Actor Frameworks for the JVM Platform: A Comparative Analysis

Rajesh K. Karmani, Amin Shali, Gul Agha
Open Systems Laboratory
Department of Computer Science
University of Illinois at Urbana-Champaign
{rkumar8, shali1, agha}@illinois.edu

ABSTRACT

The problem of programming scalable multicore processors has renewed interest in message-passing languages and frameworks. Such languages and frameworks are typically actor-oriented, implementing some variant of the standard Actor semantics. This paper analyzes some of the more significant efforts to build actor-oriented frameworks for the JVM platform. It compares the frameworks in terms of their execution semantics, the communication and synchronization abstractions provided, and the representations used in the implementations. It analyzes the performance of actor-oriented frameworks to determine the costs of supporting different actor properties on JVM. The analysis suggests that with suitable optimizations, standard Actor semantics and some useful communication and synchronization abstractions may be supported with reasonable efficiency on the JVM platform.

Categories and Subject Descriptors

D.1.3 [**Software**]: PROGRAMMING TECHNIQUES—*Concurrent Programming*; D.3.3 [**Software**]: PROGRAMMING LANGUAGES—*Language Constructs and Features*, Concurrent programming structures

General Terms

Languages, Performance

Keywords

Actors, Libraries, Frameworks, Java, JVM, Comparison, Semantics, Abstractions, Performance

1. INTRODUCTION

The growth of multicore computers has made it imperative for application programmers to write concurrent programs [1]. The dominant model for concurrent programming, popularized by Java, is a *shared memory* model: multiple threads working with a shared memory. The shared memory model is unnatural for developers, leading to programs that are error-prone and unscalable [2]. Not surprisingly, researchers and practitioners have shown an increasing interest in using *actor-oriented programming*. Some languages based on the Actor model include Erlang [3], E language [4], SALSA [5], Ptolemy [6] and Axum [7].

Ed Lee [2] has argued that in adopting a new language or library, programmers are motivated as much by its syntax as by its semantics. Perhaps for this reason, despite the development of a number of novel Actor languages, there continue to be efforts to develop Actor frameworks based on familiar languages such as C/C++ (*Act++* [8], *Broadway* [9], *Thal* [10]), Smalltalk (*Actalk* [11]), Python (*Stackless Python* [12], *Parley* [13]), Ruby (*Stage* [14]), .NET (Microsoft's *Asynchronous Agents Library* [15], *Retlang* [16]) and Java (*Scala* Actors library [17], *Kilim* [18], *Jetlang* [19], *ActorFoundry* [20], *Actor Architecture* [21], *Actors Guild* [22], *JavAct* [23], *AJ* [24], and *Jsasb* [25]).

In this paper, we analyze actor-oriented frameworks that execute on the JVM platform. Supporting actors through frameworks in a language with a different programming model can be complicated. Moreover, because the actor code runs on compilers and runtime systems for other languages, the resulting execution can be inefficient. Either to simplify the implementation, or to improve performance, many actor-oriented frameworks compromise one or more semantic property of the standard Actor model. For example, execution efficiency may be improved by unfair scheduling, or by implementing message-passing by passing references rather than copying messages. Our goal in this paper is to understand the semantic properties of Actor-oriented frameworks, what abstractions they support, and how they are implemented.

A programming frameworks can be analyzed along two dimensions: the linguistic support the framework provides for programmers, and the efficiency of executing code written using the framework. In case of the actor frameworks, linguistic support comes in two forms. First, by supporting the properties of the Actor model, a framework can enable scalable concurrency which facilitates compositional programming. Second, by providing programming abstractions that simplify expression of communication and synchronization between actors, a framework can allow programming idioms to be expressed in succinct notation.

As mentioned earlier, many actor-oriented frameworks compromise one or more of the semantic properties of actors. We discuss the significance of each of these properties in or-

PPPJ '09, August 27–28, 2009, Calgary, Alberta, Canada.

der to understand the impact of compromising the property from the "ease of programming" point of view (§3). We then describe some common communication and synchronization abstractions in actor frameworks (§4). Finally, we analyze the implementation mechanisms in Actor frameworks and study how the cost of providing actor properties may be mitigated (§5). Our analysis suggests that while a naïve implementation of actor properties may be highly inefficient, a sophisticated implementation of actor framework on JVM may provide efficient execution without compromising essential actor properties.

The main contribution of this paper is to provide a basis for understanding how various actor frameworks differ, both from a programmability point of view, and from a performance point of view. We hope that the results will guide developers in selecting a suitable framework and facilitate in the development of other actor-oriented frameworks.

2. OVERVIEW OF THE ACTOR MODEL

The Actor model is an inherently concurrent model of programming [26]. In the Actor model, systems comprise of concurrent, autonomous entities, called *actors*, and *messages*. Each actor has a unique, immutable *name* which is required to send a message to that actor. An actor name cannot be guessed but may be communicated to other actors. Each actor has its own mutable *local state*; actors do not share this local state with other actors–each actor is responsible for updating its own local state.

Actors communicate by sending *asynchronous messages* to other actors (see Figure 1). The receiving actor processes the messages one at a time in a single atomic step. Each step consists of all actions taken in response to a given message, enabling a *macro-step semantics* [27].

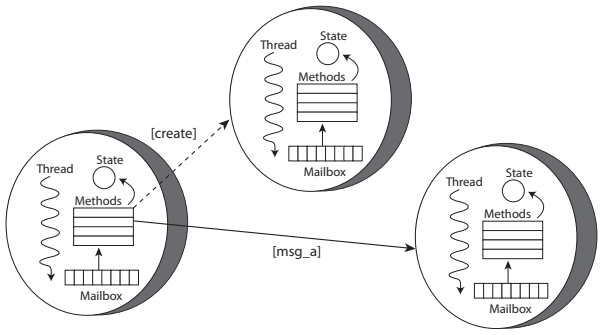

Figure 1: Actors are concurrent entities that exchange messages asynchronously.

