Actor Profiling in Virtual Execution Environments

Andrea Rosà
Università della Svizzera italiana

Lydia Y. Chen
IBM Research Lab Zurich

Walter Binder
Università della Svizzera italiana

Abstract

Nowadays, many virtual execution environments benefit from concurrency offered by the actor model. Unfortunately, while actors are used in many applications and computing frameworks, existing profiling tools are little effective in analyzing the performance of applications using actors. In this paper, we present a new instrumentation-based technique to profile actors in virtual execution environments. Our technique adopts platform-independent profiling metrics that minimize the perturbations induced by the instrumentation logic and allow comparing profiling results across different platforms. In particular, our technique measures the initialization cost, the amount of executed computations, and the messages sent and received by each actor. We implement our technique within a profiling tool for Akka actors on the Java platform. Evaluation results show that our profiling technique can help performance analysis of actor utilization and communication between actors in large-scale computing frameworks.

1. Introduction

Nowadays, actors are widely adopted in parallel and distributed applications. The fundamental principles of the actor model (i.e., location-unaware addressing, absence of shared states, and asynchronous event-based processing) enable great opportunities for scalability in many environments. Thus, actors are implemented in several programming languages [4, 14, 18] and libraries [2, 6, 7] for many virtual execution environments, and are used by numerous applications, including parallel [31] and distributed [5, 35] frameworks.

Despite the spread of actors in nowadays applications, there is a lack of effective profilers centering on actors. The typical characteristics of the actor model—such as opaque addressing, large exchange of asynchronous messages, and application logic strongly tied to the type of the message received—turn conventional parallelism [19, 22, 25] and communication [16, 33, 34] profiling techniques little effective in monitoring actor-based applications. While actor profilers should focus on the computations executed by actors, as well as on the communication between them [29], existing profiling techniques shed little light on these points.

In this paper, we present a novel technique for actor profiling. Given the wide presence of actor libraries for multiple platforms, our approach has been designed for portability. Our technique relies on bytecode instrumentation, which is applicable for any actor library and programming language running on a virtual execution environment. We employ platform-independent dynamic metrics to track computations and communication, ensuring largely reproducible profiles that are comparable across different platforms. Moreover, the collected metrics are mostly not perturbed by the inserted instrumentation code, yielding accurate measurements. To demonstrate how our approach can be applied to an existing platform, we implement our technique in a profiling tool for Akka [2], one of the most used actor libraries on the Java platform. Our implementation is based on the DiSL instrumentation framework [26]. Evaluation results conducted on actor-based workloads and large-scale computing frameworks show that the derived profiler can effectively track computations and communication in Akka, helping in the performance analysis of applications using actors.

This paper presents the following contributions. We describe a portable technique to profile platform-independent metrics on actors in virtual execution environments, particularly focused on the initialization cost, the executed computations, and the communication between actors (Sections 4 and 5). From our technique, we derive a novel profiler for Akka actors, being the first tool providing portable, comparable, and accurate metrics on the computations executed by actors in Akka applications (Section 6). Finally, we analyze actor utilization and the communication occurred between actors in applications relying on Akka, including the Apache Spark [35] and Apache Flink [5] computing frameworks, pinpointing performance drawbacks and suggesting potential improvements (Section 7). A discussion on the benefits and limitations of our technique (Section 8) and a comparison of the proposed approach with related work (Section 9) complement the description of our technique.

First, we present some background information on the technologies involved in our approach (Section 2) and summarize our design goals (Section 3).

2. Background

In this section, we present some background information on the actor model, Akka, and DiSL.

Actors and Akka. The actor model [23] conceives actors as the atomic elements of a concurrent application [13]. Actors continuously listen for incoming messages in their mailbox, processing them one at a time. Depending on the type of the received message, actors can execute different actions, i.e., carry on computations, change their state or their defined behavior, create other actors, or send messages to other actors.

The model defines strict properties that actors must satisfy. First, actors cannot share states with other actors; this avoids the occurrence of data races. Second, actors communicate only via asynchronous message exchange, simplifying the design of applications and improving scalability. Actors never wait for

the reception of a specific message, helping avoid deadlocks in the system. Finally, actors are referenced only through opaque identifiers that mask the physical location of each actor, making actors easy to distribute across cores or machines. As a consequence, actors can benefit particularly applications where computations can be decomposed into independent tasks linked by a well-defined workflow. Frequently, actors are used along with other mechanisms to obtain concurrency, such as threads or futures [32].

Among the libraries implementing actors on the Java Virtual Machine (JVM), Akka [2] is one of the most used. In particular, Akka actors have deprecated the original Scala actors since 2013, and more than 250 applications hosted on GitHub rely on Akka actors [1], including large-scale computing frameworks such as Apache Spark [35] and Flink [5].

In Akka, every object of type *akka.actor.Actor* is an actor. Actors are known to each other via a unique and immutable identifier which hides the real location of the actor. To receive messages, actors must implement the *receive* method with a Scala *PartialFunction* defining what message types an actor can handle and how it should behave upon the reception of a message. Communication between actors occurs asynchronously through the exchange of immutable objects as messages. Actors can send messages to other actors through two methods. The *tell [!]* method sends a non-blocking message to another actor and returns immediately, while the *ask [?]* method sends a message and returns a *Future*, representing a possible reply.

DiSL. DiSL [26] is a dynamic program analysis framework based on Java bytecode instrumentation. DiSL is based on Aspect-Oriented Programming (AOP) principles that allow a concise implementation of runtime monitoring tools. In DiSL, developers write instrumentation code in the form of *snippets*. The language constructs provided by DiSL allow one to specify where a snippet should be woven through *guards* (predicate methods that enable the evaluation of conditionals at weave-time to determine whether a snippet should be woven or not), *markers* (specifying which parts of a method to instrument, such as method bodies, basic blocks, etc.), and *annotations* (specifying where a snippet shall be inserted wrt. a marker, for example before or after a basic block).

Snippets have access to complete *context information* provided via method arguments. Context information can be of two types: static (i.e., static reflective information limited to constants) or dynamic (i.e., including local variables and the operand stack). DiSL supports also *synthetic local variables* (that enable data passing between different snippets woven into the same method body) and *thread-local variables* (that are implemented by additional instance fields in *java.lang.Thread*). Both variables can be expressed as annotated static fields (i.e., *@SyntheticLocal* and *@ThreadLocal*). The DiSL weaver guarantees complete bytecode coverage to ensure that analysis results represent overall program execution. In particular, complete bytecode coverage enables instrumention of the Java class library. DiSL uses a separate process for instrumentation to reduce perturbations and interferences in the observed JVM.

3. Design Goals

Our goal is to provide a technique to profile actors in virtual execution environments. Our approach centers on portable, accurate, and platform-independent metrics to produce comparable and portable profiles that are little affected from perturbations arising from the inserted instrumentation code. Here, we give a high-level overview of the used techniques and properties of the used profiling metrics.

