Bachelorarbeit

Dynamic Analysis of Message-Passing Go Programs

Erik Daniel Kassubek

Gutachter: Prof. Dr. Thiemann

Betreuer: Prof. Dr. Thiemann

Prof. Dr. Sulzmann

Albert-Ludwigs-Universität Freiburg

Technische Fakultät

Institut für Informatik

Lehrstuhl für Programmiersprachen

14. Februar 2023

Bearbeitungszeit

14. 11. 2022 – 14. 02. 2023

Gutachter

Prof. Dr. Thiemann

Betreuer

Prof. Dr. Thiemann

Prof. Dr. Sulzmann

ERKLÄRUNG

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Ort, Datum	Unterschrift

Zusammenfassung

(TODO: Zusammenfassung (Abstract) schreiben)

Inhaltsverzeichnis

1 Einführung

(TODO: Einführung schreiben)

2 Hintergrund

3 Tracer

Um ein Program analysieren zu können, wird der Ablauf eines Programmdurchlaufs aufgezeichnet. Dazu wurde ein Tracer implementiert, durch welchem die Channel- und Lock-Operationen, sowie andere Operationen wie Select und das erzeugen einer neuen Routine, ersetzt, bzw. erweitert werden. Diese führen die eigentlichen Operationen aus und zeichnen gleichzeitig den Trace auf. Die Ersetzung durch die Drop-In Replacements kann dabei automatisch durch einen Instrumenter erfolgen, welche mit Hilfe des Abstract Syntax Tress die Ersetzungen vornimmt.
Anders als in vielen anderen Programmen, welche den Trace von Go-Programmen analysieren, wie z.B. [1] oder [2] wird dabei der Tracer selbst implementiert und basiert nicht auf dem Go-Runtime-Tracer [3]. Dies ermöglicht es, den Tracer genau auf die benötigten Informationen zuzuschneiden und so einen geringeren negativen Einfluss auf die Laufzeit des Programms zu erreichen.

(EXTEND: Tracer Einführung)

3.1 Trace

(DRAFT: Trace) Der Aufbau des Trace basiert auf [4]. Er wird aber um Informationen über Locks erweitert. Der Trace wird für jede Routine separat aufgezeichnet. Die Syntax

des Traces in EBNF gibt sich folgendermaßen:

$$
\begin{array}{lcll}
T & = & '' \, [\, '', \, \{ \, U \, \}, \, '' \,] \, '' ; & \text{Trace} \\
U & = & '' \, [\, '', \, \{ \, t \, \}, \, '' \,] \, '' ; & \text{lokaler Trace} \\
t & = & signal(i) \mid wait(i) \mid pre(as) \mid post(i,j,a) & \text{Event} \\
 & & \mid post(default) \mid close(x) \mid lock(y,b,c) & \\
 & & \mid unlock(y); & \\
a & = & x, \, (''\,!\,'' \mid ''\,?\,''); & \\
as & = & a \mid (\{ \, a \, \}, [\, '' \, default \, '' \,]); & \\
b & = & ''-'' \mid ''t'' \mid ''r'' \mid ''tr'' & \\
c & = & ''0'' \mid ''1'' &
\end{array}
$$

wobei i die Id einer Routine, j einen Zeitstempel, x die Id eines Channels und y die Id eines Locks darstellt. Die Events haben dabei folgende Bedeutung:

- `signal(i)`: In der momentanen Routine wurde eine Fork-Operation ausgeführt, d.h. eine neue Routine mit Id i wurde erzeugt.

- `wait(i)`: Die momentane Routine mit Id i wurde soeben erzeugt. Dies ist in allen Routinen außer der Main-Routine das erste Event in ihrem lokalen Trace.

- `pre(as)`: Die Routine ist an einer Send- oder Receive-Operation eines Channels oder an einem Select-Statement angekommen, dieses wurde aber noch nicht ausgeführt. Das Argument as gibt dabei die Richtung und den Channel an. Ist $as = x!$, dann befindet sich der Trace vor einer Send-Operation, bei $as = x?$ vor einer Receive-Operation. Bei einem Select-Statement ist as eine Liste aller Channels für die es einen Case in dem Statement gibt. Besitzt das Statement einen Default-Case, wird dieser ebenfalls in diese List aufgenommen, also $as = [\, x_1?, \ldots, x_n?, default]$.

- `post(i, j, a)`: Dieses Event wird in dem Trace gespeichert, nachdem eine Send- oder Receive-Operation erfolgreich abgeschlossen wurde. i gibt dabei die Id und j den momentanen Zeitstempel der sendenden Routine an. a gibt wieder an, ob es

sich um eine Send- ($a = x!$) oder Receive-Operation ($a = x?$) auf dem Channel x handelt. Durch die Speicherung der Id und des Zeitstempels der sendenden Routine bei einer Receive-Operation lassen sich die Send- und Receive-Operationen eindeutig zueinander Zuordnen.

- `post(default)`: Wird in einem Select-Statement der Default-Case ausgeführt, wird dies in dem Trace der entsprechenden Routine durch $post(default)$ gespeichert.

- `close(x)`: Mit diesem Eintrag wird das schließen eines Channels x in dem Trace aufgezeichnet.

- `lock(y, b, c)`: Der Beanspruchungsversuch eines Locks mit Id j wurde beendet. b gibt dabei die Art der Beanspruchung an. Bei $b = r$ war es eine R-Lock Operation, bei $b = t$ eine Try-Lock Operation und bei $b = tr$ ein Try-R-Lock Operation. Bei einer normalen Lock-Operation ist $b = -$. Bei einer Try-Lock Operation kann es passieren, dass die Operation beendet wird, ohne das das Lock gehalten wird. In diesem Fall wird c auf 0, und sonst auf 1 gesetzt.

