Diplomarbeit

Leistungssteigerung der Speicherarchitektur des SHAP-Mehrkernprozessors

Andrej Olunczek

geboren am 20. Februar 1984 in Cottbus

Matrikelnummer: 30 66 276

Januar 2012

Technische Universität Dresden
Fakultät Informatik
Institut für Technische Informatik

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

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich diese Diplomarbeit vollkommen selbständig, unter ausschließlicher Verwendung der angegebenen Hilfsmittel und Quellen angefertigt habe. Alle Zitate sind als solche gekennzeichnet.

Dresden, 31. Januar 2012

Andrej Olunczek

Marken

Die in dieser Arbeit genannten Marken sind Handelsmarken und Markennamen ihrer jeweiligen Inhaber und deren Eigentum. Die Wiedergabe von Marken, Warenbezeichnungen u.ä. in diesem Dokument berechtigt auch ohne besondere Kennzeichnung nicht zur Annahme, dass diese frei von Schutzrechten sind und frei benutzt werden dürfen.

ZUSAMMENFASSUNG

In der vorliegenden Arbeit wird der Entwurf eines Objekt-Caches für den Java-Prozessor SHAP vorgestellt. Der Objekt-Cache soll in erster Linie ein höheres Maß an Parallelverarbeitung auf Thread-Ebene ermöglichen. Da die Parallelverarbeitung durch einen gemeinsam genutzten Speicherzugang limitiert wird, sollen häufig genutzte Daten im Cache gehalten werden, um so den Speicher-Bus zu entlastet.

Dem Entwurf geht eine Analyse des bestehenden Systems voraus, um abzugrenzen, wie der zu implementierende Cache aufgebaut sein muss, um möglichst hohen Nutzen zu haben. Die Analyse beinhaltet aus diesem Grund Simulationen verschiedener Cache-Konfigurationen, die miteinander verglichen werden. Die prototypische Implementation wurde auf einem FPGA getestet und die Leistungssteigerung ausgewertet. So konnte der maximale Speed-Up um bis zu 86% erhöht werden. Die absolute Rechenleistung wurde sogar um bis zu 123% gesteigert.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen

AES	Advanced Encryption Standard
BRAM	Block-RAM
CMP	Chip Multiprocessor
CPU	Central Processing Unit
DDR-RAM	Double Data Rate RAM
DMA	Direct Memory Access
FIFO	First In - First Out
FPGA	Field Programmable Gate Array
GC	Garbage Collector
HISC	High Level Instruction Set Computer
IDEA	International Data Encryption Algorithm
JGF	Java Grande Forum
JOP	Java Optimized Processor
JVM	Java Virtual Machine
LRU	Least Recently Used
LUT	Lookup Table
MMU	Memory Management Unit
NMRU	Not Most Recently Used
RAM	Random Access Memory
SHAP	Secure Hardware Agent Platform
SRAM	Static RAM
TLB	Translation Lookaside Buffer
UART	Universal Asynchronous Receiver / Transmitter
UDP/IP	User Datagram Protocol / Internet Protocol
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits
VLSI	Very Large Scale Integration

1 Einleitung

Mit der Möglichkeit mehrere Milliarden Transistoren auf einem Chip unterzubringen, steigt auch die Möglichkeit zunehmend mehr Rechenkerne innerhalb eines Prozessor-Chips zu platzieren. Solche Mehrkernprozessoren (im englischen auch Chip-Multiprocessors, CMPs) sind einfacher zu entwickeln und kostengünstiger herzustellen, als einen einzelnen komplexeren Einkernprozessor mit ähnlichen Leistungsdaten [LSK04]. Die Kommunikation zwischen den Kernen eines CMPs ist wesentlich einfacher bereitzustellen, als externe Kommunikationsmöglichkeiten zwischen mehreren einzelnen Einkernprozessoren. Der Nachteil von CMPs ist die Tatsache, dass meist alle Rechenkerne über eine gemeinsame Schnittstelle mit dem externen Speichersystem kommunizieren müssen. Das Problem, dass es zunehmend schwieriger wird, die immer schneller werdenden Prozessoren schnell genug mit Daten versorgen zu können, wird als „Memory Wall“ bezeichnet. Um den Zugriff auf Daten zu beschleunigen und den externen Speicher-Bus zu entlasten, werden Caches im Chip integriert, um häufig benötigte Daten zwischenzuspeichern.

Die Vorteile eines Caches sollen jetzt auch für den Java-Prozessor SHAP genutzt werden. Das primäre Ziel dieser Arbeit ist es nicht, die Zugriffszeit auf den Speicher zu verkürzen, sondern möglichst viele Speicherzugriffe abzufangen, um mehr Bandbreite auf dem Speicher-Bus frei zu bekommen. Die freigewordene Speicherbandbreite soll dazu genutzt werden, die Parallelverarbeitung auf einer größeren Anzahl von Rechenkernen zu ermöglichen. Bisher ergibt sich ab einer bestimmten Anzahl Kernen kein weiterer Leistungsgewinn, da der Speicher-Bus ausgelastet ist. Nutzt zum Beispiel ein Kern 12% der Bandbreite, können maximal acht Kerne ausgelastet werden, da ab dem neunten Kern alle Kerne Wartezyklen einfügen müssen, um auf Daten zu warten. Schafft man es nun z.B. die benötigte Bandbreite auf 6% zu halbieren, können 16 Kerne ausgelastet werden, ohne dass es zu einer Limitierung der Rechenleistung der einzelnen Kerne kommt.

Der Speicherzugriff ist stark abhängig von der jeweiligen Prozessorarchitektur, die üblichen Cache-Architekturen müssen daher hinsichtlich der Art und des Ablaufs der Speicherzugriffe angepasst und parametrisiert werden. Da SHAP Java-Bytecode als Befehlssatz nutzt, sind die Daten in Objekte gegliedert. Der Speicherzugriff erfolgt also in einer speziellen objektorientierten Form, wobei die Objekte im Speicher sowohl die Befehle als auch die Daten enthalten. Für die Befehle gibt es schon einen Methoden-Cache, diese Arbeit untersucht daher nur den Zugriff auf die Daten. Auf Grund des speziellen Speicherzugriffs beziehen sich die Analysen in dieser Arbeit nur auf SHAP als Kontext.

In Kapitel 2 wird der Speicherzugriff näher erläutert. Zudem werden Möglichkeiten für den Einsatz eines Caches allgemein und an Beispielen bisheriger Ansätze erläutert. Zudem wird

auch kurz die Mehrkernarchitektur von SHAP skizziert. Kapitel 3 beschäftigt sich mit der Analyse des bisherigen Systems, dem Ablauf und der Verteilung von Speicherzugriffen. Des Weiteren werden mehrere Cache-Varianten simuliert, um eine Vorauswahl für den Aufbau und die Dimensionierung eines Caches zu treffen. Das 4. Kapitel stellt den Entwurf und die prototypische Implementierung eines Objekt-Caches für den SHAP-Mehrkernprozessor vor. Weiterhin erfolgt die Auswertung der Leistungssteigerung gegenüber der bestehenden Lösung und eine Abschätzung des Ressourcenbedarfs auf einem FPGA. Schlussendlich werden in Kapitel 5 die Ergebnisse der Arbeit zusammengefasst und ein kurzer Ausblick gegeben.

