Dynamic Parallel Checker

Semesterarbeit - Technische Dokumentation

Studenten:

Fabian Keller
Semester 5
f3keller@hsr.ch

Dominik Heeb
Semester 5
d1heeb@hsr.ch

Dozent:

Prof. Dr. Luc Bläser
lblaeser@hsr.ch

Eine Studie über Dynamic Parallel Checking Methoden

Inhaltsverzeichnis

1 Abstract

Die Arbeit "Dynamic Parallel Checker" behandelt die Entwicklung und Implementation eines Algorithmus zur Erkennung von Nebenläufigkeitsfehlern (Race Conditions) während der Laufzeit (dynamisch).
Der entwickelte Algorithmus basiert auf dem Vector Clock Algorithmus von Colin J. Fidge und Friedmann Mattern [ACSC, 1988]. Mit dem Vector Clock Algortihmus ist es möglich, nebenläufige Schreib- und Lesezugriffe in eine partielle Ordnung zu bringen, die sog. Happend-Before Beziehung. Locks und Unlocks sowie Thread oder Task Starts und Joins implizieren dabei eine Synchronisation der Vector Clocks zwischen den involvierten Threads bzw. Tasks, also eine Happend-Before Beziehung. Pro Thread oder Task werden alle nötigen Ereignisse, Zugriffe sowie Synchronisationen, in einer History protokolliert. Dies erlaubt es, anschliessend Data Races zu identifizieren: Ein Data Race liegt vor, wenn Zugriffe ohne Happened-Before Beziehung mit mindestens einer Schreiboperation auf dieselben Ressourcen stattgefunden haben.
Die Implementation des Algorithmus instrumentiert Microsoft Intermediate Language Code (MSIL) mit Hilfe der Mono Cecil Bibliothek. Daher ist es möglich, mit dem Dynamic Parallel Checker alle für das .NET-Framework entwickelten Applikationen zu überwachen. Der bestehende MSIL Code wird um Codestellen erweitert, welche die Dynamic Parallel Checker Library aufrufen und mit Informationen beliefern. Diese Informationen werden schliesslich von Algorithmus verwertet, um Data Races zu detektieren.

2 Technischer Bericht

2.1 Einleitung und Übersicht

Dieses Kapitel beinhaltet eine Beschreibung der verwendeten Konzepte und Technologien dieses Projekts. Für das Verständnis des gesamten technischen Berichts empfiehlt sich das Lesen dieses Teils. Bei bereits vorhandener Fachkenntnis bezüglich der Thematik kann dieser Teil gerne übersprungen werden und direkt die Implementation betrachtet werden.

2.1.1 Dynamic Checker

Um Race Conditions (sh. 2.1.2) zu erkennen, gibt es verschiedene Ansätze. Zwei davon sind, der Dynamic Checker und der Static Checker. In diesem Projekt wird ein Dynamic Checker entwickelt. Ein Dynamic Checker hat die Aufgabe, Race Conditions zur Laufzeit (dynamisch) zu erkennen. Der Static Checker im Gegensatz behandelt das Erkennen zur Entwicklungszeit. Um einen Dynamic Checker realisieren zu können, muss ein fertiges Programm instrumentiert werden. Mit Instrumentation ist gemeint, dass der kompilierte Code angepasst wird, so dass er während der Laufzeit einen Erkennungsalgorithmus ausführen kann, jedoch nicht das Verhalten des Programms verändert.
Der Checker analysiert hauptsächlich Lese- und Schreibzugriffe auf Variablen. Um die Präzision zu erhöhen, müssen auch Lock/ Unlock und Thread.Start usw. ausgelesen werden.
Mehr Informationen dazu unter: 2.2.4 Vector Clock Algorithmus

2.1.2 Race Condition

Bei Race Conditions handelt es sich um Speicherzugriffsfehler. Sie entstehen, wenn mehrere Threads auf gemeinsame Ressourcen (Variablen, Fields, Collections, usw.) ohne genügende Synchronisation zugreifen. Diese können zu unerwarteten Resultaten oder Effekten führen. Eine Untergruppe der Race Conditions bilden die Data Races. Sie entstehen durch eine ungenügende Synchronisation, beim Zugriff auf Speicherstellen. Race Conditions können durch richtige Synchronisation (z.B. mit Sperren eines kritischen Abschnittes, oder durch Zustandssynchronisation) verhindert werden. In Abbildung 1 wird ein Code beschrieben, welcher diese Synchronisation komplett weglässt und daher Data Races entstehen.

Variable = 100	
Thread 1	Thread 2
Variable + 200	Variable - 100

Abbildung 1: Ein unsynchronisierter Zugriff auf eine Variable

Wenn zwei Threads den Code unter Abbildung 1 nebenläufig ausführen, ist nicht deterministisch, wie die Threads zueinander verzahnt oder gleichzeitig laufen. Die Situation in Abbildung 2 kann dadurch entstehen. Thread 1 und Thread 2 lesen den Wert in ihr eigenes Register. Die Addition erfolgt ebenfalls auf dem Register, wodurch jeder einen eigenen neuen Wert besitzt. Thread 1 schreibt seinen neuen Wert in das Memory, welcher vom neuen Wert von Thread 2 überschrieben wird. Durch diesen Vorgang entsteht das Resultat 0. Da die zwei Threads unterschiedliche Register besitzen, werden die Änderungen an den Variablen nicht an den anderen Thread übertragen.
Bei korrekter Synchronisation wären beide Thread-Blöcke nacheinander durchgeführt worden, was zu einem korrekten Resultat von 200 geführt hätte.

Thread 1	Thread 2
Lesen in Register = 100	
	Lesen in Register = 100
Addieren im Register 100 + 200	Addieren im Register 100 - 100
Speichern in Memory 300	
	Speichern in Memory 0
Resultat = 0	

Abbildung 2: Ablauf eines unsynchronisierten Zugriff

Data Races wie unter Abbildung 2 bleiben evtl. unentdeckt, da kein effektiver Fehlzustand im System entsteht, sondern Werte nicht richtig angepasst werden. Race Conditions zeichnen sich auch daher aus, dass sie stark vom Verarbeitungsablauf abhängen und daher nicht immer auftreten müssen. Wenn also die Threads in Abbildung 2 in der richtigen Reihenfolge verarbeitet werden, wird evtl. kein Fehler entstehen.

Thread-Synchronisation

Data Races können durch richtige Synchronisation verhindert werden. Eine Möglichkeit zur Synchronisation sind Locks. Diese Sperren erlauben innerhalb des Locks nur einem Thread zu arbeiten. Dies kann Data Races verhindern, wenn das Locking korrekt durchgeführt wird, also kein Code im System nebenläufig ohne Lock auf die Variable zugreift. Es sollte jedoch beachtet werden, das Synchronisation relativ teuer ist und verhindert, das der Code optimiert wird. Daher sollte nicht der gesamte Code synchronisiert werden.