Man betrachte als Beispiel das folgende Programm in Go:

```
func main() {
    x := make(chan int)
    y := make(chan int)
    a := make(chan int)

    var v sync.RWMutex

    go func() {
        v.Lock()
        x <- 1
        v.Unlock()
    }()
    go func() { y <- 1; <-x }()
    go func() { <-y }()
```

```
        select {
        case <-a:
        default:
        }
    }
```

Erweitert man diesen mit dem Tracer, erhält man folgendes Programm:

```
    func main() {
        tracer.Init()
        defer tracer.PrintTrace()

        x := tracer.NewChan[int](0)
        y := tracer.NewChan[int](0)
        a := tracer.NewChan[int](0)

        v := tracer.NewRWLock()

        func() {
            GoChanRoutineIndex := tracer.SpawnPre()
            go func() {
                tracer.SpawnPost(GoChanRoutineIndex)
                v.Lock(); x.Send(1); v.Unlock()
            }()
        }()

        func() {
            GoChanRoutineIndex := tracer.SpawnPre()
            go func() {
                tracer.SpawnPost(GoChanRoutineIndex)
                {
                    x.Receive(); x.Send(1)
                }
            }()
        }()

        func() {
```

```
            GoChanRoutineIndex := tracer.SpawnPre()
            go func() {
                tracer.SpawnPost(GoChanRoutineIndex)
                y.Send(1); x.Receive()
            }()
        }()

        tracer.PreSelect(true, a.GetIdPre(true))
        select {
        case sel_XVlBzgba := <-a.GetChan():
            a.Post(true, sel_XVlBzgba)
        default:
            tracer.PostDefault()
        }
        time.Sleep(4 * time.Second)
    }
```

Dieser ergibt den folgenden Trace:

$$
\begin{aligned}
&[[signal(2),\ signal(3),\ signal(4),\ pre(3?, default),\ post(default)]\\
&[wait(2),\ lock(1,-,1),\ pre(1!),\ post(2,2,1!),\ unlock(1)]\\
&[wait(3),\ pre(2!),\ post(3,1,2!),\ pre(1?),\ post(2,2,1?)]\\
&[wait(4),\ pre(2?),\ post(3,1,2?)]]
\end{aligned}
$$

Aus diesem lässt sich der Ablauf des Programms eindeutig nachverfolgen. **(EXTEND: Tracer)**

3.2 Instrumenter

(DRAFT: Instrumenter) Um den Trace zu erzeugen, müssen verschiedene Operationen durch Funktionen des Tracers ersetzt bzw. erweitert werden. Wie man an dem Beispiel unschwer erkennen kann, besitzt die Version mit dem Tracer einen deutlich längeren

Programmcode, was bei der Implementierung zu einer größeren Arbeitslast führen kann. Da sich der Tracer auch negativ auf die Laufzeit des Programms auswirken kann, ist es in vielen Situationen nicht erwünscht, ihn in den eigentlichen Release-Code einzubauen, sondern eher in eine eigenständige Implementierung, welche nur für den Tracer verwendet werden. Um dies zu automatisieren wurde ein zusätzliches Programm implementiert, welches in der Lage ist, den Tracer in normalen Go-Code einzufügen. Die Implementierung, welche ebenfalls in [5] zur Verfügung steht, arbeitet mit einem Abstract Syntax Tree. Bei dem Durchlaufen dieses Baums werden die entsprechenden Operationen in dem Programm erkannt, und durch ihre entsprechenden Tracer-Funktionen ersetzt bzw. ergänzt. Neben dem Ersetzen der verschiedenen Operationen werden außerdem einige Funktionen hinzugefügt. Zu Begin der Main-Funktion des Programms wird der Tracer initialisiert. Zusätzlich wird eine zusätzliche Go-Routine gestartet, in welcher ein Timer läuft. Ist dieser abgelaufen, wird der Trace ausgegeben **(TODO: (bzw. die Analyse gestartet))**, auch wenn das Programm noch nicht vollständig durchlaufen ist. Dies führt dazu, dass auch Programme, in welchen ein Deadlock aufgetreten ist, analysiert werden können. **(EXTEND: Instrumenter)**

4 Zusammenfassung

(TODO: Zusammenfassung schreiben)

5 Acknowledgments

(TODO: Acknowledgments schreiben)

Abbildungsverzeichnis

Tabellenverzeichnis

ToDo Counters

(TODO: Remove ToDo Counter List)

To Dos: 6; 1, 2, 3, 4, 5, 6

Parts to extend: 3; 1, 2, 3

Draft parts: 2; 1, 2

Literaturverzeichnis

[1] S. Taheri and G. Gopalakrishnan, "Goat: Automated concurrency analysis and debugging tool for go," in *2021 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 138–150, 2021.

[2] I. Danyliuk, "Visualizing concurrency in go," 2016. `https://divan.dev/posts/go_concurrency_visualize/`.

[3] The Go Team, "Go documentation: trace," 2022. `https://pkg.go.dev/cmd/trace`.

[4] M. Sulzmann and K. Stadtmüller, "Two-phase dynamic analysis of message-passing go programs based on vector clocks," *CoRR*, vol. abs/1807.03585, 2018.

[5] E. Kassubek, "GoChan." `https://github.com/ErikKassubek/GoChan`, 2022. v 0.1.