Bytecode Instrumentation. Our profiling technique is based on bytecode instrumentation, such that our approach can be applied to various virtual execution environments, since it does not depend on the details of the underlying hardware. In contrast, binary instrumentation and low-level metrics collected with hardware performance counters are inherently platform-dependent. Our approach aims at generating exact profiles, as opposed to sampling-based profiling, where only part of a program's execution is profiled. While sampling introduces less overhead, it may result in inaccurate profiles.

Bytecode Count. We rely on the number of executed bytecodes to measure computations carried out by actors, in contrast to many existing profilers that focus on metrics such as CPU time. We prefer the bytecode count to CPU time for several reasons [17]. The former is platform-independent, as the number of executed bytecodes remains the same in different hardware or virtual machine implementations (assuming that the same class library is used), while measuring CPU time accurately may require special support from the operating system or the hardware, limiting portability.

The bytecode count allows one to compare profiles obtained on different platforms, since they are based on the same platform-independent metric, and guarantees fully reproducible measurements for deterministic applications (when also the thread scheduling is deterministic). In contrast, CPU-time profiling is usually impossible to reproduce faithfully, and different platform characteristics affect CPU time significantly, resulting in incomparable profiles [28].

Finally, the bytecode count is accurate, i.e., profiling it does not affect the generated profile (apart from possibly different thread scheduling due to the inserted instrumentation code). On the contrary, measurements of CPU time are affected by the instrumentation logic which increases CPU time consumption. Nevertheless, our approach requires full bytecode coverage to provide accurate results. This implies that a concrete profiling tool implementing the proposed technique must be able to instrument the bytecodes of every loaded class. Section 6 presents an implementation of our approach that employs full bytecode coverage on the JVM.

Platform-independent Metrics. We aim at collecting a set of platform-independent metrics that are suitable for a broad range of actor applications and are not perturbed by the instrumentation code. Our approach does not attempt to track detailed metrics covering all possible aspects of such applications. Rather, it aims at collecting high-level metrics centered on actors that are not specific to a given platform or actor library, from which users can easily derive additional metrics

Table 1. Operations defined on Actor Profile (AP).

Operation	Description
createAP(A a, LONG ic): AP	Creates a new actor profile associated with actor *a*, and registers *ic* as its initialization count. The operation fails if an AP associated with *a* already exists, or if $ic \leq 0$.
getAP(A a): AP	Returns the AP associated with actor *a*. The operation fails if there is no AP associated with *a*.
getActor(AP ap): A	Returns the actor associated with *ap*.
getInitializationCount(AP ap): LONG	Returns the initialization count associated with *ap*.
registerMessageSent(AP ap, TYPE t): AP	Registers the transmission of a message of type *t* in *ap*, incrementing the message-sent counter for message type *t* by one. Returns *ap* after the update.
getMessagesSent(AP ap, TYPE t): INT	Returns the message-sent counter for message type *t*, associated with *ap*.
registerMessageReceived(AP ap, TYPE t, LONG cc): AP	Registers the reception of a message of type *t* in *ap*; handling the message has taken *cc* bytecodes. Increments the message-received counter (by one) and the computation count (by *cc*) for message type *t*. Returns *ap* after the update. The operation fails if $cc \leq 0$.
getMessagesReceived(AP ap, TYPE t): INT	Returns the message-received count for message type *t*, associated with *ap*.
getComputationCount(AP ap, TYPE t): LONG	Returns the computation count for message type *t*, associated with *ap*.
getMessageTypes(AP ap): $\mathscr{T}$	Returns a set of types $\mathscr{T}$. Each $t \in \mathscr{T}$ corresponds to the type of at least one message sent or received.

Table 2. Operations defined on Shadow Stack (SS).

Operation	Description
createSS(INT n, TYPE T): SS<n,T>	Creates a shadow stack that allows the inspection of the top *n* elements. The stack can contain only elements of type *T*, and is associated to the thread executing the operation. Pushes ϵ *n* times on the stack. The operation fails if $n < 1$.
push(SS<n,T> ss, T e): SS<n,T>	Pushes element *e* on the top of the stack *ss*, returning *ss* after the insertion. The operation fails if $e = \epsilon$.
top(SS<n,T> ss, INT i): T	Returns the element stored *i* positions from the top of the stack *ss*. $i = 0$ denotes the top of the stack. *ss* is not modified. Note that this operation can return ϵ. The operation fails if $i < 0$ or $i \geq n$.
pop(SS<n,T> ss): SS<n,T>	Removes the element *top(ss,0)*. Returns *ss* after the removal. The operation fails if *top(ss,0)* $= \epsilon$.

and statistics. In particular, our technique collects metrics on the *initialization count*, the *computation count*, and the *messages sent and received* by each actor. The first two metrics are expressed as bytecode counts: the initialization count measures the number of bytecodes executed in the dynamic extent of each actor constructor, while the computation count tracks all bytecodes executed by actors in response to the reception of messages. These two metrics enable one to analyze the performance of actors spawned by the application, to investigate how much they are utilized, and to assess the cost of their initialization.

On the other hand, tracking messages sent and received provides a high-level view of the communication between actors in the profiled program. While our approach focuses on platform-independent metrics, it could be easily extended to collect other platform-specific metrics related to communication, such as the amount of communication incurred over the network.

4. Profiling Data Structures

Here, we define the data structures used by our profiling technique in terms of abstract data types. We start with the data structure used to collect all information related to actors. Then, we introduce an auxiliary structure to store and access calling-context information. We assume the existence of the types A (denoting actors), TYPE, INT and LONG, as well as the possibility of creating aggregate types (here expressed with the mathematical notation $\mathscr{T}$).

4.1 Actor Profile

Every information regarding actors is collected in a data structure called *Actor Profile (AP)*. An AP is associated with exactly one actor and stores data related to its initialization and to the messages sent and received. Table 1 summarizes all relevant AP operations. A new AP instance must be created along with the creation of a new actor, to be able to track subsequent communication and computations executed by the actor. Since each actor has a unique identifier (e.g., in Java, a reference to the actor instance), each actor can be mapped to the corresponding AP.

Conceptually, an AP instance can be represented as a sequence of tuples. The first tuple stores the actor reference and its initialization count. Subsequent tuples store a set of counters (i.e., the number of messages sent, the number of messages received, and the total amount of computations executed) for each message type; that is, AP collects separate metrics for each message type. We denote $\mathscr{T}$ the set of types for which a tuple is present in a given AP. When an AP is created, $\mathscr{T} = \emptyset$. The functions *registerMessageSent* and *registerMessageReceived* check whether the type t of the sent (resp. received) message $\in \mathscr{T}$ before updating the corresponding counters. If not, t is added to $\mathscr{T}$, and a related tuple is created.

