Belegarbeit

Anforderungsanalyse für Daten-Caches für die SHAP-Mikroarchitektur

Stefan Alex

geboren am 21.01.1986 in Radebeul

Matrikelnummer: 3165659

Januar 2009

Technische Universität Dresden
Fakultät Informatik
Institut für Technische Informatik

Betreuender Hochschullehrer: Prof. Dr.-Ing. habil. Rainer G. Spallek
Betreuer: Dipl.-Inf. Martin Zabel

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1 Einführung

Caches sind ein wichtiger Ansatz zur Überwindung der Geschwindigkeitslücke zwischen Prozessor und Arbeitsspeicher. Er sieht vor, häufig genutzte Daten und Befehle möglichst nah und schnell verfügbar am Prozessor zu speichern, um somit die Zugriffslatenz zu verringern. Es existieren mehrere Strategien zum Aufbau von Caches, angefangen bei kostengünstigen, einfachen Lösungen, bis zu kostenintensiven, aber flexiblen und effizienten Systemen. Weiterhin wurden eine Reihe von Optimierungen vorgeschlagen, die durch wenig Aufwand Flaschenhälse eliminieren und einfache Architekturen aufwerten können.

Der Cache-Ansatz soll auch für die SHAP-Mikroarchitektur angewendet werden. Dabei handelt es sich um einen eingebetteten Bytecode-Prozessor. Er wurde speziell für die direkte Ausführung von Java optimiert und unterstützt nebenläufige Garbage-Collection in Hardware. Das machte ein komplexes Speicher-Managementsystem nötig und erfordert nun eine genaue Untersuchung der möglichen Architektur des Caches und seiner Positionierung im Prozessor-Speicher-System.

Zur Analyse werden für existierende Beispielanwendungen die Speicherzugriffe aufgezeichnet. Die Auswertung kann einerseits mit einem analytischen Modell geschehen. Dabei werden über die Zugriffsfolge Parameter bestimmt und mittels einer Zielfunktion die Leistungsdaten ermittelt. Die zweite Möglichkeit besteht in der Simulation des Caches und seines Verhaltens. In dieser Studienarbeit wurden beide Ansätze verfolgt, obwohl dem Zweiten aufgrund der größeren Flexibilität und Genauigkeit mehr Aufmerksamkeit geschenkt wurde.

In Kapitel 2 werden die Grundlagen der Cache-Speicher und der analytischen Modelle vorgestellt. Kapitel 3 beschäftigt sich mit der Vorbetrachtung der Simulation und beschreibt die entwickelten Tools eingehender. Das vierte Kapitel liefert eine kurze Auswertung der Lokalität der Beispielanwendungen und soll bereits eine Vorabschätzung der zu erwarteten Resultate liefern. Im letzten Kapitel werden schließlich die Simulationsergebnisse ausgewertet und Schlussfolgerungen über eine mögliche Implementierung gezogen.

2 Grundlagen

Moderne Rechnerarchitekturen leiden unter dem Problem, dass zwar der Prozessor Befehle mit einer hohen Geschwindigkeit verarbeiten kann, jedoch der Speicher nicht in der Lage ist, ihm diese Befehle in angemessener Zeit zu liefern. Um diese Geschwindigkeitslücke zu verringern, wurden Caches [1] entwickelt.

Häufig genutzte Speicherwörter werden statt aus dem langsamen Hauptspeicher (DDR-SDRAM-Technologie - zum Beispiel 11.25 ns Zugriffszeit[2] [1]) aus schnellen, prozessornahen Speichern angefordert (SRAM-Technologie - zum Beispiel 0.45 ns Zugriffszeit[3] [2]). Diese sind mit einem Mehraufwand in der Herstellung verbunden, können aber in der Regel 90 % der benötigten Daten aufnehmen [3].

2.1 Lokalitätsprinzip

Das Lokalitätsprinzip geht davon aus, dass der Zugriff eines Programms auf Adressen nicht gleichverteilt erfolgt. Die 90/10-Regel besagt, dass ein Programm während 90 % seiner Ausführungszeit mit 10 % des Codes beschäftigt ist [3].

Die Anfänge gehen auf die Entwicklung des Pagings in den sechziger Jahren zurück. Peter J. Denning [4] definiert das *Working Set* eines Prozesses als die Seiten, die in den Speicher geladen werden müssen, damit der Prozess bis zur nächsten Ausführungsphase effektiv arbeiten kann. Es umfasst zu einem bestimmten Zeitpunkt die Adressen, die vorher innerhalb eines Zeitfensters aufgerufen wurden. Denning entwickelte daraus ein Modell zur Analyse des Programmverhaltens. Lokalität beschreibt er als *Locality of Reference*. Es ist zu beobachten, dass der Aufruf von Seitenreferenzen über die Zeit zu Clustern neigt. Als Ursachen für Lokalität ist der Aufbau eines Programms zu sehen. Oft wird in speziellen Kontexten gearbeitet [5]. Ein Programm ist modular aufgebaut. Auch sorgt ein Schleifendurchlauf für starke Lokalität.

Zeitliche Lokalität (*temporal locality*) tritt auf, wenn auf kürzlich verwendete Daten oder Instruktionen in Zukunft wieder zugegriffen wird. Gründe sind der Ablauf von Schleifen, der Aufruf gleicher Methoden oder die Verwendung von Hot-Spot-Speicherstellen, zum Beispiel häufig gebrauchte globale Variablen.

Örtliche Lokalität (*spatial locality*) tritt auf, wenn benachbarte Adressen in kurzer Zeit zusammen aufgerufen werden. Gründe dafür können Instruktionen, die sequentiell im Speicher

[1] französisch *cacher* - verstecken

[2] entspricht der RAS-CAS-Latenz des Qimonda 512-Mbit DDR2 SDRAM HYB18TC512800CF

[3] SAMSUMG 36MbQDRII+ K7S3236U4C SRAM

abgelegt wurden, sein. Auch ein Zugriff auf bestimmte Datenstrukturen wie Arrays oder Matrizen kann zu örtlicher Lokalität führen.

2.2 Cache-Architekturen

2.2.1 Aufbau

Ein Cache besteht aus zwei Speicherabschnitten. Ein Bereich enthält die aus dem Hauptspeicher[4] stammenden Daten. In einem zweiten Bereich ist für jedem Eintrag ein *Tag* gespeichert, so dass die ursprüngliche Hauptspeicheradresse rekonstruiert werden kann.

Der Daten-Speicher des Caches ist in nicht-überlappende Blöcke fester Größe unterteilt, den *Cache-Lines* oder Cache-Blöcken. Eine Cache-Line hat dabei typischerweise eine Größe von 4 bis 64 aufeinanderfolgenden Bytes [6], die (meistens) in der selben Reihenfolge wie in der niedrigeren Speicherhierarchieebene übernommen werden und an festen Grenzen ausgerichtet sind. Die Block-Größe ist in der Regel höher als die Größe eines Speicherwortes der Rechnerarchitektur. Aus diesem Grund liegt nicht nur ein gewünschtes Datum im Cache, sondern auch seine unmittelbaren Nachbarn. Damit wird versucht, die örtliche Lokalität auszunutzen.

Neben der Cache-Line und dem Tag existiert zu jedem Eintrag ein *Valid*-Bit, welches die Daten als gültig kennzeichnet. Abhängig von der Schreibstrategie (siehe 2.2.2.2) kann ein *Dirty*-Bit enthalten sein. Es gibt an, ob der Eintrag im Cache aktueller als im Hauptspeicher ist.

Caches können nach der Art der Daten, die sie speichern, unterschieden werden. In einem gemeinsamen Cache (*unified cache*) sind sowohl Instruktionen, als auch Programdaten enthalten. Sie werden gleichartig behandelt. Ein geteilter Cache (*split cache*)[5] trennt sie hingegen in verschiedene Speicher. Das Ziel ist die Ausnutzung der Pipeline-Architektur, bei der parallel auf Daten und Instruktionen zugegriffen wird. Die Bandbreite ist höher, da beide Caches unabhängig voneinander angesteuert werden [6].

Caches können mehrere Hierarchieebenen umfassen. Bei solchen Systemen befinden sich mehrere Speicher zwischen dem Prozessor und dem Hauptspeicher. Bei einem Zugriff wird zuerst der dem Prozessor am nächsten liegende Cache betrachtet. Sind die Daten nicht vorhanden, wird die nächste Ebene angefragt. Erst, wenn kein Cache die Daten bereitsstellen kann, wird auf den Hauptspeicher zugegriffen. Bei Multi-Level-Cache-Systemen unterscheidet man folgende Ladestrategien:

Inclusive Caches: eine Zeile, die in einer höherern Stufe (näher am Prozessor) enthalten ist, muss auch in allen darunterliegenden Ebenen vorhanden sein.

Non-Inclusive Caches: Einträge einer höheren Stufe befinden sich nicht zwangsweise in einer Tieferen. Wird ein Block in einer hohen Ebene verdrängt, so nimmt ihn der darun-

[4] im Folgenden wird die unter dem Cache liegende Speicherhierarchieebene als Hauptspeicher bezeichnet

[5] dieses Konzept ist als *Harvard*-Architektur bekannt, ein gemeinsamer Cache wird auch als *Von-Neumann*-Architektur bezeichnet

terliegende Cache auf und hält ihn vor, bis er gebraucht oder wiederum verdrängt wird. Der Cache arbeitet als *Victim-Cache*.

Eine weitere Unterscheidung ist die Platzierung des Caches. Er kann entweder *On-Chip* auf dem Prozessor liegen. Dieser Fall führt zu einem Geschwindigkeitsgewinn, da der Prozessor wie auf Register mit geringen Zugriffszeiten anfragen kann. Er bedingt allerdings auch eine Obergrenze bezüglich der Speicherkapazität. Andererseits kann der Cache auf einem separaten Chip (*Off-Chip*) [8] liegen. Dabei können schnelle synchrone SRAM-Zellen verwendet werden, um die Latenz gering zu halten [7]. Eine gängige Realisierung ist die Verwendung eines Split-Caches On-Chip auf dem Prozessor sowie eines Unified-Caches On-chip. Ein Level-3-Cache kann sowohl On-Chip oder Off-Chip realisiert werden [7].

Eine weitere wichtige Eigenschaft eines Caches ist seine Implementierung:

Direkt abbildender Cache (*Direct-mapped-cache*): Dieser Ansatz stellt die simpelste Cache-Implementierung dar. Er verlangt keine bestimmten Strukturen, sondern ist mit einem einfachen RAM realisierbar [7]. Ein bestimmtes Speicherwort kann nur an einer Stelle im Cache abgelegt werden (*many-to-one-mapping*). Bei einem Aufruf entfallen somit aufwendige Vergleiche. Die gesuchte Cache-Line kann sofort adressiert und verglichen werden. Die Position wird durch einen Teil der Hauptspeicheradresse, der Indexadresse, bestimmt.

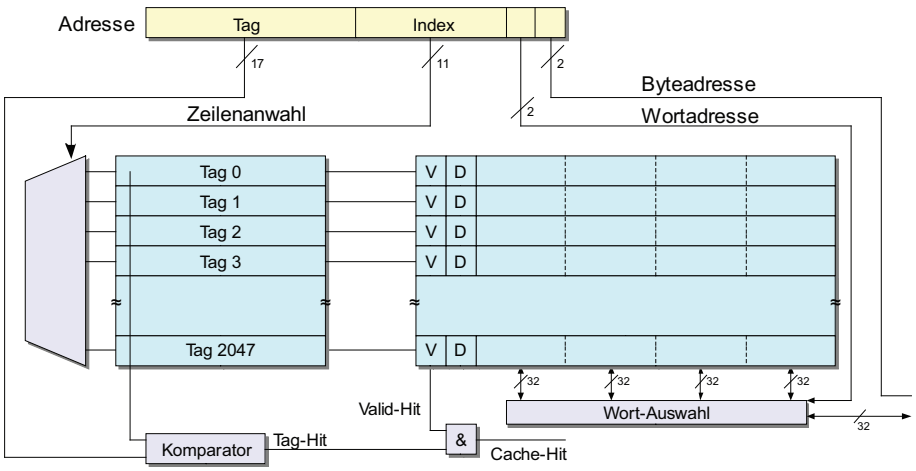

Abbildung 2.1: Ein direkt abbildender Cache (nach [7]).

Das Problem bei dieser Architektur ist die hohe Konfliktrate. Keine zwei Einträge, die sich durch die Index-Adresse gleichen, können zur selben Zeit im Cache liegen. Handelt es sich um zwei häufig gebrauchte Speicherworte, sinkt die Effektivität. Dagegen gibt es Ansätze, Kollisionen gezielt durch den Compiler zu eliminieren [6]. Auch wird zur Lösung ein *Write-Buffer*/*Victim-Cache* vorgeschlagen [3]. Der Ansatz sieht vor, verdrängte Blöcke in einem kleinen, vollassoziativen Cache (1-4 Einträge) zwischenzuspeichern. Wird der Block nach der Verdrängung wieder referenziert, wird lediglich der Victim-Cache abgefragt. Bei kleineren direkt abbildenden Caches können so ein Viertel aller

Fehlschläge beantwortet werden. Ein weiterer Vorteil ist, dass bei einer Verdrängung erst die Leseoperation des neuen Datums stattfinden kann und die Schreiboperation verschoben wird. Damit kann die Prozessorblockierung eher aufgehoben werden.

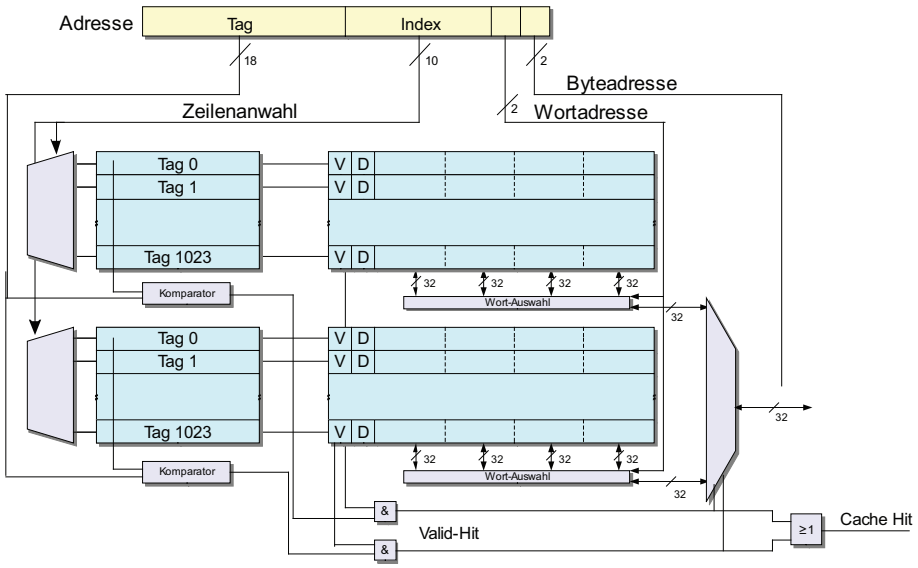

Abbildung 2.2: Ein 2-fach satzassoziativer Cache (nach [7]).

Teilassoziativer n-Wege Cache (*Set-associative-cache*): der Datenbereich wird in *Sets* unterteilt. Jedes Set kann gleichzeitig *n* Cache-Lines aufnehmen. Das Ziel-Set eines bestimmten Eintrags wird durch die Index-Adresse ermittelt (ein direkt abbildender Cache ist demnach ein 1-Weg Cache). Innerhalb des Sets bestimmt eine Ersetzungsstrategie, an welche Stelle der Eintrag gespeichert wird und welcher ältere Block verdrängt, also aus dem Cache gelöscht wird (*many-to-few-mapping*).

Der Aufwand erhöht sich mit dieser Architektur, es sind mehr Vergleicher als beim direkt abbildenden Cache[6] und die die Implementierung einer Ersetzungsstrategie nötig, um ein Datum zu finden bzw. zu speichern.

Vollassoziativer Cache (*Fully-associative-Cache*): hier handelt es sich um den Extremfall des set-assoziativen Caches, bei dem das *n* mit der Anzahl der Blöcke übereinstimmt. Der Cache besteht nur aus einem Set, in dem Wahlfreiheit bezüglich der Positionierung eines Datums herrscht. Als Folge steigt der Implementierungsaufwand. Da die Indexadresse entfällt, ist die Tag-Länge bis auf die Wort- und Byteauswahl-Bits die Länge der Speicheradresse. Ebenso muss jede Zeile mit einem Vergleicher und entsprechenden Multiplexern versehen werden. Dem Aufwand steht eine hohe Flexibilität gegenüber. Konflike zwischen oft genutzten Einträgen können auf einen Minimum begrenzt werden. Es ist möglich, mehrere Speicherbereiche, die sich in der Index-Adresse überlappen, gleichzeitig zu cachen [8].

[6] alle Vergleiche innerhalb eines Sets werden parallel durchgeführt [3].

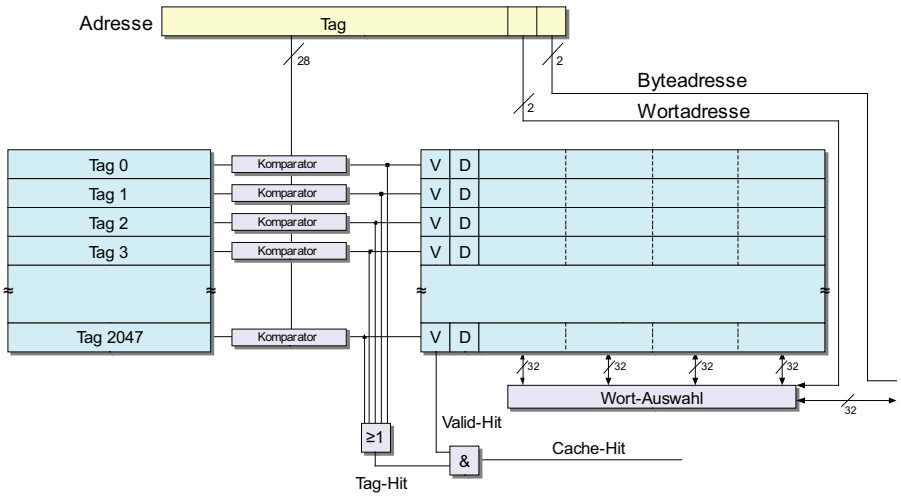

Abbildung 2.3: Ein vollassoziativer Cache (nach [7]).

2.2.2 Funktionsweise

2.2.2.1 Cache-Zugriff

Bei einem Zugriff können die Daten im Cache gefunden werden oder nicht. Man spricht von einem *Cache-Hit* oder einem *Cache-Miss*. Die Klassifikation kann auch nach der Art des Zugriffs erfolgen. Ein Schreibzugriff resultiert in einem *Write-Hit* oder einem *Write-Miss*. Bei einem Lesezugriff spricht man von einem *Read-Hit* oder einem *Read-Miss*. Im Fall eines Fehlschlags beim Lesen wird das Wort aus dem Hauptspeicher geholt und dem Prozessor zur Verfügung gestellt. Es wird gleichzeitig in den Cache geladen, um bei einem weiteren Zugriff bereitzustehen. Darüber hinaus ist es nötig, weitere Speicherwörter zu laden, um die Cache-Line zu füllen[7].

2.2.2.2 Schreibstrategien

Bei einem Schreibzugriff unterscheidet man verschiedene Strategien:

Copy-Back: Inkonsistenzen zwischen Cache und Speicher sind erlaubt. Die Einträge im Cache entsprechen dem letzten gültigen Stand, während im Hauptspeicher ältere Daten enthalten sein können. Bei einem Write-Hit wird nur der Wert im Cache aktualisiert. Das hat den Vorteil, dass Schreibzugriffe auf ein Datum beschleunigt werden. Erst beim Beenden oder einer Verdrängung wird der Speicher aktualisiert. Bei einem Cache-Miss wird der Wert in den Cache geladen und dort verändert. Für die Kennzeichnung der Daten dient das Dirty-Bit.

Write-Through: der Hauptspeicher wird sofort nach einer Schreiboperation aktualisiert. Cache und Hauptspeicher sind immer konsistent. Das Verfahren ist mit weniger Aufwand verbunden, da keine Dirty-Bits die aktuelleren Cache-Einträge markieren müssen.

Es wird die *Write-Allocation*-Strategie unterschieden:

[7] die Strategie, erst das angefragte Wort aus dem Speicher zu holen, ist als *Wrap Around* bekannt [7]

Write-Through with No-Write-Allocation bei einem Write-Miss wird das Datum nicht in den Cache geschrieben. Es sind keine weiteren Lesevorgänge nötig, um die Cache-Line zu füllen.