The standard Actor semantics provides encapsulation, fair scheduling, location transparency (location independent naming), locality of reference, and transparent migration. These properties enable compositional design and simplify reasoning [27], and improve performance as applications and architectures scale [28]. For example, because actors communicate using asynchronous messages, an actor does not hold any system resources while sending and receiving a message. This is in contrast to the shared memory model where threads occupy system resources such as a system stack and possibly other locks while waiting to obtain a lock. Thus actors provide failure isolation while potentially improving performance.

Asynchronous messaging is a key source of nondeterminism in Actor programs: the order in which messages are processed affects the behavior of an actor. In many applications, application programmers want to prune some of the nondeterminism by restricting the possible orders in which messages are processed. Two commonly used abstractions that constrain the message order are *request-reply messaging* and *local synchronization constraints* (we will discuss these abstractions in §4).

3. ACTOR PROPERTIES

We discuss four important semantic properties of actor systems: encapsulation, fairness, location transparency and mobility. Using examples, we argue for the advantages of preserving these semantics in Actor implementations.

3.1 Encapsulation

Encapsulation is one of the core principles of object oriented programming. Preserving encapsulation boundaries between objects facilitates reasoning about safety properties such as memory safety, data race freedom, safe modification of object state. For example, Java is considered a memory-safe language because it hides memory pointers behind object references that provide safe access to objects (e.g. pointer arithmetic is not allowed). Memory-safety is important for preserving object semantics: it permits access to an object's state only using well-defined interfaces. In the context of the Actor model of programming, there are two important requirements for encapsulation: *state encapsulation* and *safe messaging*.

State Encapsulation.

An actor cannot directly (i.e., in its own stack) access the internal state of another actor. An actor may affect the state of another actor only by sending the second actor a message.

The code in Figure 2 shows an implementation of a counting semaphore in the Scala Actors. The main actor, in addition to sending an `enter`() message, executes `enter`() in its own stack. Because of a lack of enforcement of Actor encapsulation in the library, the code violates the Actor property that an actor may not directly access the internal state of another actor. As a consequence, in a multi-threaded, shared memory implementation of the Actor model, two actors can concurrently enter the critical section and thus violate the semantics of a counting semaphore.

In Kilim, actors have memory references to other actors' mailboxes. Again, this violates the desired encapsulation property. Actor implementations that enforce state encapsulation do so using indirection. Because such indirection also provides location transparency, we discuss this separately later in this section.

Safe Messaging.

There is no shared state between actors. Therefore, message passing should have call-by-value semantics. This may require making a copy of the message contents, even on shared memory platforms. In Scala Actors, Kilim, JavAct and Jetlang a message carries references to its contents on shared memory platforms, thus introducing shared state between the actors. These frameworks encourage the programmers to use immutable objects inside their objects or explic-

```
import scala.actors.Actor
import scala.actors.Actor._

object semaphore {
  class SemaphoreActor() extends Actor {

    ...

    def enter() {
      if (num < MAX) {
        // critical section
        num = num + 1;
  } } }

  def main(args : Array[String]) : Unit = {
    var gate = new SemaphoreActor()
    gate.start
    gate ! "enter"
    gate.enter
  }
}
```

Figure 2: A program written in the Scala Actors shows violation of state encapsulation which may cause two actors to simultaneously execute the critical section.

itly copy the objects in the program to avoid shared state. An alternate proposal is to add a type system based on linear types to enable safe, zero-copy messaging [18]. Such a type system is not part of currently available distributions. While such a type system would be useful, the current proposal may be too restrictive and complex to be widely used in practice.

We believe that both aspects of encapsulation are important for writing large-scale actor programs. Without enforcing encapsulation, the Actor model of programming is effectively reduced to guidance for taming multi-threaded programming on shared memory machines. It is difficult to provide semantics for, or reason about the safety properties of actor-oriented languages that do not guarantee encapsulation. For example, a macro-step semantics [27] simplifies testing and reasoning about safety properties in actor programs [29]. Without enforcing actor encapsulation, it would be incorrect to assume the macro-step semantics.

3.2 Fair Scheduling

The Actor model assumes a notion of *fairness*: a message is eventually delivered to its destination actor, unless the destination actor is permanently "disabled" (in an infinite loop or trying to do an illegal operation). Another notion of fairness states that no actor can be permanently starved. Note that if an actor is never scheduled, pending messages directed to it cannot be delivered. The notion of guarantee of message delivery is formulated to also imply that no actor is permanently starved.

Previous work [27] has shown that in the presence of fairness, reasoning about liveness properties in actor programs can be done in a modular fashion. For example, with fairness, composing an actor system A, with an actor system B that consists of actors which are permanently busy does not affect the progress of the actors in A.

```
import scala.actors.Actor
import scala.actors.Actor._

object fairness {
  class FairActor() extends Actor {

    ...

    def act() { loop { react {
      case (v : int) => {
        data = v
      }
      case ("wait") => {

        // busy-waiting section
        if (data > 0) println(data)
        else self ! "wait"
      }
      case ("start") => {
         calc ! ("add", 4, 5)
         self ! "wait"
      }
  } } }
} }
```

Figure 3: A program written in the Scala Actors showing an Actor "busy-waiting" for a reply. In the absence of fair scheduling, such an actor can potentially starve other actors.

Consider the program in Figure 3 in which an actor is "busy-waiting" for a reply from another actor, say a *calculator*. In the absence of fairness, calculator may never be scheduled (starvation). Therefore, the first actor never receives the desired reply, and the system cannot make progress. Similarly *non-cooperative actors* (i.e. actors running an infinite loop or blocked on an I/O or system call) can occupy a native thread and potentially starve other actors.

A commonly occurring example of this scenario is the composition of browser components with 3^{rd} party plug-ins, which are frequently blamed for browser crashes and hang-ups. Because browsers usually do not provide any fairness guarantee for the execution of different browser components (which include plug-ins), the problem of hang-ups is exacerbated.

Scala Actors, ActorFoundry, SALSA and Actor Architecture ensure fair scheduling of actors, but as always, such a guarantee is limited by the resource constraints of the JVM and the underlying platform.