Aggregated per-actor metrics (e.g., total number of sent/received messages, computation count, average computation count per message) can be easily derived from an *AP*. For example, the computation count of an actor *a* can be computed as $\sum_{t \in getMessageTypes(getAP(a))} getComputationCount(getAP(a), t)$.

4.2 Shadow Stack

Our instrumentation technique needs to store context information related to executed methods[1] and to make them available to subsequent callees. As this operation is done frequently, we define an auxiliary data structure, the *Shadow Stack (SS)* to support storing and accessing this kind of information.

Our shadow stack is similar to a usual stack, but supports access to several top elements rather than just the top of the stack. Its operations are summarized in Table 2. Upon creation of the data structure, one must define the element type *T* of the stack, and how many top elements $n \geq 1$ it provides access to. Upon creation, *n* special elements are pushed onto the stack; we call them *empty elements* ϵ of type *T*. The empty elements

[1] For brevity, we use "method" to refer to either a method or constructor.

```
1   actor A {...}
2   actor B {
3       B() { ... }                    // constructor of B
4   }
5   actor C subtype of B {
6       C() {                          // constructor of C
7           B();                       // account to current actor
8           A a = new A();             // account to new actor
9       }
10  }
```

Figure 1. Multiple constructors and nested actor creation.

cannot be popped from the stack. Hence, the stack is never empty.

Elements on the stack can be inspected with the *top* operation, which provides access to the top of the stack as well as to subsequent elements (up to the *n-1*th element from the top). For example, *top(s,0)* returns the element on the top of the stack, *top(s,1)* returns the element immediately below the top, and so on. Note that *top* may return ϵ elements in case the stack has not been filled up enough with *push* operations.

Although the data structure can be used for other purposes than storing and accessing calling-context information, in this paper we restrict its applicability to this task. Each shadow stack is thread-local, accessible only by the owning thread. In each method, there can be at most one *push* operation for each shadow stack; such *push* operations are only allowed on method entry. For each *push*, there must be a corresponding *pop* on method completion. No other *push* or *pop* is allowed. Consequently, at the end of each method the state of each shadow stack is the same as upon method entry. A shadow stack does not have to be updated upon method entry; that is, it is not required that a shadow stack strictly represents the call stack of its owning thread. If a method is not relevant for an instrumentation, there is no need to create a corresponding shadow-stack element.

Shadow stacks can be efficiently implemented by embedding them into the frames of the call stack and into thread-local variables, without requiring any heap-allocated array, as we will show in Section 6.

5. Metric Collection

Here, we describe our technique for profiling actors. We present our approach by means of pseudocode using AOP notations to express where instrumentation code is woven (e.g., before or after method bodies). Our representation can be easily translated into a platform-specific instrumentation (e.g., Section 6 shows a concrete implementation for Akka on the JVM).

We assume that the inserted instrumentation code has access to the following context information: the receiver of the currently executing method (*currentObject()*), the size of the currently executed basic block of code in terms of bytecodes (*currentBBSize()*), and the message being sent (resp. received) (*currentMessage()*), which is typically passed as argument to send (resp. receive) methods. The function *typeOf(...)* returns the type of an object. We use the notation *[TL]* to declare thread-local variables, which are created during thread initialization.

5.1 Initialization Count

To track the initialization count of each actor, the profiler counts all bytecodes executed in the dynamic extent of an actor constructor and associates this count with the created actor. Before detailing our proposed profiler, we describe some cases requiring special attention.

Multiple Constructors and Nested Actor Initializations. Correctly accounting the executed bytecodes to a newly created actor is complicated by the presence of *multiple constructors* (i.e., an actor constructor may call another constructor for initializing the same actor) and *nested actor initializations* (i.e., an actor may create another actor within its constructor). In the former case, the inserted instrumentation code needs to determine when the initialization of an actor is complete, updating the corresponding counter only at that moment. In the latter case, the profiler must separate the initialization counts of the actors.

For example, consider the scenario illustrated in Figure 1. Actor *C* is a subtype of actor *B*. When actor *C* is created, the constructor of actor *B* is invoked within the constructor of *C* (line 7). The profiling logic must ensure that the bytecodes executed in the constructor of actor *B* are still accounted to the actor being created. Moreover, actor *C* creates actor *A* within its constructor (line 8). The bytecodes executed for creating *A* shall be accounted only to actor *A*, and not also to actor *C*, so as to avoid counting initialization bytecodes multiple times. These situations occur frequently in actor applications. We rely on shadow stacks to detect the aforementioned cases, ensuring accurate accounting of initialization counts.

Profiling Technique. We report our approach to profile the initialization count of actors in Figure 2(a). The instrumentation logic relies on four thread-local variables. The *bc* counter (line 2) is used to keep track of the number of bytecodes executed by each thread over its execution. Before each basic block of code, *bc* is incremented by the size of the basic block (lines 9–11). The *bcNested* counter (line 3) tracks the initialization count of nested actors.

The thread-local variables *ss_actor* and *ss_bcEntrance* are shadow stacks storing the actors under initialization and the bytecode count upon constructor entrance, respectively (lines 4–6). Following the rules for manipulating shadow stacks, the relevant data is pushed at the beginning of constructors (lines 14–18) and popped at the end of constructors (lines 35–36). While *ss_bcEntrance* provides access only to the topmost element, *ss_actor* also provides access to one element below.

The initialization count (line 22) is computed as the difference between the value of *bc* at the end and at the beginning of the constructor (the latter has been stored on *ss_bcEntrance*). This difference is reduced by the bytecodes executed for creating nested actors, stored in *bcNested*.

Shadow stacks are necessary to identify the presence of multiple constructors (of the same actor) or nested actor initializations. The former case occurs if *top(ss_actor,1)=top(ss_actor,0)* (implying *top(ss_actor,1)*$\neq\epsilon$), meaning that the constructor being executed has been called by another constructor of the same actor. In this case, no specific action is taken, as the *AP*

```
1  onThreadInitialization() {
2      [TL] LONG bc = 0;
3      [TL] LONG bcNested = 0;
4      [TL] SS<2,A> ss_actor = createSS(2, A);
5
6      [TL] SS<1, LONG> ss_bcEntrance = createSS(1, LONG);
7  }
8
9  onBasicBlockEntrance() {
10     bc = bc + currentBBSize();
11 }
12
13
14 onActorConstructorEntrance() {
15     ss_actor = push(ss_actor, currentObject());
16
17     ss_bcEntrance = push(ss_bcEntrance, bc);
18 }
19
20
21 onActorConstructorExit() {
22     LONG initCount = ((bc − top(ss_bcEntrance,0)) − bcNested);
23
24     IF (top(ss_actor,1)=ϵ) {
25         createAP(top(ss_actor,0), initCount);
26         bcNested = 0;
27     }
28     ELSE {
29         IF (top(ss_actor,1)≠top(ss_actor,0)) {
30             createAP(top(ss_actor,0), initCount);
31             bcNested = bcNested + initCount;
32         }
33     }
34
35     ss_actor = pop(ss_actor);
36     ss_bcEntrance = pop(ss_bcEntrance);
37 }
```

