Sound Dynamic Deadlock Prediction in Linear Time

Florian Rudaj

Agenda

- Einleitung
- Die Vorhersage von Deadlocks
- Dynamische Deadlock-Analyse
- Sync-preserving Deadlocks
- Fazit

Einleitung

- In nebenläufigen Programmen müssen Ressourcen geteilt werden
- Gleichzeitige Zugriffe auf geteilte Ressourcen können zu Inkonsistenzen führen
- Mutex und Locks dieser als Lösung

Einleitung

- Deadlocks sind schwer zu verhindern
- Aber daraus entstehende Problematik: Deadlocks
- Deadlocks sind schwer reproduzierbar und damit schwer zu debuggen

```go
func simple_deadlock() {
    x := 0
    y := 0
    var xMutex sync.Mutex
    var yMutex sync.Mutex
    go func() {
        xMutex.Lock()
        yMutex.Lock()
        x, y = doWork(x, y)
        yMutex.Unlock()
        xMutex.Unlock()
    }()
    yMutex.Lock()
    xMutex.Lock()
    x, y = doOtherWork(x, y)
    xMutex.Unlock()
    yMutex.Unlock()
}
```

Die Vorhersage von Deadlocks

- Deadlocks müssen vorhergesagt werden, um Programmabstürze zu verhindern
- Verschiedene Ansätze statische und dynamische Deadlock-Analyse

Die Vorhersage von Deadlocks

- Statische Deadlock-Analyse
  - kann Abwesenheit von Deadlocks beweisen, aber nicht gut skalierbar und liefert False Positives
- Dynamische Deadlock-Analyse
  - Effizienter
  - Liefert wenige oder keine False-Positives

Dynamische Deadlock-Analyse

```go
func simple_deadlock() {
    x := 0
    y := 0
    var xMutex sync.Mutex
    var yMutex sync.Mutex
    go func() {
        xMutex.Lock()
        yMutex.Lock()
        x, y = doWork(x, y)
        yMutex.Unlock()
        xMutex.Unlock()
    }()
    yMutex.Lock()
    xMutex.Lock()
    x, y = doOtherWork(x, y)
    xMutex.Unlock()
    yMutex.Unlock()
}
```

	T1	T2
1		
2	acq(y)	
3	acq(x)	
4	rel(x)	
5	rel(y)	
6		acq(x)
7		acq(y)
8		rel(y)
9		rel(x)

Dynamische Deadlock-Analyse

- Lock-Dependency-Methode sehr einfache Analyse
- Aus Acquire- und Releaseoperationen wird Graph aufgestellt
- Deadlock wenn Kreis im Graph existiert

Dynamische Deadlock-Analyse

```
1        T1              T2
2     acq(y)
3     acq(x)
4     rel(x)
5     rel(y)
6                   acq(x)
7                   acq(y)
8                   rel(y)
9                   rel(x)
```

```
1     y -> x
2     x -> y
```

Dynamische Deadlock-Analyse

	T1	T2
1		
2	acq(y)	
3	acq(x)	
4	rel(x)	
5	rel(y)	
6	acq(x)	
7	acq(y)	
8	rel(y)	
9	rel(x)	

```
1    y -> x
2    x -> y
```

Sync-preserving Deadlocks

- Die Events aus dem Original-Trace werden umgeordnet
- Dafür Sync-preserving Correct Reordering

