Speichermodelle

Eine Ausarbeitung im Rahmen des KvBK-Seminars

Kenan Gündogan
Friedrich-Alexander-Universität Erlangen-Nürnberg
kenan.guendogan@fau.de

1. ÜBERSICHT

Diese Ausarbeitung erläutert die Kernprobleme, die bei der nebenläufigen Programmierung anhand gemeinsam genutzter Variablen entstehen können. Anschließend werden die im C11/C++11 Standard neu eingeführten Speichermodelle vorgestellt und ihre essentielle Bedeutung für die parallele Programmierung anhand von Fallbeispielen aufgezeigt. Angesichts der Tatsache, dass eine weitere Steigerung der *Performance* derzeitiger Anwendungen im Grunde nur durch eine Parallelisierung der Software erreicht werden kann, ist eine klare Spezifikation der Sichtbarkeitsregeln für gemeinsam genutzte Variablen zwischen verschiedenen Aktivitätsfäden durch die Programmiersprache essentiell. Eine solche Spezifikation erleichtert Korrektheitsüberlegungen für den Programmierer, bietet dem Rechensystem einen definierten Rahmen für Optimierungen, und verbessert die Portabilitätsaspekte nebenläufiger Programme.

2. EINLEITUNG

Im Allgemeinen werden Aktivitätsfäden *(Threads)* und gemeinsam genutzte Datenstrukturen *(Shared Variables)* zur Realisierung von Nebenläufigkeit in Programmen herangezogen [1]. Dies ist vor allem der Tatsache geschuldet, dass Aktivitätsfäden schon vor dem Aufkommen von Viel- und Mehrkernarchitekturen gut von den meisten Betriebssystemen unterstützt wurden. Zudem wird gemeinsam genutzter Speicher *(Shared Memory)* direkt von der Hardware zur Verfügung gestellt, sodass die Benutzung von geteilten Datenstrukturen schnell und effizient ist. Die Verwendung von Referenzen auf große gemeinsam genutzte Datenstrukturen erleichtert die Parallelisierung von bereits existierenden sequentiellen Programmen, weswegen keine komplette Neugestaltung der Software vorgenommen werden muss. Diese Arbeit stellt das Speichermodell *(Memory Model)* als das Fundament der nebenläufigen Programmierung für geteilten Speicher nutzende Systeme vor, denn das Speichermodell ist die Antwort auf die scheinbar simple Frage: Welcher Aktivitätsfaden sieht wann welchen geschriebenen Wert? [1]

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

3. PROBLEMSTELLUNG

3.1 Datenwettlauf - Definition *(Data Race)*

Eine Wettlaufsituation *(Race Condition)* ist im C11/C++11 Standard von einem Datenwettlauf *(Data Race)* zu unterscheiden [12]. Ein Datenwettlauf tritt demnach genau dann ein, wenn folgende zwei Bedingungen erfüllt sind [16]:

1. Mehrere nebenläufige Aktivitätsfäden greifen gleichzeitig auf dieselbe gemeinsam genutzte Variable zu, wobei mindestens ein Aktivitätsfaden schreibend zugreift.
2. Die Aktivitätsfäden verwenden dabei keinerlei Mechanismen zur Serialisierung dieser Speicherzugriffe.

Diese Definition wird nachfolgend für das Verständnis des sog. *Data-Race-Free* Speichermodells essentiell sein [3].

3.2 Manipulierung der Anweisungsreihenfolge

Ein Programm ist eine festgelegte Folge von Instruktionen für einen Prozessor [4]. Einzelne Instruktionen können jedoch zur Steigerung der *Performance* sowohl statisch durch die Software, als auch dynamisch durch die Hardware, oder durch etwaige Laufzeitinterpreter umgeordnet werden, solange dabei die programminternen Datenabhängigkeiten nicht verletzt werden [2]. Die Umsortierung der Instruktionen findet Software-seitig durch abstrakte Maschinen wie den Kompilierer, den Assemblierer, den Binder als auch durch potentielle Laufzeitinterpreter wie bspw. die *Java Virtual Machine* statt. Auf der Seite der Hardware nimmt der Prozessor die Umsortierungen vor. Dabei werden zum Beispiel folgende Optimierungstechniken angewendet:

- *Out-of-order Execution:* Der Prozessor (oder auch der Kompilierer) ordnet Instruktionen um.
- Spekulative Ausführung von Bedingungen: Der Prozessor führt spekulativ Instruktionen aus, die hinterher vielleicht wieder revidiert werden müssen.
- Lokale Puffer *(L1-Cache, Store Buffer)*: Verbergung der Latenzen des langsameren Hauptspeichers.
- *Register Promotion*: Häufig genutzte Variablen werden durch den Kompilierer in Registern vorgehalten, um Hauptspeicherzugriffe zu minimieren.