2.1.3 Vector Clock

Die einzelnen Lese- und Schreibzugriffe in einem System können nebenläufig in unterschiedlichen Threads ablaufen und dadurch nicht in eine totale Ordnung gebracht werden. Um nun jedem einzelnen Zugriff einen Zeitstempel zuzuordnen verwenden der Dynamic Parallel Checker eine Vector Clock. Diese Vector Clock wird benötigt, um eine Aussage über die Nebenläufigkeit der einzelnen Zugriffe zu machen.
Die Vector Clock basiert auf der Lamport's Clock von Leslie B. Lamport [MCA, 1978]. Jeder Teilnehmer in einem System, in diesem Fall wären dies die einzelnen Threads, besitzt ein eigener Zeitstempel. Der eigene Zeitstempel kann unabhängig inkrementiert werden. Eine Synchronisation zwischen den Zeitstempeln der Threads findet jedoch nur statt, wenn ein Synchronisationspunkt (Memory Barrier) vorliegt. Ein Synchronisationspunkt ist z.B. wenn ein Thread1 ein Lock auf ein Objekt a macht und zuvor ein anderer Thread2 ein Unlock auf das Objekt a gemacht hat. Dadurch ist sichergestellt, dass jeder Zugriff vor dem Unlock von Thread1 sicher vor jedem Zugriff nach dem Lock von Thread2 stattgefunden hat. In diesem Fall wird nun der Zeitstempel von Thread2 mit dem Zeitstempel von Thread 1 synchronisiert. Der Zeitstempel von Thread1 bleibt wie gehabt. Mehr Informationen über die einzelnen Synchronisationspunkte finden Sie im Kapitel 2.2 Implementation.
Die Vector Clock erweitert die Lamport Clock nun dadurch, dass jeder Thread nicht nur einen globalen Zeitstempel besitzt sondern einen Vektor, der für jeden Thread einen eigenen Zeitstempel mitführt. D.h. jeder Thread besitzt nun einen eigenen Vektor mit den Zeitstempeln der anderen Threads. Jedoch befindet sich darin nicht der aktuelle Zeitstempel sondern der Zeitstempel der letzten Synchronisation mit dem jeweiligen Thread. Mit Hilfe der Vector Clock können die einzelnen Lese- und Schreibzugriffe eines Thread in eine partielle Ordnung gebracht werden.
Eine Vector Clock wie in Abbildung 3 besteht aus einem Vektor, der Key-Value Paare beinhaltet. Der Key ist die Thread- oder TaskId und der Value ist der spezifische Wert des Zeitstempels.

$$\begin{pmatrix} ThreadId_1 & 2 \\ ThreadId_2 & 0 \\ ThreadId_3 & 3 \\ ... & ... \end{pmatrix}$$

Abbildung 3: Beispiel einer Vector Clock

2.1.4 Happened-Before Beziehung

Um mit Hilfe der Vector Clock die Nebenläufigkeit von Lese- und Schreibzugriffen zu bestimmen, verwenden wir die Happened-Before Beziehung von Leslie B. Lamport [MCA, 1978]. Diese Beziehung wird in der Lamport Clock und in der Vector Clock verwendet, um eine partielle Ordnung innerhalb mehrerer Ereignissen herzustellen. Bei nebenläufigen Programmen kann keine totale Ordnung erreicht werden, daher muss mit einer partiellen Ordnung gearbeitet werden. Unserem Algorithmus genügt die partielle Ordnung soweit, da ihn lediglich die Zugriffe interessieren, die nebenläufig stattgefunden haben.
Die Happened-Before Beziehung vergleicht die Zeitstempel von zwei unterschiedlichen Zugriffen und kann dann eine Aussage darüber machen, ob diese nun nebenläufig oder zwingend nacheinander stattgefunden haben. Folgende Eigenschaften definieren die Happened-Before Beziehung: (-> = "Happened Before")

- Auf demselben Thread oder Task: a -> b, wenn die Zeit von a < b. (Zeit ist durch Vector Clock gegeben)
- Wenn eine Synchronisation zwischen zwei Threads oder Tasks durchgeführt wurde, dann gilt a -> b, wenn a der Thread oder Task ist von dem aus synchronisiert wird und b der Thread oder Task ist zu dem synchronisiert wird.
- Für drei Zugriffe mit Synchronisation a, b, c. Wenn a -> b und b -> c, dann gilt a -> c (Transitivität).

Die Happened-Before Beziehung kann somit eine Aussage darüber machen, ob zwischen zwei unterschiedlichen Lese- oder Schreibzugriffen eine Synchronisation stattgefunden hat. Die Synchronisation zeigt, dass diese Zugriffe zwingend sequentiell abgelaufen sind und dadurch keine potenzielle Data Race darstellen können. Die Zugriffe können folgende Beziehungen zueinander haben:

Happened Before

(x1, x2, x3, ...) -> (y1, y2, y3, ...)
x1 < y1
x2 <= y2
x3 <= y3
... <= ...

Abbildung 4: Beispiel Happened-Before Beziehung

Die Happened-Before Beziehung in Abbildung 4 ist wie folgt definiert: Der Zeitstempel eines Threads 1 in einer Vector Clock ist kleiner oder gleich (=<) dem Zeitstempel des selben Threads 1 in der Vector Clock von Thread 2. Mindestens eine Komponente der gesamten Vector Clock muss aber echt kleiner (<) sein als in der anderen Vector Clock. In dieser Beziehung hat sicher eine Synchronisation zwischen den beiden Threads 1 und 2 stattgefunden

und dadurch sind diese beiden Zugriffe keine potenzielle Race Condition.
Beispiel (T1 happened before T2):

$$T1 = \begin{pmatrix} T1 & 1 \\ T2 & 0 \end{pmatrix}, T2 = \begin{pmatrix} T1 & 1 \\ T2 & 2 \end{pmatrix}$$

Concurrent

Concurrent (Nebenläufig) bedeutet, dass zwischen zwei Zugriffen auf zwei unterschiedlichen Threads keine Synchronisation durch z.B. einen Lock geschah. Diese Zugriffe können eine potentielle Race Condition darstellen, falls sie auf die selbe Ressource zugegriffen haben.
Sollten also beim Vergleich der Vector Clocks von zwei Zugriffen nicht alle Elemente aus Vektor 1 grösser gleich (>=) und mindestens eines echt grösser (>), oder kleiner gleich (<=) und mindestens eines echt kleiner (<) sein wie das dazugehörige Element im Vektor 2, passierten diese Zugriffe nebenläufig. Mit anderen Worten, wenn die erste Definition von "Happened-Before" auf die Vector Clocks nicht angewendet werden kann. Unser Algorithmus interessiert sich genau für diese Zugriffspaare und daher ist die Happened-Before Beziehung zentral für das Funktionieren des Algorithmus.
Beispiel:

$$T1 = \begin{pmatrix} T1 & 2 \\ T2 & 1 \end{pmatrix}, T2 = \begin{pmatrix} T1 & 1 \\ T2 & 2 \end{pmatrix}$$

2.1.5 Microsoft IL Code

Der Microsoft IL (Intermediate Language) Code [ECMA, 2012] wird verwendet, um einen sprach- und plattformunabhängigen Zwischencode zwischen dem Compiler und dem Prozessor zu bilden. Da der IL Code einheitlich definiert ist, ist er unabhängig von der verwendeten Hardware. In diesem Zustand wird das Programm zum Kunden ausgeliefert und erst auf der Maschine durch einen JIT (Just in Time) Compiler zur Laufzeit kompiliert. Dieses Verfahren hat den Vorteil, dass der Code auf das jeweilige System abgestimmt wird und dadurch die Funktionen des Prozessors optimal ausnützt. Der Nachteil ist jedoch der Performanceverlust, da jede Methode vor dem Ausführen noch kompiliert werden muss.
Für die Instrumentation, welche für den Dynamic Parallel Checker verwendet wird, ist der IL Code ideal, da er vereinheitlicht ist und daher nur eine Implementation entwickelt werden muss, welche für alle .NET Sprachen, sowie jedes Laufzeitsystem welches die .NET Runtime unterstützt funktioniert.
Der MSIL Code ist Stack-basiert. Dies bedeutet das alle Operationen auf einem Stack nach dem LIFO Prinzip (Last In, First Out) durchgeführt werden. Der Stack wird dazu verwendet, die Werte vorzubereiten, zu verarbeiten und wieder vom Stack abzubauen. Abbildung 5 zeigt, wie eine Operation durchgeführt wird und in welchen Schritten die Parameter auf dem Stack vorbereitet werden müssen.