Sync-preserving Deadlocks

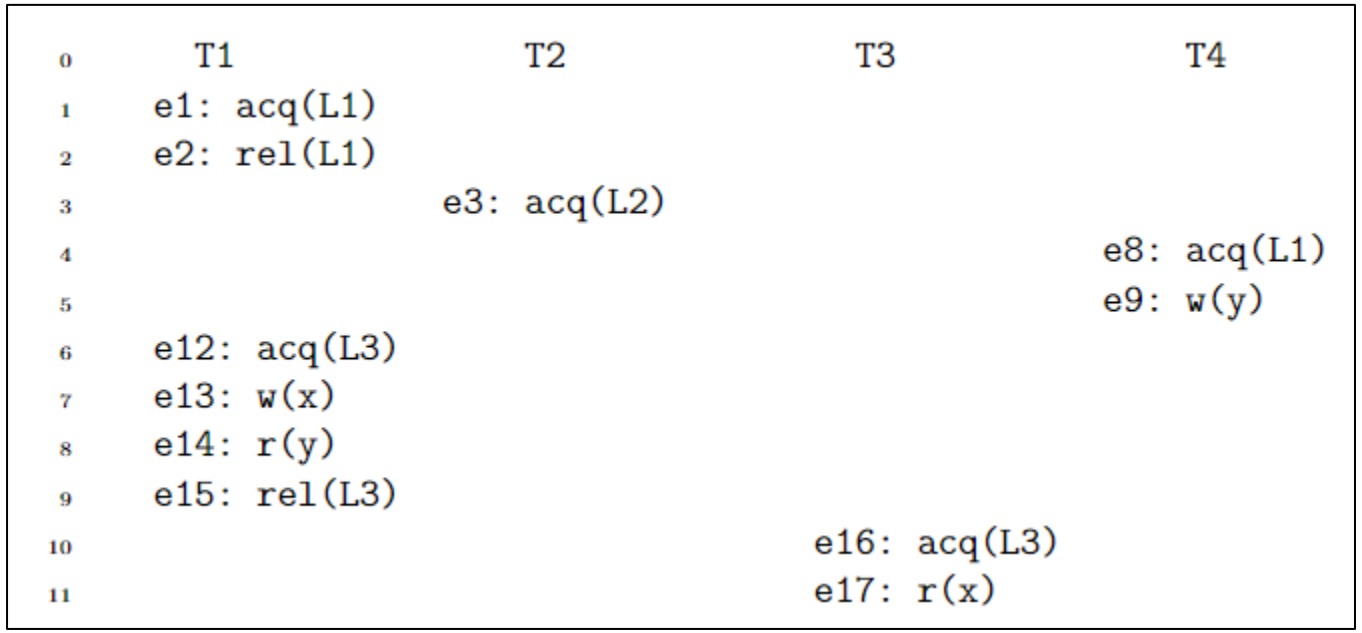
- Eine Umordnung ist ein Correct Reordering wenn folgende Regeln eingehalten werden
  - Subset: Es besteht ausschließlich aus Teilen des originalen Trace
  - Thread-Order: Die Reihenfolge der Operationen innerhalb eines Threads wurde nicht verändert
  - Last-Write: Wenn ein Read auf eine Variable existiert, muss das letzte Write auf diese Variable existieren
  - Lock-Semantik: Zwischen zwei Acquire-Events zweier verschiedener Threads muss es eine Release-Operation im ersten Thread geben

Sync-preserving Deadlocks

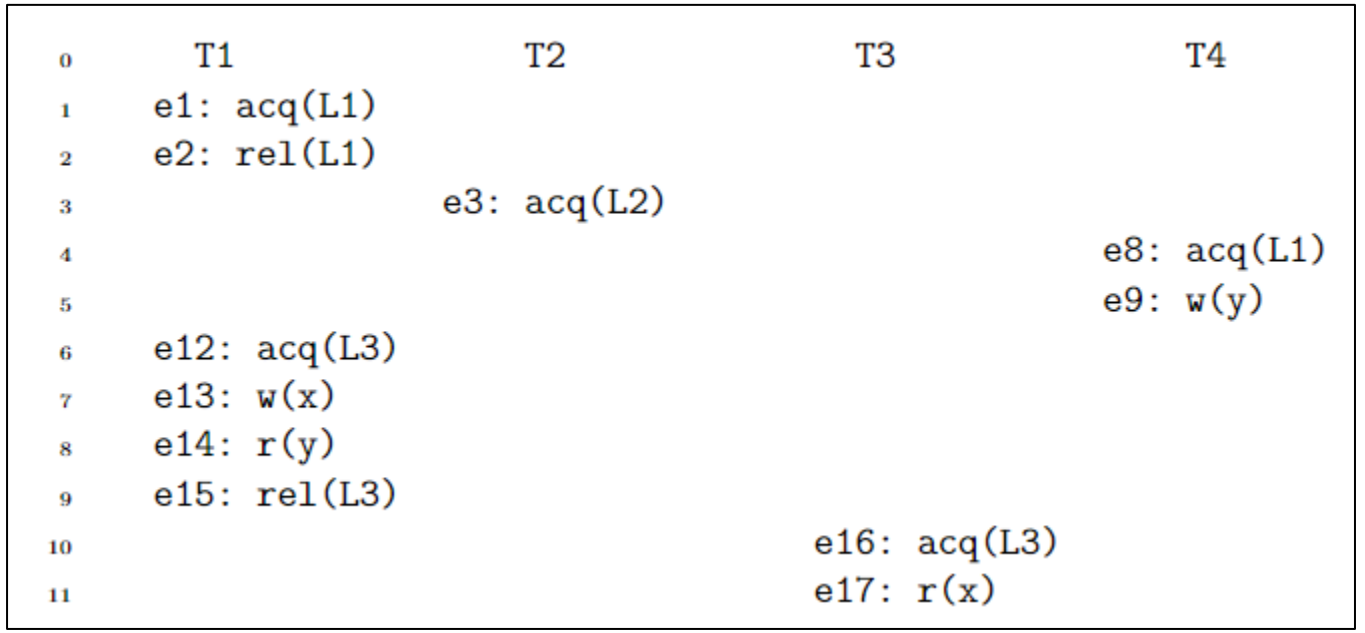
Deadlock-Pattern: <e4, e18>

	T1	T2	T3	T4
1	acq(L1)			
2	rel(L1)			
3		acq(L2)		
4		acq(L3)		
5		w(z)		
6		rel(L3)		
7		rel(L2)		
8				acq(L1)
9				w(y)
10				r(z)
11				rel(L1)
12	acq(L3)			
13	w(x)			
14	r(y)			
15	rel(L3)			
16			acq(L3)	
17			r(x)	
18			acq(L2)	
19			rel(L2)	
20			rel(L3)	

