Schriftliche Ausarbeitung im Rahmen eines Seminars in der Lehrveranstaltung „verteilte und parallele Systeme“ an der Fachhochschule Bonn-Rhein-Sieg in Sankt Augustin zum Thema:

Relaxed Memory Consistency Models

Von Thomas Wild, 07.01.2003

Inhaltsverzeichnis Seite

Einführung

Unten aufgeführte Tabelle veranschaulicht ein paralleles Programm, das auf einem Parallelrechner mit zwei Prozessoren und gemeinsamen Speicher läuft. Prozessor 1 schreibt zwei Werte in die Speicherstellen „data1“ und „data2“. Prozessor 2 liest diese beiden Speicherstellen in zwei seiner Register. Die beiden Prozessoren synchronisieren sich über die Variable „flag“. Das Programm wurde in der Erwartung geschrieben, dass register1 und register2 die Werte enthalten, die von Prozessor 1 geschrieben wurden. Doch das muss nicht zwingend der Fall sein. Speicherzugriffe können z. B. von der Hardware gepuffert werden oder der Compiler hat die Zugriffe umsortiert, um den Programmablauf zu optimieren.
So kann es passieren, daß Prozessor 2 erst den schreibenden Zugriff von Prozessor 1 auf flag sieht und erst viel später die schreibenden Zugriffe auf data1 und data2. Prozessor 2 würde dann alten Werte von data1 und data2 in register1 und register2 schreiben.Wie das Programm nun tatsächlich abläuft hängt von dem Speicherkonsistenzmodell ab, das das Rechnersystem implementiert.

Prozessor 1	Prozessor 2
data1 = 64	while(flag != 1) {;}
data2 = 55	register1 = data1
flag = 1	register2 = data2

Das Speicherkonsistenzmodell von Einzelprozessorsystemen garantiert dem Programmierer, dass Speicheroperationen anscheinend nacheinander ausgeführt werden, in der Reihenfolge, die durch das Programm vorgegeben ist. Das heißt, dass eine Leseoperation den Wert zurückgibt, der zuletzt in diese Speicherposition geschrieben wurde. Einzelprozessorsysteme müssen die Speicheroperationen aber nicht zwingend nacheinander, in der Reihenfolge ausführen, die durch das Programm bestimmt ist. Hardware und Compiler können Speicheroperationen umsortieren oder sich überschneiden lassen. Solange dabei die Datenabhängigkeiten eingehalten werden, wird das Speicherkonsistenzmodell nicht verletzt.

Würde das Programm auf einem einzelnen Prozessor ausgeführt werden, dann könnte der Prozessor die Schreiboperationen auf data1 und data2 z. B. vertauschen oder erst data1 schreiben und dann register1. Das würde das Ergebnis der Ausführung nicht beeinflussen, da die Datenflussabhänigkeiten eingehalten würden.

Bei Multiprozessorsystemen wird es komplizierter. Das Modell des Einzelprozessorsystems lässt sich nicht einfach übertragen. Im Gegensatz zu Einzelprozessorsystemen reicht es nicht aus, nur Daten- und Kontrollflussabhängigkeiten für jeden einzelnen Prozessor einzuhalten, um Konsistenz zu gewährleisten. Im obigen Beispiel gibt es keine Datenabhängigkeiten unter den Operationen von Prozessor 1. Der Compiler oder die Hardware könnte die Ausführungsreihenfolge ändern und die Variable „flag“ bereits auf 1 setzen, bevor die Werte von „data1“ und „data2“ gesetzt werden. Prozessor 2 würde demzufolge nicht mehr darauf warten, dass Prozessor 1 die Werte von „data1“ und „data2“ setzt, sondern die alten Werte übernehmen.

Sequential Consistency
Das strengste Konsistenzmodell für Mehrprozessorsysteme mit gemeinsamen Speicher ist sequentielle Konsistenz. SC ist eine Erweiterung des Modells für Einzelprozessorsysteme. SC verlangt, dass Speicheroperationen nacheinander, in einer gewissen sequentiellen Reihenfolge zu erfolgen scheinen. Das heißt, dass die Reihenfolge, die jeder einzelne Prozessor sieht immer dieselbe ist. Die Operationen jedes einzelnen Prozessors sollen dabei in Programmreihenfolge erfolgen. Dieses Modell schränkt die möglichen Optimierungen durch Compiler oder Hardware stark ein. Speicheroperationen dürfen nicht mehr überlappen oder sich gegenseitig überholen. SC bietet ein einfaches Programmiermodell aber die Performance ist durch diese Einschränkungen relativ schlecht. Eine Lösung dieses Problems bieten die sogenannten entspannten Konsistenzmodelle, die mehr Optimierungen ermöglichen, dafür aber ein komplizierteres Programmiermodell mit sich bringen.

