CS636 - Analysis of Concurrent Programs 2020-2021-II: Paper Reading 2

Name: Sarthak Singhal Roll No.: 170635

1. Adversarial Memory

Paper Summary

With the huge increase in parallel programs, data races are a threat to the systems. Almost none of the previous static and dynamic race detection algorithms can differentiate between destructive races that cause erroneous behaviour and benign races which can be intentional for performance perspective. This necessary classification of benign and destructive races is impractical to do manually as one needs deep understanding of the codebase which limits utility of these algorithms. This paper introduces a dynamic analysis technique that helps to precisely identify only destructive race conditions together with concrete execution as a witness to the error.

First, a precise race detector is used to fetch the variables which are racy(either benign or destructive). Then an attempt is made to generate an execution which can cause incorrect program behaviour due to the race condition on a particular racy variable. This adversarial execution exploits the full range of all possible behaviours that are allowed by relaxed memory models that are found in current architectures. Even though relaxed memory models are allowed but most of the time memory system shows sequentially-consistent behaviour which makes testing problematic. So, the authors designed an adversarial memory system, JUMBLE, which exploits the flexibility of relaxed memory models to crash the application. The authors also introduce Progressive Java Memory Model(PJMM) which is restriction of HB memory model removing the ability for reads to see future writes. Based on the transition rules of PJMM, if a variable has data race then a read will only see most recent write otherwise it can see many other stale writes which are legal. Vector clocks are used as a notion of tracking the happens-before relation. Write buffer is also compressed frequently to limit its size. PJMM simplifies the implementation of JUMBLE without significantly limiting its ability to expose destructive races.

In the implementation of JUMBLE, all the writes to a particular racy variable are stored in the write buffer and all reads are adversarial. Based on some heuristics JUMBLE tries to pick values from visible set which are likely to cause erroneous behaviour. Some heuristics include picking up the oldest value from visible set, another can be pick any random value from the visible set. Experiments show that JUMBLE performs very well in exposing destructive races on various benchmarks. JUMBLE shows this level of result without incurring any additional overhead i.e. its slowdown is roughly same as the EMPTY tool in RoadRunner. This work shows highly effective dynamic analysis approach to expose destructive race conditions.

Key insights/Strengths

- JUMBLE ignores all the benign races and detects just the destructive ones. This helps the developers to focus on the real issues as sometimes on trying to fix benign races real bugs maybe introduced while destructive races can be mistaken as benign.
- Together with detecting the destructive races JUMBLE also creates an execution that is a witness to the produced error. This can help to clearly reason about the origin of the bug.
- The heuristics used in the write buffer compression optimization are very subtle. Most importantly the garbage collection heuristics help to reduce the overheads in storing the write buffer while at the same time maintaining the efficiency in detecting destructive races.
- The *Oldest* heuristic to pick value for read operation from the visible set picks the oldest element from the visible set. The idea used to return the most-recently written value sometimes is subtle and important. As without it sometimes the system can run into infinite loop in busy-waiting loops if most stale value is returned all the times.
- The evaluation is nice with explanation of each individual program together with manual verification for correct behaviour.
- The slowdown is similar to Empty tool i.e. just overhead of RoadRunner. This low overhead is because few write-buffer operations are performed as small number of racy memory locations are tracked and each one is updated only a small number of times based on the evaluation on certain programs.

Weaknesses

- JUMBLE doesn't explore all possible interleavings, so many destructive races that can cause problems in some interleavings may remain undetected.
- The maximum buffer size is kept to be 32 and the oldest entries are removed if there is a possibility of increase in the buffer size. This can lead to missing destructive races which maybe only possible with the oldest entry that has been removed. The authors claim that this max limit on buffer size didn't impact precision on some programs but on other programs there maybe a chance of missing destructive races.
- JUMBLE tracks a single syntactic field returned by precise data race detector at a time. This can lead to missing some errors that are possible only due to multiple fields e.g. there is an if condition that is dependent on values of 2 variables and will only give erroneous result if values of both variables are stale.
- For arrays the reads are jumbled only for indices 0 and 1. This can also lead to missing errors that are only possible due to some other indices of the arrays.

Unsolved problems/potential future work

- Tracking single syntactic field can lead to missing some errors only possible due to multiple fields. Heuristics should be designed to incorporate multiple variables in a particular analysis.
- Instead of jumbling accesses of only indices 0 and 1 for all the arrays, some heuristics must be used to jumble accesses of different indices for different arrays. E.g. the precise race detector can return a list of indices for all arrays which are mostly involved in dataraces. Then few top indices from this list can be used for analysis.
- Adapt Jumble's adversarial memory approach to detect destructive race conditions for hardware-level memory models like TSO, PSO, and PSLO.
- Many of the previous tools explore the traditional approach of many or all possible thread interleavings assuming sequential consistency. This approach uses adversarial memory to find destructive data races. So, future tools can exploit both the scheduling and memory model non-determinism for detecting concurrency errors.

