Dynamic data race prediction - Locksets

Martin Sulzmann

The Lockset Method

The **lockset** is the set of locks that are held when processing a read/write event.

It can be used for several purposes.

Data race check based on locksets

If two conflicting events hold the same lock y, then both events must belong to two distinct critical sections involving lock y. As critical sections are mutually exclusive, two conflicting events that share the same lock cannot be in a data race.

Hence, if the locksets of two conflicting events e and f are disjoint, then (e, f) is a *Lockset data race pair*.

Critical section

A critical section is identified by a pair of matching $acq(y)$ and $rel(y)$ events.

To define matching acquire/release pairs, we attach thread ids and trace positions to events. We write $t\#e_k$ to denote some event e in thread t at trace position k. Recall Trace and event notation.

Consider acquire event $t_1\#acq(y)_k$ and release event $t_2\#rel(y)_l$.

We say that $(t_1\#acq(y)_k, t_2\#rel(y)_l)$ is a *matching acquire/release pair* if

1. $t_1 = t_2$, and
2. $k < l$ and there is no $t\#rel(y)_m$ where $k < m < l$.

The first condition states that $acq(y)$ and $rel(y)$ belong to the same thread. The second condition states that there is no other $rel(y)$ between $acq(y)_k$ and $rel(y)_l$.

We write $CS(t\#acq(y)_k, t\#rel(y)_l)$ to denote the set of events that are part of the *critical section* for a matching acquire/release pair $(t\#acq(y)_k, t\#rel(y)_l)$. We only accept events from the same thread t.

We write $t'\#e_m \in CS(t\#acq(y)_k, t\#rel(y)_l)$ if $t = t'$ and $k \leq m \leq l$.

Hence, $acq(y)$ and $rel(y)$ are part of the critical section; and any event in the same thread between them is also part of this critical section.

Lockset

Let e an event. The *lockset* of e, written $LS(e)$, consists of all ys such that e appears in a critical section belonging to lock y. More formally, we define $LS(e) = \{y \mid \exists a = t\#acq(y)_k, r = t\#rel(y)_l.e \in CS(a,r)\}$.

Example

Recall the earlier trace.

```
Trace A:

     T1          T2

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

Trace A contains two critical sections for lock variable y.

Thread $T1$ contains the critical section $CS(T1\#acq(y)_2, T1\#rel(y)_3)$.

Thread $T2$ contains the critical section $CS(T2\#acq(y)_4, T2\#rel(y)_6)$.

We consider the locksets of events $w(x)_1$ and $w(x)_5$.

$LS(w(x)_1) = \{\}$.

$LS(w(x)_5) = \{y\}$.

The two locksets are disjoint because $LS(w(x)_1) \cap LS(w(x)_5) = \{\}$. Hence, the conflicting events $w(x)_1$ and $w(x)_5$ represent a data race.

Lockset summary and limitations

The lockset method is complete, because any conflicting pair of events that represent a data race can be shown to be a lockset data race pair. However, the lockset method is unsound.

Like HB, the lockset method ignores write-read dependencies (and therefore the earlier HB unsoundness example also applies to lockset). There is a further reason for unsoundness because lockset enables reordering of critical sections. By reordering critical sections (to exhibit the data race) we may run into a deadlock.

Consider the following trace.

```
     T1         T2

1.   acq(y1)
2.   acq(y2)
3.   rel(y2)
4.   w(x)
5.   rel(y1)
6.              acq(y2)
7.              acq(y1)
8.              rel(y1)
9               w(x)
10.             rel(y2)
```

There are two lock variables y_1 and y_2. There are two conflicting events $w(x)_4$ and $w(x)_9$. Their lockset is as follows.

$LS(w(x)_4) = \{y_1\}$.

$LS(w(x)_9) = \{y_2\}$.

Based on the lockset data race check, we argue that $w(x)_4$ and $w(x)_9$ represents a data race. But is this an actual data race? No!

The reason is that there is no valid trace reordering (of the above trace) under which the two writes on x appear right next to each other. We prove this statement by contradiction.

Suppose, there exists a valid trace reordering. For example, $\ldots, w(x)_4, w(x)_5$. As the program order must remain intact, the events in thread T1 must appear before $w(x)_4$ and the events in thread T2 must appear before $w(x)_5$. But that means, thread T1 acquired locks y_1 and y_2 and the same applies to thread T2! This is impossible (and if we would try we would run into a deadlock).

Hence, the lockset method is unsound and the above is an example of a false positive.

Comparing HB and Lockset

- The lockset method is complete but unsound.
- The HB method is incomplete and unsound.

In practice, it appears that the lockset method gives rise to significantly more false positives than the HB method.

One can combine the HB and lockset method to achieve Efficient, Near Complete and Often Sound Hybrid Dynamic Data Race Prediction (extended version).

Lockset computation

We maintain the following state variables.

