Diplomarbeit

Ein relationales Datenbanksystem für die SB-PRAM

Concurrency-Control und der TPC-B–Benchmark

Christian Jacobi

Universität des Saarlandes
Fachbereich 14 — Informatik
Lehrstuhl für Rechnerarchitektur und Parallelrechner
Prof. Dr. W. J. Paul

Juni 1999

Eidesstattliche Erklärung
Hiermit erkläre ich an Eides Statt, daß ich die vorliegende Arbeit selbstständig verfaßt und nur die angegebenen Quellen benutzt habe. Ich habe diese Arbeit keinem anderen Prüfungsamt vorgelegt.

Saarbrücken, im Juni 1999

Christian Jacobi

Software is like entropy.
It is difficult to grasp,
weighs nothing,
and obeys The Second Law of Thermodynamics;
i.e., it always increases.

Danke

An dieser Stelle möchte ich allen danken, die zum Gelingen der vorliegenden Arbeit beigetragen haben.

Mein Dank gilt zunächst meinen Eltern, ohne deren Hilfe und Unterstützung mein Studium nicht möglich gewesen wäre.

Mein Dank gilt auch Herrn Prof. Paul für die Unterstützung während des gesamten Studiums.

Danken möchte ich auch meinen Freunden

- Cédric Lichtenau für die Betreuung dieser Arbeit;
- Michael Bosch, Michael Klein und Jochen Preiß für das Korrekturlesen der Arbeit sowie für die vielen Diskussionen, die zum Gelingen der Arbeit beitrugen;
- Peter Bach, Jörg Fischer, Stefan Kunde, Andreas Paul und Jochen Röhrig für das gute Klima im Labor sowie für das stets offene Ohr für Fragen;
- allen Mitarbeitern des Lehrstuhls Paul für das gute Arbeitsklima.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Datenbanksysteme stellen die wohl wichtigste Komponente beim Einsatz von Rechnersystemen in Wirtschaft und Verwaltung dar. Mit Datenbanksystemen ist die Verwaltung der dort anfallenden Datenmengen einfach und sicher möglich. Durchgesetzt hat sich das relationale Datenmodell [Cod70], wenn auch in den letzten Jahren die postrelationalen (Objekt-Orientierten) Datenbanksysteme [Sto96] an Bedeutung gewonnen haben.

Die zu verwaltenden Datenmengen sind in den letzten Jahren enorm gewachsen. Die Anforderungen an Datenbanksysteme sind daher in vielen Fällen so gestiegen, daß sie nur noch von großen Parallelrechnern mit mehreren Prozessoren erfüllt werden können. Parallelrechner lassen sich einteilen in

Shared–Nothing–Systeme: Jeder Prozessor besitzt einen eigenen Hauptspeicher und eigene Festplatten. Die Prozessoren können über ein Netzwerk miteinander kommunizieren.

Shared–Disk–Systeme: Jeder Prozessor besitzt einen eigenen Hauptspeicher; der Zugriff auf gemeinsame Festplatten ist jedem Prozessor möglich.

Shared–Everything-Systeme: Die Prozessoren teilen sich einen gemeinsamen Hauptspeicher und gemeinsame Festplatten.

Nach Bhide sind Shared–Everything–Systeme den restlichen Modellen überlegen [BS87, Bhi88]. Im Gegensatz dazu behaupten Gray und Reuter, daß sich das Shared–Nothing–Modell durchsetzen wird [GR93, Seite 59ff.]. Gray und Reuter betrachten dabei Realisierbarkeits–Aspekte und behaupten, daß sich nur Shared–Nothing–Systeme effizient realisieren lassen. Auch Bhide weist in [Bhi88] auf Realisierungs–Probleme für Shared–Everything–Systeme hin.

An der Universität Saarbrücken wurde dennoch ein Shared–Everything–System, die SB–PRAM, realisiert. Die SB–PRAM [AD+93] ist ein experimenteller Parallelrechner. Zum Zeitpunkt der Fertigstellung dieser Arbeit läuft ein Prototyp der SB–PRAM mit 16 Prozessoren. Eine SB–PRAM mit 64 Prozessoren und vier Gigabyte gemeinsamen Hauptspeicher befindet sich im Aufbau.

Simulationen nach scheint die SB–PRAM als Datenbankserver sehr geeignet zu sein. In [GJM+95] wurde der TPC-B–Benchmark [Rab92] simuliert — mit

überraschend guten Ergebnissen. Ziel der vorliegenden Arbeit war es, den TPC-B–Benchmark auf der SB–PRAM zu implementieren und auszuführen, um die Simulationsergebnisse zu überprüfen. Zur Zeit ist die Festplatten–Anbindung der SB–PRAM jedoch noch nicht vollständig implementiert, weswegen entgegen der ursprünglichen Planung der Benchmark nicht ausgeführt werden konnte. Das Datenbanksystem ist jedoch für den TPC-B–Benchmark vollständig implementiert; die Messergebnisse werden veröffentlich, sobald die Festplatten–Anbindung fertiggestellt ist (voraussichtlich Herbst 1999).

Bei der Implementierung des Datenbanksystems wird auf die Arbeit von Spengler [Spe97] aufgebaut. Dort sind die Basiskomponenten beschrieben, die für das Datenbanksystem der SB–PRAM nötig sind.

Eine der wichtigsten Komponenten des Datenbanksystems ist die *Concurrency-Control*. Diese Komponente regelt den Zugriff der Prozessoren auf die gemeinsamen Daten. Möchte ein Prozessor Daten lesen oder verändern, so muß er die Daten zuerst durch die Concurrency-Control sperren. Dadurch werden gleichzeitige Lese- und Schreib-Operationen auf den gleichen Daten vermieden.
In vielen Anwendungen kann die Concurrency-Control zum Flaschenhals des Datenbanksystems werden. Dies gilt insbesondere für massiv-parallele Architekturen wie die SB–PRAM. Daher wird in dieser Arbeit ein neuer Ansatz zur Implementierung der Concurrency-Control vorgestellt. Dieser Ansatz ist auf der SB–PRAM um einen Faktor 200 schneller als der übliche Ansatz, wie er zum Beispiel in Spenglers Arbeit [Spe97] verfolgt wird. Der Beschreibung und Analyse der neuen Concurrency-Control ist ein wesentlicher Teil dieser Arbeit gewidmet.

Die vorliegende Arbeit gliedert sich wie folgt: Im folgenden Kapitel wird der Begriff der Datenbank formal gefaßt und die Anforderungen an ein Datenbanksystem werden erläutert. Dabei werden auch die Datenbanksprache SQL und das Transaktionskonzept vorgestellt. Anschließend wird in Kapitel 3 die SB–PRAM vorgestellt.

In Kapitel 4 wird die Implementierung des Systempuffers vorgestellt. Dieser stellt die Verbindung zwischen Festplatten und den weiteren Komponenten des Datenbanksystems dar.

In Kapitel 5 wird dann die Concurrency–Control–Komponente inklusive einer Deadlock-Erkennung entwickelt. Die Korrektheit und Fairneß der Concurrency-Control und der Deadlock-Erkennung werden bewiesen. Eine Zusammenfassung dieses Kapitels ist für die Konferenz Euro-Par 99 angenommen und wird als [JL99] veröffentlicht.

In Kapitel 6 werden ein einfacher Parser für SQL-Befehle und der TPC-B–Benchmark vorgestellt. Es werden auch Ergebnisse einfacher Messungen gezeigt, die ohne die Festplatten–Anbindung auskommen. Die Ergebnisse dieser Arbeit werden abschließend in Kapitel 7 zusammengefaßt.

Die Quelltexte des Datenbanksystems sowie die Programmierumgebung der SB–PRAM sind verfügbar unter `ftp://ftp-wjp.cs.uni-sb.de/pub/pram`.

Kapitel 2

Relationales Datenmodell und grundlegende Begriffe

In diesem Kapitel soll formal definiert werden, was man unter einer relationalen Datenbank versteht. Dabei werden die Begriffe aus der Datenbank-Theorie erläutert, deren Kenntnis für die folgenden Kapitel unabdingbar sind. Nachdem der Begriff der Datenbank formal definiert wurde, werden die für diese Arbeit notwendigen Operationen auf Datenbanken erklärt. Im Anschluß wird die Sprache SQL vorgestellt, die eine einfache Formulierung dieser Operationen erlaubt. Abschließend wird der Begriff der Transaktion erläutert und darauf aufbauend eine Liste von Anforderungen an Datenbank-Management-Systeme (DBMS, DBS) gegeben.

2.1 Attribut, Tupel, Relation

Informell ist eine Relation gegeben durch eine zweidimensionale Tabelle bestehend aus einem Tabellenkopf und einem Tabellenrumpf (siehe Abbildung 2.1). Der Tabellenkopf gibt an, welche Bedeutung der Inhalt des Tabellenrumpfes hat.

A_1	A_2	...	A_n
t_1^1	t_2^1	...	t_n^1
t_1^2	t_2^2	...	t_n^2
$\vdots$			$\vdots$
t_1^m	t_2^m	...	t_n^m

Abbildung 2.1: Tabellarische Darstellung einer Relation

Definition 2.1 (Relationsschema) Sei $A = \{A_1, \ldots, A_n\}, n \geq 1$, eine endliche Menge, die sogenannten **Attribute**, und sei $D = \{D_1, \ldots, D_s\}, s \geq 1$, eine Menge von nichtleeren Mengen (sogenannte **Wertebereiche, Domains**). Weiterhin sei $dom : A \longrightarrow D$ eine Abbildung. Dann ist das **Relationsschema**

RS gegeben durch das Tupel

$$RS = (A, D, dom).$$

Ein Relationsschema entspricht dem Tabellenkopf aus der informellen Beschreibung einer Relation. Außer den Namen der Spalten wird durch das Relationsschema auch der Wertebereich[1] der einzelnen Spalten festgelegt. Die Zeilen heißen Tupel:

Definition 2.2 (Tupel) Ein **Tupel über dem Relationsschema** $RS = (A, D, dom)$ ist eine Funktion

$$t : A \to \bigcup_{i=1}^{n} dom(A_i)$$

mit

$$t.A_i := t_i := t(A_i) \in dom(A_i).$$

Der Wert $t.A_i$ wird **Ausprägung** von t auf dem Attribut A_i genannt. Synonym werden Tupel auch **Datensätze** oder **Records** genannt. Mit $d(RS) \cong \bigotimes_{i=1}^{n} dom(A_i)$ bezeichnen wir die Menge aller solcher Funktionen über RS.

Nun können wir definieren, was wir mit Relation bezeichnen:

Definition 2.3 (Relation) Sei RS ein Relationsschema. Dann ist eine **Relation R über RS** eine endliche Menge von Tupeln über RS. Insbesondere enthält eine Relation nur paarweise verschiedene Tupel.

Definition 2.4 (Datenbank) Eine **Datenbank** ist eine endliche Menge von Relationen. Die Tupel der Relationen einer Datenbank bezeichnet man als **Nutzdaten** oder einfach Daten. Alle weiteren Informationen zu einer Datenbank (zum Beispiel die Relationsschemata) bezeichnet man als **Metadaten**.

2.2 Operationen auf relationalen Daten

In diesem Abschnitt werden die für diese Arbeit wichtigen Operationen auf einer Datenbank definiert. Die Software, die diese Operationen ausführt, nennt man Datenbank-System (DBS). Eine Auflistung der Anforderungen an ein DBS wird in Abschnitt 2.4 gegeben.

Definition 2.5 (Projektion) Sei $RS = (A, D, dom)$ mit $A = \{A_1, \ldots A_n\}$ ein Relationsschema. Für $A' = \{A'_1, \ldots A'_{n'}\} \subset A$ ist das Relationsschema $RS_{|A'}$ definiert durch

$$RS_{|A'} := (A', D, dom_{|A'}).$$

[1] Auf der SB-PRAM sind die Wertebereiche der Typen *(unsigned) integer, float* und *string* erlaubt.

Die **Projektion von** RS **nach** $RS_{|A'}$ ist die Abbildung

$$\begin{aligned} p : d(RS) &\longrightarrow d(RS_{|A'}), \\ t &\longmapsto p(t) := t_{|A'}. \end{aligned}$$

Ist $R \subseteq d(RS)$ eine Relation über dem Relationsschema RS, so ist $R_{|A'} := p(R)$, das Bild von R unter p, eine Relation über $RS_{|A'}$. Man beachte, daß $R_{|A'}$ weniger Tupel als R enthält, wenn zwei in R verschiedene Tupel unter p das gleiche Bild haben.

Definition 2.6 (Selektion) Sei $RS = (A, D, dom)$ ein Relationsschema und R eine Relation über RS. Dann heißt für $a \in A, c \in dom(a)$

$$R_{a,c} := \{t \in R | t.a = c\}$$

die **Selektion**[2] **in** R **nach** $a = c$.

Definition 2.7 (Einfügen) Die Operation **Einfügen von** t **in** R fügt der Relation R über dem Relationsschema RS das Tupel $t \in d(RS) \setminus R$ hinzu:

$$R \longleftarrow R \cup \{t\}\,.$$

Definition 2.8 (Update (Änderung)) Sei R eine Relation über dem Relationsschema $RS = (A, D, dom)$ mit $A = \{A_1, \ldots, A_n\}$. Sei $U = \{a_1, \ldots a_s\} \subseteq A$ eine Teilmenge der Attribute und sei jedem a_i ein Ausdruck P_i in den Variablen A zugeordnet. Weiterhin sei $a \in A$ und $c \in dom(a)$. Dann ersetzt eine **Update-Operation mit den Parametern** $U, \{P_i\}, a, c$ alle Tupel $t \in R$ mit $t.a = c$ durch Tupel $u(t) \in d(RS)$ mit

$$u(t).A_i := \begin{cases} t.A_i & : \quad A_i \notin U \\ P_j(t) & : \quad A_i \in U \text{ und } A_i = a_j \end{cases}\,.$$

Dabei muß natürlich stets $P_j(t) \in dom(a_j)$ gelten. Welche Ausdrücke P_j erlaubt sind, wird im nächsten Abschnitt am Beispiel erläutert und in Abschnitt 6.1 definiert.
Insgesamt ergibt sich somit durch Updates eine neue Relation

$$R \longleftarrow (R \setminus R_{a,c}) \cup \{u(t) | t \in R_{a,c}\}\,.$$

Man beachte, daß wie bei der Projektion auch bei einem Update die Relation kleiner werden kann.

Nur der Vollständigkeit halber sei hier noch die Operation Löschen erwähnt. Sie wird im weiteren Verlauf dieser Arbeit nicht benötigt:

[2]Professionelle Datenbanksysteme erlauben dem Anwender wesentlich komplexere Selektionskriterien als den einfachen Vergleich. Zum Beispiel können auch die weiteren Vergleichsoperatoren "$\leq, <, \neq, >, \geq$" sowie logische Verknüpfungen benutzt werden. Für diese Arbeit reicht jedoch der einfache Selektionsbegriff aus.

Definition 2.9 (Löschen) Sei R eine Relation über $RS = (A, D, dom)$ und $a \in A, c \in dom(a)$. Die Operation **Löschen** löscht alle Tupel $t \in R$ mit $t.a = c$:

$$R \longleftarrow R \setminus R_{a,c}.$$

In der Datenbank-Theorie werden häufig diese Begriffe wie auch hier über Mengen definiert. Dies beinhaltet insbesondere die Duplikaten-Eliminierung bei den Operationen Projektion und Update. Bei Systemen, die die Verwaltung von Datenbanken und die oben definierten Operationen unterstützen, wird auf diese Duplikaten-Eliminierung aus Laufzeitgründen verzichtet. Man müßte also eigentlich obige Definitionen über Multimengen fassen.

2.3 SQL

SQL (Structured Query Language) ist eine Sprache, die unter anderem die Formulierung der im letzten Abschnitt dargestellten Datenbankoperationen erlaubt. Die Einfachheit und große Ausdruckskraft dieser Sprache hat dazu geführt, das fast alle Datenbanksysteme SQL unterstützen. In dieser Arbeit wird jedoch nur eine sehr kleine Teilmenge von SQL betrachtet. Unter anderem erlaubt SQL auch die Definition von Relationsschemata und weiterer Metainformationen einer Datenbank; dieser Teil von SQL wird in dieser Arbeit komplett übergangen. Eine ausführliche Darstellung von SQL findet man in [MS93].
Den in dieser Arbeit benutzten Teil von SQL macht man sich am besten durch einige einfache Beispiele klar. In Abschnitt 6.1 findet man eine Definition der Syntax des hier benutzten Teils von SQL.

SQL-Beispiele

Angenommen wir wollten einen kleinen Tante-Emma-Laden verwalten. Dazu benutzen wir eine Relation Artikel, die unter anderem die Attribute Name und Preis enthält. Da der Besitzer des Ladens nicht gerne mit Kleingeld hantiert, kann jeder Kunde einen Betrag auf sein eigenes Konto einzahlen und dann solange Einkäufe tätigen, bis sein Guthaben erschöpft ist. Dazu benutzen wir die Relation Kunden, die für jeden Kunden den Namen und sein aktuelles Guthaben enthält.
Eine interessante Frage für den Besitzer des Ladens ist es, welches Guthaben ein bestimmter Kunde (zum Beispiel Herr Lichtenau) hat. Diese Frage läßt sich in SQL wie folgt formulieren:

```
SELECT Guthaben INTO g FROM Kunden
    WHERE Name = "Lichtenau"
```

Bei der Abarbeitung dieser SQL-Anfrage wird zunächst die Selektion $\mathsf{Kunden}_{\mathsf{Name,\ Lichtenau}}$ berechnet. Auf die entstehende Relation wird dann die Projektion nach Guthaben berechnet und das Ergebnis in der Variablen g gespeichert.
Analog kann man mit einer SQL-Anfrage den Preis eines bestimmten Artikels ermitteln:

```
SELECT Preis INTO p FROM Artikel
    WHERE Name = "Schokoriegel"
```

Kauft nun Herr Lichtenau einen Schokoriegel, wird ihm der entsprechende Preis (von zum Beispiel 0,50 e) von seinem Guthaben abgezogen. Dies läßt sich in SQL als UPDATE-Befehl formulieren:

```
UPDATE Kunden SET Guthaben = Guthaben - 0.50
    WHERE Name = "Lichtenau"
```

Dies entspricht einer Update-Operation aus dem vorherigen Abschnitt mit den Parametern ({Guthaben}, {Guthaben-0.50}, Name, Lichtenau).
Wollte der Kaufmann die Artikel-Relation auch zur Inventur benutzen, so würden wir der Relation ein weiteres Attribut Anzahl hinzufügen, um den aktuellen Vorrat eines jeden Artikels zu speichern. Jedesmal, wenn ein Artikel gekauft wird, müßte dann die Relation Artikel aktualisiert werden:

```
UPDATE Artikel SET Anzahl = Anzahl - 1
    WHERE Name = "Schokoriegel"
```

Um einen neuen Kunden (zum Beispiel Herrn Preiß) mit einem Guthaben von 0 e in die Kundenrelation einzufügen, muß der Kaufmann einen INSERT-Befehl absetzen:

```
INSERT INTO Kunden (Name, Guthaben)
    VALUES ("Preiß", 0)
```

2.4 Der Transaktions-Begriff, Datenbanksysteme

In diesem Abschnitt soll der Begriff der "Transaktion" erläutert werden. Dieser Begriff ist zentral für Datenbank-Anwendungen. Eine wesentlich ausführlichere Darstellung dieses Begriffes, als das an dieser Stelle möglich ist, findet man in [GR93]. Nach der Einführung von Transaktionen ist es dann möglich, die Anforderungen an ein Datenbanksystem zu nennen.

Transaktionen

Die mit Abstand häufigste Verwendung von Datenbanken ist die Modellierung von real existierenden Zuständen. Jedem Zustandwechsel der realen Welt entspricht eine *Transaktion*, die den Zustandwechsel in der Datenbank vornimmt. Die im vorherigen Abschnitt als Beispiel gegebene Datenbank ist eine mögliche Modellierung eines Tante-Emma-Ladens. Als Modellierungs-Eigenschaft (Konsistenzbedingung) soll zu jedem Zeitpunkt gelten, daß

1. für jeden Artikel der in der Datenbank eingetragene Preis mit dem Preisschild des Artikels übereinstimmt;
2. die in der Datenbank gespeicherte Anzahl gleich der Anzahl der tatsächlich vorhandenen Artikel ist;
3. für jeden Kunden das in der Datenbank gespeicherte Guthaben gleich seinem tatsächlichen Guthaben[3] ist.

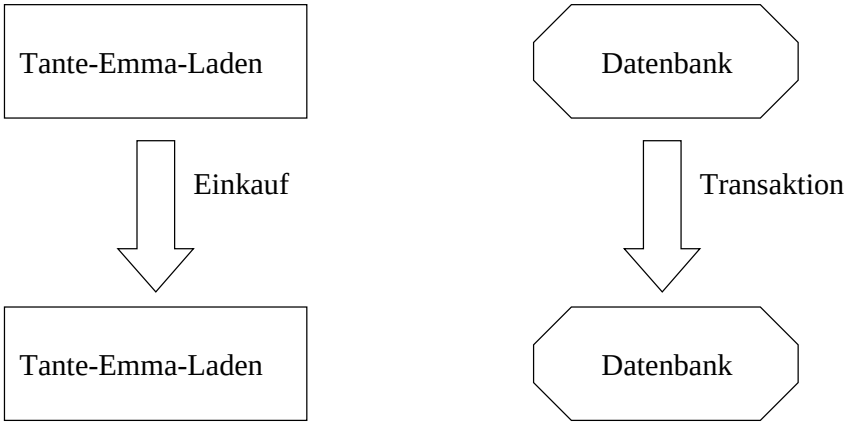

Abbildung 2.2: Der tatsächliche Zustand ist durch die Abstraktion "Datenbank" gegeben; ein Zustandswechsel (Einkauf) wird auf der Datenbankebene Transaktion genannt

Bei jedem Kaufvorgang ändert sich der Zustand des Tante-Emma-Ladens. Diese Änderung der realen Welt muß sich bei Einhaltung obiger Regeln in der Datenbank niederschlagen. Eine Folge von SQL-Anweisungen, die den realen Zustandwechsel auf die Datenbank überträgt, bezeichnet man als Transaktion (siehe Abbildung 2.2). In unserem Beispiel lautet die zu einem Kaufvorgang gehörende Transaktion in SQL:

```
defvar p float;
BEGIN WORK;                          %% Transaktion "Einkauf"
SELECT Preis INTO p FROM Artikel
      WHERE Name = <Artikelname>;
UPDATE Kunden SET Guthaben = Guthaben - p
      WHERE Name = <Kundenname>;
UPDATE Artikel SET Anzahl = Anzahl - 1
      WHERE Name = <Artikelname>;
COMMIT WORK
```

Dabei markieren BEGIN WORK bzw. COMMIT WORK den Beginn bzw. das Ende des Transaktionsblocks. Ein weiteres Beispiel für einen Zustandswechsel

[3]Man kann über die Existenz eines "tatsächlichen Guthabens" streiten; das soll jedoch nicht Thema dieser Arbeit sein. Dazu sei zum Beispiel auf *K. Marx, Das Kapital* verwiesen.

ist eine Preiserhöhung. Dabei werden die Preisschilder der Artikel gewechselt. Nach der Konsistenzbedingung 1 muß dann der in der Datenbank gespeicherte Preis des Artikels ebenfalls geändert werden. Dies ergibt folgende Transaktion:

```
BEGIN WORK;                    %% Transaktion "Preiserhöhung"
UPDATE Artikel SET Preis = Preis + <Delta>
      WHERE Name = <Artikelname>;
COMMIT WORK
```

Eine eingehendere Diskussion und Motivation des Transaktions-Begriffs findet man in [GR93, Kapitel 1]. Wir bezeichnen im folgenden eine Reihe von logisch zusammengehörigen SQL-Befehlen als Transaktion. Beim Modellieren einer Datenbank gehört es zu den wichtigsten Aufgaben, zu entscheiden, welche SQL-Befehle logisch zusammenhängend sind, also zu einer Transaktion gehören.

Datenbanksysteme

Ein Datenbankssystem ist eine Software, die es Benutzern erlaubt, Datenbanken zu definieren (administrieren) und SQL-Befehle in Form von Transaktionsblöcken auszuführen. Bei der Abarbeitung von Transaktionen muß das DBS das **ACID-Prinzip** [GR93] befolgen. Die Abkürzung "ACID" steht für

Atomicity Jede Transaktion wird ganz oder gar nicht ausgeführt. Im Falle eines Transaktionsabbruchs müssen alle von dieser Transaktion geänderten Daten wieder hergestellt werden — ein sogenannter *Rollback*. Ein Abbruch kann zum Beispiel durch einen Systemcrash oder einen *Deadlock* (siehe Abschnitt 5.5) verursacht werden. Auch bei Verletzung von Konsistenzbedingungen muß die Transaktion abgebrochen werden.

Consistency Jede Transaktion entspricht einem korrekten Zustandswechsel; die Konsistenzbedingungen werden durch die Transaktionen nicht verletzt. Dies setzt korrekte Transaktionsprogramme voraus.

Isolation Jede Transaktion wird so bearbeitet, daß aus Sicht des Benutzers alle anderen Transaktion entweder schon abgeschlossen oder noch nicht gestartet sind. Die Isolation-Eigenschaft wird auch mit *Serialisierbarkeit* bezeichnet.

Durability Die Änderungen einer korrekt beendeten Transaktion überstehen jeden Systemcrash.

Am Beispiel des Tante-Emma-Ladens läßt sich die Notwendigkeit des ACID-Prinzips einsehen. Wenn die Kauftransaktion nach dem ersten Update unterbrochen wird, wird zwar dem Kunden der Warenwert vom Guthaben abgezogen, die Inventuränderung findet jedoch nicht statt. Kauft der Kunde die Ware tatsächlich, stimmt der Anzahl-Eintrag in der Relation nicht mehr mit der Wirklichkeit überein. Findet der Kauf nicht statt, ist dem Kunden zu unrecht der Wert vom Guthaben abgezogen worden.

Als weitere Konsistenzbedingung könnte gelten, daß die in der Relation Artikel gespeicherten Anzahlen stets größer oder gleich Null sind. Das Datenbanksystem darf dann keine Transaktion zulassen, welche diese Konsistenzbedingung verletzt. Die Konsistenzbedingungen werden vom Administrator der Datenbank festgelegt und leiten sich häufig aus den Modellierungs-Eigenschaften ab.
Um die Notwendigkeit des Isolations-Prinzips zu verdeutlichen, nehmen wir an, daß der Preis von Schokoriegeln erhöht wird. Gleichzeitig kauft ein Kunde einen Schokoriegel. Dann ist nicht klar, welcher Preis dem Kunden abgezogen werden soll; in diesem Fall kann die Semantik der Select-Anweisung in Transaktion "Einkauf" nicht festgelegt werden. Wegen des Isolations-Prinzips wird die Transaktion "Preiserhöhung" entweder vor oder nach der Transaktion "Einkauf" bearbeitet, wodurch dieses Problem gelöst ist. Die Notwendigkeit der Durability-Eigenschaft ist offensichtlich.

Für die Einhaltung des ACID-Prinzips sorgen hauptsächlich die Systemkomponenten **Concurrency-Control** (siehe Kapitel 5) und **Logging/Recovery-Control**. Die Aufgabe der Concurrency-Control ist die Trennung von parallel bearbeiteten Transaktionen durch Sperren auf die Objekte (z.B. Tupel), die von den Transaktionen gelesen oder geschrieben werden. Die Logging/Recovery-Control stellt sicher, daß nach einem Systemcrash die noch nicht beendeten Transaktion "zurückgerollt" werden (Atomicity) und die schon beendeten Transaktionen erhalten bleiben (Durability).

Kapitel 3

Die SB-PRAM

3.1 Vom theoretischen PRAM-Modell zur SB–PRAM

Um parallele Algorithmen spezifizieren und analysieren zu können, wurde in der theoretischen Informatik das Modell eines abstrakten Parallelrechners entwickelt. Dieses hardware-unabhängige Modell — genannt *Parallel Random Access Machine (PRAM)* — wird in [KR90] definiert[1]:

Definition 3.1 (PRAM) Eine **PRAM** ist eine parallele Registermaschine [AHU74] mit n Prozessoren $P_0, \ldots, P_{n-1}$, die auf einen globalen, gemeinsamen Speicher SM (Shared Memory) zugreifen können. In einem Schritt führt jeder Prozessor P_i entweder eine Registeroperation oder einen Zugriff auf SM aus. Ein Schritt hat konstante Dauer.