Write-Through with Write-Allocation ein Write-Miss führt zum Laden des Datums in den Cache. Spätere Lesezugriffe auf diesen Wert können ohne den Speicher beantwortet werden.

Write-Through sorgt nur noch für Geschwindigkeitsgewinne bei Leseoperationen. Schreiboperationen, die ca. 30 % aller Befehle ausmachen [7], benötigen die komplette Hauptspeicherlatenz.

2.2.2.3 Ersetzungsstrategien

Wenn ein Block in ein Cache-Set geladen werden soll, muss unter Umständen ein alter Eintrag verdrängt werden. Ist eine Zeile als ungültig [8] gekennzeichnet, ist sie der Ersetzungskandidat [8]. Andernfalls besteht das Ziel darin, sogenannte Kollision zu vermeiden. Damit bezeichnet man die Verdrängung eines Eintrages, der in Zukunft wieder referenziert wird (*Live Block*) [14]. Die Ersetzungsstrategie sollte also einen Kandidaten auswählen, der in Zukunft nicht oder erst spät wieder angefragt wird. Um solchen Entscheidungen nahe zu kommen, existieren verschiedene Möglichkeiten:

First-In, First-Out (FIFO): es wird der Eintrag verdrängt, der sich am längsten in dem Cache-Set befindet. Dieses Verfahren ist sehr einfach zu implementieren, es genügt lediglich ein Zähler, der bei jeder Ladeoperation inkrementiert wird und dann auf den nächsten zu ersetzenden Eintrag zeigt. Der Nachteil ist die schlechte Ausnutzung der Lokalität. Häufig gebrauchte Speicherwörter werden zwangsweise verdrängt, um später wieder nachgeladen zu werden. Die Strategie eigenet sich aber besonders für Befehls-Caches verbunden mit *Prefetching* [8].

Least-Recently-Used (LRU): es wird der Eintrag verdrängt, auf den am längsten nicht mehr zugegriffen wurde. Für dieses Verfahren existieren mehrere Implementierungen, auf die in Kapitel 3 näher eingegangen wird. Es ist mit einem sehr hohen Hardwareaufwand verbunden, da für jedes Set eine Liste mit den Zugriffsinformationen der Einträge gespeichert werden muss.

Random: die Auswahl innerhalb eines Sets erfolgt zufällig. Eine mögliche Implementierung wird in [8] vorgestellt. Für alle Sets existiert ein modulo-n-Zähler. Der Wert, der die zu ersetzende Cache-Zeile angibt, ist aber für alle Sets gleich. Der Zähler wird bei jedem Zugriff auf jedes beliebige Set inkrementiert.

Not-Most-Recently-Used (NMRU): um den hohen Hardwareaufwand von LRU zu begegnen, wurde das NMRU-Verfahren vorgestellt. Nicht mehr die gesamte Reihenfolge

[8] *Invalid* - das Valid-Bit ist nicht gesetzt

der Zugriffe wird protokolliert, sondern für jedes Set wird der Block markiert, der als letztes verwendet wurde. Die Auswahl unter den verbleibenden Blöcken erfolgt nach dem Zufallsprinzip. Häufig genutzte Einträge werde somit seltener verdrängt.

Zwar können die Ersetzungsstrategien entscheidend zur Effektivität des Caches beitragen, der Hardwareaufwand wird aber gerade bei steigender Assoziativität stark erhöht. Für große Caches zeigt sich eine Abnahme der Leistungsunterschiede zwischen den Strategien [3].

Mit der Ersetzungsstrategie wird das Phänomen der zeitlichen Lokalität ausgenutzt [6]. Ein Eintrag, auf den kürzlich zugegriffen wurde, wird wahrscheinlich in nächster Zeit wieder aufgerufen, während es für ältere Einträge immer unwahrscheinlicher wird. Least-Recently-Used nutzt genau dieses Verhalten.

2.2.3 Multi-Core-Problematik

Greifen mehrere Komponenten auf den Hauptspeicher zu, treten Kohärenzprobleme auf. Wenn das Copy-Back-Verfahren angewandt wird, so befinden sich in dem Cache eines Prozessors aktuellere Daten als im Hauptspeicher. Ein anderer Prozessor würde veraltete Daten (*stale data*) oder inkonsistente Zustände lesen. Schreibt anderseits eine Komponente Daten in den Hauptspeicher, so enthält der Cache eines anderen Prozessors unter Umständen veraltete Daten.

Wenn es im gesamten System nur eine Komponente mit einem Cache gibt (die anderen Teilnehmer können DMA- oder IO-Komponenten sein), so bieten sich folgende Lösungen an [7]:

1. gemeinsam genutzte Speicherabschnitte können als *Non-cachable data* gekennzeichnet werden. Hier erfolgt kein Caching.

2. bei jedem Taskwechsel wird der Cache geleert. Startet der Prozessor seine Ausführung erneut, muss er den Cache neu laden.[9]

3. eine Logik, die den Bus überwacht und entsprechend die Cacheeinträge verwaltet (*Bus Snooping*).

Wenn mehrere CPUs mit privaten Caches gleichzeitig auf einem Hauptspeicher arbeiten, ergeben sich komplexere Kohärenzprobleme. Allerdings kann auf Caches in den meisten Fällen nicht verzichtet werden. Gerade bei Systemen mit einem Adressbus für alle Prozessoren (busorientierte Systeme) sind sie zur Gewährleistung einer angemessenen Ausführungsgeschwindigkeit unabdingbar [6].

Write-Through stellt die einfachste Lösungsvariante [6] [7] dar. Der Speicher ist dabei immer auf dem aktuellsten Stand. Für einen Schreibzugriff existiert ein Snoop-Mechanismus. Jeder Cache-Controller überwacht den Bus und stellt die Änderung fest. Es existieren zwei Reaktionsmöglichkeiten:

[9] man spricht von *Cache-Flush*, wenn alle mit dirty gekennzeichneten Daten zurückgeschrieben und danach alle Valid-Bits gelöscht werden, und von *Cache-Clear*, wenn ohne Zurückschreiben (bei Write-Through) alle Eintrage für ungültig erklärt werden

1. bei der Invalidierungsstrategie werden die Daten, die woanders geändert wurden, im Cache des Prozessors invalidiert. Das Protokoll setzt voraus, dass nur eine CPU zu einem Zeitpunkt die Cache-Line schreibt. Der Prozessor muss sicherstellen, dass er die einzig valide Kopie der Datei besitzt.

2. bei der Aktualisierungsstrategie kann der Cache die neue Zeile übernehmen und als valid markieren. Diese *Broadcast-Write-Through-Policy* sollte nur für eine kleine Prozessoranzahl eingesetzt werden, da eine hohe Last entsteht.

Neben dem Write-Through-Ansatz existieren verschiedene weitere Kohärenzprotokolle, die auch Copy-Back erlauben:

MESI-Protokol: die Caches synchronisieren sich über eine Snoop-Logik und spezielle Signale [7]: dem *Invalidate*-Signal, dem *Shared*-Signal und dem *Retry*-Signal. Jede Cache-Line kann nun in jedem Cache einen von vier Zuständen annehmen, der ihren Status beschreibt.

Ungültig (*invalid*): der Cache-Eintrag ist ungültig

Gemeinsam (*shared*): der Cache-Eintrag ist gültig und befindet sich in mehreren Caches

Exklusiv (*exclusive*): der Cache-Eintrag ist gültig und befindet sich nur in diesem Cache

Modifiziert (*modified*): der Cache-Eintrag ist gültig, der Speicher ist ungültig und der Eintrag ist nur in diesem Cache

Liest ein Prozessor eine Zeile, so wird das von den anderen Teilnehmern über die Bus-Leitung registriert. Hat ein Cache die Zeile bereits geladen, so sendet er das Shared-Signal und zeigt an, dass der Eintrag in den Zustand *shared* versetzt werden soll. Meldet sich kein anderer Teilnehmer, so setzt der lesende Cache die Zeile auf den Zustand *exclusive*. Liest ein anderer Prozessor die Zeile, wird dies über einen Bus-Snoop festgestellt und die Zeile mit *shared* markiert. Schreibt eine CPU die Zeile, so sendet er ein Invalidate-Signal auf die Bus-Leitung. Die Zeile ist als modifiziert im schreibenden Cache und als ungültig in den Übrigen gekennzeichnet. Will eine andere CPU die modifizierte Zeile lesen, so wird er von dem Besitzer angewiesen zu warten (Retry-Signal), bis er sie in den Speicher zurückgeschrieben hat. Danach ist die Zeile im Zustand *shared*. Will die andere CPU schreiben, so ist die Zeile im ersten Cache nach dem Zurückschreiben invalidiert.

Das MESI-Protokoll ist sowohl für Copy-Back, wie auch für Write-Through anwendbar. Zusätzlich werden Mischformen unterstützt [7].

MEI-Protocol: hierbei wird auf den Shared-Zustand verzichtet. Es darf nur eine Kopie einer Cache-Line geben [10].

MOESI-Protocol: es wird dem MESI-Protokol ein weiterer Zustand *owned* hinzugefügt. Dieser markiert einen Eintrag, der *modified* und *shared* vorliegt. Er wird erreicht, wenn eine als modifiziert gekennzeichnete Cache-Line gelesen wird. Die Line liegt nur beim ursprünglichen Prozessor als owned vor, bei allen anderen als *shared*. Ändert ein anderer Prozessor den Wert, wird der Eintrag in den übrigen Caches invalidiert [9].

Bei einer NUMA-Architektur (*non-uniform memory access*) ist kein gemeinsamer Hauptspeicher vorgesehen, es existieren lediglich Referenzen zu entfernten Speichern. Ein Zugriff auf ein fremdes Datum wird über das Verbindungsnetzwerk der Knoten untereinander beantwortet. Bei NC-NUMA (*non-cache NUMA*) wird gänzlich auf Caching verzichtet. Bei CC-NUMA (*cache-coherent NUMA*) existiert in der Regel ein verzeichnisbasiertes Mehrprozessorsystem (*Dicteionary-based Multiprozessor*) [6]. Eine Datenbank speichert dabei den Zustand und den Ort der Daten, wobei sich hier der Flaschenhals des Datenbankzugriffs ergibt. Dies verstärkt sich noch, wenn Zeilen bei mehreren Knoten untergebracht sind.

Die COMA-Architektur (*Cache Only Memory Access*) sieht keine Heimatspeicher mehr für Seiten vor. Sie können frei im System migrieren und in den Caches vorgehalten werden, dessen Knoten mit der Bearbeitung beschäftigt sind.

2.2.4 Adressierung

Es ergibt sich die Frage nach der Adressierung [7] des Caches. Bei virtuellen Speichersystemen arbeitet der Prozessor in einem Adressraum, der sich von dem physikalisch adressierten Hauptspeicher abhebt. Vor jedem Zugriff muss eine Übersetzung der Seitenadressen stattfinden. Ein Cache kann sowohl physikalisch, virtuell oder gemischt adressiert sein. Der Vorteil der virtuellen Adressierung ist, dass ein Cache-Zugriff gleichzeitig mit der Übersetzung stattfinden kann. Probleme ergeben sich bei der Sicherstellung der Kohärenz. Snoop-Logik, die physikalisch-adressierende Komponenten belauscht, wäre unwirksam. Hier ist zusätzliche Steuerlogik nötig. Weitere Probleme ergeben sich durch mehrere Prozesse auf einer CPU. Jeder hat einen eigenen virtuellen Adressraum, der sich aber mit anderen überdeckt. Als Lösung käme das Löschen des Caches vor jedem Prozesswechsel oder das Integrieren von ID-Informationen in die virtuelle Adresse in Frage. Ein weiteres Problem stellt *Address-Aliasing* dar, d.h. unterschiedliche virtuelle Adressen verschiedener Tasks zeigen auf die selbe reale Adresse. Dabei kann nicht mehr sichergestellt werden, ob ein gecachter Eintrag nicht bereits durch den anderen Prozess geändert wurde.

Bei einer virtuell/physikalischen Adressierung bildet der Tag den realen Anteil, das Set wird durch die virtuelle Adresse ausgewählt (*virtually addressed, physically tagged*). Die Adressumsetzung und die Cache-Adressierung können hierbei parallel erfolgen.

2.2.5 Systemaufbau

Für die Platzierung des Caches existieren mehrere Varianten. Der Cache kann in Reihe (*Look-through-Cache/Inline-Cache*) geschaltet sein. Diese Lösung ist typisch, wenn mehrere Caches

auf einen Bus zugreifen. Eine Hauptspeicheranfrage wird vom Cache-Controller abgefangen und verarbeitet. Der Speicherbus wird nicht belastet. Der Cache kann dabei mit einem schnellen Bus angesteuert werden. Die Alternative ist eine Parallelschaltung (*Look-aside-Cache*). Hauptspeicher- und Cachezugriff finden gleichzeitig statt, wobei nach einem Cache-Hit die Anfrage an den Hauptspeicher abgebrochen wird. Ein Nachteil dieser Variante stellt die stärkere Auslastung des Speichers [7] dar. Demgegenüber steht ein Geschwindigkeitsgewinn, da im Falle eines Fehlerschlags der Speicherzugriff bereits läuft.

2.2.6 Translation Lookaside Buffer

Bei einem Translation Lookaside Buffer handelt es sich um einen sogenannten *Deskriptor-Cache* [7]. Die Adressumsetzung von virtuellen in physische Adressen wird zwischengespeichert. Es wird dabei ebenfalls das Lokalitätsprinzip ausgenutzt. Kürzlich benutzte Seiten werden mit hoher Wahrscheinlichkeit in Zukunft wiederverwendet. Der Cache ist in der Regel vollassoziativ mit einer kleinen Anzahl von Einträgen (z.B. 64 Einträge [6]) aufgebaut. Als Tag dient die virtuelle Adresse, die Physische bildet den Eintrag. Ist eine Adresse nicht im TLB enthalten, so wird dies wie ein Cache-Miss behandelt. Dabei findet ein Zugriff auf den Speicher statt und der neue Wert kann geladen werden. Es existiert eine Ersetzungsstrategie, die einen alten Wert auswählt und löscht.

2.2.7 Caches für Java-Prozessoren

Im Folgenden sollen zwei Forschungsarbeiten kurz vorgestellt werden, die sich mit Caches für objektorientierte Umgebungen beschäftigten. In *An object-aware memory architecture* [11] der Sun Labs wurde eine mögliche Speicherarchitektur vorgeschlagen, die zusammen mit einer virtuellen Maschine Besonderheiten der objektorientierten Sprache unterstützt und ausnutzt. Im Speziellen geht es um einen virtuell adressierten *Object-Cache*[10].

Jede Cache-Line enthält ein Java-Objekt. Der Nachteil, dass dadurch keine benachbarten Objekte bei einem Zugriff gecached werden, wird dadurch entkräftet, das die Allokation direkt im Cache erfolgt (Copy Back). Da Studien besagen, es herrsche wenig Lokalität zwischen alten Objekten, würde dieser Nachteil wenig Einfluss haben. Er wirke nur auf Objekte, die alt genug sind, um bereits aus dem Cache verdrängt worden zu sein.

Ein Objekt wird durch eine Objekt-ID (OID) referenziert. Mittels des Offsets findet der Zugriff innerhalb des Objektes statt. Hierbei handelt es sich um die virtuelle Adresse. Die Cache-Adresse ist eine Konkatenation aus beiden Teilen. Die Index-Bits sind als Kombination aus den höchstwertigsten Offset-Bits und den niederwertigsten OID-Bits mittels XOR-Verknüpfung gebildet.

Weiterhin wird ein sogenannter Objekt-*Skew* vorgeschlagen, der negative Offsets in Positive umwandelt, um die Verwendung von mehreren Cache-Lines pro Objekt zu meiden. Entsprechende Informationen würden in der OID gespeichert und bei der Adressgenerierung berück-

[10] der Cache kann über eine Adresserweiterung auch physisch adressierte Daten speichern

sichtigt. In der Arbeit wird außerdem ein Verfahren zur In-Cache-Garbage Collection vorgeschlagen.

In *Object-Oriented Architectural Support for a Java Processor* [12] wird ein Vorschlag einer Prozessorarchitektur zur direkten Ausführung von Java in Hardware gemacht. Die Speicherhierarchie sieht einen 8 KB virtuell adressierten Objekt-Cache, einen 4 KB direkt abbildenden Befehls-Cache und einen 4 KB direkt abbildenden Daten-Cache vor.

Das Ziel des Objekt-Caches ist, keine Einbußen durch Indirektionen beim Objektzugriff entstehen zu lassen, aber dennoch schnelle Relokationen von Objekten zu erlauben.

Da 32 Bit für eine Referenz und 32 Bit für den Offset einen zu großen Tag ergeben, wurden Analysen der Objektgröße durchgeführt. Demnach ist ein 8 Bit Offset ausreichend für 99 % der Objekte und 12 Bits für Arrays. Größere Objekte würden durch die direkte Adresse verwaltet oder mittels Compilerunterstützung in kleinere Objekte zerteilt werden können.

Weiterhin wird eine *Object Table* vorgeschlagen, der entsprechend einem *Translation Lookaside Buffer* die Liste der kürzlichen Adressumsetzungen enthält.

Schließlich wurden für den Cache verschiedene Varianten zur Bestimmung der Index-Adresse diskutiert. Diese Ausführung findet sich ausführlicher in [13]. Die besten Ergebnisse wurde bei einer XOR-Verknüpfung der niederwertigsten Referenz-Bits mit den höchstwertigsten Offset-Bits erreicht.

2.3 Analytische Cache-Modelle

2.3.1 Three C's Modell

Das *Three-C*'s-Modell [3] eignet sich zur Klassifikation von Caches-Fehlschlägen und damit zur Ableitung von Optimierungsmöglichkeiten. Ein Fehlschlag wird in eine von drei Kategorien eingeteilt:

Cold/Compulsory Miss: auf einen Eintrag wird das erste Mal zugegriffen. Er befindet sich nicht im Cache. Dieser Fehler lässt sich schwer umgehen. Verschiedene *Prefetching*-Strategien können angewandt werden.

Capacity Miss: die Cache-Größe reicht nicht aus. Ein altes Datum, auf das wieder zugegriffen wird, ist verdrängt wurden. Die Lösung ist ein größerer Cache.

Conflict Miss: ein Datum ist verdrängt und wird wieder referenziert (Kollision). Im Gegensatz zu einem Capacity-Miss wäre dieser in einem voll-assoziativen Cache mit Least-Recently-Used-Ersetzungsstrategie nicht aufgetreten. Der Wert ist kurz vor der Verdrängung referenziert wurden. Als Lösungen kommen in Frage, die Assoziativität zu erhöhen oder die Ersetzungsstrategie zu erweitern.

2.3.2 Modell nach Agarwal, Horowitz und Hennessy

Im Jahr 1989 wurde das analytisches Cache-Modell von Anant Agarwal, Mark Horowitz und John Hennessy vorgestellt [14]. Es hatte zum Ziel, schnell und ohne großen Simulationsaufwand bzw. ohne die Notwendigkeit der Herstellung von Hardwareprototypen Aussagen über die Leistungsfähigkeit verschiedener Cache-Architekturen zu liefern. Es kann für Einprozess- und Mehrprozess-Systeme angewandt werden. Dazu wird eine vorhandene Adressfolge in Zeitabschnitte gleiche Länge, die sogenannten *Time Granules* unterteilt. Über jede Time Granule werden Parameter bestimmt, die in die Berechnung der Miss-Rates einfließen:

- die durchschnittliche Anzahl an verschiedenen Speicherzugriffen pro tg
- die Anzahl unterschiedlicher Speicherzugriffe der Adressfolge
- die Wahrscheinlichkeit, dass nach einem Zugriff auf eine bestimmte Adresse ein Zugriff auf die Folgeadresse erfolgt (Stufe 1)
- die Wahrscheinlichkeit, dass wenn k mal jeweils auf die Folgeadresse zugegriffen wurde, ein weiteres mal auf die Folgeadresse zugegriffen wird (Stufe $k+1$)
- die Anzahl an Time-Granules

Das Modell verwendet ein zweistufiges Markov-Modell zur Darstellung der örtlichen Lokalität. Es beschreibt die Wahrscheinlichkeiten für Adressdurchläufe mit der Schittweite 1 (*run*). Weiterhin wird mit Wahrscheinlichkeiten für die Blockzuordnung gearbeitet. Man geht davon aus, dass ein Block eine gleiche Wahrscheinlichkeit hat, in ein bestimmtes Set gemappt zu werden. Daraus werden potentielle Kollisionen bestimmt. Zur genaueren Charakterisierung wird der Wert der *Collision rate* einberechnet, die angibt, wie oft in einer Time Granule aus einer potentiellen eine wirkliche Kollision wird. Die Bestimmung von c geschieht über repräsentative Caches.