2 Grundlagen

2.1 Objektorientierte Speicherzugriffe

Da Java eine objektorientierte Programmiersprache ist, ist auch der Speicher in Form von Objekten strukturiert. Die Zuteilung von Speicherplatz für neue Objekte und der Zugriff darauf wird von der Memory Management Unit (MMU) geregelt. In dieser wird der gesamte Speicher verwaltet. Wie in der Spezifikation der Java Virtual Machine (JVM) festgelegt, entfernt ein integrierter Garbage Collector (GC) nicht mehr benötigte Objekte, da Java als Programmiersprache kein manuelles Löschen von Objekten vorsieht.

Im SHAP wird eine Variante der Copying Garbage Collection genutzt [Rei07, Rei08]. Nach dem Entfernen der nicht mehr benötigten Objekte werden alle weiterhin bestehenden Objekte zusammengerückt, um einen definierten, zusammenhängenden freien Speicherbereich für neue Objekte zu haben. Da sich durch diesen Vorgang die physische Position der Objekte im Speicher ändert, gibt die MMU nach außen nur eine Objektreferenz aus. Diese ist eine virtuelle Adresse, das sogenannte Handle, mit dem das Objekt immer adressiert werden kann. Intern verwaltet die MMU eine Tabelle, mit deren Hilfe die Objektreferenzen in reale physische Adressen umgewandelt werden. Durch diese sogenannte Indirektion wird vermieden, dass man nach dem Verschieben eines Objektes alle Verweise auf das Objekt im System finden und ändern muss.

Wenn die CPU auf ein Wort in einem Objekt zugreifen will, geschieht dass mit Hilfe der Objektreferenz und einem Offset-Wert für das jeweilige Datenwort innerhalb des Objektes. Dafür wird zuerst die Objektreferenz in die physikalische Adresse aufgelöst und im Anschluss

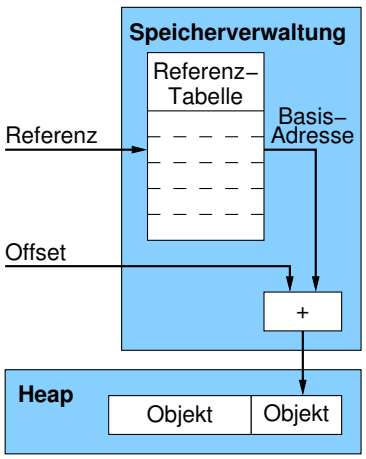


Abbildung 2.1: Verwaltung von Objekten im Speicher [ZRS08]

der Offset addiert, um die reale Adresse des gesuchten Datenwortes zu ermitteln (Abbildung 2.1).

Da die Tabelle mit der Zuordnung zwischen Objektreferenz und physikalischer Adresse mehrere tausend Einträge umfasst und somit sehr groß sein kann, wird auch diese beim SHAP im externen Speicher gehalten. Es gibt demnach zwei Ursachen, warum auf den Speicher zugegriffen wird. Das sind der Zugriffe auf die Referenztabelle durch die MMU und Zugriffe auf den sogenannten Heap. Im Heap liegen die Klassen mit den enthaltenen Methoden und die dynamisch angelegten Objekte, die sich von diesen Klassen ableiten. Die Zugriffe auf Klassen und Objekte unterteilen sich nochmal in zwei unterschiedliche Arten:

- **Konstante Daten**: Methodentabellen, der Programmcode der einzelnen Methoden, Referenzen der Objekte auf ihre Klasse und die Länge von Arrays.
- **Veränderliche Daten**: statische Felder, Datenfelder der Objekte und Daten der Arrays.

2.2 Caches

2.2.1 Motivation

Um die Leistungswerte der Speicherarchitektur zu verbessern, gibt es mehrere Möglichkeiten. Im Allgemeinen werden dazu verschiedene Cache-Strukturen verwendet, um Speicherzugriffe abzufangen und zwischenzuspeichern, ohne jedes Mal direkt auf den externen Hauptspeicher zugreifen zu müssen [HP07, PH09]. Cache-Module nutzen die Eigenschaften von Programmabläufen aus, dass oftmals in zeitlich kurzen Abständen auf die gleichen Daten (zeitliche Lokalität) oder benachbarte Daten (räumliche Lokalität) zugegriffen wird.

2.2.2 Allgemeiner Aufbau

Verschiedene Caches unterscheiden sich nach Typ, Assoziativität, Ersetzungsstrategie und Schreibstrategie. Aber alle Caches haben zwei wesentliche Hauptbestandteile.

Das ist zum einen eine Tag-Einheit, die ermittelt, ob ein Wert im Cache liegt (Cache-Hit). Liegt der angeforderte Wert nicht im Cache (Cache-Miss), wird zusätzlich ermittelt, welche Zeile durch den neuen Wert zu ersetzen ist. Für assoziative Caches gibt es die unterschiedlichsten Ersetzungsstrategien, um aus mehreren möglichen Cache-Zeilen eine auszuwählen. Die Tag-Einheit speichert auch sogenannte Valid-Bits ab, die die Gültigkeit der Daten in der Cache-Zeile markieren. Je nach Schreibstrategie werden Daten auch im Cache gespeichert, ohne diese an den Hauptspeicher weiterzugeben. Um diese Daten zu markieren, damit sie später nicht aus Versehen überschrieben werden, werden in der Tag-Einheit auch sogenannte Dirty-Bits gespeichert. Vor dem Überschreiben können die Daten dann in den Hauptspeicher transferiert werden.

Der zweite wichtige Teil des Caches ist der Cache-Speicher, der die einzelnen Cache-Zeilen enthält. Aus welcher Cache-Zeile die angeforderten Daten gelesen werden oder in welche Zei-

le neue Daten geschrieben werden hängt vom Ergebnis der Tag-Einheit ab. Welche Daten im Cache zwischengespeichert werden hängt vom jeweiligen Typ des Caches ab.

2.2.3 Typen

Unterteilt wird allgemein zwischen Daten-Cache und Befehls-Cache (Instruction-Cache). Wobei man die Daten-Caches noch weiter unterteilen kann, je nach Art der Daten, die gespeichert werden. Das können Konstanten, statische Variablen, Stack-Daten oder dynamische Daten aus dem Heap sein [SPH09, Sch09]. Es können aber nicht nur Daten- und Befehls-Caches, sondern auch Kombination von beiden sein, sogenannte Unified Caches.

2.2.4 Assoziativität

2.2.4.1 Direkt Abbildender Cache

Der direkt abbildende Cache (direct-mapped) ist die einfachste Art einen Cache aufzubauen (Abbildung 2.2). Ein Teil der Speicheradresse, der Index-Teil, gibt genau an, in welche Cache-Zeile die Daten zu einer bestimmten Speicheradresse gehören. Über einen Vergleich des sogenannten Tag-Teiles der Adresse kann man herausfinden, ob der im Cache liegende Wert zur gewünschten Speicheradresse gehört oder nicht. In der Regel umfasst eine Cache-Zeile auch mehrere Speicherwörter, so dass die niederwertigsten Bits der Adresse das gesuchte Wort innerhalb der Cache-Zeile adressieren. Nachteil dieser Variante ist, dass sich unter Umständen zwei häufig genutzte Datensätze auf dieselbe Cache-Zeile abbilden und sich somit immer wieder gegenseitig verdrängen [Ale09].

