	✓	✓	✗	✓	✓
	Large messages	✗	✗	✗	✓	✗	✗
	Sharing	✗	✗	✓	✓	✗	✗
	Mutator work	✓	✓	✗	✓	✗	✓
Protocol	Responsiveness	✗	✗	✗	✗	✗	✓
	Scalability	✗	✓	✓	✗	✓	✗
	Message Overhd.	✓	✓	✗	✓	✓	✗
	Footprint	✓	✓	✓	✓	✓	✗

Fig. 15. Characterisation of benchmarks.

trees: Based on the GCBench benchmark (Ellis et al. 2017) and code based on the binary trees benchmark from the CLBG (The Computer Language Benchmarks Game 2017).

trees′: Variation of **trees**; spawns many more actors, increasing parallelism, and the number of trees created.

rings: A ring of actors that pass around a number for *n* times. We adapted this from (?) to create several rings running at the same time (to increase parallelism).

heavyRing: A ring of actors passing around large (object) graphs. Developed by us to stress Orca's weak points.

mailbox: A highly contended mailbox, from the CAF benchmarks (team 2017).

serverSimulation: We simulate a server implementation based on a number of actors processing messages in parallel. We measure the differences between processing time of pairs of message, and visualise the jitter in processing times. Global pauses introduce jitter across all servers. Long pauses mean larger outliers.

Appendix A shows some important characteristics of our benchmarks, such as the number of actors and the total amount of memory allocated, and specify the heap sizes used. We benchmark on an AMD Opteron 6276 with 32 cores (2 hyper-threads per core), 8 NUMA nodes, 128GB RAM, 16KB L1, 2MB L2, 6MB L3, running Debian 8 (jessie). We use Pony 0.8.0, Erlang 18.3, and Scala 2.11.6 (openjdk 1.8). We spent a limited amount of time experimenting with tuning the JVM, C4 and other collectors, and performed no tuning of Pony programs.

We used the built-in telemetry of C4, JVM, and we extended the Pony runtime with telemetry information. We used SystemTap to obtain GC times for Erlang, and Erlang's built-in telemetry to measure copying on message sending. As in (Blackburn et al. 2004), we set the heap size to 3 times the minimum required to run the benchmark (maximum live set size) on the JVM—the latter measured when running with parallel collector, using a similar methodology of (Singer et al. 2010). Note that it is not possible to set the heap size for Orca and Beam. For the JVM results we used the flags `-verbose:gc`, `-XX:+PrintGCDetails`, and `-XX:+PrintGCTimeStamps`. We set the same initial and maximum heap size (using the flags `-Xms` and `-Xmx`). For Erlang, we obtained the overhead of message sending and copying to a receiver using built-in telemetry. We obtained the wall-clock GC times through SystemTap (SystemTap 2017) probes for `gc_major__start`, `gc_major__end`, `gc_min-or__start`, and `gc_minor__end`. For Orca, we used its built-in support for telemetry which gives what