Das Modell erlaubt die Berechnung von Hit-Rates für verschiedene Cache-Organisationen. Als Parameter sind die Anzahl an Sets, der Grad der Assoziativität, die Block-Größe, das Multi-Programming-Level und die Größe des Time-Slices eines Prozesses möglich. Die Zielfunktion ist zusammengesetzt aus Teilfunktionen, die jeweils verschiedene Fehler klassifizieren:

Start-up effects: der Prozess wird das erste Mal gestartet. Die initialen Arbeitsdaten müssen geladen werden.

Non-stationary behavior: die Arbeitsdaten des Prozesses ändern sich mit der Zeit. Verschiedene Referenzen werden das erste Mal außerhalb des Startup-Phase aufgerufen.

Intrinsic interference: eine verdrängte Referenz wird wieder aufgerufen.

Extrinsic Interference: eine durch einen anderen Task verdrängte Referenz wird wieder aufgerufen.

Für den Mehrprozessfall wird Round-Robin-Scheduling mit festen Zeitintervallen angenommen. Die Bestimmung der extrinsischen Interferenz kann sowohl für LRU- wie für zufällige Ersetzung geschehen.

In der Arbeit werden die mit dem Modell ermittelten Ergebnisse mit Simulationen verglichen. Es zeigen sich gute Fehlerraten für direkt abbildende Caches, die abhängig von der Block-Größe zwischen 4 und 15 % liegen. Für 2-Wege assoziative Caches ergibt sich ein mittlerer Fehler von 13 bis 23 %.

2.3.3 Modell nach Brehob und Enbody

Das analytische Modell nach Brehob und Enbody [15] [16] verwendet das Modell der Stack Distanz zur Klassifikation von zeitlicher Lokalität. Daraus wurde ein Cache-Modell entwickelt, dass aus den Stack Distanzen einer Adressfolge eine Hit-Rate einer bestimmten Cache-Konfiguration ermitteln soll.

2.3.3.1 Stack Distanz

Die Stack Distanz ist ein Modell zur Charakterisierung von zeitlicher Lokalität eines Programms. Wird eine Referenz in der Adressfolge das erste Mal aufgerufen, wird sie auf dem Stack abgelegt. Bei einem weiteren Aufruf wird sie aus ihm entfernt und wieder als oberstes Element hinzugefügt. Die Tiefe der Referenz im Stack gibt an, wie viele unterschiedliche Adressen in der Zwischenzeit verwendet wurden. Dieser Wert wird als Stack Distanz bezeichnet.

Für jeden Zugriff in einer Adressfolge kann eine einzelne Distanz berechnet werden. Brehob und Enbody beschreiben die *Stack Distance Distribution* als eine Funktion, die zu jeder Stack Distanz ihren Anteil am Vorkommen in der Adressfolge bestimmt.

Die Diagramme 2.4 und 2.5 zeigen fiktive Verteilungen für zwei Adressfolgen von unterschiedlichen Programmen. Im linken Diagramm sieht man eine sehr große zeitliche Lokalität, da ein hoher Anteil der Zugriffe auf kürzlich verwendete Referenzen stattfindet. Das rechte Programm hingegen hat im Durchschnitt große Abstände zwischen zwei Zugriffen auf eine Referenz.

Die *Cumulative Stack Distance Distribution* summiert für jede Stack Distanz das Vorkommen dieser und aller kleineren Distanzen in der Adressfolge. Das Diagramm 2.6 zeigt für zwei fiktive Programme die CSDD-Werte an. Es lassen sich bereits erste Rückschlüsse auf die Wirksamkeit eines Caches schließen. Bei 256 betragen die Verteilungen 0.5 und 0.75. Das heißt, ein vollassoziativier LRU-Cache mit 256 Einträgen Größe hätte für die untere Adressfolge eine Hit-Rate von 50 % und für die oberere könnten 75 % aller Anfragen aus dem Cache beantwortet werden.

Die Stack Distanz kann auch zur Klassifikation der Fehler nach dem Three-C's Modell verwendet werden. Erfolgt zum Beispiel bei einem Cache mit 256 Einträgen ein Zugriff mit einer Distanz kleiner als 256, so liegt ein Conflict-Miss vor. Bei einer Stack Distanz größer als 256

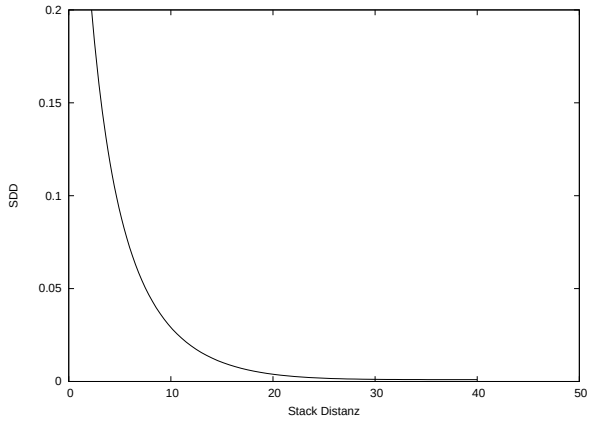

Abbildung 2.4: Ein Beispiel hoher zeitlicher Lokalität.

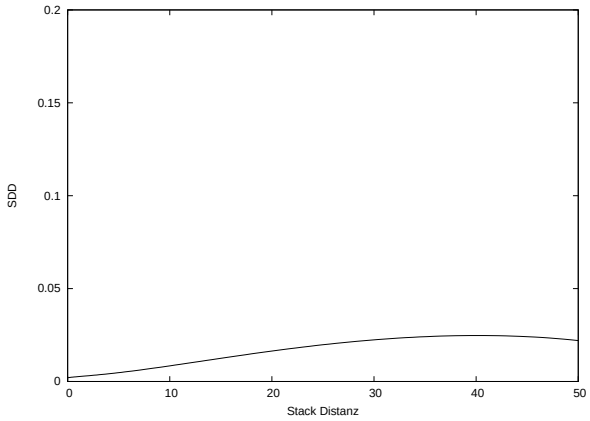

Abbildung 2.5: Ein Beispiel niedriger zeitlicher Lokalität.

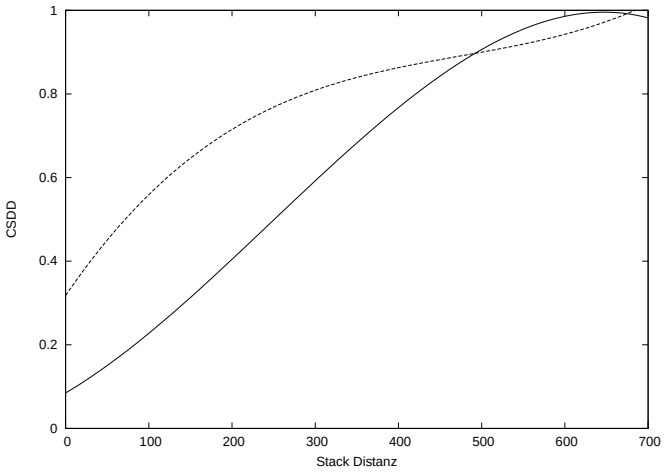

Abbildung 2.6: Beispiele für die Cumulative Stack Distance Distribution.

ist ein Capacity-Miss aufgetreten. Bei einer Stack Distanz von unendlich, davon spricht man, wenn die Referenz vorher nicht auf dem Stack vorhanden war, liegt ein Cold-Miss vor.

2.3.3.2 Cache-Modell

Das Cache-Modell berechnet für jeden Zugriff aus seiner Stack Distanz eine Treffer-Wahrscheinlichkeit. Als Cache-Parameter lassen sich die Assoziativität und die Cache-Größe angeben. Die Blockgröße kann indirekt durch die Stack Distanz dargestellt werden. Dazu werden nicht die Hauptspeicheradressen, sondern die Blockadressen ohne die Wortauswahlbits verwendet. Das heißt, zwei Adressen, deren Wort sich im selben Cache-Block befindet, werden als gleich behandelt. Eine hohe Block-Größe würde bei entsprechender örtlicher Lokalität mehr gleiche Zugriffe bedingen und die Hit-Raten erhöhen.

Mit der Stack Distanz wird errechnet, ob eine Referenz zwischen zwei Zugriffen verdrängt wurde. Das Modell geht von der Annahme aus, dass ein Eintrag eine gleich Wahrscheinlich-

keit hat, in ein bestimmtes Set gemapped zu werden. Diese Vermutung widerspricht zwar der örtlichen Lokalität, die Abweichung soll aber zu vernachlässigen sein.

Es wird von eine Fehlerrate von 1-3 % ausgegangen. Die vorgestellten Messungen zeigen bei Integer-Benchmarks gute Ergebnisse, bei Floating-Point-Berechnungen rauschen die Resultate jedoch stark.

2.3.4 Look-Back-Window

In [17] wird versucht, sowohl zeitliche, wie örtliche Lokalität zu quantifizieren. Obwohl die Arbeit auf den Bereich des High-Performance-Computings abzielt, ist sie dennoch hier anwendbar.

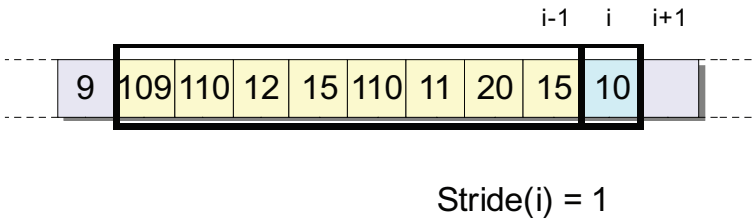

Abbildung 2.7: Ein Look-Back-Window.

Während für die zeitliche Lokalität der Ansatz der Stack Distanz[11] gegangen wird, quantifizieren sie die örtliche Lokalität mittels eines *Look-Back-Windows*. Mit ihm wird die Schrittweite eines Speicherzugriffs bestimmt. Das Look-Back-Window enthält die kürzlich verwendeten Referenzen. Die Schrittweite eines neuen Zugriffes wird nun definiert als der minimale Abstand zu den Referenzen des Windows. Werden diese Werte für eine Adressfolge gesammelt, lassen sich ähnlich der Stack Distanz Verteilungsfuntkionen zur Einschätzung der örtlichen Lokalität erstellen. In der Arbeit wird angestrebt, einen einheitlichen Score-Wert zur Charakterisierung der Lokalität zu erhalten. Dies wird durch die Summation der gewichteten Anteile bestimmter Schrittlängen erreicht.

[11] hier in abgeänderter Form als *Reuse-Distance* vorgestellt

3 Testumgebung

3.1 SHAP Bytecode-Prozessor

Bei der SHAP (Secure Hardware Agent Platform) [18] handelt es sich um eine eingebettete Mikroarchitektur, die zur Ausführung von Java in Hardware unter Echtzeit-Bedingungen geeignet ist. Ihre Hauptkomponenten bilden die CPU zur Ausführung des Java-Bytecodes, der Memory Manager zur Verwaltung des Java-Heaps und zur nebenläufigen Garbage Collection in Hardware, ein Methoden-Cache, ein integrierter Memory-Controller für den Zugriff auf externen Speicher und einen Wishbone-Bus zum Anschluss verschiedener externer Geräte.

Der Zugriff auf ein Objekt durch den Kern geschieht durch die Angabe der Objektreferenz und eines Offsets [19]. Die Referenz wird nach dem Indirektionsprinzip in eine physische Adresse umgerechnet. Das hat den Vorteil, dass Referenzen trotz Verschiebeoperationen durch die Garbage Collection erhalten bleiben. Wird auf ein neues Objekt zugegriffen, so fragt der Speicherport die Basisadresse ab und hält sie fortan. Man spricht vom *Aktivieren der Referenz*. Der Speichermanager ist mit einem Multi-Port-System ausgestattet, um dem Prozessor, dem Methodencache und der DMA je eine eigene Schnittstelle zur Verfügung zu stellen und unnötige Adressumsetzung zu vermeiden.

3.1.1 Weitere Java-Prozessoren

Im Folgenden wird eine kurze Auswahl der Cache-Architekturen weiterer Java-Prozessoren vorgestellt:

picoJava I: der von Sun Microsystems entwickelte Prozessor enthält einen Befehl- und einen Daten-Cache. Beide sind für die Größen von 0 bis 16 KB konfigurierbar.

picoJava II: die zweite Generation des von Sun entwickelten Prozessors verfügt ebenfalls über einen Befehls- und einen Daten-Cache. Der Daten-Cache ist als Zwei-Wege-assoziativer Cache mit einer Cache-Line-Größe von 16 Byte implementiert. Er ist für Größen von 0 bis 16 KB konfigurierbar.

aJile: dieser von aJile Systems entwickelte Microcontroller ist für die direkte Bytecodeausführung unter Echtzeitbedingungen geeignet. Während der aJ-100 [23] noch auf einen Cache verzichtet, enthalten der aJ-102 [24] und der aJ-200 [25] [1] einen 32 KB On-Chip Unified Cache.

[1] Verfügbar ab 2009

Java Optimized Processor: der JOP [26] implementiert die virtuelle Maschine in Hardware. Er enthält einen Stack und einen Methoden-Cache, hat jedoch keine Daten-Cache für den Objekt-Heap.

3.2 Testapplikationen

Der SHAP ist in der Lage, die CLDC[2]-API auszuführen. Für die Cache-Analyse wurde versucht, verschiedenartige passende Anwendungen bereitzustellen, die nach Möglichkeit zur selbstständigen Arbeit ohne Input in der Lage sind. Im SHAP-Projekt wurden bereits folgende Testanwendungen verwendet:

Sudoku: das Programm ermittelt alle Lösungen eines gegebenen Sudokufelds. Dieses wird in einem einzelnen Objekt gespeichert. Die Ergebnisse werden rekursiv berechnet und mittels *wait/notfiy* einem Ausgabe-Thread zur Verfügung gestellt.

FScript: hierbei handelt es sich um einen Interpreter der Skriptsprache FScript, der einen gegebenen Beispielalgorithmus abarbeitet. Die Anwendung besteht aus einem Lexer und einem Parser. Bestimmte Daten werden während der Ausführung für längere Zeit oder ständig gehalten. Dazu gehören eine Variablenliste, eine Parameterliste oder eine Funktionsmap. Während der Ausführung werden eine große Anzahl an Objekten erzeugt und verdrängt. Das Working Set sollte höheren Schwankungen unterlegen sein.

Bei dem Java Grande Forum [27] handelt es sich um eine Vereinigung mit dem Ziel, Java als Sprache zu analysieren und daraus neue Ansätze für Standards zu entwickeln. Sie ist im Bereich des Hochleistungsrechnen tätig. Von ihr entwickelt und gefördert wurde das Java Grande Framework, welches Benchmarks für sogenannte *Grande Applications* zur Verfügung stellt. Dabei handelt es sich um Anwendungen, die besondere Anforderungen im Hochleistungsrechnen, im verteilten und parallelen Rechnen, und im speicher- und datenintensiven Rechnen stellen.
Da sich die Anwendungen der Java Grande Benchmark Suite gut skalieren lassen und als Open Source vorliegen, wurden einige von ihnen angepasst für die Cache-Analyse herangezogen. Folgende sequentiell-arbeitende Anwendungen wurden ausgewählt:

HeapSort: der Sortieralgorithmus verarbeitet 15000 Integer-Werte aus einem Array. Anschließend erfolgt eine Validation des Ergebnisses. Der Algorithmus arbeitet iterativ.

Crypt: hierbei handelt es sich um die IDEA- (International Data Encryption Algorithm) Ver- und Entschlüsselung. Es werden 15000 Werte verarbeitet. Danach erfolgt eine Validation. Das Programm arbeitet mit Byte-Arrays für die Daten und verschiedenen Short- und Integer-Arrays für die Schlüssel. Jeder 64-Bit-Datenblock geht durch 8 Verarbeitetungsrunden. Die Berechnung erfolgt sequentiell.

[2] Java Connected Limited Device Configuration (CLDC): JSR 30, JSR 139

RayTracer: ein 3D-RayTracer-Benchmark. Es enthält mehrere Klassen, die verschiedene Aspekte der 3D-Umgebung repräsentieren, wie `View`, `Vec`, `Light`, `Ray`, etc. Nach dem Aufbau der Objekte zur Repräsentation erfolgt die Abarbeitung in zwei Schleifen zur Erzeugung des 2D-Bildes.

Weiterhin wurden folgene Mutlithread-Anwendungen ausgewählt:

Crypt: die IDEA-Verschlüsselung. Mehrere Thread arbeiten gleichzeitig an Teilen des selben Textes und unter Umständen mit den selben Schlüssel-Arrays.

Crypt & HeapSort: hierbei handelt es sich um eine Anwendung, die HeapSort und Crypt jeweils einen Thread zuordnet und nebenläufig ausführt.

Programmteile, die eine Unterstützung von double-Gleitkommaoperationen verlangten, wurden entweder entfernt oder durch die SHAP-interne `SoftFloat`[3]-Klasse ersetzt.

3.3 Protokollierungseinheiten

Der erste Schritt bestand darin, für bestimmte Beispielanwendungen die Folge von Adressaufrufen (*Traces*) aufzuzeichnen. Dazu wurden Protokollierungseinheiten direkt im Entwurf des SHAP-Prozessors implemtiert. Die Messung erfolgte dann auf einem Xilinx Spartan 3E XC3S500E FPGA. Weiterhin wurden Protokollierungseinheiten für den an der Technischen Universität Dresden entstandenen Prozessorsimulator DITO [29] entwickelt.

3.3.1 SHAP-Protokollierungseinheiten

Im SHAP-Prozessor wurden Protokollierungseinheiten an drei Stellen implementiert:

1. zwischen Kern und Speicherport[4]
2. zwischen Speicherport und Speichermanager[5]
3. zwischen Speichermanager und Speicher-Block-Manager[6]

Jede Einheit besteht aus zwei Komponenten. Eine dient der Aufzeichnung der Folge der aufgerufenen Adressen und der Art des Aufrufs. Eine Zweite protokolliert Timing-Werte, indem sie für jede Operationen die Vorkommen und die Gesamtausführungsdauer bestimmt. Daraus können später Durchschnittswerte errechnet werden.

Zur Ermittelung der Adressfolge wurde der Prozessor von seiner ursprünglichen Geschwindigkeit von 45 MHz auf 1,125 MHz herunter getaktet. Eine weitere Verlangsamung war wegen der DCM-Begrenzungen des FPGAs nicht möglich.

[3] SoftFloat simuliert Float-Gleitkommaoperationen gemäß IEEE-754 durch 32-Bit-Integer-Werte
[4] im Folgenden als Kern bezeichnet
[5] im Folgenden als Port bezeichnet
[6] im folgenden als Speicher bezeichnet

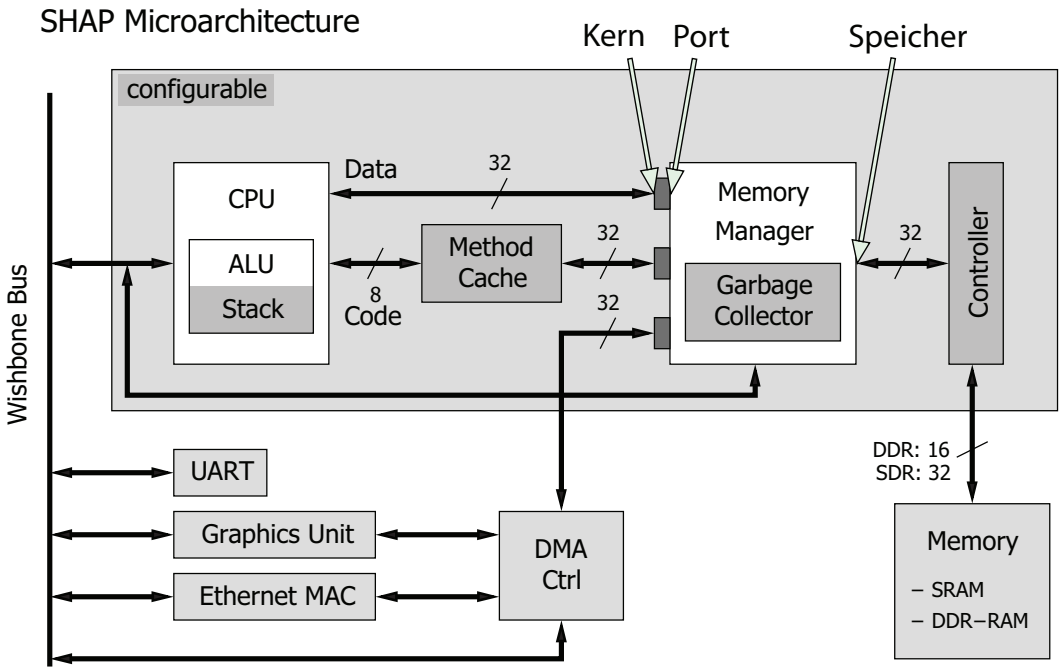

Abbildung 3.1: SHAP-Protokollierungseinheiten

Da der DDR-SDRAM aufgrund seiner Refresh-Zeiten nicht mit herunter getaktet werden konnte, ergab sich das Problem, dass Speicheranfragen beschleunigt wurden und die verlangsamte Programmausführung weniger Wartetakte enthielt. Dies wiederum beeinflusste den Scheduler. Um die Störungen zu minimieren, wurden künstlich Wartezeiten erzeugt, indem die Speicherantworten durch ein Schieberegister bzw. durch eine Zähllogik verlangsamt wurden.

