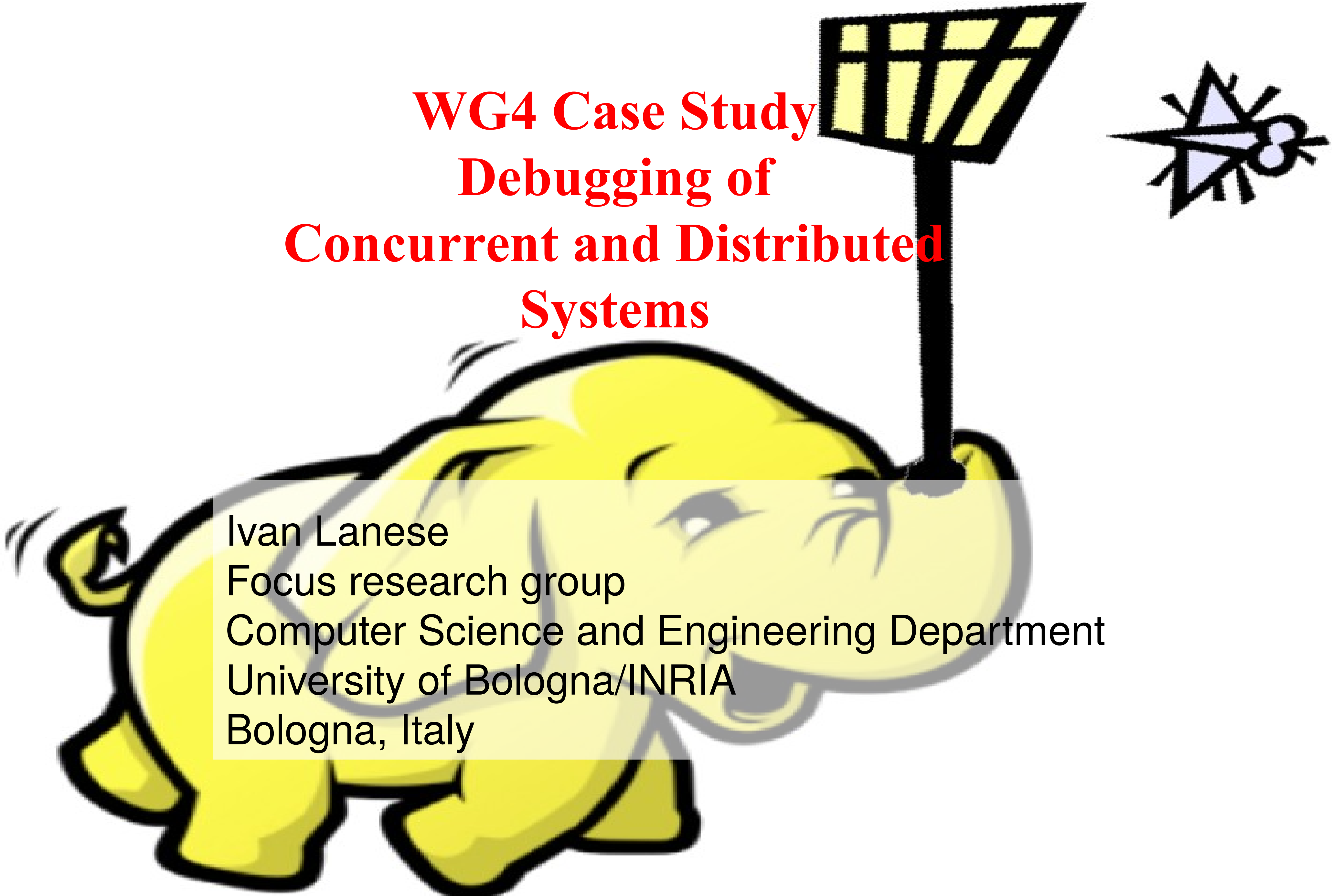
WG4 Case Study Debugging of Concurrent and Distributed Systems



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

Roadmap

- Reversible debugging
- State of the art: sequential debugging
- Debugging concurrent/distributed systems: causal-consistent reversible debugging
- Future directions

Roadmap

- Reversible debugging
- State of the art: sequential debugging
- Debugging concurrent/distributed systems: causal-consistent reversible debugging
- 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, a very little amount of research concentrates on debugging

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" error
  - Just execute backward from where the program stopped or where a wrong result appeared till the desired point is reached
- The overhead due to storing history information is a main limitation for reversible debuggers

