Ownership-Based Isolation for Concurrent Actors on Multi-core Machines

Olivier Gruber and Fabienne Boyer

Université de Grenoble
first.last@imag.fr

Abstract. The deep copy of messages that traditionally ensures the memory isolation of actors severely hinders the performance of actor systems on multi-core machines. Several approaches have been proposed in the state of the art to circumvent this overhead, but they require to choose two properties out of the three desired ones: safety, programmability, and efficiency. In this paper, we introduce a novel runtime ownership model that supports the first memory isolation model of actors with these three properties—it is safe, developer-friendly, and efficient.

1 Introduction

In recent years, the actor paradigm has regained much attention as a concurrency model to exploit parallelism in Object-Oriented Programming Languages (OOPL). The argument [7] is that memory isolated actors that cooperate through messaging provides a concurrency model that scales well on multi-core machines. Unfortunately, achieving memory isolation of actors traditionally relies on deep-copying messages—a solution that severely hinders performance [15,4,14,5].

Several approaches have been proposed to circumvent this copy overhead. Some actor frameworks abandon safety for the sake of performance [10,6], advocating to pass messages by reference. These frameworks are efficient but unsafe. Other frameworks [16,7] retain safe messaging and achieve high performance through migrating messages, but at the expense of programmability because of the introduction of special type systems and certain constraints on the shape of permitted messages. A notable exception is the ActorFoundry framework [12] that preserves the programming model of a pass-by-value semantics while avoiding the copy overhead when it is safe to do so. Unfortunately, the proposed static analysis fails to identify all opportunities to optimize out deep copies, producing mixed results from a performance perspective.

This paper discusses a different path that does not require to choose between safety, programmability, and efficiency. With our approach, you can have your cake and eat it too—our solution is safe, developer-friendly, and efficient. Our approach is safe because we guarantee that actors remain strictly memory isolated. It is developer-friendly because messages have unconstrained shapes and we neither introduce any special types nor annotations. Furthermore, developers precisely control if an object belongs to a message or to an actor. To that end,

G. Castagna (Ed.): ECOOP 2013, LNCS 7920, pp. 281–301, 2013.

we propose a runtime ownership model based on the concept of first reachability from either a message or an actor. Put simply, an object belongs to a message (resp. actor) if it is first reachable from that message (resp. actor). This ownership model is simple to use and feels completely natural to object-oriented developers. Our solution that migrates messages is efficient compared to passing messages by reference with a typical overhead between 5% to 10%—an overhead that is independent of the shape and size of messages. Furthermore, our approach is multi-core friendly since all our runtime mechanisms are lock-free.

We implemented our complete proposal in the JikesRVM [1], a research high-performance Java virtual machine. We used the actor model of Kilim [16] as a starting point, modifying the actor model as little as necessary in order to integrate our novel memory isolation. We chose Kilim because it is regarded as the best performing framework for Java [9]. All our actor benchmarks ran almost unmodified on both frameworks, the sole meaningful difference being that we require explicit continuations while Kilim proposes automated continuations. Our approach requires modifying the Java virtual machine, adding a write barrier and an extra field in the object header that contains the reference of the owner of an object.

This paper is organized as follows. In Section 2, we quickly recall the main points of the actor paradigm and discuss related work, focusing on memory isolation. In Section 3, we introduce the design of our novel memory isolation based on a runtime ownership model. In Section 4, we illustrate the use of this ownership model through concrete examples. In Section 5 we evaluate our design and we conclude in Section 6.

2 Related Work

The actor paradigm defines an actor system as a collection of concurrent and autonomous entities, called actors, that cooperate through asynchronous messages. Some models propose that messages be sent to actors while other models propose that messages be sent to mailboxes, an actor owning one or more mailbox. In this case, mailboxes are shared objects whose references can be exchanged through messages. The execution model is event-driven, with actors reacting to received messages, one reaction at a time. While each actor is a single-threaded entity, multiple actors execute concurrently, thereby exploiting parallelism. For safe concurrent executions, actors are memory isolated entities, passing messages between actors by value. A straightforward design is to deep-copy messages [15,9,13], which severely degrades the performance of actor systems [9].

For the sake of performance, certain actor frameworks [10,6] abandon safety entirely, advocating a by-reference semantics and explicit deep copies. This approach expects developers to do the right thing, passing messages by reference when it is safe to do so and making deep copies whenever necessary. The approach is attractive for its peak performance and the fact that it preserves standard object-oriented programming practices. The main criticism is undoubtedly the assumption that developers do not make mistakes, which ultimately raises

the question of when can an actor system be trusted to be bug free and therefore safe to use.

The ActorFoundry [12] proposes an original approach that retains the safety of pass-by-value semantics while avoiding the overhead of deep copies when it is safe to do so. To achieve this goal, this framework relies on static analysis that achieves encouraging preliminary results. The analysis combines a novel live variable analysis along with context-sensitive call graph creation and field-sensitive points-to analysis. The approach is especially attractive because it preserves the traditional object-oriented programming model, avoiding the complex and error-prone task of annotating code or using special type systems. Unfortunately, the proposed static analysis still fails to identify all opportunities to optimize out deep copies, producing mixed results from a performance perspective.

Other actor frameworks advocate a pass-by-migration semantics that provides a safe zero-copy messaging. The idea behind migration is that a message can be accessed by only one actor at a time. To achieve this goal, these frameworks control object aliasing through special type systems or annotations [11]. For example, Kilim [16] advocates a static type checking mechanism based on linear types that only allows for tree-shaped messages. Scala [6] has recently proposed a new type system [7] that introduces safe migration. The proposed type systems is interesting because it is somewhat simpler and removes important limitations regarding permitted message shapes. The approach is certainly attractive but the presence of a dual type system raises several questions. First, it is unclear from published papers what are the implications of the restrictions on permitted message shapes. It is unclear if format translations are often necessary, and it they are, what are the performance implications. Second, it is unclear how complex the use of such type systems actually is for the average developer and how severely it impacts traditional object-oriented programming practices.