Bei gleichzeitigem Zugriff mehrerer Prozessoren auf die gleiche Speicherzelle entstehen Konflikte, die mittels Konventionen gelöst werden [FW78, LPV81]. Die im folgenden betrachtete PRAM gehört zur Klasse der *Priority-CRCW-PRAMs*. Dies bedeutet:

CRCW: Mehrere Prozessoren dürfen entweder lesend oder schreibend auf die gleiche Speicherzelle zugreifen. CRCW steht für *concurrent read/concurrent write*.

Priority: Den Prozessoren wird eine eindeutige Priorität zugeordnet. Bei konkurrierenden Schreibzugriffen setzt sich der Prozessor mit der größten Priorität durch, schreibt also seinen Wert in die Speicherzelle.

An der Universität Saarbrücken wird am Lehrstuhl für Rechnerarchitektur eine Hardware-Simulation einer Priority-CRCW-PRAM entwickelt. Der Aufbau dieser sogenannten *SB-PRAM* (im folgenden auch einfach PRAM genannt) ist in Abbildung 3.1 dargestellt.

[1] Definition leicht abgeändert

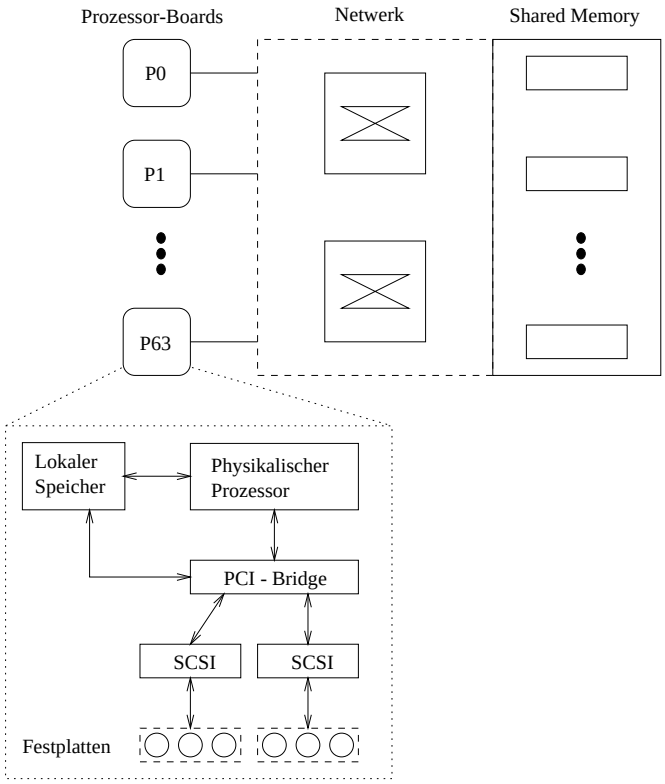

Abbildung 3.1: Schematischer Aufbau der SB-PRAM

Insgesamt gibt es $C = 32 \cdot 64 = 2048$ Prozessoren[2], denen entsprechend dem Priority-Modell eine Priorität zwischen 0 und 2047 zugeordnet ist. Der Befehlssatz der Prozessoren ist an den *Berkley-RISC-Befehlssatz* [PS82] angelehnt, ihre Instruktionsrate beträgt ungefähr 0.22 Millionen Instruktionen pro Sekunde (MIPS).

Für die Implementierung der SB-PRAM werden die Prozessoren zu 64 Gruppen von je 32 Prozessoren zusammengefaßt. Es gibt $C_{pP} = 64$ Prozessorplatinen, denen jeweils eine Prozessorgruppe zugeordnet ist. Auf jeder Prozessorplatine befindet sich ein Prozessorchip — ein sogenannter physikalischer Prozessor —, der die Befehle der Prozessoren in der zugeordneten Gruppe ausführt. Dazu werden die Prozessoren der entsprechenden Prozessorgruppe von 0 bis 31 durchnummeriert. In 32 Taktzyklen führt der Prozessorchip jeweils einen Befehl für die ihm zugeordneten Prozessoren in der Reihenfolge der Numerierung aus. Den Zeitraum von 32 Taktzyklen nennt man *Runde*.

Die einzelnen Prozessorplatinen sind über ein Netzwerk mit C_{pP} Speicherboards verbunden. Der Gesamtspeicher beträgt $L_{SM} = 1\text{GW}$[3]. Eine weiterführende Betrachtung der SB-PRAM findet man in [Kel92, DS92, AD+93, KPS94, Sch95, Bac96, Lic96, Wal97].

[2] In SB-PRAM-Terminologie werden diese Prozessoren "virtuelle Prozessoren" genannt.
[3] $1\text{GW} = 2^{30}\text{W} = 2^{30} \cdot 4$ Byte

3.2 Festplattenanbindung der SB-PRAM

Für jedes Datenbanksystem sind Festplatten zur Speicherung der Datenmengen unerläßlich. In diesem Abschnitt soll daher erläutert werden, wie Festplatten von den Prozessoren der SB-PRAM angesprochen werden. Eine ins Detail gehende Beschreibung der Festplattenanbindung findet man in [Kun99].

Grundsätzlich ist die Steuerung der Festplatten eine Aufgabe des PRAM-Betriebssystems *PRAMOS* [BR97]. Das Betriebssystem stellt dem Programmierer die Funktion scsirw() zur Verfügung, die die Festplattenzugriffe ausführt. Für die Entwicklung eines Datenbanksystens ist es jedoch wichtig, die prinzipielle Funktionsweise eines scsirw()-Aufrufs zu kennen.

Jede Prozessorplatine verfügt über eine PCI-Bridge und zwei daran angeschlossenen SCSI-Controller (siehe Abbildung 3.1). An diese Controller werden die Festplatten der SB-PRAM angeschlossen. Weiterhin ist jeder Prozessorchip an einen lokalen Speicher angeschlossen, welcher wiederum in Verbindung mit dem PCI- und SCSI-Controller steht. Der lokale Speicher dient als Puffer zwischen Prozessoren und Festplatten.

Dem Betriebssystem wird für jede Prozessorplatine ein Prozessor aus der entsprechenden Prozessorgruppe zugeordnet. Dieser Betriebssystem-Prozessor — auch *Serviceprozessor* genannt — kontrolliert den SCSI-Bus und stellt dadurch die Verbindung zu den Festplatten her. Weiterhin gibt es in jeder Prozessorgruppe mindestens einen sogenannten *Copyprozessor*. Dessen Aufgabe wird im folgenden erläutert.
Es sei angenommen, daß ein beliebiger Prozessor P einen Block von einer Festplatte laden möchte. Dies läuft folgendermaßen ab (Schreibzugriffe verlaufen analog):

1. Prozessor P stellt fest, an welcher Prozessorplatine die betreffende Festplatte angeschlossen ist. Sei dies Platine i. Der Service-Prozessor der Prozessorgruppe i sei mit P_s bezeichnet.

2. Prozessor P schickt eine Nachricht an den Service-Prozessor P_s. Dazu schreibt er einen *Service-Request* in eine parallele Warteschlange, die P_s zugeordnet ist. Der Service-Request enthält als Information die Nummer der Festplatte und die Position und Länge des Blocks, der von Festplatte gelesen werden soll. Als weitere Information enthält der Service-Request die Adresse A eines Speicherbereichs im gemeinsamen Speicher. An die Adresse A soll der von Festplatte gelesene Block geschrieben werden. Außerdem enthält der Service-Request den Zeiger auf eine Variable status. status ist mit SCSI_WAIT initialisiert.

3. Der Prozessor P_s liest den neuen Eintrag aus der Warteschlange und schickt über den SCSI-Treiber den Befehl an die Festplatte, den gewünschten Block zu lesen und in den lokalen Speicher der Prozessorplatine zu schreiben.

4. Nachdem die Festplatte dem Prozessor P_s die Ausführung des Lesebefehls quittiert hat, veranlaßt P_s die Copyprozessoren der Gruppe i, den Block

aus dem lokalen Speicher an die Adresse A im gemeinsamen Speicher zu kopieren.

5. Nach Beendigung des Kopiervorgangs quittiert Prozessor P_s über die Variable status die Ausführung der Leseoperation an Prozessor P. Dazu setzt er status := SCSI_OK.

Ein scsirw-Aufruf benötigt wenige Instruktionen, da nur der Service-Request in die Warteschlange eingefügt werden muß. Danach kann Prozessor P parallel zum eigentlichen Festplattenzugriff andere Arbeit verichten.
Die Festplattenanbindung der SB-PRAM ist noch nicht vollständig implementiert.

3.3 Zeitmodell

Im folgenden werden wir Aussagen über Zeitpunkte machen, zu denen Prozessoren bestimmte Instruktionen ausführen. Dazu werden hier zwei Zeitmodelle vorgestellt. Die Terminologie und Notation stammt aus [Röh96].

Üblicherweise sieht der Programmierer der SB-PRAM einfach 2048 Prozessoren, die gemäß dem PRAM-Modell parallel arbeiten. Alle 2048 Prozessoren führen ihre Befehle scheinbar gleichzeitig aus. Wenn wir von diesem Modell ausgehen, reden wir im folgenden von *R-gleichzeitiger* Ausführung der Befehle. Wir numerieren die *R-Zeitpunkte* von 0 an durch. Zu jedem R-Zeitpunkt führen alle 2048 Prozessoren genau eine Instruktion aus. Den R-Zeitpunkt, zu dem Instruktion I ausgeführt wird, bezeichnen wir mit $t^R(I)$.

Die Definition der Priority-CRCW-PRAM ermöglicht die Definition einer *sequentiellen Semantik*. Zur parallelen Ausführung jeder Instruktionenfolge gibt es eine äquivalente sequentielle Anordnung der ausgeführten Instruktionen. Die sequentielle Semantik erleichtert einige Beweise. Wir definieren den *V-Zeitpunkt* einer Instruktion I, die von Prozessor P zum R-Zeitpunkt t ausgeführt wird, als

$$t^V(I) := 2048 \cdot t + \mathrm{prio}(P).$$

Dabei sei $\mathrm{prio}(P) \in \{0, \ldots, 2047\}$ die Priorität des Prozessors P. Werden alle Instruktionen, die von den 2048 Prozessoren parallel ausgeführt werden, sequentiell in Reihenfolge ihrer V-Zeitpunkte ausgeführt, so bleibt die Semantik der Instruktionenfolge erhalten. Wenn nicht weiter spezifiziert, benutzen wir im folgenden immer das V-Zeitmodell. Eine Aussage wie "P_1 führt den Befehl vor P_2 aus" bezieht sich auf die V-Zeitpunkte, zu denen der Befehl ausgeführt wird.

Wird eine Instruktionenfolge (zum Beispiel eine Funktion eines Programms) $\mathcal{I} = I_1, \ldots, I_n$ von einem Prozessor ausgeführt, so bezeichnen wir mit $t_B^R(\mathcal{I}) := t^R(I_1)$ den R-zeitlichen Beginn und mit $t_E^R(\mathcal{I}) := t^R(I_n)$ das R-zeitliche Ende der Ausführung von $\mathcal{I}$. Analog bezeichnen wir mit $t_B^V(\mathcal{I}) := t^V(I_1)$ und $t_E^V(\mathcal{I}) := t^R(I_n)$ den V-zeitlichen Beginn und das Ende der Ausführung von $\mathcal{I}$.

Man beachte, daß V-Zeitpunkte zwar eine sequentielle Interpretation einer parallelen Rechnung ermöglichen, die Rechnung selbst dennoch parallel ausgeführt

wird. Die V-Zeitpunkte sind nichts weiter als ein Hilfsmittel, die parallele Welt in die gewohnte sequentielle Welt abzubilden.

3.4 Multipräfix

Neben den gewöhnlichen Lese/Schreib-Zugriffen (load/store) gibt es auf der SB-PRAM die Multipräfix-Operationen, um auf den gemeinsamen Speicher zuzugreifen:

Definition 3.2 (Multipräfix) Eine *Multipräfix-Operation* ist von der Form $mp \circ (V, D)$ mit $\circ \in \{add, and, or, max\}$, V eine ganzzahlige Variable im globalen Speicher und D ein ganzzahliger Wert. Sei P eine Prozessor, der zum V-Zeitpunkt t die Multipräfix-Operation $mp \circ (V, D)$ ausführt. Dann erhält P als Ergebnis der Operation den Wert V_{alt} von V zu Beginn von t zurück, und nach der Ausführung enthält V den Wert $V_{neu} := V_{alt} \circ D$.
Die Multipräfix-Operation *add* arbeitet sowohl auf vorzeichenbehafteten als auch vorzeichenlosen Ganzzahlen, *and* und *or* entsprechen bitweisem ver-und-en bzw. ver-oder-n. Bei der Multipräfix-Operation *max* findet ein vorzeichenbehafteter Vergleich zur Maximumsberechnung statt.

Die Multipräfix-Operationen *mp*$\circ$ sind auch als *Fetch-and-*$\circ$ bekannt. Neben den Multipräfix-Operationen gibt es auf der SB-PRAM *Sync-Operationen*. Sie entsprechen genau den Multipräfix-Operationen, jedoch liefern sie dem Prozessor kein Ergebnis zurück.
Multipräfix-Operationen sind für den Programmierer ein sehr mächtiges Werkzeug. Zum Beispiel basieren sämtliche Synchronisierungsmechanismen für die SB-PRAM wie *Locks* (siehe Abschnitt 3.6) und *Barriers* [Röh96] auf paralleler Nutzung von Multipräfix-Operationen. Es ergibt sich jedoch in Bezug auf parallele Speicherzugriffe für die SB-PRAM folgende Restriktion:

Restriktion: Auf einer Variable V im gemeinsamen Speicher darf in einer Runde (äquivalent zu einem R-Zeitpunkt) höchstens einer der Operationstypen *load, store, mpadd, mpand, mpor, mpmax, syncadd, syncand, syncor* und *syncmax* ausgeführt werden.

Um Speicherzugriffe verschiedenen Typs voneinander zu trennen, kann das sogenannte *Modulo-Bit* benutzt werden. Nach jeder Runde wird dieses Bit negiert. Um zum Beispiel *load-* und *store-Zugriffe* auf die gemeinsame Variable V zu trennen, können die Befehle *load_m0, store_m1* benutzt werden. *load_m0* wartet, bis das Modulo-Bit gleich 0 ist und führt dann die Load-Operation aus. Analog wartet *store_m1* mit dem Schreiben bis das Modulo-Bit gleich 1 ist.

3.5 Programmierung der SB-PRAM

Um eine bequeme Programmierung der SB-PRAM zu ermöglichen, wurde der GNU-C-Compiler gcc für die SB-PRAM portiert [Bos94]. Die Syntax entspricht der standardisierten C-Syntax, die für die SB-PRAM um zwei Variablenklassifizierer erweitert wurde:

shared: Mit dem Schlüsselwort shared wird dem Compiler bekanntgegeben, daß die als nächstes deklarierte Variable eine gemeinsame Variable aller Prozessoren ist. Zum Beispiel wird bei der Deklaration shared int S; zur Laufzeit nur einmal Speicher für die Variable S alloziert und alle Prozessoren können auf S zugreifen.

private: Im Gegensatz zu shared wird bei einer private-Deklaration für jeden Prozessor einzeln Speicherplatz alloziert. Zum Beispiel erzeugt die Deklaration private int P; für jeden Prozessor P_i eine eigene Variable P, auf die nur P_i Zugriff hat. Wird kein Klassifizierer angegeben, so gilt die Deklaration als privat.

Um Multipräfix- und Sync-Operationen in C benutzen zu können, wurden die entsprechenden Assembler-Befehle in [Röh96, Anhang B] in C-Makros gekapselt. Die Syntax lautet

```
sbp_mp<op>(<Adr>, <Wert>),
sbp_sync<op>(<Adr>, <Wert>),
sbp_ldg(<Adr>)                          ldg≙load global
sbp_stg(<Adr>, <Wert>)                  stg≙store global
```

Dabei steht <op> für eine Operation aus {add, and, or, max}, <Adr> steht für die Adresse einer gemeinsamen Variable und <Wert> für einen beliebigen C-Ausdruck, der zu einer Ganzzahl ausgewertet wird.
Zur Trennung verschiedener Operationen mittels Modulo-Bit (vgl. Abschnitt 3.4) stehen die gleichen Makros mit dem zusätzlichen Kürzel _m0 oder _m1 zur Verfügung (zum Beispiel sbp_mpadd_m0(S, 1)).

Die Quelltexte, die in dieser Arbeit angegeben sind, entsprechen nicht genau der C-Syntax, sondern einer C-ähnlichen und (hoffentlich) besser lesbaren Pseudo-Sprache.

3.6 Synchronisation von konkurrierenden Prozessen

Der Versuch mehrerer Prozessoren, gleichzeitig Änderungen an einer komplexen Datenstruktur (zum Beispiel Liste oder B–Baum) vorzunehmen, muß normalerweise sequentialisiert werden. Es darf in der Regel zu einem Zeitpunkt nur einem Prozessor erlaubt sein, Änderungsanweisungen durchzuführen. Andere Prozessoren müssen verzögert werden. Eine Realisierungsform dafür sind sogenannte *Sperren (Locks)*.
Eine Sperre ist ein abstrakter Datentyp mit den Operationen *sperren* und *freigeben*. Man ordnet jeder Instanz I einer Datenstruktur eine Sperre L_I zu. Bevor ein Prozessor P einen *kritischen Bereich* (zum Beispiel Auslesen oder Ändern der Daten) für I betritt, sperrt er die entsprechende Sperre L_I. Alle weiteren Prozessoren, die die Datenstruktur sperren wollen, werden verzögert. Am Ende des kritischen Bereichs wird die Sperre von P wieder freigegeben. Erst dann kann ein weiterer Prozessor die Sperre erhalten und den kritischen Abschnitt betreten. Man sagt, die Instanz I wird durch die Sperre L_I *geschützt*. Es gibt

auch Sperren, die in bestimmten Situation mehreren Prozessoren gleichzeitig Zugriff auf die Datenstruktur ermöglichen. In den folgenden Abschnitten werden verschiedene Sperren vorgestellt. Zu jeder Sperre wird ein Anwendungsbeispiel gegeben.

3.6.1 Einfache Sperre

Die *einfache Sperre* (*simple lock*) erlaubt immer nur einem Prozessor den Zugriff auf die durch die Sperre geschützte Datenstruktur. Implementieren läßt sich eine einfache Sperre durch eine ganzzahlige, gemeinsame Variable s, auf der die Funktionen simple_lock(s) und simple_unlock(s) operieren. s sei mit 0 initialisiert. Folgende Implementierung der einfachen Sperre stammt aus [Röh96]:

- simple_lock(s) :≡ { while(sbp_mpmax(s, 1) ≠ 0); } (3.1)

  Der Prozessor, der V-zeitlich als erster die mpmax-Instruktion ausführt, erhält den Wert 0 zurück. Dieser Prozessor *erhält die Sperre* und beendet die Funktion simple_lock(). Alle weiteren Prozessoren erhalten den Wert 1 zurück und müssen in der while-Schleife warten.

- simple_unlock(s) :≡ { s:=0; }

  Der Wert von s wird auf 0 zurückgesetzt. Die V-zeitlich nächste mpmax-Instruktion liefert demnach 0 zurück und setzt s dann wieder auf 1.

Ein Prozessor muß unter Umständen lange in der while-Schleife auf die Sperre warten. Während der Prozessor wartet, verrichtet er keine weitere Arbeit. Dies wird *busy-waiting* genannt. Auf anderen Architekturen gibt es zum busy-waiting die Alternative des Kontext-Wechsels. Wenn ein Prozessor die Sperre nicht sofort erhalten kann, wechselt er den Kontext und bearbeitet einen anderen *Prozeß (Thread)* [PS85]. Dies wird auf der SB-PRAM jedoch nicht unterstützt. Alle in dieser Arbeit vorgestellten Sperren sind busy-waiting–Sperren.

Beispiel I (paralleler Stack): Es soll nun ein Anwendungsbeispiel für die einfache Sperre gegeben werden. Betrachten wir einen abstrakten Datentyp Stack mit den Operationen push() und pop(). Der Stack sei für Uniprozessor-Systeme ausgelegt. Wir können den Stack einfach für die PRAM verwenden, indem wir jedem Stack S eine einfache Sperre s_S zuordnen. Zu Begin einer push()– oder pop()–Operation wird diese einfache Sperre s_S gesperrt (simple_lock(s_S)). Am Ende der Operation wird die Sperre wieder freigegeben (simple_unlock(s_S)). Dadurch wird sichergestellt, daß zu jedem Zeitpunkt nur ein Prozessor auf den Stack S zugreift.

Die einfache Sperre ist nicht *fair*. Fairneß wird ausführlich erst in Abschnitt 5.2 behandelt. Hier genügt es, folgendes Verständnis von Fairneß zu haben: jede angeforderte Sperre wird nach endlicher Zeit vergeben. Abbildung 3.2 zeigt, warum die vorgestellte einfache Sperre nicht fair ist. Drei Prozessoren fordern

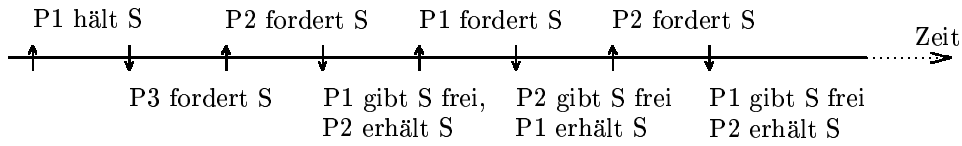

Abbildung 3.2: Prozessor P_3 verhungert, da Prozessoren mit kleinerer Priorität abwechselnd die Sperre S erhalten.

die einfache Sperre S an. In der Funktion simple_lock() liefert die mpmax-Instruktion dem Prozessor mit kleinster Priorität den Wert 0 zurück. Daher erhalten abwechselnd die Prozessoren P_1 und P_2 die Sperre, Prozessor P_3 wird übergangen. Prozessor P_3 *verhungert*, erhält also nie die Sperre S.

In [Röh96] wird eine faire einfache Sperre (fair_lock) entwickelt. Diese vergibt Sperren in der gleichen Reihenfolge, in der die Prozessoren die Funktion simple_lock() aufrufen. Es werden also keine Sperren "überholt", weshalb die Implementierung fair ist.

3.6.2 Weitere Sperren

Als Verallgemeinerung der einfachen Sperre wird die häufig verwendete *Lese/Schreib-Sperre* (*Reader/Writer-Lock, R/W-Lock*) vorgestellt. Bei dieser Sperre gibt es zwei verschiedene Sperrmodi:

Reader: mehrere Prozessoren dürfen gleichzeitig die Lese/Schreib-Sperre im Reader-Modus halten. Dies ermöglicht gleichzeitiges Auslesen von gemeinsamen Datenstrukturen durch mehrere Prozessoren.

Writer: nur ein Prozessor darf die Lese/Schreib-Sperre im Writer-Modus halten. Der Writer-Modus erlaubt exklusive Änderungsoperationen auf Datenstrukturen, die durch R/W-Locks geschützt werden. Es werden nicht gleichzeitig Sperren im Reader- und im Writer-Modus vergeben.

Den Reader/Writer-Lock gibt es in einer fairen Ausführung und in einer Ausführung, in der angeforderte Writer-Modi priorisiert vergeben werden. Die Funktionen rw_lock() und rw_unlock() bekommen als Parameter den Sperrmodus READER oder WRITER übergeben.
Sei l ein Reader/Writer-Lock. Wir sagen "Prozessor P fordert eine Schreib-Sperre auf l an", wenn P eine Sperre im Writer-Modus anfordert (analog für Lese-Sperre).

Beispiel II (parallele Liste): Als Anwendungsbeispiel für den Reader/Writer-Lock soll nun eine parallele Liste vorgestellt werden. Dazu gehen wir von einem sequentiellen abstrakten Datentyp Liste und den Operationen suchen() und einfuegen() aus. Wir ordnen jeder Liste L einen Reader/Writer-Lock rw_L zu. Die Funktion suchen() fordert zu Beginn eine Lese-Sperre auf rw_L (rw_lock(rw_L, READER)), die am Ende der Suche wieder freigegeben wird (rw_unlock(rw_L, READER)). Bei der Funktion einfuegen() wird zunächst eine Schreib-Sperre auf rw_L angefordert, die nach dem Einfügen

wieder freigegeben wird. Durch den Reader/Writer-Lock wird sichergestellt, daß mehrere Prozessoren gleichzeitig in der Liste suchen können, und daß nur ein Prozessor in die Liste einfügen kann. Außerdem ist kein Lese-Zugriff anderer Prozessoren möglich, während ein Prozessor ein neues Element in die Liste eingefügt.

Als dritter Sperrtyp soll noch die sogenannte *Gruppensperre* (*group lock*) vorgestellt werden. Eine i-Gruppensperre, $i \in \mathbb{N}_{\geq 2}$, verwaltet i Gruppen. Prozessoren fordern über die Funktion group_lock() eine Sperre für eine Gruppe j an, $1 \leq j \leq i$. Zu jedem Zeitpunkt dürfen für eine Gruppensperre g beliebig viele Sperren der gleichen Gruppe, niemals jedoch Sperren unterschiedlicher Gruppen vergeben sein. Bei der Anforderung und Freigabe einer Gruppensperre wird die Gruppennummer j als Parameter an die Funktionen group_lock(g, j) beziehungsweise group_unlock(g, j) übergeben. Die Implementierung der Gruppensperre für die SB-PRAM ist fair.

Beispiel III (verbesserter Stack): In [Röh96] wird eine gegenüber Beispiel I verbesserte Implementierung eines Stacks erklärt. Diese erlaubt die parallele Ausführung mehrere pop()-Operationen sowie mehrerer push()-Operationen. Jedoch können nicht gleichzeitig pop()- und push()-Operationen ausgeführt werden. Um dies zu verhindern, wird jeder Stack-Instanz S eine 2-Gruppensperre gr_S zugeordnet. Die pop()-Operation wird durch ein group_lock(),group_unlock()-Paar geschützt, wobei jeweils als Gruppennummer 1 übergeben wird. Analog werden push()-Operationen durch gr_S mit Sperren für die Gruppe 2 geschützt. Somit wird sichergestellt, daß nicht gleichzeitig ge-pop-t und ge-push-t wird.

Für weitere Details sowie zur Implementierung der besprochenen Sperren sei auf [Röh96] verwiesen. Dort finden sich auch weitere Anwendungen dieser Sperren. Wir nennen die in diesem Abschnitt vorgestellten Sperren *Standard-Sperren*.

3.6.3 Die Begriffe *Sperre halten* und *warten*

In diesem Abschnitt soll das Verhalten der obigen Sperren nochmals betrachtet werden, um eine formale Definition der Begriffe "Sperre halten" und "auf Sperre warten" zu ermöglichen.
Wir können den Ablauf jeder Sperranforderung in zwei Phasen aufteilen. Die Phase 1 heißt *Initialisierungs- und Warte-Phase*; Phase 2 heißt *End-Phase*. Die End-Phase beginnt, sobald die angeforderte Sperre sicher erhalten werden kann oder bereits erhalten ist. Formal: Die Phase 2 einer Sperranforderung beginnt, sobald die Restzeit der Sperranforderung unabhängig vom Verhalten der anderen Prozessoren ist. Dieser Zeitpunkt ist durch die Implementierungen der verschiedenen Sperren definiert.

Beispiel: Sei l eine (unfaire) einfache Sperre wie in Abschnitt 3.6.1 vorgestellt. Seien P_1 und P_2 die einzigen Prozessoren, die auf l Operationen ausführen. Zu Beginn sei die Sperre frei und nur Prozessor P_1 fordere eine Sperre auf

l an. Da die Sperre frei ist und von keinem anderen Prozessor angefordert wird, ist die Sperre sicher erhalten, sobald P_1 zum ersten Mal den mpmax-Befehl ausgeführt hat (siehe (3.1), Seite 17)). Bis zu diesem mpmax befindet sich P_1 in Phase 1, anschließend in Phase 2. Unter anderem findet der Vergleich auf $\neq 0$ in Phase 2 statt.

