Testing and Debugging for Concurrent Programs

Yi-Fan Tsai

yifan.tsai@colorado.edu

Outline

Concurrency Bugs in Real World
    Deadlock Bugs
    Non-Deadlock Bugs

Testing
    Coverage Criteria
    Systematic Testing

Debugging
    Fault Localization
    Reconstruction

References

Concurrency Programming is Chanllenging!

- Writing correct concurrent programs is notoriously difficult.
- Addressing this challenge requires advances in multiple directions, including bugs detection, program testing, programming model design, etc.
- Designing effective techniques in all these directions will significantly benefit from a deep understanding of *real world concurrency bug characteristics*.

[LPSZ08]

Application Set and Bug Set

105 concurrency bugs are randomly selected from 4 representative server and client open-source applications.

Application	Non-Deadlock	Deadlock
MySQL	14	9
Apache	13	4
Mozilla	41	16
OpenOffice	6	2
Total	74	31

Deadlock Bugs I

- 97% of the deadlock bugs are guaranteed to manifest if certain partial order between 2 threads is enforced.
- 22% are caused by one thread acquiring resource held by itself.
    - Single-thread based deadlock detection and testing techniques can help eliminate these simple bugs.
- 97% involve 2 threads circularly waiting for at most 2 resources.
    - Pairwise testing on the acquisition/release sequences to two resources can expose most bugs.
- 97% can deterministically manifest, if certain orders among at most 4 resource acquisition/relase operations are enforced.

Deadlock Bugs II

- The most common fix strategy is to let one thread give up acquiring one resource, such as a lock.
    - This strategy is simple, but it may introduce other non-deadlock bugs.

Non-Deadlock Bugs I

- Atomicity-Violation
  - Programmers tend to assume a small code region will be executed atomically.
  - Example:
    thread1: if (thd→proc_info) fputs(thd→proc_info, ...);
    thread2: thd→proc_info=NULL;
    thread1: if (thd→proc_info) fputs(thd→proc_info, ...);
- Order-Violation
  - Programmers commonly assume an order between two operations from different threads.
  - Example:
    parent thread: mThread = PR_CreateThread(...);
    child thread: mState = mThread→State;
    parent thread: mThread = PR CreateThread(...);

Non-Deadlock Bugs II

  - This is a different concept from atomicity violation. The example emphasizes that the assignment should happen *before* the read access. Even if memory accesses are proected by the same lock, their execution order still may not be guranteed.
- Multiple-Variable Bugs
  - Example: mOffset, mLength together mark the region of useful characters stored in dynamic string mContent.
    thread1: /* change the mContent */
    thread2: putc(mContent[mOffset + mLength - 1]);
    thread1: /* calculate and set mOffset and mLength */

Lessons from Non-Deadlock Bugs I

- 97% of non-deadlock bugs are covered by two patterns, atomicity-violation and order-violation.
- 32% are order-violation bugs.
  - A relatively not well-addressed topic.
- 96% are guranteed to manifest if certain partial order between 2 threads is enforced.
  - Testing can pairwise test program threads.
- 66% involve only one variable.
  - Focusing on concurrent accesses to one variable is a good simplifaction.
- 34% involve concurrent accesses to multiple variables.
  - A relatively not well-addressed topic! [LPH+07]

Lessons from Non-Deadlock Bugs II

- 90% can deterministically manifest, if certain order among no more than 4 memory accesses is enforced.
  - Testing can focus on the partial order among every small groups of accesses. This simplifies the interleaving testing space from exponential to polynomial regarding to the total number of accesses.
  - Most of the exceptions come from those bugs that involve more than 2 threads and/or more than 2 variables.

Testing

Testing

Requirements

- *Fast response*: Most bugs should be found very quickly.
- *Reproducibility*.
- *Coverage*: It should complete with precise guarantees.

Stategies

- *Stress testing* provides fast response during initial stages of software development.
- *Heuristic-based fuzzing* uses heuristics to direct an execution towards an interleaving that manifests a bug. These techniques often provide fast response. [Sen08]
- *Stateless model checking* systematically enumerates all schedules. It provides coverage guarantees and reproducibility.

[CBM10]

Coverage Criteria

- A fundamental problem of concurrent program bug detection and testing is that *the interleaving space is too large*.
- Real world testing resource can only check a small portion of the interleaving spaces.
- In order to systematically explore the interleaving space and effectively expose concurrent bugs, good *coverage criteria* are desired.

