Dynamic data race prediction - Happens-before and vector clocks

Martin Sulzmann, Peter Thiemann

2024-05-08

Overview

Data race prediction

Two read/write operations on some shared variable are *conflicting* if the operations happen in different threads and at least one of the operations is a write.

Dynamic analysis method

We consider a specific program run represented as a trace. The trace contains the sequence of events as they took place during program execution. We carry out the analysis based on this trace.

Exhaustive versus efficient and approximative methods

To identify conflicting events that are in a race, we could check if there is a valid reordering of the trace under which both events occur right next to each other. In general, this requires to exhaustively compute all reorderings which is not efficient.

Efficient methods approximate by only considering certain reorderings.

Instead of explicitly constructing a reordering, we calculate a *happens before relation.*

Happens-before

A partial order on events to determine if one event happens before another event.

If events are unordered, we assume that these events may happen concurrently (i.e., in any order).

Ground truth: the **must happen before relation** (mhb).

Any happens-before (partial) order approximates the must happen before relation.

- If the order is too large, it orders events that may not be ordered by mhb: it misses data races and may yield false negatives.
- If the order is too small, it does not order events that may be ordered by mhb: it hallucinates data races and may thus yield false positives.

Vector clocks

- A distributed time stamp
- A representation for a happens-before relation.

If two conflicting operations are unordered under the happens-before relation, then we report that these operations are in a (data) race.

Trace and event notation

We write $t\#e_i$ to denote event e at trace position i in thread t.

In plain text notation, we sometimes write `t#e_i`.

We assume the following events.

```
e ::=  acq(y)         -- acquire of lock y
   |   rel(y)         -- release of lock y
   |   w(x)           -- write of shared variable x
   |   r(x)           -- read of shared variable x
```

Consider the trace

$[1\#w(x)_1, 1\#acq(y)_2, 1\#rel(y)_3, 2\#acq(y)_4, 2\#w(x)_5, 2\#rel(y)_6]$

and its tabular representation

```
     T1          T2

1.   w(x)
2.   acq(y)
3.   rel(y)
4.               acq(y)
5.               w(x)
6.               rel(y)
```

Instrumentation and tracing

We ignore the details of how to instrument programs to carry out tracing of events. For our examples, we generally choose the tabular notation for traces. In practice, the entire trace does not need to be present as events can be processed *online* in a stream-based fashion. A more detailed *offline* analysis may get better results if the full trace is available.

Trace reordering

To predict if two conflicting operations are in a race we could *reorder* the trace. Reordering the trace means that we simply permute the elements.

Consider the example trace.

```
     T1          T2

1.   w(x)
2.   acq(y)
3.   rel(y)
4.               acq(y)
5.               w(x)
6.               rel(y)
```

Here is a possible reordering.

```
     T1          T2

4.               acq(y)
1.   w(x)
2.   acq(y)
3.   rel(y)
5.               w(x)
6.               rel(y)
```

This reordering is *not valid* because it violates the lock semantics. Thread T2 holds locks y. Hence, its not possible for thread T1 to acquire lock y as well.

Here is another invalid reordering.

```
     T1          T2

2.   acq(y)
3.   rel(y)
4.               acq(y)
1.   w(x)
5.               w(x)
6.               rel(y)
```

The order among elements in trace has changed. This is not allowed.

Valid trace reordering

For a reordering to be valid the following rules must hold:

1. The elements in the reordered trace must be part of the original trace.
2. The order among elements for a given trace cannot be changed. See The Program Order Condition.
3. For each release event rel(y) there must exist an earlier acquire event acq(y) such there is no event rel(y) in between. See the Lock Semantics Condition.
4. Each read must have the same *last writer*. See the Last Writer Condition.

A valid reordering only needs to include a subset of the events of the original trace.

Consider the following (valid) reordering.

```
     T1          T2

4.               acq(y)
5.               w(x)
1.   w(x)
```

This reordering shows that the two conflicting events are in a data race. We only consider a prefix of the events in thread T1 and thread T2.

Exhaustive data race prediction methods

A “simple” data race prediction method seems to compute all possible (valid) reordering and check if they contain any data race. Such *exhaustive* methods generally do not scale to real-world settings where resulting traces may be large and considering all possible reorderings generally leads to an exponential blow up.

