Debugging of Parallel and Distributed Programs

João Lourenço       José C. Cunha       Vitor Duarte
Center for Informatics and Information Technologies
Universidade Nova de Lisboa, Portugal

Joao.Lourenco@di.fct.unl.pt
Why do programs have errors?

Problem

Domain knowledge errors

Devised computational solution

Specification errors

Algorithmic errors

Compile errors

Runtime errors

Communication errors

Coding errors

System failure

Inadequate performance

User errors

Solve
Program generation

Specification bugs

Programming bugs
Observing a program

execution

correctness
Distributed computations
Program histories

- A (sequential) program is a specification of sequence of events
  - With multiple alternative paths

- **Local History** $h_i \rightarrow$ sequence of events generated by executing the program “i”
  - $h_i = e_i(0), e_i(1), \ldots, e_i(f)$
  - $k^{th}$ event in $h_i$ produces the local state $s_k$

- **Global History** $H \rightarrow$ union of the local histories of N processes
  - $H = U h_1 U h_2 U \ldots U h_N$
A **distributed computation** is a partially ordered set (**poset**) defined as

\[ C^D = (H, \rightarrow) \]

- \( H = \) Global history
- \( \rightarrow = \) Lamport’s *happens before* relation
A **cut** of a distributed computation is a subset $C$ of its global history $H$ which contains an initial prefix for each of its local histories $C = \{h_1^x, h_2^y, \ldots, h_n^z\}$
The **frontier of a cut** is the set of the last states/events in a cut

\[ F = \{s_1^x, s_2^y, ..., s_n^z\} \]

- The frontier of a cut defines a **global state**
A **cut is consistent** if for all events in its frontier, all their past events are also included in the cut.
A **global state is consistent** if it corresponds to the frontier of a consistent cut.
“Time” in distributed computations

- Consistent cuts and consistent global states are used to identify particular instants in distributed computations

- Past & Future...
  - In sequential computations are defined wrt time
  - In distributed computations are defined wrt consistent cuts, as to the left and to the right of the consistent global state in a time-space diagram respectively
A run of a distributed computations is a total ordering $R$ of all the events in $H$ and is consistent with each $h_i$

- All the events for all processes $P_i$ appear in the same order in both $R$ and $h_i$
- Respects the happens-before relation for any two states/events in the same process
- May not respect the happens-before relation for casually related events in different processes

A run may correspond to an impossible execution of a distributed program
Distributed Computation & Run
Observation of distributed computations
Consistent observations

- Any permutation of a run R is a **possible observation** of it
  - The communication channels may not preserve message order

- A **consistent (inconsistent) observation** is one that corresponds to a consistent (inconsistent) run
Consistent run

- A **run is consistent** if all of its states/events verify the causal precedence relation
  
  - Any consistent run is a possible execution of a distributed program
Consistent run
Observation of distributed computations
Observing a program
Debugging phases / steps

- **State-based debugging**
- **Deterministic re-execution**
- **Systematic state exploration**
- **Global predicate detection**

**To obtain reproducible behavior**

**To analyze alternative paths**

**To evaluate correctness properties**
Distributed debugging activity
Observation of distributed programs
Tracing
Eliminating defects from concurrent software: the challenge [1]

- Concurrency is pervasive
  - Infrastructures
  - Operating systems
  - Middleware
  - Server software (web, email)
  - Desktop software
  - Embedded software
  - High performance
  - Databases
  - Web services
Concurrency is difficult

- Difficult to design
  - Need to reason about orderings between concurrent events
  - Invariants are inherently more complicated

- Difficult to debug
  - Bugs hard to locate and to reproduce
  - Ensuring code coverage is hard and time-consuming
Why is concurrent debugging difficult?

- Concurrency introduces non-determinism
- Multiple executions of the same test may have different interleaving (and different results!)
- Debugging affects the timing
- Lack of a functional IDE (e.g., like eclipse)
- Result: Most bugs are found later in the software development cycle
  - Higher costs to fix
“Know Your Enemy – Concurrency” (Sun)

“Debugging sequential programs is an order of magnitude easier than debugging parallel programs... It’s not clear if we even understand the problems involved with complex parallel programs – given this, how can we hope to solve the debugging problem?” (Microsoft)

“Concurrency increases the complexity of the design, testing and maintenance of the code... may introduce errors that are hard to detect” (Intel)
What should we do?

- Perhaps find adequate combinations of different areas
  - Model checking
  - Invariant checking / Theorem proving
  - Type systems
  - Program (static) analysis
  - Runtime (dynamic) analysis
  - Unit testing
  - State based debugging

- Fundamental need of powerful testing and debugging tools for concurrent software
Relevant technologies

- Formal verification
- Static analysis
- State space exploration
- Trace analysis
- Noise makers
- Race detection
- Execution steering and replay
- Virtualization
Distributed Debugging...
What’s new? [1]

- Many dynamic interacting entities
- Execution on multiple processors
- Global non-deterministic behavior
- Need to adapt to new hardware, OS, middleware and/or programming languages
Are all bugs equal?

- Yields a correct result, although it takes longer than acceptable
- Unwanted side effects caused by non-reentrant code and shared data
- Ordering failures but also deadlocks
- Violations of precedence or mutual exclusion relations

- Apparently incomprehensible behavior

Byzantine
Timing
Interleaving
Synchronization
Ordering
Sequential errors
Some debugging challenges

- Systems
  - Collecting, storing and gathering logs

- Ordering
  - Combining logs

- Completeness
  - Gaps, losses, detail, code coverage

- Non-determinism
  - How to eliminate/reduce the “probe effect”

- Language issues
  - Supporting high-level abstractions