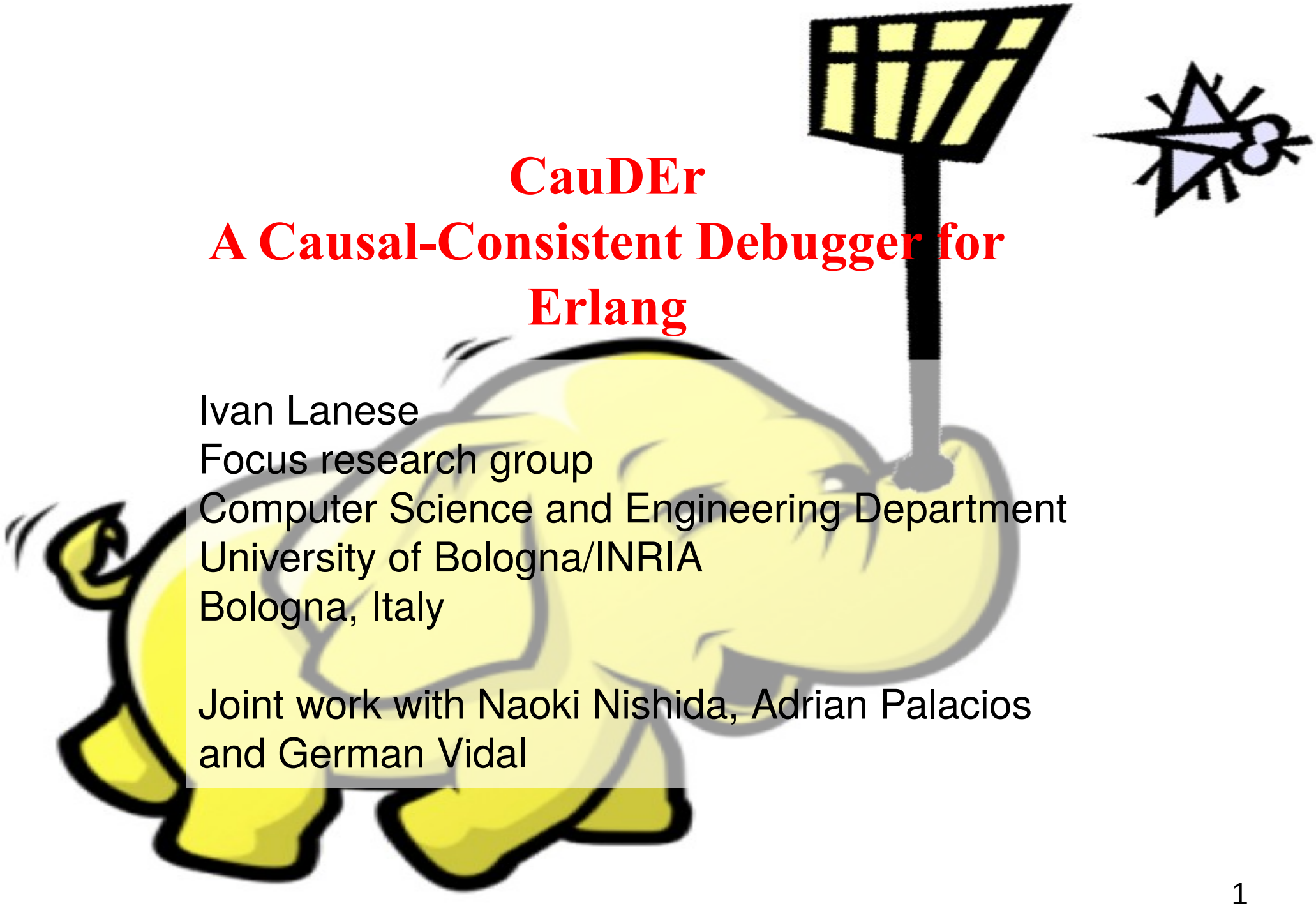

CauDEr
A Causal-Consistent Debugger for Erlang

Ivan Lanese
Focus research group
Computer Science and Engineering Department
University of Bologna/INRIA
Bologna, Italy

Joint work with Naoki Nishida, Adrian Palacios and German Vidal

Roadmap

- Debugging
- Causal-consistent reversible debugging
- CauDEr
- Future directions

Why debugging?