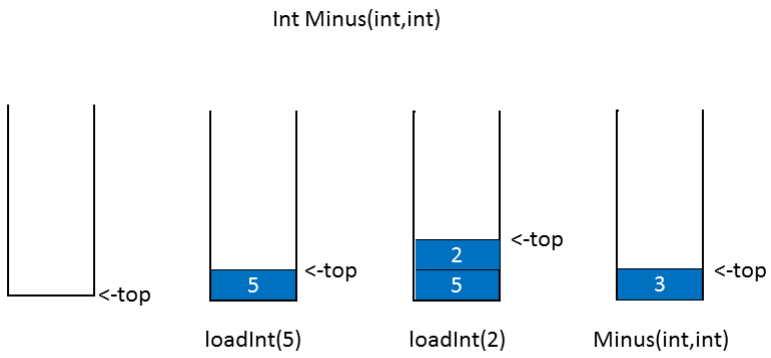

Abbildung 5: Beispiel einer Minus Operation mit den Parametern (5,2)

Eine Instruktion im MSIL Code setzt sich jeweils aus einem OpCode und einem Operand zusammen.

```
ldsflda     class TestProgram.NewObject TestProgram.Program::_obj1
```

In diesem IL Code ist "ldsflda" der OpCode und "class TestProgram.NewObject TestProgram .Program::_obj1" der Operand. Der OpCode (Operation Code) identifiziert was die Operation durchführen soll. Hier "ldsflda" (Load Static Field Address). Dabei wird vom Operand definierten Static Field die Speicheradresse ausgelesen und auf den Stack gelegt. Der Operand ist daher jeweils die Ressource, welche für den OpCode verwendet wird. Der Typ des Operand ist durch den OpCode definiert.
Um erfolgreich den Source Code für den Dynamic Parallel Checker zu instrumentieren (sh.

Abbildung 6), muss der Aufbau des Stacks immer sichergestellt werden. Daher müssen Werte um Sie aus dem Stack herauszulesen, immer dupliziert werden, damit das Programm erfolgreich durchgeführt werden kann.

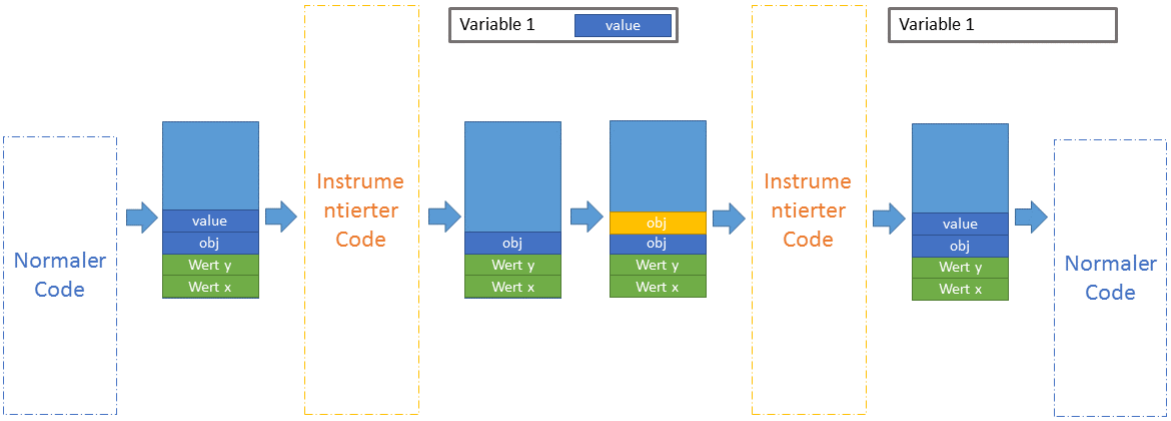

Abbildung 6: Beispiel Code Instrumentation, Stack Verhalten

In Abbildung 6 ist dargestellt, wie eine Instrumentation den Stack verwaltet und dabei die benötigte Information "obj" aus dem Stack extrahiert und den Ursprungszustand wiederherstellt. Da bei einem LIFO Stack nur auf das oberste Element zugegriffen werden kann, wird der Top-Wert "value" in eine lokale Variable gespeichert. Nach dem Abbau des duplizieren "obj" wird der Wert wieder aus der Variable auf den Stack gelegt.

2.1.6 Instrumentation

Für die Durchführung des Vector Clock Algorithmus (sh. 2.2.4) ist es wichtig die Informationen über Lese- / Schreibzugriffe sowie Locks und Thread.Starts zu erhalten. Der bestehende IL Code wird angepasst (instrumentiert), damit diese Informationen aus dem IL Code an die Dynamic Parallel Checker Library übermittelt werden. Dazu wird der Code analysiert und an Stellen mit wichtigen Zugriffen um Code erweitert. Wie in Kapitel 2.1.5 bereits beschrieben, muss dabei der korrekte Aufbau des Stack jederzeit sichergestellt werden. Für die Instrumentation des Codes wird die Library Mono.Cecil aus dem Mono Projekt verwendet [MONO, 2015]. Die Library erlaubt es den IL Code zu lesen und zu bearbeiten.

```
IL_001a:  box         [mscorlib]System.Int32
IL_001f:  stsfld      object TestProgram.Program::_a
```

Abbildung 7: Beispiel ohne Instrumentation

Abbildung 7 zeigt einen typischen Schreibzugriff im MS IL. Der Befehl "box" führt ein Boxing eines Integers durch. Dabei wird der Wert des Integers von einem Valuetype zu einem Referenztyp umgebaut. Auf dem Stack liegt nach dem Boxing die Adresse des Integer-Werts. Diese Adresse wird dann verwendet um den Wert in die Variable "_a" mit dem Befehl "stsfld"(Store Static Field) zu speichern.

```
IL_001a:  box         [mscorlib]System.Int32
//----------Instrumented Code...----------------
IL_001f:  ldsflda     object TestProgram.Program::_a
IL_0024:  ldc.i4.s    31
IL_0026:  ldstr       "System.Void TestProgram.Program::Test()"
IL_002b:  call        void [DPCLibrary]DPCLibrary.DpcLibrary::
                                   WriteAccess(int32,int32,string)
//------------...End Instrumentation--------------
IL_0030:  stsfld      object TestProgram.Program::_a
```

Abbildung 8: Instrumentierter IL Code

Abbildung 8 zeigt den Schreibzugriff nach der Instrumentation des Dynamic Parallel Checkers. Wie in Abbildung 7, startet der Code mit "box" und endet mit "stsfld". Dazwischen wurde der Code um den Aufruf der Dynamic Parallel Checker Library erweitert. Um zu wissen, auf welche Variable der Schreibzugriff durchgeführt wird, wird mit "ldsflda" (Load static Field Address) die Adresse der Ressource ausgelesen und auf den Stack gelegt. Danach wird die Zeilennummer des Schreibzugriff auf den Stack gelegt (Die Zeilennummer zeigt die Zeile vor der Instrumentation), um bei einem Fehler aufzuzeigen, welche Codezeilen diesen verursacht haben. Als nächstes wird der Methodennamen als String auf den Stack gelegt. Als letzte instrumentierte Zeile wird mit der Methode "WriteAccess(int32,int32,string)" die Dynamic Parallel Checker Library aufgerufen und die Informationen übergeben. Nach dem Instrumentierten Code hat der Stack wieder den gleichen Zustand wie nach dem "Box" Befehl in Abbildung 7.

2.2 Implementation

In diesem Kapitel wird die Implementation des Dynamic Prallel Checker genauer erklärt. Unter 2.2.1 ist das Beispiel beschrieben an welchem, in den folgenden Kapiteln, die Funktion des Dynamic Parallel Checkers beschrieben wird. Der Dynamic Parallel Checker wurde mit C#.Net entwickelt.