Approximative data race prediction methods

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

Complete means that all valid reorderings that exhibit some race can be predicted. If incomplete, we refer to any not reported race as a false negative.

Sound means that races reported can be observed via some appropriate reordering of the trace. If unsound, we refer to wrongly a classified race as a false positive.

We consider here Lamport’s happens-before (HB) relation that approximates the possible reorderings. The HB relation can be computed efficiently but may lead to false positives and false negatives.

Lamport's happens-before (HB) relation

Let T be a trace. We define the HB relation < as the smallest strict partial order that satisfies the following conditions:

Program order

If $t\#e_i, t\#f_{i+n} \in T$ for some $n > 0$, then $t\#e_i < t\#f_{i+n}$.

Critical section order

If $t_1\#rel(x)_k, t_2\#acq(x)_{k+n} \in T$ with $t_1 \neq t_2$ and $n > 0$, then $t_1\#rel(x)_k < t_2\#acq(x)_{k+n}$.

- Program order states that events in the same thread are ordered according to their trace position.
- Critical section order states that critical sections are ordered according to their trace position.
- For each acquire the matching release must be in the same thread. Hence, the critical section order only needs to consider a release and a subsequent acquire.

Example

Consider the trace

$[1\#w(x)_1, 1\#acq(y)_2, 1\#rel(y)_3, 2\#acq(y)_4, 2\#w(x)_5, 2\#rel(y)_6]$.

From program order we obtain

$$1\#w(x)_1 < 1\#acq(y)_2 < 1\#rel(y)_3$$

and

$$2\#acq(y)_4 < 2\#w(x)_5 < 2\#rel(y)_6.$$

From critical section order we obtain

$$1\#rel(y)_3 < 2\#acq(y)_4.$$

As $<$ is a strict partial order, we can derive by transitivity that

$$1\#w(x)_1 < 2\#w(x)_5.$$

Happens-before data race check

If there are conflicting events e and f but neither $e < f$ nor $f < e$, then (e, f) is a *HB data race pair*.

The argument is that if neither $e < f$ nor $f < e$ we are able to reorder the trace such that e and f appear right next to each other (in some reordered trace).

Note. If (e, f) is a *HB data race pair* then so is (f, e). In such a situation, we consider (e, f) and (f, e) as two distinct representative for the same data race. When reporting (and counting) HB data races we only consider a specific representative.

Event sets

Let ES_e be the set of events that happen up-to and including event e. That is, $ES_e = \{f \mid f < e\} \cup \{e\}$.

Example - Trace annotated with event sets

We write `ei` to denote the event at trace position `i`.

```
     T1          T2            ES

1.   w(x)                     ES_e1 = {e1}
2.   acq(y)                   ES_e2 = {e1,e2}
3.   rel(y)                   ES_e3 = {e1,e2,e3}
4.               acq(y)       ES_e4 = ES_e3 U {e4} = {e1,e2,e3,e4}
5.               w(x)         ES_e5 = {e1,e2,e3,e4,e5}
6.               rel(y)       ES_e6 = {e1,e2,e3,e4,e5,e6}
```

Observations:

- To enforce the critical section order we add the event set $ES_{rel(y)}$ of some release event to the event set $ES_{acq(y)}$ of some subsequent acquire event.
- To enforce the program order, we accumulate events within one thread (in essence, building the transitive closure).

To decide if $e < f$ we can check for $ES_e \subset ES_f$.

Consider two conflicting events e and f where e appears before f in the trace. To decide if e and f are in a race, we check for $e \in ES_f$ or $f \in ES_e$. If yes, then there is no race (because $e < f$ or $f < e$). Otherwise, there is a race.

Set-based race predictor (state)

We compute event sets by processing the events in the trace from beginning to end.