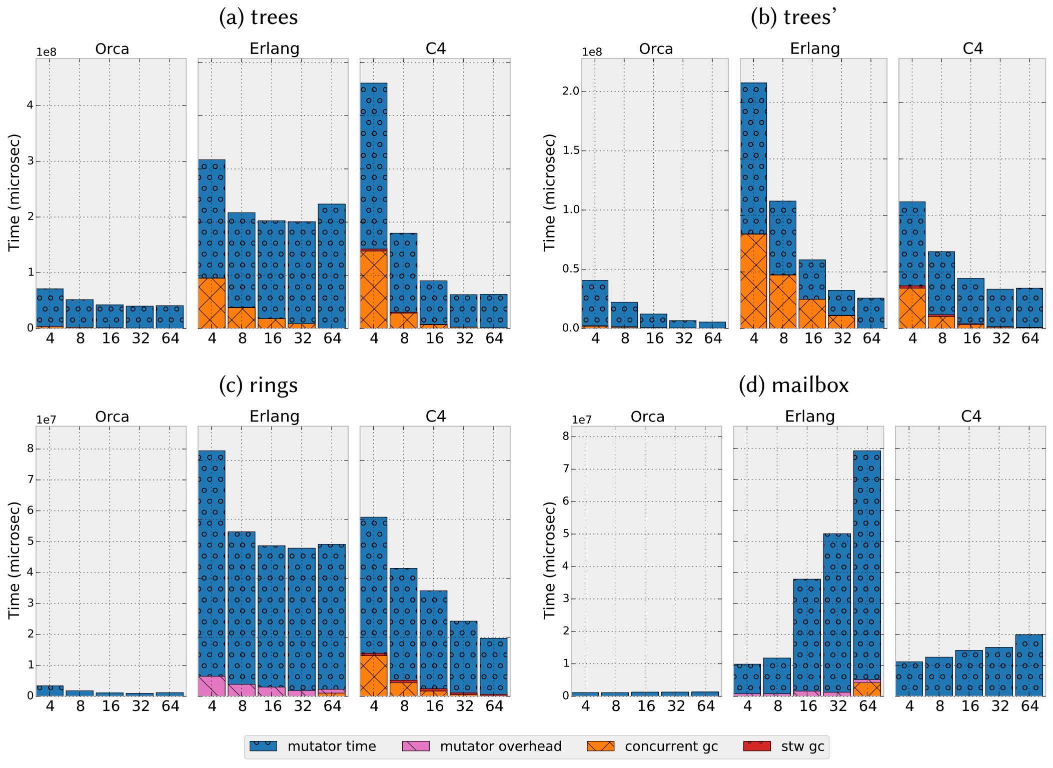


Fig. 16. Strong scalability on 4–64 cores. (stw=stop-the-world.)

and how many messages are sent, time spent on GC, scanning, allocated bytes, etc. We use the unix time command to get total execution time and footprint (maximum resident set size). Telemetry overhead for Pony and Java is very low—1%–2% in our tests, but quite high for Erlang. Thus we report for Erlang: (total execution times without telemetry)× % of copy and GC times measured with telemetry. Finally, we used `numactl` to restrict core counts and select cores of close proximity. We repeated each benchmark 10 times and report averages in our plots and standard deviations in Appendix A.3.

6.2 Scalability

Figure 16 shows execution times for **trees**, **trees′**, **rings** and **mailbox**, in Pony, Erlang, and Akka, over a variable number of cores — from 4 to 64 — so that we test scalability.

The measurements are in *microseconds* — smaller is better. We keep the same scale for one program across different languages, but differ it across different programs. We measure stop-the-world time, concurrent garbage collection time, mutator overhead time and mutator time.

- *stop-the-world time*: All the cores are doing GC or waiting for GC to finish. This is is zero for Orca or Beam. For the JVM-based GC's, the telemetry gives the start and end times of stop-the-world steps; we calculate the intervals, and sum them up.
- *concurrent garbage collection time*: GC work concurrently with mutator threads. This is the sum of the times spent on concurrent GC by all threads/actors divided by the number of cores.

- *mutator overhead time*: extra work due to GC. In Beam, this is time copying message contents, in Orca, this is time spent tracing on message sent/receipt, and we take it as 0 for the JVM collectors.
- *mutator time*: time executing application code. This includes time when actors are idle. This is the total execution time (from start to end of application) minus the sum of three variables described above.

We were unable to measure mutator overhead in the different JVM collectors, using solely the available JVM flags. We leave this for future work. Thus application time presented for the JVM collectors include mutator overhead time.

Although we use a small set of micro-benchmarks to test scalability, it is interesting to observe that Pony, when running Orca, scales better than C4 and Erlang. Both **trees** and **trees´** benchmarks allocate large amounts of data; all four benchmarks send thousands of application messages; and all benchmarks, except **trees**, spawn many actors. We do not observe long mutation times in Pony even though these include object allocation, of which there are many in these tests. Similarly, we we do not observe long GC times, even though many of the objects created in **trees** and **trees´** are short-lived and thus trigger many GC cycles.

We believe we see better mutator times in Pony because, and not only: 1) Pony does not require the programmer to send extra messages for actor termination, contrarily to Erlang and Akka; and mainly 2) Pony uses the type system to avoid read and write barriers, whereas Akka uses JVM write barriers. While these benchmarks send many messages, they do not send large data structures. Thus they do not stress scanning upon message send and receive. We will discuss the overhead of message scanning later, however note Pony's message exchanging appears to be faster than Erlang, for programs that exchange many primitive values (no scanning required), such as **rings** and **mailbox** — the time spent on sending and receiving in Pony is not visible while we can observe it in Erlang. We believe this is due to the absence of selective receive in Pony, and because message sending uses a single atomic operation, and none on receiving.

Orca does not require any stop-the-world steps, and actors can collect their own heaps independently, without any synchronisation mechanisms. Moreover, it is enough for these particular benchmarks to schedule garbage collection only between behaviors, allowing us to avoid stack scanning. Finally, the mark-and-don't-sweep nature of the collector allows for better cache locality — as discussed previously, dead objects will not pollute the memory caches. All these properties together allow for collection significantly faster than in Akka. Note than in Erlang, where heaps are also collected individually, garbage collection time is also rather small.

6.3 Responsiveness

Figure 17 shows the results from our **serverSim** benchmark. Each server processes, in isolation, requests of the same size, and the number of servers equals the number of hardware threads on the machine (64). Each request causes the server to create and operate on a binary tree of depth 14.

