Sound Dynamic Deadlock Prediction in Linear Time

Florian Rudaj
Fakultät für Informatik und
Wirtschaftsinformatik, Hochschule Karlsruhe

6. November 2023

Inhaltsverzeichnis

1 Einleitung

Sei es in Betriebssystemen, Web-Servern oder Echtzeitsystemen – in fast jeder modernen Applikation ist die Nebenläufigkeit (engl. „Concurrency“) von Tasks von großer Bedeutung. Beispielsweise könnte ohne Nebenläufigkeit ein Betriebssystem nicht mehr als ein Programm gleichzeitig laufen lassen. Ein IoT-Gerät müsste für die Netzwerkkommunikation die Aufnahme von Sensordaten stoppen. Allein diese Beispiele zeigen die Relevanz von Nebenläufigkeitin heutigen Systemen.

Nichtsdestotrotz gibt es neben den Vorteilen der Nebenläufigkeit, wie z.B. dem Performancegewinn durch das Verteilen von Tasks auf mehreren Prozessorkernen, auch Probleme, die mit ihr einhergehen – sog. „Concurrency Bugs.“ Grundlage dieser Concurrency Bugs sind die nebenläufigen Zugriffe verschiedener Tasks auf dieselben Ressourcen. Diese Zugriffe können dazu führen, dass ein Programm, welches die Tasks nebenläufig ausführt, abstürzt oder sogar in einer Endlosschleife verweilt, ohne tatsächlich Arbeit zu verrichten.

Es gibt mehrere Typen von Concurrency Bugs. Der wohl bekannteste von allen ist der sog. „Deadlock“. Dieser kommt zustande, wenn zwei Threads bereits eine Ressource reserviert haben, die der jeweils andere zum gleichen Zeitpunkt ebenso reservieren will. Wenn ein Deadlock vorkommt stürzt das Programm in Folge ab. Die Forschung beschäftigt sich schon sehr lange damit, Deadlocks zuverlässig und effizient zu erkennen bzw. vorherzusagen. Dabei wurden statische und dynamische Lösungsansätze entwickelt. Statische Lösungsansätze versuchen einen Deadlock anhand des Quellcodes zu erkennen, wohingegen dynamische Ansätze die Ausführung des Programms analysieren. Probleme dieser Lösungsansätze waren jedoch häufig, dass sie entweder zu viele false positives (Deadlocks, die aber keine sind) angezeigt oder eine zu schlechte Laufzeit haben[1].

Die vorliegende Arbeit beschäftigt sich jedoch mit einem kürzlich erbrachten und bemerkenswerten Fortschritt in der Vorhersage von Deadlocks. Nachdem in den nachfolgenden Kapiteln zuerst auf Concurrency Bugs und Deadlocks im Besonderen eingegangen wird, werden Methoden vorgestellt, die es ermöglichen, in linearer Laufzeit sowie mit sehr hoher Präzision Deadlocks vorherzusagen[1].

2 Nebenläufige Programme

Man bezeichnet ein System als nebenläufig, wenn es dazu in der Lage ist mehrere Tasks simultan abzuarbeiten. Bei dieser Definition wird oft fälschlicher-

weise angenommen, dass die Nebenläufigkeit der Parallelität gleichzusetzen ist. Allerdings heißt Nebenläufigkeit jedoch nicht gleichzeitig, dass die Tasks auch parallel bearbeitet werden. Ein System kann nämlich auch dann schon als nebenläufig bezeichnet werden, wenn es lediglich mehrere Tasks in einer Warteschlange hält und zwischen diesen hin und her wechselt, bevor diese vollständig bearbeitet wurden[2]. Genau so macht es ein Betriebssystem, welches auf einer Maschine mit nur einem Prozessorkern läuft. Es definiert Zeitschlitze, in denen jeweils ein Prozess Zugriff auf den Prozessorkern hat, bevor der nächste Prozess an der Reihe ist. Ob der jeweilige Prozess fertig bearbeitet wurde, ist keine Bedingung dafür, dass der nächste Prozess bearbeitet werden kann. Unfertig bearbeitete Prozesse werden wieder in die Warteschlange eingereiht.