3 Ownership and Memory Isolation

During our search for an alternative path to provide memory isolation for actor systems, we went through two important design steps, which we retrace in the following two subsections. Both models are based on migration. Since the second step builds on the first, we felt that it helped the clarity of the paper to present the historical evolution of our design.

3.1 Allow Aliasing

The idea behind this first design is quite simple. Rather than trying to prohibit or tightly-control aliasing, like most approaches using special type systems strive to do, we want to allow totally unconstrained aliasing. Our motivation is to maintain object-oriented programming as developers understand it today. If we allow aliasing, we need to shift our focus on preventing the use of illegal references on migrated objects. To better understand illegal references, we should discuss the two different forms of aliasing that create illegal references when migrating

messages: references from the actor state into messages and references from messages to the actor state or into other messages.

The latter form of aliasing can be easily controlled through the use of coloring and a read barrier. We use colors as follows. An actor colors any new object that it creates with its color. As we deep migrate a message, we change the color of each migrated object from the color of its sending actor to the color of its receiving actor. Therefore, a simple read barrier on the color can detect illegal references. In the case of shared objects between two messages, the first migration operation will change the color of the shared objects and the second migration operation will detect an invalid color. In the case of a reference from a message to the actor state, the referenced part of the actor state will be migrated and colored, bringing us back to the former aliasing: the actor keeping references to migrated objects.

This form of aliasing is more difficult to handle properly, even though the principle is straightforward: any reference retained by the sending actor on a migrated object is illegal and must be therefore unusable. The challenge comes from finding these illegal references. Illegal references may be retained in local variables, in arguments of method invocations, and in objects (including arrays). A possible solution could be to scan the actor, garbage-collection style. This would include a complete scan of all objects reachable from both the actor itself and the thread stack of the current reaction. Since the overhead of such a scan would have to be paid at each send; we consider this approach impractical.

We advocate another path: we can allow illegal references to exist as long as we forbid their use. At first glance, the implementation seems rather simple, leveraging a read barrier to check the validity of references before they are used. First, we need a read barrier on reading references out of objects and arrays. Second, we need a read barrier on reading references out of local variables and arguments. But this is not enough since illegal references could also be found on the operand stack[1]. Indeed, any method invocation may send a message and therefore migrate some objects whose references might have been pushed on the operand stack of caller invocations, higher on the thread stack.

At first, this path seems impractical too. First, we can expect an important overhead because of the sheer number of read barriers. Second, the implementation is really delicate because of the necessary scan of the operand stack upon every return of a method invocation. Within an interpreter, one could possibly scan the operand stack after each method invocation, introducing a serious overhead. Within a high performance virtual machine using Just-In-Time (JIT) compilation, the operand stacks is spread across the hardware registers and spill areas in stack frames, complexifying even further the search for illegal references after every method invocation.

Although it seems that we reached a dead end with this design, a simple solution exists if we are to revisit the immediate nature of the send operation. If messages were to be migrated when the current reaction completes rather than

[1] We use Java parlance, focusing on the Java virtual machine for the sake of clarity, although the problem is absolutely not Java specific.

immediately when sent, there would be no need for most of the previous read barriers. In particular, we would not need to bother with any of the read barriers related to the thread stack: local variables, arguments, and operand stack. Since we would be migrating messages once the current reaction completed, the thread stack can be considered as empty as far as memory isolation is concerned. Indeed, when a reaction completes, we are back executing code from the actor framework, and since the framework is the trusted code base, we need not search for illegal references.

Therefore, we propose to introduce the concept of *tail migration*. Developers can continue to shape messages however they please and send them whenever convenient. Each send operation only records the reference of the message, constructing a list of messages that are pending migration. Upon the completion of a reaction, the system automatically processes the pending messages, migrating each one of them to its intended actor. Notice that inter-message aliasing would only be discovered at the time of the effective migration, relying on the above deep-coloring of migrated messages.

Even with tail migration, we must rely on the use of a read barrier on reading references out of objects and arrays. In Java, this means inserting a read barrier on the GETFIELD and AALOAD bytecodes, where GETFIELD loads an object reference from an object field and AALOAD loads an object reference from an array object. With this read barrier in place, illegal references can be retained by objects and arrays, but they cannot be used, so safety is guaranteed.

Even though safety is guaranteed, we must consider the following point. Since an actor can retain illegal references, it can potentially force large graphs of objects to stay alive in other actors. This suggests to extend the garbage collector so that it discovers illegal references and nullifies them. While nullifying illegal references solves the problem, it does not seem an appropriate solution from a programming perspective since it silently removes any trace of memory isolation violations, thereby concealing the illegal behaviors of certain actors. We propose to use a special bit pattern instead of null; this special bit pattern would be treated by the garbage collector as a null reference, but it would be treated by the read barrier as an illegal reference, raising an appropriate exception such as IllegalPointerException.

To summarize, this design preserves the traditional programming style of OOPLs and provides the safe migration of unconstrained graphs of objects. Unfortunately, we can expect a relatively high overhead since it relies on a read barrier [2,17]. Furthermore, we feel that the lazy discovery of illegal references induces a programming model that is just too cumbersome for developers. Indeed, cross-message references are only discovered when the reaction completes, not when they are created. Even worse, other illegal references are only discovered if there is an attempt to use them, which means that the threat of illegal pointer exceptions always remains. Furthermore, it is our experience that understanding the real source of these exceptions when they finally occur is a daunting task for most developers, therefore greatly limiting the usability of this approach in practice.