Nachdem P_1 die Anforderung beendet hat, beginnt P_2 eine Sperranforderung auf l. Prozessor P_2 muß nun warten, da P_1 die Sperre noch nicht freigegeben hat. Sobald P_1 die Sperre freigibt und P_2 einen weiteren mpmax ausführt, ist P_2 von anderen Prozessoren unabhängig und beginnt Phase 2 seiner Anforderung.

Auch der Reader/Writer-Lock und die Gruppensperre haben einen zum mpmax-Befehl der einfachen Sperre analogen Befehl, durch den die beiden Phasen definiert werden.

Definition 3.3 Sei P ein Prozessor und l eine Sperre. Wir sagen P **wartet auf die Sperre** l, wenn P eine Sperranforderung auf l begonnen hat, die noch nicht Phase 2 erreicht hat. Wir sagen P **hält die Sperre** l, wenn P seine letzte Sperranforderung auf l beendet hat und die Sperrfreigabe (Operation unlock()) noch nicht begonnen hat.

Man beachte, daß nach dieser Definition ein Prozessor eine Sperre noch *nicht* hält, wenn er Phase 2 begonnen hat. Erst nach dem Ende von Phase 2 hält der Prozessor die Sperre.

3.7 Die SB-PRAM als Datenbankserver

In diesem Abschnitt soll eine stichwortartige Zusammenfassung der Arbeit von Spengler [Spe97] gegeben werden. Es kann hier nicht auf die Details eingegangen werden. Dazu sei auf [Spe97] verwiesen.

Seitenorientierte Datenspeicherung
Alle Daten im PRAM-Datenbanksystem werden seitenorientiert gespeichert. Eine *Seite (Page)* ist ein Speicherbereich fester Größe $L_{\mathrm{PG}} = 4KB$, in dem relationale Tupel oder Metadaten (bzw. Repräsentationen davon) abgelegt werden. In einer Seite werden nur Tupel derselben Relation gespeichert.
Jeder Relation wird eine eindeutige Identifizierungs-Nummer rid > 0 (Relations-ID) zugeordnet. Jeder Seite, die Daten der Relation rid speichert, wird eine eindeutige Nummer pid (Page-ID) zugeordnet. Jede Seite wird somit durch das Paar (rid,pid) eindeutig bestimmt. Die Datenseiten jeder Relation werden zu doppelt verketteten Listen zusammengefaßt, um eine Traversierung der Datenseiten zu ermöglichen.

Seitenspeicherung auf Festplatten
Die Seiten werden in einer *Round-Robin-Verteilung* auf Festplatten gespeichert. Das heißt: sind die Festplatten $HD_0, \ldots, HD_{i-1}$ für die Speicherung der Seiten

der Relation rid reserviert, so wird die Seite (rid,pid) auf Festplatte pid mod i gespeichert, und zwar an Position $\lfloor \mathsf{pid}/i \rfloor \cdot 4\text{KB}$.

Transaktionsbearbeitung Ein Prozessor arbeitet zu jedem Zeitpunkt an nur einer Transaktion. Es sind demnach keine Kontext-Wechsel nötig. Umgekehrt wird jede Transaktion nur von einem Prozessor bearbeitet. Es gibt also keine *Intra-Transaktions-Parallelität*[4].

Zugriffspfade
Die Zugriffspfade [GR93, Kapitel 15] ermöglichen dem DBS den Zugriff auf relationale Daten. Als Standard-Zugriffspfad wird die sequentielle Suche durch die Liste aller Datenseiten einer Relation genutzt. Da dies bei großen Relationen und kleinen Ergebnismengen sehr ineffektiv ist, werden weitere Zugriffspfade zum direkten Zugriff auf gespeicherte Daten verwendet. Legt man für eine Teilmenge der Attribute einer Relation einen Zugriffspfad an, lassen sich Selektionen nach diesen Attributen effektiv durchführen.
Die Bereitstellung effizienter Zugriffspfade zum Suchen bestimmter Datensätze bezeichnet man als *Indizierung* der Relation. Der Zugriffspfad selbst wird *Index* genannt. Die Menge der Attribute, auf die sich ein bestimmter Index bezieht, bezeichnet man als *Schlüsselattribute*. Sind die Ausprägungen der Datensätze einer Relation auf den Schlüsselattributen paarweise verschieden, so nennt man die Attribute und den Index *eindeutig*.
Für jede Relation wird ein Index als *Primärindex* ausgezeichnet. Die Schlüsselattribute des Primärindex bezeichnet man als Primärschlüssel. Der Primärschlüssel muß eindeutig sein. Der Primärindex bestimmt die Zuordnung von Tupeln auf Seiten (siehe unten). Die weiteren (nicht notwendig eindeutigen) Indizes bezeichnet man als *Sekundärindizes*.
Als Index können im PRAM-Datenbanksystem B^+-Bäume und Hash-Indizes verwendet werden.

B-Bäume sind balancierte Mehrweg-Suchbäume und wurden zuerst von Bayer und McCreight vorgestellt [BM72]. Bei B-Bäumen werden in allen Knoten des Baums Daten gespeichert. B^+-Bäume sind eine allgemein bekannte und häufig verwendete Variante der B-Bäume, bei denen die Daten nur in den Blättern des Baums gespeichert werden. Die inneren Knoten dienen nur als "Wegweiser" bei der Suche im Baum. Die B^+-Variante vereinfacht die Verkettung der Datenseiten des Baums. Diese Verkettung wird benötigt, um mehrere Datensätze den Indexattributen nach sortiert zu selektieren.
Wird der B^+-Baum als Primärindex benutzt, werden in den Blättern die vollständigen Datensätze gespeichert. Die Blätter des Baums sind die Datenseiten der Relation. Man nennt den Primärindex *unmittelbar*. In einem Sekundärindex werden in den Blättern des B^+-Baums Zeiger in Form von (rid,pid)-Tupeln auf die Datenseiten der Tupel gespeichert. Der Index heißt dann *mittelbar*.

[4]Dieses Prinzip muß vermutlich aufgegeben werden, wenn mit dem PRAM-Datenbanksystem komplexere Transaktionen als der TPC-B–Benchmark (siehe Kapitel 6) verarbeitet werden sollen. Dann könnten komplexe Operationen wie Joins oder Sortierung parallel bearbeitet werden (vgl. [Gem95]).

Die SB-PRAM–Implementierung der B^+-Bäume folgt den Ausführungen in [GR93, CLR90, Jan94]. Spengler zeigt, wie diese Implementierung für gleichzeitige Zugriffe mehrerer SB-PRAM–Prozessoren verändert werden kann [Spe97, Seite 39f.]. Für die SB-PRAM implementiert wurden die B^+-Bäume im Rahmen eines Fortgeschrittenpraktikums des Autors am Lehrstuhl Prof. Paul, vervollständigt wurde diese Implementierung im Rahmen der vorliegenden Diplom-Arbeit.

Als zweite Indexform können im PRAM-Datenbankystem Hash-Indizes als Primärindex benutzt werden. Wie der Name sagt, liegt den Hash-Indizes die Idee des *Hashings* [Knu73, Meh84] zugrunde. Für jedes Tupel wird durch den Wert des Primärschlüssels über eine Hash-Funktion die Seite bestimmt, in der das Tupel gespeichert wird. Dabei können *Seitenüberläufe* auftreten, wenn zu viele Tupel durch die Hash-Funktion auf die gleiche Seite abgebildet werden. Dieses Problem läßt sich durch verkettete *Überlauf-Seiten* oder *offene Adressierung* [CLR90] lösen. Für Details und Varianten von Hashing (wie zum Beispiel *extensible Hashing*) in Datenbanksystemen sei auf [GR93] verwiesen.

Logging und Recovery

Wie in Kapitel 2 erklärt, muß bei einem Transaktionssystem die *Durability*-Eigenschaft gewahrt werden. Dies wird in Datenbanksystemen durch eine Logging/Recovery-Komponente erreicht, die die Änderungen der Transaktionen auf Festplatte protokolliert. Aus diesen Protokollen läßt sich nach einem Systemfehler ein transaktionskonsistenter Zustand herstellen. Spengler beschreibt in seiner Arbeit [Spe97, Kap. 9] eine Logging/Recovery-Komponente für die SB-PRAM. Diese wurde bei Spengler nicht implementiert, und ihre Implementierung hätte auch den Rahmen dieser Arbeit gesprengt. Wir geben Zeitmessungen in dieser Arbeit immer unter Vernachlässigung der Logging-Komponente an, obwohl diese einen erheblichen Einfluß auf die Geschwindigkeit des Datenbanksystems haben kann [GR93, Kap. 9].

Kapitel 4

Systempufferverwaltung

4.1 Anforderungen an die Systempufferverwaltung

Alle Daten im PRAM-Datenbanksystem werden seitenorientiert auf Festplatten gespeichert (vgl. Abschnitt 3.7 und [Spe97]). Der *Systempuffer* (*Page Array*) ist der Teil des gemeinsamen Speichers, in dem die zur Bearbeitung von Transaktionen benötigten Seiten abgelegt werden. Der Systempuffer ist in durchnummerierte Bereiche der Seitengröße $L_{\mathrm{PG}} = 4$ Kilobytes, sogenannte Slots, eingeteilt. In jedem Slot wird eine Seite *gepuffert*. Die Anzahl dieser Slots wird mit S_{PA} bezeichnet; somit ergibt sich die Größe des Systempuffers zu $L_{\mathrm{PG}} \cdot S_{\mathrm{PA}} = S_{\mathrm{PA}} \cdot 4$ KB. Zu jedem Slot i gibt es eine Kontrollstruktur PCB[i], die Verwaltungsinformationen über die Seite in Slot i enthält. Die Kontrollstruktur wird im nächsten Abschnitt definiert.

Die gesamte Datenmenge einer Datenbank übersteigt die Kapazität des Systempuffers im Normalfall bei weitem. Daher ist es nötig, im Systempuffer nicht vorhandene Seiten von Festplatte zu laden, sobald diese von anderen Teilen des DBS, wie zum Beispiel der B^+-Baum–Verwaltung, angefordert werden. Ist der Systempuffer bereits vollständig gefüllt, so muß eine Seite aus dem Systempuffer *verdrängt* werden. Die verdrängte Seite heißt *Opfer*. Ist das Opfer durch Transaktionen verändert worden, wird es vor der Verdrängung auf Festplatte zurückgeschrieben.

Die Schnittstelle zwischen Systempuffer und den weiteren Komponenten des Datenbanksystems besteht aus drei Funktionen:

fix(rid,pid) Die Funktion fix() lädt die Seite (rid,pid) in den Puffer, falls sie dort noch nicht vorhanden ist. Weiterhin verhindert fix(), daß die Seite verdrängt wird. Die Funktion liefert die Kontrollstruktur zu dieser Seite zurück, über die der weitere Zugriff auf die Seite möglich ist. fix() wartet nicht, bis die Seite von Festplatte geladen wurde, sondern initiiert lediglich den Festplattenzugriff (vgl. Abschnitt 3.2).

unfix(p) Die Funktion unfix() gibt die Seite mit Kontrollstruktur p zur Verdrängung frei. Haben mehrere Prozessoren einen fix()-Aufruf für diese Seite ausgeführt, muß jeder dieser Prozessoren unfix() aufrufen, bevor die Seite verdrängt werden darf.

testbuf(p) Die Funktion testbuf() liefert TRUE, wenn die Seite vollständig von Festplatte geladen wurde. Dies muß nach einem fix()-Aufruf überprüft werden, bevor auf die Seite zugegriffen werden darf.

Zu jedem fix()-Aufruf muß es genau einen unfix()-Aufruf geben, und der unfix()-Aufruf darf erst beginnen, wenn die Seite vollständig geladen wurde. Man nennt diese Systempuffer-Schnittstelle fix–use–unfix–Schnittstelle.
Ist bei einem fix()-Aufruf die Seite schon gepuffert, so spricht man von einem *buffer-hit*, ansonsten von *buffer-miss*. Durch eine geschickte Wahl der *Verdrängungsstrategie*, also der Auswahl des Opfers, soll eine möglichst große Anzahl von *buffer-hits* erreicht werden. Insofern stellt der Systempuffer auch einen *Cache* für die Seiten dar.

Im folgenden Abschnitt wird der Aufbau des Systempuffers und die Implementierung der Funktionen fix(), unfix() und testbuf() beschrieben. Anschließend wird die Verdrängungstrategie erklärt.

Effizienzbetrachtungen zu der in diesem Kapitel vorgestellten Implementierung des Systempuffers findet man zum Beispiel in [Spe97, GR93, Här87, NDD92]. Allerdings beziehen diese Analysen sich weitestgehend auf Systeme mit wenigen Prozessoren und lassen sich nicht ohne weiteres auf massiv-parallele Architekturen wie die SB-PRAM übertragen. Ob die Ergebnisse dieser Analysen auch für die SB-PRAM gelten, müssen Messungen noch ergeben.

Die hier beschriebene Systempufferverwaltung basiert auf den Ausführungen in [Spe97]. Der dort vorgestellte Systempuffer implementiert die fix–use–unfix–Schnittstelle nur indirekt. Spenglers Systempuffer unterstützt außerdem nur blockierende Festplattenzugriffe. Auch ist das Protokoll zum Sperren einzelner Teile des Systempuffers in beiden Implementierungen unterschiedlich. Die Systempufferverwaltung hängt eng mit der im nächsten Kapitel vorgestellten neuen *Concurrency-Control* zusammen, weswegen sie im Rahmen dieser Arbeit neu implementiert werden mußte. Daher ist der Systempufferverwaltung auch in dieser Arbeit ein Kapitel gewidmet.

4.2 Aufbau des Systempuffers

Zur Verwaltung der gepufferten Seiten werden Zusatzinformationen in sogenannten *Page Control Blocks* (*PCB*) gespeichert. Dazu wird jedem Slot i ein Page Control Block PCB[i] zugeordnet, wobei PCB[ ] ein eindimensionales Feld ist, das über die Slotnummer adressiert wird.

Definition 4.1 (Page Control Block) Ein **Page Control Block (PCB)** ist eine Struktur mit folgenden Komponenten:

- rid: Relations-ID der durch diesen PCB beschriebenen Seite.
- pid: die entsprechende Page-ID.
- users: users zählt die Anzahl der fix()-Aufrufe minus unfix()-Aufrufe. Es gilt immer users≥ 0.

status: entweder SCSI_WAIT oder SCSI_OK, je nachdem, ob die Seite bereits vollständig von Festplatte geladen ist (vgl. Abschnitt 3.2).

buf: ein Zeiger auf den $L_{\mathrm{PG}} = 4\mathrm{KB}$ großen Systempuffer-Slot, in dem die Seite abgespeichert wird.

Der Struktur werden später weitere Komponenten hinzugefügt.

Durch die Implementierung der Systempufferverwaltung wird sichergestellt, daß jede Seite höchstens einmal im Puffer vorhanden ist. Liegt die Seite (rid,pid) im Systempuffer in Slot i, macht es daher Sinn, den Page Control Block PCB[i] als *den* (rid,pid)-PCB zu bezeichnen. Wir unterscheiden im folgenden nicht immer zwischen Seiten und den zugehörigen Kontrollblöcken.

Eine gepufferte Seite heißt *fixiert*, wenn in ihrem PCB users> 0 gilt, sonst *nicht fixiert*. Ist eine Seite fixiert, so wird sie nicht verdrängt.

Bei der Suche nach der Seite (rid,pid) im Systempuffer wird zunächst der (rid,pid)-PCB gesucht. Im Erfolgsfall ist dann mit dem (rid,pid)-PCB über die Komponente buf der Zugriff auf die Seite möglich. Eine Suche nach dem (rid,pid)-PCB ist über das Feld PCB[ ] jedoch nur ineffektiv möglich. Um die Suche zu beschleunigen, wird Hashing [Knu73, Meh84] verwendet. Die PCBs werden dazu nach den Komponenten rid, pid in eine Tabelle gehasht. Kollisionen werden durch einfach verkettete Listen gelöst. Wir setzen daher obige Definition fort, um Hashing zu ermöglichen:

Definition 4.1 (forts.): Ein **PCB-Kopf** ist der Listenkopf einer Hash-Liste und besteht aus folgenden Komponenten:

first: Ein Zeiger auf den ersten PCB in der Liste. Eine leere Liste wird durch first=NULL markiert.

lock: Mit diesem Reader/Writer-Lock werden lesende Zugriffe (Suchen) von schreibenden Zugriffen (Einfügen/Löschen) auf der Hash-Liste und auf den in der Liste gespeicherten PCBs getrennt (vgl. Beispiel II in Abschnitt 3.6.2).

Die PCB-Struktur wird erweitert um die Komponenten

head: head ist ein Zeiger zum Listenkopf der Hash-Liste, zu der dieser PCB gehört.

next: next ist ein Zeiger zum nächsten Listeneintrag. Das Listenende wird durch next=NULL markiert.

Abbildung 4.1 zeigt den schematischen Aufbau des Systempuffers. Die Hash-Tabelle heißt PAGEARRAY und ist ein PCB-Kopf–Feld der Größe S_{PAHT} (abkürzend für Pagearray-Hashtable). Um eine Seite (rid,pid) zu finden, wird mit einer Hash-Funktion h_{PA} ein PCB-Kopf in dem Feld PAGEARRAY bestimmt. Anschließend wird in der entsprechenden Hash-Liste nach dem (rid,pid)-PCB gesucht.

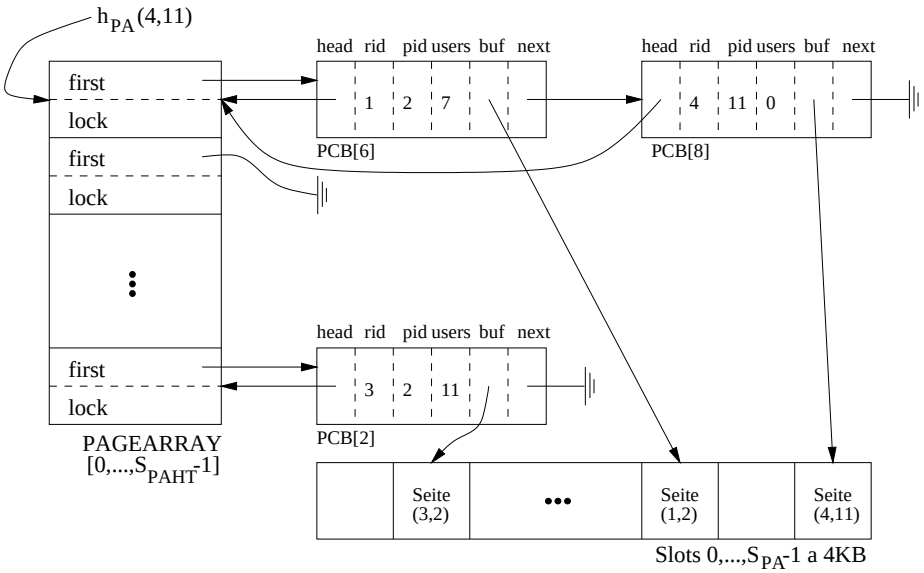

Abbildung 4.1: Aufbau des Systempuffers

Beschreiben wir nun die verwendete Hash-Funktion. Dazu zerlegen wir die Binärdarstellung der Relations-ID rid in r_1 Bytes

$$(rid_0, rid_1, \ldots rid_{r_1-1}) \in \{0, \ldots, 255\}^{r_1},$$

so daß

$$\mathsf{rid} = \sum_{i=0}^{r_1-1} rid_i \cdot 256^i$$

gilt. Analog betrachten wir die pid als Folge von r_2 Bytes $(pid_0, \ldots pid_{r_2-1})$. Die Größe der Hash-Tabelle S_{PAHT} muß im folgenden prim sein.
Als Hash-Funktion dient die Funktion

$$h_{\mathrm{PA}}(\mathsf{rid}, \mathsf{pid}) := \left(\sum_{i=0}^{r_1-1} a_i \cdot rid_i + \sum_{j=0}^{r_2-1} b_j \cdot pid_j \right) \bmod S_{\mathrm{PAHT}},$$

wobei die Koeffizienten $a_i, b_j \in \{0, \ldots, S_{\mathrm{PAHT}} - 1\}$ zum Systemstart zufällig und gleichverteilt gewählt werden. Die Klasse der Hash-Funktionen dieser Form ist *universell* [CLR90, Kap. 12.3.3]. Für universelles Hashing ergibt sich eine erwartete Listenlänge an jedem Listenkopf von $O(\frac{S_{\mathrm{PA}}}{S_{\mathrm{PAHT}}})$ mit kleinen Vorfaktoren [Meh84]. Wir nehmen im folgenden die Größe der Hash-Tabelle als $S_{\mathrm{PAHT}} \approx 2 \cdot S_{\mathrm{PA}}$ an. Dies garantiert im Schnitt kurze Laufzeiten für die Suche nach Seiten im Systempuffer [Meh84].

Gilt $S_{\mathrm{PAHT}} < 2^{24}$, so kann die Hash-Funktion durch übliche 32-Bit–Arithmetik ausgewertet werden: die Produkte in den Summen sind alle kleiner als $2^{24} \cdot 255 < 2^{32}$, also mit 32 Bit darstellbar. Daher kann jedes Zwischenergebnis der Auswertung der Hash-Funktion mit 32-Bit–Arithmetik direkt modulo S_{PAHT} reduziert werden, so daß keine Zwischenergebnisse $\geq 2^{32}$ entstehen können.

Bei einer Seitengröße von $L_{\mathrm{PG}} = 4\mathrm{KB}$ stellt die Bedingung $S_{\mathrm{PAHT}} < 2^{24}$ keine Einschränkung dar, da die maximale Systempuffergröße von $S_{\mathrm{PAHT}} \cdot L_{\mathrm{PG}} \approx 2^{36}$ Byte die Speichergröße der SB-PRAM um einen Faktor 16 überschreitet.

Der Rest dieses Abschnitts ist der Beschreibung der Implementierung der Funktionen fix(), unfix() und testbuf() gewidmet. Wir beginnen mit der Beschreibung der Funktion fix(). Diese lädt — wenn nötig — eine Seite in den Puffer und fixiert sie. Dazu wird zunächst überprüft, ob die Seite schon gepuffert ist. Ist dies nicht der Fall, muß die Seite in den Puffer geladen werden; dabei muß darauf geachtet werden, daß bei parallelen Zugriffen auf den Systempuffer die Seite nicht mehrmals geladen wird. Dies wird im folgenden detailliert erklärt. In Quelltext 4.1 (Faltblatt am Ende des Kapitels) ist Pseudo-Code für die Funktion fix() angegeben. Die mit (∗) markierten Zeilen können zunächst ignoriert werden.

In den Zeilen (1) und (2) wird die Hash-Funktion ausgewertet und auf die entsprechende Liste eine Lese-Sperre gesetzt. Danach wird in Zeile (3) die angeforderte Seite in der einfach verketteten Liste l :=PAGEARRAY[listennr].first gesucht. Nun sind die Fälle buffer-hit und buffer-miss möglich, je nachdem, ob die Seite in der Liste gefunden wurde. Der Fall des buffer-hit wird in den Zeilen (5)-(9) behandelt, der buffer-miss in den Zeilen (11)-(28). Wir betrachten zunächst den buffer-hit.

Sei also die gewünschte Seite (rid,pid) im Systempuffer vorhanden und bezeichne p den dazugehörigen PCB. Dieser ist dann bei der Suche in der Liste l gefunden worden. Mittels Multipräfix-Addition wird p.users inkrementiert (6); dadurch wird die Seite (rid,pid) fixiert. Zum Schluß der fix()-Funktion wird in Zeile (8) die Lese-Sperre auf der Liste wieder zurückgenommen und der gefundene PCB p zurückgegeben; über p.buf kann dann auf die Seite im Systempuffer zugegriffen werden.

Nun zum Fall des buffer-miss (Zeilen (11)-(28)): wird der gesuchte PCB in der Liste l nicht gefunden, so gibt fix() zunächst die Lese-Sperre auf der Liste wieder frei. In Zeile (12) fordert fix() durch einen get_pcb()-Aufruf einen freien Slot an. Der zugehörige PCB sei mit new_pcb bezeichnet. Die Funktion get_pcb() wird im nächsten Abschnitt erläutert.
Da der neue PCB new_pcb in die Liste eingefügt werden soll, wird in Zeile (13) eine Schreib-Sperre auf der Liste l gesetzt. Zwischen Sperrfreigabe (11) und Erhalt der Schreib-Sperre (13) kann bereits ein anderer Prozessor, der die Schreib-Sperre (13) zuerst erhielt, einen PCB für die gleiche Seite (rid,pid) eingefügt haben. Daher wird in Zeile (14) nochmals eine Suche nach dem gewünschten PCB ausgeführt. Während dieser Suche hat der Prozessor durch die Schreib-Sperre (13) exklusiven Zugriff auf die Liste, und daher kann kein anderer Prozessor gleichzeitig die Seite (rid,pid) laden.
Hat tatsächlich ein anderer Prozessor bereits einen PCB für die Seite (rid,pid) eingefügt, so wird in den Zeilen (16)-(21) wie im Fall des buffer-hit vorgegangen. Lediglich der erworbene freie PCB new_pcb wird durch einen put_pcb()-Aufruf zurückgegeben.
Hat andererseits noch kein anderer Prozessor einen entsprechenden PCB eingefügt, so wird in den Zeilen (22)-(26) der neue PCB in die Hash-Liste eingefügt.

Dazu wird zunächst das Laden der gewünschte Seite von Festplatte initiiert (Funktion scsirw(), vgl. Abschnitt 3.2), und anschließend werden die Komponenten des neuen Kontrollblocks initialisiert. Bei der Initialisierung wird auch die users-Komponente auf 1 gesetzt, wodurch die Seite fixiert wird. Danach wird in Zeile (26) der neue PCB am Listen-Anfang eingefügt. Zum Abschluß wird die Sperre auf der Liste wieder freigegeben (27) und der neue PCB zurückgeliefert.

Es ist nicht möglich, daß ein Prozessor einen PCB für die Seite (rid,pid) in die Liste l_1 einfügt, und ein zweiter Prozessor einen PCB für die gleiche Seite in eine andere Liste l_2 einfügt, da die Listennummer durch die vom Prozessor unabhängige Hash-Funktion h_{PA} bereits eindeutig bestimmt ist.

Quelltext 4.2 Die Funktionen unfix() und testbuf()

```
unfix(p)
PCB p;
{
   sbp_mpadd(p.users, -1)
}

testbuf(p)
PCB p;
{
   return (p.status = SCSI_OK)
}
```

Quelltext 4.2 zeigt Quelltext für die Funktionen unfix() und testbuf(). Die Funktion unfix() besteht einfach aus einer Multipräfix-Dekrementierung der users-Komponente. Dadurch wird die Fixierung in der Funktion fix() rückgängig gemacht.
Die testbuf()-Funktion überprüft, ob die Seite vollständig in den Puffer geladen wurde. Dies ist der Fall, wenn durch den SCSI-Treiber der Wert SCSI_OK in die Komponente status geschrieben wurde (vgl. Abschnitt 3.2). Die Funktion testbuf() überprüft genau das.

4.3 Verdrängungsstrategie *Clock*

Es bleibt zu klären, wie die Funktionen get_pcb() und put_pcb() implementiert werden. Dazu wird eine parallele Queue [Röh96] PCBQUEUE bereitgestellt, in der die PCBs der freien Slots liegen. Die Funktion get_pcb() versucht, einen freien PCB aus der PCBQUEUE zu lesen. Ist die PCBQUEUE leer, so wartet get_pcb(), bis ein anderer Prozessor einen freien PCB einfügt. In die PCBQUEUE eingefügt wird ein PCB p durch die Funktion put_pcb(p). Der weiter unten vorgestellte Verdrängungs-Algorithmus *Clock* ruft bei der Verdrängung einer Seite die Funktion put_pcb() auf.
In der Funktion put_pcb() wird die Daten-Rückspeicherung auf Festplatte vorgenommen. Dazu erweitern wir die PCB-Struktur um die Komponente dirty. Beim Laden einer Seite p von Festplatte wird p.dirty=0 gesetzt. Eine Transaktion, die die Seite verändert, muß p.dirty auf 1 setzen. Bei einem put_pcb(p)-Aufruf wird

die Seite p auf Festplatte geschrieben, falls p.dirty=1 ist. Danach wird der PCB in die PCBQUEUE geschrieben.