2.2.4.2 n-fach satzassoziativer Cache

Um dem Problem entgegenzuwirken, dass häufig genutzte Daten auf die selbe Cache-Zeile abgebildet werden, nutzt man n-fach satzassoziative Cache-Strukturen [HP07]. Der Cache besteht dann quasi aus n baugleichen direkt abbildenden Caches (siehe Abbildung 2.3). Dabei muss

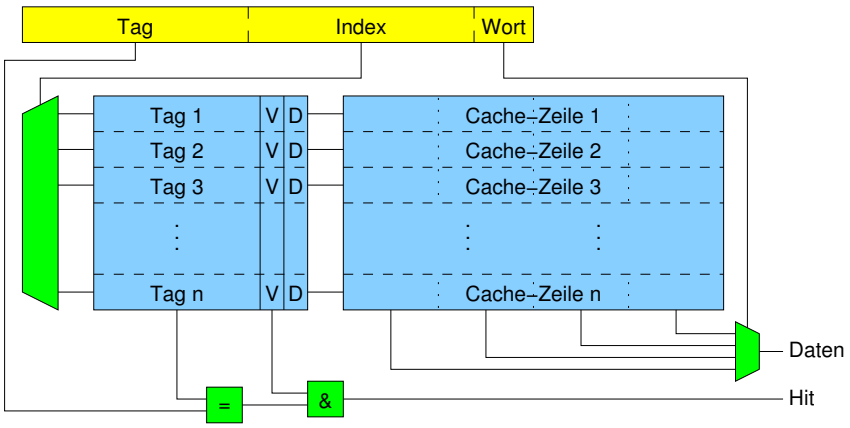


Abbildung 2.2: Schema für einen direkt abbildenden Cache

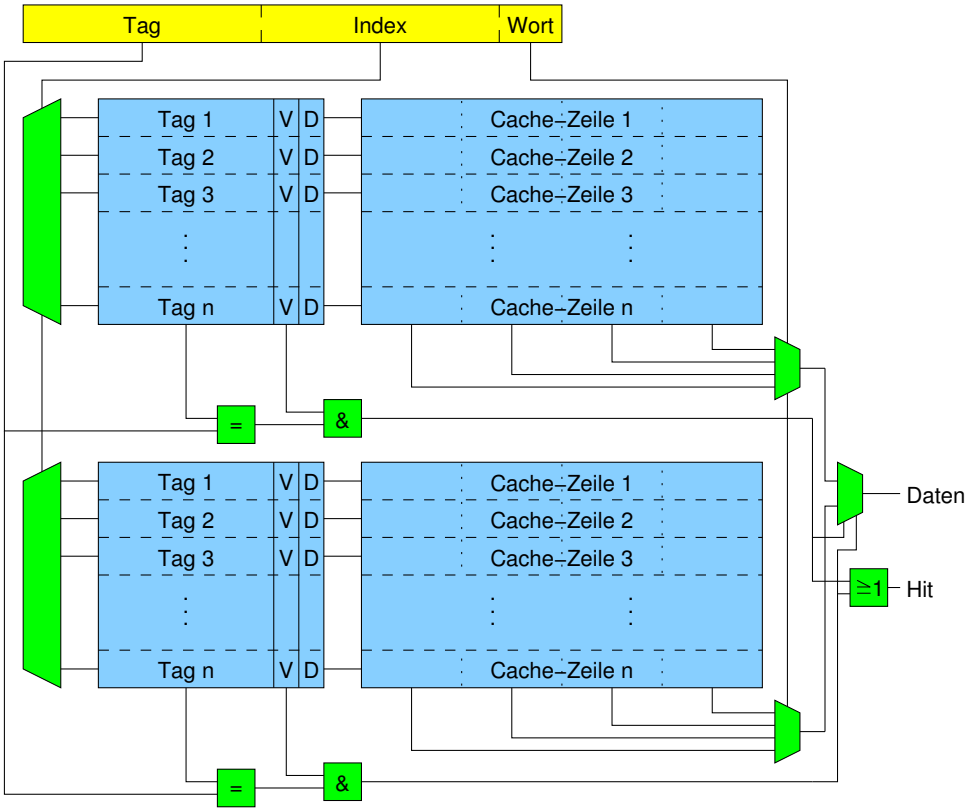


Abbildung 2.3: Schema für einen n-fach satzassoziativen Cache

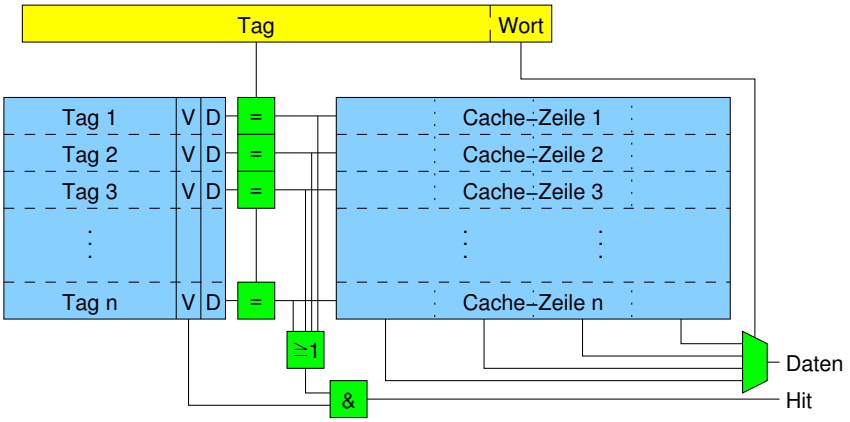


Abbildung 2.4: Schema für einen vollassoziativen Cache

aber eine zusätzlich benötigte Logik entscheiden, in welchen der n möglichen Cache-Zeilen ein neuer Wert geschrieben werden soll. Eine Gruppe aus n Cache-Zeilen einer Index-Adresse nennt man einen Satz, deshalb spricht man auch von Satzassoziativität. Außer zusätzlicher Logik für die Ersetzungsstrategie, die entscheidet, welche der n Cache-Zeilen überschrieben wird, benötigt man nun aber auch n Vergleicher, um festzustellen, ob ein angeforderter Datensatz auch im Cache liegt. Es ist ein höherer Hardwareaufwand zu erwarten, als bei einem direkt abbildenden Cache.

2.2.4.3 Vollassoziativer Cache

Die höchste Steigerung der Assoziativität ist der vollassoziative Cache. Dort kann prinzipiell jeder Datenblock in jeder beliebigen Cache-Zeile stehen. Dazu ist aber zum einen eine komplexe Logik nötig, die entscheidet, welcher Datensatz in welcher Cache-Zeile gespeichert werden soll. Zum anderen ist für jede Zeile auch ein eigener Vergleicher nötig, um festzustellen, ob der angeforderte Datensatz im Cache liegt (Abbildung 2.4). Der Hardwareaufwand ist also sehr hoch [PH09]. Ein wesentliches Kriterium für die Leistungsfähigkeit dieses Cache-Aufbaues ist die Ersetzungsstrategie, die entscheidet, welcher Datensatz ersetzt wird, wenn der Cache voll ist.

2.2.5 Ersetzungsstrategien

Es gibt viele verschiedene Ersetzungsstrategien [Ale09, Wik12], zum Beispiel:

- **First-In-First-Out (FIFO)**: Bei dem FIFO-Prinzip wird derjenige Wert überschrieben, der am längsten im Cache liegt. Damit werden allerdings Datensätze, die häufig benötigt werden, immer wieder aus dem Cache verdrängt und kurz danach wieder neu angelegt [Ale09].
- **Random**: Im Random-Modus wird zufällig ein Wert ausgewählt, der dann ersetzt wird.
- **Least-Recently-Used (LRU)**: Der LRU-Modus ersetzt die am längsten ungenutzte Cache-Zeile. Diese Ersetzungsstrategie ist aber mit erheblichem Hardwareaufwand verbunden, da immer gespeichert werden muss, in welcher Reihenfolge auf die Cache-Zeilen zugegriffen wurde [Ale09, Wik12].
- **Not-Most-Recently-Used (NMRU)**: Reduziert den Aufwand von LRU, in dem nur gespeichert wird, auf welche Zeile als letztes zugegriffen wurde. Zwischen allen anderen Cache-Zeilen wird dann zufällig eine ausgewählt, die ersetzt wird.