(a) Platform-independent profiling.

```
1
2  @ThreadLocal static long bc = 0;
3  @ThreadLocal static long bcNested = 0;
4  @ThreadLocal static Object ss_actor_tl_0 = null;
5  @SyntheticLocal static Object ss_actor_sl;
6  @SyntheticLocal static long ss_bcEntrance_sl;
7
8  @Before(marker=BasicBlockMarker.class)
9  private static void onBasicBlockEntrance(final BasicBlockStaticContext bbsc) {
10   bc += bbsc.getBBSize();
11 }
12
13 @Before(marker=BodyMarker.class, guard=AkkaActorConstructor.class)
14 public static void onActorConstructorEntrance(final DynamicContext dc) {
15   ss_actor_sl = ss_actor_tl_0;
16   ss_actor_tl_0 = dc.getThis();
17   ss_bcEntrance_sl = bc;
18 }
19
20 @After(marker=BodyMarker.class, guard=AkkaActorConstructor.class)
21 public static void onActorConstructorExit() {
22   long initCount = (bc − ss_bcEntrance_sl) − bcNested;
23
24   if (ss_actor_sl==null) {
25     Profiler.createAP(ss_actor_tl_0,initCount);
26     bcNested = 0;
27   }
28   else {
29     if (ss_actor_sl != ss_actor_tl_0) {
30       Profiler.createAP(ss_actor_tl_0,initCount);
31       bcNested += initCount;
32     }
33   }
34
35   ss_actor_tl_0 = ss_actor_sl;
36
37 }
```

(b) DiSL snippets for Akka profiling.

Figure 2. Profiling of actor initialization count.

```
1  onThreadInitialization()
2      [TL] INT openReceiveMethods = 0;
3  }
4
5  onReceiveMethodsEntrance() {
6      ss_bcEntrance = push(ss_bcEntrance, bc);
7      openReceiveMethods = openReceiveMethods + 1;
8  }
9
10
11 onReceiveMethodsExit() {
12     IF (openReceiveMethods=1) {
13
14         registerMessageReceived(getAP(currentObject()),
15                                 typeOf(currentMessage()), bc − top(ss_bcEntrance,0));
16     }
17     openReceiveMethods = openReceiveMethods − 1;
18     ss_bcEntrance = pop(ss_bcEntrance);
19 }
```

(a) Platform-independent profiling.

```
1
2  @ThreadLocal static int openReceiveMethods = 0;
3
4  @Before(marker=BodyMarker.class, guard=ScalaPartialFunctionReceive.class)
5  public static void onReceiveMethodsEntrance() {
6    ss_bcEntrance_sl = bc;
7    ++openReceiveMethods;
8  }
9
10 @After(marker=BodyMarker.class, guard=ScalaPartialFunctionReceive.class)
11 public static void onReceiveMethodsExit(final DynamicContext dc) {
12   if (openReceiveMethods==1) {
13     Object message = dc.getMethodArgumentValue(0, Object.class);
14     Profiler.registerMessageReceived(Profiler.getAPFromPF(dc.getThis()),
15                                      message.getClass(), bc − ss_bcEntrance_sl);
16   }
17   −−openReceiveMethods;
18
19 }
```

(b) DiSL snippets for Akka profiling.

Figure 3. Profiling of messages received and computation count. The figure must be read along with Figure 2.

for the actor will be created when the outmost constructor ends. On the other hand, the case of nested actor initializations occurs if *top(ss_actor,1)*$\neq\epsilon \wedge$ *top(ss_actor,1)*$\neq$*top(ss_actor,0)* (lines 28–33), which indicates that the constructor being executed has been called within the constructor of another actor, as the two actors on the stack are different. In this case, the profiler creates a new *AP* for the nested actor (line 30) and adds its initialization count to *bcNested*. Following this mechanism, the initialization count of the caller actor will exclude the bytecodes executed for the initialization of the nested actor. Finally, the condition *top(ss_actor,1)*$=\epsilon$ (line 24) indicates that the constructor being executed is not nested within any other constructor. Here, the profiler must create a new *AP* (line 25) and reset *bcNested* (line 26), such that the initialization count of subsequently created actors can be accounted correctly.

5.2 Computation Count and Messages

Profiling message exchange between actors requires knowledge on which methods an actor library offers to send and receive messages. Typically, actor libraries provide multiple methods for carrying out message exchange, each with a different underlying logic where the message sent/received is passed as argument (for example, Akka offers two different methods for sending messages). We provide a platform- and library-independent approach that instruments all methods available for sending and receiving messages. Our profiler assumes that a single message is sent or received at a time, a common approach in popular actor libraries.

We assume that in the dynamic extent of a receive method, receive methods for different actors will not be invoked. In practice, this assumption holds for all actor libraries we have inspected. Still, the message exchange may be implemented in a receive method that is called by other receive methods for the same actor. For example, in Akka *ask [?]* calls *tell [!]* to send a message. Our profiler has to detect such nested receive invocations to avoid double accounting.

Figure 3 illustrates our profiling logic for receiving messages. The thread-local variable *openReceiveMethods* (line 2) counts how many receive methods are on the call stack. Data on the message received is collected if the receive method being executed is the only one on the call stack (lines 12–16). Since we assume that calls to receive methods of other actors are not possible, our profiler employs an integer counter rather than a shadow stack, as keeping references to actors is not necessary, differently from initialization-count profiling. Nevertheless, the counter is updated similarly to a shadow stack, being incremented (resp. decremented) at the entrance (resp. exit) of method bodies (lines 7 and 17).

Profiling computation count leverages the shadow stack *ss_bcEntrance*, already used to track initialization count. Executed bytecodes can be tracked by simply subtracting the top of *ss_bcEntrance* from the current bytecode counter *bc* (line 14–15). If a new actor is created as response to a received message, the executed bytecodes are accounted to both the initialization count of the created actor and to the computation count of the receiving actor. This behavior is desired, because it allows us to capture overall initialization count for all actors, as well as as overall computation count for handling all received messages. However, the sum of all actor initialization and computation counts may exceed the total number of executed bytecodes.

Profiling sent messages follows the same logic, with the difference that computation count is not tracked in this case, for two reasons. First, sending of messages is library code, it is not handled by user-defined functions. Second, the sending primitives are typically invoked within the user-defined code that handles message reception, for which we already calculate computation count. Thus, it is not necessary to use *ss_bcEntrance*. Profiling of sent messages uses a separate thread-local counter (i.e., *openSendMethods*) and updates metrics via *registerMessageSent*. We do not show the code due to the lack of space.