2.2.1 Beispiel

Der Vektor Clock Algorithmus wird in diesem Kapitel mit Hilfe von folgendem Pseudo-Code Beispiel genauer erläutert.

```
//... Deklaration a,b,locka,lockb...

Thread.Start(() =>                  // start thread2 (T2)
{
        lock(lockb) {
                b = 3;
        }
        b = 4;
        lock(locka) {
                Console.WriteLine(a);
        }
});
Thread.Start(() =>                  // start thread3 (T3)
{
        lock(locka) {
                a = 2;
        }
        b = 5;
        lock(lockb) {
                Console.WriteLine(b);
        }
})
lock(locka) {
        a = 3;
}
lock(lockb) {
        b = 6;
}
Console.WriteLine(a);
```

Abbildung 9: Beispiel für die Erklärung des Algorithmus

Da der Code in Abbildung 9 asynchron durchgeführt wird, werden die 3 enstehenden Thread parallel ausgeführt. Abbildung 10 zeigt welche relevanten Aufrufen die einzelnen Threads parallel durchführen. "w(), r()" stehen für Write und Read Vorgänge des Threads. Der Buchstabe in der Klammer für die Ressource.

T1	T2	T3
lock(locka)	lock(lockb)	lock(locka)
w(a)	w(b)	w(a)
unlock(locka)	unlock(lockb)	unlock(locka)
lock(lockb)	w(b)	w(b)
w(b)	lock(locka)	lock(lockb)
unlock(lockb)	r(a)	r(b)
r(a)	unlock(locka)	unlock(lockb)

Abbildung 10: Darstellung der parallelen Vorgänge von Abbildung 9

2.2.2 Vector Clock pro Thread

Die Vector Clock eignet sich optimal für die History des Dynamic Parallel Checkers. Jeder Thread erhält seine eigene Vektor Clock, in der er die Zeitstempel der anderen Threads mitführt. Es wird nicht der aktuelle Zeitstempel mitgeführt, sondern der Zeitstempel der bei der letzten direkten oder indirekten Synchronisation mit diesem Thread mitgeliefert wurde. Bei einem Synchronisationspunkt wird die Vector Clock mit der Vector Clock aus der Lock-History synchronisiert (sh. 2.2.3). Die Reihenfolge der Synchronisation ist jedoch nicht deterministisch und hängt vom Thread-Scheduler ab.
Unser Algorithmus verwendet die Vector Clock des aktuellen Threads, um diese mit den Vector Clocks der History der anderen Threads zu vergleichen. Dadurch kann in Erfahrung gebracht werden, welche Zugriffe nebenläufig zu dem aktuellen Zugriff stattgefunden haben. Dies zeigt die Notwendigkeit, dass jeder Thread seine eigene Vector Clock hat.

2.2.3 Lock-History

Während der kompletten Überwachung wird eine Lock-History wie z.B. in Abbildung 11 geführt. Diese beinhaltet die Information, welcher Thread zuletzt auf welche Ressource zu welcher Zeit (Vector Clock) einen Lock durchgeführt hat. Diese Information wird benötigt um z.B. eine Synchronisation zwischen dem Thread, der den letzten Unlock auf eine Ressource durchgeführt hat, mit dem aktuellen Thread vorzunehmen. Bei jedem Unlock wird ein neuer Lock-History Eintrag erstellt, falls noch keiner vorhanden ist. Sollte jedoch bereits einer vorhanden sein, wird dieser einfach überschrieben.

Lock-History:

Vektor	Ressource	ThreadId
(2,1,0)	locka	1
(0,1,0)	lockb	2

Abbildung 11: Beispiel Lock-History

Synchronisation der Vector Clock

Wenn z.B. im Beispielprogramm (Abbildung 9) Thread1 (T1) vor Thread3 (T3) in den Lock(a) Bereich kommt, schreibt er beim Verlassen des gesicherten Bereichs einen Eintrag in die Lock-History. Wenn nun T3 den Lock(a) beziehen möchte, prüft der Algorithmus zuerst, ob ein Eintrag in der Lock-History vorhanden ist. Ist dies der Fall, wird die eigene Vector Clock (a3, b3, c3) mit der Vector Clock des History Eintrags (a1, b1, c1) synchronisiert (Abbildung 12).

Vorgehensweise bei der Synchronisation:

- Bei jeder ThreadId oder TaskId (sh. 3.2.1), die in den beiden Vector Clocks vorkommt, wird der grössere Wert aus beiden Vector Clocks verwendet. z.B. MAX(a1, a2)
- Die synchronisierende Vector Clock inkrementiert den Wert der eigenen ThreadId oder TaskId. z.B. MAX(c1, c3) + 1

$$T1 = \begin{pmatrix} T1 & 2 \\ T2 & 1 \end{pmatrix}, T3_{alt} = \begin{pmatrix} T2 & 2 \\ T3 & 3 \end{pmatrix} => T3_{neu} = \begin{pmatrix} T1 & MAX(2,0) = 2 \\ T2 & MAX(1,2) = 2 \\ T3 & MAX(0,3) + 1 = 4 \end{pmatrix}$$

Abbildung 12: Beispiel Vector Clock Synchronisation

Somit stehen alle Zugriffe, von vor der Synchronisation von Thread1, in der Happened-Before Beziehung mit allen Zugriffen von Thread3 nach der Synchronisation. Somit können diese Zugriffe keine Race Condition darstellen.

2.2.4 Vector Clock Algorithmus

Mit Hilfe der Instrumentierung 8 wird bei jedem Auftreten eines Ereignisses (Lese- oder Schreibzugriff, Lock oder Unlock auf Ressource, Start eines Threads usw.) die Dynamic Parallel Checker Library aufgerufen und anschliessend der Algorithmus durchlaufen.

Handelt es sich bei dem Ereignis um einen Lock auf eine Ressource, wird das Ablaufdiagramm in Abbildung 13 durchlaufen. Zuerst wird überprüft, ob in der Lock-History bereits ein Eintrag , auf die gleiche Ressource, vorhanden ist. Ist dies der Fall, findet eine Synchronisation zwischen der Vector Clock des aktuellen Threads und der Vector Clock des Eintrags in der Lock-History statt. Die Synchronisation ist in 2.2.3 beschrieben.

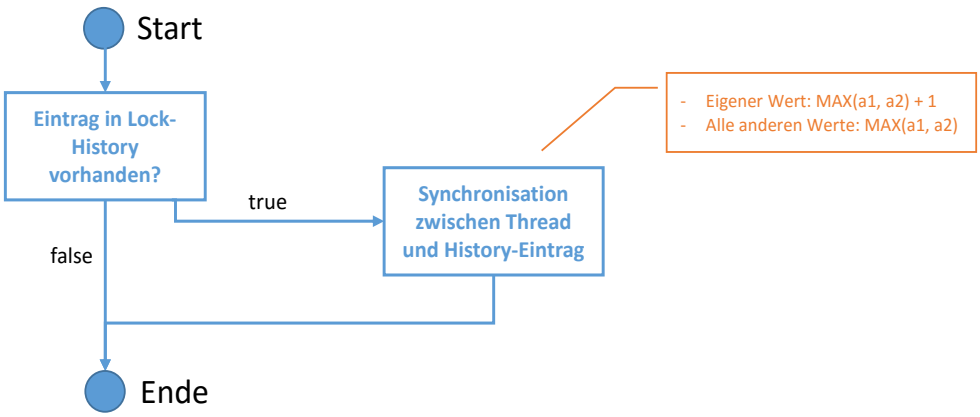

Abbildung 13: Algorithmus - Lock

Handelt es sich bei dem Ereignis um einen Unlock auf eine Ressource, wird das Ablaufdiagramm in Abbildung 14 durchlaufen. Es wird sichergestellt, dass die aktuelle Vector Clock zu der Ressource in die Lock-History geschrieben wird. Dabei wird überprüft, ob bereits ein Eintrag zu dieser Ressource in der Lock-History vorhanden ist. Wenn ja, wird dieser mit der neuen Zeit und der ThreadId oder TaskId überschrieben. Falls kein Eintrag vorhanden ist, wird ein neuer erstellt. Zudem wird der eigene Wert in der Vector Clock des aktuellen Threads oder Tasks inkrementiert. Dies dient dazu, dass beim nächsten Auftreten eines Locks auf dieselbe Ressource, nach der Synchronisation, die Happened-Before Beziehung gilt.

