Università
della
Svizzera
italiana

**Faculty
of Informatics**

Actor Profiling in Virtual Execution Environments

Andrea Rosà*, Lydia Y. Chenˆ, and Walter Binder*

*Università della Svizzera italiana (USI), Faculty of Informatics, Lugano, Switzerland
ˆIBM Research Lab Zurich, Rüschlikon, Switzerland

GPCE 2016
October 31st, 2016
Amsterdam, The Netherlands

Overview

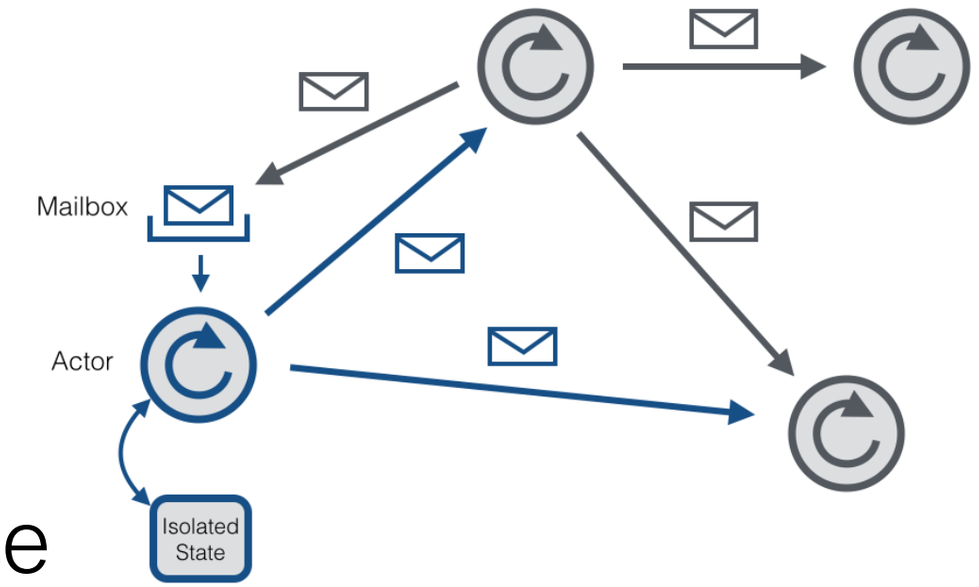
- In this talk, we present a new technique for **actor profiling** in **virtual execution environments**

- Other contributions:
  - Profiling tool for Akka actors
    - Centered on computations executed by actors
  - Evaluation of actor performance in Akka applications

Background

Actors

- Atomic concurrent entities
  - Cannot share state
  - Can communicate only via asynchronous messages
  - Execute computations in response to a message
  - Message type dictates executed computation



Actors in practice

- Many implementations for Java, C++, Python, .NET, Haskell, …
- On the JVM: Akka is the most used one
- Main applications:
  - Computing workers (e.g., Signal/Collect)
  - Communication endpoints (e.g., Apache Spark, Apache Flink)
- Existing general-purpose profilers cannot recognize actors
- Existing actor profilers do not measure computations

Actor profiling

Design goals

Portability Accuracy

- **Bytecode instrumentation**
  - Can be applied to many virtual execution environments
  - Allows computing bytecode-level metrics

Design goals

Portability Accuracy

- Bytecode instrumentation
- Bytecode count
  - Platform-independent
  - Little affected from perturbations inserted by instrumentation code
  - Makes profiles reproducible and comparable across different environments
  - Requires full bytecode coverage

Design goals

Portability Accuracy

- Bytecode instrumentation
- Bytecode count
- Platform-independent metrics
- High-level
- Applicable to many actor-based programs
- Not specific to given platforms or libraries