3.2 Control Aliasing

Our second design builds on the first, combining tail migration with a novel ownership model. Our ownership model defines two *owner classes*: the Actor class and the Message class. In other words, each actor and each message are individual owners. Each owner defines one ownership domain including the owner object and all the objects owned by that owner. Therefore, owner objects are created as owning themselves. Other objects are created free (not owned) and will remain free until garbage collected or until absorbed by an ownership domain. A free object is absorbed as soon as it becomes reachable from an owner object, directly or indirectly. In other words, the ownership propagates through reference assignments:

$$L.f = R; \tag{1}$$

When the field f of the left-hand-side object L, owned by an owner O, refers to a right-hand-side object R, the ownership propagates from L to R as follows:

1. **If the object L is free**, there is no ownership to propagate. If the object R was free, it remains free. If it was owned, its owner remains unchanged.
2. **Object L is owned and R is free**. When a free object is first referenced by an owned object, we propagate the ownership: R becomes owned by the owner of L.
3. **The objects L and R are owned**. The assignment is illegal unless L and R are owned by the same owner.
4. **The object L is owned and R is shared**. The assignment is always legal and there is no ownership propagation.

Even with isolated ownership domains, the concept of shared objects is necessary to model shared references to trusted language objects. For instance, mailbox objects are typically shared by actors to send and receive messages. Other language objects must also be shared such as Java enumeration, Java classes, or class loader strings. In general, it is also accepted that immutable objects ought to be shared. Shared objects are never absorbed, they remain free until garbage collected. However, it is important to assert that, aside from references to well-identified shared objects, ownership domains are entirely isolated. Any assignment attempting to create a reference across the boundary between two ownership domains would raise an *illegal assignment exception*, immediately identifying the source of a future illegal reference.

It is important to point out that the absorption of an object is a *deep absorption*, applying the same write barrier on all encountered references. Indeed, in the last case above, despite the fact that R is free, the graph of objects reachable from R might be composed of either owned or free objects. Free objects are absorbed but objects owned by other owners would represent an illegal situation forcing an exception to be thrown. It is also important to point out that once

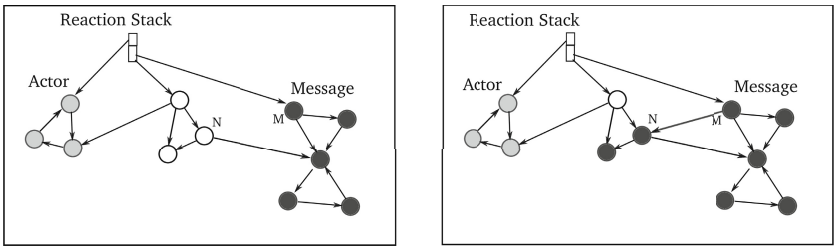

Fig. 1. Ownership Propagation

owned, an object remains owned by the same owner until it becomes garbage and it is reclaimed.

Figure 1 illustrates an example of ownership propagation. On the left-hand side, we have an actor, the stack of its current reaction, and a message M. There is also a small graph of free objects, only reachable from the reaction stack. Going to the right-hand side, the actor is adding the object N to the message M, which propagates the ownership of the message M onto the object N and those reachable from N.

Note that propagating ownership does not entail any object copy. It is similar to coloring but instead of propagating a color, we propagate the identity of the owner object. In Java, this means the above write barrier must be added to the PUTFIELD and AASTORE bytecodes, PUTFIELD stores an object reference in an object field and AASTORE stores an object reference in an array object. The simplest implementation is to have an extra hidden reference per object, called the *owner reference*. When the object is free, that owner reference is null. When the object is owned, that owner reference refers to the owner object.

Notice that local variables, method arguments, and free objects are allowed to refer to any object—free or owned. This is important because this allows a total programming freedom when manipulating the state of an actor or of a message. In other words, despite the fact that objects may be owned, either by the actor or by a message, the actor developers retain a complete programming freedom. For instance, an enumeration on a hash table would be a free object, even if the hash table is owned. The enumeration object will be using direct references on owned objects that are internal to the collection implementation. In other words, aliasing of owned objects through free objects is legal and not limited in any way. This model is correct, never endangering isolation, since we adopted a tail migration rather than an immediate migration when sending messages.

4 Examples

In this section, we illustrate the use of our ownership model through a simple yet complete example. The examples are written in Ownership-Kilim (O-Kilim), our actor framework based on Kilim [16]. The purpose of this section is two-fold. First, we want to illustrate the minimal differences between O-Kilim and

Kilim. Second, we want to highlight how natural our ownership model feels to object-oriented developers.

Like Kilim, we have three special classes: the Task class, the Mailbox class, and the Message class. The Task class is the concept of actor in Kilim. The three classes are part of the trusted code base, but only the Mailbox class is final. Mailboxes are shared across tasks, allowing tasks to send and receive messages. There are two key differences between Kilim and our proposal. First, O-Kilim relies on our ownership model. Second, Kilim advocates automated continuations, relying on blocking operations on mailboxes with an automated wind-unwind of the thread stack upon blocking and resuming tasks. O-Kilim assumes explicit continuations and run-to-completion reactions. The O-Kilim framework guarantees that each actor executes at most one reaction at any one time, irrespective of the number of mailboxes it is bound to. An actor binds to a mailbox by registering a listener, whose MailboxListener interface is given in Listing 1.1. Only one actor may bind to a given mailbox.

Developers are expected to extend the Actor class and Message class as they see fit. Instances of both the Actor and Message class are owners, each instance defining a separate ownership domain. Remember that owners are instantiated as owning themselves and therefore their constructors already execute within the confinement of their ownership domain. In Listing 1.1, we also show the simplest actor constructor that first binds to a given mailbox and then starts.

