Cooperative Scheduling Behaviour from Modeling to Runtime

Vlad Serbanescu, Frank de Boer, and Mahdi Jaghouri

Leiden Institute of Advanced Computer Science
Centrum Wiskunde and Informatica
{vlad.serbanescu,frank.s.de.boer,jaghouri}@cwi.nl
http://www.cwi.nl

Abstract. Modeling applications at the design phase of any project is highly important in order to build a reliable, fast and robust system. Understanding the control flow of execution from just the model is crucial for the applications next software engineering phases such as implementation, testing, release and maintenance. If we add that a large portion software systems are running several jobs in parallel, with a good model we can observe data consistency and detect deadlocks efficiently. One of the toughest models to execute, especially in a parallel or distributed environment is the Actor-based model which introduces the notion of cooperative scheduling, a high-level synchronization mechanism which allows an actor to continue to execute messages from its queue when the current message execution is suspended . In this paper we will investigate some of the challenges of translating the cooperative scheduling behavior into the Java Runtime Environment. We will analyze the Abstract Behavioral Specification (ABS) Language concepts which are very well suited for Agent-based applications modeling and provide a benchmark for testing several solutions of efficiently running cooperative scheduling behavior.

Keywords: Cooperative Scheduling, Mult-Agent Systems, High-Performace Computing, Language Design

1 Introduction

-high-level description of cooperative scheduling
-position the paper with scala and Akka, state of the art
- problem statement: how stacks need to be saved in OO, how expensive a thread is
- strongly typed actors messages. - the scope of the paper is implementation in the Java Runtime Environment/Java Virtual MAchine(JVM) - using ABS as a mute programming language, real software development context, support for object-oriented design and foreign lanuage interface.

2 ABS Language Concepts

Our reference modeling language for analyzing cooperative scheduling is ABS[4], an actor-based modeling language whose semantics offer high-level synchronization mechanisms for parallel and even distributed applications. This is a very powerful modeling language with extensive support for concurrent programming[5], resource analysis[6], deadlock analysis[7] and remote communication [8, 9]. Our focus in this paper are the semantics of the language that offer high-level constructs for modeling asynchronous communication between actors using messages and cooperative scheduling of these messages by an actor. The second construct can have optional annotations that define custom schedulers in order to satisfy an actor's specific behavior. Asynchronous method calls can also be annotated with costs and deadlines which can be combined with cooperative scheduling policies to create a very powerful scheduler.

2.1 Asynchronous Method Invocation

In ABS all actors communicate with each other using asynchronous method invocation generated by a very simple construct "a!m()" which sends a message to another actor "a" to execute method "m". The semantics allow synchronous method calls that only change the internal state of an actor. ABS also has support for grouping Actors into Concurrent Object Groups (COG) that allow synchronous communication between actors of the same COG, but for the scope of our analysis and implementation simplicity we will assume that each actor is defined in its own group. Furthermore the semantics of ABS impose that each actor has a queue that stores all invocations coming from other actors as messages that are to be executed. Each actor executes the messages sequentially in its own context therefore all actors run in parallel. An important observation to make here is that in modeling an application, ABS assumes there is no "message overtaking", that is the order of messages delivered from one particular actor to another is the same as the order in which they were generated. This is a significant constraint that our implementation needs to take into account and an important synchronization mechanism as will be described in our modeled application later in this section. The result of an asynchronous method invocation can be captured in an future which the caller actor can use to retrieve the result and use it, but also to block its current execution in order to synchronize with the callee actor. This is done through the "future.get" statement that a future supplies and has the same semantics that futures have to synchronize concurrent objects in other modeling and programming languages.

2.2 Await Construct

The "await" statement of ABS offers a high level mechanism that controls execution within an actor based on its internal state or the state of other actors in the system. As explained in Section 2.1, futures can be used to synchronize between actors. However, in combination with the "await" construct, futures can also

be used to control and suspend message execution within an actor. A statement like "await future?" differs from "future.get" as it doesn't block the entire execution of the actor, but instead block the current message from which the "await" statement was called and allows the actor to continue with executing subsequent messages that are available in its queue. The same behavior of an actor can also be achieved by combining the "await" statement with a boolean condition that describes a particular state of the actor. For example "await isPrime(this.x)" will suspend the current message executing this statement until the field x of the actor is set to a prime number. In both uses of the "await" contruct the suspended message will be made available once the future is completed or the boolean condition evaluates to true. The effectiveness of this mechanismm will be illustrated in a real example in Section 2.3. While this behavior is very trivial to model, we will see some of the challenges encountered when implementing this behavior in Sections 4 and 5.

