Entwurfsentscheidungen und Einflußfaktoren der Speichersynthese für allgemeine Prozessor-Systeme

Renate Beckmann
Lehrstuhl Informatik XII
Universität Dortmund

Forschungsbericht Nr. 546

August 1994

Zusammenfassung

Dieser Bericht befaßt sich mit der Synthese von Speichern für allgemeine Prozessor-Systeme. Die Qualität solcher Speicher hängt sehr von den Zugriffseigenschaften der Anwendungsprogramme ab, die wiederum von dem Anwendungsbereich und dem entsprechenden Prozessor-System abhängen. Die Aufgabe dieser Speichersynthese ist es, für gegebene Anwendungsbereiche auf einem bestimmten Rechner-System (mit einem oder mehreren Prozessoren) eine optimale Speicherhierarchie (Cache, Hauptspeicher, usw.) hinsichtlich Zugriffszeit und Kosten zu entwerfen. Da heutige Speichersysteme zunehmend komplexer und ausgefeilter werden, wird für deren Entwurf Werkzeugunterstützung notwendig. Hier werden die einzelnen Entwurfsentscheidungen sowie die verschiedenen Faktoren, die einen optimalen Speicherentwurf beeinflussen und entsprechende Zusammenhänge näher beschrieben.

Abstract

This report deals with the memory synthesis for general purpose processors. The performance of such memory systems highly depends on the access behavior of the application programs, which, in turn depends on the application domain and the host processor system. Processor memory synthesis has to design a memory hierarchy for a given appliation domain and a given host processor system, which has an optimal performance relative to access time and costs. Due to the increasing complexity of memory systems a tool support is necessary. This report describes the memory design decisions as well as the factors influencing an optimal memory design, and the relations between them.

Inhaltsverzeichnis

1. Einleitung

Durch die zunehmend komplexer werdenden mikroelektronischen Systeme wird auch deren Entwurf immer komplexer und zeitintensiver. Damit der Entwurf solcher Systeme (sowohl bzgl. Zeit als auch Komplexität) handhabbar wird, wurden im Laufe der Zeit automatische Entwurfswerkzeuge für immer höhere Abstraktionsebenen entwickelt: Layout-Editoren sind zum großen Teil durch automatische Plazierungs- und Verdrahtungswerkzeuge ersetzt worden. Weitere Werkzeuge für die Logik-Synthese und die High-Level-Synthese sind entwickelt worden. Mit Hilfe von High-Level-Synthese-Werkzeugen können heute bereits Digital-Filter für Sprache und Video schnell und zuverlässig entworfen werden, und laut Ankündigungen von kommerzieller Seite (Synopsis, IBM) werden High-Level-Synthese-Werkzeuge nun auch kommerziell zur Verfügung gestellt.

Bisher unterstützen diese Werkzeuge den Entwurf einzelner Schaltungen. Eine Erweiterung auf System-Ebene beinhaltet eine Reihe von noch nicht ausreichend untersuchten Aufgaben, wie z.B. Hardware-/Software-Codesign, Zeitverhalten auf System-Ebene, Code-Generierung für eingebettete Systeme (embedded systems) und Speichersynthese.

Die ersten der o.a. Aufgaben auf System-Ebene sind bereits seit einiger Zeit Gegenstand der Forschung, bis auf wenige Ausnahmen jedoch nicht die Speichersynthese:

- Speichersynthese für DSP-Anwendungen [Meer92, Bala93]
- Register-Allokation in Compilern [Aho86] und High-Level-Synthese-Werkzeugen [Gajs92]
- Allokation von Multiport-Speichern in High-Level-Synthese-Werkzeugen [Bala88, Kim92]
- Puffer-Allokation in High-Level-Synthese-Werkzeugen [Kolk93, Prad93]

In diesem Bericht geht es um die Speichersynthese für allgemeine Prozessor-Systeme wie Workstations und PCs, die in der Literatur bisher nicht behandelt wird. Um die steigende Geschwindigkeit von Prozessoren mit einer genügenden Speicher-Bandbreite zu unterstützen, sind Speicher-Systeme immer komplexer geworden. Komplexe Hierarchien mit unterschiedlichen Arten von Speichermodulen und fortgeschrittene Techniken, wie kompliziert organisierte Caches, Speicherverschränkung, Pipelining, Bus-Snooping, usw., werden nicht mehr nur in teuren Großrechner, sondern auch in kleineren Systemen (wie Workstation, PC) eingesetzt.

Die gegenwärtige Situation im Bereich Speichersynthese für allgemeine Prozessor-Systeme läßt sich wie folgt charakterisieren:

- Für die Speichersynthese für allgemeine Prozessor-Systeme gibt es noch keine automatische Werkzeugunterstützung, die Aufgabe wird ausschließlich durch Experten von Hand ausgeführt.
- Der Entwurf eines Speichersystems wird sehr stark durch Heuristiken geleitet, deren Qualität nur schwer beurteilt werden kann.
- Die Zusammenhänge zwischen den verschiedenen Entwurfsentscheidungen sind sehr komplex und unstrukturiert, so daß es schwierig ist, hier einen kompletten Überblick zu bekommen.

Die Speichersynthese für allgemeine Prozessor-Systeme ist zur Zeit also eher eine Kunst als eine Wissenschaft, d.h. sie ist noch keine Ingenieur-Disziplin. Dies führt dazu, daß selbst wichtige industrielle Firmen Entwurfsentscheidungen treffen, die zu einem Speicher-System führen, dessen Leistung wohl nicht ihren Erwartungen entspricht.[1]

Diese Situation ist also durchaus nicht zufriedenstellend. Um dies zu ändern, wird zunächst der Aufgabenbereich der allgemeinen Speichersynthese strukturiert, indem die verschiedenen Entwurfsentscheidungen sowie die relevanten Faktoren des Umfeldes zusammengestellt und auf verschiedene Beziehungen zueinander untersucht werden. Damit wird das nötige Entwurfswissen zusammenge-

1. Beispielsweise sind in modernen SPARC-Systemen wie der SPARC-10 die Blockgrößen der Caches und die Anzahl der Module bei der Speicherverschränkung nicht gut aufeinander abgestimmt.

stellt. Dies ist eine notwendige Voraussetzung, um ein Konzept für diese Aufgabe entwickeln zu können.

Die Aufgabe

Speicherhierarchien, die mit einem Speichersynthese-Werkzeug entworfen werden sollen, bestehen aus Speichermodulen unterschiedlicher Größe mit entsprechend unterschiedlichen Zugriffsgeschwindigkeiten (Abb. 1). Hierbei spielt vor allem die Verwendung eines Caches eine besondere Rolle, der

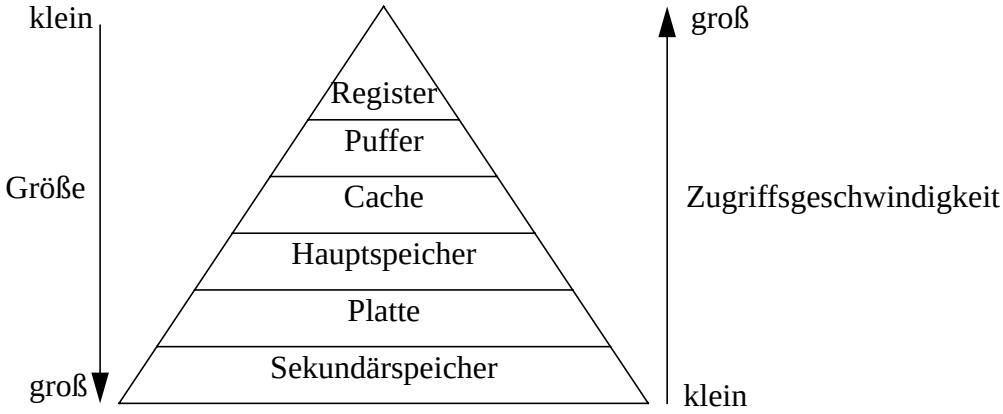

Abb. 1: Speicherhierarchie

durch die zunehmende Integrationsdichte heute auf dem Prozessorchip integriert werden kann. Eine unnötige Zeitverzögerung wird verhindert, wenn viele der von einem laufenden Prozeß benötigten Daten in Zwischenspeichern auf dem Prozessor-Chip gehalten werden (z.B. durch Prefetching oder Assoziativität beim Cache, s.u.). Für die einzelnen Speichermodule gibt es verschiedene Maßnahmen, um den Zugriff weiter zu beschleunigen (z.B. Speicherverschränkung beim Hauptspeicher, die die Zugriffe parallelisiert, s.u.). D.h. die Möglichkeiten für den Entwurf von Speicherhierarchien sind sehr komplex und vielfältig, wobei sich die einzelnen Entwurfsentscheidungen stark beeinflussen. Mit der Einführung von Multiprozessor-Systemen, bei denen mehrere Prozessoren dieselben Speichermodule benutzen können, um einen Datenaustausch zu beschleunigen, sind außerdem Maßnahmen zur Erhaltung der Datenkonsistenz verstärkt notwendig.

Die Aufgabe der hier beschriebenen Speichersynthese für allgemeine Prozessor-Systeme ist es also, abhängig von den Anwendungsprogrammen und dem zugrundeliegenden Rechnersystem eine Speicherhierarchie zu entwerfen, die hinsichtlich Zugriffszeit und Kosten (Fläche bzw. Preis) optimal ist. Hierbei geht es um den abstrakten Entwurf, der konzeptionelle Entscheidungen wie Größe, Assoziativität, Grad der Speicherverschränkung, usw. trifft, nicht aber um die Abbildung auf Transistorebene mittels Bausteinbibliotheken oder Modulgeneratoren.

Die Optimalität eines solchen Speicherentwurfs hängt, wie schon erwähnt, stark von zwei Dingen ab: Zum einen spielt das Zugriffsverhalten der laufenden Anwendungsprogramme auf den Speicher eine große Rolle: Greift z.B. ein Programm nur auf wenig verschiedene Daten zu, genügt ein kleiner Zwischenspeicher (Cache), auf den schneller zugegriffen werden kann als auf größere, oder läßt sich das Zugriffsverhalten gut vorhersagen, weil meistens auf den Inhalt der dem letzten Zugriff folgenden Adresse zugegriffen wird, können die benötigten Daten vor der Benutzung in den Cache geladen werden. Zum anderen hat auch die Architektur des (Multi-)Prozessor-Systems Einfluß: Werden z.B. die Befehle mittels Pipelining abgearbeitet, empfiehlt es sich, einen getrennten Cache für Daten und Instruktionen und eine Speicherverschränkung vorzusehen, damit diese Zugriffe parallel ablaufen können.

Die Einflußfaktoren sowie die Zusammenhänge beim Entwurf von Speichern sind also sehr vielfältig und stehen nicht (annähernd) vollständig zur Verfügung, was jedoch für die Entwicklung einer automatischen Werkzeugunterstützung notwendig ist.

Der Bericht

Dieser Bericht gibt zunächst einen Einblick in die Problematik der o.a. Speichersynthese. Im zweiten Kapitel werden die besonderen Problemeigenschaften beschrieben, und es wird eine Konzeptidee für eine erste Werkzeugunterstützung beim Entwurf von Speicherhierarchien für allgemeine Prozessor-Systeme vorgestellt.

Die Kapitel drei bis sechs enthalten eine umfassende Sammlung der verschiedenen Einflußfaktoren, Entwurfsentscheidungen und Ziele bei der Speichersynthese und beschreiben diese genauer: Es werden jeweils die möglichen Wertebereiche und einzelne Bemerkungen darüber, wie diese Entwurfsentscheidungen zu treffen sind, angegeben. Sie sind nach Klassen (s.u.) sortiert. Die Art der Entwurfsentscheidungen vor allem für den Cache konnten zum großen Teil aus der Literatur entnommen werden.

Kapitel sieben beschreibt die verschiedenen Beziehungen und Bedingungen der Einflußfaktoren und Entwurfsentscheidungen untereinander und zueinander. Sie sind z.T. aus zahlreichen Literaturquellen zusammengetragen, wo verteilt in verschiedenen Analysen Beziehungen zwischen einigen wenigen Entwurfentscheidungen angegeben sind. Sie wurden gegeneinander abgewogen und bewertet. Ein weiterer großer Teil konnte in Diskussionen gesammelt werden.

Hierbei beschränkt sich der Bericht auf den Cache, einzelne Puffer und den Hauptspeicher, wobei der Cache sehr ausführlich behandelt wird, weil der Entwurf besonders kritisch bzgl. der Leistung des Speichersystems ist. Die Register werden gewöhnlich beim Prozessorentwurf ausreichend mit berücksichtigt [Bala88, Cheu90, Kim92]. Der Entwurf von Platte und Sekundär- bzw. Tertiärspeicher wird hier nicht behandelt.

2. Die Speichersynthese

2.1 Eigenschaften der Speichersynthese

- *Speichersynthese auf hoher Ebene*
  Bei der Speichersynthese wie sie hier beschrieben wird geht es um den Entwurf auf hoher Abstraktionsebene (Register-Transfer-Ebene). Es sollen für eine Speicherhierarchie konzeptionelle Entwurfsentscheidungen getroffen werden, d.h. aus welchen Speichermodulen sie besteht und wie diese arbeiten. Für die niedrigeren Ebenen gibt es Modulgeneratoren oder Bibliotheken, die mit Hilfe einiger der hier festzulegenden Werte Speicherbausteine generieren oder zur Verfügung stellen.
- *Konfigurierung = Strukturierung + Dimensionierung*
  Die Speichersynthese oder das Konfigurieren eines Speichers setzt sich aus zwei Teilaufgaben zusammen: Es muß festgelegt werden, aus welchen Modulen die Speicherhierarchie bestehen soll, wieviele Puffer, Caches und Hauptspeicher es gibt. Das sei hier Strukturierung genannt. Strukturentscheidungen beeinflussen die Anzahl der Entwurfsentscheidungen. Für jede Art von Speichermodul gibt es eine feste Anzahl von Entwurfsentscheidungen, die zu treffen sind. Stellt man diese als Parameter dar, so müssen die Parameter eingestellt werden, d.h. es muß dimensioniert werden.
- *wenig Strukturierung, viel Dimensionierung*
  Es gibt relativ wenige Strukturierungsentscheidungen (wie z.B. ob der Cache für Daten und Instruktionen getrennt wird, ob es einen zweiten größeren Cache gibt, und wieviele Prozessoren einen Cache auf welche Ebenen und Hauptspeichermodule gemeinsam benutzen). Hingegen gibt es für jedes Speichermodul einige Entwurfsentscheidungen, die aus einer großen Menge von Möglichkeiten eine auswählen müssen. D.h. die Anzahl der Entwurfsentscheidungen ist auf wenige Hundert beschränkt, aber der Entwurfsraum ist wegen der Kombination der großen Anzahl und Art von Möglichkeiten je Entwurfsentscheidung sehr groß.

- *unterschiedliche Arten von Entwurfsentscheidungen*
  Eine Entwurfsentscheidung kann von unterschiedlichster Art sein: boolesch (z.B. Wahl einer Speicherverschränkung)[1], numerisch (z.B. Größe des Speichermoduls), symbolisch (z.B. Auswahl des Verfahrens für das Rückschreiben) oder strukturell (z.B. Vorhandensein eines Caches).
- *viele Zusammenhänge zwischen Einflußfaktoren und Entwurfsentscheidungen und untereinander*
  Alleine die Beziehungen und Abhängigkeiten der verschiedenen Entwurfentscheidungen untereinander sind sehr zahlreich (z.B. wird bei einem kleinen Cache auch eine kleine Blockgröße gewählt oder das Prefetching ist nur bei kleinen Blockgrößen sinnvoll, usw.). Wie bereits oben beschrieben hängt der Speicherentwurf auch sehr stark von der entsprechenden Anwendung und der zugrunde liegenden Rechnerarchitektur ab. Die Speichersynthese wird also sehr stark beeinflußt oder gelenkt durch diese Beziehungen und Abhängigkeiten.
- *Beziehungen von Entwurfentscheidungen zur Leistung vage und unscharf*
  Es gibt keine direkt abzulesenden Beziehungen zwischen den getroffenen Entwurfsentscheidung und der Leistung des Speicher. Es ist z.B. nicht eindeutig zur Entwurfszeit zu beurteilen, welchen konkreten Einfluß die Cache-Größe auf die durchschnittliche Zugriffszeit hat (je kleiner der Cache, desto schneller ein Zugriff, jedoch desto häufiger muß auf den langsameren Hauptspeicher zugegriffen werden, weil das Datum nicht im Cache ist).
- *Bewertung mittels Simulator*
  Aufgrund der Vagen Beziehungen zwischen Entwurfsentscheidungen und Leistung ist eine Bewertung der Entscheidungen zur Entwurfszeit außerordentlich schwierig. Dies führt zu der Notwendigkeit, sowohl einen Simulator, der den Speicherentwurf bewertet, als auch Zyklen zur Entwurfsverbesserung vorzusehen.
- *Speichersynthese als Optimierungsproblem*
  Beim Entwurf eines Speichers geht es nicht nur darum, irgendeine korrekte und realisierbare Speicherhierarchie zu entwickeln, sondern bzgl. Zugriffszeit und Kosten zu optimieren. Dies ist gerade in Hinblick auf die vagen Beziehungen zwischen Entwurfs- und Leistungsfaktoren schwierig. Hierzu sind verschiedene aus der Literatur bekannte Optimierungs- und Simulationsverfahren notwendig.
- *Speichersynthese, ein neues Forschungsfeld*
  Es gibt in der Literatur viele Untersuchungen und Analysen über einige der Entwurfsentscheidungen beim Cache und deren Auswirkungen auf indirekte Leistungsfaktoren. Diese Analysen basieren, bis auf sehr wenige Ausnahmen, auf einer festen Menge von Anwendungsprogrammen und einer ausgewählten Rechnerarchitektur. Sie betrachten nicht, wie sich die Analyse-Ergebnisse bei wechselnden Anwendungsprogrammen und Rechnerarchitekturen verhalten, obwohl dies sehr wohl Einfluß auf die Leistung des jeweiligen Caches / Hauptspeichers hat (Einflußfaktoren). Es sind keine konkreten Entwurfsregeln oder gar Systeme beschrieben, wie Caches oder Hauptspeicher unter Berücksichtigung der Einflußfaktoren zu entwerfen sind. Bei der Automatisierung der Speichersynthese steht die Forschung also noch am Anfang und ist sehr auf das Wissen derjenigen angewiesen, die diese Arbeit bisher von Hand erledigt haben.

2.2 Konzeptidee eines Speichersynthese-Werkzeugs

Eine Speicherhierarchie kann als hochgradig generisches Modell repräsentiert werden, dessen Parameter die Entwurfsentscheidungen darstellen. Die Speichersynthese läßt sich dann als das Einstellen dieser Parameter darstellen. Zum einen gibt es Parameter, die die Struktur festlegen, d.h. aus welchen Speichermodulen die Speicherhierarchie besteht, und wie diese logisch gesehen zueinander angeord-

1. Einige boolesche Entwurfsentscheidungen sind nicht explizit angegeben, diese werden durch numerische Folgeentscheidungen mit ausgedrückt. Ist z.B. bei der Speicherverschränkung die Anzahl der Module gleich bzw. größer eins, entspricht dies einer Entscheidung gegen bzw. für die Speicherverschränkung.

net werden. Zum anderen wird jedes dieser Speichermodule je nach Art durch eine feste Menge von Parametern beschrieben (Dimensionierung). Die Aufgabe der Speichersynthese entspricht dann dem Wählen bzw. Einstellen dieser Parameter.

Ebenso können die Einflußfaktoren der zugrundeliegenden Rechnerarchitektur sowie der Anwendung mit Hilfe von Parametern beschrieben werden: Die Parameter der Rechnerarchitektur können direkt aus den Prozessor-/Systembeschreibungen abgelesen werden. Die Parameter der Anwendung müssen zunächst durch Analyse von (typischen) Anwendungsprogrammen bzw. der zugehörigen Adreßsequenz von Speicherzugriffen extrahiert werden.

Wie bereits erwähnt gibt es sehr viele Beziehungen und Bedingungen zwischen den Parametern der Rechnerarchitektur, der Anwendung und der Speicherhierarchie. Diese schränken die Wahlmöglichkeiten der Speicher-Parameter stark ein. Dieses Konzept läßt sich sehr gut auf das Konzept der logischen Programmierung mit Randbedingungen oder Constraints (CLP - constraint logic programming) abbilden, das eben gerade die Implementierung der Bedingungen zwischen den zahlreichen Parametern unterstützt. Zudem kommen die bekannten Vorteile der logischen Programmierung, wie hohe Abstraktionsebene, schnelle Implementierung, gute Wartbarkeit besonders dem Problem der Speichersynthese zugute, weil es auf diesem Gebiet noch wenig Erfahrung gibt. Dies führt dazu, daß u.U. häufigere Änderungen am Konzept notwendig sind.

Abbildung 2 zeigt den konzeptionellen Aufbau eines Speichersynthese-Werkzeugs. Zuerst werden

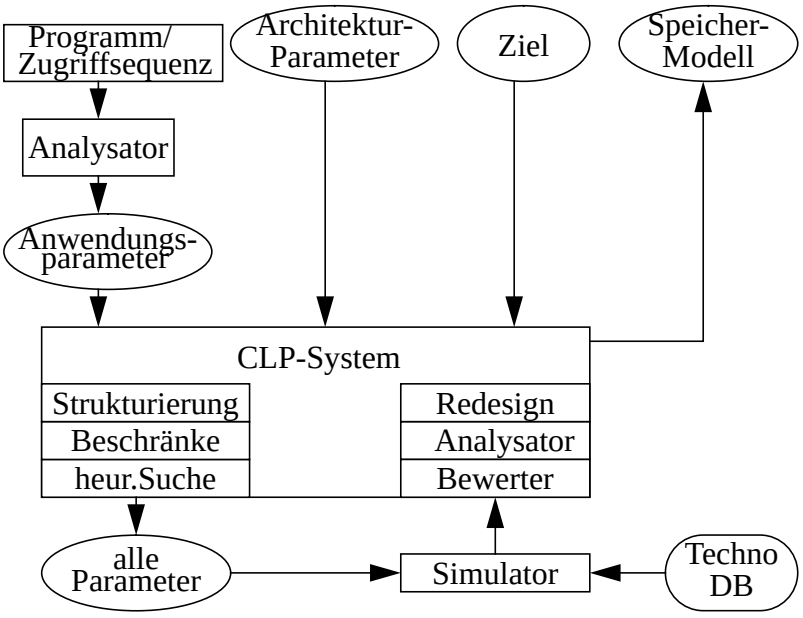

Abb. 2: Aufbau eines Speichersynthese-Werkzeugs

alle für die Speichersynthese nützlichen und vorhandenen Einflußfaktoren angegeben. Dafür wird zunächst ein für das Anwendungsgebiet typisches Programm analysiert, um die relevanten Einflußfaktoren in Form von Anwendungsparametern zu extrahieren. Die Architektur-Parameter können der Beschreibung der Rechnerarchitektur, für den die Speicherhierarchie entworfen werden soll, entnommen werden. Desweiteren können obere Schranken, die einzuhalten sind, oder Ziele angegeben werden, bzgl. dessen die Speicherhierarchie zu optimieren ist (z.B. Zugriffszeit oder Kosten).

Mit Hilfe von Bedingungen zu den Anwendungs- und Architektur-Parametern und dem angegebenen Ziel wird nun die Struktur der Speicherhierarchie durch die Anzahl ihrer Module festgelegt (Strukturierung). Jedes Speichermodul hat eine feste Menge von Parametern mit klar definierten Wertebereichen. Mit Hilfe von weiteren Bedingungen werden nun diese Wertebereiche der Speicher-Parameter eingeschränkt (Beschränke in Abb. 2).

Hier können zwei Probleme auftreten: Sind die Bedingungen zu stark bzw. inkonsistent (overconstrained), kann keine Lösung gefunden werden. Für diesen Fall wird die Menge der Bedingungen in verschiedene Klassen je nach “Härtegrade” aufgeteilt, die die Wichtigkeit der Bedingungen widerspiegeln. Die Bedingungen werden nun schrittweise je nach Härtegrad ausgewertet. Kommt es zu einer Inkonsistenz, wird die zuletzt ausgewertete Klasse von Bedingungen zurückgenommen.

Das zweite Problem bezieht sich darauf, daß die Bedingungen zu schwach sind (underconstrained), d.h. nach Auswertung aller Klassen von (konsistenten) Bedingungen sind die Speicher-Parameter noch nicht eindeutig festgelegt. Dann muß eine heuristische Suche angewendet werden, d.h. die Speicher-Parameter müssen innerhalb ihres durch die Bedingungen eingeschränkten Wertebereiches festgelegt werden. Hierfür gibt es verschiedene Heuristiken, die sich an dem angegebenen Ziel orientieren.

Wie bereits erwähnt, kann die Qualität des Speicherentwurfs nicht direkt anhand der Speicher-Parameter bewertet werden, sondern es ist eine Simulation nötig. Hierzu werden alle Eingaben sowie die Speicher-Parameter an den Simulator weitergegeben. Existiert zum eingegebenen Anwendungsprogramm noch keine Adreßsequenz der Speicherzugriffe, muß eine Adreßsequenz generiert werden, die den extrahierten Anwendungsparametern entspricht. Der Simulator führt eine ereignisgesteuerte Simulation durch. Die technischen Daten, wie z.B. Zugriffszeit, werden der Technologie-Datenbank entnommen. Die Ergebnisse der Simulation werden bzgl. der Zielvorgaben bewertet.

Ist das Ergebnis zufriedenstellend, wird die erzeugte Speicherhierarchie ausgegeben. Ansonsten muß analysiert werden, welche Ursachen (z.B. zu hohe Fehlschlagsrate beim Cache, oder zu lange Zugriffszeit) für das ungenügende Ergebnis verantwortlich sind. Sind die Ursachen gefunden, werden einige der Speicher-Parameter in einem Redesign-Zyklus verändert. Hierfür gibt es verschiedene Redesign-Regeln. Die restlichen Speicher-Parameter müssen daraufhin entsprechend mittels Bedingungen und heuristischer Suche angepaßt werden. Dieser neue Entwurf wird wieder simuliert und bewertet. Erfüllt der Entwurf die Bewertungsvorgaben, wird abgebrochen, sonst muß ein neuer Redesign-Zyklus gestartet werden.

2.3 Die verschiedenen Klassen von Parametern

Hier soll eine grobe Übersicht über alle für die Speichersynthese relevanten Einflußfaktoren und Entwurfsentscheidungen gegeben werden:

- Speicher-Parameter
  Die Entwurfsentscheidungen beziehen sich auf die zu entwerfende Speicherhierarchie, d.h. sie entsprechen in dem hier vorgestellten Konzept den Speicher-Parametern. Speicher-Parameter werden also von einem Speichersynthese-Werkzeug berechnet, können aber, falls gewünscht, auch eingegeben werden. Sie lassen sich weiter unterteilen in Parameter für Puffer, Cache, Hauptspeicher, MMU/TLB und Multiprozessor-System.
- Anwendungsparameter
  Die Faktoren der Anwendungsprogramme, die für die Speichersynthese relevant sind, sind in den Anwendungsparametern festgehalten. Einige dieser Parameter sollen, soweit möglich, vorgegeben werden, andere werden mit Hilfe von Analysen von Beispielprogrammen oder deren Adreßsequenzen bestimmt.
- Architektur-Parameter
  Die für die Speichersynthese relevanten Eigenschaften der zugrundeliegenden Rechnerarchitektur, für die die Speicherhierarchie entworfen werden soll, werden durch die Architektur-Parameter repräsentiert. Sie beinhalten also relevante Details über den inneren Aufbau der Prozessoren des Rechner-Systems. Sie sollen soweit wie möglich vorgegeben werden, um den Speicher optimal darauf anpassen zu können.

- Ziel-Parameter
  Die direkten und indirekten Bewertungsmaßstäbe werden anhand der Ziel-Parameter überprüft. Sie können als zu erreichen vorgegeben werden, meist in Form von oberen oder unteren Schranken.

In der Literatur ist der Cache das am meisten untersuchte Speichermodul. Hierzu gibt es eine Sammlung von Entwurfsentscheidungen (Cache-Parameter) und einige Untersuchungsreihen bzgl. einiger dieser Parameter. Parameter der anderen Speichermodule, sowie die Beziehungen und Bedingungen zu Anwendung und Architektur werden bis auf wenige Ausnahmen nicht beschrieben.

Die Sammlung, Strukturierung und Implementierung eben dieser Bedingungen zur Festlegung der zu treffenden Entwurfsentscheidungen stellt eine der Hauptaufgaben der Speichersynthese dar.

Die folgenden Kapitel enthalten eine Sammlung sowohl der Speicher- als auch der Anwendungs-, Architektur- und Ziel-Parameter sowie der Bedingungen und Beziehungen hauptsächlich der Cache-Parameter zu den restlichen Parametern.

3. Speicher-Parameter

Allgemein verwendete Literatur: [Baeh91, Baer80, Debl90, East90, Ever90, Fluc89, Goor89, Grant89, Grant91, Hama90, Haye88, Henn90, Hsie89, Huck89, Knie89, Koho80, Lang89, Marw93, Muld91, Ober89, Rhei92, Scho88, Siew82, Ston90, Swaa92, Tane84, Tava90, Unge89, Weck82, Weik93a/b].

3.1 Allgemeine Puffer

Zusätzliche Literatur: [Perle93].

Ein Puffer ist ein einfacher, sehr kleiner Zwischenspeicher, der an verschiedenen Stellen in der Speicherhierarchie benutzt wird. Eine besondere Verwendung findet der Puffer als Instruktionspuffer (s.u.). Ein Puffer ist durch folgende numerischen (Größe) und symbolischen (Zugriffsart) Parameter beschrieben:

3.1.1 Größe

Ein Puffer ist in der Regel sehr klein und enthält nur wenige Daten der jeweils verwendeten Datengröße (Instruktion, Adresse, usw.). Hier wird die Anzahl dieser Daten angegeben.

- Wertebereich: {1-8}
- Bemerkungen
  - Je nach Verwendung des Puffers können hier auch andere Wertebereiche relevant sein.

3.1.2 Zugriffsart

Es gibt für einen Puffer verschiedene mögliche Art des Zugriffs auf die eingelagerten Daten.

- ohne Zugriffseinschränkung:
  Es kann auf jedes Datum im Puffer zugegriffen werden.
- Stapel:
  Es kann nur auf das zuletzt eingelagerte Datum zugegriffen werden (LIFO-Prinzip).
- Schlange:
  Es kann nur auf das als erstes eingelagerte Datum zugegriffen werden, dieses wird dann aus der Schlange entfernt (FIFO-Prinzip).

- Wertebereich: {ohne, Stapel, Schlange}
- Bemerkungen:

- In der Regel wird auf dieser Ebene der Puffer ohne Zugriffseinschränkung verwendet, außer z.B. beim Instruktionspuffer (s.u.).

3.2 Instruktionspuffer

Zusätzliche Literatur: [Kael92, Kuri91].

Bei einem gemischten Cache (s.u.) kann zusätzlich ein Puffer für Instruktionen eingeführt werden, um das Holen der Instruktionen zu beschleunigen. Ein Instruktionspuffer ist durch folgende numerischen (Größe) und symbolischen (Zugriffsart) Parameter beschrieben

3.2.1 Größe

Hier wird die Größe des Instruktionspuffers in “Anzahl der Instruktionen” festgelegt.

- Wertebereich: {1 - 16}
- Bemerkungen:
  - Ein Instruktionspuffer ist relativ klein. Ist ein große Puffer nötig, würde sich eher ein Cache für Instruktionen lohnen.

3.2.2 Zugriffsart

Es gibt verschiedene Arten, auf einen Instruktionspuffer zuzugreifen:

- Prefetch-Schlange:
  Instruktionen werden in eine Schlange geladen, sobald der Bus frei ist. Führt die CPU eine Instruktion aus, wird sie (nach dem FIFO-Prinzip) aus dem Puffer entfernt, d.h. die CPU konsumiert sie. Dieser Puffer nimmt also keine Rücksicht auf die Lokalität von Programmen.
- Instruktionspuffer:
  Klassische Instruktionspuffer enthalten typischerweise ein zusammenhängendes Segment eines Programms. Sie sind als Schlangen oder Stapel organisiert. Instruktionen werden der Reihe nach geladen. Ist bei einer Schleifenverzweigung die Zieladresse nicht im Puffer, wird dieser vollständig gelöscht und mit Instruktionen, die dem Sprungziel folgen, gefüllt. Auch Schlangen-basierte Caches werden hierzu gezählt. Der Puffer sollte so groß gewählt werden, daß lokale Zugriffe enthalten sein können.

- Wertebereich: {kein Puffer, Prefetch-Schlange, Instruktionspuffer}
- Bemerkungen:
  - Prefetch-Schlangen unterstützen nicht das lokale Verhalten von Programmen, z.B. Instruktionen in Schleifen müssen immer wieder neu geholt werden.
  - Mehrere Schlangen-basierte Instruktionspuffer können eine geteilte Lokalität unterstützen und entsprechend ähnlich gute Leistungen zeigen, wie ein Mehrwege-Satz-assoziativer Instruktionscache (s.u.), sie benötigen aber weniger Verwaltungsaufwand und entsprechend weniger Fläche.
  - Für Verzweigungen gibt es Branch-Target-Buffer, die die voraussichtliche Zieladresse und die Wahrscheinlichkeit, mit der diese Zieladresse bei der Verzweigung gewählt wird, speichern.

3.3 Cache

Zusätzliche Literatur: [Ara89, Alpe88, Bars92, Devi91, Ewy93, Goks89, Kola92, Przy88, Sawa89, Smit82, Smit85].

Ein Cache ist ein kleiner Speicher, der logisch gesehen zwischen der CPU und dem Hauptspeicher angesiedelt ist und Daten des Hauptspeichers (s.u.) zwischenspeichert, d.h. alle Daten aus dem Cache sind auch im Hauptspeicher vorhanden. Es kann auch am Cache vorbei direkt auf den Hauptspeicher

zugegriffen werden, d.h. man kann am Cache “vorbei gucken” (lookaside buffer).

Der Cache besteht aus verschiedenen Komponenten (Abb. 3):

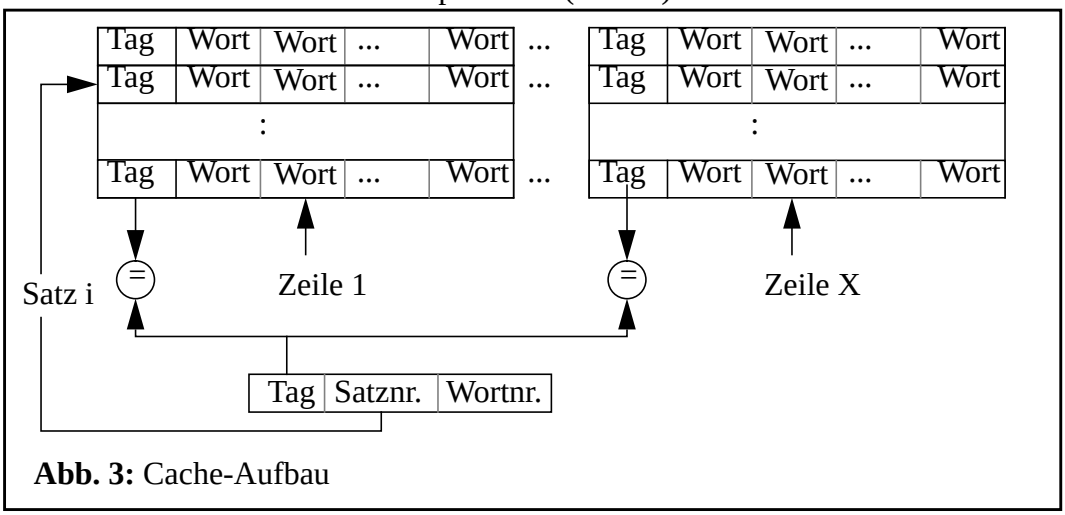

Abb. 3: Cache-Aufbau

- Wort (word): die kleinste adressierbare Einheit, meist 1, 2 oder 4 Byte.
- Zeile (line) oder Block: die Speichereinheit, in der Daten vom Hauptspeicher in den Cache transportiert werden, ein ganzzahliges Vielfaches (meist eine 2-er Potenz) eines Wortes. Speziell beim Cache heißt ein Block auch Zeile.
- Satz (set): der Bereich eines Caches, der voll-assoziativ (s.u.) nach einem Datum durchsucht wird, ein ganzzahliges Vielfaches (meist eine 2-er Potenz) einer Zeile. D.h. die Anzahl der benötigten Adreßvergleicher ist gleich der Anzahl der Zeilen in den Sätzen. Ist diese gleich S, spricht man auch von einem S-Wege-Satz-assoziativen Cache. Beim direct-mapped Cache enthält jeder Satz genau eine Zeile, der voll-assoziative Cache besteht aus genau einem Satz.
- Tag: Bits, die den Teil einer Adresse sowie einige Bits für die Verwaltung z.B. von Zugriffsrechten enthalten: Da der Hauptspeicher größer ist als der Cache, werden bestimmte Daten aus dem Hauptspeicher an der gleichen Cache-Position abgelegt. Um diese Daten noch eindeutig identifizieren zu können, werden diejenigen Adreßbits, die benötigt werden, um das Datum im Hauptspeicher zu adressieren abzüglich der Bits für die Cache-Adresse im zugehörigen Tag gespeichert. Dabei gilt: |Tag|=log(Adreßraum*Assoziativität/Cachegröße).

Ein Cache beschleunigt den Zugriff auf Daten, besonders, wenn die Anwenderprogramme eine gewisse Lokalität aufweisen, was normalerweise der Fall ist. Dabei entsteht allerdings ein Verwaltungsmehraufwand für das Ein- und Auslagern sowie für das Erhalten der Konsistenz der Daten, besonders bei mehreren Prozessoren.

Die verschiedenen Cache-Parameter (Entwurfsentscheidungen) und ihre Beziehungen werden im folgenden näher beschrieben. Dabei beziehen sich die ersten Parameter (Vorhandensein eines Caches, Anzahl der Level, Trennung in Daten- und Instruktionscache und Vorhandensein eines Instruktionspuffers) sowie das Vorhandensein eines Schreibpuffers auf die sogenannten Strukturentscheidung. Sie beeinflussen die Anzahl der Komponenten im generischen Speichermodell und somit die Anzahl der einzustellenden Parameter. Die restlichen Parameter fallen in den Bereich der Dimensionierung. Sie lassen sich weiter aufteilen in numerische (Größe, Zeilengröße, Anzahl der Subzeilen und Assoziativität) und symbolische Entwurfsentscheidungen. Letztere wählen aus einer fest vorgegebenen Menge von Verfahren, Strategien oder Methoden eine aus. Tabelle 1 gibt zu jeder Entwurfsentscheidung den

Parameter	Struktur	symbolisch	numerisch
Vorhandensein eines Caches	+		
Anzahl der Level	+		

Tab. 1: Typ der Entwurfsentscheidungen der Cache-Parameter

Parameter	Struktur	symbolisch	numerisch
Trennung	+		
Vorhandensein eines Instruktionspuffers	+		
Levelnummer (ergibt sich automatisch)			+
Integration		+	
Inhalt (ergibt sich automatisch)		+	
Adressierung		+	
Größe			+
Zeilengröße			+
Anzahl der Subzeilen			+
Assoziativität			+
Ersetzungsstrategie		+	
Ladestrategie		+	
Schreiben - Aktualisieren des Hauptspeichers - Schreibstrategie - Vorhandensein eines Schreibpuffers	 +	 + + 	
Start		+	

Tab. 1: Typ der Entwurfsentscheidungen der Cache-Parameter

Typ an. Nähere Erläuterungen folgen im Anschluß.