3.3 Location Transparency

In the Actor model, the actual location of an actor does not affect its name. Actors communicate by exchanging messages; each actor has its own address space which could be completely different from that of others. The actors an actor knows could be on the same core, on the same CPU, or on a different node in a network. *Location transparency* provides an infrastructure for programmers so that they can program without worrying about the actual physical locations.

Because one actor does not know the address space of another actor, a desirable consequence of location transparency

Table 1: Comparison of Execution Semantics

	SALSA (v1.1.2)	**Scala Actors** (v2.7.3)	**Kilim** (v0.6)	**Actor Architecture** (v0.1.3)	**JavAct** (v1.5.3)	**ActorFoundry** (v1.0)	**Jetlang** (v0.1.7)
State Encapsulation	Yes	No	No	Yes	Yes	Yes	Yes
Safe Message-passing	Yes	No	No	Yes	No	Yes	No
Fair Scheduling	Yes	Yes	No	Yes	No	Yes	No
Location Transparency	Yes	No	No	Yes	Yes	Yes	Yes
Mobility	Yes	No	No	Yes	Yes	Yes	No

is state encapsulation. Location transparent naming also facilitates runtime migration of actors to different nodes, or *mobility*. In turn, migration can enable runtime optimizations for load-balancing and fault-tolerance.

Location transparency is supported by SALSA, Actor Architecture, JavAct, ActorFoundry and Jetlang, while in Scala Actors and Kilim, an actor's name is a memory reference, respectively, to the object representation of the actors (Scala) and to the actors' mailbox (Kilim).

3.4 Mobility

Mobility is defined as the ability of a computation to move across different nodes. In their seminal work, Fuggetta et al. classify mobility as either strong or weak [30]. *Strong mobility* is defined as the ability of a system to support movement of both code and execution state. *Weak mobility*, on the other hand, only allows movement of code (except for the initial state, which may be transferable). In actor systems, weak mobility is useful when migrating an *idle* actor (i.e. an actor that is blocked due to an empty mailbox), while strong mobility means that it is meaningful for an actor to migrate while it is still processing a message.

Because actors provide modularity of control and encapsulation, mobility is quite natural to the Actor model. Object-oriented languages may allow mobility at the level of objects but all thread stacks executing through the object need to be made aware of this migration. Moreover, when the stack frame requires access to an object on a remote node, the execution stack needs to be moved to the remote node to complete the execution of the frame and then migrated back to the original node [31].

At the system level, mobility is important for load balancing, fault-tolerance and reconfiguration. Previous work has shown that mobility is essential for achieving scalable performance, especially for dynamic, irregular applications over sparse data structures [32]. In such applications, different stages may require a different distribution of computation. In other cases, the optimal distribution is dependent on runtime conditions such as data and work load. We should point out that strong mobility (when augmented with discovery services) will enable the programmer to declaratively exploit heterogeneous system resources such as GPUs, DSPs, other task-specific accelerators, and high clock frequency cores.

Weak mobility is supported by SALSA, Actor Architecture, JavAct and ActorFoundry. Strong mobility can be provided in frameworks such as ActorFoundry, which allow capturing the current execution context (*continuation*) and enforce Actor encapsulation.

3.5 Discussion

Table 1 summarizes semantic properties supported by some of the more popular Actor frameworks on the JVM platform. Some frameworks improve execution efficiency by ignoring aspects of the Actor semantics. For example, as we mentioned earlier, actor message-passing entails sending the message contents by value. In languages such as C, C++ or Java, which can have arbitrary aliasing patterns, sending messages by value involves making a deep copy of the message contents to prevent any unintended sharing among actors. Deep copying is an expensive operation, even when performed at the level of native instructions (see §5).

Actor implementations such as Kilim [18] and Scala Actors [17] provide by-reference semantics for message-passing, and when required, leave the responsibility of making a copy of message contents to the programmers. We argue that such an approach creates a double jeopardy for the programmers. To begin with, they have to think in a message-passing Actor model, then they need to revisit their design in order to figure out which messages actually need to be copied, and finally, they need to ensure that the contents of these messages are actually copied.

The temptation to ignore encapsulation is stronger in the case of an Actor framework as opposed to an Actor language. For example, in order to ensure that an actor is unable to access the state of another actor directly, a language may provide an abstraction such as a mailbox address or a channel but implement it using direct references in the compiled code for efficiency. This is similar to how Java implements object references to abstract away pointers. In an Actor-based framework, such abstractions (or indirections) have to be resolved at runtime, something that is relatively inefficient.

Even the notion of scheduling fairness is subtle in Actor implementations. Note that the execution of actor programs is *message-driven*, i.e. actors are scheduled for execution on the arrival of a message, and actors are assumed to be *cooperative*, (i.e., an actor 'yields' control when no message is pending for it). However, nothing prevents an actor from executing an infinite loop, or blocking indefinitely on an I/O or system call.

In order to provide fair scheduling, the implementation requires some support for pre-emption. In frameworks where each actor is mapped to a JVM thread (also called the 1:1 architecture) the Actor implementation is as fair as the underlying virtual machine or operating system (on many platforms, a JVM thread maps to a native thread). In other cases, explicit scheduling by the frameworks may be required to support fair scheduling.

4. PROGRAMMING ABSTRACTIONS

We now discuss two useful programming abstractions for communication and synchronization in actor programs.

4.1 Request-Reply Messaging Pattern

Request-reply messages express the most common pattern of messaging and synchronization in actor programs. In this pattern, the sender of a message blocks waiting for the reply to arrive before it can proceed [33, 34, 35]. This RPC-like pattern is sometimes also called *synchronous messaging*. For example, an actor that requests a stock quote from a broker needs to wait for the quote to arrive before it can make a decision whether or not to buy the stock (see Figure 4).