Allerdings sind Systeme, die Tasks parallel bearbeiten, auch gleichzeitig nebenläufig. Angenommen die Maschine, auf der unser Betriebssystem läuft, hat nun mehrere Prozessorkerne zur Verfügung. Damit hat unser Betriebssystem die Möglichkeit so viele Tasks gleichzeitig bearbeiten zu lassen, wie es Prozessorkerne gibt. Mit der gewachsenen Anzahl der Prozessorkerne und die dadurch ermöglichte Parallelität verfügt das System auch über die Fähigkeit die gleiche Anzahl an Tasks in viel kürzerer Zeit zu bearbeiten. Die parallele Bearbeitung bedeutet also einen Performancegewinn.

Dies ist jedoch nicht der einzige Vorteil den Nebenläufigkeit mit sich bringt. Man stelle sich zum Beispiel einen Browser vor über den der User durch Eingabe einer URL zu einer Website gelangt. Nach Eingabe der URL und betätigen der Enter-Taste bemerkt der User, dass das Laden der Website sehr lange dauert und entscheidet sich stattdessen eine andere Website zu besuchen. Die Eingabe der anderen URL ist dem User nur möglich, weil das Laden der Website und das Erkennen von User-Eingaben im UI parallel, oder zumindest nebenläufig, bearbeitet wird. Ansonsten würde der User warten müssen, bis die Website vollständig geladen ist oder ein Fehlerfall auftritt. Neben dem Performancegewinn bietet die Nebenläufigkeit also auch noch den Vorteil der Reaktionsfähigkeit.

Nebenläufigkeit kann auch dabei helfen eine Anwendung skalierbarer zu machen. Ein Beispiel dafür sind Webserver, die die Anfragen von mehreren hundert Clients gleichzeitig bearbeiten müssen. Durch die Nebenläufigkeit können auf unterschiedlichen Prozessorkernen mehrere Anfragen parallel verarbeitet werden. Falls eine dieser Anfragen beispielsweise einen länger dauernden I/O-Zugriff ausführt kann während der dadurch auftretenden Wartezeit die Anfrage eines anderen Clients bearbeitet werden. Nebenläufigkeit stellt also auch sicher, dass die zur Verfügung stehenden Ressourcen möglichst effizient ausgeschöpft werden.

Doch der Einsatz von Nebenläufigkeit bringt auch Gefahren mit sich. Ein

Beispiel dafür ist der folgende in der Programmiersprache Go geschriebene Code. Die Funktion „raceCondition“ verfügt über eine Variable „data“, die auf den Wert 0 geprüft wird. Wenn der Wert 0 ist, soll der Wert zurückgegeben werden. Wenn der Wert jedoch nicht 0 ist, wird -1 zurückgegeben. In den Zeilen 3 – 5 startet die Funktion einen neuen Thread, der die Variable „data“ um eins erhöht. Bei Ausführung der Funktion „raceCondition“ ergeben sich drei mögliche Ausgabewerte. Die beiden offensichtlichsten Ausgabewerte sind 0 und -1. Wenn der nebenläufige Thread aus den Zeilen 3-5 es noch nicht geschafft „data“ um eins zu erhöhen wird 0 ausgegeben. Der Wert -1 wird zurückgegeben, wenn der nebenläufige Thread es vor der Prüfung auf 0 geschafft hat „data“ hochzuzählen. Der dritte Fall ist weniger offensichtlich und gleichzeitig sehr problematisch. Er tritt ein, wenn die Prüfung von „data“ auf den Wert 0 erfolgreich ist, jedoch zwischen der Prüfung und der Rückgabe von „data“ der nebenläufige Thread das Inkrement um eins durchgeführt hat. In diesem Fall gibt „raceCondition“ 1 zurück.

```
1  func raceCondition() int {
2          data := 0
3          go func() {
4                  data++
5          }()
6          if data == 0 {
7                  fmt.Println("")
8                  return data
9          }
10         return -1
11 }
12
13 func main() {
14         var outputs []int
15         for i := 0; i < 100000; i++ {
16                 outputs = append(outputs, raceCondition())
17         }
18         print_shares([]int{-1, 0, 1}, outputs)
19 }
```

