EFFICIENT EXECUTION OF FINE-GRAINED ACTORS ON MULTICORE PROCESSORS

BY

RAJESH KUMAR

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Doctoral Committee:

Professor Gul Agha, Chair
Professor Darko Marinov
Professor Ralph Johnson
Dr. Mark S. Miller, Research at Google

ABSTRACT

The Actor model is a promising model for programming new computing platforms such as the multicores and cloud computers, primarily due to features such as inherent concurrency and isolation of state. However, the model is often perceived to be fundamentally inefficient on stock multicore processors. Consequently, we find that standard semantic properties of the model, including encapsulation and fairness, are ignored even by languages and frameworks that claim to be based on the Actor model.

In this work, we propose and implement both static (compiler) and dynamic (runtime) techniques that overcome these perceived inefficiencies, while retaining key actor semantics, even in a framework setting. We compare the performance of ActorFoundry with other frameworks for small benchmarks and programs. The results suggest that key actor semantics can be supported in an actor framework without compromising execution efficiency.

We also validate our results for a large real-world application, i.e. a Java game called Quantum [1] having more than 25k lines of code. Quantum is a real-time strategy game, which employs a few threads for handling IO, UI and network events. We port the Quantum game to ActorFoundry, so that the asynchrony due to threads and communication between them is expressed using actors and messages.

Next, we introduce additional concurrency in Quantum by actorizing all game objects. This results in relatively fine-grained actors. We are able to run a game instance with more than 10,000 game objects, which keeps an 8-core computer at full throttle. According to our knowledge, this is the largest execution of a real client-side actor program. The performance is comparable to an Actor framework implementation that does not provide the standard actor semantics. Moreover, our set of static (compiler) and dynamic (runtime) techniques allow an actor framework to compare well against a shared memory model in terms of execution efficiency.

To my parents, siblings and my wife, for their love and support.

ACKNOWLEDGMENTS

First of all, I would like to thank my advisor and mentor, Gul Agha, for his faith, encouragement, advice and an unreasonable level of support through my graduate school. It has been an honor to interact with and learn from him on many aspects of academic and general life. He will always remain an inspiration and a fatherly figure.

Next I would like to thank my committee member Darko Marinov for his prompt advice and words of support. I had the pleasure of working with him for a significant part of my graduate school. I enjoyed my conversations with him and will always remember his frank and humorous style.

My committee members Ralph Johnson and Mark Miller have been instrumental in getting this dissertation to completion with their generous advice and invaluable time.

I have had the pleasure of learning from many professors in the department including P Madhusudan, Vikram Adve, Indranil Gupta, Robin Kravets, and Grigore Rosu. Members of the OSL Research Group have provided valuable feedback to my research and great company. I have been fortunate to make many friends along the way. I would like to give a shout-out to Timo Latvala, Vijay Korthikanti, Vilas Jagannath, Kirill Mechitov, Peter Dinges, Parya Moinzadeh, Reza Shiftehar, Koushik Sen, Amin Shali, Liping Chen, Myungjoo Ham, Sameer Sundresh, Ashish Vulimiri, and Minas Charalambides. Outside the OSL, I had some very good interactions with and had the pleasure of working along with Samira Tasharofi, Stas Negara, Nicholas Chen, Steven Lauterburg, and Noyan Baykal.

I would like to thank the department staff for their timely help in all matters related to fellowships, travel, and other paperwork. I would like to acknowlege the support of Mr and Mrs Sohaib Abbasi for their support through a generous fellowship.

Over the course of graduate program, I met with some amazing people

and made some very good friendships that I will cherish all my life. I will intentionally make the mistake of trying to name them: Yaniv, Santiago, Artur, Noomi, Emma, Fakhruddin, Sarika, Anshu, Kiran, Brian, Lee, Anusha, Soumyadeb, Akash, Lavanya, Natarajan, Yoav, Alina, Regina, Neha, Preeyaa, Aneel, Rakesh, Rekha, Pratim, Ankur, Shruti, Sruthi, Pavithra, Sarah, Jay, Amandeep, Supreet, Vrashank, Komal, Piyush, Tarun, Rajhans, Neha, Nishana, Jayanthi, Nisha, Abhishek, Utsav, Piyush, Kiran, Meghana, Cosmin, Camilo, Ralf, Mara, Anthony, Cassie, Megan, Sibin, Radha, Shweta, Mohit, Asfand, Ibraheem, Usman, Bilal, Hassaan, Atif, Asma, Aya, Gina and many others whom I sincerely apologize to. I will keep looking out for them through my life and meet them whenever possible.

I can not forget the support and patience of my friends and family through the challenging times of graduate school.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Due to the ubiquity of platforms such as the multicore processors, cloud computers and sensor networks, Actor-oriented programming has been getting increasing attention from both scientists and practitioners [2].

The key features that make the Actor model promising are isolation of state, modular reasoning and scalable execution. Because actors are isolated and do not share state, the programs do not suffer from low-level memory hazards such as data races. Because the actor model facilitates modular reasoning about certain safety and liveness properties, it is simpler to assign blame for program bugs. For example, an actor that is unable to make progress has only itself to blame for blocking during the processing of a message. Because programs are decomposed into concurrent actors, the programs can theoretically scale to as many processors as the number of actors. These features are enabled due to some key semantic properties of the Actor model, namely encapsulation, fairness, location transparent naming and mobility [3]. The actor model guarantees encapsulation in two ways: *temporal encapsulation* and *safe messaging*. Temporal encapsulation means that an actor interacts with the environment, including other actors, only through messaging. (The word temporal does not imply any timing requirement here as it does in an earlier description of the term in [4].)

Possibly because of one or more of the above-mentioned features, many new frameworks and languages based on the Actor model have been put out recently by both researchers and practitioners. It is not unusual to come across a new actor library or framework on a monthly basis.

However, a closer observation suggests that these frameworks compromise some key semantic properties of the Actor model [5]. There may be multiple reasons for this: lack of awareness, time-to-market, or trade-off for efficiency. It is known that encapsulation, and specifically safe messaging is compromised in frameworks such as Scala Actors and Kilim [6, 7] in order to obtain

execution efficiency .

In this dissertation, we seek to refute the perception about inefficiency, and show that all key semantic properties of Actor model can be implemented, even in an actor framework, with performance comparable to other implementations. More specifically, we challenge the notion, with partial success, that message-passing models such as the Actor model are fundamentally less efficient on multicores than shared memory models.

In the model that we consider, there are two important deviations from the pure actor model [8]. In our model, not every program entity is an actor. Traditional objects are first-class entities as well and can be part of the internal state of an actor. This deviation is tolerated for pragmatic reasons associated with building an actor framework for an existing object-oriented language, and leveraging its existing code base. Almost all the existing frameworks that we considered have made this deviation. Asynchronous Sequential Processes describe a similar model in a formal way [9]. Secondly, actors can block during the processing of messages, typically when they are waiting to receive the reply of a "query" message. This deviation is made to accomodate the mainstream programmers' familiarity with sequential control flow. Such a linguistic construct was introduced in high-level actor languages early on and in current frameworks. For example, both Scala Actors [6] and Kilim [7] provide such constructs.

Observe that by restricting accessibility of sequential objects to single actors, no concurrent access to the state of an actor is introduced. Thus, such a system can be easily modeled by the actor semantics provided in [3].

In our quest for these semantics *and* improved performance, we analyze various JVM-based Actor frameworks, and compare their semantics and performance for small benchmarks and programs. We observe that frameworks that provide these semantic properties are significantly more inefficient than frameworks that do not provide these properties.

We implement static and dynamic techniques in order to improve the performance of a framework called ActorFoundry [10] that enforces actor properties. Specifically, we first introduce lightweight stacks and a CPS transform from the Kilim project [11] in order to implement actors with a low cost of creation and context-switching. Second, we introduce a scheduler with a monitoring-based technique for guaranteeing scheduling fairness. Third, we exclude known immutable types in Java such as String and Integer from

being copied inside a message that is sent to a local actor. (This work was published in [5].)

As a result of implementing these techniques, the performance of ActorFoundry is comparable to other frameworks and languages. However, in this work, we made a compromise regarding a key property, safe messaging. The programmer has to explicitly annotate the messages that are safe to send without making a copy of the message contents.

In order to have a uniform semantics for message passing, we then implement a static analysis framework for Java bytecode that (conservatively) removes the need for copying message data on shared memory hardware. The static analysis infers instances of messages that result in transfer of ownership. Such messages can be sent without copying resulting in efficient execution. The tool implementation SOTER is found to be effective and useful for a suite of examples [12].

To summarize, we implement both static and dynamic techniques to mitigate the perceived inefficiencies and improve the performance of an Actor framework. The initial results suggest good performance for Actor frameworks, when measured for small benchmarks and programs. We then validate these results for a large real-world program. We also compare its performance against a shared memory implementation.

In this regard, we focus on games as the application domain. Games are particularly promising because games, especially of the real-time strategy genre, tend to have internal concurrency (with complex interactions), independent of the number of external user requests. Morever, games represent a popular domain among client-side applications. For example, games consistently dominate the list of top apps in Google Play market and Apple's App Store. According to Business Insider Intelligence, in a recent check of Apple's iPhone App Store, games represented 55 percent of the top 200 paid apps, and 33 percent of the top 200 free apps.

Specifically we look at an open-source game called Quantum [1], which has more than 25k lines of Java code. Quantum is a real-time strategy game that involves multiple players controlling pre-owned planets, trees, and creatures that orbit planets. Players can create more creatures by building trees, and colonize other planets in the universe by moving creatures to them. The goal of the game is it to eliminate all enemy creatures and take over enemy planets.

The original version from the web has a strict game loop that updates all games objects every game cycle. This version has some concurrency for I/O operations. For example, it has separate threads for receiving user input, producing sound and network play. In the first stage of actorizing Quantum, we replace these threads with actors.

In the second stage, we introduce new concurrency by converting game objects such as planets, trees and creatures into actors. We replace the strict game loop, and instead enable each game object to update itself asynchronously and send messages for exchanging state information with other games objects. We are able to run a game instance with more than 10,000 game objects and run the computer (a Core 2 Duo and an 8-core i7) at full throttle. According to our knowledge, this is the largest execution of a real client-side actor program.

Although the game implementation has some communication-computation overlap, many CPU cycles are spent in delivering a message (by the sender), and later by the receiver in pulling it from its mailbox, and then repeating these steps for the reply, if needed. Independent experiments on a Core 2 Duo processor suggest that it takes two orders of magnitude more time for a request-reply message in comparison to reading the value simply through a method call.

Next, we introduce a "short-circuit" implementation for request-reply messages. If the recipient actor of such a message is *local*, and is neither busy nor blocked, the sending actor calls the message handler in its own stack instead of delivering a message to recipient's mailbox [13]. Experimental results suggest that this technique improves performance by an order of magnitude.

The work presented in this dissertation suggests that key actor semantics can be supported in an Actor framework without compromising execution efficiency. Moreover, our set of static (compiler) and dynamic (runtime) techniques allow an Actor framework to compare well against a shared memory model in terms of execution efficiency, even on a hardware with a low number of cores.

1.1 Not In Scope

For large-scale distributed systems, such as supercomputers, modern server infrastructures that power Google, Amazon, Facebook and Twitter, and cloud infrastructures there is no choice but to think in terms of message-passing. However, the scalability of servers in client-server architectures is out of scope for this work. We look at the efficiency of local execution on multicore processors and interactive concurrency.

A large body of research work has been published on various aspects of the Actor model. In the terms of efficiency, we acknowledge that garbage collection for Actor systems [14, 15, 16] is not in scope for this work. Similarly, power efficiency of actor execution on multicores [17, 18] is not in the scope.

Considerable amount of work has been done on expressing concurrency in the Actor model including coordination [19, 20, 21, 22, 23, 24, 25, 26], session types [27], real-time constraints [28, 29], and combining actor model with other models of concurrency [30, 31, 32].

Recent work has also focused on building tools for model checking [33, 34, 35, 36] and testing [37, 38, 39, 40] of actor programs. This is not further discussed in this work.

1.2 Outline

The dissertation is organized as follows. In the next chapter, we describe the Actor model and its key semantic properties. In chapter 3, we describe the programming model and its execution semantics model that we consider in this work.

In chapter 4, we discuss the related work in this area. In chapter 5, we discuss the perceived inefficiencies of implementing key properties of the model in both qualitative and quantitive sense. In the next chapter, we discuss optimization strategies in order to mitigate some of the perceived inefficiencies.

In chapter 7, we discuss a static analysis algorithm that enables efficient yet safe messaging in many cases. In chapter 8, we present our experience with looking at real-world programs and converting them into Actor programs. Finally, we conclude our thesis and present some ideas for future work in chapter 9.