Anhand des oben bereits eingeführten Programms kann man sequentielle Konsistenz wie folgt erläutern. Die sequentielle Konsistenz verlangt, daß alle Prozessoren ihre Operationen in Programmreihenfolge ausführen. So muss Prozessor 1 erst data1 schreiben, dann data2 und zuletzt flag. Prozessor 2 muss erst innerhalb der while-Schleife flag lesen, dann register1 schreiben und zuletzt register2. Dabei kann es zu Verschränkungen zwischen den einzelnen Speicheroperationen kommen. So könnte Prozessor 1 erst data1 und dann data2 von Prozessor 1 geschrieben werden bevor Prozessor 2 das erste Mal auf flag zugreift. Es kann aber auch sein, daß erst Prozessor 1 data1 schreibt und dann Prozessor 2 den ersten Zugriff auf flag ausführt bevor Prozessor1 data2 schreibt. Diese Ausführungsreihenfolgen und andere sind möglich. Alle Prozessoren sehen aber immer dieselbe Ausführungsreihenfolge, egal um welche Ausführungsreihenfolge es sich dabei handelt. Prozessor 2 kann also erst die while-Schleife verlassen, wenn Prozessor 1 alle seine Speicheroperationen ausgeführt hat. Die beiden Prozessoren können sich über die gemeinsame Variable flag synchronisieren und das Programm wird korrekt ausgeführt.

Prozessor 1	Prozessor 2
data1 = 64	while(flag != 1) {;}
data2 = 55	register1 = data1
flag = 1	register2 = data2

Processor Consistency
Nach Goodman ist ein Multiprozessor prozessorkonsistent, wenn das Ergebnis irgendeiner Ausführung dasselbe ist, als wenn die Operationen jedes einzelnen Prozessors in der sequentiellen Reihenfolge, die durch sein Programm bestimmt wird, erscheinen. Das heißt, dass es im Gegensatz zur sequentiellen Konsistenz keine für alle Prozessoren einheitliche Reihenfolge mehr geben muss, in der Speicheroperationen sichtbar werden. Die Schreibzugriffe zweier Prozessoren können von einem dritten Prozessor durchaus in einer anderen Reihenfolge gesehen werden, als die beiden schreibenden Prozessoren sie sehen. Jeder Prozessor muss aber immer noch die Schreibzugriffe eines anderen Prozessors in der Reihenfolge sehen, in der der andere Prozessor sie ausgibt.

Eine formellere Definition nach Gharachorloo besagt, dass bevor ein Lesezugriff erfolgen darf, alle vorhergehenden Lesezugriffe erfolgt sein müssen. Und bevor ein Schreibzugriff erfolgen darf müssen alle vorhergehenden Zugriffe (lesen und schreiben) ausgeführt worden sein. Diese Definition erlaubt gegenüber der Definition von Goodman, dass eine Leseoperation, die auf eine Schreiboperation folgt mit der Schreiboperation in der Ausführungsreihenfolge vertauscht werden darf. Eine sequentiell konsistente Ausführung kann erzwungen werden, wenn anstatt von Schreiboperationen read-modify-write-Operationen verwendet werden.

Anhand von unten aufgeführtem Programm kann man die Auswirkungen der Prozessorkonsistenz erklären. Prozessor 1 und Prozessor 2 wollen sich über die gemeinsame Variable flag synchronisieren, so daß immer nur einer von beiden Prozessen die kritische Region betritt. Bei einem sequentiell konsistenten Speicher wäre das kein Problem. Beide Prozessoren würden ihre Anweisungen in Programmreihenfolge abarbeiten und es würde maximal ein Prozessor zu einem beliebigen Zeitpunkt die kritische Region bearbeiten.

Die Prozessorkonsistenz erlaubt allerdings, daß eine Lesoperation, die auf eine Schreiboperation folgt mit dieser vertauscht wird. So könnte Prozessor 1 und Prozessor 2 erst den Wert von flag2 und flag1 auslesen und anschließend erst flag1 und flag2 auf 1 setzen. Beide Prozessoren hätten in diesem Fall eine 0 gelesen und würden die kritische Region betreten.