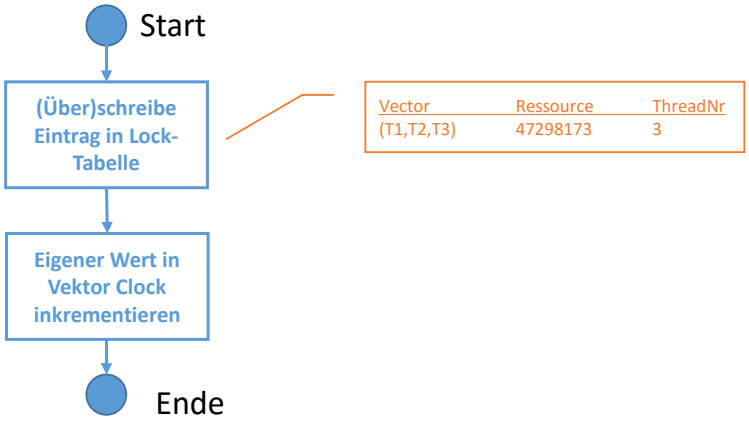

Abbildung 14: Algorithmus - Unlock

Handelt es sich bei dem Ereignis um einen Lese- oder Schreibzugriff auf eine Ressource, wird das Ablaufdiagramm in Abbildung 15 durchlaufen. Zuerst wird das Ereignis in die Programm-History des aktuellen Threads oder Tasks abgelegt. Sollte während der selben Vector Clock Zeit im aktuellen Thread bereits ein Zugriff auf diese Ressource stattgefunden haben, wird der Zugriffstyp (Read oder Write) aktualisiert. Solange nur Lesezugriffe stattgefunden haben, ist der Zugriffstyp immer "Read". Sobald jedoch ein Schreibzugriff stattgefunden hat, wird dieser Typ auf "Write" geändert und wird nie mehr den Type "Read" erhalten. Dadurch werden die Zugriffe pro Zeitabschnitt eines Threads zusammengefasst und die Datenstruktur optimiert.
Anschliessend wird das aktuelle Ereignis auf Race Conditions geprüft. Dafür benötigt der Algorithmus alle Zugriffe, von den anderen Threads oder Tasks, die nebenläufig zu dem aktuellen Zugriff aufgetreten sind. Dies kann mit Hilfe der Happened-Before Beziehung überprüft werden. In einem dynamischen Checker hat das Scheduling einen grossen Einfluss auf die Reihenfolge der einzelnen Zugriffe und dadurch auch auf die Happened-Before Beziehungen zwischen zwei Zugriffen. Für jeden nebenläufigen Zugriff wird überprüft, ob auf die selbe Ressource zugegriffen wurde, wie im aktuellen Zugriff. Ist dies der Fall, wird überprüft, um was für einen Zugriffstyp es sich handelt (Lese- oder Schreibzugriff). Sollte einer der beiden Zugriffe ein Schreibzugriff sein, ist es eine Race Condition. Mehr Informationen zu Race Conditions in Kapitel 2.1.2. Sollte eine Race Condition erkannt werden, wird diese mit Hilfe

eines Logging Frameworks dem Benutzer gemeldet.

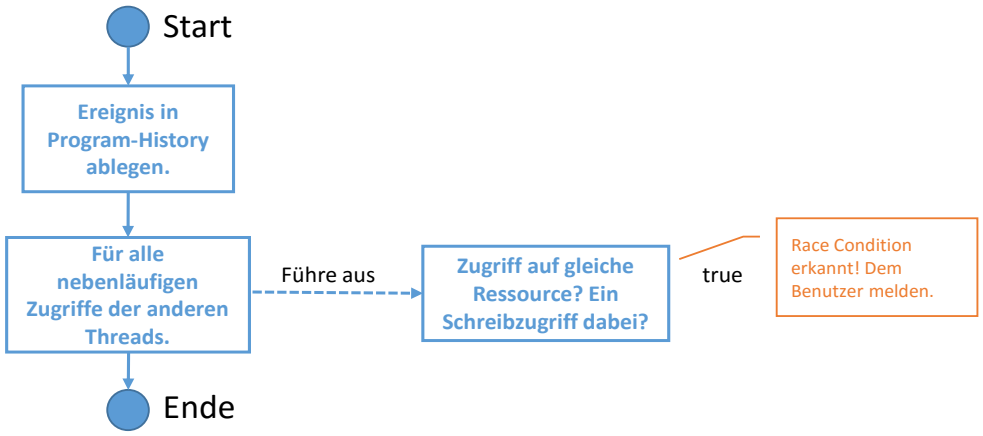

Abbildung 15: Algorithmus - Read/Write

Algorithmus Beispiel

In diesem Kapitel wird der Algorithmus auf das Beispielprogramm in Abbildung 9 angewendet. Dabei ist zu beachten, dass dieses Beispiel lediglich eine mögliche Ablaufsequenz aufzeigt. Änderst sich das Scheduling, ändert sich auch das komplette Verhalten des Algorithmus.

- *Alle Threads werden gestartet. Jede Instanz eines Threads oder Tasks instanziiert eine neue Vector Clock. Die Lock-History sowie die Programm-History ist noch leer.*

Vektor	Ressource	ThreadNr

ThreadNr	Ressource	Vektor	ZugriffsTyp

$$T1 = \begin{pmatrix} T1 & 1 \end{pmatrix}, T2 = \begin{pmatrix} T2 & 1 \end{pmatrix}, T3 = \begin{pmatrix} T3 & 1 \end{pmatrix}$$

- *lock(lockb), w(b), unlock(lockb) von Thread2*

Vektor	Ressource	ThreadNr
(0,1,0)	lockb	2

ThreadNr	Ressource	Vektor	ZugriffsTyp
2	b	(0,1,0)	write

$$T1 = \begin{pmatrix} T1 & 1 \end{pmatrix}, T2 = \begin{pmatrix} T2 & 2 \end{pmatrix}, T3 = \begin{pmatrix} T3 & 1 \end{pmatrix}$$

- *lock(locka), w(a), unlock(locka) von Thread1*

Vektor	Ressource	ThreadNr
(1,0,0)	locka	1
(0,1,0)	lockb	2

ThreadNr	Ressource	Vektor	ZugriffsTyp
2	b	(0,1,0)	write
1	a	(1,0,0)	write

$$T1 = \begin{pmatrix} T1 & 2 \end{pmatrix}, T2 = \begin{pmatrix} T2 & 2 \end{pmatrix}, T3 = \begin{pmatrix} T3 & 1 \end{pmatrix}$$

- *lock(locka), w(a), unlock(locka) von Thread3. Keine Race Condition, weil Happened Before Beziehung zwischen beiden Schreibzugriffen auf Ressource a.*