6. Concrete Implementation: Akka Profiling

Here, we show how our platform-independent profiling technique can be applied to a concrete virtual execution environment. We have implemented our approach within a profiling tool for Akka [2], based on the DiSL bytecode instrumentation framework [26]. We chose Akka, because it is one of the most used actor libraries for the JVM. We now discuss how our instrumentation scheme can be translated to DiSL code.

6.1 Shadow-Stack Translation

In Java, thread-local shadow stacks can be efficiently embedded within the frames of the call stack and in thread-local variables. Our translation avoids heap allocations and leverages synthetic-local and thread-local variables offered by DiSL.

Table 3. Shadow stack translations from platform-independent (PI) profiling to DiSL code.

PI	DiSL
ss = createSS(n,T)	*@ThreadLocal T ss_tl_0 = ϵ;* *...* *@ThreadLocal T ss_tl_<n-2> = ϵ;* *@SyntheticLocal T ss_sl;*
push(ss, x)	*ss_sl = ss_tl_<n-2>;* *ss_tl_<n-2> = ss_tl_<n-3>;* *...* *ss_tl_0 = x;*
top(ss, j)	*if (j == n-1) return ss_sl* *else return ss_tl_<j>;*
pop(ss)	*ss_tl_0 = ss_tl_1;* *...* *ss_tl_<n-2> = ss_sl;*

Table 3 reports the general scheme to translate operations on shadow stacks to DiSL code, assuming that the preconditions of the operations are met. A shadow stack *SS<n,T> ss* can be implemented with *n-1* thread-local variables and one synthetic local variable. For $n \geq 2$, the thread-local variable *ss_tl_<i>* provides access to the element *top(ss, i)* ($i = [0, n-2]$). The element *top(ss, n-1)* is stored in the synthetic local variable *ss_sl*. Regardless of the size of the shadow stack, only *n-1* elements are stored in thread-local variables. All other elements are embedded within the frames of the Java call stack, thanks to the synthetic local variable. All thread-local variables are initialized to the empty element ϵ.[2]

Manipulation of the shadow stack can occur only on method entrance and exit, as first (resp. last) instructions executed in the method. At the beginning of each method that manipulates the stack, a push occurs by copying *ss_tl_<n-2>* into *ss_sl*, then copying the value of *ss_tl_<i>* into *ss_tl_<i+1>* (*i*=[*0,n-3*]), and finally assigning the pushed element to *ss_tl_0*. When the method ends, it must undo the *push* operation, by first copying *ss_tl_<i+1>* into *ss_tl_<i>* (*i*=[*0,n-3*]), then copying *ss_sl* into *ss_tl_<n-2>*. For *n=2*, a single thread-local variable suffices, while for *n=1*, no thread-local variable is needed.

For *n>2*, an alternative translation of shadow stacks would store variables *ss_tl_<i>* (*i*=[*0,n-2*]) in a single thread-local array *ss_tl*. The array serves as a ring buffer where the top *n-1* elements are accessible, while elements falling out from it due to *push* operations are saved into synthetic local variables. On method exit, fallen-out elements are restored into the array. This solution would avoid *n* data copying operations at each *push* and *pop*, at the cost of accessing an array on the heap, and is likely to be more efficient for large values of *n*. We do not show this translation here, because it is not needed in the case of our profiler ($n \leq 2$).

6.2 Instrumentation for Akka Profiling

Figures 2(b) and 3(b) show DiSL snippets that implement our instrumentation for Akka on the JVM. As illustrated in Figure 2(b), the shadow stacks *ss_actor* and *ss_bcEntrance* are

[2] For this use case, we consider ϵ=*null* for *T=Object*, and ϵ=0 for *T=long*. Note that these two values cannot be pushed on the respective shadow stack.

translated according to the aforementioned scheme[3]. The Java annotations specify where the snippets shall be woven (e.g., before a basic block or before/after a method body), while guards specify into which methods snippets shall be woven. For example, the guard *AkkaActorConstructor* specifies that the snippet must be applied only to methods called *<init>* of each class being subtype of *akka.actor.Actor*.

Static and dynamic context classes (i.e., *DynamicContext* and *BasicBlockStaticContext*) expose static and dynamic context information to DiSL snippets. These classes are used within snippets to obtain the size of basic blocks, the receiver object, and the message being received (passed as a method argument). While markers and context classes are already provided by DiSL, we have implemented guards specific to the Akka instrumentation.

In the snippets, we use the class *Profiler* to store and handle APs associated to actors. Operations on the APs are translated as static methods in *Profiler*. Actor instances are mapped to AP instances through a special thread-safe, indentity-based hash map that does not prevent actor instances from being reclaimed by the garbage collector.

In Akka applications, the receive function of actors must be implemented through a special object (of type *scala.runtime.AbstractPartialFunction*) univocally associated with an actor during its creation. Thus, the DiSL instrumentation in Figure 3(b) must be applied to such classes, and not to actors. This is specified by the guard *ScalaPartialFunctionReceive*; the method *Profiler.getAPFromPF* is in charge of looking up the AP of the actor associated with the *AbstractPartialFunction* being instrumented.

7. Evaluation

Here, we show how our approach supports performance analysis of actor applications. We apply our Akka profiler to benchmarks and real-world applications. We start by comparing actor initialization and computation count in Savina [24], an actor-based benchmark suite (Section 7.1). Then, we analyze the communication occurred between actors in two well-known computing frameworks, Apache Spark [35] and Apache Flink [5] (Section 7.2). Finally, we briefly discuss the runtime overhead of our profiler (Section 7.3).[4]

7.1 Actor Utilization

Initialization and computation count help better understand how actors are used. This is particularly important in applications relying on actors to carry out computational work. In the following text, we analyze such metrics in Savina [24], an actor-based suite composed of 30 benchmarks implemented in 10 different actor libraries for the JVM—including Akka—that rely solely on actors to obtain concurrency.

To provide a concise metric that enables one to compare initialization and computation count, we introduce a new metric, the *utilization* of each actor, defined as the ratio of computation count and initialization count. Small utilization values are symptoms of bad performance, as the system spends more resources in creating actors rather than executing computations. On the other hand, large utilization values pinpoint that adding more computing actors may speed up the application.

Table 4 summarizes statistics on messages, actors, and their utilization in all the benchmarks utilizing Akka. The right part of the table shows, for each benchmark, the average utilization among all actors, along with the standard deviation and the 20^{th}, 50^{th}, and 80^{th} percentiles of the distribution.[5] As can be seen from the table, benchmarks in the suite vary a lot in terms of the number of actors spawned, messages processed, and actor utilization, while the number of types for actors and message is rather low for all benchmarks. Utilization can also vary a lot among actors of the same benchmark. For example, utilization in recmatmul shows a very high standard deviation, ranging from poorly utilized actors to extremely utilized ones.