Die Übertragung der Adressen erfolgte direkt während der Programmausführung mittels USB zu einem Desktop-PC. Bei der Protokollierungseinheit des Speichers ergab sich bei einigen Testanwendungen das Problem, dass die reale Geschwindigkeit der USB-Verbindung von 4 MBit/s nicht genügte, die Adressen zu übertragen, ohne den Zwischenspeicher auf dem FPGA zu überlasten. Auch der Wechsel auf eine Ethernet-Verbindung zeigte kaum eine Verbesserung, da die Adressaufrufe nicht gleichverteilt, sondern gerade in der Startphase des Programms stark gehäuft auftraten. Die Lösung bestand in der Unterteilung der Adressfolge nach ihren jeweiligen Aufrufern (CPU, Method-Cache, Segment-Manager, Referenz-Manager, DMA), die sich an der Speicherschnittstelle über Request-Tags ermitteln ließen. Für jeden wurde eine eigene Adressfolge protokolliert. Anschließend wurde ein Trace aufgezeichnet, der nur die Folge der Aufrufer (3-Bit-kodiert) enthält. Mit diesen Informationen konnte der ursprüngliche Adresstrace wieder hergestellt werden. Da die Adressaufrufe bei verschiedenen Abläufen nicht deterministisch sind, ergab sich eine Abweichung, die aber unter 0.05 % liegt.

Für die Protokollierung der Timing-Werte war keine Verlangsamung des Prozessors nötig, da die entsprechenden Daten während der Programmausführung gesammelt und nach Abschluss übertragen wurden.

Im folgenden werden die einzelnen Protokollierungseinheiten näher beschrieben:

Kern-Protokollierungseinheit: das entsprechende Zugriffsprotokoll ist in [20] beschrieben. Ein Zugriff erfolgt in zwei Teilen: Zuerst wird die Objektreferenz aktiviert. Der Kern kann dann mit einem entsprechenden Offset Lese- und Schreiboperationen innerhalb des Objektes veranlassen. Die Protokollierungseinheit registriert die Befehle *Referenz setzen*, *Lese-Offset setzen* und *Schreib-Offset setzen* und kodiert es als 16 Bit-Vektor. Jeweils zwei Zugriffe werden gleichzeitig per USB übertragen. Für je zwei Byte wird ein Paritätsbit in ungerader Parität ermittelt. Aller fünfzehn Werte wird ein 15-Bit-Paritätsvektor übertragen.

Die Timing-Komponente protokolliert die Zeit-Werte. Sie erkennt zusätzlich den Befehl *Daten schreiben*.

Port-Protokollierungseinheit: die Protokollierungeinheit überträgt die Daten in dem Format wie die Kern-Einheit. Der Unterschied besteht darin, dass ein Schreibbefehl durch ein Kommando erfolgt.

Speicher-Protokollierungseinheit: das entsprechende Zugriffsprotokoll ist in [20] beschrieben. Die Protokollierungseinheit bietet die Möglichkeit, die Adressfolge nach bestimmten Request-Tags zu selektieren. Zu jedem Speicheraufruf werden der Aufrufer, die Art des Aufrufes, die Adresse und die Blocklänge übertragen. Unterscheiden sich zwei Aufrufe nur in den untersten 14 Bit der Adresse, werden lediglich diese übertragen und mit einer speziellen Kennzeichnung versehen. Somit kann die Menge an übertragenen Daten reduziert werden.

Es ist nicht möglich, mehrere Protokollierungseinheiten gleichzeitig zu betreiben bzw. die Adressfolge und die Timing-Werte parallel zu ermitteln.

Für die Protokollierung war eine Modifikation der Programme nötig. Alle Operationen, die Daten auf die Standardausgabe schreiben, wurden entfernt. Geschähe dies nicht, führten sie zu einer Blockierung des Programms, dessen Polling-Anfragen auf die UART-Schnittstelle die Ergebnisse beeinträchtigt hätten.

Des Weiteren wird zu Beginn der Programmausführung und nach dessen Abschluss ein Signal über den Wishbone-Bus an die Protokollierungseinheit gesandt. So wird ihr mitgeteilt, die Protokollierung zu starten und zu beenden. Damit kann einerseits vermieden werden, dass das Laden des Programms protokolliert wird. Andererseits kann den Timing-Modulen mitgeteilt werden, wann die gesammelten Werte übertragen werden können.

3.3.2 DITO-Protokollierungseinheiten

In dem Prozessorsimulator wurden Protokollierungseinheiten am Kern und am Speicher implementiert. Die eigentlichen Module wurden direkt in den Simulator in C++ integriert. Sie registrieren Listener an bestimmten Stellen des virtuellen Prozessormodells, welches sie

über Speicher-Anfragen informiert. Dazu waren geringfügige Modifikation an den TADL-Beschreibungen notwendig, die aber keinen Einfluss auf die Simulation des Prozessors haben. Die Protokollierung erfolgt an beiden Meßpunkten gleichzeitig. Es werden neben den Adresszugriffen ebenfalls Timing-Werte gesammelt.

3.4 Cache-Simulator

Der Cache-Simulator ist ein in Java entwickeltes Tool, das verschiedene Teilaufgaben kombiniert:

- die Analyse von zeitlicher und örtlicher Lokalität in Programmen
- die Simulation eines Translation Lookaside Buffers
- die Simulation von verschiedenen Cache-Architekturen verbunden mit analytischen Modellen

Das Programm ist streng modular aufgebaut und erlaubt Erweiterungen auf einfache Weise. Es folgt dem Model-View-Controller-Prinzip, das die Trennung von Aufgaben der Darstellung, des Modells und der Steuerung der Simulation vorsieht. Die Darstellung, also die Repräsentation nach außen, läuft über die zentrale Klasse `HelloCacheSimulator`, die über ein Konsolen-Menü Befehle annimmt und an die Controller-Klasse `SimController` weitergibt. Diese stellt lediglich eine Schnittstelle zu den drei Hauptcontrollerklassen `CacheSimulator`, `TLBSimulator` und `LocalityAnalyzer` dar.

Der Simulator wird vor der Ausführung mit einem Testdatensatz initialisiert. Das beinhaltet das Laden der mit den Protokollierungseinheiten aufgezeichneten Trace-Dateien und die Spezifikation des Aufzeichnungsortes. Die Klasse `CacheAccessManager` stellt daraufhin *Trace-Reader* bereit. Dabei handelt es sich um Klassen, welche den abstrakten `TraceReader` erweitern. Dadurch kann völlig unabhängig von der Herkunft der Adressfolge vorgegangen werden. Ein `TraceReader` stellt die `CacheAccess`-Objekte zur Verfügung. Jeder Speicherzugriff im Cache wird mit einem `CacheAccess` gekapselt. Es erlaubt Lese-, Schreib- und Invalidierungsoperationen. Die Notwendigkeit, Cache-Einträge zu invalidieren ergab sich durch die Kopieroperationen an der Speicherschnittstelle. Ein `CacheAccess` enthält ebenfalls einen Request-Tag, der einen bestimmten Aufrufer kennzeichnet. Nur bei einem am Speicher aufgenommenen Trace hat es eine Wirkung, da dort unterschiedliche Tags registrierbar sind. Für die Translation Lookaside Buffer Simulation werden `TLBAccess`-Objekte verwendet. Sie kapseln nur die Referenzaktivierungen. Neben den Zugriffen existiert die Klasse `TraceInformation`. Sie enthält die Daten, welche die jeweilige Timing-Komponente der Protokollierungseinheit aufgezeichnet hat.

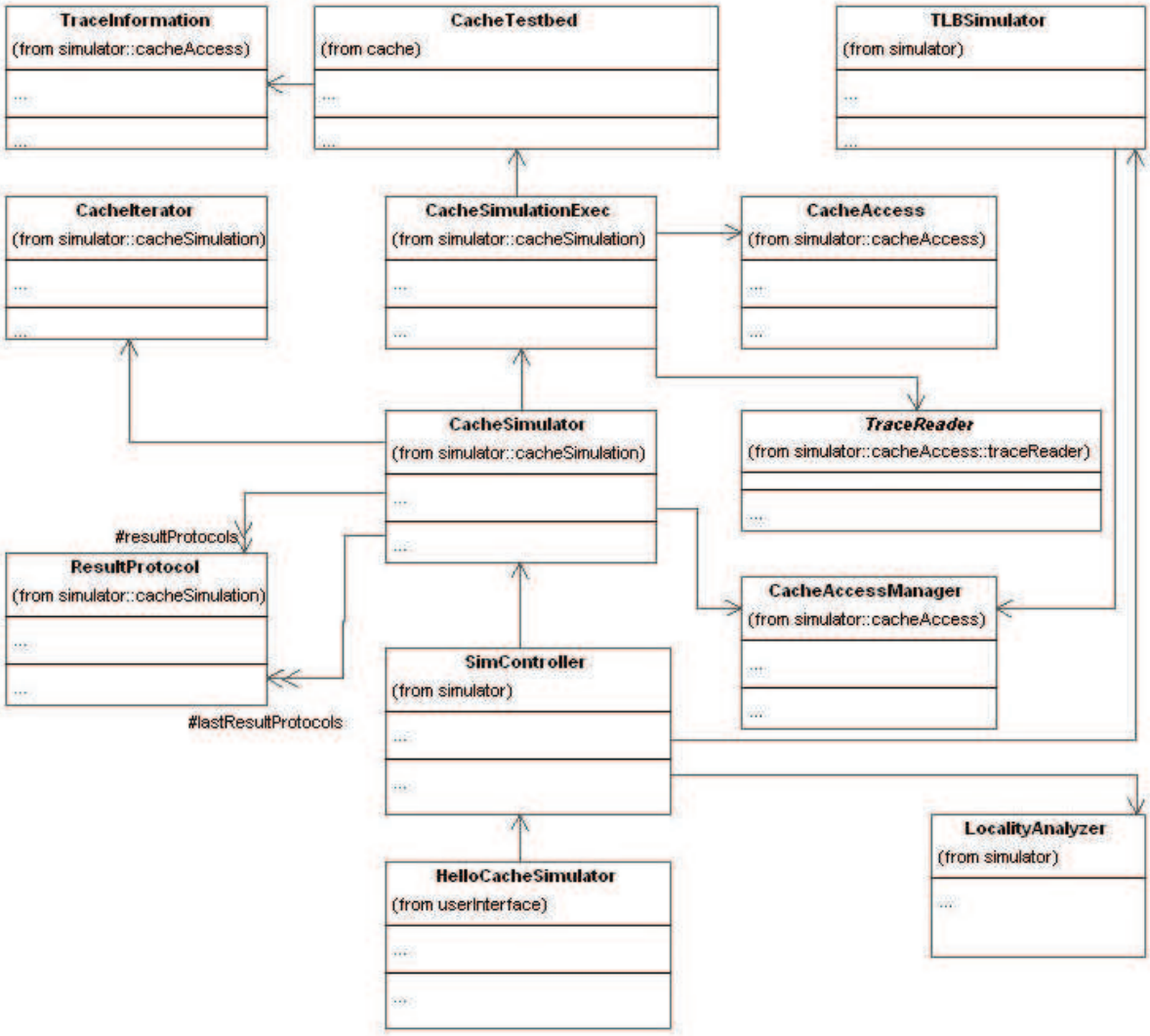

Abbildung 3.2: Die Top-Level-Struktur des Cache-Simulators.

Im folgenden werden die einzelen Fähigkeiten des Simulators kurz aufgezählt:

Lokalitätsanalyse: die Lokalität kann auf drei Arten analysiert werden:

- Bestimmung der Stack Distanz-Verteilung und der kumulativen Stack Distanz-Verteilung
- Bestimmung der Markov-Wahrscheinlichkeiten für bestimmte Schrittweiten
- Bestimmung der Wahrscheinlichkeiten durch das Look-Back-Window und des Score-Wertes

Für alle Varianten besteht die Möglichkeit, eine bestimmte Anzahl an niederwertigen Adressbits wegzulassen. Die Ergebnisse können entweder auf der Konsole, als Textdatei oder als GnuPlot-Skript ausgegeben werden.

Translation Lookaside Buffer-Simulation: es können Grenzwerte für die Anzahl der

TLB-Einträge und die Ersetzungsstrategien angegeben werden. Der Buffer wird anschließend für alle Möglichkeiten innerhalb der Parameter als vollassoziativer Cache simuliert. Die Ergebnisse enthalten Angaben über Hit-Rates, Geschwindigkeitsgewinn und Hardware-Aufwand. Sie können auf der Konsole, als Textdatei, oder als GnuPlot-Skript ausgegeben werden.

Cache-Simulation: sie stellt den komplexesten Bestandteil des Programms dar. Über das Menü oder über eine Initialisierungsdatei können Simulationsparameter gesetzt werden, innerhalb derer ein Cache-Iterator Caches erstellt und zurückgibt. Die Parameter sind:

- Assoziativität des Caches
- Cache-Größe
- Block-Größe
- Wort-Größe
- Write-Buffer-Größe (Anzahl der Einträge)
- Ersetzungsstrategie
  - First-In-First-Out
  - Random
  - Least-Recently-Used
  - Not-Most-Recently-Used
- Schreibstrategie
  - Copy-Back
  - Write-Through with Write-Allocation
  - Write-Through with No-Write-Allocation
- Addressdecoder
  - Classic-Address-Decoder
  - Hash-Address-Decoder
- Iteration über die Addresse - Verschiebung der Index-Bits zwischen den Tag-Bits
- Request-Tags - der simulierte Cache reagiert nur auf verschiedene Request-Tags

Urspünglich war angedacht, die Adressverteilung, also die Zuordnung der Bits der Adresse zu den Tag-, Index- und Wort-Bits, wahlfrei über einen Iterator bestimmen zu lassen. Da dies allerdings mit sehr langen Simulationszeiten verbunden wäre, wurde die Funktion fallengelassen. Eine Implementation liegt aber vor. Es ist dennoch möglich, eine Adressverteilung manuell einzugeben, allerdings hängt diese immer von den anderen Cache-Paramtern ab, weswegen die Funktion in der Auswertung nicht benutzt wurde.

Weitere Simulationsparameter sind:

- `simStep`: der Cache kann in Einzelschritten simuliert werden. Das Füllen der Cache-Lines wird dann auf der Konsole ausgegeben. Dieser Parameter gibt die Anzahl der Adressen an, die pro Schritt abgearbeitet werden.
- `simPhase`: der Paramter gibt an, wieviele möglichen Cache-Konfigurationen hintereinander ohne Unterbrechnung simuliert werden sollen. Für den Fall, dass solange simuliert werden soll, bis der Iterator keine neuen Caches mehr findet, gibt es einen extra Funktionsaufruf.
- `threads`: die Anzahl der Caches, die gleichzeitig in unterschiedlichen Threads simuliert werden sollen. Der Parameter dient der besseren Lastverteilung auf Multiprozessor-Rechnern.
- `simLength`: gibt eine Obergrenze an, wieviele Speicherzugriffe aus einem Trace entnommen werden sollen.
- `simAnalytical`: gibt an, ob die analytischen Modelle ebenfalls simuliert werden sollen.

Der Simulator errechnet eine Reihe von Ergebnissen:

- die Anzahl an Read-Misses, Read-Hits, Write-Misses, Write-Hits, zusätzlich die Read-Hit-Rate, Write-Hit-Rate und die Gesamt-Hit-Rate
- die Write-Buffer-Hits
- eine Fehlerklassifikation gemäß des Three C's-Modell
- die Anzahl an eingesparten Speicherzugriffen unterteilt nach Lese- und Schreibzugriff
- den Speed-Up
- eine Hardware-Score
- die Hit-Rate nach dem Brehob/Enbody-Modell
- die Hit-Rates nach den Agarwal-Modell

Die Resultate können anschließend auf der Konsole, in eine Textdatei oder als GnuPlot-Skript ausgegeben werden.

3.4.1 Cache-Modell

Der Cache-Simulator erlaubt die Berechnung eines Score-Wertes der Hardware. Dazu werden zu den Elementen der Register-Transfer-Ebene (Multiplexer, Komparatoren, Register, SRAM, etc.) die benötigten Transistoren in Abhängigkeit ihrer Größe näherungsweise bestimmt. Für jede Komponente des Cache-Modells existiert eine Berechnungsvorschrift. Somit kann für das

Modell ein aufsummierter Wert errechnet werden. Obwohl es bezüglich der Anforderungen Vereinfachungen gibt, stellen die ermittelten Werte zumindest eine Tendenz dar, wie der Aufwand sich bei der Änderung bestimmter Cache-Parameter verhält.

Im folgenden werden die Bestandteile des Cache-Modells erläutert.

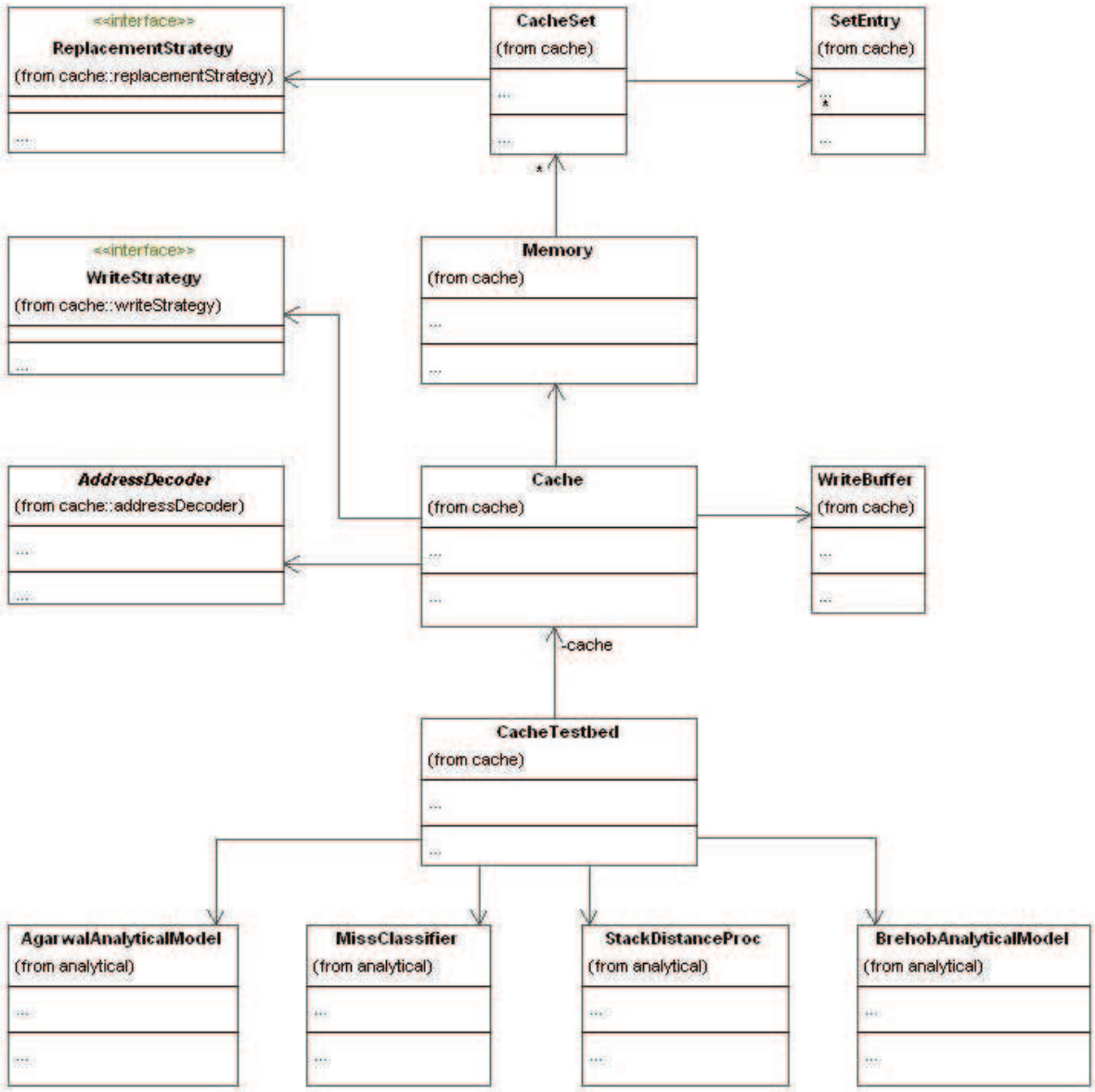

Abbildung 3.3: Das Cache-Modell des Simulators.