We maintain the following state variables when processing event e_i.

```
D(t)

  Each thread t maintains the set of events that happen before e_i.
  (initially empty)

R(x)

  Most recent set of concurrent read events on x.
  (initially empty)

W(x)

  Most recent write event on x.
  (initially undefined)

Rel(y)

  Rel(y) contains the event set of the most recent release event on lock y.
  (initially empty)
```

Set-based race predictor (code)

Each event invokes its processing function. We write `e@operation` for the processing function for event `e` of the form `operation`. We add thread information as an additional argument.

```
e@acq(t,y) {
   D(t) = D(t) U Rel(y) U { e }
}

e@rel(t,y) {
   D(t) = D(t) U { e }
   Rel(y) = D(t)

e@fork(t1,t2) {
  D(t1) = D(t1) U { e }
  D(t2) = D(t1)
}

e@join(t1,t2) {
  D(t1) = D(t1) U D(t2) U { e }
  }

e@write(t,x) {
  If W(x) exists and W(x) not in D(t)
  then write-write race (W(x),e)

  For each r in R(x),
  if r not in D(t)
  then read-write race  (r,e)

  D(t) = D(t) U { e }
  W(x) = e
}

e@read(t,x) {
  If W(x) exists and W(x) not in D(t)
  then write-read race (W(x),e)

  R(x) = {e} U (R(x) \ D(t))
  D(t) = D(t) cup { e }
}
```

Examples (Set-based data race detection)

Race detected

```
    T0            T1

1.  fork(T1)
2.  acq(y)
3.  wr(x)
4.  rel(y)
5.                wr(x)
6.                acq(y)
7.                rel(y)
```

According to the HB relation, the two writes on x are not ordered. Hence, we report that they are in a race.

Here is the annotated trace.

```
    T0            T1

1.  fork(T1)
2.  acq(y)                     D(T0) = [0:fork(T1)_1,0:acq(y)_2]
3.  wr(x)                      W(x) = undefined
4.  rel(y)                     Rel(y) = [0:fork(T1)_1,0:acq(y)_2,0:wr(x)_3,0:rel(y)_4]
5.                wr(x)        W(x) = 0:wr(x)_3
6.                acq(y)       D(T1) = [0:fork(T1)_1,0:acq(y)_2,0:wr(x)_3,0:rel(y)_4,1:wr(x
7.                rel(y)       Rel(y) = [0:fork(T1)_1,0:acq(y)_2,0:wr(x)_3,0:rel(y)_4,1:wr
```

Race not detected

```
    T0            T1

1.  fork(T1)
2.  acq(y)
3.  wr(x)
4.  rel(y)
5.                acq(y)
6.                rel(y)
7.                wr(x)
```

The HB relation does not reorder critical sections. We report that there is no race. This report is a false negative because there is a valid reordering that demonstrates that the two writes on x are in a race.

Here is the annotated trace with the information computed by the set-based race predictor.

```
   T0            T1

1.  fork(T1)
2.  acq(y)                      D(T0) = [0:fork(T1)_1,0:acq(y)_2]
3.  wr(x)                       W(x) = undefined
4.  rel(y)                      Rel(y) = [0:fork(T1)_1,0:acq(y)_2,0:wr(x)_3,0:rel(y)_4]
5.                acq(y)        D(T1) = [0:fork(T1)_1,0:acq(y)_2,0:wr(x)_3,0:rel(y)_4,1:acq(
6.                rel(y)        Rel(y) = [0:fork(T1)_1,0:acq(y)_2,0:wr(x)_3,0:rel(y)_4,1:acc
7.                wr(x)         W(x) = 0:wr(x)_3
```

- For each acquire in thread t we show D(t)
- For each release on y we show Rel(y)
- For each write on x we show the state of W(x) before processing the read
- If there are any reads we also show R(x)

Earlier race not detected

```
    T0            T1

1.  fork(T1)
2.  acq(y)
3.  wr(x)
4.  wr(x)
5.  rel(y)
6.                wr(x)
7.                acq(y)
8.                rel(y)
```

The write at trace position 3 and the write at trace position 6 are in a race. However, we only maintain the most recent write. Hence, we only report that the write at trace position 4 is in a race with the write at trace position 6.