Um freie PCBs in die Queue zu schreiben, müssen in der Regel Seiten aus dem Systempuffer verdrängt werden. Nach einem Vorschlag in [GJM+95] und [Spe97] bietet sich für die SB-PRAM der *Clock-Algorithmus* [Här87] an. Ein dedizierter *Clock-Prozessor* P_{Clock} übernimmt die Suche nach verdrängbaren Seiten, sobald in der PCBQUEUE weniger als TH_{PCB} (für *Threshold*) freie PCBs gespeichert sind. Der Clock-Prozessor verdrängt die gefunden Seiten und fügt die freien PCBs in die PCBQUEUE ein. TH_{PCB} ist eine zum Systemstart festzulegende Konstante. Durch einen geeignet[1] gewählten Wert von $TH_{\text{PCB}} \approx 1/10 \cdot S_{\text{PA}}$ soll gewährleistet werden, daß bei einem buffer-miss die Wartezeit auf einen freien Slot möglichst niedrig ist. Umgekehrt darf der Wert jedoch nicht zu hoch sein, da sonst zu viele Seiten verdrängt werden.

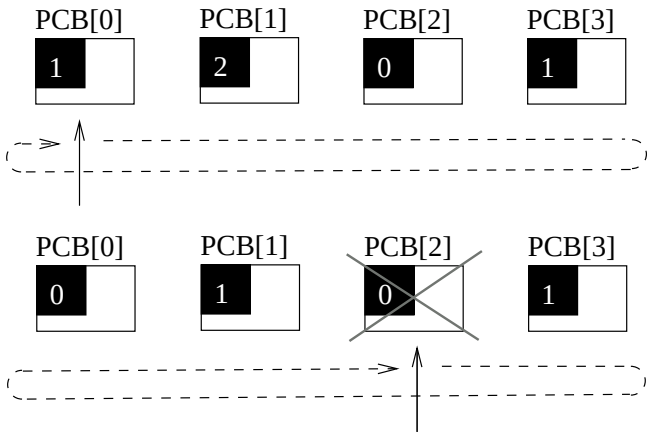

Abbildung 4.2: Aufbau des Systempuffers

Keine Seite sei fixiert. Oben: Ausgangslage. Unten: Nachdem der Clock-Prozessor bei den PCBs 0 und 1 die clock-Komponente dekrementiert hat, "findet" er den PCB 2, der bereits einen clock-Wert von 0 hat. PCB 2 wird verdrängt. Die nächste Suche wird bei PCB 3 fortgesetzt.

Wir erweitern die PCB-Struktur um die Komponente clock, welche beim Einfügen eines PCBs in eine Hash-Liste mit prio≥ 0 initialisiert wird. Der Wert prio wird der Funktion fix() als Parameter übergeben (siehe Quelltext 4.1, Zeilen mit (∗)). Der Parameter prio bestimmt die Priorität der Seite beim Verdrängen. Seiten mit hohen Prioritäten werden seltener verdrängt. Somit kann verhindert werden, daß häufig benutzte Seiten — wie zum Beispiel die oberen Ebenen von B^+-Bäumen — verdrängt werden.

Bei der Suche nach verdrängbaren Seiten durchläuft der Clock-Prozessor P_{Clock} schrittweise zyklisch alle Slots (bzw. die PCBs) in Reihenfolge ihrer Numerie-

[1] Der angegebene Wert von $1/10 \cdot S_{\text{PA}}$ ist eine Ad-hoc Schätzung. Er muß sich in der Praxis erst bewähren und muß eventuell geändert werden.

rung. Im Schritt i untersucht P_{Clock} den PCB $p(i) := \mathsf{PCB}[i \mod S_{\text{PA}}]$ (siehe Abbildung 4.2).
Ist der PCB $p(i)$ fixiert, so setzt der Clock-Prozessor die Suche nach einer verdrängbaren Seite direkt beim nächsten PCB $p(i+1)$ fort. Ob die Seite $p(i)$ fixiert ist, läßt sich am Wert der users-Komponente erkennen. Genau wenn users > 0 ist, ist die Seite fixiert.
Sei umgekehrt $p(i)$ nicht fixiert. Ist $p(i).\mathsf{clock} = 0$, so wird die Seite durch Löschen aus der entsprechenden Hash-Liste $p(i).\mathsf{head}$ und einen $\mathsf{put\_pcb}(p(i))$-Aufruf verdrängt. Falls $p(i).\mathsf{clock} > 0$, so wird $p(i).\mathsf{clock}$ dekrementiert und es beginnt der nächste Clock-Schritt. Bei erneuten Zugriffen auf die Seite mittels fix() wird der clock-Wert wieder auf prio gesetzt.

Es muß sichergestellt werden, daß während der Verdrängung kein anderer Prozessor die Seite erneut fixiert. Dazu setzt P_{Clock} vor der Verdrängung von $p(i)$ eine Schreib-Sperre im entsprechenden Listenkopf $p(i).\mathsf{head}$. Möchte dann während der Verdrängung ein anderer Prozessor die $p(i)$-Seite fixieren, so muß er vorher eine Lese-Sperre auf $p(i).\mathsf{head}$ setzen (siehe Funktion fix()). Dies ist nicht möglich, da der Clock-Prozessor eine Schreib-Sperre hält.

Kapitel 5

Concurrency-Control

Bei der parallelen Ausführung von Transaktionen entstehen Konflikte, wenn zwei oder mehr Transaktionen gleichzeitig auf einen Datensatz zugreifen wollen. Diese Konflikte werden durch Sperren gelöst. Bevor eine Transaktion einen Datensatz lesen oder schreiben darf, muß sie eine entsprechende Lese- oder Schreibsperre auf diesen Datensatz setzen. Dadurch werden inkompatible Zugriffe, zum Beispiel konkurrierende Schreibzugriffe, serialisiert. Die Komponente des PRAM-Datenbanksystems, welche die Sperranforderungen der Transaktionen bearbeitet, heißt *Concurrency-Control* (CC).

Auf der PRAM ist es durch den gemeinsamen Speicher möglich, daß jeder Prozessor die eigenen Sperranforderungen selbst bearbeitet. Die nötige Kommunikation zwischen den konkurrierenden Prozessoren findet über den gemeinsamen Speicher statt. Es ist also kein dedizierter Concurrency-Control–Prozessor nötig, wie dies bei vielen anderen Architekturen der Fall ist. Wenn jeder Prozessor seine Sperranforderungen selbst bearbeitet, spricht man gelegentlich von *autonomous locking* [MR94].

Um unnötige Serialisierung zum Beispiel bei gleichzeitigen lesenden Zugriffen zu vermeiden, gibt es verschiedene Sperrtypen, zum Beispiel *shared-* und *exclusive-*Sperren. Zu welchem Zeitpunkt welcher Sperrtyp angefordert werden muß und wann die Sperren wieder freigegeben werden, bestimmt das *Sperrprotokoll*. Das beim PRAM-Datenbanksystem verwendete Sperrprotokoll wird in Abschnitt 5.1 erläutert.

In den Abschnitten 5.2 und 5.3 wird die SB-PRAM–Implementierung der Concurrency-Control erklärt. Die dort gezeigte Implementierung hat gegenüber der üblichen Implementierung [Spe97, GR93] erhebliche Vorteile, die in Abschnitt 5.4 erläutert werden. Diese Implementierung läßt sich leicht auf andere Shared-Memory-Systeme mit Fetch-and-Add portieren.

In den meisten Datenbanksystemen, so auch beim PRAM-DBS, können zyklische Wartesituationen, sogenannte *Deadlocks*, entstehen. In Abschnitt 5.5 wird dieser Begriff definiert und die Concurrency-Control um eine Deadlock-Erkennung erweitert. Die Korrektheit der Deadlock-Erkennung wird bewiesen.

Teile dieses Kapitels sind für die Konferenz EURO-PAR 99 angenommen und werden als [JL99] veröffentlicht.

5.1 Sperrprotokoll

In diesem Abschnitt soll das für das SB-PRAM–Datenbanksystem verwendete Sperrprotokoll erläutert werden. Eine ausführliche Betrachtung der hier verwendeten Begriffe findet man in [GR93, Kapitel 7].

Im PRAM-Datenbanksystem wird das 2-Phasen-Sperrprotokoll (Two-Phase-Locking, 2PL) verwendet. Die Abarbeitung einer Transaktion wird in zwei Phasen aufgeteilt:

1. In der ersten Phase werden die benötigten Sperren angefordert, und die Transaktion wird ausgeführt. Es werden keine erhaltenen Sperren freigegeben.

2. Erst in Phase zwei werden alle in Phase eins erhaltenen Sperren freigegeben.

Die Aufteilung einer Transaktion in zwei Phasen hat nichts mit der Aufteilung einer Sperranforderung in zwei Phasen zu tun (vgl. Abschnitt 3.6.3). Man kann beweisen, daß bei Einhaltung des 2-Phasen-Sperrprotokolls das Isolations-Prinzip (siehe Seite 9) gilt [GR93].

Weiterhin wird im PRAM-Datenbanksystem das Konzept des *Granular Locking* umgesetzt. Dies bedeutet, daß Sperren auf unterschiedlichen Ebenen, genannt *Granulate*, erworben werden könnnen. Für das PRAM-DBS werden die Granulate *Relation, Seite, Tupel* unterstützt (siehe Abbildung 5.1).
Durch die Möglichkeit, Sperren auf verschiedenen Granulaten zu setzen, können einfache Transaktionen gezielt die benötigten Tupel oder Seiten sperren. Dadurch wird eine hohe Parallelität erreicht, da nur der benötigte Teil der Daten blockiert wird. Umgekehrt können jedoch auch sehr komplexe Transaktionen, die viele Millionen Tupel verarbeiten, eine ganze Relation durch eine einzige Sperre sperren. Dabei muß jedoch darauf geachtet werden, daß keine anderen Sperren auf feinerem Granulat übergangen werden.

Beispiel 1: Ein Prozessor P_1 kann durch eine einzige Sperre die Relation R exklusiv sperren, während ein anderer Prozessor P_2 ein Tupel aus dieser Relation R gesperrt hält. Somit haben sowohl P_1 als auch P_2 Zugriff auf die Relation, obwohl P_1 eine exklusive Sperre gesetzt hat. Prozessor P_1 hat die Sperre von P_2 "übersehen".

Um solche Situationen zu vermeiden, muß ein Prozessor P bei einer Sperranforderung auf ein bestimmtes Granulat bereits alle gröberen Granulate gesperrt haben. Im Beispiel muß P_2 dann neben der Sperre auf dem Tupel t auch eine Sperre auf der Seite halten, in der t gespeichert wird. Weiterhin muß P_2 auch eine Sperre auf der Relation selbst halten. Wenn dann P_1 eine exklusive Sperre auf R anfordert, muß P_1 warten, da P_2 selbst eine Sperre auf R hält.

Für Granular Locking reichen die zwei Sperrmodi *shared* und *exclusive* nicht aus. Denn würden nur diese zwei Sperrmodi benutzt, so müßte für jede exklusive Sperre auf Tupel-Ebene auch die gesamte Relation exklusiv gesperrt

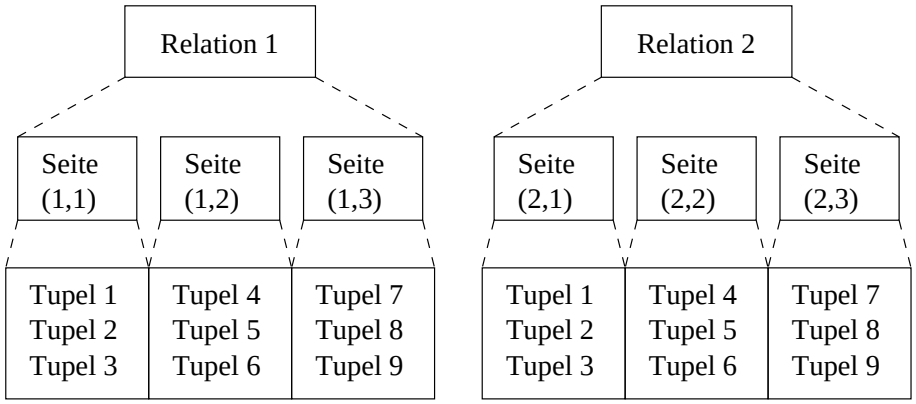

Abbildung 5.1: Granular Locking

Möchte ein Prozessor das Tupel 2 der Relation 1 exklusiv sperren, muß er vorher eine IX-Sperre auf Relation 1 und Seite (1,1) setzen.

werden. Dadurch würde die parallele Ausführung von Transaktionen praktisch vollständig verhindert.
Um bei Granular Locking weiterhin parallele Ausführung mehrerer Transaktionen zu ermöglichen, werden daher neben den schon angesprochenen *shared*- und *exclusive*-Sperren sogenannte *Warnsperren* (*Intention Locks*) eingeführt. Die Warnsperren zeigen auf grobem Granulat an, daß auf feinerem Granulat Sperren gehalten werden. Warnsperren sind untereinander kompatibel, jedoch zu anderen Sperrmodi (zum Beispiel *exclusive*) inkompatibel (s.u.).
Um auf feinem Granulat eine Sperre zu erhalten, müssen *top-down* zunächst alle gröberen Granulate mit geeigneten Warnsperren gesperrt werden (siehe Abbildung 5.1). Diese Warnsperren werden entsprechend dem 2-Phasen-Sperrprotokoll bis zum Transaktionsende gehalten. Bei der Freigabe von Sperren werden die Sperren umgekehrt *bottom-up* von feinem zu grobem Granulat freigegeben. Im PRAM-Datenbanksystem werden folgende Sperrmodi unterstützt:

X Eine *eXclusive*-Sperre erlaubt lesenden und schreibenden Zugriff auf alle Daten des betreffenden Granulats sowie der feineren Granulate.

S Eine *Shared*-Sperre erlaubt nur lesenden Zugriff auf die Daten des betreffenden und der feineren Granulate.

IS Eine *Intentional-Shared*-Sperre auf einem Granulat erlaubt die Anforderung von weiteren IS- oder S-Sperren auf nächst-feinerem Granulat. Auf dem aktuellen Granulat sind weder lesende noch schreibende Zugriffe erlaubt.

IX Eine *Intentional-eXclusive*-Sperre erlaubt die Anforderung von weiteren Sperren beliebigen Typs auf nächst-feinerem Granulat. Auf dem aktuellen Granulat sind weder lesende noch schreibende Zugriffe erlaubt.

SIX Eine *Shared & Intentional-eXclusive*-Sperre erlaubt lesenden Zugriff auf alle Daten des aktuellen Granulats sowie den Erwerb von beliebigen anderen Sperren auf nächst-feinerem Granulat.

Umgekehrt heißt das, daß vor dem Anfordern einer X-, IX- oder SIX-Sperre das nächst-gröbere Granulat bereits IX- oder SIX-gesperrt sein muß. Bevor eine S- oder IS-Sperre angefordert werden darf, muß auf nächst-gröberem Granulat eine IX- oder IS-Sperre gesetzt sein. Alle Daten, die eine Transaktion liest, müssen X-, S- oder SIX-gesperrt werden. Zu verändernde Daten müssen X-gesperrt werden.

Tabelle 5.1 zeigt die *Kompatibilitätsmatrix* der fünf im PRAM-Datenbanksystem unterstützten Sperrmodi.

	vorhandener Sperrtyp					
geforderter Sperrtyp	frei	X	S	IS	IX	SIX
X	+	−	−	−	−	−
S	+	−	+	+	−	−
IS	+	−	+	+	+	+
IX	+	−	−	+	+	−
SIX	+	−	−	+	−	−

Tabelle 5.1: Kompatibilitätmatrix: + bedeutet kompatibel, − inkompatibel

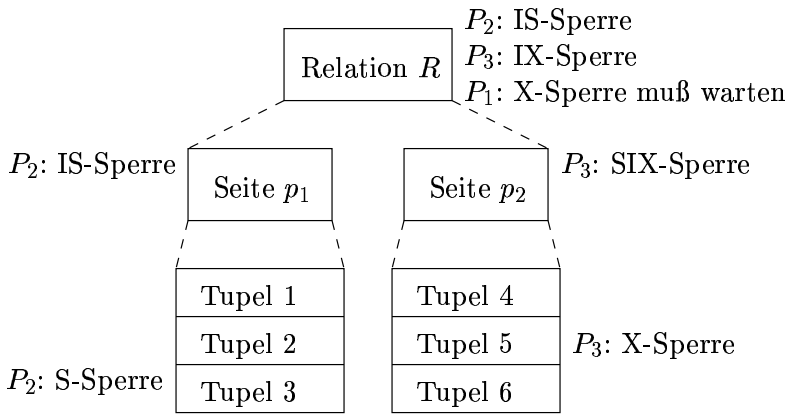

Abbildung 5.2: Mögliche Situation bei Granular Locking

Beispiel 2 Prozessor P_2 möchte das Tupel 3 der Relation R im Modus S sperren (siehe Abbildung 5.2). Dazu fordert er zunächst eine IS-Sperre auf R und auf der Seite p_1 an, in der das Tupel gespeichert wird. Danach kann P_2 ein S-Sperre auf Tupel 3 setzen.

Sei p_2 eine weitere Seite. Ein Prozessor P_3 sperrt zunächst die Relation R im Modus IX; diese Warnsperre ist kompatibel zur Warnsperre von P_2 (Modus IS). Anschließend sperrt P_3 die Seite p_2 im Modus SIX. Somit

hat P_3 lesenden Zugriff auf alle Tupel in Seite p_2. Die SIX-Sperre erlaubt P_2, nach einem bestimmten Tupel zu suchen und dieses im Modus X zu sperren, um es zu verändern. P_3 sperrt zum Beispiel Tupel 5.

Anschließend versucht Prozessor P_1 wie in Beispiel I, die gesamte Relation R exklusiv zu sperren. P_2 und P_3 halten schon Sperren auf R. Da die Sperrmodi IS und IX zu X inkompatibel sind, kann P_1 die X-Sperre auf R nicht sofort setzen. P_1 muß warten, bis P_2 und P_3 ihre Sperren auf R freigeben. Im Gegensatz zu Beispiel 2 "übersieht" P_1 also keine Sperren.

5.2 Die CC-Sperre

Die Standard-Sperren (einfache Sperre, Gruppensperre, Reader/Writer-Lock) aus [Röh96], die in Abschnitt 3.6 vorgestellt wurden, unterstützen die oben beschriebenen Sperrmodi mit der Kompatibilitätsmatrix 5.1 nicht direkt. Man kann jedoch aus mehreren Standard-Sperren eine Sperre zusammensetzen, die die für die Concurrency-Control geforderten Modi unterstützt. Dies wird im folgenden getan. Das Zusammensetzen dieser sogenannten CC-Sperre aus mehreren Standard-Sperren ist eine neue Idee, die gegenüber der herkömmlichen Implementierung der CC-Sperre große Vorteile hat (vgl. Abschnitt 5.4). Wir werden die Korrektheit und Fairneß der neuen CC-Sperre beweisen.

5.2.1 Implementierung der CC-Sperre

Sei o ein Objekt. Unser Ziel ist es, eine Sperre für o zu entwickeln, die die Sperrmodi und die Kompatibilitätsmatrix des vorherigen Abschnitts unterstützt.
Betrachten wir zunächst die X-Sperre gegenüber den anderen Sperrmodi. Ist o X-gesperrt kann keine andere Sperre, auch keine weitere X-Sperre, auf o vergeben werden. Sei nun andererseits o mit einer φ-Sperre, $\varphi \neq X$, belegt. Dann kann keine X-Sperre auf o gesetzt werden. Man kann demnach X-Sperren und sonstige Sperren durch einen Reader/Writer–Lock rw trennen. Er erlaubt entweder eine X-Sperre (Writer) oder mehrere sonstige Sperren (Reader). Im folgenden müssen wir uns daher nur noch um die Kompatibilität von nicht-X-Sperren kümmern.
Streichen wir also aus der Kompatibilitätsmatrix die X-Zeile und die X-Spalte. Dann sieht man, daß alle Sperrmodi kompatibel zu einer schon vorhanden IS-Sperre sind. Ebenso ist eine angeforderte IS-Sperre zu allen schon vergebenen Sperren auf o kompatibel. Wir können demnach auch die IS-Zeile und die IS-Spalte aus der Matrix streichen.

Wir haben bisher schon einiges erreicht: um eine X-Sperre auf dem Objekt o zu erhalten, muß lediglich eine Schreib-Sperre auf rw gesetzt werden. Um eine IS-Sperre zu erwerben, muß eine Lese-Sperre auf rw gesetzt werden. Es bleiben die Fälle S, IX, SIX-Sperre zu behandeln. Diese erwerben zunächst ebenfalls eine Lese-Sperre auf rw. Im folgenden wird gezeigt, wie die Kompatibilität zwischen diesen drei Sperrmodi nach der rw-Sperre überprüft wird.

Die drei verbleibenden Sperrmodi sind paarweise inkompatibel. Jedoch können gleichzeitig mehrere S-Sperren vergeben werden; ebenso können mehrere IX-Sperren vergeben werden. Wir trennen die drei Sperrmodi zunächst mit einer 3-Gruppensperre gr, wobei jede Gruppe einem der drei Sperrmodi zugeordnet wird ($S \hat{=} 1, IX \hat{=} 2, SIX \hat{=} 3$).
Möchte also ein Prozessor auf das Objekt o eine S-Sperre setzen, so fordert er zunächst die Lese-Sperre auf rw und anschließend eine Gruppe-1-Sperre auf gr. Analog wird eine IX-Sperre ausgeführt. Da bei einer Gruppensperre Sperren der gleichen Gruppe untereinander kompatibel sind, haben wir die Behandlung von S- und IX-Sperren abgeschlossen.

Es bleibt als letztes Problem, untereinander inkompatible SIX-Sperren zu trennen. Wir können mehrere SIX-Sperren durch eine einfache Sperre el trennen. Fordert also ein Prozessor eine SIX-Sperre auf o, wird zunächst rw lesegesperrt, danach eine Gruppe-3-Sperre auf gr gesetzt, und abschließend werden die eventuell konkurrierenden SIX-Sperren durch el serialisiert.

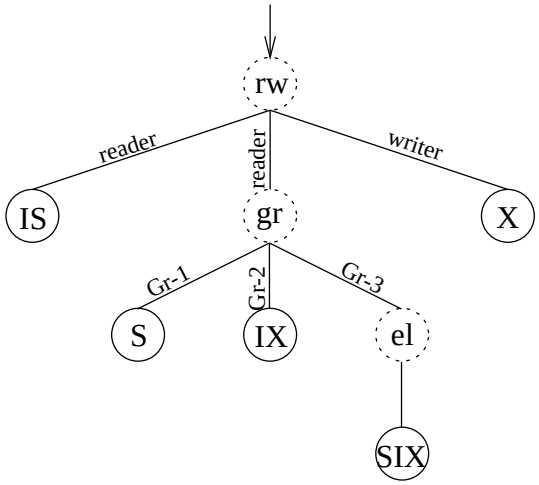

Abbildung 5.3: Sperr-Baum

Insgesamt ergibt sich ein *Sperr-Baum* (siehe Abbildung 5.3), dessen Blätter mit Sperrmodi beschriftet sind, an dessen inneren Knoten verschiedene Standard-Sperren aus [Röh96] sitzen, und dessen Kanten mit den verschiedenen Sperrmodi dieser Standard-Sperren beschriftet sind. Um nun das Objekt o im Modus φ zu sperren, wird von der Wurzel bis zum Blatt φ jede auf diesem Pfad liegende Standard-Sperre im entsprechenden Modus gesperrt. Um eine Sperre wieder freizugeben, werden in umgekehrter Reihenfolge vom Blatt bis zur Wurzel die entsprechenden Standard-Sperren freigegeben.

Definition 5.1 (CC-Sperre) Eine **CC-Sperre** ist eine Struktur mit folgenden Komponenten:

rw: Reader/Writer-Lock zur Trennung von X- und sonstigen Sperren.

gr: 3-Gruppensperre zur Trennung von S, IX, SIX-Sperren.

el: Einfache Sperre zur Trennung mehrerer SIX-Sperren.

Es sind die Funktionen cc_lock(l, m) und cc_unlock(l, m) definiert, welche auf einer übergebenen CC-Sperre l eine m-Sperre (m$\in \{X, S, IS, IX, SIX\}$) setzen beziehungsweise freigeben (vgl. auch Abschnitt 5.5).

Satz 5.2 (Korrektheit) Sei l eine CC-Sperre. Dann sind zu keinem Zeitpunkt zueinander inkompatible Sperrmodi auf l vergeben.
Beweis: Wir führen einen Widerspruchsbeweis. Seien φ und ψ zueinander inkompatible Sperrmodi, die gleichzeitig auf einer CC-Sperre vergeben sind.

1. Fall $\varphi = X, \psi \neq X$: dann muß der Reader/Writer-Lock rw gleichzeitig eine Lese- und eine Schreib-Sperre vergeben haben.
2. Fall $\varphi = X, \psi = X$: dann hat rw mehrere Schreib-Sperren gleichzeitig vergeben.
3. Fall $\varphi = S, \psi \in \{IX, SIX\}$: dann hat die Gruppensperre gr gleichzeitig eine Gruppe-1-Sperre und eine Gruppe-2/3-Sperre vergeben. Analog für $\varphi = IX, \psi = SIX$.
4. Fall $\varphi = SIX, \psi = SIX$: dann hat die einfache Sperre el mehrere Sperren gleichzeitig vergeben.

Alle Fälle führen auf einen Widerspruch, womit die Aussage des Satzes bewiesen ist. ■

5.2.2 Fairneß der CC-Sperre

In Kapitel 3 habe wir Sperren *fair* genannt, wenn jede angeforderte Sperre nach endlicher Zeit vergeben wird. In diesem Abschnitt wollen wir weitergehende Aussagen über die Fairneß der CC-Sperre machen. Wir werden eine obere Schranke für die Anzahl der Sperranforderungen angeben, die eine gegebene Sperranforderung "überholen" können. Dazu werden zunächst einige Fairneß-Kriterien über die Standard-Sperren vorausgesetzt. Diese Kriterien gelten für die SB-PRAM–Implementierungen nach [Röh96] oder lassen sich direkt aus den dort angegebenen Analysen und Quelltext-Fragmenten ableiten. Aus diesen Kriterien leiten wir weitere Eigenschaften der Standard-Sperren her, die sich nicht offensichtlich aus [Röh96] ergeben. Ab Seite 44 werden wir die Erkenntnisse über die Standard-Sperren auf die CC-Sperre anwenden. Mit Satz 5.15, der abschließend Auskunft über die maximal mögliche Anzahl von "Überholungen" in der CC-Sperre gibt, werden wir die Diskussion abschließen.

Notation: Im folgenden werden wir häufig über verschiedene Ausführungen von Operationen argumentieren. Um diese Ausführungen unterscheiden zu können, versehen wir sie mit symbolischen Namen. So sagen wir zum Beispiel "Seien l und l' Ausführungen der Funktion simple_lock()". Weiterhin werden wir im folgenden nicht zwischen Sperranforderungen und der zugehörigen Ausführung der Sperrfunktion unterscheiden.

Alle in dieser Arbeit behandelten Sperrtypen außer der einfachen Sperre unterstützten mehrere Sperrmodi. Wir werden eine Sperranforderung l für den Modus φ mit l^{φ} bezeichnen, wenn wir den Modus verdeutlichen wollen.

Es sei an die Bedeutung von V-Zeitpunkten und die Notation $t_B^V(\mathcal{I})$, $t_E^V(\mathcal{I})$ erinnert (Abschnitt 3.3, Seite 14).

Voraussetzung: Zu jeder Sperrausführung l muß eine eindeutig bestimmte Freigabe $u(l)$ der erhaltenen Sperre existieren. $u(l)$ darf nicht begonnen werden, bevor l beendet wurde. Dies wird vom Datenbanksystem garantiert.

Ist L eine Sperre (einfache Sperre, R/W-Lock, Gruppensperre oder CC-Sperre), und hat ein Prozessor P eine Sperranforderung l auf L beendet, so beginnt P die nächste Sperranforderung auf L erst nach dem Ende von $u(l)$. Dies wird durch eine *Double-Lock–Erkennung* garantiert (siehe Abschnitt 5.3.2).

Bei den folgenden Betrachtungen wird davon ausgegangen, daß die Programmausführung nicht von asynchronen Interrupts unterbrochen wird. Daher können wir von einer synchronen Abarbeitung der Sperranforderungen ausgehen (vgl. [Röh96, Seite 27f]).