Anhand der Optimierungstechniken wird ersichtlich, dass das System die vom Programm vorgegebene Instruktionsreihenfolge nicht einhalten muss, wenn es dadurch den Prozessor besser auslasten kann.

Listing 1: Signalübermittlung

```
1  #include <stdatomic.h>
2  static atomic_int a = 0, b = 0; //Gem. gen. Var.
3  static int        c = -1;       //Lokal gen. Var.
4
5  static void *thread1(void *p) {
6   //Atomar a = 5 setzen
7   atomic_store_explicit(&a,5,memory_order_relaxed);
8
9   //Atomar b = 1 setzen
10  atomic_store_explicit(&b,1,memory_order_relaxed);
11  return NULL;
12 }
13
14 static void *thread2(void *p) {
15 //Lade b atomar und pruefe, ob b bereits gesetzt
16 if(atomic_load_explicit(&b,memory_order_relaxed)){
17   c=atomic_load_explicit(&a,memory_order_relaxed);
18   assert(c == 5); //Kann schiefgehen
19  }
20  return NULL;
21 }
```

Nebenläufiges C-Programm mit zwei Aktivitätsfäden. Über `b` *signalisiert Faden 1, dass er* `a = 5` *gesetzt hat, d.h. Faden 2 übernimmt nur unter der Bedingung* `b == 1` *den Wert von* `a`*.*

Solche Optimierungen zielen auf eine Verbesserung der nichtfunktionalen Eigenschaften des Programms ab, insbesondere die der *Performance*. Solange eine sequentielle Programmausführung stattfindet, werden die funktionalen Attribute des Programms aufgrund der programminternen Datenabhängigkeiten nicht verändert, d.h. ein sequentielles Programm kann nie etwas von seinen eigenen Umsortierungen erfahren. Das trifft auch auf einen einzelnen Aktivitätsfaden zu, der an und für sich rein sequentiell abgearbeitet wird.

Nebenläufige Programme bestehen aus mehreren sequentiellen Aktivitätsfäden, die aber im Verbund parallel ausgeführt werden. Wenn die Aktivitätsfäden untereinander über gemeinsam genutzte Variablen *(Shared Variables)* interagieren, dann können die Optimierungen, d.h. Instruktionsvertauschungen eines Aktivitätsfadens als Seiteneffekt *(visible side effect)* in einem anderen Aktivitätsfaden auftreten und zu sog. Sichtbarkeitsanomalien führen.

3.3 Sichtbarkeitsanomalien

Eine Sichtbarkeitsanomalie kann nur dann auftreten, wenn mehrere Aktivitätsfäden auf gemeinsam genutzten Variablen arbeiten und min. einer der Speicherzugriffe eine Schreiboperation darstellt. Ob und wie solche Seiteneffekte auftreten, lässt sich nicht vorhersagen, da dies sowohl von der zeitlichen Verzahnung der Aktivitätsfäden auf der Befehlssatzebene als auch von den tatsächlich vom Rechensystem vorgenommenen Optimierungen abhängt.

Listing 1 zeigt ein nebenläufiges C-Programm, das eine simple Signalübermittlung *(Message Passing)* von Aktivitätsfaden 1 zu Aktivitätsfaden 2 implementiert, wobei `a` und `b` gemeinsam genutzte Variablen sind. Die Signalvariable `b` aus Faden 1 dient dazu, dem Faden 2 mitzuteilen, ob `a` bereits geschrieben worden ist. Faden 2 hingegen überprüft parallel, ob `b` gesetzt wurde und falls dem so ist, führt Faden 2 die Anweisung `a = c` aus. Sämtliche Möglichkeiten der Instruktionsumordnung lassen sich prinzipiell auf vier Fälle abbilden und werden graphisch in *Tabelle 1* dargestellt.

Tabelle 1: Mögliche Instruktionsumsortierungen

Fall 1		Fall 2	
	`if(b == 1)` `return NULL`	`a = 5` `b = 1`	
`a = 5 | b = 1` `b = 1 | a = 5`			`if(b == 1)` `c = a` *(a = 5)*
Fall 3		**Fall 4**	
`b = 1` `a = 5`		`b = 1`	
	`if(b == 1)` `c = a` *(a = 5)*		`if(b == 1)` `c = a` *(a = 0)*
		`a = 5`	

Der erste Fall ist irrelevant, da der Aktivitätsfaden 1 für diese Betrachtung zu spät ausgeführt wird. Der zweite Fall beschreibt die Situation, in der keine Instruktionsumordnung stattfindet, d.h. das Programm wird getreu dem Quelltext ausgeführt und die Signalübermittlung verläuft erfolgreich. Der dritte Fall verdeutlicht, dass aufgrund der zeitlichen Verzahnung eine Instruktionsvertauschung im Aktivitätsfaden 1 nicht zwangsläufig als Seiteneffekt im Aktivitätsfaden 2 auftreten muss. Trotz der Vertauschung der Variablen `a` und `b` erfolgt eine korrekte Signalübermittlung.