Roadmap

- Reversible debugging
- State of the art: sequential debugging
- Debugging concurrent/distributed systems: causal-consistent reversible debugging
- Future directions

State of the art: sequential debugging

- Reversible debuggers exist
  - GDB, UndoDB
- Many reversible debuggers deal only with sequential programs
- Some of them allow one to debug concurrent programs
  - They register scheduler events
  - The same scheduling is used when the program is replayed

Sequential reversible debugging strategy

- Take an execution containing a failure and move backward and forward along it looking for the bug
- The exact same execution can be explored many times forward and backward
  - Non-determinism is no more a problem

A reversible debugger: GDB

- GDB supports reversible debugging since version 7.0 (2009)
- Uses record and replay
  - One activates the recording modality
  - Executes the program forward
  - Can explore the recorded execution backward and forward
  - When exploring, instructions are not re-executed

GDB reverse commands

- Like the forward commands (step, next, continue), but in the backward direction
- Reverse-step: goes back to the last instruction
- Reverse-next: goes back to the last instruction, does not go inside functions
- Reverse-continue: runs back till the last stop event
- ...
- Breakpoints and watchpoints can be used also in the backward direction

A commercial reversible debugger: UndoDB

- Already presented at the Grenoble meeting
- From UndoSoftware, Cambridge, UK
  http://undo-software.com/
- Built as an extension of GDB
- Available for Linux and Android
- Allows reversible debugging for programs in C/C++

UndoDB commands

- Close to GDB commands
- Some more high-level commands and configuration commands
- Commands to write a recorded execution to file, and reload it
  - Useful to record on client premises and explore at company premises

UndoDB winning feature

Performance

- Comparison with GDB, on recording gzipping a 16MB file

	Native	UndoDB	GDB
Time	1.49 s	2.16 s (1.75 x)	21 h (50000 x)
Space	-	17.8 MB	63 GB

Roadmap

- Reversible debugging
- State of the art: sequential debugging
- Debugging concurrent/distributed systems: causal-consistent reversible debugging
- Future directions

Causal-consistent reversible debugging

- 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) have been undone
  - 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 propose the following debugging strategy: follow causality links backward from misbehavior to bug
- Which primitives do we need to enable such a strategy?

A proposal: the **roll** primitive

- The main primitive we propose is **roll t n**
- Undoes the last **n** actions of thread **t**...
- ... in a causal-consistent way
  - Before undoing an action one has to undo all (and only) the actions depending on it
- A single **roll** may cause undoing actions in many threads

Different interfaces for **roll**

- One interface for each possible misbehavior
  - This depends on the language
- Examples are:
  - Wrong value in a variable: **rollvariable id** goes to the state just before the last change of variable **id**
  - Unexpected thread: **rollthread t** undoes the creation of thread **t**

Using roll-like primitives

- The programmer can follow causality links backward
- No need for the programmer to know which thread or instruction originated the misbehavior
  - The primitives find them automatically
- The procedure can be iterated till the bug is found
- Only relevant actions are undone
  - Thanks to causal consistency
- Looking at which threads are involved gives useful information
  - If an unexpected thread is involved, an interference between the two threads has happened

CaReDeb: a causal-consistent debugger

- Only a prototype to test our ideas
- Debugs programs in the μOz language
  - Toy functional language with threads and asynchronous communication via ports
- Written in Java
- Available at http://www.cs.unibo.it/caredeb
- Description and underlying theory in [Giachino, Lanese & Mezzina, FASE 2014]

Roadmap

- Reversible debugging
- State of the art: sequential debugging
- Debugging concurrent/distributed systems: causal-consistent reversible debugging
- Future directions

Future directions

- Enable causal-consistent debugging of real languages
  - Many constructs
  - One needs to understand their causal semantics
  - Large implementation effort
- 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?

UndoSoftware interests

- Debugging concurrent and distributed systems is challenging and requires a huge effort
  - They are not interested in solutions which are not efficient
  - They will not tackle it in the near future
- A main interest is parallel recording
- Debugging of MPI programs is also of interest

Finally

Thanks!

Questions?

Our target language: μOz

- A kernel language of Oz
  [P. Van Roy and S. Haridi. Concepts, Techniques and Models of Computer Programming. MIT Press, 2004]
- Oz is at the base of the Mozart language
- Thread-based concurrency
- Asynchronous communication via ports
- Shared memory
  - Variable names are sent, not their content
- Variables are always created fresh and never modified
- Higher-order language
  - Procedures can be communicated