Die sequentiell konsistente Ausführung läßt sich erzwingen, indem die Schreiboperationen durch atomare Read-Modify-Write-Operationen ersetzt würden. Der gelesene Wert wird dabei ignoriert und anschließend der beabsichtigte Wert geschrieben. Gharachorloo sagt, daß ein Lesezugriff erst erfolgen darf, wenn alle vorhergehenden Lesezugriffe erfolgt sind. Eine Read-Modify-Write-Operation beinhaltet einen Lesezugriff und muss atomar ausgeführt werden. Für Prozessor 1 würde das bedeuten, daß flag2 nicht ausgelesen werden darf, bevor die Read-Modify-Write-Operation auf flag1 vollendet ist, weil die Read-Modify-Write-Operation einen vorhergehenden Lesezugriff enthält.

Prozessor1	Prozessor2
flag1 = 1	flag2 = 1
if(flag2 == 0)	if(flag1 == 0)
kritische Region	kritische Region

Weak Ordering
Auch schwache Konsistenz genannt führt die Synchronisationsoperation als einen neuen Speicherzugriffsoperationstyp ein. Bei Ausführung einer Synchronisationsoperation werden alle schwebenden Schreiboperationen ausgeführt und keine neuen Schreiboperationen gestartet. Synchronisationsoperationen selbst sind zueinander sequentiell konsistent. Speicherzugriffe auf Daten (lesen und schreiben) zwischen zwei Synchronisationsoperationen können beliebig vertauscht werden.

Das unten aufgeführte Programm veranschaulicht die Funktionsweise. Die Schreiboperationen auf data1 und data2 auf Prozessor 1 dürfen sich gegenseitig überholen. Nachdem die sync-Operation beendet ist, müssen diese Schreiboperationen für alle anderen Prozessoren im Hauptspeicher sichtbar sein. Erst jetzt ist es dem Prozessor 1 möglich flag auf 1 zu setzen. Die lesenden Zugriffe auf data1 und data2 von Prozessor 2 dürfen erst ausgeführt werden nachdem die sync-Operation auf Prozessor 2 vollendet ist.

Prozessor 1	Prozessor 2
data1 = 64	while(flag != 1) {;}
data2 = 55	sync
sync	register1 = data1
flag = 1	register2 = data2

Release Consistency
Die schwache Konsistenz ist relativ ineffizient, weil sie erst alle schwebenden Schreibzugriffe fertig stellen, und neue Schreiboperationen zurückhalten muss. Mit der Freigabekonsistenz lässt sich dies verbessern. Bei der Freigabekonsistenz müssen nicht nachdem ein Prozess eine kritische Region verlässt alle ausstehenden Schreiboperationen abgearbeitet werden. Es muss lediglich sichergestellt werden, dass die ausstehenden Schreiboperationen vor dem erneuten Betreten der kritischen Region abgeschlossen sind. Dadurch befinden sich die Daten in der kritischen Region immer in einem konsistenten Zustand, wenn die kritische Region betreten wird. Die Freigabekonsistenz definiert zwei Synchronisationsoperationen: acquire und release. Eine CPU, die ein gemeinsames Datum innerhalb einer kritischen Region beschreiben will führt erst ein acquire auf einer Synchronisationsvariablen aus. Dadurch erhält sie exklusiven Zugriff auf dieses Datum und alle anderen gemeinsamen Daten. Die CPU kann nun diese Daten beliebig verarbeiten. Wenn die CPU fertig ist, führt sie ein release auf der Synchronisationsvariablen aus. Durch das release werden nicht zwangsläufig alle ausstehenden Schreiboperationen zu Ende ausgeführt. Darüber hinaus werden auch neue Schreiboperationen nicht daran gehindert zu starten. Das Release ist aber erst abgeschlossen, wenn alle zuvor gestarteten Schreiboperationen beendet sind. Ein anschließendes Acquire kann erst ausgeführt werden, wenn alle vorherigen Release-Operationen abgeschlossen wurden. Die Freigabekonsistenz ermöglicht es, dass zwei aufeinanderfolgende Datenzugriffsoperationen sich gegenseitig überholen können. Ein Datenzugriff, gefolgt von einem Acquire kann das Acquire überholen. Ein Release gefolgt von einem Datenzugriff kann den Datenzugriff überholen.