[LJZ07]

Criterion All: All-Interleavings

- The interleaving space gets a “complete coverage” if all feasible interleavings of shared accesses from all threads are covered.
- Property Set: $|\Gamma_{\texttt{ALL}}| = \prod_{i=1}^{M} \binom{\sum_{j=i}^{M} N_j}{N_i}$
  - M is the number of threads
  - N_i is the number of access events from thread i.

Criterion TPair: Thread-Pair-Interleavings

- The interleaving space gets a “complete coverage” if all feasible interleavings of all shared memory accesses from any pair of threads are covered.
- Fault Model: The model assumes that most concurrency bugs are caused by the interaction between two threads, instead of all threads.
- Property Set: $|\Gamma_{\texttt{TPair}}| = \sum_{1 \le i < j \le M} \binom{N_i + N_j}{N_i}$
    - M is the number of threads
    - N_i is the number of access events from thread i.

Criterion SVar: Single-Variable-Interleavings

- The interleaving space gets a "complete coverage" if all feasible interleavings of all shared accesses to any specific variable from any pair of threads are covered.
- Fault Model: This model is based on the observation that many concurrency bugs invole conflicting accesses to one shared variable, instead of multiple variables.
- Property Set: $|\Gamma_{\texttt{SVar}}| = \sum_{1 \le i < j \le M} \sum_{v \in V} \binom{N_{i,v} + N_{j,v}}{N_{i,v}}$.
    - V is the set of shared variables.
    - $N_{i,v}$ is the number of accesses from thread i to shared variable v.

Criterion PI: Partial-Interleavings

- Criterion DefUse: Define-Use
    - All possible define-use pairs are covered.
    - Fault Model: A read access uses a variable defined by a wrong writer.
    - Property Set: $|\Gamma_{\texttt{DefUse}}| = N^r + \sum_{1 \leq i \neq j \leq M} \sum_{v \in V} (N^r_{i,v} \cdot N^w_{j,v})$
        - N^r denotes the total number of read accesses.
- Criterion PInv: Pair-Interleavings
    - For each consecutive access pair from any thread, all feasible interleaving accesses to it have been covered.
        - A consecutive access pair accesses the same shared variable from one thread.
    - Fault Model: Atomicity violations.
    - Property Set: $|\Gamma_{\texttt{PInv}}| = PN + \sum_{1 \leq i \neq j \leq M} \sum_{v \in V} (PN_{i,v} \cdot N_{j,v})$
        - PN: the number of all consecutive access pairs.

Criterion LR: Local-or-Remote

- Criterion LR-Def: Local-or-Remote-Define
    - For each read-access r in the program, both of the following cases have been covered - r reads a variable defined by local thread (or the initial memory state) and r reads a variable defined by a different thread.
    - Property Set: $|\Gamma_{LR-Def}| = 2N^r$.
- Criterion LR-Inv: Local-or-Remote-interleaving
    - For every consecutive access pair from any thread accessing any shared variable, both of the follwing cases have been covered - the pair has an unserializable interleaving access and the pair does not have one.
        - An unserializable interleaving is an interleaving that does not have equivalent effects to a serial execution. [LTQZ06]
    - Property Set: $|\Gamma_{LR-Inv}| = 2PN$.

Systematic Testing

- “Heisenbugs” occasionally surface in concurrent systems that have otherwise been running reliably for months. Slight changes to a program, such as adding debugging statements, sometimes drastically reduce the likelihood of erroneous interleavings, adding frustration to the debugging process.
- CHESS takes complete control over the scheduling of threads and asynchronous events, thereby capturing all the interleaving nondeterminism in the program. [1]

[MQB$^+$08]

---

[1] CHESS is able to find assertion failures, deadlocks, livelocks, and “sluggish I/O behavior”.

CHESS Architecture

- The scheduler is implemented by redirecting calls to concurrency primitives, such as locks and thread-pools, alternate implementations provided in a wrapper library.
- The wrappers provide enough hooks to CHESS to control the thread scheduling. CHESS enables only one thread at a time.
- CHESS repeatedly executes the same test driving each iterations of the test through different schedule.

Preemption Bounding

- A real-world may preempt a thread at just about any point in its execution.
- CHESS explores thread schedules giving priority to schedules with fewer preemptions.
- In experience, very serious bugs are reproducible using just two preemptions. Bounding the number of preemptions is a very good strategy to tackle state-space explosion.