2.2.6 Schreibstrategien

Ein weiterer Punkt ist das Cache-Verhalten bei Schreibzugriffen. Es gibt zwei wesentliche Formen: Write-Back und Write-Through.

Beim Write-Back wird ein zu schreibendes Datum nur in den Cache geschrieben, nicht in den Hauptspeicher. Veränderte Werte im Cache werden durch ein Dirty-Bit markiert und bei Verdrängung oder dem Leeren des Caches in den Hauptspeicher zurückgeschrieben. Bei einem Cache-Miss kann es dann zu längeren Latenzen kommen, wenn doch einmal Daten zurückgeschrieben werden müssen.

Bei der zweiten Variante, dem Write-Through, werden die neuen Daten auch in den Hauptspeicher kopiert. Dadurch hat man zwar keinen Geschwindigkeitsvorteil bei Schreiboperationen, dafür gibt es keine nachträgliche Latenzen, wenn Daten zurückgeschrieben werden müssen.

Für den Fall, dass der zu schreibende Wert nicht im Cache liegt gibt es zwei Strategien. Der Cache wird entweder komplett ignoriert (No-Write-Allocation) oder es wird eine neue Cache-Zeile angelegt, in der der Wert dann gespeichert wird (Write-Allocation). In der Regel wird Write-Back mit Write-Allocation und Write-Through mit No-Write-Allocation verwendet [HP07].

2.2.7 Hierarchien

In Mehrkernprozessoren werden oft mehrere Cache-Ebenen benutzt. So kann jeder Kern seinen eigenen Cache haben, zusätzlich kann aber auch ein weiterer Cache zur Verfügung gestellt werden, der von allen Kernen genutzt wird. Man spricht hier von einer Cache-Hierarchie (Abbildung 2.5).

In den unterschiedlichen Ebenen können durchaus verschiedene Cache-Typen zum Einsatz kommen. So kann zum Beispiel in der untersten Cache-Ebene (nahe dem Prozessor) ein geteilter Cache vorhanden sein, also ein Daten- und ein Befehls-Cache. Diese greifen dann beide über einen gemeinsamen Unified Cache einer höheren Ebene auf den Speicher zu.

Bei mehreren Cache-Ebenen wird zudem unterschieden, ob gleiche Daten immer in allen Ebenen verfügbar sind (Inclusive) oder nur in jeweils einer der Ebenen (Exclusive). Es existieren aber auch Mischformen von Beidem [LJW+09].

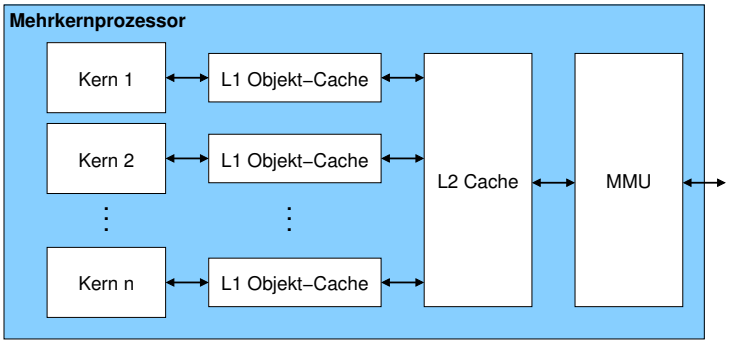


Abbildung 2.5: Cache-Hierarchie

2.2.8 Kohärenz

Laufen mehrere Kerne mit je einem eigenen Cache parallel, kann es zu Kohärenzproblemen kommen. Immer dann, wenn ein Kern Daten in den Hauptspeicher oder den eigenen Cache schreibt, die aber zuvor schon von einem anderen Kern in dessen Cache zwischengespeichert wurden, muss diesem Cache mitgeteilt werden, dass sich die Daten geändert haben und er sie als ungültig markieren oder aktualisieren muss. Für dieses Problem gibt es verschiedene Kohärenzprotokolle, mit deren Hilfe die Caches untereinander kommunizieren, um festzustellen, welche Daten im eigenen Cache nicht mehr aktuell sind [PH09].

Dieses Problem ist in der JVM etwas entschärft, da jeder Thread in einer eigenen Kopie des Hauptspeichers arbeitet, bzw. arbeiten kann [LY99]. Es kann also jeder Thread einen eigenen lokalen Cache nutzen. Eine Synchronisation mit dem Hauptspeicher ist nur beim Betreten eines `synchronized`-Bereiches oder beim Zugriff auf eine mit `volatile` gekennzeichnete Variable notwendig.

2.3 Bisherige Ansätze

2.3.1 Lokaler Objekt-Cache

Eine Möglichkeit zur Beschleunigung von Speicherzugriffen ist das Zwischenspeichern von einzelnen Objekten, wie es beim Java Optimized Processor (JOP) realisiert ist [HPS10, Sch11]. Dazu wird pro Rechenkern ein Cache implementiert.

Es wird zweckmäßiger Weise pro Cache-Zeile nur ein Objekt abgelegt (Abbildung 2.6), allerdings nur so viele Wörter, wie die Cache-Zeile fassen kann. Es können nur kleinere Objekte komplett im Cache gehalten werden, bei größeren Objekten nur der vordere Bereich, also die ersten Offsets. Zum Cache-Aufbau gibt es unterschiedliche Ansätze. So wird bei einem Zugriff auf ein Objekt entweder gleich die komplette zugehörige Cache-Zeile gefüllt oder es werden nur benötigte Wörter in den Cache geladen und dabei jedes Wort einzeln mit einem Valid-Bit markiert ob es im Cache liegt. Es wurde auch eine Variante untersucht, zu jedem Objekt die physikalische Adresse mit zwischenzuspeichern [HPS10].

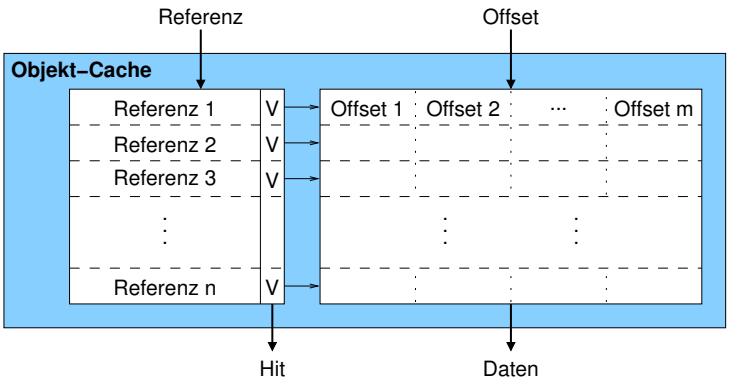


Abbildung 2.6: Schema für einen Objekt-Cache

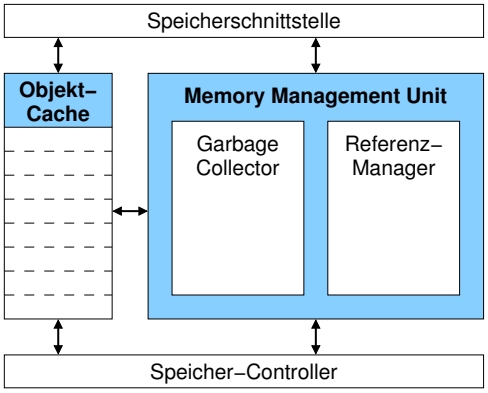


Abbildung 2.7: Ein globaler Cache neben der MMU

Um das Problem mit der Kohärenz zu lösen, werden zu schreibende Daten sofort in den Hauptspeicher transferiert. Beim Betreten eines `monitorenter`-Bereiches oder wenn auf eine mit `volatile` gekennzeichnete Variable zugegriffen wird, wird der Cache geleert, um den aktuellen Stand der Daten aus dem Hauptspeicher holen zu können [Sch11].