We plot the difference between the time of completion of a request and its immediate precedessor; we take this as a measure of GC-induced jitter. Namely, garbage is accumulated forcing GC to trigger at various points, depending on GC protocol. Orca experiences very little jitter, whereas the performance of JVM-based collectors vary considerably more. We have tried different workloads, and we saw that Erlang has very little jitter for smaller workloads. However, for bigger workloads, Erlang behaves poorly because it spends more time on GC and more messages are accumulated in actors' queues. Results for smaller workloads are in Appendix A. Both Orca and Erlang are able to collect actors' heaps individually, meaning GC is evenly amortised, and each actor's collection touches only a few megabytes of memory as opposed to an entire heap. Moreover, Orca opts to

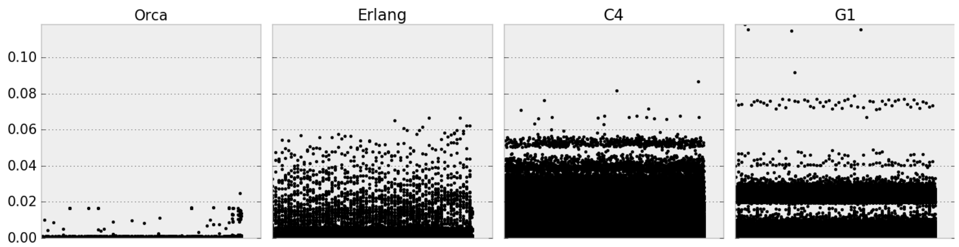


Fig. 17. Responsiveness. X-axis: request ID, Y-axis: Jitter/difference between finishing time (seconds) of subsequent requests. Java measurements are from a warmed-up VM and does not include JIT'ing.

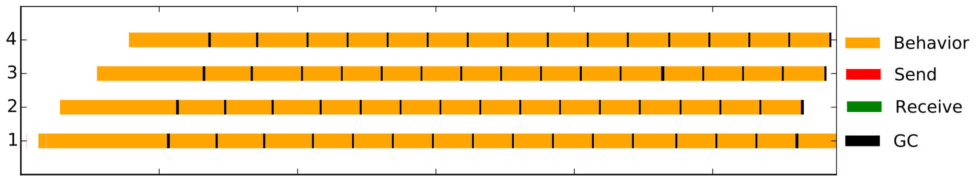


Fig. 18. CPU usage of **serverSim**. Y-axis: the different core IDs. X-axis the application's timeline in CPU cycles.

do many short collection cycles (this is shown eg in Figure 18 which shows this the apllication's timeline on a shorter version of the **serverSim** benchmark[3]) which lead to only small variation in response times.

6.4 Orca vs NoGC

We now compare Pony running Orca (Pony-Orca) with a version of Pony that does not garbage collect (Pony-NoGC). We ran **trees**, **trees′**, **rings** and **mailbox**, and measured execution times on different numbers of cores, using Pony-Orca and Pony-NoGC. With this experiment we do *not* intend to measure the overhead of Orca in Pony — this would not be the right approach for such results, as garbage collection reduces memory pressure, and the number of page requests to the operating system. Indeed, we ran the **trees** and **trees′** benchmarks with smaller arguments (less allocation), or otherwise they would crash without garbage collection. Instead, with this test, we show that Orca is very important for Pony's performance in memory intensive programs.

Pony-NoGC does not garbage collect at all — no actor nor object collection — and it does not send any GC related runtime messages, nor does any tracing upon message sending or receiving. For comparison, we switched off actor collection when running Pony-Orca, but kept Orca messages and scanning on message passing. Scalability results are in Figures 19.

On benchmarks that do not allocate much memory, such as **rings** and **mailbox**, garbage collection may affect performance (slightly in the former), or may not have any impact at all (in the latter). However, without garbage collection, benchmarks as **trees** and **trees′** show slow downs and poor scalability. This happens because the runtime needs to request more memory pages to the operating system, an expensive operation.

Memory pressure also affects Pony's responsiveness, as we can see in Figure 20. The **serverSim** benchmark is very similar to the **trees** benchmark, in that it allocates many trees. Over time, the

[3]We run it on fewer cores, with fewer actors, and fewer requests, for presentation purposes only. However this gives us similar behaviour to the "full" version, in the sense that GC is triggered many times for short periods.

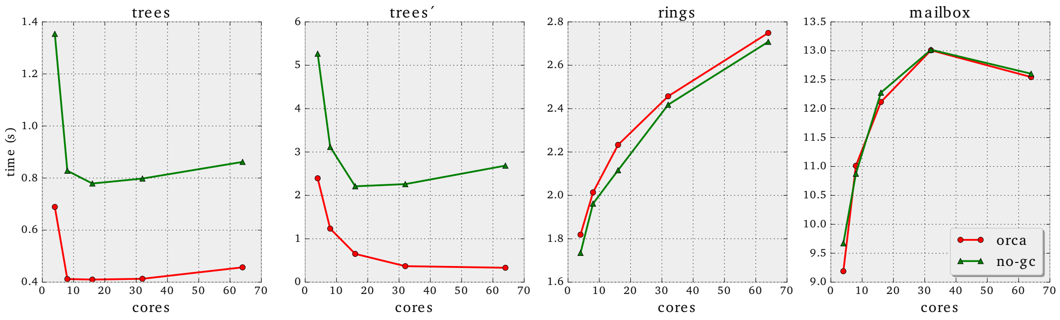


Fig. 19. Scalability of Pony-Orca and Pony-NoGC. Execution times (seconds) on 4–64 cores. Lower is better.