CHAPTER 2

BACKGROUND

In this chapter we present a concise definition of the Actor model, its key semantic properties, and describe their implications on programmability and performance.

2.1 The Actor Model

The Actor model is an inherently concurrent model of programming [8]. In the Actor model, programs comprise of concurrent, autonomous entities, called *actors*, which can send and receive *messages*. 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 local state in its own thread of control.

Actors send *asynchronous messages* for exchanging data as well as synchronizing with each other (see Figure 2.1). The receiving actor processes the messages, one message at a time, in a single atomic step. Each step consists of all the actions taken in response to a given message, enabling a *macro-step semantics* [3]. The union of all potential responses by an actor constitute its behavior.

Each actor has a unique, immutable *name* which is required to send a message to that actor. An actor name cannot be forged but may be communicated to other actors.

The standard Actor semantics provide encapsulation, fairness, location transparent naming, and mobility. These properties enable compositional design, high-level reasoning [3], and scalable performance as applications and architectures scale [13]. For example, because actors communicate using asynchronous messages, an actor does not occupy any system resources while sending and receiving a message. This is in contrast to the shared memory

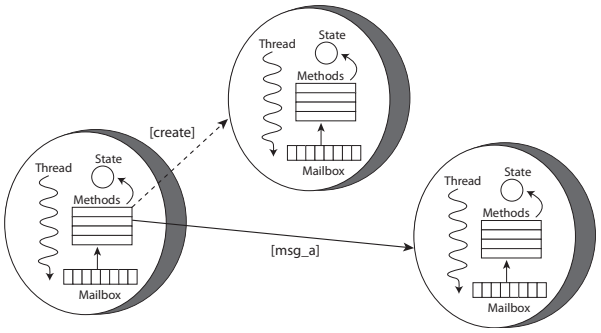

Figure 2.1: Actors are concurrent entities that exchange messages asynchronously.

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* (See [5]).

2.2 Actor Properties

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

2.2.1 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

```
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 2.2: A program written in the Scala Actors shows violation of temporal encapsulation which may cause two actors to simultaneously execute the critical section.

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: *temporal encapsulation* and *safe messaging*.

Temporal 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 a message to the other actor.

The code in Figure 2.2 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.

Actor implementations that enforce temporal 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.

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 [3] simplifies testing and reasoning about safety properties in actor programs [41]. Two concurrent messages that require m and n instructions for processing would need an exponential number of schedules in m and n to completely reason about their behavior. However, given macro-step semantics, it needs executing only two schedules! Without enforcing actor encapsulation, it would be incorrect to assume the macro-step semantics.

2.2.2 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 i.e. every actor makes progress. Note that if an actor is starved (i.e. never scheduled), pending messages directed to it cannot be delivered. Thus, the notion of guarantee of message delivery implies that no actor is permanently starved. A notion of fairness that is weaker than both previous notions is that the system makes progress. However, in this case, a given actor may not be making any progress.

```
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 2.3: 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.

Without a fairness assumption, one cannot reason about liveness properties of concurrent programs [3]. 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. Note that fairness in itself, however, does not guarantee liveness.

Consider the program in Figure 2.3 in which an actor is "busy-waiting" for a reply from another actor, say a *calculator* actor. In the absence of fairness, the calculator actor 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 native application components with 3^{rd} party plug-ins. For example, if browsers do not provide any fairness guarantee for the execution of different browser components and plug-ins, the plug-ins may result in crashes and hang-ups.

Moreover, more than one message may be sent to an actor concurrently (by the same sender–due to asynchronous messaging, or by different senders–due to asynchronous operation of these senders). Without a fairness requirement guaranteeing that each message be eventually delivered, one cannot ensure reason about a liveness property that depends on the delivery of a message.

2.2.3 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 that 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. Location transparency enables building robust, scalable programs by providing the runtime the flexibility to distribute and replicate components.

Because one actor does not know the address space of another actor, a desirable consequence of location transparency is temporal encapsulation. Location transparent naming also facilitates runtime migration of actors to different nodes, or *mobility*. Such mobility may be guided by annotations or a meta-level logic either to facilitate efficiency in execution or to ensure security.

2.2.4 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 [42]. *Strong mobility* is defined as the ability of a system to move code, data and the execution state. *Weak mobility*, on the other hand, only allows movement of code and data.

In an actor system, strong mobility implies that the system is allowed to migrate an actor even when it is busy (i.e., it is processing a message), while weak mobility implies migrating an actor only when it is not processing a message.

Because actors provide encapsulation and modularity of control, mobility is 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, even in the case of weak mobility. 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 [43].

At the system level, mobility can enable runtime optimizations for load balancing, reconfiguration and certain aspects of fault-tolerance [44, 45]. Previous work has shown that mobility is helpful in achieving scalable performance, especially for dynamic, irregular applications over sparse data structures [46]. 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) enables the programmer to declaratively exploit heterogeneous system resources such as GPUs, DSPs, other task-specific accelerators, and high clock frequency cores.

CHAPTER 3

PROGRAMMING AND EXECUTION MODEL

In this work, we are interested in actor programs developed in a mainstream language such as Java (referred to as *host language*), which are typically imperative in style and can leverage a large body of existing code.

In the model that we consider, one can think of an actor as a sequential program augmented with three (actor) operators: *receive*, *send* and *create*. An actor program can be viewed as a collection of reactive entities that concurrently execute the following sequence of statements in a loop: receive a message (and block if no message is available), decode the message, and process it. During the processing of a message, an actor can send more messages to its *acquaintances*, create new actors, and update its local state by assigning new values to variables.

An actor can also send a blocking or RPC-like message. This blocks the sending actor, and behaves like a *future* that must be resolved before the sender can make any further progress. The blocking message returns with a response value or an acknowledgement.

Note that we do not allow composition of actor programs with any existing abstraction of concurrency such as threads or processes that are available for the host language.

We also assume that there is no intra-actor or internal concurrency. However, the implementations can choose to parallelize the processing of messages in cases where it is *safe*, i.e. it preserves one-message-at-a-time semantics, or in other words, serializability.

3.1 ActorFoundry - a Java-Based Actor Framework

In order to illustrate the model, consider a simple imperative actor language called ActorFoundry that runs on the JVM and extends Java syntax. In

essence, we describe restrictions on Java programs that make them legal ActorFoundry programs. For a similar approach, see the Joe-E language [47].

A class that extends `osl.manager.Actor` defines a certain actor behavior. New actors are created as "instances" of such classes by calling the `create(class, args)` method, where `args` correspond to the arguments of a constructor in the corresponding class. Each newly created actor has a unique name that is initially known only to the creator at the point where the creation occurs.

The instance variables of an actor class comprise the local state of its instance actor. These variables are created by, owned by and visible to only the corresponding instance actor. ActorFoundry does not restrict the type of instance variables; the variables may be ordinary Java objects, mutable or immutable, or names of actors.

Note that Java allows static variables that implicitly introduce global, shared state among objects. We assume that programmers do not use static variables, or the static variables only refer to immutable objects as in [47]. The ActorFoundry compiler does not currently enforce this restriction.

Certain (public) methods are annotated with `@message`. These serve as handlers for incoming messages. If a corresponding method is not available, the message is processed with just the no-op instruction.

Example: Listing 1 shows the HelloWorld program in ActorFoundry. The program comprises of two actor definitions, `HelloActor` and `WorldActor`. An instance of the `HelloActor` can receive one type of message, the `greet` message, which triggers the execution of `greet` method. The `greet` method serves as P's entry point, similar to how the `main` method serves as the point of entry in a Java program.

3.1.1 Execution Semantics

The semantics of ActorFoundry closely follow the semantics described in [3], and can be informally described as follows. Consider an ActorFoundry program P that consists of a set of actor definitions.

At the beginning of execution of P, the mailbox of each actor is empty and some actor in the program must receive a message from the runtime or the environment. The ActorFoundry runtime first creates an instance of

```
public class HelloActor extends Actor {

  ActorName other = null;

  @message
  public void greet() throws RemoteCodeException
  {
      call(stdout, "print", "Hello");
      other = create(WorldActor.class);
      send(other, "audience");
  }
}

public class WorldActor extends Actor {

  @message
  public void audience() throws RemoteCodeException
  {
      send(stdout, "print", "World");
  }
}
```

Listing 1: HelloWorld program in ActorFoundry

`HelloActor` and then sends the `greet` message to it, which serves as P's entry point. Because only one instance of each actor definition is created in this example program, we refer to the instance of `HelloActor` as `helloActor`.

Each actor can be viewed as executing a loop with the following steps: remove a message from its mailbox (often implemented as a queue), decode the message, and execute the corresponding method. If an actor's mailbox is empty, the actor blocks–waiting for the next message to arrive in the mailbox. Such blocked actors are referred to as *idle actors*.

The processing of a message may cause the actor's local state to be updated, new actors to be created and messages to be sent. Because ActorFoundry does not restrict the syntax or control structures available in Java, an actor may execute an infinite loop inside the body of its methods or method calls made on its encapsulated objects. Because of the encapsulation property of actors, there is no interference between messages that are concurrently processed by different actors.

An actor communicates with another actor that it knows in P by sending asynchronous (non-blocking) messages using the `send` statement: `send`*(a,msg)* has the effect of *eventually* appending the *contents* of *msg* to the mailbox of the actor *a*. Thus, messages have *by-copy* semantics. However, the call to `send` returns immediately i.e. the sending actor does not wait for the message to arrive at its destination.

The model also allows blocking or RPC-like messages expressed using the `call` statement. The call to `call` blocks the sending actor, and returns the return value of the corresponding method in the receiving actor, or an implicit, empty acknowledgement if the return type is `void`.

An exception that is raised by a actor during the processing of an asynchronous message is handled by itself or the runtime. It is never sent back *implicitly* by the runtime to the sending actor. On the other hand, if an exception is raised during the processing of an RPC-like message, and it is not handled by the receiver itself, the runtime sends an asynchronous exception message back to the sender. When a waiting actor receives an exception message, *and* it matches the identity of the message that it was waiting on, the sender gets unblocked and the control transfers to an exception message handler.

Because actors operate asynchronously, and the network has indeterminate delays, the arrival order of messages is *nondeterministic*. However, we assume that messages are *eventually* delivered (by relying on the guarantee provided by underlying communication layers), and are eventually processed by the destination actors (as fair as the JVM scheduler).

An actor program 'terminates' when every actor created by the program is idle and the actors are not open to the environment (otherwise the environment could send new messages to their mailboxes in the future). Note that an actor program need not terminate–in particular, certain interactive programs and operating systems may continue to execute indefinitely.

Going back to our example, on receiving a `greet` message, the `helloActor` sends a (blocking) `print` message to the `stdout` actor (a built-in actor representing the standard output stream) along with the contents "Hello". As a result, "Hello" will be printed on the standard output stream. Next, it creates an instance of the `WorldActor`. The `helloActor` sends an asynchronous `audience` message to the `worldActor`, which in turn sends a `print` message to `stdout` along with the contents "World". Thus the `helloActor` delegates the printing of "World" to the `worldActor`. Note that because the message sent by `helloActor` to `stdout` is blocking, "Hello" will always be printed before "World". Further details about ActorFoundry programming environment can be found on its webpage [10].

CHAPTER 4

RELATED WORK

The ubiquitous availability of multicore processors has made it imperative for application programmers to consider introducing parallelism and concurrency in their programs [48]. The dominant model for concurrent programming, popularized traditionally by OS programming and recently 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 [49]. Not surprisingly, researchers and practitioners have shown an increasing interest in alternate models, such as the *actor model of programming*. Some languages based on the Actor model include Erlang [50], E language [51, 52], SALSA [53], Ptolemy [54] Axum [55] and Dart [56].

Ed Lee [49] 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++* [57], *Broadway* [58], *Thal* [59]), Smalltalk (*Actalk* [60]), Python (*Stackless Python* [61], *Parley* [62]), Ruby (*Stage* [63]), .NET (Microsoft's *Asynchronous Agents Library* [64] and Orleans [65], *Retlang* [66]) and Java (*Scala* Actors library [67], *Kilim* [7], *Jetlang* [68], *ActorFoundry* [69], *Actor Architecture* [70], *AmbientTalk* [71], *Actors Guild* [72], *JavAct* [73], *AJ* [74], *Jsasb* [75], and *JCoBox* [76]).

We analyze various actor-oriented frameworks that execute on the JVM platform. A programming framework 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.

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 is to understand the semantic properties of actor-oriented frameworks, and how they are implemented.

