Studienarbeit
'Träge Release-Konsistenz'

Peter Weisse
weisse@informatik.hu-berlin.de

Humboldt Universität Berlin
Fachbereich Informatik, Lehrstuhl für
Softwaretechnik und Theorie der Programmierung II

Betreuer: Dr. Frank Müller

21. Februar 2000

Zusammenfassung

Ein verteiltes System ist eine Menge voneinander unabhängiger Computer, die dem Benutzer des Systems den Eindruck vermitteln, es handle sich um einen einzigen Computer [30, 30].

Ein verteiltes System, welches einen einzigen virtuellen Adreßraum benutzt, wird als DSM-System (*Distributed Shared Memory, verteilter gemeinsam genutzter Speicher*) bezeichnet[6, 7, 27, 28, 5].

Die Performance einer DSM-Struktur ist stark abhängig von der Nachrichtenmenge, die über das Netzwerk übertragen werden muss. Eine Strategie, den Netzverkehr zu vermindern, ist die Replikation gemeinsam genutzter Variablen auf mehreren Maschinen. Dadurch wird das Problem der gemeinsamen Nutzung von Variablen darauf reduziert, wie verschiedene Kopien einer Menge von Datenstrukturen konsistent gehalten werden können. Dazu werden verschiedene Konsistenzmodelle definiert.

Gegenstand dieser Arbeit ist eine Erweiterung der Funktionalität des bestehenden DSM-Systems [27, 28] um das MRMW-Protokoll (*Multi-Reader Multi-Writer, viele Leser, viele Schreiber*). Dies wird durch die Benutzung des trägen Release-Konsistenzmodelles erreicht. Es werden der gegenseitige Ausschluß[29] und Barrieren mit Hilfe des MRMW-Protokolles[3] implementiert.

Eine erfolgte Änderung am bestehenden DSM-System ist die Erzeugung eines Moduls, welches Zeitstempel verwalten kann. Zusätzlich wurden die Module, die Diffs und Twins verwalten, den neuen Erfordernissen angepaßt.

Nicht behandelt werden alle Aspekte einer zufriedenstellenden Speicherfreigabe. Im Besonderen besteht das Problem der Dealloziierung der erzeugten Diffs und die Nichtbeachtung eventueller Zählerüberläufe. Eine zukünftige Aufgabe ist die Verknüpfung des MRMW-Protokolles zu anderen, bereits bestehenden Protokollen. Diese Einschränkungen sollen im Rahmen einer anschließenden Diplomarbeit untersucht und beseitigt werden.

Inhaltsverzeichnis

1 Einführung **3**

2 Terminologie **5**

3 Grundlagen **6**
3.1 Synchronisation 7
3.2 Konsistenz 9
3.3 Release-Konsistenz 10
3.4 Eifrige vs. Träge Release-Konsistenz 13

4 Gegenseitiger Ausschluß **15**

5 Barrieren **23**

6 Veränderungen am DSM-System **28**
6.1 Das Modul Timestamp 29
6.2 Das Modul Diff 34
6.3 Das Modul Twin 39
6.4 Das Modul Release 40
6.5 Die Aufrufhierarchie 42

7 Schlußwort / Ausblick **44**

Abbildungsverzeichnis **45**

Tabellenverzeichnis **45**

Literatur **46**

1 Einführung

Die nebenläufige und parallele Programmierung eröffnet neue Möglichkeiten, Algorithmen zu entwerfen und reale Abläufe zu modellieren.

Ein verteiltes System ist eine Menge voneinander unabhängiger Computer, die dem Benutzer des Systems den Eindruck vermitteln, es handle sich um einen einzigen Computer [30, 31]. Die Grundlage dieser Systeme sind Multicomputer bzw. Multiprozessoren. Entgegengesetzt zu Multiprozessoren sind Multicomputer einfacher aufzubauen und schwieriger zu programmieren. Li (1986)[25] und Li und Hudak (1989)[24] definierten ein verteiltes DSM-System auf der Basis von Multicomputern, welches einfach aufzubauen und zu programmieren war.

Ein verteiltes System, dass einen einzigen virtuellen Adreßraum benutzt, wird als DSM-System (*Distributed Shared Memory, verteilter gemeinsam genutzter Speicher*) bezeichnet [27, 28, 30, 31]. Eine Menge von Workstations, die über ein LAN verbunden sind, benutzen einen virtuellen gemeinsamen Adressraum. Der Zugriff auf eine lokal vorhandene Seite erfolgt hardwaremäßig, mit voller Geschwindigkeit. Der Versuch, auf eine Seite einer anderen Maschine zuzugreifen, bewirkt einen Hardware-Seitenfehler, wodurch ein Trap in das Betriebssystem erzeugt wird. Das Betriebssystem sendet daraufhin eine Nachricht an die entfernt gelegene Maschine, die die benötigte Seite sucht und sie an den anfordernden Prozessor sendet. Die zuerst fehlgeschlagene Anweisung wird wiederholt und kann jetzt abgeschlossen werden.

Die Performance einer DSM-Struktur ist stark abhängig von der Nachrichtenmenge, die über das Netzwerk übertragen werden muss. Eine Strategie, den Netzverkehr zu vermindern, ist die Replikation gemeinsam genutzter Variablen auf mehreren Maschinen. Dadurch wird das Problem der gemeinsamen Nutzung von Variablen darauf reduziert, wie verschiedene Kopien einer Menge von Datenstrukturen bzw. Seiten konsistent gehalten werden können. Dazu werden verschiedene Konsistenzmodelle definiert.

Gegenstand dieser Arbeit ist eine Erweiterung der Funktionalität des bestehenden DSM-Systems [27, 28] um das MRMW-Protokoll (*Multi-Reader Multi-Writer, viele Leser, viele Schreiber*). Dies wird durch die Benutzung des trägen Release-Konsistenzmodelles erreicht. Es werden der gegenseitige Ausschluß[29] und Barrieren mit Hilfe des MRMW-Protokolles[3] implementiert.

Diese Arbeit ist in folgende Abschnitte gegliedert. Der Abschnitt 2 erläutert die zentralen Begriffe dieser Arbeit und der verwendeten Semantik.

Das Kapitel 3 geht allgemein auf Konsistenzmodelle und im Besonderen auf das implementierte Konsistenzmodell näher ein. Im Kapitel 4 wird über die Implementation des gegenseitigen Ausschlusses und im letzten Abschnitt 5 über die Barrieren referiert. Abschnitt 6 zeigt die Veränderungen auf, die am bestehenden DSM-System nötig waren. Zum Schluß wird im Kapitel 7 eine Zusammenfassung dieser Arbeit und ein Ausblick auf zukünftige Aufgabenstellungen umrissen.

Nicht behandelt werden alle Aspekte einer zufriedenstellenden Speicherfreigabe. Im Besonderen besteht das Problem der Dealloziierung der erzeugten Diffs und die Nichtbeachtung eventueller Zählerüberläufe. Unter der Voraussetzung, dass alle Prozesse an einer Barriere teilnehmen, könnte dort die gesamte Garbage Collection durchgeführt werden. Eine zukünftige Aufgabe ist die Verknüpfung des MRMW-Protokolles zu anderen, bereits bestehenden Protokollen. Diese Einschränkungen sollen im Rahmen einer anschließenden Diplomarbeit untersucht und beseitigt werden.

2 Terminologie

In der Informatik ist die Verwendung von Begriffen aus dem Englischen sehr verbreitet. Diese Studienarbeit einer deutschen Universität versucht, so viele Begriffe wie möglich aus dem Deutschen zu verwenden. Problematisch sind Verben, deren Konjugation im Deutschen nicht festgeschrieben ist.

Die Begriffe, zu denen es keine prägnante deutsche Entsprechung gibt, wurden englisch belassen. Dazu gehören Prozess, Thread, Mutex.

Prozesse stehen für UNIX-Prozesse. Ein Prozess ist dabei die spezifische Laufzeitumgebung eines Programmes. Ein Prozess hat einen eigenen Kontrollfluß und Speicher. Er wird über eine Kombination aus Hostnamen und Portadresse identifiziert. Prozesse werden vom System geschedult, d.h sie erhalten CPU-Zeit nach einem vom System vorgegebenen Plan. Wenn ein Prozess die CPU wieder abgeben muß, dann findet ein Kontextwechsel statt. Hierbei müssen alle prozessrelevanten Informationen gespeichert werden.

Threads sind nebenläufige Einheiten, die gemeinsamen Speicher, aber getrennte Kontrollflüsse haben. Bei der betrachteten Bibliotheksimplementierung der POSIX-Threads sind Threads immer einem Prozess zugeordnet, soweit es sich um ein UNIX-ähnliches System handelt. Threads sind im Gegensatz zu Prozessen ,,leichtgewichtiger", d.h die Thread-relevanten Informationen, die für die Ausführung bzw. Unterbrechung wichtig sind, haben einen kleineren Umfang. Da Threads einem Prozess zugeordnet sind, muß man bei einem Thread-Kontextwechsel keine prozessspezifischen Daten speichern. Wenn der zugeordnete Prozess die CPU abgeben muß, dann werden vom System alle prozessspezifischen Daten gerettet. Threads können darüber hinaus Multiprozessoren mit gemeinsamen Speicher ausnutzen, um die Programmausführung zu beschleunigen.

Als Mutex wird hier das Objekt bezeichnet, dass zur Realisierung eines gegenseitigen Ausschlusses (*mutual exclusion*)[29] benutzt wird. Durch ein mutex lock kann dieser Mutex (für kurze Abschnitte) von einem Thread angefordert werden und mittels mutex unlock wieder abgegeben werden.

3 Grundlagen

Die Möglichkeiten zur Synchronisation von Prozessen in verteilten Systemen sind vielfältig. Es wird zwischen absoluter und logischer Zeitsynchronisation unterschieden. Die absolute Zeitsynchronisation ist in verteilten Systemen nicht, oder nur mit großem Aufwand, realisierbar. Eine logische Synchronisation von Uhren ist möglich, wenn das Fehlen der Synchronisation nicht auffällt.

Die Anzahl der zu verschickenden Nachrichten beeinflußt die Performance des Programms wesentlich. Durch die Einführung verschiedener Konsistenzmodelle kann die Menge von Nachrichten vermindert werden. Schwächere Konsistenzmodelle bedingen zusätzliche Einschränkungen, die der Programmierer zu beachten hat.

Abbildung 1 zeigt die verschiedenen Typen des Zugriffs auf gemeinsame Variablen in verteilten Systemen.

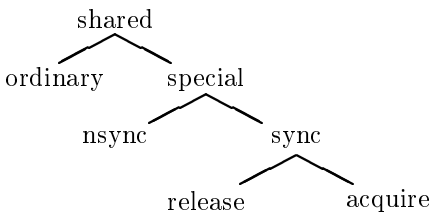

Abbildung 1: Zugriffstypen auf den Speicher bei verteilten Anwendungen

3.1 Synchronisation

Ein nichttriviales Problem ist die Synchronisation von Prozessen in verteilten Systemen.

In einem Computer mit einem Prozessor kann die interne Uhr als absolute Zeitsynchronisation benutzt werden. Dabei ist es unwichtig, ob diese Uhr von der realen Zeit abweicht. Wenn alle Prozesse auf der Maschine dieselbe Uhr verwenden, sind sie doch intern konsistent.

Sobald jedoch mehrere CPUs mit jeweils einer eigenen Uhr eingesetzt werden, verändert sich diese Situation. Es kann nicht garantiert werden, dass alle Uhren mit genau derselben Frequenz laufen. Diese *Uhrabweichung* bedingt unterschiedliche Auslesungen der Werte und somit die Ursache für Synchronisationsfehler.