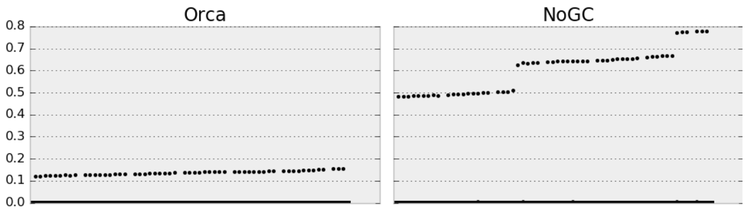


Fig. 20. Responsiveness of Pony-Orca and Pony-NoGC. X-axis: request ID, Y-axis: Jitter/difference between finishing time of subsequent requests. All times in seconds. Lower is better.

benchmark	#app	#INC	#DEC	#cycles	$\frac{\#INC}{\#app}$
1. trees	22369658	0	14	4194453	0.0
2. trees′	20884713	0	138	3847755	0.0
3. heavyRing	775	640	705	642	0.8
4. rings	64001668	0	1408	137	0.0
5. mailbox	496000996	0	992	1	0.0

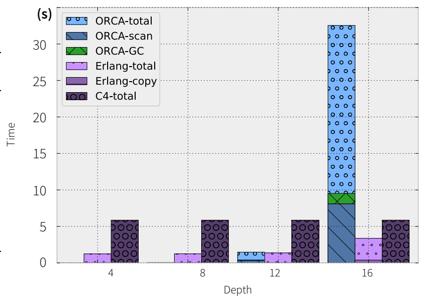


Fig. 21. Orca overhead due to tracing on message send and receive.

amount of memory allocated increases and Pony becomes less responsive, also due to the increasing number of page requests.

6.5 Mutator Overhead

As explained in previous Sections, Orca relies on tracing upon message sending and receiving, and on additional runtime messages. This extra work adds some overhead to the program's execution, which we call mutator overhead. We now discuss how much overhead Orca added to our benchmarks. Figure 21 shows the number of application, increment, and decrement messages sent for each benchmark, and the total number of GC cycles performed by all actors. Although 1, 2, 4, and 5 send many application messages, they never send increment messages. Most of the benchmarks require many decrement messages, however the ratio of decrement messages over number of GC cycles is quite small, is mainly due to the use of weighted reference counts.

Figure 21 also shows in microseconds the mutator time, mutator overhead (time spent on tracing upon sending and receiving) and GC time of the **heavyRing** benchmark. We ran this benchmark using 4 cores only[4] and we vary the depth of the tree being sent from 4 to 16. This benchmark stresses one of Orca's weaknesses: message tracing, and considering that all this benchmark does is receive a mutable object graph and forward it, it is expected to observe quite some overhead due to tracing. Indeed we obtained 21% of total execution time, for the largest graph. Notably, changing these structures to be passed as read-only completely avoids this overhead because no tracing is needed. Similar optimisations are not possible in Beam or on the JVM. We have run such experiments, passing immutable graphs instead, and as expected, we observed that the execution time for different depths keeps "constant" and always below 1 second. Passing immutable data structures in Pony, independently of their size, is almost for free.

Moreover it is also interesting to observe some GC time in this benchmark. Even though objects do not become unreachable, Orca still performs some GC cycles [5].

6.6 Footprint

Figure 22 shows the footprint — maximum resident set size — obtained using a core count of 64 for all the benchmarks, except for the **heavyRing** benchmark which was obtained using depth 16 for the tree being passed around. The reason why Orca is in general worse than C4 is that in the current implementation of Orca in Pony, GC cycles are triggered only *between* behaviours. Thus, they are not as frequent as in C4 causing a larger footprint. Adding support for GC in the middle of behaviours is orthogonal to the Orca protocol and the reason why it has not yet been implemented is the hassle of stack scanning. Moreover, Orca also sends more messages due to GC, and it keeps reference counting structures, which require extra space.

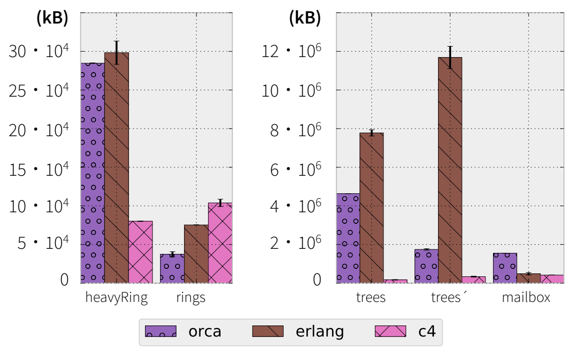


Fig. 22. Footprint (KB). Note difference in y-axis in two plots.

