A Scala Library for Actor-Based Cooperative Scheduling

Anonymous Author(s)

Abstract

In this paper we introduce and evaluate a new actor-based library in Scala with a scalable implementation of cooperative scheduling of messages within an actor. This allows for multiple entry points for suspending and resuming execution while processing a message. By means of a typical benchmark we evaluate our proposed solution and compare it to other thread-based implementations of cooperative scheduling.

Keywords Actors, Scala, Cooperative Scheduling, Functional Programming, Object-Orientation, Java Library

ACM Reference Format:
Anonymous Author(s). 2017. A Scala Library for Actor-Based Cooperative Scheduling. In *Proceedings of ACM SIGPLAN Conference on Programming Languages, New York, NY, USA, January 01–03, 2017 (PL'17),* 10 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Actor-based models of computation in general assume a run-to-completion mode of execution of the messages. The Abstract Behavioral Specification (ABS) Language extends the Actor-based model with a high-level synchronization mechanism which allows actors to suspend the execution of the current message and schedule in a cooperative manner another queued message. This extension is a powerful means for the expression and analysis of fine-grained run-time dependencies between messages.

In ABS, asynchronous messages are modeled as methods of an actor. Sending an asynchronous message will schedule the execution of the corresponding method in the callee, returning immediately a future that, upon completion, will hold the result of the method execution, or in case of errors, the thrown exception. This "programming to interfaces" paradigm enables static type checking of message passing at compile time. This is contrast to the typical approach in Scala where messages are allowed to have any type and thus are only checked at run-time whether the receiver can handle them.

ABS further extends the Actor-based model with a high-level synchronization mechanism that allows actors to suspend in a cooperative manner the execution of the current message and start another queued message. The continuation of the suspended message, which is automatically put back in the message queue, may also be assigned an enabling condition. A typical use case is awaiting the completion of a future in the same method that has sent the corresponding message. This simplifies the programming logic by enabling the programmer to process the outcome of an asynchronous method call in the same place where it was sent (in a way similar to dealing with synchronous calls). Bear in mind that the continuation will be executed in the same actor thread thus (unlike the `onComplete` hooks in Scala that may run in a different thread) pose no threat to actor semantics.

The main challenge is the development of an efficient and scalable source-to-source implementation of cooperative scheduling. The basic feature of our proposed solution is the implementation of messages as described in [16] by means of lambda expressions (as provided by Java 8), i.e., the method call specified in a message is translated into a corresponding lambda expression which is passed and stored as an object of type Callable or Runnable (depending on whether it returns a result). We show how this basic feature is integrated in a general Scala/Java run-time system for ABS which also caters for synchronous calls that give rise to the suspension of the entire call stack generated from a message.

We describe how we implemented ABS actors and cooperative scheduling using Java thread pools and executor services. As elaborated in Section 6, we took an approach with negligible performance penalties regarding message passing and cooperative scheduling. We describe step by step how to support up to millions of messages, actors, and suspended continuations.

By means of a typical benchmark we evaluate our proposed solution and compare it to other thread-based implementations in Java or Erlang(a language that uses lightweight threads) of cooperative scheduling.

Related Work. In this paper we provide a Scala implementation of the ABS language as a full-fledged Actor-based programming language which features cooperative scheduling and fully supports the "programming to interfaces" paradigm. In contrast, Java libraries for programming actors like Akka [11] mainly provide pure asynchronous message passing which does not support the use of application programming interfaces (API) because, for example, in Akka/-Java a message is only typed as a Java Object, so there is no static typing of messages, nor are they part of the actor interface. A first approach for supporting programming to

A note.
PL'17, January 01–03, 2017, New York, NY, USA
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

interfaces was proposed in Takka library [13]. Distinguishing features of the ABS comprise of high-level constructs for cooperative scheduling which allow the application of formal methods, e.g., formal analysis of deadlock [9]. The Actor model in Scala [12] does provide a suspension mechanism, but its use is *not* recommended because it actually blocks the whole thread and causes degradation of performance. It is possible to register a *continuation* piece of code to run upon completion of a future, but that may run in a separate thread which however breaks the actor semantics and may cause race conditions inside the actor.

The Abstract Behavioral Specification (ABS) [15] Language has been developed for modeling parallel and distributed systems based on the actor model and provides extensive support for formal analysis like functional analysis [7], resource analysis [2] and deadlock analysis [8].

There exist various implementation attempts for cooperative scheduling in ABS. The approach followed in the Java backend of ABS [15, 17] and in the Erlang backend [10] is process-oriented in the sense that sending a message is implemented by the generation of a corresponding process. We will compare these backends with our solution in Java which allows to store messages (as objects of type Callable or Runnable) before executing them (data-oriented). Our solution builds on this work and extends it with a compiler with full support for cooperative scheduling in the context of both synchronous and asynchronous calls, functional programming and foreign language interface.

The focus of our paper is on an efficient implementation of cooperative scheduling in Java. Implementations in other languages allow for different approaches: In the Haskell backend for ABS [3], for example, the use of continuations allows to queue a message as a process and dequeue it for execution and the C implementation of cooperative scheduling for the Encore language [6] uses low-level (e.g., machine code) operations for storing and retrieving call stacks from memory.