Sync-preserving Deadlocks



Sync-preserving Deadlocks



Subset: Es besteht ausschließlich aus Teilen des originalen Trace
Thread-Order: Die Reihenfolge der Operationen innerhalb eines Threads wurde nicht verändert
Last-Write: Wenn ein Read auf eine Variable existiert, muss das letzte Write auf diese Variable existieren
Lock-Semantik: Zwischen zwei Acquire-Events zweier verschiedener Threads muss es eine Release-Operation im ersten Thread geben

Sync-preserving Deadlocks

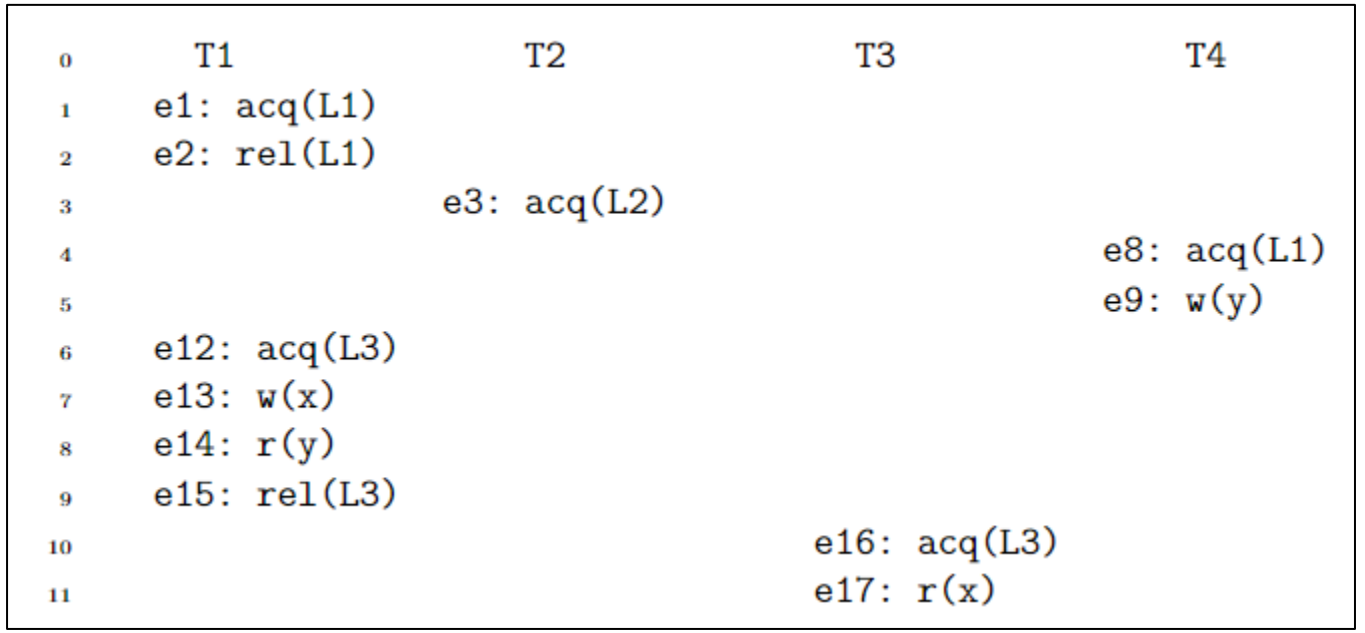
- Damit ein Correct Reordering Sync-preserving ist, müssen alle Acquire-Events auf denselben Lock in der gleichen Reihenfolge sein wie im Original-Trace

Sync-preserving Deadlocks

L1: T1 -> T4
L2: T2 -> T3
L3: T2 -> T1 -> T3

	T1	T2	T3	T4
0				
1	acq(L1)			
2	rel(L1)			
3		acq(L2)		
4		acq(L3)		
5		w(z)		
6		rel(L3)		
7		rel(L2)		
8				acq(L1)
9				w(y)
10				r(z)
11				rel(L1)
12	acq(L3)			
13	w(x)			
14	r(y)			
15	rel(L3)			
16			acq(L3)	
17			r(x)	
18			acq(L2)	
19			rel(L2)	
20			rel(L3)	