Der vierte Fall kann nur deshalb auftreten, weil die atomaren Schreiboperationen in den Zeilen 7 und 10 aufgrund der Markierung mit `memory_order_relaxed` vertauscht werden dürfen. Dabei führt die Vertauschung zu einem fehlerhaften Programmzustand, da die Umsortierung der Schreiboperationen in den Zeilen 7 und 10 des Aktivitätsfadens 1 als sichtbarer Seiteneffekt im Aktivitätsfaden 2 auftritt. Die Signalübermittlung ist dadurch inkorrekt, da der Aktivitätsfaden 2 die Instruktion `a = c` ausführt, bevor der Aktivitätsfaden 1 `c = 5` gesetzt hat, wodurch `a` fälschlicherweise mit `0` statt mit `5` belegt wird.

Die Manipulierung der Instruktionsfolge verursacht im Fall vier einen Programmfehler. Dadurch verdeutlicht *Listing 1* die Problematik, die durch Instruktionsumsortierungen enstehen können. Die Instruktionsfolge eines Programms wird von nun an als Speicherzugriffsfolge betrachtet und besteht damit aus Schreib- und Leseoperationen, die vom Programmierer über den Quelltext festgelegt wird. Wie zuvor beschrieben, kann eine Modifikation der Speicherzugriffsfolge in einem Aktivitätsfaden als Seiteneffekt in einem anderen Aktivitätsfaden auftreten, und zu einem Programmfehler führen. Klare Spezifikationen erlaubter Modifikationen an der Speicherzugriffsfolge und deren korrekte Verwendung lösen die Problemstellung. Solche Spezifikationen werden als Speichermodelle bezeichnet.

4. SPEICHERMODELLE

4.1 Speichermodelle als Schichtenübergreifende Schnittstelle

(Memory Models)

Ein Speichermodell dient als Schnittstelle zwischen dem Programmierer und dem ausführenden Rechensystem, indem es die zulässigen Modifikationen an der Speicherzugriffsfolge definiert [14]. Ein Rechensystem besteht aus einer Schichtenfolge von (abstrakten) Maschinen [5], wobei jeder dieser Schichten aus Optimierungsgründen Modifizierungen an der Speicherzugriffsfolge vornehmen kann [14].

Nach Adve und Gharachorloo betrifft die problemorientierte Programmiersprache als oberste Schicht den Programmierer *(high level language code)* sowie den Kompilierer [14]. Der Programmierer kann durch das Speichermodell der Programmiersprache unabhängig von der Rechnerarchitektur hinsichtlich der Funktionalität und der Korrektheit seines Quelltextes reflektieren [14]. Die Programmiersprache abstrahiert also für den Programmierer die Details der darunterliegenden Schichten bzgl. der Instruktionsumordnungen weg und kann dadurch ein von der Hardware unabhängiges Speichermodell garantieren. Die unteren Schichten der Maschinenbefehlsebene bis hin zum Prozessor betreffen die Hardware-Architekten und diejenigen Programmierer *(low level language code)*, die Kompilierer konstruieren oder sich mit Maschinenkode auseinandersetzen, d.h. das Speichermodelle verschiedene Bereiche der Informatik übergreifend beeinflussen [14].

Speichermodelle der Hardware könnnen sich durchaus von denen der Kompilierer unterscheiden *(hardware-software mismatch)* [1], was jedoch dank der Abstraktion durch die Programmiersprache für Funktionalitäts- und Korrektheitsüberlegungen ignoriert werden kann. Die Hardware-Speichermodelle werden in dieser Arbeit jedoch ausgespart, da der Fokus auf den Speichermodellen von Programmiersprachen liegt.

4.2 Zielsetzung von Speichermodellen

Ein Speichermodell erfüllt hierbei folgende drei Funktionalitäten:

1. Performance:
Ein Speichermodell hat einen großen Einfluss auf die *Performance* von Programmen, denn es gilt: Je stärker ein Speichermodell die Manipulationen der Speicherzugriffsfolge einschränkt, desto weniger Optimierungen kann der Kompilierer bzw. die Hardware vornehmen, was sich negativ auf die *Performance* auswirkt [14].

2. Programmierbarkeit:
Anhand des Speichermodells ist der Programmierer in der Lage, Funktionalität und vor allem Korrektheit seiner Programme sicherzustellen [14].