In [20, 21, 22] und [23] stellte Lamport einen klassischen Algorithmus vor, der zeigte, dass die Uhrsynchronisation möglich ist. Er legte dar, dass die Uhrsynchronisation nicht absolut sein muß. Wenn zwei Prozesse miteinander interagieren, dann ist es nicht notwendig, dass ihre Uhren synchronisiert werden, weil das Fehlen der Synchronisation nicht auffällt und somit keine Probleme verursacht. Darüber hinaus hat er gezeigt, dass es normalerweise nicht darauf ankommt, dass alle Prozesse sich genau auf eine Zeit einigen, sondern dass sie sich auf die Reihenfolge einigen, in der die Ereignisse auftreten. Für eine bestimmte Klasse von Algorithmen ist es die interne Konsistenz, auf die es ankommt, nicht, ob sie der realen Zeit entsprechen. Für diese Algorithmen werden die Uhren auch als *logische* Uhren bezeichnet.

Um logische Uhren zu beschreiben, definierte Lamport[20] eine Relation *passiert_vor*. Der Ausdruck $a \rightarrow b$ wird gelesen als *passiert_vor* und bedeutet, dass alle Prozesse übereinstimmen, dass zuerst Ereignis a passiert und danach Ereignis b.

- Wenn a und b Ereignisse im selben Prozess sind und a vor b geschieht, dann gilt $a \rightarrow b$.
- Wenn Ereigniss a das Senden einer Nachricht von einem Prozess ist und Ereignis b das Empfangen dieser Nachricht in einem anderen Prozess, dann trifft ebenfalls $a \rightarrow b$ zu. Eine Nachricht kann nicht empfangen werden, bevor sie gesendet wurde, weil es eine endliche Zeit dauert, bis die Nachricht angekommen ist.
- Die Transitivität dieser Relation gilt: wenn $a \rightarrow b$ und $b \rightarrow c$, dann $a \rightarrow c$.

Wenn zwei Ereignisse x und y in verschiedenen Prozessen auftreten, die keine Nachrichten austauschen (nicht einmal indirekt über Dritte), dann trifft $x \rightarrow y$ nicht zu, aber auch nicht $y \rightarrow x$. Diese Ergebnisse heißen *nebenläufig*, was bedeutet, dass man nichts darüber aussagen kann, welches dieser Ereignisse zuerst passiert ist.

Lamports Algorithmus folgt direkt aus der *passiert_vor*-Relation. Er gibt eine Methode an, die für jedes Ereignis a einen Zeitwert $C(a)$ definiert, über den sich alle Prozesse einig sind. Diese Zeitwerte müssen die Eigenschaft besitzen, dass, wenn $a \rightarrow b$ gilt, auch $C(a) < C(b)$ gilt. Mit anderen Worten, wenn a vor b geschieht, muß auch $C(a) < C(b)$ sein. Wenn a das Senden einer Nachricht von einem Prozess beschreibt und b den Empfang dieser Nachricht durch einen anderen Prozess, dann müssen $C(a)$ und $C(b)$ so angegeben werden, dass jeder die Werte von $C(a)$ und $C(b)$ mit $C(a) < C(b)$ akzeptiert. Darüber hinaus muß die Uhrzeit C immer ansteigen.

Wenn eine Nachricht ankommt und die Uhr des Empfängers einen Wert zeigt, der vor der Zeit liegt, zu der die Nachricht abgeschickt wurde, stellt der Empfänger seine Uhr nach vorne, damit sie einen Tick mehr als die Sendezeit anzeigt.

Mit dieser Methode existiert die Möglichkeit, allen Ereignissen in einem verteilten System eine Zeit zuzuordnen, die den folgenden Bedingungen genügt:

- Wenn a in demselben Prozess vor b passiert, dann gilt $C(a) < C(b)$.
- Wenn a und b das Senden und Empfangen einer Nachricht darstellen, dann gilt $C(a) < C(b)$.
- Für alle Ereignisse a und b gilt $C(a) \neq C(b)$.

Die logischen Uhren wurden für das Modell der trägen Release-Konsistenz im bestehenden DSM-System implementiert (siehe Kapitel 6.1 auf Seite 29).

3.2 Konsistenz

Ein Konsistenzmodell ist ein Vertrag zwischen Software und Speicher[2]. Wenn die Software bereit ist gewissen Regeln zu gehorchen, verspricht der Speicher korrekt zu arbeiten. Verletzt die Software diese Regeln, gilt der Vertrag nicht mehr und die Korrektheit der Speicheroperationen ist nicht länger sichergestellt. Es gibt ein großes Spektrum an möglichen Verträgen, von solchen, die der Software nur kleine Einschränkungen auferlegen, bis hin zu solchen, die eine normale Programmierung unmöglich machen. Die Software mit den geringeren Einschränkungen bieten keine so gute Performance wie diejenigen mit den größeren Forderungen.

Folgende Konsistenzmodelle werden unterschieden: [30, 31, 2, 6, 7, 8]

- Strenge Konsistenz
- Sequentielle Konsistenz [20, 21, 22, 23]
- Kausale Konsistenz [14]
- PRAM-Konsistenz [26]
- Prozessor-Konsistenz [13]
- Schwache Konsistenz [11, 10]
- Release-Konsistenz [12, 17, 16, 19, 18]
- Eintritts-Konsistenz [9, 15, 1]
- Bereichs-Konsistenz [15]

In dieser Arbeit wird auf die Release-Konsistenz näher eingegangen. Für alle anderen Konsistenzmodelle wird auf die im Quellenverzeichnis genannte Literatur verwiesen.

3.3 Release-Konsistenz

Im Release-Konsistenzmodell erfolgt eine wesentliche Unterscheidung zwischen Ein- und Austritt in den kritischen Bereich. *Acquire*-Zugriffe teilen dem Speichersystem mit, dass ein kritischer Abschnitt betreten werden soll. *Release*-Zugriffe besagen, dass ein kritischer Bereich gerade verlassen wurde. In jedem Falle ist der Programmierer dafür verantwortlich, den expliziten Code in sein Programm aufzunehmen, der angibt, wann sie ausgeführt werden sollen.

Bei der Release-Konsistenz ist es außerdem möglich, *Barrieren* statt kritischer Bereiche zu verwenden. Eine Barriere ist ein Synchronisierungs-Mechanismus, der einen Prozess daran hindert, Phase *n+1* eines Programmes zu beginnen, bis alle Prozesse Phase *n* beendet haben. Wenn ein Prozess an eine Barriere kommt, dann muß er warten, bis alle anderen Prozesse auch dort angekommen sind. Wenn der letzte Prozess ankommt, dann werden alle gemeinsam genutzten Variablen synchronisiert, und alle Prozesse werden fortgesetzt. Die Ankunft an der Grenze bedeutet *Release*, ihr Verlassen bedeutet *Acquire*.

Neben diesen Synchronisierungs-Zugriffen ist auch das Lesen und Schreiben gemeinsam genutzter Variablen möglich. Acquire und Release müssen nicht auf den gesamten Speicher angewendet werden. Statt dessen können sie auch eingesetzt werden, um nur bestimmte gemeinsam genutzte Variablen zu bewachen. In diesem Falle werden nur diese Variablen konsistent gehalten. Die gemeinsam genutzten Variablen, die konsistent gehalten werden, nennt man auch *bewacht*.

Ein verteilter gemeinsam genutzter Speicher ist dann Release-konsistent, wenn er folgenden Regeln gehorcht:

- Bevor ein Zugriff (Read/Write) auf eine gemeinsam benutzte Variable ausgeführt wird, müssen alle zuvor abgesetzten Acquires erfolgreich abgeschlossen sein.

- Bevor ein Release ausgeführt werden darf, müssen alle vorherigen Read- und Writevorgänge des Prozesses abgeschlossen sein.

- Die Acquire- und Release-Zugriffe müssen dem Modell der sequentiellen Konsistenz folgen [17, 16, 19, 18]. In [30, 31] wird die sequentielle Konsistenz abgeschwächt und nur eine Prozessor-Konsistenz gefordert.

In Abbildung 2 wird eine gültige Ereignisabfolge für die Release-Konsistenz dargestellt. Prozess P1 setzt ein Acquire ab, verändert eine gemeinsam genutzte Variable zweimal und führt dann ein Release aus. Prozess P2 setzt ein Acquire ab und liest x. Dabei ist sichergestellt, dass der Wert von x zurückgegeben wird, den es zum Zeitpunkt des letzten Release hatte, nämlich 2. Wenn das Acquire vor P1's Release erfolgt, dann wird es verzögert, bis das Release erfolgt ist. Wenn P3 kein Acquire absetzt, bevor es eine gemeinsam genutzte Variable liest, ist der Speicher nicht verpflichtet, den aktuellen Wert von x zurückzugeben, so daß die Rückgabe von 1 erlaubt ist.

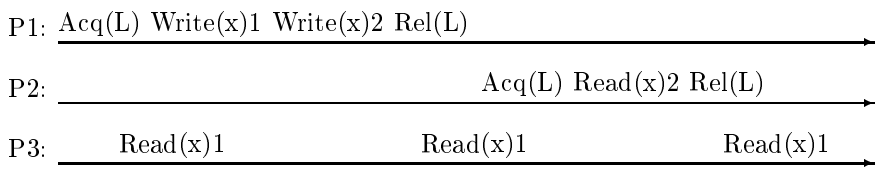

Abbildung 2: Eine gültige Ereignisabfolge für die Release-Konsistenz

Die Einschränkung der Software unter Verwendung aller schwachen Konsistenzen besteht in der Verpflichtung der verschiedenen Prozesse, zwischen zwei Synchronisationspunkten stets nur auf disjunkte Daten schreibend zuzugreifen. Darauf hat der Programmierer zu achten.

Es wird zwischen der eifrigen Release-Konsistenz (ERC, *Eager Release Consisteny*) und der trägen Release-Konsistenz (LRC, *Lazy Release Consisteny*) unterschieden.

Eifrige Release-Konsistenz

Der Vertrag zwischen der Software und dem Speicher besagt, dass, wenn die Software ein Acquire absetzt, der Speicher sicherstellt, dass alle lokalen Kopien der bewachten Variablen aktualisiert werden, so dass sie gegebenenfalls mit den entfernt gelegenen Variablen konsistent sind. Wenn ein Release erfolgt, dann werden die bewachten Variablen, die verändert wurden, den anderen Maschinen mitgeteilt. Ein Acquire garantiert nicht, dass alle lokalen Veränderungen unmittelbar allen anderen Maschinen mitgeteilt werden. Analog müssen bei einem Release nicht unbedingt die Veränderungen auf anderen Maschinen zur Kenntnis genommen werden.

Träge Release-Konsistenz

Das Modell der trägen Release-Konsistenz besagt, dass lokale Modifikationen einer Seite zwischen Acquire und Release nicht bei der Releaseoperation global propagiert werden. Dies erspart unnötigen Kommunikationsaufwand, da nicht alle Prozesse, die eine Kopie der Seite besitzen, diese Modifikationen auch verwenden wollen. Statt dessen versucht der Prozess bei einem Acquire, den aktuellen Wert der Variablen von den Maschinen, auf denen er sich befindet, zu erhalten. Um festzustellen, welche Variablen übertragen werden müssen, wird ein Zeitstempel-Protokoll benutzt. Bei einem Acquire werden alle *vorherigen* Veränderungen der Seite erfragt. Der Term *vorherig* ist im Sinne der *Happened-before-once*-Relation zu interpretieren.