3.3.1 Vorhandensein eines Caches (Strukturierung)

Ein Prozessor-System besitzt nicht notwendigerweise einen Cache.

- Wertebereich: {ja, nein}
- Bemerkungen:
  - Es müssen die Vorteile (z.B. schneller Zugriff) und die Nachteile (z.B. zusätzlicher Verwaltungsaufwand für das Ein- und Auslagern von Daten) abgewägt werden.

3.3.2 Anzahl der Cache-Level (Strukturierung)

Die Anzahl der Cache-Level gibt an, wieviel Ebenen von Cache vorhanden sind.

- Wertebereich: {1, 2}
- Bemerkungen:
  - Gibt es zwei Cache-Level, befindet sich der 1.-Level-Cache in der Regel mit auf dem Chip des Prozessors (on-chip), während sich der 2.-Level-Cache auf einem anderen Chip befindet (off-chip). Dies bedingt einen größeren Verwaltungsaufwand, weil Daten in beide Caches ein- und ausgelagert werden können.
  - Die weiteren Parameter müssen für jeden dieser Caches einzeln bestimmt werden.

3.3.3 Trennung des Caches in Daten- und Instruktionscache (Strukturierung)

Ein Cache kann in zwei getrennte Caches geteilt werden. Dann werden in dem einen Cache die Instruktionen für die Anwenderprogramme zwischengespeichert (Instruktionscache), während der andere Cache die Operanden zu den Instruktionen enthält (Daten-Cache). Wird der Cache nicht getrennt, gibt es nur einen gemischten (unified) Cache.

- Wertebereiche: {ja, nein}
- Bemerkungen:
  - Die Trennung kommt nur für den 1.-Level-Cache in Frage, für eine 2.-Level-Cache lohnt der

Aufwand nicht, da auf jeden Fall eine Zeitverzögerung für einen off-chip-Zugriff entsteht.
- Bei einer Trennung ist mehr Hardware-Aufwand nötig, weil die Zugriffslogik zweimal vorhanden sein muß.
- Ein einzelner erfolgreicher Zugriff dauert bei getrennten Caches länger als bei einem gemischtem Cache, weil zuerst entschieden werden muß, auf welchen Cache zuzugreifen ist.
- Der Datenfluß (Datenmenge pro Zeit) zur CPU wird durch die Trennung erhöht, falls es zu jedem dieser beiden Caches eine eigene Verbindung (Bus) gibt.
- Durch die Trennung wird das Pipelining unterstützt, weil Instruktionen und Daten parallel geholt werden können, d.h. der durchschnittliche Zugriff wird beschleunigt.
- Durch die Trennung kann jeder Cache getrennt optimiert werden.
- Durch die Trennung wird jeder Cache kleiner, und somit besser auf dem Chip an die entsprechende Stelle plazierbar.
- Durch die Trennung kann es ein Konsistenzproblem geben, auch bei nicht selbst modifizierenden Programmen. Es können z.B. die Organisationsbits wie Zugriffsrechte in einer Zeile, die sowohl Daten als auch Instruktionen enthält, in den getrennten Caches unterschiedlich belegt sein.

3.3.4 Vorhandensein eines Instruktionspuffers (Strukturierung)

Bei einem gemischten Cache kann man zusätzlich Puffer für Instruktionen einführen, um das Holen der Instruktionen zu beschleunigen.

- Wertebereich: {ja, nein}
- Bemerkungen:
  - Ein Instruktionspuffer unterstützt das lokale Zugriffsverhalten von Programmen, sofern sich Zugriffe auf einen örtlich zusammenhängenden Bereich im Adreßraum konzentrieren. Ansonsten muß der Puffer immer wieder neu geladen werden.
  - Ist die örtliche Lokalität auf wenige Bereiche verteilt, können auch mehrere Puffer vorgesehen werden.

3.3.5 Levelnummer

Die Levelnummer gibt zu einem Cache an, ob es sich um einen 1.- oder 2.-Level-Cache handelt, siehe auch Anzahl der Level der Cache-Hierarchie.

- Wertebereich: {1, 2}

3.3.6 Integration

Ein Cache kann mit auf dem Prozessor-Chip integriert sein, dann nennt man das on-chip, ansonsten ist der Cache off-chip angeordnet.

- Wertebereich: {on-chip, off-chip}
- Bemerkungen:
  - Ein einzelner Cache auf den Chips des Hauptspeichers bringt nicht genügend Geschwindigkeitssteigerung, er sollte sich auf dem Chip des Prozessors befinden.
  - Der 2.-Level-Cache ist immer off-chip, der 1.-Level-Cache bringt on-chip die größte Geschwindigkeitssteigerung.

3.3.7 Inhalt

Der Inhalt gibt an, ob es sich um einen gemischten (unified), Daten- oder Instruktionscache handelt, siehe auch Trennung des Caches in Daten- und Instruktionscache.

- Wertebereich: {gemischt, Daten, Instruktionen}

3.3.8 Adressierung

Ein Cache kann mit der realen Adresse, unter der die Daten im Hauptspeicher abgelegt sind, oder mit der virtuellen Adresse, wie sie von den Prozessen benutzt wird, adressiert werden.

- Wertebereich: {virtuell, real}
- Bemerkungen:
  - Befindet sich das gesuchte Datum im Cache, ist ein Zugriff bei virtueller Adressierung schneller, weil die Adresse nicht übersetzt werden muß.
    Der Zugriff auf reale Caches ist entsprechend langsamer, weil die virtuellen Adressen erst übersetzt werden müssen. Dies kann beschleunigt werden, indem die Adreßübersetzung und die Satzauswahl bei einem assoziativen Cache (s.u.) parallelisiert werden, falls sich die beiden Adreßbereiche nicht überlappen.
  - Ein real adressierter Cache bleibt bei Prozeßwechsel gültig. Ein virtueller Cache wird bei Prozeßwechsel ungültig, da verschiedenen Prozesse dieselbe virtuelle Adresse für verschiedene reale Adressen verwenden können. Um dies zu verhindern, muß zu jeder Adresse der zugehörige Prozeßname mit abgespeichert.
  - Das Problem der Synonyme entsteht bei virtueller Adressierung eines Caches, wenn mehrere Prozesse über verschiedene virtuelle Adressen auf die gleichen realen Adressen zugreifen. Dies tritt z.B. auf, wenn mehrere Prozesse den gleichen Code benutzen (code sharing). Synonyme müssen entdeckt und eliminiert werden, indem die virtuellen Adressen mit Hilfe von TLBs in reale Adressen übersetzt und mit Hilfe von weiteren Tabellen in alle möglichen virtuellen Adressen rückübersetzt werden.
  - Externe Ein- und Ausgaben sind bei virtueller Adressierung des Caches langsamer, falls sich das aktuelle Datum nur im Cache und nicht im Hauptspeicher befindet, weil erst die externen realen Adressen in virtuelle übersetzt werden müssen, um das Datum im Cache zu finden.

3.3.9 Größe (Kapazität)

Unter Cache-Größe versteht man die Speicherkapazität eines Caches ohne die Tag-Bits.

- Wertebereich: { 1 - 256 KByte für einen 1.-Level-Cache,
  256 - 4096 KByte für einen 2.-Level-Cache}
- Bemerkungen:
  - Die Cache-Größe ist in der Regel eine 2-er Potenz.
  - Die Größe des on-chip-Caches wird durch den noch vorhandenen Platz auf dem Chip beschränkt, hierbei müssen auch die benötigten Tag-Bits für Adresse und Verwaltung berücksichtigt werden.
  - Ein Daten-Cache ist üblicherweise größer als ein Instruktionscache, weil Instruktionszugriffe i. allg. ein sequentielleres Zugriffsverhalten zeigen als Datenzugriffe (aufeinanderfolgende Instruktionen sind an aufeinanderfolgenden Adressen gespeichert, außer bei Sprüngen). Damit lassen sich die in naher Zukunft benötigten Instruktionen besser abschätzen.
  - Der 1.-Level-Cache sollte die Arbeitsmenge (s.u.) eines üblichen Anwendungsprogramms über eine gewisse Zeitspanne aufnehmen können. Ansonsten ergibt sich eine zu große Fehlschlagsrate.
  - Ein 2.-Level-Cache sollte mindestens doppelt so groß sein wie ein 1.-Level-Cache, damit sich sein Einsatz lohnt: Auf den 2.-Level-Cache wird nur zugegriffen, falls ein Datum nicht im 1.-Level-Cache gefunden wird. Gilt das Prinzip, daß der 2.-Level-Cache eine Obermenge an Daten des 1.-Level-Caches enthält, was aus vielerlei Gründen sinnvoll ist, und ist der 2.-Level-Cache nur wenig größer als der 1.-Level-Cache, dann hat der 2.-Level-Cache eine sehr hohe Fehlschlagsrate, weil nur selten auf den 2.-Level-Cache erfolgreich zugegriffen wird (Sind sie gleich groß, beträgt die Fehlschlagsrate des 2.-Level-Caches 100%). Zudem ist die Zugriffszeit hier

nicht so kritisch, weil die Zeit für einen erfolgreichen Zugriff durch den 1.-Level-Cache bestimmt wird.
- Je größer der Cache, desto langsamer ist aus technologischen Gründen ein Zugriff. Das ist insbesondere beim 1.-Level-Cache kritisch.
- Je größer der Cache, desto weniger Fehlschläge gibt es aufgrund von Kollisionen (collision miss) oder von zu geringer Kapazität (capacity miss).

3.3.10 Zeilengröße

Die Zeilen eines Caches sind Speicherbereiche konstanter Größe, die mit Hilfe eines Tags gemeinsam verwaltet wird. Die Größe einer Zeile bezieht sich auf die Speicherkapazität ohne Tag.

- Wertebereich: { 1 - 256 Worte für einen gemischten 1.-Level-Cache,
  16 - 64 Worte für einen Daten-Cache,
  8 - 16 Worte für einen Instruktionscache,
  > 256 Byte für einen 2.-Level-Cache}
- Bemerkungen:
  - Die Zeilengröße in Byte ist in der Regel eine 2-er Potenz.
  - Die Zeilengröße ist ein ganzzahliges Vielfaches eines Wortes. Ein Speicherwort bezeichnet die kleinste vom Prozeß adressierbare Speichereinheit.
  - Die Zeilengröße entspricht in der Regel der Speichereinheit, in der Daten zwischen dem Cache und dem Hauptspeicher übertragen werden. Allerdings können auch halbe oder viertel Zeilen übertragen werden (subblock placement).
  - Die Zeilengröße sollte ein ganzzahliges Vielfaches der Verbindungsbreite zwischen CPU und Cache sein.
  - Die Verbindungsbreite zwischen Cache und Hauptspeicher sollte ein ganzzahliges Vielfaches von der Zeilengröße sein.
  - Bei großen Zeilen werden mehrere hintereinanderliegende Daten schnell eingelagert. Das ist bei sequentiellem Programmverhalten von Vorteil. Jedoch erhöht sich bei einem Fehlschlag die Zeit für das Einlagern einer einzelnen Zeile aus dem Hauptspeicher (miss penalty).
  - Bei großen Zeilen kann es zu einer Speicherverschmutzung (pollution) kommen, wenn viele der Daten einer eingelagerten Zeile nicht gebraucht und statt dessen benötigte Daten dadurch verdrängt werden.
  - Die Fehlschlagsrate bedingt durch Kollisionen (conflict miss ratio) steigt bei großen Zeilen, weil weniger Zeilen zur Verfügung stehen und stattdessen mehr verschiedene Daten aus dem Hauptspeicher auf die gleiche Zeile im Cache abgebildet werden, dies macht sich besonders bei kleinen Caches bemerkbar.
  - Bei großen Zeilen sinkt die Fehlschlagsrate bedingt durch Prozeßwechsel und erstmaligem Zugriff (compulsory miss ratio), weil bei jedem Fehlschlag viele Daten auf einmal eingelagert werden.
  - Bei großen Zeilen ist eine geringere Verwaltung nötig, weil es insgesamt weniger Tags gibt und weniger Zeilen für assoziative Suche, Ersetzungsstrategie, Konsistenz und Sicherung verwaltet werden müssen.
  - Bei großen Zeilen kommt es seltener vor, daß sich ein vom Prozeß angefordertes Datum in zwei verschiedenen Zeilen befindet (line crosser).
  - Benutzen mehrere Prozesse Daten gemeinsam (sharing), dann führen große Zeilen vermehrt dazu, daß die Prozesse auf verschiedene Daten in der gleichen Zeile zugreifen, es kommt also zu einem Schein-Sharing (false sharing). Dies führt bei Mehrprozessorbetrieb zu unnötigem Zeitverlust und Umlagern von Daten.

3.3.11 Subzeilen

Eine Zeile kann in ein bis vier Subzeilen eingeteilt sein, die getrennt voneinander ein- und ausgelagert, aktualisiert oder ungültiggeschrieben werden können. Wird eine Zeile nicht in Subzeilen unterteilt, wird dies durch die Anzahl der Subzeilen pro Zeile von eins modellliert. Jede Subzeile hat ihre eigenen Gültigkeits- und Organisationsbits, allerdings gibt es nur einen Adreßtag. Dies gibt etwas mehr Flexibilität und spart Speicher für die Adressen.

- Wertebereich: {1, 2, 4}
- Bemerkungen:
  - Subzeilen einer Zeile liegen im Adreßraum hintereinander. Mehr als vier Subzeilen bringen keinen Gewinn mehr, es ist dann flexibler kleinere Zeilen zu wählen, die jeweils in verschiedenen Bereichen des Adreßraums liegen dürfen.
  - Mit Subzeilen sind größere Zeilen möglich, da die meisten Aktionen nicht auf einer gesamten Zeile ausgeführt werden müssen.
  - Eine Verzögerung durch Fehlschlag kann mit Subzeilen verkürzt werden, da ein Prozessor nur warten muß, bis der entsprechende Teil der Zeile eingelagert ist.
  - Da auch Teile einer Zeile ungültig geschrieben werden können, gibt es bei Multiprozessor-Systemen weniger “Streit” um eine Zeile (false sharing).
  - Mit Subzeilen ist das Durchschreiben (s.u.) eher möglich, weil nicht die ganze Zeile im Hauptspeicher aktualisiert werden muß.
  - Mit Subzeilen ist eher Prefetching möglich, weil auch Teile einer Zeile vorab eingelagert werden können.

3.3.12 Assoziativität

Die Assoziativität eines Caches bezeichnet die Anzahl der Zeilen eines Satz (set), deren Adressen jeweils gleichzeitig mit der eingegebenen Adresse des gewünschten Datums verglichen werden. Der Cache wird also in Sätze aufgeteilt, innerhalb derer eine Zeile an einer beliebigen Zeilenpositionen abgespeichert werden kann. Wird die Adresse eines Datums eingegeben, werden die entsprechenden Bits dieser Adresse dazu benutzt, den richtigen Satz im Cache auszuwählen. Die restlichen Bits dieser Adresse werden mit den Adreßtags jeder Zeile dieses Satzes gleichzeitig verglichen, d.h. es sind entsprechend viele Vergleicher nötig. Die beiden Extreme sind der direct-mapped (jeder Satz enthält genau eine Zeile, diese wird also direkt adressiert) und der voll-assoziative Cache (es gibt nur einen Satz, und die Adressen jeder Zeile werden gleichzeitig verglichen) (Abb. 4a-c).

- Wertebereich: {1 - 16, voll}
- Bemerkungen:
  - Die Assoziativität ist meistens eine 2-er Potenz.
  - Je höher die Assoziativität, desto
    ... seltener treten Kollisionen auf (conflict miss ratio).
    ... mehr Hardware ist nötig, um die entsprechenden Vergleicher für die Adreßbits zu realisieren.
    ... länger sind die Tags, und der dafür benötigte Speicher, weil innerhalb eines Satzes die Position einer Zeile nicht mehr mitbestimmend für die Adresse ist.
    ... langsamer ist aufgrund der Vergleicher und technologischer Gründe ein Zugriff für einen Treffer (hit time).
  - 2:1-Regel: Viele Simulationen haben gezeigt, daß ein direct-mapped-Cache von der Größe 2*X dieselbe Fehlschlagsrate hat wie ein zwei-Wege-Satz-assoziativer Cache der Größe X.

3.3.13 Ersetzungsstrategie (replacement)

Mit Hilfe der Ersetzungsstrategie wird entschieden, welche Zeile im Cache ausgelagert wird, um einer neuen einzulagernden Zeile Platz zu machen. Ersetzungsstrategien arbeiten satzweise, d.h. das auszu-

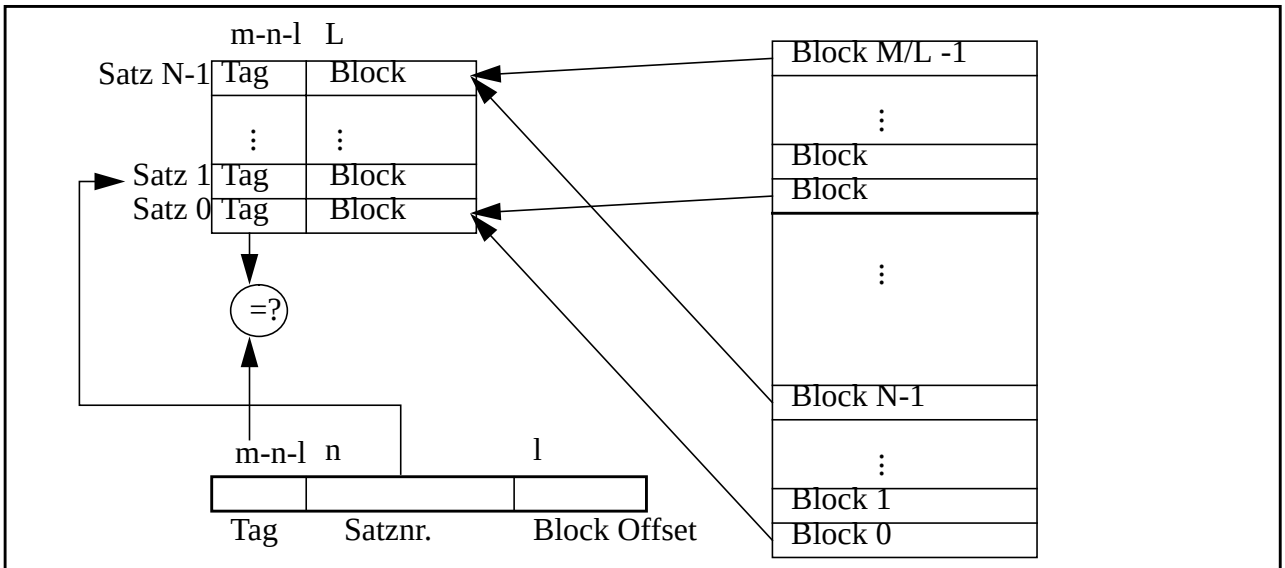

Abb. 4a: direct-mapped-Cache

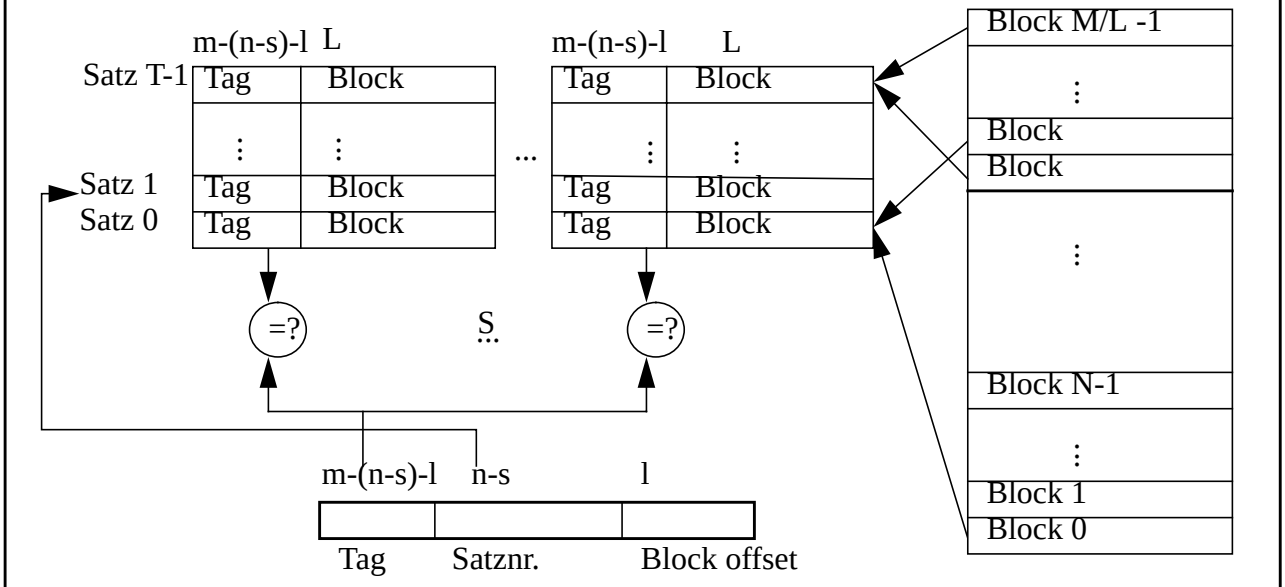

Abb. 4b: S-Wege-Satz-assoziativer Cache

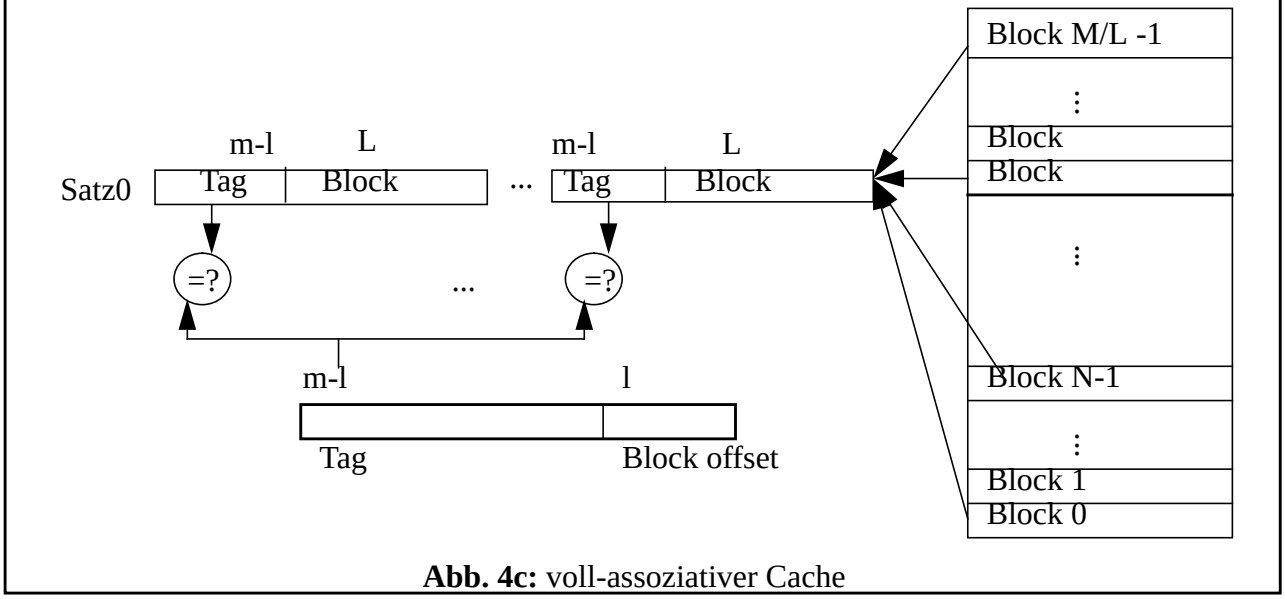

Abb. 4c: voll-assoziativer Cache

lagernde Datum wird nur aus dem Satz bestimmt, in den die neue Zeile eingelagert werden soll. Dieser kann direkt aus der Adresse abgelesen werden. Hier seinen nur die für einen Cache relevanten Ersetzungsstrategien beschrieben. Diese und weitere sind beim Hauptspeicher näher erläutert, wo sie dazu benutzt werden zu entscheiden, welche Seite aus dem Hauptspeicher auszulagern ist.

- Random:
  Die auszulagernde Zeile wird zufällig ausgewählt Dies ist einfach zu realisieren, die auszulagernde Zeile ist schnell zu bestimmen, aber die Fehlschlagsrate ist nur bei Programmen mit irregulärem Zugriffsverhalten gering.
- FIFO (first-in-first-out):
  Die zuerst eingelagerte Zeile als erstes wieder ausgelagert. Dies ist ebenfalls einfach zu realisieren, allerdings führt es nur bei sequentiellem Zugriffsverhalten zu einer geringen Fehlschlagsrate
- LRU (least recently used):
  Die am längsten nicht benutzte Zeile wird als erstes ausgelagert. Dies Verfahren kann mit Hilfe von Dreiecksmatrizen oder Listen realisiert werden. Es ist mit einigem Aufwand verbunden, führt aber zu einer geringen Fehlschlagsrate.

• Wertebereich: {Random, FIFO, LRU}

• Bemerkungen:
  - Für einen direct-mapped-Cache wird keine Ersetzungsstrategie gebraucht, weil jeder Satz nur eine Zeile enthält.
  - Für einen 1.-Level-Cache kommen wegen der Geschwindigkeitsanforderung nur in Hardware implementierte Strategien in Frage, die schnell sind, also Random, FIFO oder LRU. Andere Strategien sind beim Hauptspeicher beschrieben.
  - Die Ersetzungsstrategie beim Cache hat großen Einfluß auf die Fehlschlagsrate, besonders bei kleinen Caches (also da am besten LRU).

3.3.14 Ladestrategie (prefetching)

Die Ladestrategie entscheidet, ob und wann ein Datum in den Cache geholt wird, dabei wird gegebenenfalls zusätzlich das Datum geholt, das dem zugegriffenen Datum folgt, also an der nächsten Adresse liegt (1 block lookahead). Andere Verfahren zur Auswahl des zusätzlich zu holenden Datums sind hier zu langsam. Folgende Strategien stehen zur Auswahl:

- zugriffsgesteuert (always prefetch):
  Bei jedem Zugriff wird die jeweils nächste Zeile in den Cache geholt. Dies Verfahren ist einfach und führt zu einer kleinen Fehlschlagsrate, allerdings ist die Zugriffsrate auf den Hauptspeicher und die Transferrate sehr hoch.
- tag-gesteuert (tagged prefetch):
  Jede Zeile hat einen Tag, der beim Ein- und Auslagern auf 0 und beim Zugriff auf 1 gesetzt wird. Beim Zugriff auf eine 1-Zeile (also mindestens der 2. Zugriff) wird die nächste Zeile in den Cache geholt. Der Aufwand ist hier höher und auch die Fehlschlagsrate, aber die Zugriffsrate auf den Hauptspeicher und die Transferrate ist geringer.
- fehlschlagsgesteuert (fetch on miss):
  Bei jedem Fehlschlag wird diese und die jeweils nächste Zeile in den Cache geholt. Die Fehlschlagsrate ist hier meist zu hoch, aber die Zugriffsrate auf den Hauptspeicher und die Transferrate noch geringer.
- anfragegesteuert (demand fetch, ohne Prefetching):
  Nur bei einem Fehlschlag wird die entsprechende Zeile in den Cache geholt. Dies Verfahren ist sehr einfach, hat aber die größte Fehlschlagsrate und kleinste Zugriffsrate auf den Hauptspeicher und die Transferrate.

- Wertebereich: {zugriffsgesteuert, tag-gesteuert, fehlschlagsgesteuert, anfragegesteuert}
- Bemerkungen:
  - Ziel ist es, eine möglichst kleine Fehlschlagsrate zu erreichen, ohne den gegenwärtigen Prozeß durch Prefetching zu behindern, wobei gilt: Je mehr Prefetching bis zu einer bestimmten Grenze desto kleiner ist die Fehlschlagsrate (darüber hinaus kommt es zu einer Speicherverschmutzung) und desto größer ist die Zugriffsrate auf den Hauptspeicher und die Transferrate.
  - Ein Prefetching der jeweils nächsten Zeile mit einer Zeilengröße L ist besser als eine Zeilengröße von 2*L ohne Prefetching, weil bei der kleineren Zeilengröße nach dem Transfer von L Bytes die angeforderten Daten bereits benutzt werden können.

3.3.15 Schreiben

Hier geht es darum zu entscheiden, wie der Hauptspeicher beim Schreiben eines Datums aktualisiert wird, falls ein Cache vorhanden ist. Eventuelle Konsistenzmechanismen für Multiprozessorsysteme werden in dem entsprechenden Kapitel beschrieben.

a) Aktualisieren des Hauptspeichers (updating)
Wird auf ein Datum im Cache schreibend zugegriffen, muß entschieden werden, wie der Hauptspeicher aktualisiert wird:

- Rückschreiben (copy back):
  Das Datum wird zunächst nur im Cache geändert und in regelmäßigen Abständen wird der Hauptspeicher aktualisiert.
- Durchschreiben (write through):
  Das Datum wird im Cache geändert und gleichzeitig bei jedem Schreiben wird der Hauptspeicher aktualisiert

- Wertebereich: {Rückschreiben, Durchschreiben}
- Bemerkungen:
  - Rückschreiben eignet sich gut, wenn mehrmals geschrieben wird, bevor andere Prozesse das Datum im Hauptspeicher lesen wollen.
  - Rückschreiben benutzt in der Regel fetch-on-write (s.u.), weil nicht bei jedem Schreiben auf den Hauptspeicher zugegriffen werden soll.
  - Rückschreiben macht eine kompliziertere Zugriffslogik für den Cache notwendig (invalidate/ dirty bit), um sicherzustellen, daß immer auf die aktuelle Version zugegriffen wird.
  - Rückschreiben wird meist bei Multiprozessor-Systemen benutzt, um den Datenverkehr zum Hauptspeicher gering zu halten, damit die Verbindung zum Hauptspeicher, die von mehreren Prozessoren benutzt wird, nicht zu überlasten.
  - Durchschreiben eignet sich nur bei seltenem Schreiben, weil sonst der Datenverkehr zu Hauptspeicher zu groß wird
  - Durchschreiben vereinfacht das Konsistenzproblem, weil der Hauptspeicher immer die aktuelle Version der Daten enthält, dies gilt insbesondere für DMA (direct memory access), Multiprozessoren und nach einem Systemabsturz.
  - Durchschreiben benutzt in der Regel die write-around Technik, da sowieso jeder Schreibzugriff ein Fehlschlag ist, es sein denn, es wird auf geschriebenen Daten mehrmals lesend zugegriffen. Hierfür ist ein entsprechend großer Schreibpuffer notwendig.

b) Schreibstrategie
Beim Schreiben eines Datums gibt es zwei Möglichkeiten:

- fetch-on-write:
  Soll ein Datum geschrieben werden, wird es zuerst in den Cache geholt und dort geändert.
- write-around:

Das Datum wird direkt im Hauptspeicher geändert.

- Wertebereiche: {fetch-on-write, write-around}
  - Ein einzelner Schreibzugriff mit write-around ist schneller als mit fetch-on-write, weil das Datum nicht erst in den Cache geholt werden muß. Wird mehrfach (mehr als drei mal) auf das gleiche Datum im Cache zugegriffen, ohne das es zwischenzeitlich ausgelagert wird (also fetch-on-write), ist die durchschnittliche Dauer für einen Schreibzugriff schneller als mit write-around, da das Datum nach dem ersten Zugriff im Cache zur Verfügung steht.

c) Vorhandensein eines Schreibpuffer (Struktur)
Um ein Datum in den Hauptspeicher zurückzuschreiben, kann es in einem Puffer zwischengespeichert und von da aus sobald wie möglich in den Hauptspeicher geschrieben werden (Puffer-Parameter s.o.).

- Wertebereich: {ja, nein}
- Bemerkungen:
  - Ein Schreibzugriff kann durcheinen Schreibpuffer beschleunigt werden, da der Prozessor nur warten muß, bis das Datum in den Puffer geschrieben wurde. Das Schreiben vom Puffer in den Hauptspeicher kann im Hintergrund geschehen.
  - Ein Schreibpuffer verkompliziert Lesezugriffe, weil auf der Suche nach dem aktuellen Daten auch der Puffer durchsucht werden muß.

3.3.16 Start bei Prozeßwechsel

Hier geht es darum, was mit dem Cache passiert, wenn ein Prozeßwechsel stattfindet:

- Kaltstart (cold start):
  Der Cache wird bei Prozeßwechsel gelöscht, so daß bei erstmaligen Beschreiben des Caches mit den Daten des neuen Prozesses keine Ersetzungsstrategie angewendet werden muß.
- Warmstart (warm start):
  Der Cache-Inhalt bleibt erhalten, d.h. aber auch, daß diese Daten bei einem virtuell adressierten Cache nur dann gültig bleiben, wenn der Prozeßname mit gespeichert wird.

- Wertebereich: {kalt, warm}
- Bemerkungen:
  - Ein Kaltstart eignet sich bei seltenen Prozeßwechseln und wenn der neue Prozeß nahezu den gesamten Cache benötigt.
  - Ein Warmstart eignet sich bei häufigen Prozeßwechseln und wenn der neue Prozeß noch alte Daten im Cache unberührt läßt. Dies ist vor allem bei einem großem Cache der Fall, weil dann ein reaktivierter Prozeß u.U. noch viele seiner alten Daten im Cache vorfindet, d.h. er muß diese Daten nicht erst erneut in den Cache holen. (working set restoration). Damit ergeben sich weniger Kaltstart-Fehlschläge (compulsory misses).

3.3.17 Sonstiges

In der Literatur sind noch weitere exotische Cache-Varianten beschrieben, die jedoch eher für wenige Spezialfälle in Frage kommen. Sie lassen sich nur schwer in einem automatischen Entwurfswerkzeug verwenden.

- halb gemischter Cache (semi-unified cache)
  Der halb gemischte Cache [Drac93] besteht aus einem Daten- und einem Instruktionscache, je mit einer Assoziativität von eins. Der Daten- bzw. Instruktions-Cache dient gleichzeitig als 2.-Level-Cache für Instruktionen bzw. Daten. Bei einem Fehlschlag in einem 1.-Level-Cache wird in dem entsprechenden 2.-Level-Cache gesucht. Ist das Datum dort vorhanden, werden die Zeilen des 1.-

und 2.-Level-Cache gegeneinander ausgetauscht. Ist das Datum dort nicht vorhanden, muß eine neue Zeile aus einem off-chip-Cache oder aus dem Hauptspeicher eingelagert werden. Die auszulagernde Zeile kann zwischen den beiden Caches mit einer Austauschstrategie (Random oder LRU) gewählt werden, wie bei einem 2-fach-Satz-assoziativen Cache. Als Vorteile für den halb gemischten Cache sind angegeben: a) Die globale Fehlschlagsrate für beide Caches mit je einer Größe von X entspricht der eines 2-fach-Satz-assoziativen Caches der Größe 2*X. b) Die Zugriffszeit auf einen dieser Caches ist gleich der eines direct-mapped Caches. c) Es kann auf Daten und Instruktionen parallel zugegriffen werden, wie bei getrennten Caches. d) Der Platzbedarf für Daten und Instruktionen kann dynamisch gegeneinander abgewogen werden, wie bei einem gemischten Cache. e) Die off-chip-Fehlschlagsrate ist deutlich reduziert, gegenüber einem direct-mapped-Cache. Die Nachteile sind eine verzögerte Zugriffszeit für den Zugriff auf einen 2.-Level-Cache, was auch den parallelen Zugriff von Daten und Instruktionen einschränkt, und der zusätzliche Hardware-Aufwand für eine Verbindung zwischen den beiden Caches.

- victim-Cache
  Der victim-Cache [Joup90] ist ein voll-assoziativer Cache mit ein bis fünf Zeilen, der zusätzlich zum “normalen” direct-mapped-Cache angelegt wird. Wird ein Datum aus dem “normalen” Cache ausgelagert, wird es zunächst in den victime-Cache geschrieben. Ein direct-mapped-Cache zusammen mit einem victim-Cache verringert die Zugriffszeit für einen Treffen gegenüber einem zweifach-Satz-assoziativen Cache (weil direct-mapped schneller ist als x-fach-Satz-assoziativ), und die Verzögerung durch einen Fehlschlag wird stark verringert, falls sich das Datum im victim-Cache befindet. Andernfalls muß aus dem vergleichsweise langsamen Hauptspeicher geladen werden, und die zusätzliche Verzögerung für das Durchsuchen des victim-Caches fällt dann nicht sehr ins Gewicht. Dies bringt nur einen Vorteil, wenn es nur wenige Zeilen im Cache gibt, um die mehrere Daten “konkurieren”, da sonst der victim-Cache zu groß sein müßte. In der Regel ist es jedoch sinnvoller, den “normalen” direct-mapped-Cache stattdessen mit einer geringen Assoziativität zu versehen.
- column-Cache
  Der column-Cache soll mit Hilfe eines direct-mapped-Caches einen Cache mit Assoziativität von zwei emulieren. Soll eine neue Zeile in den Cache eingelagert werden, und ist diese Position im Cache durch eine noch benötigte Zeile besetzt, wird mit Hilfe einer Flip-Funktion ein Bit des Adreßtag der neuen Zeile negiert und damit entsprechend eine neue Position im Cache bestimmt. Zusätzlich wird für jede Zeile ein Bit benötigt um anzuzeigen, ob der Adreßtag modifiziert wurde. Ein Zugriff auf den column-Cache dauert länger, weil bei jedem Zugriff zusätzlich das Flip-Bit überprüft werden muß. Problem treten auf, wenn die neu bestimmte Position wieder ein Datum enthält, was noch gebraucht wird oder seinerseits verschoben wurde. Dann ist ein weiteres Bit nötig.

3.4 Hauptspeicher

Zusätzliche Literatur: [Bhav91, Burn70, Burr92, Cheo90, Cher91, Cheu87, Cheu90, Jess87, Li91, Rau91].

Der Hauptspeicher ist relativ groß und wird auch Arbeits- oder Primärspeicher genannt. Es kann direkt mit dem Instruktionssatz der Prozossoren auf den Hauptspeicher zugegriffen werden. Er sollte möglichst viele Programme und Daten des laufenden Prozesses enthalten, um einen Zugriff auf Sekundärspeicher zu vermeiden, die vergleichsweise sehr langsam sind (ca. 100-faches der Zugriffszeit).

Der logische Hauptspeicher selbst kann auch aus mehreren physikalischen Speicherbausteinen (Halbleiter-ICs) von unterschiedlicher Art und Zugriffsgeschwindigkeit bestehen. Ein Speicherbelegungsplan (memory map) legt fest, durch welchen Speicherbausteinen von welcher Art die verschiedenen Teile des Hauptspeichers realisiert sind.

