A Scala Library for Actor-Based Cooperative Scheduling[*]

Subtitle[†]

Anonymous Author(s)

Abstract

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 runtime dependencies between messages. In this paper we introduce and evaluate a new run-time system in Java for the use of ABS as a full-fledged programming language. The main challenge is the development of an efficient and scalable implementation of cooperative scheduling. By means of a typical benchmark we evaluate our proposed solution and compare it to other thread-based implementations of cooperative scheduling . . . .

CCS Concepts • **Software and its engineering → General programming languages**; • **Social and professional topics →** *History of programming languages*;

Keywords Actors, Scala, Cooperative Scheduling, Functional Programming, Object-Orientation, Run-Time

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),* 9 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

The Actor-based model of computation [4] is particularly tailored to the description of distributed systems. Actors represent processes that execute in parallel and interact via asynchronous communication of messages. The Abstract Behavioral Specification (ABS) [16] Language has been developed for modeling distributed systems based on the actor model and provides extensive support for formal analysis like functional analysis [6], resource analysis [18] and deadlock analysis [20]. In this paper, we bring the concepts and advantages of ABS to programming level by implementing a library that can be directly used from Scala and Java.

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. More details on syntax and behavior of ABS is presented in Section **??**.

The main challenge is the development of an efficient and scalable source-to-source implementation of cooperative scheduling which, in the presence of synchronous calls, gives rise to the suspension of the entire call stack generated from a message. The continuation of the suspended message including the stack frame needs to be saved in order for execution to resume properly once the message is rescheduled. The basic feature of our proposed solution is the implementation of messages 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 [30] which is passed and stored as an object of type

[*]Title note
[†]Subtitle note

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

Callable or Runnable (depending on whether it returns a result).

In this paper, we describe how we implemented ABS actors and cooperative scheduling using Java thread pools and executor services. As elaborated in Section **??**, we took an approach with negligible performace 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. Our main motivation is a Java implementation of the ABS language as a full-fledged Actor-based programming language which fully supports the "programming to interfaces" paradigm. In contrast, Java libraries for programming actors like Akka [1] 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. Furthermore, 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 [11]. The Actor model in Scala [2] 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.

There exist various implementation attempts for cooperative scheduling in ABS. The approach followed in the Java backend of ABS [5, 16] and in the Erlang backend [7] 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).

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 [8], 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 [9] 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 [3], 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 3 presents how our solution transitions from a process-oriented approach to a data-oriented approach. Section **??** details how continuations are transformed into data and how this data is scheduled on the actors. Section 4 describes the main challenges of implementing the run-time system for the actor-based model. In Section 5 we show the experimental evaluation of our solution followed by the conclusions drawn in Section 6.

2 ABS Language Concepts

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 [16].

2.1 Method Invocations

Abstracting from the syntax of the actual parameters, in ABS an asynchronous invocation of a method m of an actor a is described by a statement Future f=a!m(), where f is a future used to store the return value (asuming that m contains a return statement). In our backend this invocation will be stored in a queue of the called actor a. Futures can be passed around as references and provide a get operation described by f.get which results in the value returned and blocks the current process if the corresponding method invocation has not yet computed the return value.

Again abstracting from the syntax of the actual parameters, ABS additionally features synchronous method calls described in the standard manner by statements of the form x=a.m() (also assuming here that the method m returns a value). In case the called actor a belongs to a different COG (i.e., Concurrent Object Group which shares one thread of control, see [16]) than the caller the semantics of such a synchronous call can be translated by Future f=a!m(); x= f.get, for some future f (of the appropriate type). In case the called actor a and the caller belong to the same COG such a synchronous call can be translated by Future f=a!m( ); x= f.get;resume, where the auxiliary statement resume (as introduced in [25]) is implememted in our Java backend by a specific scheduling policy which preserves the synchronous call stack. This will be described in more detail in Section 4.

2.2 Await Construct

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. In both cases the executing process is suspended and another (enabled) process (belonging to the same actor, or more generally, to the same COG) can be scheduled for execution. In contrast, a statement like f.get does not allow the scheduling of another process of the same actor (COG). It thus blocks the entire actor (COG). Listing 1 sketches a pattern of method definitions which gives rise to a suspended stack of synchronous calls; assume an asynchronous call to m1; this may generate a synchronous call to m2 and subsequently this invocation may suspend on the future f. The main challenge addressed in this paper is to have an efficient implementation of such a suspended stack of synchronous calls.