Die Happened-before-once Relation

Sie beschreibt die partielle Ordnung, beschrieben mit $\xrightarrow{hb1}$, mit der Synchronisationsoperationen, wie Acquire, Release und Speicherzugriffe geordnet sind:

- Wenn a1 und a2 Acquire, Release oder Speicherzugriffe auf dem gleichen Prozessor sind, und a1 geschieht vor a2 in der Programmreihenfolge, dann ist a1 $\xrightarrow{hb1}$a2.
- Wenn a1 ein Release auf Prozessor p1 und a2 das korrespondierende Acquire auf Prozessor p2, dann ist a1 $\xrightarrow{hb1}$a2. Für einen Lock korrespondiert ein Acquire mit einem Release, wenn zwischen ihnen kein Acquire bzw. Release existiert. Für eine Barriere korrespondiert ein Acquire mit einem Release, wenn das Acquire und das Release der gleichen Barriereninstanz zugeordnet sind.
- Wenn a1 $\xrightarrow{hb1}$a2 und a2 $\xrightarrow{hb1}$a3 , dann a1 $\xrightarrow{hb1}$a3.

3.4 Eifrige vs. Träge Release-Konsistenz

Im direkten Vergleich wird die höhere Effizienz der trägen Release-Konsistenz gegenüber der eifrigen Release-Konsistenz deutlich.

Tabelle 1 stellt die Unterschiede der beiden Protokolle in knapper Form dar.

	ERC	**LRC**
Datenübergabe	bei Release	bei Acquire
Kommunikation	mit n-1 Prozessen	mit 1 Prozess
Effizienz	immer Verbreitung	nur wenn nötig

Tabelle 1: Vergleich ERC und LRC

Die Datenübergabe erfolgt bei der eifrigen Release-Konsistenz durch die Verbreitung der eigenen Änderungen bei Ausführung einer Release-Operation. Dabei wird mit allen Prozessen kommuniziert, die eine Kopie der Speicherseite besitzen. Es existiert keine Unterscheidung, ob die aktualisierten Kopien jemals benötigt werden.

Mit dem Protokoll der trägen Release-Konsistenz werden bei Aufruf der Acquire-Operation alle Änderungen bis zu diesem Zeitpunkt beschafft. Es wird nur mit dem einen Prozess, der das letzte Release durchführte, kommuniziert. Dadurch wird implizit die Effizienz erhöht, da nur bei beabsichtigter Benutzung der betreffenden Speicherseite über vergangene Modifikationen informiert wird.

Es sei dabei auf die zu beachtenden Einschränkungen des Programmierers in Kapitel 3.3 auf Seite 12 verwiesen.

Gravierende Unterschiede zwischen ERC und LRC werden besonders in dem folgenden Beispiel einer schlecht programmierten Schleife mit einem Acquire-Datenzugriff-Release-Tupel im Schleifenrumpf deutlich (siehe Abbildung 3). Viele Prozesse schreiben wiederholt unabhängig voneinander auf getrennte Speicherbereiche. Solange die Prozesse sich in dieser Schleife befinden, wäre kein Informationsaustausch untereinander nötig.

Bei Existenz des ERC-Modells müssen bei jedem Release die modifizierten Variablen an die anderen Prozessoren weitergegeben werden. Dies verursacht unnötige Bandbreitenverschwendung.

```
P[i]: i = 1..n
    Solange ( Bedingungsvariable ) {
      Acquire(L)
      x[i] = ...;    // Write(x[i])
      Release(L)
    }
```

Abbildung 3: Eifrige vs. Träge Release-Konsistenz: Beispiel

Mit Hilfe des LRC-Modells wird bei Release nichts ausgeführt. Bei dem folgenden Acquire wird festgestellt, dass die aktuellen Daten schon vorhanden sind und es werden keine Nachrichten erzeugt. Wiederholte Acquire-Release-Paare des gleichen Prozessors sind bei fehlender Konkurrenz von außen kostenlos.[6, 7]

4 Gegenseitiger Ausschluß

Gegenseitiger Ausschluß[29] ist ein wichtiges Konzept, um den Zugriff auf gemeinsam genutzte Ressourcen in verteilten Systemen zu schützen. Eine Anzahl von Prozessen synchronisiert ihren Zugriff auf gemeinsame Daten durch den Eintritt in eine kritische Sektion. Die verteilten Algorithmen müssen, um korrekt zu sein, den gegenseitigen Ausschluß (mutual exclusion) garantieren, frei von Verklemmungen sein (deadlock free) und das Verhungern einzelner Prozesse ausschließen (absence of starvation).

Nachteile strengerer Konsistenzmodelle

In verteilten Systemen sind sie in der Hauptsache über die Performance des Programmes zu spüren. Ein wichtiger Faktor bezüglich der Geschwindigkeit ist die Anzahl verschickter Nachrichten. Im Vergleich zum hardwaremäßigen direktem Zugriff ist das Senden und Empfangen von Nachrichten bzw. Signalen erheblich langsamer. Die durch die Definition neuer Konsistenzmodelle eingesparten Nachrichten erreichen den größten Geschwindigkeitsgewinn.

Ein Problem der sequentiellen Konsistenz ist in Abbildung 4 ersichtlich.

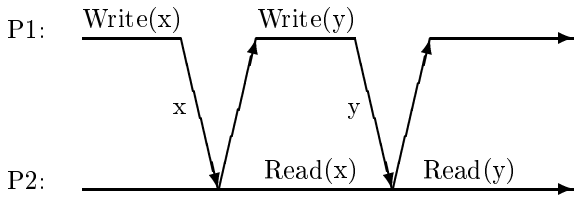

Abbildung 4: Performance-Problem: Sequentielle Konsistenz

Jeder Schreibzugriff auf den verteilten gemeinsam genutzten Speicher ist *sofort* allen anderen Prozessen sichtbar. Durch den Versand zu vieler, eigentlich unnötiger Nachrichten und der daraus folgenden Latenzzeit ergibt sich ein hoher Zeitverbrauch. In diesem Beispiel liest der Prozess P2 diese Variablen, aber P2 wäre auch von den Veränderungen informiert worden, wenn er keinerlei Read-Operationen auf den Variablen durchgeführt hätte.

Ein wichtiger Design-Aspekt unter der Nutzung strengerer Konsistenzmodelle liegt in der Wahl der Speicherabschnittsgröße, die eine Seite darstellt. Die Verwendung von größeren Einheiten für DSM hat Vor- und Nachteile.

Der größte Vorteil ist, dass es nicht sehr viel länger dauert, größere Abschnitte zu übertragen - mehr Zeit wird benötigt für die Einrichtung einer Netzwerkübertragung. Die Übertragung von Daten in großen Einheiten reduziert die Anzahl der Transfervorgänge. Diese Eigenschaft ist wesentlich, weil viele Programme ein lokales Zugriffsmuster aufweisen, d.h., wenn ein Programm auf ein Wort einer Seite zugegriffen hat, dann ist es wahrscheinlich, daß es in allernächster Zukunft auf weitere Worte derselben Seite zugreifen wird.

Andererseits führt eine zu große effektive Seitengröße zu einem neuen Problem, der sogenannten *falschen gemeinsamen Nutzung*[4], wie in Abbildung 5 dargestellt.

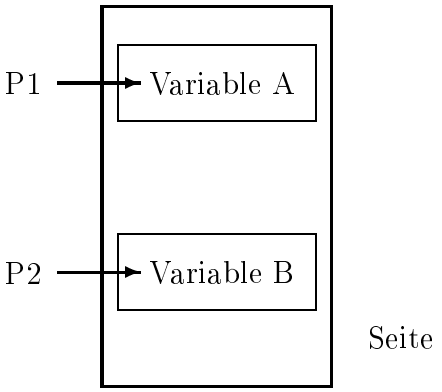

Abbildung 5: Performance-Problem: Falsche gemeinsame Nutzung

Es existiert eine Seite, die zwei nicht voneinander abhängige gemeinsam genutzte Variablen A und B enthält. Prozess P1 führt häufig Zugriffe auf A durch, wobei er die Speicherstelle liest und beschreibt. Analog verwendet Prozess P2 die Variable B. Unter diesen Umständen wird die Seite, die die beiden Variablen enthält, ständig zwischen den beiden Maschinen hin und her geschoben.

Das Problem dabei ist, dass die Variablen zwar nicht voneinander abhängig sind, aber weil sie sich zufällig auf derselben Seite befinden, erhält ein Prozeß, der eine davon anfordert, auch die andere. Je größer die effektiven Seiten sind, desto öfter tritt die falsche gemeinsame Nutzung auf, und je kleiner die effektiven Seiten sind, desto weniger tritt sie auf. Bei normalen virtuellen Speichersystemen gibt es nichts, was mit diesem Problem vergleichbar wäre.

Intelligente Compiler erkennen das Problem und platzieren die Variablen so im Adreßraum, dass sie die falsche gemeinsame Nutzung verhindern und

die Performance verbessern können. Wenn die falsche gemeinsame Nutzung jedoch darin besteht, dass Prozess P1 ein Element aus einem Array und Prozess P2 ein anderes Element aus demselben Array benutzt, dann kann selbst ein intelligenter Compiler wenig gegen dieses Problem unternehmen.

Die Implementierung des gegenseitigen Auschlusses unter Benutzung des trägen Release-Konsistenzmodelles ist Grundlage dieser Arbeit. Das Ziel ist die Verringerung der Kommunikation der Prozesse untereinander unter Beibehaltung des verteilten Speichermodelles. Unter Verwendung des Release-Konsistenzmodelles ist die Wahl der Granularität von Speicherabschnitten gegenstandslos. Zugriffe auf eine Speicherseite beinhalten Locks auf einen Teildatenbereich dieser Seite. Damit ist das MRMW-Protokoll[3] eigentlich erweitertes MRSW-Protokoll, mit dem der Programmierer darauf zu achten hat, nicht gleichzeitig zwei Teildatenbereiche einer Seite zu beschreiben. Näheres zu diesen Einschränkungen ist in Kapitel 3.3 auf Seite 12 zu finden.

Die Modellierung des gegenseitigen Ausschlusses

Mit Hilfe zweier Arten von Synchronisierungs-Variablen bzw. -Operationen wird eine effizientere Implementation des gegenseitigen Ausschlusses ermöglicht. *Acquire*-Zugriffe teilen dem Speichersystem mit, dass ein kritischer Abschnitt betreten werden soll. *Release*-Zugriffe besagen, dass ein kritischer Bereich gerade verlassen wurde.

Ein Acquire beinhaltet einen Lock auf einen Teildatenbereich dieser Speicherseite. Beim Release wird dieser wieder freigegeben.

Ein Prozess P1 hat ein Acquire auf die Seite L mit einem Lock auf den Teildatenbereich x durchgeführt und ist noch nicht zum Release gekommen. Wenn jetzt ein zweiter Prozess P2 ein Acquire auf die gleiche Seite und auf den gleichen Teildatenbereich ausführt, bleibt er solange stehen, bis P1 das Release beendet hat. Danach erfragt P2 die Modifikationen der Seite von P1.

Bei folgender Situation (Abbildung 6) wird das Acquire von P2 solange verzögert, bis das Release von P1 durchgeführt wurde:

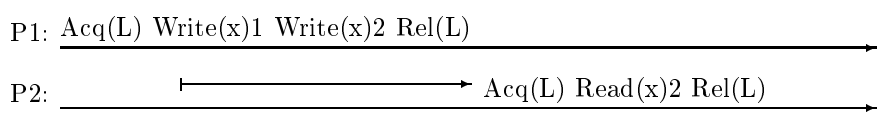

Abbildung 6: Acquire-Release-Modellierung eines geg. Ausschlusses

In der Realität werden sogar nur die Veränderungen des Teildatenbereichs x ausgetauscht.

Das MRMW-Protokoll

Das prinzipielle Vorgehen des MRMW-Protokolls erfolgt in der gleichen Reihenfolge.