In contrast, in the Actor-based modeling language Rebeca [18], for example, messages are queued and processed in a run-to-completion mode of execution. Cooperative scheduling is a powerful means for the expression and analysis of fine-grained run-time dependencies between messages.

The rest of this paper is organized as follows: In Section 2 we give an overview of the main features that the target actor-based model has. Section 4 presents how our solution transitions from a process-oriented approach to a data-oriented approach. Section 5 describes the implementation the runtime system for the actor-based model. In Section 6 we show the experimental evaluation of our solution followed by the conclusions drawn in Section 7.

2 Cooperative Scheduling in ABS

In this section we informally describe the main features of the flow of control underlying the semantics of cooperative scheduling in the ABS language. For a detailed description of the syntax of the ABS language we refer to [15].

In ABS a statement of the form `await f?` suspends the executing process which can only be rescheduled if the method invocation corresponding to the future `f` has computed the return value. Similarly, a statement of the form `await b`, where `b` denotes a boolean expression, suspends the executing process which can only be rescheduled if `b` evaluates to true. Finally a `suspend` statement invokes cooperative scheduling without a condition. In all cases the executing process can be suspended and another (enabled) message can be scheduled for execution. In contrast, a statement like `f.get` does not allow the scheduling of another process of the same actor. It thus blocks the entire actor. Listing 1 gives an example ABS model comparing cooperative scheduling to pure asynchronous communication as in pure actor model. It is important to observe the call by CooperativeMaster that is followed by an `await f?`. This allows the master to send computation requests to the workers and then wait for the results. This call suspends the current message execution of the master, but still allows it to process other messages. In contrast, PureMaster receives the results in another call (success) enforcing the programmer to divide the logic in two separate methods. In Section 6.2 we will discuss the performance of ABS actors in comparison with the Akka Actor model implementation.

Listing 1. Cooperative scheduling vs. pure async approach

```
1  class PureMaster (Int numWorkers) implements IMaster {
2   Unit sendWorkRoundRobin(...) {
3    IWorker worker = chooseWorker();
4    worker ! work(...);
5   }
6   Unit success(Result result) { /* process results */ }
7  }
8  class CooperativeMaster (Int numWorkers) implements IMaster {
9   Unit sendWorkRoundRobin(...) {
10   IWorker worker = chooseWorker();
11   Future<Result> fut = worker ! work(...); // start computation
12   await fut?;
13   // process restuls
14  }}
```

3 Scala API

Here we should briefly describe how to use the API in Scala, plus the two design patterns for await and get.

Since the core of ABS actors is written in Java it can be used in both Scala and Java, but due to the native functional approach in Scala, we describe the examples in Scala. Defining an actor is simply done by extending the LocalActor class. Thus by instantiating an instance of such actor classes, automatically the object will have the message queue and conceptually one thread of execution. The actor's reference

can then be used and passed in between objects and actors normally. It can then receive messages asynchronously and react to them. The Actor API provides three methods for asynchronous communication and concurrency control.

Sending Asynchronous messages The sendmethod has the following signature:

```
1 <V> ABSFuture<V> send(Callable<ABSFuture<V>> message);
```

Messages an actor can receive must return a Future. The method take a *message* parameter that specifies the task to be run asynchronously by the actor. An important observation to make here is that we cannot enforce the *message* to be a method provided by one of the interfaces that the actor extends, although it is recommended to only use methods exposed by these interfaces. What we can enforce though is that all messages that are sent to actor must return a result of type Futuresuch that we can use the design pattern described below. One can send messages to actors as shown in line xx of example yy.

Spawning an internal task The spawn method has the following signature:

```
1 <V> ABSFuture<V> spawn(Guard guard, Callable<ABSFuture<V>> message);
```

This method of the API together with the run-time system presented in Section 5 is used to emulate the *await* semantics and cooperative scheduling. The method takes as parameters an enabling condition (guard) and task to run (message) once the guard is satisfied. The method returns a Futurewhich is the return type of the message to be executed.

Design Pattern for ABS Await semantics The *spawn* method of the API together with the run-time system presented in Chapter **??** is used to emulate the *await* semantics and cooperative scheduling. A simple example is shown in listing 3.

```
1 //TODO add an example for await with spawn here
```

The pattern is very simple, it works as a normal call to spawnif the condition is not satisfied, while the continuation that follows await is ran directly if the condition is satisfied straight away.

Blocking an Actor The getSpawn method has the following signature:

```
1 <T,V> ABSFuture<T> getSpawn(ABSFuture<V> f, CallableGet<T, V> message);
2 <T,V> ABSFuture<T> getSpawn(ABSFuture<V> f, CallableGet<T, V> message,
3                             int priority, boolean strict);
```

This method has a similar behaviour spawnbut may only be used together with a future Guard (ABSFuture) and also propagates the value of this future (inside CallableGet) to the *spawned* task once it is ready. This method must be used to read a future as reading it directly will cause the blocking of the whole thread if the value is not available. Future references can be passed between actors. Reading the future value using getSpawn, allows the actor to run other tasks to be run in case the future is not completed without forcing the actor to block. Once the future is ready, the *message* parameter will contain the value of this future together with the task that is to be run. This limits the numbers of threads that live in the system, but are blocked, without breaking the actor encapsulation. If we want messages and continuations to execute in a particular order (such as preserving a synchronous call stack) we can set a higher or lower *priority* for a message. This is shown in an example in Listing**??**. Furthermore, there may the case when we would like the actor to block and therefore the setting the boolean parameter *strict* will determine if the actor will block.