Focusing on low values of utilization $u < 10$, Table 4 shows that bitonicsort and fjcreate utilize actors rather scarcely on average. Analyzing the percentiles of the utilization distribution, one can see that at least the 20% of the actors are little utilized in 9 benchmarks, while the percentage raises to at least 50% and 80% in 5 and 3 benchmarks, respectively. Moreover, benchmarks where the percentage of little utilized actors is higher (i.e., fjcreate, bitonicsort, and barber) also leverage a high number of actors, a sign that an excessive amount of actors could be spawned given the computations to be carried out. Finally, we highlight that the number of messages processed in bitonicsort—where most of the actor are little utilized—is very high, i.e., more than 2.5M. This is an indication that single messages trigger the execution of (too) small computations.

Potential root causes of poor actor utilization are 1) the presence of too many actors wrt. the amount of computations to be carried out, and 2) unbalanced division of computations among actors. As corrective actions aimed at better utilizing actors in the system—assuming that the amount of computations cannot be changed—the developer may consider to remove some actors, in the first case. Such an action is aimed at avoiding the initialization of actors that would be utilized scarcely. In the latter case, developers could redesign the division of computations to actors, where possible, to balance them more evenly. This is the action we suggest in bitonicsort, where 15892 actors are little utilized due to bad design rather than due to little computations to be carried out (manually verified).

On the other hand, focusing on actors with a high utilization (u>100000), the table shows a high average utilization

[3] To avoid casts, we store *Object* references instead of *Actor* references in *ss_actor_tl_0* and *ss_actor_sl*.

[4] The measurements were collected on a multicore platform (Intel Xeon E5-2680, 2.7 GHz, 16 cores, 128 GB RAM, CPU frequency scaling and Turbo mode disabled) running Oracle JDK 1.8.0_66 b17 Hotspot Server VM 64-bit on Ubuntu Linux Server 64-bit version 14.04.3 64-bit. In the evaluation of Spark and Flink (see Section 7.2), the master and each worker are deployed each on separate machines with the aforementioned technical specification. The evaluation is run using DiSL 2.1, Spark 1.5.2, and Flink 1.0.

[5] The reported results refer to a standard benchmark run with default parameters for all benchmarks. We excluded benchmark sor due to a bug in the benchmark that impedes program completion.

Table 4. Actor utilization for the Savina benchmark suite [24].

Benchmark	Actors		Messages		Utilization				
	#	# types	#	# types	AVG	STD	20^{th} perc.	50^{th} perc.	80^{th} perc.
apsp	40	4	108291	8	626	178	679	680	682
astar	25	5	1126	9	85659	46159	33412	102194	112408
banking	1005	5	208231	9	170	48	128	163	211
barber	5007	7	41474	10	304	14844	4	4	4
big	125	5	4800605	8	39076	18946	6424	48916	49078
bitonicsort	190525	16	2674789	8	12	127	6	6	7
bndbuffer	85	6	160204	10	700944	222883	757762	769162	783645
chameneos	105	5	800404	9	10331	36335	6977	7047	7103
cigsmok	205	5	2405	8	1549	887	815	1380	2237
concdict	26	6	400066	10	8920	4924	427	11539	11559
concsll	26	6	320066	11	53231	234589	6466	8988	9014
count	6	5	1000008	7	150864	292090	0	315	341271
facloc	1370	5	743792	9	253	6314	2	4	21
fib	150052	4	450149	6	285	915	4	22	289
filterbank	66	14	1419465	11	20819	114765	5	580	3784
fjcreate	40004	4	80003	5	3	3	3	3	3
fjthrput	64	4	600063	5	2057	498	2182	2182	2184
logmap	25	6	992878	11	18066	19199	408	5640	40936
nqueenk	25	5	29140	9	1060159	542017	615780	1303146	1368435
philosopher	25	5	1198130	10	24101	11740	16278	30055	30398
pingpong	6	5	120006	10	28394	45128	0	321	77835
piprecision	25	5	8673	9	1858180	949326	1105397	2309469	2358476
quicksort	2007	4	6011	6	10829	31468	2256	5397	15204
radixsort	66	6	6100066	6	65701	19222	70194	70669	70980
recmatmul	25	5	1818	8	4969990	10166347	4	5	11649055
sieve	15	5	91343	8	145413	152496	315	96522	303587
threadring	104	4	100304	7	1595	297	1639	1641	1660
trapezoid	105	5	305	6	30925	6921	32375	32376	32690
uct	199977	5	879898	13	572591	95530	491944	573138	651467

Table 5. Messages sent by actors in Apache Spark [35] and Apache Flink [5]. Values reported are the sum of the messages sent by the master and all workers.

No application			pi			kmeans			pagerank	
Uptime [s]	Spark	Flink	# tasks	Spark	Flink	# points	Spark	Flink	Spark	Flink
150	7907	4663	250	8137	8880	10k	8316	235313	74068	129407
300	18153	13661	500	8297	9113	100k	8373	172638		
450	28449	22962	750	8468	9517	1M	8992	194050		
600	38605	31906	1000	8303	9020	10M	10630	209950		

in 7 benchmarks, i.e., recmatmul, count, sieve, bndbuffer, nqueenk, piprecision, and uct. Depending on the environment settings and the resources available in the system, utilization could be reduced by adding more actors to better utilize system resources. For example, count spawns only 6 actors, 2 of which being highly utilized. In platforms where more cores are available, splitting computations further by adding actors might yield speedups, as additional actors could be executed by idle cores without contention. In contrast, such an action may not benefit other benchmarks showing high actor utilization, as they already employ more actors than available cores.

Finally, we note that our analysis on actor utilization shows that some benchmarks in the suite are little representative of realistic workloads. The high number of actors spawned for computing simple algorithms such as in fib, and the excessively fine-grained granularity of actors utilized by benchmarks such as fjcreate or bitonicsort make the benchmarks arguably not representative of actor-based applications.

Overall, our profiler is a helpful tool in the analysis of applications where concurrency is based on Akka actors, enabling the identification of little utilized and highly utilized actors thanks to dedicated metrics.

7.2 Communication

In this section, we show how our tool can guide the analysis of the communication between endpoints in the Apache Spark [35] and Apache Flink [5] frameworks. Both are widely adopted for big-data processing, graph computations, and stream processing, and are designed around a master/slave architecture, where a single master orchestrates and manages several workers. Akka actors are utilized as endpoints that handle communication between each computing entity.

We evaluate communication occurred between actors under three applications executed on both frameworks, pi, kmeans, and pagerank. While the first application is a useful benchmark as it enables a fine-grained control on the number of tasks executed by workers, kmeans and pagerank allow us to evaluate the frameworks during a machine-learning and a graph-processing application, respectively. We complement the evaluation by looking at the communication only due to monitoring messages, without any application being executed. Finally, we investigate communication wrt. different input data sizes. Table 5 reports the result of our analysis.[6] From the table, one can see that Spark exchanges more messages than Flink when no application is executed. However, the situation is opposite when applications are executed in the frameworks. This holds for all three applications considered here. While the difference between the frameworks is minor in pi, Flink sends 1.75x more messages than Spark in pagerank

[6] To guarantee a fair comparison of the two frameworks, both of them have been configured with the same settings, in terms of number of workers (i.e., 4), number of available cores, and amount of memory available to the cluster. Moreover, the input set and the parameters of the algorithms are the same in both frameworks. The input for kmeans is a set of randomly generated points, while the input for pagerank is taken from [15].

and, surprisingly, 23x more messages in kmeans on average, given the same operating conditions and the same computation to be performed.