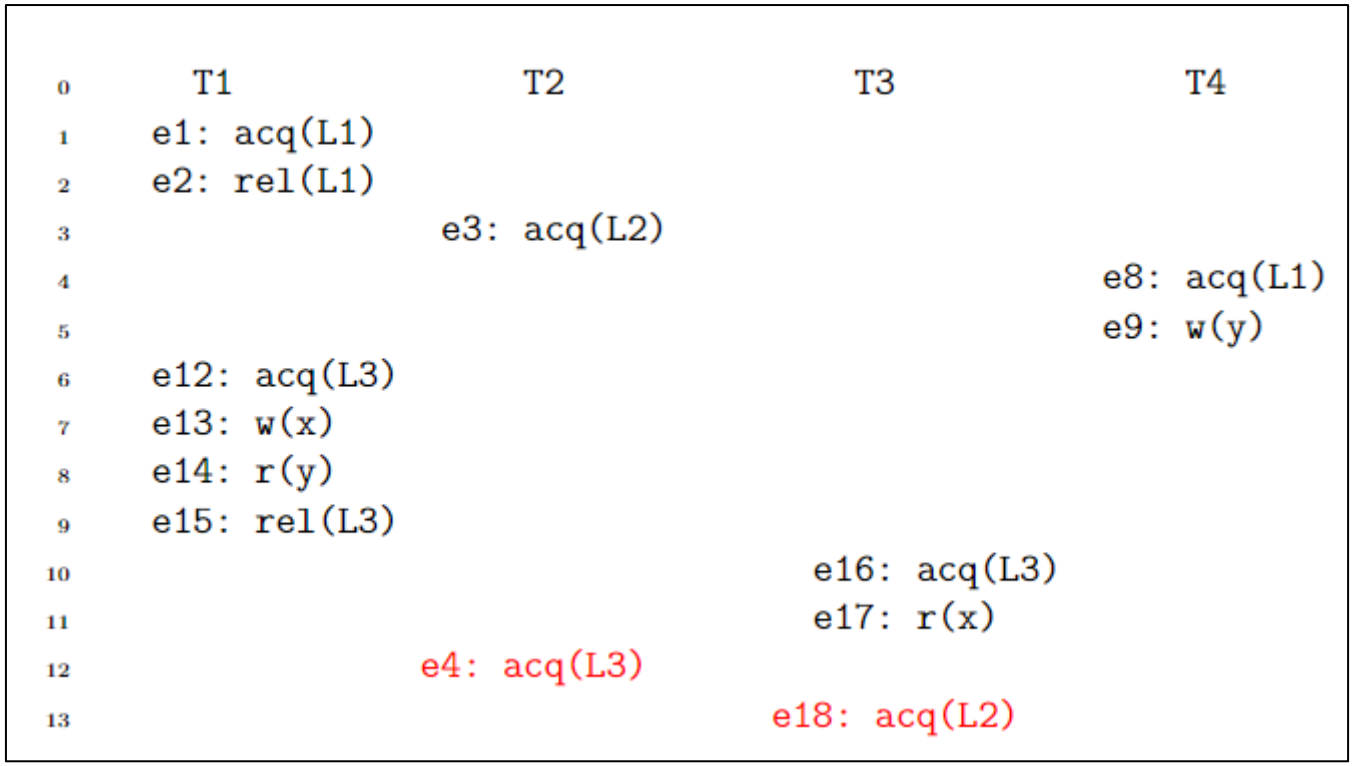
Sync-preserving Deadlocks



L1: T1 -> T4
L2: T2
L3: T1 -> T3

-> Das Correct Reordering ist auch Sync-preserving!

Sync-preserving Deadlocks



Sync-preserving Deadlocks

	T0	T1	T2
e1.	fork(T2)		
e2.			acq(z)
e3.			acq(y)
e4.			acq(x)
e5.			rel(x)
e6.			rel(y)
e7.			rel(z)
e8.	acq(z)		
e9.	fork(T1)		
e10.		acq(x)	
e11.		acq(y)	
e12.		rel(y)	
e13.		rel(x)	
e14.	join(T1)		
e15.	rel(z)		

Deadlock-Pattern: <e4, e11>
S = {e3, e10}
S‘ = SPClosure(S)
S‘ = {e3, e10}
S‘ = {e3, e2, e10, e9, e8} wegen Thread-Order
➔ Zwei Acq-Events für Lock z e2 und e8
➔ Nach Lock-Semantik muss e7 hinzugefügt werden
S‘ = {e3, e2, e10, e9, e8, e7}
S‘ = {e3, e2, e10, e9, e8, e7, e6, e5, e4} wegen Thread-Order
➔ e4 aus dem Deadlock Pattern ist in S‘
➔ Kein Sync-preserving Deadlock

Fazit

- Die Vorhersage von Sync-preserving Deadlocks kann ohne False-Positives geschehen
- False-Negatives können dennoch auftreten