Listing 1.1. Server Actor

```
1  public interface MailboxListener<T> {
2    void onMessage(Mailbox<Message<T>> mb, Message<T> msg);
3  }
4
5  public class Server extends Task implements MailboxListener<Message> {
6    Mailbox<Message> mymb;
7    public Server(Mailbox<Message> mb) {
8      mymb = mb;
9      mymb.addMessageSubscriber(this);
10     start();
11   }
12   public void onMessage(Mailbox<Message> mb, Message msg) {
13     Mailbox<Message> niu = new Mailbox<Message>();
14     new Parser(niu);
15     niu.put(msg);
16     return;
17   }
18 }
```

In Listing 1.1, the server's reaction creates actors (Parser) that concurrently process received requests. Each request will be about parsing a text document into an in-memory W^3C document. Notice that messages can be forwarded

without hassle, one of the main performance advantages of migration. Line 15, we can see that forwarding a message is as simple as putting into a mailbox, which does not migrate the message immediately but remembers that it should. At the end of the reaction, line 16, the message will be migrated, without any overhead.

Notice also that mailboxes and actors are created as regular objects. Notice however that created actors are not referenced; in fact, actors are not shared objects and cannot be referenced across ownership domains. Furthermore, the reference to an actor is only legal within local variables and arguments belonging to a reaction on that actor. Hence line 14, the newly created actors cannot be referenced; indeed, any communication with an actor must happen through mailboxes.

Listing 1.2. Main Initialization

```
1  public static void main(String args[]) {
2    Mailbox<Message> mbserver = new Mailbox<Message>();
3    new Server(mbserver);
4    int maxClient = Integer.parse(args[0]);
5    for (int i = 0; i < maxClients; i++) {
6      Mailbox<Message> niu = new Mailbox<Message>();
7      new Client(i, niu, mbserver);
8      niu.put(new Message(Message.START, args[i+1]));
9    }
10 }
```

Listing 1.2 illustrates the corresponding main initialization of an actor system. We create one server actor and a certain number of client actors, giving to each client the mailbox of the server and a url of a document. Listing 1.3 shows the simple message class we use throughout this example.

Listing 1.3. Message Class

```
public class Message extends kilim.Message {
  public static final int START = 1;
  public static final int REQUEST = 2;
  public static final int RESULT = 3;
  int kind;
  Mailbox mb;
  String url;
  Document doc;
  Message(int kind, String url) { ... }
  Message(int kind, Mailbox mb, byte[] text) { ... }
  Message(int kind, Document doc) { ... }
}
```

Listing 1.4 shows the client actor, reading the text document from the url (line 17), requesting the server to parse the document (line 18) and absorbing the in-memory XML document when receiving it (lines 22 to 24). Notice the extraction (line 23) before the absorption (line 24). Indeed, the XML document is owned by the message, it must therefore be extracted before it can be absorbed by the actor. Also notice the use of a local variable to remember the document reference since the msg.doc field will be nullified by the extraction, preserving the invariants of our ownership model.

Listing 1.4. Client Actor

```
1  public class Client extends Task implements MailboxListener<Message> {
2    Mailbox<Message> mymb;
3    Mailbox<Message> mbserver;
4    int id;
5    Document doc;
6    public Client(int no, Mailbox<Message> mb, Mailbox<Message> mbs) {
7      id = no;
8      mbserver = mbs;
9      mymb = mb;
10     mymb.addMessageSubscriber(this);
11     start();
12   }
13   public void onMessage(Mailbox<Message> mb, Message msg) {
14     byte[] text;
15     switch (msg.type) {
16     case Message.START:
17       text = readDocument(msg.url); // read text document
18       mbserver.put(new Message(Message.REQUEST, mymb, text));
19       break;
20     case Message.RESULT:
21       Document tmp = msg.doc
22       if (filter(tmp)) {
23         extract(tmp);
24         doc = tmp;
25       }
26       break;
27     }
28   }
29 }
```

The extraction of an object from an ownership domain is a deep extraction, recursively forcing all reachable objects to be free again. Also, the extraction of an object from its ownership domain nullifies any remaining reference within that ownership domain that refers to objects that were just freed. This is important to preserve our invariants: (i) there is no references from an ownership domain to free objects, (ii)ownership domains remain fully encapsulated: there cannot

be any cross-domain references stored in reference fields of objects (including arrays).

However, notice (line 21 and 22) that messages can be freely manipulated from local variables and arguments, including references on internal objects. We believe this is one major advantage of our approach—one that preserves a completely natural programming model for object-oriented developers. The method filter (called line 22) would scan the document and decide if it should be absorbed or not. This filter method would be written exactly as it would be written in pure Java. The only constraint in our model is that cross-domain references, between two messages or between a message and an actor, are illegal. However, one can always extract an object from one domain and absorb it into another. This is exactly what happens line 23 that extracts the document from the ownership domain of the message and line 24 that absorbs the document in the actor state.

And there is nothing more complex than that in our model in order to control ownership and therefore ensure the safe isolation of actors. The rule could not be simpler: just assign the reference of your object were it belongs first, then alias it through local variables and arguments as necessary to manipulate it. It is our experience that this model feels natural to object-oriented developers; they write code as they are used to.

5 Evaluation

To validate our novel design for software memory isolation based on the safe migration of unconstrained messages, we implemented O-Kilim on top of a modified JikesRVM—a high performance research virtual machine for Java. We modified the virtual machine to include our ownership and we implemented the O-Kilim framework on top, providing tasks (actors) and mailboxes with a tail-migration semantics. The execution model is event-based, one actor reacting to only one message at a time, with each reaction running to completion. However, our framework may use multiple worker threads to execute multiple reactions concurrently across multiple actors.

We modified the JikesRVM version 3.1.0, implementing our write barrier in both the baseline compiler and the optimized compiler. All given numbers are obtained with the optimized compiler, with the O2 optimization level for both the boot image and the benchmark code. The JikesRVM is setup so that it does not use dynamic recompilation at runtime. In other words, Java code is compiled only once and runtime statistics are turned off. All benchmarks are run with a warmup run that forces all Java code to be compiled at the O2 level. Then, we measure ten successive runs, during which there is no longer any compilation overhead since we turned off dynamic recompilation. The given numbers are the mean average of the ten timed runs. We used the Immix garbage collection, with a maximum heap size of 2GB.