3. Portabilität:
Wegen uneinheitlicher Umsortierung der Speicherzugriffsfolge verschiedener Hardware [13] ist der Anwendungsprogrammierer darauf angewiesen, dass die Programmiersprache, in der das Programm verfasst ist, bestimmte Speichermodelle garantiert, um anhand derer die Korrektheit sicherstellen zu können.

Damit ist ein Speichermodell unabdingbar für Rechensysteme, die eine nebenläufige Ausführung auf gemeinsam genutzten Speicher *(Shared Memory Systems)* ermöglichen [14].

4.3 Konsistenzen von Speichermodellen
(Memory Consistency Models)

Speichermodelle unterscheiden sich hauptsächlich in ihrer Stärke. Schwächere Speichermodelle erlauben mehr Umsortierungen der Speicherzugriffsfolge als stärkere Speichermodelle, woraus sich gewisse Vor- und Nachteile ergeben. Der Vorteil schwacher Speichermodelle liegt in der besseren *Performance* von Programmen, wohingegen der Vorteil starker Speichermodelle in den leichteren Korrektheitsüberlegungen liegt. Der tatsächlich relevante Grund für starke Speichermodelle ist jedoch daduch bedingt, dass manche nebenläufige Algorithmen ein starkes Speichermodell für einen korrekten Ablauf benötigen [1]. Daraus folgt, dass für einen parallel ausgeführten Algorithmus das verwendete Speichermodell zur Steigerung der *Performance* so schwach wie möglich, aber zur Korrektheitserfüllung so stark wie nötig ausgewählt werden sollte.

In den folgenden Unterkapiteln werden nun die im C11/ C++11 Standard eingeführten Speichermodelle anhand von C-Programmen erläutert. Zudem stellt dieser Sprachstandard atomare Operationen zur Verfügung, die über die Bibliothek `stdatomic.h` importiert werden können. Diese Bibliotheksfunktionen können mit den unten aufgeführten Konstanten aufgerufen werden.

4.3.1 Relaxed Consistency

Die *Relaxed Consistency* ist das schwächste Speichermodell, da es lediglich die Atomarität von Instruktionen zur Vermeidung von Datenwettläufen garantiert *(s. 3.1 Datenwettlauf)*, aber nicht die Speicherzugriffsmanipulationen beschränkt [6]. Mit der Konstante `memory_order_relaxed` markierte Operationen werden gemäß diesem Speichermodell behandelt, d.h. derartige Operationen sind dadurch nur atomar, dürfen aber vom System beliebig umsortiert werden, soweit es die Datenabhängigkeiten zulassen. In *Listing 1* wurden die Operationen mit `memory_order_relaxed` markiert, weshalb es zur Vertauschung der beiden atomaren Operationen *(Zeile 7, 10)* im Aktivitätsfaden 1 kommen kann. Im Aktivitätsfaden 2 sind die atomaren Operationen zwar auch derart markiert, jedoch können diese Anweisungen aufgrund ihrer bedingten Abhängigkeit *(s. Listing 1: Zeile 16-17)* nicht umsortiert werden.

Ein typischer Anwendungsfall ist die Implementierung einer Zählervariablen, die von mehreren Aktivitätsfäden inkrementiert wird, da hierbei lediglich die Atomartität der Inkrementierung zu gewährleisten ist [6].

4.3.2 Release-Acquire Consistency

Die *Release-Acquire Consistency* ist stärker als die *Relaxed Consistency*. Bei diesem Speichermodell hängen die Umsortierungsmöglichkeiten davon ab, ob der aktuell betrachtete Speicherzugriff eine Schreib- oder Leseoperation ist. Die Besonderheit dieses Speichermodells liegt darin, dass sie drei Konstanten zur Markierung von atomaren Operationen zur Verfügung stellt [7]:

1. `memory_order_acquire` *(Erwerbende Semantik)*
Diese Konstante kann lediglich auf atomare Funktionen angewendet werden [10], die einen lesenden Zugriff auf eine gemeinsam genutzte Variable enthalten *(s. Listing 2: Zeile 10, 19; Listing 3: Zeile 7, 16)*, also reine Ladeoperationen bzw. Lesen-Ändern-Schreibende Operationen. Eine akquirierende Operation bildet eine Umsortierungsgrenze, da alle im momentan betrachteten Aktivitätsfaden dahinter stehenden Schreib- und Lesezugriffe nicht vor diese akquirierende Ladeoperation vorgezogen werden dürfen. Infolgedessen wird für den akquirierenden *(acquire)* Aktivitätsfaden stets der global aktuelle Wert der gemeinsam genutzten angefragten Variable garantiert [9].

