Dynamic deadlock prediction

Martin Sulzmann, Peter Thiemann

2024-05-29

Overview

Our assumptions are:

- Concurrent programs making use of acquire and release (mutex operations).
- A single program run where events that took place are recorded in some trace.

Goal:

- Analyze the trace to check if we might run into a deadlock.

The literature distinguishes between *resource* and *communication* deadlocks.

A **resource deadlock** arises if

- a set of threads are blocked, and
- each thread in the set is waiting to acquire a lock held by another thread in the set.

Communication deadlocks arise if threads are blocked due to (missing) channel events such as send and receive.

Here, we only consider resource deadlocks. Whenever we say deadlock we refer to a resource deadlock.

Lock graphs

We introduce a well-established deadlock prediction method based on *lock graphs*. The basic idea is as follows:

- Locks are nodes.
- There is an edge from lock x to lock y if a thread holds locks x while it acquires lock y. This can be calculated based on lock sets.
- Check for cycles in the lock graph.
- If there is a cycle, we report that there is a potential deadlock.

We will also consider an improved representation based on *lock dependencies*.

Literature

- Using Runtime Analysis to Guide Model Checking of Java Programs (Havelund, 2000)

Introduces lock trees. Each thread maintains its own sequence of locking steps. This approach is known under the name the *Goodlock* approach. Goodlock can only deal with two threads.

- Detecting Potential Deadlocks with Static Analysis and Run-Time Monitoring (Argawal, Wang and Stoller, 2005)

Introduces lock graphs, a generalization of lock trees. Lock graphs are capable of checking for deadlocks among three threads and more.

- A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks (Joshi, Park, Sen and Naik, 2009)

Introduces lock dependencies.

Lock graph based dynamic deadlock prediction

We explain the idea of dynamic deadlock prediction based on lock graphs via several examples. The lock graph construction is as follows.

- Locks are nodes.
- There is an edge from lock x to lock y if a thread holds locks x while it acquires lock y.

Consider the example.

```
    T1          T2

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

Based on the above trace, the program runs through without any apparent problem.

For the above trace, we find the following lock graph.

```
y -> x
x -> y
```

That is, we may run into a situation where

- some thread holds lock y and acquires lock x,
- some thread holds lock x and acquires lock y.

This may lead to a deadlock!

Consider the following reordering of the above trace.

```
    T1          T2

1.  acq(y)
5.              acq(x)
2.  B-acq(x)
6.              B-acq(y)
```

We write `B-acq` to denote that the acquire is blocked.

As we can see, all threads are blocked => deadlock!

Lock graph deadlock criteria:

If the lock graph contains a cycle, then there is a potential deadlock.

Precision (false positives)

How *precise* is the analysis method based on lock graphs?

Same thread

Consider the following example where we also record the lockset for each thread.

```
    T1          T2      LS_T1      LS_T2

 1.  acq(y)              {y}
 2.  acq(x)              {y,x}                 yields y -> x
 3.  rel(x)              {y}
 4.  rel(y)              {}
 5.  acq(x)              {x}
 6.  acq(y)              {x,y}                 yields x -> y
 7.  rel(y)              {x}
 8.  rel(x)              {}
```

We encounter the same (cyclic) lock graph that we have seen for some earlier example.

```
y -> x
x -> y
```

In this case, it is a false positive because all events take place in the *same thread*.

Guard locks

Here is another example.

```
    T1            T2           T1_LS          T2_LS

 1.  acq(z)                     z
 2.  acq(y)                     z,y                      implies z -> y
 3.  acq(x)                     z,y,x                    implies y -> x  z -> x
 4.  rel(x)                     z,y
 5.  rel(y)                     z
 6.  rel(z)
 7.             acq(z)                      z
 8.             acq(x)                      z,x          implies z -> x
 9.             acq(y)                      z,x,y        implies x -> y  z -> y
 10.            rel(y)                      z,x
 11.            rel(x)                      z
 12.            rel(z)
```

In summary, we obtain the following lock graph.