Folgendes Beispiel veranschaulicht die Funktionsweise.
Prozessor 1 führt ein Acquire aus und kann die folgenden Schreiboperationen bis zum nächsten Release beliebig ausführen. Prozessor 2 liest flag solange, bis der Wert 1 gelesen wird und führt dann ein Acquire aus. Das Acquire ist erst beendet, wenn alle ausstehenden Schreiboperationen vollendet sind. Und somit stehen in data1 und data2 die Werte, die von Prozessor 1 geschrieben wurden.

Prozessor 1	Prozessor 2
acquire	while(flag != 1) {;}
data1 = 64	acquire
data2 = 55	register1 = data1
flag = 1	register2 = data2
release	release

Hardwareaspekte von Konsistenzmodellen
Fortschritte in der Hardware, und wie dadurch die Leistung von Konsistenzmodellen verbessert werden kann.

In heutigen Prozessoren gibt es gewisse Merkmale, die es ermöglichen Konsistenzmodelle zu optimieren.
Diese Merkmale sind im Einzelnen:

Out-of-order Scheduling
Ein Prozessor kann mehrere aufeinander folgende Anweisungen innerhalb eines sog. „Instruction Window" gleichzeitig untersuchen. Nachdem die Datenabhängigkeiten einer Anweisung im Instruction Window festgestellt wurden, kann diese Anweisung an eine Funktionale Einheit zur Ausführung übergeben werden. Die Ergebnisse werden aber in Programmreihenfolge sichtbar.

Non-Blocking-Loads:
Ein Prozessor ist nicht blockiert, während er einen Ladevorgang ausführt. Der Prozessor kann andere Anweisungen, die von dem Ladevorgang unabhängig sind weiter ausführen. Dies schließt auch weitere Ladevorgänge ein. Ein Ladevorgang kann das „Instruction Window" nicht verlassen, bevor es nicht einen Wert zurückgegeben hat. Daher ist die Effektivität dieser Technik direkt von der Größe des „Instruction Window" abhängig. Denn die Größe bestimmt, wie viele Anweisungen sich mit dem Ladevorgang überlappen können.

Speculation:
Heutige Prozessoren unterstützen spekulative Ausführung, wie z. B. Branch Prediction. So ist es z. B. möglich Anweisungen jenseits eines Speculation Point (z. B. eine Verzweigung im Programmablauf) zu decodieren, zu verteilen und auszuführen. Die Ergebnisse dieser Anweisungen dürfen aber solange nicht sichtbar werden, wie alle Spekulationen von denen sie abhängen nicht eingetreten sind. Wenn sich eine Spekulation als falsch herausstellt, werden alle davon abhängigen Anweisungen und deren Ergebnisse verworfen.

Prefetching:
Prozessoren können Cachezeilen anfordern, die wahrscheinlich in naher Zukunft gebraucht werden. Die Zeit, die zum Einlagern der Cachezeile in den Cache des Prozessors benötigt wird kann mit andere anstehender Arbeit gefüllt werden. Prefetch-Operationen können entweder durch Software explizit (Prefetch-Anweisungen) initiiert werden oder zur Laufzeit durch die Hardware.

Anwendung dieser Merkmale zur Optimierung von Konsistenzmodellen

Hardware Prefetching aus dem Instruction Window

Angenommen das Instruction Window enthält mehrere dekodierte Speicherzugriffsoperationen. Diese Speicheroperationen müssen nacheinander ausgeführt werden, um die Konsistenz nicht zu gefährden. Der Prozessor kann aber nicht-bindende Prefetch-Operationen ausführen. Dadurch bleiben einige Verzögerungen, die durch Speicherzugriffe entstehen verborgen.
Die I/O-Verzögerungen, die man durch diese Technik verbergen kann sind durch die Größe des Instruction Window beschränkt. Allerdings können Prefetch-Operationen, die zu früh ausgeführt werden die Performance auch beeinträchtigen. Das liegt z. B. an zusätzlichem Netzwerkverkehr. Oder andere Prozessoren erleiden zusätzliche Verzögerungen, weil sie ihre Cachezeilen vorzeitig verlieren.