Here is the annotated trace.

```
    T0             T1

1.  fork(T1)
2.  acq(y)                       D(T0) = [0:fork(T1)_1,0:acq(y)_2]
3.  wr(x)                        W(x) = undefined
4.  wr(x)                        W(x) = 0:wr(x)_3
5.  rel(y)                       Rel(y) = [0:fork(T1)_1,0:acq(y)_2,0:wr(x)_3,0:wr(x)_4,0:rel(
6.                 wr(x)         W(x) = 0:wr(x)_4
7.                 acq(y)        D(T1) = [0:fork(T1)_1,0:acq(y)_2,0:wr(x)_3,0:wr(x)_4,0:rel(y
8.                 rel(y)        Rel(y) = [0:fork(T1)_1,0:acq(y)_2,0:wr(x)_3,0:wr(x)_4,0:rel(
```

Read-write races

```
    T0             T1             T2

1.  wr(x)
2.  fork(T1)
3.  fork(T2)
4.  rd(x)
5.                 rd(x)
6.                                acq(y)
7.                                wr(x)
8.                                rel(y)
```

The two reads are in a race with the write in T2.

Here is the annotated trace.

```
    T0             T1             T2

1.  wr(x)                                        W(x) = undefined
2.  fork(T1)
3.  fork(T2)
4.  rd(x)                                        W(x) = 0:wr(x)_1
5.                 rd(x)                         W(x) = 0:wr(x)_1
6.                                acq(y)         D(T2) = [0:wr(x)_1,0:fork(T1)_2,0:fork(T2)_3,2
7.                                wr(x)          W(x) = 0:wr(x)_1  R(x) = [0:rd(x)_4,1:rd(x)_5]
8.                                rel(y)         Rel(y) = [0:wr(x)_1,0:fork(T1)_2,0:fork(T2)_3,
```

There is not write-write race because D(T2) contains the earlier 0:wr(x)_1 when processing the write in T2.

Can we find a more compact representation for D(t)?

The size of D(t) may grow linearly in the size of the trace.

To check for a race we check if some element is in D(t).

If there are n events, set-based race prediction requires $O(n^2)$ time.

From event sets to timestamps

Timestamps

Each thread maintains a timestamp.

We represent a timestamp as a natural number.

Each time we process an event we increase the thread's timestamp.

Initially, the timestamp for each thread is 1.

Example - Trace annotated with timestamps

```
     T1          T2      TS_T1        TS_T2

1.   w(x)                  2
2.   acq(y)                3
3.   rel(y)                4
4.               acq(y)                 2
5.               w(x)                   3
6.               rel(y)                 4
```

Representing an events by its thread id and timestamp

Let e be an event in thread t and j its timestamp.

Then, we can uniquely identify e via t and j.

We write $t\#j$ to represent event e. In the literature, $t\#j$ is called an *epoch*.

Example - Trace annotated with epochs

```
     T1          T2           Epochs

1.   w(x)                     1#1
2.   acq(y)                   1#2
3.   rel(y)                   1#3
4.               acq(y)       2#1
5.               w(x)         2#2
6.               rel(y)       2#3
```

We generallly use the timestamp "before" processing the event.

From event sets to sets of epoch

Instead of events sets we record the set of epochs.

We group together epochs belonging to the same thread. For example, in case of $\{1\#1, 1\#2\}$ we write $\{1\#\{1, 2\}\}$.

Example - Trace annotated with sets of epochs

```
     T1          T2             Sets of epochs

1.   w(x)                      {1#1}
2.   acq(y)                    {1#{1,2}}
3.   rel(y)                    {1#{1,2,3}}
4.               acq(y)        {1#{1,2,3}, 2#1}
5.               w(x)          {1#{1,2,3}, 2#{1,2}}
6.               rel(y)        {1#{1,2,3}, 2#{1,2,3}}
```

From sets of timestamps to vector clocks

Insight:

For each thread only keep most recent timestamp.

For example, in case of $\{1\#\{1, 2\}\}$ we write $\{1\#2\}$

Example - Trace annotated with most recent timestamps

```
     T1          T2             Sets of most recent timestamps

1.   w(x)                      {1#1}
2.   acq(y)                    {1#2}
3.   rel(y)                    {1#3}
4.               acq(y)        {1#3, 2#1}
5.               w(x)          {1#3, 2#2}
6.               rel(y)        {1#3, 2#3}
```

Vector clocks

A vector clock encodes a set of most recent timestamps from n threads as a vector of length n.

```
V  ::=  [i1,...,in]    -- vector clock with n time stamps
```

The entry i_t is the timestamp for thread t. The above vector clock V stands for the set of timestamps

```
V  ~~  {1#i1, ..., n#in}
```

We use the entry 0 to indicate missing information about a thread.