Listing 2: dekker.c

```
1  #include <stdatomic.h>
2  static atomic_int a  = 0, b  = 0;//Global gen.Var.
3  static int        r1 = 0, r2 = 0;//Lokal gen.Var.
4
5  static void *thread1(void *param) {
6   //Atomar a = 1; Freigebend: Sperrt abwaerts
7   atomic_store_explicit(&a,1,memory_order_release);
8
9   //Atomar b laden; Erwerbend: Sperrt aufwaerts
10  r1=atomic_load_explicit(&b,memory_order_acquire);
11  return NULL;
12 }
13
14 static void *thread2(void *param) {
15  //Atomar b = 1; Freigebend: Sperrt abwaerts
16  atomic_store_explicit(&b,1,memory_order_release);
17
18  //Atomar a laden; Erwerbend: Sperrt aufwaerts
19  r2=atomic_load_explicit(&a,memory_order_acquire);
20  return NULL;
21 }
```

Prinzipiell geschieht hier bei Faden 1: `{a=1; r1=b;}` *und bei Faden 2:* `{b=1; r2=a;}`. ***Sperre abwärts:*** *Alle Speicherzugriffe, die sich oberhalb der Release-Schreiboperation befinden, können nicht unter diese Release-Schreiboperation verschoben werden.* ***Sperre aufwärts:*** *Alle Speicherzugriffe, die sich unterhalb der akquirierenden Ladeoperation befinden, können nicht über diese akquirierende Ladeoperation verschoben werden.*

Tabelle 2: Mögliche Resultate von dekker.c

Fall 1		Fall 2		Fall 3		Fall 4	
a=1 r1=b	b=1 r2=a	a=1 r1=b	b=1 r2=a	a=1 r1=b	b=1 r2=a	r1=b a=1	r2=a b=1
r1=1	r2=0	r1=0	r2=1	r1=1	r2=1	r1=0	r2=0

Die unterste Zeile gibt pro Fall jeweils an, welche Werte die Variablen `r1` *und* `r2` *bei Programmende aufweisen.*

2. `memory_order_release` *(Freigebende Semantik)*

Diese Konstante kann lediglich auf atomare Funktionen angewendet werden [10], die einen schreibenden Zugriff auf eine gemeinsam genutzte Variable enthalten *(s. Listing 2: Zeile 7, 16; Listing 3: Zeile 10, 19)*, also sowohl reine Schreiboperationen als auch Lesen-Ändern-Schreibende Operationen [9]. Eine freigebende Operation bildet eine Umsortierungsgrenze, da alle im aktuellen Aktivitätsfaden zuvor aufgeführten Schreib- und Lesezugriffe nicht hinter diese freigebende Schreiboperation versetzt werden dürfen. Ein Aktivitätsfaden, der eine Variable freigibt *(release)*, macht all seine getätigten Speicherzugriffe (bis zur Freigabe) für all die Aktivitätsfäden sichtbar, die diese Variable chronologisch später erwerben *(acquire)* [9]. Die *Release*-Schreiboperation etabliert dadurch eine *happens-before*-Beziehung zwischen dem *Release*-Aktivitätsfaden und all denjenigen Aktivitätsfäden, welche die vom *Release*-Aktivitätsfaden beschriebene Variable chronologisch später akquirieren. Damit treten alle bereits ausgeführten Speicheroperationen des *Release*-Aktivitätsfadens als sichtbarer Seiteneffekt bei den akquirierenden Aktivitätsfäden auf. D.h. wenn Aktivitätsfaden A eine Variable `z` freigibt *(release)* und Aktivitätsfaden B die Variable `z` hinterher akquiriert, dann gilt die *happens-before*-Beziehung zwischen A und B.

Listing 3: rekked.c *(Umgedrehter Dekker)*

```
1  #include <stdatomic.h>
2  static atomic_int a  = 0, b  = 0;//Gem. gen. Var.
3  static int        r1 = 0, r2 = 0;//Lokal gen. Var.
4
5  static void *thread1(void *param) {
6   //Atomar b laden; Erwerbend: Sperrt aufwaerts
7   r1=atomic_load_explicit(&b,memory_order_acquire);
8
9   //Atomar a = 1; Freigebend: Sperrt abwaerts
10  atomic_store_explicit(&a,1,memory_order_release);
11  return NULL;
12 }
13
14 static void *thread2(void *param) {
15  //Atomar a laden; Erwerbend: Sperrt aufwaerts
16  r2=atomic_load_explicit(&a,memory_order_acquire);
17
18  //Atomar b = 1; Freigebend: Sperrt abwaerts
19  atomic_store_explicit(&b,1,memory_order_release);
20  return NULL;
21 }
```