2.3.2 Globaler Objekt-Cache

In [JSQ07] wird eine Architektur vorgestellt, in der der Objekt-Cache in der MMU beziehungsweise parallel dazu gehalten wird (Abbildung 2.7). Außer einer Beschleunigung des Speicherzugriffes wird hier auch ausgenutzt, dass mehr Speicherbandbreite für die MMU zur Verfügung steht, die für das Kompaktieren des Speichers nach der Garbage Collection benötigt wird. Für die Adressierung einer Cache-Zeile wird nicht die Objekt-Referenz genutzt, sondern eine Verkettung von Teilen der Referenz und des Offsets.

2.3.3 Cache-Hierarchien

Ein Beispiel für eine Architektur, die Cache-Hierarchien nutzt, wird in [LJW+09] vorgestellt. Diese Architektur hat neun Kerne, wobei jeder Kern seinen eigenen Level-1-Cache hat. Jeweils drei Kerne nutzen einen gemeinsamen Level-2-Cache. In diesem L2-Cache hat jeder der drei Kerne seinen eigenen Bereich, der mit dem L1-Cache des jeweiligen Kernes verbunden ist. Zusätzlich gibt es im L2-Cache einen Bereich, den sich alle drei Kerne teilen. Auf diesen gemeinsamen Bereich greifen alle drei Kerne auch direkt und ohne Umweg über den L1-Cache zu.

2.3.4 Objektverwaltung im Cache

Im High Level Instruction Set Computer für Java (jHISC) [SLF04] werden kleine kurzlebige Objekte nur im Cache gehalten, also gar nicht in den Hauptspeicher transferiert. Diese Möglichkeit wurde auch schon in [CG93] diskutiert. Dazu werden Objekte, die nicht größer als die Cache-Zeile sind, direkt im Cache allokiert. Über das Zählen von Referenzen auf dieses Objekt

wird im Cache dessen Erreichbarkeit überprüft, so dass das Objekt sofort freigegeben werden kann, wenn es nicht mehr erreichbar ist. Wenn das Objekt zu alt ist, oder den Status als ältestes Objekt im Cache hat und der Platz für neuere Objekte benötigt wird, wird es in den Hauptspeicher verdrängt.

2.3.5 Zusammenfassen von Speicherzugriffen

Eine andere Variante zur Beschleunigung, die ohne Cache auskommt, nutzt REALjava [TSP10]. Dort werden mehrere Speicherzugriffe zu einer sogenannten Supersequence zusammengefasst, um die Häufigkeit von Speicheranfragen zu reduzieren. Allerdings ist dort die Architektur eine andere als beim SHAP. Nur einfache Java-Bytecodes werden von REALjava-Coprozessoren erledigt. Komplexere Bytecodes, Speicherzugriffe und weiteres werden von einer Standard-CPU abgearbeitet. Da die Speicherzugriffe über die CPU geführt werden, ist die Latenz sehr hoch und die CPU wird jedes Mal durch einen Interrupt in ihrer momentanen Arbeit unterbrochen. Durch das Zusammenfassen von Speicherzugriffen wird versucht, die durchschnittliche Latenz zu senken und die Gesamtperformance der CPU zu erhöhen. Dazu ist aber eine aufwändige Umsortierung der Bytecode-Folge mit einem eigenen Recompiler notwendig, der die bestehenden Bytecodes entsprechend umsortiert und verändert.

2.3.6 Methoden-Cache

Ein lokaler Methoden-Cache ist eine weitverbreitete Technik [SP05, HCT10, Sch04] und wird auch im SHAP genutzt [PZS07]. Da es für die Geschwindigkeit des Prozessors wichtig ist, eine möglichst geringe Latenz zu haben, wenn man einen neuen Bytecode ausführen will, wird der aktuelle Programm-Code möglichst nah am Kern gehalten. Um dieses Ziel zu erreichen, wird beim Betreten einer Methode (`invoke`) diese komplett in den Cache geladen. Auch wird versucht die aufrufende Methode im Cache zu behalten, da diese ja weiterhin gebraucht wird, wenn die aktuelle aufgerufene Methode beendet wird (`return`).

Im Jamuth-Prozessor [UW07] gibt es neben einem Befehls-Cache einen speziellen Scratch-RAM, der komplette Methoden dauerhaft vorrätig hält, die zeitkritisch sind oder sehr häufig benötigt werden.

2.4 Die SHAP-Mehrkernarchitektur

Die Secure Hardware Agent Platform (SHAP) ist eine, an der Professur für VLSI-Entwurfssysteme, Diagnostik und Architektur der Fakultät Informatik der Technischen Universität Dresden entwickelte, konfigurierbare Prozessorarchitektur, die Java-Bytecode nativ ausführen kann [PZR07]. Sie besteht aus einer frei wählbaren Anzahl Rechenkerne, die über je einen Daten-Port und einen mit einem Methoden-Cache versehenen Befehls-Port an eine gemeinsame Memory Management Unit angebunden sind [ZRS08]. Über diese greifen sie auf einen gemeinsamen Hauptspeicher zu (Unified Memory Architecture, siehe Abbildung 2.8).

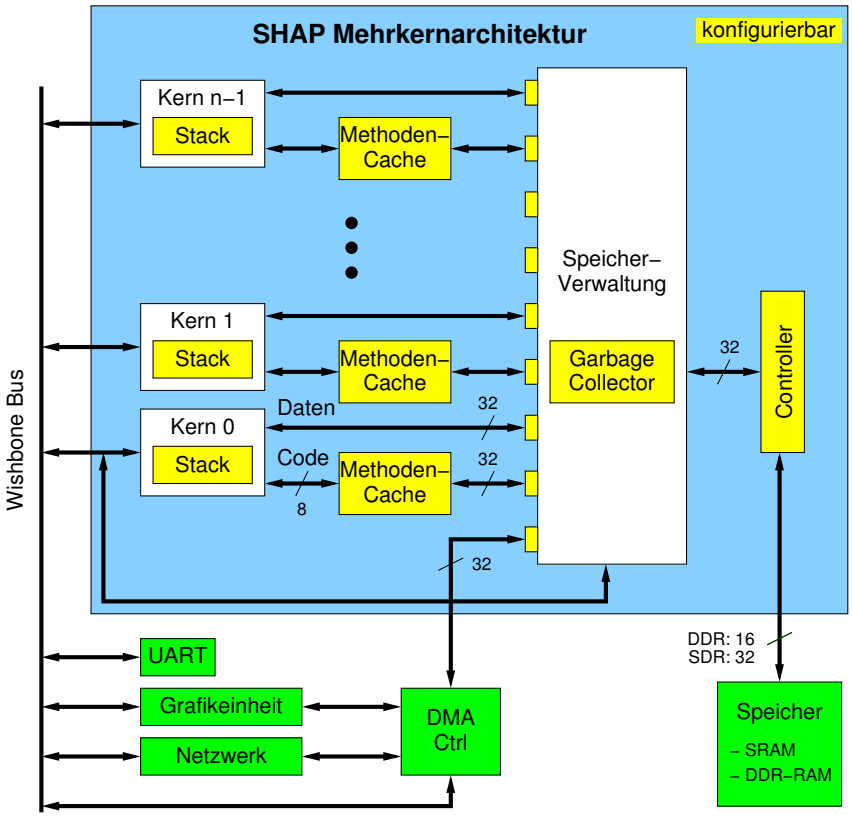


Abbildung 2.8: Die SHAP-Mehrkernarchitektur [ZS09]