Table 4.1: Comparison of Execution Semantics (SL = SALSA v1.1.2, SA = Scala Actors v2.7.3, KA = Kilim v0.6, AA = Actor Architecture v0.1.3, JA = JavAct v1.5.3, AF = ActorFoundry v1.0, JL =Jetlang v0.1.7, ✓ means the property is satisfied, × means the property is not satisfied)

	SL	**SA**	**KA**	**AA**	**JA**	**AF**	**JL**
Temporal Encapsulation	✓	×	×	✓	✓	✓	✓
Safe Message-passing	✓	×	×	✓	×	✓	×
Fair Scheduling	✓	✓	×	✓	×	✓	×
Location Transparency	✓	×	×	✓	✓	✓	✓
Mobility	✓	×	×	✓	✓	✓	×

4.1 Comparison

Table 4.1 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, up to the Actor's name, to prevent any

unintended sharing among actors. Deep copying is an expensive operation, even when performed at the level of native instructions (see §6).

Actor implementations such as Kilim and Scala Actors violate the temporal encapsulation property. In Kilim, actors have memory references to other actors' mailboxes, while Scala actors have direct reference to the object representation of other actors.

Moreover, Kilim [7] and Scala Actors [67] provide by-reference semantics for message-passing, thus introducing shared state between the actors. These frameworks leave the responsibility of making a copy of message contents to the programmers, when required. We argue that such an approach creates a double hazard 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.

A similar approach is adopted by JavAct and Jetlang a message carries references to its contents on shared memory platforms. These frameworks also encourage the programmers to use immutable objects inside their objects. An alternate proposal is to add a type system based on linear types to enable safe, zero-copy messaging [7]. 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.

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). This is the model adopted by JVM-based frameworks such as Scala Actors, ActorFoundry, SALSA and Actor Architecture. As discussed, such a guarantee is limited by the resource constraints of the JVM and the underlying platform. In other cases, explicit scheduling by the frameworks may be required to support fair scheduling.

Note that fairness is an abstraction in the sense that it ignores the problem of such resource limitations. This is analogous to how recursion is defined in sequential languages by ignoring memory limitations [77]. For our purposes, for example, assuming no message loss and a first-in-first-out (FIFO) scheduling of messages arriving at an actor would be considered fair, while a last-in-first-out (LIFO) protocol would not be. This is despite the fact that there may be some suitably large limit on memory available to store messages.

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).

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.

We already discussed the significance of each of these properties in order to understand the impact of compromising the property from the "ease of programming" point of view (§2.2). Next, we discuss the costs associated with naïve implementations of these properties. Later, we study how the cost of providing actor properties may be mitigated (§6). 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.

Recently, Pritish et al. propose runtime optimizations to improve performance for Charm++ actors or *chares* on multicore processors [78]. However, note that chares do not observe by-value semantics for messaging on multi-

cores.

CHAPTER 5

PERFORMANCE ISSUES FOR IMPLEMENTING SEMANTIC PROPERTIES

In this chapter, we study the perceived inefficiencies associated with implementing actor properties. Perhaps not so surprisingly, we observe that these perceived costs are the true costs of a straightforward or naïve implementation of these properties.

5.1 A Qualitative Analysis

Figure 5.1 illustrates the structure of an actor in a system which supports the semantic properties discussed in previous section. Similar to an object, the actor encapsulates state and behavior. Since actors are concurrent and autonomous, every actor maintains separate control information or stack. In addition, every actor has a mailbox, which has an explicit address represented by ActorName in the figure. An actor can communicate (send messages) with other actors in the system if it knows their address.

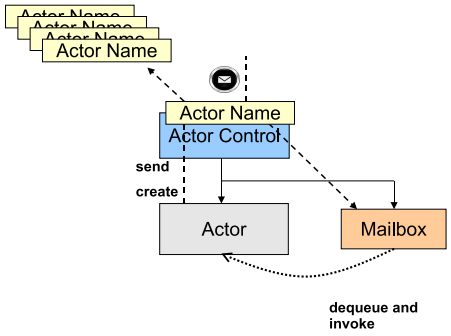

Figure 5.1: Actor Anatomy in Actor Foundry v0.1.14

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? Before we do

so, it would be instructive to analyze qualitatively what may be required to implement the above mentioned components in a framework written for an existing language, such as Java, which we will refer to as *host language*.

The state and behavior together can be represented simply as an object in the host language. The control of an actor can be encoded in the host language's abstraction of concurrency, such as a thread in Java. This allows the implementation to be as fair as the underlying language or VM. This approach may work well if the framework is targetted for writing coarse-grained actors as in SALSA, E, AmbientTalk, and CHARM++. On the other hand, it may have serious performance implications for fine-grained actors due to costs associated with creation and context-switching. Many light-weight implementations, such as Stackless Python and Kilim, allow multiple actors to be mapped to a single native thread. However, such implementations would have to implement fairness in their scheduling strategy.

The mailbox can be implemented as a queue object in the host language. Every actor may have a separate mailbox, or alternatively multiple actors that are mapped to a single thread may have a common mailbox.

As discussed in section 2.2, object-level state encapsulation (as in Java) is not sufficient to enforce actor encapsulation. On the other hand, the reference of an actor's mailbox can be used as the actor address to enforce actor encapsulation. This technique is implemented in Kilim. However, in order to implement location independence and mobility, some proxy object which maps to a physical address is still required. Location independence and mobility also require a naming and routing service, and a transportation layer to send actors and messages from one node to another.

5.2 A Quantitative Analysis

As noted in Table 4.1, 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. We choose Actor Foundry v0.1.14 as our research framework not only because it supports the standard properties, we find it to be highly modular and extensible. Moreover, its source distribution along with many (small) examples is available off-the-web. On the flip side, the

original version was not under active development or usage, and we did not find any programs written by independent developers.

In order to address the questions we posed earlier, we quantitatively analyze the performance of Actor Foundry v0.1.14 to understand the costs associated with actor semantics.

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).

5.2.1 A Crude Comparison using the Threadring Benchmark

In order to analyze the cost of supporting these properties, we implement a small benchmark called `Threadring` [79] in which 503 concurrent entities pass a token around in a ring 10 million times. `Threadring` provides a crude estimate for the overhead of message-passing and context switching.

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 5.2 for a full comparison.

These results suggest that a faithful but naïve implementation of the standard actor semantics can have a significantly high execution overhead. We analyze the performance bottlenecks in a framework supporting the semantic properties. For this purpose, we choose Actor Foundry v0.1.14 as our implementation platform.

5.2.2 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 [69]. The goal was to develop a modular Actor framework for a new, upcoming object-oriented

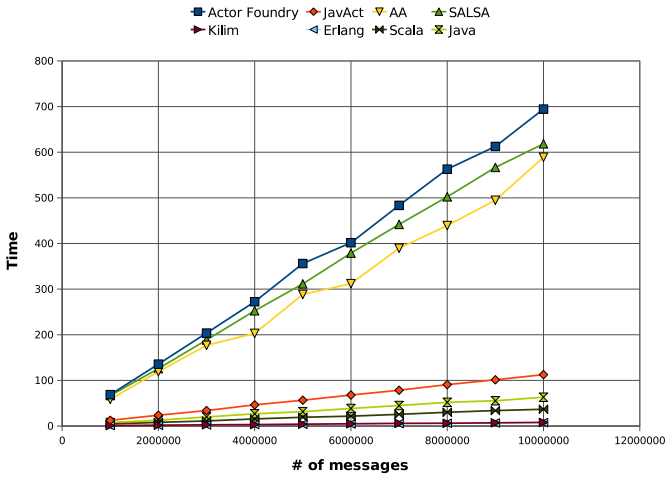

Figure 5.2: Threadring Performance - Actor Foundry v0.1.14 compared with other concurrent languages and frameworks

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. 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.

Note that this approach towards deep copying is tolerant of any cycles in the object graph. Actor Foundry v0.1.14 also supports distributed execution of actor programs, location transparency and weak mobility [42]. Actor Foundry v0.1.14 does not implement an automatic actor garbage collection mechanism [14].

5.2.3 Cost of Creating Actors

Since each actor in Actor Foundry v0.1.14 maps to a JVM thread, actor creation entails thread creation. Therefore, the cost of creating an actor is a function of the cost of creating a JVM thread.

5.2.4 Cost of Context Switching

Similarly, switching an actor entails thread context switching. 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.

As these results suggest, the creation of actors and context switches in Actor Foundry v0.1.14 are a major source of inefficiency for this actor implementation.

5.2.5 Cost of Sending Messages

We further profile the execution to identify 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 result is another large source of inefficiency. Figure 5.3 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.

Despite such a high overhead, frameworks such as SALSA, Actor Architecture and Actor Foundry v0.1.14 are useful for programming coarse-grained

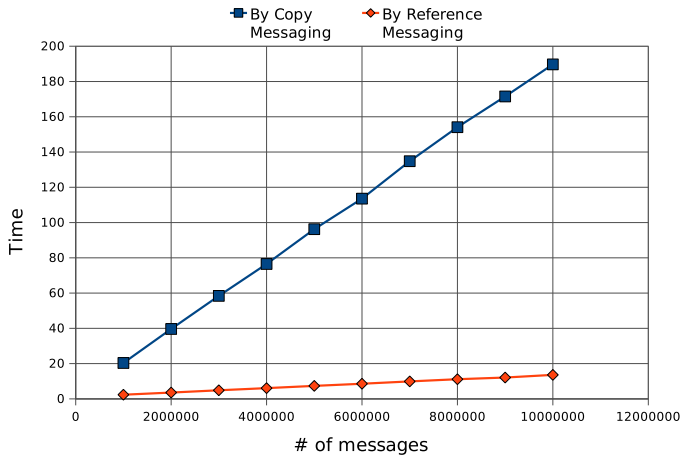

Figure 5.3: Graph showing the cost of sending messages in ActorFoundry by reference versus by making a deep copy.

concurrency (relatively higher computation-to-communication ratio) or distributed applications (higher tolerance for communication overhead and typically coarse-grained).

However, these overheads are prohibitive for an application with fine-grained actors. In the next two sections, we discuss implementation strategies which reduce the cost of supporting actor semantics.

CHAPTER 6

TECHNIQUES FOR IMPROVING PERFORMANCE - ROUND 1

In this chapter, we study strategies that mitigate the inefficiences and optimize performance, *without compromising the semantic properties.*

6.1 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, which are typically expensive operations. In particular, thread context switching involves saving the complete computation stack of the thread, program counter and state of other registers. Although this saves the execution state of an Actor, it is an overkill since actors do not share state.

Prior experience with ThAL language [59] suggests that light-weight continuations provide significant performance improvement in the creation as well as context switching of actors. The ThAL compiler creates continuation methods for blocks of code that need to be executed when an actor resumes after blocking. An alternate approach is to create continuation actors, which correspond to actor semantics in terms of execution driven by asynchronous messages.

In order to provide similar support in ActorFoundry, we integrate Kilim's light-weight Task abstraction and its bytecode post-processor ("weaver") [11]. 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 calls its continuation. Actors are assumed to be cooperative; an actor continues to process messages until it runs out of message. At this point, it goes into the idle state and waits for a new message to arrive. 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 §6.2. 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.

6.2 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 na-

tive threads. An unfortunate worst case is when the number of native threads exceeds what can fit in the available JVM heap size, resulting in long delays and potentially program crash. (This scenario 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` [79]. `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 matchmaking 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 6.1 shows that the modified (fair) scheduler incurs negligible overhead for the three benchmarks.

6.3 Zero-Copy Messaging on Shared Memory Platforms

Note that in `Threadring`, the message content is an Integer (token), which is an immutable type and can be safely shared between actors. As a first solution, we disable deep-copying for some known immutable Java types such as Number, Integer and String. This brings down the running time of `Threadring` to 30s.

We also introduce two new methods in the ActorFoundry framework:

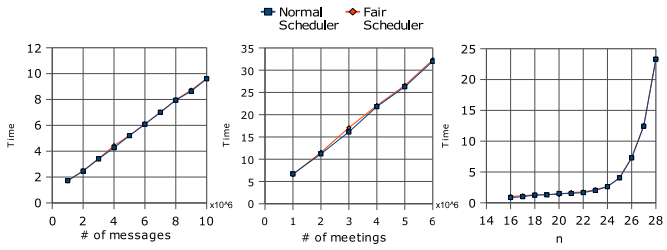

Figure 6.1: Overhead of Fairness for (a) Threadring (b) Chameneos-redux (c) Naïve fibonacci calculator

`sendByRef`() and `callByRef`(). These respectively correspond to `send`() and `call`(), but allow the programmer to send message contents by reference. Thus, these methods enable an explicit declaration of ownership transfer semantics for messages. However, we believe it is desirable to have a compiler and runtime system to infer cases when it is safe to do so.