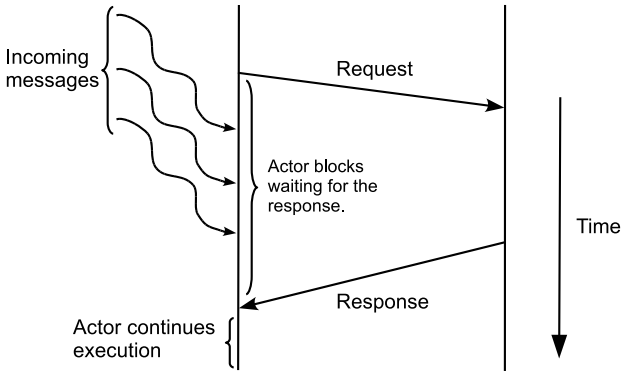

Figure 4: Request-reply messaging pattern blocks the sender of a request until it receives the reply. All other incoming messages during this period are deferred for later processing.

Without a high-level abstraction for request-reply messaging, the programmer has to explicitly encode the following steps in their program: an actor sends the request, waits for the reply to arrive and for each incoming message, the actor checks whether the message is a reply to the request or is another message that happened to arrive between the request and the reply.

Request-reply messaging is almost universally supported in Actor languages and frameworks. For example, it is available as a primitive in Scala Actors, SALSA, Actor Architecture and ActorFoundry.

4.2 Local Synchronization Constraints

Observe that each actor operates asynchronously and message passing is also subject to arbitrary communication delays; therefore, the order of messages processed by an actor is nondeterministic. However, sometimes an actor needs to process messages in a specific order. For example a single element buffer has to alternate the processing of *put* and *get* messages [36]. This requires that the order in which messages are processed is restricted. Synchronization constraints simplify the task of programming such restrictions on the order in which messages are processed. For example, they can allow the programmer to declare that an actor postpone the processing of a message until it receives some sequence of messages, or until a condition on the actor's state is satisfied.

Figure 5 shows the state diagram of a bounded buffer. This state diagram explains how a bounded buffer actor accepts different messages based on its current state.

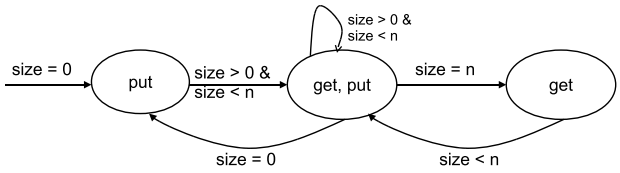

Figure 5: Bounded Buffer State Diagram

From a programming language design perspective, it is desirable to separate the specification of when a message is processed from the specification of how a message is processed. Such a separation makes the process of reasoning about the system more modular. In this case, method implementations specify how to respond to a message, and synchronization constraints specify when to respond to a message [36]. Separating synchronization constraints also makes it easier for programmers to change the implementation of the methods (in ways not affecting the state variables that are used in the synchronization constraints), or change the constraints, without affecting the other.

The following code is an example of a synchronization constraint in the ActorFoundry. Here, the method `disablePut` defines a synchronization constraint for the message `put`. Note that the constraint is required to be a side-effect free function (to enable efficient evaluation as well as simplify the semantics).

```
@Disable(messageName = "put")
public Boolean disablePut(Integer x) {
  if (bufferReady) {
   return (tail == bufferSize);
  }
  else {
    return true;
  }
}
```

In the above code, the constraint returns true if the `put` message cannot be processed in the actor's current state. At runtime, a message is processed if it is not *disabled* i.e., no constraint returns true for the message (see Figure 6). A disabled message is placed in a queue called *save queue* for later processing.

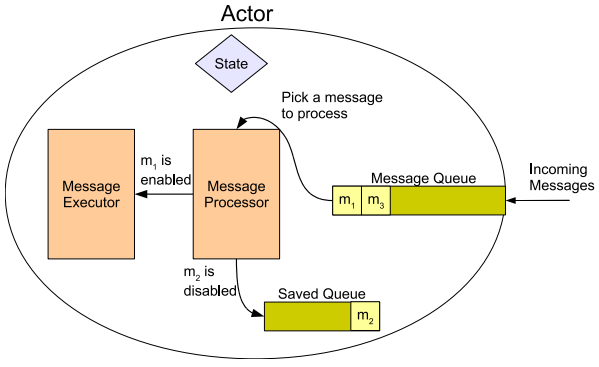

Figure 6: Implementation semantics of local synchronization constraints in ActorFoundry

Table 2: Comparison of Programming Abstractions

	SALSA (v1.1.2)	**Scala Actors** (v2.7.3)	**Kilim** (v0.6)	**Actor Architecture** (v0.1.3)	**JavAct** (v1.5.3)	**ActorFoundry** (v1.0)	**Jetlang** (v0.1.7)
Request-reply messaging	Yes	Yes	No	Yes	No	Yes	No
LSC	No	Yes	No	No	No	Yes	No
Join expressions	Yes	Yes	No	No	No	Yes	No

Whenever a message is processed successfully by an actor, it is possible that a previously disabled message (a message in save queue) is no longer disabled. This is because the state of the actor may change after processing a message, and this change of state may enable other messages.

Among the other frameworks on the JVM platform, only Scala Actors library provides local synchronization constraints. Its support for constraints is based on pattern matching on incoming messages (as in Erlang). Messages that do not match the receive pattern are postponed and may be matched by a different pattern later in execution.

4.3 Discussion

Language abstractions are syntactic sugar which do not change the underlying Actor semantics. For example, previous research formally shows how an actor program with these constraints can be translated to an equivalent program without the constraints [37].

In addition to request-reply messaging and local synchronization constraints, some Actor frameworks have provided other communication and synchronization abstractions which we could not discuss in detail due to space constraints. For example, both Scala and ActorFoundry provide join expressions [38, 39], which allow an actor to respond to a set of messages as if it were one message. SALSA provides three other abstractions for synchronization: *token-passing continuations*, *join blocks* and *first-class continuations*. Token-passing continuations are designed to specify a partial order of message processing. In addition, a token can be used as a result to pass on to a function for further processing. Join blocks can be used to specify a barrier for concurrent activities, such that the results from all the activities are available in a subsequent message. Join blocks are quite similar to join expressions. First-class continuations allow delegating computation to a third party, enabling dynamic replacement or expansion of messages grouped by token-passing continuations.