When correlating the magnitude of the communication with the uptime of the system, the table shows a linear correlation for both frameworks, as expected. On the contrary, the relationship between the amount of messages sent and the number of tasks in pi is weakly correlated in both frameworks. For example, a pi computation over 750 tasks triggers more messages than the same computation over 1000 tasks. Finally, while an increasing data size in kmeans clearly generates an increasing number of messages in Spark, we cannot confirm such a trend in Flink, as the computation over 10k points generates 62675 more messages than a computation over 100k points.

In Figure 4 we show the kmeans execution times for both frameworks wrt. the size of the input data.[7] From the figure, we can see that Spark shows a significantly lower execution time than Flink, with differences as high as 7x in favor of the former, showing that less communication here results in better performance. Interestingly, the differences between the two frameworks becomes smaller with increasing data sizes. This indicates that Flink may not be well optimized for processing small amounts of data. Comparing the trends of execution time and messages sent, we can observe that the two metrics correlate very well in Spark (correlation coefficient $= 0.99$). In contrast, in Flink the correlation between the two metrics is low (correlation coefficient $= 0.18$).

Overall, actor communication profiling reveals that communication in Spark is more optimized than in Flink, for the given workload. Our results pinpoint that further investigations on the root causes of the substantial message exchange in Flink are needed, as well as on the correlation between tasks and messages in both frameworks.

7.3 Runtime Overhead

Our Akka profiler introduces moderate runtime overhead of 5% – 20% for long-running workloads (execution time $\geq$ 30s), when measuring the elapsed wall time between JVM startup and termination. For short-running workloads, the runtime overhead is up to a factor of 2, due to load-time instrumentation by DiSL. Because of space limitations, we do not present detailed performance measurements here.

8. Discussion

Here we discuss the strengths and limitations of our approach.

Benefits. Our goal has been to enable platform-independent actor profiling in virtual execution environments. Our approach is a step into this direction. We track high-level platform-independent metrics that make our approach applicable to several actor libraries on different virtual execution environments. Our profiling technique is based on bytecode instrumentation and does not resort to any platform-specific feature. As a consequence, it is fully portable. Moreover, it instruments every actor and tracks every message sent or received, resulting in exact profiles.

[7] Time reported in Figure 4 refers to the execution of the kmeans computation without any instrumentation. Time needed for transferring the input data and setting up the cluster is not included.

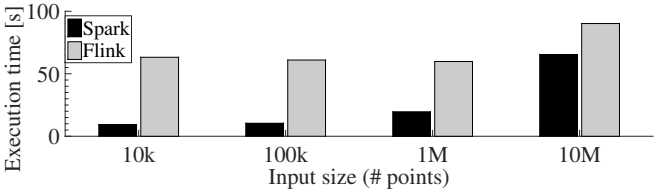

Figure 4. Execution time of a kmeans computation in Apache Spark and Apache Flink wrt. input data size.

In our profiling technique, bytecode count is a central metric. Apart from being platform-independent, this metric is little affected by perturbations caused by instrumentation, makes profiles reproducible and comparable (assuming a fully deterministic program), and ensures accurate results in presence of full bytecode coverage, in contrast to commonly profiled metrics such as CPU time. In addition, tracking messages sent and received can provide an overall view of the communication between actors, still with a platform-independent approach.

We have shown how the proposed technique can be applied to Akka actors on the Java platform. Our Akka profiler is able to track all computations executed by actors with platform-independent metrics, in contrast to sampling-based techniques (which may lack accuracy) or binary instrumentation (which lacks portability). Moreover, the profiler does not resort on any platform-specific mechanisms such as hardware performance counters.

The translations applied for Akka profiling are straightforward. Applying our approach to other actor libraries or virtual environments requires the ability to perform instrumentation with full bytecode coverage and some knowledge on the target library, in particular how actors are defined and which methods the library offers for sending and receiving messages.

Limitations. The bytecode-count metric suffers from some limitations. The metric cannot track any code which does not have a bytecode representation, such as native methods executed by actors, resulting in a potential underestimation of the initialization and computation counts. For programs that heavily depend on native code, dynamic bytecode metrics may not be relevant. Nonetheless, as constructors cannot be native, the profiler covers all constructor invocations, ensuring that the *AP* of each actor is initialized. VM-level activities such as garbage collection (GC) are also not tracked by our approach. As a result, actors can trigger significant GC execution, but the cost of GC will not be represented in the profiles.

The bytecode count represents computations of different complexity with the same unit. As a result, it may be little representative of the real amount of computations (e.g., machine instructions) executed by actors. In addition, the bytecode count may not well represent computations in cases where the just-in-time (JIT) compiler applies on-the-fly optimizations to the executed applications. For example, the JIT compiler may apply inlining optimizations that remove invocation instruc-

tions. In these cases, some bytecodes may not be executed but are still accounted by our approach, resulting in possibly misleading values of initialization and computation counts.

9. Related Work

In the following text, we discuss work related to the proposed technique. First, we compare our approach with other tools for profiling computations and communication. Next, we complement the discussion on the use case presented by positioning our technique wrt. other profiling tools for Akka.

Computation Profiling. Profiling the amount of computations executed by actors is similar to techniques for investigating the granularity of computing tasks. Typically, profiling tools for task granularity [21] represent computations as CPU time. However, CPU-time profiles are affected by perturbations arising from instrumentation code [28], are not reproducible, are not comparable between different environments, and often require support from the underlying platform to be profiled accurately. In contrast, we rely on bytecode count to measure computations, which provides accurate, portable, comparable, and platform-independent results. Overall, we are not aware of any other technique that provides accurate and portable platform-independent metrics to quantify initialization cost and computations executed by actors.

Tracking actor computations is also related to the investigation of parallelism in concurrent applications. Various profiling tools have been designed to analyze parallelism. Cilkview [22] studies the execution time of different computing paths, while Harmony [25] collects parallel block vectors, which link the execution of basic blocks to the number of running threads. Other metrics dedicated to parallelism are bottle graphs [19], which relate per-thread computations with execution time. Overall, all these tools and metrics focus on threads as the main computing entities. While actors are often dispatched and executed by threads, our approach centers on actors, making our technique orthogonal to these tools.