Listing 1. ABS Example

```
m1( ){
  ...;
  this.m2( );
  ... //continuation of m1 <=>C(m_1)
}

m2( ){
  ...
  await f?
  .... //continuation of m2 <=>C(m_2)
}
```

3 Cooperative Scheduling Implementation Schemes

In this section we present how the Java 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. from a very simple approach to using several libraries and features that the latest version of Java provides. The first feature we will use is the Executor interface [26] that facilitates parallel programming in Java. 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 Java 8 and it is the construct for a lambda expression [30]. 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;
- Generation of asynchronous calls and message passing;
- Releasing control or suspension of a call when encountering an await;
- Saving the call stack of a synchronous call upon releasing control.

3.1 Every asynchronous call is a thread and each actor has a lock.

The trivial straightforward approach for implementing cooperative scheduling in Java 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 simply suspends the thread;
- The call stack of a synchronous call is saved in the suspended thread

3.2 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 new threads that compete for the lock;
- Suspension of a call simply suspends the thread;
- The call stack of a synchronous call is saved in the suspended thread;

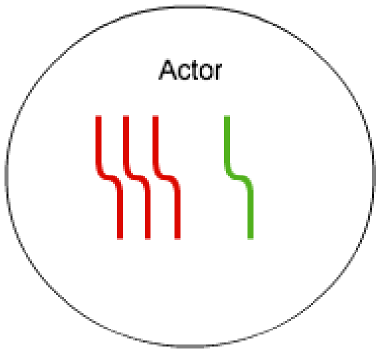

Figure 1. Basic process-oriented approach

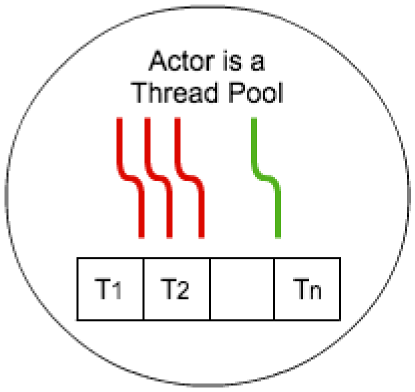

Figure 2. Actor-as-executor approach

3.3 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.

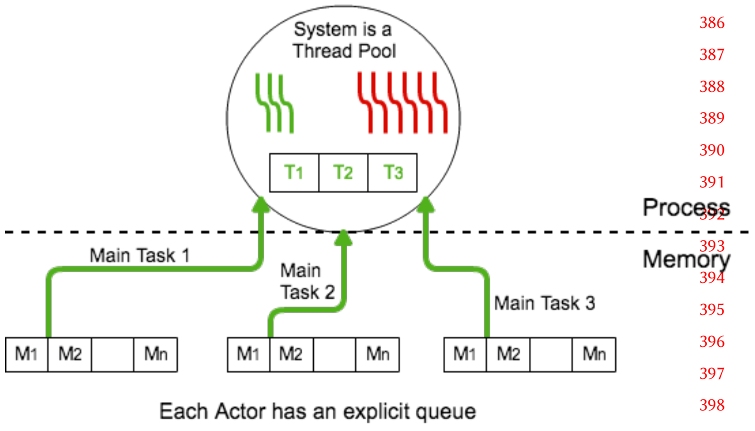

Figure 3. System as a thread pool

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

- The actor is initialized as an object with a lock, 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.

3.4 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. Finally, let's consider the example in Listing 1. In this case 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. $C(m_2); C(m_1)$). 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 only 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, but at compile time we can easily identify all of these occurrences from the ABS code. We can simply mark them at compile time 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. The details of how to construct these continuations and schedule them are presented in Section **??**.

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.

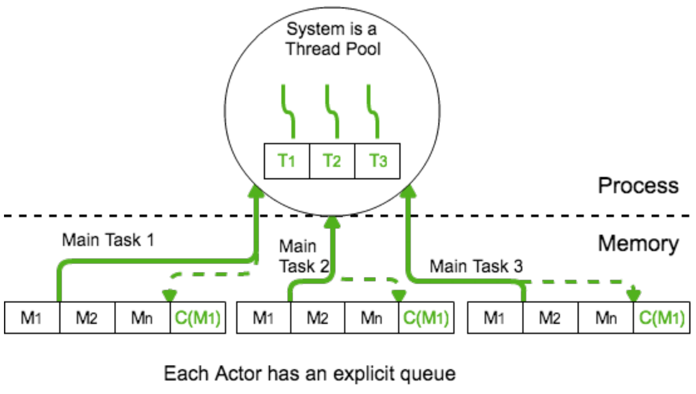

Figure 4. Full data-oriented approach

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.