7 CONCLUSION

Orca represents a point in the GC design space where local tracing of messages both at the sender and the receiver side is used to avoid global GC stops. While this tracing is algorithmically costly, scanning can be performed locally in a thread without impacting the rest of the system. Thus, as the number of cores increases, the global impact of a single message send decreases. This can be contrasted with a stop-the-world GC where additional threads may increase the latency

[4]The ring is composed by 64 actors that behave sequentially — an actor only sends the tree after receiving it.
[5]Actors consume their references to the trees being passed and need to inform the owner that they no longer can reach it.

between triggering a collection and the start of the actual collection. Thus, even in an actor system where the entire heap is effectively shared by all actors, the necessary tracing can be performed concurrently, without blocking. Theoretically, a parallel stop-the-world collector should be able to perform a global collection more efficiently than Orca because each object would be marked only once (as opposed to twice per message send plus once per collection), and because of heap partitioning techniques such as generational GC operating on a global scale. For such techniques to be beneficial, however, their gains must outweigh any time lost waiting for threads to stop or due to negative coherency effects for propagating mark bits to cores sharing objects. Marking algorithms that can run in parallel with a mutator can reduce stop-the-world latency, but at the cost of less efficient marking which may slow down the mark-phase, and expensive write-barriers slowing down mutators.

Orca's design allows an application to be partitioned into as many local heaps as there are actors, and to be collected in parallel. This makes it relatively straightforward to extend Orca with an upper-bounded heap, actor-local thresholds for collection, more coarse-grained garbage collection like collecting all actors on a scheduler thread at the same time, or stop the entire system to collect garbage. Some optimisations, like collecting several actors together could be implemented by fusing their local heaps which would void the need for tracing when sending messages between these actors. While this may be helpful for groups of actors frequently exchanging messages, a downside is the complication of doing work stealing for load-balancing if stealing must keep all actors sharing a single heap on the same thread. We leave this for future work.

On a similar note, in the context of the Encore actor language, Yang and Wrigstad (2017) construct a layer on-top of Orca that supports concurrent operations on shared mutable state. This reintroduces write-barriers, but only for the isolated sections of a program that need this feature and relies on Encore's type system for lock-free primitives (Castegren and Wrigstad 2017). Since locks can be constructed from these primitives, it is possible to handle locks in a similar fashion, moving closer to thread-based concurrency. This too is an interesting direction for future work.

Acknowledgements

We are deeply grateful to Tim Wood for extensive discussions and suggestions about effective communication of our ideas. We thank Rakhilya Mekhtieva for her contributions to the formal proofs, Sebastian Blessing and Andy McNeil for their contributions to the initial implementation, as well as the anonymous reviewers for their insightful comments. This work was initially funded by Causality Ltd, and has also received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement 695412) and the FP7 project UPSCALE, the Swedish Research council through the grant Structured Aliasing and the UPMARC Linneaus Centre of Excellence, the EPSRC (grant EP/K011715/1), the NSF (award 1544542) and ONR (award 503353).

REFERENCES

ActorFoundry 2017. http://osl.cs.illinois.edu/software/actor-foundry/. (Retrieved July 2017).

Gul Agha. 1986. Actors: a Model of Concurrent Computation in Distributed Systems. *MIT Press* 11, 12 (1986).

Akka 2017. Akka – Scala actor library. http://akka.io. (Retrieved July 2017).

Joe Armstrong. 2007. A History of Erlang. In *History of Programming Languages (HOPL) III.* DOI:http://dx.doi.org/10.1145/1238844.1238850

Joshua S. Auerbach, David F. Bacon, Rachid Guerraoui, Jesper Honig Spring, and Jan Vitek. 2008. Flexible Task Graphs: a Unified Restricted Thread Programming Model for Java. In *Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES).* DOI:http://dx.doi.org/10.1145/1375657.1375659

David F. Bacon, Perry Cheng, and V. T. Rajan. 2003. A Real-Time Garbage Collecor with Low Overhead and Consistent Utilization. In *Symposium on Principles of Programming Languages (POPL).* DOI:http://dx.doi.org/10.1145/2502508.2502523

Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. 2004. Myths and Realities: The Performance Impact of Garbage Collection. *SIGMETRICS Perform. Eval. Rev.* (2004). DOI:http://dx.doi.org/10.1145/1012888.1005693

Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Einar Broch Johnsen, Ka I. Pun, S. Lizeth Tapia Tarifa, Tobias Wrigstad, and Albert Mingkun Yang. 2015. Parallel Objects for Multicores: A Glimpse at the Parallel Language Encore. In *Formal Methods for Multicore Programming*. LNCS, Vol. 9104. DOI:http://dx.doi.org/10.1007/978-3-319-18941-3_1