- Jede Seite erhält ein Attribut, mit dem sie den Zustand UNMAPPED, READONLY , READWRITE oder INVALID annehmen kann.
- Zum Startzeitpunkt sind alle Seiten READONLY.
- Bei einem Schreibzugriff auf die Variable x (Write(x)) erfolgt eine Zugriffsverletzung, die DSM-Software erstellt eine Kopie der Seite und markiert diese als READWRITE (siehe Abbildung 7).

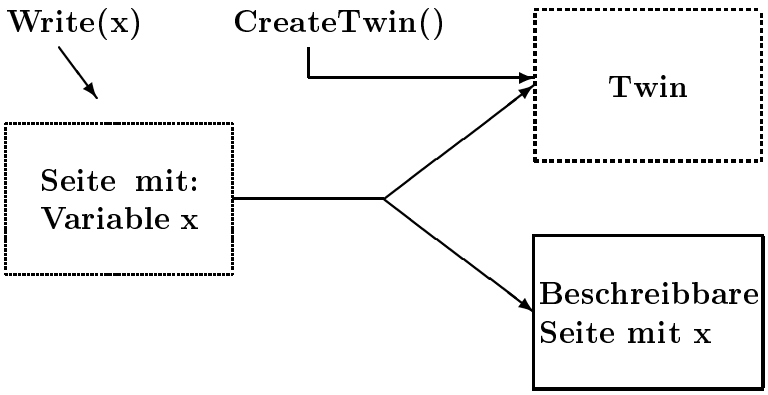

Abbildung 7: Erstellung eines Zwillings

Ein durchgezogener Rand einer Speicherseite der Abbildung 7 bedeutet, dass die Seite bescheibbar zur Verfügung steht. Gepunktete Seiten sind READONLY bzw. eigene Zwillinge, also nicht beschreibbare Seiten. Die Funktion *CreateTwin()* erstellt den Zwilling einer Seite und speichert ihn ab.

Alle folgenden Schreibzugriffe können ohne Eingriff der DSM-Software durchgeführt werden. Der Zugriff auf die jetzt lokal vorhandene Seite erfolgt hardwaremäßig, mit voller Geschwindigkeit.

Wie diese lokalen Veränderungen auf die Kopien der anderen Prozessoren übertragen werden, ist von der Verwendung des implementierten Release-Konsistenzmodelles abhängig.

Innerhalb der Release-Operation wird der Diff erstellt, der die Unterschiede zwischen Seite und Twin in komprimierter Form enthält. Dieser Vorgang ist in Abbildung 8 dargestellt.

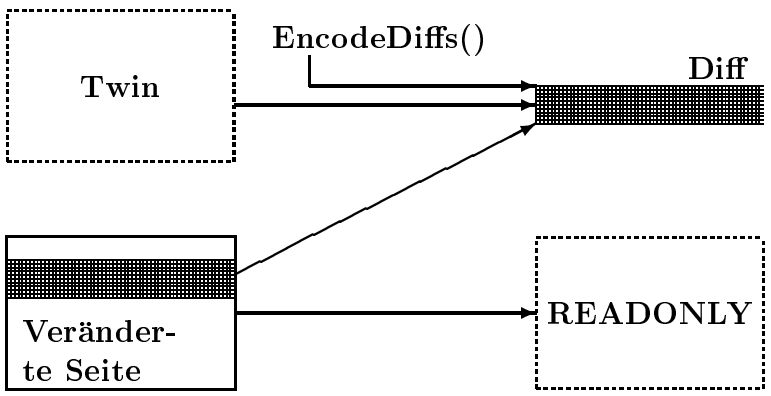

Abbildung 8: Erstellung eines Diffs

Diffs werden generiert, wenn

- Ein Prozessor die Modifikationen einer Seite erfragt
- Eine Schreibkopie von einem anderen Prozessor für eine Seite angefordert wird, die das Attribut READWRITE besitzt, d.h. sie ist modifiziert worden.
- Bei Entritt eines Prozesses in eine Barriere

Eifrige Invalidierung

Modifikationen des gemeinsamen Speichers werden sofort zum Zeitpunkt der Durchführung der Synchronisierungsoperation Release global sichtbar gemacht (siehe Abbildung 9). Dies geschieht unabhängig davon, ob die anderen Prozesse die Veränderungen bestimmter Speicherseiten überhaupt zur

Kenntnis nehmen wollen. Es wird versucht, alle existierenden Kopien einer modifizierten Seite zu invalidieren.

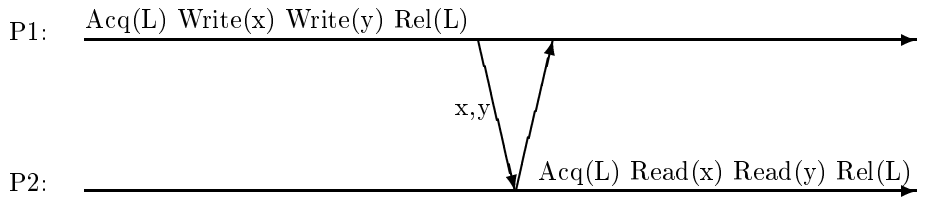

Abbildung 9: Eifrige Invalidierung

Besitzt die modifizierte Seite das Attribut

- READONLY, invalidieren
- READWRITE, die Kopie hängt eine Diff-Beschreibung seiner Modifikationen an die Antwortnachricht und invalidiert seine Kopie. Der empfangene Prozess fügt diese Diff-Beschreibungen in seine Kopie ein [16, 19].

Zur Identifizierung, wer eine Invalidierungsnachricht erhalten soll, wird eine ungefähre Copyset benutzt. Eine Copyset ist eine Menge, die beschreibt, welche Prozesse eine Kopie der Seite besitzen.

Träge Invalidierung

Modifikationen des gemeinsamen Speichers werden zum Zeitpunkt des *nachfolgenden* Acquire sichtbar gemacht. Der Prozess, der das Acquire durchführt, informiert sich über die *vorherigen* Modifikationen dieser Seite. Die Bedeutung der Terme *nachfolgenden* und *vorherig* enthält die *Happened-before-once*-Relation (siehe Kapitel 3.2 auf Seite 9).

Die Ausführung eines jeden Prozesses ist unterteilt in *Intervalle*. Das Intervall beschreibt die globale Zeit im Sinne der *passiert_vor*-Relation. Bei jedem Acquire bzw. Release beginnt ein neues Intervall. Intervalle von verschiedenen Prozessen sind partiell sortiert:

- Intervalle auf einem einzelnen Prozessor sind durch die Programmreihenfolge geordnet.

- Ein Intervall auf Prozessor p passiert vor einem Intervall auf Prozessor q, wenn das Acquire von q korrespondiert mit dem Release des Prozessors p.

Eine einzelner *Timestamp* besteht aus einer Prozessidentifikation und einem Intervall. Die Zeitmarke beschreibt einen Prozess auf einem über den Hostnamen identifizierbaren Rechner eindeutig. Der Prozess ist dort unter einem definierten Port zu erreichen. Seine logische Uhrzeit wird durch eine Intervallnummer bestimmt.

Die partielle Ordnung der Intervalle kann durch einen *Timestamp-Vektor* in jedem Intervall repräsentiert werden. Zur Konstruktion eines Timestamp-Vektors sind im Kapitel 6.1 auf Seite 29 ausführliche Informationen zu finden. Ein Timestamp-Vektor existiert einmal für jeden beteiligten Prozess. Er verwaltet die eigene Zeitmarke und enthält einen Eintrag für jeden Prozessor. Der Eintrag für Prozessor p im Intervall i ist gleich i. Der Eintrag für Prozessor $q \neq p$ beschreibt das größte Intervall des Prozessors q, dass vor dem jetzigen Intervall des Prozessors p, entsprechend der partiellen Ordnung, liegt. Dieser q-te Eintrag j ist so zu interpretieren, dass j das jüngste Intervall ist, in dem p ein Update von q erhalten hat. Ein Prozessor generiert bei einem Acquire einen neuen Timestamp-Vektor , indem er das paarweise Maximum seines vorherigen Timestamp-Vektors und dem Release-Timestamp-Vektor des Fremdprozessors ermittelt. Danach sendet er q seinen Timestamp-Vektor , damit er von q alle Modifikationen, die er noch nicht kennt, die aber q bekannt sind, erhält.

Implementation

Der Schreibzugriff auf eine invalidierte Seite verursacht eine Zugriffsverletzung. Der Prozessor p ermittelt den letzten Releaser dieses Teildatenbereichs. Existiert kein solcher Prozess (bei Programmstart) bzw. ist der Prozessor selbst der letzte Releaser, bekommt der fragende Prozess die Schreibrechte und kann mit seiner Programmabarbeitung fortfahren.

Ist ein fremder Prozessor q als der letzte Releaser erkannt, wird der gesamte eigene Timestamp-Vektor zu diesem Prozeß q geschickt. Dies geschieht *piggybacked* auf dem Rücken der Nachricht, die das Schreibtoken[29] auf dieser Seite verlangt. Mit Übersendung des Tokens schickt q alle Diffs, die Prozess p laut seinem Timestamp-Vektor unbekannt sind (siehe Abbildung 10). Eine Zeitmarke ist unbekannt, wenn sich im eigenen Timestamp-Vektor

keine Zeitmarke mit gleichem Hostnamen und gleichem Port befindet bzw. bei Übereinstimmung die Intervallnummer niedriger als die der fremden Zeitmarke ist. Der Prozessor p erhält das Token und fügt alle ihm unbekannten Diffs jüngerer Intervalle ein. Damit ist seine Speicherseite bezüglich dieses Teildatenbereichs aktualisiert und konsistent. Es wird ersichtlich, dass die Diffanforderung, die in Abbildung 10 als Pfeil dargestellt ist, aus zwei Nachrichten besteht.

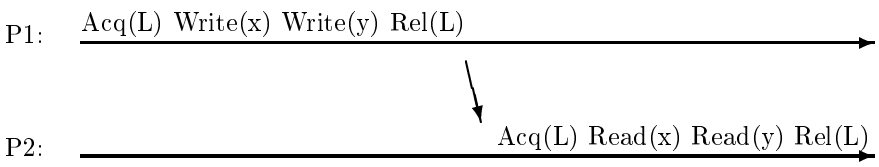

Abbildung 10: Träge Invalidierung

5 Barrieren

Eine Barriere kann mit Hilfe der Acquire-Release-Intervalle modelliert werden.

Barrieren sind Synchronisierungs-Mechanismen, der einen Prozess daran hindern, Phase *n+1* eines Programmes zu beginnen, bis alle Prozesse Phase *n* beendet haben. Wenn ein Prozess an eine Barriere kommt, dann muß er warten, bis alle anderen Prozesse auch dort angekommen sind. Wenn der letzte Prozess die Barriere erreicht, werden alle gemeinsam genutzten Variablen synchronisiert. Erst danach werden alle Prozesse fortgesetzt. Die Ankunft an der Grenze bedeutet *Release*, ihr Verlassen bedeutet *Acquire*.

Das Modell einer Barriere

In Abbildung 11 wird ein einziger zentralisierter *Barrieren-Manager* benutzt, der alle Ankunfts- und Abfahrtsmessages sammelt.