Design Pattern for ABS Get semantics The example in Listing **??** shows the pattern to write the ABS Get semantics using getSpawn. The pattern is the same as await, only it uses getSpawnwith the s*trict* flag set to true to enforce blocking behaviour.

```
1 //TODO add an example for get with getSpawn here
```

Design Pattern for preserving a synchronous call stack The example in Listing **??** shows the pattern to execute a chain of synchronous calls correctly using getSpawn. To do this, we call the method synchronously as normal, but if its returned Futureis not ready, we read it using getSpawnbut give the continuation *message* a higher priority such that it will execute first when the future is ready.

```
1 //TODO add an example for get with synccalls here
```

4 Implementation Schemes in Java

In this section we present how the Scala language is used as a backend for ABS and investigate the evolution of the scheme used to implement cooperative scheduling in the Java Runtime environment. We start from a very trivial approach to using several libraries and features that the latest version of Java SDK provides. The first feature we will use is the Executor interface that facilitates thread programming. The objects that implement this interface provide a queue of tasks and an efficient way of running those tasks on multiple threads. Throughout this section we will use the terms executor and thread pool interchangeably to refer to this feature. The second notion discussed in our solution was introduced starting with the Java 8 version, and it is the construct for a lambda expression [1]. This allows us to model a method call as a Runnable or Callable and we will refer to this model as a **task**. We will look at each solution in terms of four Actor features:

- Creation of the actor itself;
- Generating asynchronous calls and message passing;
- Releasing control or suspension of a call upon await;
- Saving the call stack of a synchronous call upon releasing control.

[1] https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

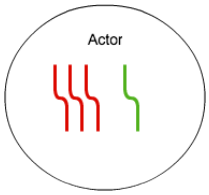

Figure 1. Basic process-oriented approach

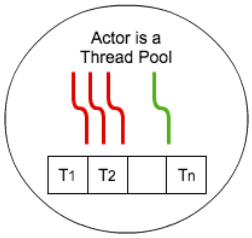

Figure 2. Actor-as-executor approach

4.1 Every asynchronous call is a thread and each actor has a lock

The trivial straightforward approach for implementing cooperative scheduling in the library is for an asynchronous call to generate a new native thread with its own stack and context. We would then model each actor as an object with a lock for which threads compete. The disadvantage here is that this lock-per-actor must be checked by every message handler upon start, and freed upon completion. Whenever an await statement occurs the thread would be suspended by the JVM's normal behavior. When the release condition is enabled, a suspended thread would become available and in turn compete with the other threads in order to execute on the actor. The main drawback of this approach is the large number of threads that are created, which restricts any application from having more method calls than the number of live native threads. Figure 1 provides an illustration of this base *process-oriented* approach. Here the idea is that the threads in the circle belong to one actor, thus sharing a lock(the green thread holds the lock, while the red ones are waiting to acquire it). The four Actor features are implemented as follows:

- The actor is initialized as an object with a lock;
- Asynchronous calls are created as new threads that compete for the lock
- Suspension of a call suspends the thread;
- The call stack of a synchronous call is saved in the suspended thread.

4.2 Every actor is a thread pool.

To reduce the number of live threads in an application, we can model each invocation as a task using lambda expressions and organize each actor as a thread pool. This gives the actor an implicit queue to which tasks are submitted. We obtain a small reduction in the number of threads corresponding to the number of tasks that have been submitted, but not started. Once they are started the threads still have to compete for the actor's lock in order to execute, but the number of live threads can be restricted to the number of threads allowed by each Thread Pool, while the rest of the invocations remain in the thread pool queue as tasks. This approach is illustrated in Figure 2. The implementation of the four Actor features is as follows:

- The actor is initialized as an object with a lock and a thread pool;
- Asynchronous calls are created as lambda expressions and assigned new threads that compete for the lock once they start;
- Suspension of a call suspends the thread;
- The call stack of a synchronous call is still saved in the suspended thread.

4.3 Every system has a thread pool.

In the previous two approaches we modeled the concept of an actor being restricted to one task at a time by introducing a lock on which threads compete. However as all invocations are modeled as tasks that don't need a thread before they start, there is no point in starting more than one task only to have it stuck on the actor's lock. Therefore we introduce *one thread pool per system* or *the system's executor*, and instead of submitting the messages directly to the thread pool, we add an indirection by storing them into an explicit queue first and introduce as a second phase the submission to the thread pool. We assign a separate task for each actor to iterate through its associated queue and submit the next available message to the system thread pool. We will refer to this task from now on as the **Main Task** of an actor. This removes the requirement to store every message handler as a thread, after it starts and attempts to acquire a lock, saving a task as data in the heap instead. However it comes at the cost of having to manage message queues manually as we need an explicit queue for all the messages that have not yet been submitted (to the thread pool).