```
z -> y
y -> x
z -> x
z -> x
x -> y
z -> y
```

from which we can derive the following cycle

```
y -> x
x -> y
```

This case is again a false positive because there is a common *guard lock* z.

Lock dependencies

Instead of a lock graph we compute *lock dependencies* on a per thread basis. A lock dependency $D = (t, l, ls)$ is constructed if thread t acquires lock l while holding locks ls. Thus, we can eliminate some of the false positives that arise using lock graphs (but not all).

Computation

We compute lock dependencies by processing each event in the trace (in the order events are recorded).

We maintain the following state variables.

```
ls(t)

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

Ds

  The set of lock dependencies.
```

For each event we call its processing function.

```
acq(t,y) {
    Ds = Ds U { (t,y,ls(t)) }  if ls(t) != emptyset
    ls(t) = ls(t) U {y}
}

rel(t,y) {
    ls(t) = ls(t) \ {y}
}

fork(t1,t2) {
}

...
```

We write `S1 \ S2` to denote set difference: `S1 \ S2` contains all elements in `S1` that are not in `S2`.

Cycle check

We write $D = (id, l, ls)$ to refer to some lock dependency in thread id where lock l is acquired while holding locks ls.

A deadlock (warning) is issued if there is a cyclic lock dependency chain $D_1, \ldots, D_n$:

- (LD-1) $ls_i \cap ls_j = \emptyset$ for $i \neq j$, and
- (LD-2) $l_i \in ls_{i+1}$ for $i = 1, ..., n-1$, and
- (LD-3) $l_n \in ls_1$.

Note: each D_i results from some distinct thread i.

Examples

Same thread

Recall

```
   T1          T2      LS_T1      LS_T2

 1.  acq(y)             {y}
 2.  acq(x)             {y,x}                 yields (T1,x,{y})
 3.  rel(x)             {y}
 4.  rel(y)             {}
 5.  acq(x)             {x}
 6.  acq(y)             {x,y}                 yields (T1,y,{x})
 7.  rel(y)             {x}
 8.  rel(x)             {}
```

In summary, we find the following lock dependencies.

```
(T1,x,{y})
(T1,y,{x})
```

No deadlock warning is issued because the dependencies are from the same thread T1.

Guard lock

Recall

```
   T1             T2            T1_LS           T2_LS

 1.  acq(z)                      z
 2.  acq(y)                      z,y                          (T1,y,{z})
 3.  acq(x)                      z,y,x                        (T1,x,{y,z})
 4.  rel(x)                      z,y
 5.  rel(y)                      z
 6.  rel(z)
 7.             acq(z)                        z
 8.             acq(x)                        z,x            (T2,x,{z})
 9.             acq(y)                        z,x,y          (T2,y,{x,z})
 10.            rel(y)                        z,x
 11.            rel(x)                        z
 12.            rel(z)
```

For the above, we compute the following lock dependencies.

```
D1 = (T1, y, {z})
D2 = (T1, x, {y,z})

D3 = (T2, x, {z})
D4 = (T2, y, {x,z})
```

No deadlock warning is issued due to the common guard lock z.

Precision (false positives and false negatives)

Lock dependencies improve over lock graphs but still suffer from false positives and false negatives.

False positive

Consider the following example.

```
    T1          T2

1.  acq(y)
2.  acq(x)
3.  rel(x)
4.  rel(y)
5.  acq(z)
6.  wr(a)
7.  rel(z)
8.              acq(z)
9.              rd(a)
10.             rel(z)
11.             acq(x)
12.             acq(y)
13.             rel(y)
14.             rel(x)
```

We encounter the following lock dependencies.

```
D1 = (T1, x, {y})

D2 = (T2, y, {x})
```

We issue a deadlock warning because D1, D2 form a cyclic lock dependency chain.

A false positive! Due to the write-read dependency, events in thread T1 must happen before the events in thread T2.

False negatives

Consider the following example.

```
     T1          T2         T3

1.  acq(x)
2.  fork(T2)
3.              acq(y)
4.              rel(y)
5.  join(T2)
```

```
6.  rel(x)
7.                          acq(y)
8.                          acq(x)
9.                          rel(x)
10.                         rel(y)
```

We compute the following dependency.

```
D1 = (T3,x,{y})
```

There appears to be no deadlock (no cycle) but there is a trace reordering under which we run into a deadlock.