```
ls(t) : Set(Lock)

  The set of locks held by thread t at a certain time.

LS : Event -> Set(Lock)

  A mapping from an event e to its lockset.
```

We write `e@operation` to denote that event `e` will be processed by `operation`.

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

e@rel(t,y) {
 ls(t) = ls(t) - {y}

e@fork(t1,t2) {
}

e@join(t1,t2) {
}

e@write(t,x) {
  LS(e) = ls(t)
}

e@read(t,x) {
  LS(e) = ls(t)
}
```

For sets S1 and S2 we write S1 - S2 for the set difference (the set that contains all elements in S1 that are not in S2).

We only record the lockset for write and read events.

We also cover fork and join events. As we can see, the computation of locksets is agnostic to the presence of fork and join events.

Observation

Unlike Lamport's happens-before that is sensitive to the order of critical sections, the computation of locksets is not affected if we reorder critical sections.

Consider trace A.

```
     T1          T2

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

It holds that $LS(e1) = \{\}$ and $LS(e5) = \{y\}$.

For the following (valid) reordering

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

It holds that $LS(e1) = \{\}$ and $LS(e5) = \{y\}$.

Further examples

We annotate the trace with lockset information.

Example 1

```
      T0      T1         Lockset
1.   acq(y)
2.   w(x)               {y}
3.   rel(y)
4.   r(x)               {}
5.           w(x)       {}
6.           acq(y)
7.           rel(y)
```

The locksets of the two writes on `x` in thread T0 and T1 are disjoint. Hence, the lockset method reports a data race.

Example 2

```
      T0      T1         Lockset
1.   w(x)               {}
2.   acq(y)
3.   w(x)               {y}
4.   rel(y)
```

```
5.             acq(y)
6.             w(x)      {y}
7.             rel(y)
```

The lockset of the write at trace position 1 and the write at trace position 6 are disjoint. Hence, we expect that the lockset method signals a data race.

To be efficient, an implementation based on the lockset method only keeps track of the most recent locksets. That is, each thread maintains a list of the most recent reads/writes and their locksets.

Applied to the above example, we encounter the following behavior.

- Thread T0 processes $w(x)_1$ and records $LS(w(x)) = \{\}$.
- Once thread T0 processes $w(x)_3$ it records $LS(w(x)) = \{y\}$.
- Hence, the history of earlier locksets for writes on x is lost.
- Hence, algorithms that only record locksets for most recent writes/reads will not signal a data race for this example.

Example 3

```
      T0      T1        Lockset
1.   w(x)              {}
2.   acq(y)
3.   rel(y)
4.           acq(y)
5.           w(x)      {y}
6.           rel(y)
```

The locksets are disjoint. Hence, the algorithm signals a data race.

However, in the actual program, thread T0 forks thread T1. We assume that after the release at trace position 3 there is a go statement to create thread T1. For example, the above trace could result from the following program.

```go
x = 3
acq(y)
rel(y)

go func() {
    acq(y)
    x = 4
    rel(y)
}()
```

This “fork” information is not recorded in the trace. As we only compare locksets, we encounter here another case of a false positive.

Example 4

```
      T0      T1      T2        Lockset
1.   acq(y)
2.           w(x)              {}
3.   rel(y)
4.                   acq(y)
5.                   w(x)      {y}
6.                   rel(y)
```

Shouldn't the lockset at trace position 2 include lock variable y!?

No!

- The write at trace position 2 seems to be protected by lock variable y.
- However, thread T1 does not "own" this lock variable!