Listing 1: Beispiel einer Race Condition (abgeleitet von [2])

Um zu beweisen, dass bei jeder Ausführung völlig unklar ist welches Ergebnis zurückgegeben wird, wird in der Main-Funktion die Race Condition 100.000 mal ausgeführt. Die Funktion „print_shares“ in Zeile 18 nimmt die

Liste der Ausgaben von „raceCondition“ entgegen und gibt die Häufigkeit sowie Anteile der Ergebnisse -1, 0 und 1 in der Konsole aus. Für eine Ausführung dieses Programms ergibt sich beispielhaft die folgende Ausgabe:

```
Counts of -1: 118
Share of -1: 0.118000 Percent

Counts of 0: 99876
Share of 0: 99.876000 Percent

Counts of 1: 6
Share of 1: 0.006000 Percent
```

Listing 2: Konsolenausgabe bei Race Condition

Hier ist zu erkennen, dass vor allem der Fall, bei dem „raceCondition“ 0 zurückgibt, auftritt. Aber auch die anderen beiden Fälle treten auf. Besonders ärgerlich ist, dass der Fall, bei dem 1 zurückgegeben wird, auftritt und gleichzeitig sehr selten ist. Damit das Programm in die Zeilen 7 und 8 gelangen darf, muss die Variable „data“ den Wert 0 enthalten. Allerdings hat der nebenläufige Thread durch die simulierte Arbeit in Zeile 7 manchmal genügend Zeit den Inhalt von „data“ vor der Rückgabe zu ändern.

Ein solcher Concurrency Bug nennt sich Race Condition. Genauer gesagt handelt es sich hier um einen sog. „Data Race“, da es darauf ankommt, welcher Thread seine Arbeit mit den Daten in der Variable „data“ zuerst verrichtet. In Unit-Tests wird ein solcher Concurrency Bug nicht erkannt, da er, wie in der Konsolenausgabe zu erkennen ist, dafür viel zu selten auftritt. In Produktionsumgebungen jedoch, wo eine solche Funktion tausende Male am Tag ausgeführt wird, kann ein solcher Bug auftreten. Dieser ist dann aber schwer oder gar nicht zu replizieren, da er so selten vorkommt. Bei Concurrency Bugs kann es auch vorkommen, dass sie erst ab einem gewissen Skalierungsniveau auftreten. Beispielsweise wird ein Dienst, der einen solchen Bug enthält, 1-mal pro Tag aufgerufen. Wenn die Wahrscheinlichkeit, zu der der Concurrency Bug auftritt, die gleiche ist wie in unserem obigen Beispiel kann dieser Bug jahrelang unentdeckt bleiben. Wenn die Anfragen auf den Dienst jedoch zunehmen, tritt dieser Bug immer häufiger auf und es kommt dadurch wohlmöglich zu Inkonsistenzen in den Daten und Ausgaben des Dienstes.

Um Race Conditions zu verhindern, können Zugriffe auf Variablen, die sich während der Ausführung eines bestimmten Abschnitts im Code nicht verändern dürfen, synchronisiert werden. Dies geschieht durch die Verwen-

dung eines sog. „Mutex“. Nebenläufige Threads können sich durch das Erwerben eines Locks auf einen Mutex den Zugriff auf eine Variable, die zu dem Mutex gehört, reservieren[2]. Angewandt auf das obige Beispiel sieht der Code der Funktion „raceCondition“ folgendermaßen aus:

```
1  func raceCondition() int {
2          data := 0
3          var dataAccess sync.Mutex
4          go func() {
5                  dataAccess.Lock()
6                  data++
7                  dataAccess.Unlock()
8          }()
9          dataAccess.Lock()
10         if data == 0 {
11                 print("")
12                 return data
13         }
14         dataAccess.Unlock()
15         return -1
16 }
```

Listing 3: Durch Mutex verhinderte Race Condition (abgeleitet von [2])

Hier ist in den Zeilen 5 - 7 zu sehen, dass der nebenläufige Thread den Mutex zuerst sperrt, bevor er „data“ inkrementiert. Anschließend wird der Mutex wieder freigegeben, da die Änderung an der geteilten Variable vollständig durchgeführt wurde. Der Main-Thread hingegen sperrt in Zeile 9 den Mutex, um sicher zu gehen, dass sich der Inhalt von „data“ in den Zeilen 10 – 12 nicht verändert.