Denis Caromel and Ludovic Henrio. 2004. *A Theory of Distributed Objects*. Springer-Verlag. DOI:http://dx.doi.org/1007/b138812

Elias Castegren and Tobias Wrigstad. 2016. Kappa: Reference Capabilities for Concurrent Programming. In *European Conference on Object Oriented Programming (ECOOP)*. DOI:http://dx.doi.org/10.4230/LIPIcs.ECOOP.2016.5

Elias Castegren and Tobias Wrigstad. 2017. Relaxed Linear References for Lock-free Data Structures. In *European Conference on Object-Oriented Programming (ECOOP)*, DOI:http://dx.doi.org/10.4230/LIPIcs.ECOOP.2017.6

Sylvan Clebsch and Sophia Drossopoulou. 2013. Fully Concurrent Garbage Collection of Actors on Many-core Machines. In *Conference on Object-Oriented Programming Languages, Applications and Systems (OOPSLA)*. DOI:http://dx.doi.org/10.1145/2509136.2509557

Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny Capabilities for Safe, Fast Actors. In *Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE!)*. DOI:http://dx.doi.org/10.1145/2824815.2824816

Cliff Click, Gil Tene, and Michael Wolf. 2005. The Pauseless GC Algorithm. In *International Conference on Virtual Execution Environments (VEE)*. DOI:http://dx.doi.org/10.1145/1064979.1064988

David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-first Garbage Collection. In *International Symposium on Memory Management (ISMM)*. DOI:http://dx.doi.org/10.1145/1029873.1029879

Tamar Domani, Gal Goldshtein, Elliot K. Kolodner, Ethan Lewis, Erez Petrank, and Dafna Sheinwald. 2002. Thread-local Heaps for Java. In *International Symposium on Memory Management (ISMM)*. DOI:http://dx.doi.org/10.1145/512429.512439

John Ellis, Pete Kovac, and Hans Boehm. 2017. GCBench. (Retrieved July 2017). Developed by first two authors, modified by the third, from http://www.hboehm.info/gc/gc_bench.html.

Roger Henriksson. 1998. *Scheduling Garbage Collection in Embedded Systems*. Ph.D. Dissertation. Lund University.

Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *International Joint Conference on Artificial Intelligence (IJCAI)*. http://dl.acm.org/citation.cfm?id=1624775.1624804

Aaron Hillegass. 2011. *Objective-C Programming*. Addison-Wesley.

Shams M Imam and Vivek Sarkar. 2014. Savina – An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries. In *International Workshop on Programming based on Actors Agents and Decentralized Control (AGERE!)*. DOI:http://doi.acm.org/10.1145/2687357.2687368

Filip Pizlo, Ethan Blanton, Anthony Hosking, Petr Maj, Jan Vitek, and Lukas Ziarek. 2010. SCHISM: Fragmentation-Tolerant Real-Time Garbage Collection. In *Programming Language Design and Implementation Conference (PLDI)*. DOI:http://dx.doi.org/10.1145/1809028.1806615

Filip Pizlo, Athony L. Hosking, and Jan Vitek. 2007. Hierarchical Real-time Garbage Collection. In *Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. DOI:http://dx.doi.org/10.1145/1254766.1254784

Jeremy Singer, Richard E. Jones, Gavin Brown, and Mikel Luján. 2010. The Economics of Garbage Collection. In *International Symposium on Memory Management (ISMM)*. DOI:http://dx.doi.org/10.1145/1806651.1806669

Jesper Honig Spring, Filip Pizlo, Rachid Guerraoui, and Jan Vitek. 2007. Reflexes: Abstractions for Highly Responsive Systems. In *International Conference on Virtual Execution Environments (VEE)*. DOI:http://dx.doi.org/10.1145/1254810.1254837

George Steed. 2016. *A Principled Design of Capabilities in Pony*. Master's thesis. Imperial College London.

SystemTap 2017. SystemTap website. (Retrieved July 2017). https://sourceware.org/systemtap/.

The CAF team. 2017. https://github.com/actor-framework/benchmarks/. (Retrieved July 2017). https://github.com/actor-framework/benchmarks/

Gil Tene, Balaji Iyengar, and Michael Wolf. 2011. C4: The Continuously Concurrent Compacting Collector. In *International Symposium on Memory Management (ISMM)*. DOI:http://doi.acm.org/10.1145/2076022.1993491

The Computer Language Benchmarks Game 2017. The Computer Language Benchmarks Game. http://benchmarksgame.alioth.debian.org. (Retrieved July 2017).