Die Entwurfsentscheidungen zur Planung des logischen Speichers sind hier in Form von Parametern beschrieben. Auch sie lassen sich einteilen in Struktur- (Vorhandensein von Puffern vor den Modulen bei der Speicherverschränkung) numerische (Größe, Seitengröße, Anzahl der Module bei der Speicherverschränkung) und symbolische (alle anderen Parameter) Entwurfsentscheidungen.

3.4.1 Größe

Die Größe bezieht sich auf die Speicherkapazität des Hauptspeichers. Hier ist die reale Größe des Hauptspeichers gemeint.

- Wertebereich: {4 MByte - 2 GByte}
- Bemerkungen:
  - Der Hauptspeicher sollte groß genug gewählt werden, um einen Zugriff auf den Sekundärspeicher während der Prozeßlaufzeit zu verhindern.
  - Der virtuelle Adreßraum kann sehr viel größer als der reale Hauptspeicher sein.

3.4.2 Speicherverwaltung

Die Speicherverwaltung strukturiert den Speicher in Teilbereiche und verwaltet den Zugriff auf diese Bereiche über verschiedene Tabellen, in denen die reale Adresse, Zugriffsrechte und Informationen für die Freispeicherverwaltung enthalten ist. Die verschiedenen Möglichkeiten werden im folgenden näher erläutert.

- Wertebereich: {Identität, Seitenadressierung, Segmentierung,
  Segmentierung mit Seitenadressierung.}
- Bemerkungen:
  - Abhängig von der gewählten Art der Speicherverwaltung sind weitere Parameter festzulegen (s.u.).

a) Identität

Bei der Identität entspricht der virtuelle Adreßraum dem realen. Der Datenzugriff erfolgt also direkt über die realen Adressen im Hauptspeicher.

- Bemerkungen:
  - Bei der Identität sind die virtuellen Adressen gleich den realen Adressen
  - Die Identität wird nur bei einfachen PCs, kleineren Minicomputern und älteren Maschinen benutzt
  - Der Adreßraum ist auf die Größe des Hauptspeichers beschränkt.
  - Zusammenhängende virtuelle Speicherbereiche müssen auch real zusammenhängend abgespeichert werden, das führt zu einen schlechten Speicherausnutzung.
  - Der Mehrprozeßbetrieb wird über insgesamt zwei Grenzregister geregelt, die die Grenzen des laufenden Prozesses speichern und bei Prozeßwechsel umgeladen werden. Der Mehrprozeßbetrieb ist umständlich, weil die Daten der Prozesse beim Umladen immer an die gleiche Stelle wieder eingelagert werden müssen.
  - Es kann immer nur der gesamte von einem Prozeß benötigte Speicher ein- und ausgelagert werden. Er muß immer an die gleiche Stelle geschrieben werden.
  - Beim Binden und Laden sind Adreßmodifikationen nötig

b) Seitenadressierung

Bei der Seitenadressierung wird der Speicher in Seiten konstanter Größe eingeteilt. Verschiedene Prozesse können dieselbe Seitenadresse (virtuell) benutzen, die dann bei einem Zugriff auf verschiedene Speicherbereiche (real) abgebildet werden. Innerhalb der Seiten wird jedoch real adressiert. D.h. der virtuelle Speicher ist deutlich größer als der reale. Für jeden Prozeß gibt es eine Seitentabelle, in der

zu jeder virtuellen Adresse des Prozesses die reale enthalten ist. Desweiteren hält ein Systemregister die Basisadresse der Seitenadresse des laufenden Prozesses.

Der Zugriff erfolgt also über eine virtuelle Adresse, die aus einer virtuellen Seitenadresse und dem realen Offset besteht. Die virtuelle Seitenadresse verknüpft mit der entsprechenden Basisadresse aus einem Systemregister zeigt auf den Eintrag in der Seitentabelle, der die reale Seitenadresse enthält. Diese verkettet (concat) mit den Offset ergibt die reale Adresse im Hauptspeicher (Abb. 5).

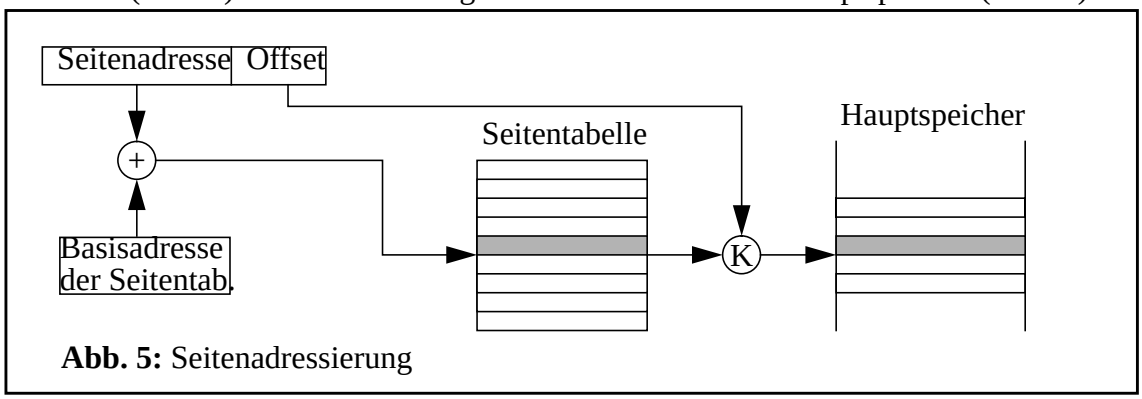

Abb. 5: Seitenadressierung

Beim Zugriff werden auch die Zugriffsrechte, die in der Seitentabelle gespeichert sind, überprüft. Die Seitentabelle wird im Hauptspeicher gehalten. Die virtuelle Seitengröße entspricht dabei der Größe der Kacheln, in die der reale Speicherbereich strukturiert wird.

- Wertebereich: [0.5 - 8] KByte
- Bemerkungen:
  - Seiten / Kachel haben eine konstante Größe, dadurch ist es möglich, eine Seite an einer beliebigen Stelle einzulagern, ohne andere Seiten verschieben zu müssen.
  - Durch die konstante Seitengröße kann es zu einer internen Fragmentierung kommen, wenn die zu speichernden Daten nicht die gesamte Seite ausnutzen.
  - Eine Adreßkorrektur ist nur beim Binden, nicht beim Laden erforderlich, weil jeder Prozeß bei einer virtuellen Adressen 0 beginnen kann.
  - Die Seitentabelle kann sehr groß und in dieser Form nur als Ganzes ausgelagert werden.

c) Segmentierung

Bei der Segmentierung wird der Hauptspeicher in Segmente unterschiedlicher Länge aufgeteilt, die sich jeweils zu Programmlaufzeit in eine Richtung ausdehnen können und bzgl. der Zugriffsrechte homogen sind. Im Prinzip läuft der Zugriff über die virtuelle Adresse ähnlich wie bei der Seitenadressierung, allerdings muß der Offset zur realen Adresse addiert werden, und es muß überprüft werden, ob die Länge des Segmentes, die in der Segmenttabelle gespeichert ist, nicht überschritten wird.

- globale Segmentierung (segmented name space):
  Bei der globalen Segmentierung gibt es im System eine globale Segmenttabelle, die die Segmentbezeichner der Segmente aller Prozesse verwaltet. Hierbei müssen in einer Adresse Segmentname und Offset separate Wertebereiche haben (2-dimensionale Adressierung), weil sonst auf das nächste Segment zugegriffen würde, das eventuell zu einem anderen Prozeß gehört. Ein globales Segment-Register enthält die Anfangsadresse der Segmenttabelle.
- lokale Segmentierung (lineare Semtentierung):
  Bei der linearen Segmentierung gibt es für jeden Prozeß eine lokale Segmenttabelle. Hier ist ein Überlauf des Offsets in den Segmentnamen möglich, weil das folgende Segment zum gleichen Prozeß gehört. Dies ist sinnvoll, wenn der Adreßraum innerhalb des Segmentes nicht ausreicht, und ein logischen Segment deshalb in mehrere Segment aufgeteilt wird. Es gibt also keine 2-dimensional Adressierung mehr. Ein Segmenttabellen-Basis-Register enthält die Anfangsadresse der Segmenttabelle, es muß bei jedem Prozeßwechsel entsprechend umgeladen werden.

- Kombination:
  Es gibt globale (z.B. für Prozeß-Stacks) und lokale Segmenttabellen (z.B. für Daten und Code von jedem Prozeß). Für die lokalen Segmenttabellen des laufenden Prozesses und für die globalen Segmenttabellen gibt es jeweils ein Register mit der Anfangsadresse. Die Register für die Anfangsadressen der lokalen Segmenttabellen müssen entsprechend bei Prozeßwechsel umgeladen werden.

• Wertebereich: {globale Segmentierung, lokale Segmentierung, Kombination}

• Bemerkungen:
  - Bei der Segmentierung kann es zu einer externen Fragmentierung des Speichers kommen, weil die Segmente unterschiedlich lang sein können und somit beim Einlagern eines Segmentes in einen Speicherbereich dieser nicht immer voll ausgenutzt wird.
  - Adreßmodifikationen sind beim Laden nicht und beim Binden nur bei der lokalen Segmentierung notwendig.
  - Beim Verschieben eines Segmentes, was gelegentlich wegen der externen Fragmentierung oder des dynamischen Ausdehnens eines Segmentes notwendig ist, werden alle Adressen durch Verändern der Basisadresse automatisch angepaßt.
  - Bei der globalen Segmentierung ist das Sharing von Daten besonders einfach, weil beim Zugriff über die Globale Segmenttabelle auch auf das gleiche Datum zugegriffen wird. Bei lokalen Tabellen müssen diese entsprechend auf das gleiche Datum zeigen (s.u.).
  - Bei der globalen Segmentierung kann eine Datei als ein nicht eingelagertes Segment behandelt werden. Der Zugriff eines Programms auf eine Datei entspricht dann einem Segment-Fehler. Hier kann dann leicht wieder fortgesetzt werden.

d) Segmentierung mit Seitenadressierung

Bei der Kombination von Segmentierung und Seitenadressierung werden die Segmente des Speichers weiter in Seiten unterteilt.

- 2-stufig:
  Bei der einfachen Kombination aus Segmentierung und Seitenadressierung besteht die virtuelle Adresse aus der Segmentnummer, der Seitennummer und dem Offset. Mit der Segmentnummer, die auf die Segmenttabellen-Basis-Adresse aus einem Register addiert wird, kann ein Eintrag aus der Segmenttabelle adressiert werden. Dieser wird zur Seitennummer addiert und adressiert einen Eintrag in der Seitentabelle. Dieser enthält die reale Adressen, die noch mit dem Offset verkettet wird und liefert die Adresse im Hauptspeicher (Abb. 6).

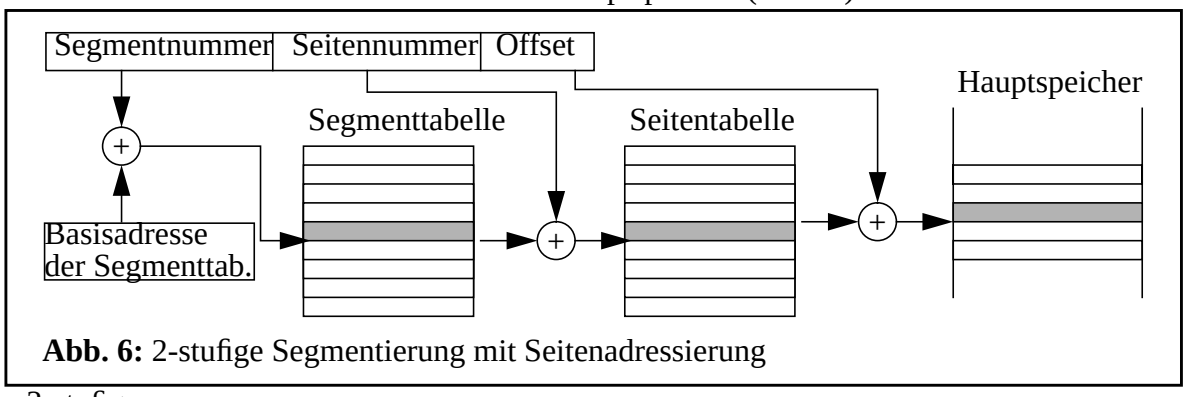

Abb. 6: 2-stufige Segmentierung mit Seitenadressierung

- 3-stufig:
  Bei der dreistufigen Kombination von Segmentierung und Seitenadressierung besteht die logische (virtuelle) Adresse aus einem Selektor und einem Offset. Der Selektor verkettet mit der Segmenttabellen-Basis-Adresse aus einem Register adressiert eine Segment-Deskriptor-Tabelle, deren Inhalt zum Offset addiert wird. Das Ergebnis ist eine lineare Adresse, die aus drei Teilen besteht. Die höherwertigen Bits verkettet mit einer weiteren Basis-Adresse aus einem

weiteren Register adressieren ein Seitentabellen-Verzeichnis, das die Basisadressen der verschiedenen Seitentabellen enthält. Letztere verkettet mit den mittleren Bits der linearen Adressen adressieren schließlich die Seitentabellen, deren Inhalt die reale Adresse der Seite liefert, die noch mit niederwertigen Bits der linearen Adresse verkettet werden muß (Abb. 7).

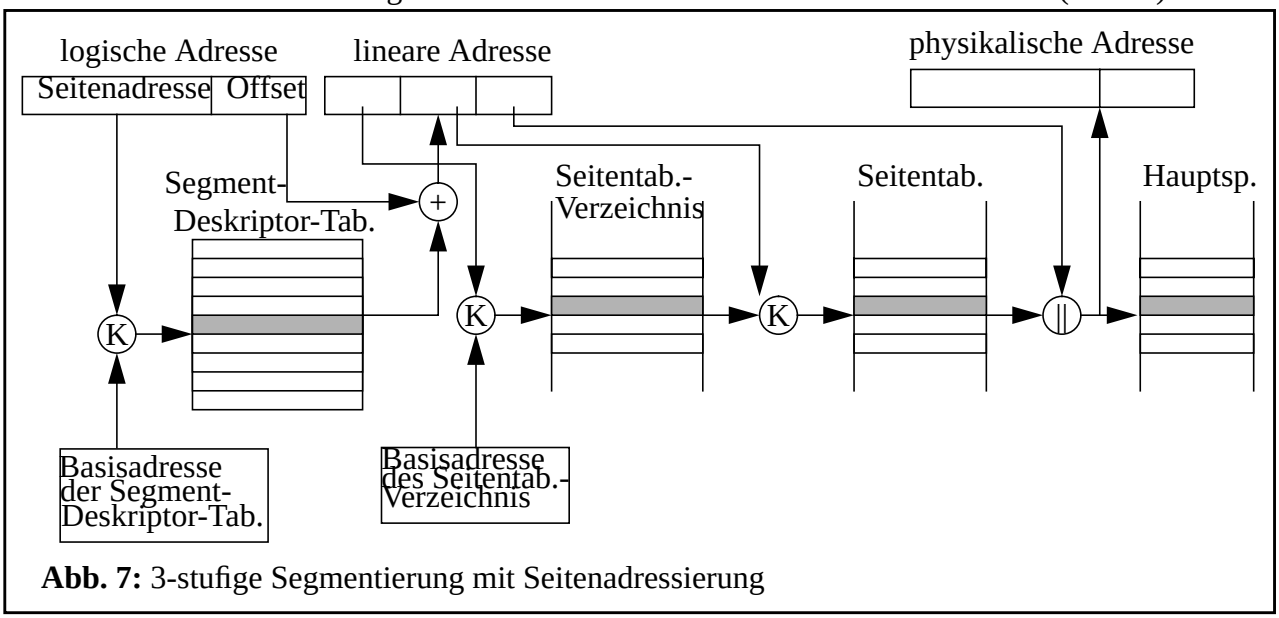

Abb. 7: 3-stufige Segmentierung mit Seitenadressierung

- Wertebereich: {2-stufig, 3-stufig}
- Bemerkungen:
  - Bei der Kombination von Segmentierung und Seitenadressierung gibt es keine externe, nur interne Fragmentierung.
  - Bei der 2-stufigen Kombination von Segmentierung und Seitenadressierung müssen die Seitenadressen von allen Seiten eines Segmentes in einer Seitentabelle physikalisch zusammenhängend abgespeichert werden.
  - Bei der 3-stufigen Kombination von Segmentierung und Seitenadressierung kann die Größe der Seitentabellen auf die Kachelgröße beschränkt werden, es werden entsprechend mehrere Seitentabellen für die Seiten eines Segmentes benötigt, die in einem Seitentabellen-Verzeichnis verwaltet werden. Die Seiten eines Segmentes lassen sich so getrennt ein- und auslagern, und müssen nicht alle gleichzeitig und physikalisch zusammenhängend im Hauptspeicher gehalten werden.

3.4.3 Sharing

Sharing auf Hauptspeicherebene bedeutet, daß verschiedene Prozesse Daten aus einer Seite / einem Segment nutzen. Es gibt verschiedene Möglichkeiten, dies zu verwalten:

- Über globale Deskriptor-Tabelle:
  Von mehreren Prozessen genutzte Segmente werden in die globale Segmenttabelle eingetragen. Die Prozesse greifen dann über dieselben virtuelle Adresse auf die Daten zu.
- Über gemeinsam genutzte lokale Deskriptor-Tabelle:
  Mehrere Prozesse, die Daten gemeinsam benutzen, haben dieselbe lokale Deskriptor-Tabelle und benutzen entsprechend dieselben virtuelle Adresse für gemeinsam benutzte Daten.
- Über gemeinsam genutzte Seite:
  Verschiedene Prozesse können Daten aus derselben Seite / demselben Segment benutzen. Der Zugriff erfolgt über verschiedene lokale Deskriptor-Tabellen. Dies führt zu einem Problem, wenn die Seite / das Segment Adressen enthält, die weiterverarbeitet werden, da die verschiedenen Prozesse verschiedene virtuelle Adressen benutzen.
  Bei der Seitenadressierung müssen deshalb alle Prozesse, die eine Seite gemeinsam benutzen,

dieser auch dieselbe Seitennummer geben.
Bei der Segmentierung muß dieses auch gelten, oder die gespeicherten Adressen aus dem Segment dürfen nicht die Segmentnummer enthalten, diese befindet sich in der Segmentbasisadresse oder in einem Register.

- Wertebereich: {GDT, LDT, Seite}
- Bemerkungen:
  - Die einfachste Verwaltung von gemeinsam benutzter Segmente geht über die globale Deskriptor-Tabelle.

3.4.4 Ladestrategie

Daten können vom Hintergrundspeicher (Platte oder Sekundärspeicher) in den Hauptspeicher geladen werden, wenn sie benötigt werden. Dies verursacht allerdings vorher einen Seiten- oder Segment-Fehler. Um dies zu verhindern, kann eine Seite eingelagert werden, bevor zum ersten Mal auf sie zugegriffen wird.

- paging ohne automatisches Einlagern
  Ein Seite wird ausgelagert, ohne daß eine neue Seite eingelagert wird.
- demand paging mit automatischem Einlagern
  Wird auf ein Datum zugegriffen, daß sich nicht im Hauptspeicher befindet, wird auf den Sekundär-Speicher zugegriffen. Die Seite, in der sich das Datum befindet, wird in den Hauptspeicher eingelagert. Dies ist die übliche Methode.
- prepaging:
  Eine Seite wird eingelagert, bevor auf sie zugegriffen wurde, um beim ersten Zugriff auf diese Seiten die lange Verzögerung durch einen Sekundärspeicherzugriff zu vermeiden. Hiermit kann man bei Prozeßwechsel die Seiten aus der Arbeitsmenge(working set) des reaktivierten Prozesses vorab wieder einlagern.

- Wertebereich: {paging ohne, demand paging, prepaging}
- Bemerkungen
  - Paging ohne automatischen Einlagern ist nur für bestimmte Anwendungen, z.B. Realzeit-Systeme von Bedeutung.
  - Üblicherweise werden Daten nur dann eingelagert, wenn sie benötigt werden.
  - Während ein Prozeß läuft, werden keine Daten für diesen Prozeß vorab vom Sekundärspeicher in den Hauptspeicher geladen, sondern erst, wenn sie gebraucht werden.
  - Wenn bei einem Prozeßwechsel ein alter Prozeß wieder angestoßen wird, ist es vorteilhaft, vorab mittels prepaging die Seiten aus seiner Arbeitsmenge in den Hauptspeicher zu holen.

3.4.5 Plazierungsstrategie

Die Plazierungsstrategie legt fest, an welche freie Stelle im Hauptspeicher ein einzulagerndes Segment plaziert werden soll. Dies ist nur bei der Segmentierung von Belang, da die Segmente eine unterschiedliche Größe haben und somit nicht an jede freie Stelle passen.

- keine Strategie:
  Bei der Seitenadressierung, auch in Kombination mit der Segmentierung, bedarf es keiner besonderen Strategie, wo eine Seite einzulagern ist, weil alle Seiten die gleiche Größe haben, und somit jede Seite an jede freie Stelle in den Hauptspeicher paßt. Hier wird die Position durch die Ersetzungsstrategie bestimmt.
- best fit:
  Die best-fit-Strategie lagert ein Segment an die kleinste freie Stelle in den Hauptspeicher ein, wo es gerade noch hineinpaßt.
- first-fit

Bei der first-fit-Strategie wird das einzulagernde Segment an die erste gefundene freie Stelle (ab dem letzten Einfügen) im Hauptspeicher plaziert, in die es hineinpaßt.

- Wertebereich: {keine Strategie, best-fit, first-fit}
- Bemerkungen:
  - Best-fit und first-fit sind Plazierungsstrategien nur für die reine Segmentierung. Wird Seitenadressierung (mit) verwendet, entscheidet die Ersetzungsstrategie, wohin eine Seite eingelagert wird.
  - Bei der best-fit-Strategie entstehen viele kleine freie Stellen im Speicher, in die in der Regel kein anderes Segment mehr hineinpaßt. Sie können erst wieder genutzt werden, wenn ein angrenzendes Segment ausgelagert wird, oder wenn der Speicher bereinigt wird, d.h. die belegten Teile des Hauptspeichers werden zusammengeschoben, so daß die vielen kleine freien Stellen zu einer größeren zusammengeschoben werden.
  - Bei der first-fit-Strategie entstehen durch das Einlagern eines Segmentes in der Regel größere freie Stellen im Hauptspeicher als bei der best-fit-Strategie, allerdings lassen diese sich eher für weitere einzulagernde Segmente nutzen.

3.4.6 Ersetzungsstrategie

Soll eine Seite aus dem Sekundärspeicher in den Hauptspeicher eingelagert werden, ohne daß Platz dafür vorhanden ist, muß eine Seite aus dem Hauptspeicher ausgelagert werden. Welche das ist, entscheidet die Ersetzungsstrategie.

- Random:
  Diese Strategie wählt zufällig eine Seite aus, die ausgelagert wird. Dafür ist kein besonderer Hardware-Aufwand erforderlich, es ist allerdings nur für sehr irreguläres Zugriffsverhalten geeignet. Random läßt sich lokal oder global, für feste oder variable Anzahl von Kacheln einsetzen und ist nicht frei von Anomalien, d.h. daß trotz Vergrößerung der Speicherkapazität die Fehlerrate steigen kann (s.u.).
- FIFO (first-in-first-out):
  FIFO arbeitet nach dem Schlangen-Prinzip. Die Seite, die als erstes eingelagert wurde, wird auch als erstes wieder ausgelagert. Dafür reicht ein Zeiger im Betriebssystem aus, allerdings werden häufig benutzte Seiten zu früh ausgelagert. FIFO ist also nur für Prozesse mit sequentiellem Zugriffsverhalten geeignet. FIFO läßt sich lokal oder global, für feste oder variable Anzahl von Kacheln einsetzen und ist nicht frei von Anomalien.
- LIFO (last-in-first-out):
  Die zuletzt eingelagerte Seite wird als erstes wieder ausgelagert. Die Ergebnisse sind für fast alle Programme schlecht, deshalb wird dieses Verfahren in der Regel nur für spezielle Stapel (stacks) angewendet. FIFO ist nicht frei von Anomalien.
- Clock:
  Beim Clock-Verfahren wird für jede Kachel ein used-bit benötigt, daß beim Einlagern einer Seite und beim Zugriff auf ‘1’ gesetzt wird. Zusätzlich gibt es einen Zeiger, der auf die der zuletzt benutzte Seite (eingelagert oder zugegriffen) folgenden Seite zeigt. Soll eine Seite eingelagert werden, kreist der Zeiger über alle Seiten, trifft er auf ein used-bit ‘1’, wird es auf ‘0’ gesetzt, trifft er auf ein used-bit ‘0’ wird diese betreffende Seite aus- und die neue Seite eingelagert und der Zeiger auf die nächste Seite gesetzt. Spätestens wenn der Zeiger einmal auf jede Seite gezeigt hat, trifft er auf ein used-bit ‘0’. Damit wird verhindert, daß häufig benutzte Seiten ausgelagert werden. Bei rein sequentiellen Zugriffsverhalten entspricht das Verhalten des Verfahrens dem von FIFO. Clock ist lokal für feste Anzahl von Kacheln geeignet und ist nicht frei von Anomalien.
- LRU (least recently used):
  Bei LRU wird jeweils die Seite eines Prozesses ausgelagert, auf die am längsten nicht zugegrif-

fen wird. LRU ist frei von Anomalien. LRU läßt als Liste oder als Dreiecksmatrix (falls zu groß, aufteilen) implementieren.
  - In die Liste werden die Kachelnummern eingetragen, beim Zugriff auf eine Seite wird diese in der Liste gelöscht (falls vorhanden) und vorne eingehängt. Bei einem Seitenfehler wird die Seite am Ende der Liste ausgelagert und aus der Liste entfernt.
  - Bei der Implementierung als Dreiecksmatrix gibt es für jede Seite eine Zeile und eine Spalte. Beim Zugriff auf eine Seite werden die entsprechenden Einträge der Zeile auf ‘0’ und der Spalte auf ‘1’ gesetzt. Ausgelagert wird dann jeweils die Seite, deren Einträge in der Zeile alle ‘1’ und in der Spalte alle ‘0’ sind. Wegen der Symmetrie genügt es, nur eine Hälfte, nämlich die Dreiecksmatrix, zu speichern. Bei großer Anzahl von Seiten kann die Matrix sehr groß werden, gegebenenfalls teilt man dann die Matrix in Teilmatrizen auf.
    LRU wird vorwiegend für wenige Einträge in lokalen Speichern wie Caches benutzt und eignet sich vor allem als festes, lokales Verfahren.
- LFU (least frequently used):
  Die am wenigsten referenzierte Seite wird ausgelagert. Gibt es mehrer solche Seiten, wird unter diesen LRU benutzt. Dieses Verfahren wird selten angewendet.
- Working-Set-Verfahren (WS):
  Beim Working-Set-Verfahren werden zu jedem Prozeß die Seiten im Hauptspeicher gehalten, auf die im letzten Zeitintervall T zugegriffen wurde. Diese Seiten nennt man auch die Arbeitsmenge (working set) WS des Prozesses:

  $$WS(t,T) := \{\text{Seite } i \mid \text{Seitenzugriff auf Seite } i \text{ im Intervall } [t-T,T]\}$$

  Die Implementierung ist relativ aufwendig. Für jede Seite wird ein periodischer Zähler einführen, der in jedem Zeittakt um 1 erhöht wird. Wird auf die Seite zugegriffen, wird deren Zähler zurückgesetzt, läuft ein Zähler über, wird die entsprechende Seite freigegeben.Bei einem Seitenfehler wird eine freigegebene Seite, die nicht notwendigerweise zum gleichen Prozeß gehört, ausgelagert, d.h. es handelt sich um ein globales variables Verfahren. Außerdem werden Seiten auch ohne Seitenfehler freigegeben, und ein Seitenfehler führt nicht immer zur Freigabe einer Seite, d.h. der Speicherinhalt paßt sich automatisch an den Speicherbedarf der Prozesse an.
  Neben der aufwendigen Realisierung besteht ein weiteres Problem in der Wahl des Parameters T. Wird er zu groß gewählt, gibt es bei Seitenfehlern häufig keine freigegebene Seite und eines der oben beschriebenen Verfahren muß zusätzlich angewendet werden. Wird T zu klein gewählt, werden zu früh Seiten freigegeben, obwohl dies nicht nötig ist, d.h. der Speicher wird schlecht ausgenutzt. Das Working-Set-Verfahren ist frei von Anomalien.
- PFF (page fault frequency):
  PFF stellt eine Vereinfachung des Working-Set-Verfahrens dar. Ist die Zeit zwischen zwei Seitenfehlern eines Prozesse größer als ein vorgegebener Parameter P, erhält der Prozeß eine weitere Seite zugeteilt, ist sie kleiner als P, werden alle Seiten, auf die seit dem letzten Seitenfehler nicht zugegriffen wurde, freigegeben. Hierfür wird ein Timer und für jede Seite ein used-bit benötigt. Allerdings ist PFF nicht frei von Anomalien.
- Min:
  Min ist ein theoretisches Austauschverfahren für die Seitenadressierung, daß unter Kenntnis aller Seitenzugriffen (auch die in der Zukunft) eine optimale Vergabe vornimmt. Es wird immer die Seite ausgelagert, auf die in Zukunft am längsten nicht zugegriffen wird. Min ist natürlich frei von Anomalien. Min ist kein praktisch anwendbares Verfahre, da die Seitenzugriffe in der Zukunft nicht bekannt sind. Es wird dazu benutzt, um Austauschalgorithmen für eine feste Größe des lokalen Ausschnitts zu vergleichen.

- Wertebereich: {Random, FIFO, LIFO, Clock, LRU, LFU, WS, PFF}
- besondere Eigenschaften:

Tabelle 2 gibt verschiedene Eigenschaften der Ersetzungsstrategien an, die im Anschluß erläutert

	Random	FIFO	LIFO	Clock	LRU	LFU	WS	PFF
ben. orient.	-	-	-	+	+	+	+	+
fest/var.	f/(v)	f/(v)	f/(v)	f	f	f/v	v	v
global/lokal	g/l	g/l	g/l	l	l	g/l	l	l
Stack-Alg.	-	-	+	-	+	-	+	-
Eignung	irreg. Zugr.	seq. Zugr.	stack. Zugr.		wenig Kacheln			
HW-Aufwand	-	-	-	1 Bit/ Kachel	Δ-Matrix		Zähler/ Kachel	1 Bit/Kachel +Timer
Zugriff	-	-	-	1 Bit setzen	n Bits setzen		n Zähl erh.	1 Bit setzen
Seitenfehler	-	-	-	<n Bits setzen	Matrix durch-suchen		Zähler testen	Zeit testen, Kachel suchen
zus. Verf.	-	-	-	-	-	LRU	+	+

Tab. 2: Eigenschaften der Ersetzungsstrategien

werden.

- benutzungsorientiert:
  Ein Verfahren ist benutzungsorientiert, wenn die Bestimmung der auszulagernden Seite von ihrer Benutzung abhängt.
- fest/variabel:
  Der Platz für jeden Prozeß ist fest oder dynamisch. Bei dynamischer Platzzuteilung muß nicht bei jedem Einlagern auch ein Auslagern erfolgen (wenn ein Prozeß mehr Speicher bekommt), ebenso muß nicht jedem Auslagern ein Einlagern vorangehen (wenn einem Prozeß Speicher entzogen wird).
- global/lokal:
  Es gibt globale und lokale Verfahren und solche, die sich global und lokal einsetzten lassen. Lokale Verfahren verwalten jeden Prozeß getrennt, globale nicht. Werden die Verfahren global eingesetzt, so impliziert das eine variable Anzahl von Kachel für jede Prozeß.
- Stack-Algorithmus oder Anomalien:
  Eine Ersetzungsstrategie ist ein Stack-Algorithmus (stack property), falls bei zunehmender Speicherkapazität die Fehlerrate nicht steigt. Man sagt auch, die Strategie ist frei von Anomalien.

- Bemerkungen:
  - Bei der reinen Segmentierung müssen alle Segmente von beendeten oder unterbrochenen Prozessen als Ganzes ausgelagert werden.
  - Bei der Kombination von Seitenadressierung und Segmentierung ist es möglich, Teile der Segmente auszulagern. Dafür bedarf es dann einer der beschriebenen Strategien.
  - Für die Fehlerraten der einzelnen Algorithmen gilt empirisch i.a. folgende Beziehung:

$$f_{RAND} \quad f_{FIFO} \gtrsim f_{CLOCK} \geq f_{LRU} \geq f_{MIN}$$

3.4.7 Speicherverschränkung(Interleaving)

Durch eine Speicherverschränkung wird der Hauptspeicher in M verschiedene Module (Banks) auf-

geteilt, in denen parallel und unabhängig voneinander nach Daten gesucht werden kann. Eine Adreßentschlüsselungslogik verteilt die ankommenden Adressen der Lese- und Schreibbefehle nach einer bestimmten Verteilungsstrategie auf die verschiedenen Module. Während in einem Modul noch nach Daten gesucht wird, können bereits neue Adressen auf die anderen Module verteilt werden, d.h. in den verschiedenen Modulen kann parallel gearbeitet werden. Vor jedem Modul gibt es einen Puffer, für den Fall, daß mehrere aufeinander folgendende Adressen auf das gleichen Modul zugreifen. Die Speicherverschränkung bringt einen Geschwindigkeitsvorteil, weil der Übertragen der Adressen zum Hauptspeicher sowie auf die einzelnen Module erheblich schneller geht als das Abarbeiten des Lese-/Schreibbefehls in den einzelnen Modulen. Dies ist besonders bei Pipelining-Rechnern und Multiprozessor-Systemen von Bedeutung, da dort in jedem Takt neue Speicherzugriffe generiert werden können.

a) Anzahl der Module
Hier wird festgelegt, in wieviele Module der Hauptspeicher aufgeteilt wird.

- Wertebereich: {1 - 32}
- Bemerkungen:
  - Ab einer bestimmten Anzahl von Modulen ist der Leistungsvorteil nur noch gering.
  - Einige Verteilungsstrategien schränken die mögliche Anzahl der Module ein, z.B. die sequentielle Verteilung, bei der diese Anzahl prim sein soll (s.u.).

b) Puffergröße vor den Bänken
Alle Verteilungsstrategien können nicht garantieren, daß die Adressen gleichmäßig auf alle Module verteilt werden. Um trotzdem die volle Bandbreite der Speicherverschränkung nutzen zu können, wird vor jedem Modul ein Anfrage-Puffer eingesetzt.

- Wertebereich: {0 - 10} Anfragen (Adresse und Datum)
- Bemerkungen:
  - Für SIM ohne Verschiebung (s.u.) bringen Puffer vor den Modulen nicht viel, weil beim Auftreten einer Häufung bei einem Modul diese sich fortsetzt.

c) Verteilungsstrategie
Die Verteilungsstrategie bestimmt, wie die ankommenden Adressen auf die verschiedenen Module verteilt werden. Dies geschieht entweder sequentiell über die Modulo-Funktion (SIM = sequentially interleaved memory) oder pseudo-zufällig (PRIM = pseudo-random interleaved memory).

- SIM (sequentially interleaved memory):
  - einfaches SIM
    SIM oder auch sequentielle Speicherverschränkung, liefert aus der ankommende Adresse A modulo M die Nummer des Moduls, auf das die Adresse verteilt wird, wenn M die Anzahl der Module ist. Dies liefert gute Resultate bei sequentiellem Zugriffsverhalten oder wenn die Schrittweiten (stride) der Zugriffssequenzen prim zu M sind (siehe Anwendungsparameter.).
  - PSIM:
    PSIM verteilt die Adressen auf die Module wie SIM, allerdings ist hier die Anzahl der Module immer prim.
  - SIM mit Verschiebung (skewing):
    Hier wird zur Adresse A ein Verschiebungsfaktor ((A div M) modulo M) addiert und dann modulo M gerechnet, um die Nummer des Moduls, auf das die Adresse verteilt wird, zu erhalten. Dies liefert in der Regel eine gleichmäßigere Verteilung der Adressen auf die Module.
- PRIM (pseudo-randomly interleaved memory):

- PRIM mit Hashfunktion:
  Hier wird die ankommende Adresse A mittels einer Hashfunktion h auf die Modulnummer M = h(A) verteilt. Es ist darauf zu achten, daß h auch die höherwertigen Bits der Adresse zur Verteilung benutzt, da diese gerade das Modul adressieren, während die hinteren Bits die Position in den Modulen kennzeichnen. Würden die höherwertigen Bits nicht benutzt, würde h nur die Position innerhalb der Module permutieren, jedoch nicht die Modulnummer, d.h. an der Verteilung der Adressen auf die Module würde sich nichts ändern. Desweiteren sollte h bijektiv sein.
- PRIM mit Matrix:
  Hier werden Bits der Adresse A durch AND und XOR (Multiplikation und Addition modulo 2) mit einem Bitmuster verknüpft, um die neue Adresse, deren höherwertigen Bits auch die Modulnummer angeben, zu erhalten. Dies geschieht durch Multiplikation des Bitmusters einer Adresse A mit einer Matrix H. Wenn die Adresse n Bits hat und $m = 2^M$ höherwertige Bits für die Modulnummer vorgesehen sind, liefere die Multiplikation die m Bits für die errechnete Modulnummer, die Adresse innerhalb der Module wird nicht verändert. A ist eine $n \times 1$, H eine $m \times n$ und die errechnete Modulnummer eine $m \times 1$ Matrix. H sollte eine Permutation der Bits für die Modulnummer liefern, d.h. die untersten m Zeilen der Matrix sind linear unabhängig.
- irreduzible Polynome (IPOLY):
  Hier wird die Adresse A dargestellt als Polynom $A(x) = a_{n-1}x^{n-1}+...+a_0x^0$ mit x=2 (z.B. für A=101 ist $A(2)=x^2+1$). Wenn A(x) vom Grad n ist und sei P(x) ein irreduzibles Polynom (Primpolynom) vom Grad m, dann läßt sich A(x) eindeutig darstellen als V(x)*P(x)+R(x) mit R(x) vom Grad kleiner m. V(x) liefert dann die Adresse innerhalb eines Moduls und R(x)=A(x) modulo P(x) den Modulindex (z.B. $A(x)=V(x)*P(x)+R(x) =(a_{n-1}x^{n-1-m}+...+a_m)x^m+(a_{m-1}x^{m-1}+...+a_0)$). Alle Polynome sind aus dem Golois Feld GF(2), d.h. alle Koeffizienten sind 0 oder 1 und Multiplikation und Division modulo 2 entsprechen dann AND und XOR. Die neu errechnete Adresse $B(x)=(A(x) \text{ div } x^m)*x^m + (A(x) \text{ mod } P(x))$ ist eine Permutation, also bijektiv.
  Adreßsequenzen, deren Schrittweiten eine 2-er-Potenz ist, werden kurzfristig gleichmäßig verteilt (Es gibt Teilsequenzen der Länge M, die auf verschiedene Module verteilt werden). Bei ungeraden Schrittweiten werden die Adressen sogar langfristig gleichmäßig verteilt: Für eine Sequenzlänge gegen unendlich werden alle Module gleichhäufig angesprochen. Wenn P(x) irreduzible ist, sind die unteren m Zeilen der Matrix (s.o.) regulär. Wenn der ggT von P(x) und A(x) vom Grad g ist, werden nur 2^{m-g} Module angesprochen.