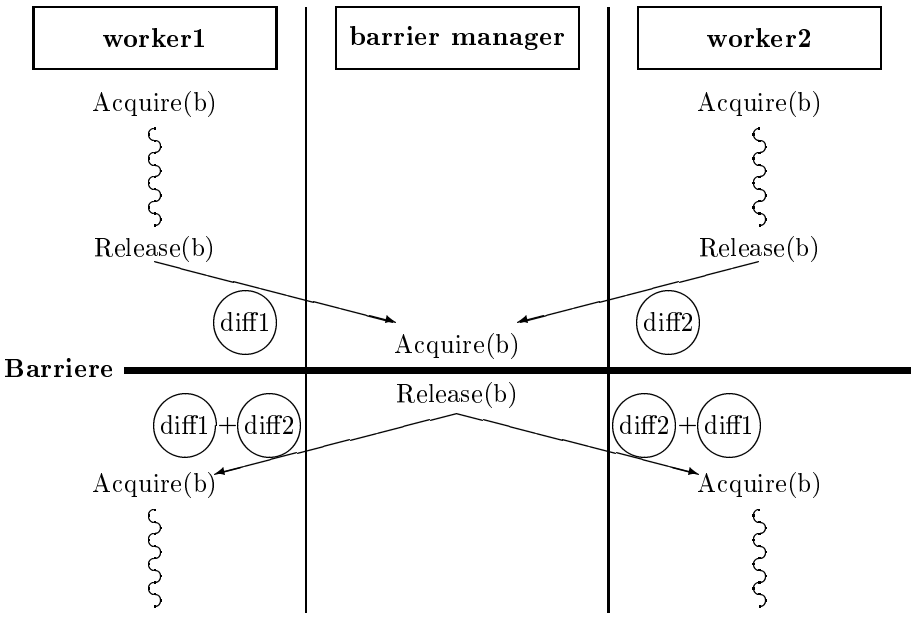

Abbildung 11: Acquire-Release-Modellierung einer Barriere

Alle Prozesse führten initial ein Acquire auf die Barriere b durch. Das implizite Lock auf die Barriere ist als Multi-Writer-Lock definiert. Bei Ankunft an die Barriere wird das Release(b) aufgerufen. Nach dem letzten Prozess setzt der Barrieren-Manager ein Acquire(b) ab. Dadurch erfragt er von allen Worker-Prozessen die Teildatenbereiche, die diese im letzten Intervall verändert haben und generiert einen Satz aktueller Seiten. Nach seinem Release(b) erfragen alle Worker-Prozesse (durch ein eigenes Acquire(b)) diese Modifikationen. Nach Verlassen der Barriere besitzen alle Prozesse einen konsistenten Speicher, wie er mit dem Protokoll der sequentiellen Konsistenz aussehen würde. Die Timestamp-Vektoren werden ebenso aktualisiert.

Die Implementation einer Barriere

Im Gegensatz zu diesem theoretischen Modell wird im bestehenden DSM-System[27, 28] eine verteilte baumstrukturierte Barriere verwendet[6, 7] (siehe Abbildung 12).

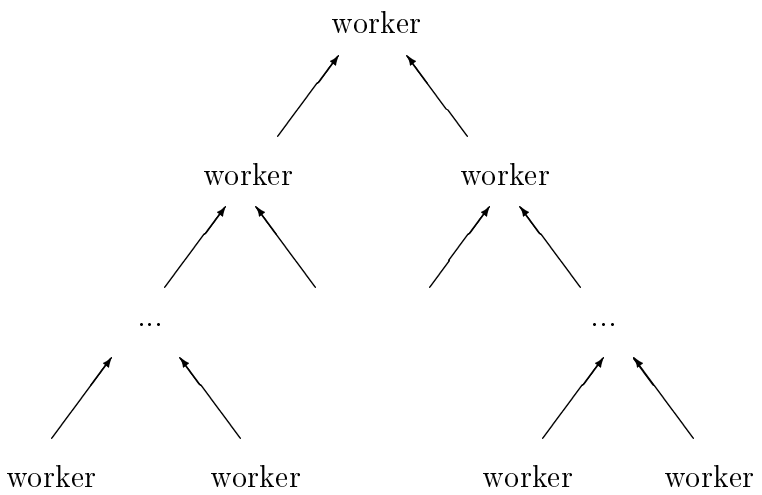

Abbildung 12: Eine baumstrukturierte Barriere

Sie vermeidet den Flaschenhals der Kommunikation mit einem zentralisierten Barrierenmanager. In einer Baumbarriere wird die Arbeit des Mana-

gers auf alle Knoten verteilt. Alle Worker-Threads organisieren sich in einem Baum wie in Abbildung 12 gezeigt. Der erste an der Barriere ankommende Knoten bildet die Wurzel des Baumes. Alle weiteren Knoten, die die Barriere erreichen, sortieren sich als logische Kindknoten innerhalb des binären Baumes an.

Es werden zwei Arten von Nachrichten unterschieden. Betreten die Knoten die Barriere, senden sie ein Signal an ihren Vaterknoten, also im Baum empor. Erhalten sie ein Antwortsignal von ihrem Vaterknoten, verlassen sie die Barriere und arbeiten weiter.

Der Ablauf in einer Baumbarriere

Bevor ein Knoten seine Ankunftsnachricht an seinen Elternknoten sendet, wartet er erst auf das Ankunftssignal seiner beiden Kindknoten(Abbildung 13). Da dieser auch erst auf das Eintreffen seiner Kinder wartet, startet erst der zuletzt ankommende Knoten die Nachrichtenkette im Baum. Alle im Baum befindlichen Knoten senden das Signal an ihren jeweiligen Vaterknoten. Wenn die Wurzel des Baumes die Signale seiner beiden Kindknoten erhält und selbst an der Barierre ist, sind alle Prozesse an der Barriere beteiligt(Abbildung 14).

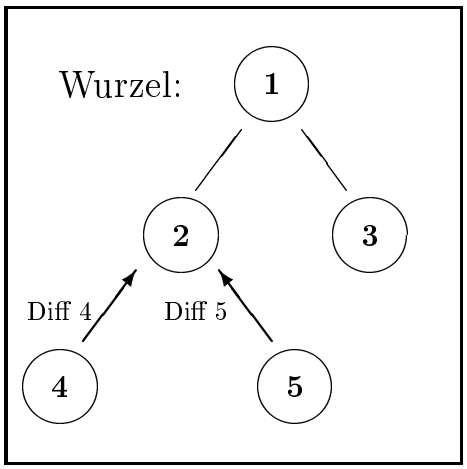

Abbildung 13: Ankunft an der Barriere

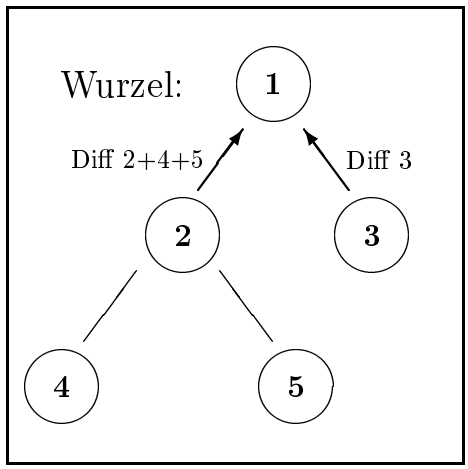

Abbildung 14: Weiterleitung der Diffs

Sie schickt daraufhin das Signal für das Weiterarbeiten an ihre beiden Kindknoten und verläßt die Barriere. Diese schicken die Abfahrts-Nachricht

weiter zu ihren Kindern, bis alle Prozesse die Barriere verlassen haben (siehe Abbildung 15). In Abhängigkeit von der verwendeten Hardware kann dazu auch ein eventuell effizienterer *Broadcast* verwendet werden.

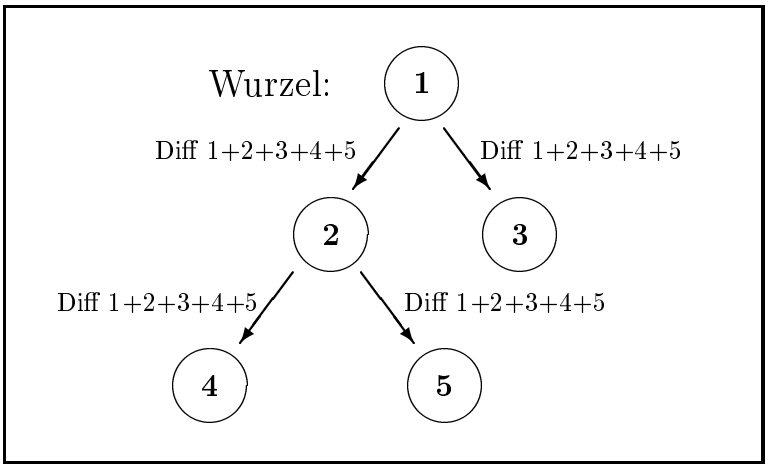

Abbildung 15: Abfahrts-Nachricht aus der Barriere

Die Effizienz einer Baumbarriere

Die Tiefe des Baumes beträgt $\lceil log_2 n \rceil$. Er ist deshalb für große n sehr effizient.

Die Größe einer Nachricht beeinflußt nicht die Performance des verteilten DSM-Systems. Die zentrale Idee bei der Implementation der trägen Release-Konsistenzmodelles ist es, die nötigen Aktualisierungsnachrichten an die Ankunfts- bzw. Abfahrts-Nachrichten anzuhängen. Damit könnte der gemeinsame verteilte Speicher ohne Performanceverlust konsistent gemacht werden. Unter der Voraussetzung, dass alle Knoten an einer Baumbarriere teilnehmen, wären nach der Barriere alle Speicherseiten aktualisiert.

Tatsächlich sind die Benutzung aller Nachrichten in beide Richtungen nötig. Jeder ankommende Knoten packt die von ihm erzeugten Veränderungen aller seiner Intervalle in einen Diff-String. Damit ist die Eigenschaft der Prozessor-Konsistenz zugesichert. Im Anhang der Ankunftsnachricht schickt er seine Diffs zu seinem Vaterknoten im Baum. Dieser ergänzt den Anhang um die von ihm generierten Diffs, ohne den Inhalt der empfangenen Diff-Zeichenkette zu dekodieren.

Nach Erhalt der Ankunftssignale besitzt die Wurzel des gesamten Baumes die Diffs aller an der Barriere beteiligten Knoten. Sie aktualisiert ihre eigenen Speicherseiten und hängt den großen Diff-String an die Abfahrts-Nachricht an. Jeder Knoten im binären Baum führt schrittweise in der Reihenfolge der Intervalle jedes Prozesses die Speicherveränderungen aus, damit die verlangte Prozessor-Konsistenz eingehalten wird. Nachdem alle Knoten die Barriere verlassen haben, ist der Speicher konsistent.

Um Zählerüberläufe zu vermeiden, werden nach der Barriere alle Intervalle neu initialisiert. Ebenso können die erzeugten Twins und Diffs aller Prozesse dealloziiert werden. Dies ist nur unter der Bedingungung möglich, dass alle Prozesse an der Barierre teilnehmen.

6 Veränderungen am DSM-System

Für die Implementation des MRMW-Protokolles wurden die Module *Timestamp* und *Release* erzeugt und die Module *Diff, Twin, Pagefunc, Pagetab* und *Barrier* verändert. An dieser Stelle werden die erzeugten bzw. veränderten Module näher vorgestellt, zu denen umfangreichere Programmierarbeiten nötig waren.

Im Verlauf der Software-Entwicklung stellte sich das Problem der internen Kodierung der Zeitmarken und Diffstrings. Eine Darstellung in Strukturform (*struct*) würde einen aufwandslosen Zugriff auf die Memberdaten garantieren. Damit wäre z.B. ein schneller Intervallvergleich zwischen zwei Zeitmarken möglich. Bestimmte Einträge in Difflisten könnten effizienter bearbeitet werden. Demgegenüber steht die Bedingung, dass Diffstrings und Zeitmarken sehr oft zu anderen Prozessen verschickt werden müssen. Dazu ist bei einer Darstellung in Strukturform eine Kodierungsfunktion nötig, die die gesamte Struktur zum Senden in Stringform ver- bzw. bei Erhalt einer empfangenen Nachricht entpackt.

