# Dynamic data race prediction (Overview)

Martin Sulzmann, Peter Thiemann

2024-05-08

## Goals

- Main sources of bugs in concurrent programs: data races and deadlocks
  - data races may go unnoticed
  - deadlocks may occur rarely
  - ... programs are not 100% reliable!
- How can we determine if a program has a data race or a deadlock?
  - static analysis of the program text (without running the program)
  - dynamic analysis (observe the running program)

## Dynamic versus static analysis methods

### Dynamic analysis

- Execute the program and observe its behavior
- Determine if there is any potential bad behavior based on this specific program run

### Static analysis

- Predict all possible program runs without actually executing the program
- (usually) sound, but overapproximative
  - if there is a bug, it will be flagged
  - but also non-bug may be flagged

### Here we consider *dynamic* data race prediction:

- Execute the program
- Record the events that took place in a trace
- Analyze the trace to check if there is any potential bad behavior (in our case if there is a potential data race)

We restrict our attention to programs that make use of operations *acquire* and *release* (aka lock/unlock).

## Motivating example

Consider the following Go program where we emulate acquire/release via a buffered channel.

```
func example1() {
    var x int
    y := make(chan int, 1)

    acquire := func() {
        y <- 1
    }
    release := func() {
        <-y
    }

    // Thread T2
    go func() {
        acquire()
        x = 3 // P1
        release()

    }()

    // Thread T1 = Main Thread
    x = 4 // P2
    acquire()
    release()

    time.Sleep(1 * 1e9)
    fmt.Printf("%d \n", x)

}
```

The variable `x` is *shared* between threads T1 and T2. Access to variable `x` at location P2 is not protected by a lock. Hence, there is a potential data race that involves the conflicting operations labeled as P1 and P2.

## Program trace

Events are collected in a program **trace**. A trace is a linear sequence of events and represents an interleaved execution of the program.

An **event** registers interesting program behavior. For the running example, we assume write events `w(x)`, where `x` is a shared variables, and acquire/release events `acq(y)` and `rel(y)`, where `y` is a lock. Each event is connected to one of the operations we are interested in.

For each event, the trace records the thread where the event happened. Each event can be identified by its position in the trace.

There is a tabular notation for traces with a separate column for each thread and separate row for each event. The trace position of an event corresponds to the row number.

Here is an example trace that results from running the example program.

```
Trace A:

     T1          T2

e1.   w(x)
e2.   acq(y)
e3.   rel(y)
e4.                acq(y)
e5.                w(x)
e6.                rel(y)
```

The above trace tells us something about the specific program run we consider. This run first executes the operations in thread T1 (trace positions 1-3) followed by the operations in thread T2 (trace positions 4-6).

What about the data race? Recall that by "looking" at the program text we "guess" that there is a potential data race. However, the observed program behavior represented by the above trace does not exhibit a data race right away.

## Conflicting events and data race

Two events are **conflicting events** if they are read/write events for the same variable that come from different threads and at least one of them is a write event.

A **data race** arises if two conflicting events appear right next to each other in the trace. That implies that both events may happen in any order.

The above trace contains two conflicting events at trace positions 1 and 5. There is `w(x)` from thread T1 at position 1 and `w(x)` from thread T2 at position 5. The events *do not* appear right next to each other in the trace. Hence, we cannot conclude that there is a data race.

## Rerun of program versus reordering of trace

Suppose we rerun the program with a different schedule where thread T2 executes before T1. Here is the resulting trace.

Trace B:

```
     T1          T2
e4.               acq(y)
e5.               w(x)
e6.               rel(y)
e1.   w(x)
e2.   acq(y)
e3.   rel(y)
```

The same operations are executed. Hence, we encounter the same events in a different order. To highlight this point, we refer to the events using the trace positions of the original trace (trace A).

What about the data race? There is still *no* data race present in the above trace (B). The conflicting events, the writes in T1 and T2, do not appear right next to each other in the trace.

But if we try again and again with different schedules, eventually we obtain the following trace.

Trace C:

```
     T1          T2
e4.               acq(y)
e5.               w(x)
e1.   w(x)
e6.               rel(y)
e2.   acq(y)
e3.   rel(y)
```

As in trace B, thread T2 starts execution first. But unlike trace B, the scheduler switches to T1 after the write operation. Thus, the two conflicting writes appear right next to each other in the trace. This run materializes a data race!

Instead of rerunning the program to obtain traces A, B, and C, another method is to consider *valid trace reorderings* that result from trace A. Reordering of a trace means that we change the order of its events such that the resulting sequence of events still represents a sensible execution sequence. Indeed, traces B and C are valid trace reorderings of trace A.

## Valid trace reorderings

What are the conditions for a valid trace reordering?

We need some notation.

- For a trace $P$ and thread $T$, we write $P \downarrow T$ for the list of events in thread $T$ contained in $P$.

## Program order must be maintained

Consider the following reordering of the events in trace A.

```
Trace D:

     T1          T2

e2.   acq(y)
e3.   rel(y)
e4.                acq(y)
e1.   w(x)
e5.                w(x)
e6.                rel(y)
```

This trace seems to exhibit a data race. While trace D is a reordering of A, it is **not** a valid reordering as it violates *program order*!