When cooperative scheduling occurs, the executing task of an actor will be suspended and therefore still live in the system as a thread so the application's live threads will be equal to the number of "await" statements in the program. The system's executor can dynamically adjust the number of available threads to run subsequent available tasks, but the application will then still be limited by the maximum number of suspended threads that can exist in the main memory. Furthermore, we still require a lock to ensure that, upon release, the suspended thread will compete with the Main Task associated to the actor from which it originated. This design is presented in Figure 3 and it is important to observe the migration of messages into memory and that the red threads are only tasks that have been suspended by an

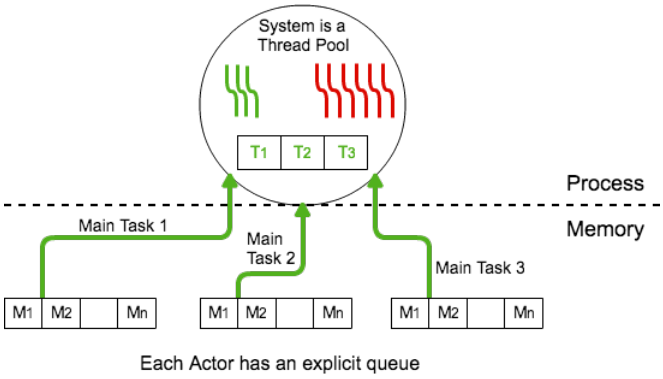

Figure 3. System as a thread pool

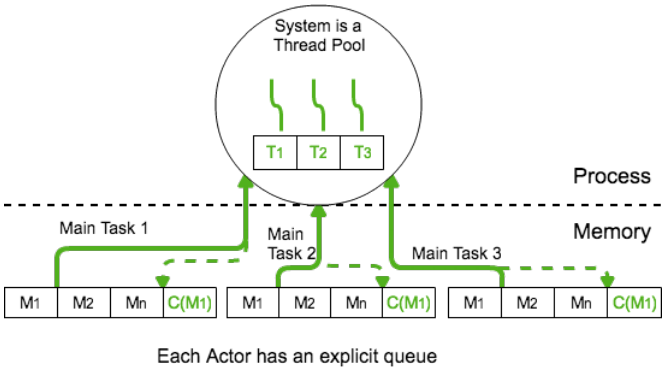

Figure 4. Full data-oriented approach

"await". We observe the following changes in the four actor features:

- The actor is initialized as an object with a lock for threads that will be suspended by `await`, an explicit queue and an implementation of the **Main Task**;
- Asynchronous calls are created as new tasks that will be run by the **Main Task** and will compete for the lock only with suspended tasks;
- Suspension of a call restarts the **Main Task** to iterate the actor's queue and parks the current task as a thread;
- The call stack of a synchronous call is still saved in the suspended thread.

4.4 Fully asynchronous environment.

In the absence of synchronous calls, to eliminate the problem of having live threads when cooperative scheduling occurs, we can also use lambda expressions to turn the continuation of an await statement (which is determined by its lexical scope) into an internal method call and pass its current state as parameters to this method. Essentially what we do is allocate memory for the continuation on the heap, instead of holding a stack and context for it. We will benchmark this trade-off between the memory footprint of a native thread and the customized encoding of a thread state in memory. As there are no more suspended threads in the system to compete with the actor's Main Task, we can eliminate the lock per actor. The four features of the Actor are as follows:

- The actor is initialized as an object with an explicit queue and an implementation of the of the **Main Task**, but without a lock;
- Asynchronous calls are created as new tasks that will be run by the **Main Task** and will compete for the lock only with suspended tasks;
- Suspension of a call first saves the continuation as a lambda expression guarded by the release condition, and the **Main Task** continues to iterate through the queue;
- The process of saving the call stack of a synchronous call is detailed in the next paragraph.

Synchronous calls context. In the presence of synchronous calls, the problem that remains is how to save this call stack without degrading performance? To do this we can try to alter the bytecode to resume execution at runtime from a particular point, but we want our approach to be independent of the runtime and be extensible to other programming languages. Therefore we try to approach the problem differently; if we can turn a continuation, that does not originate from a stack of calls, into a task, is it possible to extend this to synchronous calls as well? We know that this issue arises when methods that contain an "await" statement are called synchronously like the example in Listing 1. When the `await f?` is encountered, the continuation consists of both the lexical scope of the release statement that is followed by the a complete synchronous call chain that has been generated(i.e. $C(M_1)$ in Figure 4). At compile time, when translating source-to-source from ABS to Scala, we can identify all of these occurrences from the ABS code. We can mark them and transform them into asynchronous calls followed by an "await" statement on the future generated from the call. Now we can use the same approach for translating these continuations into tasks using lambda expressions and thus eliminating any suspended threads in the system. The final model of our solution is presented in Figure 4.