All experiments were run on an HP-Z400, with an Intel(R) Xeon(R) CPU W3520@2.67GHz, 64bit, 4 cores, 8 threads, 8GB RAM with memory bus at

2.4GHz, 4x32KB L1 for both data and instructions, and 4x256KB L2, and 8MB L3. Ubuntu 12.04 is installed, 64bit version, with the Linux kernel version 2.6.35.

5.1 Actor Benchmarks

Although there is no official benchmark for actor frameworks [9], three typical benchmarks have been consistently used in the literature [9,16,8].

1. The *Ring benchmark* evaluates message passing, a ring of actors are passing a token message around the ring a certain number of times. Our default setting is to pass 4 million times the token around 500 actors.
2. The *Chameneos benchmark* [8] has N creatures, called Chameneos, that have a cyclic behavior where they request a broker to arrange a meeting between two creatures and when the meeting takes place, the two creatures play together for a while and change color. Our default setting is to have 20 creatures, meeting one million times. To simulate playing, we have creatures change color a certain number of times at each meeting.
3. The *QuickSort benchmark* illustrates a concurrent sorting service based on a client-server pattern. The service is implemented by an actor that receives multiple requests from client actors, each one to sort one collection. Concurrent requests are processed in parallel. For each request, the service creates an actor and forwards the collection to sort. The created actor sorts the collection and sends back the sorted collection to the client. Each client has a cyclic behavior: creates a collection, requests that the collection be sorted, waits for the sorted collection, and then scans the sorted collections to check that it is indeed sorted. We use 500 clients that each issue 100 requests to sort a collection with 100 elements. The collection is a Java list of comparable objects.

With these benchmarks, we can actually establish that our framework has a performance behavior that is comparable to Kilim, despite the slight changes in the actor model: run-to-completion reactions and the presence of our ownership model. Having established the soundness of our prototype with respect to one of top-performing actor frameworks [9], we can feel confident that our comparison of different memory isolation schemes is meaningful.

First of all, it is important to assess how similar the benchmarks are when running them on O-Kilim versus Kilim. The benchmarks are not only algorithmically the same, but they are almost identical syntactically since we have preserved the Kilim concepts of Tasks (actors) and Mailboxes, almost unmodified. Rather than offering a blocking operation to get messages from a mailbox, we offer a callback mechanism to notify an actor that a message is available, triggering the actor's reaction[2]. This is actually the only syntactical difference as our ownership is transparent, as illustrated in our example in Section 4.

[2] For all benchmarks, explicit continuations were trivial, something that might not always be true for all applications.

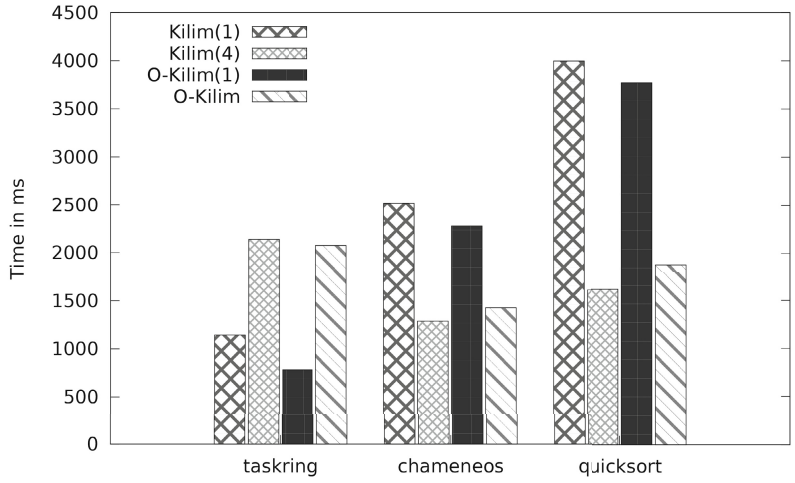

Fig. 2. Unsafe O-Kilim vs Unsafe Kilim

In Figure 2, we ran our benchmarks on Kilim (version 0.6) and on O-Kilim, both sending messages by reference. In other words, we measure the Java performance of both frameworks, providing no memory isolation in this first experiment. We ran all benchmarks with either one or four worker threads, a worker thread being a Java thread solely used to run actors' reactions. At first glance, we can observe that both frameworks behave remarkably the same across all benchmarks.

Regarding the Ring benchmark, both frameworks are unable to benefit from multiple workers as the benchmark has no builtin parallelism. Unsurprisingly, it is more efficient to schedule a sequence of reactions on a single thread than passing them around multiple threads, experiencing the delays necessary to wake up worker threads that are sleeping. However, both frameworks are able to benefit from using multiple worker threads on the two other benchmarks (the Chameneos and QuickSort). We can notice that single-threaded performance are close across all benchmarks and so are the achieved speedups with four workers. From these performance figures, we deduce that our O-Kilim prototype is a fair vehicle, compared to Kilim, in order to evaluate different memory isolation schemes.

Focusing on memory isolation, we compare different message passing schemes in Figure 3. The benchmarks are the same as before, but we run them on a single-threaded version of the O-Kilim framework. The rationale is that the overhead of memory isolation is a single-threaded overhead since it incurs no locking and no concurrent behavior. The *by-ref scheme* means that we ran the benchmarks on the unmodified JikesRVM, running the O-Kilim framework that passes messages by Java reference. This is equivalent to what we showed in Figure 2 under O-Kilim numbers. The *migration scheme* means that we ran the benchmarks on a modified JikesRVM to include our ownership model, running the O-Kilim framework. We