Abhay Vardhan and Gul Agha. 2002. Using passive object garbage collection algorithms for garbage collection of active objects. In *International Symposium on Memory Management (ISMM)*. DOI:http://dx.doi.org/10.1145/512429.512443

Wei-Jen Wang. 2013. Conservative Snapshot-Based Actor Garbage Collection for Distributed Mobile Actor Systems. *Telecommunication Systems* 52, 2 (2013). DOI:http://dx.doi.org/10.1007/s11235-011-9509-1

Tobias Wrigstad, Filip Pizlo, Fadi Meawad, Lei Zhao, and Jan Vitek. 2009. Loci: Simple Thread-Locality for Java. In *European Conference on Object-Oriented Programming (ECOOP)*. DOI:https://doi.org/10.1007/978-3-642-03013-0_21

Albert Mingkun Yang and Tobias Wrigstad. 2017. Type-assisted Automatic Garbage Collection for Lock-free Data Structures. In *International Symposium on Memory Management (ISMM)*. DOI:http://dx.doi.org/10.1145/3092255.3092274

A EVALUATION (CONTINUATION)

Table 1 shows some important characteristics of our benchmarks. We list the number of application actors and the total amount of memory allocated. We also show (in Megabytes) the heap size with which we run them.

Benchmark	#actors	# MB allocated			Heap size (MB)		
		Pony	Akka	Erlang	Orca	Beam	JVM
trees	14	226859	282602	?			6000
trees′	131	163641	198885	?			7500
heavyRing	65	?	?	?	**NA**	**NA**	2G
rings	1281	0.01	5881.36	?			21
mailbox	497	0.02	**NI**	**NI**			**NI**

Table 1. Benchmarks characteristics. **NI** stands for Not Implemented, **NA** for Not Applicable, ? for unable to measure. While perfectly possible, we have not implemented support for upper-bounding the heap in Orca and there is as of yet no telemetry that allows us to simply sum up the size of each actor's local heap.

A.1 Scalability Results for all JVM collectors

Here we compare the scalability of the various JVM collectors with each other. Based on the results shown here we decided to report only C4 in Section 6. In Figure 23 we compare the results for **trees**, **trees′**, and **rings** with four different JVM collectors. We did not include **heavyRing** because although this benchmark spawns 64 actors, it has sequential behaviour, and thus it is not interesting to evaluate scalability.

A.2 All responsiveness results

In Figures 24 and 25 we complete the information from Figure 17. Namely, in Figure 24, we repeat the results for Orca, Beam, C4, and G1 from Figure 17, and in addition we show responsiveness results for CMS, G1GC and Parallel. Here servers process trees of depth 14. Interestingly, in this benchmark, G1GC seems to have better average responsiveness than C4, but worse outliers.

In Figure 25, we compare the responsiveness results of for Orca, Beam, C4, and G1. In this benchmark, the servers process requests of depth 8.

A.3 Standard deviations

In Table 2 we report the standard deviations of total execution times. Averages are reported in plots. The column heads denote tree depth.

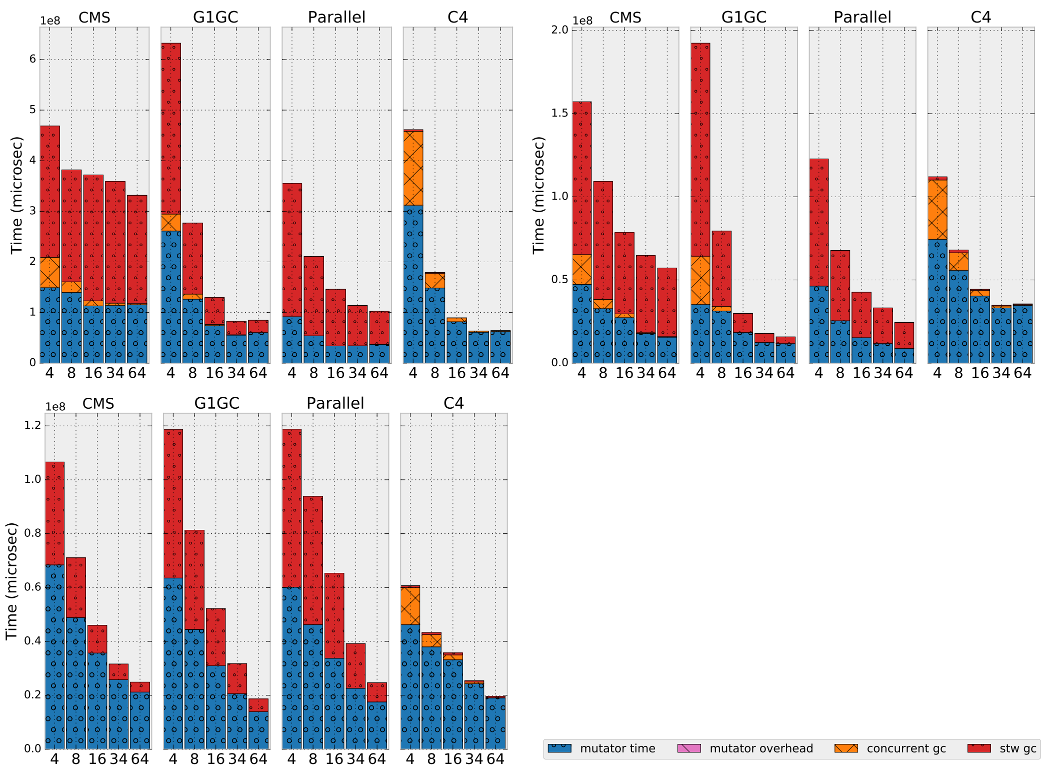


Fig. 23. Scalability: Comparison of the different JVM collectors with benchmarks **trees**, **trees′**, and **rings** (bottom). Measurements for 4–64 cores.