6.4 Performance Revisited

Figure 6.2 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 6.3 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 4.1, 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 en-

capsulation, fairness, location transparency and mobility. We believe that further significant optimizations for location transparency and mobility are possible.

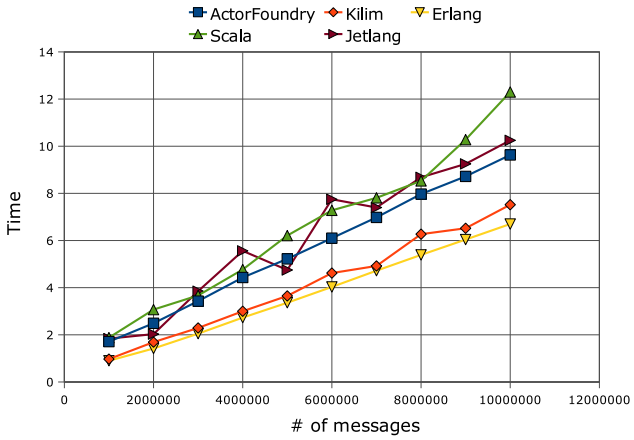

Figure 6.2: Threadring Performance

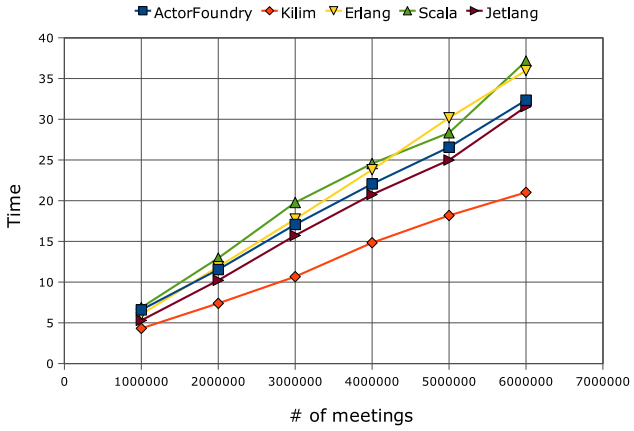

Figure 6.3: Chameneos-redux Performance

6.5 Discussion

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 the work since we realize that Actor frameworks are still evolving.

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 largely relieves the programmer of the burden of reasoning in terms of a dual semantics for message passing. Although a static analysis is necessarily conservative, we believe it is effective much of time. We present details in the next section.

CHAPTER 7

SAFE AND EFFICIENT MESSAGING

In this chapter, we present our static analysis algorithm that detects message contents that can be safely passed by reference. It does so by inferring whether a message site has transfer of ownership semantics. This chapter is part of the joint work that we did with Stas Negara, and we would like to acknowledge his significant contributions.

7.1 Ownership Transfer

An important observation is that many message passing programs tend to have simple structure where messages have an *ownership transfer* semantics, i.e. an actor hands off an object or data stream to another actor by passing it in a message [80]. For example, actors in different stages of a pipeline transfer the data to the next stage after processing it. If message sites that result in transfer of data ownership can be identified in Actor programs, performance may be significantly enhanced [5] by sending references to data instead of making a copy.

7.2 Illustrative Example

We discuss a couple of examples to illustrate the problem of identifying messages that transfer the ownership of its contents, as well as to motivate the different techniques we use to solving the problem. Figure 7.1 presents a code fragment of a `RefMessenger` actor in ActorFoundry v1.0: a `RefMessenger` actor can receive two types of messages, `store` and `transfer` (lines 3 and 6); in response to message `transfer`, it sends two messages `compute` to the actor that is bound to `relayActor` by calling the method `relayPrint` (lines 8-9). We would like to check whether a `RefMessenger` actor transfers the

ownership of `item` to the `relayActor` at line 13, which would make it safe to pass by reference the object `data` to `relayActor` (through the call at line 8).

In order to answer this question, we perform a static points-to analysis; the analysis detects all objects that may "escape" to `relayActor` through the message at line 13. We also perform a static live variable analysis in order to check whether, after sending the message at line 13, an "escaped" object may be accessed by the `RefMessenger` actor. Both these analyses are interprocedural and are performed on a call graph, a directed graph that represents calling relationships between procedures (subroutines) in a program. We provide a brief overview of call graph construction later in the section.

```
1  public class RefMessenger extends Actor {
     ...
2    StringBuffer localName = null;
3    @message public void store(String name) {
4       localName = new StringBuffer(name);
5    }
6    @message public void transfer() {
7       StringBuffer data = new StringBuffer("Hi ");
8       relayPrint(data);
9       relayPrint(localName);
10      data.append(localName);
11   }
12   void relayPrint(StringBuffer item) {
13     send(relayActor, "compute", item);
14   }
15 }
```

Figure 7.1: A simplified version of `RefMessenger` example in ActorFoundry v1.0.

Points-to Analysis: A *points-to analysis* produces a *points-to graph*, which establishes what variables may point to what memory locations. For example, a points-to graph can tell that the two reference variables, `data` and `item`, point to the same `StringBuffer` object allocated at line 7. This particular graph is a result of interprocedural points-to analysis, where allocated objects are represented as nodes denoted with s_i:`T`, where i shows the line number where the object is allocated, and `T` represents its type.

An object that is pointed by variable `item` may escape to `relayActor` through the message at line 13 (Figure 7.1). Thus, performing an interprocedural points-to analysis enables detection of the fact that the object s_7:`StringBuffer` has been sent or *escaped*. However, this analysis is not sufficient to answer the question whether this object is accessed by the

`RefMessenger` actor after having escaped to `relayActor`. We employ a live variable analysis in order to answer the latter question.

Live Variable Analysis: A *live variable analysis* of a program is a dataflow analysis that calculates the set of variables that may be read before being written to in the program. An interprocedural live variable analysis performed on the code fragment in Figure 7.1[1] detects that variable `data` is live after the call to method `relayPrint` at line 8 completes, because its value is read and modified at line 10. Although the scope of the variable `data` is limited to the method `transfer`, and is not visible inside the method `relayPrint`, the object it points to is live throughout the method `relayPrint`, including at the point right after the message to actor `relayActor` is sent (line 13). We require an interprocedural analysis to detect the escaping of the object pointed to by variable `data`, because its intraprocedural counterpart treats every method in isolation and misses the fact that the object is live in method `relayPrint`.

Earlier in the section, we established that the variable `data` points to the object s_7:`StringBuffer`, and that this object escapes to actor `relayActor`. Live variable analysis shows that this object is live in actor `RefMessenger` after escaping to actor `relayActor`. Consequently, we conclude that it is *not* safe to pass the variable `data` by reference, because passing it by reference would result in sharing the object s_7:`StringBuffer` between the `RefMessenger` actor and `relayActor`.

Call Graph Construction: The construction of a *call graph* has a significant impact on both the precision and the speed of interprocedural analysis. Note that our model is that of an *open system*, i.e. we do not assume any information about the outside world while analyzing a particular actor. Thus we need to consider all possible messages that an actor can receive from the outside world.

Consider the `RefMessenger` actor in Figure 7.1. The actor can receive two types of messages, `store` and `transfer`. The execution of `RefMessenger` actor starts when it receive either one of them, and hence the methods `store`

[1]Static analysis performed directly on Java source code serves only for the demonstration purposes. Our tool SOTER performs both points-to and live variable analises on a low-level intermediate representation (IR) in a static single assignment (SSA) form.

and `transfer` serve as two separate entry-points. Our analysis recognizes that they are received by the same instance of `RefMessenger`. This enables us to handle code such as that given in Figure 7.1 where the instance field `localName` is initialized in one message handler (line 4) but escapes in another one (line 13, through the call at line 9). Our analysis correctly detects that the object s_4:`StringBuffer` escapes to actor `relayActor` at line 13.

Context sensitivity plays an important role in call graph construction. A context-insensitive analysis produces more imprecise and smaller call graph compared to a context-sensitive analysis. In a context-insensitive call graph, every invoked method, distinguished by its signature, is represented with a single node regardless of the context in which this method is invoked.

Consider the code example from Figure 7.2 and its context-insensitive call graph. Were we to use this call graph for our points-to analysis, we would decide that linked list `l1` has to be passed to actor `myActor` by value because there are objects that `l1` transitively points to that are live after the program point where `l1` is passed to actor `myActor` (line 7). Although this decision is safe (i.e., it does not produce a data race), it is too conservative and misses an opportunity for optimization.

```
1  public class TestActor extends Actor {
     ...
2    @message public void test(){
3      LinkedList<A> l1 = new LinkedList<A>();
4      LinkedList<A> l2 = new LinkedList<A>();
5      l1.add(new A(1));
6      l2.add(new A(2));
7      send(myActor, "process", l1);
8      l2.add(new A(3));
9    }
10 }
```

Figure 7.2: A code example, whose analysis is highly affected by context-sensitivity of the call graph.

An example of a context-sensitive call graph is a graph that distinguishes invocations of the same method on different receiver instances. Such a call graph would have many more nodes than its context-insensitive counterpart. However it provides much better precision. A *receiver instance context call graph* has two distinct nodes for method `add` of class `LinkedList` $<$ `A` $>$: one node represents invocations on the linked list `l1`, and another node represents invocations on the linked list `l2`. The corresponding points-to graph shows that linked lists `l1` and `l2` transitively point to non-intersecting sets of instances of class `A`, and, consequently, an analysis based on such points-

to graph would correctly decide to pass linked list `l1` to actor `myActor` by reference (line 7).

Therefore, for our static analysis, we construct a receiver instance context call graph in order to exploit the better precision such a call graph offers. Although a context-sensitive call graph is much bigger, it is also considerably sparser than a context-insensitive call graph. As a result, Andersen's points-to analysis performed on a context-sensitive call graph does not take much longer as demonstrated in [81]. For the final step of our analysis, namely the live variable analysis, we describe a custom interprocedural algorithm that scales well for large programs.

7.3 Static Analysis Algorithm

We describe our static analysis using a simple, illustrative actor program presented in Figure 7.3. The program consists of four classes, the last two of which specify actor behavior:

1. Class `MutableValue` is a wrapper around an integer value. The value is assigned when an instance of class `MutableValue` is created and may be changed during the lifetime of this object.

2. Class `ValueHolder` holds a field `MutableValue` and provides a method `getMutableValue` to access the encapsulated object. Instances of class `ValueHolder` are passed between actors. In ActorFoundry, objects between actors are passed by copy, which is implemented using serialization/deserialization of objects. Therefore, both `ValueHolder` and `MutableValue` classes implement the `java.io.Serializable` interface.

3. Class `SumActor` specifies an actor that can receive message `sum` with two arguments of type `ValueHolder`. This message computes the sum of two integer values of `MutableValue` fields of the arguments, stores this sum in the `MutableValue` field of the second argument, and then prints it to the console.

4. Class `ExecutorActor` specifies an actor that can receive message `boot`. The message handler creates an instance of `SumActor` and several in-

stances of `MutableValue` and
`ValueHolder`, and then sends two `sum` messages to the created `SumActor`.

7.3.1 Call Graph Construction

In the first step of our analysis, we construct the program's call graph. As noted earlier, we construct a receiver instance context call graph. It requires identifying all messages that actors of this program can receive, since these serve as entry-points in the constructed call graph. The call graph for the program in Figure 7.3 has two entry-points: one for message `sum` of `SumActor` and another one for message `boot` of `ExecutorActor`.

Figure 7.4 shows a fragment of the constructed call graph that starts from the entry-point for message `boot`. We have omitted the part of the call graph that starts from the entry-point for message `sum`: it does not present any interesting case for our analysis because the functionality of this message does not involve sending messages to other actors. Moreover, we do not show calls to methods that are not defined within the code of class `ExecutorActor` (e.g. framework calls that are inside the body of methods `create` and `send`), and calls that construct the output `String` at line 8. The omitted parts do not affect the analysis, and have been omitted in order to simplify the presentation.

7.3.2 Points-to Analysis

We use the call graph to perform *flow-insensitive* Andersen's points-to analysis [82]. Figure 7.5 illustrates a fragment of the resulting points-to graph for the code from Figure 7.3. This fragment is relevant to our analysis: it presents objects that escape to the actor `sumActor` (lines 20 and 21) and pointers that point to them directly or indirectly.

Our points-to analysis is both *context-sensitive* and *field-sensitive*, i.e. it distinguishes instance fields of different instances of the same class. This allows us to distinguish instance field `mv` of different instances of `ValueHolder` as shown in Figure 7.5, where every instance field `mv` is represented with a separate pointer, whose name prefix corresponds to the name of the containing `ValueHolder` instance (`s18.mv`, `s19.mv`, `s6.mv`).