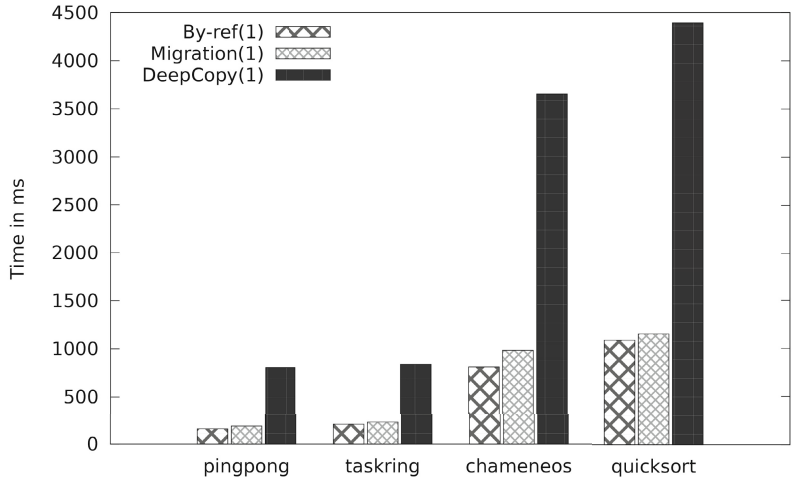

Fig. 3. Messaging Overheads

discuss the details of the corresponding modifications of the JikesRVM in the next subsection. The *deep-copy scheme* means that we ran the benchmarks on a modified JikesRVM in which we added an optimized deep-copy mechanism, but there is no write barrier of course. To implement this deep copy, we use the low-level capabilities of the JikesRVM to efficiently scan and clone objects, combined with the use of a hash table to preserve cycles when deep copying.

These performance figures confirm the established belief [9,12,7] that the deep-copy scheme severely hinders performance. This is even the case despite the fact that we rely on an optimized deep copy and not the costly Java serialization [13]. The Table 1 summarizes the overheads of the Figure 3. All measurements are done with the O-Kilim framework and the same benchmarking code, only modifying the semantics of the send operation in the O-Kilim framework.

Table 1. Deep-copy vs Migration Overheads

-	**Ring**	**Chameneos**	**QuickSort**
By-ref	783ms	2279ms	2505ms
Deep-Copy	3190ms	5072ms	8568ms
overhead	247%	122%	265%
Migration	917ms	2491ms	2761ms
overhead	17.11%	9.3%	10.21%

The Ring benchmark measures a pure messaging overhead since each actor is creating a message that it passes to the next actor down the ring. Consequently, this benchmark gives us an estimate of the overhead of creating and

sending small messages—a single object containing a single integer field (primitive type). The deep-copy overhead is high, even though the messages are small. The reason is that there is no processing at all to balance any messaging overhead. The same is true for migrating message, so the Ring benchmark measures our worst overhead, which is around 20%. The Chameneos benchmark shows the positive effect of adding processing. Both the deep-copy and migration benefits from added processing, with overheads that are cut in half. However, the QuickSort benchmark shows that the deep-copy overhead soars with larger messages whereas the migration overhead remains identical, around 10%.

From these figures, we confirm previously published results [15,4,14,5] stating that a deep-copy approach is much more costly than migrating messages. We focus now on characterizing more accurately our overhead, changing the amount of processing as well as the shape and size of the processed data. First, we will change the amount of processing in the Chameneos benchmark, varying the number of times meeting creatures are changing colors.

Table 2. Varying Processing in Chameneos Benchmark

Color Changes	**1**	**50**	**100**
Time(1W)	985ms	2491ms	3940ms
overhead	21.45%	9.30%	4.45%
Time(4W)	1345ms	1583ms	1875ms
overhead	14.37%	10.85%	0.4%
ratio	1.36	0.63	0.47

The performance figures are in Table 2. The *Time(1W)* line gives the execution times for the Chameneos benchmark, on a single worker, with 20 creatures, changing colors at each meeting either once, fifty times, or one hundred times. Changing color is a simple method with two nested switch with 3 cases each, deciding the new color of a creature based on its current color and the color the other creature had when the meeting began. The added processing does not entail any messaging, it is just a pure Java method manipulating Java enums. The *Time(4W)* line gives the execution times for the same Chameneos benchmark but using four worker threads.

Notice our high overhead when creatures only change color once (very limited processing per message). We are back to an overhead of around 20%, as it was the case with the Ring benchmark earlier. Also notice that when processing per message is limited, our ability to leverage multiple workers is challenged—we actually execute 1.36 slower with four workers than with one. In contrast, notice the positive effect of adding processing, with decreasing overheads for 50 and 100 color change. Despite the fact that the added processing is really small our overhead drops significantly: adding 1.4 micro-seconds per meeting drops our overhead to 9.30% and adding 2.8 micro-seconds per meeting drops it to 4.45%.

Moreover, notice the positive effects on our ability to leverage multiple workers. With 50 color changes, we reduce the execution time by 0.63 with an overhead at 10.85%. With 100 color changes, we reduce the execution time by 0.47, with an overhead at 0.4%. We believe that these figures indicate that modern processors are able to absorb our isolation overhead through concurrent execution (both hyper-threaded cores and advanced superscalar architectures) when there is enough processing per message. An effect that we will be confirmed on a novel XML benchmark below.

But before moving to XML processing, we can question the validity of our remarks since we only added synthetic processing (a loop manipulating Java enums). Consequently, we also conducted similar experiments with the QuickSort benchmark, varying not only the shape and size of the sorted collections, but also the cost of the element comparison.