trees	4	8	16	32	64
ORCA	0.233	0.195	0.495	0.341	0.346
BEAM	3.537	3.685	6.111	3.092	1.462
C4	35.459	11.204	3.876	1.149	1.291

trees2	4	8	16	32	64
ORCA	0.203	0.131	0.108	0.215	0.06
BEAM	1.88	0.982	0.511	0.455	0.297
C4	1.899	1.496	1.034	1.516	1.668

ring	4	8	16	32	64
ORCA	0.021	0.028	0.096	0.248	0.285
BEAM	0.793	0.287	0.489	0.161	1.146
C4	1.378	0.878	1.464	1.209	1.335

mailbox	4	8	16	32	64
ORCA	0.574	0.164	0.149	0.105	0.063
BEAM	0.615	0.32	5.414	0.747	0.198
C4	0.894	1.892	1.011	0.794	1.026

Table 2. Scalability. Standard deviations of total execution times (in seconds) for different core counts.

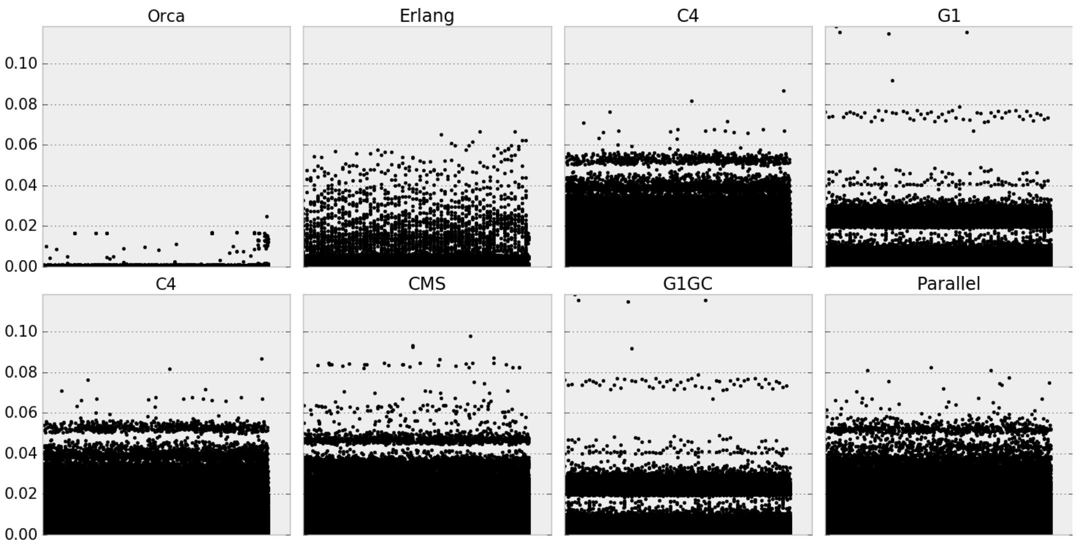


Fig. 24. Resposiveness results for all protocols. Requests to servers trigger the creation of trees of Depth 14. Results are in seconds. This Figure completes the plots in Figure 17

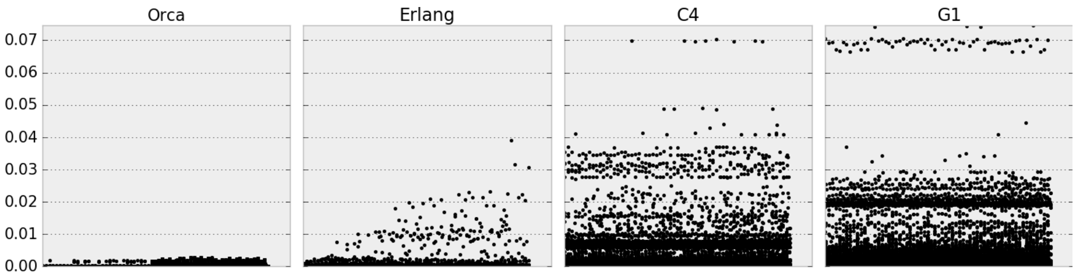


Fig. 25. Responsiveness results for Orca, Erlang, C4, G1, and Go. Requests to servers trigger the creation of trees of Depth 8. Results are in seconds. This Figure completes the plots in Figure 17.