Wenn nun die gleiche Main-Funktion wie bereits in dem vorigen Code-Beispiel ausführt, werden folgende Ergebnisse in die Konsolenausgabe geschrieben:

```
Counts of -1: 2063
Share of -1: 2.063000 Percent

Counts of 0: 97937
Share of 0: 97.937000 Percent

Counts of 1: 0
Share of 1: 0.000000 Percent
```

Listing 4: Konsolenausgabe ohne Race Condition

Wie anhand der Konsolenausgabe zu erkennen ist, gibt es keinen einzigen Fall mehr, bei dem sich der Wert von „data“ nach der Prüfung auf 0 doch noch zu 1 ändert. Das Reservieren und Freigeben eines Mutex sichert also die Konsistenz einer Variable innerhalb einer kritischen Sektion. Der Begriff „kritische Sektion“ bezeichnet dabei einen Abschnitt im Code, der Lese- oder Schreiboperationen auf eine zwischen Threads geteilte Variable ausführt.

Allerdings bringt diese Methode der Konsistenzsicherung weitere Probleme mit sich. Eines dieser Probleme ist der sog. Deadlock. Er tritt dann auf, wenn alle Threads eines Programms zur Bearbeitung ihres Tasks einen Mutex reservieren müssen, der aber bereits von einem anderen Thread gehalten wird. Das bedeutet, dass im Falle eines Deadlocks alle Threads aufeinander warten und keine Arbeit verrichtet wird. Ein einfaches Beispiel eines Deadlocks zeigt der folgende Code:

```
func simple_deadlock() {
        a := 0
        b := 0
        var aMutex sync.Mutex
        var bMutex sync.Mutex
        go func() {
                aMutex.Lock()
                bMutex.Lock()
                //simulate work
                if a == 1 && b == 1 {
                        a++
                        b++
                }
                bMutex.Unlock()
                aMutex.Unlock()
        }()
        bMutex.Lock()
        aMutex.Lock()
        //simulate work
        if a == 0 && b == 0 {
                b++
                a++
        }
        aMutex.Unlock()
        bMutex.Unlock()
}
```

Listing 5: Beispiel für einen Deadlock

Im obigen Codebeispiel gibt es zwei Threads. Beide Threads benötigen die Variablen a und b innerhalb einer If-Bedingung. Um zu sichern, dass die Werte, auf die die Variablen geprüft wurden, innerhalb der If-Bedingung gleichbleiben, gibt es für jede Variable einen Mutex. Nun ist es so, dass die beiden Threads eine unterschiedliche Reihenfolge haben den Mutex zu reservieren. Dies kann dazu führen, dass der nebenläufige Thread erst den Mutex für a und der Main-Thread den Mutex für b reserviert. Im nächsten Schritt versuchen beide Threads den Mutex, der vom jeweils anderen Thread bereits reserviert wurde, selbst zu reservieren. Da dies jedoch nicht möglich ist kommt es zu einem Deadlock, da alle Threads warten. Infolgedessen stürzt das Programm ab.

3 Die Vorhersage von Deadlocks

Um den Absturz von Programmen durch Deadlocks zu verhindern, wurde in der Vergangenheit viel Forschung betrieben, um Deadlocks vorherzusagen. Dies geschieht durch die Analyse des Programms. Bei den Analysemethoden für Deadlocks unterscheidet man zwischen der statischen und der dynamischen Deadlock-Analyse[1].

Die statische Deadlock-Analyse betrachtet ausschließlich den Quellcode für die Vorhersage von Deadlocks. Es gibt statische Ansätze, die dazu in der Lage sind, die Abwesenheit von Deadlocks zu beweisen. Allerdings sind statische Analysemethoden nicht gut skalierbar und zeigen in Programmen oft Deadlocks an, wo jedoch keine sind[1].

Eine Analysemethode, die solche „False-Positives“ anzeigt, wird auch „unsound“ genannt. Mit dem Begriff „sound“ wird in der Deadlock-Analyse eine Methode bezeichnet, wenn diese keine false positives vorhersagt.