A fully asynchronous environment means to change a synchronous call `x = this.m();` into an asynchronous call plus an await like `f = this!m(); x = await f?;`. There are two problems here. First, we still need to make sure that "m" must be the exact next method that will run. Second, when "m" finishes, the actor scheduler must immediately schedule the method that is waiting for its result. Both can be implemented by changing the non-deterministic behavior of the Main Task. There are two constrains that we have to impose to preserve the chain of synchronous calls. First, we set a flag "isSyncCall" in the task that is calling "m" and when that flag is true, the Main Task will immediately execute

the message that is to be enqueued instead of taking an arbitrary one from the queue. Second, assuming that there is no "message overtaking", the messages that are part of a synchronous call chain arrive in the Actor's queue in the FIFO order. We can label each call chain with a number "syncContext" and all messages part of that call chain will have this number. When the Main Task completes a message labeled with a particular number it will take from the queue the next available message with the same label and execute it. Finally, when the last message with that label is complete, then the call chain is complete and the next message will be selected arbitrarily.

5 Implementation

The implementation of the full data-oriented scheme is done through a library which contains a set of classes and interfaces that have a direct mapping to the ABS language concepts described in Section 2. The library provides an implementation of cooperative scheduling behavior, the suspension and release mechanisms while respecting the logic of synchronous calls. The library provides the solution and the optimizations discussed in Section 4. The library can be obtained as a maven project and is available online [2] with the compiler [3].

5.1 Actor Implementation

The library offers an interface **Actor** containing several methods for implementing the behavior of synchronous (*sendSync*), asynchronous (*send*) method calls and await statements (*await*) from ABS. The *await* statement provided by the library is implemented in such a way that it supports an explicit continuation that can be either generated by the compiler from ABS or used by the programmer directly in Scala using the library. The library currently supports an implementation of this interface called **LocalActor** for actors running on the same machine. An important advantage of having a task assigned to each actor and a manually processed queue is that we can start and stop the task depending on the queue state. To avoid keeping the Main Task alive, we make it part of the functionality of sending a message or completing a future to notify the Main Task. This is done by any other actor who sends an invocation to an empty queue and subsequently the Main Task stops when there are no more enabled messages in the queue.

Inside the **LocalActor** class there is a *messageQueue* defined which holds all the invocations (synchronous or asynchronous) that have been submitted to the actor as tasks. The implementation defines an inner class *MainTask* which is a Java Runnable that corresponds to the task responsible for taking messages from the queue and running them. When the queue is empty, we do not want to make this task busy-wait until a message arrives, and so, upon insertion of a new message into the queue, there is a check whether such a task exists already. This gives rise to some subtle synchronization points. For every actor, we keep a local atomic boolean flag *mainTaskIsRunning*. A first approach looks like in Listing 2:

Listing 2. Basic Synchronization Approach

```
// inside the task
class MainTask implements Runnable{
  public void run() {{
     // iterate through queue and take one message and run it
    mainTaskIsRunning.set(false);
}}

// when inserting a new message
messageQueue.add(m);
if (!mainTaskIsRunning.compareAndSet(false, true)) {
  DeploymentComponent.submit(new MainTask())
}
```

Listing 3. Complete Synchronization Approach

```
private boolean takeOrDie() {
 synchronized (mainTaskIsRunning) {

   // iterate through queue and take one ready message
   // if it exists set the next message for the main task and then
   return true;

   //if the queue if empty or no message is able to run
   mainTaskIsRunning.set(false);
   return false;
}}
private boolean notRunningThenStart() {
 synchronized (mainTaskIsRunning) {
   return mainTaskIsRunning.compareAndSet(false, true);
}}

// inside the task
class MainTask implements Runnable{

  public void run() {
  while (takeOrDie())
    // proceed to take the next message message and run it
}}

// when inserting a new message
messageQueue.add(m);
if (notRunningThenStart()) {
  DeploymentComponent.submit(new MainTask());
}
```

The problem with the code in Listing 2 is that the check of the queue for emptiness and setting the flag to "false" is not atomic, and in between these two statements, a new message may be inserted into the queue without spawning a new Main Task. To remedy this, in Listing 3, we introduce a new method that can check the queue for emptiness and set the flag to "false" in a critical section, for example inside a "synchronized" block using the *messageQueue* or *mainTaskIsRunning* as the lock. Additionally, either the insertion into

[2] https://github.com/vlad-serbanescu/abs-api-cwi.git (Local Actor branch)
[3] https://github.com/CrispOSS/jabsc (Scala Writer branch)

the *messageQueue* or *compareAndSet* of *mainTaskIsRunning* should also use the same lock. Another problem with this approach above is that, when there are more actors than available threads, there is no fairness policy when assigning a thread to an actor. To remedy this, we could change the while loop to an if statement like in Listing 4:

Listing 4. Fairness Between Actors

```
class MainTask implements Runnable{
  public void run() {
    if (!takeOrDie())
      return;
    // proceed to take the next message message and run it
    DeploymentComponent.submit(this);
  }
}
```

5.2 Actor System Implementation

The system executor uses a *newCachedThreadPool* singleton *ExecutorService* that automatically creates and allocates threads to handle the current load of the system. An actor may start a new MainTask (for processing messages in the queue) by calling the static method submit(**new** MainTask()). Inside the class there is also support to safely call shutdown() when all the actors in the application have completed execution. Further in this section we explain some details and optimizations implemented in management of actors and their corresponding tasks.