Speculative Load Execution

Normalerweise darf ein Wert, der durch Hardware Prefetching geladen wurde solange nicht benutzt werden, wie vorhergehende Speicheroperationen nicht abgeschlossen sind. Durch Speculative Load Execution kann der Vorteil den man durch das Hardware Prefetching erhalten hat weiter ausgenutzt werden. Die vorzeitig geladenen Werte werden einfach und ohne Rücksicht auf Konsistenzbedingungen zur Verwendung freigegeben. Falls die verwendete, zuvor per Prefetching geladene Speicheradresse nicht verändert wird bis zu dem Zeitpunkt, wo sie nicht-spekulativ hätte verarbeitet werden können, dann war die spekulative Verarbeitung des geladenen Werts erfolgreich. Anderenfalls muss die Berechnung zurückgerollt werden, bis zu dem Punkt, wo der falsche Wert geladen wurde. Um zu erkennen, ob ein gelesener und verwendeter Wert verändert wird, bevor er nicht-spekulativ hätte verwendet werden dürfen, benutzt der Prozessor die Cachekohärenz Mechanismen Hardware-Cachekohärenter Multiprozessorsysteme. Wenn ein Prozessor eine Cachezeile verändert, wird an alle anderen Caches, die diesen Wert beinhalten die Veränderung signalisiert. Der Prozessor beobachtet die Cachekohärenznachrichten an seinen Cache und kann so feststellen, ob ein von ihm spekulativ verarbeiteter Wert verändert wurde. Der maximale Performancegewinn ist durch die Größe des Instruction Window beschränkt.Außerdem führt zu frühes spekulatives Verarbeiten von geladenen Werten zu erhöhter Wahrscheinlichkeit eines Rollback, was den Performancegewinn ebenfalls einschränkt.

Cross-Window-Prefetching

Prefetch-Operationen können auch für Instruktionen ausgeführt werden, die zurzeit nicht im Instruction Window sind, aber es anzunehmen ist, dass die demnächst gebraucht werden. Solche Prefetches können entweder vom Compiler explizit durch Prefetchanweisungen in den Code eingefügt werden oder von der Hardware zur Laufzeit selbst bestimmt werden. Diese Technik ist zwar auch auf Einzelprozessorsystemen anwendbar, doch lässt sich mit dieser Technik auch die Performance von Konsistenzmodellen verbessern. Cross-Window-Prefetching reduziert die Einschränkungen, die man durch ein zu kleines Instruction Window hat

Compileraspekte zur Gewährleistung von Speicherkonsistenz
Auf Einzelprozessorsystemen kann man Compileroptimierungen, wie z. B. umordnen oder entfernen von Speicheroperationen verwenden, die man bei Anwendung der Konsistenzmodelle SC oder PC nicht einfach anwenden kann. Eine Ausführungsreihenfolge eines parallelen Programms kann durch einen Graphen dargestellt werden. Die Knoten sind die Speicheroperationen. In diesem Graph unterscheidet man zwei Arten von Kanten. Eine Kante von einem Knoten A zu einem Knoten B ist vom Typ „programm order“, wenn in der Programmreihenfolge A vor B kommt. Eine Kante von A nach B ist eine „conflict order“-Kante, wenn A vor B ausgeführt wird und ein Konflikt besteht zwischen A und B (Konflikt: Zwei Operationen stehen in Konflikt zueinander, wenn sie auf verschiedenen Prozessoren ausgeführt werden, auf die gleiche Speicherstelle zugreifen und zumindestens eine dieser Operationen ist eine Schreiboperation.) Wenn dieser Graph keine Kreise hat, dann scheint diese Ausführung des Programms sequentiell konsistent zu sein. Operationen, die durch programm-order-Kanten verbunden sind und nicht zu einem Kreis gehören können umgeordnet werden.

Speicherkonsistenzmodelle von Programmierumgebungen
Speicherkonsistenzmodelle wurden ursprünglich ausschließlich für die Hardwareschnittstelle entworfen. Hohe Programmiersprachen boten keine direkte Unterstützung für explizit parallele Programme auf einem gemeinsamen Speicher. Stattdessen benutzte man parallelisierende Compiler oder nicht-standardisierte Spracherweiterungen. Heutige Programmiersprachen sind in der Lage mit Parallelität auf gemeinsamen Speicher umzugehen.

POSIX Threads
POSIX ist ein IEEE Standard, der ein Thread-Interface für die C Programmiersprache definiert. POSIX Threads beinhalten Funktionen, um „Gegenseitigen Ausschluss“ mit Locks und Bedingungsvariablen zu implementieren. Programme, die diese Funktionen richtig verwenden können sich auf sequentiell konsistente Ergebnisse verlassen. Programme, die sich anders synchronisieren, wie z. B. durch Dekker´s Algorithmus können unvorhersehbare Ergebnisse erhalten. POSIX Threads sind sehr mächtig, weshalb sie manche kleinere Aufgaben auch eher ausbremsen.