Durch die eindeutige Ineffizienz dieser Funktion wurde die Darstellung in Strukturform verworfen und alle Zeitmarken und Diffstrings in Stringform kodiert. Der Versand- und Empfangsaufwand reduzierte sich gegen Null. Der relativ aufwändige Zugriff wurde mit den beiden effizient programmierten Konvertierungsfunktionen *dsm_Char2LongInt()* und *dsm_LongInt2char()* in erträglichen Maßen gehalten. Im direkten Vergleich zwischen beiden Darstellungsformen zeigte sich diese Kodierung als gleichwertig effizient und programmiertechnisch einfacher.

6.1 Das Modul Timestamp

Lamport stellte schon 1979 den klassischen Algorithmus vor, der zeigte, dass die Uhrsynchronisation in verteilten Systemen möglich ist. Er legte dar, dass die Uhrsynchronisation nicht absolut sein muß, sondern die Beachtung einer Ereignisreihenfolge als Synchronisationsalgorithmus ausreichend sein kann. Wenn zwei Prozesse miteinander interagieren, dann ist es nicht notwendig, dass ihre Uhren synchronisiert werden, weil das Fehlen der Synchronisation nicht auffällt und somit keine Probleme verursacht. Die Uhren werden dann als *logische* Uhren bezeichnet.

Um logische Uhren zu beschreiben, definierte Lamport[20, 21, 22, 23] eine Relation *passiert_vor*. Lamports Algorithmus folgt direkt aus der *passiert_vor*-Relation. Er gibt eine Methode an, allen Ereignissen in einem verteilten System eine Zeit zuzuordnen, die bestimmten Bedingungen genügt. Mehr zu diesem Thema findet sich in Kapitel 3.1 auf Seite 7.

Das Modul Timestamp implementiert die logischen Uhren für das Modell der trägen Release-Konsistenz im bestehenden DSM-System.

Eine einzelne Zeitmarke

Ein einzelner Timestamp beschreibt einen Prozess auf einem über den Hostnamen identifizierbaren Rechner eindeutig. Der Prozess ist dort unter einem definierten Port zu erreichen. Seine logische Uhrzeit wird durch eine Intervallnummer bestimmt.

Das Memberdatum Intervall beschreibt die globale Zeit im Sinne der *passiert_vor*-Relation. Beim Senden einer Nachricht wird das eigene Intervall inkrementiert.

In den folgenden Abschnitten werden die Begriffe Zeitmarke und Timestamp synonym verwendet werden.

Abbildung 16 zeigt den internen Aufbau einer Zeitmarke .

Abbildung 16: Struktur eines Timestamps

In Abbildung 17 wird beispielhaft der Timestamp des Prozesses aufge-

zeigt, der auf Rechner *localhost* unter Port *1234* erreichbar ist und sich im momentanen Intervall *5* befindet.

21	'localhost'	1234	5

Abbildung 17: Beispiel einer Zeitmarke

Der Speicherplatzbedarf des Hostnamens ist seiner Größe angepasst. Für die Dezimalzahlen wird ein Platz von vier Bytes reserviert. Sie werden mit Hilfe der beiden effizienten Konvertierungsfunktionen *dsm_Char2LongInt()* und *dsm_LongInt2char()* erzeugt bzw. ausgelesen. Zu diesem Thema wird auf Kapitel 6 auf Seite 28 verwiesen.

Ein Timestamp-Vektor

Ein Timestamp-Vektor dient der Implementation einer logischen Uhr in einem verteilten System. Er existiert einmal für jeden beteiligten Prozess. Er verwaltet die eigene Zeitmarke und alle empfangenen Timestamps der anderen Prozesse. Ein fremder Timestamp wird dann in den eigenen Timestamp-Vektor aufgenommen, wenn die eigenen Kopien der Speicherseiten mit den Speicherveränderungen, die der fremde Prozess geschickt hat, aktualisiert wurden. Im globalen Timestamp-Vektor ist also die Information über die erfolgte Prozesssynchronisation direkt erkennbar.

Abbildung 18 zeigt den internen Aufbau eines Timestamp-Vektors .

Länge	#TS	TS-Länge	Host	Port	Intervall	...

Abbildung 18: Struktur eines Timestamp-Vektors

Der erste Eintrag definiert die Länge des gesamten Timestamp-Vektors in Zeichen. Die zweite Dezimalzahl beschreibt die Anzahl von angehängten kodierten Zeitmarken . Die erste Zeitmarke definiert den eigenen globalen

Timestamp . Alle folgenden Einträge (symbolisiert mit ...) lassen die vergangene Kommunikation mit fremden Prozessen erkennen.

Im Beispiel der Abbildung 19 wird der Timestamp-Vektor des Prozesses beschrieben, der auf Rechner *localhost* unter Port *1234* erreichbar ist, sich im momentanen Intervall *5* befindet und insgesamt *3* Timestamps besitzt.

76	3	21	'localhost'	1234	5	17	'worf'	6666	5	...

Abbildung 19: Beispiel eines Timestamp-Vektors

Ausser der eigenen Zeitmarke sind noch noch zwei Timestamps fremder Prozesse kodiert. Die jeweilige Intervallnummer des Fremdprozesses zeigt, dass dem Besitzer dieses Timestamp-Vektors alle Speicheränderungen von *worf:6666* bis einschließlich Intervall *5* bekannt sind, d.h. dessen Speicherschreibvorgänge bis zu diesem Intervall in den eigenen Kopien aktualisiert wurden.

Zwei Zeitmarken des gleichen Rechners (Hostnamens) und unterschiedlichen Ports sind voneinander verschieden. Dies trifft zu, falls sich mehrere Threads auf einem Computer befinden und unter jeweils anderen Portadressen erreichbar sind. Im logischen Sinne sind sie als verschiedene Knoten eines verteilten Systems zu interpretieren.

Die Funktion 'void timestamp_vector_init()'

Diese Funktion alloziiert einen Timestamp-Vektor , der mit initialen Werten belegt ist (siehe Abbildung 20). Die erste Zeitmarke beinhaltet die Kodierung des Besitzerprozesses: den eigenen Hostnamen und Port, sowie das Intervall Eins. Alle anderen Prozesse sind zu diesem Zeitpunkt noch nicht bekannt.

21	'localhost'	1234	1

Abbildung 20: Initialer Zustand eines Timestamp-Vektors

Zum Aufbau eines Timestamp-Vektors sei auf das Kapitel 6.1 auf Seite 30 verwiesen. Die Kodierung der eigenen Eigenschaften ist relativ aufwändig. Demgegenüber wird diese Funktion nur einmal von jedem beteiligten Prozess aufgerufen. Die Funktion *void timestamp_vector_init()* muß nicht zwangsläufig aufgerufen werden, die meisten der Funktionen dieses Moduls testen vor ihrer Ausführung die Existenz des globalen Timestamp-Vektors und erzeugen ihn bei Bedarf.

Die Funktion 'void timestamp_vector_destroy()'

Diese Prozedur zerstört den eigenen globalen Timestamp-Vektor . Diese Routine ist für eine zufriedenstellende Speicherfreigabe am Ende des Programmes explizit aufzurufen.

Die Funktion 'char* timestamp_vector_get()'

Die Funktionen *char* timestamp_vector_get()* und *char* timestamp_get()* geben den eigenen Timestamp-Vektor bzw. Timestamp an den Aufrufer zurück. Sie testen, ob der Timestamp-Vektor , der Besitzer dieser Zeitmarke ist, alloziiert wurde. Im Falle der Nichtexistenz wird er durch den Aufruf von *timestamp_vector_init()* neu erzeugt. Die zurückgegebene Ergebnis-Zeitmarke ist ein direkter Pointer auf den eigenen globalen Timestamp-Vektor . Dies ermöglicht eine effiziente Ausführung der Funktionen. Der Ergebnispointer darf also weder vorher alloziiert noch nach Aufruf der Funktionen dealloziiert werden. Die Funktion *char* timestamp_vector_length()* gibt bei Bedarf die Stringlänge des Timestamp-Vektors zurück.

Die Funktion 'void timestamp_new_interval()'

Die Routine inkrementiert die Intervallnummer des eigenen Prozesses um Eins. Dies geschieht im Falle einer Acquire- bzw. Releaseoperation. Auch hier erfolgt ein Existenztest und eine eventuelle Initialisierung des Timestamp-Vektors .

Die Funktion 'int timestamp_know(char* stamp, char *tsv)'

Ihr wird eine Zeitmarke und ein Timestamp-Vektor übergeben. Es werden alle Timestamps des übergebenen Vektors mit der Zeitmarke verglichen. Eine Zeitmarke ist bekannt, wenn sich im Vektor ein Timestamp mit gleichem

Hostnamen, gleicher Portnummer und geringerem Intervall befindet. In diesem Fall ist der Rückgabewert Eins, sonst Null.

Die Funktion 'int timestamp_add (timestamp_t *stamp)'

Die Funktion fügt eine Zeitmarke in den globalen Timestamp-Vektor ein. Es sind zwei mögliche Fälle zu unterscheiden. Wenn der Timestamp *bekannt* ist (s.o.), so wird die Funktion mit dem Rückgabewert Null beendet. Nur unbekannte Zeitmarken werden in den Timestamp-Vektor aufgenommen. Der Returnwert von Eins zeigt diese Eigenschaft auf (in diesem Falle benötigt der Prozess die noch unbekannten Diffs).

DEBUG-Funktionalität

Für DEBUG-Sitzungen gedacht sind folgende Funktionen, die TRACE-Ausschriften ihrer Objekte aufzeigen: *int dump_timestamp(char* stamp)* und *void dump_timestamp_vector(char* titel, char* _tsv)*.

6.2 Das Modul Diff

In dem Modell der Release-Konsistenz werden die Synchronisationsoperationen Acquire und Release unterschieden. Zusätzlich wird die Lebensdauer eines Prozesses in Intervalle unterteilt, die bei der Ausführung der Release-Operation inkrementiert werden.

Ein Schreibzugriff auf eine Datenstruktur einer Seite, die für die träge Release-Konsistenz markiert ist, löst die Ausführung der Acquire-Funktion auf der Speicherseite aus. Dabei wird diese Seite kopiert und als Twin abgespeichert (siehe Kapitel 6.3, Seite 39). Danach kann auf die Datenstrukturen dieser Seite beliebig zugegriffen werden. Innerhalb der Release-Operation wird ein Diff erstellt, der die Unterschiede zwischen Seite und Twin in komprimierter Form enthält. Alle Diffs der Seiten werden in einer globalen Liste gespeichert, da nur sequentielle Zugriffe darauf nötig sind. Zu der Struktur, in dem diese Diffs vorliegen, wird auf Kapitel 6 auf Seite 28 verwiesen.

Der Term *Diff* wird in diesem Dokument in mehreren Zusammenhängen verwendet. Ein Diff definiert den Unterschied zwischen einer Speicherseite und seinem erstellten Twin. Aber auch die entstandene Zeichenkette, die sämtliche Differenzen einer Speicherseite in kodierter Form enthält, wird als Diff bezeichnet. Semantisch sind beide Interpretationen äquivalent.

Als *Diff-String* wird die Zeichenkette bezeichnet, die einen oder mehrere Diffs inklusive einer Zeitmarke enthält. Sie kann zu anderen Prozessen verschickt werden.

Ein einzelner Seitendiff

Der Diff einer einzelnen Seite wird bei Anforderung eines Acquires von einem anderen Prozess erstellt. Er wird in Abbildung 21 dargestellt.