Eliminating busy waiting. Cooperative scheduling through release points may suspend the current message run by the actor based on either a particular inner state or future requiring completion on a different actor. We discussed how the Main Task can start or stop based on release points, but it is important to observe how exactly does an actor verify that a release point has completed. Having a task continuously iterate through all suspended messages (busy-waiting) is inefficient. We can separate the messages that are guarded by an incomplete future until that future completes such that the Main Task does not need to check them. Subsequently they can be put in the *messageQueue* once the future guard completes. The same procedure cannot be applied for a message whose enabling condition depends on the actor state, as its state can change again, so it always has to be verified before a message is scheduled. We assign this verification to the Main Task that iterates through the queue, and simply stop the task if no message is available after one iteration. If the task does stop, it means that the actor is in a state in which it is unable to execute any of its suspended messages. Therefore, it requires another actor to either send a new invocation that will change its state or the system to send a notification about a future that may release some of its messages and re-enable the Main Task. Therefore we maintain a global hash table, mapping every future to the set of actors that are awaiting on its completion. Actors that complete a certain future can call the static method releaseAll(f) (provided at the system level) to notify actors that contain messages suspended on that particular future.

Using JVM Garbage Collection Using the approach explained so far in this section, the only extra references we need for the actors are the ones inside the global hash-table required for the notification mechanism for futures. Once the future is completed and notifications are sent, the key is deleted and the actor references become unreachable. Therefore we can leave the entire garbage collection process to the Java Runtime Environment as no other bookkeeping mechanisms are required. This way we do not keep a registry of the actors like the `context` in Scala and Akka.

6 Benchmarking the Implementation Schemes

In this section we want to evaluate Scala library in terms of the impact that the cooperative scheduling feature has on performance. We will use a benchmark that involves heavy cooperative scheduling to compare the data-oriented approach to the process-oriented approach that ABS when translated to Java. Furthermore we will use a benchmark tailored towards CPU-intensive to evaluate the potential of this Scala library to support ABS as a software-development language. We will compare our solution with both the Erlang backend of ABS which benefits from lightweight threads as well as the Akka Actor library for Scala.

6.1 Cooperative Scheduling Benchmark

The main problem that we encountered when implementing cooperative scheduling was saving the context of an execution and resuming from that context. To do this in Java using threads and context switches heavily limits the application to the number of native threads that can be created. To measure the improvement provided by our Scala library features using the simple example that creates a recursive stack of synchronous calls. Te compiled ABS model in Scala is translated as Listing 5.

Listing 5. Benchmark Example

```
trait Ainterface extends Actor with Ordered[Actor] {
  def recursive_m( i : Int,  id : Int): Int;
}

class A() extends LocalActor with Ainterface {

  var result : Int =  0;
  def recursive_m( i : Int,  id : Int): Int= {
    if ((i > 0)) {
      var msg : Callable[Int] = () => this.recursive_m((i − 1), id);
      var future_k : ABSFutureTask[Int]
          = this.sendSync ((future_k_par: ABSFutureTask[Int])=>
            this.Arecursive_mAwait0(i, id, future_k_par), msg);
      var k : Int   = future_k.get();
    }
    else {
      var msg : Callable[Int] = () => this.compute();
      var f : ABSFutureTask[Int] = this.send (msg);
      await(()=>this.Arecursive_mAwait1(f, i, id), Guard.convert(f));
```

```
    }
    return 1;
  }

  def compute(): Int= {
    result = (result + 1);
    return result;
  }

  def Arecursive_mAwait0( i : Int,  id : Int,  future_k : ABSFutureTask[Int]): Int= {
    var k : Int = future_k.get();
    return 1;
  }

  def Arecursive_mAwait1( f : ABSFutureTask[Int],  i : Int,  id : Int): Int= {
    return 1;
  }
}

object Main extends LocalActor {

  def main( args : Array[String]): Unit= {
    var i : Int =  0;
    var master : Ainterface =  new A();
    var futures : List[ABSFutureTask[Int]] =  Nil();

    while ((i < 2000)) {
      var msg : Callable[Int] = () => master.recursive_m(5, i);
      var f : ABSFutureTask[Int] = master.send (msg);
      futures = Cons(f, futures);
      i = (i + 1);
    }

    while ((!Objects.equals(futures, Nil[ABSFutureTask[Int]]()))) {
      var f1 : ABSFutureTask[Int] =  head(futures);
      futures = tail(futures);
      var r : Int = f1.get();
}}}
```

Mahdi. If we need to cut, we can put the code on github and link to it here.