Both Kilim and Jetlang have a channel-based communication model. Kilim's runtime provides typed mailboxes as channels. Moreover, in Kilim there is only one receiver for each channel. On the other hand, Jetlang provides a publish/subscribe message-passing paradigm on top of these channels. Multiple subscribers may subscribe to a channel and a single actor can subscribe to multiple channels. In the implementation, the latter feature can result in multiple messages being simultaneously processed by an actor, which is a violation of Actor state encapsulation.

Besides two party interactions, we have previously proposed other mechanisms for expressing multi-actor coordination. These include Synchronizers [40] which constrain the messages *arriving* for an actor or a group of actors, and a *Protocol Description Language* [9, 41] which can express complex messaging protocols between actors, potentially constrain messages at all actors participating in a protocol. As actor languages and frameworks get more traction, we believe that such coordination methods may be an important mechanism for writing large-scale programs.

Table 2 summarizes the support for the more common language abstractions by various Actor implementations on the JVM platform.

5. IMPLEMENTATION STRATEGIES FOR PERFORMANCE

We now address the question: can standard Actor semantic properties such as encapsulation, fair scheduling, location transparency and mobility be provided efficiently in an Actor framework on the JVM? As noted earlier (see Table 1, 2), SALSA, Actor Architecture and ActorFoundry (since v0.1.14) faithfully preserve Actor semantic properties such as fairness (as fair as the underlying JVM and OS scheduler), encapsulation, location transparency and mobility. In order to analyze the cost of supporting these properties, we implemented a small benchmark called `Threadring` [42] in which 503 concurrent entities pass a token around in a ring 10 million times. `Threadring` provides a crude estimate of overhead for creating an actor and stress tests message-passing and context switching.

For our experiments we used a Dell XPS laptop with Intel Core™ 2 Duo CPU @2.40GHz, 4GB RAM and 3MB L2 cache. The software platform is Sun's Java™ SE Runtime Environment 1.6.0 and Java HotSpot™ Server VM 10.0 running on Linux 2.6.26. We set the JVM heap size to 256MB for all experiments (unless otherwise indicated).

The Actor Foundry v0.1.14 takes about 695s to execute this benchmark. Both SALSA and Actor Architecture have similar execution times. In contrast, Kilim and Scala Actors perform an order of magnitude faster. JavaAct performs better than Actor Foundry v0.1.14 but does not come close to either Kilim or Scala's efficiency. For comparison, note that, an Erlang implementation takes about 8s while a Java Thread implementation takes 63s. See Figure 7 for a full comparison.

This shows that a faithful but naïve implementation of the standard Actor semantics can make the execution overhead of actor programs significantly high, making frameworks such as SALSA, Actor Architecture and Actor Foundry v0.1.14 efficient only for coarse-grained concurrent or distributed applications.

To address the question we posed earlier, we optimize and analyze the performance of Actor Foundry v0.1.14 to understand the costs, and study strategies in order to mitigate these costs.

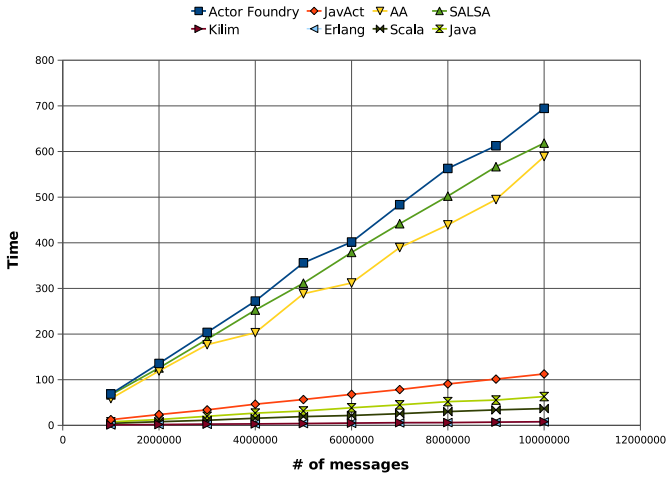

Figure 7: Threadring Performance - Actor Foundry v0.1.14 compared with other concurrent languages and frameworks

5.1 Overview of ActorFoundry

Actor Foundry v0.1.14 was originally designed and developed at the Open Systems Laboratory by Astley et al. around 1998-2000 [20]. The goal was to develop a modular Actor framework for a new, upcoming object-oriented language called Java. Actor Foundry v0.1.14 provides a simple, elegant model in which the control and mailbox are hidden away in the framework while the programmer is concerned with an actor's local state and behavior only. Figure 8 illustrates the programming model. To leverage the Actor semantics, programmers are provided with a small set of methods as part of the Actor Foundry v0.1.14 API. The API includes:

- `send(actorAddress, message, args)`. Sends an asynchronous message to the actor at specified address along with arguments.
- `call(actorAddress, message, args)`. Sends an asynchronous message and waits for a reply. The reply is also an asynchronous message which is either simply an acknowledgment, or contains a return value.
- `create(node, behavior, args)`. Creates a new actor with the given behavior at the specified node. The argument node is optional, if it is not specified, the actor is created locally.

Actor Foundry v0.1.14 maps each actor onto a JVM thread. Messages are dispatched to actors by using the Java Reflection API. The message string is matched to a method name at runtime and the method is selected based on the runtime type of arguments. Any Java object can be part of a message in Actor Foundry v0.1.14; the only restriction being that the object implements `java.lang.Serializable` interface. All message contents are sent by making a deep copy by using Java's Serialization and Deserialization mechanism. Actor Foundry v0.1.14 also supports distributed execution of actor programs, location transparency and weak mobility [30]. Actor Foundry v0.1.14 does not implement an automatic actor garbage collection mechanism [43].