Die MMU ist das Herzstück des Mehrkernprozessors und verwaltet den Speicher auf Objektbasis. Mit einem nebenläufig arbeitenden Garbage Collector [Rei07, Rei08] wird der Speicherplatz nicht mehr benötigter Objekte wieder freigegeben. In der MMU wird demnach auch die virtuelle Objektadresse der Rechenkerne in die physikalische Adresse des Hauptspeichers umgerechnet. Dazu verwaltet diese im Hauptspeicher eine Tabelle, in der die Größe, die Basisadresse und der Bias-Wert eines jeden Objektes gespeichert sind. Die Adressierung innerhalb eines Objektes erfolgt über einen Offset-Wert. SHAP verwendet sowohl positive als auch negative Offsets. Die negativen Offsets enthalten Referenzen auf andere Objekte, die positiven enthalten primitive Werte (Integer, Character, Float, etc.). Mit Hilfe des Bias-Wertes erkennt die MMU wo der „Nullpunkt“ innerhalb des Objektes ist. Anhand eines negativen Offsets erkennt der Garbage Collector auch, dass es sich um eine Referenz handelt und kann damit seinen Referenzierungsgraphen aufbauen, um zu entscheiden, welche Objekte noch erreichbar sind. Die nicht mehr erreichbaren Objekte werden dabei gelöscht.

Da der GC nach dem Löschen nicht mehr erreichbarer Objekte den restlichen Speicher kompaktiert, indem er die Objekte zusammenschiebt, kann sich auch die Basisadresse der Objekte ändern. Diese Änderung wird aber vor den Rechenkernen verborgen gehalten und nur in der MMU behandelt. Die Referenz auf ein Objekt bleibt immer die gleiche. Man spricht vom sogenannten Indirektionsprinzip (siehe Abbildung 2.1 auf Seite 3).

Das Zwischenspeichern von Objekten in SHAP wurde bereits in einer anderen Arbeit ana-

lysiert [Ale09]. Der Schwerpunkt lag damals aber auf einer Beschleunigung des Einkernprozessors. Im Ergebnis der Arbeit war aber nicht eindeutig zu klären, ob die zu erwartenden Leistungsgewinne eine Implementierung eines Caches rechtfertigen. Diese und andere Arbeiten [SPH09, HPS10, Sch11] zeigen, dass durchaus schon bei geringen Cache-Größen moderate Leistungsgewinne zu erreichen sind.

3 Analyse des bestehenden Systems

3.1 Analyse der Zugriffe

3.1.1 Voraussetzungen

Um zu analysieren, wie man die benötigte Speicherbandbreite durch den Daten-Port reduzieren kann, wurden verschiedene Benchmarks auf dem SHAP als Einkernprozessor ausgeführt. Dabei wurde protokolliert, welche Datenaufrufe abgesetzt werden und wie sich diese im Einzelnen gliedern. Dazu wurde die SHAP-eigene Trace-Architektur [Ale10] genutzt, um über die Gigabit-Ethernet-Schnittstelle Aktivitäten am Daten-Port mitschreiben zu können. Ausgewertet wurde, welche Bytecodes wann, in welchen Objekten und von welchen Offsets Daten laden oder speichern.

Die Benchmarks stammen aus drei Benchmark-Suiten. Zum einen aus der JEM-Suite [JEM12, SPU10], die speziell für den eingebetteten Bereich erarbeitet wurde. Des Weiteren ausgewählte Benchmarks aus der JGF-Suite [JGF12, BSW+99], die zum Leistungstest und zur Optimierung von JVMs entwickelt wurde. Und zu guter Letzt ein FScript-Test [FSc12], der ein Testprogramm einer eigenen Script-Sprache interpretiert und ausführt. Details zu den Benchmarks sind in Tabelle 3.1 aufgelistet.

Suite	Benchmark	Art der Anwendung
	FScript-Test	Einfache Script-Sprache. Interpretiert Strings.
JEM	AES	AES-Verschlüsselung in vier Threads.
	Bubblesort	Sortiert Daten.
	Kfl	Steueralgorithmus zur Be- und Entladung von Eisenbahnwagen.
	Lift	Steueralgorithmus zur Automatisierung in Fabriken.
	Matrixmul	Multipliziert Matrizen.
	NQueens	Platziert n Damen kollisionsfrei auf einem Schachfeld.
	Sieve	Sieb des Eratosthenes zur Primzahlberechnung.
	UdpIp	Ein UDP/IP-Stack für industrielle Anwendungen.
JGF	Crypt	IDEA Verschlüsselung.
	Heapsort	Ein weiterer Sortieralgorithmus.
	Sparsematmult	Multiplikation dünnbesetzter Matrizen.

Tabelle 3.1: Verwendete Benchmarks

3.1.2 Zugriffsarten

Die durchgeführte Analyse zeigt, dass die Auslastung des Speicher-Busses je nach Anwendung sehr stark variieren kann. Während zum Beispiel die Anwendung Crypt aus der JGF-Suite kaum Bandbreite für die Daten braucht, dafür aber relativ viel für die MMU und den GC, benötigt die Matrixmultiplikation der JEM-Suite eine sehr hohe Bandbreite für die Daten, während der Ausführung aber kaum Bandbreite für die MMU und den Methoden-Cache (Abbildung 3.1). Im Durchschnitt kann man sagen, dass zwei Drittel der benötigten Bandbreite vom Daten-Port genutzt werden, aber nur ein gutes Viertel für den Methoden-Cache und weniger als zehn Prozent für den Garbage Collector und die MMU. Diese Werte variieren aber je nach Benchmark extrem. Diese Arbeit beschränkt sich auf den großen Anteil des Daten-Ports. Da der Crypt-Benchmark diesen fast gar nicht benutzt, wird er im Folgenden nicht weiter betrachtet.

Die Anfragen über den Daten-Port unterteilen sich in drei verschiedene Arten. Die drei Bereiche sind das Aktivieren der Referenz, also das Übersetzen der virtuellen in die physische Adresse durch das Laden der Basis- und Bias-Werte aus der Referenztabelle. Dazu kommen dann noch sowohl das Laden als auch das Speichern der eigentlichen Daten durch einen entsprechenden Offset-Zugriff. Die in Abbildung 3.2 dargestellten Messergebnisse zeigen, dass etwa 40% der Zugriffe des Daten-Ports auf die Referenztabelle erfolgen. Weitere 50% sind lesende Offset-Zugriffe und weniger als 10% sind speichernde Offset-Zugriffe. Es lohnt sich also, sowohl Referenzzugriffe als auch lesende Offset-Zugriffe für ein eventuelles Zwischenspeichern durch einen Cache näher zu untersuchen.