7.3.3 Live Variable Analysis

The crux of our algorithm is a custom interprocedural live variable analysis, which takes the call graph and points-to graph as input. We perform the live variable analysis in order to detect objects that may be accessed after being passed to other actors. Even the fastest, polynomial time algorithms for this analysis that are precise, for example the algorithm in [83], can effectively handle only small size programs as shown in [84]. Because any actor program is analyzed together with the ActorFoundry framework, which is a relatively large 38.7KLOCs software, we cannot employ such algorithms (e.g. applying an implementation of the algorithm from [83] we ran out of memory even for the smallest actor programs). In order to be able to handle programs of such a large scale, we elaborate a custom algorithm which conservatively assumes that every instance field is live as long as the containing object is live.

The key idea behind our approach is to split an interprocedural analysis into two intraprocedural phases. Figure 7.10 illustrates our two-phase approach. In the first phase (marked with number 1), we perform a standard intraprocedural live variable analysis for a subset of call graph nodes. In the second phase (marked with number 2), we solve a forward data-flow problem defined on the nodes of the constructed call graph. For this problem we do not consider the internal control flow of the call graph nodes. As a result, this analysis is just like a regular intraprocedural analysis, except that we use call graph nodes instead of basic blocks, and call edges instead of control flow edges. Our evaluation (Section 7.5) shows that this algorithm scales well for large programs.

The algorithm takes as input the receiver instance context call graph, *callGraph*, and the results of points-to analysis, *pointstoGraph*, for a given program. The output of the algorithm, *passByValue*, specifies for each argument of every message passing site in the program, whether it needs to be copied. For a particular argument *arg* of a call site *cs* the value of *passByValue[cs,arg]* is *true* when *arg* needs to be copied, and *false* when it is safe to pass *arg* by reference.

The initialization of our algorithm (lines 1-14 in Figure 7.6) computes the set of all message passing sites in a program, *passingCallSites*, and the set of all call graph nodes, *passingNodes*, that contain at least one message passing site. Also, it initializes all entries of *passByValue* to *false*. The algorithm

visits each call site of every call graph node. For every visited call site *cs*, the procedure *isMessagePassingCallSite(cs)* returns *true* if *cs* involves sending a message to another actor (and thus, may escape objects), and *false* in the contrary case. If a call site *cs* may send messages, it is added to *passingCallSites* (line 8) and its containing call graph node is added to *passingNodes* (line 7). In ActorFoundry call sites that may send messages to other actors are calls to methods `send`, `call`, and `create` of class `Actor`. For the program in Figure 7.3, the set *passingCallSites* contains call sites at lines 12, 20, and 21. And the set *passingNodes* includes call graph nodes that contain these call sites. In Figure 7.4 these are `executorActor.add` node for call sites at lines 20 and 21, and `executorActor.execute` node for call site at line 12.

The algorithm then computes *reachingNodes* (line 15 in Figure 7.6) - the set of all call graph nodes that can reach *passingNodes*. The *reachingNodes* is computed by the procedure *transitiveClosure*, which takes the *callGraph* and the *passingNodes* as arguments.

Figure 7.7 shows the procedure *transitiveClosure*, which computes all reaching nodes, *reachingNodes*, that can reach the initial set of nodes, *initNodes*, as a transitive closure of *initNodes* in a particular call graph *callGraph*. The nodes in *reachingNodes* are the only nodes in the call graph, from which the control flow may reach message passing call sites. So, *reachingNodes* contains all call graph nodes that are relevant to our analysis. For our example program *reachingNodes* includes the following nodes from Figure 7.4: `executorActor.add`, `executorActor.execute`, and `executorActor.boot`.

Next, our algorithm applies a standard intraprocedural live variable analysis to collect local variables that are live just after the relevant call sites (line 16 in Figure 7.6). A call site is relevant to our analysis if it is either a message passing call site or is represented as a node in the set *reachingNodes*. Figure 7.8 presents procedure *computeLiveVariables* that takes as input *reachingNodes* and returns *callSiteLiveVariables* which specifies the set of live variables at the program point just after a relevant call site.

For every node *n* from *reachingNodes*, procedure *computeLiveVariables* performs a standard local live variable analysis (line 2) that calculates the set of live variables for every program point in the analyzed node *n*. In order to reduce the memory consumption, we keep the results only for the program points that are relevant to our analysis, i.e. those program points

that are just after relevant call sites (lines 4-9). Relevant call sites and the corresponding sets of live variables for the example program shown in Figure 7.3 are as follows: line 7 - {`mv`}; line 12 - {`sumActor`, `vh`}; line 13 - {}; line 20 - {`sumActor`, `vh2`, `vh3`}; line 21 - {}.

As we demonstrated in Section 7.2, if a variable *var* is live at the program point just after a call site that represents a call of some call graph node *n*, then it is live in node *n* as well. We call such variable *var* a *node live variable* for node *n*, because it is live at every program point inside node *n*. If node *n* contains other call sites, which represent calls of other call graph nodes, then variable *var* is live in those nodes too and so on. This propagation of variable *var* is a forward data-flow problem defined on the nodes of the underlying call graph. Our algorithm uses procedure *propagateLiveVariables* to compute node live variables for every node from *reachingNodes* (line 17 in Figure 7.6).

Figure 7.9 shows procedure *propagateLiveVariables* that propagates live variables forward in the call graph. It takes as input *reachingNodes* and the sets of live variables for all relevant call sites, *callSiteLiveVariables*. The output of this procedure is *nodeLiveVariables*, which specifies for every node from *reachingNodes* the set of node live variables. The initialization part of the procedure (lines 1-8) defines for every node *n* from *reachingNodes* initial values for sets *IN[n]* and *OUT[n]*, which represent correspondingly the set of node live variables at the entry and at the exit of node *n*. Both initial entry and exit sets are a union of all live variables from all call sites that call node *n* (lines 2-7). The computation part of the procedure (lines 9-17) is a fixed-point algorithm for a forward data-flow problem, where the transfer function is identity (line 15), and the meet operator is union (line 13). Procedure *getPredecessors* (line 11) returns a set of call graph nodes that immediately precede the given node. For *reachingNodes* and *callSiteLiveVariables* shown previously for our code example in Figure 7.3, procedure *propagateLiveVariables* computes the following *nodeLiveVariables*: `executorActor.boot` - {}, `executorActor.execute` - {`mv`}, `executorActor.add` - {`mv`}.

In the end (lines 18-31 in Figure 7.6), our algorithm computes for every call site *cs* from *passingCallSites* the set of all live variables, *liveVariables*, as a union of local live variables for call site *cs* and node live variables of the call graph node that contains call site *cs* (line 21). Next, we use *pointstoGraph* to compute the set of all live objects (lines 22-24). Then, for every argument *arg*

of call site *cs* we compute all objects that *arg* points to, which is the set of objects that may escape to other actors (line 26). Finally, if the intersection of the objects that are live after call site *cs*, *liveObjects*, and the objects that may escape, *escapedObjects*, is not empty, we mark that *arg* should be passed by value (lines 27-29).

For the example program in Figure 7.3, our algorithm establishes that: call site at line 20 – argument `vh1` can be passed by reference, argument `vh2` should be passed by value; call site at line 21 – argument `vh2` can be passed by reference, argument `vh3` should be passed by value. Argument `vh2` of the call site at line 20 should be passed by value, because variable `vh2` is live at the program point right after the call site at line 20. Argument `vh3` of the call site at line 21 should be passed by value, because according to the points-to graph in Figure 7.5, it transitively points to the object *s5:MutableValue*, which is live, as it is the same object node live variable *mv* of node `executorActor.add` points to.

7.4 Algorithm Properties

Our interprocedural live variable analysis consists of two related but distinct phases. In the first phase, we perform a standard intraprocedural live variable analysis for a subset of call graph nodes. Specifically, as shown above, we consider only those nodes of the call graph that are relevant to our analysis. Program statements of a call graph node are translated into an intermediate representation (IR) in a static single assignment (SSA) form, where every variable is assigned exactly once. Such representation significantly reduces both the time and the complexity of intraprocedural live variable analysis. In the second phase, we solve a forward data-flow problem defined on the nodes of the constructed call graph. For this problem we do not consider the internal control flow of the call graph nodes. As a result, this analysis is just like a regular intraprocedural analysis, except that we use call graph nodes instead of basic blocks and call edges instead of control flow edges.

Figure 7.10 illustrates our two-phase approach. In the first phase (marked with number 1) we perform intraprocedural live variable analysis on the control flow graphs of individual call graph nodes. In the second phase (marked with number 2) we propagate live variables forward in the call graph, dis-

regarding the internal control flow of the call graph nodes. Splitting an interprocedural analysis into two phases, both of which are intraprocedural by nature, makes our algorithm fast and scalable for large programs as demonstrated in Section 7.5. The trade off is the reduced precision of our analysis. Although our analysis conservatively assumes that every instance field is live as long as the containing object is live, the evaluation results presented in Section 7.5 show that it is able to detect the majority of optimization opportunities for a variety of actor programs.

Complexity. Our algorithm consists of a standard Andersen's points-to analysis, followed by a standard intraprocedural live variable analysis for a subset of call graph nodes n, and a forward data flow problem for live variable propagation. The last two phases are intraprocedural by nature and have a comparable complexity. Hence, the complexity of our algorithm is $O($ complexity of Andersen's points-to analysis $+(n+1)*$ complexity of the intraprocedural live variable analysis).

Soundness. An argument *arg* of a message passing call site *cs* in a call graph node n is marked by our algorithm to be passed by value, if *arg* transitively points to at least one object o that is live after message passing call site *cs* (lines 18-31 in Figure 7.6). Considering that the employed Andersen's points-to analysis is sound, it is sufficient to demonstrate that our interprocedural live variable algorithm does not miss any objects that are live after some message passing call site. If there is a live object o then there should be at least one live variable *var* that transitively points to o. There are three kinds of variables that can be live after some message passing call site *cs* in a call graph node n:

- Variable *var* is a local variable in the call graph node n. Such variable is detected as live at the program point right after the message passing call site *cs* in the first phase of our algorithm, where we perform a standard intraprocedural live variable analysis.

- Variable *var* is a local variable in an immediate or a transitive caller node of the call graph node n. The second phase of our algorithm propagates such variable to the call graph node n, and variable *var* becomes a node live variable for node n.

- Variable *var* is an instance field of the class whose method is represented

with the call graph node n. Our algorithm conservatively assumes that every instance field is live as long as the containing object is live. In this case, the containing object for variable *var* is object `this` in the method represented with the node n. Object `this` is always live in any instance method, and so *var* is live as well.

Thus, if variable *var* is live after some message passing call site, our algorithm detects this regardless of the kind of *var*. Consequently, all objects variable *var* points to are detected as live, including object o.

Termination. Observe that the first phase of our algorithm performs a standard intraprocedural live variable analysis for a subset of call graph nodes, *reachingNodes*. An intraprocedural live variable analysis for a call graph node terminates, and the number of nodes in a call graph is finite. Thus, the first phase terminates. In the second phase of our algorithm, we solve a data-flow problem using a fixed-point algorithm (lines 9-17 in Figure 7.9). For every node n from the set of nodes *reachingNodes* the set *OUT[n]* of node live variables never shrinks. Considering that the number of variables in a program is finite and the number of call graph nodes is finite, the fixed-point algorithm eventually reaches a point, when *OUT[n]* does not change for any node $n \in$ *reachingNodes*, and terminates. Thus, the second phase terminates. Both phases of our algorithm terminate and so, our algorithm terminates.

7.5 Implementation and Evaluation

SOTER (for *Safe Ownership Transfer enablER*[2]) is a Java implementation of the static analysis described in Section 7.3. SOTER uses IBM T. J. Watson Libraries for Analysis (WALA) framework [85] that provides a flow-insensitive Andersen's points-to analysis and an infrastructure for implementing data-flow analysis. For both the call graph construction and the point-to analysis, we only specify the entry points as well as some configuration options such as context-sensitivity.

Our analysis algorithm is language-independent. The current implementation takes Java bytecode as input, and thus can be easily extended to

[2]Soter is also the name of the Greek god of safety, deliverance, and preservation from harm.

handle programs in any language or framework that compiles to Java bytecode (e.g. Kilim, Jetlang, SALSA). We initially implemented support for ActorFoundry [10]. Later, we extended SOTER to support Scala [67] programs, which took only a couple of weeks of part-time effort. SOTER's source code as well as ActorFoundry and Scala subject programs can be found at http://osl.cs.uiuc.edu/soter

We performed all experiments on a 4-core 2.4GHz, 3GB RAM machine. Any ActorFoundry actor program is analyzed together with ActorFoundry framework, a relatively large software, whose bytecode size is 726KB. In the worst case our analysis took around 24 seconds.