4 Implementation

The implementation of the full data-oriented scheme is done through a Java 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 3 using the preprocessed continuations generated as detailed in Section **??**. The library can be obtained as a maven project and is available online [29].

4.1 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 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

Listing 2. Basic Synchronization for the Demand-Driven 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())
}
```

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 3. Complete Synchronization for the Demand-Driven 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 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);
 }
}
```

4.2 Deployment Component

In ABS there are two scenarios that re-enable tasks; an internal state of the actor is valid(i.e. a boolean condition that guard a continuation evaluates to true) or a future that guards a continuation is completed by another actor. These later release points require the system to have a notification mechanism for actors with empty queues and newly enabled messages. Therefore we maintain a global hash table, mapping every future to the set of actors that are awaiting on its completion.

ABS uses the concept of **Deployment Component** to describe a system or a machine on which actors run. Therefore in our library we use this class to manage the two elements that the solution requires at the system level. The first is the system executor which currently in the library is a *newFixedThreadPool* singleton *ExecutorService* initialized with a fixed number of threads equal to the number of cores in the system. An actor may start a new Main Task by simply calling the static method submit(**new** MainTask()) offered by the Deployment Component. Inside the class there is support to safely call shutdown() when all the actors in the application have completed execution. The second element is the notification mechanism together with a *ConcurrentHashMap* that contains mappings of futures that hold release points on actors in the system. Actors that complete a certain future can call the static method releaseAll(f) to notify actors that contain messages suspended on that particular future.

Eliminating busy waiting. Cooperative scheduling through the "await" statements 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 how exactly does an actor verify that a release point has completed? Clearly, having a task continuously iterate through all suspended messages (busy-waiting) is inefficient. While we can permanently mark a message that needs a future to complete as available, by the nature of ABS, we cannot do that for a message which is released by a particular valid state, as its state can change again by the time it is run, so it always has to be verified before it is fetched. Instead, we assign this verification to the Main Task that iterates through the queue, and simply stop the task if no message is available. 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.

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. some more text...

5 Benchmarking the Implementation Schemes

5.1 Cooperative Schedulin 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 simple example that is illustrated in Listing 5 which involves heavy cooperative scheduling.