The **Program order Condition** states:

- Let $P$ be some trace and $P'$ be a permutation of $P$.
- $P'$ preserves program order if, for each thread $T$, $P \downarrow T = P' \downarrow T$.

Trace D does not preserve program order with respect to A because e1 occurs after e2 and e3.

In trace A:

```
[e1, e2, e3]
```

In trace D:

```
[e2, e3, e1]
```

where we use list notation to represent a sequence of events.

## Lock semantics must be maintained

Consider the following reordering of the events in trace A.

```
Trace E:

     T1          T2

e1.   w(x)
e2.   acq(y)
```

```
e4.                 acq(y)
e5.                 w(x)
e6.                 rel(y)
e3.   rel(y)
```

Trace E is a reordering of A, but it violates the *lock semantics*.

The **Lock Semantics Condition** states:

- Let $P$ be some trace and $P'$ be some reordering of $P$.
- If $P' = P_1 acq(y) P_2 acq(y) P_3$, then there must be some event $rel(y)$ in $P_2$.
- If $P' = P_1 rel(y) P_2$, then there must be some $acq(y)$ such that $P_1 = P_1' acq(y) P_1''$ and there is no $rel(y)$ in $P_1''$.

Trace D violates the Lock Semantics Condition, because it contains two acquire events acq(y) without a rel(y) event in between.

## Last writer must be maintained

We explain the *Last Writer Condition* with another example.

Consider the program

```
func example3() {
    x := 1
    y := 1

    // Thread T1
    go func() {
        x = 2   // w(x)
        y = 2   // w(y)
    }()

    // Thread T2 = Main Thread
    if y == 2 { // r(y)
        x = 3   // w(x)
    }

}
```

This program can give rise to three different traces F1, F2, and F3. Here, we augment the read event with the actual value that is read to show the difference. (We ignore the initial assignments as they happen before thread T1 is started.) In F1, the read of y occurs after both writes in T1, so it sees the updated value 2 and executes the true-branch of the conditional. In F2 and F3, the read of y occurs before the w(y), so it sees the old value 1 and does **not** execute the true-branch of the conditional.

`Trace F1:`

```
     T1            T2

e1.   w(x)
e2.   w(y)
e3.                 r(y)=2
e4.                 w(x)

----------
Trace F2:

     T1            T2

1.   w(x)
2.                 r(y)=1
3.   w(y)

----------
Trace F3:

     T1            T2

1.                 r(y)=1
2.   w(x)
3.   w(y)
```

Suppose now we obtain trace F1 from a run of the program and we seek a reordering to exhibit a data race.

```
Trace G:

      T1            T2

e3.                 r(y)
e4.                 w(x)
e1.   w(x)
e2.   w(y)
```

Trace G is a reordering of F1. The two writes on x appear now right next to each other. It seems that we encounter a data race. However, the reordering G is not valid because it violates the *Last Writer Condition*.

If $P_1 r(x) P_2$ is a trace, then the *last writer* of r(x) is some w(x) such that $P_1 = P_1' w(x) P_1''$ and $P_1''$ does not contain another $w(x)$.

The **Last Writer Condition** states:

- Let P be some trace with a read event r(x) and P' be some reordering of P.

- The event r(x) must have the same *last writer* in P' as in P.

The event e2 is the last write of e3 in trace F1. In the reordering G, e3 has no last writer. Hence, the Last Writer Condition is violated.

The objective of the Last Writer Condition is to rule out infeasible traces. It works for our example because G is different from F1, F2, and F3 (the only feasible traces).

## Summary and outlook

The above examples show that detection of a data race can be challenging. We need to find the *right reordering* of a trace and program run. We can hope to rerun the program over and over again to encounter such a trace, but this is clearly very time consuming and likely we often encounter the same (similar) traces again. Finding a devious schedule under which the data race manifests itself can be tough to find.

What to do? We record the trace of a specific program run. Two conflicting events may not appear in the trace right next to each other. However, we may be able to predict that there is some trace reordering under which the two conflicting events appear right next to each other (in the reordered trace). This approach is called **dynamic data race prediction**.

Exhaustive predictive methods attempt to identify as many reorderings as possible (all!). Exhaustive methods do not scale to real-world settings because program runs and the resulting traces may be large and considering all possible reorderings generally leads to an exponential blow up.

Here, we consider efficient predictive methods. By efficient we mean a run-time that is linear in the size of the trace. As we favor efficiency over exhaustiveness, we may compromise completeness and soundness.

**Complete** means that all valid reorderings that exhibit some race can be predicted. If a method is incomplete, we call every race that is not reported race a **false negative**.

**Sound** means that all races reported by the method can be exhibited with some valid reordering of the trace. If a method is unsound, it may report a trace as racy even if there is no valid reordering that exhibits a race. Such a report is a **false positive**.

An exhaustive analysis would be sound and complete, but we need to compromise to obtain an efficient anaysis. Specifically, we discuss two popular, efficient dynamic data race prediction methods:

- Happens-before (incomplete)
- Lockset (unsound)