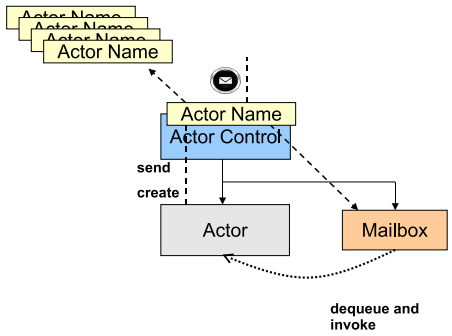

Figure 8: Actor Anatomy in Actor Foundry v0.1.14

5.2 Continuations based actors

Note that each actor in Actor Foundry v0.1.14 maps to a thread; hence actor creation and switching entails thread creation and thread context switching. In particular, thread context switching involves saving the complete computation stack of the thread, program counter and state of other registers. Another source of context switching overhead in the kernel mode is due to kernel crossings.

Prior experience with ThAL language [10] suggests that continuations based actors provide significant improvement in terms of creation as well as context switch overhead. To provide similar support in ActorFoundry, we integrate Kilim's light-weight Task abstraction and its bytecode post-processor ("weaver") [44]. In our context, Kilim's post-processor transformation presents two challenges.

First, the transformation does not work when messages are dispatched using Java Reflection API, since the weaver is unable to transform Java library code. This prevents the continuations from being available in the actor code. To overcome this, we generate custom reflection for each actor behavior. A method matching a message is found by comparing the message string to a method's name and the type of message arguments to type of method's formal arguments. Once a match is found, the method is dispatched statically. A desirable side-effect is that static dispatch is more efficient than Java Reflection.

Second, the transformation requires introducing a scheduler for ActorFoundry which is aware of cooperative, continuations based actors. We introduce such a scheduler as follows. The scheduler employs a fixed number of JVM threads called *worker threads*. All worker threads share a common scheduler queue. Each worker thread dequeues an actor from the queue and call its continuation. Actors are assumed to be cooperative; an actor continues to run until it yields waiting for a message. When scheduled, an actor may process multiple messages. This scheduling strategy increases locality and reduces actor context switches. On the other hand, it can cause starvation in the system. We discuss this issue in the context of fairness in §5.4. Our scheduler is message-driven: an actor is put on the scheduler queue if and only if it has a pending message.

With this implementation, the running time for Threadring example is reduced to about 267s. Further cleanup of the framework and disabling the logging service brings the running time down to about 190s.

5.3 Zero-copy messaging on shared memory platforms

We profile the execution to identify further performance bottlenecks. A faithful implementation of the actor message-passing semantics in Actor Foundry v0.1.14 means that message contents are deep-copied using Java's Serialization and Deserialization mechanism, even for immutable types. It turns out that deep copying of message contents remains the biggest bottleneck. Figure 9 compares the overhead of deep copying versus that of sending message contents by reference for the `Threadring` benchmark. Note that in `Threadring`, the message content is an integer (token), which is an immutable type and can be safely shared between actors. We disable deep-copying for some known immutable types. This brings down the running time of `Threadring` to 30s.

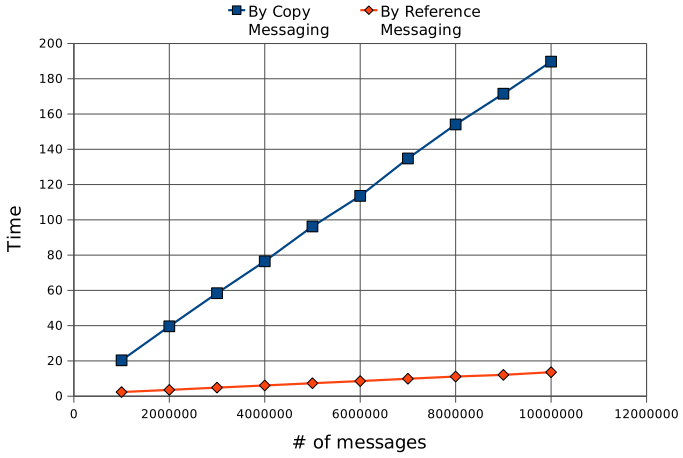

Figure 9: Graph showing the cost of sending messages in ActorFoundry by reference versus by making a deep copy.

5.4 Fair Scheduling

In order to guarantee scheduling fairness, we modify the scheduler described earlier to include a *monitoring thread*. At regular intervals, the monitoring threads checks whether the system has made "progress". A system is said to have made *progress* if any of the worker threads have scheduled an actor from the schedule queue in the preceding interval. If the monitoring thread does not "observe" any system progress and the schedule queue has actors waiting to be scheduled, it spawns a new JVM thread. This *lazy thread creation* mechanism ensures that enabled actors are not permanently starved.

There are some trade-offs in lazy thread creation. If the duration between observations is too small and actors carry out relatively coarse-grained computations, the monitoring thread may incorrectly observe that no progress has been made. In the worst case, this approach may result in some extra native threads. An unfortunate worst case is when the number of native threads exceeds what can fit in the available JVM heap size, resulting in a crash. (This possibility could be prevented by checking the heap size before creating a thread). Moreover, frequently checking progress incurs a higher overhead. On the other hand, a large gap between observations may decrease the responsiveness of an application in the presence of non-cooperative actors. In other words, there is a trade-off between responsiveness, overhead and precision. In our current implementation, the monitoring thread wakes up every 250ms to make observations.

We implemented another small benchmark called `Chameneos-redux` [42]. `Chameneos-redux` comprises of two sets of concurrent entities called *Chameneos* and another concurrent entity called *Broker*. The first set contains three Chameneos while the second set contains ten. Initially each Chameneos in the first set sends a message to the Broker. The Broker provides match-making service by picking two random Chameneos and sending each of them the other's information. After a match, the Chameneos send another message to the Broker and so on. The Broker is required to complete six million matches, after which it polls each Chameneos for total individual matches. At the end, the Broker prints the sum of matches across all Chameneos (in this case, twelve million). After the first round, the same interaction occurs for the second set which has ten Chameneos.

We compare the overhead of fairness for `Threadring`, `Chameneos-redux` and a naïve implementation of `fibonacci`. These benchmarks consist of cooperative actors only. Figure 10 shows that the modified (fair) scheduler incurs negligible overhead for the three benchmarks.