- Developers spend 50% of their programming time finding and fixing bugs
- The global cost of debugging has been estimated in $312 billions annually
- The cost of debugging is bound to increase with the increasing complexity of software
  - Size
  - Concurrency, distribution
- Surprisingly, very little research on debugging
  - Compare, e.g., to research on model checking

Standard debugging strategy

- When a failure occurs, one has to re-execute the program with a breakpoint before the expected bug
- Then one executes step-by-step forward from the breakpoint, till the bug is found
- Limitations:
  - High cost of replaying
    - » Time, use of the actual execution environment
  - Difficult to precisely replay the execution
    - » Concurrency or non-determinism
  - Difficult to find the exact point where to put the breakpoint
    - » If the breakpoint is too late, the execution needs to be redone
    - » Frequently many attempts are needed

Reversibility for debugging

- Reversible debuggers extend standard debuggers with the ability to execute the program under analysis also backward
- Avoids the common “Damn, I put the breakpoint too late” problem
  - Just execute backward from where the program stopped or where a wrong result appeared till the desired point is reached
- Some reversible debuggers also ensure that nondeterminism is resolved in the same way

Causal-consistent reversibility

- We are interested in debugging concurrent/distributed programs
- Since [Danos & Krivine, CONCUR 2004] the notion of reversibility for concurrent systems is causal-consistent reversibility
  - Any action can be undone, provided that its consequences (if any) are undone beforehand
  - Concurrent actions can be undone in any order, but causal-dependent actions are undone in reverse order
- At any point, many actions can be undone

Debugging and causality

- Causal-consistency relates backward computations with causality
- Debugging amounts to find the bug that caused a given misbehavior
- We use the following debugging strategy: follow causality links backward from misbehavior to bug
  - Causal-consistent reversible debugging
  - Originally proposed in [Giachino, Lanese & Mezzina, FASE 2014]

The **roll** primitive

- Causal-consistent debugging based on **roll** *n pid* semantics
- Undoes the last *n* steps of process *pid*...
- ... in a causal-consistent way
  - Before undoing an action one has to undo all (and only) its consequences
- A single **roll** may cause undoing steps in many processes
- Different interfaces for **roll** are needed, one for each kind of misbehavior that can occur in the language

Erlang and Core Erlang

- We target the Erlang language
- Emilio already explained why Erlang is interesting in his talk yesterday
- Functional language
- Based on the actors concurrency model
  - Processes are actors that communicate asynchronously by message passing
  - Each process has its own local mailbox
  - No shared memory
- During compilation, Erlang is first translated to Core Erlang

Different interfaces for **roll**

- One interface for each possible misbehavior
- In Erlang:
  - Wrong value in a variable: **roll var** *id* goes to the state just before the variable *id* has been created
  - Unexpected message: **roll send** *msgId* goes to the state where the message *msgId* has been sent
  - Wrong message received: **roll rec** *msgId* goes to the state where *msgId* has been received
  - Unexpected process: **roll spawn** *pid* goes to the state where process *pid* has been created

Using roll-like primitives

- The programmer can follow causality links backward
- The procedure can be iterated till the bug is found
- Only relevant steps are undone
  - Thanks to causal consistency
- No need for the programmer to know which process or expression originated the misbehavior
  - The primitives find them automatically
- Looking at which processes are involved gives useful information
  - The involvment of an unexpected process means that an interference has happened

CauDEr: Causal-Consistent Debugger for Erlang

- Only a prototype to test our ideas
- Supports a subset of Core Erlang
  - Sequential language + actor primitives
- Written in Erlang
- Available at https://github.com/mistupv/cauder
- Description and underlying theory in
  [Lanese, Nishida, Palacios & Vidal, FLOPS 2018]

Demo time!

Future directions

- We are currently working on a strong revision of CauDEr
  - Enable to record an execution and replay it in the debugger
  - Support Erlang instead of Core Erlang
- Is the **roll** primitive good?
  - Which is the impact on actual debugging?
  - It would be interesting to setup an experiment
- Are there other useful primitives?

Finally

Thanks!

Questions?