- Wertebereich: { SIM, PSIM, SIM mit Verschiebung,
  PRIM mit Hashfunktion, PRIM mit Matrix, IPOLY}

- Bemerkungen:
  - Eine gute Wahl der Verteilungsstrategie setzt voraus, daß man die Schrittweite üblicher Zugriffssequenzen kennt, oder anders gesagt, je mehr über die Adressen der Zugriffssequenzen bekannt ist, desto besser kann man die Verteilungsstrategie daraufhin optimieren. Es ist jedoch in der Regel schwierig, hier Vorhersagen zu treffen.
  - In den meisten Fällen gilt bzgl. der Leistungsfähigkeiten, daß PRIM mittels irreduziblen Polynoms zwischen SIM und PRIM als Matrix-Multiplikation liegt.
  - Bei Multiprozessor-Systemen, treten verschiedene Adreßsequenzen mit unterschiedlichen Schrittweiten gleichzeitig auf, d.h. das Zugriffsverhalten ist zunehmend irregulär.
  - zu SIM
    - Bei einer einfachen Verteilungsstrategie nach dem Modulo-Prinzip sollte die Anzahl der Module zur Schrittweite der Zugriffssequenzen prim sein, da ansonsten die Adressen nur auf einen Teil der Module verteilt werden. Sei z.B. folgende Adreßsequenz gegeben: {a,a+n,a+2n,a+3n,...} mit Schrittweite n und Anzahl der Module m. Sind n und m prim

zueinander, dann wird die Adreßsequenz gleichmäßig auf alle m Module verteilt. Ist hingegen n=x*m, werden alle Adressen zum Modul ‘a mod m’ verteilt.
  - SIM eignet sich für einige Adreßsequenzen sehr gut und für andere sehr schlecht, d.h. die Ergebnisse hängen stark von den Schrittweiten ab. Allerdings geht die Adreßumsetzung einfach und sehr schnell.
  - Die Beschleunigung der Speicherzugriffe durch Speicherverschränkung mit einfachem SIM liegt bei ca. $\sqrt{m}$ (Wurzel m), wenn m die Anzahl der Module ist, d.h. die Beschleunigungsrate nimmt monoton ab.
- zu PRIM
  - XOR-basiertes PRIM als Matrix implementiert liefert gute Ergebnisse für fast alle Schrittweiten der Adreßsequenzen und somit die besten Ergebnisse, wenn die Schrittweiten variieren, allerdings werden Puffer vor den Modulen benötigt, um diese Leistung zu erreichen, und die Adreßumsetzung kostet Zeit.
  - Um ein gutes irreduzibles Polynom zu finden, müssen alle Polynome vom Grad m darauf getestet werden, welches sich besser eignet, das ist sehr aufwendig.

d) Maßnahmen bei fehlerhaften Bänken

Falls ein oder mehrere Module defekt sind, werden in der Regel nur noch die Hälfte der Module genutzt, damit die Adreßlogik einfach bleibt (ein Bit weniger zur Adressierung). Eine andere Möglichkeit ist, die korrekt arbeitenden Module in Gruppen à 2^{m-1}, 2^{m-2}, usw. Module aufzuteilen, wobei die Bits der Modulnummer nacheinander angeben, ob sich das angesprochene Modul in der Gruppe befindet oder nicht. Damit werden die Adressen des defekten Moduls auf alle anderen Module verteilt.

- Wertebereich: {mit Umstrukturierung, ohne}
- Bemerkungen:
  - Die Möglichkeit der Umstrukturierung trifft vor allem für SIM-Verteilungsstrategien zu. Bei den PRIM-Strategien greift dies erst nach der Adreßumsetzung. Ansonsten müßte nachträglich ein/e neue/s Verteilungs-Funktion/Matrix/Polynom konstruiert werden, was sehr aufwendig wäre.

3.4.8 Blockzugriffsverfahren

Soll mehr als nur ein Datum aus/in dem/n Hauptspeicher gelesen/geschrieben werden, wie z.B. beim Einlagern einer Zeile in den Cache, gibt es besondere Zugriffsverfahren, die dies beschleunigen. Für einen Speicherzugriff werden Zeilen- und Spaltenadresse und jeweils für Zeile und Spalte ein Gültigkeitssignal benötigt. Vor dem Zugriff wird bei einem Bus dieser belegt und danach wieder freigegeben. Soll jetzt ein ganzer Block von Daten übertragen werden, wird bei allen Blockzugriffsverfahren der Bus nur einmal am Ende freigegeben. Außerdem bleiben Zeilenadresse und dazugehöriges Gültigkeitssignal die ganze Zeit angelegt.

- page-mode:
  Im page-mode wird jeweils beim nächsten Datum des selben Blocks die Spaltenadresse und das jeweilige Gültigkeitssignal neu angelegt.
- static-column-mode:
  Beim static-column-mode bleibt das Gültigkeitssignal auch für die Spalte konstant, es wird nur eine neue Spaltenadresse angegeben.
- nibble-mode:
  Beim nibble-mode bleibt die Spalteadresse konstant, es wird nur ein neues Gültigkeitssignal auch für die Spalte angelegt.

- Wertebereich: {kein Blockzugriff, page-mode, static-column-mode, nibble-mode}

- Bemerkungen:
  - Durch die Blockzugriffsverfahren kann auf mehrere Daten eines Blocks schneller zugegriffen werden, weil nicht alle Adressen und Gültigkeitssignale neu angelegt werden müssen.
  - Blockzugriffsverfahren kommen z.B. beim Ein- und Auslagern von ganzen Cache-Zeilen oder sogar Caches zum Einsatz.

3.4.9 Erweiterter Hauptspeicher (extended memory, expanded memory)

Der erweiterte Hauptspeicher ist eine Erweiterung des Hauptspeiches um einen großen aber langsameren Speicher, der wie ein Puffer zwischen Hauptspeicher und Hintergrundspeicher (Platte oder Sekundärspeicher) wirkt. Dies bringt insofern eine Geschwindigkeitssteigerung, als dieser Puffer noch erheblich schneller als der Sekundärspeicher ist. Es gibt verschiedene Möglichkeiten, diese Puffer anzuordnen:

- Caching:
  Beim Caching verhält sich der erweiterte Speicher wie ein Cache zwischen Haupt- und Sekundärspeicher. Ein Seiten- oder Segmentfehler im Hauptspeicher führt also immer dazu, daß zuerst auf den erweiterten Speicher zugegriffen wird, bevor der Sekundärspeicher angesprochen wird.
- Fremdspeicher (remote memory):
  Werden in einem Multiprozessor-System die Hauptspeicher der anderen Prozessoren als erweiterter Speicher wie beim Caching benutzt, spricht man von Fremdspeichern.
- Direktanschluß:
  Beim Direktanschluß wird ein zweiter Speicherbus benötigt, d.h. es kann direkt auf den erweiterten Speicher zugegriffen werden, ohne den Umweg über den Hauptspeicher. So können langsamere Speicherbausteine für den erweiterten Speicher benutzt werden, aber der Zugriff ist dennoch relativ schnell, weil nicht zuerst auf den Hauptspeicher zugegriffen werden muß. Es ist auch möglich, diesen zweiten Speicherbus zu einem Netzwerk auszubauen, an dem mehrere Speichermodule hängen, wie z.B. bei der GigaSUN (VME-Bus für acht erweiterte Speicher).
- Kombination von Caching und Direktanschluß

- Wertebereich: {Caching, Fremdspeicher, Direktanschluß, Kombination Caching / Direktanschluß}
- Bemerkungen:
  - Ein erweiterter Hauptspeichers kann die Leistung für Anwendungen mit großen Datenstrukturen signifikant erhöhen (z.B. ab 2 MByte Daten). Der Direktanschluß ist deutlich am schnellsten, allerdings wird ein zweiter Bus benötigt.
  - Das Verhältnis der Zugriffszeiten zwischen dem schnellen Hauptspeicher und dem erweiterten Hauptspeicher ist entscheidend dafür, welcher Ansatz besser ist, Direktanschluß oder Caching (Fremdspeicher sind eine Art Caching-Variante). Sei n die durchschnittliche Anzahl von Zugriffen auf eine Seite ohne Seitenfehler, C_{fast} die Zugriffszeit für den Hauptspeicher, C_{slow} die für den erweiterten Speicher, C_{fault} der Overhead für einen Seitenfehler, d die durchschnittliche Anzahl von Seiten, die ausgelagert wird und p die Seitengröße in Worten, dann gilt:
    $p\ (1 + d)\ (C_{fast} + C_{slow}) + C_{fault} + n\ C_{fast} \leq n\ C_{slow}$
  - Der erweiterte Hauptspeicher sollte eine Hit-Rate von über 0.9995 haben.
  - Es können bei Multiprozessor-Systemen nicht nur die Speicher der jeweils anderen Prozessoren benutzt werden, sondern sie können auch einen erweiterten Hauptspeicher gemeinsam benutzen. Dies führt zum Konzept der Speicher-Server.

3.4.10 Ein-/Ausgabe

Die Kommunikation des Rechners mit der Peripherie (Drucker, Bildschirm, usw.) läuft über die Ein- und Ausgabe:

- einfache Ein-/Ausgabe:
  Hier läuft für eine Ein- oder Ausgabe im Prozessor selbst ein Programm ab, für jeden Buchstaben die Ein- oder Ausgabe wird ein Ein- oder Ausgabebefehl abgearbeitet, d.h. die Ein-/Ausgabe läuft über den Prozessor.
- DMA (direct memory access):
  Hier spezifiziert der Prozessor die Speicheradresse, das Peripheriegerät und die Anzahl der Bytes, die übertragen werden sollen. Die Übertragung wird von der Peripherie selbst gesteuert. Diese Geräte heißen DMA-Geräte und benötigen selbst einen DMA-Controller. Der Prozessor kann während der Datenübertragung parallel weiterarbeiten. Für die Übertragung von Daten gibt es zwei Möglichkeiten:
  - two-cycle-transfer:
    Der Prozessor gibt die Speicheradresse, die Anzahl der zu übertragenden Daten und die Adresse der Peripherie an den DMA-Controller. Bei der Ausgabe lädt der DMA-Controller die Daten aus dem Speicher in seinen Puffer und von da aus an den Ein-/Ausgabe-Speicher. Die Adresse wird um 1 erhöht und das nächste Datum wird übertragen. Hier sind zwei Zyklen nötig, deshalb der Name.
  - fly-by-transfer:
    Hier wird vom DMA-Controller die Adresse an den Hauptspeicher angelegt und das Peripherie-Gerät angesprochen. Die Daten fließen dann direkt vom Hauptspeicher zum Peripherie-Gerät. Hier wird nur ein Zyklus benötigt.

  Um DMA-fähig zu sein, muß die Konsistenz der Daten im Hauptspeicher sichergestellt sein.
- Ein-/Ausgabe über den Speicher (memory mapped i/o):
  Hier gibt es keine expliziten Ein-/Ausgabebefehle, dies passiert über das Lesen und Schreiben eines für die Ein- und Ausgabe reservierten Speicherbereichs, d.h. die Adressen für die Register der Ein-/Ausgabe-Geräte werden auf Hauptspeicheradressen abgebildet. Dafür geht ein kleiner Teil des Adreßraums für den Hauptspeicher verloren, aber es werden keine expliziten Ein-/Ausgabe-Befehle benötigt und der ganze Instruktionssatz sowie die verschiedenen Adressierungsarten können ausgenutzt werden.

- Wertebereich: {einfache Ein-/Ausgabe, two-cycle-transfer, fly-by-transfer, E/A über Speicher}
- Bemerkungen:
  - DMA-Controller sind heute z.T. so kompliziert, daß sie wie Koprozessoren ausgebaut sind. Sie bestehen aus verschiedenen Status-, Steuer-, Befehls- und Adreß-Registern, Daten- und Adreßzählern und einem Datenpuffer.

3.4.11 Sonstiges

Folgende Punkte sollen hier zunächst nicht näher behandelt werden:

- Anzahl Ports
- Fehlerkorrektur
- Freispeicherverwaltung

3.5 TLB

Zusätzliche Literatur: [Tell90].

Um virtuelle Adressen eines Prozesses in reale Adressen des Hauptspeichers zu übersetzen, werden für jeden Prozeß Tabellen benötigt (je nach Speicherverwaltung eine Seiten- oder Segmenttabelle, oder beides), die zu einer virtuellen Adresse die reale angibt. Diese können aufgrund von Mehrprozeßsystemen und großem virtuellen Adreßraum sehr groß werden und befinden sich, falls möglich, im Hauptspeicher. Diese Tabellen werden durch die MMU verwaltet, wobei deren Parameter im Zusammenhang mit dem Hauptspeicher beschrieben werden.

Um zu verhindern, daß bei einer Adreßübersetzung mehrmals auf den Hauptspeicher zugegriffen wird, werden die zuletzt benötigten Adressen (virtuell - real) in einem Puffer, dem TLB (translation lookahead buffer) gehalten. Anstatt also bis zu dreimal auf den langsameren Hauptspeicher (bei der 3-stufige Segmentierung) zuzugreifen, braucht nur einmal auf den kleinen, sehr schnellen TLB zugegriffen zu werden, wo zu einer virtuellen Adresse direkt die reale Adresse und die Zugriffsrechte gespeichert sind. Letztere werden in der CPU überprüft.

Logisch gesehen liegt der TLB bei einem real adressierten Cache zwischen der CPU und dem Cache und bei einem virtuell adressierten Cache zwischen dem Cache und dem Hauptspeicher, bzw. einem 2.-Level-Cache, falls vorhanden (Abb. 8).

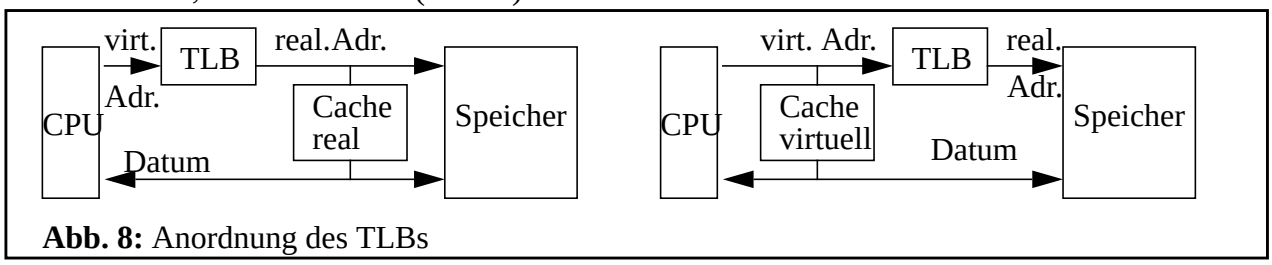

Abb. 8: Anordnung des TLBs

Vom Aufbau her gesehen ist der TLB ein spezieller Adreß-Cache, dem Betriebssystem zur Verwaltung des virutellen Speichers dient. Die Parametern und deren Beziehungen entsprechen also denen eines Caches. Speziell haben die Parameter folgende zusätzliche Eigenschaften und Einschränkungen:

3.5.1 Vorhandensein eines TLBs (Strukturierung)

Ein TLB muß sowohl für virtuell als auch real adressierte Caches vorhanden sein.

- Wertebereich: {ja}
- Bemerkungen:
  - Der TLB ist auch bei virtueller Adressierung sinnvoll, weil für Fehlschläge und für das Rückschreiben auf den Hauptspeicher reale Adressen nötig sind.

3.5.2 Anzahl der TLB-Level (Strukturierung)

Der TLB dient der Beschleunigung der Adreßübersetzung. Einen 2. Level off-chip-TLB würde einen zu großen Verwaltungsaufwand bedeuten.

- Wertebereich: {1}

3.5.3 Trennung des TLB für Daten und Instruktionen (Strukturierung)

Trennung des TLB für Daten- und Instruktionenszugriffe ist nicht sinnvoll.

- Wertebereich: {nein}

3.5.4 Levelnummer

Es gibt nur einen TLB-Level (s.o.).

- Wertebereich: {1}

3.5.5 Integration

Da der TLB der Beschleunigung der Adreßübersetzung dient, sollte er möglichst schnell erreichbar sein.

- Wertebereich: {on-chip, off-chip}
- Bemerkungen:

- Der TLB ist in der Regel auf dem Chip des Prozessors integriert. Bei Systemen mit externem MMU-Chip, kann der TLB auch dort plaziert sein.

3.5.6 Inhalt

Der TLB wird nicht für Daten- und Instruktionsadressen getrennt.

- Wertebereich: {gemischt}

3.5.7 Adressierung

Der TLB wird natürlich virtuell adressiert, das ist hier aber nicht von Belang, da der TLB gerade die Adreßübersetzung vornimmt.

- Wertebereich: {virtuell}

3.5.8 Größe

Hier gibt es im Prinzip drei verschiedene Größenordnungen:

- Der TLB enthält die komplette Seitentabelle eines Prozesses:
  Das ist sehr einfach und liefert eine schnelle Adreßübersetzung, ist aber nur für kleine Adreßräume möglich, weil ansonsten ein großer TLB nötig ist. Zudem dauert das Umladen bei einem Prozeßwechsel lange. In diesem Fall ist die Anzahl der Einträge:
  $$\frac{VirtuellerAdressRaum}{Hauptspeicherseitengroesse}$$
- Der TLB enthält komplette Seitentabellen von mehreren Prozessen:
  Hierbei erhalten einige Prozesse eine festen Tabellennummer und deren Tabelle befindet sich dauerhaft im TLB. Aufgrund der benötigten TLB-Größe ist dies nur für wenige Prozesse und für sehr kleine Adreßräume möglich. Für diese Prozesse wird eine sehr schnelle Adreßübersetzung geliefert, da in diesen Fällen kein Umladen bei Prozeßwechsel nötig ist. In diesem Fall ist die Anzahl der Einträge gleich
  $$AnzahlProzesse \times \frac{VirtuellerAdressRaum}{Hauptspeicherseitengroesse}$$
- Der TLB enthält einen Teil der Seitentabelle eines Prozesses:
  In diesem Fall ist ein TLB mit akzeptabler Größe auch für sehr große virtuelle Adreßräume möglich. Auch die Umladezeit bei einem Prozeßwechsel ist akzeptabel, aber die Adreßübersetzung kann gelegentlich verlangsamt werden, wenn sich der gewünschte Tabelleneintrag nicht im TLB befindet, sondern aus dem Hauptspeicher geholt werden muß. In diesem Fall ist die Anzahl der Einträge 8 bis 1024. Die Größe beträgt somit,
  $$AnzahlEintraege \times Zeilengroesse$$
  wobei die Anzahl der Zeilen den hier zu wählende Parameter darstellt.

- Wertebereich: {8 - 4096}

3.5.9 Zeilengröße

Im Prinzip sieht der TLB folgendermaßen aus (Abb. 9):

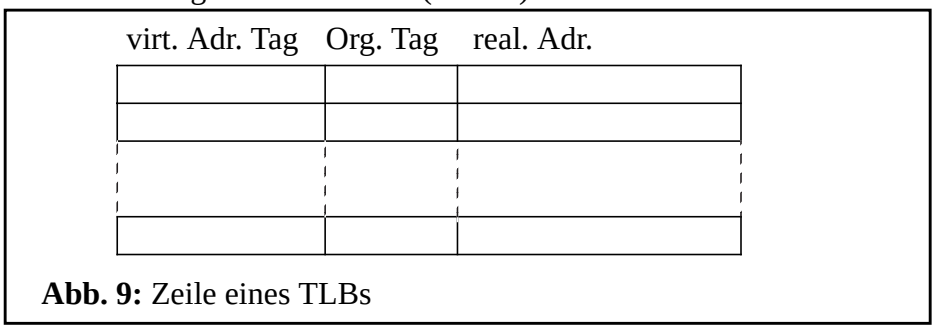

Abb. 9: Zeile eines TLBs

Eine Zeile besteht somit aus den folgenden Komponenten mit den angegebenen Längen:

- Der virtuelle Adreßtag enthält die restlichen nicht über die Adresse des TLB-Eintrags eingegangenen Bits der virtuellen Adresse[1]. Enthält der TLB z.B. die komplette Seitentabelle eines Prozesses, so ist der Tag leer und er vergrößert sich mit zunehmender Assoziativität des TLBs:

$$VirtAdrTag = \log\left\lceil \frac{VirtAdressRaum}{HauptspeicherSeitenGroesse \times AnzahlTlbEintraege} \times Assoziativtaet \right\rceil + ProzessId$$

- Organisationsbits wie z.B. Zugriffsrechte Kennzeichnung über die Gültigkeit benötigen

$$OrgTag = [4-8]\,Bits$$

- Der reale Adreßteil gibt die Adressen der Seiten im Hauptspeicher an:

$$RealAdr = \log\left(\frac{Hauptspeichergroesse}{Hauptspeicherseitengroesse}\right)$$

- Die Anzahl der benötigten Bits beträgt je Eintrag beträgt somit,

$$VirtAdrTag + OrgTag + RealAdr$$

  wobei die Speicherkapazität hier ‘*RealAdr*’ entspricht und hier bereits fest bestimmt ist.

• Wertebereich: {realAdr (s.o.)}

3.5.10 Assoziativität

Sind komplette Seitentabellen im TLB enthalten, wird keine Assoziativität benötigt, ansonsten sollte sie sehr groß bis voll-assoziativ sein, da der TLB relativ klein ist und somit die Fehlschlagsrate sehr hoch wäre, d.h. Adreßübersetzung würde doch hauptsächlich über langsame Hauptspeicherzugriffe laufen.

• Wertebereich: {1, 8 - 32, voll}

3.5.11 Ersetzungsstrategie

Sind komplette Seitentabellen im TLB enthalten, entfällt eine Ersetzung oder bezieht sich auf ganze Tabellen, ansonsten muß sie wie beim 1.-Level-Cache sehr schnell und gut sein, also kommen nur Random oder LRU in Frage.

• Wertebereich: {Random, LRU}

3.5.12 Ladestrategie (Prefetching)

Prefetching entfällt hier, weil es zu aufwendig wäre, d.h. ein Einlagern in den TLB passiert nur, wenn auf die entsprechende Adresse zugegriffen wird.

1. Innerhalb einer Seite wird z.B. ausschließlich real adressiert, d.h. dieser Teil der Adresse muß nicht im TLB gespeichertwerden, weil er nicht übersetzt wird.

- Wertebereich: {anfragegesteuert}

3.5.13 Schreiben

a) Aktualisieren des Hauptspeichers (updating)
Es ist eine Art Durchschreiben nötig, da Änderungen in den Organisationstags (wie valid oder r/w-Permission) sofort weitergeleitet werden müssen, allerdings wird das von der Konsistenzstrategie übernommen.

- Wertebereich: {Durchschreiben}

b) Schreibstrategie
Es wird fetch-on-write gewählt, weil sich in der Regel der Eintrag bereits im TLB befindet. Das liegt daran, daß sich ganze Seitentabellen im TLB befinden, oder daß Änderungen nur dann auftreten, wenn das zugehörige Datum vorher angefragt wurde, d.h. die Adresse wurde entweder mit dem TLB übersetzt oder mit den Tabellen im Hauptspeicher und anschließend im TLB gespeichert.

- Wertebereich: {fetch-on-write}

c) Schreibpuffer
Gegebenenfalls fungiert ein Cache als Hintergrundspeicher des TLBs, so daß ein Schreibpuffer nicht nötig ist.

- Wertebereich: {nein}

3.5.14 Start

Bei kompletten Seitentabellen für mehrere Prozesse wird Warmstart gewählt, ansonsten Kaltstart, weil ja bei Prozeßwechsel alle Einträge ungültig werden.

- Wertebereich: {kalt, warm}

3.6 Multiprozessor (Speicher)

Zusätzliche Literatur: [Capp92, Catt88, Chai90, Dewa93, Dubo90, Gara90, Grau90, Hage92, Hill90, Hill92, Heon89, Leno92, Li89, Lilj93, Lins92, Nitz91, Rau91, Sand92, Sing91, Sten90, Stum90, Thak90, Toma93a/b, Vran91, Wang89, Wheel92, Zhou90].

Multiprozessor-Systeme enthalten bis zu einige Hundert Prozessoren. Ein Programm kann beschleunigt werden, indem es parallel auf den Prozessoren ausgeführt wird. Die Programmierung von Multiprozessor-Systemen ist erheblich einfacher, wenn sie gemeinsamen Speicher (shared memory) benutzen. Bei getrennten Speichern (distrubuted memory) sind spezielle Mechanismen zum Datenaustausch auf Software-Ebene nötig. Benutzen mehrere Prozessoren einen gemeinsamen Speicher, kann es sehr schnell zu einem Wettlauf auf diesen Speicher kommen, er ist überlastet.

Um dieses zu mindern, ist es sinnvoll, wenn jeder Prozessor einen privaten Cache besitzt, auf dem Daten zwischengespeichert werden, so daß ein Zugriff auf den gemeinsam benutzten Speicher vermieden wird. D.h. also, daß mehrere Kopien von Daten existieren können, was zu einem weiteren Problem führt, der Inkonsistenz. Diese Inkonsistenzen zu verhindern ist als das Cache-Kohärenz-Problem (cache coherence problem) in Multiprozessor-Systemen bekannt. Hier gibt es eine Vielzahl von Mechanismen und Protokolle, deren Verwendung stark von verschiedenen Eigenschaften des Multiprozessor-Systems (z.B. Kopplung der Prozessoren untereinander, Art der Verbindung, usw.) abhängt.

Hier sind nur die Speicher-Parameter eines Multiprozessor-Systems beschrieben, die sich nicht auf die Strukturierung und Dimensionierung einzelner Speichermodule (s.o.) beziehen, sondern auf deren Zusammenspiel. Diese Entwurfsentscheidungen sind symbolisch. Die Architektur-Parameter eines

Multiprozessor-Systems sind im Kapitel Architektur-Parameter erläutert.

3.6.1 Lokalität der Speicher

Ein Speicher kann lokal bei einem Prozessor oder verteilt angeordnet werden (Abb. 10).

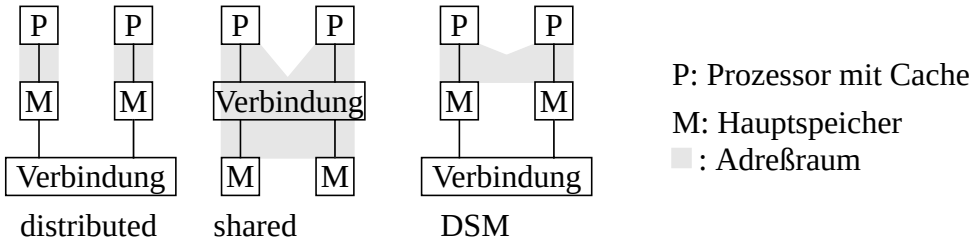

Abb. 10: Lokalität des Speicher

- verteilter Speicher (distributed memory)
  Bei Prozessoren mit verteilten Speichern, auch lose gekoppelte Prozessoren genannt, hat jeder Prozessor seinen eigenen Hauptspeicher, auf den er durch normale Adressierung zugreifen kann. Will er auf Daten in Speichern anderer Prozesse zugreifen, geht das nur über massage passing, d.h. er muß dem entsprechenden Prozeß eine Nachricht senden. Jeder Prozessor ist für die Konsistenz seiner Daten selbst verantwortlich.
  Dies ist eine einfache Organisation, bedingt jedoch ein kompliziertes Programmiermodell, da aus Sicht der Programmierung kein einheitlicher Adreßraum vorhanden ist.
- gemeinsam benutzter Speicher (shared memory)
  Bei Prozessoren mit gemeinsam benutzten Speichern, auch eng gekoppelte Prozessoren genannt, sind die Prozessoren über ein gemeinsames Netzwerk mit den Speichermodulen verbunden. Der Adreßraum von jedem Prozessor umfaßt also alle gemeinsam benutzten Speichermodule. Hier gibt es verschiedene Organisationsmechanismen, eine Datenkonsistenz sicherzustellen.
- verteilter gemeinsam benutzter Speicher (distrubuted shared memory)
  Die dritte Form ist eine Mischform. Hier ist der Speicherbereich sowohl verteilt als auch gemeinsam benutzt (DSM). Jeder Prozessor hat direkten Zugriff auf einen lokalen Hauptspeicher und über die Verbindung kann in der gleichen Art auf Speicher anderer Prozessoren zugegriffen werden, d.h. für jeden Prozessor stellen sich die verschiedenen Speichermodule als ein homogener Adreßbereich dar. Dies erfordert ebenfalls Mechanismen, eine Datenkonsistenz sicherzustellen.

• Wertebereich: {DM, SM, DSM}

3.6.2 Uniformität der Speicherzugriffe

Hier geht es darum, ob alle Prozessoren auf alle Speicher in der gleichen Zeit zugreifen können oder ob Zugriffe von bestimmten Prozessoren schneller ausgeführt werden als andere (Abb. 11).

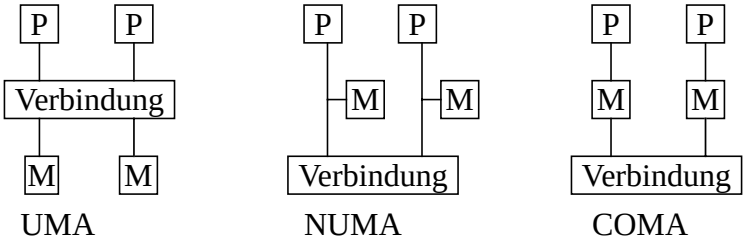

Abb. 11: Uniformität der Speicherzugriffe

- UMA (uniform memory access):
  In UMA-Systemen benötigen alle Prozessoren gleichlange, um auf einen Speicher zuzugreifen,

dies ist auch in SM-Systemen (s.o.) der Fall. Der Speicher wird gemeinsam benutzt.
- NUMA (non uniform memory access):
  In NUMA-Systemen hängt die Zugriffszeit davon ab, welcher Prozessor auf welches Speichermodul zugreift. Dies ist z.B. in DM- und DSM-Systemen (s.o.) der Fall: Der Zugriff auf den eigenen Hauptspeicher ist schneller, als der Zugriff auf das Speichermodul eines anderen Prozessors. Der Speicher ist verteilt und wird gemeinsam benutzt.
- COMA (cache only memory access):
  COMA-Systeme sind eine Mischform der UMA- und NUMA-Systeme. Einerseits dauert der Zugriff auf Speichermodule anderer Prozessoren länger als der Zugriff auf das eigene Speichermodul, andererseits erfolgt der Zugriff stets über den eigenen Speicher, d.h. der eigenen Speicher fungiert als eine Art zweiter Cache und Daten werden immer erst hier gespeichert, bevor sie zum Prozessor gehen. Jeder Prozessor hält also alle Daten, die er benötigt, im eigenen Speicher und sogesehen dauert dann der Zugriff für alle Prozessoren gleichlang, weil sie immer auf den eigenen Speicher zugreifen.
- DDF (data diffusion machine):
  Bei der DDM handelt es sich um ein hierarchisch angelegtes COMA-System.

• Wertebereich: {UMA, NUMA, COMA, DDM}

3.6.3 Granularität

Als Granularität bezeichnet man die Größe von Speicherbereichen, die gemeinsam benutzt werden können.

- fest
  In der Regel werden feste Größen, z.B. eine Seite, als gemeinsam benutzt gekennzeichnet, auch wenn nur ein Teil davon wirklich gemeinsam benutzt wird, z.B. einige Worte. Das bedingt, daß zwei Prozessoren, die verschiedene Teile in einer Seite benutzen, nicht gleichzeitig Zugriff haben (false sharing).
- dynamisch (region-oriented)
  Hier orientiert sich die Kennzeichnung von Speicherbereichen an den tatsächlich gemeinsam benutzten “Portionen”, je feinkörniger diese Einteilung ist, desto höher ist hierfür der Verwaltungsaufwand und desto seltener kommt es zu unnötigen gegenseitigen Behinderungen.

• Wertebereich: {fest, dynamisch}

3.6.4 Private Caches

Haben die Prozessoren in dem System keinen privaten Cache, gibt es keine Konsistenzprobleme. Die sequentielle Konsistenz (s.u.) ist gesichert, wenn die Speicherbefehle atomar sind, d.h. wenn sie nicht unterbrochen werden können. Caches sind aber gerade in Multiprozessor-Systemen notwendig, um die Transferrate zu Speichern (anderer Prozessoren) zu vermindern. Hier sind dann besondere Konsitenz-Mechanismen nötig (s.u.).

• Wertebereich: {ja, nein}

3.6.5 Multilevel Caches

Ein Cache auf dem 2. Level, falls vorhanden, wird zwischen dem 1.-Level-Cache und der Verbindungsstruktur zum Hauptspeicher plaziert. Er enthält gewöhnlich eine Obermenge der Daten aus dem 1.-Level-Cache, damit das Kohärenz-Protokoll handhabbar bleibt und der 1.-Level-Cache nur notwendige Kohärenz- und Ein-/Ausgabe-Operationen ausführen muß (filtern). Der 2.-Level-Cache kann prinzipiell von mehreren Prozessoren gemeinsam benutzt werden. Es sind prinzipiell drei Anordnungen sinnvoll (Abb. 12):

- privat:

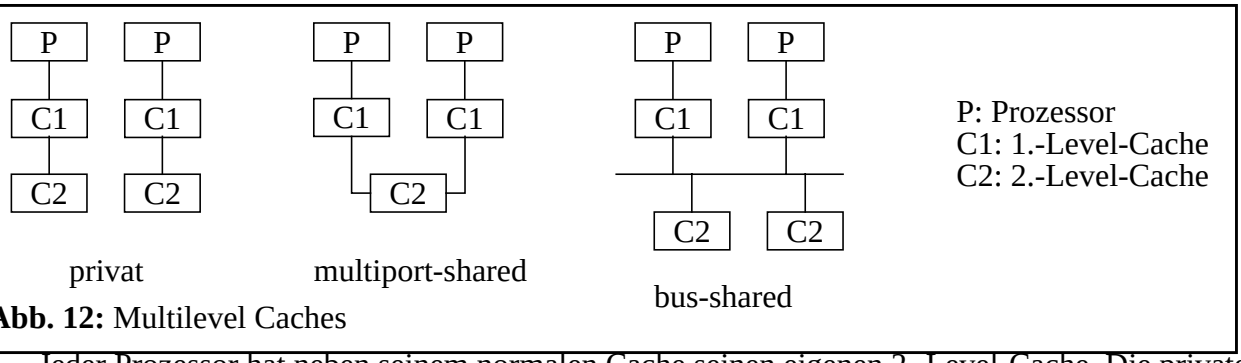

Abb. 12: Multilevel Caches

Jeder Prozessor hat neben seinem normalen Cache seinen eigenen 2.-Level-Cache. Die private Organisation ist gut für kleine direct-mapped Caches auf dem 1. Level mit kleiner Zeilengröße und einem Sektor-organisierten 2.-Level-Cache mit großen Zeilen.

- multiport-shared:
  Einige wenige (≤4) Prozessoren benutzen neben einem private 1.-Level-Cache einen weiteren Cache gemeinsam und greifen über getrennte Ports zu. Dies erfordert eine hohe Assoziativität für den 2.-Level-Cache, wenn er den Inhalt sämtlicher 1.-Level-Caches beeinhalten soll (inclusion property).
- bus-shared:
  Einige (≤ 24) Prozessoren benutzen einen gemeinsamen weiteren Cache und sind über einen Bus damit verbunden. Dies ist flexibler, werden aber viele Prozessoren zusammengeschlossen, ist wieder ein komplettes Kohärenz-Protokoll zwischen den Cache-Ebenen nötig

- Wertebereich: {ohne, privat, multiport-shared, bus-shared}
- Bemerkungen:
  - Ein zweiter Cache vermindert bei einem Fehlschlag die Verzögerungszeit (s.u.) und reduziert den Verkehr zum Hauptspeicher.
  - Der 2.-Level-Cache kann die Aktionen der Kohärenz-Protokolle auf das notwendige filtern, indem Buch geführt wird, welche Daten im 1.-Level-Cache vorhanden sind und jede Information aus dem 1.-Level-Cache auch im 2.-Level-Cache enthalten ist. Er ist gerade in Multiprozessor-Systemen besonders wichtig.
  - Normalerweise wird bei zwei Cache-Leveln der 1.-Level-Cache virtuell und mit Durchschreiben und der 2.-Level-Cache real mit Rückschreiben gewählt.
  - Trick: Wenn der virtuelle 1.-Level-Cache auch Rückschreiben benutzt, ist das schneller. Aber beim Prozeßwechsel, wenn der Cache ungültig wird, würde ein Rückschreiben aller Daten sehr lange dauern. Deshalb werden die Daten nicht rückgeschrieben, sondern nur ungültiggeschrieben. Erst wenn sie überschrieben werden sollen, werden sie zurückgeschrieben, dann reicht auch ein kleiner Schreibpuffer.
  - Die Ersetzungsstrategie beim 2.-Level-Cache kann erweitert werden, so daß zuerst nur die Daten ausgetauscht werden, die in keinem der 1.-Level-Caches enthalten sind. Dies kann nicht garantiert werden, sonst müßte der 2.-Level-Cache eine zu hohe Assoziativität haben, um noch solch ein Datum zu finden.
  - Um die inclusion-property zu garantieren gilt (A: Assoziativität, B: Blockgröße, S: Anzahl Sätze):

$$A_2 \geq \sum A_1(angeschlosseneCaches) \times max(\frac{B_2}{B_1}, \frac{S_1}{S_2})$$

3.6.6 Kopien im Cache

Daten dürfen aus dem/den Speicher(n) in Caches kopiert werden, ist dies nicht erlaubt, treten keine Konsistenzprobleme auf, es genügt das normale Aktualisieren mit Durchschreiben oder Rückschrei-

ben.

Dürfen Daten in den Cache geholt werden, gibt es verschiedenen Möglichkeiten für Kopien:

- Wandern (migration):
  Ein Datum existiert immer nur einmal in einem Cache (keine Kopien), wird ein Datum angefordert, so “wandert” es von seinem derzeitigem Ort zu dem anfragenden Cache.
- Lesekopien (read replication):
  Von einem Datum existiert genau ein schreibbares Exemplar in einem Cache, auf alle anderen Kopien des Datums darf nur lesend zugegriffen werden.
- Schreibkopien mit Lese- und Schreibmöglichkeiten (full replication):
  Es darf mehrere Kopien von Daten in verschiedenen Caches geben, auf die auch schreibend zugegriffen werden darf. Allerdings müssen dann gewisse Vorkehrungen getroffen werden, um die Datenkonsistenz zu erhalten (s.u.).

• Wertebereich: {unerlaubt, Wandern, Lesekopien, Schreibkopien}

3.6.7 Aktualisieren anderer Caches, Propagierte Information (update):

Wird ein Datum in einem Caches geändert, so muß dies zu allen Caches propagiert werden, die eine Kopie des Datum zwischengespeichert haben. Dies ist auf unterschiedliche Art und Weise möglich:

- Ungültigschreiben (write invalidate):
  Es wird an die entsprechenden Caches eine Nachricht geschickt, daß ihre Version des Datums ab jetzt ungültig ist, es wird als ungültig markiert. Es darf nur ein Prozessor schreiben (read replication).
- Aktualisieren (write update, broadcast write):
  Die entsprechenden Caches bekommen das geänderte Datum zugeschickt, das Datum wird aktualisiert.
- Kombinationen
  Beide Verfahren lassen sich kombinieren, wenn man mit Aktualisieren beginnt und bei längeren (3 Schreibzugri**f**e) Schreibsequenzen (write runs) zum Ungültigschreiben übergeht.