```
     T1          T2          T3

1.  acq(x)
2.  fork(T2)
7.                          acq(y)
8.                          B-acq(x)
3.              B-acq(y)
5.  B-join(T2)
```

The above is an example of a *cross-thread* critical section that includes several events due to the fork/join dependency. However, the computation of lock dependencies is agnostic to cross-thread critical sections. This ignorance may lead to false negatives as shown above.

Go-style mutexes behave like semaphores

So far, we assumed that a thread that acquires lock x must also release x. This assumption holds for Java and C++. However, Go-style mutexes behave differently as they act more like semaphores. In Go, the acquire and release of a lock is not tied to a single thread.

Consider the following example.

```
    T1        T2          T3

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

Thread T1 acquires lock x but this lock is released by thread T2. A similar observation applies to lock y and threads T2 and T3.

Go-style mutexes pose new challenges for lock graph based deadlock prediction.

Consider the following example.

```
    T1          T2          T3

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

If we apply the lock graph method we find the following graph

```
y -> x

x -> y
```

There is a cyclic dependency and therefore we issue a deadlock warning. This is a false positive as there is no valid reordering under which threads T1 and T3 remain stuck.

For example, consider the following reordering.

```
    T1          T2          T3

1.  acq(y)
5.                          acq(x)

At this point we are not stuck, because thread T2 can continue

3.              rel(y)

and therefore thread T3 continues

6.                          acq(y)

and so on.
```

Similar arguments apply to the other reorderings.

We conclude:

- Go-style mutexes behave like semaphores
- Deadlock prediction based on lock graph results in further false positives

ThreadSanitizer (TSan)

Lockgraph based deadlock detector. Not much is known about what has been implemented. Certainly a fairly naive implementation.

```
void foo() {
    mutexA.lock();
    mutexB.lock();
    mutexB.unlock();
    mutexA.unlock();
}

void bar() {
    mutexB.lock();
    mutexA.lock();
    mutexA.unlock();
    mutexB.unlock();
    foo();
}

int main() {
    bar();
}
```

TSan reports a deadlock but this is a clear false positive.

Summary

- Dynamic deadlock prediction based on lock graphs.
- Improved representation based on lock dependencies.
- False positives and false negatives remain an issue.
- Go-style mutexes pose further challenges (false positives).
- Ongoing research to reduce the amount of false positives and false negatives.

Appendix: Some implementation in Go

Here is a simple implementation for dynamic deadlock prediction based on lock graphs.

We consider "standard" mutex operations where we assume the thread that acquires a lock must also release the lock.

```
package main

import "fmt"
```

```go
import "time"



// Set of integers.
// We use map[int]bool to emulate sets of integers.
// The default value for bool is true.
// Hence, if the (integer) key doesn't exist, we obtain false = element not part of the set
// In case of add and remove, we use pointers to emulate call-by-reference.

type Set map[int]bool

func mkSet() Set {
    return make(map[int]bool)
}

func empty(set Set) bool {
    return len(set) == 0
}

func elem(set Set, n int) bool {
    return set[n]
}

func add(set Set, n int) Set {
    s := set
    s[n] = true
    return s
}

//  union(a,b) ==> c,true
//    if there's some element in b that is not an element in a
//  union(a,b) ==> c,false
//    if all elements in b are elements in a
func union(a Set, b Set) (Set, bool) {
    r := true
    for x, _ := range b {
        if !elem(a, x) {
            r = false
            a = add(a, x)
        }
    }
    return a, r

}
```

```
func remove(set Set, n int) Set {
    s := set
    s[n] = false
    return s
}

// Mutex
type Mutex chan int

func newMutex() Mutex {
    var ch = make(chan int, 1)
    return ch
}

func lock(m Mutex) {
    m <- 1
}

func unlock(m Mutex) {
    <-m
}

// User interface
type M interface {
    init()
    acquire(tid int, n int)
    release(tid int, n int)
    check() bool
    info()
}

// Each mutex is identified via a natural number starting with 0.
// We support MAX_MUTEX number of mutexes.
const MAX_MUTEX = 3

// Each thread is identified via natural number starting with 0.
// We support MAX_TID number of threads.
const MAX_TID = 4