Definition 5.3 (Überholen) Seien l_1 und l_2 Sperranforderungen. Wir sagen "l_2 **überholt** l_1", wenn $t_B^V(l_1) < t_B^V(l_2) \ \wedge \ t_E^V(l_1) > t_E^V(l_2)$.

Folgendes Kriterium wird in [Röh96, Anhang E] bewiesen:

Kriterium 5.4 (Fairneß der Einfachen Sperre) Seien l_1 und l_2 verschiedene Sperranforderungen an eine einfache Sperre. Dann gilt das Fairneß-Kriterium

$$t_B^V(l_1) < t_B^V(l_2) \Longrightarrow t_E^V(l_1) < t_E^V(l_2).$$

Sperren werden also in der Reihenfolge ihrer Anforderung vergeben — es gibt keine Überholungen.

Definition 5.5 (Sequenzen) Sei l eine n-Gruppensperre. Wir numerieren die group_lock(l, φ_i)-Ausführungen $l_i^{\varphi_i}$ $(i \geq 1, 1 \leq \varphi_i \leq n)$ gemäß der Reihenfolge ihres Beginns, also $\forall i : t_B^V(l_i) < t_B^V(l_{i+1})$. Eine **$\varphi$-Sequenz** S^{φ} ist eine maximal große Menge aufeinanderfolgend begonnener group_lock(l, φ)-Ausführungen, die zur selben Gruppe gehören (siehe Abbildung 5.4, 1.); formal:

$$S^{\varphi} = \left\{ l_s^{\varphi_s}, l_{s+1}^{\varphi_{s+1}}, \ldots, l_{s+u}^{\varphi_{s+u}} \right\}, s \geq 1, u \geq 0$$
$$\varphi = \varphi_s = \varphi_{s+1} = \ldots = \varphi_{s+u}$$
$$\varphi_{s-1} \neq \varphi \text{ (falls } s \neq 1), \varphi_{s+u+1} \neq \varphi$$

Wir definieren den Anfangszeitpunkt einer Sequenz als $t_{\text{anf}}(S) := \min_{l \in S} \ t_B^V(l)$. Analog definieren wir den Sequenzen-Begriff für Reader/Writer-Locks und die CC-Sperre.

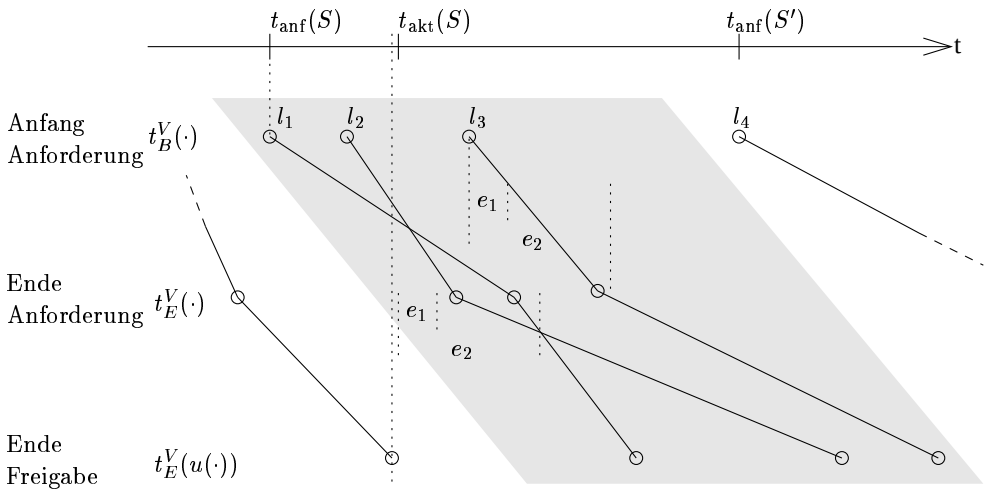

Abbildung 5.4: Beispiel Sequenzen

1. Das Bild zeigt grau hinterlegt eine Sequenz $S = \{l_1, l_2, l_3\}$. Der Anfangszeitpunkt t_{anf} ist gleich dem Anfangszeitpunkt $t_B^V(l_1)$ von l_1.
2. Die Anforderungen l_1 und l_2 werden begonnen, während noch die vorherige Sequenz aktiv ist. Die Sequenz wird zum Zeitpunkt t_B aktiviert — eine V-Zeiteinheit nach dem Ende der letzten Sperrfreigabe der vorherigen Sequenz.
3. Die Anforderungen l_1 und l_2 brauchen vom Aktivierungszeitpunkt an mindestens e_1 und höchstens e_2 Befehle. Die Anforderung l_3 ist die erste Sperranforderung in S, die nach dem Aktivierungs-Zeitpunkt begonnen wird. Daher ist $k = 3$. Vom Beginn der Anforderung l_3 an benötigt l_3 mindestens e_1 und höchstens e_2 Befehle bis zum Ende.

   Die nächste Sequenz beginnt mit der Anforderung l_4, noch bevor alle Sperren von S wieder freigegeben worden.

Folgendes Kriterium wird in [Röh96, S. 39] genannt:

Kriterium 5.6 (Sequenzen-Fairneß der Gruppensperre) Seien l_1^φ und l_2^ψ Sperranforderungen an eine Gruppensperre, $\varphi \neq \psi$. Dann gilt das Fairneß-Kriterium

$$t_B^V(l_1) < t_B^V(l_2) \Longrightarrow t_E^V(l_1) < t_E^V(l_2) \tag{5.1}$$

Dieses Kriterium besagt, daß eine Sperranforderung aus einer Sequenz nicht von einer Sperranforderung einer späteren Sequenz überholt wird.

Mit der Sequenzen-Fairneß können wir nun den Aktivierungs-Zeitpunkt einer Sequenz definieren:

Definition 5.7 (Aktivierung) Seien sämtlichen Sperranforderungen an eine Gruppensperre in Sequenzen $S_i, i \geq 1$ aufgeteilt. Die Sequenzen seien ihren Anfangszeitpunkten t_{anf} nach geordnet. Zu jeder Sperranforderung l gehört eine Freigabe $u(l)$. Wir definieren den **Aktivierungs-Zeitpunkt** $t_{\text{akt}}(S_i)$ der

Sequenz S_i (siehe Abbildung 5.4, 2.):

$$\begin{aligned} t_{\text{akt}}(S_1) &:= 0 \\ t_{\text{akt}}(S_i) &:= \max\left\{ t_E^V(u(l)) \;\middle|\; l \in S_{i-1} \right\} + 1 \quad \text{für } i > 1 \end{aligned}$$

Eine Sequenz wird also *aktiviert*, wenn jede Sperre der Vorgänger-Sequenz freigegeben wurde. Man beachte, daß nach dieser Definition eine Sequenz aktiviert werden kann, noch bevor die erste Sperranforderung der Sequenz begonnen wurde ($t_{\text{akt}}(S) < t_{\text{anf}}(S)$).
Eine Sequenz S_i heißt **aktiv** zum V-Zeitpunkt t, wenn ihr Aktivierungs-Zeitpunkt vor t liegt ($t_{\text{akt}}(S_i) \leq t$), und die nachfolgende Sequenz S_{i+1} noch nicht aktiviert wurde ($t_{\text{akt}}(S_{i+1}) > t$).

Die Erfüllung des folgenden Kriteriums ergibt sich für die SB-PRAM aus den Quelltexten in [Röh96, Abschnitt 4.2.3.2] und [JLS99]:

Kriterium 5.8 (Fairneß der Gruppensperre innerhalb einer Sequenz) Sei C die Anzahl der Prozessoren. Sei $S = \{l_1, l_2, \ldots, l_n\}$ eine Sequenz von Sperranforderungen an eine Gruppensperre, seien die einzelnen Sperranforderungen zeitlich geordnet ($t_B^V(l_i) < t_B^V(l_{i+1})$). Sei weiterhin

$$k := \min\left\{ i \,\middle|\, l_i \in S, t_B^V(l_i) > t_{\text{akt}}(S) \right\} \tag{5.2}$$

der minimale Index der Sperren in der Sequenz S, die nach dem Aktivierungs-Zeitpunkt $t_{\text{akt}}(S)$ angefordert wurden. Falls k undefiniert ist, setze $k := n + 1$. Dann gilt für die Gruppensperre das Fairneß-Kriterium

$$\exists\, e_1, e_2 \in \mathbb{N}, 0 < e_1 < e_2 : \left.\begin{array}{ll} \text{(i)} & \forall i < k \;:\; t_{\text{akt}}(S) + C \cdot e_1 \;\leq\; t_E^V(l_i) \;\leq\; t_{\text{akt}}(S) + C \cdot e_2 \\ & \text{Alle Sperranforderungen, die vor dem Aktivierungs-Zeitpunkt} \\ & \text{begonnen wurden, benötigen nach der Aktivierung der Se-} \\ & \text{quenz mindestens } e_1 \text{ und höchstens } e_2 \text{ Instruktionen bis zum} \\ & \text{Ende der Sperranforderung.} \\ \text{(ii)} & \forall i \geq k \;:\; t_B^V(l_i) + C \cdot e_1 \leq t_E^V(l_i) \leq t_B^V(l_i) + C \cdot e_2 \\ & \text{Alle Sperranforderungen, die nach dem Aktivierungs-Zeit-} \\ & \text{punkt begonnen wurden, benötigen mindestens } e_1 \text{ und höch-} \\ & \text{stens } e_2 \text{ Instruktionen bis zum Ende.} \\ \text{(iii)} & k < C \\ & \text{Weniger als } C \text{ Anforderungen werden vor dem Aktivierungs-} \\ & \text{Zeitpunkt begonnen.} \\ \text{(iv)} & e_1 \text{ ist eine untere Schranke für die Anzahl der Instruktionen} \\ & \text{einer } \mathsf{group\_lock()}\text{-Ausführung. Dies folgt bereits aus (i) und} \\ & \text{(ii).} \end{array}\right\} \tag{5.3}$$

Das folgende Lemma ist offensichtlich:

Lemma 5.9 1. Sei r eine untere Schranke für die Anzahl der Instruktionen, die ein Prozessor P für eine group_lock()-Ausführung abarbeiten muß (R-Zeitdauer). Sei Δ die Länge eines R-Zeitintervalls. In diesem R-Zeitintervall kann P höchstens $\lceil \Delta/r \rceil$ group_lock()-Ausführungen beginnen.

2. Ist C die Anzahl der Prozessoren, so können alle Prozessoren zusammen in einem R-Zeitintervall der Länge Δ höchstens $C \cdot \lceil \Delta/r \rceil$ group_lock()-Ausführungen beginnen.

Satz 5.10 (Fairneß der Gruppensperre innnerhalb einer Sequenz)
Sei wieder $S = \{l_1, l_2, \ldots, l_n\}$ eine Sequenz von zeitlich geordneten Sperranforderungen an eine Gruppensperre. Sei

$$k := \min \left\{ i \,|\, l_i \in S, t_B^V(l_i) > t_{\text{akt}}(S) \right\}$$

wie in Kriterium 5.8. Seien e_1 und e_2 die Konstanten aus Kriterium 5.8. Dann gilt für alle Sperranforderungen $l_i \in S$:

$$\forall\, j \text{ mit } n \geq j > i + C \left\lceil \frac{e_2}{e_1} \right\rceil \;:\; t_E^V(l_j) > t_E^V(l_i).$$

Eine Anforderung $l_i \in S$ kann also nur von den Anforderungen l_j mit $i < j \leq i + C\,\lceil e_2/e_1 \rceil$ überholt werden.
Beweis: Setzen wir $d_1 := C \cdot e_1, d_2 := C \cdot e_2$.
Wir bezeichnen den minimal und maximal möglichen Endzeitpunkt von $l_j \in S$ mit

$$t_{\min}(l_j) := \begin{cases} t + d_1 \text{ falls } j < k \\ t_B^V(l_j) + d_1 \text{ falls } j \geq k \end{cases} \tag{5.4}$$

$$t_{\max}(l_j) := t_{\min}(l_j) - d_1 + d_2 \tag{5.5}$$

Mit diesen Definitionen gilt für alle $l_j \in S$ nach Kriterium 5.8

$$t_{\min}(l_j) \leq t_E^V(l_j) \leq t_{\max}(l_j). \tag{5.6}$$

Wir stellen für alle j fest:

$$t_{\min}(l_j) \leq t_{\min}(l_{j+1}),$$

und damit durch Induktion

$$j \leq j' \implies t_{\min}(l_j) \leq t_{\min}(l_{j'}). \tag{5.7}$$

Betrachten wir ein beliebiges aber für den Rest des Beweises festes $l_i \in S$. Eine Sperranforderung $l_j \in S, j > i$, kann l_i höchstens überholen, wenn $t_{\min}(l_j) < t_{\max}(l_i)$ gilt. Denn sonst ist wegen (5.6)

$$t_E^V(l_i) \leq t_{\max}(l_i) \leq t_{\min}(l_j) \leq t_E^V(l_j),$$

womit l_j die Anforderung l_i nicht überholt (siehe Definition 5.3). Betrachten wir also die Menge

$$U := \{l_j \in S | j \geq i \text{ und } t_{\min}(l_j) < t_{\max}(l_i)\}.$$

Höchstens die Sperranforderungen in U können l_i überholen. Es ist $l_i \in U$, und für alle j, j' mit $j' > j > i$ gilt

$$l_j \notin U \implies l_{j'} \notin U, \tag{5.8}$$

denn sonst wäre

$$t_{\min}(l_{j'}) \overset{(5.7)}{\geq} t_{\min}(l_j) \overset{l_j \notin U}{\geq} t_{\max}(l_i) \overset{l_{j'} \in U}{>} t_{\min}(l_{j'}).$$

Aus (5.8) folgt

$$\exists\, j_{\max} \quad : \quad U = \{l_i, l_{i+1}, \ldots, l_{j_{\max}}\}.$$

Insgesamt gilt damit

$$\forall\, j, n \geq j > j_{\max} \quad : \quad t_E^V(l_j) > t_E^V(l_i).$$

Es ist $j_{\max} = i + |U| - 1$. Wir vervollständigen den Beweis, indem wir $|U| \leq C \cdot \lceil e_2/e_1 \rceil + 1$ zeigen:

$$\begin{aligned}
|U| &= |\{l_j \in S | j \geq i \text{ und } t_{\min}(l_j) < t_{\max}(l_i)\}| \\
&= |\{l_j \in S | j > i \text{ und } t_{\min}(l_j) < t_{\max}(l_i)\}| + 1 \\
&\overset{(5.7)}{\leq} |\{l_j \in S | t_{\min}(l_i) \leq t_{\min}(l_j) < t_{\max}(l_i)\}| + 1 \\
&\leq k + |\{l_j \in S | j \geq k \text{ und } t_{\min}(l_i) \leq t_{\min}(l_j) < t_{\max}(l_i)\}| + 1 \\
&\overset{(5.4)}{=} k + \left|\left\{l_j \in S | j \geq k \text{ und } t_{\min}(l_i) \leq t_B^V(l_j) + d_1 < t_{\max}(l_i)\right\}\right| + 1 \\
&\overset{(5.5)}{=} k + \left|\left\{ l_j \in S \;\middle|\; \begin{array}{l} j \geq k \text{ und} \\ t_{\min}(l_i) \leq t_B^V(l_j) + d_1 < t_{\min}(l_i) - d_1 + d_2 \end{array} \right\}\right| + 1 \\
&= k + \left|\left\{ l_j \in S \;\middle|\; \begin{array}{l} j \geq k \text{ und} \\ \underbrace{t_{\min}(l_i) - d_1 \leq t_B^V(l_j) < t_{\min}(l_i) - 2d_1 + d_2}_{\Delta} \end{array} \right\}\right| + 1 \\
&\leq k + C \cdot \left\lceil \frac{e_2 - e_1}{e_1} \right\rceil + 1,
\end{aligned}$$

da im V-Zeitintervall $\Delta = \left[t_{\min}(l_i) - d_1, \ldots, t_{\min}(l_i) - 2d_1 + d_2\right)$ der R-Länge $(d_2 - d_1)/C = e_2 - e_1$ nach Lemma 5.9 und (5.3, iv) höchstens $C \cdot \left\lceil \frac{e_2 - e_1}{e_1} \right\rceil$ Sperranforderungen beginnen können. Beachtet man nun noch, daß $k < C$ (5.3,iii), so erhält man

$$\begin{aligned}
|U| &\leq C + C \cdot \left\lceil \frac{e_2}{e_1} - 1 \right\rceil + 1 \\
&= C \cdot \left\lceil \frac{e_2}{e_1} \right\rceil + 1
\end{aligned}$$

wie behauptet. ■

Aus dem Satz und Kriterium 5.6 ergibt sich das

Korollar 5.11 (Fairneß der Gruppensperre) Sei $l_1, l_2, \ldots$ eine zeitlich geordnete Folge von Sperranforderungen an eine Gruppensperre. Dann gilt für

alle i und j

$$j > i + C \cdot \left\lceil \frac{e_2}{e_1} \right\rceil \Longrightarrow t_E^V(l_j) > t_E^V(l_i).$$

Eine Sperranforderung l an eine Gruppensperre wird also von höchstens $C \cdot \left\lceil \frac{e_2}{e_1} \right\rceil$ anderen Sperranforderungen überholt.

Kriterium 5.12 (Fairneß des Reader/Writer-Locks) Sei $l_1^{\varphi_1}, l_2^{\varphi_2}, \ldots$ ($\varphi_i \in$ {READER, WRITER}) eine zeitlich geordnete Folge von Sperranforderungen an einen Reader/Writer–Lock. Der Reader/Writer-Lock erfüllt folgende Fairneß-Kriterien:

1. Seien $l_i^{\varphi_i}$ und $l_j^{\varphi_j}$ Sperranforderungen für unterschiedliche Modi ($\varphi_i \neq \varphi_j$) und $i < j$. Dann gilt $t_E^V(l_i) < t_E^V(l_j)$ (Sequenzen-Fairneß).
2. Für eine Lese-Anforderung l_i gilt mit e_1 und e_2 aus Kriterium 5.8:

   $$\forall j > i + C \cdot \left\lceil \frac{e_2}{e_1} \right\rceil : t_E^V(l_i) < t_E^V(l_j)$$

   (Fairneß in einer READER-Sequenz).
3. Für l_i^{WRITER} und l_j^{WRITER} mit $i < j$ gilt $t_E^V(l_i) < t_E^V(l_j)$. Es wird also keine Schreib-Anforderung von einer anderen Schreib-Anforderung überholt.

Bemerkung: Daß die SB-PRAM–Implementierung des Reader/Writer-Locks diese Punkte erfüllt, folgt aus den Betrachtungen in [Röh96, Abschnitt 4.2.4] zusammen mit Satz 5.10 dieser Arbeit. Daß die Punkte 1. und 2. dieses Kriteriums an die Gruppensperre erinnern, ist kein Zufall: der Reader/Writer-Lock von Röhrig besteht hauptsächlich aus einer Gruppensperre.
Punkt 3. dieses Kriteriums wird von der Implementierung in [Röh96] nicht erfüllt. Die aktuelle (und bisher unveröffentlichte) SB-PRAM–Implementierung des Reader/Writer-Locks erfüllt diesen Punkt jedoch.

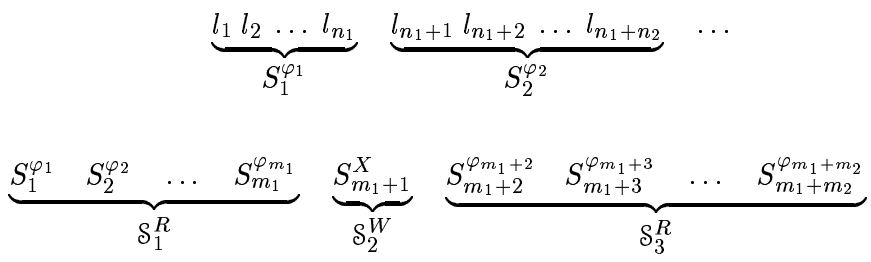

Abbildung 5.5: Sequenzen und Obersequenzen

Mit den vorgestellten Informationen über die Fairneß der Standard-Sperren gelingen nun Aussagen über die Fairneß der CC-Sperre. Dazu betrachten wir eine

Folge von Sperr-Anforderungen an eine CC-Sperre (siehe Abbildung 5.5). Die Anforderungen seien wieder V-zeitlich geordnet. Nach Definition 5.5 werden die aufeinanderfolgenden Sperr-Anforderungen gleichen Typs zu Sequenzen zusammengefaßt. Im weiteren werden wir Sequenzen von Sequenzen, sogenannte Obersequenzen, betrachten. Dazu fassen wir einerseits aufeinanderfolgende Sequenzen $S_i, S_{i+1}, \ldots$ der Typen $\varphi_i, \varphi_{i+1}, \ldots \in \{S, IX, IS, SIX\}$ zu einer Obersequenz $\mathbb{S}_j^R$ zusammen, und wir bezeichnen eine Sequenz S^X auch mit $\mathbb{S}^W$. Man beachte jedoch, daß sich die Indizes von S^X und der zugehörigen $\mathbb{S}^W$ unterscheiden können. Die Sperranforderungen in einer $\mathbb{S}^R$-Obersequenz sind genau die aufeinanderfolgenden Lese-Anforderungen an den Reader/Writer-Lock der CC-Sperre. Analog entsprechen die $\mathbb{S}^W$-Anforderungen den Schreib-Anforderungen an diesen Reader/Writer-Lock.

Mit dieser Notation läßt sich Kriterium 5.12 zu einer ersten Aussage über die Fairneß der CC-Sperre umformulieren:

Lemma 5.13 Sei $l_1, l_2, \ldots$ eine Folge von Sperranforderungen an eine CC-Sperre. Die Anforderungen werden zu Obersequenzen $\mathbb{S}_1^{\psi_1}, \mathbb{S}_2^{\psi_2} \ldots$ zusammengefaßt ($\psi_i \in \{R, W\}$). Nach Kriterium 5.12 erfüllt der Reader/Writer-Lock rw in der CC-Sperre folgendes Fairneß-Kriterium:

1. Eine Anforderung $l \in \mathbb{S}_i^{\psi_i}$ wird von keiner Anforderung $l' \in \mathbb{S}_j^{\psi_j}, j > i$, einer anderen Obersequenz überholt (Sequenzen-Fairneß).
2. Für eine Anforderung $l_j \in \mathbb{S}_i^R$ gilt:

$$\forall\, j' > j + C \left\lceil \frac{e_2}{e_1} \right\rceil \;:\; t_E^V(l_j) < t_E^V(l_{j'}).$$

3. Keine X-Anforderung wird von einer anderen X-Anforderung überholt.

Insbesondere gilt für die gesamte CC-Sperre, daß

4. keine X-Anforderung von einer nicht-X-Anforderung überholt wird und umgekehrt.
5. Keine X-Anforderung wird von einer anderen X-Anforderung überholt.

Im folgenden betrachten wir nur noch die Fairneß zwischen S-, IX- und SIX-Anforderungen. Da diese drei Sperrmodi zum Modus IS kompatibel sind, können wir Fairneß zwischen S,IX,SIX-Anforderungen und IS-Anforderungen vernachlässigen.

Lemma 5.14 Sei $l_1^{\varphi_1}, l_2^{\varphi_2}, \ldots$ eine Folge von zeitlich geordneten Sperranforderungen an eine CC-Sperre mit $\varphi_i \in \{\text{S, IX, SIX}\}$. Für alle i, j gilt:

$$j > i + 2 \cdot C \cdot \left\lceil \frac{e_2}{e_1} \right\rceil \;:\; t_E^V(l_i) < t_E^V(l_j).$$

Jede Anforderung l_i wird also von höchstens $2 \cdot C \cdot \left\lceil \frac{e_2}{e_1} \right\rceil$ Anforderungen $l_j, j > i$, überholt.

Beweis: Wir schreiben $t_B^V(l)$ für den Beginn und $t_E^V(l)$ für das Ende jeder Sperranforderung l in der ganzen CC-Sperre. Mit $t_B^V(rw(l))$, $t_B^V(gr(l))$ und mit $t_B^V(el(l))$ bezeichnen wir den Beginn der zu l gehörigen Anforderung an die Standard-Sperren rw, gr und el in der CC-Sperre. Analoge Bezeichnung benutzen wir für das Ende der Standard-Sperranforderungen.

Wir betrachten nun eine beliebige aber für den Rest des Beweises feste Anforderung l_i. Nach Lemma 5.13(2.) gilt

$$\begin{aligned} \forall\, j > i + C \cdot \left\lceil \frac{e_2}{e_1} \right\rceil \,:\, t_B^V(gr(l_i)) &= t_E^V(rw(l_i)) + C \\ &< t_E^V(rw(l_j)) + C \\ &= t_B^V(gr(l_j)). \end{aligned} \tag{5.9}$$

Man beachte hierfür, daß die Gruppensperre gr genau eine Runde (gleich C V-Zeitpunkte) nach dem Ende des Reader/Writer-Locks rw beginnt. Setze $i' := i + C \cdot \left\lceil \frac{e_2}{e_1} \right\rceil$. Dann sagt die Ungleichung (5.9) aus, daß die Anforderungen $l_{i'+1}, l_{i'+2}, \ldots$ die Gruppensperre gr nach l_i beginnen. Es ergibt sich im schlimmsten Fall also die Reihenfolge

$$\ldots, l_{i'-2}, l_{i'-1}, l_i, l_{i'}, l_{i'+1}, \ldots,$$

in der die Gruppensperre gr angefordert wird. Damit gilt nach Korollar 5.11

$$\forall\, j > i' + C \cdot \left\lceil \frac{e_2}{e_1} \right\rceil = i + 2 \cdot C \cdot \left\lceil \frac{e_2}{e_1} \right\rceil \,:\, t_E^V(gr(l_i)) < t_E^V(gr(l_j)). \tag{5.10}$$

Wir unterscheiden nun zwei Fälle, jenachdem, ob l_i eine SIX-Anforderung oder eine S- oder IX-Anforderung ist.

Fall 1: Ist l_i eine S- oder IX-Anforderung, so gilt $t_E^V(l_i) = t_E^V(gr(l_i))$, und deswegen und nach (5.10)

$$\forall\, j > i + 2 \cdot C \cdot \left\lceil \frac{e_2}{e_1} \right\rceil \,:\, t_E^V(l_i) < t_E^V(gr(l_j)) \leq t_E^V(l_j).$$

Dabei gilt in der rechts stehenden Ungleichung Gleichheit, falls l_j ebenfalls eine S- oder IX-Anforderung ist. Ist l_j eine SIX-Anforderung, so muß nach dem Ende der Gruppensperre gr noch die einfache Sperre el erhalten werden, weswegen dann rechts "echt kleiner" gilt.

Fall 2: Sei nun l_i eine SIX-Anforderung. Dann wird l_i bis zur Gruppensperre gr von höchstens $2 \cdot C \cdot \left\lceil \frac{e_2}{e_1} \right\rceil$ anderen Anforderungen überholt. Hat l_i die Gruppensperre beendet, so kann bis zur Freigabe von l_i keine S- oder IX-Sperre beendet werden, da diese die Gruppensperre wegen Inkompatibilität zu l_i nicht erhalten können. Demnach kann eine SIX-Anforderung nach der Gruppensperre gr von keiner weiteren S- oder IX-Anforderung überholt werden. Es gilt also für Anforderungen $l_j^{\varphi_j}$ mit $\varphi_j \in \{\text{S,IX}\}$ und $j > i + 2 \cdot C \cdot \left\lceil \frac{e_2}{e_1} \right\rceil$ nach (5.10)

$$t_E^V(gr(l_j)) \quad > \quad t_E^V(gr(l_i)),$$

und damit

$$t_E^V(l_j) = t_E^V(gr(l_j)) \overset{\text{inkomp.}}{>} t_E^V(gr(u(l_i))) > t_E^V(l_i). \tag{5.11}$$

Dabei bezeichnet $t_E^V(gr(u(l_i)))$ das Ende der Freigabe der Gruppensperre bei der Freigabe der SIX-Sperre l_i. Die rechte Ungleichung gilt, weil die Sperrfreigabe von l_i nicht vor dem Ende der Sperranforderung l_i beginnt.

