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[3], 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[4], resource analysis[5], deadlock analysis[6] and remote communication [7, 8]. 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: Actor-based implementation of Agent-based Modelling

-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 - 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. 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.
4. 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.
5. 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.
6. 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.
7. 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.
8. 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.

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.