3.4.1.1 SetEntry

Die Klasse repräsentiert einen Eintrag in einem Set. Er besteht aus dem Tag, einem Valid-Bit und dem Dirty-Bit. Die Cache-Line wird im Simulator nicht mit gespeichert. Für jedes `SetEntry` können die Hardwareanforderungen berechnet werden. Sie bestehen aus den

SRAM-Zellen, die einer Blocklänge entsprechen und den Registern für das Tag um dem Valid-Bit. Optional werden noch ein Dirty-Bit und zwei Bits für einen MESI-Zustand dazugerechnet.

3.4.1.2 CacheSet

Ein Cache-Set enthält entsprechend der Assoziativität die Set-Einträge und eine Ersetzungsstrategie. Es werden ebenfalls Angaben über die Verdrängung von Einträgen gespeichert. Der Aufwand eines `CacheSet` ist die Summe der `SetEntry`'s und der Ersetzungsstrategie.

3.4.1.3 ReplacementStrategy

Hinter diesem Interface verbirgt sich die Ersetzungsstrategie. Sie stellt zum einen eine Lese-Methode zur Verfügung, mit der intern Zugriffe protokolliert werden. Zum anderen enthält sie eine Schreibmethode, welche die Nummer des zu ersetzenden Eintrags ausgibt.

Für unterschiedliche Strategien ergeben sich unterschiedliche Hardwareanforderungen. Bei First-In-First-Out wird ein $\log n$ - Zähler angenommen. n ist hierbei die Assoziativität. Der Zähler zeigt auf das nächste zu ersetzende Datum.

Für Least-Recently-Used existieren zwei gängige Implementierungen. Einerseits als Schieberegister. Bei diesem werden die Einträge als Liste gespeichert. Die Position in der Liste markiert das Alter eines Eintrags. Bei den entsprechenden Operationen kann sich nun die Reihenfolge ändern. Für eine derartige Realisierung sind $n \log n$-Register sowie $n-1 \log n$-Vergleicher nötig. Weiterhin entfällt ein großer Teil auf die Steuerlogik.

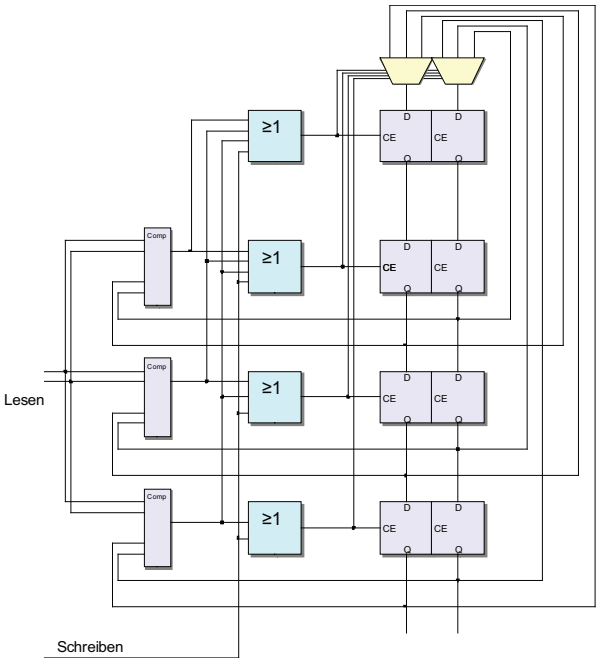

Abbildung 3.4: Beispiel eines Schieberegisters für die LRU-Ersetzungsstrategie.

Eine zweite Möglichkeit ist die Implementierung als Dreiecksmatrix. Die Nummern der Set-Einträge sind hierbei durch die Matrix miteinander in Beziehung gebracht. Der Wert entält die

entsprechenden Alterungsinformationen. Befindet sich in einer Zelle eine 1, so ist der Eintrag der Zeilennummer jünger als der Eintrag der Spaltennummer. Für eine Verdrängung wird derjenige ausgewählt, der in seiner Zeile nur Nullen und in der Spalte nur Einsen enthält [8]. Zu realisieren ist diese Schaltung mit $(n-1)!$ Registern für die Matrix und einer Steuerlogik.

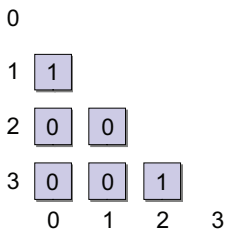

Abbildung 3.5: Beispiel einer Dreiecksmatrix für die LRU-Ersetzungsstrategie.

Für die Berechnung wird die Strategie mit dem geringeren Aufwand ausgewählt.

Not-Most-Recently-Used benötigt ähnlich der FIFO-Implementierung ein $\log n$-Register zur Speicherung des zuletzt benutzten Eintrages. Für Random wird von keinen Hardwareanforderungen ausgegangen [7].

3.4.1.4 Memory

Diese Klasse hält alle Cache-Sets. Mittels einer Set-Adresse kann auf sie zugegriffen werden.

3.4.1.5 WriteStrategy

Das Interface stellt Lese- und Schreiboperationen zur Verfügung. Eine implementierende Klasse leitet die Anfragen an die Klasse `Memory` weiter und sorgt entsprechend der Strategie für Allokationen im Cache. Sie stellt außerdem Informationen über Verdrängungen zur Verfügung. Es werden keine Hardwareanforderungen berechnet.

3.4.1.6 AddressDecoder

Es werden zwei Arten von Adress-Dekodern unterstützt. Ein `ClassicAddressDecoder` lässt die Bits unverändert und extrahiert Word- und Indexadresse, sowie den Tag.

Der `HashAddressDecoder` sorgt hingegen für eine Umrechnung des Index mit verschiedenene Tag-Bits. Dadurch wird versucht, eine bessere Streuung des Indexadressen über den Cache zu erreichen. Für beide Adress-Dekoder können Bit-Verteilungen angegeben werden. Sie spezifizieren, welche Bits einer Adresse jeweils für Tag, Wort- oder Indexadresse verwendet werden.

[7] entgegen dem Ansatz in [8], wo eine konkrete Implementierung vorgestellt wurde

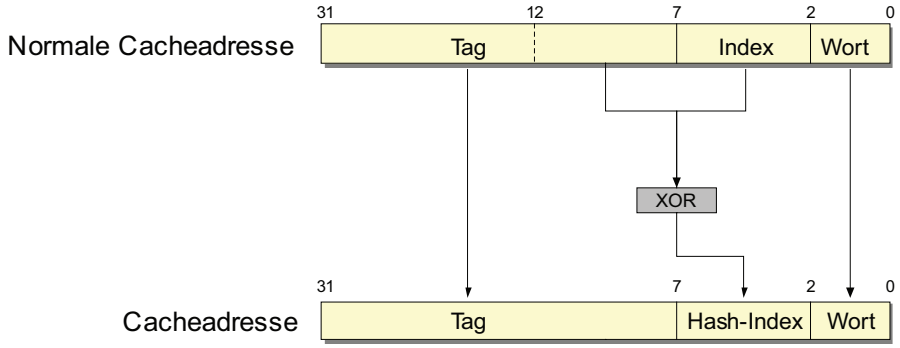

Abbildung 3.6: Adressgenerierung durch einen `HashAddressDecoder`.

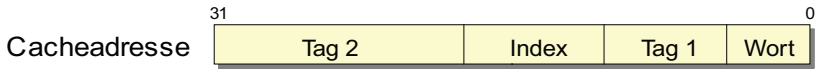

Abbildung 3.7: Eine Cache-Adresse mit spezieller Bit-Verteilung.

3.4.1.7 WriteBuffer

Er stellt den Victim-Cache dar. Verdrängte Einträge werden hier zwischengespeichert. Er kann als vollassoziativer Cache nach der First-In-First-Out-Strategie angesehen werden. Als Tag wird die Adresse des verdrängten Eintrages ohne die Wortbits verwendet. Der Write-Buffer ist optional. Als Hardwareanforderungen ergeben sich für die Daten SRAM-Zellen entsprechend der Anzahl der Einträge multipliziert mit der Blockgröße, und die entsprechenden Register. Jeder Write-Buffer-Eintrag enthält neben dem Tag Status-Bits, die kennzeichnen, welches Wort aus dem Block bereits in den Hauptspeicher zurückgeschrieben wurde.

3.4.1.8 Cache

Diese Klasse enthält den Cache-Speicher, die Schreibstrategie, den Adress-Dekoder und den Write-Buffer. Sie nimmt Lese-, Schreib- und Invalidierungsbefehle entgegen, sorgt für die Dekodierung der Adresse und leitet sie weiter. Die Hardwareanfoderungen hängen stark von der Assoziativität des Caches ab:

- die Anforderungen der weiteren Komponenten
- Multiplexer zur Auswahl eines Sets (entfällt bei vollassoziativen Cache)
- Komparatoren zum parallelen Vergleich der Tags des angewählten Sets
- Multiplexer zur Auswahl eines Wortes aus dem Block pro Eintrag des ausgewählten Sets
- Multiplexer zur Auswahl des Wortes aus den Einträgen des Sets
- Steuerlogik

3.4.2 Timing-Modell

3.4.2.1 Motivation

Im Cache-Simulator steht ein Timing-Modell zur Berechnung des Speed-Ups zur Verfügung. Es ist in der Klasse `CacheTestbed` implementiert. Vor der Ausführung müssen ihm Informationen über das Programm zur Verfügung gestellt werden. Diese beinhalten die durchschnittliche Ausführungszeit von Lese- und Schreiboperationen sowie die durchschnittliche Wartezeit zwischen Hauptspeicherzugriffen.

Eine genauere Betrachung des Timings ist für die Einschätzung der Leistungsfähigkeit eines Caches unabdingbar. Die Hit-Rate allein ist nicht aussagekräftig. Da die Blockgröße in der Regel höher ist als die Wortgröße, sind bei einem Read-Miss statt einer Leseoperation durch den Aufrufer mehrere Leseoperation durch den Cache nötig. Reicht die Zeit zwischen zwei Anfragen aus, um die Cache-Line zu füllen, so entsteht kein Nachteil. Ist das nicht der Fall, so verlangsamt der Cache die Ausführungsgeschwindigkeit des Prozessors. Verschärft wird die Situation zusätzlich, wenn in dem Cache eine Verdrängung eines als dirty markierten Eintrages stattfand. In dem Fall sind erst eine Reihe von Schreiboperationen notwendig, bevor gelesen werden kann. Der Hauptspeicherbus wird für längere Zeit blockiert. Ein Write-Buffer kann für einen gewissen Lastenausgleich sorgen, indem er Schreiboperationen puffert und den Bus zwischenzeitlich freigeben kann.

Für die Berechnung des Geschwindigkeitsgewinns wird davon ausgegangen, dass der Aufrufer während der kompletten Lese- oder Schreibzeit blockiert [8]. Kann ein Zugriff aus dem Cache beantwortet werden, so wird diese Blockierzeit als eingespart angesehen. Anschließend gilt die durchschnittliche Zeit zwischen den Befehlen bis zum nächsten Zugriff.

Im Folgenden wird das genaue Vorgehen der Timing-Analyse bei den unterschiedlichen Operationen und Zuständen beschrieben.

3.4.2.2 Invalidierungsbefehl

Handelt es sich um einen Invalidierungsbefehl, muss dieser sowohl an den Hauptspeicher als auch an den Cache weitergeleitet werden. Das Modell errechnet, ob der Cache aus vorherigen Operation noch Speicherzugriffe durchführt. Ist das der Fall, so wird die Wartezeit bis zur Freigabe des Speicherbusses negativ angerechnet. Die Freigabe hängt von der Art des Cache-Aufbaus ab: Ist ein Write-Buffer vorhanden, so braucht er nur das aktuelle Wort zurückzuschreiben. Ist der Write-Buffer bereits gefüllt oder nicht implementiert, so wird die Schreiboperation aus dem Cache durchgeführt. In dem Fall muss der gesamte Block geschrieben werden. Intern wird dieses Modell mittels eines Zeitkontos für Schreiboperationen implementiert. Ist der Bus wegen einer Leseoperation blockiert, wird diese auf jeden Fall beendet. Intern wird die Stillstandszeit bereits bei der vorhergehenden Operation berechnet. Da es sich bei den Invalidierungsbefehlen um Kopierbefehle für den Hauptspeicher handelt, müssen bei einem Cache

[8] die Blockierung während eines Invalidierungsbefehls wird nicht durch den Cache beschleunigt, da er die eigentlichen Kopieroperationen an den Speicher weitergibt

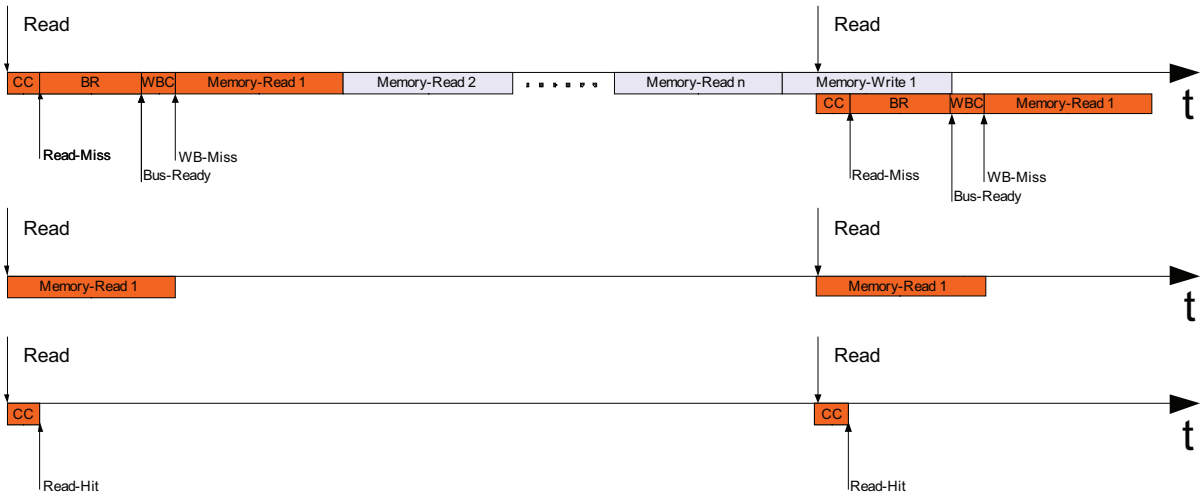

Abbildung 3.8: Beispiel eines Geschwindigkeitsverlustes durch den Cache (oben), eines normalen Speicherzugriffs (mitte) und eines Geschwindigkeitsgewinns durch den Cache (unten).

mit Copy-Back-Schreibstrategie unter Umständen Einträge zurückgeschrieben werden[9]. Das Durchsuchen des Caches wird mit einem Takt Zeitverlust angerechnet. Er wird zu der Busfreigabezeit addiert. Enthält das Cache-Modell entsprechende Daten, wird zusätzlich noch die Zeit für die Schreiboperation addiert. Der entstandene Wert stellt eine Verlangsamung der Ausführung dar. Nach Abschluss des Invalidierungsbefehls wird mit der durchschnittlichen Wartezeit bis zum nächsten Befehl gerechnet. In diesem Zeitraum kann der Write-Buffer, wenn er vorhanden und gefüllt ist, Werte in den Hauptspeicher schreiben und das Zeitkonto reduzieren.

3.4.2.3 Lesebefehl

Bei einer Leseoperation wird überprüft, ob der Cache den Aufrufer des Befehles akteptiert oder nicht. Falls nicht, wird er dennoch auf das Datum durchsucht. Für diese Operation wird ein Takt als Zeit angerechnet. Ist das Datum enthalten, wird es als *Non-Cached-Hit* gezählt und die Beschleunigung bestimmt. Bei einem Fehlschlag wird die Zeit bis zur Busfreigabe berechnet und die Operation auf dem Hauptspeicher angenommen. Die Verzögerungszeit zur Busfreigabe wird negativ angerechnet.

Bei einem Lesezugriff eines cachenden Anfragers wird ebenfalls ein Takt zum Durchsuchen berechnet. Bei einem Hit werden die gesparten Takte minus dem Wartetakt gezählt. Anschließend beginnt der Zeitraum bis zum nächsten Befehl. Bei einem Miss muss zuerst der Hauptspeicherbus freigegeben werden. Anschließend wird der Write-Buffer, sofern er vorhanden ist, durchsucht. Dafür wird ein Takt benötigt. In dieser Zeit wird von keinen Hauptspeicheroperationen ausgegangen. Da ein neuer Block in den Cache geladen wird, muss überprüft werden,

[9] vereinfacht wird davon ausgegangen, dass der Write-Buffer keine Daten enthält, die durch die Kopieroperationen betroffen sind

ob es eine Verdrängung eines als dirty gekennzeichneten Eintrages gab. In diesem Fall muss er entweder im Write-Buffer zwischengespeichert oder komplett in den Hauptspeicher geschrieben werden. Dafür wird die Zeit bestimmt. Nun wird im Falle eine Write-Buffer-Hit ein Takt oder die Zeit für eine Leseoperation im Fall eines Write-Buffer-Miss angenommen, um den zu lesenden Wert dem Anfrager zur Verfügung zu stellen. Dauert die Summe dieser Operationen länger als ein normaler Lesezugriff, so hat der Cache das System verlangsamt und die Anzahl der bis jetzt eingesparten Taktzyklen wird reduziert. Im anderen Fall wird sie erhöht. Wurde der neue Wert aus dem Hauptspeicher gelesen, so wird die Lesezeit für die restlichen Blockeinträge mit der Zeit vor dem nächsten Befehl verrechnet. Bleibt noch Zeit übrig, werden Write-Buffer-Operationen simuliert. Reicht die Zeit nicht aus, wird die zusätzliche Dauer negativ angerechnet.

3.4.2.4 Schreibbefehl

Bei einer Schreiboperation eines nicht-cachenden Anfragers wird die Zeit für die Freigabe des Speicherbusses negativ angerechnet. Es wird davon ausgegangen, dass der Cache aktualisiert wird. Dies fließt nicht in die Berechnung mit ein.

Bei cachenden Anfragern wird nach den Schreibstrategien unterschieden. Für einen Copy-Back-Cache wird erneut ein Takt zum Durchsuchen gezählt. Bei einem Hit wird die eingesparte Schreibzeit minus einem Takt positiv angerechnet. Im Falle eines Fehlschlags ähnelt das Vorgehen einem Read-Miss. Es wird auf den Speichertakt gewartet, um anschließend den Write-Buffer zu durchsuchen. Bei einer Verdrängung wird der Eintrag auf den Write-Buffer gelegt oder zurückgeschrieben. Anschließend finden die Ladeoperationen zur Allokation des Eintrages im Cache statt. Wie bei den Lesezugriffen werden die Zeiten gegeneinandergerechnet und der Gewinn oder Verlust bestimmt. Ganz anders verhält sich das Write-Through-Verfahren. Es wird kein Takt für das Durchsuchen des Caches angenommen, da der Hauptspeicherzugriff definitiv stattfindet. Im Fall eines Hits wird lediglich die Zeit berechnet, die es dauert, den Hauptspeicherbus freizugeben. Dies führt zu einer Verlangsamung. Ein Geschwindigkeitsgewinn ist nicht zu erwarten. Im Falle eines Write-Miss werden bei Write-Through with Write-Allocation die Zeiten berechnet, die das Allokieren des Wertes im Cache dauern. Die Ausführungszeit wird mit der Wartezeit für den nächsten Befehl verrechnet.