Prinzipiell geschieht hier bei Faden 1: `{r1=b; a=1;}` *und bei Faden 2:* `{r2=a; b=1;}`*, abgesehen davon ist rekked.c identisch zu dekker.c. Dabei dienen Listing 2 und 3 zur Darstellung, welche Umsortierungen von Speicherzugriffen unter dem Release-Acquire Speichermodell gültig sind und wie sich die Umsortierungen konkret auf die Variablen* `r1` *und* `r2` *auswirken.*

Tabelle 3: Mögliche Resultate von rekked.c

Fall 1		Fall 2		Fall 3	
r1 = b a = 1	r2 = a b = 1	r1 = b a = 1	r2 = a b = 1	r1 = b a = 1	r2 = a b = 1
r1 = 0	r2 = 1	r1 = 1	r2 = 0	r1 = 0	r2 = 0

Die unterste Zeile gibt pro Fall jeweils an, welche Werte die Variablen `r1` *und* `r2` *bei Programmende aufweisen.*

3. `memory_order_acq_rel` *(Erwerben, dann freigeben)*

Diese Konstante kann lediglich auf atomare Lesen-Ändern-Schreibende Funktionen *(read-modify-write operations)* angewendet werden [10]. Eine erwerbend-freigebende Operation bildet eine Umsortierungsgrenze, da sämtliche Schreib- und Lesezugriffe im betreffenden Aktivitätsfaden weder vor noch hinter diese Operation positioniert werden dürfen. Es handelt sich hierbei um eine Kombination der ersten beiden Konstanten mit entsprechender Semantik [9].

Nun wird anhand zweier C-Programme *(s. Listing 2, 3)* das *Release-Acquire* Speichermodell praktisch erörtert. Es handelt sich dabei folgend um den sog. Dekker-Algorithmus und um dessen programmtechnisches Gegenteil [18]:

Dekker-Algorithmus *(s. Listing 2 sowie Tabelle 2)*

Die Fälle 1 bis 3 stellen die erwarteten Resultate dar, da hier die Belegungen der Variablen `r1` und `r2` lediglich von der zeitlichen Verzahnung der Aktivitätsfäden abhängt. Beim vierten Fall handelt es sich jedoch um eine besondere Situation, da hierbei die stattfindenden Speicherzugriffsmanipulationen in Form des Resultats `r1 = r2 = 0` sichtbar werden. Die *Release*-Schreiboperation *(Zeile 7 bzw. 16)* verhindert zwar, dass oberhalb von ihr liegende Speicherzugriffe nach unten hin verschoben werden können, doch die akquirierende Ladeoperation *(Zeile 10 bzw. 19)* sperrt nicht von oben nach unten (sondern von unten nach oben), weshalb die Ladeoperation *(Zeile 10 bzw. 19)* über die Schreibope-

ration hinweg *(Zeile 7 bzw. 16)* nach oben verschoben werden kann. Wie in der Tabelle 2 dargestellt, tritt der vierte Fall `r1 = r2 = 0` genau dann ein, wenn beide Aktivitätsfäden jeweils sowohl ihre Ladeoperation *(Zeile 10 bzw. 19)* vorziehen als auch den initialen Wert `0` der Variablen `a` bzw. `b` laden.

Rekked-Algorithmus *(s. Listing 3 sowie Tabelle 3)*
Beim verkehrten Dekker-Algorithmus (mnemonisch *Rekked* genannt) tritt jedoch der umgedrehte vierte Fall `r1 = r2 = 1` von Dekker nicht auf. Die nach unten hin sperrende *Release*-Schreiboperation *(Zeile 10 bzw. 19)* befindet sich an unterster Stelle, sodass darüberliegende Speicherzugriffe nicht nach unten hin verschoben werden können, d.h. die akquirierende Ladeoperation *(Zeile 7 bzw. 16)* kann nicht unter die *Release*-Schreiboperation versetzt werden. Zudem kann die *Release*-Schreiboperation *(Zeile 10 bzw. 19)* nicht über die akquirierende Ladeoperation verschoben werden, da die akquirierende Ladeoperation *(Zeile 7 bzw. 16)* nach oben hin eine Sperre bildet. Infolgedessen kann überhaupt keine Umsortierung der Anweisungen stattfinden und das Resultat des Rekked-Algorithmus hängt ausschließlich von der zeitlichen Verzahnung der beiden Aktivitätsfäden ab.