Vektor	Ressource	ThreadNr
(1,0,2)	locka	3
(0,1,0)	lockb	2

ThreadNr	Ressource	Vektor	ZugriffsTyp
2	b	(0,1,0)	write
1	a	(1,0,0)	write
3	a	(1,0,2)	write

$$T1 = \begin{pmatrix} T1 & 2 \end{pmatrix}, T2 = \begin{pmatrix} T2 & 2 \end{pmatrix}, T3 = \begin{pmatrix} T1 & 1 \\ T3 & 3 \end{pmatrix}$$

- *w(b) von Thread2*

Vektor	Ressource	ThreadNr
(1,0,2)	locka	3
(0,1,0)	lockb	2

ThreadNr	Ressource	Vektor	ZugriffsTyp
2	b	(0,1,0)	write
1	a	(1,0,0)	write
3	a	(1,0,2)	write
2	b	(0,2,0)	write

$$T1 = \begin{pmatrix} T1 & 2 \end{pmatrix}, T2 = \begin{pmatrix} T2 & 2 \end{pmatrix}, T3 = \begin{pmatrix} T1 & 1 \\ T3 & 3 \end{pmatrix}$$

- *w(b) von Thread3. Eine Race Condition, weil keine Happend-Before Beziehung.*

Vektor	Ressource	ThreadNr
(1,0,2)	locka	3
(0,1,0)	lockb	2

ThreadNr	Ressource	Vektor	ZugriffsTyp
2	b	(0,1,0)	write
1	a	(1,0,0)	write
3	a	(1,0,2)	write
2	b	(0,2,0)	write
3	b	(1,0,3)	write

$$T1 = \begin{pmatrix} T1 & 2 \end{pmatrix}, T2 = \begin{pmatrix} T2 & 2 \end{pmatrix}, T3 = \begin{pmatrix} T1 & 1 \\ T3 & 3 \end{pmatrix}$$

- *lock(lockb), w(b), unlock(lockb) von Thread1. Zwei Race Conditions sind aufgetreten.*

Vektor	Ressource	ThreadNr
(1,0,2)	locka	3
(3,1,0)	lockb	1

ThreadNr	Ressource	Vektor	ZugriffsTyp
2	b	(0,1,0)	write
1	a	(1,0,0)	write
3	a	(1,0,2)	write
2	b	(0,2,0)	write
3	b	(1,0,3)	write
1	b	(3,1,0)	write

$$T1 = \begin{pmatrix} T1 & 4 \\ T2 & 1 \end{pmatrix}, T2 = \begin{pmatrix} T2 & 2 \end{pmatrix}, T3 = \begin{pmatrix} T1 & 1 \\ T3 & 3 \end{pmatrix}$$

- *lock(locka), r(a), unlock(locka) von Thread2. Eine Race Condition ist aufgetreten.*

Vektor	Ressource	ThreadNr
(1,3,2)	locka	2
(3,1,0)	lockb	1

ThreadNr	Ressource	Vektor	ZugriffsTyp
2	b	(0,1,0)	write
1	a	(1,0,0)	write
3	a	(1,0,2)	write
2	b	(0,2,0)	write
3	b	(1,0,3)	write
1	b	(3,1,0)	write
2	a	(1,3,2)	read

$$T1 = \begin{pmatrix} T1 & 4 \\ T2 & 1 \end{pmatrix}, T2 = \begin{pmatrix} T1 & 1 \\ T2 & 4 \\ T3 & 2 \end{pmatrix}, T3 = \begin{pmatrix} T1 & 1 \\ T3 & 3 \end{pmatrix}$$

- *lock(lockb), r(b), unlock(lockb) von Thread3. Eine Race Condition ist aufgetreten.*

Vektor	Ressource	ThreadNr
(1,3,2)	locka	2
(3,1,4)	lockb	3

ThreadNr	Ressource	Vektor	ZugriffsTyp
2	b	(0,1,0)	write
1	a	(1,0,0)	write
3	a	(1,0,2)	write
2	b	(0,2,0)	write
3	b	(1,0,3)	write
1	b	(3,1,0)	write
2	a	(1,3,2)	read
3	b	(3,1,4)	read

$$T1 = \begin{pmatrix} T1 & 4 \\ T2 & 1 \end{pmatrix}, T2 = \begin{pmatrix} T1 & 1 \\ T2 & 4 \\ T3 & 2 \end{pmatrix}, T3 = \begin{pmatrix} T1 & 3 \\ T2 & 1 \\ T3 & 5 \end{pmatrix}$$

- *r(a) von Thread1. Eine Race Condition ist aufgetreten.*

Vektor	Ressource	ThreadNr
(1,3,2)	locka	2
(3,1,4)	lockb	3

ThreadNr	Ressource	Vektor	ZugriffsTyp
2	b	(0,1,0)	write
1	a	(1,0,0)	write
3	a	(1,0,2)	write
...	...	...	...
2	a	(1,3,2)	read
3	b	(3,1,4)	read
1	a	(4,1,0)	read

$$T1 = \begin{pmatrix} T1 & 4 \\ T2 & 1 \end{pmatrix}, T2 = \begin{pmatrix} T1 & 1 \\ T2 & 4 \\ T3 & 2 \end{pmatrix}, T3 = \begin{pmatrix} T1 & 3 \\ T2 & 1 \\ T3 & 5 \end{pmatrix}$$

3 Schlussfolgerungen

Der entwickelte Dynamic Parallel Checker ermöglicht eine hohe Erkennungsrate von Race Conditions. Zur Zeit befinden sich jedoch noch viele Fehlmeldungen (False-Positives) unter den Meldungen, weil einige Synchronisationspunkte noch nicht instrumentiert werden (sh. Fehlende Synchronisationsoperationen).
Das heisst, dass das System einige Zugriffe als Race Conditions erkennt, obwohl zuvor vielleicht eine Synchronisation stattgefunden hat, sodass diese Zugriffe nicht mehr nebenläufig stattgefunden haben. Sobald die Methoden aus Kapitel Fehlende Synchronisationsoperationen instrumentiert sind, werden die Fehlmeldungen auf ein Minimum reduziert. Dies ist unumgänglich, wenn dieser Algorithmus in der Praxis genutzt werden sollte.

3.1 Performance

3.1.1 Testvorgang

Im Rahmen des Projektabschlusses wurde der Dynamic Parallel Checker einigen Performance Tests ausgesetzt. Dabei wurden Tools gesucht welche mit .Net programmiert wurden. Die Messungen zeigten starke Differenzen in der Performance des Checkers auf.

Testfall 1: TestTool

Dieses Tool wurde im Rahmen des Projekt für das Testen des Algorithmus entwickelt. Es handelt sich um eine Console-Application, welche verschiedene synchrone und asynchrone Read/Writes durchführt.

Resultat

Ohne DPC	Mit DPC
ca. 70ms	ca. 205ms

Faktor 2.9x

Das Resultat zeigt einen moderaten Anstieg der Verarbeitungszeit. In der Analyse zeigt sich in diesem Testfall, dass das TestTool wenige komplexe asynchrone Verarbeitungen durchführt. Daher werden wenige Vector Clocks erfasst und berechnet. Da das Tool vor allem aus Code besteht, welcher instrumentiert werden muss, ist die Grösse des Tools von 18KB ohne Instrumentierung auf 32KB angestiegen

Testfall 2: Tic-Tac-Toe

Um die Performance des Algorithmus mit vielen Read/Writes im synchronen Ablauf zu testen, wurde ein Tic-Tac-Toe Spiel gefunden und getestet. Da der Code rein synchron ist, werden wenige Vector Clocks generiert. Getestet wurde der Vorgang in welchem der Computergegner versucht den besten Zug zu evaluieren und durchzuführen. Dieser Vorgang ist stark rekursiv und führt zu vielen Read / Writes

Resultat

Ohne DPC	Mit DPC
ca. 80ms	ca. 833715ms

Faktor 10421x

Die Analyse des Resultats zeigt auf, das der extreme Performanceverlust dadurch entsteht, das das Tic-Tac-Toe Spiel durch ineinander verschachtelte For-Schleifen, welche in jedem Durchgang eine Rekursion starten, mit weiteren verschachtelten For-Schleifen, zu hohen Read/Write zahlen führen. Als Test wurden die Rekursionen eingeschränkt. Dadurch zeigte sich ein exponentieller Anstieg der Verarbeitungszeit durch die Rekursionen.