3.4.3 Analytische Modelle

Im Cache-Simulator wurden die in Kapitel 2 beschriebenen analytischen Modelle implementiert. Das Modell nach Agarwal hat dabei weniger den Zweck einer Miss-Rate-Vorhersage, als einer Klassifikation der Cache-Fehlschläge. Da die Berechnung und die Cachesimulation parallel erfolgen, wurden einige Parameter ersetzt. Sowohl die Berechnungen des Markov-Modells wie die Collision Rate werden im Cache-Modell mit höherer Genauigkeit bestimmt.

3.5 Simulationsablauf

Die Simulation ist mit einem hohen Aufwand verbunden. Die Parameter ergeben einen großen Elaborationsraum. Deswegen wurde nach einem Muster vorgegangen. Für eine konstante Adressverteilung und für die Least-Recently-Used-Ersetzunsstrategie wurden Cache-Konfigurationen in gängigen Parametern für Assoziativität, Write-Buffer-Größe, Cache-Größe, Block-Größe und Schreibstrategie simuliert. Anschließend erfolgte punktuell die Auswahl von Ergebnissen, die eine Simulation der übrigen Paramter sinnvoll erscheinen lies.

Diese ersten Durchgänge erfolgten mit einer maximalen Obergrenze von 2.500.000 Zugriffen pro Simulation. Die Ergebnisse dienten zur Feststellung der idealen Cache-Konfigurationen an den verschiedenen Implementierungsstellen. Um diese dann untereinader zu vergleichen, wurden die Adressfolgen komplett simuliert.

Die analytischen Modelle sowie die Adressfolgen vom Prozessorsimulator sind nur an bestimmten Stellen zu Vergleichszwecken einbezogen wurden. Die Auswertung enthält die Ergebnisse der Folgen, die direkt von SHAP-Prozessor stammen.

4 Lokalitätsanalyse

4.1 Objektaktivierungen

Für die Analyse der Objektaktivierungen wurde der Befehl "Referenz setzen" (*mem_stref*) des Kerns ausgewertet. Er bewirkt, dass im Speicherport zu einer Referenz die Basis für die Adressumsetzung geladen wird. Die Beobachtungen der Lokalität werden erste Rückschlüsse über die Leistung eines Translation-Lookaside-Buffers liefern. Als Metrik wurde die Stack Distanz verwendet.

	0	1	2	3	4	5	6	7	8	9	10
Crypt	0.38	0.32	0.08	0.22	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	0	≈ 0
Crypt 2	0.23	0.52	0.07	0.19	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	0	≈ 0
Crypt 3	0.23	0.52	0.06	0.19	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	0	≈ 0
HeapSort	0.16	0.81	0.03	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0
RayTracer	0.05	0.12	0.01	0.10	0.12	0.03	0.10	0.13	0.05	0.07	0.03
Sudoku	0.82	0.15	0.03	0.01	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0

Tabelle 4.1: Verteilung der Stack Distanzen

Bei dem HeapSort-Sortieralgorithmus erfolgen 81.4 % aller Aktivierungsbefehle auf Referenzen, deren letzte Aktivierung nur einen Stack-Eintrag zurückliegt. Bei Crypt sind die Wahrscheinlichkeiten für die ersten 4 Distanzen stärker ausgeglichen. Bei 37.8 % aller Aufrufe wird eine bereits aktivierte Referenz erneut aufgerufen. Der Crypt-Algorithmus arbeitet mit mehreren Arrays, die Kodier- und Dekodierschlüssel enthalten. Sie werden auf eine zentrale Datenstruktur angewandt und in mehreren Durchgängen verrechnet. Einige der Schlüssel werden für den jeweils nächsten Zyklus übernommen. Durch die Verwendung mehrere Objekte steigen die Stack Distanzen im Durchschnitt, bleiben aber klein, da sich die Arbeitsmenge nur langsam ändert.

Sudoku zeigt eine starke Lokalität bedingt durch die Programmstruktur. Da das Sudoku-Feld immer durch ein Objekt repräsentiert ist, wird auch größtenteils auf diesem gearbeitet. In Folge sind 81.6 % aller Aktivierungen auf bereits aktivierte Objekte. RayTracer stellt das Gegenteil dar. Die Programmstruktur ist stark modularisiert. Einzelne Elemente der 3D-Szene werden darin gekapselt. Die Lokalität sinkt dadurch.

Abbildung 4.1 zeigt die Cumulative Stack Distance Distribution. Es lässt sich ablesen, dass ein Translation Lookaside Buffer mit 4 Einträgen bereits einen größen Anteil der Anfragen

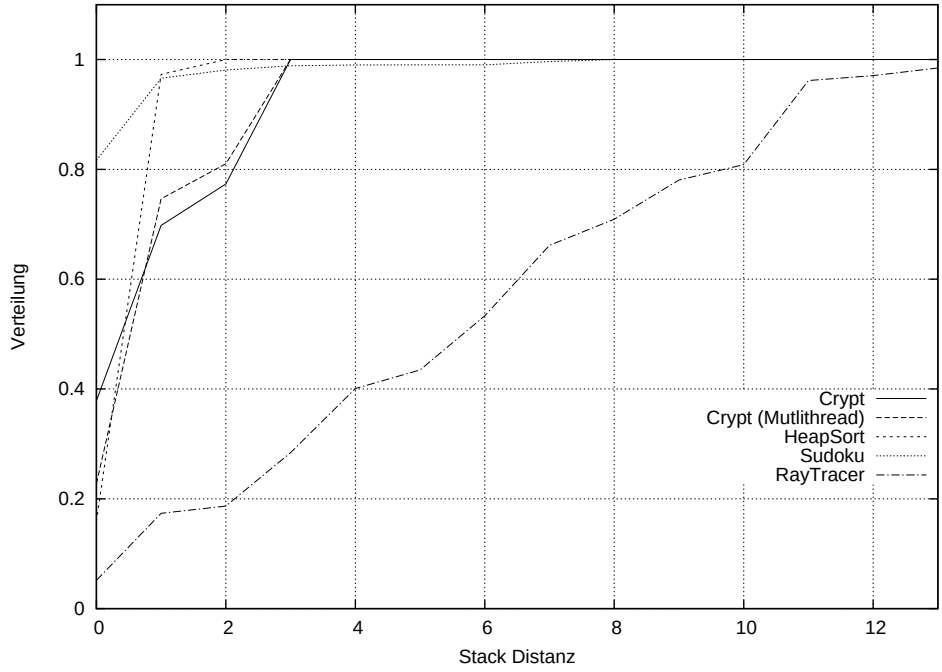

Abbildung 4.1: Cumulative Stack Distance Distribution der Objektaktivierungen

enthalten kann. Der RayTracer steht dem allerdings entgegen. Für eine effiziente Ausführung ist ein wesentlich größerer Buffer notwendig.

4.2 Lokalität der Objektzugriffe am Kern

Die Analyse der Lokalität innerhalb der Objekte basiert auf den Stack Distanz- und Look-Back-Window-Modellen. Dabei wurden die kompletten Speicheradressen bestehend aus der Konkatenation von Referenz und Offset betrachtet. Die zeitlichen Lokalitäten sind im Diagramm 4.2 dargestellt. Der RayTracer zeigt auch hier die schlechtesten Werte. Für ihn würde eine ideale Cache-Implementierung eine Größe von 16 Einträgen (64 Byte) haben. Für die anderen Anwendungen werden bereits mit 8 Einträgen (32 Byte) bei einen vollassoziativen Cache mit Least-Recently-Used-Ersetzung aktzeptable Treffer-Raten von 90 % erreicht.

Die örtliche Lokalität verdeutlicht Tabelle 4.2. Sie enthält zum einen die Wahrscheinlichkeit, dass ein Adressaufruf die Schrittweite 1 oder die Schrittweite 2 hat. Damit ist gemeint, das bei einem Zugriff innerhalb des Look-Back-Windows eine direkt benachbarte, bzw. eine zwei Schritt entfernte Adresse liegt. Eine Abschätzung über mögliche Blockgrößen fällt bei diesem Modell schwer, da die Ausrichtung der Adressen im Cache nicht betrachtet wird. Es kann aber angenommen werden, dass für einen Cache mit einer Blockgröße von 4 Wörtern die Stride-1-Werte gelten. Somit zeigen die Wahrscheinlichkeiten an, wie hoch die Chance ist, dass bei einem Zugriff das Datum bereits durch einen vorhergehenden Aufruf in den Cache geladen wurde.

Der dritte Wert der Tabelle ist ein Score-Wert, mit dem die Lokalität quantifiziert wird. Je

näher der Wert an 1 liegt, um so höher die Lokalität.

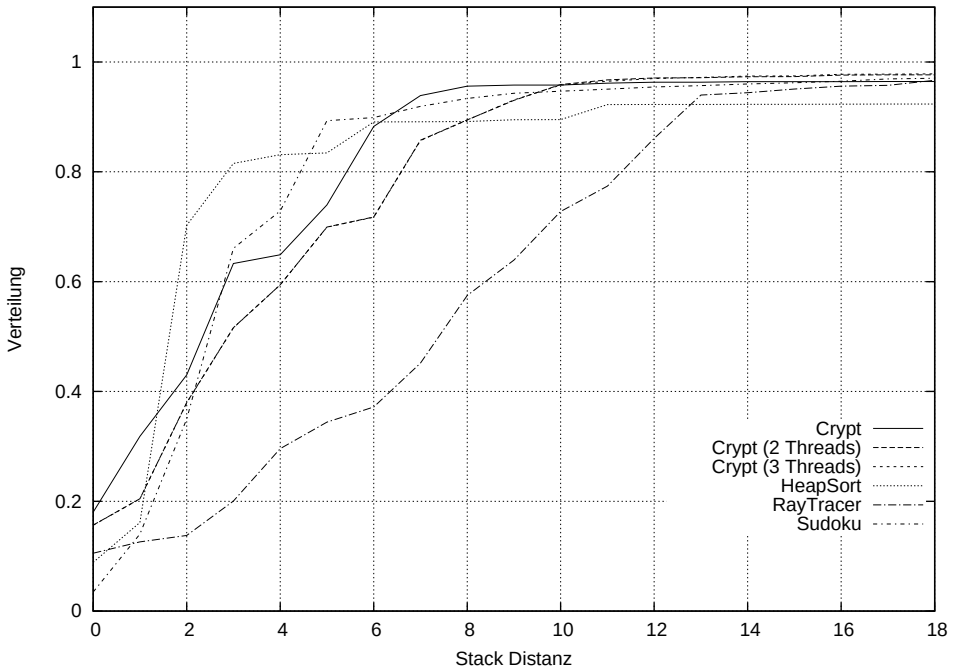

Abbildung 4.2: Zeitliche Lokalität der Objektzugriffe

Anwendung	Stride 1 - Wahrscheinlichkeiten	Stride 2 - Wahrscheinlichkeiten	Score
Crypt	36.68 %	0.32 %	0.3843
Crypt 2	31.46 %	6.95 %	0.3641
Crypt 3	31.31 %	6.96 %	0.3630
HeapSort	16.8 %	5.89 %	0.2236
RayTracer	0.46 %	1.82 %	0.0250
Sudoku	64.58 %	5.85 %	0.6774

Tabelle 4.2: Örtliche Lokalität der Objektzugriffe

4.3 Lokalität der Speicherzugriffe

Das Diagramm 4.3 stellt die Cumulative Stack Distance Distribution für die Adresszugriffe auf den Speicher dar. Die besten Werte zeigen zu Anfang HeapSort und Sudoku. Hier kann bereits mit sehr kleinen Caches ein großer Anteil der Einträge gehalten werden. Eine gute Cache-Größe für alle Anwendungen wird bei 512 Einträgen erreicht (2 KB). Die tatsächlichen Werte für eine günstige Implementierung werden dieen Wert allerdings übersteigen, da ein vollassoziativer Cache nur mit hohem Aufwand zu implementieren ist.

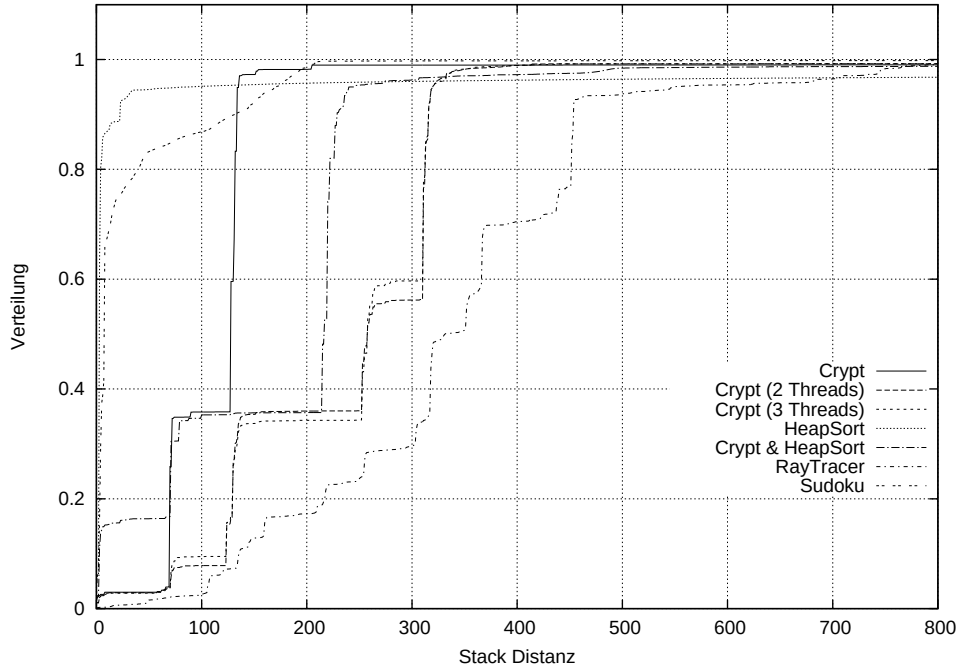

Abbildung 4.3: Zeitliche Lokalität der Speicherzugriffe

Die Tabelle 4.3 zeigt die örtliche Lokalität. Im Gegensatz zu den Zugriffen am Kern steigt die Lokalität insbesondere beim RayTracer, aber auch bei den meisten anderen Anwendungen. Ein Grund dafür kann die Adressumrechnung sein. Der Offset wird mit einem Bias verrechnet. Es kommt somit zu einer dichteren Adressverteilung. Auch können die Einträge, die am Kern nicht gecached werden, wie zum Beispiel der Methoden, die örtliche Lokalität nennenswert erhöhen.

Anwendung	Stride 1 - Wahrscheinlichkeiten	Stride 2 - Wahrscheinlichkeiten	Score
Crypt	57.74 %	0.19 %	0.6497
Crypt 2	57.77 %	0.19 %	0.6499
Crypt 3	57.77 %	0.19 %	0.6499
Crypt & HeapSort	56.96 %	0.74 %	0.5923
HeapSort	57.77 %	7.87 %	0.6433
RayTracer	88.52 %	0.42 %	0.8923
Sudoku	33.86 %	9.75 %	0.4768

Tabelle 4.3: Örtliche Lokalität der Speicherzugriffe

5 Auswertung

5.1 Prozessor-Cache

5.1.1 Vor- und Nachteile

Ein Implementierung des Caches am Kern bietet mehrere Vorteile. So kann durch seine Nähe zum Prozessor eine Anfrage nach minimaler Zeit beantwortet werden. Die Blockierung wird so schnell wie möglich aufgehoben.

Der Cache wäre virtuell adressiert. Es kann mit der Referenz und dem Offset auf ihn zugegriffen werden. Die Verschiebung eines Objektes im Speicher durch den Garbage-Collector würde keinen Einfluß auf den Cache-Inhalt haben. Dies birgt gerade bei Anwendungen mit einem hohen Objektaufkommen Vorteile, da sonst unablässig Teile des Caches invalidiert werden müssten. Die virtuelle Adressierung hat aber einen Nachteil beim Laden von Blöcken. Es muss erst eine Adressumsetzung geschehen, bevor ein Wert gelesen werden kann.

Auch der Aufwand steigt. Referenz und Offset ergeben zusammen 25 Bit gegenüber der 23 Bit Hauptspeicheradresse. Die Tags werden somit größer.

Ein weiterer Nachteil ist die Einschränkung, dass wegen dem Speichermanager kein Copy-Back als Schreibstrategie angewandt werden kann. Da die Objekte im Hauptspeicher aktuell sein müssen, ist ein ständiges Rückschreiben erforderlich.

Ein Nachteil des Caches ergibt sich auch bei bei Multi-Core-Umgebungen. Da jeder Prozessor über einen eigenen Speicherport auf den CPU zugreifen würde, müsste jeweils ein privater Cache implementiert werden. Arbeiten mehrere Prozessoren auf den gleichen Daten, so ist zusätzlich eine Snoop-Logik und ein entsprechendes Protokoll nötig. Dazu würde sich Write-Through eignen. Aus diesem Nachteil kann sich auch ein Vorteil ergeben. Verfügt jeder CPU über einen eigenen Cache, wird der Engpass des Speicherzugriffes vermindert. Jeder kann einen großen Teil seiner Arbeit lokal erledigen. Ein gemeinsamer Cache für die Prozessoren würde mehr Wartezeiten hervorrufen.

Die Leistungsfägikeit eines Caches hängt maßgeblich davon ab, wieviel Blockierungszeit er vermeidet. In Tabelle 5.1 sind für den Kern in Abhängigkeit der Programme und der Operationen die Zeiten aufgeführt

Der Cache wird das System dann maßgeblich beschleunigen, wenn die Zeit zwischen den Anfragen gering, aber die Lese- und Schreibdauer hoch ist. Die zur Verfügung stehenden Anwendungen verhalten sich dabei sehr unterschiedlich. Crypt und RayTracer arbeiten wenig auf dem Speicher, während die Anderen häufiger Anfragen durchführen. Dafür ist die Wartezeit bei den Erstgenannten wesentlich höher.

	Referenz aktivieren	Lesen	Schreiben	Warten
Crypt	1,55 Takte	53.55 Takte	3.01 Takte	582.71 Takte
Crypt (2 Threads)	16.24 Takte	42.33 Takte	3.01 Takte	539.14 Takte
Crypt (3 Threads)	17.56 Takte	42.39 Takte	3.02 Takte	541.16 Takte
HeapSort	1.46 Takte	11.40 Takte	3.04 Takte	9.36 Takte
RayTracer	12.95 Takte	41.41 Takte	5.55 Takte	630.5 Takte
Sudoku	1.33 Takte	11.1 Takte	3.1 Takte	12.04 Takte

Tabelle 5.1: Timingdaten am Kern

5.1.2 Translation Lookaside Buffer-Implementierung

Für einen Cache am Kern kommt zusätzlich die Implementierung eines Translation Lookaside Buffers in Frage. Dieser würde nach dem Speicherport die Adressumsetzung cachen. Für die Simulation wurden die Referenzaktivierung am Kern gemessen und mit den Zeiten am Port verrechnet. Der Speicherport cached bereits neben der aktuellen Referenz eine Weitere. Deswegen wurde am Kern gemessen, da eine zukünftige Implementierung den Speicherport nur eine Adresse zwischenspeichern lassen würde.

Der Translation Lookaside Buffer wird als vollassoziativier Cache simuliert. Er enthält als Tag die virtuelle Adresse und als Referenz die Basisadresse und den Bias, einen Wert, mit dem der Offset verrechnet wird.