Example - Trace annotated with vector clocks

```
     T1          T2            Vector clocks

1.   w(x)                     [1,0]
2.   acq(y)                   [2,0]
3.   rel(y)                   [3,0]
4.               acq(y)       [3,1]
5.               w(x)         [3,2]
6.               rel(y)       [3,3]
```

Mapping event sets to vector clocks

Based on the above, each event set can be represented as the set $(\{1\#E_1, ..., n\#E_n\}$ where sets E_j are of the form $\{1, ..., k\}$.

We define a mapping Φ from event sets to vector clocks as follows.

$\Phi(\{1\#E_1, ..., n\#E_n\}) = [max(E_1), ..., max(E_n)]$

Properties

1. $D_e \subset D_f$ iff $\phi(D_e) < \phi(D_f)$
2. $\phi(D_e \cup D_f) = sync(\phi(D_e), \phi(D_f))$
3. Let e, f be two events where e appears before f and $e = t\#k$. Then, $e \notin D_f$ iff $k > \Phi(D_f)(t)$.

We define $<$ and *sync* for vector clocks as follows.

```
Synchronize two vector clocks by taking the larger time stamp

     sync([i1,...,in],[j1,...,jn]) = [max(i1,j1), ..., max(in,jn)]

Strict order on vector clocks

      [i1,...,in]  < [j1,...,jn])

if ik<=jk for all k=1...n and there exists k such that ik<jk.
```

FastTrack, a race detector based on vector clocks

Further vector clock operations.

```
Lookup of time stamp

   [i_1,...,i_t,...,i_n](t) = i_t


Increment the time stamp of thread t

    inc([...,i_t,...],t) = [...,i_t+1,...]
```

We maintain the following state variables.

```
Th(t)

  Vector clock of thread t

R(x)

  Vector clock for reads from x

W(x)

  Epoch of most recent write on x


Rel(y)

  Vector clock of the most recent release on lock y.
```

Initially, the timestamps in R(x), W(x), and Rel(y) are all set to zero.

In Th(t), all time stamps are set to zero except the time stamp for entry t, which is set to one.

Event processing is as follows.

```
acq(t,y) {
  Th(t) = sync(Th(t), Rel(y))
  inc(Th(t),t)
}

rel(t,y) {
   Rel(y) = Th(t)
   inc(Th(t),t)
}

fork(t1,t2) {
  Th(t2) = sync(Th(t1), Th(t2))
  inc(Th(t1),t1)
}
```

```
join(t1,t2) {
  Th(t1) = sync(Th(t1),Th(t2))
  inc(Th(t1),t1)
  }

write(t,x) {
  If not (R(x) < Th(t))
  then write-read race detected

  If W(x) != 0
  then let j#k = W(x)
       if k > Th(t)(j)
       then write-write race detected
  W(x) = t#Th(t)(t)
  inc(Th(t),t)
}

read(t,x) {
  If W(x) != 0
  then let j#k = W(x)
       if k > Th(t)(j)
       then read-write race detected
  R(x) = sync(Th(t), R(x))
}
```

Examples (from before)

- For each event we annotate its vector clock
- For each write we show the state of W(x) before processing the read
- If there are any reads we also show R(x)

Race detected

```
    T0            T1

1.  fork(T1)                    [1,0]
2.  acq(x)                      [2,0]
3.  wr(z)                       [3,0]  W(z) = undefined
4.  rel(x)                      [4,0]
5.                wr(z)         [1,1]  W(z) = 0#3
6.                acq(x)        [1,2]
7.                rel(x)        [4,3]
```

Race not detected

```
    T0            T1

1.  fork(T1)                    [1,0]
2.  acq(x)                      [2,0]
3.  wr(z)                       [3,0]  W(z) = undefined
4.  rel(x)                      [4,0]
5.                acq(x)        [1,1]
6.                rel(x)        [4,2]
7.                wr(z)         [4,3]  W(z) = 0#3
```

Earlier race not detected

```
    T0            T1

1.  fork(T1)                    [1,0]
2.  acq(y)                      [2,0]
3.  wr(x)                       [3,0]  W(x) = undefined
4.  wr(x)                       [4,0]  W(x) = 0#3
5.  rel(y)                      [5,0]
6.                wr(x)         [1,1]  W(x) = 0#4
7.                acq(y)        [1,2]
8.                rel(y)        [5,3]
```