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), false);
   }
   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;
  }
}
```

The model creates an Actor of type "A" and sends a large number of messages to it to execute a method recursive_m(5,id). 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 actorâĂŹs 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.

5.2 NQueens Benchmark

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 4 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,

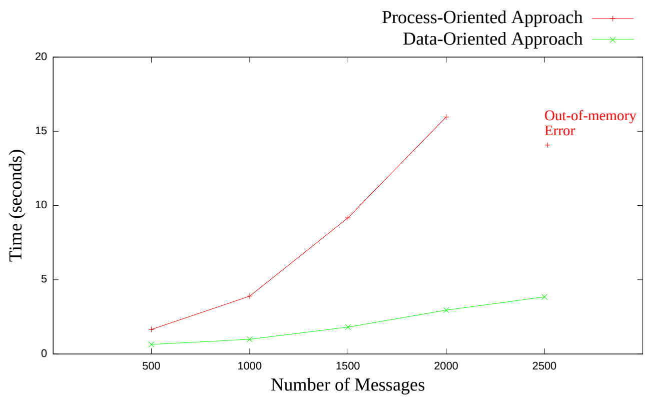

Figure 5. Performance figures for pure Cooperative Scheduling

we use the Savina benchmark for programming with actors [28]. We chose the problem 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 model. 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 application. This feature is a powerful programming concept, but we would like to see the cost of developing a benchmark in comparison to an existing actor implementation like the Akka Actor library. We compared the implementation offered by the Savina benchmark using Akka actors with our own with a board size from 10-15 on the same machine as before. The results in Figure7 show that the cooperative scheduling feature slows the program down by a factor of 2x, but this remains constant despite significant computation, even when the number of solutions computed reaches over 2,000,000.

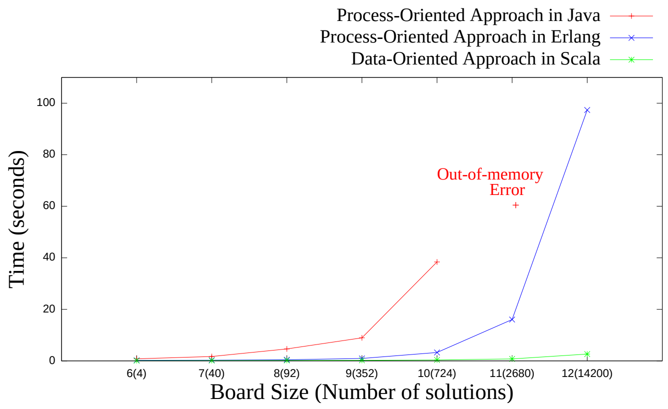

Figure 6. Performance figures for the NQueens problem in Erlang and Scala Library

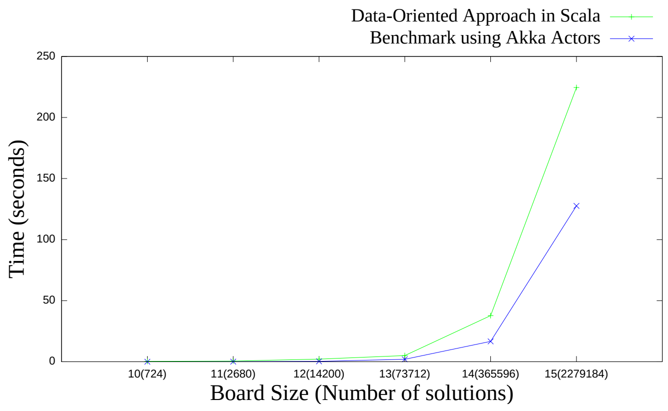

Figure 7. Performance figures for the NQueens problem implementations in Akka and Scala Library

6 Conclusions

In this paper we proposed a Java library for efficiently implementing the cooperative scheduling behavior of ABS. We observed significant improvements due to the trade-off of saving continuations including the call stack as data in memory instead of native Java threads.

To provide support for the functional data types required by ABS we extended the backend to the Scala programming language together with this library. We want to study how well these models execute into the Java Runtime Environment compared to their direct implementations without the use of modeling languages. Furthermore having a portable JVM library gives us a basis for industrial adoption of the ABS language. Finally this provides an opportunity to use ABS rich features in formal verification, resource analysis and deadlock detection as well as extensions that support real-time and distributed programming, in a software development context.

7 Acknowledgments

Partly funded by the EU project FP7-612985 UpScale: From Inherent Concurrency to Massive Parallelism through Type-based Optimizations (`http://www.upscale-project.eu`). We would like to thank the reviewers for the time and effort to provide a critical and constructive review to this paper.

References

[1] Munish K. Gupta: Akka Essentials. Packt Publishing. p. 334. ISBN 1849518289, 2012.

[2] Philipp Haller, Martin Odersky: Scala Actors: Unifying thread-based and event-based programming. Theor. Comput. Sci. 410(2-3): 202-220 (2009)

[3] Marjan Sirjani, Ali Movaghar: An actor-based model for formal modelling of reactive systems: Rebeca. Technical Report. 2001.

[4] Gul Agha: Actors: A model of concurrent computation in distributed systems. Massachusetts Institute of Technology Cambridge Artificial Intelligence Lab. 1985.

[5] Jan Schäfer: A Programming Model and Language for Concurrent and Distributed Object-Oriented Systems. Dissertation. University of Kaiserslautern, 2010.

[6] Din, C. C., Bubel, R., and HÃďhnle, R. (2015, August). KeY-ABS: a deductive verification tool for the concurrent modelling language ABS. In International Conference on Automated Deduction (pp. 517-526). Springer International Publishing.

[7] Georg Göri, Einar Broch Johnsen, Rudolf Schlatte, Volker Stolz: Erlang-Style Error Recovery for Concurrent Objects with Cooperative Scheduling. ISoLA (2) 2014: 5-21.

[8] Elvira Albert, Nikolaos Bezirgiannis, Frank S. de Boer, Enrique Martin-Martin: A Formal, Resource Consumption-Preserving Translation of Actors to Haskell. Proceedings LOPSTR 2016.