Design goals

Portability Accuracy

- Bytecode instrumentation
- Bytecode count
- Platform-independent metrics

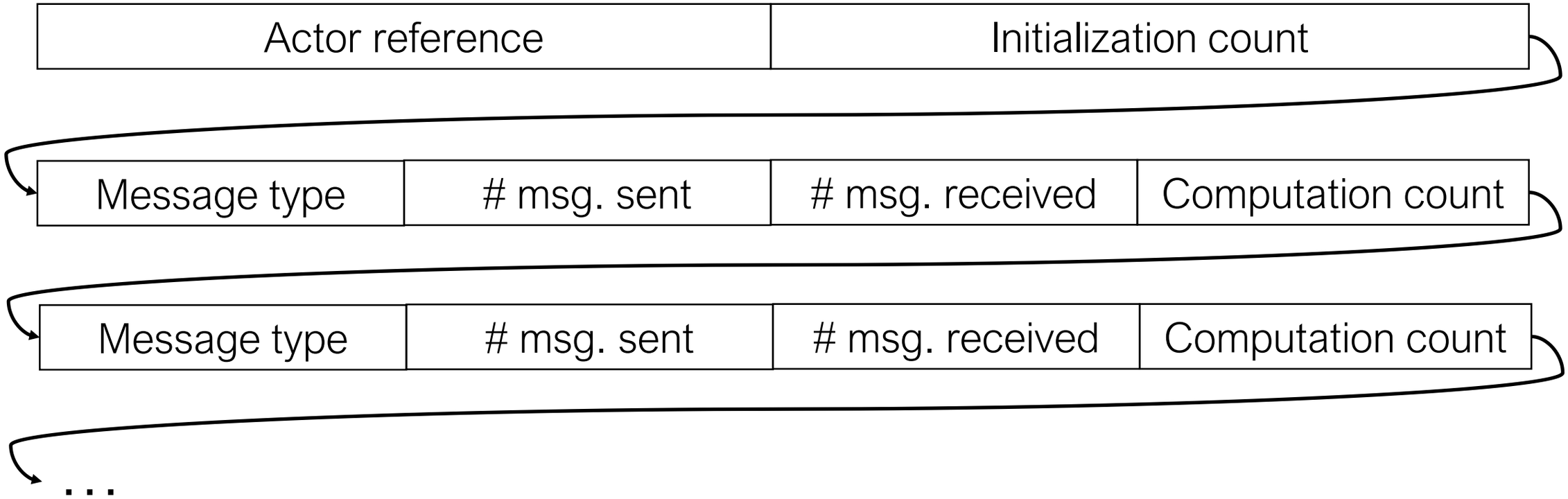
- Initialization count
- Computation count

} Expressed as bytecode count

- Messages sent
- Messages received

Profiling Data Structures

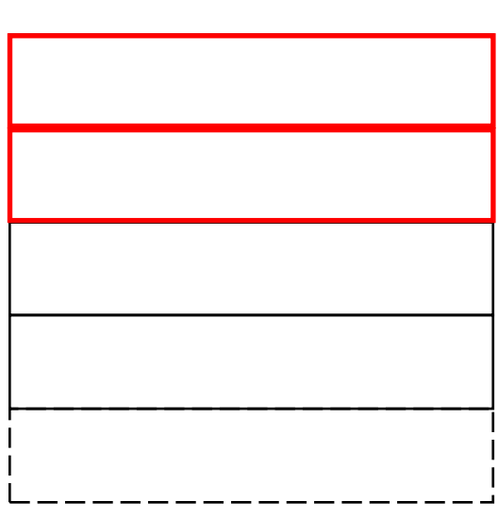
- **Actor Profile (AP)**
  - Stores information related to one actor
  - Created along with actor creation



Profiling Data Structures

- **Shadow Stack (SS)**
  - Stores context information related to executed methods
  - Provides information to subsequent callees
    - `createSS(INT n, TYPE T): SS<n,T>`

Number of inspectable top elements



- `createSS(2, INT): s`
- `top(s,0)`
- `top(s,1)`
- ~~`top(s,2)`~~

- At most one `push` is allowed (at method entrance)
- `push` must be followed by one `pop` (at method exit)

Initialization Count Profiling

- Challenges:
  - Multiple constructors
    - Instrumentation must determine end of actor initialization
  - Nested actor creation
    - Instrumentation must separate initialization counts of actors

```
actor A {...}
actor B {
    B() {...}
}
actor C subtype of B {
    C() {
        B();
        A a = new A();
        ...
    }
}
```

- Shadow stacks help detect such situations

Initialization Count Profiling

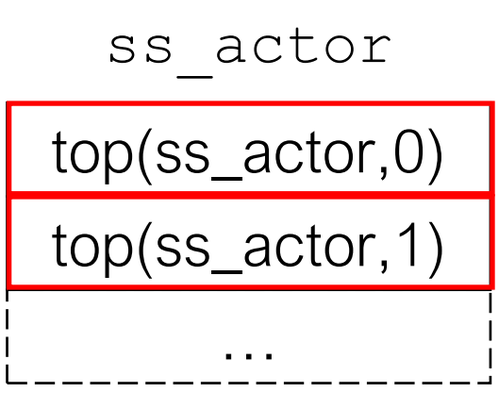
```
onThreadInitialization() {
    [TL] LONG bc = 0;
    [TL] LONG bcNested = 0;
    [TL] SS<2, ACTOR> ss_actor = createSS(2, ACTOR);
    [TL] SS<1, LONG> ss_bcEntrance = createSS(1, LONG);
}

onBasicBlockEntrance() {
    bc = bc + currentBBSize();
}

onActorConstructorEntrance() {
    ss_actor = push(ss_actor, currentObject());
    ss_bcEntrance = push(ss_bcEntrance, bc);
}
```