• Wertebereich: {invalidate, update, Kombination}

• Bemerkungen:
  - Ungültigschreiben wird in Verbindung mit Durchschreiben oder eher noch Rückschreiben verwendet. Aktualisieren ist nur sinnvoll in Verbindung mit Durchschreiben.
  - Für Ungültigschreiben gibt es einen Trick: Fordert ein Prozessor ein Datum über einen Bus an, können all anderen Prozessoren an dem Bus die eigene Kopie aktualisieren, d.h. es wird auch Aktualisiert, aber nur, wenn ein Datum angefordert wird.

3.6.8 Software/Hardware-Methoden

Die Methoden zur Konsistenzerhaltung der Caches können in Software oder Hardware implementiert sein:

- Software-Methoden:
  Benutzer, Compiler oder Betriebssystem stellen sicher, daß Lese- und Schreibbefehle untereinander koordiniert werden, z.B. durch Verbieten des Zwischenspeichern eines Datums im Cache. Software-Methoden sind in der Regel nicht sehr aufwendig bzgl. Hardware, aber die meisten benötigen eine Hardware-Unterstützung. Sie sind nicht so effizient wie Hardware-Methoden, weil sie zur Compile-Zeit arbeiten, d.h. genaue Angaben über das Sharing von Daten liegen noch nicht vor, weil der genaue Programmverlauf nicht vorhergesagt werden kann. Außerdem sind sie nicht so leicht zu programmieren.
- Hardware-Methoden:

Sie sind in der Regel effizienter als Software-Methoden, weil sie zur Laufzeit arbeiten und der Programmverlauf klar ist. Der dafür benötigte Hardware-Aufwand ist aufgrund der Technologie-Fortschnitte aber tragbar. Außerdem sind sie auf allen Ebenen der Software transparent, d.h. sie bieten ein allgemeines einfaches Programmiermodell, so daß die Software sich nicht mit dem Konsistenz-Problem befaßt ist. Man nennt sie auch Cache-Kohärenz-Protokolle.
  - Synchronisationsmechanismen:
    s.u.
  - gemischte Formen:
    In neuerer Zeit gibt es auch einige Methoden, die sowohl auf Software als auch auf Hardware abstützen.
- Wertebereich: {Software, Hardware, Synchronisation, gemischt}
- Bemerkungen:
  - Ein Schreibzugriff dauert bei Multiprozessor-Systemen länger als bei 1-Prozessor-Systemen, weil Maßnahmen zur Konsistenzerhaltung nötig sind. Hierbei verzögern Hardware-Methoden weniger als Software-Methoden.

3.6.9 Cache-Kohärenz-Protokolle

Es gibt zwei verschiedene Hardware-Protokolle, die sicherstellen, daß die Konsistenz der Informationen in den Caches erhalten bleibt:

- Snooping-Protokoll (snooping cache protocol):
  Wird ein Datum geschrieben, bekommen alle Caches eine Nachricht, unabhängig davon, ob sie eine Kopie des Datum zwischengespeichert haben oder nicht. Hierzu ist eine Busverbindung zu den Caches nötig, damit die Nachricht bei allen Caches gleichzeitig ankommt. Dieses Verfahren ist geeignet, wenn in vielen verschiedenen Caches Kopien von Daten existieren, nicht sehr viele Prozessoren da sind und ist ideal und kosteneffizient für Bus-Systeme.
- Verzeichnis-Protokoll (directory-based):
  Hier wird in einem Verzeichnis buchgeführt, welche Caches Kopien von den jeweiligen Daten halten. Nur diese werden gegebenenfalls benachrichtigt. Hier sind noch einige Entwurfsentscheidungen zu treffen (s.u.). Dieses Verfahren wird bei Verbindungsstrukturen, die keine Busse sind, bevorzugt und ist von Vorteil, wenn nur wenige Caches Kopien eines Datums halten.
- Hybride Methoden:
  Hierbei werden Cluster von Prozessoren gebilden. Auf den verschiedenen Ebenen können dann verschiedene Methoden zum Einsatz kommen.

- Wertebereich: {Snooping, Verzeichnis, hybrid}
- Bemerkungen:
  - Snooping-Protokolle können mit Aktualisieren oder Ungültigschreiben arbeiten, letzteres erlaubt nur Lesekopien.
  - Verzeichnis-Protokolle benutzen Ungültigschreiben, Aktualisieren wäre wegen des mehrstufigen Durchreichens der Daten vor allem bei allgemeinen Verbindungsstrukturen zu aufwendig.

a) Verzeichnis-Verteilung

Wird ein Verzeichnis-Protokoll gewählt, gibt es zwei Möglichkeiten der Verteilung:

- zentral:
  Ein Prozessor hält die gesamte, globale Tabelle, welche Caches welche Datenkopien halten. Dies ist einfach, jedoch kommt es leicht zu Zeitverzögerungen, weil dieser Prozessor überbelastet ist. Ein zentraler Manager ist also nur für sehr kleine Multiprozessor-Systeme geeignet.
- verteilt (distributed):
  Die Tabelle, welche Caches welche Datenkopien halten, wird zwischen den Prozessoren aufge-

teilt. Diese Verfahren sind skalierbar.
- aufgeteiltes Verzeichnis:
  Mehrere Prozessoren erhalten jeweils ein Teil des globalen Verzeichnisses.
- Verkettete Listen
  Für jede Datum gibt es eine einfach oder doppelt verkettete Liste, in der alle Caches verkettet sind, die eine gültige Kopie des Datums halten. Ein zentrales Verzeichnis hält für jedes Datum einen Zeiger auf den Anfang dieser Liste, und jeder Cache hat für jede Datenkopie einen Zeiger auf den nachfolgenden Cache dieser Liste. Dies ist ähnlich effizient wie ein vollständiges Verzeichnis (s.u.).
- Besitzer (owner based):
  Jeder Prozessor ist zu einigen Daten der Besitzer (owner), und kontrolliert für diese Daten den Zugriff.

• Wertebereich: {zentral, verteilt aufgeteiltes Verzeichnis, verteilte Listen, verteilte Besitzer}

b) Dynamik der Verzeichnis-Verteilung

Für die Festlegung, welcher Prozessor die Tabelle oder Teile davon verwaltet, gibt es zwei Möglichkeiten:

- fest
  Die Zuordnung, welcher Prozessor die Tabelle oder Teile davon verwaltet, ist fest.
- dynamisch:
  Im Laufe des Betriebs eines Multiprozessor-Systems wird die Verwaltung der Tabelle oder Teile davon von verschiedenen Prozessoren durchgeführt, d.h. die Verwaltung ist dynamisch verteilt.

• Wertebereich: {fest, dynamisch}

• Bemerkungen:
- Bei dezentraler Verwaltung durch Besitzer ist eine Broadcast-Nachricht die schnellste Möglichkeit, um den Besitzer eines Datums zu finden, der den Eintrag verwaltet.
- Bei dezentraler, dynamischer Verwaltung hält jeder Prozessor eine Tabelle für die gesamten Daten. Allerdings werden nicht bei allen Schreibzugriffen die Tabellen aller Prozessoren aktualisiert, sondern nur die der beteiligten (letzter und zukünftiger Besitzer). Um den derzeitigen Besitzer eines Datums zu finden, müssen eventuell die Tabellen mehrerer Prozessoren durchsucht werden, weil deren Einträge veraltet sein können (probably owner). Um diese Kette nicht zu lang werden zu lassen, werden von Zeit zu Zeit die Einträge aller Tabellen mittels Broadcast-Nachrichten aktualisiert.

c) Vollständigkeit der Information bei zentralen Verzeichnissen

Bei zentralen Verzeichnissen kann die Menge der Informationen über Kopien von Daten sehr groß werden und einen zu großen Teil des Speicher einnehmen. Deshalb gibt es für zentrale Verzeichnisse die Möglichkeit, die Menge der Informationen zu begrenzen:

- vollständiges Verzeichnis:
  Hierbei kann das Verzeichnis die vollständigen Informationen halten (full-map directory), d.h. zu jedem Prozessor und jedem Datum gibt es einen Eintrag, ob der Prozessor das Datum hält. Vollständige Verzeichnisse sind sehr effizient, weil notwendige Nachrichten ganz gezielt verschickt werden können, aber sie sind nicht skalierbar, d.h. nicht erweiterbar bzgl. der Anzahl der Prozessoren und benötigen viel Speicher.
- Teilverzeichnis:
  Hier wird nur protokolliert, wenn ein Prozessor eine Kopie eines Datums in seinem Cache hält. Dafür steht eine begrenzte Zahl von Verweisen zur Verfügung (partial directory). Es sind spezielle Maßnahmen nötig, wenn die Anzahl der gemeinsam benutzten Seiten die vorgesehene maximale Anzahl von Verweisen überschreitet. Es können dann eine oder alle Kopien ungültig-

geschrieben werden, um einen freien Zeiger für die neue Kopie zu bekommen.

- Wertebereich: {vollständiges Verzeichnis, Teilverzeichnis}

3.6.10 Synchronisationsmechanismen

Synchronisationsmechanismen gehören nicht zu den eigentlichen Speicherkohärenz-Methoden, dienen aber ebenfalls der Erhaltung der Konsistenz und sind deshalb der Vollständigkeit halber aufgeführt.

Wenn mehrere Prozessoren auf ein Datum schreiben wollen, muß sichergestellt werden, daß dies nicht gleichzeitig geschieht (mutual exclusion, wechselseitiger Ausschluß). Dies wird durch eine Synchronisation der Zugriffe in Software oder Hardware sichergestellt. Die angegebenen Verfahren sind für gemeinsam benutzte Speicher (shared memories).

- Hardware-Primitiven
  Hardware-Synchronisationsmechanismen sind teurer als Software-Methoden in dem Sinne, daß sie zusätzliche Hardware benötigen. Sie sind aber effizienter, weil Inkonsistenz-Bedingungen flexibel zur Laufzeit von Programmen entdeckt werden können, und es sind keinerlei Konsistenz-Maßnahmen auf Software-Ebene nötig
  TEST&SET und RESET, sowie COMPARE&SWAP gehören zu den sogenannte Read-Modify-Write Mechanismen.
  - TEST&SET und RESET

```
TEST&SET(lock)
{ temp-lock; lock-1; return temp;}
RESET(lock)
{lock-0;}
```

    Die o.a. Schleife wird vom Mikrocode oder der Software solange ausgeführt, bis der Rückgabewert 0 ist, d.h ein anderer Prozessor hat das lock wieder freigegeben. Dann erst kann in den kritischen Abschnitt eingetreten werden. Dieses Verfahren ist sehr komplex: Wenn N Prozessoren auf eine Speicherstelle zugreifen wollen, muß der Speicher N lock-Operationen hintereinander ausführen, auch wenn nur eine davon erfolgreich ist.
  - COMPARE&SWAP

```
COMPARE&SWAP(r1,r2,w)
{temp-w;
 if(temp=r1)then{w-r2;z-1;} else {r1-temp;z-0;}}
```

    Interprozeß-Interrupts verhindern diese Warteschleifen. Statt immer wieder die TEST&SET-Schleife auszuführen, sendet der Prozeß, der das Signal gesperrt hat, Interrupts an die wartenden Prozesse, die dann ähnliche Aktionen, COMPARE&SWAP ausführen: Eine temporäre Variable temp wird mit dem Speicherwort w geladen, entspricht das nicht dem Inhalt von Register r1, so hat ein anderer Prozeß w inzwischen beschrieben (bzw. r1 wird zum ersten Mal mit w geladen), temp (der neue Inhalt von w) wird in r1 geschrieben, ohne nochmal auf den Speicher zuzugreifen. Stimmt der Inhalt von temp (also w) mit r1 überein, dann kann der neue Wert von w in r2 nach w geschrieben werden. Dies vermindert auch den Netzverkehr, weil nur im erfolgreichen Fall (z=1) der Speicher beschrieben wird. Diese Verfahren ist dennoch ähnlich komplex wie TEST&SET, weil zumindest jeder Prozeß lesend auf den Speicher zugreift.
  - fetch&add
    Wenn N Prozesse einen Wert im Speicher inkrementieren wollen, wird bei fetch&add nicht N mal auf den Speicher zugegriffen, sondern eine Hilfsvariable wird N mal sequentiell inkrementiert (was schneller ist, weil nicht auf den Speicher zugegriffen werden muß) und der

Speicherwert wird nur einmal um N erhöht. Dies ist sehr effizient, um z.B. Durchläufe einer Schleife mit mehreren Prozessoren zu parallelisieren, falls sie unabhängig voneinander sind. Die Schleifenvariable wird sequentiell in einer Variablen hochgezählt, jeder Prozessor bekommt den entsprechenden Wert mitgeteilt und die Speicherzelle der Schleifenvariablen wird nur einmal aktualisiert.

- full/empty-bit
  Dies Verfahren ist für heterogene Systeme geeignet. Hierbei ist eine Speicherzelle mit einem Tag versehen. Sie darf nur geladen werden, wenn sie als voll markiert ist. Nach dem Laden wird sie als leer markiert, d.h. andere Prozesse dürfen diese Zelle nicht mehr laden. Beim Zurückschreiben wird die Zelle wieder als voll markiert. Auch bei diesem Verfahren treten Wartezyklen auf, wenn ein Prozeß eine Speicherzelle laden will, die als leer markiert ist.

- Software-Methoden
  Software-Methoden werden vom Betriebssystem, Compiler oder Benutzer initiiert und gesteuert. Sie sind nicht sehr effizient, da sie zur Compile-Zeit arbeiten, aber billig und skalierbar und somit für sehr große Multiprozessoren geeignet
  - Semaphore
    Ein Semaphor ist eine ganzzahlige nicht negative Variable s mit einer Warteschlange, auf die Prozesse nur mit zwei Operationen zugreifen: Mit P (Passeer = Betreten) wird zu Beginn eines kritischen Abschnitts s dekrementiert, falls s>0, andernfalls wird der Prozeß in die Warteschlange eingereiht. Mit V (Verlaaten = Verlassen) wird s bei Verlassen eines kritischen Abschnitts inkrementiert und ein Prozeß aus der Warteschlange ausgewählt. Der Anfangswert von s gibt an, wieviele Prozesse sich gleichzeitig im kritischen Abschnitt befinden dürfen. Sowohl s als auch die Warteschlange muß durch eine der bekannte Hardware-Methoden geschützt werden.
  - Barriere
    Barrieren werden benutzt, um kritische Abschnitte von parallelen Prozessen zusammen und gemeinsam abzuarbeiten. Hierbei muß jeder Prozeß warten, bis alle Prozesse eine Barriere erreicht haben. Der letzte Prozeß weckt alle wartenden Prozesse auf. Dies ist sehr effizient, um iterative Algorithmen parallel abzuarbeiten.
  - message-passing
    Beim message-passing werden Nachrichten zwischen Sender und Empfänger ausgetauscht.
    - synchron
      Im synchronen Fall wartet der Sender darauf, daß der Empfänger eine Bestätigung schickt.
    - asynchron
      Hierbei arbeitet der Sender nach Abschicken der Nachricht weiter, ohne auf eine Bestätigung Empfängers zu warten. Ist dieser bei Eintreffen der Nachricht nicht bereit, diese aufzunehmen, wird die Nachricht entweder zwischengespeichert oder sie geht verloren.

4. Rechnerarchitektur-Parameter

Allgemein verwendete Literatur: [Baeh91, Baer80, Debl90, East90, Ever90, Fluc89, Goor89, Grant89, Grant91, Hama90, Haye88, Henn90, Hsie89, Huck89, Knie89, Koho80, Lang89, Marw93, Muld91, Ober89, Rhei92, Scho88, Siew82, Ston90, Swaa92, Tane84, Tava90, Unge89, Weck82, Weik93a/b].

Die Architektur des Rechnersystems, für den eine Speicherhierarchie entworfen werden soll, hat einen großen Einfluß auf die Speichersynthese, weil für diese Architektur die Speicherzugriffe optimiert werden sollen. Zur Klärung einiger Begriffe sei hier ganz kurz der Aufbau eines Rechners beschrieben:

Ein Rechner besteht aus der Zentraleinheit, auch CPU (central processor unit) oder Prozessor genannt,

und dem Hauptspeicher. Die CPU enthält das Rechenwerk (operation unit) und das Steuerwerk (control unit) inklusive einiger Spezialregister, sowie den Registersatz zum Zwischenspeichern von Daten. Zum Rechenwerk gehören die ALU (arithmetic logic unit), der Akkumulator und einige Spezialregister, und zum Steuerwerk gehören Befehlsregister und -zähler, Dekoder, Mikroprogrammeinheit und das Adreßwerk. Die Mikroprogrammeinheit ist entweder fest verdrahtet oder frei programmierbar (Mikroprozessor). Multiprozessor-Systeme enthalten entsprechend mehrere Prozessoren und Hauptspeicher.

Ein Prozessor-System arbeitet eine Folge von Instruktionen, die im Hauptspeicher abgelegt sind, ab. Die Folge von Mikroinstruktionen, die nötig ist, um eine Instruktion abzuarbeiten, nennt man Instruktionszyklus. Dazu hören z.B. Instruktion holen, dekodieren, Operanden holen und Instruktion ausführen. Die Zykluszeit eines Prozessors ist definiert als Zeit, die nötig ist, die kürzeste Mikroinstruktion durchzuführen. Mit Hilfe der Zykluszeit werden andere Aktionen des Prozessors gemessen. Ihr Kehrwert ist die Taktrate.

4.1 Prozessor

Im folgenden seien die Architektur-Parameter eines einzelnen Prozessors, die für die Speichersynthese von Belang sind, beschrieben.

4.1.1 Mikroprogrammierbarkeit

Hier wird angegeben, ob die Mikroprogrammeinheit frei programmierbar oder fest verdrahtet ist.

- Wertebereich: {fest, programmierbar}
- Bemerkungen:
  - Für fest verdrahtete Mikroprozessoren läßt sich das Zugriffsverhalten auf den Hauptspeicher besser vorhersagen als bei frei programmierbaren.

4.1.2 RISC/CISC

Hier seien einige für die Speichersynthese relevante Unterschiede eines RISC-Prozessors (reduced instruction set) zum CISC-Prozessor beschrieben. Der Instruktionssatz eines RISC-Prozessors enthält gewöhnlich weniger als 100 Instruktionen mit sehr wenig verschiedenen Formaten und Adressierungsarten, es gibt Lade- und Speicherbefehle, um auf den Hauptspeicher zuzugreifen, alle anderen Befehle greifen, falls nötig, nur auf den Registersatz zu und sollten innerhalb eines Taktzyklus' abarbeitbar sein. Um den Speicherzugriff zu beschleunigen, wird eine Harvard-Architektur gewählt, d.h. es gibt getrennte Busse (Daten- und Adreßbusse) für Instruktionen und Daten. Der Registersatz enthält relativ viele meist gleichgroße Register. Dies führt zu einem relativ einfachen Steuerwerk und einheitlicher Befehlsverarbeitung, so daß sich Pipelining gut einsetzen läßt.

- Wertebereich: {RISC, CISC}
- Bemerkungen:
  - Bei RISC-Prozessoren kann nur mit speziellen Lade- und Speicherbefehlen auf den Hauptspeicher zugegriffen werden.
  - Bei RISC-Prozessoren wird eine Harvard-Architektur bevorzugt, d.h. eine Trennung des Caches für Daten und Instruktionen.
  - Manche Prozessoren lassen sich nicht eindeutig als RISC- oder CISC-Prozessoren klassifizieren, da sie nicht alle Merkmale erfüllen.

4.1.3 Pipeline-Stufen

Mit Hilfe des Pipelinings kann die Befehlsverarbeitung im Steuerwerk parallelisiert werden. Während ein Befehl noch verarbeitet wird, kann bereits ein anderer Befehl dekodiert und ein weiterer geholt

werden. Die Pipeline-Stufen geben an, in wieviel Schritten ein Befehl im Prozessor abgearbeitet wird, bzw. wieviele Befehl maximal gleichzeitig bearbeitet werden können. Ist der Prozessor nicht Pipelining-fähig, so ist die Anzahl der Stufen gleich 1.

- Wertebereich: {1 - 12}
- Bemerkungen:
  - RISC-Prozessoren eignen sich eher zum Pipelining, weil sie nur einfache Befehle mit relativ einheitlichem Ausführungsmodus haben.

4.1.4 Taktfrequenz

Die Zykluszeit ist die Zeit, die nötig ist, um den kürzesten Mikrobefehl in einem Prozessor abzuarbeiten, der Kehrwert der Zykluszeit ist die Taktfrequenz(s.o.).

- Wertebereich: {20 - 300 MZ}
- Bemerkungen:
  - DSP-Prozessoren sind verhältnismäßig langsam getaktet.
  - RISC-Prozessoren schneller getaktet als CISP-Prozessoren.

4.1.5 Instruktionssatz

a) Anzahl unterschiedlicher Instruktionen

Hier soll die ungefähre Anzahl der Instruktionen des Prozessors angegeben werden.

- Wertebereich: {20 - 300}
- Bemerkungen:
  - RISC-Prozessoren haben nur einen eingeschränkten Befehlssatz (ca. 20 - 100).

b) Komplexität der Instruktionen

Hier wird die Komplexität der Instruktionen angegeben, das ist der Umfang der Aktionen, die in einem Befehl durchgeführt werden können.

- Wertebereich: {einfach, komplex, sehr komplex}
- Bemerkungen
  - Dies hat Einfluß auf die mögliche Flexibilität der Speicherzugriffe.
  - RISC-Prozessoren haben eher nur einfache Instruktionen.

c) Anzahl Speicherzugriffe pro Instruktion

Hier kann sowohl die durchschnittliche Anzahl von Speicherzugriffen in einer Instruktion als auch eine Einteilung der Instruktionen in Klassen mit unterschiedlichen Anzahlen von Speicherzugriffen angegeben werden.

- Wertebereich: {0-6}
- Bemerkungen:
  - Laden und Speichern eines Doppelwortes gilt als zwei Speicherzugriffe.

d) $\mathbf{CPI_{ex}}$

CPI_{ex} (cycles per instruction execution) gibt die durchschnittlich benötigte Anzahl von Taktzyklen zur Abarbeitung einer Instruktion im Prozessor an. Hierbei wird angenommen, daß Befehle und Operanden jeweils nach einem Takt zur Verfügung stehen (keine Verzögerungen durch den Speicher). Stattdessen kann man dann den Kehrwert benutzen, der wird als IPC (instructions per cycle) bezeichnet.

- Wertebereich: {0.2 - 4}

- Bemerkungen:
  - Bei entsprechend breiten Bussen können auch mehrere Befehle gleichzeitig abgearbeitet werden, damit kann der CPI_{ex}-Wert unter 1 sinken (z.B. DEC Alpha-Chip).
  - Je niedriger CPI_{ex} ist, desto höher sind die Anforderungen an den Speicher.

4.1.6 Anzahl von Registern im Registersatz

Die Größe des Registersatzes ist eine wichtige Eigenschaft eines Prozessors. Die Breite der Register ist in der Regel den Daten- und Adreßwortbreiten angepaßt.

- Wertebereich: {8 - 32, 128 - 500}
- Bemerkungen:
  - RISC-Prozessoren haben einen relativ großen Registersatz, in dem häufig die Register weiter in einer bestimmten Verwaltungsstruktur organisiert sind (z.B. Register-Fenster beim SPARC).
  - Sind ausreichend Register für die Abarbeitung eines sehr lokal arbeitenden Programms vorhanden, wird kein Cache benötigt, dies trifft aber nur in Ausnahmefällen zu.

4.1.7 Systembusbreiten für Adreß- und Datenbus

Hier sind die Bitbreiten für Adreß- und Datenbusse gegebenenfalls von Instruktionen und Daten zwischen Prozessor und Cache bzw. Hauptspeicher gemeint. Der Begriff Datenbus ist hier mißverständlich, da er sowohl Daten als auch Instruktionen transportieren kann, er ist als Unterschied zu weitergegebenen Adressen so benannt.

- Wertebereich: {je 16 - 256}
- Bemerkungen:
  - Die Breite des jeweiligen Adreßbusses legt auch die Größe des adressierbaren virtuellen Speichers fest. Es ist auch möglich, über zwei Transferzyklen eine entsprechend doppelt so breite Adresse zu verwenden und damit den adressierbaren Speicherbereich zu quadrieren.
  - Die Busverbindung zwischen Cache und Hauptspeicher ist in der Regel ein Vielfaches der zwischen Prozessor und Cache.

4.1.8 Prozeßwechselfrequenz

Hier wird angegeben, wie häufig im Durchschnitt ein Prozeßwechsel initiiert wird. Dies ist in der Regel die Aufgabe des Betriebssystems.

- Wertebereich: {klein, mittel, groß}
- Bemerkungen:
  - Die Prozeßwechselfrequenz beeinflußt einige Cache-Parameter: Bei häufigem Prozeßwechsel mit kleinen Prozessen ist ein großer Warmstart-Cache vorteilhaft, weil dann reaktivierte Prozesse noch benötigte Daten im Cache vorfinden. Ebenso beschleunigen große Zeilen das Umladen eines (virtuellen) Caches.
  - Ein Prozeßwechsel wiederum hängt auch von der Speicherorganisation ab. Muß ein Prozessor auf ein Datum aus dem Haupt- oder gar Sekundärspeicher warten, wird der Prozeß unterbrochen, d.h. es findet ein Prozeßwechsel statt.
  - Je höher die Prozeßwechselfrequenz, desto höher ist i.a. die Fehlschlagsrate.

4.2 Multiprozessor (Architektur)

Im folgenden werden die verschiedenen Architektur-Parameter oder -Aspekte näher beschrieben, die sich auf das Zusammenspiel von Prozessoren in einem Multiprozessor-System beziehen und die für die Speichersynthese interessant sind:

4.2.1 Anzahl der Prozessoren

Die Größenordnung des Multiprozessor-Systems spielt bei vielen Speicher-Parametern eine große Rolle. Ein System kann aus einem (Uniprozessor), wenigen (≤4), vielen (≤ 40) oder gar sehr vielen (einige Hundert) Prozessoren bestehen.

- Wertebereich: {1, 2 - 4, ≤40, > 40}

4.2.2 Scalierbarkeit (scalability)

Ein Multiprozessor-System ist skalierbar, wenn ohne großen Aufwand weitere Prozessoren in das System eingefügt werden können. Dazu muß die Verbindungsstruktur und das Kohärenz-Protokoll darauf abgestimmt sein.

- Wertebereich: {ja, nein}

4.2.3 Art der Verbindung

Für die Verbindungsart zwischen den Prozessoren und/oder Speichern gibt es verschiedene Möglichkeiten (Abb. 13):

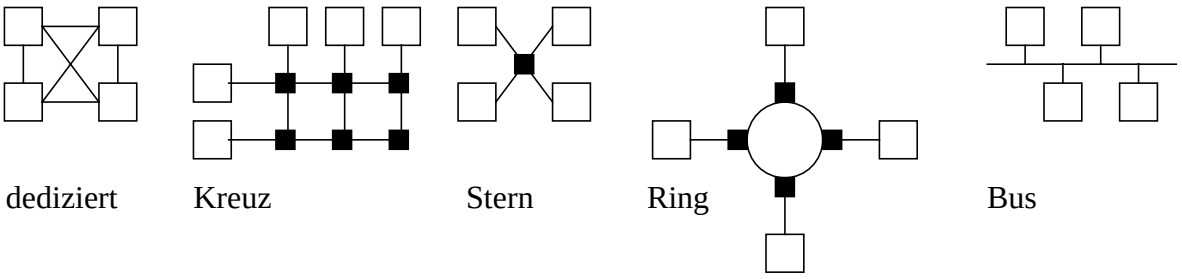

Abb.13: Art der Verbindung im Multiprozessor-System

- dedizierte Verbindung:
  Eine Verbindung, die genau zwei Einheiten miteinander verknüpft, heißt eine dedizierte Verbindung. Sie erlaubt eine schnelle Datenübertragung. Auf jeder Verbindung können parallel und unabhängig voneinander Daten übertragen werden. Es ist einleuchtend, daß für viele Prozessoren/Speicher (N) sehr viele Verbindung nötig sind (N(N-1)/2), falls jede Einheit mit jeder vergunden ist), außerdem ist es aufwendig, die Anzahl der gekoppelten Prozessoren variable zu halten.
- Kreuz (crossbar):
  Bei einer Kreuzverbindung sind N Einheiten mit M anderen Einheiten über N+M Verbindungen und N*M Kreuzungen verbunden. Es kann zu einer Zeit nur ein Kreuzungspunkt in jeder Reihe uns Spalte aktiv sein, d.h. es können maximal min{N,M} Übertragungen gleichzeitig stattfinden, die unterschiedlich lange dauern können. Als Beispiel können N Prozessoren mit M Speichermodulen auf diese Weise verbunden sein.
- Stern:
  Bei der Sternverbindung sind N Einheiten über N Verbindungen und einem Kreuzungspunkt verbunden. Es kann nur eine Übertragung zur Zeit stattfinden.
- Ring:
  Bei einer Ring-Verbindung sind die Einheiten über jeweils einen Ringcontroller mit einem gemeinsamen Ring verbunden, auf dem ein Token kreist, um Daten oder eine Empfangbestätigung derselben zu transportieren. Soll ein Datum geschickt werden, schickt der Sender eine Anfrage zum Empfänger, der seinerseits eine Empfangsbereitschaft signalisiert. Daraufhin schickt der Empfänger eine Nachricht, deren korrekten Eingang der Empfänger quittiert. Da zwei getrennte Verbindungen zwischen je zwei Einheiten existieren, kann eine Unterbrechung toleriert werden. Der Ring wird häufig für größere Entfernungen benutzt, z.B. Apollo-Netz.

  - Bus:
    Der Bus ist eine Verbindung, an der mehrere Einheiten angeschlossen sind. Es kann jeweils nur eine Einheit schreiben, aber mehrere dürfen lesen, dies muß kontrolliert werden. Alle angeschlossenen Einheiten erhalten die Nachricht zur gleichen Zeit. Bei vielen angeschlossenen Einheiten kann es zur Überlastung des Busses kommen, deshalb ist er nur für kleine bis mittlere Multiprozessor-Systeme oder -subsysteme geeignet.
    - synchroner Bus:
      Beim synchronen Bus steht eine festgelegte Zeitspanne für das Übertragen von Daten zur Verfügung. Hier bestimmt natürlich die langsamste Einheit die Zeit.
    - asynchroner Bus:
      Beim asynchronen Bus wird zusätzlich zu den zu übertragenen Daten ein weiteres Signal für den Empfänger mitgeschickt. Hat dieser die Nachricht erhalten, schickt er seinerseits ein Signal an den Sender zurück (handshaking).
  - Mischformen:
    Es gibt viele Möglichkeiten, die verschiedenen Verbindungsstrukturen zu kombinieren. Es können Untergruppen von Einheiten verschiedene Verbindungsstrukturen benutzen, die wieder über eine andere Verbindungsstruktur miteinander verbunden werden.

- Wertebereich: {dediziert, Kreuz, Stern, Ring, synchroner Bus, asynchroner Bus, gemischt}
- Bemerkungen
  - Je mehr Einheiten eine Verbindung gemeinsam benutzen, desto eher ist die Verbindung überlastet. Im umgekehrten Fall wird die Verbindung bei steigender Anzahl von Einheiten teurer. Hier muß ein Kompromiß gefunden werden.

4.2.4 Clusterung

Gruppen von Prozessoren können zu Clustern zusammengefaßt werden, mit jeweils einer unterschiedliche Verbindungsstruktur. Hier wird die Anzahl der Prozessoren in einem Cluster angegeben, ist sie 1, wird nicht in Cluster eingeteilt.

- Wertebereich: {1 - 16}
- Bemerkungen:
  - In der Regel sind die Einheiten in einem Cluster mit einer anderen Art von Verbindung gekoppelt, als die Cluster untereinander, d.h. die Art der Verbindung ist gemischt.

4.2.5 Heterogenität

Man bezeichnet ein Multiprozessor-System als homogen, falls die enthaltenen Prozessoren alle von der gleichen Art sind, d.h. sie sind alle Instanzen eines Prozessortypen, andernfalls ist das System heterogen.

- Wertebereich: {homogen, heterogen}
- Bemerkungen:
  - Bei heterogenen Systemen kann der Austausch von Daten zwischen den Prozessoren einen erheblichen Aufwand bedeuten, besonders, wenn sie Daten oder direkt Speicherbereiche gemeinsam benutzen: Zum einen können die Daten ein unterschiedliches Format besitzen, so daß zunächst eine Format-Transformation stattfinden muß. Zum anderen können die Prozessoren mit einer unterschiedlichen Geschwindigkeit arbeiten, so daß eine besondere Synchronisation nötig ist.

4.2.6 Prozeß-Wandern (migration)

Wird ein Prozeß nach einer Unterbrechung in Prozessor A von einem anderen Prozessor B weiter

bearbeitet, wandert der Prozeß zwischen den Prozessoren A und B (migration). Entsprechend müssen die benötigten Daten zwischen diesen Prozessoren wandern. Dabei kann eine unötige Verzögerung entstehen, wenn Prozessor A noch Daten dieses Prozesses als notwendig hält, während Prozessor B diese eigentlich benötigt (eine Art Schein-Sharing). Hier wird angegeben, ob solch ein Prozeß-Wandern erlaubt ist.

- Wertebereich: {ja, nein}
- Bemerkungen:
  - Prozeß-Wandern kann zu einer Art Schein-Sharing führen (s.o.).

4.2.7 Sharing

Wenn Speicherbereiche von mehreren Prozessoren gemeinsam benutzt werden, heißt das Sharing (sharing). Ist dies der Fall, kann jeder Prozessor direkt auf diese Speicherbereiche zugreifen, ansonsten müssen gemeinsam benutzte Daten mit Hilfe von Nachrichten über die Prozessoren (message passing) wechselseitig ausgetauscht werden. Letzteres verursacht keine Konsistenzprobleme.

- Wertebereich: {ja, nein}

4.2.8 Konsistenz-Modelle

Das Problem der Datenkonsistenz ist ein Hauptproblem in Multiprozessor-Systemen. Greifen mehrere Prozessoren auf gleiche Daten lesend und schreibend zu, spielt die Reihenfolgen, in der diese Speicherzugriffe stattfinden, eine große Rolle. Jedoch ist zum einen nicht immer klar, in welcher globalen (prozessorübergreifenden) Reihenfolge die Prozessoren ihre Speicherbefehle anstoßen und zum andern entspricht diese Reihenfolge nicht notwendigerweise der Reihenfolge, in der diese tatsächlich in den Speichermodulen ausgeführt werden.[1]

Durch unterschiedliche Synchronisationsmechanismen können verschiedene Ausführungsreihenfolgen (Konsistenz-Modelle), die unterschiedlich restriktiv sind, erzwungen werden. Je strikter, desto mehr Wartezeiten müssen in Kauf genommen werden. Wichtig ist, daß der Programmierer (von System-Software) das Konsistenz-Modell kennen muß, bzw. das es so gewählt sein muß, wie es vom Programmierer erwartet wird.

- sequentielle Konsistenz (sequential consistency):
  Ein System ist sequentiell konsistent, wenn das Ergebnis eines beliebigen, auf mehreren Prozessen ausgeführten Programms das gleiche ist, als wenn die Befehle von allen Prozessoren in einer bestimmten Reihenfolge sequentiell ausgeführt worden wären. Die Reihenfolge der Befehle in jedem einzelnen Prozessor muß erhalten bleiben (Prozessor-Konsistenz).
  Eine hinreichende Bedingung hierfür ist, daß ein Speicherzugriff (lesen oder schreiben) erst erlaubt ist, wenn alle eher angestossenen Lesezugriffe nicht mehr durch den neuen Befehl beeinflußt werden können und der Wert des letzten Schreibzugriffs zur Verfügung steht.
- Prozessor-Konsistenz (processor consistency):
  Hierbei muß die Reihenfolge der Befehle in jedem Prozessor erhalten bleiben, die Reihenfolge der Befehle in unterschiedlichen Prozessoren ist nicht bestimmt. Die Bedingungen hierfür sind:
  Ein Lesezugriff ist erlaubt, wenn alle früheren Lesezugriffe nicht mehr von neuen Schreibbefehlen anderer Prozessoren beeinflußt werden können, jedoch ist es möglich, daß der Wert von alten Schreibzugriffen anderer Prozessoren noch nicht zur Verfügung steht, sowie nicht notwendigerweise der von darauffolgenden Schreibzugriffen.
  Ein Schreibzugriff ist erlaubt, wenn alle eher angestoßenen Zugriffe ausgeführt sind, d.h. daß

1. Beispielsweise kann ein Speicherbefehl von Prozessor A, der vor einem Speicherbefehl von Prozessor B angestossen wurde und auf das gleiche Speichermodul zugreift, nach diesem ankommen, weil die Verbindung von Prozessor A zu dem Speichermodul länger oder langsamer ist als die von Prozessor B.

Lesezugriffe, die auf Schreibzugriffen des selben Prozessors folgen, diese “überholen” können.
- schwache Konsistenz (weak consistency):
  Bei schwachen Konsistenz-Modellen wird davon ausgegangen, daß Betriebssystem, Benutzer oder Compiler selbst Synchronisationsmechanismen einbauen. Bedingungen sind:
  Vor einem gewöhnlichen Zugriff müssen alle früheren Synchronisationsbefehle ausgeführt sein.
  Vor einem Synchronisationsbefehl müssen alle früheren gewöhnlichen Zugriffe ausgeführt sein.
  Synchronisationsbefehle sind untereinander sequentiell konsistent.
  Der Schreibzugriff auf gemeinsam genutzte Daten sollte also unter exklusivem Ausschluß erfolgen, Synchronisationsbefehle müssen extra gekennzeichnet werden, aber das Pipelining von Speicherzugriffen wird möglich.
- Versionskonsistenz (release consistency):
  Das Versionskonsistenz-Modell ist das schwächste. Hier ist eine Speicherkonsistenz nur am Ende eines kritischen Abschnitts garantiert.
  Dafür müssen folgende Bedingungen gelten: Vor einem gewöhnlichen Zugriff müssen alle früheren acquire-Befehle ausgeführt sein. Vor einem release-Befehl müssen alle früheren gewöhnlichen Zugriffe ausgeführt sein. Synchronisationsbefehle sind untereinander Prozessorkonsistent.

- Wertebereich: { sequentielle Konsistenz, Prozessor-Konsistenz,
  schwache Konsistenz, Versionskonsistenz}

- Bemerkungen:
  - Sequentielle Konsistenz entspricht der Prozessor-Konsistenz, wenn Speicherbefehle atomar sind, d.h. einmal gestartet dürfen sie nicht von Speicherbefehlen anderer Prozesse unterbrochen werden. Dies gilt normalerweise nicht, sondern muß durch Synchronisationsmechanismen erzwungen werden.
  - Für Prozessoren ohne Cache, die über Busse verbunden sind, ist sequentielle Konsistenz garantiert, wenn sichergestellt ist, daß Speicherzugriffe in der Reihenfolge der Programme ausgeführt werden und Lesezugriffe nicht an Schreibpuffern vorbei zugreifen können.
  - Können Daten in Caches zwischengespeichert werden und die Verbindung ist ein Bus, sind spezielle Hardware-Mechanismen nötig.
  - Bei allgemeinen Verbindungsstrukturen kann der Pfad für den Speicherzugriff und somit die dafür benötigte Zeit, nicht vorhergesagt werden. Sowohl mit als auch ohne Caches sind spezielle Hardware-Mechanismen nötig.
  - Je schwächer ein Konsistenz-Modell ist, desto höher sind die Anforderungen an Benutzer, Compiler oder Betriebssystem, aber desto größer ist die Leistung, weil Befehle nicht so häufig aufeinander warten müssen.