Die dynamische Deadlock-Analyse betrachtet eine beispielhafte Ausführung des Programms in Form eines Trace als Basis für die Vorhersage. Die Ereignisse innerhalb des Trace werden auf bestimmte Weise umgeordnet, sodass Deadlocks erkannt werden können. Im Gegensatz zur statischen Deadlock-Analyse wird hierbei nicht das Ziel verfolgt, einen Beweis für die Abwesenheit von Deadlocks zu erbringen. Stattdessen wird sich darauf konzentriert durch Umordnungen von Traces theoretisch mögliche Deadlocks zu entdecken. Dynamische Methoden erkennen zwar nicht alle, aber dennoch viele Deadlocks. Außerdem sind sie dadurch, dass sie keinen Beweis erbringen müssen, skalierbarer und liefern nur wenige bis keine False-Positives. Ein solcher dynamischer Ansatz wird in der vorliegenden Arbeit im Kapitel „Sync-preserving Deadlocks“ vertieft beschrieben. Er erkennt die meisten Deadlocks, ist sehr gut zu skalieren und ist darüber hinaus sound[1].

Die gerade beschriebenen Analysemethoden müssen in der Lage sein, verschiedene Typen von Deadlocks vorherzusagen. Zum einen gibt es den sog. „Resource-Deadlock“, bei dem alle Threads des Programms auf die Freigabe einer Ressource warten, die von einem anderen Thread des Programms bereits reserviert wurde. Das Deadlock-Beispiel aus dem vorherigen Kapitel ist ein solcher Resource -Deadlock.

Weiterhin gibt es den sog. „Communication-Deadlock“, der dann auftritt, wenn alle Threads darauf warten mit einem anderen Thread kommunizieren zu können. Der folgende Code enthält einen solchen Communication-Deadlock:

```
1  func communication_deadlock() {
2          ch := make(chan int)
3
4          go func() {
5                  data := <-ch
6                  fmt.Println("Received:", data)
7          }()
8
9          data := <-ch
10         fmt.Println("Received:", data)
11 }
```

Listing 6: Communication Deadlock

Ein Channel in Go ist ein Objekt, welches ermöglicht, Daten zwischen Threads für nebenläufige Berechnungen auszutauschen. In Zeile 2 des obigen Codebeispiels wird ein solcher Channel erstellt. Der nebenläufige Thread in den Zeilen 4 - 7 möchte Daten aus dem Channel empfangen. Da es keinen Sender gibt, der Daten in dem Channel platziert, blockiert der nebenläufige Thread. Da der Main-Thread in Zeile 9 ebenfalls Daten aus demselben Channel empfangen will, aber kein Sender vorhanden ist, blockiert auch dieser. Dies führt dazu, dass sich beide Threads des Programms in einem Wartezustand befinden und ein Communication-Deadlock auftritt.

4 Dynamische Deadlock-Analyse

Bei der dynamischen Deadlock-Analyse wird versucht, theoretisch mögliche Deadlocks anhand eines Trace des Programms zu erkennen. Genauer gesagt wird versucht, sog. „Deadlock-Patterns“ in einem Trace zu erkennen. Deadlock-Patterns sind Muster, die darauf hindeuten, dass ein Trace einen potenziellen Deadlock enthält. Ein Deadlock-Pattern zu erkennen ist eine notwendige, aber keine hinreichende Bedingung für die Detektion eines Deadlocks[1]. Das bedeutet, dass zwar ein Deadlock-Pattern für die Vorhersage eines Deadlocks vorhanden sein muss, jedoch nicht hinter jedem Deadlock-Pattern ein tatsächlicher Deadlock stehen muss.