Table 3. Varying Processing in QuickSort Benchmark

Settings	**A**	**B**	**C**
Time(1W)	2761ms	4972ms	10609ms
overhead	10.21%	9.05%	8.66%
Time(4W)	1234ms	2250ms	3998ms
overhead	7.39%	3.11%	4.14%
ratio	0.44	0.45	0.37

The performance figures are in Table 3. All the three settings are on collections of one hundred elements, using the unmodified QuickSort benchmark. In the first setting, called A, each collection is a Java ArrayList and elements are a simple object, with one integer field. The comparison of elements is therefore cheap: comparing primitive integers. In the second setting, called B, the elements are still simple objects, but they contain a string rather than an integer. In fact, the string is the textual representation of the integer value they contained in the setting A. The goal is to increase the cost of comparing elements, now comparing strings rather than primitive integers. In the third setting, we change the implementation of the collections, moving from Java ArrayList to Java LinkedList. The goal is to increase the overhead of accessing the elements of the sorted collection during the sort. The performance figures in Table 3 confirm our previous analysis. Across all sizes and shapes, our single-threaded overhead is around 10%. Furthermore, this overhead drops to 3% with parallel execution on four workers. Additionally, we maintain our speedup, executing up to 0.37 times faster on four workers—a speedup of 2.7 on 4 workers.

To confirm these statements, we conducted one last experiment with an *XML Parser benchmark*. We modified our QuickSort benchmark to parse XML documents concurrently rather than sorting Java collections. In other words, we retained the overall architecture of the benchmark, but we replaced our homegrown QuickSort algorithm with the use of the Java SAX parser. We use 100 clients that each issue 50 requests to parse a XML text document into a W^3C

document. The text document is a simple array of characters, while the W^3C document is a complex in-memory graph of objects, whose exact size and shape depends on the parser implementation.

Table 4. Varying Processing in XML Parser Benchmark

Settings	**Small**	**Medium**	**Large**
Time(1W)	1251ms	5664ms	17102ms
overhead	5.2%	7.68%	7.66%
Time(4W)	604ms	2926ms	7975ms
overhead	0.16%	9.13%	0%
ratio	0.48	0.51	0.46

The performance figures are in Table 4. Again, the performance figures confirm our analysis, even with a completely different processing and different data sets. The small XML document is really small with 7 tags, 5 attributes, accounting for 161 bytes. The medium document is more reasonable, with 69 tags and 32 attributes, accounting for 1217 bytes. The large document has 225 tags and 104 attributes, accounting for 5357 bytes. In this last experiment, with realistic processing, our overhead is around 5-8% with single-threaded execution and drops to zero overhead with four workers, while retaining an acceptable speedup, executing at twice the speed with 4 workers compared to one worker.

Overall, we believe that we have established that our memory isolation scheme induces a low overhead and does not impede concurrent execution. We further believe that we have reached our goal since we achieve low-cost memory isolation with minimal changes to the actor model and no change at all to the habits of object-oriented developers. However, we feel that more work is necessary on two fronts. First, a comparison analysis with solutions based on static analysis, that also preserve the object-oriented programming model, seems necessary. Second, the actor community must establish a representative benchmark suite for multi-core machines, with realistic concurrent applications that are representative of the concurrent and processing patterns targeted by actor systems.

5.2 WriteBarrier Benchmarking

In this section, we detail the design of our memory isolation mechanisms. In particular, we detail our ownership implementation, the associated write barrier. We explain how we integrated these mechanisms in the JikesRVM virtual machine. We also provide low-level performance numbers from the various benchmarks we ran in the previous subsection, allowing us to further explain the excellent performance figures we discussed earlier.

Our ownership model has a straightforward implementation. We added an *owner field* to the header of each Java object. This owner field holds the reference to the owner of the object, or null otherwise. This reference is known to the garbage collector, so a live object maintains its owner alive. The added overhead

to the garbage collection process is negligible. As explained earlier, the owner field is assigned by a deep absorption triggered by our write barrier.

The concept of write barrier is not new; it has been used intensively for supporting various garbage collection schemes [17]. In fact, the Immix garbage collector [3] we use in all our experiments uses write barriers for its own tracking of references. Traditionally, the reported performance overhead of write barriers [2,17] is typically around 2% to 6%. Traditionally, a write barrier is divided into a fast path and a slow path. The fast path handles the common case, requiring very few instructions; instructions that are inlined by the Just-In-Time compiler (JIT) on compiling the PUTFIELD and AASTORE bytecodes. The slow path handles the exceptional cases, requiring more instructions that are not inlined, therefore requiring that the fast path branches to the slow path when an exceptional condition occurs.

Listing 1.5. Write Barrier

```
static void writeBarrier(Object left, Object right) {
  if (right!=null) {
    Object lowner, rowner; // left and right owners.
    lowner = Magic.getObjectAtOffset(left,JavaHeader.OWNER_OFFSET);
    if (lowner!=null) {
      rowner = Magic.getObjectAtOffset(right,JavaHeader.OWNER_OFFSET);
      if (lowner!=rowner)
        writeBarrierSlowPath(lowner,left,rowner,right);
    }
  }
}
```

```
Assembly:
   ; ECX = right
   ; EDX = left
   TEST  ECX,ECX      ;
   JEQ                ; branch if right==null
   MOV EAX  -24[EDX]  ; EAX = lowner
   TEST EAX,EAX       ;
   JEQ                ; branch if lowner==null
   MOV EBX, -24[ECX]  ; EBX = rowner
   CMP EAX            ; compare lowner and owner
   JNE                ; branch to slow path if lowner!=rowner
```

Our write barrier is no exception to this split into a fast path and a slow path. Our fast path is given in Listing 1.5, both in Java and the corresponding IA-32 assembly code. The write barrier is written in Java because the JikesRVM is a meta-circular Java Virtual Machine (JVM), written itself in Java. We insert our write barriers like the garbage collector does, when the JIT compiler expands runtime services, after the final High-Level Intermediate Representation (HIR) is produced and before it is lowered to the Low-level Intermediate Representation (LIR). We have not touched the LIR translation to machine code. Since our

write barrier is inserted on assignment bytecodes (PUTFIELD and AASTORE), we have a left-hand-side object and a right-hand-side object, called left and right in the Java source. Notice the added owner field in the object header, holding the reference to the owner if any.