[9] Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Einar Broch Johnsen, Ka I Pun, Silvia Lizeth Tapia Tarifa, Tobias Wrigstad, Albert Mingkun Yang: Parallel Objects for Multicores: A Glimpse at the Parallel Language Encore. SFM 2015: 1-56

[10] Elvira Albert, Frank S. de Boer, Reiner HÃďhnle, Einar Broch Johnsen, Rudolf Schlatte, Silvia Lizeth Tapia Tarifa, Peter Y. H. Wong: Formal modeling and analysis of resource management for cloud architectures: an industrial case study using Real-Time ABS. Service Oriented Computing and Applications 8(4): 323-339 (2014).

[11] Elena Giachino, Cosimo Laneve, Michael Lienhardt: A framework for deadlock detection in core ABS. Software and System Modeling 15(4): 1013-1048 (2016)

[12] Joakim Bjork, Frank S. de Boer, Einar Broch Johnsen, Rudolf Schlatte, Silvia Lizeth Tapia Tarifa: User-defined schedulers for real-time concurrent objects. ISSE 9(1): 29-43 (2013)

[13] De Boer, Frank S., Dave Clarke, and Einar Broch Johnsen. "A complete guide to the future." European Symposium on Programming. Springer Berlin Heidelberg, 2007.

[14] Nobakht, Behrooz, and Frank S. de Boer. "Programming with actors in Java 8." International Symposium On Leveraging Applications of Formal Methods, Verification and Validation. Springer Berlin Heidelberg, 2014.

[15] Mehdi Dastani, Bas Testerink: Design patterns for multi-agent programming. IJAOSE 5(2/3): 167-202 (2016)

[16] Johnsen, Einar, Reiner HÃďhnle, Jan SchÃďfer, Rudolf Schlatte, and Martin Steffen. "ABS: A core language for abstract behavioral specification." In Formal Methods for Components and Objects, pp. 142-164. Springer Berlin/Heidelberg, 2012.

[17] Nobakht, B., de Boer, F. S., Jaghoori, M. M., and Schlatte, R. (2012, March). Programming and deployment of active objects with application-level scheduling. In Proceedings of the 27th Annual ACM Symposium on Applied Computing (pp. 1883-1888). ACM.

[18] Albert, Elvira, Puri Arenas, Antonio Flores-Montoya, Samir Genaim, Miguel GÃşmez-Zamalloa, Enrique Martin-Martin, GermÃąn Puebla, and Guillermo RomÃąn-DÃŋez. "SACO: static analyzer for concurrent objects." In International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 562-567. Springer Berlin Heidelberg, 2014.

[19] Albert, Elvira, Frank de Boer, Reiner HÃďhnle, Einar Broch Johnsen, and Cosimo Laneve. "Engineering virtualized services." In Proceedings of the Second Nordic Symposium on Cloud Computing & Internet Technologies, pp. 59-63. ACM, 2013.

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

[21] Serbanescu, Vlad, Keyvan Azadbakht, and Frank de Boer. "A java-based distributed approach for generating large-scale social network graphs." Resource Management for Big Data Platforms. Springer International Publishing, 2016. 401-417.

[22] Bezirgiannis, Nikolaos, and Frank de Boer. "ABS: a high-level modeling language for cloud-aware programming." International Conference on Current Trends in Theory and Practice of Informatics. Springer Berlin Heidelberg, 2016.

[23] Franco Zambonelli, Nicholas R. Jennings, Michael Wooldridge. Organisational Abstractions for the Analysis and Design of Multi-agent Systems Agent-Oriented Software Engineering. Volume 1957 of the series Lecture Notes in Computer Science pp 235-251

[24] Johnsen, Einar Broch, Olaf Owe, and Ingrid Chieh Yu. "Creol: A type-safe object-oriented model for distributed concurrent systems." Theoretical Computer Science 365.1-2 (2006): 23-66.

[25] Johnsen, Einar Broch, Olaf Owe, and Eyvind W. Axelsen. "A run-time environment for concurrent objects with asynchronous method calls." Electronic Notes in Theoretical Computer Science 117 (2005): 375-392.

[26] https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html

[27] Hahnle, Reiner, and Muschevici, Radu (2016, October). Towards incremental validation of railway systems. In International Symposium on Leveraging Applications of Formal Methods (pp. 433-446). Springer International Publishing.

[28] Imam, Shams., and Sarkar, Vivek. (2014). Savina-an actor benchmark suite. In 4th International Workshop on Programming based on Actors, Agents, and Decentralized Control, AGERE.

[29] https://github.com/vlad-serbanescu/abs-api-cwi.git

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

A Appendix

Text of appendix …