4.3 Technologie-Daten

Die Technologie-Datenbank gibt eine Liste von Speicherbausteinen und deren Beschreibung wieder. Sie wird sowohl beim Syntheseprozeß verwendet, um die Leistung grob abzuschätzen, als auch bei der Simulation, die den Speicherentwurf bewerten soll. Sie enthält zu einem Speicherbaustein folgende Informationen:

a) Name
Hier wird ein Bezeichner für den Speicherbaustein angegeben, um ihn identifizieren zu können.

b) Typ
Hier wird angegeben, um welchen Typ von Speicherbaustein (RAM, ROM, usw.) es sich handelt.

c) statisch/dynamisch
Ein Baustein kann statisch oder dynamisch sein.

d) Technologie

Hier wird die Technologie des Bausteins (CMOS, NMOS, bipolar, usw.) angegeben.

e) Organisation
Hier wird die Größe einer Speicherzelle dieses Bausteins angegeben. Die meisten Bausteine der Firma inmos sind z.B. 1-, 4- oder 8-Bit-weise organisiert. Werden größere Speicherworte benötigt, werden mehrere Bausteine zusammengeschaltet.

f) Speicherkapazität
Hier wird angegeben, wieviel Information auf dem Baustein gespeichert werden kann.

g) Zugriffszeiten
Hier wird die Lese- und Schreib-Zugriffszeit für diesen Baustein angegeben. Für spezielle Zugriffsmodi müssen noch detailliertere Zeit-Werte für das Anlegen der verschiedenen Signale (CAS, RAS, usw.) oder die Zugriffszeit für einen Block angegeben werden.

h) Preis
Hier wird der Preis eines Bausteins angegeben.

5. Anwendungsparameter

Verwendete Literatur: [Catt88, Chen89, Dela90, Harp86, Hoba89, Hyat93, Lin89, McNi88, Meer92, Mesl92, Milu89, Ogur89, Rama89, Vogt90, Woud90].

Anwendungen sind die Programme, die auf dem zu entwerfenden Rechnersystem (inkl. Speicherkonfiguration) ausgeführt werden. Sind die Zugriffseigenschaften dieser Anwendungsprogramme bekannt, kann daraufhin die Speicherkonfiguration optimiert werden. Dazu können die Zugriffseigenschaften, auch Anwendungsparameter genannt, direkt eingegeben werden. Einige dieser Eigenschaften könne auch durch Analyse eines Programms oder seiner Zugriffssequenz extrahiert oder abgeleitet werden.

Die Analysen von Zugriffssequenzen liefert genauere Ergebnisse als die von Programmen, da hier die exakten Adressen der Zugriffe und deren Reihenfolge feststeht. Stehen diese jedoch nicht zur Verfügung, müssen die Programme analysiert werden. Die Zugriffssequenzen werden entsprechend der Ergebnisse der Programmanalyse generiert, um am Ende des Speichersyntheseprozesses eine Bewertung der Synthese mittels Simulation durchzuführen.

Zunächst werden die Anwendungsparameter beschrieben, die direkt in die Speichersynthese eingehen. Diese werden z. T. aus anderen Parametern abgeleitet, die durch Analyse von Programmen oder deren Zugriffssequenzen extrahiert werden. Diese sind danach beschrieben.

5.1 Allgemein

5.1.1 Breite

Die Breite der möglichen Anwendungsprogramme hat Einfluß auf die mögliche Optimierung des Speichers. Je spezieller die Anwendung ist, für die das Rechnersystem entworfen werden soll, desto genauer kann die Speicherkonfiguration auf die Anwendung hin angepaßt werden.

- Universalrechner (genereal purpose):
  Das sind die ganz normalen Rechner-Systeme, deren Prozessoren, die eine möglichst breite Palette an Anwendungen möglichst gut unterstützen soll (z.B. SPARCstation XX). Hier werden die üblichen Programme (SPICE, latex, C-Compiler, usw.), grob analysiert, um für die Speichersynthese aller Universalrechner Durchschnittswerte für die üblichen Fälle zu haben, da keine genaueren Kenntnisse über die Anwendungen zur Verfügung stehen.
- ASIP (application specific instruction set processor):
  Die Prozessoren in dem Rechnersystem sind programmierbar, aber deren Instruktionssatz ist so

ausgelegt, daß bestimmte Anwendungsgebiete besonders gut unterstützt werden. Hier werden mehrere übliche Programme aus dem entsprechenden Anwendungsgebiet analysiert, um Durchschnittswerte für die Speichersynthese aller Rechner aus diesem Anwendungsgebiet (s.u.) zu haben.
  - ASIC (application specific integrated circuit):
    Damit werden Rechner-Systeme mit fest verdrahtetem Programm in den Prozessoren bezeichnet, d.h. es läuft nur ein Anwendungs-Programm auf einem Prozessor. Hier kann das entsprechende Programm genauer auf die u.a. Parameter hin untersucht werden.

- Wertebereich: {Universalrechner, ASIP, ASIC}
- Bemerkungen:
  - Je spezieller die Anwendung, desto genauer kann das entsprechende Programm analysiert und der Speicher daraufhin dimensioniert werden, besonders, wenn der Programmablauf wenig von den Eingaben abhängt (s.u.). Bei allgemeineren Anwendungen werden die typischen Programme untersucht, allerdings muß der Speicher größer ausgelegt werden, weil die Art der Programme und deren Speicherbedarf doch sehr schwanken kann.
  - ASICs sind in der Regel nicht mikroprogrammierbar.

5.1.2 Anwendungsgebiet

Es gibt Anwendungsgebiete, deren Programme ein ganz bestimmtes Zugriffsverhalten zeigen. Ist dieses bekannt, kann die Speichersynthese daraufhin optimiert werden. Einige Beispiele sind hier:

  - Objektorientierte Anwendungen:
    Sie zeigen häufig ein stark lokales Zugriffsverhalten, wenn auf die Objekte zugegriffen wird. Hier kann z.B. ein extra Speicher für die Objekte vorgesehen werden, deren Parameter anders gewählt sind, als für die restlichen Daten.
  - Matrizenkalkulation:
    Bei Programmen zur Matrizenberechnung wird ebenfalls entweder sehr sequentiell oder mit wenig verschiedenen Schrittweiten der Adressen (stride) auf Daten zugegriffen. Dies ist z.B. unter Verwendung von Speicherverschränkung bei der Wahl der Anzahl der Module und der Verteilungsstrategie zu beachten.
  - DSP (digital signal processing):
    Anwendungen aus dem DSP-Bereich greifen sehr schnell und sehr sequentiell auf Daten zu. Dabei sind Zeitbedingungen exakt einzuhalten. Speicher werden z.B. auch dafür benutzt, gegebenenfalls Signale zu verzögern. Die Speicherhierarchie wird häufig nicht voll ausgenutzt. Viele DSP-Prozessoren haben z.B. keinen Cache und der benutzte Speicher ist z.T. on-chip und z.T. off-chip. Aus diesen Gründen entspricht das Vorgehen beim Entwurf von Speichern für DSP-Prozessoren nicht dem hier beschriebenen Ansatz, sondern es ist eine Spezialbehandlung für den Speicherentwurf bei dieser Anwendung nötig, wie z.B. in [Meer92].

- Wertebereich: {objektorientiert, Matrizenkalkulation, DSP, usw.}
- Bemerkungen:
  - Bei objektorientierten Programmen kann ein extra Speicher für die Objekte vorgesehen werden, der aufgrund der Lokalität relativ klein sein kann. So können leichter ganze Objekte ein- bzw. ausgelagert werden, was sehr sinnvoll sein kann.
  - Bei Matrizenkalkulationen reicht die SIM-Verteilungsstrategie für die Speicherverschränkung nicht aus.
  - Speicherkonfigurationen für DSP-Prozessoren werden nicht nach dem gewöhnlichen Schema der Speicherhierarchie (1.- und 2.-Level-Cache, Hauptspeicher) entworfen.

5.1.3 Zeitanforderung

Hier gibt es Echtzeit-Anwendungen oder nicht, d.h. für Echtzeit-Anwendungen sind die angegebenen Zeitbedingungen fest.

- Wertebereich: {Echtzeit, keine Echtzeit}
- Bemerkungen:
  - Bei Echtzeit-Anwendungen sind die angegebenen Zeitvorgaben für Zugriffe fest und unbedingt einzuhalten. Handelt es sich nicht um Echtzeit-Programme, können die Zeitvorgaben geringfügig überschritten werden, wenn dies andere Vorteile (Fläche, Kosten, Busauslastung, usw.) hat.

5.1.4 Lokalität (temporal locality)

Ein Programm / Zugriffssequenz verhält sich lokal, wenn in jedem Zeitintervall fester Länge nur auf wenige (maximal X) verschiedene Daten / Adressen zugegriffen wird. Der englische Begriff ‘temporal locality’ bedeutet genauer, daß nach einem Zugriff auf eine Zelle sehr wahrscheinlich in kurzer Zeit wieder auf diese Zelle zugegriffen wird.

- Wertebereich: {klein, mittel, groß}
- Bemerkungen:
  - Die Lokalität kann direkt bei der Analyse von Zugriffssequenzen extrahiert werden. Bei Programmen muß sie aus den Analyseergebnissen abgeleitet werden.
  - Bei sehr großer Lokalität ist es u.U. sinnvoll, einen Cache nicht durch zu intensives Prefetching zu “verschmutzen”.
  - Je höher die Lokalität, desto niedriger ist die Fehlschlagsrate.

5.1.5 Sequentialität (spatial locality)

Bei sehr sequentiellem Zugriffsverhalten wird fast immer auf die Folgeadresse (die Adresse, die der Adresse des vorangegangenen Zugriffs folgt) zugegriffen. Der englische Begriff ‘spatial locality’ bedeutet genauer, daß bei einem Zugriff in kurzer Zeit auf die Folgeadresse zugegriffen wird, es ist jedoch aufwendiger, dies zu analysieren.

- Wertebereich: {klein, mittel, groß}
- Bemerkungen:
  - Die Sequentialität der Zugriffe kann eigentlich nur durch Analyse der Zugriffsequenzen untersucht werden. Bei Programmen kann man gegebenenfalls grob analysieren, ob die Instruktionsabfolge immer gleich ist, aber das ist kein echtes Kriterium, da die tatsächlichen Zugriffssequenzen nicht zur Verfügung stehen.
  - Bei großer Sequentialität sind für den Cache große Zeilen und Prefetching angebracht, um die Fehlschlagsrate zu verringern. Das wird besonders bei Instruktionszugriffen der Fall sein.
  - FIFO beim Cache ist nur bei sehr großer Sequentialität sinnvoll.

5.2 Programm

Programme werden direkt analysiert, falls die entsprechenden Zugriffssequenzen nicht zur Verfügung stehen, weil z.B. der Rechner, auf dem die Programme laufen sollen, noch nicht fertig realisiert ist oder weil noch kein Compiler exisitiert.

In diesem Fall können nur relativ grobe Analysen gemacht werden, weil durch den Programmablauf entstehende Speicherzugriffe stark vom benutzten Compiler und den zur Verfügung stehenden Registern des Rechners abhängen.

5.2.1 Dynamik

Hier geht es um die Abhängigkeit des Programmablauf von den Eingaben.

- Wertebereich: {klein, mittel, groß}
- Bemerkungen:
  - Ist das Programm fest verdrahtet, spielt die Dynamik eine Rolle: Je weniger der Programmablauf und die daraus resultierenden Speicherzugriffe von der Eingabe des Programme abhängen, desto weniger kann die Zugriffssequenz variieren, d.h. desto genauer kann der Speicher angepaßt werden.
  - Die Dynamik eines Programms ist schwer zu analysieren.

5.2.2 Anzahl Sprünge und Verzweigungen

Eine hohe Anzahl an Sprüngen und Verzweigungen haben Einfluß auf die Sequentialität eines Programms

- Wertebereich: {klein, mittel, groß}
- Bemerkungen:
  - Je mehr Sprünge und Verzweigungen in einem Programm sind, desto seltener wird jeweils auf die Folgeadresse zugegriffen, d.h. das Programm verhält sich weniger sequentiell.
  - Je mehr Sprünge und Verzweigungen, desto ungünstiger ist ein Prefetching für Instruktionen.

5.2.3 Schleifenintensität

Bei starker Schleifenintensität kommen die meisten Zugriffe aus einer Schleife heraus. Dazu kann man drei Werte analysieren:

- Anzahl Schleifen:
  Hier wird die Anzahl der Schleifen bei der Programmausführung gezählt.
- durchschnittliche/maximale Größe der Schleifenrümpfe:
  Das ist die durchschnittliche/maximale Anzahl von Speicherzugriffen in einem Schleifenrumpf.
- durchschnittliche Anzahl von Durchläufen:
  Das gibt an, wie oft eine Schleife durchlaufen wird.

Die Multiplikation dieser drei Werte dividiert durch die Gesamtanzahl der Speicherzugriffe ergibt die Schleifenintensität.

- Wertebereich: {klein, mittel, hoch}
- Bemerkungen:
  - Bei kleinen Schleifenrümpfen, die oft durchlaufen werden, ist das Zugriffsverhalten natürlich sehr lokal.
  - Werden Schleifen oft durchlaufen, sollte der Cache alle Werte in einem Schleifenrumpf aufnehmen können, d.h. die Cache-Größe ist, falls möglich, entsprechend groß zu wählen.
  - Durch häufig durchlaufene Schleifen können Zugriffsequenzen mit konstanten Schrittweiten der Adressen (strides) entstehen. Dies muß bei einer Speicherverschränkung (Anzahl der Module, Verteilungsstrategie) berücksichtigt werden.

5.2.4 Variablenanalyse

Hier werden die in den Programmen verwendeten Variablen genauer untersucht.

a) Maximale Anzahl gleichzeitig lebendiger Variablen und deren Größe

Die Multiplikation dieser beiden Werte ergibt bei guter Ressourcen-Verteilung den maximal benötigten Speicher für die Daten.

- Wertebereich: {natürliche Zahlen}

- Bemerkungen:
  - Ist dieser Wert nicht zu groß, können diese Variablen bei optimaler Verteilung auf die Speicherzellen alle im Cache gehalten werden, d.h. die Cache-Größe ist entsprechend zu wählen.
  - Können sowieso alle Variablen des Programms im Cache oder gar in Registern gehalten werden, was der Ausnahmefall ist (ganz bestimmte Anwendungen), ist die Dimensionierung von Cache und Hauptspeicher trivial, weil z.B. Ersetzungsstrategie, Prefetching, usw. keine Rolle spielen.

b) durchschnittliche Lebenszeit

Die Lebenszeit einer Variablen geht von ihrem ersten (hoffentlich schreibenden) bis zum letzten lesenden Zugriff.

- Wertebereich: {natürliche Zahlen}

c) durchschnittliche Zugriffsfrequenz

Die Zugriffsfrequenz einer Variable ist die Lebenszeit dividiert durch die Anzahl der Zugriffe auf diese Variable.

- Wertebereich: {rationale Zahlen}
- Bemerkungen:
  - Je größer die durchschnittliche Zugriffsfrequenz, desto eher lohnt ein Cache, d.h. Variablen mit hoher Zugriffsfrequenz sollten nicht aus dem Hauptspeicher geholt werden müssen.
  - Eine hohe durchschnittliche Zugriffsfrequenz ist auch ein Maß für Lokalität.

d) Verhältnis von Lese- und Schreibzugriffen

Hier werden die Anzahl der Lese- und Schreibzugriffe zueinander ins Verhältnis gesetzt.

- Wertebereich: {rationale Zahlen}
- Bemerkungen:
  - Bei einer hohen Schreibrate sollte im Cache die Rückschreibestrategie gewählt werden, die Durchschreibestrategie ist nur für eine kleine Schreibrate sinnvoll, da ansonsten die Verbindung zum Hauptspeicher überlastet wird.

e) durchschnittliche Länge von Schreibsequenzen (write runs)

Hier wird die Anzahl der Schreibzugriffe zwischen je zwei Lesezugriffen auf eine Variable gezählt.

- Wertebereich: natürliche Zahlen
- Bemerkungen:
  - Bei langen Schreibsequenzen (≥ 3) ist auf jeden Fall Rückschreiben günstig, auch wenn die Schreibrate gering ist. Bei kurzer Schreibsequenz (≤ 2) ist Durchschreiben u. U. günstiger. Die Strategie kann das auch dynamisch für jede Variable gewechselt werden, d.h. wenn zum zweiten oder dritten mal hintereinander schreibend zugegriffen wird, wird von Durchschreiben zu Rückschreiben gewechselt.
  - Bei Multiprozessoren ist für lange Schreibsequenzen Ungültigschreiben (invalidate) sinnvoll.

f) Verhältnis von Instruktions- und Datenzugriffen

Hier wird die Anzahl der Instruktions- und Datenzugriffen zueinander ins Verhältnis gesetzt.

- Wertebereich: {rationale Zahlen)
- Bemerkungen:
  - Unterscheiden sich die beiden Anzahlen stark (wie z.B. bei RISC-Prozessoren), werden die übrigen Werte (s.o.) zusätzlich für Instruktions- und Datenzugriffe getrennt ausgewertet.

- Unterscheiden sich die o.a. Werte stark für Instruktions- und Datenzugriffe, sind getrennte Zwischenspeicher (Caches) sinnvoll.

g) relative Anzahl von Abhängigkeiten
Eine Variable ist von einer anderen abhängig, wenn sie aus ihr berechnet wird, also A1 := f(...A2...).

- Wertebereich: {klein, mittel, groß}
- Bemerkungen:
  - Bei großer Abhängigkeit kann die Reihenfolge der Speicherzugriffe nur geringfügig variiert werden, um eine korrekte Abarbeitung des Programms zu gewährleisten, d.h. es ist eine strenge Synchronisation nötig.

5.2.5 Sharing

Hier wird angegeben, wie intensiv Daten von Programmen auf verschiedenen Prozessoren gemeinsam benutzt werden.

a) Grad des Sharing
Hier ist größenordnungsmäßig die Anzahl und die Größe der gemeinsam benutzen Daten gemeint.

- Wertebereich: {klein, mittel, groß}
- Bemerkungen:
  - Werden viele Daten gemeinsam benutzt, sind aus Effizienzgründen Hardware-Mechanismen zur Erhaltung der Konsistenz notwendig.

b) Anzahl der Prozessoren, die Daten gemeinsam benutzen
Hier wird angegeben, wieviele Prozessoren jeweils Daten gemeinsam benutzen.

- Wertebereich: {1, ≤ 3, ≤ 10, > 10}
- Bemerkungen:
  - Benutzen sehr viele Prozessoren Daten gemeinsam, kann die Verwaltung dieser Daten über eine globale Seiten-/Segmenttabellen zu einem Engpaß führen, wenn sie gleichzeitig auf verschiedenen Daten, die in derselben Tabelle verwaltet werden, zugreifen wollen. Werden diese Daten über mehrere lokale Tabellen verwaltet, muß allerdings die Daten-Konsistenz sichergestellt werden.
  - Greifen viele Prozessoren häufig auf gleiche Daten zu, sind lokale Kopien in den lokalen Caches sinnvoll. Für das Aktualisieren anderer Caches ist nur bei wenigen Prozessoren das Snooping-Protokoll eignet, da ein Bus benötigt wird, der bei vielen Prozessoren leicht überlastet wäre. Benutzen viele Prozessoren Daten gemeinsam, muß die Verzeichnis-Methode verwendet werden.
  - Bei vielen Prozessoren wird die Konsistenz-Erhaltung vereinfacht, wenn Durchschreiben zur Aktualisierung des Hauptspeichers benutzt wird, allerdings führt dies zu mehr Datentransfer auf den Verbindungen. Ist also die Schreibrate sehr hoch, muß dann trotzdem Rückschreiben verwendet werden.

5.3 Zugriffssequenz

Die Zugriffssequenz eines Programms ist die Folge von Speicherzugriffen, die bei Ablauf eines Programms ausgelöst wird. Zu jedem Zugriff sind folgende Informationen wichtig:

- Adresse:
  die Adresse des Speichers, auf die zugegriffen wird.
- Lese-/Schreibzugriff:
  die Information, ob es sich um eine Lese- oder Schreibzugriff handelt.

- Instruktions-/Datenzugriff:
  die Information, ob auf eine Instruktion oder ein Datum zugegriffen wird.
- Datum:
  Handelt es sich um einen Schreibzugriff, ist das abzuspeichernde Datum anzugeben.
- Prozeßname:
  Hier wird der Name des Prozesses angegeben, der den Speicherzugriff ausgelöst hat.

Steht zu einem Programm das Ablaufprotokoll (trace) zur Verfügung, werden diese Informationen entsprechend herausgefiltert, um die Zugriffssequenz zu erhalten. Die im folgenden beschriebenen Parameter beziehen sich auf Zugriffssequenzen.

5.3.1 Anzahl Sprünge

Wird bei Instruktionszugriffen nicht auf die Folgeadresse der letzten Instruktion zugegriffen, handelt es sich um einen Sprung. Hier ist die relative Anzahl von Sprüngen in einem Programmablauf gemeint.

- Wertebereich: {klein, mittel, groß}
- Bemerkungen:
  - Bei sehr großer Anzahl von Sprüngen ist ein Prefetching für Instruktionen zwecklos.

5.3.2 Schrittweiten

Der Abstand der Adressen zwischen zwei Speicherzugriffen wird als Schrittweite bezeichnet. Hier sollen die häufig auftretenden Schrittweiten angegeben werden.

- Wertebereich: {Liste von Schrittweiten}
- Bemerkungen:
  - Ist die Schrittweite zwischen den Sprüngen immer gleich (stride), sollte die Anzahl der Module bei einer Speicherverschränkung prim dazu sein.
  - Gibt es mehrere verschiedene Schrittweiten, die wiederholt auftreten, ist die Verteilungsstrategie SIM in der Regel nicht geeignet.

5.3.3 Größe der Arbeitsmenges (working set)

Zur Arbeitsmenge gehören alle Daten, auf die in den letzten T Zeiteinheiten zugegriffen wurde. Dieser Parameter gibt die Anzahl der verschiedenen Zugriffe in einem Intervall T an. Dabei kann das Maximum aus einigen Stichproben-Intervallen genommen werden.

- Wertebereich: {1 - T}
- Bemerkungen:
  - Die Arbeitsmenge des laufenden Prozesses sollte (für ein möglichst großes T) komplett im Cache gehalten werden, d.h. der Cache muß entsprechend groß sein. Ist die nicht möglich sollte dies gegebenenfalls für ein kleineres T gelten.
  - Die Größe der Arbeitsmenge für ein festes T ist auch ein Maß für die Lokalität: Je kleiner die Arbeitsmenge desto größer die Lokalität und umgekehrt.

5.3.4 Speicherzellenanalyse

Hier werden die von den Zugriffssequenzen benutzten Speicherzellen genauer untersucht.

a) Anzahl benutzter Speicherzellen
Hier wird die Anzahl der verschiedenen Speicherzellen gezählt, auf die zugegriffen wurde.

- Wertebereich: {natürliche Zahl}
- Bemerkungen:

- Wird nur auf sehr wenige Speicherzellen zugegriffen, und können diese alle im Cache oder sogar im Register gehalten werden, so ist der Entwurf einer Speicherhierarchie trivial.

b) durchschnittliche Zugriffsfrequenz
Der Parameter ist bei der Variablenanalyse (s.o.) beschrieben, wobei hier Speicherzellen und nicht Variablen gemeint sind.

c) Verhältnis von Lesen- und Schreibzugriffen
Der Parameter ist bei der Variablenanalyse (s.o.) beschrieben, wobei hier Speicherzellen und nicht Variablen gemeint sind.

d) durchschnittliche Länge von Schreibsequenzen (write runs)
Der Parameter ist bei der Variablenanalyse (s.o.) beschrieben, wobei hier Speicherzellen und nicht Variablen gemeint sind.

e) Verhältnis von Instruktions- und Datenzugriffen
Der Parameter ist bei der Variablenanalyse (s.o.) beschrieben, wobei hier Speicherzellen und nicht Variablen gemeint sind.

f) Anzahl von Adressen mit gleichen niederwertigen Bits
Hier wird angegeben auf wieviele verschiedene Adressen mit gleichen X (4 - 10) niederwertigen Bits in einem Intervall der Länge T durchschnittlich zugegriffen wird.

- Wertebereich: {1 - T}
- Bemerkungen:
  - Diese Anzahl gibt an, wieviel verschiedene Adressen mit gleichen niederwertigen Bits gleichzeitig im Cache gehalten werden sollten und ist somit ein Maß für die benötigte Assoziativität eines Caches.

6. Ziel-Parameter

Das eigentliche Ziel des Rechnerentwurfs ist die Abarbeitung von Programmen in minimaler CPU-Zeit bei minimalen Kosten gemessen in Fläche bzw. Preis. Dies setzt eine minimale Speicherzugriffszeit voraus. Diese Ziele sind z.T. widersprüchlich. Wie sich diese Ziele aus Unterzielen zusammensetzen geht aus den entsprechenden Beschreibungen der Ziel-Parameter hervor.

6.1 Leistung

6.1.1 CPU-Zeit (CPU time)

Die CPU-Zeit eines Programms ist die Zeit, in der das Programm die CPU belegt, bzw. die nötig wäre, um es auf dem Rechner auszuführen, wenn keine anderen Programme zwischenzeitlich zu bearbeiten wären. Die CPU-Zeit zu optimieren ist das eigentliche Zeit-Ziel der Synthese. Sie u.a. bestimmt am Ende, wie lange ein Benutzer auf die Ausführung eines Programms auf dem entworfenen Rechner warten muß. Um dies zu erreichen, werden beim Entwurf verschiedene Unterziele angestrebt.

$$CpuZeit = AnzahlInstruktionen \times CPI' \times Zykluszeit$$

6.1.2 CPI'

Der CPI'-Wert gibt, wie der CPI-Wert, die durchschnittliche Anzahl der benötigten Zyklen für einen Befehl an, bezieht aber die tatsächliche, durchschnittlich benötigte Anzahl von Taktzyklen für die

Zugriffszeit von Speicherbefehlen mit ein.

$$CPI' = CPI_{ex} + \frac{AnzahlSpeicherzugriffe}{AnzahlInstruktionen} \times \varnothing Zugriffszeit$$

6.1.3 Durchschnittliche Zugriffszeit (average access time)

Die durchschnittliche Zugriffszeit gibt an, wie lange im Durchschnitt auf einen Speicherzugriff gewartet werden muß. Hier wird die echte Zeit gemessen und nicht die Anzahl der benötigte Zyklen, d.h. derselbe Speicher kann für verschiedene CPUs unterschiedliche Leistung zeigen: Liegt z.B. für Architektur A die Zykluszeit eben unter der durchschnittlichen Speicherzugriffszeit, muß auf einen Speicherzugriff zwei Zyklen gewartet werden, aber bei Architektur B, die vielleicht etwas langsamer ist, nur ein Zyklus. Dies kann sogar dazu führen, das am Ende Architektur B schneller ist!

$$\varnothing Zugriffszeit = Transport_{CPU-Cache1} + Zugriff_{Cache1} + Fehlrate_1 \times FehlZeit_1$$

Die durchschnittliche Zugriffszeit setzt sich also zusammen aus der Transportzeit zwischen CPU und dem 1.-Level-Cache, der Zugriffszeit für eine Treffer im 1.-Level-Cache und der Fehlschlagsrate multipliziert mit der dafür benötigten Verzögerungszeit.

6.1.4 Treffer/Fehlschlag

Ein Speicherzugriff heißt Treffer (hit), wenn sich das gewünschte Datum im angesprochenen Speichermodul befindet. Andernfalls muß das Datum aus einem darunterliegenden Speichermodul geholt werden, dies bezeichnet man als Fehlschlag (miss). Ein Fehlschlag beim Zugriff auf den Hauptspeicher nennt man Seiten- oder Segmentfehler (page/segmentation fault).

6.1.5 Treffer-Zeit (hit time)

Die Treffer-Zeit bezeichnet die durchschnittliche Zeit, die benötigt wird, um ein Datum, das sich im Cache befindet, zur Verfügung zu stellen.

- Zeit für einen Schreibzugriff:
  Hier ist die Zeit gemeint, die die CPU warten muß, um einen Schreibzugriff im Cache auszuführen. Dazu gehört auch das Warten auf das Schreibrecht. Das Aktualisieren des Hauptspeichers und gegebenenfalls anderer Caches läuft dann im Hintergrund ab, die CPU kann weiterarbeiten.
- Zeit für einen Lesezugriff:
  Hier ist die Zeit gemeint, die der Cache benötigt, um ein Datum zur Verfügung zu stellen.

Die Zugriffszeit auf einen Cache ist abhängig von der Größe und der Assoziativität des Caches.

6.1.6 Trefferrate / Fehlschlagsrate (hit rate / miss rate)

Die Trefferrate/Fehlschlagsrate (Fehlrate) bezieht sich auf die Treffer/Fehlschläge relativ zur Anzahl der Zugriffe:

$$TrefferRate = \frac{AnzahlTreffer}{AnzahlZugriffe} \qquad Fehlschl\dot{R}ate = \frac{AnzahlFehlschl}{AnzahlZugriffe}$$

6.1.7 Anzahl der Fehlschläge

Beim Cache teilt man die Fehlschläge nach drei verschiedenen Ursachen ein, die die Art des Fehlschlag bestimmen:

- Start-Fehlschläge (compulsory miss)

Diese Fehlschläge treten nach einem Prozeßwechsel (zwangläufig) auf, wenn zum ersten Mal (wieder) auf Daten des (re-)aktivierten Prozesses zugegriffen wird.

- Kapazitätsfehlschläge (capacity miss)
  Fehlschläge aufgrund mangelnder Kapazität treten auf, falls der Cache nicht groß genug ist, um alle benötigten Daten des Prozesses aufzunehmen. Dies ist nicht ungewöhnlich, da der Cache nicht genug Platz für alle Daten des Speichers bieten kann, weil er kleiner ist.
- Konflikt-Fehlschläge (conflict miss)
  Ein Konflikt tritt auf, falls auf Daten aus verschiedenen Zeilen zugegriffen werden muß, die sich aber nicht alle gleichzeitig im Cache befinden können, da sie beim Einlagern an die gleiche Stelle geschrieben würden, obwohl an anderer Stelle noch Platz vorhanden wäre. D.h. es kann immer nur ein Teil dieser beiden eingelagert werden (Dies läßt sich durch Erhöhen der Assoziativität im Cache verhindern, s.o.).

$$AnzahlFehlschl = StartFehlschl + KapazitaetsFehlschl + KonfliktFehlschl$$

6.1.8 Verzögerung durch Fehlschlag (miss penalty)

Bei einem Fehlschlag muß das Datum von dem darunterliegenden Speichermodul zur Verfügung gestellt und dann zum Speichermodul, in dem der Fehlschlag ausgelöst wurde, bzw. direkt zur aufrufenden Stelle transportiert werden.

$$Fehlzeit_1 = Transport_{Cache1-Cache2} + Zugriff_{Cache2} + Fehlrate_2 \times FehlZeit_2$$

$$Fehlzeit_2 = Transport_{Cache2-Hptsp} + Zugriff_{Hptsp} + Fehlrate_{Hptsp} \times FehlZeit_{Hptsp}$$

$$Fehlzeit_{Hptsp} = Transport_{Hptsp-Seksp} + Zugriff_{Sektsp}$$

Die Verzögerungszeit bei einem Fehlschlag im 1.-Level-Cache ($Fehlzeit_1$) setzt sich zusammen aus der Transportzeit zwischen 1.- und 2.-Level-Cache, der Zugriffszeit auf den 2.-Level-Cache und der Fehlschlagsrate des 2.-Level-Caches multipliziert mit dessen Verzögerungszeit ($Fehlzeit_2$). Letztere ist die Summe aus Transportzeit zwischen 2.-Level-Cache und Hauptspeicher, Zugriffszeit für einen Treffer im Hauptspeicher und dessen Fehlschlagsrate multipliziert mit der dadurch ausgelösten Verzögerungszeit ($Fehlzeit_{Hptsp}$). Letztere setzt sich wiederum aus der Transportzeit zwischen Hauptspeicher und Sekundärspeicher und der Zugriffszeit auf den Sekundärspeicher zusammen.

Ist kein 2.-Level-Cache vorhanden, so ist $Transport_{Cache1\text{-}Cache2}$, $Transport_{Cache2\text{-}Hptsp}$, $Zugriff_{Cache2}$ gleich ‘0’, $Fehlrate_2$ gleich ‘1’ und die $Transport_{Cache1\text{-}Hptsp}$ muß noch hinzuaddiert werden.

6.1.9 Zugriffszeit auf den Hauptspeicher

Die Zugriffszeit auf den Hauptspeicher ist abhängig von den verwendeten Speicherbausteinen, dem Zugriffsmodus und der Speicherverschränkung.

6.1.10 Zugriffszeit auf den Sekundärspeicher

Die Zugriffszeit auf den Sekundärspeicher wird einer Technologie-Datenbank entnommen (s.u.) und soll hier nicht detaillierter für Festplatte, Band, u.a. betrachtet werden.

6.1.11 Transportzeit

Die Transportzeit zwischen zwei Speichermodulen hängt von der Größe des zu transportierenden Datensatzes und der Bandbreite und der Länge der Verbindung ab. Die Bandbreite bezeichnet die Transportkapazität einer Verbindung, also wieviele Daten sich in welcher Zeit transportieren las-

sen.Hierbei spielt auch die Auslastung der Verbindung eine Rolle, da u.U. zunächst gewartet werden

$$Transport_{A-B} = Auslastung \times Datenmenge \times Bandbreite_{A-B}$$

$$Bandbreite_{A-B} = \frac{Menge}{Zeit}$$

muß, bis die Verbindung frei ist. Die Auslastung wird u.a. durch Prefetching erhöht (s.o.).

6.2 Kosten

6.2.1 Fläche

Die benötigte Fläche eines Speichermoduls ist besonders für einen on-chip-Cache interessant, da dieser noch auf den Chip des Prozessors Platz haben muß. Die zur Verfügung stehende Fläche ist also begrenzt, dies schränkt die Cache-Größe ein.

$$Flaeche_{Cache} = Konst \times \langle TagBits + OrgBits + Zeilengroesse \rangle \times AnzahlZeilen$$

$$TagBits = \frac{Adressbreite}{\lceil \log (CacheGroesse) \rceil} \times \log (Assoziativitaet)$$

Die Fläche eines Caches ist proportional zu Anzahl der Bits in einer Zeile multipliziert mit der Anzahl der Zeilen. Eine Zeile setzt sich zusammen aus Tagbits, wenigen Organisationsbits (ca. 2-16 Bits) und der Speicherkapazität oder Zeilengröße. Die Anzahl der Tagbits ist die Größe des Adreßraumes (real oder virtuell, je nach Adressierung des Caches) dividiert durch die Cachegröße und davon der Logarithmus, multipliziert mit der Assoziativität des Caches.

6.2.2 Preis

Der Preis ist vor allem für off-chip-Speichermodule von Belang.[1] Ein off-chip-Speichermodul kann aus mehreren verschiedenen Speicherbausteinen zusammengesetzt sein, entsprechend ist der Preis die Summe der Preise der verwendeten Bausteine. Er ist abhängig von der Zugriffszeit und der Größe des Bausteins. Hier gibt es verschiedene Varianten, die in einer Technologie-Datenbank aufgelistet sind.

7. Beziehungen zwischen den Parametern

Allgemein verwendete Literatur: [Alpe88, Baeh91, Baer80, Debl90, East90, Ever90, Fluc89,Goor89, Grant89, Grant91, Hama90, Haye88, Henn90, Hsie89, Huck89, Knie89, Koho80, Kola92, Lang89, Marw93, Muld91, Ober89, Paet91, Rhei92, Scho88, Siew82, Ston90, Swaa92, Tane84, Tava90, Unge89, Weck82, Weik93a/b].

Die Beziehungen zwischen den einzelnen Parametern sind hier nur qualitativ angegeben, weil sich die quantitativen Angaben in den verschiedenen Literaturquellen sehr stark unterscheiden, sofern sie überhaupt vorhanden sind.

Hier wurden hauptsächlich die Beziehungen zwischen den Entwurfsparametern untereinander und zwischen den Entwurfsparametern und den Einflußfaktoren beschrieben, da die Einflußfaktoren ja mehr oder weniger gegeben sind. Falls nicht, sind unter den Bemerkungen zu den einzelnen Einflußfaktoren einiger ihrer Beziehungen untereinander angegeben.

1. Für on-chip-Speichermodule spielt der Preis keine sehr große Rolle, da nur der verbliebene Platz auf dem Prozessor-Chip zur Verfügung steht. Hier lohnt es sich, möglichst viel Platz zur Verfügung zu stellen, da der zusätzliche Nutzen in der Regel die Kosten übersteigt. Weitere Chips können nicht verwendet werden, so daß dafür keine weiteren Kosten hinzukommen können.

Die Beziehungen zu den Entwurfsparametern sind für jede Art von Speichermodulen beschrieben und gelten natürlich dann für jedes dieser Module. Die beschriebenen Beziehungen der Cache-Parameter untereinander gelten beispielsweise für einen 1.-Level-Cache und für einen 2.-Level-Cache eines jeden Prozessors, außer es ist explizit angegeben, für welches Speichermodul die Beziehungen gelten.

Abbildung 14 gibt einen Überblick über das komplizierte Beziehungsgeflecht zwischen den Ent-

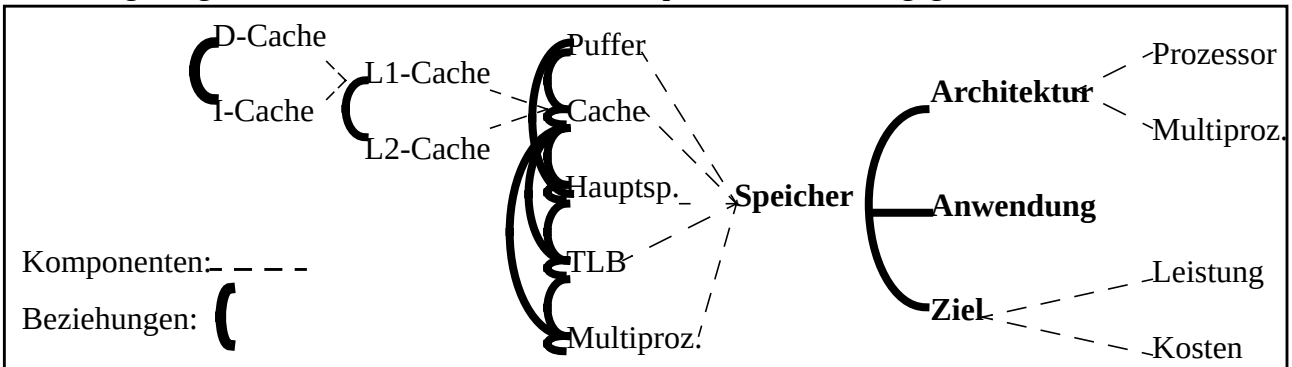

Abb. 14: Beziehungsgeflecht zwischen den Parametern

wurfs- (Speicher-), Einfluß- (Rechnerarchitektur- und Anwendungs-) und Ziel-Parametern.