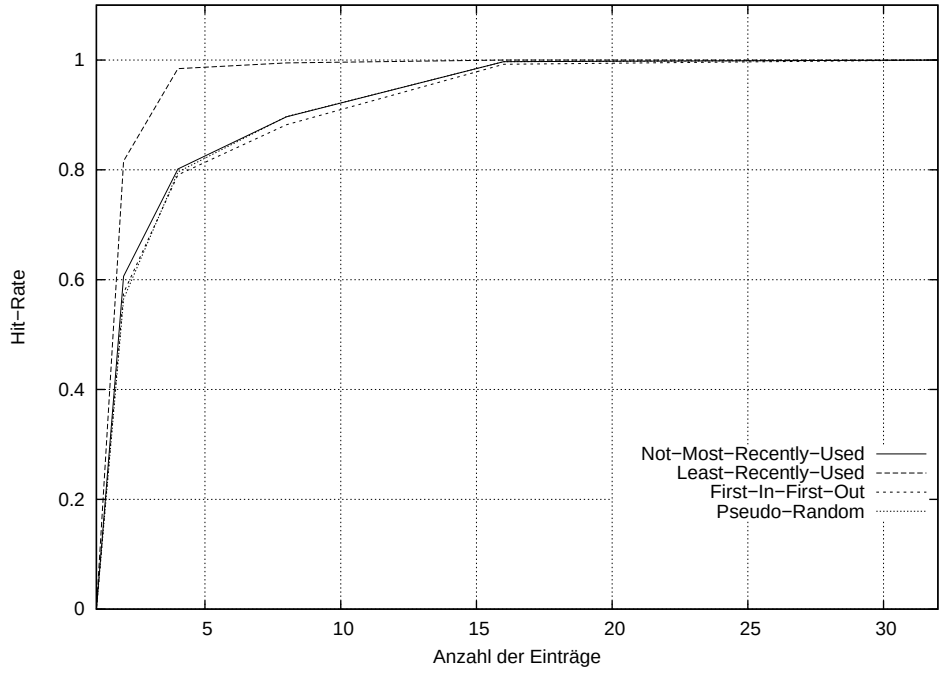

Abbildung 5.1: Translation Lookaside Buffer Hit-Raten

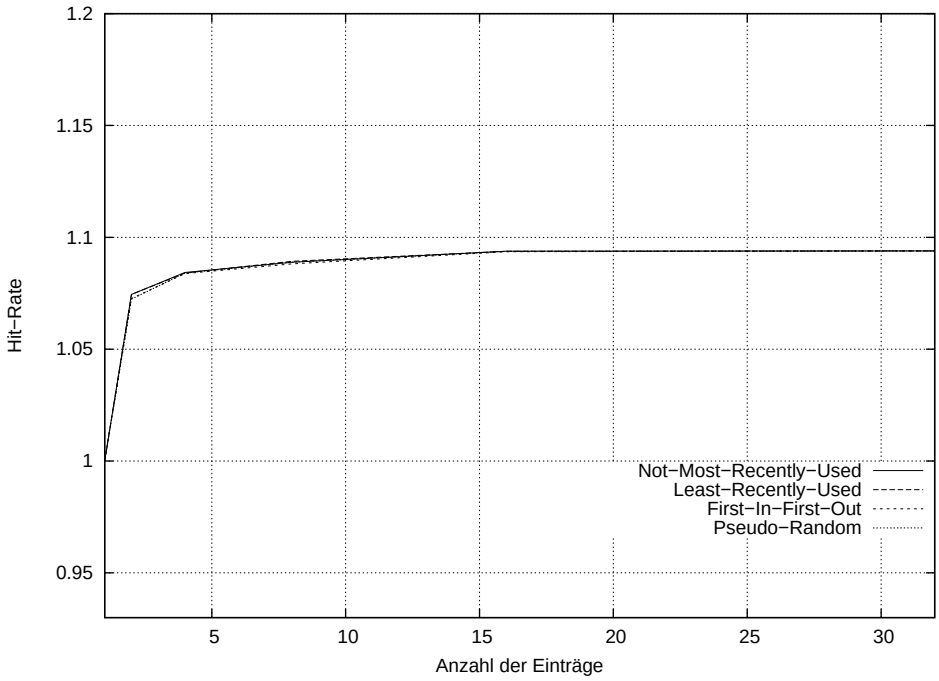

Abbildung 5.2: Translation Lookaside Buffer Speed-Up

Über die Anwendungen RayTracer, Crypt, HeapSort und Sudoku wurden Durchschnittswerte gebildet.

Bei einer Größe von einem Eintrag weisen alle Translation Lookaside Buffer eine Hit-Rate von 0 auf. Dieser Wert ist nachvollziehbar, da keine gleichen Anfragen hintereinader auf den Buffer stattfinden. Eine aktivierte Referenz wird im Speicherport gehalten.

Es zeigt sich zu Beginn ein starker Anstieg der Hit-Rate mit der Größe. Die Ersetzungsstrategie Least-Recently-Used verhält sich dabei am günstigsten. Gemäß den Ergebnissen würde eine derartige Implementierung mit 4 Einträgen eine andere Ersetzungsstrategie mit 8 Einträgen noch übertreffen.

Der Speed-Up erreicht Werte zwischen 7 und 9 %. Die Ursache liegt in der HeapSort-Anwendung, die durch den TLB um bis zu 24 % beschleunigt wird.

Die Diagramme zeigen bei einem TLB von 4 Einträgen bereits gute Ergebnisse. Der Speed-Up, der durch eine Verdopplung der Größe gewonnen würde, rechtfertigt nicht mehr die Erhöhung des Hardwareaufwandes um dem Faktor 2.3. Für die Ersetzungsstrategien ergeben sich nur geringe Unterschiede beim Geschwindigkeitsgewinn. Eine Zufallsstrategie kann mit dem geringsten Aufwand implementiert werden.

5.1.3 Cache-Implementierung

Für die initialen Simulationen wird eine Blockgröße von 32 Bit und die Write-Through with Write-Allocation-Strategie angenommen.

Die Assoziatvitäten 4 und 8 erreichen bei 128 Byte Cachegröße bereits Treffer-Raten von

97 %. Für einen 2-Wege Cache werden entsprechende Werte erst bei der doppelten Kapazität erreicht. Da alle weiteren Parameter übereinstimmen, entstehen die Fehler aufgrund von Kollisionen. Der Speed-Up steigt von 1.417 bei der Assoziativität 2 auf nur 1.431 bei der Assoziativität 4. Für die weitere Analyse wird deshalb der 2-Wege Cache mit 128 KByte Kapazität gewählt.

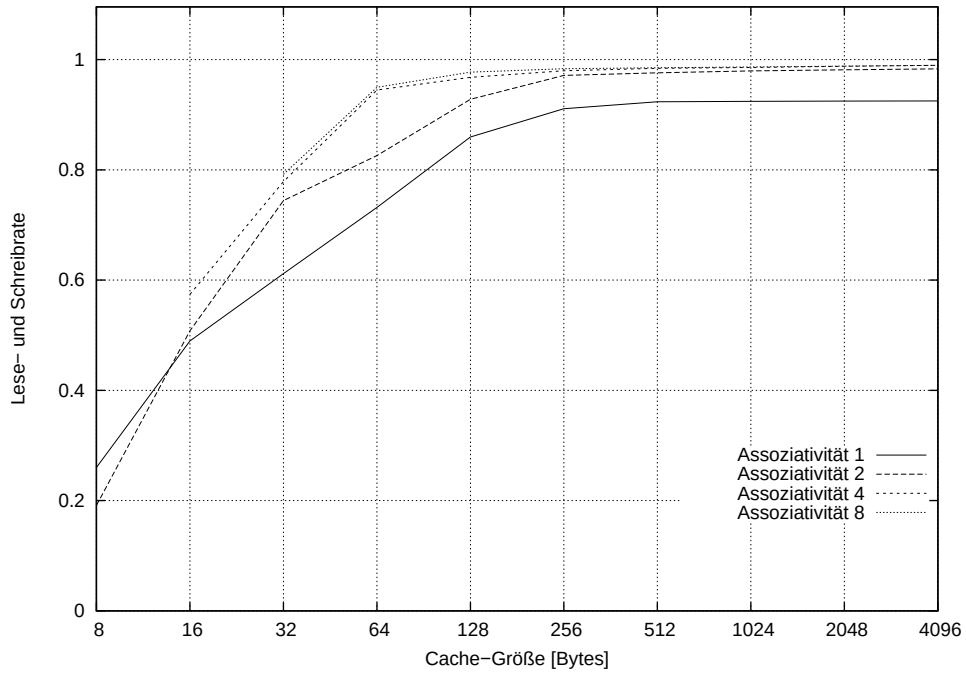

Abbildung 5.3: Hit-Rate in Abhängigkeit der Cachegröße und der Assoziativität

In Diagramm 5.4 ist der Einfluss der Schreibstrategie und der Blockgröße dargestellt. Der Speed-Up verringert sich bei dem Sprung von 4 Byte auf 8 Byte nur minimal. Die höhere Blockgröße sollte für die Implementierung bevorzugt werden, da bei gleichbleibender Cachegröße der Hardwareaufwand sinkt. Die Anzahl der Blöcke wird reduziert, es sind nur noch die Hälfte der Tags nötig. Auch die Ersetzungsstrategie kann einfacher implementiert werden.

Bei den Schreibstrategien zeigt sich Write-Through with No-Write-Allocation als die leicht bessere Variante. Die Unterschiede in den Leistungsdaten betreffen die Write-Hit-Rate. Bei No-Write-Allocation ist sie 0. Durch das Lesen geladene Daten werden also nicht geschrieben. Bei Write-Through with Write-Allocation sinkt die Lese-Rate leicht. Allerdings ist der Verlust gering, da Schreiboperationen nur einen geringen Anteil haben.

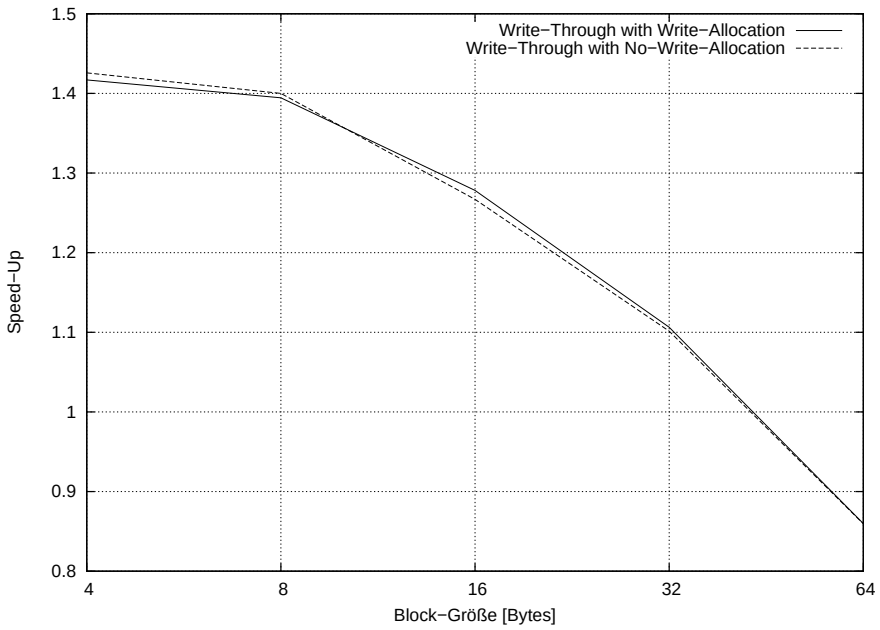

Abbildung 5.4: Speed-Up in Abhängigkeit der Blockgröße und der Schreibstrategie

Die Simulation wurde mit einer Block-Größe von 8 Byte und Write-Through with No-Write-Allocation fortgesetzt. Das Diagramm zeigt die Hit-Rate bei einer Verschiebung der Index-Bits. Zwischen Index- und Wortauswahlbits befinden sich eine bestimmte Anzahl an Tag-Bits. Diese sind in der x-Achse dargestellt.

Die Analyse der Adressverteilung ergab eine Leistungsverbesserung durch die Verwendung des Hash-Adress-Dekoders mit gleichzeitiger Verschiebung der Index-Bits auf die niederwertigsten Referenzbits. Diese beginnen bei dem Adressoffset 14.

Im Diagramm ist zusätzlich die Standardabweichung dargestellt.

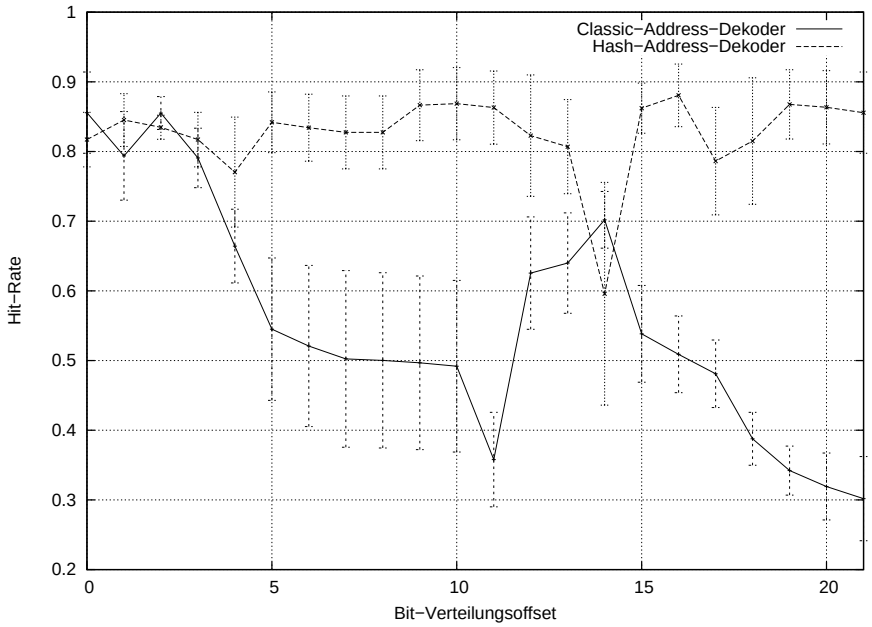

Abbildung 5.5: Hit-Rate in Abhängigkeit der Adressverteilung

Die Erhöhung der Write-Buffer-Größe mit 4 Einträgen steigert die Beschleunigung im Durchschnitt um 2 Prozentpunkte. Der Aufwand wächst allerdings sehr stark. Dem 128 Byte großen 2-Wege Cache stünde ein 32 Byte vollassoziativer Write-Buffer gegenüber, dessen Aufwand fast die Hälfte des Cache-Aufwandes ist.

Für die Ersetzungsstrategien lassen sich kaum Unterschiede feststellen. Bei der Assoziativität 2 verhalten sich Not-Most-Recently-Used und Least-Recently-Used gleich. Allerdings kann auf die Implementierung in Anbetracht der geringen Geschwindigkeitsgewinne verzichtet werden.

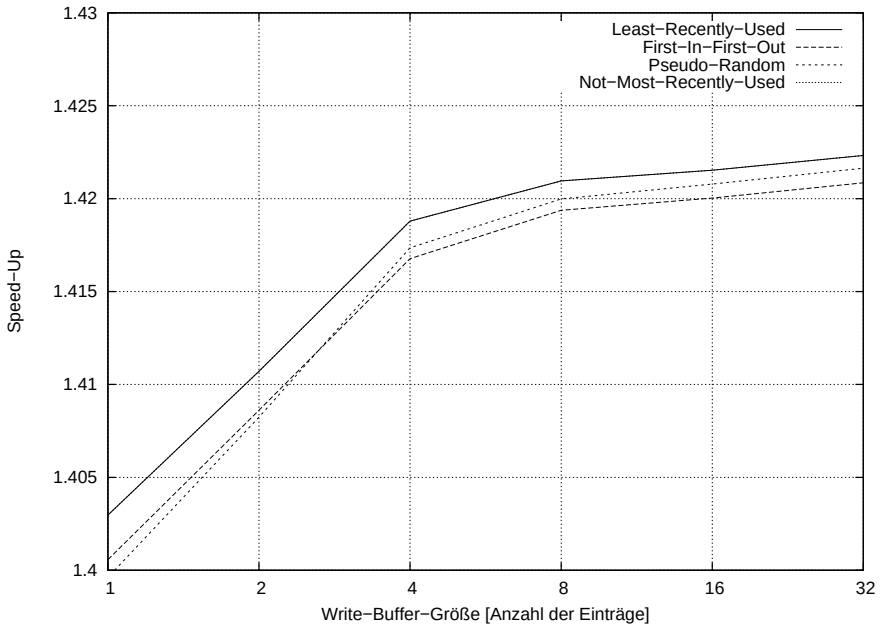

Abbildung 5.6: Speed-Up in Abhängigkeit der Write-Buffer-Größe und der Ersetzungsstrategie.

5.2 Speicherport-Cache

Ein Cache zwischen Speicherport und Speichermanager würde die selben Anfragen wie ein Cache am Kern zu verarbeiten haben. Der Unterschied besteht darin, dass der Offset im Port mit einem Bias-Wert verrechnet wird. Infolgedessen ergeben sich andere Speicheradressen und eine andere Streuung der Einträge im Cache.

Zur Analyse wurden die Mengen der Ergebnisse des Kerns und des Ports direkt verglichen. Jeweils die selben Cache-Konfigurationen wurden gegenübergestellt. Es ergab sich, dass bei der Lese-Rate jedes dritte Ergebnis des Ports besser war und bezüglich der Schreib-Rate jedes sechste Ergebnis höhere Trefferraten aufwies. Auffällig ist, dass der Port bei niedriger Assoziativität günstigere Leistungsdaten aufweist. Hier zeigt sich eventuell eine bessere Streuung durch die Offset-Umrechnungen. Häufig angefragte Speicherwörter werden seltener in das gleiche Set gemapped. Betrachtet man die Geschwindigkeiten, so hat der Port-Cache den Vorteil, dass er Leseoperationen schneller durchführen kann. Dafür ist sein Geschwindigkeitsgewinn im Falle eines Cache-Hits geringer, da die eigentliche Anfrage vom Kern kommend mindestens einen Wartetakt im Speicherport verbringt. Somit lässt sich nur ein kleinerer Teil der Blockierungszeit vermeiden. Die Ergebnisse bestätigen das. Der Port-Cache ist besser, wenn die Blöcke entsprechend groß sind. Allerdings führt dann ein Cache eher zu einem Geschwindigkeitsverlust, als zu einem Gewinn, da das Laden und Verdrängen von Blöcken länger dauert. Diese Zeit lässt sich beim Port-Cache reduzieren. Da man bei der Implementierung immer von optimistischen Hit-Raten ausgehen sollte und das Nachladen von Blöcken nicht die Mehrheit der Fälle darstellt, ist die Priorität die möglichst starke Verringerung der Blockierungszeit. Die Option, den Cache am Speicherport zu implementieren, ist nicht empfehlenswert.

5.3 Hauptspeicher-Cache

Der Cache am Speicher vermeidet die Nachteile eines Caches am Kern, hat aber auch nicht seine Vorteile. Er wäre physikalisch addressiert. Eine Blockierung kann nicht in dem Maße aufgehoben werden, wie ein Cache am Kern dies könnte. Dafür bietet er am Speicher nicht nur die Möglichkeit, Anfragen des Prozessors zu beantworten, sondern auch des Segment-Managers, des Referenz-Managers, des Methoden-Caches und weitere Komponenten wie DMA oder Prozessoren. Auch die Anfragen zur Adressumsetzung werden im Cache vorgehalten. Somit würde eine extra Realisierung eines Translation-Lookaside-Buffers entfallen.

	Lesen	Schreiben	Kopieren	Warten
Crypt	8,9 Takte	3.24 Takte	19.61 Takte	57.10 Takte
Crypt (2 Threads)	8.91 Takte	3.17 Takte	19.58 Takte	44.31 Takte
Crypt (3 Threads)	8.93 Takte	3.17 Takte	19.57 Takte	42.01 Takte
HeapSort	7.10 Takte	3.16 Takte	14.33 Takte	14.01 Takte
RayTracer	8.99 Takte	3.48 Takte	17.32 Takte	17.68 Takte
Sudoku	7.32 Takte	3.06 Takte	13.37 Takte	17.22 Takte
Crypt & HeapSort	8.70 Takte	3.24 Takte	17.98 Takte	37.49 Takte

Tabelle 5.2: Timingdaten am Speicher

Das Diagramm 5.7 zeigt die Hit-Rate für verschiedene Assoziativitäten und Cachegrößen. Für die Blockgröße wurden bei der Simulation 4 Byte angenommen. Der Cache arbeitet für alle Aufrufer.

Bei einer Größe von 4 KByte liegen die Hit-Raten zwischen 96 und 97 %. Das ergibt einen Speed-Up von 1.29 bis 1.30. Der Wechsel von 2 KByte auf 4 KByte statt einer Erhöhung der Assoziativität bei 2 KByte ist sinnvoll. Eine Steigerung der Assoziativität bringt bei diesen Cachegrößen eine starken Komplexitätszuwachs. Bei einer Kapazität von 4 KByte sind die Unterschiede zwischen den Assoziativitäten minimal, weswegen ein direkt abbildender Cache gewählt werden kann.