// We represent the lock graph via an adjacency matrix.
// Edge x --> y is represented via edge[x][y] = true.
// Thread i adds the edge x --> y if
// thread i holds the lock x while acquiring the lock y.
type LG struct {
    lockset [MAX_TID]Set
    edge    [MAX_MUTEX][MAX_MUTEX]bool
```

```
    mutex   [MAX_MUTEX]Mutex
    g       Mutex
}

// We use pointer receivers as we update some of the internal structs.
// Interface M is implemented by *LG.

func (m *LG) init() {
    for i := 0; i < MAX_TID; i++ {
        m.lockset[i] = mkSet()
    }

    for i := 0; i < MAX_MUTEX; i++ {
        for j := 0; j < MAX_MUTEX; j++ {
            m.edge[i][j] = false
        }

    }

    for i := 0; i < MAX_MUTEX; i++ {
        m.mutex[i] = newMutex()

    }

    m.g = newMutex()

}

func (m *LG) info() {

    fmt.Printf("\n *** INFO ***")
    lock(m.g)

    for i := 0; i < MAX_TID; i++ {
        fmt.Printf("\n Thread %d holds the following locks:", i)
        for j := 0; j < MAX_MUTEX; j++ {
            if elem(m.lockset[i], j) {
                fmt.Printf(" %d", j)
            }
        }
    }

    fmt.Printf("\n Lock graph:")
    for i := 0; i < MAX_MUTEX; i++ {
        for j := 0; j < MAX_MUTEX; j++ {
            if m.edge[i][j] {
```

```
                fmt.Printf("\n  %d --> %d", i, j)
            }
        }

    }

    unlock(m.g)

}

// 1. Acquire the actual lock.
// 2. Update the lock graph.
// 3. Update the lockset.
// Optimization note:
//   There's no need to iterate over all the elements in the lockset.
//   It's sufficient to add an edge x -> n where
//   x is the 'most recent' lock added.
///  Example:
//    Consider the lock graph x -> y, y -> z where
//    via the transitive closure we obtain x -> z.
//    Hence, there's no need to add the edge x -> z.
func (m *LG) acquire(tid int, n int) {
    lock(m.mutex[n])

    lock(m.g)
    for i := 0; i < MAX_MUTEX; i++ {
        if elem(m.lockset[tid], i) {
            m.edge[i][n] = true
        }
    }

    m.lockset[tid] = add(m.lockset[tid], n)

    unlock(m.g)
}

// 1. Update the lockset.
// 2. Release the actual lock.
func (m *LG) release(tid int, n int) {

    lock(m.g)
    m.lockset[tid] = remove(m.lockset[tid], n)
    unlock(m.g)

    unlock(m.mutex[n])
```

```
}

// Check for cycles in the lock graph.
func (m *LG) check() bool {

    directNeighbors := func(m LG, x int) Set {
        next := mkSet()

        for y := 0; y < MAX_MUTEX; y++ {
            if m.edge[x][y] {
                next = add(next, y)
            }
        }
        return next
    }

    // Cycle check for a specific node x by breadth-first traversal.
    // 1. Builds the set of all neighbors of neighbors and so on starting
    // with a specific node x.
    // 2. Check if we encounter/visit x.
    checkCycle := func(m LG, x int) bool {
        visited := mkSet()
        goal := directNeighbors(m, x)
        stop := false

        for !stop {
            new_goal := mkSet()
            for y, _ := range goal {
                new_goal, _ = union(new_goal, directNeighbors(m, y))
            }

            visited, stop = union(visited, goal)
            goal = new_goal

        }

        if elem(visited, x) {
            return true
        }
        return false
    }

    // Cycle check for each node.
    r := false

    lock(m.g)
```

```
    for x := 0; x < MAX_MUTEX; x++ {
        r = r || checkCycle(*m, x)

    }

    unlock(m.g)

    if r {
        fmt.Printf("\n *** cycle => potential deadlock !!! ***")
    }

    return r

} // check

// Examples

// Program with a potential deadlock.
func example1(m M) {
    // Introduce some short-hands.
    x := 0
    y := 1
    t0 := 0
    t1 := 1
    // MUST call init at the start!
    m.init()