2. Relaxer

Paper Summary

With the huge increase in parallel programs, data races are a threat to the systems. Many of the data races are intentional to avoid synchronization overheads. This presence of explicit data races increases the need for tools to help the developers to write and test high performance parallel software. Many tools are already present but most of them check just the violation of SC which doesn't confirm the presence of a potential bug. This paper introduces an active testing technique, RELAXER, to detect concurrency bugs under relaxed memory models.

RELAXER consists of 2 phases. In the first phase it examines a sequentially consistent execution of a given program and using a dynamic analysis technique try to predict potential violations of sequential consistency under a given relaxed memory model. A trace is generated by running a given program on a random thread schedule and sequentially consistent memory. Using an iterative algorithm RELAXER generates a potential cycle from the given execution trace. These potential cycles are passed on to the second phase which requires an operational implementation of relaxed memory model under consideration. The paper describes operational implementations which are conservative approximations for the memory models SC, PSO, TSO, and PSLO. Then in the second phase which is parameterized by the memory model, RELAXER tries to create executions under the given memory model in which the potential cycles provided from phase 1 are actual happens-before cycles and hence violate SC. RELAXER does this by controlling the thread scheduler(e.g. delaying some threads) and the given memory model. Then it checks whether the generated execution which violates SC actually lead to errors. RELAXER reports a bug only if there is an erroneous output in the execution in which SC violations occur.

RELAXER provides implementation for each memory operation for the memory models: SC, TSO, PSO, and PSLO which help to simulate the memory model and perform controlled scheduling of the phase 2 algorithms. RELAXER is evaluated on 10 benchmarks which comprise of mutual exclusion, concurrent data structures and parallel application benchmarks. The evaluation confirms that RELAXER can predict and create real sequential consistency violations and also create the input happens-before cycle with high probability. It also confirms that RELAXER can produce real bugs involving relaxed memory models. RELAXER is unsound as it can't confirm all the predicted cycles due to some practical limitations. Even if all the predicted cycles are confirmed it might be the case that the cycle is not buggy and hence the error will not be reported. Dekker's algorithm had some benign cycles which were also not reported buggy by RELAXER. RELAXER can also be easily adapted to other memory models for which operational approximation can be developed.

Key insights/Strengths

- Together with distinguishing harmful violations of sequential consistency from benign ones RELAXER also creates an execution that is a witness to the produced error. This can help to debug and clearly reason about the origin of the bug.

- RELAXER provides implementations of the operational semantics of various relaxed memory models. This can be used by the programmers to test and debug as using RELAXER they can observe the run-time behaviour of the program under the corresponding model which might be counter-intuitive before.
- Some previous tools like SOBER just find violations of sequential consistency which doesn't necessarily mean that there is an error. While RELAXER focuses on creating executions in which there are violations of SC but they should give errors also i.e. SC violations with no errors will not be reported by RELAXER.
- In the evaluation section, the explanation of individual bugs in some of the benchmarks is given. It helps to understand the various possibilities of bugs in some standard algorithms.
- Some previous tools have limitations like SOBER can just be applied to TSO and not any other memory model. While RELAXER can incorporate any memory model for which we can develop an operational approximation.

Weaknesses

- Operational implementation is conservative approximation of full relaxed memory models. So, there might be some bugs that can be missed if they are only possible in the full version of relaxed memory models and not the conservative one. E.g. too strict implementation of memory barrier in PSO can't capture all the behaviours of PSO. Hence, it can miss some bugs that could have been detected in the original version of PSO.
- RELAXER only focuses on length-4 cycles. This can lead to missing program errors that are only caused due to greater length cycles.
- RELAXER can have false negatives. It may not confirm the cycles even if all are feasible. It can happen because of the requirement of a complex thread schedule or maybe the need of a longer time for a store to remain buffered which is not practical.
- RELAXER also doesn't compare the runtime performance with the prior tools due to which it doesn't justify clearly the real motivation behind using RELAXER.

Unsolved problems/potential future work

- RELAXER is currently implemented for C. The implementation can be ported to other more industry friendly languages like Java, C++, etc. This can increase the reach of RELAXER to much more software applications.
- The operational implementations can be made a bit more relaxed so that they become more similar to the original relaxed memory models. This can lead to detection of more number of concurrency bugs which might be missed due to slight stricter nature of the operational implementations.
- The authors had to manually replace calls to synchronization primitives like pthread locks

with calls to RELAXER stubs that translate the operations into loads, stores, CASs, and memory barriers. Instead of doing this a dynamic analysis framework like RoadRunner can be used which will provide some functions that would be automatically called on lock acquire, write operation, etc.