Auch für eine SIX-Anforderung l_j mit $j > i + 2 \cdot C \cdot \left\lceil \frac{e_2}{e_1} \right\rceil$ gilt nach (5.10) $t_E^V(gr(l_i)) < t_E^V(gr(l_j))$, und daher

$$\begin{aligned} t_B^V(el(l_i)) &= t_E^V(gr(l_i)) + C \\ &< t_E^V(gr(l_j)) + C \\ &= t_B^V(el(l_j)), \end{aligned}$$

und mit Kriterium 5.4 gilt dann

$$\begin{aligned} t_E^V(l_i) &= t_E^V(el(l_i)) \\ &< t_E^V(el(l_j)) \\ &= t_E^V(l_j). \end{aligned} \tag{5.12}$$

Die Ungleichungen (5.11) und (5.12) zusammen ergeben

$$\forall\, j > i + 2 \cdot C \cdot \left\lceil \frac{e_2}{e_1} \right\rceil \quad : \quad t_E^V(l_i) < t_E^V(l_j)$$

Aus den beiden Fälle zusammen ergibt sich die Behauptung des Lemmas. ■

Wir fassen die Ergebnisse für die CC-Sperre zusammen:

Satz 5.15 (Fairneß der CC-Sperre) Seien C die Anzahl der Prozessoren und e_1, e_2 die Konstanten aus Kriterium 5.8. Unter den Vorraussetzungen der Kriterien für die Standard-Sperren erfüllt die CC-Sperre folgende Fairneß-Kriterien:

1. Eine X-Anforderung an eine CC-Sperre wird von keiner anderen Anforderung überholt.
2. Eine IS-Anforderung wird von keiner X-Anforderung überholt.
3. Eine φ-Anforderung, $\varphi \in \{S, IX, SIX\}$ wird von keiner X-Anforderung und von höchstens $2 \cdot C \cdot \left\lceil \frac{e_2}{e_1} \right\rceil$ anderen ψ-Anforderungen, $\psi \in \{S, IX, SIX\}$, überholt.

Bemerkung: auf der SB-PRAM gilt $\lceil e_2/e_1 \rceil = 3$, wie sich aus der Implementierung der Gruppensperre [Röh96, Abschnitt 4.2.3][JLS99] ergibt. Sperranforderungen an die CC-Sperre werden also von höchstens $6C$ späteren Anforderungen überholt.

5.3 Aufbau der Concurrency-Control

Aufbauend auf den neuen Sperr-Typ CC-Sperre wird in diesem Abschnitt eine Concurrency-Control entwickelt. Als Schnittstelle dienen zwei Funktionen:

db_lock(granul, rid, pid, tid, φ) Die Funktion db_lock() sperrt das Objekt rid, pid, tid im Modus φ. Der Parameter granul gibt das Granulat des Objekts an. Handelt es sich bei dem Objekt um eine Seite oder ein Tupel, so wird die Seite mittels der Funktion fix() von db_lock() in den Systempuffer geladen und fixiert.

db_unlock(granul, rid, pid, tid, φ) Die Funktion db_unlock() gibt eine zuvor erhaltene Sperre wieder frei. Eine von db_lock() fixierte Seite wird von db_unlock() de-fixiert.

Fordert ein Prozessor eine Sperre auf ein Objekt, so soll üblicherweise auf das Objekt (oder feinere Granulate) zugegriffen werden. Daher ist es sinnvoll, ein Objekt o in den Systempuffer zu laden, sobald eine Sperre auf das Objekt o gefordert wird. Dadurch wird außerdem eine einfache Implementierung der db_lock()-Funktion ermöglicht.
Die Implementierung der Funktionen db_lock() und db_unlock() wird im folgenden erläutert.

5.3.1 Die Sperrtabelle

Zu jedem Objekt (Relation, Seite oder Tupel) werden Informationen in sogenannten Object Control Blocks gespeichert:

Definition 5.16 (Object Control Block)
Ein **Object Control Block (OCB)** ist eine Struktur mit folgenden Komponenten:

- granulat: $\in$ {g_relation, g_page, g_tupel} gibt an, auf welcher Ebene sich das Objekt befindet.
- rid: Relations-ID des Objekts
- pid: Page-ID des Objekts, falls Objekt nicht Relation ist.
- tid: Tupel-ID des Tupels, falls Objekt Tupel ist.
- page: Ein Zeiger auf den entsprechenden Page Control Block (PCB), falls Objekt Seite oder Tupel ist.
- t_ocb: Eine Hash-Tabelle von OCBs.
- nxt_ocb: dient zur Verkettung von OCBs in den Hash-Listen von t_ocb.
- lock: eine CC-Sperre.

Diese Struktur wird im nächsten Abschnitt erweitert. Wir unterscheiden im folgenden nicht immer zwischen einem Objekt und dem zugehörigen OCB.

Die Relations-OCBs werden in einem eindimensionalen Feld RELOCB[ ] gespeichert, das über die entsprechende Relations-ID rid addressiert wird.

Zur Verwaltung der OCBs der Seiten-Ebene wird die PCB-Struktur um einen OCB-Zeiger p_ocb (*Page-OCB*) erweitert. Für jede Seite p im Systempufer zeigt p.p_ocb auf den OCB der Seite. Bei einer Sperranforderung auf eine Seite (rid,pid) findet ein fix()-Aufruf statt, wodurch die Seite in den Systempuffer

geladen und fixiert wird. Der zur Seite gehörende PCB p wird von fix() zurückgeliefert. Über p.p_ocb hat man dann bereits den (rid,pid)-OCB gefunden. Es wird nicht zu jeder Seite auf Festplatte ein OCB im Speicher gehalten — was angesichts der Anzahl der Seiten auch nicht möglich wäre —, sondern nur zu den im Systempuffer liegenden Seiten.
Der zu einer Seite p gehörende OCB p.p_ocb wird beim Laden der Seite von Festplatte initialisiert und beim Verdrängen wieder de-initialisiert. Dazu wird die Funktionen fix() und der Verdrängungsalgorithmus angepaßt.

Die OCB-Komponente t_ocb (*Tupel-OCB*) wird nur bei Objekten des Granulats Seite benutzt. In dieser Hash-Tabelle werden die benutzten OCBs der zur Seite gehörenden Tupel gespeichert. Bei einer Sperranforderung auf auf ein Tupel t wird zunächst über die Funktion fix() die entsprechende Seite in den Systempuffer geladen und fixiert. Der zugehörige PCB p wird von fix() zurückgeliefert. Über die PCB-Komponente p_ocb erhält man den OCB o_{pg} der Seite. Über die Hash-Tabelle o_{pg}.t_ocb hat man dann Zugriff auf den OCB des zu sperrenden Tupels.
Das Verfahren zum Finden, Einfügen und Löschen von Einträgen in die Hash-Tabelle t_ocb verläuft analog zur PAGEARRAY–Hash-Tabelle. Bei einer Sperranforderung auf ein Tupel t wird der OCB des Tupels in der Hash-Tabelle gesucht. Wird er nicht gefunden, so wird die entsprechende Hash-Liste schreib-gesperrt und ein neuer OCB für das Tupel eingefügt. Zuvor wird wie in der Funktion fix() überprüft, ob nicht bereits ein anderer Prozessor einen OCB für das Tupel t eingefügt hat.
Zur Verwaltung von freien OCBs werden die Funktionen get_ocb(), put_ocb() bereitgestellt. Diese benutzen einen zum Systemstart reservierten Pool von OCBs und eine analog zur PCBQUEUE benutzte Queue OCBQUEUE. Die "Verdrängung" nicht benutzter OCBs erfolgt jedoch nicht durch einen eigenen Clock-Algorithmus; beim Verdrängen einer Seite aus dem Systempuffer werden die benutzten Tupel-OCBs der Seite durch den Clock-Prozessor P_{Clock} durch put_ocb()-Aufrufe freigegeben.

Als weitere Struktur wird der *Lock Control Block* (*LCB*) eingeführt. In einer einfach verketteten LCB-Liste LCBLIST_i wird für jeden Prozessor P_i gespeichert, welche Sperren von P_i auf welchen Objekten gehalten werden. Auf diese Listen greift nur der jeweilige Prozessor zu, es gibt also keine parallelen Zugriffe auf die Listen. Daher sind keine Synchronisations-Maßnahmen (zum Beispiel Sperren) nötig.

Definition 5.17 (Lock Control Block) Ein **Lock Control Block (LCB)** ist eine Struktur mit folgenden Komponenten:

rid:	Relations-ID
pid:	Page-ID
tid:	Tupel-ID
granulat:	Granulat
lmode:	$\in \{X, S, IS, IX, SIX\}$, gesetzter Sperrmodus

ocb: ein Zeiger auf den zugehörigen OCB

next: LCB-Zeiger zur Verkettung in der Liste

Auf die Komponenten rid, pid, tid und granulat könnte auch über den durch ocb bestimmten OCB zugegriffen werden. Aus Geschwindigkeitsgründen werden diese Informationen redundant im LCB gespeichert.

Mit den eingeführten Strukturen ist es nun möglich, den Ablauf einer Sperranforderung und der Freigabe einer bereits erworbenen Sperre zu erläutern. Dabei gehen wir von einer $\varphi-$Sperre auf ein Tupel tid in der Seite pid der Relation rid aus (φ ist der angeforderte Sperrmodus). Sperren auf gröberes Granulat laufen analog ab.

db_lock(g_tupel, rid, pid, tid, φ) :$\equiv$

1. Zunächst wird durch einen Aufruf der Funktion fix(rid,pid) die Seite fixiert und — falls nötig — in den Systempuffer geladen; der zugehörige Page Control Block sei mit p bezeichnet. p wird von fix() zurückgeliefert. o_{pg} := p.p_ocb ist ein Zeiger auf den (rid,pid)-OCB.

2. Anschließend wird in der Hash-Tabelle o_{pg}.t_ocb nach dem tid-OCB gesucht. Befindet sich der tid-OCB nicht in der Hash-Tabelle, wird durch einen get_ocb()-Aufruf ein freier OCB o reserviert und in die Hash-Tabelle eingefügt (analog zu Suchen/Einfügen in der Funktion fix()). Befindet sich der tid-OCB schon in der Hash-Tabelle, so sei er ebenfalls mit o bezeichnet.

3. Auf die CC-Sperre o.lock wird durch die Funktion cc_lock() eine φ-Sperre gesetzt. Die cc_lock()-Funktion wartet, bis die Sperre gesetzt werden kann (busy-waiting). Sie liefert entweder LOCK_OK oder LOCK_DEADLOCK zurück. LOCK_DEADLOCK wird bei erkannten Deadlocks (siehe Abschnitt 5.5) zurückgeliefert und erfordert einen *Rollback* der entsprechenden Transaktion.

4. Es wird ein LCB l mittels der C-Funktion malloc()[1] reserviert. Der LCB l wird initialisiert und in die entsprechende LCB-Liste LCBLIST eingefügt.

Umgekehrt verläuft eine Freigabe einer erworbenen φ-Sperre auf (rid,pid,tid) wie folgt:

db_unlock(g_tupel, rid, pid, tid, φ) :$\equiv$

1. Suche den (rid,pid,tid)-LCB in der LCB-Liste LCBLIST. Sei der LCB mit l bezeichnet.

[1] Man könnte hier wie bei den PCBs und den OCBs auf die Standard–C–Funktion malloc() verzichten und einen LCB-Pool ähnlich der PCBQUEUE oder OCBQUEUE benutzen. Dort mussten allerdings Speicherbereiche für den gemeinsamen Zugriff reserviert werden, wozu die PRAM–C–Funktion shared_malloc() verwendet werden muß. Diese ist im Vergleich zu malloc() sehr langsam. Da LCBs nur für den exklusiven Zugriff eines Prozessors reserviert werden, lohnt sich ein eigener Pool an dieser Stelle nicht. Wir können hier die Standard-Funktion malloc() benutzen.

2. o:=l.ocb ist der (rid,pid,tid)-OCB. Durch den Funktionsaufruf cc_unlock(o.lock, φ) wird die zugehörige CC-Sperre freigegeben.

3. o.page ist ein Zeiger auf den (rid,pid)-PCB. Ein unfix(o.page)-Aufruf gibt die Seite im Systempuffer wieder frei.

4. Der LCB l wird abschließend aus der Liste LCBLIST gelöscht.

Quelltext für diese Funktionen findet man in [JLS99].

5.3.2 Double-Locks und Lock-Conversion

Bei der Abarbeitung von Transaktionen kommt es gelegentlich vor, daß das gleiche Objekt von der gleichen Transaktion mehrmals gesperrt wird. Zum Beispiel können in einer Transaktion zwei Anfragen auf die gleiche Relation vorkommen, so daß die Relation zweimal IS-gesperrt wird. Diese Situation nennt man *Double-Locks*. Problematisch sind *Double-Locks*, wenn zweimal eine X-Sperre angefordert wird, da diese zueinander inkompatibel sind. Gleiches gilt für SIX-Sperren.
Die PRAM–Concurrency-Control unterstützt Double-Locks. Bevor die Sperre auf ein Objekt *o* gesetzt wird, wird in der LCBLIST nach einem Lock Control Block für das Objekt *o* gesucht. Gibt es einen solchen LCB, und ist die schon vergebene Sperre vom gleichen Modus wie die neu angeforderte, so wird die Sperranforderung nicht ein zweitesmal ausgeführt, sondern als erfolgreich beendet von der Concurrency-Control zurückgeliefert.

Es kann bei der Transaktionsbearbeitung jedoch auch vorkommen, daß das gleiche Objekt in zwei verschiedenen Modi gesperrt werden soll. In einem solchen Fall spricht man nicht von Double-Lock, sondern von *Lock-Conversion*. Zum Beispiel könnte eine Transaktion eine Seite erst S-sperren und anschließend eine X-Sperre auf der gleichen Seite anfordern.
Lock-Conversion wird im PRAM-DBS nicht unterstützt. Jedoch kann man die CC-Sperre abändern, so daß Lock-Conversion unterstützt wird. Dazu müssen die verwendeten Standard-Sperren lediglich um Lock-Conversion ergänzt werden. In [Edl95, Wil88] werden Standard-Sperren für Shared-Memory-Rechner mit Fetch-and-Add vorgestellt, die Lock-Conversion unterstützen.

5.4 Gegenüberstellung mit dem üblichen Verfahren

Die im letzten Abschnitt vorgestellte Concurrency-Control basiert auf einer neuen Idee, welche gegenüber der herkömmlichen Implementierung [Spe97, GR93] erhebliche Vorteile besitzt. Diese sollen in diesem Abschnitt erläutert werden. Dazu wird zunächst die herkömmliche Implementierung kurz vorgestellt.

Der wesentliche Unterschied zwischen beiden Implementierungen liegt in der *CC-Sperre*. Die in Abschnitt 5.2 vorgestellte CC-Sperre benutzt einen *Sperr-Baum*, der auf den Standard-Sperren aus [Röh96] aufbaut. In [Spe97] sieht eine

CC-Sperre hingegen wie folgt aus[2]:

Definition 5.18 (CC-Sperre) Eine **CC-Sperre** ist eine Struktur mit folgenden Komponenten:

- lmode: Gewährter Sperrtyp $\in \{X, S, IS, IX, SIX\}$. Bei Vergabe mehrer kompatibler Sperren zeigt diese Komponente den *Gruppensperrmodus*[3] an.
- grcnt[ ]: Array, mit dem die Anzahl gewährter, kompatibler Sperren in Abhängigkeit des Sperrtyps gezählt werden.
- actives: Zeiger auf eine Liste, die alle LCBs kompatibler Sperren enthält, die auf dem Objekt vergeben wurden.
- waits: Zeiger auf eine LCB-Liste, in der alle Sperranforderungen zwischengespeichert werden, deren Sperrtypen inkompatibel zu lmode sind.
- rw: fairer Reader/Writer-Lock

Bei der Anforderung einer φ-Sperre auf eine CC-Sperre l wird zunächst eine Schreib-Sperre auf l.rw gesetzt. Gibt es schon wartende Sperranforderungen (die Liste l.waits ist nicht leer), wird die neue Sperranforderungen an die Warte-Liste l.waits angehängt. Anschließend wird die Sperre auf l.rw freigegeben.
Gibt es noch keine wartenden Sperranforderungen, wird die Kompatibilität von φ und l.lmode überprüft. Sind die beiden Modi inkompatibel, so wird die neue Sperranforderung in die Warte-Liste eingefügt. Andernfalls wird der neue Gruppensperrmodus berechnet und die gewährte Sperre in die actives-Liste eingefügt. Anschließend wird die Sperre auf l.rw freigegeben.

Bei der Freigabe einer φ-Sperre werden wartende Sperranforderungen in der Reihenfolge ihrer Speicherung in l.waits aktiviert, falls sie nach der Freigabe der φ-Sperre kompatibel zu den anderen aktiven Sperren sind. Auch die Freigabe einer φ-Sperre findet in einem durch l.rw geschützten Abschnitt statt. Details findet man in [Spe97, Seite 59ff.].

Das Hauptproblem entsteht bei dieser CC-Sperre bei häufig gesperrten Objekten, bei denen die Sperrmodi weitestgehend kompatibel sind. Dies tritt bei *granular locking* häufig bei groben Granulaten auf. Zum Beispiel müssen bei Seitenzugriffen immer die entsprechenden Relationen mit Warnsperren belegt werden, wobei die benutzten IS- und IX-Sperren kompatibel sind. Bei der in Abschnitt 5.2 vorgestellten CC-Sperre werden diese Sperranforderungen nicht sequentialisert, da die Standard-Sperren kompatible Sperranforderungen nicht sequentialisieren.

Die in [Spe97] benutzte CC-Sperre sequentialisiert selbst kompatible Sperren, weil bei jeder Sperranforderung der Reader/Writer–Lock l.rw schreib-gesperrt

[2] In [Spe97] wird die CC-Sperre nicht separat betrachtet, sondern direkt im OCB verankert. Die hier genannte Definition ist aus der OCB-Definition extrahiert.

[3] Der *Gruppensperrmodus* entspricht in etwa dem "stärksten" vergebenen Sperrmodus auf der CC-Sperre. Dazu führt man eine partielle Ordnung $\prec$ auf den Sperrmodi ein: IS$\prec$S, IX, SIX

wird. Spengler gibt in seiner Arbeit eine Länge des kritischen Abschnitts von 60 Instruktionen an. Dies ist nicht sonderlich viel. Betrachtet man aber zum Beispiel eine Relation R, die von jeder Transaktion mit einer Warnsperre belegt wird, ergeben sich erhebliche Laufzeit-Nachteile dieser CC-Sperre. Denn fordern 2000 Prozessoren gleichzeitig eine Warnsperre auf R, so muß der letzte Prozessor $2000 \cdot 60 = 120.000$ Instruktionen warten, bis er den Reader/Writer–Lock bekommt — obwohl alle Prozessoren kompatible Sperren gefordert haben.
Bei Transaktionsende müssen die Sperren wieder freigegeben werden. Dazu fordern die Prozessoren wieder eine Sperre auf l.rw. Der Prozessor, der die CC-Sperre als erster setzen konnte, muß nun jedoch auf den fairen Reader/Writer-Lock l.rw warten, falls noch nicht alle Sperranforderungen bearbeitet sind. Somit stellen die 120.000 Instruktionen (ca. eine halbe Sekunde, vgl. Kapitel 3) eine untere Schranke für die Transaktionsdauer und damit eine obere Schranke für den Transaktionsdurchsatz im schlimmsten Fall dar.

Die neue CC-Sperre kann Sperranforderungen in weniger als 100 Instruktionen ausführen, wenn nicht auf inkompatible Sperren gewartet werden muß. Muß auf die angeforderte Sperre nicht gewartet werden, so benötigt eine komplette Sperranforderung über die Funktion db_lock() inklusive fix()-Aufruf, LCBLIST durchsuchen etc. (siehe Abschnitt 5.3) zwischen 400 und 600 Instruktionen. Die neue Concurrency-Control ist in bestimmten Situationen also ungefähr um einen Faktor 200 schneller, als die Concurrency-Control von Spengler ($\approx$600 zu $\approx$120.000 Instruktionen). Eine solche Situation tritt zum Beispiel beim TPC-B–Benchmark auf (siehe nächstes Kapitel), wo jede Transaktion jede Relation im IX-Modus sperrt.

5.5 Deadlocks

Im PRAM-Datenbanksystem können, wie bei den meisten anderen Datenbanksystemen auch, zyklische Wartesituationen entstehen. Diese zyklischen Wartesituationen sind als *Deadlocks* bekannt. In diesem Abschnitt wird der Deadlock-Begriff definiert und die Concurrency-Control wird um eine Deadlock-Erkennung erweitert. Wir beweisen die Korrektheit der Deadlock-Erkennung.

5.5.1 Definitionen

Es sei an die Aufteilung jeder Sperranforderung in zwei Phasen und die Definition von *auf Sperre warten* und *Sperre halten* erinnert (Abschnitt 3.6.3). Wir benutzen diese Definitionen auch für CC-Sperren. Im folgenden bezeichne C immer die Anzahl der Prozessoren.

Definition 5.19 (Wartegraph) Seien P und P' Prozessoren, l eine CC-Sperre, und φ und ψ seien zueinander inkompatible Sperrmodi. Zu einem Zeitpunkt t warte P' auf eine φ-Sperre auf l, und P halte eine ψ-Sperre auf l. Wir sagen, daß Prozessor P' zum Zeitpunkt t auf P wartet, geschrieben $P' \prec^t P$.
Der **Wartegraph** zum Zeitpunkt t ist ein gerichteter Graph $\mathcal{W}^t = (V, E^t)$ mit Knotenmenge V und Kantenmenge E^t. Die Menge V ist gleich der Menge der

Prozessoren $\{0, \ldots, C-1\}$. Die Kantenmenge wird definiert durch

$$(P', P) \in E^t :\Longleftrightarrow P' \prec^t P.$$

Man beachte, daß für jeden Prozessor P zu zu jedem Zeitpunkt höchstens eine CC-Sperre existiert, auf die P wartet (busy-waiting).

Definition 5.20 (Deadlock) Ein Prozessor P befindet sich zum Zeitpunkt t im **Deadlock**, wenn der Wartegraph $\mathcal{W}^t$ einen Zyklus $\langle P = P_1, P_2, \ldots P_m = P\rangle$ (das heißt $P = P_1 \prec^t P_2 \prec^t \ldots \prec^t P_m = P$) besitzt, in dem P vorkommt (O.B.d.A. steht $P = P_1$ an erster Stelle in diesem Zyklus).
Ein Prozessor P **hängt** zum Zeitpunkt t **von einem Deadlock ab**, wenn es in $\mathcal{W}^t$ einen Pfad $P \longrightarrow^*_{\mathcal{W}^t} P'$ gibt, und P' befindet sich zum Zeitpunkt t im Deadlock.

Lemma 5.21 Befindet sich P zum Zeitpunkt t im Deadlock, und werden keine Sperranforderungen abgebrochen, so befindet sich P auch zu jedem Zeitpunkt $t' \geq t$ im Deadlock.

Beweis: Wir zeigen die Aussage durch Induktion über t'.

Induktionsanfang $t' = t$: gilt nach Voraussetzung

Induktionsschritt $t'-1 \to t'$: Angenommen, P befände sich zum Zeitpunkt t' nicht mehr im Deadlock. Dann würde der zum Zeitpunkt $t'-1$ existierende Wartegraphen-Zyklus $\langle P = P_1, P_2, \ldots, P_m = P\rangle$ zum Zeitpunkt t' nicht mehr existieren. Also ist mindestens eine Kante (P_j, P_j+1) beim Übergang von $E^{t'-1}$ zu $E^{t'}$ verschwunden; sei l die CC-Sperre zu dieser Kante. Zum Zeitpunkt t' wartet P_j also nicht mehr auf l, oder P_{j+1} hält seine Sperre auf l nicht mehr.

Wir betrachten zunächst die zweite Möglichkeit. Zum Zeitpunkt $t'-1$ war P_{j+1} noch in Phase 1 seiner Sperranforderung und wartete dort auf P_{j+2}; demnach kann P_{j+1} zum Zeitpunkt t' die Sperrfreigabe für l nicht begonnen haben (busy-waiting). Deswegen hält P_{j+1} zum Zeitpunkt t' noch immer die Sperre auf l.

Nun zu der Möglichkeit, daß P_j nicht mehr wartet. Zum Zeitpunkt $t'-1$ befand sich P_j in Phase 1 der Sperranforderung für l. Zum Zeitpunkt t' hält P_{j+1} l gesperrt; P_j hängt also noch von P_{j+1} ab (Satz 5.2) und befindet sich demnach noch in Phase 1, da die Sperranforderung von P_j nicht abgebrochen wurde. P_j wartet also noch auf l. Widerspruch. ■

Nur wegen dieses Lemmas sind Deadlocks von Interesse. Das Lemma sagt aus, daß Deadlocks nicht von selbst "verschwinden" und daher eine sich in einem Deadlock befindende Transaktion nicht terminiert. In Datenbanksystemen wird dieses Problem üblicherweise durch eine Deadlock-Erkennung gelöst, die bei einem gefundenen Deadlock den Zyklus durch Abbruch (mindestens) einer Transaktion aufbricht. Im folgenden beschränken wir uns auf das Auffinden von Deadlocks und vernachlässigen den Abbruch von Transaktionen.

5.5.2 Erkennung von Deadlocks

Die Definition von Deadlocks legt es nahe, zur Deadlock-Erkennung eine Zyklen-Suche im Wartegraphen auszuführen. Eine Zyklen-Suche kann effizient durch *Tiefensuche (Depth-First-Search, DFS)* ausgeführt werden [CLR90, Kap 23.3f]. Dieser Ansatz zur Deadlock-Erkennung ist allgemein bekannt und wird häufig verwendet. In Datenbanksystemen für massiv-parallele Architekturen wie die SB-PRAM stellt jedoch die Verwaltung des Wartegraphen ein Problem dar. Würden wir den Graphen in der üblichen Adjazenzlisten-Darstellung im Speicher halten, müßten Änderungs- und Lese-Zugriffe auf die Adjazenzlisten durch Sperren getrennt werden[4]. Dadurch würden wir aber den Gewinn durch die neue CC-Sperre wieder zerstören, da die Wartegraph-Verwaltung Zugriffe sequentialisieren würde.
Im PRAM-Datenbanksystem wird der Wartegraph daher nur "implizit gespeichert". Wir benutzen die Idee der Adjazenzmatrizen, berechnen allerdings $v \prec^t u$, anstatt den Matrixeintrag (v, u) auszulesen. Die Knotenmenge ist gleich der Prozessormenge und ändert sich daher nicht, muß also nicht gespeichert werden. Eine Traversierung der Knotenmenge ist mit einer einfachen "for v=0 to C-1"-Schleife möglich.

Quelltext 5.1 Erste Version der Funktion dfs_findcycle

```
        dfs_findcycle(v)
        node v;
        {
(1)         status[v] := besucht
(2)         foreach u ≻ v        //traversiere zu v adjazente Knoten
                if (status[u] = nicht_besucht)
(3)                 dfs_findcycle(u)
                else if (status[u] = besucht)
                    if (u = myID)
(4)                     Prozessor myID befindet sich im Deadlock,
                        synchronisiere alle beteiligten Prozessoren und
                        breche eine Transaktion ab.
            status[v] := abgeschlossen
            return;
        }
```

Quelltext 5.1 zeigt Pseudo-Code für eine erste Implementierung von DFS. Diese wird später geändert. Der Prozessor, der DFS ausführt, habe die Nummer myID. Jeder Knoten v wird mit einer Komponente status[v] versehen. In diesem Feld wird gespeichert, ob der Knoten schon "besucht" wurde (vgl. [CLR90]). Die status-Komponente aller Knoten wird vor jeder Zyklensuche mit nicht_besucht initialisiert. Während der Zyklensuche ändert sich der Status eines Knoten zunächst auf besucht, abschließend auf abgeschlossen. Damit mehrere Prozessoren gleichzeitig eine Zyklensuche ausführen können, hat jeder Prozessor eine private Version des Feldes status[ ].

[4] Zumindest gilt das für die offensichtliche Implementierung der Graphen-Routinen.

Bei einer Tiefensuche werden von einem Knoten v aus alle zu v adjazenten Knoten u, $u \succ v$, besucht. Falls ein Knoten u mit $u \succ v$ existiert, der schon besucht, aber noch nicht abgeschlossen wurde (u.status = besucht), wurde ein Zyklus gefunden und eine Transaktion muß abgebrochen werden.