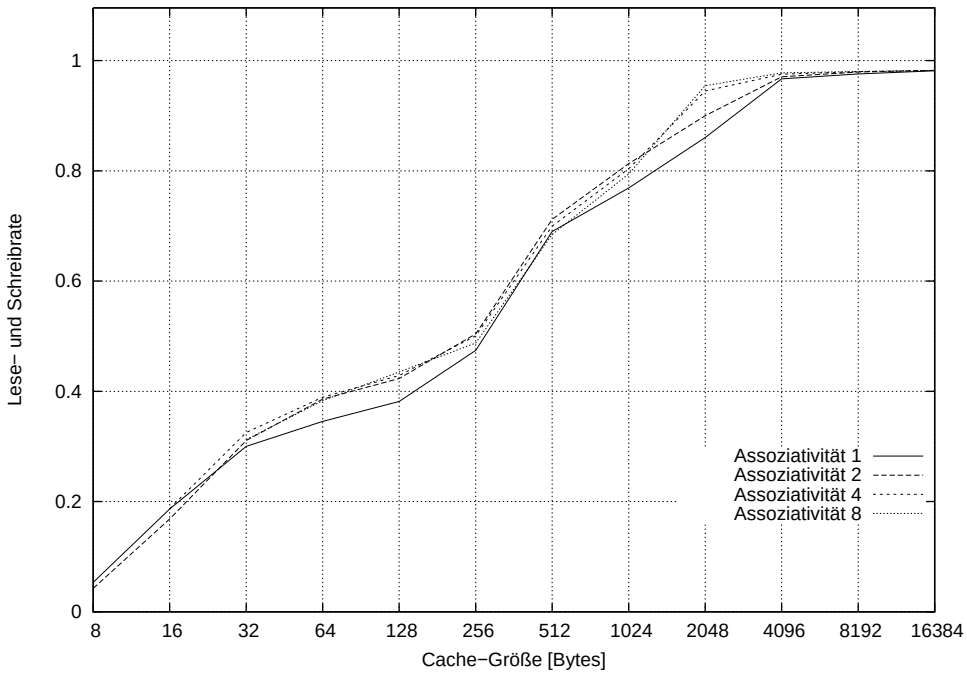

Abbildung 5.7: Hit-Rate in Abhängigkeit der Assoziativität und der Cachegröße

Die Vermutung, das wenn auf Caching der Methodencache-Aufrufe verzichtet wird, die Kapazität reduziert werden kann, lässt sich nicht bestätigen. Die Leistungsdaten liegen für die einzelnen Anwendungen entweder gleichauf oder darunter.

Als nächstes werden die Schreibstrategien und die Blockgröße für einen direkt abbildenden Cache mit 4 KByte Kapazität evaluiert.

Für die dargestellten Ergebnisse gilt, dass die Cachegröße konstant bleibt, also die Anzahl der Blöcke mit steigender Blockgröße sinkt. Die günstigste Lösung bezüglich des Aufwands ergibt sich bei 16 Byte.

Bei den Schreibstrategien verhalten sich Write-Through with Write-Allocation und Copy-Back gleich. Sie haben sie selbe Allokationspolitik. Bei Copy-Back kommt es allerdings ab einer höheren Blockgröße zu Einbußen bezüglich der Geschwindigkeit. Aufgrund von Rückschreiboperationen, die bei größeren Blöcken länger dauern, wird der Bus mehr belastet. In der Simulation wurde ein Write-Buffer der Größe 1 angenommen. Auch hier kommt es zu Verzögerungen aufgrund der Schreiboperationen. Die Anzahl an eingesparten Schreibzyklen ist gering. Der Geschwindigkeitsgewinn wird durch das längere Schreiben überdeckt.

Die Schreibstrategien unterscheiden sich nicht wesentlich in der Geschwindigkeit. Deshalb sollte auf Copy-Back wegen des zusätzlichen Hardwareaufwandes verzichtet werden. Für die weiteren Betrachtungen wird Write-Through with No-Write-Allocation angenommen. Diese Strategie zeigt minimal bessere Leistungsdaten.

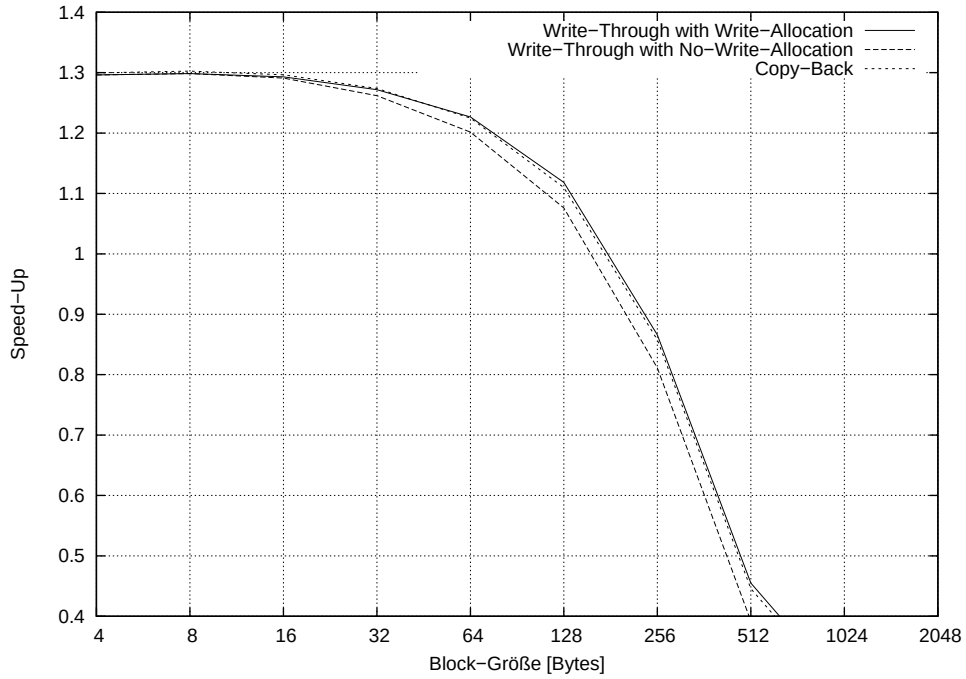

Abbildung 5.8: Speed-Up in Abhängigkeit der Schreibstrategie und der Block-Größe

Die Adressverteilung bewirkt wie beim Kern bessere Ergebnisse, wenn der Hash-Adress-Dekoder mit einer bestimmten Verschiebung gewählt wird. Zwischen Index und Wortbits sollten 9 Tag-Bit liegen. Hierbei ergab sich auch die geringste Standardabweichung. Die Ergebnisse sind für die verschiedenen Anwendungen am konstantesten.

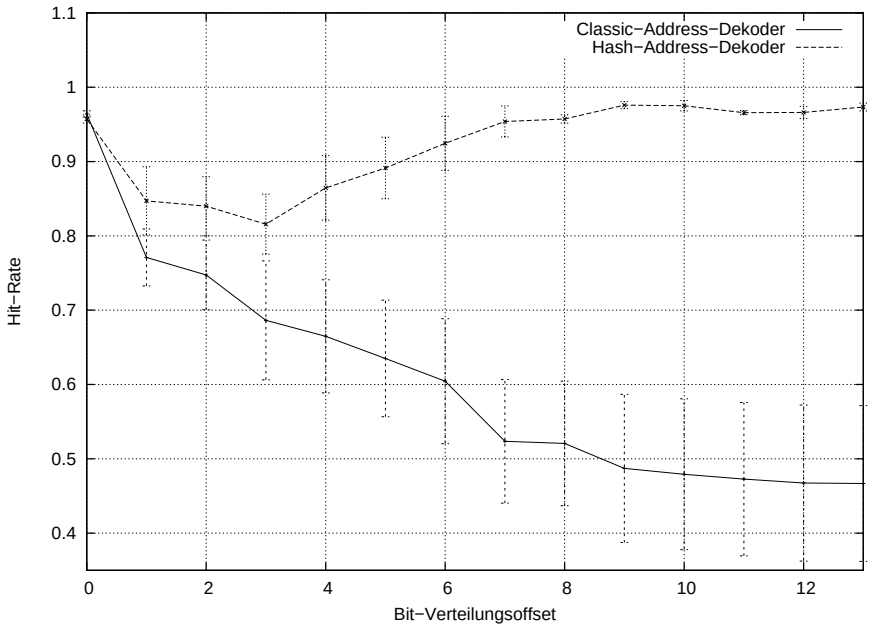

Abbildung 5.9: Hit-Rate in Abhängigkeit der Adressverteilung

Es wurde ein direkt abbildender Cache als beste Variante ermittelt. Dadurch entfällt die Notwendigkeit einer Ersetzungsstrategie.

Durch den Write-Buffer entsteht kein Geschwindigkeitsgewinn. Teilweise verlangsamt er das System durch die zusätzliche Abfrage. Es empfiehlt sich allerdings die Implementierung mit einem Eintrag, um die Write-Through-Anfrage aufzunehmen.

Für den Cache kommt die Problematik hinzu, dass er im Falle eines Multi-Core-Einsatzes mehreren Prozessoren zur Verfügung stehen muss. Diese Situation wurde mit der Crypt & HeapSort-Anwendung simuliert. Sie zeigt einen Speed-Up von 1.19, der unter Berücksichtigung der Ausführungszeiten beider Einzelanwendungen nicht auf Trashing-Effekte oder Überlastungen des Caches hinweist.

Betrachtet man die Aufrufer, so zeigen die Ergebnisse, dass neben dem CPU, dem Refernz- und Segmentmanager auch die Anfragen des Methoden-Caches zwischengespeichert werden sollen. Zwar haben ein Drittel der Ergebnisse eine höhere Hit-Rate für die Variante ohne Methoden-Cache, jedoch bringt bei den Implementierungen, die hohe Leistungsdaten erreichen, ein Caching einen zusätzlichen Leistungsschub. Ein konkretes Muster lies sich nicht erkennen.

5.4 Vergleich

Für die jeweils günstigsten Cache-Konfiguration am Speicher und am Kern wurden die Simulationen in voller Länge durchgeführt und gegenübergestellt.

5.4.1 Kern

Assoziativität: 2

Cache-Größe: 128 Byte

Block-Größe: 8 Byte

Schreibstrategie: Write-Through with No-Write-Allocation

Ersetzungsstrategie: Pseudo-Random

Adress-Dekoder: Hash-Address-Decoder

Index-Verschiebung: 15

Write-Buffer-Größe: 1

Hardwareaufwand: 38582

	Hit-Rate	Read-Hit-Rate	Write-Hit-Rate	Speed-Up
Crypt	0.80	0.831	0.0	1.085
Crypt (2 Threads)	0.785	0.823	0.0	1.069
Crypt (3 Threads)	0.785	0.823	0.0	1.068
HeapSort	0.847	0.93	0.0	1.75
RayTracer	0.85	0.864	0.0	1.06
Sudoku	0.976	0.981	0.0	1.75

Tabelle 5.3: Ergebnisse der Simulation am Kern

5.4.2 Speicher

Assoziativität: 1

Cache-Größe: 4 KByte

Block-Größe: 128 byte

Schreibstrategie: Write-Through with No-Write-Allocation

Ersetzungsstrategie: -

Adress-Dekoder: Hash-Address-Decoder

Index-Verschiebung: 9

Write-Buffer-Größe: 1

Hardwareaufwand: 1341306

	Hit-Rate	Read-Hit-Rate	Write-Hit-Rate	Speed-Up
Crypt	0.993	0.996	0.013	1.135
Crypt (2 Threads)	0.951	0.954	0.241	1.163
Crypt (3 Threads)	0.978	0.981	0.193	1.18
HeapSort	0.978	0.981	0.938	1.345
RayTracer	0.988	0.988	0.268	1.41
Sudoku	0.999	0.999	0.948	1.344
Crypt & HeapSort	0.97	0.974	0.766	1.19

Tabelle 5.4: Ergebnisse der Simulation am Speicher

Eine eindeutige Aussage für eine Implementierung lässt sich nicht treffen. Für den Speicher-Cache würde sprechen, dass er für jede Anwendung eine moderate Beschleunigung aufweist. Der Cache am Kern hingegen beschleunigt nur einzelne Anwendungen, dafür ist der Speed-Up wesentlich höher. Da der Cache am Kern viel preiswerter zu implemtieren ist, spricht vieles für diese Lösung. Gerade für die zukünftigen Multi-Core-Ausbaustufen des SHAPs bieten sich mehrere kleine Caches für die Kerne an, anstatt alle gemeinsam auf Einen zugreifen zu lassen.

5.5 Kritik

Für die Messungen wurden einige Vereinfachungen gemacht. Das Timing ist mit Durchschnittswerten berechnet. Bei der Wartezeit zwischen den Operationen ist diese Annahme aber nicht immer optimal. Der negative Effekt von schnellen, sequentiellen Operationen wird nicht beachtet, wenn diese durch längere Wartezeiten im Durchschnittswert ausgeglichen werden.

Die Blockierungszeiten sind idealisierte Werte. Gerade am Speicher arbeiten mehrere nebenläufige Komponenten. Die Blockierung eines Aufrufers hat nicht unbedingt die Blockierung aller Weiteren zur Folge. Hier gab es aber keine Alternative zur Messung. Der Schwerpunkt der Operationen liegt am Speicher auf dem Methoden-Cache und den Daten-Anfragen des CPUs. Es kann angenommen werden, dass der CPU komplett blockiert, wenn eine der beiden Anfragen gestellt wird. Das Vorkommen der anderen Aufrufer ist im Vergleich zu diesen beiden eher gering.

Weiterhin wurden Vereinfachungen bezüglich der Schreibstrategie gemacht. Gerade der Write-Buffer eignet sich auch zum Zwischenspeichern der Ergebnisse um somit die Blockierung aufzuheben. Der Effekt ist aber aufgrund der geringen Anzahl von Schreiboperationen eher gering.

6 Zusammenfassung

In dieser Arbeit wurde eine Evaluation möglicher Cache-Strategien für die SHAP-Microarchitektur durchgeführt. Die verwendeten analytischen Modelle erwiesen sich dabei aufgrund des spezialisierten Anwendungsfalls als ungenügend und gingen in die Endauswertung nicht mit ein. Sie dienten nur bei einer ersten Übersicht der Evaluationsraums als Orientierungshilfe. Die eigentliche Arbeit wurde mit einen Cache-Simulator durchgeführt, der speziell für die Anforderungen angepasst wurde. So ist es möglich gewesen, die verschiedenen Aspekte der Implementierungspunkte zu beachten und genaue Aussagen in Hinblick auf Treffer-Raten und Zeitgewinnen zu liefern. Gerade das zweite Leistungsmerkmal ist entscheidend, da sich die verschiedenen Cache-Positionen und Parameter darin am deutlichsten wiederspiegeln.

Das Ergebnis lässt leider keine endgültige Aussage über eine Implementierung zu. Sowohl ein Cache am Prozessor, wie am Speicher bieten jeweils Vor- und Nachteile. So können mit einem Prozessorcache verschiedene Anwendungen in hohem Maße beschleunigt werden, während andere fast nicht profitieren. Ein Cache am Speicher muss mit höherem Aufwand realisiert werden, liefert jedoch für alle Anwendungen eine gewisse Beschleunigung. Die Tendenz geht zur ersten Variante. Gerade für eingebettete Systeme sind Fragen der Chip-Fläche und der Leistungsaufnahme essentiell. Es sollte auf kompakte Lösungen zurückgegriffen werden. Ein Cache am Prozessor stellt so eine Lösung dar.

Literaturverzeichnis

[1] Qimonda AG: *HYB18TC512[80/16]0CF 512-Mbit Double-Data-Rate-Two SDRAM Internet Data Sheet*, Rev. 1.10, Edition 2008-07, München, Germany, 2008

[2] Samsung Electronics: *K7S3236U4C/K7S3218U4C 36Mb QDRII+ SRAM Specification*, Rev. 1.0, August 2008

[3] Hennessy, J. L.; Patterson, D. A.: *Computer Architecture: A Quantitative Approach.*, third edition, Morgan Kaufmann, San Mateo, CA, USA, 2002.

[4] Denning, P. J.: *The Working Set Model For Program Behavior*, In: SOSP '67 Proceedings of the first ACM symposium on Operating System Principles, ACM, New York, USA, 1967, S. 15.1 - 15.2

[5] Denning, P. J.: *Virtual Memory*, In: ACM Computing Surveys (CSUR), ACM, New York, USA; September 1970, S. 153 - 189 Volume 2 , Issue 3

[6] Tanenbaum, A. S.; Goodman, J.: *Computerarchitektur - Strukturen, Konzepte, Grundlagen*, 4. Auflage, Pearson Studium, München, Germany, 2005, ISBN 3-8273-7148-1

[7] Flik, T: *Mikroprozessortechnik und Rechnerstrukturen*, 7. Auflage, Springer-Verlag, Berlin, Heidelberg, 2005, ISBN 3-540-22270-7

[8] Liebig, H: *Rechnerorganisation: die Prinzipien*, 3. Auflage, Springer-Verlag, Berlin, Heidelberg, 2003, ISBN 3-540-00027-5

[9] Advanced Micro Devices, Inc.: *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, Rev. 3.14, Pub.-No. 24593, September 2007

[10] Heath, S.: *Embedded Systems Design*, Second Edition, Newnes, 2003, ISBN 0750655461, 9780750655460

[11] Wright, G; Seidl, M. L.; Wolczko, M: *An object-aware memory architecture*, Sun Microsystems, Sun Labs, Menlo Park, CA, USA, Februar 2005

[12] Vijaykrishnan, N.; Ranganathan, N, Gadekarla, R: *Object-Oriented Architectural Support for a Java Processor*, In: ECOOṔ98, the 12th European Conference on Object-Oriented Programming, July 1998

[13] Vijaykrishnan, N.; Ranganathan, N: *Supporting object accesses in a Java processor*, Dept. of Comput. Sci. and Eng., Pennsylvania State Univ., University Park, PA;, In: Computers and Digital Techniques, IEE Proceedings, Volume: 147, Issue: 6, S. 435-443, November 2000

[14] Agarwal, A; Horowitz, M; Hennessy J.: *An Analytical Cache Model*, Computer Systems Laboratory, Stanford University, In: ACM Transactions on Computer Systems, Vol. 7, No. 2, S. 184-215, Mai 1989

[15] Brehob, M; Enbody, R.J.: *An analytical model of locality and caching*, Technical Report MSUCPS:TR99-31, Michigan State University, Department of Computer Science and Engineering, 1999

[16] Brehob, M. W.: *On The Mathematics Of Caching*, Dissertation, Michigan State University, Department of Computer Science and Engineering, 2003

[17] Weinberg, J; McCracken, M. O.; Strohmaier, E; Snavely, A: *Quantifying Locality In The Memory Access Patterns of HPC Applications*, In: Proceedings of the 2005 ACM/IEEE conference on Supercomputing, S. 50, 2005, ISBN 1-59593-061-2

[18] Preußer, T. B.; Zabel, M; Reichel, P: *The SHAP Microarchitecture and Java Virtual Machine*, Technische Berichte, Institut für Technische Informatik, Technische Universität Dresden, Germany, April 2007

[19] Zabel, M.; Reichel, P; Spallek, R. G.: *Multi-Port-Speichermanager für die Java-Plattform SHAP*, Dresdner Arbeitstagung Schaltungs- und Systementwurf (DASS 2008), Fraunhoder-Institut für Integrierte Schaltungen, Institutsteil Entwurfsautomatisierung, S. 143 - 147, 2008

[20] Zabel, M; Preußer, T. B; Reichel, P: *SHAP Reference Manual*, Technische Universität Dresden, Fakultät Informatik, Institut für Technische Informatik, 8. April 2008

[21] Sun Microsystems: *picoJava I Data Sheet - Java Processor Core*, Palo Alto, CA, USA, Dezember 1997

[22] Puffitsch, W; Schoeberl, M.: *picoJava-II in an FPGA*, JTRES '07: Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems, ACM, September 2007

[23] aJile Systems, Inc.: *aJ-100 Real-time Low Power Java Processor*, Reference Manual, Version 2.1, 6. Dezember 2001

[24] aJile Systems, Inc.: *Real-Time Low-power Network Direct Execution Microprocessor for the Java Platform aJ-102*, Preliminary Product Brief

[25] aJile Systems, Inc.: *Real-Time Low-power Network Direct Execution Microprocessor for the Java Platform aJ-200*, Preliminary Product Brief

[26] Schoeberl, M: *JOP Reference Handbook: Building Embedded Systems with a Java Processor*, 18. September 2008, ISBN 978-1438239699

[27] Java Grande Forum: www.javagrande.org

[28] Edinburgh Parallel Computing Centre, Edinburgh University Scotland: Java Grande Benchmarking, www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html

[29] Preußer, T. B.: *Entwicklung eines Prozessorsimulators unter besonderer Berücksichtigung des Organic Computing*, Diplomarbeit, Institut für Technische Informatik, Fakultät Informatik, Technische Universität Dresden, Oktober 2003