PageIndex	#Diffs	Off/Len	Diff	...	Off/Len	Diff

Abbildung 21: Struktur eines Seiten-Diffs

Das Attribut *Pageindex* definiert die Nummer der ursprünglichen Seite. Dann werden die Anzahl der in diesem String kodierten Diffs festgelegt (*#Diffs*). Alle folgenden Einträge wiederholen sich um diese Menge. Die

nächsten beiden Attribute (*Off/Len, Offset/Länge*) beschreiben die Anfangsposition des Diffs auf der Seite und seine Länge. Um Übertragungsmenge zu sparen, sind sie in ein Zahlenfeld kodiert. Dies beschränkt ihre Größe auf jeweils 32767. Indirekt wird damit die Seite auf diese Größe beschränkt, da das Attribut Offset keinen größeren Wert annehmen kann. Der eigentliche *Diff* beinhaltet die Veränderungen der Speicherseite an der Position *Off* mit der Länge *len*. Im Beispiel der Abbildung 22 wird der Diff der Seite *4* beschrieben, der *3* Diffs kodiert.

4	3	24/4	'abcd'	48/4	'wxyz'	96/8	'12345678'

Abbildung 22: Beispiel eines Seiten-Diffs

Der erste Diff von dieser Seite wurde an der Position *21* mit der Länge *4* erzeugt. Die Werte, die sich vorher an dieser Stelle befanden, wurden mit den Werten *'abcd'* überschrieben. In diesem Beispiel werden nur Zeichenketten zur besseren Darstellung eines Diffs benutzt. Die anderen beiden Diffs beschreiben die Offsets *122* und *200*. Die Offsets und Längen aller Diffs sind Vielfache von Vier. Dies ist durch die Kodierungsfunktionen bedingt, die möglichst effizient arbeitet.

Ein Diff-String

Diese Zeichenkette beinhaltet alle Seitendiffs, die ein Prozess in einem definierten Intervall erzeugt hat. Er besteht aus einer Kette von Einzeldiffs.

Die Abbildung 23 zeigt den Aufbau eines solchen Diff-Strings.

Stringlänge	Zeitmarke	#Diffs	Diff1	...	Diffn

Abbildung 23: Struktur eines Diffs

Der erste Eintrag *Stringlänge* definiert die Gesamtlänge der Zeichenkette. Die Zuordnung des Diff-Strings zu einem Prozess und einer Intervallnummer

erfolgt durch die *Zeitmarke* . Der Timestamp besitzt die Attribute HOST, PORT und INTERVALL. Der Parameter *#Diffs* legt die Anzahl der folgenden Seitendiffs fest, die nach obigen Schema aufgebaut sind.

Die Abbildung 24 beschreibt den Diff der Länge *28*, der von dem Prozess erstellt wurde, der auf dem Computer *'worf'* unter der Portadresse *1234* erreichbar ist.

28	'worf'	1234	5	1	4	1	128/4	'wxyz'

Abbildung 24: Beispiel eines Diffs

Er befand sich zum Zeitpunkt der Erzeugung dieser Zeichenkette im Intervall *5*. Es ist *1* einzelner Diff kodiert (s.o.). Von Seite *4* wurde *1* Zeichenkette von Position *128* der Länge *4* mit dem Inhalt *'wxyz'* erzeugt.

Die Funktion 'void dsm_InitDiffs()'

Diese Routine initialisiert alle Listenstrukturen. Es existieren zwei verschiedene globale Listen für jeden Thread: *current_diff_list* und *old_diff_list*.

In der Liste *current_diff_list* werden alle Differenzen abgespeichert, die der Thread selbst erzeugt hat. Der kodierte Timestamp in ihnen entspricht der thread-eigenen Zeitmarke mit geringerer Intervallnummer. Diese Diffs werden bei einem Mutex und innerhalb einer Barriere in die zu versendende Nachricht kodiert.

In die Liste *old_diff_list* werden alle Diffs abgespeichert, die der Thread von anderen Prozessen innerhalb eingegangener Nachricht empfangen und dekodiert hat. Diese Diffs werden bei einem Mutex in die zu versendende Nachricht kodiert. Sie dürfen aus Effizienzgründen nicht bei einer Barriere verwendet werden. Mit der Teilnahme von n Prozessen an einer Barriere teilen alle Threads ihre eigenen Diffs den anderen teilnehmenden Prozessen ihre eigenen Differenzen mit. Nach Austritt aus der Barriere besitzten n Prozesse alle Informationen der anderen Barrierenmitglieder. Es ist nicht nötig, die Zeichenketten der Liste *old_diff_list* zu übertragen.

Die Funktion 'int dsm_CreateDiff(long index, char **diff, int *len)'

Die Routine bekommt eine Seitennummer *index* als Parameter. Sie alloziiert eine Ergebniszeichenkette *diff*, vergleicht die Seite mit ihrem Zwilling und packt die Unterschiede nach obigen Format in die Zeichenkette. Nach Ausführung dieser Operationen definiert der Parameter *difflen* die Länge von *diff*. Diese Funktion wird im DSM-System von dem Protokoll der *Bereichs-Konsistenz* direkt benutzt.

Die Funktion 'char *dsm_ApplyDiff(long index, char *diff)'

Diese Operation besitzt die inverse Funktionalität der obigen Routine. Sie bekommt eine Seitennummer *index* und eine Zeichenkette *diff* als Parameter. Sie wendet den Diff auf die entsprechende Seite an, d.h. sie aktualisiert ihre Seitenkopie mit den Veränderungen des fremden Prozesses. Diese Funktion wird im DSM-System von dem Protokoll der *Bereichs-Konsistenz* direkt benutzt.

Die Funktion 'int dsm_CreateAllDiffs()'

Sie durchsucht alle von dem eigenen Prozess erzeugten Twins (siehe Kapitel 6.3, Seite 39). Für jede veränderte Seite (d.h. es existiert ein Unterschied zwischen einer Seite und ihrem Twin) wird ein Diff einer einzelnen Seite nach obigem Format erzeugt. Anschließend werden alle Einzeldiffs zu einem Diff-String zusammengepackt und mit einem Header versehen (*Timestamp, #Pages*, s.o.). Der resultierende Diff-String wird in der eigenen globalen Liste abgespeichert. Diese Funktion wird i.a. am Ende eines Intervalles aufgerufen.

Die Funktion 'int dsm_encode_diffs(char **diff, char* tsv_string)'

Die Routine erhält als Parameter eine Zeichenkette *diff* und einen Timestamp-Vektor *tsv_string*. Sie alloziiert und belegt einen Diff-String, der alle dem Timestamp-Vektor *tsv_string* unbekannten Diffs enthält. Zu der Definition eines *unbekannten Diffs* sei auf das Kapitel 6.1 auf der Seite 29 verwiesen. Ist beim Aufruf dieser Funktion der Parameter *tsv_string* gleich NULL, werden alle besitzenden Diffs in den Ergebnisstring kodiert. Sollte die übergebene Zeichenkette *diff* nicht leer sein, so wird angenommen, dass er bereits kodierte Diffs enthält. Er wird in diesem Falle realloziiert und nur die eige-

nen (von diesem Prozess erzeugten) Diffs hinten in die Zeichenkette kodiert. Diese beiden Eigenschaften sind bei einer Barriere nötig.

Die Funktion 'void dsm_decode_and_add_diffs(char *diffstream)'

Die Parameterzeichenkette *diffstream* kodiert eine Diff-Struktur. Die Funktion ermittelt den jeweiligen Timestamp und testet, ob ihr dieser Diff *unbekannt* (s.o.) ist. Trifft diese Aussage zu, werden alle zugehörigen Einzeldiffs dekodiert, die zugehörigen Speicherseiten aktualisiert und der Diff-String in der globalen Liste abgespeichert.

Die Funktion *void dsm_destroy_DiffList()* Diese Operation zerstört alle bestehenden Diffs und die Difflisten *current_diff_list*, *old_diff_list* und *barrier_diff_list*. Diese Routine ist für eine zufriedenstellende Speicherfreigabe am Ende des Programmes explizit aufzurufen.

DEBUG-Funktionalität

Für DEBUG-Sitzungen gedacht sind folgende Funktionen, die TRACE-Ausschriften ihrer besitzenden Objekte in Textform aufzeigt: *void dump_DiffList(char* titel)* und *void dump_diffstring(char* titel, char *string)*.

6.3 Das Modul Twin

Die Funktion 'int dsm_CreateTwin(long index)'

Der Parameter *index* definiert den Index einer Speicherseite, über den mittels *dsm_GetPage(index)* ein direkter Zugriff möglich ist. Die Twins werden mit Hilfe von Listen verwaltet, da nur ein sequentieller Zugriff benötigt wird. Die entsprechende Seite wird kopiert und die Kopie gesichert. Die Ausführung dieser Funktion setzt voraus, dass die entsprechende Seite bereits gelockt ist.

Die Funktion 'void dsm_DeleteTwin(long index)'

Die Routine zerstört den über den Parameter *index* definierten Zwilling.

DEBUG-Funktionalität

Für DEBUG-Sitzungen gedacht sind folgende Funktionen, die TRACE-Ausschriften ihrer besitzenden Strukturen aufzeigt: *void dsm_DumpPage(char* titel, long index)* und *void dsm_DumpTwin(char* titel, long index)*.

6.4 Das Modul Release

Das Modul Release stellt die Schnittstelle zwischen DSM-System und dem trägen Release-Konsistenzmodell. Sie bietet alle Funktionen, die von der DSM-Software benötigt werden, um das MRMW-Protokoll nutzen zu können.

Alle Operationen dieses Moduls sind von einem lokalen Mutex umhüllt. Dieser verhindert die mißbräuchliche zeitgleiche Nutzung globaler Listenstrukturen bzw. Funktionen zweier Threads auf einem gemeinsamen Knoten.

Die Funktion 'void dsm_ReleaseInit(long FirstReleasePage)'

Diese Operation initialisiert die eigenen Listenstrukturen und den lokalen Mutex. Sie ist zwingend aufzurufen. Sie führt die initialen Funktionsaufrufe der Module *Timestamp* und *Diff* auf.

Die Funktion 'void dsm_SaveTwin(long TwinIndex)'

Die Routine speichert den Zwilling einer Seite in der eigenen globalen Liste ab. Der Twin wird solange in der Liste aufgehoben, bis das Programm beendet bzw. ein Diff von diesem Twin und der zugehörigen Speicherseite durchgeführt wird. Dann wird der Twin ersatzlos zerstört.

Die Funktion 'void dsm_CloseInterval()'

Sie wird bei Ende eines Intervalles aufgerufen. Der Intervallzähler wird inkrementiert. Zusätzlich werden alle Diffs des Intervalles erzeugt und abgespeichert.

Ein Intervall wird beendet bei:

- Diffanforderung eines anderen Prozesses
- Diffempfang eines anderen Prozesses
- Eintritt in eine Barriere

Die Funktion 'int dsm_SendInterval(char **diff, char* tsv_string)'

Die Prozedur ruft direkt die Funktion *int dsm_encode_diffs(char **diff, char* tsv_string)* des Moduls *Diff* auf. Dazu sei auf das Kapitel 6.2 auf Seite 34 verwiesen. Es werden alle dem Parameter *tsv_string* unbekannten Diffs in einen Diff-String *diff* kodiert.

Die Funktion 'void dsm_ReceiveInterval(char *diff)'

Die Parameterzeichenkette *diff* definiert einen solchen Diff-String. Das eigene Intervall wird mittels der Funktion *void dsm_CloseInterval()* beendet (Intervallerhöhung und Differstellung). Danach werden die im Parameter kodierten Diffs entpackt und die entsprechenden Speicherseiten aktualisiert. Dies entspricht dem direkten Aufruf der Funktion *void dsm_decode_and_add_diffs(char *diffstream)* im Modul *Diff*.