Our fast path captures the most common cases: (i) when free objects are assigned (left owner is null) and (ii) when legal references are assigned within an ownership domain (left and right owners are the same). The Table 5 gives the number of write barriers and the decomposition into fast and slow paths for various benchmarks, reusing previous settings. For Chameneos, we reused 20 creatures, 1000000 rendez-vous, and 50 color change per meeting. For the XML parser, we reused 100 actors, 50 requests over our medium-size document. For QuickSort, we reused 500 clients, 100 requests on a array list of 50 objects.

Table 5. Write Barrier and Absorption

	Ring	Chameneos	QuickSort	Parser
Messages	4M	4M	50,000	10,000
WriteBarrier	4M	16M	40M	21.4M
fast	1%	35%	32%	99%
slow	99%	65%	68%	1%
Absorption	4M	4M	10.2M	4.4M
Extraction	0	0	5.1M	2.2M
Overhead	17.11%	9.3%	10.21%	7.68%
total exec time	917ms	2780ms	6658ms	1488ms

Table 5 also shows other performance numbers related to the absorption or extraction of objects. We give the number of absorbed and extracted objects. We also recall the total execution times and the overhead induced by memory isolation. These numbers show that our various benchmarks do cover a wide mix of overheads. We cover from 4 million write barriers up to 40 millions, with different fast/slow ratios, from 1% up to 99% fast paths. The number of absorbed objects also varies greatly, from 1 million up to 10 millions, and so is the number of extracted objects, from 0 to 5 millions. This variety makes us believe that our performance figures given earlier are indeed representative of the quality of our proposed design for the safe memory isolation of actors.

6 Conclusion

This paper proposed a novel runtime ownership model that provides object-oriented developers with a completely natural programming model. Our ownership model does not require special types or annotations; the model is straightforward, an object belongs to the first owner it is reachable from. This rule is easy to understand and simple to put into practice in object-oriented

languages. Regarding the actor model it is integrated in, our approach has essentially one requirement: to use an event-driven execution that permits the tail-migration of messages. Our approach does not constrain the shape of permitted messages, any shape or size can be migrated at no cost. The overhead of our approach comes from the ownership management (write barrier, absorption, and extraction). This overhead is less than 20% across all our benchmarks and can typically be expected to be around 5%. This work suggests several directions for future work. First, it would be interesting to see how different static analysis techniques could help reduce the overhead of our ownership model. This is a research direction that we feel promising and that we intend to pursue. Second, we feel that it would be important for the actor community to develop realistic benchmarks that would help stronger evaluations of different actor systems.

Acknowledgments. We wish to thank Laurent Daynès for his comments on this work and valuable insights on Isolates and XIMI in particular. We also thanks the anonymous reviewers for their valuable comments.

References

1. Alpern, B., Augart, S., Blackburn, S.M., Butrico, M., Cocchi, A., Cheng, P., Dolby, J., Fink, S., Grove, D., Hind, M., McKinley, K.S., Mergen, M., Moss, J.E.B., Ngo, T., Sarkar, V., Trapp, M.: The Jikes Research Virtual Machine project: Building an open source research community. IBM Systems Journal 44(2) (May 2005)
2. Blackburn, S.M., Hosking, A.L.: Barriers: friend or foe? In: ISMM 2004: Proceedings of the 4th International Symposium on Memory Management, pp. 143–151. ACM, New York (2004)
3. Blackburn, S.M., McKinley, K.S.: Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance. In: ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM (June 2008)
4. Czajkowski, G.: Application isolation in the Java Virtual Machine. In: Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), pp. 354–366 (October 2000)
5. Golm, M., Kleinöder, J., Bellosa, F.: Beyond Address Spaces - Flexibility, Performance, Protection, and Resource Management in the Type-Safe JX Operating System. In: Proceedings of the 8th USENIX Workshop on Hot Topics in Operating Systems (HotOS), Elmau/Oberbayern, Germany, pp. 3–8 (May 2001)
6. Haller, P., Odersky, M.: Actors that unify threads and events. In: Murphy, A.L., Vitek, J. (eds.) COORDINATION 2007. LNCS, vol. 4467, pp. 171–190. Springer, Heidelberg (2007)
7. Haller, P., Odersky, M.: Capabilities for uniqueness and borrowing. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 354–378. Springer, Heidelberg (2010)
8. Kaiser, C., Pradat-Peyre, J.-F.: Chameneos, a concurrency game for java, ada and others. In: ACS/IEEE International Conference on Computer Systems and Applications. Book of Abstracts, p. 62 (January 2003)
9. Karmani, R.K., Shali, A., Agha, G.: Actor frameworks for the jvm platform: a comparative analysis. In: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java, PPPJ 2009, pp. 11–20. ACM, New York (2009)

10. Rettig, M.: JetLang (2008-2009), http://code.google.com/p/jetlang
11. Müller, P., Rudich, A.: Ownership transfer in universe types. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications, OOPSLA 2007, pp. 461–478. ACM, New York (2007)
12. Negara, S., Karmani, R.K., Agha, G.: Inferring ownership transfer for efficient message passing. In: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, PPoPP 2011, pp. 81–90. ACM, New York (2011)
13. Palacz, K., Czajkowski, G., Daynes, L., Vitek, J.: Incommunicado: Efficient Communication for Isolates. In: Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), pp. 262–274 (November 2002)
14. Java Community Process. Application Isolation API Specification, http://jcp.org/en/jsr/detail?id=121
15. Schäfer, J., Poetzsch-Heffter, A.: Jcobox: generalizing active objects to concurrent components. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 275–299. Springer, Heidelberg (2010)
16. Srinivasan, S., Mycroft, A.: Kilim: Isolation-Typed Actors for Java. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 104–128. Springer, Heidelberg (2008)
17. Yang, X., Blackburn, S.M.B., Frampton, D., Hosking, A.L.: Barriers reconsidered, friendlier still! In: Proceedings of the Eleventh ACM SIGPLAN International Symposium on Memory Management, ISMM 2012, Beijing, China, June 15-16 (2012)