Testfall 3: Parallel File Search

Das Tool Parallel File Search wurde als Testfall evaluiert, da in diesem Tool durch verschachtelte "Parrallel.Foreach" viele Tasks gestartet werden. Die einzelnen Tasks besitzen wenig Read/Writes. Jedoch durch die Anzahl Tasks (für jeden Ordner im Filesystem wird ein Task gestartet) muss der Algorithmus viele Vector Clocks (für jeden Task eine Vector Clock) erstellen. Im Test wurden 358 Ordner überprüft.

Resultat

Ohne DPC	Mit DPC
ca. 26ms	ca. 36992ms

Faktor 1422x

In der Analyse zeigt sich, das der Parallel File Search viele Tasks startet. Diese müssen als Vector Clock erfasst werden und mit allen anderen überprüft werden. Dadurch steigt der Aufwand des Algorithmus mit der Anzahl Thread/Tasks an.

3.1.2 Schlussfolgerung Performance

Durch die Performance-Messungen zeigt sich, dass der Dynamic Parallel Checker Algorithmus stark abhängig ist von der Implementation der Anwendung welche überprüft werden soll. Testfall 2 zeigte auf, dass Tools mit stark verschachtelten For-Schleifen die Performance des Algorithmus stark beeinflussen, da sie viele Read und Writes produzieren. Testfall 3 zeigt auf, dass viele Threads und Tasks die gestartet werden, viele Vector Clocks ergeben. Dadurch muss der Algorithmus all diese Vector Clocks miteinander vergleichen. Der Arbeitsaufwand steigt dadurch steil an.
Die Schlussfolgerung aus diesen Testfällen ist, dass der Vector Clock Algorithmus je grösser die zu verarbeitende Applikation ist, langsamer wird. Dadurch wäre der Algorithmus effizienter bei der Verarbeitung kleinerer Codes, wie z.B. Unit-Tests (sh. Vision 4.3).

3.2 Technische Findings

3.2.1 Task/Threads

Das .Net Framework von Microsoft kennt zwei asynchrone Operationstypen (Task oder Threads). Der Unterschied dieser zwei Typen liegt auf der Ausführungsumgebung. Threads werden direkt gestartet und vom Prozessor parallel verarbeitet. Task wurden zusammen mit der Task Parallel Library eingeführt. Dieser Typ wird nicht direkt ausgeführt, sondern in eine Queue geladen und dann von Worker Threads berechnet. Die Worker Threads werden vom .Net Framework automatisch gestartet. Die Anzahl wird durch die Laufzeitumgebung bestimmt, z.B. wie viele Kernen der Prozessor hat. Der Vorteil dieser Methode gegenüber der Threads ist es, das die Belastung auf den Arbeitsspeicher und Prozessor bei vielen parallelen Vorgängen optimiert und gesenkt wird. Der Nachteil ist, das es schneller zu Deadlocks kommen kann, wenn die Threads untereinander abhängig sind.
Für die Entwicklung des Dynamic Parallel Checker ist der Unterschied zwischen Task und Threads dahingehend wichtig, da die ID des Threads bei Tasks der ID des Worker Thread entspricht. Um den Detailgrad der Erkennung zu erhöhen, wurde bei Tasks die Task ID und nicht die Thread ID für die Erkennung verwendet.

3.2.2 Garbage Collector

Der Garbage Colletor (GC) verwaltet die Belegung und Freigabe von Arbeitsspeicher einer Anwendung. Bei der Erzeugung eines Objektes, wird dem Objekt die Speicherplatzadresse auf dem Heap zugewiesen. Da nicht unendlich Arbeitsspeicher vorhanden ist, greift der GC ein, und reorganisiert die Speicherplatzbelegung. Dabei kann es vorkommen, dass bestehenden Objekten eine neue Speicherstelle und somit auch eine neue Speicheradresse zugewiesen wird. Dies kann für den Dynamic Parallel Checker ein grosses Problem darstellen.
Der Dynamic Parallel Checker identifiziert die Objekte, auf die ein Zugriff stattgefunden hat, mit Hilfe deren Adressen. D.h. sobald der Garbage Collector den Arbeitsspeicher reorganisiert hat, haben die selben Objekte nicht mehr die gleiche Adresse und der Dynamic Parallel Checker funktioniert dadurch nicht mehr. Es gibt keine Möglichkeit die Änderungen des GCs in Erfahrung zu bringen und dadurch die gesamte History anzupassen. Sollte der GC eine Reorganisation durchführen, meldet die Dynamic Parallel Checker Library Fehlmeldungen (False-Positives).

3.2.3 Short Branches

Der MSIL Code verwendet für Befehle wie "try/catch/finally/if/else usw." Branches. Ein Branch ist ein Befehl im IL Code welcher eine Sprungmarke definiert. Dabei wird als Operand die Code Zeile zu welcher der Branch springen soll definiert. MSIL kennt 13 verschiedene Branches (sh. [ECMA, 2012] III.3.5 - III.3.18 ohne III.3.16), alle funktionieren nach dem selben Prinzip. Der Operand des Befehls beinhaltet die Code Zeile, zur welcher gesprungen werden soll.
Durch die Instrumentation für den Dynamic Parallel Checker verändern sich die Zeilennummern. Daher ist es wichtig, dass diese nachgeführt werden. Mono.Cecil unterstützt den Entwickler soweit, das über die direkte Referenz auf die Zielcodezeile der Wert nachgetragen wird. Jedoch besitzen alle Branches neben dem Standard OpCode, auch ein "*.s" OpCode. Zum Beispiel gibt es für "brtrue IL_3A303", den Code "brtrue.s IL_03", wobei "*.s" nur für Code Zeilennummern kleiner "IL_FF" verwendet werden kann. Durch die Instrumentation des Dynamic Parallel Checkers kann dieser Wert aber überschritten werden. Mono.Cecil hat keine automatische Erkennung solcher Branch-Fehler.

```
method.Body.SimplifyMacros();
/*--------------Code Instrumentation---------------*/
method.Body.OptimizeMacros();
```

Abbildung 16: Branch Optimierung durch Mono.Cecil

Die Methode SimplifyMacros in Abbildung 16 wird dazu verwendet, das Mono.Cecil alle "brtrue.s" in "brtrue" umwandelt. Dadurch kann die Zeilennummer auf die Grösse, die nötig ist, gesetzt werden. Der Befehl OptimizeMacros versucht dann nach der Instrumentation "brtrue" wieder auf "brtrue.s" zu wandeln. Dadurch wird der Code wenn möglich wieder optimiert.
Wenn der Code in Mono.Cecil eingelesen wird, werden alle Adressen in den Branches als Referenz auf die jeweilige Instanz der Ziel-Methode gehalten. Wenn die Methode gelöscht wird, ist die Referenz ungültig und Mono.Cecil zeigt auf eine Unbekannte Adresse. Dadurch wird der Code instabil. Daher sollte wenn möglich Code nicht aus der Instrumentation gelöscht werden, sondern der OpCode und Operand manuell ersetzt werden.