Communication Profiling. Although there are many profilers for parallel and distributed systems, profiling communication at the level of messages exchanged is, as far as we know, rather overlooked. A notable exception is the work conducted by Vetter [33], which proposes a technique for profiling communication activities inside an application using message sampling. Unfortunately, sampling can guarantee only approximated results. Moreover, this technique focuses on message latency as main metric, which suffers from similar drawbacks as profiling CPU time. Instead, our approach targets platform-independent metrics such as the amount of messages sent and received, which can be tracked with no perturbations. Moreover, as we rely on comprehensive bytecode instrumentation, our profiles are accurate, unlike those resulting from sampling.

Focusing on distributed systems, several tools provide network profiling, such as Magpie [16], SNAP [34] and X-Trace [20]. However, their main goal is to track metrics and information at the level of the network stack, being little helpful when investigating message passing between actors. Moreover, such metrics are inherently platform-dependent and require dedicated support from the underlying operating system to be tracked, unlike our approach. Finally, message exchange is rarely considered by distributed monitoring systems [27] (which focus mainly on CPU, memory, and load monitoring) and tracing infrastructures such as Dapper [30] or Zipkin [12] (which mainly target application latency and system errors).

Akka Profiling. Our profiling tool for Akka enables the collection of the initialization and computation count for Akka actors. To the best of our knowledge, no other profiler for Akka focuses on such metrics. Lightbend Monitoring [9] includes metrics specific to Akka actors, such as the number of actors running in the system, statistics on mailbox size, time in mailbox, and time to process messages. Our case study provides a tool complementary to Lightbend Monitoring, as it focuses on metrics that are not provided by the latter.

Another profiling tool for Akka is Akka Tracing [11], which focuses especially on slow requests, debugging, and message latency in distributed systems, while Kamon [8] targets message processing time, mailbox size, time-in-mailbox per message, and errors. Plugins with similar focus are integrated into other profiling tools such as AppDynamics [3] and NewRelic [10]. In general, these tools pay little attention on computations and communications. Moreover, some of these profilers, e.g., Kamon, are based on instrumentation frameworks that cannot guarantee full bytecode coverage, unlike DiSL. Thus, extending such tools to include metrics on executed computations is likely to result in inaccurate results.

10. Conclusions

In this paper, we have presented a novel technique to profile actors in virtual execution environments. Our technique is based on bytecode instrumentation, and reconciles portability and accuracy to provide high-level platform-independent metrics on the initialization cost and on the computations executed by actors, as well as on the communication between them. From the proposed approach, we have derived a profiling tool for Akka actors on the Java platform. Evaluation results show that our approach helps the performance analysis of applications using actors. In particular, with the support of the derived tool, we have located inefficiencies in actor utilization and communication in Savina and Apache Flink, and we have pinpointed possible improvements. We plan to make a public release of the derived Akka profiler in the near future.

As part of our future work, we plan to complement our Akka profiler with platform-specific metrics such as the number of executed machine instructions and network traffic, and to enable the reconstruction of the message flow between actors. With the support of these new metrics, we plan to deepen the performance evaluation of Apache Flink, investigating the root causes of inefficient communication and higher execution time wrt. Apache Spark, the presence of unhanded messages, and the reception of messages that trigger no work.

References

[1] Akka Actor Corpus. http://actor-applications.cs.illinois.edu/akka.html.

[2] Akka. http://akka.io.

[3] AppDynamics. https://www.appdynamics.com/java/akka/.

[4] Elixir. http://elixir-lang.org.

[5] Apache Flink. https://flink.apache.org.

[6] GPars. http://gparsdocs.de.a9sapp.eu.

[7] Jetlang. https://github.com/jetlang/.

[8] Kamon. http://kamon.io/integrations/akka/actor-router-and-dispatcher-metrics/.

[9] Lightbend Monitoring. https://www.lightbend.com/products/monitoring/.

[10] NewRelic. https://newrelic.com.

[11] Akka Tracing. https://github.com/levkhomich/akka-tracing/.

[12] Zipkin. http://zipkin.io.

[13] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

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

[15] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group Formation in Large Social Networks: Membership, Growth, and Evolution. In *KDD*, pages 44–54, 2006.

[16] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modelling. In *OSDI*, pages 259–272, 2004.

[17] W. Binder, J. G. Hulaas, and A. Villazón. Portable Resource Control in Java. In *OOPSLA*, pages 139–155, 2001.

[18] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-Oriented Programming in Ambienttalk. In *ECOOP*, pages 230–254, 2006.

[19] K. Du Bois, J. B. Sartor, S. Eyerman, and L. Eeckhout. Bottle Graphs: Visualizing Scalability Bottlenecks in Multi-threaded Applications. In *OOPSLA*, pages 355–372, 2013.

[20] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A Pervasive Network Tracing Framework. In *NSDI*, pages 271–284, 2007.

[21] K. Hammond, H.-W. Loidl, and A. S. Partridge. Visualising granularity in parallel programs: A graphical winnowing system for Haskell. In *HPFC*, volume 95, pages 208–221, 1995.

[22] Y. He, C. E. Leiserson, and W. M. Leiserson. The Cilkview Scalability Analyzer. In *SPAA*, pages 145–156, 2010.

[23] C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI*, pages 235–245, 1973.

[24] S. M. Imam and V. Sarkar. Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries. In *AGERE!*, pages 67–80, 2014.

[25] M. Kambadur, K. Tang, and M. A. Kim. Harmony: Collection and Analysis of Parallel Block Vectors. In *ISCA*, pages 452–463, 2012.

[26] L. Marek, A. Villazón, Y. Zheng, D. Ansaloni, W. Binder, and Z. Qi. DiSL: A Domain-specific Language for Bytecode Instrumentation. In *AOSD*, pages 239–250, 2012.

[27] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817 – 840, 2004.

[28] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Evaluating the Accuracy of Java Profilers. In *PLDI*, pages 187–197, 2010.

[29] A. Rosà, L. Y. Chen, and W. Binder. Efficient Profiling of Actor-based Applications in Parallel and Distributed Systems. In *ICOOOLPS (to appear). Position paper.* URL: http://www.inf.usi.ch/phd/rosaa/icooolps16.pdf, 2016.

[30] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Technical report, Google, Inc., 2010. URL http://research.google.com/archive/papers/dapper-2010-1.pdf.

[31] P. Stutz, A. Bernstein, and W. Cohen. Signal/Collect: Graph Algorithms for the (Semantic) Web. In *ISWC*, pages 764–780, 2010.

[32] S. Tasharofi, P. Dinges, and R. E. Johnson. Why Do Scala Developers Mix the Actor Model with Other Concurrency Models? In *ECOOP*, pages 302–326, 2013.

[33] J. Vetter. Dynamic Statistical Profiling of Communication Activity in Distributed Applications. In *SIGMETRICS*, pages 240–250, 2002.

[34] M. Yu, A. Greenberg, D. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim. Profiling Network Performance for Multi-tier Data Center Applications. In *NSDI*, pages 57–70, 2011.

[35] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *NSDI*, pages 2:1–2:14, 2012.