The goal of the evaluation is to assess the effectiveness and usefulness of SOTER. To achieve this goal, we applied SOTER on a variety of ActorFoundry programs.

7.5.1 ActorFoundry

For ActorFoundry actor programs, we would like to answer two questions:

- **Effectiveness:** How many opportunities to safely pass a message contents by reference are detected by SOTER in comparison to the total number of such opportunities and to what fairly sophisticated programmers can manually achieve?
- **Usefulness:** What is the performance improvement achieved by SOTER?

Table 7.1 presents results that assess the effectiveness of SOTER. Each row displays data for a particular actor program, whose name appears in the second column. The first column reflects the general category of an actor program.

These categories include programs from the ActorFoundry distribution, **'Benchmarks'** refers to the programs used in an earlier study [35, 38], **'Synthetic'** category is attributed to actor programs written specifically to test our analysis, and **'Real world'** programs are those written by advanced students in the Software Engineering course in Computer Science at Illinois. All presented actor programs except those from **Synthetic** category were written without the knowledge of a tool such as ours. The third column, **LOC**,

Table 7.1: The effectiveness of SOTER on different ActorFoundry actor programs. Size of ActorFoundry library, AFL=726KB. #1 = threadring, #2 = concurrent, #3 = copymessages, #4 = performance, #5 = pingpong, #6 = refmessages, #7 = rpcping, #8 = sor, #9 = chamenos, #10 = fibonacci, #11 = leader, #12 = philosophers, #13 = pi, #14 = shortestpath, #15 = quicksortCopy, #16 = quicksortCopy2, #17 = clownfish, #18 = rainbow_fish, #19 = swordfish, #20 = threadfin

Program	LOC	Passed args.	Ideal by ref.	SOTER by ref.	Human misses	SOTER/ Ideal ratio	Analysis time (se)
AF distribution							
#1	43	7	7	7	1	100%	3.4
#2	204	12	12	7	N/A	58%	3.8
#3	80	19	18	10	5	56%	12.5
#4	126	14	14	12	N/A	86%	3.6
#5	62	9	9	8	N/A	89%	3.5
#6	20	3	3	2	2	67%	3.3
#7	65	9	9	9	N/A	100%	3.4
#8	320	36	36	18	10	50%	3.8
Benchmarks							
#9	187	12	12	4	1	33%	3.5
#10	53	28	28	24	N/A	86%	3.5
#11	81	12	12	2	N/A	17%	3.4
#12	77	6	6	6	N/A	100%	3.3
#13	73	6	6	4	N/A	67%	3.4
#14	126	59	59	52	N/A	88%	3.5
Synthetic							
#15	76	3	3	3	N/A	100%	12.5
#16	92	8	8	6	N/A	75%	12.4
Real world							
#17	700	87	87	59	N/A	68%	24.0
#18	591	68	68	67	N/A	99%	3.6
#19	615	83	83	13	N/A	16%	4.1
#20	471	101	101	98	N/A	97%	12.8

Table 7.2: The performance improvement achieved by SOTER.

Program	# actors	Execution time (ms)		Speed up	Ideal execution time (ms)
		Before	After		
threadring	504	25870	1880	13.76	1880
concurrent	601	187510	15990	11.73	1390
copymessages	31810	7730	3710	2.08	610
sor	6402	76960	61620	1.25	5890
chameneos	14	56890	36640	1.55	1620
leader	30001	14050	8190	1.72	7380
philosophers	60001	9550	1380	6.92	1380
pi	3002	4210	3890	1.08	3880
quicksortCopy	200002	24660	4530	5.44	4530
quicksortCopy2	200002	16320	4870	3.35	3580

shows the number of lines of code in the program (not counting comments and blank lines). Although the size of the programs may seem relatively small, they represent a wide variety of programmers and purpose. Moreover, these programs are written on top of an Actor library. A library encapsulates much of the functionality required to express an actor program, and therefore the actor code itself has a smaller size than it would have without a library-based approach.

The fourth column, **Passed arguments**, represents the total number of message passing call site arguments present in the code of an actor program. The following two columns show correspondingly the number of arguments that could be safely passed by reference, having an ideal understanding of the analyzed program,[3] and the number of arguments that SOTER reports as safe to be passed by reference. The next column, **Human misses**, presents the number of arguments that are safe to be passed by reference, which are missed by developers (advanced CS students at Illinois), who manually optimized the program. N/A in this column means that the program is not manually optimized. The following column displays the effectiveness of SOTER, i.e. the ratio of detected opportunities to safely pass arguments by reference to the total number of such opportunities.

SOTER is quite effective: on average it is able to detect around 71% of available optimization opportunities. Moreover, it detects some opportunities missed by developers. The last column shows how long it takes SOTER to

[3]Note that complete knowledge of the semantics of the analyzed program yields far better results than any possible static analysis.

analyze the corresponding actor program. Our analysis is quite fast: for ActorFoundry actor programs it does not exceed 24 seconds.

Table 7.2 shows the performance improvement achieved by SOTER by comparing the execution time of actor programs before and after application of SOTER. We exclude actor programs whose execution time is too small to base our evaluation on. For the majority of actor programs evaluated, SOTER speeds up the execution more than twice, and for two of them, by more than an order of magnitude. The last column reflects the execution time of actor programs, where all arguments that could be safely passed by reference having an ideal understanding of the actor program are indeed passed by reference. Note that for the majority of actor programs, the ideal execution time and the execution time after applying SOTER are very close. However, for some actor programs, there is still considerable room for improvement even after applying SOTER, which is mainly due to the conservatism of our static analysis. We discuss possible extensions of our analysis in the paper [12].

```
public class MutableValue implements java.io.Serializable{
  private int value;
  public MutableValue(int value){
    this.value = value;
  }
  public int getValue(){
    return value;
  }
  public void setValue(int value){
    this.value = value;
  }
}

public class ValueHolder implements java.io.Serializable{
  private MutableValue mv;
  public ValueHolder(MutableValue mv){
    this.mv = mv;
  }
  public MutableValue getMutableValue(){
    return mv;
  }
}

public class SumActor extends Actor{
  @message public void sum(ValueHolder vh1, ValueHolder vh2){
    int val = vh1.getMutableValue().getValue();
    MutableValue mv = vh2.getMutableValue();
    mv.setValue(mv.getValue() + val);
    System.out.println("Sum:" + mv.getValue());
  }
}
```

```
1  public class ExecutorActor extends Actor{
2    @message
3    public void boot(Integer val)
4                throws RemoteCodeException{
5      MutableValue mv = new MutableValue(val);
6      ValueHolder vh = new ValueHolder(mv);
7      execute(vh);
8      System.out.println("val:" + mv.getValue());
9    }
10   private void execute(ValueHolder vh)
11                throws RemoteCodeException{
12     ActorName sumActor = create(SumActor.class);
13     add(sumActor, vh);
14   }
15   private void add(ActorName sumActor, ValueHolder vh3){
16     MutableValue mv1 = new MutableValue(1);
17     MutableValue mv2 = new MutableValue(2);
18     ValueHolder vh1 = new ValueHolder(mv1);
19     ValueHolder vh2 = new ValueHolder(mv2);
20     send(sumActor, "sum", vh1, vh2);
21     send(sumActor, "sum", vh2, vh3);
22   }
23 }
```

Figure 7.3: A running example of an actor program. Import statements are omitted due to space considerations.

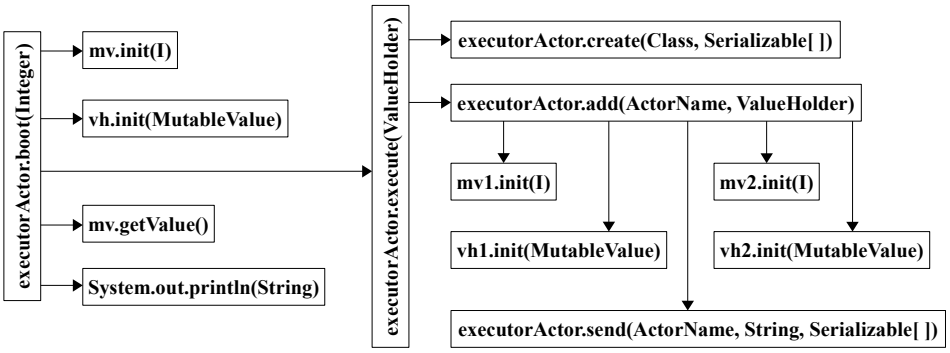

Figure 7.4: Filtered call graph for the code fragment from Figure 7.3.

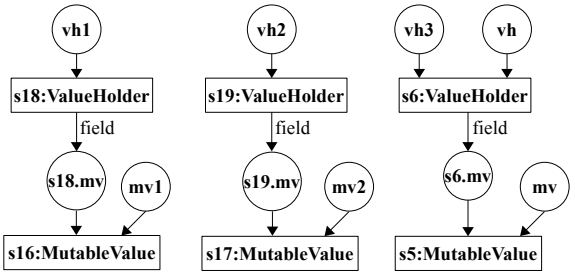

Figure 7.5: Filtered points-to graph for the code fragment from Figure 7.3.

```
input: callGraph, pointstoGraph
output: passByValue
1  passingNodes = ∅;
2  passingCallSites = ∅;
3  foreach (Node n: callGraph){
4    callSites = getContainedCallSites(n);
5    foreach (CallSite cs: callSites){
6      if (isMessagePassingCallSite(cs)){
7        passingNodes = passingNodes ∪ n;
8        passingCallSites = passingCallSites ∪ cs;
9        foreach (Argument arg: cs){
10         passByValue[cs,arg] = false;
11       }
12     }
13   }
14 }
15 reachingNodes =
     transitiveClosure(callGraph, passingNodes);
16 callSiteLiveVariables =
     computeLiveVariables(reachingNodes);
17 nodeLiveVariables =
     propagateLiveVariables(reachingNodes,
                            callSiteLiveVariables);
18 foreach (CallSite cs: passingCallSites){
19   Node n = getContainingNode(cs);
20   liveObjects = ∅;
21   liveVariables =
       callSiteLiveVariables[cs] ∪ nodeLiveVariables[n];
22   foreach (LiveVariable var: liveVariables){
23     liveObjects = liveObjects ∪
                     getPointedObjects(pointstoGraph, var);
24   }
25   foreach (Argument arg: cs.getArguments()){
26     escapedObjects =
         getPointedObjects(pointstoGraph, arg);
27     if ((escapedObjects ∩ liveObjects) ≠ ∅){
28       passByValue[cs,arg] = true;
29     }
30   }
31 }
```

Figure 7.6: Overview of our algorithm for interprocedural live variable analysis.

```
procedure transitiveClosure
input: callGraph, initNodes
output: reachingNodes
1  reachingNodes = ⊘;
2  foreach (Node n: initNodes){
3    workList = {n};
4    while (workList ≠ ⊘){
5      workNode = pop(workList);
6      if (workNode ∉ reachingNodes){
7        reachingNodes = reachingNodes ∪ workNode
8        callers = getCallers(callGraph, workNode);
9        append(workList, callers);
10     }
11   }
12 }
13 return reachingNodes;
```

Figure 7.7: Collecting all call graph nodes that reach *initNodes*.

```
procedure computeLiveVariables
input: reachingNodes
output: callSiteLiveVariables
1  foreach (Node n: reachingNodes){
2    OUT = performLocalLiveVariableAnalysis(n);
3    callSites = getContainedCallSites(n);
4    foreach (CallSite cs: callSites){
5      if (isMessagePassingCallSite(cs) OR
6        (getCalledNode(cs) ∈ reachingNodes)){
7        callSiteLiveVariables[cs] = OUT[cs];
8      }
9    }
10 }
11 return callSiteLiveVariables;
```

Figure 7.8: Performing local live variable analysis for *reachingNodes* and storing its relevant part in *callSiteLiveVariables*.

```
procedure propagateLiveVariables
input: reachingNodes, callSiteLiveVariables
output: nodeLiveVariables
1  foreach (Node n: reachingNodes){
2    IN[n] = ∅;
3    callSites = getCallingCallSites(n);
4    foreach (CallSite cs: callSites){
5      IN[n] = IN[n] ∪ callSiteLiveVariables[cs];
6    }
7    OUT[n] = IN[n];
8  }
9  do{
10   foreach (Node n: reachingNodes){
11     predecessors = getPredecessors(n);
12     foreach (Node pred: predecessors){
13       IN[n] = IN[n] ∪ OUT[pred];
14     }
15     OUT[n] = IN[n];
16   }
17 } while (changes to any OUT occur);
18 foreach (Node n: reachingNodes){
19   nodeLiveVariables[n] = OUT[n];
20 }
21 return nodeLiveVariables;
```

Figure 7.9: Propagating live variables through *reachingNodes*.