4 Backlog

4.1 Fehlende Synchronisationsoperationen

Der Umfang dieser Arbeit beschränkte sich aus zeitlichen Gründen auf einen gewissen Umfang. Die bereits instrumentierten Synchronisationspunkte befinden sich im Anhang. (sh. Liste instrumentierte mscorlib Methoden)
Folgende Synchronisationspunkte könnten bei einer Weiterentwicklung dieses Projekts noch instrumentiert werden:

- Thread.Abort()
- Thread.Interrupt()
- Task.Result()
- Task.WaitAll()
- Task.WaitAny()
- Task.ContinueWith()
- Task.WhenAll() / Task.WhenAny()
- Task.Delay()
- Task.Factory.ContinueWhenAny() / Task.Factory.ContinueWhenAll()
- Task.Factory.FromAsync()
- async await
- volatile Variablen
- Semaphoren
- Finalizer
- Parallel.Invoke() / Parallel.For() / Parallel.ForEach()

4.2 Funktionsumfang

Deadlock Erkennung

Die implementierte Funktionalität erlaubt es zur Zeit nur, Race Conditions zu erkennen. Von Vorteil wäre es, wenn zusätzlich noch Deadlocks erkannt werden. Dafür müsste aber der vorhandene Algorithmus noch erweitert werden. Vor jedem Lock auf eine Ressource müsste die Dynamic Parallel Checker Library informiert werden und eine eigene Liste geführt werden, welcher Thread welchen Lock bezogen hat. Sobald ein Thread den Lock freigibt, müsste dieser Eintrag wieder gelöscht werden. Dadurch könnte erkannt werden, dass zwei unterschiedliche Threads sich gegenseitig sperren.

Dependencies instrumentieren

In dieser Version wird eine spezifische Applikation instrumentiert. Sollte die Anwendung noch Abhängigkeiten zu weiteren, selbst entwickelten Libraries besitzen, werden Race Conditions, aus diesen Libraries, nicht erkannt. Dies, aufgrund der Tatsache, dass die Abhängigkeiten nicht instrumentiert werden. Diese zusätzliche Funktionalität, würde den Umfang des Überprüfungsbereichs erweitern. Dabei ist zu beachten, dass die Microsoft Core Library (mscorlib) nicht mit instrumentiert wird. Dies würde den Rahmen der Überwachung sprengen.

4.3 Vision

Unit Testing

Um die Erkennungsrate des Algorithmus erhöhen zu können, müssen Tools mehrmals den Test durchlaufen. Daher ist eine Vision, den Algorithmus in ein Concurrent-UnitTesting Framework einzubauen. Dieses erlaubt dem Entwickler das Erstellen von ConcurrentTests, welche nicht nur einen synchronen Test durchführen, sondern die Tests parallel zu einander ausführt um Race Conditions zu provozieren. Der Dynamic Parallel Checker Algortihmus überwacht die Unit Tests und meldet auftretende Race Conditions. Dadurch würden Tools vor dem Deployment und im Entwicklungszyklus jederzeit auf Race Conditions getestet. Dies sensibilisiert die Entwickler auf parallele Probleme und erhöht die Qualität der Software. Das Unit Testing Framework kann durch einen eigenen Scheduler in der Erkennung noch weiter verbessert werden. Durch ein gezieltes Verzahnen der Threads könnte provoziert werden, dass diese Race Conditions oder Deadlocks generieren

Glossar

Totale Ordnung	Eine totale Ordnung liegt vor, wenn sich alle Elemente in eindeutiger Weise sortieren lassen (wenn jedes Element kleiner gleich (<=) dem nächsten Element ist). z.B. die endliche Menge der natürlichen Zahlen.
Partielle Ordnung	Teilmengen könnten in eine Ordnung gebracht werden, jedoch die einzelnen Teilmengen können in der Gesamtmenge nicht mehr in eine Ordnung gebracht werden.
Thread	Ist in der Informatik ein Ausführungsstrang oder eine Ausführungsreihenfolge in der Abarbeitung eines Programms. Der Entwickler kann ein zusätzlicher Ausführungsstrang (Thread) erstellen und starten um Instruktionen quasi parallel zum Hauptprogramm auszuführen.
Task	Ein Task ist ein Paket aus Instruktionen, die quasi parallel von mehreren Worker Threads abgearbeitet werden.

Abbildungsverzeichnis

Literaturverzeichnis

[ACSC, 1988] "Timestamps in Message-Passing Systems That Preserve the Partial Ordering", http://zoo.cs.yale.edu/classes/cs426/2012/lab/bib/fidge88timestamps.pdf, 02.1988

[ECMA, 2012] "Standard ECMA-335 Common Language Infrastructure (CLI)", http://www.ecma-international.org/publications/standards/Ecma-335.htm, 06.2012

[MCA, 1978] "Time, Clocks, and the Ordering of Events in a Distributed System", http://research.microsoft.com/en-us/um/people/lamport/pubs/time-clocks.pdf, 07.1978

[MONO, 2015] "Mono.Cecil Library", http://www.mono-project.com/docs/tools+libraries/libraries/Mono.Cecil/, 12.2015

[MSDN, 2015] "MSDN - Memory Management and Garbage Collection", https://msdn.microsoft.com/de-de/library/0xy59wtx(v=vs.110).aspx, 12.2015

Anhang

Persönliches Fazit

Fabian Keller

Die parallele Programmierung ist ein immer aktuelleres Thema in der Praxis. Daher denke ich, dass ich mit der Thematik dieser Studienarbeit für das spätere Berufsleben sehr viel gewinnen konnte. Das Thema hat mich seit Beginn sehr interessiert. Auch wenn die Thematik eine gewisse Komplexität mit sich brachte, konnten wir uns sehr schnell in dem Thema zurecht finden und Fortschritte erzielen. Besonders von der Einsicht in die tieferen Schichten der Programmiersprache und das genaue Analysieren der Microsoft Core Library (mscorlib) konnte ich persönlich sehr profitieren. Zudem war das Entwickeln eines neuen Algorithmus für die Erkennung von Race Conditions eine Herausforderung, die mir sehr gefiel. Die Zusammenarbeit im Team funktionierte hervorragend. Beide Teammitglieder haben sich gegenseitig ergänzt und dadurch ein sehr gut funktionierendes Team gebildet.

Dominik Heeb

Die Arbeit "A study of dynamic parallel checking methods" war für mich eine sehr spannende und lehrreiche Arbeit. Fabian und ich hatten zu Beginn nur wenige Kenntnisse über das Thema "Dynamic checking". Das Thema Concurrency hat mich persönlich sehr interessiert, da ich es schon mehrmals verwendet, aber noch nie tiefer angeschaut habe. Durch dieses Projekt hatten wir die Möglichkeit einen ganz anderen Blickwinkel über die Programmierung zu erhalten. Da wir früh schon eine grundsätzliche Idee für den Algorithmus hatten, war die Entwicklung effizienter von statten gegangen als wir uns gedacht hatten. Ich fand sehr gut, dass wir nicht einfach einen Algorithmus aus einem Paper implementiert haben, sondern durch diese Idee einen Algorithmus selbst entwickeln konnten. Die Arbeit im Team war sehr angenehm, da wir beide mit dem selben Enthusiasmus herangingen.

Liste instrumentierte mscorlib Methoden

Bereits instrumentierte Methoden:

- Thread.Start()
- Thread.Join()
- Task.Start()
- Task.Wait()
- Task.Run()
- Task.Factory.StartNew()

Liste instrumentierte MSIL Operationen

Bereits instrumentierte MSIL Operationen:

- ldsfld (Load static field)
- stsfld (Store static field)
- ldfld (Load field)
- stfld (Store field)
- ldelem (Load element of array)
- stelem (Store element of array)
- initobj (Init object)
- call (Call method)
- callvirt (Call virtual method)

Mehr Informationen zu den einzelnen Operation Codes: siehe [ECMA, 2012].