Der `volatile` *Bezeichner*
Um bestimmte Compileroptimierungen zu unterdrücken kann man in den Programmiersprachen ANSI C, C++ und Java den volatile-Bezeichner verwenden. Ein entscheidender Grund für diesen Bezeichner war, dass man die Möglichkeit braucht in bestimmten Programmen, wie z. B. Gerätetreibern oder Interupthandler Compileroptimierungen zu unterdrücken. Man verwendet ihn aber auch, um Datenkonsistenz in Multiprozessorsystemen mit gemeinsamen Speicher zu realisieren. Die Java Language Specification enthält die genaueste Spezifikation der Semantik des volatile-Bezeichners. Demnach muss jeder Zugriff auf eine als volatile deklarierte Variable einen Hauptspeicherzugriff auslösen. Das bedeutet z. B., dass Werte einer solchen Variable nicht in Registern zwischengespeichert werden können. Des weiteren muss der Zugriff auf mehrere als volatile deklarierte Variablen in der Reihenfolge erfolgen, die durch das Programm vorgegeben ist. Speicherzugriffe auf volatile Variablen können sich nicht gegenseitig überholen. Es gibt aber keine Einschränkung bzw. Vorgaben über die Ausführungsreihenfolge einer volatilen gegenüber einer nicht-volatilen Variablen.

Das Java-Konsistenzmodell
Das Java-Konsistenzmodell besteht aus einer Menge von Regeln auf einer abstrakten Darstellung des Systems. Es gibt einen sogenannten Main Memory. Dieser enthält die sogenannte Master Copy aller Variablen. Jeder Thread hat einen Working Memory. Wenn ein Thread eine Anweisung ausführt, dann erfolgen use- und/oder assign-Operationen auf den Werten im Working Memory des Threads. Diese use- und assign-Operationen werden in Programmreihenfolge ausgeführt. Die jeweilige Implementierung verschiebt Werte zwischen dem Main Memory und den einzelnen Working Memories aufgrund mehrerer definierter Regeln. Java-Programme verwenden die Schlüsselwörter „synchronized“ und „volatile“, um die Speicherzugriffe zu ordnen. Synchornized bewirkt eine Sperrung der zugeordneten Variablen. Anschließende wird die zugehörige Anweisung (Block) oder Methode ausgeführt. Hinterher wird die Sperre wieder aufgehoben. Das Konsistenzmodell von Java ist sehr komplex. Aber es gibt zwei Regeln, die die Reihenfolge der Speicherzugriffe betreffen, die besonders wichtig sind. Die erste Regel besagt, dass nach einem sperrenden Zugriff muss sich ein Thread so verhalten, als ob jede Variable V vor der nächsten Benutzung durch den Thread aus der Working Copy des Threads entfernt wurde. Das heißt, der Thread muss sich einen gültigen Wert aus der Master Copy holen, wenn er diese Variable V benutzen will. Es sei denn, der Thread verwendet diese Variable definierend (Wertzuweisung) nach dem Setzen der Sperre und vor der ersten benutzenden Verwendung.
Die zweite Regel besagt, dass vor dem Aufheben der Sperre alle Variablen die durch diesen Thread definiert wurden zurück in die Master Copy geschrieben werden müssen.
Eine Programmierrichtlinie besagt, dass wenn eine Variable jemals von einem Thread definiert wird und von einem anderen Thread benutzt oder definiert wird, dann sollen sämtliche Zugriffe auf diese Variable durch synchronized-Blöcke oder synchronized-Methoden umschlossen sein.

OpenMP – Speicherkonsistenz mit der FLUSH-Direktive
Die Flush-Direktive in OpenMP spezifiziert einen Synchronisationspunkt. An diesem Punkt muss die Implementierung für einen konsistenten Speicher sorgen. Diese Direktive gewährleistet, dass alle Schreiboperationen, die von einem Thread vorgenommen wurden für alle anderen sichtbar werden. Compiler müssen dafür sorgen, dass z. B. Register zurück in den Hauptspeicher geschrieben werden oder Schreibpuffer geleert werden. Leseoperationen, die auf eine Flush-Direktive folgen liefern neue Werte. Einige OpenMP-Direktiven führen implizit Flush-Direktiven aus. (barrier, parallel, for, …)