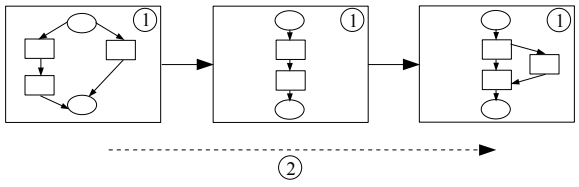

Figure 7.10: Two phases of our live variable analysis. In the first phase (marked with number 1) we consider internal control flow graphs of individual call graph nodes for the classical intraprocedural live variable analysis. In the second phase (marked with number 2) we propagate live variables forward in the call graph, disregarding the internal control flow of the call graph nodes.

CHAPTER 8

REAL-WORLD CASE STUDIES

Our work thus far suggests good performance improvements can be achieved for Actor frameworks; however the performance has been measured for small benchmarks and programs. We would like to observe how these techniques scale to large programs, and be able to validate these results for real-world applications. Specifically, the applications characteristics we are looking for are a large number of actors including many fine-grained actors, and sufficiently complex but arbitrary interactions between them. (Data parallelization or map-reduce do not qualify since a large body of research work on optimizing these specific patterns already exists.)

We believe that an application with such characteristics magnifies the effect of (a) multicore architecture, and (b) distribution, on efficient execution. However, in this work we focus on the effect of multicores only.

In our quest for large real-world applications, domains such as transaction processing systems, NoSQL databases, simulations, and games fit the characteristics discussed above. Games are particularly promising because they tend to have internal concurrency (with complex interactions), independent of the number of external user requests. Morever, games is a popular domain among client-side applications. For example, games consistently dominate the list of top apps in Android Market and Apple Store. According to the business and technology news website, Business Insider Intelligence, in a recent check of Apple's iPhone App Store, games represented 55 percent of the top 200 paid apps, and 33 percent of the top 200 free apps. Many studies report that games represent more than a third of the time spent on mobile apps and three-fourth of the money spent on mobile apps.

We look at multiple games, both of the real-time strategy and turn-based strategy genres. For example, Herzog3D is a single-threaded, real-time strategy game. For every user input, the game loop updates the state of every game object. This approach results in an over-constrained and inefficient

execution. In a natural implementation of a real-time strategy game such as Herzog3D, many games objects would execute autonomously in parallel, react to events from the environment and interact with each other obeying the relaxed constraints. Actors are a natural model for describing these game objects. Although we did not complete porting Herzog3D to ActorFoundry, we next describe our experience with a similar real-time strategy game called Quantum.

8.1 Case Study - Quantum Game

Specifically, we look at an open-source Java game called Quantum [1]. Quantum is an open-source port of the Dyson game to Java, and has more than 25k lines of source code. Dyson is a real-time strategy game that involves multiple players controlling pre-owned *planets*, *trees*, and *creatures* that orbit planets. Players can create more creatures by building trees, and colonize other planets in the universe by moving creatures to them. Creatures can be in one of three modes: orbit, move or attack. The goal of the game is it to eliminate all enemy creatures and take over enemy planets.

The game play comprises of three kinds of objects: planets, creatures and trees. The original version of Quantum from the web has a strict game loop that updates all games objects in every iteration or *game cycle*. In every iteration, creatures interact with planets and other creatures. This version has some concurrency for I/O operations. For example, it has separate threads for receiving user input, producing sound and network play.

8.1.1 Porting Quantum to ActorFoundry

First, we boot this game on ActorFoundry runtime by writing an actor with a single message, whose handler invokes the main method of the original game. This message serves as the application's entry point.

In the first stage of actorizing Quantum, we replace the built-in threads with actors. These include the SoundManager.

In the second stage, we introduce new concurrency by converting game objects such as planets, trees and creatures into actors. We replace the strict game loop, and instead enable each game object to update itself asyn-

chronously by sending an "update" message to itself. The objects also send messages for exchanging state information with other games objects.

After this transformation, we are able to run a game instance with more than 10,000 game objects and run the computer (a Core 2 Duo and an 8-core i7) at full throttle. According to our knowledge and from publicly available information, this is the largest execution of a real-world client-side actor program.

8.1.2 Game Architecture

The execution of the game makes one important assumption: a fair scheduler that enables actors to make progress "uniformly". This ensures that every game object or actor gets updated at the same rate relative to other objects. Furthermore, the graphics renderer executes in a separate thread and is able to read the internal state of all actors directly *without* exchanging messages.

8.1.3 Further Performance Issues

We also compare its performance with that of the multi-threaded version, and observe that it is significantly slower than the two versions (4 minutes 45 seconds vs 35 seconds).

We note that during each update step, a creature sends a message to its host planet. A creature also sends some messages to another creature at nearly every step. Although the game implementation has some communication-computation overlap, many CPU cycles are spent in delivering a message (by the sender), and later by the receiver in pulling it from its mailbox, and then repeating these steps for the reply, if needed. Separate experiments on a Core 2 Duo processor suggest that it takes two orders of magnitude more time for a request-reply message in comparison to reading the value simply through a method call.

Next, we introduce a "short-circuit" implementation for request-reply messages. If the recipient actor of such a message is *local*, and is neither busy nor blocked, the sending actor calls the message handler in its own stack instead of delivering a message to recipient's mailbox [13]. Separate experiments suggest that this technique improves performance by an order of

magnitude. As a result, the performance of Quantum improves to 4 minutes 15 seconds. While this suggests a fine-grained actor implementation is considerably slower than a multi-threaded version on shared memory multicore processors, it should be noted that we have not implemented a number of low-level optimization techniques such as those related to improving cache performance. Moreover, observe that as multicore architectures scale and behave like distributed computers, shared memory may not be available at the same scale [86, 87, 88]. These arguments are further discussed in Section 9.1.

8.2 Other Game Scenarios

We also look at another open-source Java game called Domination [89]. Domination is a turned-based strategy game with a multi-threaded, networked implementation. The game implementation employs Java threads for handling IO, UI and network events. We convert the multi-threaded game into an actor version, where the asynchrony due to threads and communication between them is expressed using actors and messages. However, no new concurrency or parallelism is introduced. Hence the actors in this version are coarse-grained. Similar to Quantum, additional parallelism can be introduced in Domination by actorizing other objects in the game and the search algorithm. This will naturally result in relatively fine-grained actors. We are able to boot the coarse-grained version of the game in ActorFoundry and resulting performance is comparable to the original multi-threaded version.

CHAPTER 9

CONCLUSION AND FUTURE WORK

We believe that this work makes a significant impact in identifying inefficiencies and improving the performance for executing fine-grained actors on modern multicore processors.

A recent paper describing an industrial strength implementation of the Actor model (Akka actors) suggests that some of the design choices and implementation were in part influenced by the questions we raised, and the static and dynamic techniques we proposed in this work [90].

The source code and the documentation for ActorFoundry and our case study is available for download on this link:
http://osl.cs.illinois.edu/software/actor-foundry.

9.1 Multicore Architecture

The two key features of multicore architecture that impact performance are shared memory and cache hierarchy. In our current implementation, actors are mapped dynamically to threads that share a scheduler queue (through the shared memory and shared cache). Because the worker threads share a common scheduler queue, there exists thread contention over accessing this protected data structure. The performance cost of this contention can be prohibitive, specially if there is a large number of threads or the computation per message is small.

Moreover, it is possible that an actor is scheduled by Thread-1 for its first message, and it is scheduled by Thread-2 for its next message. Since there is no static assignment of actors to the threads, an actor with a pending message can be scheduled by any idle thread. While this strategy seems appropriate for achieving load-balancing, it may be detrimental for the overall performance due to its sub-optimal cache behavior– given that it produces

poor temporal and spatial locality. The effects of poor locality such as frequent cache misses and degraded performance have been well-studied, even in the context of sequential computing [91, 92]. This strategy is akin to managing and switching between multiple contexts, and has shown to have a high cache-performance cost [93].

Our work establishes a research platform to study this trade-off. The initial tasks could be to distribute and isolate the currently shared data structures (including the work load) between the different threads running on a single shared memory multicore node (processor). The goal would be to find a sweet spot in terms of an implementation strategy that provides a static mapping to improve cache behavior, and yet allows some dynamicity to achieve load-balancing. We anticipate the resulting runtime architecture on the single multicore node to be nearly distributed. This trend of distributing computation on a single node is also suggested by some processor architectures from Intel [86] and Tilera [87].

Further down, our study can also provide insights into finding the optimal number of actors per thread that results in optimal cache behavior. A similar problem in the context of virtual machines on cloud computers has been recently studied [94].

It has also been argued recently that for certain classes of programs, such as those with a divide-and-conquer pattern, can be solved efficiently by combining the Actor model with task-based approaches [30]. Such an integration allows parallel code to be written inside an actor where the parallel threads can share state or access separate portions of a shared data structure.

9.2 Distribution

Our current work does not focus on optimizations related to distribution of computation across nodes in a grid or cloud. As a large number of actors are created and distributed, the inefficiency of managing name tables is exacerbated. Note that the name tables are required to implement location-independence and mobility. A naïve implementation that adds an entry into name tables for every actor that is ever created can result in a large space overhead as well as (cumulative) lookup time. However, we believe this overhead can be mitigated through various implementation strategies. We

propose a runtime technique that adds entries in name tables selectively and reactively. Specifically, an entry in the name table is required when actors or their names escape their original "birth" node. The dynamic technique can be augmented with a static analysis that will try to identify the coupling between actors so that highly-coupled actors can be clustered together on the same node.

9.3 Lessons Unrelated to Performance

In this work, as we converted the various game objects into actors, we faced some challenges in expressing the (relaxed but semantically correct) game constraints between the fine-grained actors.

During the exercise of actorizing games, we also learned some of the design patterns of actor-oriented programming. Also note that we are manually converting multi-threaded and single-threaded programs into actor programs. We hope to document these steps for future research on automatic or semi-automatic refactorings for introducing actor design into existing programs.

REFERENCES

[1] M. Zechner, "Quantum," *http://code.google.com/p/quantum-game/*, 2009.

[2] R. K. Karmani and G. Agha, "Actors," *Springer's Encyclopedia of Parallel Computing*, 2011.

[3] G. Agha, I. A. Mason, S. Smith, and C. Talcott, "A Foundation for Actor Computation," *Journal of Functional Programming*, vol. 7, no. 01, pp. 1–72, 1997.

[4] H. Kopetz, "Component-based design of large distributed real-time systems," in *Journal of IFAC, Pergamon Press*, 1997, pp. 53–60.

[5] R. K. Karmani, A. Shali, and G. Agha, "Actor frameworks for the JVM platform: a comparative analysis," in *PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*. New York, NY, USA: ACM, 2009, pp. 11–20.

[6] P. Haller and M. Odersky, "Capabilities for uniqueness and borrowing," in *ECOOP 2010 Object-Oriented Programming*, ser. Lecture Notes in Computer Science, T. DHondt, Ed. Springer Berlin / Heidelberg, 2010, vol. 6183, pp. 354–378.

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

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

[9] D. Caromel, L. Henrio, and B. P. Serpette, "Asynchronous sequential processes," *Information and Computation*, vol. 207, no. 4, pp. 459 – 495, 2009.

[10] *The Actor Foundry: A Java-based Actor Programming Environment*, Open Systems Laboratory, University of Illinois at Urbana-Champaign, 1998-09. [Online]. Available: http://osl.cs.uiuc.edu/af

[11] S. Srinivasan, "A thread of one's own," in *Workshop on New Horizons in Compilers*, vol. 4. Citeseer, 2006.

[12] S. Negara, R. K. Karmani, and G. Agha, "Inferring ownership transfer for efficient message passing," in *In proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. New York, NY, USA: ACM, 2011.

[13] W. Kim and G. Agha, "Efficient support of location transparency in concurrent object-oriented programming languages," in *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 1995, p. 39.

[14] N. Venkatasubramanian, G. Agha, and C. 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.

[15] A. Vardhan and G. Agha, "Using passive object garbage collection algorithms for garbage collection of active objects," in *ISMM '02: Proceedings of the 3rd international symposium on Memory management*. New York, NY, USA: ACM, 2002, pp. 106–113.

[16] W.-J. Wang, C. Varela, F.-H. Hsu, and C.-H. Tang, "Actor garbage collection using vertex-preserving actor-to-object graph transformations," in *Advances in Grid and Pervasive Computing*, ser. Lecture Notes in Computer Science, vol. 6104. Springer Berlin Heidelberg, 2010, pp. 244–255.