- the main concepts: async calls, await on boolean and futures, each object has its own queue. - example for coroutines (Paul Klint paper at SEN)

2.3 Proof of concept

Agent-based modeling has been shown to be a powerful means to express organizational abstractions, including auctioning systems [10, **?**]. However, it is still a major challenge to provide an efficient implementation of these high-level agent-based modeling concepts, e.g., the deliberation cycle which integrates a goal-oriented computation with an event-based communication approach. To lower the barrier for adoption by industry, Dastani and Testerink [3] propose a Java methodology which guides the development of agent-based models. This includes a corresponding library of object-oriented design patterns for modeling agent-based concepts, called OO2APL.

A major part of the complexity in OO2APL stems from the implementation of a middleware for management of the deliberation cycle of and the event-based communication between agents. Furthermore, this middleware is tightly coupled with the high-level design patterns for agents. In contrast, Listing 1.1 shows that modeling agents directly by actors allows to abstract in a concise manner from the underlying implementation of the deliberation cycle and the event-based communication mechanism.

Listing 1.1. Generic Agent Model

```
1  data Goal = Goal;
2  data Belief = Belief;
3  data Message = Message;
4
5  interface Agent {
6        Unit message(Message m);
7  }
8
9  class A1(Set<Belief> init_beliefs, Set<Goal> init_goals)  implements Agent {
10       Set<Belief> beliefs = init_beliefs;
11       Set<Goal> goals = init_goals;
12
13       Unit message(Message m) {
```

```
14              case m {
15                    Message =>  { }
16                    _ => { }
17              }
18        }
19
20        Unit goal_rule(Goal g) {
21              case g {
22                    Goal => { }
23                    _ => { }
24              }
25        }
26
27        Unit run() {
28               while(this.goals != EmptySet) {
29                for (g in this.goals) {
30                   this.goal_rule(g);
31                }
32                suspend;
33              }
34        }
35
36  }
```

Listing 1.2 shows a concrete instantiation of an agent in an auctioning system.

Listing 1.2. Agent Model

```
1   data Message =
2   Announce(Agent announceCaller, Route toSell, Price price) |
3   Bid(Agent bidCaller, Route toBuy, Price bid) |
4   Sold(Agent soldCaller, Route soldItem) |
5   Result(Set<Agent> winners, List<Price> prices, List<Agent> unhappy);
6
7   class BiddingAgent(Belief init_value, Goal init_goal, Rat risk) implements Agent {
8         Belief belief = init_value;
9         Set<Goal> goals = set[init_goal];
10
11        Unit message(Message m) {
12              case m {
13                    Announce(caller, slot, price) =>  { ...   }
14                Sold(caller, slot) => { ... }
15              }
16        }
17  }
```

a generic agent model

generic mechanism for querying the belief base by pattern matching

no need for explicit registry of agents, so get for free garbage collection

we have a natural match between ABS methods and plan execution of agents, which the cooperative scheduling provides additional flexibility tin the implementation of the deliberation cycle.

-expressive generic agent model
-software engineering context
- support for functional data types, FLI, type-checking
-sequence of events that maximize agent throughput

3 Cooperative Scheduling Implementation Schemes

-sequence diagram of what happens in an actor during synchronous and asynchronous method calls.

3.1 Modeling Language Concepts in the JVM

An Actor has a lock and Every Asynchronous call is a Thread The trivial straightforward approach for implementing cooperative scheduling in Java is to model each actor as an object with a lock for which execution threads compete. Each asynchronous call would then generate a new thread with it own stack and context. Whenever a control switch statement occurs the thread would be suspended by the JVM's normal behaviour and the available threads would then compete for the actor's lock. When the release condition is enabled the suspended thread would become available and in turn compete with the other available threads in order to execute on the actor.
We have one lock per Actor for ensuring single thread execution.

Every Actor is a Thread Pool -free queue, less space -Each async. call is modeled as task.

Every System has a Thread Pool -each actor is modeled as a task with a queue of lambda expressions. -particular uses of an executor service.

Synchronous calls context -to save execution context we save the current call stack as a Thread and optimally suspend it

Continuations as tasks - fully asynchronous environment
- modeling continuations as labeled functions and converting them to lambda expressions.
- allocating more memory in the heap and saving memory on the stack.