Prioritized Search

GAMBIT extends CHESS with prioritized search that combines the speed benefits of heuristic-guided fuzzing with the soundness, progress, and reproducibility guarantees of stateless model checking. [CBM10]

- Techniques for state-space explosion
    - Partial-order reduction
    - Preemption bounding
- Priority function
    - New happens-before executions
    - Random search
    - Tester guide
    - Known patterns

Debugging

Fault Localization

- Fault-detection tools for concurrent programs find data-access patterns among thread interleavings, but they report benign patterns as well as actual faulty patterns.
- The fault-localization technique can pinpoint faulty data-access patterns in multi-threaded concurrent programs.

[PVH10]

Technique

- Online pattern identification
    - The system records unserializable and conflicting interleaving patterns, and subsequently associates them with passing and failing runs.
    - For example, pattern $\texttt{W}_{1,100} - \texttt{W}_{2,200} - \texttt{R}_{1,105}$ represetns an unserializable pattern (atomicity violation).
        - Between the write and read accesses to a variable from thread 1 at statement 100 and 105, thread 2 writes to the same variable at statement 200.
- Pattern suspiciousness ranking
    - Fault localization assumes that entities (patterns) executed more often by failing executions than passing executions are more suspect.
    - $\texttt{suspiciousness}(s) = \frac{\%\mathit{failed}(s)}{\%\mathit{failed}(s)+\%\mathit{passed}(s)}$.
    - Prioritized ranking guides the developer toward the most likely cause of a fault and mitigates false positives.

Reconstruction

- Many approaches to detect bugs report too little information or too much information.
    - A single communication event is not enough to understand concurrency bugs.
    - Replay makes programmers sift through an execution trace to comprehend bugs.
- Reconstructions of buggy executions are short, focused fragments of the interleaving schedule surrouding a program event such as shared-memory communication.

[LWC11]

Communication Graph Debugging

- The process begins with the observation of a bug or a bug report.
- A test case is designed to trigger the bug, and runs the test multiple times. A communication graph is collected from each execution, and the labeled as buggy or nonbuggy, depending on the outcome of the test.
- Reconstructions are built from edges in buggy graphs. Statistical features are used to compute the likelihood and rank the edges and reconstructions.

Example I

```
1 class Queue{
2     dequeue(){
3         if (qsize==0) return null;
4         size--;
5         return items[...];}
6     size() { return qsize;} }
7 if (q.size()==0) continue;
8 q.dequeue().get();
```

- Problematic senario may happen when thread 1 reads qsize at line 3. The value may be written by thread 2 at line 4 rather than the value read by thread 1 at line 6.
- Identifying the communication 4 → 3 is insufficient because it occurs in both buggy and nonbuggy executions.

Context-Aware Communication Graphs

- In a context-aware graph, a node is a pair (I, C) representing the execution of a static instruction I in communication context, C.
- Example: edge $(4, L_R - R_R - R_W) \to (3, R_W - R_W - L_R)$ only occurs in buggy exxecutions' graphs.
    - The context of the sink nodes implies that the most recent event is a remote write which can correspond to thread 2's write at line 4.

References I

Katherine E. Coons, Sebastian Burckhardt, and Madanlal Musuvathi.
Gambit: effective unit testing for concurrency libraries.
In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 15–24, New York, NY, USA, 2010. ACM.

Shan Lu, Weihang Jiang, and Yuanyuan Zhou.
A study of interleaving coverage criteria.
In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 533–536, New York, NY, USA, 2007. ACM.

References II

Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou.
Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs.
In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 103–116, New York, NY, USA, 2007. ACM.

Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou.
Learning from mistakes: a comprehensive study on real world concurrency bug characteristics.
In *Proceedings of the 13th international conference on Architectural support for programming languages and*

References III

operating systems, ASPLOS XIII, pages 329–339, New York, NY, USA, 2008. ACM.

Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou.
Avio: detecting atomicity violations via access interleaving invariants.
In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 37–48, New York, NY, USA, 2006. ACM.

References IV

Brandon Lucia, Benjamin P. Wood, and Luis Ceze.
Isolating and understanding concurrency errors using reconstructed execution fragments.
In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 378–388, New York, NY, USA, 2011. ACM.

Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu.
Finding and reproducing heisenbugs in concurrent programs.
In Richard Draves and Robbert van Renesse, editors, *OSDI*, pages 267–280. USENIX Association, 2008.

References V

Sangmin Park, Richard W. Vuduc, and Mary Jean Harrold.

Falcon: fault localization in concurrent programs.
In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 245–254, New York, NY, USA, 2010. ACM.

Koushik Sen.
Race directed random testing of concurrent programs.
In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 11–21, New York, NY, USA, 2008. ACM.