[17] V. A. Korthikanti and G. Agha, "Towards optimizing energy costs of algorithms for shared memory architectures," in *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, ser. SPAA '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1810479.1810510 pp. 157–165.

[18] X. Zhao and N. Jamali, "Fine grained per-core frequency scheduling for power efficient multicore execution," in *Proceedings of the 2nd IEEE International Green Computing Conference (IGCC 2011) [Work-in-Progress Workshop]*. Orlando, Florida: IEEE, July 2011, pp. 1–8.

[19] S. Frlund and G. Agha, "A language framework for multi-object coordination," in *ECOOP*, ser. Lecture Notes in Computer Science, O. Nierstrasz, Ed., vol. 707. Springer, 1993, pp. 346–360.

[20] S. Frølund and G. Agha, "Abstracting interactions based on message sets," in *ECOOP '94: Selected papers from the ECOOP '94 Workshop on Models and Languages for Coordination of Parallelism and Distribution, Object-Based Models and Languages for Concurrent Systems*, ser. Lecture Notes In Computer Science. Springer-Verlag, 1995, pp. 107–124.

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

[22] G. Agha and C. J. Callsen, "Actorspace: an open distributed programming paradigm," in *Proceedings of the fourth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP)*. New York, NY, USA: ACM, 1993, pp. 23–32.

[23] N. Jamali, P. Thati, and G. A. Agha, "A actor-based architecture for customizing and controlling agent ensembles," *Intelligent Systems and their Applications, IEEE*, vol. 14, no. 2, pp. 38–44, 1999.

[24] G. Agha, N. Jamali, and C. Varela, "Agent naming and coordination: Actor based models and infrastructures," in *Coordination of Internet agents*. Springer-Verlag, 2001, pp. 225–246.

[25] N. Jamali and G. Agha, "Cyberorgs: A model for decentralized resource control in multi agent systems," in *Proceedings of Workshop on Representations and Approaches for Time-Critical Decentralized Resource/Role/Task Allocation, at the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 03), Melbourne, Australia, July*, 2003.

[26] P. Dinges and G. Agha, "Scoped synchronization constraints for large scale actor systems," in *COORDINATION*, ser. Lecture Notes in Computer Science, M. Sirjani, Ed., vol. 7274. Springer, 2012, pp. 89–103.

[27] M. Charalambides, P. Dinges, and G. Agha, "Parameterized concurrent multi-party session types," in *Proceedings 11th International Workshop on Foundations of Coordination Languages and Self Adaptation, Newcastle, U.K., September 8, 2012*, ser. Electronic Proceedings in Theoretical Computer Science, N. Kokash and A. Ravara, Eds., vol. 91. Open Publishing Association, 2012, pp. 16–30.

[28] S. Ren and G. A. Agha, "Rtsynchronizer: language support for real-time specifications in distributed systems," *SIGPLAN Not.*, vol. 30, no. 11, pp. 50–59, 1995.

[29] S. Ren, G. Agha, and M. Saito, "A modular approach for programming distributed real-time systems," in *Journal of Parallel and Distributed Computing, vol. 36, no. 1, pp 4-12, 1996. Also published in School on Embedded Systems, European Educational Forum 1996, pp 52-*, 1996.

[30] S. Imam and V. Sarkar, "Integrating task parallelism with actors," in *ACM SIGPLAN Notices*, vol. 47, no. 10. ACM, 2012, pp. 753–772.

[31] S. Tasharofi, P. Dinges, and R. Johnson, "Why do scala developers mix the actor model with other concurrency models?" in *ECOOP'13 - Object-Oriented Programming, 27th European Conference, Montpellier, France, July 1–6, 2013*, 2013, accepted for publication.

[32] S. T. Heumann, V. S. Adve, and S. Wang, "The tasks with effects model for safe concurrency," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2013, pp. 239–250.

[33] K. Sen, A. Vardhan, G. Agha, and G. Rosu, "Efficient decentralized monitoring of safety in distributed systems," in *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society, 2004, pp. 418–427.

[34] M. M. Jaghoori, A. Movaghar, and M. Sirjani, "Modere: the model-checking engine of rebeca," in *Proceedings of the 2006 ACM symposium on Applied computing*. ACM, 2006, pp. 1810–1815.

[35] S. Lauterburg, M. Dotta, D. Marinov, and G. Agha, "A framework for state-space exploration of java-based actor programs," in *ASE '09: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 468–479.

[36] P. Bokor, J. Kinder, M. Serafini, and N. Suri, "Efficient model checking of fault-tolerant distributed protocols," in *Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, 2011, pp. 73–84.

[37] K. Sen and G. Agha, "Automated systematic testing of open distributed programs," in *Fundamental Approaches to Software Engineering*. Springer Berlin Heidelberg, 2006, pp. 339–356.

[38] S. Lauterburg, R. K. Karmani, D. Marinov, and G. Agha, "Evaluating ordering heuristics for dynamic partial-order reduction techniques," in *Fundamental Approaches to Software Engineering (FASE) with ETAPS*, 2010.

[39] V. Jagannath, M. Gligoric, S. Lauterburg, D. Marinov, and G. Agha, "Mutation operators for actor systems," in *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, 2010, pp. 157–162.

[40] S. Tasharofi, R. K. Karmani, S. Lauterburg, A. Legay, D. Marinov, and G. Agha, "Transdpor: a novel dynamic partial-order reduction technique for testing actor programs," in *Formal Techniques for Distributed Systems*. Springer, 2012, pp. 219–234.

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

[42] A. Fuggetta, G. P. Picco, and G. Vigna, "Understanding code mobility," *IEEE Transactions on Software Engineering*, vol. 24, no. 5, pp. 342–361, 1998.

[43] 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.

[44] S. Chakravorty, C. L. Mendes, and L. V. Kalé, "Proactive fault tolerance in mpi applications via task migration," in *High Performance Computing-HiPC 2006*. Springer, 2006, pp. 485–496.

[45] C. Engelmann, G. R. Vallee, T. Naughton, and S. L. Scott, "Proactive fault tolerance using preemptive migration," in *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*. IEEE, 2009, pp. 252–257.

[46] R. Panwar and G. Agha, "A methodology for programming scalable architectures," in *Journal of Parallel and Distributed Computing, vol. 22 pp 479-487*, 1994.

[47] A. Mettler, "Language and framework support for reviewably-secure software systems," Ph.D. dissertation, EECS Department, University of California, Berkeley, Dec 2012. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-244.html

[48] H. Sutter and J. R. Larus, "Software and the concurrency revolution," *ACM Queue*, vol. 3, no. 7, pp. 54–62, 2005.

[49] E. A. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, 2006.

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

[51] M. S. Miller, "Robust composition: Towards a unified approach to access control and concurrency control," Ph.D. dissertation, 2006.

[52] "The E language," *http://www.erights.org/elang*, 2000.

[53] C. Varela and G. Agha, "Programming dynamically reconfigurable open systems with SALSA," *ACM SIGPLAN Notices*, vol. 36, no. 12, pp. 20–34, 2001.

[54] E. A. Lee, "Overview of the ptolemy project," University of California, Berkeley, Tech. Rep. UCB/ERL M03/25, 2003.

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

[56] The Dart Team, "Dart programming language specification," *http://www.dartlang.org/docs/spec*, 2013.

[57] D. Kafura, "ACT++: building a concurrent C++ with actors," *Journal of Object-Oriented Programming*, vol. 3, no. 1, pp. 25–37, 1990.

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

[59] W. Kim, "Thal: An actor system for efficient and scalable concurrent computing," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1997.

[60] J. 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*. Cambridge University Press, 1989, p. 109.

[61] C. Tismer, "Stackless python," *http://www.stackless.com/*, 2004-09.

[62] J. Lee, "Parley," *http://osl.cs.uiuc.edu/parley/*, 2007.

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

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

[65] S. Bykov, A. Geller, G. Kliot, J. Larus, R. Pandya, and J. Thelin, "Orleans: A framework for cloud computing," Microsoft Research, Tech. Rep. MSR-TR-2010-159, 2010.

[66] M. Rettig, "Retlang," *http://code.google.com/p/retlang/*, 2007-09.

[67] P. Haller and M. Odersky, "Actors That Unify Threads and Events," in *9th International Conference on Coordination Models and Languages*, ser. Lecture Notes in Computer Science, vol. 4467. Springer, 2007.

[68] M. Rettig, "Jsasb," *https://jsasb.dev.java.net/*, 2008.

[69] M. Astley, *The Actor Foundry: A Java-based Actor Programming Environment*, Open Systems Laboratory, University of Illinois at Urbana-Champaign, 1998-99. [Online]. Available: http://osl.cs.uiuc.edu/foundry

[70] M.-W. Jang, *The Actor Architecture Manual*, Department of Computer Science, University of Illinois at Urbana-Champaign, March 2004. [Online]. Available: http://osl.cs.uiuc.edu/aa

[71] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. DHondt, and W. De Meuter, "Ambient-oriented programming in ambienttalk," *ECOOP 2006–Object-Oriented Programming*, pp. 230–254, 2006.

[72] T. Jansen, "Actors guild," *http://actorsguildframework.org/*, 2009.

[73] S. R. J.-P. Arcangeli, F. Migeon, "Javact : a java middleware for mobile adaptive agents," February 2008. [Online]. Available: http://www.irit.fr/PERSONNEL/SMAC/arcangeli/JavAct.html

[74] W. Zwicky, "Aj: A systems for buildings actors with java," M.S. thesis, University of Illinois at Urbana-Champaign, 2008.

[75] R. Young, "Jetlang," *http://code.google.com/p/jetlang/*, 2009.

[76] J. Schäfer and A. Poetzsch-Heffter, "JCoBox: Generalizing active objects to concurrent components," in *ECOOP 2010–Object-Oriented Programming*. Springer, 2010, pp. 275–299.

[77] W. D. Clinger, "Foundations of actor semantics," Ph.D. dissertation, Massachusetts Institute of Technology, 1981.

[78] P. Jetley and L. V. Kale, "Optimizations for message driven applications on multicore architectures," in *18th annual IEEE International Conference on High Performance Computing (HiPC 2011)*, December 2011.

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

[80] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi, "Language support for fast and reliable message-based communication in Singularity OS," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 4, pp. 177–190, 2006.

[81] M. Sridharan and S. J. Fink, "The complexity of Andersen's analysis in practice," in *SAS '09: Proceedings of the 16th International Symposium on Static Analysis*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 205–221.

[82] L. O. Andersen, "Program analysis and specialization for the C programming language," Ph.D. dissertation, University of Copenhagen, DIKU, 1994.

[83] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *POPL '95: Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, 1995, pp. 49–61.

[84] M. Sridharan, S. J. Fink, and R. Bodik, "Thin slicing," in *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2007, pp. 112–122.

[85] "WALA Static Analysis Library," http://wala.sourceforge.net/.

[86] Intel Corporation, "Single-chip Cloud Computer," *http://techresearch.intel.com/ProjectDetails.aspx?Id=1*.

[87] Tilera Corporation, "TILE-Gx Processor Family," *http://tilera.com/products/processors/TILE-Gx_Family*.

[88] R. Murphy, "On the effects of memory latency and bandwidth on supercomputer application performance," in *Proceedings of the 2007 IEEE 10th International Symposium on Workload Characterization*, ser. IISWC '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 35–43.

[89] Y. Mamyrin, "Domination," *http://domination.sourceforge.net/*, 2005-10.

[90] P. Haller, "On the integration of the actor model in mainstream technologies: the scala perspective," in *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions*. ACM, 2012, pp. 1–6.

[91] W. W. Hwu and P. P. Chang, "Achieving high instruction cache performance with an optimizing compiler," in *Proceedings of the 16th annual international symposium on Computer architecture*, ser. ISCA '89. New York, NY, USA: ACM, 1989. [Online]. Available: http://doi.acm.org/10.1145/74925.74953 pp. 242–251.

[92] S. Carr, K. S. McKinley, and C.-W. Tseng, "Compiler optimizations for improving data locality," in *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS-VI. New York, NY, USA: ACM, 1994. [Online]. Available: http://doi.acm.org/10.1145/195473.195557 pp. 252–262.

[93] J. C. Mogul and A. Borg, "The effect of context switches on cache performance," in *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS-IV. New York, NY, USA: ACM, 1991. [Online]. Available: http://doi.acm.org/10.1145/106972.106982 pp. 75–84.

[94] Q. Wang and C. A. Varela, "Impact of cloud computing virtualization strategies on workloads' performance," in *4th IEEE/ACM International Conference on Utility and Cloud Computing(UCC 2011)*, Melbourne, Australia, December 2011.