Die foreach-Schleife in Zeile 2 wird üblicherweise durch eine Traversierung der Adjazenzliste des Knoten v implementiert. Wie oben erklärt, werden in der PRAM-Implementierung diese Adjazenzlisten nicht gespeichert. Die foreach-Schleife kann jedoch durch folgendes Programmfragment ersetzt werden:

Quelltext 5.2 Implementierung der foreach-Schleife

```
for u=0 to C-1
   if u ≻ v
      do something
```

Es muß nun geklärt werden, wie die Bedingung $u \succ v$ überprüft werden kann. Dazu führen wir zunächst für jeden Prozessor P_i eine Variable waitfor[i] vom Typ "Zeiger auf CC-Sperre" sowie eine Variable waitmode[i]$\in$ {X,S,IX,IS,SIX} ein. Wartet Prozessor P_i auf eine φ-Sperre auf l, so ist waitfor[i]=l der Zeiger auf die CC-Sperre. Weiterhin ist waitmode[i]=φ. Wartet P_i auf keine Sperre, so ist waitfor[i]=PSEUDO und waitmode[i] nicht spezifiziert. PSEUDO ist ein Zeiger auf eine "Pseudo-Sperre", die noch erklärt wird.
Wir erweitern außerdem die Definition der CC-Sperre (siehe Definition 5.1) um ein Feld mode[ ], so daß für jeden Prozessor P_i und jede CC-Sperre l in l.mode[i] der Sperrmodus von Prozessor P_i auf l gespeichert wird. Hält Prozessor P_i die Sperre l nicht gesperrt, so wird l.mode[i]:=frei gesetzt.
Der Zeiger PSEUDO zeigt auf eine CC-Sperre, die von keinem Prozessor angefordert wird, und für die PSEUDO.mode[i]=frei für alle Prozessoren P_i gilt. Alle Sperrmodi sind zu frei kompatibel.

Bemerkung: Um die sechs Möglichkeiten (frei und 5 Sperrmodi) für mode[i] zu kodieren, werden 3 Bit benötigt. Damit kann man in einem Wort der Länge 32 Bit für 10 Prozessoren den Wert von mode speichern. Die einzelnen Bits in einem 32-Bit-Wort können dann mit syncor gesetzt und mit mpand gelöscht und gelesen werden. Diese Speicherzugriffe können serialisierungsfrei ausgeführt werden. Der Speicherverbrauch eines OCBs steigt durch die Komponente mode[ ] in der CC-Sperre um $C/10 = 2048/10 \approx 205$ Worte. Dies ist ein gewaltiger Overhead, da vorher die Größe eines OCBs bei 15 Worten lag, jetzt bei 220 Worten. Der hierfür benötigte Speicher verringert die Größe des Systempuffers. Es bleibt zu hoffen, daß der positive Effekt einer nicht unnötig sequentialisierenden Concurrency-Control inklusive Deadlock-Erkennung den negativen Effekt des kleineren Systempuffers übertrifft.

Wir erweitern die Operationen der CC-Sperre cc_lock() und cc_unlock(), so daß dort die Variablen und Komponenten waitfor, waitmode und mode gesetzt werden (vgl. Quelltext 5.3 auf Faltblatt am Ende des Kapitels). Für die Zeile (15)

wird d_2 in Quelltext 5.4 festgelegt. Die Zeile (l2) wird zur Initialisierungs-Phase der Sperranforderung gezählt. Die Phase 2 der Sperranforderung endet mit Zeile (l4). Die Zeilen (l1),(l5) und (l6) werden also nicht zur Sperranfoderung gezählt. Die Freigabe einer Sperre beginnt erst mit Zeile (u2). Damit ergibt sich die Definition 3.3 (*auf Sperre warten, Sperre halten*) zu

- P wartet auf eine φ-Sperre auf der CC-Sperre l, wenn (l2) begonnen wurde, und sich die Sperranforderung noch in der ersten Phase befindet.
- P hält eine φ-Sperre auf l, wenn ($l4$) beendet wurde, und (u2) noch nicht begonnen wurde.

Es ist nach Definition $u \succ v$ genau dann, wenn es eine CC-Sperre l gibt, so daß gilt:

1. Prozessor v wartet auf φ-Sperre auf l
2. Prozessor u hält ψ-Sperre auf l
3. φ und ψ sind inkompatibel.

Prozessor v wartet auf l:=waitfor[v]. Mittels l.mode[u]=ψ und waitmode[v]=φ sind die Bedingungen 2 und 3 leicht überprüfbar. Es ergibt sich Quelltext 5.4 als Programmfragment für den Adjazent-Test.

Lemma 5.22 Ein Prozessor P führe den Adjazenz-Test (Quelltext 5.4) aus, es sei t der Zeitpunkt des Speicherzugriffs in Zeile (t1) und testerg das Ergebnis des Tests. Dann gilt

$$\mathsf{testerg} = \mathsf{TRUE} \Longrightarrow v \prec^t u.$$

Beweis: Wir nehmen testerg = TRUE nach dem Test an. Dann gilt φ ist inkompatibel zu ψ, und daher ist $\psi \neq$frei.
1. Es ist $\psi \neq$frei=PSEUDO.mode[u] und daher gilt zum Zeitpunkt t waitfor[v] $\neq$ PSEUDO. Prozessor v muß also eine Sperranforderung auf l begonnen haben und befindet sich zum Zeitpunkt t zwischen Ende-(l2) und Anfang-(l4). Da zwischen (l4) und (l6) mehr als d_1 V-Zeitpunkte vergehen, kann v zum Zeitpunkt $t + d_1$ keine neue Anforderung begonnen haben, und v hat daher waitmode[v] noch nicht verändert. Zum Zeitpunkt $t + d_1$ wird in Zeile (t2) also der Sperrmodus φ gelesen, den v angefordert hatte.
2. Sei $t + d_2$ der Zeitpunkt, zu dem l.mode[u] aus dem gemeinsamen Speicher gelesen wird. Es ist in Zeile (t4) $\psi \neq$ frei und damit zum Zeitpunkt $t + d_2$ l.mode[u]=$\psi \neq$frei. Prozessor u hat also zum Zeitpunkt $t + d_2$ Zeile (l6) schon beendet, und da Zeile (l5) $\geq d_2$ V-Zeiteinheiten benötigt, hat u zum Zeitpunkt t schon ($l4$) beendet. Damit hat u die Sperranforderung seit mindestens d_2 V-Zeiteinheiten beendet. Jedoch hat u die Zeile (u1) zum Zeitpunkt $t + d_2$ und damit erst recht zum Zeitpunkt t noch nicht beendet und daher die Sperrfreigabe noch nicht begonnen. Prozessor u hält zum Zeitpunkt t also eine ψ-Sperre auf o.
3. Es ist φ zu ψ inkompatibel. Da u zum Zeitpunkt t die ψ-Sperre auf l hält und v die Sperranforderung schon begonnen hat, hängt v vom Verhalten von u

ab. Daher ist v noch in Phase 1 der Sperranforderung und wartet also auf u; es gilt $v \prec^t u$. ■

Da das Programmfragment TEST keine Sprung-Befehle enthält, ergeben sich d_1 und d_2 zu Konstanten, die direkt aus dem Maschinen-Code des Fragments abgelesen werden können. Man beachte, daß hierin ein wesentlicher Vorteil des PRAM-Modells liegt: jeder Befehl hat konstante Dauer. Bei der aktuellen Implementierung ist $d_2 = 6 \cdot C$.

Lemma 5.23 Für zwei Prozessoren P_1 und P_2 gelte $P_1 \prec^t P_2$. Die den Konflikt verursachende CC-Sperre sei l. Der durch P_1 angeforderte Sperrmodus sei mit φ bezeichnet, der von P_2 gehaltene Sperrmodus sei ψ. Weiterhin gebe es einen Zyklus im Wartegraphen $\mathcal{W}^t$, in dem die Kante (P_1, P_2) vorkommt. Dann gilt für ein (kleines) $d \in \mathbb{N}$ zum Zeitpunkt $t + d$ und alle folgenden Zeitpunkte mindestens bis zur Auflösung des Deadlocks:

$$\begin{aligned} \mathsf{waitfor}[P_1] &= l \\ \mathsf{waitmode}[P_1] &= \varphi \\ \mathsf{o.mode}[P_2] &= \psi \end{aligned}$$

Beweis: Nach Definition von $\prec^t$ wartet P_1 auf eine φ-Sperre für l. P_1 hat also zumindest die Ausführung von (l2) begonnen und wird daher nach kurzer Zeit die Variable $\mathsf{waitfor}[P_1]$ wie in der Behauptung setzen; $\mathsf{waitmode}[P_1]$ wurde bereits in Zeile (l1) gesetzt. Prozessor P_2 hält schon die ψ-Sperre auf l, ebenfalls nach Definition von $\prec^t$. Nach Definition von "Sperre halten" hat er Zeile (l4) schon ausgeführt und wird daher nach kurzer Zeit in Zeile (l6) $l.\mathsf{mode}[P_2]{=}\psi$ setzen.

Da sich die Prozessoren P_1 und P_2 im Deadlock befinden und demnach auf Sperren warten, werden diese Variablen bis zum Ende des Deadlocks nicht mehr verändert (busy-waiting und Lemma 5.21). ■

Auch hier ergibt sich wie in der Bemerkung zu Lemma 5.22 die Konstante d direkt aus dem Maschinen-Code der Programm-Fragmente: $d = 9 \cdot C$.

Wir möchten nun folgenden Satz über die Korrektheit der Deadlock-Erkennung beweisen:

Satz 5.24 (Korrektheit) Sei P ein Prozessor. P führe eine Zyklensuche durch, die mit dem Funktions-Aufruf dfs_findcycle(P) zum Zeitpunkt t begonnen habe. Dann ist die Deadlock-Erkennung korrekt, d.h.

1. befindet sich P zum Zeitpunkt $t-d$ (d aus Lemma 5.23) in einem Deadlock, so findet der Algorithmus einen Deadlock, in dem P sich befindet (beachte: P kann sich in mehreren Deadlocks gleichzeitig befinden).
2. jeder durch den Algorithmus gefundene Deadlock ist tatsächlich ein Deadlock, in dem P sich befindet.

Aussage 1 trifft lediglich mit Wahrscheinlichkeit 1 zu[5].

[5] Wenn der Leser von jetzt an für den Rest seines Lebens nur noch Münzen wirft, ist es nicht ausgeschlossen, daß er niemals "Kopf" zu sehen bekommt. Es wird dem Leser allerdings mit Sicherheit und nicht nur mit Wahrscheinlichkeit 1 langweilig werden.

In der Form von Quelltext 5.1 ist der Algorithmus jedoch nicht korrekt, da der Wartegraph sich während der Zyklensuche ändert. Dazu betrachten wir folgendes Beispiel:

Beispiel: Prozessor P_1 führt eine Zyklensuche aus. Dabei findet er zu gewissen Zeitpunkten $t_1 < t_2 < t_3$ den Zyklus:

$$P_1 \prec^{t_1} P_2 \prec^{t_2} P_3 \prec^{t_3} P_1.$$

Dabei sei $l_{2,3}$ die CC-Sperre, die den Konflikt zwischen P_2 und P_3 verursacht und $l_{3,1}$ analog für P_3 und P_1. Im Zeitintervall (t_2, t_3) kann Prozessor P_3 die Sperre auf $l_{2,3}$ freigeben, seine Transaktion beenden, eine neue Transaktion beginnen und anschließend eine zu P_1 inkompatible Sperre auf $l_{3,1}$ anfordern. Dadurch erkennt Prozessor P_1 einen scheinbaren Zyklus, der zum Zeitpunkt t_3 nicht existiert wegen $P_2 \not\prec^{t_3} P_3$. Solche nur scheinbaren Zyklen werden **Phantom-Deadlocks** genannt.

Ein Lösung des Problems wäre, die Deadlock-Erkennung nicht prozessor- sondern transaktionsorientiert durchzuführen. Damit ist gemeint, als Knoten des Wartegraphen nicht Prozessoren sondern Transaktionen (bzw. ihre eindeutige Transaktionsnummer) zu benutzen. Wird eine Transaktion beendet, wird der Knoten und alle inzidenten Kanten aus dem Graphen gelöscht. Bei Transaktionsbeginn wird ein neuer Knoten ohne inzidente Kanten eingefügt. In diesem Fall könnte wegen des Two-Phase-Locking–Protokolls die Situation des Beispiels nicht mehr auftreten. Man kann beweisen, daß dies für die Korrektheit der Deadlock-Erkennung genügt, sofern es keine spontanen Transaktionsabbrüche gibt ([BHG87, Seite 81]). Solche spontanten Transaktionsabbrüche können zum Beispiel durch Ausfall eines Knotens in einem verteilten System oder durch die Verletzung einer Konsistenzbedingung hervorgerufen werden.
Wir wählen hier jedoch einen anderen Weg zur Lösung des Problems, da eine transaktionsorientierte Deadlock-Erkennung schwieriger zu implementieren ist; man denke zum Beispiel an die Felder waitfor[ ] und waitmode[ ], die dann nicht durch die beschränkte Prozessornummer indiziert werden, sondern durch eine (zumindest theoretisch) unbeschränkte Transaktionsnummer. In diesem Fall würde sich eine Implementierung der Felder als Hash-Tabellen anbieten; dies führt bei parallelen Zugriffen zu Schwierigkeiten. Außerdem würde sich bei einer transaktionsorientierten Deadlock-Erkennung die Knoten-Menge V des Wartegraphen ändern, was bei der Implementierung ebenfalls zu Schwierigkeiten führen kann.

Die hier verwendete Lösungsidee ist, jeden durch den Algorithmus gefundenen Zyklus nochmals in umgekehrter Reihenfolge zu durchlaufen. Gelingt das, so garantiert das folgende Lemma die tatsächliche Existenz eines Deadlocks.

Lemma 5.25 Es gelte für Prozessoren $P_i, 1 \leq i \leq k$, und Zeitpunkte $t_1 < t_2 < t_3 < \ldots < t_k$

$$P_1 \prec^{t_k} P_2 \prec^{t_{k-1}} P_3 \prec^{t_{k-2}} \ldots \prec^{t_3} P_{k-1} \prec^{t_2} P_k \prec^{t_1} P_1$$

und Prozessor P_1 gebe keine Sperren frei. Dann gilt auch

$$P_1 \prec^{t_k} P_2 \prec^{t_k} P_3 \prec^{t_k} \ldots \prec^{t_k} P_{k-1} \prec^{t_k} P_k \prec^{t_k} P_1,$$

es "verschwinden" also keine Abhängigkeiten wie im Beispiel. Insbesondere befinden sich die Prozessoren unter diesen Voraussetzungen zum Zeitpunkt t_k im Deadlock.

Beweis: Setze $P_{k+1} := P_1$. Die Behauptung folgt dann durch Induktion über $i = k+1, \ldots, 1$ wie folgt:

Induktionsanfang $i = k+1$: P_i gibt nach Voraussetzung keine Sperre frei und demnach muß P_{i-1} zum Zeitpunkt t_k noch immer auf P_i warten. Insbesondere gibt P_{i-1} ebenfalls keine Sperre frei (busy-waiting).

Induktionsschritt $i \to i-1$: im Wortlaut wie der Induktionsanfang. ■

Dieses Lemma ermöglicht uns, den DFS-Algorithmus zu modifizieren, so daß wir den Satz beweisen können. Quelltext 5.5 zeigt die veränderte Version der Funktion dfs_findcycle(). Wir werden die Änderungen nach und nach erklären. Zeile (3) und die Änderungen an der Funktion cc_lock() werden im nächsten Abschnitt erklärt. Wir argumentieren im folgenden über einen Prozessor P, der die Tiefensuche ausführt. Dieser habe die Nummer myID.

Beweis Aussage 2 des Satzes. Wir nehmen an, der DFS-Algorthimus (Quelltext 5.5) findet durch den beschriebenen Adjazenz-Test (Quelltext 5.4) und Lemma 5.22 einen "Zyklus" $P = P_1 \prec^{t_1} P_2 \prec^{t_2} \ldots \prec^{t_{m-1}} P_m = P$. Bevor eine Transaktion abgebrochen wird, überprüft Prozessor P diesen Pfad nochmals von P_m rückwärts zu P_1 laufend (Zeile 5). Ist dieser Pfad nicht mehr vorhanden, so hat P einen Phantom-Deadlock gefunden und beginnt die Zyklensuche erneut (6). Andernfalls ergibt sich durch die Überprüfung des Pfads die Situation des Lemmas, so daß nach der Überprüfung des Pfads zu einem Zeitpunkt t' gilt: $P = P_1 \prec^{t'} P_2 \prec^{t'} \ldots \prec^{t'} P_m = P$. Für die Anwendung des Lemmas ist zu beachten, daß P selbst die Zyklensuche durchführt, währenddessen also keine Sperre freigegeben hat. Die Prozessoren befinden sich also zum Zeitpunkt t' tatsächlich im Deadlock; dies ist in Zeile (7) erkannt und wird dort behandelt. Damit ist Teil 2 des Satzes bewiesen.

Es bleibt Aussage 1 zu beweisen. Wir nehmen im folgenden an, daß Prozessor P sich zum Zeitpunkt $t-d$ in einem Deadlock befindet, und daß P die Zyklensuche durch den Aufruf dfs_findcycle(P) zum Zeitpunkt t beginnt. Es kann nun passieren, daß P einen Phantom-Deadlock findet, so daß der "Rückwärts-Test" in Zeile (5) von Quelltext 5.5 fehlschlägt. Die Tiefensuche wird dann in Zeile (6) abgebrochen und neu gestartet. Dies kann sich (zumindest theorethisch) unendlich oft wiederholen, so daß P niemals den Deadlock findet, in dem er selbst hängt.
Dieses Problem lösen wir durch einen Trick: die foreach-Schleife in Zeile (2) legt nicht fest, in welcher Reihenfolge die adjazenten Knoten besucht werden. Die Reihenfolge ist für die Tiefensuche unererheblich. Wir können daher das Programm-Fragment in Quelltext 5.2 durch das Programm-Fragment 5.6 ersetzen.

Beweis Aussage 1 des Satzes. Die Zyklensuche beginne zum Zeitpunkt t. Sei $\langle P = P_0, P_1, \ldots, P_{k-1}, P_k = P\rangle$ ein Zyklus im Wartegraphen $\mathcal{W}^{t-d}$. Bevor der Deadlock zum Zeitpunkt t_E in Zeile (7) aufgelöst wird (wenn er überhaupt aufgelöst wird), ist er nach Lemma 5.21 ebenfalls in allen Wartegraphen $\mathcal{W}^{t'}, t^E > t' > t - d$, vorhanden.
Die Kanten des Deadlocks werden daher vom Adjazenz-Test (Quelltext 5.4) erkannt wegen Lemma 5.23. Prozessor P beginnt die Tiefensuche bei Knoten P. Mit einer Wahrscheinlichkeit von mindestens $p := 1/C$ wird in Quelltext 5.6 eine Permutation mit $\pi(0) = P_1$ gewählt und bei der Zyklensuche als nächster Knoten P_1 besucht. Von dort aus wird analog mit Wahrscheinlichkeit mindestens p der Knoten P_2 besucht, u.s.w.. Insgesamt wird mit Wahrscheinlichkeit mindestens $p_{ges} = p^k$ ohne "Umwege" direkt der Pfad $\langle P_0, \ldots, P_{k-1}, P_k\rangle$ genommen und damit der Zyklus gefunden. Die Wahrscheinlichkeit, daß es genau $m \geq 0$ erfolglose und wegen nur scheinbaren Zyklen abgebrochene Durchläufe gibt, bevor der Zyklus gefunden wird, ist somit kleiner als

$$p(m) := p_{ges} \cdot (1 - p_{ges})^m.$$

Die Wahrscheinlichkeit, daß der Zyklus mit höchstens m Fehlversuchen gefunden wird, ist somit

$$\begin{aligned}
p(\leq m) &:= \sum_{i=0}^{m} p(i) \\
&= \sum_{i=0}^{m} p_{ges} \cdot (1 - p_{ges})^i \\
&= p_{ges} \cdot \frac{1 - (1 - p_{ges})^{m+1}}{1 - (1 - p_{ges})} \qquad \text{(geom. Summe)} \\
&= 1 - \underbrace{(1 - p_{ges})}_{<1}^{m+1} \\
&\xrightarrow{m \to \infty} 1.
\end{aligned}$$

■

In dieser Abschätzung ist der Erwartungswert der Fehlversuche, wie man leicht nachrechnet, $1/p_{ges}$. Bei der üblichen Zykluslänge 2 und C=2000 Prozessoren ergibt sich der Erwartungswert also zu 4 Millionen. Es ist für diesen Fall auch leicht zu überprüfen, daß $p(\leq 18 \cdot 10^6) < 0.99 < p(\leq 19 \cdot 10^6)$. Im schlimmsten Fall braucht der Algorithmus nach dieser Abschätzung also horrend viele Durchläufe, bis er den Deadlock findet. Man beachte jedoch, daß es sich bei dieser Abschätzung um eine extreme Worst-Case–Abschätzung handelt.

Bemerkung: Das beschriebene Problem der Phantom-Deadlocks ist in der Literatur bekannt als *phantom deadlock problem* und ist gelöst [BHG87, Seite 80f][AD90], wird jedoch nicht in allen Publikationen genannt [GR93, Spe97]. Insbesondere hat Spengler dieses Problem bei seiner Implementierung für die SB-PRAM übersehen. Die vorgestellte Lösung scheint neu zu sein.

5.5.3 Verknüpfung der Deadlock-Erkennung und der CC-Sperre

Es bleibt zu erklären, wie die vorgestellte Deadlock-Erkennung und die bisher entworfene Concurrency-Control verknüpft werden, und wann eine Zyklensuche durchgeführt wird.
Wir nehmen dazu zunächst an, daß ein PRAM–Prozessor mehrere Prozesse ausführen kann, die sich gegenseitig abbrechen können. Wir nehmen an, Prozessor P fordert eine Sperre an. Dann kann P in einem Prozeß die Sperre anfordern wie bisher, und in einem zweiten Prozeß die Zyklensuche durchführen. Erhält der erste Prozeß die Sperre, kann er die Zyklensuche abbrechen und P setzt die Bearbeitung wie bisher fort. Findet umgekehrt der zweite Prozeß bei der Zyklensuche einen Deadlock, so daß die Transaktion von P abgebrochen werden soll, so kann dieser Prozeß den ersten Prozeß und somit die Sperranforderung abbrechen.

Das PRAM-Programmiersystem und das Betriebssystem PRAMOS unterstützen die Abarbeitung mehrerer Prozesse durch einen Prozessor nicht. Für den speziellen Fall der Deadlock-Erkennung läßt sich dieses Problem aber durch eine Technik namens *Interleaving* lösen. Wir führen die Prozesse quasi-parallel aus, indem wir in der Funktion dfs_findcycle() (zweiter Prozeß) regelmäßig einige Befehle des ersten Prozesses ausführen. In Zeile (3) von Quelltext 5.5 wird vor jedem Schritt der Tiefensuche ein Teil der Zeilen (l3) bis (l6) der ürsprünglichen cc_lock()-Funktion (Quelltext 5.3) ausgeführt. Ist nach der Ausführung dieses Teils die angeforderte Sperre erhalten, wird durch einen goto-Sprung zur Zeile (9) die Zyklensuche abgebrochen und die Bearbeitung der Transaktion fortgesetzt. Soll die Transaktion von P abgebrochen werden, wird dies durch einen Sprung zu Zeile (10) eingeleitet.
Im Falle des Transaktionsabbruchs werden die bisherigen Änderungen rückgängig gemacht (ein sogenannter *Rollback*). Dieser Teil des DBS ist noch nicht implementiert, da er wesentlich von der noch nicht vorhandenen Logging/Recovery–Control abhängt.

Der untere Teil von Quelltext 5.5 zeigt die endgültige Version der cc_lock()-Funktion. Diese besteht nun aus den Initialisierungs-Zeilen (l1) und (l2) wie in Quelltext 5.3 und einer Endlos-Schleife, in der die Zyklensuche vorgenommen wird. Aus der Zyklensuche wird die Sperranforderung durch den beschriebenen Sprung zur Zeile (9) oder (10) beendet bzw. abgebrochen.
Man sieht leicht, daß sich an den Beweisen des vorherigen Abschnittes durch die neue Funktion cc_lock() nichts ändert.

Kapitel 6

SQL-Parser und TPC-B

In diesem Kapitel soll gezeigt werden, wie mit den in [Spe97] und dieser Arbeit entworfenen Komponenten der TPC-B–Benchmark [Rab92, Gra93] ausgeführt werden kann. Dazu erklären wir im folgenden Abschnitt einen einfachen *Parser* für den hier benutzten SQL-Teil. Anschließend geben wir einen Überblick über den TPC-B–Benchmark. Abschließend werden Messungen gezeigt.

6.1 SQL-Grammatik und Parsing

Abbildung 6.1 zeigt die SQL-Grammatik, die vom PRAM-Datenbanksystem unterstützt wird. Aus dieser Grammatik wird mit Hilfe der Unix-Programme lex und yacc [LMB92] ein *Parser* generiert. Dieser Parser erzeugt aus dem Quelltext einer Transaktion den *Ableitungsbaum* der Transaktion. Aus diesem Ableitungsbaum wird ein PRAM–C–Programm erzeugt, welches die Transaktion abarbeitet. Dieses Programm benutzt die Schnittstellen der Datenbank-Komponenten aus [Spe97] und dieser Arbeit. Zum Beispiel könnte in diesem C–Programm eine Sperre mittels der Funktion cc_lock() angefordert werden oder die Suchfunktion eines B^+-Baum ausgeführt werden. Das C–Programm, das aus dem Ableitungsbaum generiert wird, wird mit dem Compiler pgcc kompiliert, wobei die Datenbankkomponenten als Bibliothek an das Programm *gelinkt* werden. Das Ergebnis ist eine auf der SB-PRAM ausführbare Datenbank-Anwendung.

Die Grammatik in Abbildung 6.1 entspricht einem sehr kleinen Teil von SQL (siehe [MS93]). Der Parser wurde jedoch so implementiert, daß die unterstützte Grammatik leicht erweiterbar ist.

6.2 Der TPC-B–Benchmark

Der Benchmark TPC-B [Rab92, Gra93] des *Transaction Processing Performance Council (TPC)* ist ein weitbekannter Benchmark, der 1995 vom TPC für veraltet erklärt wurde. Er wurde ersetzt durch den neuen Benchmark TPC-C. Für die SB-PRAM soll trotzdem zunächst der TPC-B–Benchmark wegen seiner Einfachheit verwendet werden.
Beim TPC-B–Benchmark geht es um die Modellierung einer Bank. Die Bank hat n Filialen (*branch*), die in der Relation BRANCHES verwaltet werden. In

⟨Transaktion⟩	⟶	BEGIN WORK; ⟨Statement-List⟩ COMMIT WORK;
⟨Statement-List⟩	⟶	⟨Statement⟩; | ⟨Statement⟩; ⟨Statement-List⟩
⟨Statement⟩	⟶	⟨Update-Statement⟩ | ⟨Insert-Statement⟩
⟨Update-Statement⟩	⟶	UPDATE ⟨Rel-Name⟩ SET ⟨Set-List⟩ WHERE ⟨Where-Clause⟩
⟨Rel-Name⟩	⟶	⟨String⟩
⟨Set-List⟩	⟶	⟨Set-Statement⟩ | ⟨Set-Statement⟩, ⟨Set-List⟩
⟨Set-Statement⟩	⟶	⟨Attribut⟩ = ⟨Ausdruck⟩
⟨Attribut⟩	⟶	⟨String⟩
⟨Attribut-Liste⟩	⟶	⟨Attribut⟩ | ⟨Attribut⟩, ⟨Attribut-Liste⟩
⟨Variable⟩	⟶	$⟨String⟩
⟨Wert⟩	⟶	⟨Integer⟩ | ⟨Variable⟩
⟨Wert-Liste⟩	⟶	⟨Wert⟩ | ⟨Wert⟩, ⟨Wert-Liste⟩
⟨Ausdruck⟩	⟶	⟨Wert⟩ | ⟨Attribut⟩ | ⟨Ausdruck⟩ $[+, -, *, \div]$ ⟨Ausdruck⟩
⟨Where-Clause⟩	⟶	⟨Attribut⟩ = ⟨Variable⟩
⟨Insert-Statement⟩	⟶	INSERT INTO ⟨Rel-Name⟩ (⟨Attribut-Liste⟩) VALUES (⟨Wert-Liste⟩)

Abbildung 6.1: SQL–Grammatik

Das Unix-Programm yacc sorgt für die korrekte Priorisierung der Operatoren $+, -, *, \div$. Die Nicht-Terminale ⟨Integer⟩ und ⟨String⟩ werden vom Programm lex erkannt.

jeder Filiale gibt es 10 Kassierer (*teller*). Die Kassierer aller Filialen werden in der Relation TELLERS verwaltet. Weiterhin gibt es zu jeder Filiale 100.000 Konten (*account*); die Konten aller Filialen werden in der Relation ACCOUNTS gespeichert. Abschließend gibt es noch eine Relation HISTORY, in der Zahlungsvorgänge protokolliert werden. Tabelle 6.1 zeigt die Relationsschemata der einzelnen Relationen. Die Attribute filler in den Tabellen haben keine Bedeutung, sind aber durch die Banchmark-Spezifikation vorgeschrieben, um die Datensätze künstlich zu vergrößern.