The model creates an Actor of type "A" and sends a large number of messages to it to execute a method recursive_m(5,i. This method creates a call chain of size 5 before sending an asynchronous message to itself to execute a basic method compute() and awaits on its result. Although simple, this example allows us to benchmark the pure overhead that arises from having a runtime system with cooperative scheduling support, both in a data-oriented approach and a process-oriented approach. The results are shown in Figure 5. The performance figures presented are for one actor that is running 500-2500 method invocations. It is important to observe that each invocation generates 2 messages in the actor's queue, so as the number of calls increases the number of messages doubles. The figures show that the trade-off for storing continuations into memory instead of saving them in native threads removes limitations on the application and significantly reduces overhead.

6.2 NQueens Benchmark

Mahdi. I propose to have two ABS models, one with and one without cooperative scheduling. And compare them at the same time to Akka. I will try to make them next tuesday.

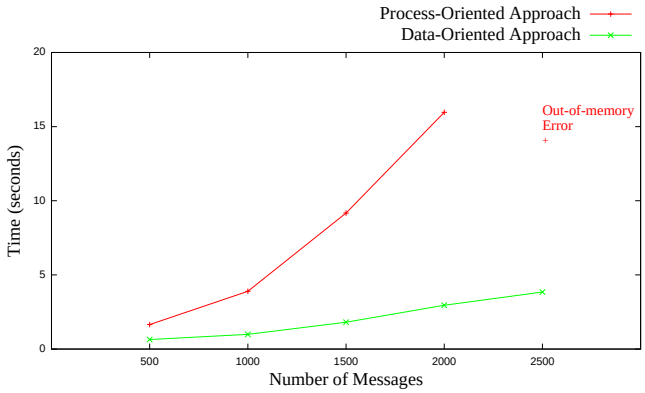

Figure 5. Performance figures for pure Cooperative Scheduling

While our solution is catered towards a widely-used software development language, we would like to also compare with other languages that implement ABS language concepts efficiently using threads and without these limitations. In Section 5 we listed several optimizations that were inferred from our implementation solution. What we want to do is compare this optimized solution to an ABS backend implemented in Erlang that uses the same process-oriented approach but does not suffer from any limitation of native threads. We want to observe if our data-oriented approach can be comparable to Erlang's lightweight threads. For this comparison, we use the Savina benchmark for programming with actors [14]. We chose the problem in Listing1 of arranging N queens on a NxN chessboard as it provides a master-slave model that relies heavily on the cooperative scheduling properties of the master. The benchmark divides the task of finding all the valid solutions to the N queens problem to a fixed number of workers that at each step have to find an intermediary solution of placing a queen K on the board before relaying the message back to the master which then assigns the next job of placing queen $K + 1$ to another worker. As the search space becomes smaller, w impose a threshold where the worker has to find the complete solution up to N queens before sending a message back to the master.

We ran the benchmark with a board size varying from 6 to 12 with a fixed number of 4 workers on a core i5 machine which supports hyper-threading. The results are shown in Figure 5. It is important to observe that as the board size increases, the number of solutions grows from 40 to 2680. The result in Figure 6 show that our approach is better once the board size reaches 12 and the number of solutions to be found is 14200. This result also strengthens ABS purpose to provide a programming language for real applications.

Furthermore, we would like to observe the performance impact that support for cooperative scheduling adds to an

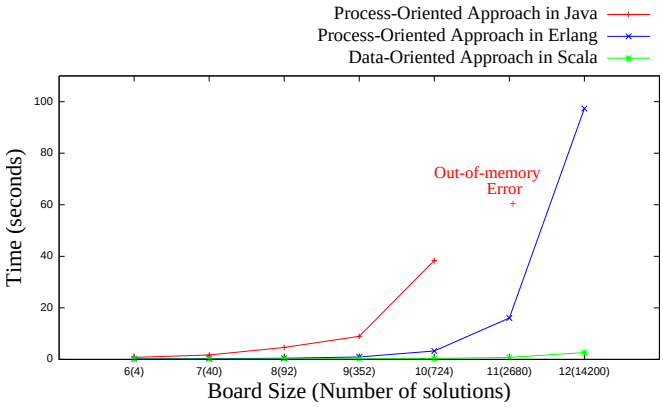

Figure 6. Performance figures for the NQueens problem in 3 ABS backends

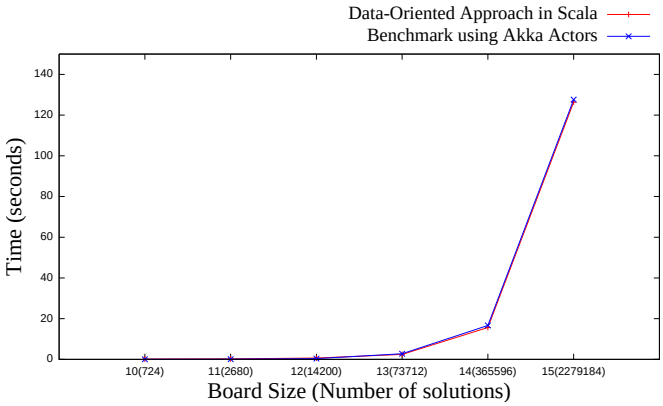

Figure 7. Performance figures for the NQueens problem implementations in Akka and Scala Library

application. This feature is a powerful programming abstraction, but we would like to see the cost of developing a benchmark in comparison to an existing actor implementation like the Akka Actor library. It is first important to observe that the ABS model for this benchmark is 154 lines of code, while the benchmark written directly in Scala using the Akka library is 351 lines of code. This reduction in the code is due cooperative scheduling and also ABS not requiring a context setup for actors and explicit build-up of messages and subsequent pattern matching on them. We compared the implementation offered by the Savina benchmark with our own for a board size from 10 to 15 on the same machine as before. The results in Figure7 show that the cooperative scheduling feature does not have any added overhead compared to the Akka Actor library while offering a seamless programming experience.