- `bc`: tracks bytecodes executed by thread over its execution
- `bcNested`: tracks initialization count of nested actors
- `ss_actor`: stores actors under initialization
- `ss_bcEntrance`: stores value of `bc` upon constructor entrance

Initialization Count Profiling

value of `bc` at constructor entrance

```
onActorConstructorExit() {
    LONG initCount = ((bc - top(ss_bcEntrance,0)) - bcNested);
    IF (top(ss_actor,1) = ε) {
          createAP(top(ss_actor,0), initCount);
          bcNested = 0;
    } ELSE {
             IF (top(ss_actor,1) != top(ss_actor,0)) {
                  createAP(top(ss_actor,0), initCount);
                  bcNested = bcNested + initCount;
             }
    }
    ss_actor = pop(ss_actor);
    ss_bcEntrance = pop(ss_bcEntrance);
}
```



- Receiver of the constructor being currently executed
- Receiver of the outer (open) constructor (if any)

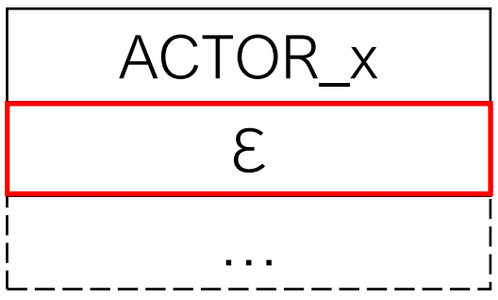
- Size and content of `ss_actor` signals multiple constructor or nested actor creation

Initialization Count Profiling

```
onActorConstructorExit() {
    LONG initCount = ((bc - top(ss_bcEntrance,0)) - bcNested);
    IF (top(ss_actor,1) = ε) {
            createAP(top(ss_actor,0), initCount);
            bcNested = 0;
    } ELSE {
                IF (top(ss_actor,1) != top(ss_actor,0)) {
                    createAP(top(ss_actor,0), initCount);
                    bcNested = bcNested + initCount;
                }
    }
    ss_actor = pop(ss_actor);
    ss_bcEntrance = pop(ss_bcEntrance);
}
```

No multiple constructors nor nested actor creations

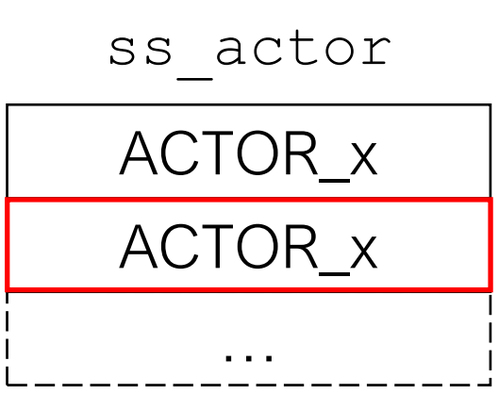
ss_actor



No other actor is being created

Initialization Count Profiling

```
onActorConstructorExit() {
    LONG initCount = ((bc - top(ss_bcEntrance,0)) - bcNested);
    IF (top(ss_actor,1) = ε) {
            createAP(top(ss_actor,0), initCount);
            bcNested = 0;
    } ELSE {
                IF (top(ss_actor,1) != top(ss_actor,0)) {
                    createAP(top(ss_actor,0), initCount);
                    bcNested = bcNested + initCount;
                }
    }
    ss_actor = pop(ss_actor);
    ss_bcEntrance = pop(ss_bcEntrance);
}
```



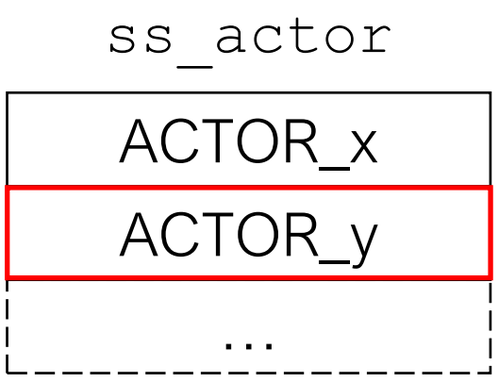
Multiple constructors detected

Different constructor, but same actor

Initialization Count Profiling

```
onActorConstructorExit() {
    LONG initCount = ((bc - top(ss_bcEntrance,0)) - bcNested);
    IF (top(ss_actor,1) = ε) {
            createAP(top(ss_actor,0), initCount);
            bcNested = 0;
    } ELSE {
                IF (top(ss_actor,1) != top(ss_actor,0)) {
                    createAP(top(ss_actor,0), initCount);
                    bcNested = bcNested + initCount;
                }
    }
    ss_actor = pop(ss_actor);
    ss_bcEntrance = pop(ss_bcEntrance);
}
```