3.2 Optimizations for the JVM

Demand-driven Approach

Optimal Usage of System Threads

Eliminating Busy Waiting

Using JVM Garbage Collection

4 Continuation Generation at Compile Time

- Mahdi's blog post

Conceptually an actor has one thread of execution, which means it can run only one method at a time. Practically, however, allocating a real thread for each actor is highly expensive. Very roughly speaking, an executor service in Java provides a queue of tasks and an efficient way of running those tasks on a few threads. Due to the optimizations provided by Java implementations, an executor in principle is the best way to scale to many concurrent tasks. We use the terms executor and thread-pool interchangeably. There are two possibilities in how to use executors when modeling actors.

One Task Per Message

Conceptually, one message handler (or method) may be seen as the unit of execution forgetting about release points and awaits for now. By submitting one task to the executor upon receiving a message, we in effect also delegate the queueing of the messages over to the executor. This is good as we may assume executors have optimized implementations for handling queues. The disadvantage here is that we will need a lock per actor that must be checked by every message handler upon start, and freed upon completion. We can reduce the number of lock-based synchronization by handling actor queues manually, as described below.

One Task Per Actor

From a different perspective, we may see an actor itself as a unit of execution, as in conceptually has exactly one thread. This justifies having one task in the executor thread pool per actor. This means that we must handle the message queue of each actor separately. The task corresponding to an actor is then responsible for taking messages one by one from the queue and running them. This removes the requirement to synchronize every message handler, but it comes at the cost of having to manage message queues manually.

There is a problem when the queue is empty. Since we do not want to make this task busy-wait until a message arrives, an idea would be to check upon insertion of a new message into the queue whether such a task exists already. This again, however, requires some careful synchronization, but we expect that this is less severe than the previous case (although we cannot prove it). To do so, for every actor, we keep a local atomic boolean flag 'running'. A first approach looks like this:

// inside the task Runnable task = () =¿ (while (!q.isEmpty()) // take one message and run it running.set(false);); // when inserting a new message q.insert(m); if (!running.getAndSet(true)) executor.submit(task); The problem with the above code 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 task. To remedy this, we need to 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 'q' or 'running' as the lock. Additionally, either the insertion into the 'q' or getAndSet of 'running' should also use the same lock obviously.

boolean isQueueEmptyAndReset(q, running) synchronized (running) if (q.isEmpty()) running.set(false); return true; return false; boolean getAndSetSync(running) synchronized (running) return running.getAndSet(true);

// inside the task Runnable task = () =¿ (while (!isQueueEmptyAndReset(q, running)) // take one message and run it); // when inserting a new message q.insert(m); if (!getAndSetSync(running)) executor.submit(task); Additionally, the flexibility on controlling the queues may even be useful. We can indeed make use of this for handling release points and await statements. We suggest to use various queues for every actor, each coupled with a predicate, for example checking a boolean or completion of a future, and then at every scheduling point, one message from one of the enabled queues is taken, and executed. This is obviously assuming that when releasing the processor (e.g., via 'await') the continuation is clearly a runnable/callable that can be put into the queue.

Fairness

Another problem with the approach above is that (when there are more actors than available threads), some actors may keep processing one message after the other from its queue, and thus keeping the thread it is running on, while some other actors are starving, i.e., not being assigned to any thread. To remedy this, we could change the while loop to an if statement like this:

// inside the task Runnable task = () =¿ (// take one message and run it, and then ... if (!isQueueEmptyAndReset(q, running)) executor.submit(this);); // when inserting a new message q.insert(m); if (!getAndSetSync(running)) executor.submit(task);

Continuations

When a method releases the processor, its continuation must be scheduled as a new task. There are two problems with that. Firstly, the continuation is not just another method, but part of it, and thus cannot just be turned into a Runnable, so that it can be enqueued. Secondly, in presence of synchronous calls, the whole stack needs to be part of the continuation.

First, we assume no stack is present and try to address the first problem. Then we propose a solution to the stack issue in an orthogonal way. Theoretically, at runtime, you simply could make a pointer to the current instruction pointer and use it as the continuation. Of course, you need to store all the local variables. This would be a viable solution, if we could at runtime create a method whose entry point is the current (or more specifically, the next) instruction (or in terms of Java, the next bytecode instruction). This method should take as parameters, all the local variables of the currently executing method, including its parameters. Another solution is to do a preprocessing and statically create these methods.

Preprocessing continuations

Assuming that code is written in ABS and then compiled to Java, we can do the preprocessing in ABS or in Java. But possibly this is better to do at Java because then this code would apply also when using ABS API directly in Java. The following explains how to do the preprocessing.