Gemessen wird beim TPC-B–Benchmark, wieviele Transaktion pro Sekunde ausgeführt werden können. Die Anzahl der Transaktionen pro Sekunde wird mit *tpsB* bezeichnet. Die Anzahl n von Filialen muß gleich *tpsB* sein.
Die TPC-B–Transaktion ist in Quelltext 6.1 dargestellt. Die Transaktion ist von den Parametern Aid, Bid, Tid und Delta abhängig. Diese Transaktions-Parameter werden mit einem in der Benchmark-Spezifikation beschriebenen Verfahren bestimmt. Weiterhin stellt das Datenbanksystem jeder Transaktion die Variablen Date, Time und ProcID (Processor-ID) zur Verfügung.

Relation	Attribute	Wertebreich	Kommentar
BRANCHES	BranchID	integer	Identifizierungs-Nummer der Filiale
	Balance	integer	
	filler	char(92)	Auffüllen auf 100 Byte Länge
TELLERS	TellerID	integer	Identifizierungs-Nummer des Kassierers
	BranchID	integer	Filiale, zu der der Kassierer gehört
	Balance	integer	
	filler	char(88)	Auffüllen auf 100 Byte Länge
ACCOUNTS	AccountID	integer	Kontonummer
	BranchID	integer	Filiale, zu der das Konto gehört
	Balance	integer	Betrag des Kontos
	filler	char(88)	Auffüllen auf 100 Byte Länge
HISTORY	BranchID	integer	
	TellerID	integer	
	AccountID	integer	
	Delta	integer	
	Date	integer	Datum bei Transaktionsbeginn
	Time	integer	Uhrzeit bei Transaktionsbeginn
	ProcID	integer	Nummer des Prozessors, der die Transaktion ausführt
	filler	char(22)	Auffüllen auf 50 Byte Länge

Tabelle 6.1: Relationsschemata der TPC-B–Relationen.
Unterstrichen sind die Primärschlüssel-Attribute.

Quelltext 6.1 TPC-B–Transaktion

```
BEGIN WORK;

UPDATE ACCOUNTS SET Balance = Balance + $Delta
      WHERE AccountID = $Aid;

UPDATE TELLERS SET Balance = Balance + $Delta
      WHERE TellerID = $Tid;

UPDATE BRANCHES SET Balance = Balance + $Delta
      WHERE BranchID = $Bid;

INSERT INTO HISTORY (BranchID, TellerID, AccountID, Delta, Date, Time, ProcID)
      VALUES ($Bid, $Tid, $Aid, $Delta, $Date, $Time, $myID);

COMMIT WORK;
```

Die TPC-B–Transaktion entspricht der Einzahlung des Betrags Delta auf das Konto Aid. Dazu werden neben dem Balance-Wert des Kontos auch die Balance-Werte des Kassierers und der Filiale aktualisiert und abschließend ein Eintrag in die HISTORY-Tabelle geschrieben. Quelltext 6.2 zeigt Pseudo-Code für einen Update-Befehl der TPC-B–Transaktion, wie er vom SQL-Parser erzeugt wird. Dabei wird als Primärindex ein B^+-Baum angenommen.

In der Benchmark-Spezifikation wird gefordert, daß das Datenbanksystem das ACID-Prinzip (siehe Abschnitt 2.4) befolgt. Wie in Abschnitt 3.7 erwähnt,

Quelltext 6.2 Update der ACCOUNT-Relation

1. Sperre die ACCOUNT-Relation im Modus IX.
2. Suche im B^+-Baum nach der Seite, in der der Aid-Datensatz gespeichert ist. Sei dies die Seite p. Sperre p exklusiv.
3. Suche in der Seite p nach dem Aid-Datensatz. Sei r ein Zeiger auf diesen Datensatz.
4. Suche im Systemkatalog nach der Position des Attributs Balance. Sei o diese Position.
5. Setze r[o] = r[o] + Delta

verfügt das PRAM-Datenbanksystem über keine Logging/Recovery–Komponente. Daher wird das *Durabilty*–Prinzip nicht befolgt. Wir ignorieren dieses Manko im folgenden.

6.3 Messungen

Ursprüngliches Ziel dieser Arbeit war es, den *tpsB*-Wert der SB-PRAM zu messen. Da jedoch bei Abschluß der vorliegenden Arbeit die Festplatten-Treiber noch nicht vollständig waren, konnte der TPC-B–Benchmark nicht ausgeführt werden. Wir messen daher hier nur die *Pfadlänge* (Anzahl der ausgeführten Instruktionen) einer TPC-B–Transaktion, bei der keine Konflikte mit anderen Prozessoren entstehen, und für die alle benötigten Seiten im Systempuffer liegen. Weitere Messungen werden später als Technischer Bericht veröffentlich, sobald die Festplatten-Treiber fertiggestellt sind (vermutlich Herbst 1999).

Als Zugriffspfade können B^+-Bäume und Hash-Indizes verwendet werden. Die HISTORY-Relation werden wir immer mit einem Hash-Index verwalten. Ansonsten geben wir Messungen für B^+-Bäume und für Hash-Indizes an. Die Höhe der B^+-Bäume nehmen wir mit 2 für die Relation BRANCHES und 3 für TELLERS an, eine Höhe von 4 für die Relation ACCOUNTS[1]. Auf Relationsebene werden nur die nötigen Warnsperren gesetzt. Alle Daten werden auf Seitenebene gesperrt (Modi S und X), auf Tupelebene gibt es keine Sperren.
Tabelle 6.2 zeigt Laufzeiten, wenn die Primärindizes durch B^+-Bäume realisiert sind. Für das ACCOUNT-Update sind zum Vergleich die Schritte aus Quelltext 6.2 angegeben. Die Sperranforderungen zu Beginn der Transaktionsbearbeitung haben leicht unterschiedliche Dauer. Dies liegt an der LCB-Liste LCBLIST (siehe Abschnitt 5.3), die vor der Bearbeitung der Anforderung durchsucht wird. Im Laufe der Transaktionsbearbeitung wird diese länger, womit sich auch die Suchzeiten in der Liste vergrößern. Daher muß für komplexe Transaktionen mit vielen hundert Sperren die LCB-Liste vermutlich durch eine komplexere Suchstruktur (dynamische Hashtabelle [Meh84] o.ä.) ersetzt werden.

[1] Bei einem geschätzten Füllgrad von 70% im B^+-Baum [NM78] hat der Baum einen Verzweigungsgrad von ungefähr 360. In einer Datenseite mit 70% Füllgrad liegen 28 Datensätze. Dann reichen die angegebenen Höhen der Bäume für mehr als 10000*tpsB* aus.

Aktion	# Instr.	Kommentar
IX-Sperre auf ACCOUNT	380	Schritt 1 (siehe Quelltext 6.2)
IX-Sperre auf BRANCH	394	
IX-Sperre auf TELLER	404	
IX-Sperre auf HISTORY	420	
ACCOUNT-Seite suchen	4120	B^+-Baum mit Höhe 4 durchlaufen, innere Seiten S sperren, Blatt X sperren (Schritt 2)
Datensatz in Seite suchen	328	(Schritt 3)
UPDATE ausführen	4	(Schritt 4,5)
TELLER-Seite suchen	3326	B^+-Baum mit Höhe 3
Datensatz in Seite suchen	394	
UPDATE ausführen	4	
BRANCH-Seite suchen	2404	B^+-Baum mit Höhe 2
Datensatz in Seite suchen	364	
UPDATE ausführen	4	
HISTORY-Seite wählen	40	Hash-Funktion auswerten
Seite X sperren	508	inklusive fix()
Datensatz einfügen	598	
Sperren freigeben	1496	14 Sperren freigeben
$\sum$	15188	

Tabelle 6.2: TPC-B–Laufzeit mit B^+-Bäumen

Aktion	# Instr.	Kommentar
ACCOUNT-Seite wählen	40	Hash-Funktion auswerten
ACCOUNT-Seite sperren	524	inklusive fix()
TELLER-Seite wählen	40	
TELLER-Seite sperren	538	
BRANCH-Seite wählen	40	
BRANCH-Seite sperren	552	

Tabelle 6.3: Änderungen der TPC-B–Laufzeit durch Hash-Index

Tabelle 6.3 gibt Auskunft über die Änderungen an den Laufzeiten, wenn statt B^+-Bäumen Hash-Indizes benutzt werden. Es ändern sich im Vergleich zu Tabelle 6.2 nur die Zeilen, in denen eine Suche mittels B^+-Baum bzw. mittels Hash-Index ausgeführt wird.

Insgesamt ergeben sich Pfadlängen von 15188 und 7072 Instruktionen, je nach gewählter Index-Form. Bei diesen Messungen musste der Prozessor wohlgemerkt auf keine Sperre warten und alle benötigten Seiten lagen im Systempuffer.
Alle Zahlen in den Tabelle 6.2 und 6.3 unterliegen Schwankungen und können unter Umständen nicht genau reproduziert werden. Zum Beispiel hängt die Dauer einer Suche im B^+-Baum davon ab, wie dicht die Knoten besetzt sind. Die Dichte der Knoten kann auf unterschiedlichen Pfaden durch den Baum unterschiedlich sein. Auch die Länge der Hash-Listen in der Systempufferverwaltung können schwanken. Bei mehreren Messungen waren die Schwankungen jedoch so gering (max. 200 Instruktionen Unterschied, ungefähr 2%), daß die angegebenen Zahlen zumindest als Schätzung Aussagekraft haben.

Kapitel 7

Zusammenfassung und Ausblick

Simulationen des TPC-B–Benchmarks in [GJM+95] haben gezeigt, daß der an der Universität Saarbrücken entwickelte Parallelrechner SB–PRAM für Datenbankanwendungen sehr geeignet ist. Ziel dieser Arbeit war es, die Simulationsergebnisse experimentell zu bestätigen. Dazu wurde das PRAM–Datenbanksystem, das im Rahmen mehrerer Diplomarbeiten [Gem95, Spe97] entwickelt wurde, vervollständigt. Da jedoch die Festplattentreiber für die SB-PRAM bei Fertigstellung dieser Arbeit noch nicht vollständig implementiert waren, konnte der TPC-B–Benchmark noch nicht ausgeführt werden. Die Funktionsfähigkeit des Datenbanksystems konnte jedoch weitestgehend ohne die Festplatten überprüft werden, so daß nach Fertigstellung der Festplattentreiber der TPC-B–Benchmark ausgeführt werden kann.

Neben vielen kleinen Änderungen an den Basiskomponenten des Datenbanksystems wurde im Rahmen dieser Arbeit die Concurrency-Control–Komponente komplett neu entwickelt. Wesentlicher Bestandteil der Concurrency-Control–Komponente ist die CC-Sperre, welche die für Granular Locking benötigten Sperrmodi unterstützt. Zur Implementierung der CC-Sperre wurde ein neuer Ansatz gefunden, bei dem die CC-Sperre durch eine Kombination mehrerer Standard-Sperren aufgebaut wird. Da die verwendeten Standard-Sperren kompatible Sperranforderungen nicht serialisieren, werden auch in der CC-Sperre kompatible Sperranforderungen nicht serialisiert. Dies ist ein wesentlicher Vorteil gegenüber der üblichen Implementierung [Spe97, GR93], weshalb auf der SB-PRAM die Concurrency-Control in einigen Situationen um einen Faktor 200 beschleunigt wird. Diese Situationen treten bei Granular Locking regelmäßig auf. Die Korrektheit und Fairneß der neuen CC-Sperre wurde bewiesen.
Die Concurrency-Control wurde um eine Deadlock-Erkennung erweitert, die es den Prozessoren erlaubt, ohne Serialisierung den Wartegraphen nach Deadlocks zu durchsuchen. Die Korrektheit der Deadlock-Erkennung wurde ebenfalls bewiesen.

Als neue Komponente wurde für das Datenbanksystem ein einfacher SQL-Parser entwickelt. Dieser erlaubt die automatische Übersetzung von SQL-Anweisungen in C–Programme, die nach dem Kompilieren auf der SB–PRAM

ausgeführt werden können. Der entwickelte Parser unterstützt nur einen kleinen Teil von SQL, der für die Ausführung des TPC-B–Benchmarks nötig ist. Der Parser wurde jedoch erweiterbar aufgebaut.

Auch ohne die Festplattenanbindung konnte die Pfadlänge (Anzahl der Instruktionen) einer TPC-B–Transaktion gemessen werden, wenn keine Festplattenzugriffe nötig sind, und nicht auf Sperren gewartet werden muß. Je nach verwendeter Index-Form — unterstützt werden B^+-Bäume oder Hash-Indizes — ergeben sich Pfadlängen von ≈ 15000 und ≈ 7000 Instruktionen. Die gemessen Pfadlängen entsprechen ungefähr den in [GJM+95] angegebenen Zahlen.

Ausblick

Nach Fertigstellung der Festplattentreiber (voraussichtlich Herbst 1999) muß zunächst der TPC-B–Benchmark ausgeführt werden. Die Ergebnisse werden veröffentlicht, sobald sie verfügbar sind. Sollten die Ergebnisse wie erwartet gut ausfallen, steht einer Erweiterung des Datenbanksystems nichts entgegen. Der nächste Schritt wäre dann, die komplexeren Benchmarks TPC-C und TPC-D auf der SB–PRAM zu implementieren. Die für TPC-B entwickelten Basiskomponenten könnten dazu weiter verwendet werden. Allerdings muß für die komplexeren Benchmarks vermutlich die Zuordnung von Transaktionen zu einzelnen Prozessoren aufgelöst werden, so daß mehrere Prozessoren eine Transaktion verarbeiten. Zum Beispiel könnten parallele Joins [Gem95] und paralleles Sortieren [Gem95, GJR99] verwendet werden, um die Verarbeitung zu beschleunigen.

Literaturverzeichnis

[AD90] P. Anastassopoulos and J. Dollimore. A distributed deadlock detection and resolution algorihtm which does not suffer from phantom deadlocks (2). University of london, DOCS, London, 1990.

[AD+93] Feri Abolhassan, Reinhard Drefenstedt, et al. On the physical design of PRAMs. *COMPJ: The Computer Journal*, 36, 1993.

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Series in Computer Science and Information Processing. Addison-Wesley, Reading,Massachusetts;Menlo Park,California;London, 1974.

[Bac96] Peter Bach. *Entwurf und Realisierung der Prozessorplatine der SB-PRAM.* Diplomarbeit der Universitaet des Saarlandes FB14 (Paul). Universität des Saarlandes, Saarbrücken, 1996.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Amsterdam;Sydney;Singapore, 1987.

[Bhi88] A. Bhide. An analysis of three transaction processsing architectures. In *International Conference On Very Large Data Bases (VLDB '88)*, pages 339–350, Palo Alto, Ca., USA, August/September 1988. Morgan Kaufmann Publishers, Inc.

[BM72] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica, Springer Verlag (Heidelberg, FRG and NewYork NY, USA) Verlag*, 1(3), February 1972. Also published in/as: ACM SIGFIDET 1970, pp.107–141.

[Bos94] Michael Bosch. Portierung und Optimierung des GNU C-Compilers für die SB-PRAM. interner Report, Universität des Saarlandes, Saarbrücken, FB Informatik, 1994.

[BR97] Michael Bosch and Jochen Röhrig. PRAMOS — Ein Betriebsystem für die SB-PRAM. interner Report, Universität des Saarlandes, Saarbrücken, FB Informatik, 1997.

[BS87] A. Bhide and M. Stonebraker. Performance issues in high performance transaction processing architectures. In D. Gawlick, M. Haynie, and A. Reuter, editors, *Proceedings of the 2nd International*

Workshop on High Performance Transaction Systems, volume 359 of *LNCS*, pages 277–300, Berlin, September 1987. Springer.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge;London, 1990.

[Cod70] E. F. Codd. A relational model for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[DS92] Reinhard Drefenstedt and Dietmar Schmidt. On the physical design of butterfly networks for PRAMs. In *FMPSC: Frontiers of Massively Parallel Scientific Computation.* National Aeronautics and Space Administration (NASA), IEEE Computer Society Press, 1992.

[Edl95] Jan Edler. *Practical Structures for Parallel Operating Systems.* PhD thesis, New York University, Mai 1995.

[FW78] Steven Fortune and James Wyllie. Parallelism in random access machines. In *ACM Symposium on Theory of Computing (STOC '78)*, pages 114–118. ACM Press, 1978.

[Gem95] Christine Gemuend. *Eine Relationelle Datenbank für die SB-PRAM - Komplexe Operatoren und Sortieralgorithmen.* Diplomarbeit der Universitaet des Saarlandes FB14 (Paul). Universität des Saarlandes, Saarbrücken, 1995.

[GJM+95] Gemund, Jakob, Massonne, Paul, and Spengler. High performance transaction systems on the SB-PRAM. In *ISTCS: 3rd Israeli Symposium on the Theory of Computing and Systems*, 1995.

[GJR99] Thomas Grün, Christian Jacobi, and Jochen Röhrig. Fast quicksort parallelization on the SB–PRAM. Bisher unveröffentlicht, 1999.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, San Francisco, 1993.

[Gra93] Jim Gray, editor. *The Benchmark Handbook for Database and Transaction Processing, 2nd Edition.* Morgan Kaufmann, San Mateo, 2nd edition, 1993.

[Här87] Theo Härder. Realisierung von Operationalen Schnittstellen. In *Datenbank-Handbuch, Lockeman and Schmidt (eds.), Springer Verlag (Heidelberg, FRG and NewYork NY, USA).* 1987.

[Jan94] Jan Jannink. Implementing Deletion in B^+-Trees. Stanford University, CS Dept., Stanford, CA; verfügbar unter http://www-db.stanford.edu:80/pub/jannink/btree, 1994.

[JL99] Christian Jacobi and Cédric Lichtenau. Highly concurrent locking in shared memory database systems. Angenommen für *EUROPAR:*

Parallel Processing, 5th International EURO-PAR Conference, erscheint in LNCS, 1999.

[JLS99] Christian Jacobi, Cédric Lichtenau, and Bernd Spengler. Eine relationales Datenbanksystem für die SB-PRAM, Quelltexte. verfügbar unter ftp://ftp-wjp.cs.uni-sb.de/pub/pram/dbs/, 1999.

[Kel92] Joerg Keller. *Zur Realisierbarkeit des PRAM Modelles.* PhD thesis, Universität des Saarlandes, Saarbrücken, FB Informatik, Saarbrücken, 1992.

[Knu73] Donald E. Knuth. *The Art of Computer Programming. Volume 3 / Sorting and Searching.* Addison-Wesley Series in Computer Science and Information Processing. Addison-Wesley, Reading, Massachusetts;Menlo Park, California;London, 1973.

[KPS94] J. Keller, W. J. Paul, and D. Scheerer. Realization of PRAMs: Processor design. *Lecture Notes in Computer Science*, 857:17–??, 1994.

[KR90] R. M. Karp and V. Ramachandran. Parallel Algorithms for Shared-Memory Machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, chapter 17, pages 870–941. North Holland, 1990.

[Kun99] Stefan Kunde. Entwicklung des SCSI-Treibers für die SB-PRAM. Praktikumsausarbeitung, Universität des Saarlandes, Saarbrücken, FB Informatik (Draft), voraus. 1999.

[Lic96] Cédric Lichtenau. *Entwurf und Realisierung des Speicherboards der SB-PRAM.* Diplomarbeit (Paul). Universität des Saarlandes, Saarbrücken, 1996.

[LMB92] John R. Levine, Tony Mason, and Doug Brown. *Lex & Yacc.* O'Reilly, Sebastopol, CA, 1992.

[LPV81] G. Lev, N. Pippenger, and L. Valiant. A fast parallel algorithm for routing in permutation networks. *IEEE trans. on comp.*, C-30, 2:93–100, 1981.

[Meh84] Kurt Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching.* EATCS Monographs on Theoretical Computer Science. Springer, Berlin;Heidelberg;New York, 1984.

[MR94] Lory D. Molesky and Krithi Ramamritham. Efficient locking for shared memory database systems. Technical Report UM-CS-1994-010, Computer Science Department, University of Massachusetts, March 94.

[MS93] Jim Melton and Alan R. Simon. *Understanding the New SQL: A Complete Guide.* Morgan Kaufmann, San Francisco, 1993.

[NDD92] V. F. Nicola, A. Dan, and D. M. Dias. Analysis of the generalized clock buffer replacement scheme for database transaction processing. In *Proc. 1992 ACM SIGMETRICS and PERFORMANCE '92 Int'l. Conf. on Measurement and Modeling of Computer Systems*, page 35, Newport, Rhode Island, USA, June 1-5 1992.

[NM78] Toshiyuki Nakamura and Tetsuo Mizoguchi. An analysis of storage utilization factor in block split data structuring scheme. In S. Bing Yao, editor, *Fourth International Conference on Very Large Data Bases*, pages 489–495, West Berlin, Germany, 13–15 September 1978. IEEE-CS.

[PS82] David A. Patterson and Carlo H. Sequin. A VLSI RISC. *IEEE Computer*, 15(9):8–21, September 1982.

[PS85] J. L. Peterson and A. Silberschatz. *Operating System Concepts, 2nd ed.* Addison-Wesley (Reading MA), 1985.

[Rab92] Francois Rabb, editor. *TPC Benchmark B, Rev 1.1.* Shanley Public Relations, 777 N. First Street, Suite 600, San Jose, CA 95112-6311, March 1992.

[Röh96] Jochen Röhrig. *Implementierung der P4-Laufzeitbibliothek auf der SB-PRAM.* Diplomarbeit der Universitaet des Saarlandes FB14 (Paul). Universität des Saarlandes, Saarbrücken, 1996.

[Sch95] Dieter Scheerer. *Der Prozessor der SB-PRAM.* PhD thesis, Universität des Saarlandes, Saarbrücken, FB Informatik, Saarbrücken, 1995.

[Spe97] Bernd Spengler. *Eine Relationale Datenbank für die SB-PRAM - Entwicklung der Basiskomponenten.* Diplomarbeit (Paul). Universität des Saarlandes, Saarbrücken, 1997.

[Sto96] Michael Stonebraker. *Object-Relational DBMSs: The Next Great Wave.* Morgan Kaufmann Publ., San Francisco, 1996.

[Wal97] Thomas Walle. *Das Netzwerk der SB-PRAM.* PhD thesis, Universität des Saarlandes, Saarbrücken, 1997.

[Wil88] James M. Wilson. *Operating System Data Structures for Shared–Memory MIMD Machines with Fetch–and–Add.* PhD thesis, New York University, 1988.

Quelltext 4.1 Pseudo-Code für die Funktion fix()

```
        PCB fix(rid, pid, prio)
        unsigned integer rid, pid, prio;
        {
(1)         listennr := h_PA(rid, pid);
(2)         rw_lock(PAGEARRAY[listennr].lock, READER);
(3)         suche nach (rid,pid)-PCB in PAGEARRAY[listennr].first
(4)         if (gefunden)
Buffer–hit
(5)             sei p der (rid,pid)-PCB
(6)             sbp_mpadd(p.users, 1);              /* fixiere Seite */
(7)             p.clock := prio;                                          (*)
(8)             rw_unlock(PAGEARRAY[listennr].lock, READER);
(9)             return p;
(10)        else
Buffer–miss
(11)            rw_unlock(PAGEARRAY[listennr].lock, READER);
(12)            new_pcb := get_pcb();       /* fordert freien Slot an */
(13)            rw_lock(PAGEARRAY[listennr].lock, WRITER);
(14)            suche nach (rid,pid)-PCB in PAGEARRAY[listennr].first
(15)            if (gefunden)     /* anderer Prozessor war schneller */
(16)                sei p der (rid,pid)-PCB
(17)                sbp_mpadd(p.users, 1);          /* fixiere Seite */
(18)                p.clock := prio;                                      (*)
(19)                rw_unlock(PAGEARRAY[listennr].lock, WRITER);
(20)                put_pcb(new_pcb);     /* gibt freien Slot zurück */
(21)                return p;
(22)            else
(23)                initiiere Laden der Seite von Festplatte
                        nach new_pcb.buf
(24)                initialisiere new_pcb         /* u.a. Seite fixieren */
                    Initialisiere OCB (siehe Abschnitt 5.3)
(25)                new_pcb.prio := prio;                                 (*)
(26)                setze new_pcb an den Listenanfang von
                        PAGEARRAY[listennr].first
(27)                rw_unlock(PAGEARRAY[listennr].lock, WRITER);
(28)                return new_pcb;
        }
```

Quelltext 5.3 Die Funktionen cc_lock() und cc_unlock()

cc_lock(l,φ)	$\equiv$	waitmode[myID]:=φ	(l1)
		waitfor[myID]:=l	(l2)
		$\text{cc\_lock}_{\text{alt}}$(l,$\varphi$)	(l3)
		waitfor[myID]:=PSEUDO	(l4)
		warte $\lceil d_2/C \rceil$ Runden (füge 6 No-Operation-Befehle ein)	(l5)
		l.mode[myID]:=φ	(l6)
cc_unlock(l,φ)	$\equiv$	l.mode[myID]:=frei	(u1)
		$\text{cc\_unlock}_{\text{alt}}$(l,$\varphi$)	(u2)

Quelltext 5.4 Implementierung des Adjazenz-Tests

			V-Zeitpunkt des Speicherzugriffs	
TEST	$\equiv$	l:=waitfor[v]	t	(t1)
		φ:=waitmode[v]	$t + d_1$	(t2)
		ψ:=l.mode[u]	$t + d_2$	(t3)
		testerg := φ inkompatibel zu ψ		(t4)
		//Test über Table-Lookup		

Quelltext 5.5 Funktionen dfs_findcycle() und cc_lock()

```
        dfs_findcycle(v)
        node v;
        {
(1)         status[v] := besucht
(2)         foreach u ≻ v        //traversiere zu v adjazente Knoten
(3)             führe einige Instruktionen
                    der cc_lock-Funktion, Zeilen (l3)-(l6) aus)
                if (myID hat die Sperre erhalten)
                    goto (9)
                if (status[u] = nicht_besucht)
(4)                 dfs_findcycle(u)
                    status[u] := abgeschlossen
                else if (status[u] = besucht)
                    if (u = myID)
                        // scheinbarer Zyklus gefunden
(5)                     überprüfe gefundenen Zyklus rückwärts
(6)                     falls Zyklus "verschwunden", reinitialisiere
                            status[ ] und starte Zyklensuche neu
(7)                     Prozessor myID befindet sich im Deadlock,
                            synchronisiere alle beteiligten Prozessoren und leite
                            Abbruch einer Transaktion ein.
                        if (Transaktion von myID soll abgebrochen werden)
                            goto (10)
                        else
(8)                         reinititalisiere status[ ], starte Zyklensuche neu
            return;
        }

        cc_lock(l,φ)
        {
(l1)        waitmode[myID]:=φ
(l2)        witfor[myID]:=l
            forever
                initialisiere status[ ] := nicht_besucht
                dfs_findcycle(myID)
(9)         return LOCK_OK
(10)        return LOCK_DEADLOCK
        }
```

Quelltext 5.6 Implementierung der foreach-Schleife

```
Wähle zufällig und gleichverteilt eine Permutation π der Zahlen {0,...,C − 1}
for u'=0 to C-1
    u:=π(u')
    if u ≻ v
        do something
```