4.3.3 Sequentielle Konsistenz *(Sequential Consistency)*

Die sequentielle Konsistenz ist allgemein das stärkst mögliche Speichermodell, da sie jegliche Umsortierung der Speicherzugriffsfolge eines Aktivitätsfadens unterbindet. Ein nebenläufiges Programm ist dann sequentiell konsistent, wenn sich bei zeitlich beliebiger Verzahnung der Operationen aller Aktivitätsfäden stets die Instruktionen eines einzelnen Aktivitätsfadens in sequentieller, d.h. in der vom Programm festgelegten Reihenfolge erscheint [15]. Wenn das Resultat eines sequentiell konsistenten nebenläufigen Programms bei mehrfacher Ausführung variiert, dann ausschließlich aufgrund der unterschiedlichen zeitlichen Verzahnung der Aktivitätsfäden. Der Vorteil sequentieller Konsistenz liegt zwar in den einfacheren Korrektheitsüberlegungen, da keinerlei Umsortierungen der Speicherzugriffe stattfinden kann, jedoch überwiegen die Nachteile der aufwändigen Realisierung. Zur Gewährleistung sequentieller Konsistenz müssen viele Optimierungen unterbunden werden, worunter die Ausführungsgeschwindigkeit leidet [1].

Um bei Dekkers Algorithmus *(s. Listing 2)* das ungewollte Ergebnis `r1 = r2 = 0` zu verhindern, muss die Vertauschung der Schreiboperation mit der Ladeoperation in den Zeilen 7 und 10 bzw. 16 und 19 unterbunden werden, indem alle atomaren Funktionen *(Zeile 7, 10, 16, 19)* mit der Konstanten `memory_order_scq_cst` aufgerufen werden [8]. Alternativ kann auch das Anhängsel `_explicit` weggelassen werden, z.B.: `atomic_store(&a, 1)`. Aus Tabelle 4 geht hervor, dass sich die Resultate des Dekker-Algorithmus bei sequentiell konsistenter Ausführung nur aufgrund der unterschiedlichen zeitlichen Verzahnung der Anweisungen der Aktivitätsfäden unterscheiden. Das sequentiell konsistente Speichermodell vermeidet bei Dekkers Algorithmus den ungewollten Fall `r1 = r2 = 0`. Beim Rekked-Algorithmus jedoch genügt bereits das schwächere *Release-Acquire* Speichermodell zur Vermeidung des ungewollten Falls `r1 = r2 = 1`. Anhand des Rekked-Algorithmus geht hervor, dass manchmal auch schwächere Speichermodelle ungewollte Fälle vermeiden können. In C/C++ gilt die sequentielle Konsistenz nur für diejenigen atomaren Funktionen, die mit der Konstanten `memory_order_seq_cst` versehen sind. Atomare Funktionen mit der Konstante `memory_order_seq_cst` bilden eine Sperre, die weder von oben nach unten, noch von unten nach oben Umsortierungen von Speicherzugriffen zulässt. Das Rechensystem kann jedoch Instruktionen, die sich zwi-schen zwei `memory_order_seq_cst`-Funktionen befinden, innerhalb dieser Sperren umsortieren.

Tabelle 4: Dekker - Sequentiell konsistent

Fall 1		Fall 2		Fall 3	
`a = 1` `r1 = b`			`b = 1` `r2 = a`	`a = 1` `r1 = b`	`b = 1` `r2 = a`
	`b = 1` `r2 = a`	`a = 1` `r1 = b`			
`r1 = 0`	`r2 = 1`	`r1 = 1`	`r2 = 0`	`r1 = 1`	`r2 = 1`

Mögliche Resultate des Dekker-Algorithmus bei sequentiell konsistenter Ausführung, d.h. alle atomaren Funktionen in Listing 2 werden hierfür mit der Konstanten `memory_order_seq_cst` aufgerufen. Die unterste Zeile gibt pro Fall jeweils an, welche Werte die Variablen `r1` und `r2` bei Programmende aufweisen. Die Variablenwerte von `r1` und `r2` unterscheiden sich von Fall zu Fall nur aufgrund der unterschiedlichen zeitlichen Verzahnung der Aktivitätsfäden.

4.4 Data-Race-Free Speichermodell bei C/C++

Bei C/C++ gilt das *Data-Race-Free* Speichermodell, d.h. dass die zuvor erläuterten Speichermodelle lediglich für Datenwettlauffreie *(s. 3.1 Datenwettlauf - Definition)* Programme gelten [1]. Im Umkehrschluss bedeutet dies, dass ein Programm mit einem Datenwettlauf undefiniertes Verhalten *(undefined behavior)* aufweist, über dessen Laufzeitverhalten keine Aussagen mehr getroffen werden können. Im Falle eines Datenwettlaufs ist das Resultat eines Programms beliebig, da Datenwettläufe enthaltende Programme keine gültigen C/C++-Programme mehr darstellen. Dadurch fällt dem Programmierer die nicht triviale Aufgabe zu, Datenwettläufe jeglicher Art zu erkennen und zu vermeiden, da das Programm ansonsten undefiniertes Verhalten zu beliebiger Zeit während der Laufzeit aufzeigen kann [1].