6.5 Die Aufrufhierarchie

Die zentrale Rolle des Modules *release* als Schnittstelle zur DSM-Software ist in Abbildung 25 deutlich erkennbar. Die Richtung der Pfeile entspricht den *#include*-Anweisungen im Quelltext.

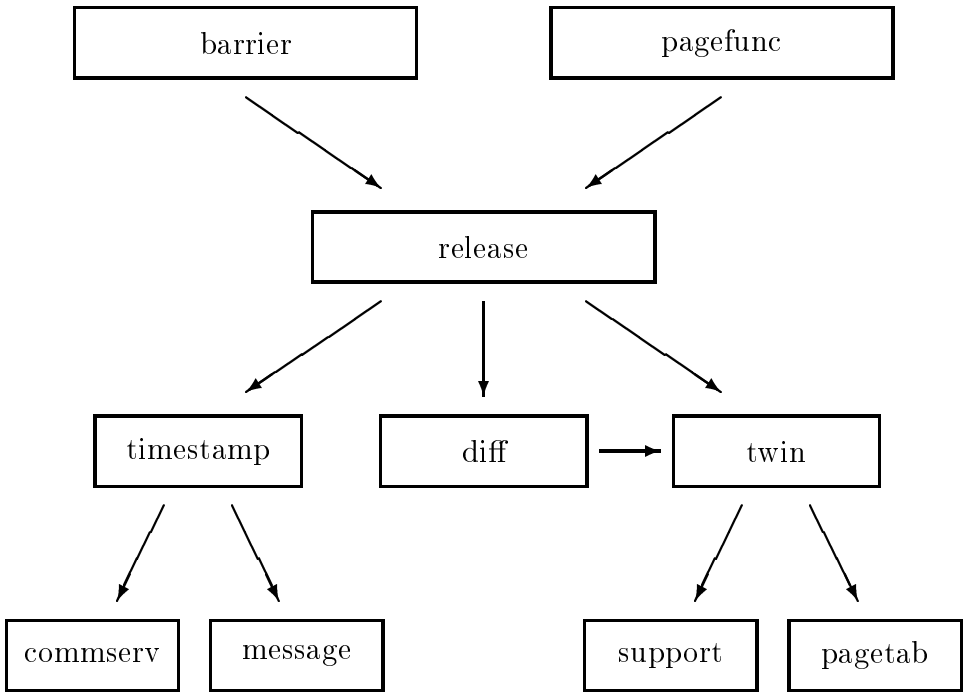

Abbildung 25: Die Aufrufhierarchie der Module

Das Modul *commserv* wird für die Funktionen *char* dsm_selfhost()* und *int dsm_selfport()* benötigt. Sie liefern den Namen des Computers, auf dem sich der ausführende Prozess befindet und die Portadresse, unter der der Prozess erreichbar ist. Beide Angaben dienen der Identifizierung eines Prozesses. Mehr Angaben dazu sind unter Kapitel 6.1 auf Seite 29.

Im Modul *message* sind die beiden Funktionen *dsm_Char2LongInt()* und *dsm_LongInt2char()* definiert. Sie werden für die Konvertierungen von Dezimalzahlen in Zeichenketten und umgekehrt benutzt. Zu diesem Thema sind im Kapitel 6 auf Seite 28 mehr Informationen.

Das Modul *support* unterstützt die umfangreiche Funktionalität zum Abspeichern beliebiger Datenstrukturen in sortierten Listen. Mit Hilfe dieses

Moduls werden alle *Diffs* und *Twins* abgespeichert.

Die Speicherseiten des DSM-Systems werden im Modul *pagetab* verwaltet. Innerhalb der Datenstruktur befindet sich ein Pointer, der für die Aufnahme des Zwillings einer Seite reserviert ist. Der direkte Zugriff auf die Seiten (Status, Access, Lock) wird über verschiedenste Funktionen unterstützt. Ob und welche Speicherseite über das träge Konsistenzmodell verwaltet wird, definieren *int dsm_isRelease(void)* und *int dsm_isDSMRelease(char *addr)*.

7 Schlußwort / Ausblick

Gegenstand dieser Arbeit war eine Erweiterung der Funktionalität des bestehenden DSM-Systems [27, 28] um das MRMW-Protokoll. Dies wurde durch die Benutzung des trägen Release-Konsistenzmodelles erreicht. Es wurden der gegenseitige Ausschluß und Barrieren mit Hilfe des MRMW-Protokolles implementiert [29, 3].

Eine erfolgte Änderung am bestehenden DSM-System war die Erzeugung eines Moduls, welches Zeitstempel verwalten kann. Zusätzlich wurden die Module, die Diffs und Twins verwalten, den neuen Erfordernissen angepaßt.

Für die Implementation des MRMW-Protokolles wurden die Module *Timestamp* und *Release* erzeugt und die Module *Diff, Twin, Pagefunc, Pagetab* und *Barrier* verändert. Diese Arbeiten sind bereits vollständig abgeschlossen.

Nicht behandelt wurden alle Aspekte einer zufriedenstellenden Speicherfreigabe. Im Besonderen besteht das Problem der Dealloziierung der erzeugten Diffs und die Nichtbeachtung eventueller Zählerüberläufe. Unter der Voraussetzung, dass alle Prozesse an einer Barriere teilnehmen, könnte dort die gesamte Garbage Collection durchgeführt werden. Eine zukünftige Aufgabe ist die Verknüpfung des MRMW-Protokolles mit anderen Protokollen.

Anschließend wird dem Vergleich des MRMW-Protokolles mit bereits bestehenden Protokollen besondere Aufmerksamkeit gewidmet werden. Dies wird insbesondere das MRSW-Protokoll betreffen. Es werden umfangreiche Effektivitätstests durchgeführt werden.

Diese Einschränkungen sollen im Rahmen einer anschließenden Diplomarbeit untersucht und beseitigt werden.

Abbildungsverzeichnis

1 Zugriffstypen auf den Speicher bei verteilten Anwendungen . . 6
2 Eine gültige Ereignisabfolge für die Release-Konsistenz 11
3 Eifrige vs. Träge Release-Konsistenz: Beispiel 14
4 Performance-Problem: Sequentielle Konsistenz 15
5 Performance-Problem: Falsche gemeinsame Nutzung 16
6 Acquire-Release-Modellierung eines geg. Ausschlusses 17
7 Erstellung eines Zwillings 18
8 Erstellung eines Diffs 19
9 Eifrige Invalidierung 20
10 Träge Invalidierung 22
11 Acquire-Release-Modellierung einer Barriere 23
12 Eine baumstrukturierte Barriere 24
13 Ankunft an der Barriere 25
14 Weiterleitung der Diffs 25
15 Abfahrts-Nachricht aus der Barriere 26
16 Struktur eines Timestamps 29
17 Beispiel einer Zeitmarke 30
18 Struktur eines Timestamp-Vektors 30
19 Beispiel eines Timestamp-Vektors 31
20 Initialer Zustand eines Timestamp-Vektors 31
21 Struktur eines Seiten-Diffs 34
22 Beispiel eines Seiten-Diffs 35
23 Struktur eines Diffs 35
24 Beispiel eines Diffs 36
25 Die Aufrufhierarchie der Module 42

Tabellenverzeichnis

1 Vergleich ERC und LRC 13

Literatur

[1] S. V. Adve, A. L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. A comparison of entry consistency and lazy release consistency implementations. Technical Report TX 77005-1892, Dept. of CS, Rice University, 1995.

[2] S. V. Adve and M. D. Hill. Weak ordering—A new definition. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)*, pages 2–14, 1990.

[3] C. Amza, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Software dsm protocols that adapt between single writer and multiple writer. Technical report, Dept. of CS, Rice University, University of Rochester, 1995.

[4] C. Amza, A. L. Cox, K. Rajamani, and W. Zwaenepoel. Tradeoffs between false sharing and aggregation in software distributed shared memory. Technical report, Dept. of CS, Rice University, 1997.

[5] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, February 1996.

[6] G. R. Andrews. *Concurrent Programming – Principles and Practice.* Benjamin Cummings, 1991.

[7] G. R. Andrews and R. A. Olsson. *The SR Programming Language – Concurrency in Practice.* Benjamin Cummings, 1993.

[8] B. Bershad, M. Zekauskas, and W. Sawdon. The midway distributed shared memory system. In *COMPCON Conference Proceedings*, pages 528–537, 1993.

[9] N. Bershad and M. Zekausas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. TR CMU-CS-91-170, Carnegie-Mellon Univ., 1991.

[10] M. Dubios and C. Scheurich. Memory access dependencies in shared-memory multiprocessors. *IEEE Trans. Software Engineering*, 16(6):660–673, June 1990.

[11] M. Dubios, C. Scheurich, and F. Briggs. Synchronization, coherence, and event ordering in multiprocessors. *IEEE Computer*, 21(2):9–21, February 1988.

[12] K. Gharachorloo, D. Lenosik, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *International Symposium on Computer Architecture*, pages 15–26, 1990.

[13] J. R. Goodman and W.-C. Hsu. Code scheduling and register allocation in large basic blocks. In *Intl. Conference on Supercomputing*, pages 442–452, July 1988.

[14] P. Hutto and M. Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *Distributed Computing Systems*, pages 302–311, 1990.

[15] Liviu Iftode, Jaswinder Pal Singh, and Kai Li. Scope consistency: a bridge between release consistency and entry consistency. In *Symposium on Parallel Algorithms and Architectures*, pages 277–287, June 1996.

[16] P. Keleher. *Lazy Release Consistency for Distributed Shared Memory.* PhD thesis, Department of Computer Science, Rice University, December 1994.

[17] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *International Symposium on Computer Architecture*, pages 13–21, 1992.

[18] P. Keleher and C. Tseng. Enhancing software dsm for compiler-parallelized applications. In *International Parallel Processing Symposium*, pages 490–499, 1997.

[19] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. An evaluation of software-based release consistent protocols. *J. Parallel Distrib. Comput.*, 29:126–141, September 1995.

[20] L. Lamport. Time, clocks and ordering of events in distributed systems. *Comm. ACM*, 21(7):558–565, June 1978.

[21] L. Lamport. How to make a multiprocessor that correctly executes multiprocess programs. *ACM Trans. Comput. Systems*, 28:690–691, September 1979.

[22] L. Lamport. Concurrent reading and writing of clocks. In *ACM Trans. Comput. Systems*, pages 8(3):305–310, November 1990.

[23] L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. Technical report, Digital Equipment Corporation, Systems Research Center, February 1993.

[24] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Systems*, 7(4):321–359, November 1989.

[25] K. Li and P. R. Hudak. Memory coherence in shared virtual memory systems. In *Proceedings 1986 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239, New York, NY, 1986. ACM.

[26] R. Lipton and J. Sandberg. Pram: A scalable shared memory. Technical Report CS-TR-180-88, Princeton Univ., 1988.

[27] F. Mueller. Distributed shared-memory threads: DSM-threads. In *Workshop on Run-Time Systems for Parallel Programming*, pages 31–40, April 1997.

[28] F. Mueller. On the design and implementation of DSM-threads. In *Proc. 1997 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 315–324, June 1997. (invited).

[29] F. Mueller. Prioritized token-based mutual exclusion for distributed systems. In *International Parallel Processing Symposium*, pages 791–795, 1998.

[30] A. S. Tanenbaum. *Distributed Operating System.* Prentice-Hall, 1995.

[31] A. S. Tanenbaum. *Verteilte Betriebssysteme.* Prentice-Hall, 1995.