7.1 Speicher

7.1.1 Puffer

- Größe - Zeilengröße
  Ein Puffer hat nur wenige Zeilen, d.h. seine Größe wird zum großen Teil durch seine Zeilengröße bestimmt.

7.1.2 Cache

Hier sind die Beziehungen der einzelnen Parameter eines Caches untereinander beschrieben. In Abbildung 15 sind die Möglichkeiten der Cache-Struktur-Parameter dargestellt. Abbildung 16 gibt

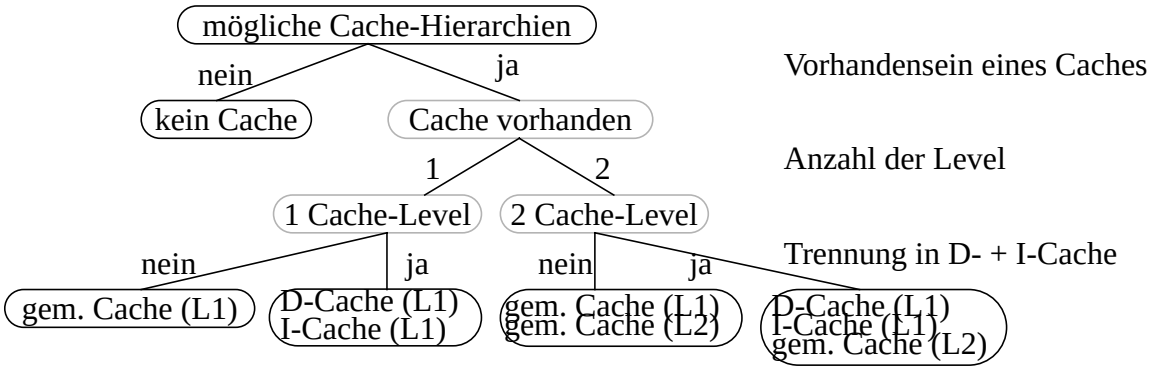

Abb. 15: Möglichkeiten der wichtigsten Cache-Struktur-Parameter

an, welche Parameter eines einzelnen Caches zueinander in Beziehung stehen.

- Anzahl der Level - Integration
  Gibt es nur einen Cache, sollte dieser on-chip sein, die Integration auf dem Chip des Hauptspeicher bringt nicht genügend Geschwindigkeitsvorteile.
- Levelnummer - Integration
  Der 2.-Level-Cache ist immer off-chip.
- Levelnummer - Inhalt

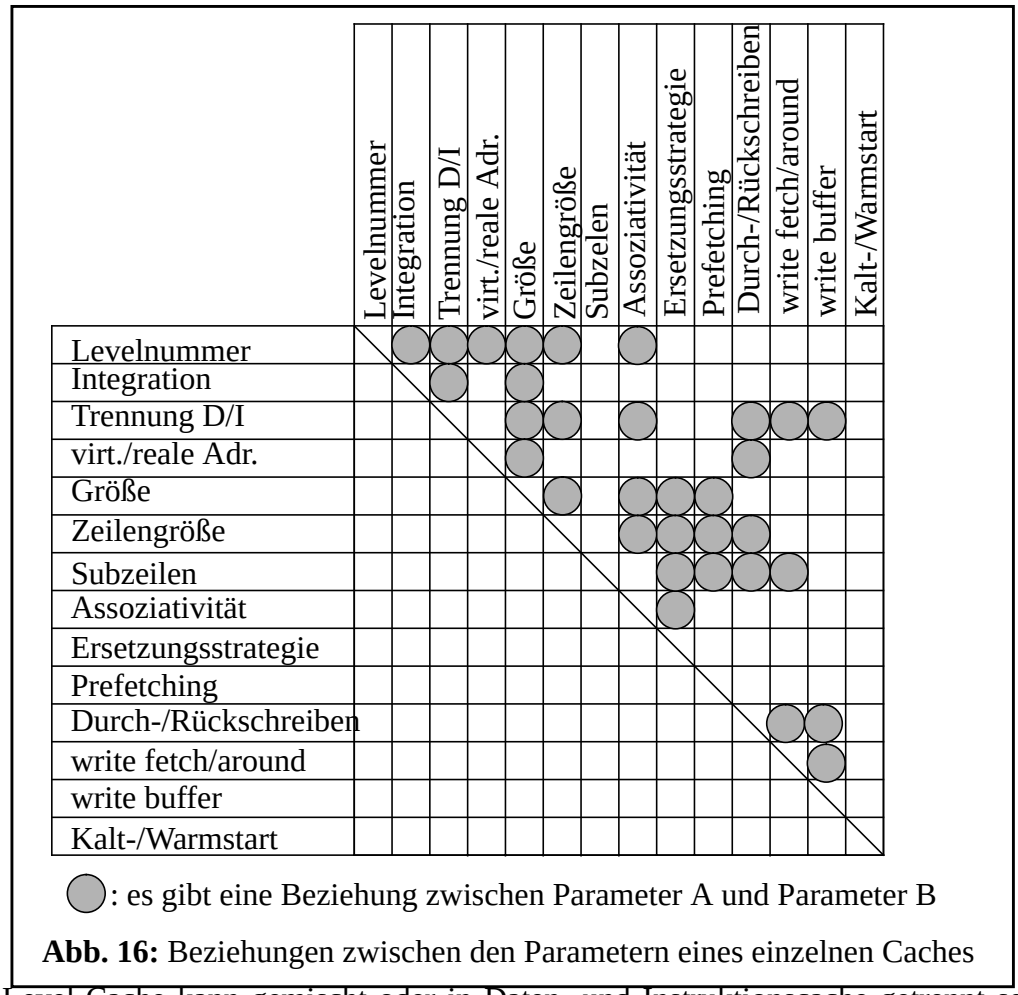

Abb. 16: Beziehungen zwischen den Parametern eines einzelnen Caches

Der 1.-Level-Cache kann gemischt oder in Daten- und Instruktionscache getrennt sein, der 2.-Level-Cache ist immer gemischt, da hier der Aufwand für eine Trennung nicht lohnt.

- Levelnummer - Adressierung
  Der 1.-Level-Cache kann sowohl virtuell als auch real adressiert sein, der 2.-Level-Cache wird real adressiert, weil der Geschwindigkeitsverlust für die Adreßübersetzung hier keine so große Auswirkungen hat.
- Levelnummer - Größe
  Der 1.-Level-Cache ist in der Regel viel kleiner (1-256 KByte) als der 2.-Level-Cache (256K-4MByte). Gibt es nur einen Cache, so ist er mittel groß.
- Levelnummer - Zeilengröße
  Die Zeile eines 1.-Level-Caches ist meist etwas kleiner (4-256Byte) als die des 2.-Level-Caches (32-256Byte).
- Levelnummer - Subzeilen
  Subzeilen lohnen sich nur für einen 1.-Level-Cache, da beim 2.-Level-Cache der Speicherbedarf für die zusätzlichen Adreßbits bei getrennten Caches nicht so kritisch ist.
- Levelnummer - Assoziativität
  Der 1.-Level-Cache hat in der Regel eine geringe Assoziativität (1-8), um einen schnellen Zugriff zu gewährleisten. Der 2.-Level-Cache kann voll-assoziativ sein, weil die Zugriffszeit durch den 1.-Level-Cache bestimmt wird, auf jeden Fall ist sie größer als 1.
- Levelnummer - Ersetzungsstrategie
  Beim 1.-Level-Cache kommen nur Hardware-Strategien in Frage, also Random, FIFO oder LRU. Beim 2.-Level-Cache wird Random bevorzugt.

- Integration - Größe
  Die Größe des on-chip-Caches ist durch den noch vorhandenen Platz auf dem Chip begrenzt.
- Inhalt - Zeilengröße
  Die Zeilengröße des Instruktionscaches ist kleiner (8-16 Worte) als die des Daten-Caches (16-32 Worte). Gibt es einen unifizierten Cache, beträgt die Zeilengröße 4-256 Worte.
- Inhalt - Assoziativität
  Die Assoziativität beim unifizierten- und beim Daten-Cache sollte relativ hoch sein, um Konflikt-Fehlschläge zu vermeiden. Beim Instruktionscache genügt oft eine geringe Assoziativität oder ein direct-mapped Cache, weil sich diese Zugriffe sehr sequentiell verhalten.
- Inhalt - Aktualisieren
  Beim Instruktionscache entfällt das Schreiben, außer bei selbstmodifizierenden Programmen.
- Inhalt - Ersetzungsstrategie
  Beim Instruktionscache kann FIFO ausreichen, wenn die Instruktionszugriffe sehr sequentiell sind.
- Inhalt - Ladestrategie
  Beim Instruktionscache ist ein Prefetching wegen des sequentiellen Zugriffsverhaltens angebracht.
- Adressierung - Größe - Start
  Bei einem großen Cache sind für einen unterbrochenen und neu gestarteten Prozeß evtl. noch gültige Daten im Cache, wenn dieser real oder virtuell mit Prozeßname adressiert und Warmstart gewählt wird.
- Adressierung - Aktualisieren
  Durchschreiben kommt nur für einen real adressierten Cache in Frage, weil ansonsten die Adreßübersetzung bei jedem Schreibzugriff zuviel Zeit kosten würde.
- Adressierung - Start
  Beim virtuell adressierten Cache wird meist Warmstart gewählt. Beim Warmstart ist eine Prozeßname notwendig, um die virtuellen Adressen eindeutig einer realen Adresse zuordnen zu können, da verschiedenen reale Adressen in verschiedenen Prozessen die gleiche virtuelle Adresse haben können.
- Größe
  Die Größe ist in der Regel eine 2-er Potenz.
- Größe - Integration
  Die Größe des on-chip-Caches ist durch den noch vorhandenen Platz auf dem Chip begrenzt.
- Größe - Zeilengröße
  Bei kleinen Caches sollte die Zeilengröße auch nur klein sein, da sonst zu wenig Zeilen zur Verfügung stehen, was zu häufigen Konflikt-Fehlschlägen führen würde.
- Größe - Assoziativität
  Bei einem großen Cache spielt die Assoziativität nicht eine so große Rolle wie bei kleinen Caches. Kleine Caches benötigen ein hohe Assoziativität. Die 2:1-Regel besagt, daß ein direct-mapped-Cache mit Größe 2*S eine größere Fehlschlagsrate hat, als ein zwei-Wege-Satz-assoziativer Cache der Größe S.
- Größe - Ersetzungsstrategie
  Bei großen Caches ist die Ersetzungsstrategie nicht so entscheidend, Random reicht meist aus. Bei kleinen Caches sollte LRU gewählt werden.
- Größe - Ladestrategie
  Bei kleine Caches führt Prefetching schnell dazu, daß noch benötigte Daten ausgelagert werden, deshalb kommt anfragegesteuertes Laden in Frage. Bei großen Caches kommt mehr Prefetching in

Frage.

- Größe - Start
  Bei großen Cache kann durch Warmstart das noch Vorhandensein von alten Daten beim Reaktivieren eines Prozesses ausgenutzt werden.
- Zeilengröße
  Die Zeilengröße ist in der Regel eine 2-er Potenz.
- Zeilengröße - Subzeilen
  Mit Subzeilen sind größere Zeilen möglich, da die meisten Aktionen nicht auf die gesamte Zeile ausgeführt werden müssen.
- Zeilengröße - Assoziativität
  Bei großen Zeilen ist mehr Assoziativität nötig, weil ansonsten zu viele Fehlschläge auftreten (siehe auch Größe - Assoziativität).
- Zeilengröße - Ersetzungsstrategie
  Bei großen Zeilen ist die Ersetzungsstrategie wichtiger, also LRU, als bei kleinen, hier genügt eher Random (siehe auch Größe - Ersetzungsstrategie).
- Zeilengröße - Ladestrategie
  Prefetching lohnt sich nur bei kleinen Zeilen, da sonst das Einlagern zu lange dauert. Eine Zeilengröße 2*L ohne Prefetching bringt die gleichen Daten in den Cache wie eine Zeilengröße L mit Prefetching, aber der Prozeß kann nach den ersten L Daten schon weiterarbeiten.
- Zeilengröße - Aktualisieren
  Bei großen Zeilen kommt nur Rückschreiben in Frage, weil ein Schreibzugriff sonst zu lange dauern würde.
- Subzeilen - Aktualisieren
  Mit Subzeilen ist das Durchschreiben eher möglich, weil nicht die ganze Zeile im Hauptspeicher aktualisiert werden muß.
- Subzeilen - Prefetching
  Mit einer Aufteilung der Zeilen in Subzeilen ist ein vermehrtes Prefetching möglich, da auch nur Teile einer Zeile vorab in den Cache geholt werden können, d.h. die Verbindung zu Hauptspeicher wird nicht soviel und nicht für so lange Zeit blockiert.
- Assoziativität
  Die Assoziativität ist meistens eine 2-er Potenz.
- Assoziativität - Ersetzungsstrategie
  Bei direct-mapped Caches ist keine Ersetzungsstrategie nötig, weil das zu ersetzende Datum mit der Adresse ausgewählt ist. Bei kleiner Assoziativität spielt die Ersetzungsstrategie eine größere Rolle, also LRU, als bei großer Assoziativität, hier genügt machmal Random.
- Aktualisieren - Schreibstrategie
  Bei Durchschreiben wird meist write-around gewählt, weil sowieso auf den Hauptspeicher zugegriffen wird, außer wenn das geschriebene Datum danach mehrmals gelesen wird. Für Rückschreiben kommt nur fetch-on-write in Frage, denn das Datum muß ja in den Cache geholt werden. Bei write-around wird Durchschreiben gewählt, weil direkt ein Schreiben auf dem Hauptspeicher stattfindet, und somit ein Rückschreiben überflüssig ist.
- Aktualisieren - Schreibpuffer
  Beim Durchschreiben ist in der Regel ein Schreibpuffer nötig, weil ansonsten ein Schreibzugriff zu zeitaufwendig wäre.
- Schreibstrategie - Schreibpuffer

Für write-around wird meist ein Schreibpuffer verwendet.

7.1.3 Hauptspeicher

- Größe - Speicherverwaltung
  Bei der Seitenadressierung (mit Segmentierung) kommt es zu einer internen und bei der Segmentadressierung zu einer externen Fragmentierung. Das vermindert die nutzbare Speicherkapazität. Die interne Fragmentierung ist meist nicht sehr groß, die externe Fragmentierung läßt sich durch Verschiebungen beheben.
- Speicherverwaltung - Plazierung
  Bei Seitenadressierung (mit oder ohne Segmentierung) ist keine Plazierungsstrategie nötig, weil jede Seite an jede Stelle plaziert werden kann.
- Speicherverwaltung - Ersetzungsstrategie
  Bei der reinen Segmentierung ist keine Ersetzungsstrategie nötig, weil bei Prozeßwechsel immer alle Segmente eines Prozesses ausgelagert werden müssen.
- Verteilungsstrategie - Puffer
  SIM ohne Verschiebung bringt wenig Zeitvorteil, wenn keine Puffer vor den einzelnen Modulen vorgesehen sind, da es leicht zu Kollisionen (aufeinander folgende Zugriffe gehen an das gleiche Modul) kommen kann, so daß die Zugriffe doch nacheinander abgearbeitet werden müssen.
- Anzahl der Module - Verteilungsstrategie
  Bei Verteilungsstrategie PRIM muß die Anzahl der Module prim sein.

7.1.4 TLB

- alle Parameter
  Es gelten natürlich alle Beziehungen zwischen den Parametern eines Caches, da der TLB eine Art Cache ist. Einzelne Parameter sind aufgrund der Besonderheiten eines TLBs bereits fest vorgegeben (s.o.).
- Größe - Zeilengröße
  Die Größe wird hier durch Anzahl der Zeilen festgelegt, d.h. sie beträgt Anzahl der Zeilen multipliziert mit Zeilengröße.
- Größe - Assoziativität
  Sind komplette Seitentabellen im TLB enthalten, wird keine Assoziativität benötigt, sie ist eins. Sonst ist sie sehr groß, weil der TLB relativ klein ist.
- Größe - Start
  Sind komplette Seitentabellen im TLB enthalten, wird Warmstart gewählt, ansonsten Kaltstart.

7.1.5 Multiprozessor (Speicher)

- Mögliche Kombination einiger Parameter
  Es sind nicht alle Kombinationen von Parametern sinnvoll. Die folgende Graphik gibt die möglichen Kombinationen einiger Architektur- und Speicher-Parameter von Multiprozessor-Systemen wieder (Abb. 17):
- Lokalität der Speicher - Uniformität
  Will man bei gemeinsam genutzten Speichern, daß der Zugriff auf alle verfügbaren Daten für jeden Prozessor gleichschnell geht, muß der Zugriffsweg auch der gleiche sein, d.h. gemeinsam genutzter Speicher (SM). Das garantiert bei Verwendung eines Busses auch eine sequentielle Konsistenz.
- Lokalität - Cache-Kohärenz-Protokoll
  Bei verteilten Speichern geht der Datenaustausch über Nachrichten (message passing), es ist kein

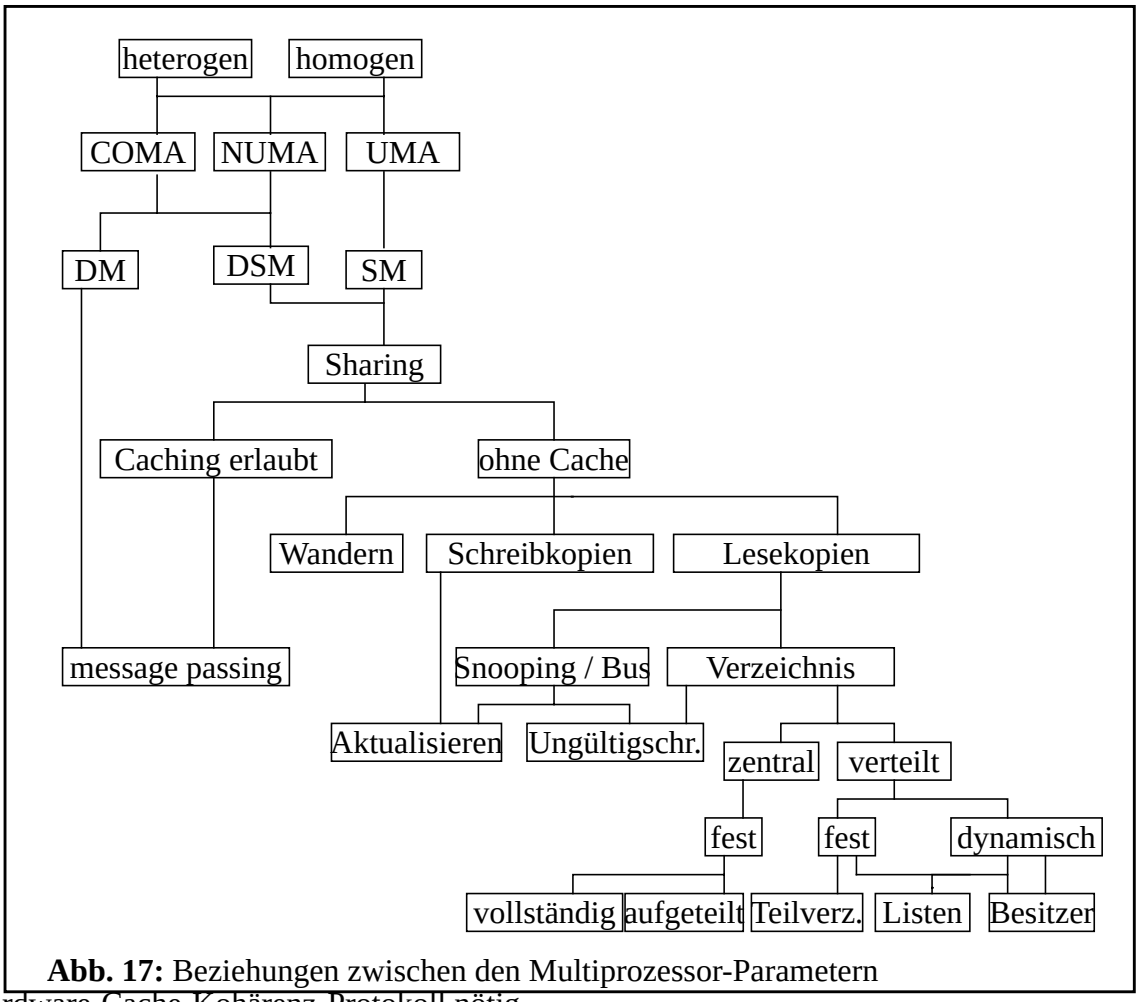

Abb. 17: Beziehungen zwischen den Multiprozessor-Parametern

Hardware-Cache-Kohärenz-Protokoll nötig.

- Cache-Kopien - Cache-Kohärenz-Protokoll
  - Gibt es jeweils nur eine Kopie eines Datums in einem der Caches und “wandert” diese jeweils zu dem Prozessor, der sie ändern will, ist kein Cache-Kohärenz-Protokoll nötig, sondern nur eine Koordination zwischen diesem Cache und Zugriffen von außen (DMA).
  - Lesekopien können mit Snooping- oder Verzeichnis-Methoden verwaltet werden.
  - Schreibkopien erfordern das Snooping-Protokoll
- Cache-Kopien - Aktualisieren der Caches
  - Schreibkopien bedingen ein Aktualisieren (write update), da ansonsten beim ersten Verändern einer der Kopien alle anderen Kopien ungültig würden.
  - Mit Invalidate-Nachrichten sind mehrere Lesekopien aber nur eine schreibbare Version erlaubt, d.h. es gibt nur Lesekopien.
- Cache-Kohärenz-Protokoll - Aktualisieren
  Verzeichnis-Protokolle verwenden Ungültigschreiben, Snooping-Protokolle unterstützen das Aktualisieren der Caches.
- Verzeichnis-Verteilung -Aktualisieren
  Bei verteilten Verzeichnissen sind Broadcast-Nachrichten die schnellste Möglichkeiten, ein Datum und seinen Besitzer zu finden. Speziell bei der dynamischen Verteilung ist von Zeit zu Zeit ein Aktualisieren der Caches nötig.
- Verzeichnis-Verteilung - Dynamik der Verzeichnis-Verteilung
  - Ein zentrales Verzeichnis wird nicht dynamisch von verschiedenen Prozessoren verwaltet.
  - Beim verteilt aufgeteilten Verzeichnis-Protokoll sind Teile des Verzeichnisses fest auf verschie-

dene Prozessoren verteilt.
  - Listen- und Besitzer-basierte Protokolle haben ein festes Verzeichnis, das die Verweise auf die ersten Kopien bzw. die Besitzer enthält. Weitere Kopien werden dann dynamisch verwaltet.
  - Besitzer-basierte Protokolle können auch vollständig dynamisch verteilt sein. Dann ist entweder eine Broadcast-Nachricht (Bus) nötig, um den Besitzer zu finden (schlechte Leistung) oder jeder Prozessor hat ein Verzeichnis über die Besitzer der Daten. Beim letzteren Verfahren werden nicht immer alle Verzeichnisse aktualisiert, falls sich ein Besitzer ändert, sondern der alte Besitzer hat einen Verweis auf den neuen Besitzer. Aber auch hier ist von Zeit zu Zeit eine Broadcast-Nachricht nötig, damit die Verweisketten nicht zu lang werden.

- Vollständigkeit der Verzeichnisse - Cache-Kopien
  Ein Teilverzeichnis erlaubt mehr Kopien von Daten bei moderater Verzeichnis-Größe, als ein vollständiges Verzeichnis.

7.1.6 Puffer - Cache

- Puffer - Vorhandensein eines Schreibpuffers
  Das Vorhandensein eines Schreibpuffers im Cache entscheidet, ob ein allgemeiner Puffer vorhanden ist.
- Zeilengröße - Zeilengröße
  Wird der Puffer als Schreibpuffer oder Instruktionspuffer beim Cache verwendet, stimmen die Zeilengrößen beider Speichermodule überein.
- Zugriffsart - Instruktionspuffer
  Wird der Puffer als Instruktionspuffer für den Cache verwendet, bestimmt die Art des Instruktionspuffers die Zugriffsart des Puffers.

7.1.7 Puffer - Hauptspeicher

- Größe - Puffergröße vor den Bänken
  Die Puffergröße vor den Bänken bei einer Speicherverschränkung des Hauptspeichers entscheidet, ob vor jeder Bank ein Puffer installiert ist und wie groß diese Puffer sind.
- Größe - Verteilstrategien
  Bessere Verteilungsstrategien können bereits mit Hilfe von kleinen Puffern vor den Bänken erheblich verbessert werden .
- Zugriffsart - Speicherverschränkung
  Puffer von den Bänken werden als Schlange verwendet.

7.1.8 Cache - Cache

a) Instruktionscache - Daten-Cache

- Größe - Größe
  Der Instruktionscache kann kleiner sein als der Daten-Cache, weil die Zugriffe i.a. sequentieller sind.
- Assoziativität - Assoziativität
  Der Daten-Cache ist stärker assoziativ als der Instruktionscache, weil beim Instruktionscache das Zugriffsverhalten i.a. sequentieller sind.

b) 1.-Level-Cache - 2.-Level-Cache

- Anzahl der Level - Aktualisieren
  Bei 2 Cache-Leveln benutzt meist der 1.-Level-Cache Durchschreiben und der 2.-Level-Cache Rückschreiben.

- Anzahl der Level - Adressierung
  Bei 2 Cache-Leveln ist der 1.-Level-Cache meist virtuell und der 2.-Level-Cache real.
- Größe - Größe
  Der 2.-Level-Cache muß, falls vorhanden, erheblich größer sein als der 1.-Level-Cache, weil er sonst eine zu hohe Fehlschlagsrate hätte.
- Assoziativität - Assoziativität
  Der 2.-Level-Cache kann stärker assoziativ sein als der 1.-Level-Cache, weil der dadurch bedingte Zeitverlust da nicht so kritisch ist, die Zugriffszeit wird durch den 1.-Level-Cache bestimmt.
- Ersetzungsstrategie - Ersetzungsstrategie
  Auch hier gilt, daß diese Aufgabe beim 2.-Level-Cache nicht so zeitkritisch ist wie beim 1.-Level-Cache, d.h. man ist nicht auf Random oder LRU eingeschränkt, sondern kann aufwendigere Algorithmen wie das Working-Set-Verfahren o.ä. in Betracht ziehen.
- Laden - Laden
  Beim 2.-Level-Cache kann das Prefetching intensiver betrieben werden als beim 1.-Level-Cache, da sich die dafür benötigte Zeit nicht so gravierend auswirkt.

7.1.9 Cache - Hauptspeicher

- Cache-Größe - Adressierung - Assoziativität - Seitengröße des Hauptspeichers
  Ein real adressierter 1.-Level-Cache sollte nicht größer sein als die Seitengröße des Hauptspeichers multipliziert mit der Assoziativität. In diesem Fall kann die Übersetzung des virtuellen Teils der Adresse und der Zugriff auf den entsprechenden Satz mit Hilfe der Satznummer im realen Teil der Adresse parallel verlaufen. Anschließend müssen die Tags der Zeilen im adressierten Satz mit dem Tag aus der angegebenen Adresse verglichen werden.
- Aktualisieren - Speicherverschränkung
  Durchschreiben wird durch eine ausreichende Speicherverschränkung unterstützt, die die häufigen Schreibzugriffe schneller abarbeitet.
- Adressierung - DMA
  DMA ist mit real adressierten Caches schneller. Hält der virtuell adressierten Cache das aktuelle Datum, muß die reale Adresse des DMA-Zugriffs zuerst in die virtuelle Adresse übersetzt werden, um das Datum im virtuell adressierten Cache zu finden.
- Zeilen - Blockzugriffsverfahren
  Blockzugriffsverfahren kommen z.B. beim Ein- und Auslagern von Zeilen in den Cache zum Einsatz.

7.1.10 Cache - TLB

- Adressierung - Lage des TLB
  Beim real bzw. virtuell adressierten Cache liegt der TLB zwischen Prozessor und Cache bzw. Cache und Hauptspeicher.

7.1.11 Cache - Multiprozessor (Speicher)

- Größe vom 1.-Level-Cache - Multilevel-Cache
  Bei kleinem 1.-Level-Cache ist die private Organisation am günstigsten.
- Assoziativität vom 2.-Level-Cache - Multilevel-Cache
  Wird der 2.-Level-Cache von mehreren Prozessoren gemeinsam benutzt (multiport oder bus shared), muß er eine hohe Assoziativität haben, damit er noch die Daten aus allen 1.-Level-Caches enthält (inclusion property).

- Aktualisieren der Caches - Aktualisieren (des Hauptspeichers)
  Aktualisieren (update) ist nur sinnvoll, wenn die einzelnen Caches Durchschreiben benutzen, beim Ungültigschreiben wird eher Rückschreiben bevorzugt.

7.1.12 Hauptspeicher - TLB

- Speicherverwaltung - Größe und Assoziativität
  Bei der Identität ist kein TLB vorhanden. Bei mehrstufiger Adressierung (z.B. 3-stufige Segmentierung mit Seitenadressierung) speichert der TLB zu einer virtuellen direkt die reale, nicht aber die lineare Adresse. Besonders bei mehrstufiger Adressierung sollte der TLB realtiv groß und vollassoziativ sein, um selten die Adreßübersetzung über die verschiedenen Stufen durchführen zu müssen
- Größe - Zeilengröße
  Die Länge der realen Adressen des Hauptspeicher bestimmen die Zeilengröße des TLBs (s.o.).

7.1.13 Hauptspeicher - Multiprozessor (Speicher)

Diese Beziehungen sind bei den Speicher-Parametern eines Multiprozessor-Systems beschrieben.

7.1.14 TLB - Multiprozessor (Speicher)

Ein TLB ist einem einzelnen Prozessor zugeordnet und wird entsprechend aus dessen lokaler Sicht entworfen.

7.2 Speicher - Architektur

7.2.1 Cache - Prozessor

- Trennung - Pipelining
  Die Trennung eines Caches unterstützt das Pipelining, weil Instruktionen und Daten parallel geholt werden können.
- Trennung - Bandbreite
  Die Bandbreite wird erhöht, wenn Daten- und Instruktionscache getrennt sind.
- Trennung - DMA
  Eine Trennung des Caches macht eine Ein-/Ausgabe noch komplizierter
- Adressierung - Prozeßwechsel
  Ein real adressierter Cache bleibt bei Prozeßwechsel gültig, ein virtuell adressierter Cache nicht, da muß zusätzlich der Prozeßname gespeichert werden.
- Adressierung - DMA
  Ein virtueller Cache macht eine Ein-/Ausgabe langsamer, wenn das akutelle Datum nur im Cache vorhanden ist, weil die Adressen erst in virtuelle übersetzt werden müssen.
- Cache-Größe - DMA
  Eine Ein-/Ausgabe durch den Cache führt bei kleinen Caches zu größeren Fehlschlagsraten, weil diese nicht mehr voll für die Prozesse genutzt werden können, sondern auch die Ein-/Ausgabe bedienen müssen.
- Zeilengröße - Prozeßwechsel
  Bei häufigem Prozeßwechsel sind bei großen Zeilen die Daten schneller wieder nachgeladen.
- Zeilengröße - $\text{Busbreite}_{\text{CPU-Cache}}$
  Es gilt Zeilengröße = $\text{Busbreite}_{\text{CPU-Cache}}$ * a, wobei a eine 2-er-Potenz ist.
- Zeilengröße - $\text{Busbreite}_{\text{Cache-Hauptspeicher}}$

Es gilt Zeilengröße * a = $\text{Busbreite}_{\text{Cache-Hauptspeicher}}$ wobei a eine 2-er-Potenz ist.

- Aktualisierung - Bandbreite CPU-Cache
  Für Durchschreiben ist eine große Bandbreite nötig, um die relativ häufigen Schreibzugriffe auf den Hauptspeicher schnell bedienen zu können.
- Aktualisierung - DMA
  Bei Rückschreiben z.B. kann es zu Konsistenzproblemen kommen, wenn Ein-/Ausgabe-Zugriffe direkt über den Hauptspeicher laufen (DMA), weil der alte Daten enthalten kann.
- Start - Prozeßwechselfrequenz
  Kaltstart eignet sich bei seltenen Prozeßwechseln, sonst, besonders beim großen Caches, ist Warmstart besser.

7.2.2 Cache - Multiprozessor

- Vorhandensein eines Cache - Konsistenz
  Das Problem der Datenkonsistenz ist mit Cache komplizierter.
- Trennung - Konsistenz
  Durch die Trennung des Caches kann es auch bei nicht selbstmodifizierenden Programmen zu Konsistenzproblemen kommen.
- Zeilengröße - Schein-Sharing
  Je größer die Zeilen desto mehr kommt es zu Schein-Sharing.
- Aktualisieren - Multiprozessoren
  Bei Multiprozessoren ist wegen der Konsistenzprobleme Durchschreiben einfacher als Rückschreiben, aber meist wird Rückschreiben gewählt, weil sonst das Verbindungsnetz schon bei wenigen Prozessoren überlastet wäre.

7.2.3 Hauptspeicher - Prozessor

- Speicherverwaltung - Prozessor-Architektur
  Die Identität wird nur bei einfachen PCs oder sehr alten Rechnern verwendet. Der Mehrprozeßbetrieb ist umständlich. Beim Binden und Laden sind Adreßmodifikation nötig.
- Größe - Adreßbreite
  Der virtuelle Adreßraum kann sehr viel größer als der reale Hauptspeicher sein, der reale Adreßraum nicht.
- Ladestrategie - Prozeßwechsel
  Bei Prozeßwechsel können, wenn die Prozeßwechselfrequenz nicht sehr hoch ist, Daten vorab in den Hauptspeicher eingelagert werden, bei laufendem Prozeß macht das wenig Sinn.
- Speicherverschränkung - Pipelining
  Speicherverschränkung unterstützt das Pipelining, da mehrere Zugriffe parallel abgearbeitet werden können.
- Anzahl Ports - DMA
  Für DMA-fähige Speicher sind zusätzliche Ports sinnvoll.

7.2.4 Hauptspeicher - Multiprozessor (Architektur)

- Speicherverwaltung - Sharing
  Sharing von Daten ist am einfachsten über globale Tabellen möglich.
- Multiprozessoren mit Daten-Sharing führen zu zunehmend irregulärem Zugriffsverhalten, das muß bei der Wahl der Verteilungstrategie bei der Speicherverschränkung berücksichtigt werden.

- erweiterter Hauptspeicher - gemeinsame Speicher
  Multiprozessoren können nicht nur den Hauptspeicher der jeweils anderen Prozessoren benutzen (remote), sondern auch gemeinsam einen erweiterten Hauptspeicher.

7.2.5 TLB - Prozessor

- Größe - virtueller Adreßraum
  Komplette Seitentabellen von einem oder gar mehrerem Prozeß können nur im TLB gehalten werden, falls der virtuelle Adreßraum genügend klein ist.
- Zeilengröße - Adreßraum
  Die Zeilengröße des TLB wird durch den virtuellen (Tag-Größe) und realen Adreßraum bestimmt (s.o.).

7.2.6 TLB - Multiprozessor (Architektur)

Ein TLB ist einem einzelnen Prozessor zugeordnet und wird entsprechend aus dessen lokaler Sicht entworfen.

7.2.7 Multiprozessor (Speicher) - Prozessor

Die Speicher-Parameter von Multiprozessor-Systemen stehen mit den Parametern der Prozessoren nur indirekt über die Architektur-Parameter von Multiprozessor-Systemen in Beziehung und sind entsprechend dort beschrieben.

7.2.8 Multiprozessor (Speicher) - Multiprozessor (Architektur)

- Lokalität der Speicher - Konsistenzmodell
  Bei verteilten Speichern und Datenaustausch über Nachrichten (message passing) gibt es keine Konsistenzprobleme.
- Lokalität - Sharing
  Beim Daten-Sharing sind (verteilt) gemeinsam benutzte Speicher einfach zu handhaben, weil über einen einheitlichen Adreßraum zugegriffen wird.
- Multilevel-Caches - Clusterung
  Es können nur wenige Prozessoren einen 2.-Level-Caches gemeinsam benutzten ($\leq$4 für multiport shared, $\leq$ 24 für bus shared).
- Cache-Kopien - Konsistenzmodell
  Eine Möglichkeit, daßs Konsistenzproblem bei gemeinsam benutzten Speicher zu verhindern, ist das Zwischenspeichern von gemeinsam benutzten Daten in Caches zu verbieten.
- Cache-Kohärenz-Protokoll - Art der Verbindung
  - Snooping-Protokolle erfordern einen Bus.
  - Verzeichnis-Protokolle können bei Bussen und allgemeinen Netzwerken verwendet werden, für allgemeine Netzwerke ist das Verzeichnis-Protokoll zu verwenden.
- Verzeichnis-Verteilung - Anzahl Prozessoren
  Ein zentrales Verzeichnis ist nur für kleinen Multiprozessor-Systeme geeignet.
- Verzeichnis-Verteilung - Sharing
  Ein zentrales Verzeichnis ist nur für wenig Sharing geeignet.
- Cache-Kohärenz-Protokoll - Skalierbarkeit
  Reine Snooping-Protokolle sind nicht skalierbar.
- Vollständigkeit der Verzeichnisse - Skalierbarkeit
  Ein vollständiges zentrales Verzeichnis ist für skalierbare Multiprozessor-Systeme ungeeignet, da

der Prozessor mit diesem Verzeichnis schnell überlastet wäre.

- Vollständigkeit der Verzeichnisse - Sharing
  Teilverzeichnisse sind nur für wenig Sharing geeignet.

7.3 Speicher - Anwendung

7.3.1 Cache - Anwendung

- Vorhandensein eines Cache - Lokalität
  Das Anwendungsprogram muß eine gewisse Lokalität aufweisen, damit beim mehrfachen Zugriff auf die gleichen Daten diese schneller zur Verfügung stehen, ansonsten wären die Kosten eines Caches nicht gerechtfertigt.
- Instruktionspuffer - Lokalität
  Verhalten sich die Instruktionszugriffe sehr lokal, kann ein Instruktionspuffer statt eines Instruktionscaches gewählt werden, meist gibt es mehrere “Häufungspunkte”, d.h. es müßten mehrere Puffer da sein.
- Größe - Arbeitsmenge
  Der Cache sollte so groß gewählt werden, daß er die Arbeitsmenge eines üblichen Progammlaufs aufnehmen kann.
- Zeilengröße - Sequentialität
  Bei sequentiellem Programmverhalten sind große Zeilen günstig.
- Zeilengröße - Lokalität
  Für kleine Zeilen spricht die zeitliche Lokalität von Programmen (temporal locality), d.h. einmal benutzte und eingelagerte Daten werden in kurzer Zeit wieder benötigt, also sollte immer möglichst wenig ausgelagert werden. Dagegen spricht die örtliche Lokalität von Programmen (spacial locality), d.h. wird ein Datum benutzt, wird als nächstes meist ein Datum in unmittelbarer Nähe benutzt, also sollten immer möglichst viele Daten aus der näheren Umgebung mit eingelagert werden.
- Assoziativität - Sequentialität
  Bei sehr sequentiellem Programmverhalten ist ein direct-mapped-Cache ausreichend.
- Ersetzungsstrategie - Sequentialität
  Bei sehr streng (umgekehrt) sequentiellen Zugriffverhalten eines Programms kommt auch FIFO (LIFO) als Ersetzungsstrategie in Frage, i.a. aber nicht.
- Aktualisierung - Lese-/Schreib-Verhältnis
  Aktualisierungsmechanismen sind sehr vom Zugriffsverhalten der Programme abhängig. Durchschreiben ist nur bei seltenen Schreibzugriffen möglich, ansonsten wird Rückschreiben gewählt.
- Schreibstrategie - Lese-/Schreib-Verhältnis
  Fetch-on-write ist nur sinnvoll, weniger als 1.3 Lese-/Schreibzugriffe vorm Auslagern stattfinden, ansonsten wähle man write-around.