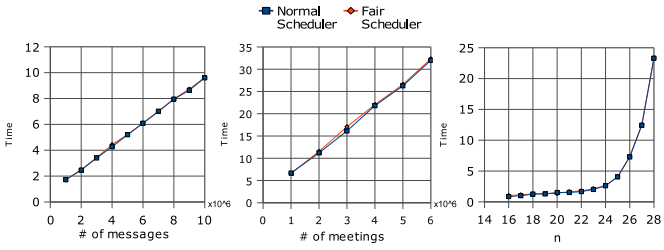

Figure 10: Overhead of Fairness for (a) Threadring (b) Chameneos-redux (c) Naïve fibonacci calculator

5.5 Performance

Figure 11 compares the performance of `Threadring` benchmark written for an optimized implementation of the ActorFoundry (v1.0) with its performance in Kilim, Scala and Jetlang. We do not include SALSA and Actor Architecture as their performance is almost an order of magnitude worse. We also include numbers for Erlang which currently holds the undisputed position of being the most widely used Actor language. Figure 12 provides a similar comparison for `Chameneos-redux` benchmark.

Observe that Kilim outperforms the rest (including Erlang) for both benchmarks, since the framework provides light-weight actors and basic message passing support only. The programming model is low-level as the programmer has to directly deal with mailboxes, and as noted in Table 1, 2, it does not provide standard Actor semantics and common programming abstractions. This allows Kilim to avoid the costs associated with providing these features.

Note that ActorFoundry's performance is quite comparable to the other frameworks. This is despite the fact that ActorFoundry v1.0 preserves encapsulation, fairness, location transparency and mobility. We believe that further signif-

icant optimizations for location transparency and mobility are possible.

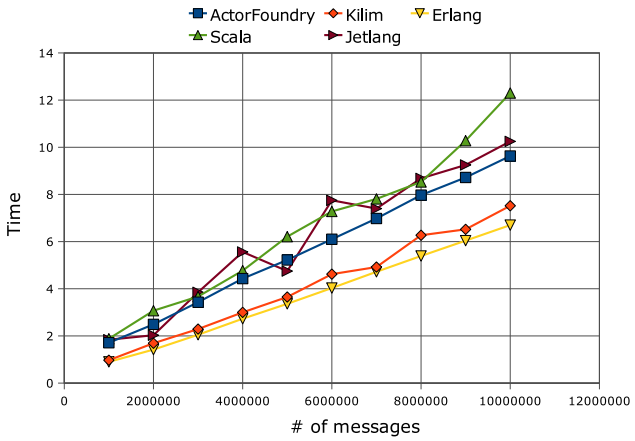

Figure 11: Threadring Performance

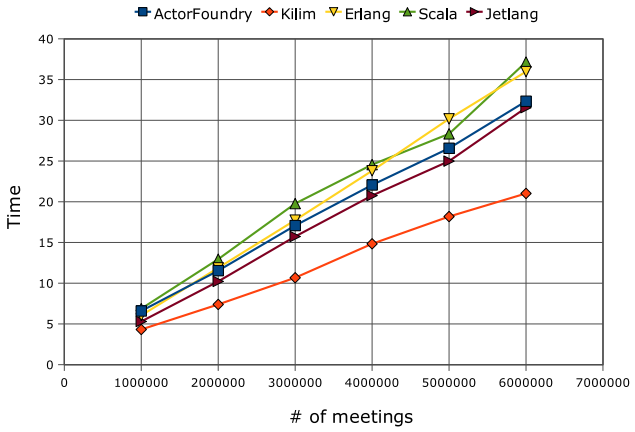

Figure 12: Chameneos-redux Performance

6. DISCUSSION AND FUTURE WORK

Engineering is mainly about picking the right tool for the job. Our experience suggests that, despite a growing interest in the Actor model, the model may not be generally well-understood beyond the basic concept of actors and asynchronous messages. Perhaps this is to be expected as the mindset of the majority of the programmers is ingrained in the currently dominant object-oriented paradigm, and the Actor model of programming requires a shift in that mindset. Such a shift will be as significant as the shift object-oriented programming brought in the world of procedural programming. It may be hard for programmers, and sometimes even for the designers of an Actor framework, to understand the implications of the various design decisions in building or using a particular framework. We have tried to take an open view in this paper since we realize that Actor frameworks are still evolving. We believe this paper can serve as a reference for framework designers evaluating different trade-offs. Our study may also motivate the design of better benchmarks for Actor implementations.

We would like to extend this work with further analyses of the cost of supporting location transparency and mobility. Preliminary results suggest that safe messaging is the dominant source of inefficiency in actor systems. Thus, safe efficient messaging remains an active research topic. We believe static analysis can determine some cases where messages contents can be safely passed by reference. Such an analysis would largely relieve the programmer of the burden of reasoning in terms of a dual semantics for message passing. Although a static analysis would necessarily be conservative, we believe it could be effective much of time. We are also exploring possible optimizations for communication and synchronization between co-located actors.

Acknowledgment

This work was funded in part by the Universal Parallel Computing Research Center at the University of Illinois at Urbana-Champaign. The Center is sponsored by Intel Corporation and Microsoft Corporation, and in part by the National Science Foundation under grant CNS 05-09321. The authors would like to thank the anonymous reviewers for their detailed and useful feedback, and Steven Lauterburg, Darko Marinov, Mirco Dotta, Niklas Gustafsson and other members of the Axum development team at Microsoft for useful discussions during the course of this work. We would like to acknowledge the support of past and present members of the Open Systems Laboratory in this research.

7. REFERENCES

[1] Herb Sutter and James R. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.

[2] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[3] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[4] The e language. *http://www.erights.org/elang*, 2000.

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

[6] Edward A. Lee. Overview of the ptolemy project. Technical Report UCB/ERL M03/25, University of California, Berkeley, 2003.

[7] Microsoft Corporation. Axum programming language. *http://msdn.microsoft.com/en-us/devlabs/dd795202.aspx*, 2008-09.