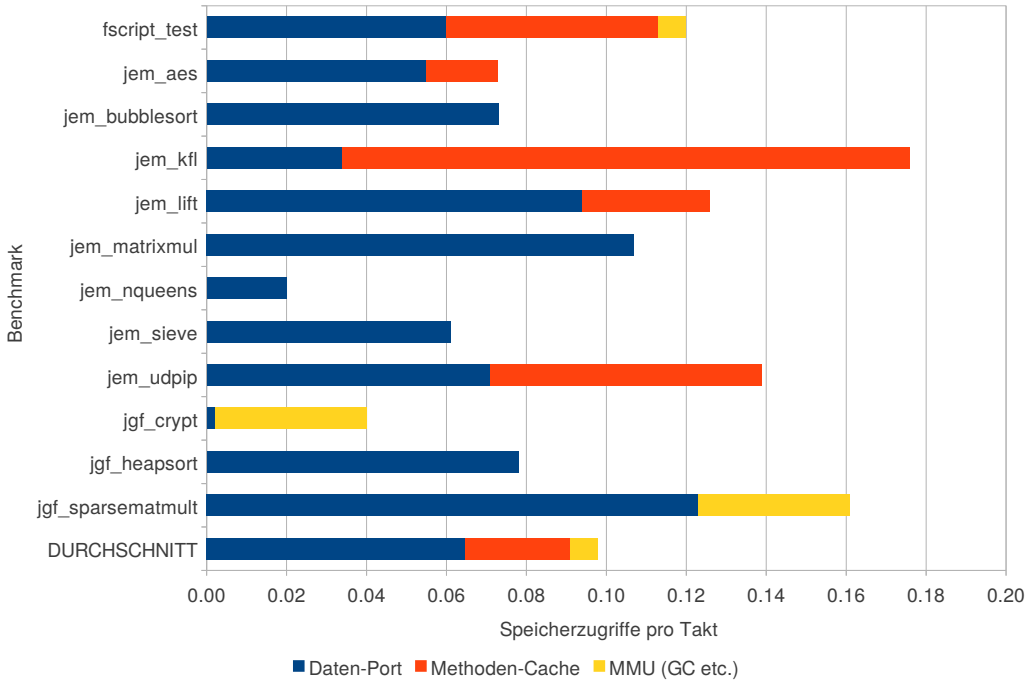


Abbildung 3.1: Zugriffe auf den Speicher pro Taktzyklus

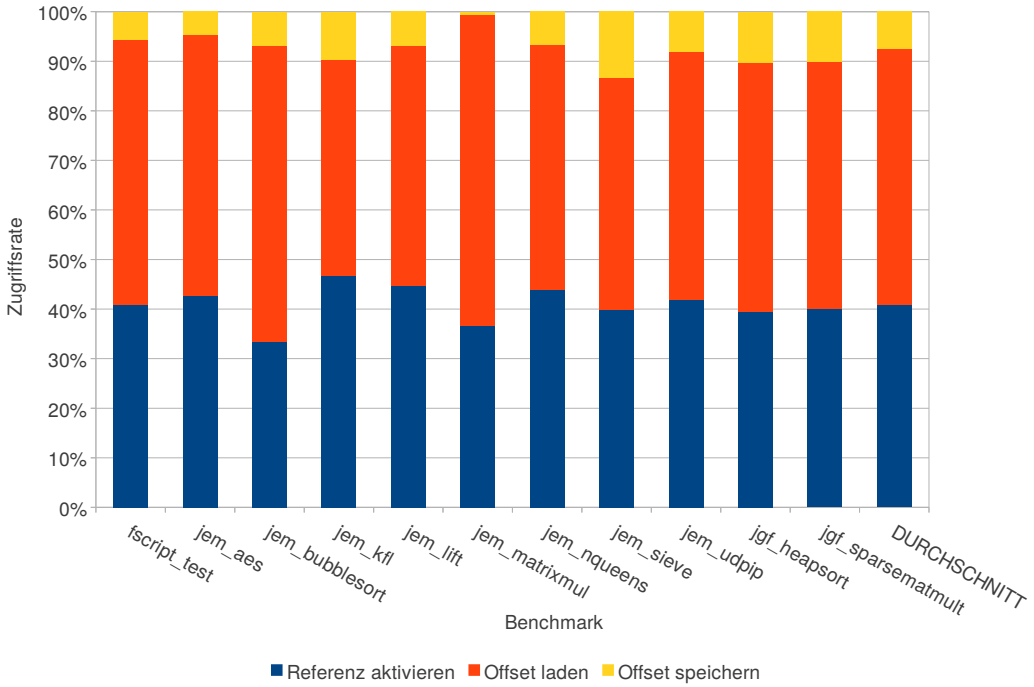


Abbildung 3.2: Zugriffsverteilung am Daten-Port

3.1.3 Herkunft der lesenden Offset-Zugriffe

Weiterhin wurde analysiert, auf welche Bereiche innerhalb eines Objektes am häufigsten zugegriffen wurde. Die Offsets 0 und -1 sind in jedem Objekt zu finden. Offset 0 enthält Werte, die zum Beispiel für das zur Synchronisation benötigte Sperren von Objekten gebraucht werden. Im Offset -1 steht hingegen die Referenz auf die eigene Klasse, zu der das Objekt gehört. Dieser Wert wird zum Beispiel für Methodenaufrufe benötigt. Die Offsets -2 und 1 sind die ersten Offsets, die in jedem Objekt nach Bedarf vorhanden sind und Nutzdaten enthalten. Die Statistik zeigt auch, dass diese beiden Offsets diejenigen sind, die am häufigsten gelesen werden (Abbildung 3.3). Je nach Objekt gibt es aber auch andere Offsets, die sehr häufig aufgerufen werden. Im Benchmark NQueens ist das zum Beispiel der Offset 137 (Tabelle 3.2).

Ein weiterer interessanter Punkt ist, welche Bytecodes die meisten Speicherzugriffe verursachen. Das sind in vielen Fällen Zugriffe auf Arrays (*aload und *astore in Tabelle 3.3). Die Ursache liegt unter anderem auch darin begründet, dass diese zwei Speicherzugriffe benötigen, einen um die Array-Länge zu lesen und einen um den eigentlichen Datenwert zu lesen oder zu schreiben. Das Lesen der Array-Länge ist nötig, um festzustellen, ob der Index des Zugriffes auch innerhalb des Arrays liegt. Ist das nicht der Fall wird eine Exception geworfen. Die Array-Länge eines Objektes ist immer auf dem Offset 1 gespeichert. Das ist auch eine der Ursachen, warum dieser Offset am häufigsten aufgerufen wird.