7.3.2 Hauptspeicher - Anwendung

- Größe - Arbeitsmenge
  Der Hauptspeicher sollte groß genug sein, daß während der Prozeßlaufzeit nicht auf den Sekundärspeicher zugegriffen werden muß, d.h. die gesamte Arbeitsmenge sollte im Hauptspeicher gespeichert sein (T gleich Prozeßlaufzeit), weil der Zugriff auf Sekundärspeicher erheblich langsamer ist.
- Speicherverwaltung - Zugriffsverhalten
  Wird auf eine Datei zugegriffen, kann bei der globalen Segmentierung dies wie ein Segmentfehler

behandelt werden, das ist einfacher.

- Speicherverwaltung - Anwendungsgebiet
  Paging ohne automatisches Einlagern ist nur für ganz spezielle Anwendungen geeignet (z.B. Echtzeit).
- Ersetzungsstrategie - Zugriffsverhalten
  Random eignet sich für irreguläres und FIFO für sehr sequentielles Zugriffsverhalten.
- Speicherverschränkung - Zugriffsverhalten
  - Sind die Schrittweiten der Zugriffssequenzen bekannt, kann eine geeignete Verteilungsstrategie gewählt werden.
  - Einfaches SIM ist für sehr sequentielles Zugriffsverhalten sehr gut, wenn die Schrittweite prim zur Anzahl der Module ist. Es kann aber für andere Schrittweiten auch sehr schlecht sein.
  - PRIM mit Matrizenmultiplikation liefert für fast alle Schrittweiten gute Ergebnisse, wenn Puffer vor den Modulen installiert sind. Allerdings ist die Adreßumsetzung zeitintensiv. Das Finden des richtigen Polynoms ist aufwendig.
  - XOR-basiertes PRIM ist für fast alle Schrittweiten geeignet.
- erweiterter Speicher - Arbeitsmenge
  Ein erweiterter Speicher ist für Anwendungen mit sehr großen Datenstrukturen sinnvoll.

7.3.3 TLB - Anwendung

- Größe - Lokalität
  Zeigen die Anwendungsprogramme / Zugriffssequenzen ein sehr lokales Zugriffsverhalten bzw. benötigen sie wenig Speicher, können im TLB vollständige Seitentabellen von einem oder sogar mehreren Programmen gehalten werden. Das ist besonders bei häufigem Prozeßwechsel günstig.
- Größe - Zeitanforderung
  Bei sehr harten Zeitanforderungen der Anwendungen sollte der TLB möglichst groß sein.

7.3.4 Multiprozessor (Speicher) - Anwendung

Viele Beziehungen zwischen Anwendung und Speicherparametern von Multiprozessor-Systemen gehen bereits über die Architektur-Parameter von Multiprozessor-Systemenein , die entsprechend der Anwendung entworfen worden sind. Diese ihrerseits beeinflussen die Speicherparameter von Multiprozessor-Systemen (s.o.).

- Lokalität der Speicher - Grad des Sharing
  Werden wenigen / einigen / vielen Daten gemeinsam benutzt, eignen sich verteilte / DSM / gemeinsam benutzte Speicher.
- Lokalität der Speicher - Anzahl Prozessoren, die Daten gemeinsam benutzen
  Gemeinsam benutzte Speicher eignen sich nur für wenige Prozessoren, die Daten gemeinsam benutzen.
- Uniformität der Speicherzugriffe - Zeitanforderung
  Erfordert die Anwendung eine einheitliche Zeitverzögerung beim Speicherzugriff oder gibt es Obergrenzen, müssen NUMA-Systeme verwendet werden.
- Granularität - Sharing
  Sind die Größen der gemeinsam benutzten Daten sehr unterschiedlich groß, kann es sinnvoll sein, die Granularität dynamisch zu halten. In der Regel bedeutet dies jedoch einen zu großen Verwaltungsaufwand.
- Software/Hardware-Methoden - Anwendungsbreite
  Nur für sehr wenige ganz bestimmte Anwendungen werden Software-Methoden bei den Cache-

Kohärenz-Protokollen verwendet, in der Regel sind sie zu langsam.

- Aktualisieren anderer Caches - Verhältnis von Lese- und Schreibzugriffen
  Wird selten von mehreren Prozessoren auf gemeinsam benutzte Daten schriebend zugegriffen, eigenen sich Broadcast-Nachrichten. Wird häufig geschrieben, kann nur Ungültigschreiben verwendet werden, weil ansonsten das Verbindungssystem schnell überlastet wäre.
- Cache-Kohärenz-Protokoll - Sharing
  - Benutzen relativ viele Prozessoren in einem kleinen Multiprozessor-System Daten gemeinsam, kann mit dem Snooping-Protokoll schneller aktualisiert werden.
  - zentrale Verzeichnisse eignen sich nur für sehr wenig Sharing, weil der entsprechende Prozessor schnell überlastet ist.
  - Vollständigen Verzeichnissen eignen sich nur, wenn die Anzahl der Prozessoren, die gleichzeitig Daten gemeinsam benutzen, stark begrenzt ist.

7.4 Speicher - Ziel

7.4.1 Puffer - Leistung

- Vorhandensein - Zugriffszeit
  Kleine Puffer an bestimmten Stellen (Cache, Hauptspeicher) verringert in der Regel die Zugriffszeit (s.o.). Die Zugriffszeit kann geringfügig erhöht werden, wenn im Puffer nach einem nicht vorhandenen Datum gesucht wird.
- Vorhandensein - Fehlschlagsrate
  Ein Prefetch-Puffer verringert Kapazitäts- und Start-Fehlschlagsrate, eine victim-Cache veringert die Konflikt-Fehlschlagsrate.

7.4.2 Puffer - Kosten

- Vorhandensein - Fläche bzw. Preis
  Ein on-chip bzw. off-chip Puffer erhöht die Fläche bzw. den Preis, abhängig von seiner Größe.

7.4.3 Cache - Leistung

- Vorhandensein eines Caches - Zugriffszeit
  Bei einem Fehlschlag dauert der Zugriff mit Cache länger, als wenn direkt auf den Hauptspeicher zugegriffen würden, weil das Ein-, Auslagern und Konsistenz-Mechanismen wegfielen. Mit dem Einsatz eines Caches wird i.a. jedoch der Speicherzugriff (lesen und schreiben) beschleunigt.
- Anzahl der Level - Zugriffszeit
  Bei zwei Leveln muß zusätzliche Zeit für Ein-, Auslagern und Konsistenz-Mechanismen vorgesehen werden.
- Trennung - Zugriffszeit
  Daten- und Instruktionscache zu trennen erhöht die Zugriffsbandbreite, weil auf beide Caches getrennt zugegriffen werden kann, d.h. Instruktionen und Operanden holen kann parallel erfolgen. Allerdings muß zusätzliche etwas Zeit für Ein- und Auslagern vorgesehen werden.
- Trennung - Fehlschlagsrate
  Bei einer Trennung des Caches verringert sich die Fehlschlagsrate.
- Levelnummer - Fehlschlagsrate
  Bei einem 2.-Level-Cache gibt es kaum noch Fehlschläge, vor allem kaum Konflikt-Fehlschläge, weil er sehr groß ist.
- Adressierung - Zugriffszeit

Ein virtuell adressierter Cache ist bei erfolgreichem Zugriff schneller als ein realer, weil keine Adreßübersetzung vorgenommen werden muß. Bei einem real adressierten Cache kann die Adreßübersetzung beschleunigt werden, wenn die Cachegröße auf Hauptspeichergröße multipliziert mit der Assoziativität des Caches beschränkt ist (s.o.).
Ein virtueller Cache hat das Problem der Synonyme, d.h. wenn gleiche virtuelle Adressen verschiedener Prozesse sich auf die gleiche reale Adresse beziehen. Dafür muß ein Mechanismus implementiert werden, der dafür sorgt, daß solche Daten nur einmal im Cache mit einheitlicher virtueller Adresse vorkommen.

- Größe - Fehlschlagsrate
  Je größer der Cache, desto geringer ist die Fehlschlagsrate (Kapazitäts-, Konflikt- und Start-Fehlschlagsrate).
- Größe - Zugriffszeit
  Je größer der Cache, desto langsamer ist aus technologischen Gründen ein Zugriff.
- Zeilengröße - Transferzeit
  Je größer die Cache-Zeile, desto schneller können mehrere Daten eingelagert werden.
- Zeilengröße - Fehlschlagsrate
  Je größer die Cache-Zeile, desto weniger Start- und desto mehr Konflikt-Fehlschläge gibt es.
- Zeilengröße - Verzögerung durch Fehlschlag
  Je größer eine Zeile aber desto länger dauert das Einlagern einer Zeile, d.h. die Verzögerung durch Fehlschlag erhöht sich.
- Subzeilen - Verzögerung durch Fehlschlag
  Eine Verzögerung durch Fehlschlag kann mit Subzeilen verkürzt werden, da ein Prozessor nur warten muß, bis der entsprechende Teil der Zeile eingelagert ist.
- Subzeilen - Fehlschlagsrate
  Da auch Teile einer Zeile ungültig geschrieben werden können, gibt es bei Multiprozessor-Systemen weniger Schein-Sharing und somit eine geringere Fehlschlagsrate.
- Assoziativität - Fehlschlagsrate
  Je größer die Assoziativität desto weniger Konflikt-Fehlschläge gibt es, bei voll-assoziativen Caches gibt es keine Konflikt-Fehlschläge. Ein zwei-Wege-Satz-assoziativer Cache der Größe X hat eine geringere Fehlschlagsrate als ein direct-mapped-Cache der Größe 2*X.
- Assoziativität - Zugriffszeit
  Je größer die Assoziativität desto größer ist aus technologischen Gründen die Zugriffszeit.
- Ersetzunsstrategie - Fehlschlagsrate
  Die Ersetzunsstrategie beeinflußt stark die Konflikt-Fehlschlagsrate, besonders bei kleinen Caches und geringer Assoziativität. LRU führt zu einer geringeren Fehlschlagsrate als Random oder gar FIFO.
- Laden - Fehlschlagsrate
  Ein “mittleres” Prefetching (in der Reihenfolge: anfragegesteuert, Tag-gesteuert, fehlschlagsgesteuert, anfragegesteuert) verringert die Fehlschlagsrate. Die Start-Fehlschlagsrate sinkt, und Kapazitäts- und Konflikt-Fehlschlagsraten können aufgrund von Speicherverschmutzung steigen.
- Laden - Verzögerung durch Fehlschlag
  Anfragegesteuertes Prefetching hat eine geringere Verzögerung durch Fehlschlag als fehlschlagsgesteuertes Prefetching.
- Aktualisierung - Zugriffszeit
  Beim Durchschreiben dauert ein durchschnittlicher Schreibzugriff länger als beim Rückschreiben, weil immer in den Hauptspeicher geschrieben werden muß.

- Schreibstrategie - Zugriffszeit
  Bei write-around geht ein Schreibzugriff schneller als bei fetch-on-write im Falle eines Fehlschlags.
- Schreibpuffer - Zugriffszeit
  Mit Schreibpuffern ist ein Schreibzugriff auf Caches schneller, weil nur gewartet werden muß, bis das Datum im Schreibpuffer steht. Das Lesen dauert allerdings länger, weil bei einem Fehlschlag zusätzlich der Schreibpuffer durchsucht werden muß.
- Aktualisierung - Zugriffszeit
  Ein Schreibzugriff ist mit Rückschreiben schneller.
- Aktualisierung - Fehlschlagsrate
  Beim Durchschreiben ist jeder Schreibzugriff ein Fehlschlag, zumindest was den Zeitaspekt angeht.
- Aktualisierung - Transfer
  Beim Durchschreiben gibt es sehr viel Datentransfer zwischen Cache und Hauptspeicher, mehr als beim Rückschreiben.
- Start - Fehlschlagsrate
  Beim Warmstart gibt es weniger Start-Fehlschläge, wenn ein reaktivierter Prozeß noch alte Daten im Cache vorfindet, d.h. die Start-Fehlschlagsrate sinkt bei einem Warmstart, beim Kaltstart hingegen bleibt sie konstant.

7.4.4 Cache - Kosten

- Vorhandensein eines Cache - Fläche
  Ein Cache kostet Fläche auf dem Chip.
- 2-Level-Cache - Preis
  Ein 2.-Level-Cache erfordert zusätzliche Hardware für den Cache selbst und für die Zugriffslogik.
- Trennung - Fläche
  Getrennte Daten- und Instruktionscache erhöhen den Hardware-Aufwand und damit die Fläche für Cache und Zugriffslogik. Außerdem wird ein zweiter Bus bzw. andere Verbindung nötig, um die Vorteile der Trennung ausnutzen zu können.
- Adressierung - Fläche
  Bei realer Adressierung beim on-chip-Cache muß die MMU noch mit auf dem Chip, d.h. es verbleibt weniger Platz für den eigentlichen Cache.
- Größe - Fläche
  Die Größe des on-chip-Caches inklusive Zugriffslogik ist durch die begrenzte Fläche auf dem Chip eingeschränkt.
- Zeilengröße - Fläche
  Je kleiner die Zeilen, desto mehr Fläche muß für Verwaltung investiert werden (Tag-, Sicherheits- und Gültigkeitsbits).
- Assoziativität - Fläche
  Je größer die Assoziativität desto größer ist der Hardware-Aufwand und damit die Fläche für die Vergleicher und die Tags. Die zusätzliche Anzahl von Bits im Tag einer jeden Zeile ist gleich dem Logarithmus der Assoziativität.
- Laden - Transferrate
  Je mehr Prefetching desto höher ist die Transferrate.
- Laden - Transferzeit

Wegen der kritischen Zeit ist nur das Holen der jeweils nächsten Zeile möglich (one block lookahead).

- Aktualisierung - Fläche
  Durchschreiben ist billiger als Rückschreiben, weil keine Gültigkeitsbits und keine Logik nötig sind, die prüft, wann zurückgeschrieben werden soll.

7.4.5 Hauptspeicher - Leistung

- Seitengröße - Zugriffseit
  Je größer eine Seite, desto länger dauert das Ein- und Auslagern einer Seite bei einem Seitenfehler oder Prozeßwechsel. Das kann kritisch sein, wenn dies während der Prozeßlaufzeit passiert.
- Speicherverwaltung - Zugriffszeit
  - Bei der Seitenadressierung können die großen Seitentabellen nur als Ganzes ausgelagert werden, das führt zu einer Zeitverzögerung bei Prozeßwechsel oder Umlagern.
  - Bei der Segmentierung kann die Verschiebung wegen externer Fragmentierung die Zugriffszeit sehr verzögern.
  - Bei der 2-stufigen Segmentierung müssen alle Seiten eines Segmentes in einer Seitentabelle zusammen gespeichert werden. Bei der 3-stufigen Segmentierung kann die Größe der Seitentabellen auf eine Seitengröße beschränkt werden, das beschleunigt das Ein- und Auslagern.
- Speicherverwaltung - Zeit für Binden und Laden
  - Bei der Identität sind sowohl beim Binden als auch beim Laden Adreßmodifikationen nötig.
  - Bei der Seitenadressierung und bei der lokalen Segmentierung sind nur beim Binden Adreßmodifikationen nötig.
  - Bei der globalen Segmentierung sind weder beim Binder noch beim Laden Adreßmodifikationen nötig.
- Fehlschlagsrate - Zugriffszeit
  Ein Fehlschlag im Hauptspeicher führt zu einer langen Verzögerungszeit, weil das Einlagern einer Seite in den Hauptspeicher vergleichsweise sehr lange dauert.
- Speicherverschränkung - Zugriffszeit
  Mit Speicherverschränkung dauert ein einzelner Zugriff etwas länger, weil die Adresse auf das entsprechende Modul verteilt werden muß. Allerdings gehen die Zugriffe insgesamt schneller, weil sie zum großen Teil in den Modulen parallel abgearbeitet werden können. Dies trifft auch bei einer Verbindung zum Hauptspeicher zu, weil die meiste Zeit für den Zugriff im Hauptspeicher benötigt wird, und nur wenig Zeit, verglichen dazu, für den Transport.
- Verteilungsstrategie - Anzahl der Module - Zugriffszeit
  Eine zunehmende Anzahl an Modulen bringt nur noch wenig Zeitvorteil, speziell ist bei SIM die Beschleunigung proportional zu der Wurzeln der Anzahl der Module.
- Verteilungsstrategie - Zugriffszeit
  - Die Adreßumsetzung beim einfachem SIM ist sehr einfach und schnell.
  - Im allgemeinen liegt PRIM mit Polynomen bezüglich der Verteilung und der Zugriffszeit zwischen SIM und PRIM mit Matrizenmultiplikation.
- erweiterter Hauptspeicher - Fehlschlagsrate
  Der erweiterte Hauptspeicher sollte eine Fehlschlagsrate unter 0.0005% haben.

7.4.6 Hauptspeicher - Kosten

- Seitengröße - Hardware
  Je größer die Seiten, desto weniger Einträge in der Seitentabelle gibt es, d.h. desto weniger Speicherplatz brauchen die Seitentabellen.

- erweiterter Speicher - technische Zugriffszeit
  Welcher Ansatz beim erweiterten Speicher besser ist, der Direktanschluß oder das Caching, wird von dem Unterschied der Zugriffszeiten für Hauptspeicher und erweitertem Speicher bestimmt.

7.4.7 TLB - Leistung

- Größe - Zugriffszeit
  Ein großer TLB (in Vergleich zu einem Cache dennoch sehr klein!) kann viele Adressen speichern, so daß mehr Zugriffe, die über reale Adressen (real adressierter Cache oder Cache-Fehlschlag) stattfinden, beschleunigt werden können.

7.4.8 TLB - Kosten

- Größe - Fläche
  Der TLB ist relativ klein, um die dafür benötigte on-chip Fläche klein zu halten.

7.4.9 Multiprozessor - Leistung

- Lokalität der Speicher - Zugriffszeit
  Die Zugriffszeit ist bei verteilten Speichern auf den Prozessor-eigenen Speicher am schnellsten. Beim Zugriff auf andere Speicher muß die Transferzeit über die Verbindungsstruktur mit einkalkuliert werden.
- Uniformität - Zugriffszeit
  COMA unterstützt das Zwischenspeichern im Prozessor-eigenen Speicher, was bei mehrmaliger Verwendung die Zugriffszeit senkt.
- Granularität - Fehlschlagsrate
  Feste Granularität erhöht das Schein-Sharing, dynamische nicht.
- Privater Cache - Zugriffszeit
  Ein privater Cache verringert gerade bei Multiprozessor-Systemen, die Wahrscheinlichkeit, daß die Verbindung zum Hauptspeicher überlastet wird und ein Zugriff zu lange dauert.
- Multilevel-Cache - Zugriffszeit
  Ein 2.-Level-Cache verkürzt die Verzögerung für einen Fehlschlag beim 1.-Level-Cache.
- Kopien - Zugriffszeit
  Die Zugriffszeit verkürzt sich mit zunehmenden Kopien (keine Kopie, Wandern, Lese-, Schreibkopien), weil nicht auf das Holen des Datums gewartet werden muß. Allerdings kann es bei häufigem Schreiben zu Überlastungen im Verbindungssystem kommen.
- Aktualisieren der Caches - Zugriffszeit
  Für einen Zugriff geht Ungültigschreiben schneller als Aktualisieren.
- Dynamik - Verteilung - Zugriffszeit
  Der Zugriff für dynamisch verteilte Verzeichnisse dauert länger als für feste oder zentrale Verzeichnisse, weil meistens das Datum erst über verschiedene Prozessoren gesucht werden muß, allerdings kommt es vor allem bei zentralen aber u.U. auch bei festen Verzeichnissen schneller zu Verzögerungen durch Überlastung.

7.4.10 Multiprozessor - Kosten

- Granularität - Hardware
  Je dynamischer und feiner die Granularität desto mehr Organisationsbits sind nötig.
- Dynamik - Verteilung - Hardware
  Dynamische verteilte Verzeichnisse benötigen viel Speicher, weil jeder Prozessor ein komplettes

Verzeichnis über alle Daten hat.

8. Literatur

[Agar89] A. Agarwal. Analysis of Cache Performance for Operating Systems and Multiprogramming. Kluwer, 1989.

[Aho86[A.V. Aho, R. Sethi, J.D. Ullmann. Compilers: Principles, Techniques and Tools. Reading, Ma: Addison-Wesley, 1986.

[Alpe88] D.B. Alpert, M.J. Flynn. Performance Trade-Offs for Microprocessor Cache Memories. IEEE Micro, August 1988.

[Baeh91] H. Bähring. Mikrorechner-Systeme: Mikroprozessoren, Speicher, Peripherie. Springer, 1991.

[Baer80] J. L. Baer. Computer Systems Architecture. London: Computer Science Press, 1980.

[Baer88] J. L. Baer, W. H. Wang. On the Inclusion Properties for Multi-Level Cache Hierarchies. In Proceedings of the 15th International Symposium on Computer Architecture, 1988.

[Bala88] M. Balakrishnan, A.K. Majumdar, D.K. Banerji, et al. Multiple Storage Adaptive Multi-Trees. IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, March 1988.

[Bala93]F. Balasa, F. Catthoor, H. De Man. Exact Evaluation of Memory Size for Multi-dimensional Signal Processing Systems. In IEEE/ACM International Conference on Computer-Aided Design, 1993.

[Bars92] H. Barsuhn, W. Lochlein, D. Wendel, U. Wille, P. Coppens. Level-2 Cache for High Performance /390-µ Processors. In EUROMICRO: European Conference, Software and Hardware: Specification and Design, 1992.

[Bror93] M. Brorsson, P. Stnstr”om. Visualisation of Cache Coherence Bottlenecks in Shared Memory Mulitprocessor Applications. IEEE Computer Architecture Newsletter, Fall 1993.

[Burn70] G.J. Burnett, E.G. Coffmann. A Study of Interleaved Memory System. In Spring Joint Computer Conference, vol. 36, pages 467–474, 1970.

[Capp92] F. Cappello, J.-L. Bechennec, J.-L. Giavitto. Design of the Processing Node of the Path64 Parallel Computer. In EUROMICRO: European Conference, Software and Hardware: Specification and Design, 1992.

[Catt88] F. Catthoor et al. Architectural Strategies for an Application-Specific Synchronous Multiprocessor Environment. IEEE Transactions on Acoustics, Speech and Signal Processing, February 1988.

[Chai90] D. Chaiken, C. Fields, K. Kurihara, A. Agarwal. Directory-Based Cache Coherence in Large-Scale Multiprocessors. IEEE Computer, June 1990.

[Chen89] C.-L. Chen, C.-K. Liao. Analysis of Vector Access Performance on Skewed Interleaved Memory. Computer Architecture News, January 1989.

[Cheo90] H. Cheong, A.V. Veidenbaum. Compiler-Directed Cache Management in Multiprocessors. IEEE Computer, June 1990.

[Cher91] D.R. Cheriton, H.A. Goosen, P.D. Boyle. Paradigm: A Highly Scalable Shared-Memory Multicomputer Architecture. IEEE Computer, February 1991.

[Cheu87] K. Cheung, G. Sohi, K. Saluja, D. Pradhan. Organization and Analysis of a Gracefully-Degrading Interleaved Memory System. Computer Architecture News, 1987.

[Cheu90] K.Cl Cheung, G.S. Sohi, K.K. Saluja, D.K. Pradhan. Design and Analysis of a Gracefully Degrading Interleaved Memory System. IEEE Transactions on Computers, January 1990.

[DeBl90] M. DeBlasi. Computer Architecture. Addison-Wesley, 1990.

[Dela90] A.Delaruelle, O. McArdle, J. van Meerbergen, C. Niessen. Synthesis of Delay Functions in DSP-Compilers. In The European Design Automation Conference, 1990.

[Devi91] Y. Deville, J. Gobert. A Class of Replacement Policies for Medium and High-Associativity Structures. Computer Architecture News, March 1991.

[Dewa93] V. S. S. Nair G. Dewan. A Case for Uniform Memory Access Multiprocessors. Computer Architecture News, September 1993.

[Drac93] N. Drach, A. Seznec. Semi-Unified Caches: Increasing Associativity of on-chip Caches. IEEE Computer Architecture Newsletter, Fall 1993.

[Dubo90] M. Dubois, Sh. Thakkar. Guest Editors' Introduction: Cache Architectures in Tightly Coupled Multiprocessors. IEEE Computer, June 1990.

[East90] I. East. Computer Architecture and Organization. Pittmann, 1989.

[Ever90] W. Everling. Rechnerorganisation. BI-Wissenschaftsverlag, 1990.

[Ewy93] B.J. Ewy, J.B. Evans. Secondary Cache Performance in RISC Architectures, 1993.

[Fulc89] J. Fulcher. An Introduction to Microcomputer Systems, Architecture and Interfacing. Addison-Wesley, 1989.

[Gajs92]D.D. Gajski, N.D. Dutt, A. Wu, St. Lin. High-Level Synthesis: Introduction to Chip and System Design, Kluwer Academic Publishers, 1992.

[Gara90] K. Garachorloo, D. Lenoski, J. Laudon, et. al. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In Annual International Symposium on Computer Architecture, 1990.

[Goks89] A.K. Goksel, R.H. Kranbeck, P.P. Thomas, et al. A Content Addressable Memory Management Unit with On-Chip Data Cache. IEEE Journal of Solid-State Circuits, June 1989.

[Goor89] A.J. van de Goor. Computer Architecture and Design. Addison-Wesley, 1989.

[Grant89] D. Grant, P.B. Denyer, J. Finlay. Synthesis of Address Generator. In IEEE International Conference on Computer-Aided Design, 1989.

[Grant91] D. Grant, P.B. Denyer. Address Generation for Array Access Based on Modulus m Counters. In European Design Automation Conference, 1991.

[Grau90] G. Graunke, S. Thakkar. Synchronization Algorithms for Shared-Memory Multiprocessors. IEEE Computer, June 1990.

[Hage92] E. Hagersten, A. Landin, S. Haridi. DDM - A Cache-Only Memory Architecture. IEEE Computer, September 1992.

[Hama90] V.C. Hamacher, Z.G. Vranesic, S.G. Zaky. Computer Organization. McGraw-Hill, 1990.

[Harp86] D.T. Harpter III, J.R. Jump. Performance Evaluation of Vector Accesses in Parallel Memories Using a Skewed Storage Scheme. Computer Architecture News, 1986.

[Haye88] J. P. Hayes. Computer Architecture and Organization, Second Edition. McGraw-Hill, 1988.

[Henn90] J. L. Hennessy, D. A. Patterson. Computer Architecture: A Quantitative Approach. Morgan Kaufman, 1990.

[Hill90] M.D. Hill, J.R. Larus. Cache Considerations for Multiprocessor Programmers. Communications of the ACM, March 1992.

[Hill92] M.D. Hill, J.R. Larus, S.K. Reinhardt, D.A. Wood. Cooperative Shared Memory: Software and Hardware Support for Scalable Multiprocessors. In Architectural Support for Programming Languages and Operating Systems, 1992.

[Hoba89] W.C. Hobart Jr., H.G. Cragon. Locality Characteristics of Symbolic Programs. In IEEE International Conference on Computer Design, pages 508 – 511, 1989.

[Hsie89] M.M. Hsieh, T.C. Wei, W.N. Loo. A Cached System Architecture Dedicated for the System I/O Activity on a CPU Board. In IEEE International Conference on Computer Design, pages 512 – 517, 1989.

[Huck89] J.C. Huck, M.J. Flynn. Analyzing Computer Architectures. IEEE Computer Society Press, 1989.

[Hyat93] C. Hyatt. A High Performance Object-Oriented Memory. In Computer Architecture News, 1993.

[Jess87] E. Jessen, R. Valk. Rechensysteme, Grundlagen der Modellbildung. Springer 1987.

[Jeon89] D.-K. Jeong, D.A. Wood, G.A. Gibson, et al. A VLSI Chip Set for a Multiprocessor Workstation - Part II: A Memory Management Unit and Cache Controller. IEEE Journal of Solid-State Circuits, December 1989.

[Joup90]N. Jouppi. Improving Direct-Mapped Cache Performance by Addition of a Small Fully-Associative Cache and Prefetch Buffer. In Proceedings of the 17th Annual International Symposium on Computer Architecture, 1990

[Kael92] D.R. Kaeli, P.G. Emma, J.W. Knight, Th.R. Puza. Contrasting Instruction-Fetch Time and Instruction-Decode Time Branch Prediction Mechanisms: Achieving Synergy Through Their Cooperative Operation. In EUROMICRO: European Conference, Software and Hardware: Specification and Design, 1992.

[Kim92] T. Kim, C. Liu. A New Approach to the Multiport Memory Allocation Problem in Data Path Synthesis. VLSI Integration, 1992

[Knie89] Th. Knieriemen. Rechneraufbau am konkreten Beispiel: Dargestellt anhand der Macintosh2-Modellreihe. Vieweg, 1989.

[Koho80] T. Kohonen. Content-Addressable Memories. Springer, 1980.

[Kola92] T. Kolarz. Performance Evaluation: An Analytical Model for Calculating Start Misses in Caches. In EUROMICRO: European Conference, Software and Hardware: Specification and Design, 1992.

[Kolk93]T. Kolks, B. Lin, H. De Man. Sizing and Verification of Communication Buffers for Communicating Processes. In IEEE/ACM International Conference on Computer-Aided Design, 1993.

[Kuri91] L. Kurian, P.T. Hulina, L.D. Coraor, D.N. Mannai. Classification and Performance Evaluation of Instruction Buffering Techniques. In Annual International Symposium on Computer Architecture, pages 150 – 159, 1991.

[Lang89] G. Langholz, J. Francioni, A. Kandel. Elements of Computer Organization. Prentice Hall,

1989.

[Leno92] D. Lenoski, J. Lauon, K. Gharachorloo, et al. The Stanford DASH Multiprocessor. IEEE Computer, March 1992.

[Li89] P. Hudak K. Li. Memory Coherence in Shared Virtual Memory Systems. IEEE Transactions on Computers, November 1989.

[Li91] K. Li, K. Petersen. Evaluation of Memory System Extensions. Computer Architecture News, May 1991.

[Lilj93] D. J. Lilja. Cache Coherence in Large-Scale Shared-Memory Multiprocessors. Computing Survey, September 1993.

[Lin89] K. S. Lin, G. A. Frantz, R. Simar Jr. The TMS320 Family of Digital Signal Processors. In Proceedings of the IEEE, September 1989.

[Lins92] R.D. Lins. A Multi-Processor-Shared Memory Architecture for Parallel Cyclic Reference Counting. In EUROMICRO: European Conference, Software and Hardware: Specification and Design, 1992.

[Lipp93]P. Lippens, J. van Meerbergen u.a. Allocation of Multiprot Memories for Hierarchical Data Streams. In IEEE/ACM International Conference on Computer-Aided Design, 1993.

[Marw92] P. Marwedel, W. Rosenstiel. Synthese von Register-Transfer-Strukturen aus Verhaltensbeschreibungen. In Informatik-Spektrum, Februar 1992.

[Marw93] P. Marwedel. Begleitmaterial zur Vorlesung "Rechnerarchitektur" im Sommersemester 93. Universität Dortmund, September 1993.

[McFa90] M. C. McFarland, A. C. Parker, R. Camposano. The High-Level Synthesis of Digital Systems. In Proceedings of the IEEE, February 1990.

[McNi88] G.D. McNiven, E.S. Davidson. Analysis of Memory Referencing Behavior for Design of Local Memories. Computer Architecture News, May 1988.

[Meer92] J. van Meerbergen, P. Lippens, et. al. Architectural Strategies for High-Throughput Applications. Journal of VLSI Signal Processing, 1992.

[Mesl92] A.M. Meslin, A.C. Pacheco, J.S. Aude. A Comparative Analysis of Cache Memory Architecture for the Multibus Multiprocessor. In EUROMICRO: European Conference, Software and Hardware: Specification and Design, 1992.

[Milu89] V.M. Milutinovic. High-Level Language Computer Architecture. Computer Science Press, 1989.

[Muld91] J.M. Mulder, N.T. Quach, M.J. Flynn. An Area Model for On-Chip Memories and its Application. IEEE Journal of Solid-State Circuits, February 1991.

[Nitz91] B. Nitzberg, V. Lo. Distributed Shared-Memory: A Survey of Issues and Algorithms. IEEE Computer, August 1991.

[Ober89] W. Oberschelp, G. Vossen. Rechneraufbau und Rechnerstrukturen. Oldenbourg, 1989.

[Ogur89] T. Ogura, J. Yamada, S.-I. Yamada, M.-A. Tan-No. A 20-KBit Associative Memory LSI for Artificial Intelligence Machines. IEEE Journal of Solid-State Circuits, August 1989.

[Paet91] Ch. Pätzhold. Problemlösen im VLSI-Entwurf: Der Speicherspezialist im CHARM-System. Diplomarbeit, Universität Dortmund, Lehrstuhl Informatik 1, 1991.

[Perle93] Ch. H. Perleberg, A. J. Smith. Branch Target Buffer Design and Optimization. In IEEE Transactions on Computers, 1993.

[Prad93]D. K. Pradhan, M. Chatterjee, S. Banerjee. Buffer Assignment For Data Driven Architectures. In IEEE/ACM International Conference on Computer-Aided Design, 1993.

[Proe92]W. E. Proebster. The Evolution of Data Memory and Storage: An Overview. In Computer System Software Engineering. Ed. P. Deville and J. Vandewalle. Kluwer, 1992

[Przy88] St. Przybylski, M. Horowitz, J. Hennessy. Performance Tradeoffs in Cache Design. In Annual International Symposium on Computer Architecture, 1988.

[Rama89] U. Ramachandran, M.y.A. Khalidi. A Design of a Memory Management Unit for Object-Based Systems. In IEEE International Conference on Computer Design, pages 522 – 523, 1989.

[Rau91] B.R. Rau. Pseudo-Randomly Interleaved Memory. In Annual International Symposium on Computer Architecture, pages 74–83, 1991.

[Rhei92] D. Rhein, H. Freitag. Mikroelektronische Speicher: Speicherzellen, Schaltkreise, Systeme. Springer-Verlag, 1992.

[Sand92] H.S. Sandhu, B. Gamsa, S. Zhou. Region-Oriented Memory Management in Shared-Memory Multiprozessors. Technical Report CSRI-269, Computer System Research Institute of the University of Toronto, 1992.

[Sawa89] K. Sawada, T. Sakurai, K. Nogami, et al. A 32-KByte Integrated Cache Memory. IEEE Journal of Solid-State Circuits, August 1989.

[Scho88] E. Schoppnies. Taschenlexikon Halbleiterelektronik. VEB Bibliographisches Institut Leibzig, 1988.

[Siew82] D.P. Siewiorek, C.G. Bell, A. Newell. Computer Structures. McGraw-Hill, 1982.

[Sing91] J.P. Singh, W.-D. Weber, A. Gupta. Splash: Stanford Parallel Applications for Shared-Memory. Computer Architecture News, March 1991.

[Smit82] A.J. Smith. Cache Memories. ACM Computing Surveys, September 1982.

[Smit85] A.J. Smith, J.R. Goodman. Instruction Cache Replacement Policies and Organizations. IEEE Transactions on Computers, March 1985.

[Sten90] P. Stenström. A Survey of Cache Coherence Schemes for Multiprocessors. IEEE Computer, June 1990.

[Ston90] H.S. Stone. High-Performance Computer Architecture. Addison-Wesley, 1990.

[Stum90] M. Stumm, S. Zhou. Algorithms Implementing Distributed Shared Memory. IEEE Computer, May 1990.

[Swaa92] M.F.X.B. van Swaaij, F.H.M. Franssen, F.V.M. Catthoor, H.J. De Man. Modeling Data Flow and Control Flow for High Level Memory Management. In The European Design Automation Conference, pages 8–13, 1992.

[Tane84] A.S. Tanenbaum. Structured Computer Organization. Prentice-Hall, 1984.

[Tava90] D. Tavangarian. Flagorientierte Assoziativspeicher und -prozessoren. Informatik Fachberichte 240, Springer, 1990.

[Tell90] P.J. Teller. Translation-Lookaside Buffer Consistency. IEEE Computer, June 1990.

[Thak90] S. Thakar, M. Dubous, A.T. Laundrie. New Directions in Scalable Shared-Memory Multiprocessor Architectures. IEEE Computer, June 1990.

[Toma93a] M. Tomasevic, V. Milutinovic. The Cache Coherence Problem in Shared-Memory Multiprocessors: Harware Solutions. IEEE Computer Society Press, 1993.

[Toma93b] M. Tomasevic, V. Milutinovic. Hardware solutions for Cache Coherence in Shared-Memory Multiprocessor Systems. In The Cache Coherence Problem in Shared-Memory Multiprocessors: Harware Solutions. IEEE Computer Society Press, 1993.

[Tsen92] Ch.-Ch. Tseng, Ch.-Z. Lin, J.-K. Kwang, K.-T. Lin. A Data Driven Hybrid Computer Architecture. In EUROMICRO: European Conference, Software and Hardware: Specification and Design, 1992.

[Unge89] T. Ungerer. Innovative Rechnerarchitekturen: Bestandsaufnahme, Trends, Möglichkeiten. McGraw-Hill, 1989.

[Vogt90] C. Vogt. A Buffer-Based Method for Storage Allocation in an Object-Oriented System. IEEE Transactions on Computers, March 1990.

[Vran91] Z.G. Vranesic, M. Stumm, D.M. Lewis, R. White. Hector: A Hierarchical Structured Shared Memory Multiprocessor. IEEE Computer, January 1991.

[Wang89] W. H. Wang, J. L. Baer, H. M. Levy. Organization and Performance of a two-Level Virtual-Real Cache Hierarchy. In Annual International Symposium on Computer Architecture, Juni 1989.

[Weck82] G. Weck. Prinzipien und Realisierung von Betriebssystemen. Teubner, 1982.

[Weik93a] G. Weikum, P. Zabback. I/O Parallelität und Fehlertoleranz in Disk-Arrays I. In Informatik Spektrum, Juni 1993.

[Weik93b] G. Weikum, P. Zabback. I/O Parallelität und Fehlertoleranz in Disk-Arrays II. In Informatik Spektrum, August 1993.

[Whee92] B. Wheeler, B.N. Bershad. Consistency Management for Virtually Indexed Caches. In Architectural Support for Programming Languages and Operating Systems, 1992.

[Woud90] R. Woudsma et al. Piramid: An Architecture-Driven Silicon Compiler for Complex DSP Applications. Proc. of ISCAS, May 1990.

[Zhou90] S. Zhou, M. Stumm, K. Li, D. Wortman. Heterogeneous Distributed Shared Memory. Technical Report CSRI-244, Computer System Research Institute of the University of Toronto, 1990.

Anhang: Liste aller Parameter