[8] D. Kafura. ACT++: building a concurrent C++ with actors. *Journal of Object-Oriented Programming*, 3(1):25–37, 1990.

[9] D.C. Sturman and G.A. Agha. A protocol description language for customizing failure semantics. In *Proceedings of 13th Symposium on Reliable Distributed Systems*, pages 148–157, Oct 1994.

[10] WooYoung Kim. *ThAL: An Actor System for Efficient and Scalable Concurrent Computing*. PhD thesis, University of Illinois at Urbana-Champaign, 1997.

[11] J.P. Briot. Actalk: a Test bed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment. In *Proceedings of the 1989 European Conference on Object-Oriented Programming*, page 109. Cambridge University Press, 1989.

[12] Christian Tismer. Stackless python. *http://www.stackless.com/*, 2004-09.

[13] Jacob Lee. Parley. *http://osl.cs.uiuc.edu/parley/*, 2007.

[14] Jonathan Sillito. Stage: exploring erlang style concurrency in ruby. In *IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering*, pages 33–40, New York, NY, USA, 2008. ACM.

[15] Microsoft Corporation. Asynchronous agents library. *http://msdn.microsoft.com/en-us/library/dd492627(VS.100).aspx*, 2008-09.

[16] Mike Rettig. Retlang. *http://code.google.com/p/retlang/*, 2007-09.

[17] P. Haller and M. Odersky. Actors That Unify Threads and Events. In *9th International Conference on Coordination Models and Languages*, volume 4467 of *Lecture Notes in Computer Science*. Springer, 2007.

[18] S. Srinivasan and A. Mycroft. Kilim: Isolation typed actors for Java. In *Procedings if the European Conference on Object Oriented Programming (ECOOP)*, 2008.

[19] Mike Rettig. Jetlang. *http://code.google.com/p/jetlang/*, 2008-09.

[20] Mark Astley. *The Actor Foundry: A Java-based Actor Programming Environment*. Open Systems Laboratory, University of Illinois at Urbana-Champaign, 1998-99.

[21] Myeong-Wuk Jang. *The Actor Architecture Manual*. Department of Computer Science, University of Illinois at Urbana-Champaign, March 2004.

[22] Tim Jansen. Actors guild. *http://actorsguildframework.org/*, 2009.

[23] S. Rougemaille J.-P. Arcangeli, F. Migeon. Javact : a java middleware for mobile adaptive agents, February 2008.

[24] William Zwicky. Aj: A systems for buildings actors with java. Master's thesis, University of Illinois at Urbana-Champaign, 2008.

[25] Rex Young. Jsasb. *https://jsasb.dev.java.net/*, 2008-09.

[26] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[27] Gul Agha, Ian A. Mason, Scott Smith, and Carolyn Talcott. A Foundation for Actor Computation. *Journal of Functional Programming*, 7(01):1–72, 1997.

[28] WooYoung Kim and Gul Agha. Efficient support of location transparency in concurrent object-oriented programming languages. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 39, New York, NY, USA, 1995. ACM.

[29] K. Sen and G. Agha. Automated systematic testing of open distributed programs. In *Fundamental Approaches to Software Engineering (FASE)*, volume 3922 of *Lecture Notes in Computer Science*, pages 339–356. Springer, 2006.

[30] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[31] T. Walsh, P. Nixon, and S. Dobson. As strong as possible mobility: An Architecture for stateful object migration on the Internet. *6th ECOOP Workshop on Mobile Object Systems: Operating System Support, Security and Programming Languages, Sophia Antipolis (France)*, 2000.

[32] Rajendra Panwar and Gul Agha. A methodology for programming scalable architectures. In *Journal of Parallel and Distributed Computing, vol. 22 pp 479-487*, 1994.

[33] Gul Agha. Concurrent object-oriented programming. In *in Communications of the ACM, Association for Computing Machinery, vol. 33, no. 9, pp 125-141, September*, 1990.

[34] G. Agha, S. Frolund, WY Kim, R. Panwar, A. Patterson, and D. Sturman. Abstraction and modularity mechanisms for concurrent computing. *IEEE Parallel & Distributed Technology: Systems & Applications*, 1(2):3–14, 1993.

[35] T. Papaioannou. *On the structuring of distributed systems: The argument for mobility*. Loughborough University of Technology, 2000.

[36] Svend Frølund. *Coordinating distributed objects: an actor-based approach to synchronization*. MIT Press, Cambridge, MA, USA, 1996.

[37] I.A. Mason and C.L. Talcott. Actor languages their syntax, semantics, translation, and equivalence. *Theoretical Computer Science*, 220(2):409–467, 1999.

[38] S. Frølund and G. Agha. Abstracting interactions based on message sets. In *Selected papers from the ECOOP '94 Workshop on Object-Based Models and Languages for Concurrent Systems*, Lecture Notes In Computer Science, pages 107–124. Springer-Verlag, 1995.

[39] Philipp Haller and Tom Van Cutsem. Implementing joins using extensible pattern matching. In *10th International Conference on Coordination Models and Languages*, volume 5052 of *Lecture Notes in Computer Science*, pages 135–152. Springer, 2008.

[40] Svend Frølund and Gul Agha. A language framework for multi-object coordination. In *ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming*, pages 346–360, London, UK, 1993. Springer-Verlag.

[41] Mark Astley, Daniel Sturman, and Gul Agha. Customizable middleware for modular distributed software. *Communications of the ACM*, 44(5):99–107, 2001.

[42] Open Source. The computer language benchmarks game. *http://shootout.alioth.debian.org/*, 2004-2008.

[43] Nalini Venkatasubramanian, Gul Agha, and Carolyn Talcott. Scalable distributed garbage collection for systems of active objects. In *in Y. Bekkers and J. Cohen (editors), International Workshop on Memory Management, ACM SIGPLAN and INRIA, St. Malo, France, Lecture Notes in Computer Science, vol. 637, pp 134-148, Springer-Verlag, September*, 1992.

[44] S. Srinivasan. A thread of one's own. In *Workshop on New Horizons in Compilers*, volume 4. Citeseer, 2006.