The idea is that every await is replaced by a method call. For example:

```
void m1(p) {
```

```
while (b) {
await g;  // let say this is await number 1
i = 0;
}
}
```

will translate to

```
void m1(p) {
while (b) {
if (!g) {
this ! await_1(i, b, p);  // assume ! represents async send
return;
}
i = 0;
}
}
```

Then we have to generate the method `|await_1()` or in fact one of these methods per await statement. We can generate these methods using the following rules:

```
Cont(S1;S2) = Cont(S2)     IF await in S2
Cont(S1;S2) = Cont(S1);S2  IF await in S1
Cont(while(b){S}) = Cont(S); while(b){S}
Cont(if(b) S1 else S2) = Cont(S1) IF await in S1
Cont(if(b) S1 else S2) = Cont(S2) IF await in S2
For the above example, we get the following:

Cont(m1) = Cont(while(b) { await g; i = 0 })
= Cont(await g; i = 0); while(b) {await g; i = 0 }
= { i = 0;  while(b) {await g; i = 0 } }
boolean await_1(i, b, p) {
i = 0;
while(b) {
if (!g) {
this ! await_1(i, b, p);
return;
}
i = 0
}
}
```

Stack calls and continuation

The idea is to change a synchronous call ‘x = o.m();‘ into an asynchronous call plus an await like ‘f = o!m(); x = await f.get();‘. There are two problems here. When the callee is the same actor (on on the same COG in ABS world), 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. Luckily, both can be implemented by one change to the scheduler. This change regards the implementation of the task, as we described in the previous post. That task will need another flag ‘isSynchCall‘ and when that flag is true, it will immediately execute the message that is to be enqueued instead of taking an arbitrary one from the queue. This will work for the return also assuming the method described below for handling the completion of futures.

Completion of futures

The problem is when an actor has no enabled messages, it may only be reenabled when a future it is waiting on completes. But how should the actor be awakened in this case? The answer is by the actor who completes the future.

There must be a global hash table, mapping every future to the set of actors that are awaiting on its completion. When a future completes (basically when a method finishes), the current actor looks up this future in this hash table, and sends a special ‘enable‘ method to all awaiting actors, and this method takes the future as the parameter. On the other side, every actor puts the messages awaiting a future into a special queue of suspended messages. The ‘enable‘ method takes the message waiting in this future and puts them to the default queue of enabled messages. Note that the messages awaiting on boolean conditions, are in separate queues as we discussed in the previous post.

- formal explanation for creating continuations.

5 Runtime Behaviour

- the new Java backend

6 Benchmarking the Implementation Schemes

-multiple stack frame problem. - Old Java, vs Erlang vs New Java benchmark, vs possibly Optimized suspended threads.

7 Conclusions

References

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

2. 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.
3. Mehdi Dastani, Bas Testerink: Design patterns for multi-agent programming. IJAOSE 5(2/3): 167-202 (2016)
4. Johnsen, E. B., Hhnle, R., Schfer, J., Schlatte, R., and Steffen, M. (2010, November). ABS: A core language for abstract behavioral specification. In International Symposium on Formal Methods for Components and Objects (pp. 142-164). Springer Berlin Heidelberg.
5. 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.
6. Albert, E., Arenas, P., Flores-Montoya, A., Genaim, S., Gmez-Zamalloa, M., Martin-Martin, E., and Romn-Dez, G. (2014, April). 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.
7. Flores-Montoya, A. E., Albert, E., and Genaim, S. (2013). May-happen-in-parallel based deadlock analysis for concurrent objects. In Formal Techniques for Distributed Systems (pp. 273-288). Springer Berlin Heidelberg.
8. erbnescu, V., Azadbakht, K., and de Boer, F. (2016). A java-based distributed approach for generating large-scale social network graphs. In Resource Management for Big Data Platforms (pp. 401-417). Springer International Publishing.
9. Bezirgiannis, N., and de Boer, F. (2016, January). ABS: a high-level modeling language for cloud-aware programming. In International Conference on Current Trends in Theory and Practice of Informatics (pp. 433-444). Springer Berlin Heidelberg.
10. 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

Appendix: Springer-Author Discount

LNCS authors are entitled to a 33.3% discount off all Springer publications. Before placing an order, the author should send an email, giving full details of his or her Springer publication, to `orders-HD-individuals@springer.com` to obtain a so-called token. This token is a number, which must be entered when placing an order via the Internet, in order to obtain the discount.

8 Checklist of Items to be Sent to Volume Editors

Here is a checklist of everything the volume editor requires from you:

☐ The final LaTeX source files

☐ A final PDF file

☐ A copyright form, signed by one author on behalf of all of the authors of the paper.

☐ A readme giving the name and email address of the corresponding author.