Ein einfacher Algorithmus, um dynamisch Deadlocks zu erkennen ist die „Lock-Dependency-Methode“. Sie basiert auf der Idee von sog. „Lock-Graphen“. Ein Lock-Graph baut sich aus den Lock-Operationen nebenläufiger Threads auf gemeinsame Ressourcen bzw. Mutexes auf. Ein Lock auf eine Ressource stellt hierbei einen Knoten im Graph dar. Eine Kante von

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

Listing 8: Lock-Graph zum obigen Trace[3]

einem Lock-Knoten zu einem anderen Lock-Knoten entsteht dann, wenn ein nebenläufiger Thread einen der beiden Locks bereits hält und den anderen reservieren möchte. Nachdem der Lock-Graph aus dem Trace aufgebaut wurde, wird nach Zyklen im Lock-Graph gesucht. Der Zyklus im Lock-Graphen ist das Deadlock-Pattern dieses Algorithmus. Wenn mindestens ein Zyklus vorhanden ist, dann geht die Lock-Dependency-Methode davon aus, dass ein Deadlock vorhanden ist[3]. Man betrachte den folgenden Trace:

```
  T1          T2
acq(y)
acq(x)
rel(x)
rel(y)
            acq(x)
            acq(y)
            rel(y)
            rel(x)
```

Listing 7: Trace, der einen Deadlock enthält[3]

Die Operationen acq (acquire) und rel (release) stehen hier für die Reservierung und die Freigabe eines Mutexes. Es ist zu sehen, dass Thread T1 erst y und dann x reserviert. Unser Graph bekommt dadurch die Knoten x und y, sowie eine Kante von y nach x. In Thread T2 wird erst x und dann y reserviert. Wir haben also wieder die Knoten x und y, die aber bereits in unserem Lock-Graph existieren. Hinzu kommt aber eine Kante von x nach y, da T2 x bereits hält, bevor y reserviert wird. Daraus ergibt sich der folgende Lock-Graph:

Der Lock-Graph wird im nächsten Schritt auf Zyklen untersucht. Wie im obigen Beispiel leicht zu erkennen ist, gibt es einen Zyklus zwischen den Knoten x und y. Die Lock-Dependency-Methode sagt hier also einen Deadlock voraus. Dies ist auch richtig, da der obige Trace auch so verlaufen kann, dass im ersten Schritt T1 y reserviert und im zweiten Schritt T2 x reserviert. Der nächste Schritt für beide Threads wäre dann die Reservierung des Mutex, der vom jeweils anderen Thread bereits gehalten wird. Da dies für beide

```
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)
```

Listing 9: Trace, bei dem nur ein Thread Arbeit verrichtet[3]

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

Listing 10: Lock-Graph zu obigem Trace[3]

Threads jedoch nicht möglich ist, resultiert diese Umordnung des Trace in einem Deadlock[3].

Allerdings sagt die Lock-Dependency-Methode auch häufig False-Positives vorher. So wie in dem folgenden Beispiel:

Hier werden alle Reservierungen und Freigaben der Ressourcen innerhalb von T1 ausgeführt. T2 benötigt keine Ressourcen, die zu reservieren wären. Aus dem obigen Trace resultiert der gleiche Lock-Graph wie schon im ersten Trace[3]:

Da in diesem Lock-Graph ein Zyklus steckt sagt die Lock-Dependency-Methode wieder einen Deadlock vorher. Dieser kann in der Realität jedoch niemals auftreten, da nur T1 Ressourcen reserviert und freigibt.

Das Vorhersagen von False-Positives ist problematisch, da Softwareentwickler anschließend viel Zeit mit der Suche und Verhinderung des Deadlocks verbringen würden, ohne dass dieser überhaupt existiert. Aus diesem Grund wurde eine weitere dynamische Analysemethode entwickelt, die keine False-Positives vorhersagt und dennoch einen Großteil der potenziellen Deadlocks erkennt. Auf diese Methode wird im nachfolgenden Kapitel eingegangen.

5 Sync-preserving Deadlocks

6 Fazit

Literatur

1. Tunç, H.C., Mathur, U., Pavlogiannis, A., Viswanathan, M.: Sound dynamic deadlock prediction in linear time. Proc. ACM Program. Lang. **7**(PLDI) (jun 2023)

2. Cox-Buday, K.: Concurrency in Go: Tools and Techniques for Developers. 1st edn. O'Reilly Media, Inc. (2017)

3. Sulzmann, M.: Dynamic deadlock prediction. `https://sulzmann.github.io/AutonomeSysteme/lec-deadlock.html` Accessed: 2023-11-01.