Sound Dynamic Deadlock Prediction in Linear Time

Florian Rudaj

Hochschule Karlsruhe
Matrikel-Nummer: 65106
rufl1020@h-ka.de

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.

hier noch Quelle zur Einleitung im Paper hinterlegen, da das mit den statischen und dynamischen Lösungsansätzen daraus kommt

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.

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 fälschlicherweise 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. 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 race_condition() 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, race_condition())
17          }
18          print_shares([]int{-1, 0, 1}, outputs)
19  }
```

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
```

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“ 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, kommt es umso häufiger vor, dass dieser Bug auftritt und wohlmöglich Inkonsistenzen in den Daten und Ausgaben des Dienstes verursacht.

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 Verwendung 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. Angewandt auf das obige Beispiel sieht der Code der Funktion „raceCondition“ folgendermaßen aus:

```
1  func race_condition() 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 }
```

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
```

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, die im folgenden Kapitel behandelt werden.

3 Deadlocks und deren Vorhersage

4 Dynamische Deadlock-Analyse

5 Bestehende Ansätze

5.1 Deadlock-Muster

6 Sync-preserving Deadlocks

7 Fazit

8 Beispiele

Dieses Kapitel enthält Beispiele wie Bilder, Tabellen und Fußnoten verwendet werden können.

8.1 Fußnote mit Link

The website *Google*[1] is a search engine.

8.2 Referenz auf Bild

Bilder können gut benutzt werden um das Forschungsmodell zu zeigen (vgl. Abb. 1).

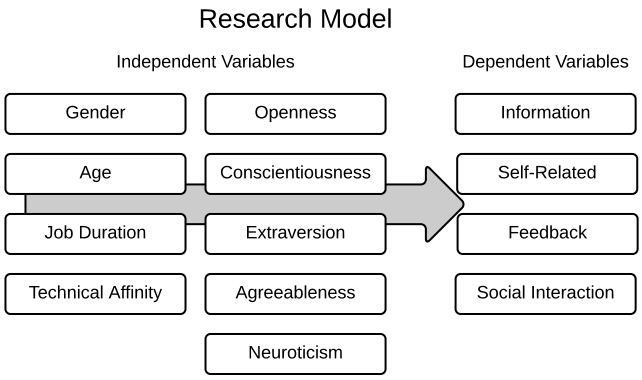

Abb. 1. Our case study investigating explanations for differences in usage motivation.

8.3 Beispiel Tabelle

8.4 Referenzen

The usage of social networking sites (SNS) for business purposes seems to be a promising approach for enhanced connectivity and communication among employees independent from space, time and position [1]. Since social media services like Facebook, Twitter and other SNS are part of our daily private lives [2], their implementation as a business support tool spread with amazing rapidity [3].

Die Referenzen finden sich in der Datei: references.bib.

[1] `https://www.google.com`

I use the software because,...	**Scale**	Loading
I can access information more easily.	Information	.825
I can access information that is relevant for me.	Information	.817
I will get informed about activities in my department.	Information	.775
I can present my ideas.	Information	.697

Tabelle 1. Dependent variables: Item texts and scales. Loading refers to the factor-loading of the principal component analysis after varimax rotation with Kaiser-Normalization.

Literatur

1. DiMicco, J., Millen, D.R., Geyer, W., Dugan, C., Brownholtz, B., Muller, M.: Motivations for social networking at work. In: Proceedings of the 2008 ACM conference on Computer supported cooperative work, ACM (2008) 711–720
2. Stocker, A., Müller, J.: Exploring factual and perceived use and benefits of a web 2.0-based knowledge management application: The siemens case references+. In: Proceedings of the 13th International Conference on Knowledge Management and Knowledge Technologies, ACM (2013) 18
3. Koch, M., Richter, A.: Enterprise 2.0: Planung, Einführung und erfolgreicher Einsatz von Social Software in Unternehmen. Oldenbourg Verlag (2009)