Nested actor creation detected



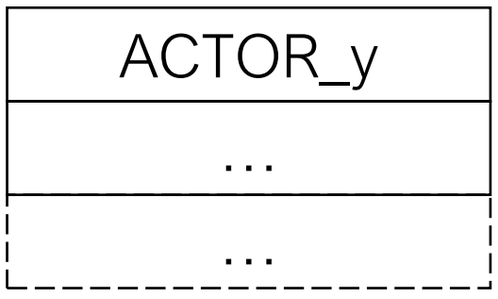
ACTOR_x is being initialized in the constructor of ACTOR_y

Initialization Count Profiling

```
onActorConstructorExit() {
    LONG initCount = ((bc - top(ss_bcEntrance,0)) - bcNested);
    IF (top(ss_actor,1) = ε) {
            createAP(top(ss_actor,0), initCount);
            bcNested = 0;
    } ELSE {
                IF (top(ss_actor,1) != top(ss_actor,0)) {
                        createAP(top(ss_actor,0), initCount);
                        bcNested = bcNested + initCount;
                }
    }
    ss_actor = pop(ss_actor);
    ss_bcEntrance = pop(ss_bcEntrance);
}
```

Subtracts `initCount` of ACTOR_x

ss_actor



Computation Count / Messages

- Similar to initialization count profiling, but simpler
  - Requires at most a single shadow stack
- Requires knowledge on sending/receiving methods

Akka Profiling

- Based on DiSL [1]
  - Framework for Java bytecode instrumentation
  - Based on AOP (*snippets*)
  - Guarantees full bytecode coverage
  - Translation of platform-independent profiling code to DiSL snippets

[1] L. Marek et al., *DiSL: A Domain-specific Language for Bytecode Instrumentation*. AOSD'12

Akka Profiling

- Shadow-stack translation

Platform-independent	DiSL (n=1)	DiSL (n=2)
`ss = createSS(n,T)`	`@SyntheticLocal T ss_sl;`	`@ThreadLocal T ss_tl = ε;` `@SyntheticLocal T ss_sl;`
`push(ss, x)`	`ss_sl = x;`	`ss_sl = ss_tl;` `ss_tl = x;`

`top(ss, j)`	`if (j != 0) return ERROR`
	`else return ss_sl;`
		`if (j > 1) return ERROR`
		`elseif (j==1) return ss_sl;`
		`else return ss_tl;`

`pop(ss)`	`no action`	`ss_tl = ss_sl;`

- n-1 entries are stored in thread-local variables
- Other entries are embedded within frames of call stack
- Avoids heap allocation

Akka Profiling

```
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  }
```

```
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  }
```

Evaluation

Evaluation

- Computing workers:
  - Savina benchmark suite [2]
  - Analyze utilization of actors
    - Utilization = computation count / initialization count
- Communication endpoints:
  - Apache Spark [3] and Apache Flink [4]
  - Compare communication between workers

---

[2] S. M. Imam et al., *Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries.* AGERE!'14

[3] M. Zaharia et al., *Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing*. NSDI'12

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

Savina

Low utilization (U < 10)

Benchmark	Actors		Messages		Utilization				
	#	# types	#	# types	AVG	STD	20th perc.	50th perc.	80th perc.
barber	5007	7	41474	10	304	14844	4	4	4
bitonicsort	190525	16	2674789	8	12	127	6	6	7
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
pingpong	6	5	120006	10	28394	45128	0	321	77835
recmatmul	25	5	1818	8	4969990	10166347	4	5	11649055

- 20% of actors are little utilized in 9 benchmarks
- 50% of actors are little utilized in 5 benchmarks
- 80% of actors are little utilized in 3 benchmarks
- Number of actors spawned is high
- Possible optimization: remove some actors

Spark / Flink

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		

- Flink sends more messages than Spark under a workload
  - kmeans: up to 23x more messages
- Applications run faster in Spark
  - Up to 7x faster

Conclusions and discussion

Conclusions

- We have presented a new technique for actor profiling in virtual execution environments
- We have derived a profiling tool for Akka actors
- We have evaluated actor utilization and communication in Akka applications

Discussion

- Limitation of bytecode count:
  - Cannot track code without bytecode representation (e.g., native methods)
    - Cannot track VM activities (e.g., garbage collection)
  - Bytecodes of different complexity are represented with the same unit
- Susceptible to dynamic optimizations

Discussion

- Future work:
  - Machine instruction count
    - Platform-specific
    - Perturbed by instrumentation
  - Network traffic
    - Platform-specific
  - Message flow between actors
- Expand analysis on use cases:
  - Flink: root causes of inefficient communication?

Thank you for the attention

- Contact details:

Andrea Rosà
andrea.rosa@usi.ch
http://www.inf.usi.ch/phd/rosaa