4.5 Differenzen zwischen C/C++ und Java

Das Hauptaugenmerk von C/C++ liegt auf der *Performance* von Programmen, wohingegen für Java die Typ- und Ausführungssicherheit im Vordergrund steht. Deshalb ist das *Data-Race-Free* Speichermodell zwar adäquat für eine *Performance*-orientierte Sprache wie C/C++, aber unzureichend für sichere Sprachen wie Java [1]. Das Speichermodellkonzept von Java sieht im Falle eines Datenwettlaufs *(data race)* die Begrenzung des daraus resultierenden nichtdeterministischen Verhaltens vor, d.h. es wird versucht, die möglichen Schäden durch solche Datenwettläufe zu begrenzen, um der Eigenschaft einer sicheren Sprache *(z.B. execution of untrusted code)* gerecht zu werden, was jedoch nicht vollends gelingt [11, 1]. Weder das *Data-Race-Free* Speichermodell von C/C++ noch das von Java erfüllt die Kriterien eines Speichermodellkonzepts, das realisierbar ist, abstrakt genug für *high-level* Programmiersprachen ist, und im Falle eines Datenwettlaufs nicht auf undefiniertes Verhalten ausweicht [17].

5. ZUSAMMENFASSUNG

Das *Data-Race-Free* Speichermodell ist das Speichermodellkonzept für geteilten Speicher nutzende parallele Software, das sich nach jahrzehnter langer Forschung herausgebildet hat [1]. Aktuelle Bestrebungen zielen darauf ab, Datenwettläufe zukünftig am Besten durch das System selbst und idealerweise auch durch die Programmiersprache zur Entlastung des Programmierers abzufangen. Zudem ist eine bessere Zusammenarbeit der Hard- und Softwarebereiche bzgl. der Speichermodelle naheliegend, sodass die Hard- und Software-Speichermodelle besser ineinander greifen können. Das Paradigma des gemeinsam genutzten Speichers ist hierbei nicht das Hauptproblem, sondern vielmehr die undisziplinierte Art der Nutzung des gemeinsamen Speichers. Zur Vereinfachung der parallelen Programmierung sind klare, möglichst einfache, gut skalierbare, und generell gültige Speichermodelle nötig, die sowohl *Performance*-kritische Programmiersprachen (z.B. C/C++) als auch sicherheitskritische Programmiersprachen (z.B. Java) bedienen können. Dafür müssen sowohl die Rechensysteme als auch die Programmiersprachen grundlegend überarbeitet werden [1].

6. REFERENCES

[1] S. V. Adve, H.-J. Boehm. *Memory Models: A Case for Rethinking Parallel Languages and Hardware.* August 2010.

[2] H.-J. Boehm. *Threads Cannot be Implemented as a Library.* November 2004.

[3] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro and T. Anderson. *Eraser: A Dynamic Data Race Detector for Multithreaded Programs.* September 1997.

[4] W. Schröder-Preikschat. *Systemprogrammierung. Grundlage von Betriebssystemen. Sachwortverzeichnis.* November 2016.

[5] W. Schröder-Preikschat. *Rechnerorganisation: Virtuelle Maschinen.* Mai 2016.

[6] http://en.cppreference.com/w/c/atomic/memory_order#Relaxed_ordering. Zugriff Dezember 2016.

[7] http://en.cppreference.com/w/c/atomic/memory_order#Release-Acquire_ordering. Zugriff Dezember 2016.

[8] http://en.cppreference.com/w/c/atomic/memory_order#Sequentially-consistent_ordering. Zugriff Dezember 2016.

[9] http://en.cppreference.com/w/c/atomic/memory_order#Constants. Zugriff Dezember 2016.

[10] http://en.cppreference.com/w/c/atomic#Functions. Zugriff Dezember 2016.

[11] Java Language Specification. Chapter 17. Threads and Locks. http://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.4.5-410. Dezember 2016.

[12] ISO IEC 9899:201x N1570. Kapitel 5.1.2.4.

[13] Paul E. McKenney. *Is Parallel Programming Hard, And, If So, What Can You Do About It?.* Dezember 2011. Seite 171.

[14] S. V. Adve, K. Gharachorloo. *Shared Memory Consistency Models: A Tutorial.* September 1995.

[15] L. Lamport. *How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs.* September 1979.

[16] S. V. Adve, M. D. Hill. *Weak Ordering - A New Definition And Some Implications.* Dezember 1989.

[17] J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis, D. Dreyer. *A Promising Semantics for Relaxed-Memory Concurrency.* November 2016.

[18] E. W. Dijkstra. *Cooperating Sequential Processes.*