Read-write races

```
    T0            T1            T2

1.  wr(x)                                     [1,0,0]  W(x) = undefined
2.  fork(T1)                                  [2,0,0]
3.  fork(T2)                                  [3,0,0]
4.  rd(x)                                     [4,0,0]
5.                rd(x)                       [2,1,0]
6.                              acq(y)        [3,0,1]
7.                              wr(x)         [3,0,2]  W(x) = 0#1  R(x) = [0#4,1#1]
8.                              rel(y)        [3,0,3]
```

Summary

False negatives

The HB method has *false negatives* because the textual order between critical sections is preserved. That is, valid reorderings that swap critical sections are ignored.

Consider the following trace.

```
     T1          T2

1.   w(x)
2.   acq(y)
3.   rel(y)
4.               acq(y)
5.               w(x)
6.               rel(y)
```

We derive the following HB relations.

The program order condition implies the following ordering relations:

$w(x)_1 < acq(y)_2 < rel(y)_3$

and

$acq(y)_4 < w(x)_5 < rel(y)_6$.

The critical section order condition implies

$rel(y)_3 < acq(y)_4$.

By transitivity, we conclude that

$w(x)_1 < w(x)_5$.

How? From above we find that

$w(x)_1 < acq(y)_2 < rel(y)_3$ and $rel(y)_3 < acq(y)_4$.

By transitivity we find that

$w(x)_1 < acq(y)_4$.

In combination with $acq(y)_4 < w(x)_5 < rel(y)_6$ and transitivity we find that $w(x)_1 < w(x)_5$.

Because of $w(x)_1 < w(x)_5$, the HB method concludes that there is no data race because the conflicting events $w(x)_1$ and $w(x)_5$ are ordered.

This report is a false negative as demonstrated by the following valid reordering.

```
     T1          T2

4.               acq(y)
5.               w(x)
1.   w(x)
```

We first execute events from thread T2 and find that the two conflicting writes are in a data race.

False positives

The first race reported by the HB method is an "actual" race. However, subsequent races may be false positives.

Consider the program

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

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

    // Thread T2 = Main Thread
    if y == 2 {
        x = 3
    }

}
```

We consider a specific execution run that yields the following trace.

```
Trace I:

     T1             T2

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

We encounter a write-read race because $w(y)_2$ and $r(y)_3$ appear right next to each other in the trace.

It seems that there is also a HB write-write data race. The HB relations derived from the above trace are as follows:

$w(x)_1 < w(y)_2$ and $r(y)_3 < w(x)_4$.

Hence, $w(x)_1$ and $w(x)_4$ are unordered. Hence, we find the write-write data race $(w(x)_4, w(x)_1)$.

We reorder the above trace (while maintaining the program order HB relations). For the reordered trace we keep the original trace positions.

```
Trace II:
     T1            T2

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

In the reordered trace II, the two writes on x appear right next to each other. But there is no program run that yields the above reordered trace!

In the reordered trace II, we violate the write-read dependency between $w(y)_2$ and $r(y)_3$. $w(y)_2$ is the last write that takes place before $r(y)_3$. The read value y influences the control flow. See the above program where we only enter the if-statement if $w(y)_2$ takes place (and sets y to 2).

We conclude that the HB relation does not take into account write-read dependencies and therefore HB data races may not correspond to *actual* data races.

We say *may not* because based on the trace alone we cannot decide if the write-read dependency actually affects the control flow.

For example, trace I could be the result of a program run where we assume the following program.

```go
func example4() {
    var tmp int
    x := 1
    y := 1

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

    // Thread T2 = Main Thread
    tmp = y // READ
    x = 3

}
```

There is also a write-read dependency, see locations marked WRITE and READ. However, the read value does not influence the control flow. Hence, for the above program trace II would be a valid reordering of trace I.

Further reading

What Happens-After the First Race?

Shows that the “first” race reported by Lamport’s happens-before relation is sound.

ThreadSanitizer

C/C++ implementation of Lamport’s happens-before relation (to analyze C/C++).

Go’s data race detector

Based on ThreadSanitizer.