    // If we include the command below, it appears that often
    // the deadlock won't be detected.
    // m.info()

    // Helper
    // acq(a);acq(b);rel(b);rel(a)
    acqRel2 := func(t int, a int, b int) {
        m.acquire(t, a)
        m.acquire(t, b)
        m.release(t, b)
        m.release(t, a)

    }

    // T0
    go acqRel2(t0, x, y)

    // T1
```

```
    acqRel2(t1, y, x)

    m.info()
    m.check()

    // m.info()

}

// Program with no deadlock
// but our deadlock detector may signal "potential deadlock".
func example2(m M) {
    x := 0
    y := 1
    t0 := 0
    t1 := 1
    m.init()
    ch := make(chan int)

    acqRel2 := func(t int, a int, b int) {
        m.acquire(t, a)
        m.acquire(t, b)
        m.release(t, b)
        m.release(t, a)

    }
    // T0
    go func() {
        acqRel2(t0, x, y)
        ch <- 1
    }()

    // T1
    <-ch
    acqRel2(t1, y, x)

    m.info()
    m.check()

    // m.info()

}

// Program with a potential deadlock where three threads are involved in.
func example3(m M) {
    x := 0
```

```
    y := 1
    z := 2

    t0 := 0
    t1 := 1
    t2 := 2

    m.init()

    acqRel2 := func(t int, a int, b int) {
        m.acquire(t, a)
        m.acquire(t, b)
        m.release(t, b)
        m.release(t, a)
    }

    // T0
    go acqRel2(t0, x, y)

    // T1
    go acqRel2(t1, y, z)

    // T2
    acqRel2(t2, z, x)

    m.info()
    m.check()
    time.Sleep(1 * time.Second)

}

// Program with a potential deadlock.
// There are two threads and three locks.
// The deadlock involves only two of the three locks.
func example4(m M) {
    x := 0
    y := 1
    z := 2

    t0 := 0
    t1 := 1

    m.init()

    // T0
    go func() {
```

```
        m.acquire(t0, x)
        m.acquire(t0, y)
        m.acquire(t0, z)
        m.release(t0, z)
        m.release(t0, y)
        m.release(t0, x)
    }()

    // T1
    m.acquire(t1, z)
    m.acquire(t1, x)
    m.release(t1, x)
    m.release(t1, z)

    m.info()
    m.check()
    time.Sleep(1 * time.Second)

}

// Program with no deadlock
// but our deadlock detector may signal "potential deadlock".
func example5(m M) {
    x := 0
    y := 1
    z := 2

    t0 := 0
    t1 := 1

    m.init()

    // T0
    go func() {
        m.acquire(t0, x)
        m.acquire(t0, y)
        m.acquire(t0, z)
        m.release(t0, z)
        m.release(t0, y)
        m.release(t0, x)
    }()

    // T1
    m.acquire(t1, x)
    m.acquire(t1, z)
    m.acquire(t1, y)
```

```
    m.release(t1, y)
    m.release(t1, z)
    m.release(t1, x)

    m.info()
    m.check()
    time.Sleep(1 * time.Second)

}

func main() {
    var lg LG

    // example1(&lg)
    // example2(&lg)
    // example3(&lg)
    // example4(&lg)
    example5(&lg)

}

// NOTES

/*

 Require a (global) lock to avoid data races during tracing and checking.
 For example,

  acquires "writes" to the lock graph, and
  check "reads" from the lock graph

  releases "writes" to the lockset, and
  info "reads" from the lockset.


 If check and info won't happen concurrently to tracing the program,
 we can simplify the instrumentation for acquire and release.
 There's no need for lock(m.g) and unlock(m.g) instructions for the following reasons.

 Consider acquire for thread tid and lock n.
  1. We first acquire its actual lock.
  2. Hence, there can't be any concurrent access to m.lockset[tid] and m.edge[i][n].

 Consider release for thread tid and lock n.
  1. We first update m.lockset[tid] before
  2. releasing the actual lock.
```

```
  Hence, there can't be any concurrent access to m.lockset[tid].

*/

// Further Observations

// -- race for example1 sometimes leads to a stalled program run
```