7 Conclusions

In this paper we proposed a Scala library for efficiently implementing the cooperative scheduling behavior of ABS which provides a powerful programming abstraction. To provide support for the functional data types required by ABS we embedded the Java run-time system into the Scala programming language. We provided a benchmark tailored towards cooperative scheduling that show significant improvements of saving continuations including the call stack as data in memory instead of using a process-oriented approach by means native Java threads. We used a second benchmark for a CPU-intensive application to show that this feature does have a negative impact on performance compared to the Akka Actor library.

Furthermore having a portable JVM library gives us a basis for industrial adoption of the ABS language and provides ABS as a powerful extension of Scala with support for formal verification, resource analysis and deadlock detection as well as extensions that support real-time programming [5], in a software development context. In particular the ABS extension of Scala provide can be integrated with the distributed ABS implementation that exists in the Haskell backend [4] to provide a distributed programming model for Actor-based applications.

In future work we plan to extend the library to statically type-check the message submitted via the *send* method in order to prevent the user from running unwanted code on the actors. static type checking with respect to actor interfaces

syntactic sugar

await the proper way

distributed actors

References

[1] Gul A Agha. 1985. *Actors: A model of concurrent computation in distributed systems.* Technical Report. MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB.

[2] Elvira Albert, Puri Arenas, Antonio Flores-Montoya, Samir Genaim, Miguel Gómez-Zamalloa, Enrique Martin-Martin, Germán Puebla, and Guillermo Román-Díez. 2014. SACO: Static Analyzer for Concurrent Objects.. In *TACAS*, Vol. 14. 562–567.

[3] Elvira Albert, Nikolaos Bezirgiannis, Frank de Boer, and Enrique Martin-Martin. 2016. A Formal, Resource Consumption-Preserving Translation of Actors to Haskell. *arXiv preprint arXiv:1608.02896* (2016).

[4] Nikolaos Bezirgiannis and Frank de Boer. 2016. ABS: a high-level modeling language for cloud-aware programming. In *International Conference on Current Trends in Theory and Practice of Informatics*. Springer, 433–444.

[5] Joakim Bjørk, Frank S de Boer, Einar Broch Johnsen, Rudolf Schlatte, and S Lizeth Tapia Tarifa. 2013. User-defined schedulers for real-time concurrent objects. *Innovations in Systems and Software Engineering* 9, 1 (2013), 29–43.

[6] Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Einar Broch Johnsen, Ka I Pun, S Lizeth Tapia Tarifa, Tobias Wrigstad, and Albert Mingkun Yang. 2015. Parallel objects for multicores: A glimpse at the parallel language Encore. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*. Springer, 1–56.

[7] Crystal Chang Din, Richard Bubel, and Reiner Hähnle. 2015. KeY-ABS: a deductive verification tool for the concurrent modelling language ABS. In *International Conference on Automated Deduction*. Springer,

517–526.

[8] Antonio E Flores-Montoya, Elvira Albert, and Samir Genaim. 2013. May-happen-in-parallel based deadlock analysis for concurrent objects. In *Formal Techniques for Distributed Systems*. Springer, 273–288.

[9] Elena Giachino, Cosimo Laneve, and Michael Lienhardt. 2016. A framework for deadlock detection in core ABS. *Software & Systems Modeling* 15, 4 (2016), 1013–1048.

[10] Georg Göri, Einar Broch Johnsen, Rudolf Schlatte, and Volker Stolz. 2014. Erlang-style error recovery for concurrent objects with cooperative scheduling. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 5–21.

[11] Munish Gupta. 2012. *Akka essentials*. Packt Publishing Ltd.

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

[13] Jiansen He, Philip Wadler, and Philip Trinder. 2014. Typecasting actors: from Akka to TAkka. In *Proceedings of the Fifth Annual Scala Workshop*. ACM, 23–33.

[14] Shams M Imam and Vivek Sarkar. 2014. Savina-an actor benchmark suite: Enabling empirical evaluation of actor libraries. In *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control*. ACM, 67–80.

[15] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. 2012. ABS: A core language for abstract behavioral specification. In *Formal Methods for Components and Objects*. Springer, 142–164.

[16] Behrooz Nobakht and Frank S de Boer. 2014. Programming with actors in Java 8. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 37–53.

[17] Jan Schäfer. 2011. *A programming model and language for concurrent and distributed object-oriented systems*. University of Kaiserslautern.

[18] M Sirjani and A Movaghar. 2001. An actor-based model for formal modelling of reactive systems: Rebeca. *Technical Report* (2001).