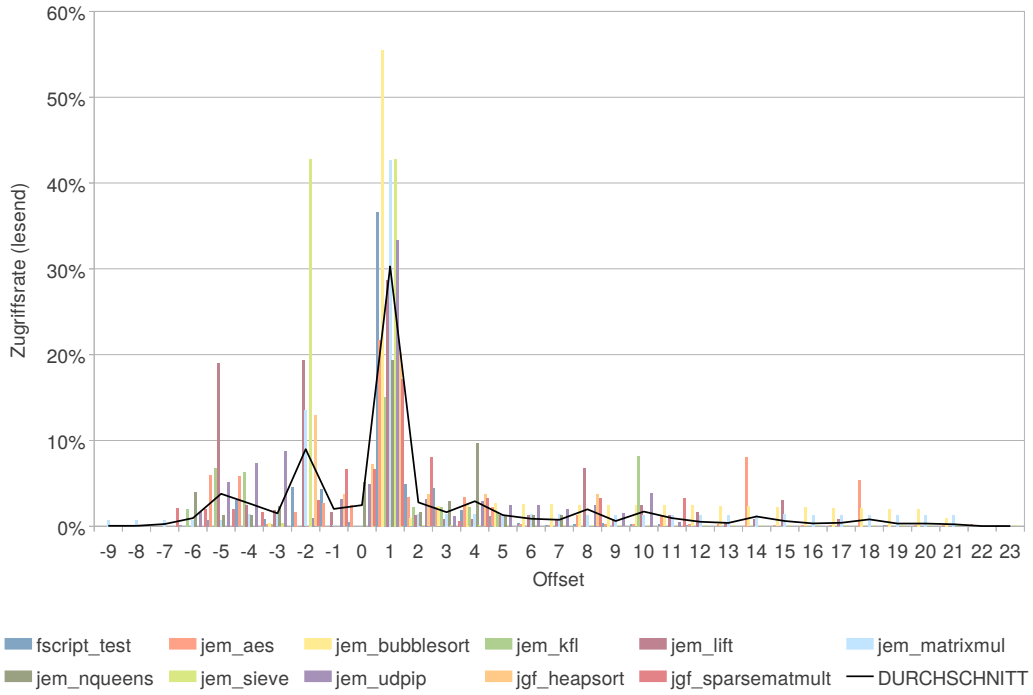


Abbildung 3.3: Verteilung lesender Zugriffe auf Offsets

	Offset #1		Offset #2		Offset #3	
FScript-Test	1	34,25%	2	5,16%	3	5,16%
JEM AES	1	21,05%	14	7,42%	-5	5,47%
JEM Bubblesort	1	49,81%	4	3,36%	5	3,27%
JEM Kfl	1	12,33%	10	6,75%	-5	5,56%
JEM Lift	1	25,50%	-2	17,00%	-5	16,62%
JEM Matrixmul	1	42,31%	-2	13,39%	-3	1,61%
JEM NQueens	137	40,40%	1	17,11%	4	11,88%
JEM Sieve	-2	33,23%	1	33,23%	53	0,54%
JEM UdpIp	1	29,59%	-3	7,60%	0	6,35%
JGF Heapsort	1	19,84%	-2	10,70%	0	9,12%
JGF SparseMatmult	1	16,94%	2	9,38%	0	8,22%

Tabelle 3.2: Häufig genutzte Offsets

	Verursacher #1		Verursacher #2		Verursacher #3	
FScript-Test	*aload	53,78%	getfield	19,93%	inv.virtual	5,18%
JEM AES	*aload	30,75%	inv.special	14,71%	getstatic	10,74%
JEM Bubblesort	*aload	79,25%	*astore	20,38%	io_write	0,33%
JEM Kfl	getstatic	46,77%	inv.static	16,03%	*aload	13,49%
JEM Lift	getfield	44,71%	*astore	20,71%	*aload	17,00%
JEM Matrixmul	*aload	82,84%	getfield	14,85%	*astore	1,55%
JEM NQueens	inv.static	40,41%	*aload	21,94%	*astore	12,23%
JEM Sieve	*astore	44,75%	getfield	33,23%	*aload	21,71%
JEM UdpIp	*aload	36,78%	getstatic	14,16%	*astore	13,95%
JGF Heapsort	getfield	13,74%	*aload	12,22%	getstatic2	12,15%
JGF SparseMatmult	inv.virtual	16,43%	getstatic2	10,96%	*aload	10,57%

Tabelle 3.3: Bytecodes, die die meisten Zugriffe verursachen

3.2 Einsatzmöglichkeiten für einen Objekt-Cache

3.2.1 Ort des Caches

Es gibt nun verschiedene Möglichkeiten, einen Cache zu implementieren. So ist zuerst zu klären, wo der Cache einzubauen ist. Es gibt die Möglichkeit, einen großen Cache für alle Kerne zu nutzen oder man baut für jeden Kern einen eigenen Cache. Ziel ist bekanntlich, die benötigte Speicherbandbreite eines Kerns zu senken. Würde man jetzt einen Cache für alle Kerne implementieren, würde sich das Problem des Flaschenhalses nur vom Hauptspeicher hin zum Cache verlagern. Um eine effektive und nachhaltige Entlastung des Zugangs zum externen Speicher zu erreichen, bietet sich demnach nur die Variante mit einem eigenen Cache für jeden Kern an.

3.2.2 Größe der Caches

Bei einem Cache je Kern sollte dieser möglichst klein sein, da er dann vielfacher Ausführung vorkommt. Auch nehmen die bereits bestehenden Speicher im SHAP schon sehr viel Platz ein [Olu09]. Es ist also zu klären mit wie wenigen Cache-Zeilen man bereits ein gutes Ergebnis erzielen kann. In Abbildung 3.4 ist zu sehen, dass schon mit vier bis acht zwischengespeicherten Objekten eine hohe Sättigung erreicht werden kann. Die sechs am häufigsten genutzten Objekte verursachen über 80% der Zugriffe. Die restlichen 20% verteilen sich dafür über sehr viele Objekte. Bei den Zugriffen handelt es sich nicht um räumliche sondern um zeitliche Lokalität, die hier ausgenutzt werden kann. Da die Objekte nicht zwangsweise nebeneinander liegen, ist ein direkt abbildender Cache hier nicht geeignet, denn dieser ist eher für räumliche Lokalität geeignet, d.h. nebeneinander liegende Adressen. Empfehlenswert ist also ein kleiner Cache mit hoher Assoziativität, was auch von anderen Arbeiten bestätigt wird [Ale09, SPH09, HPS10, Sch11].

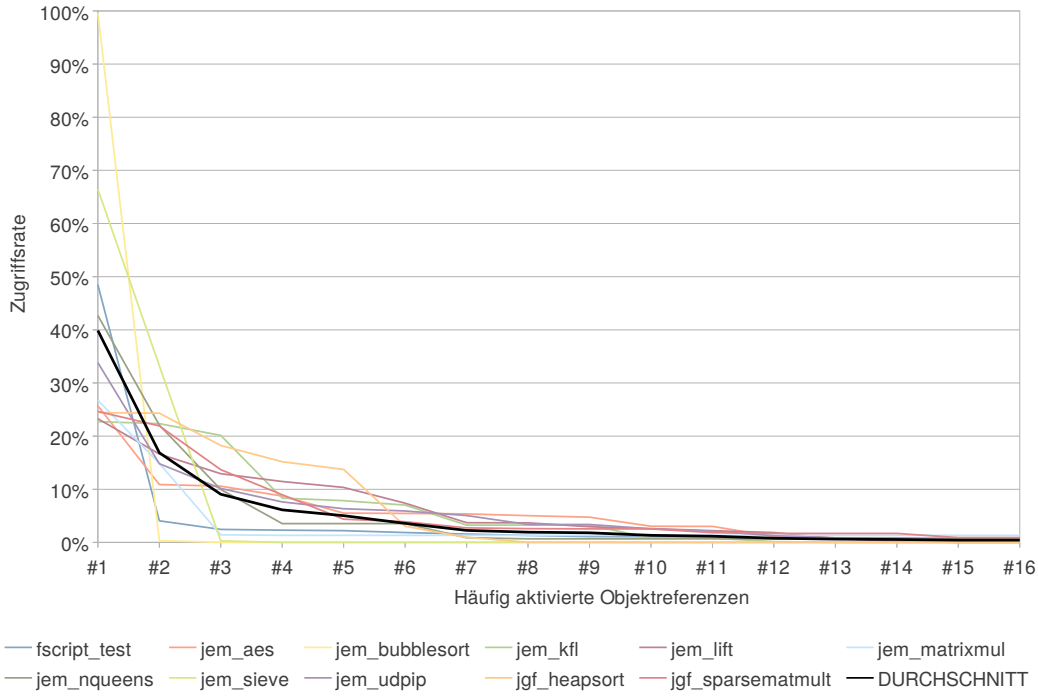


Abbildung 3.4: Zugriffsverteilung auf häufig benutzte Objekte

3.2.3 Inhalt der Caches

Im nächsten Schritt stellt sich die Frage, was eigentlich im Cache gehalten werde soll. Werden nur konstante Werte im Cache gehalten, benötigt man keine Synchronisationsmechanismen zwischen den Caches der einzelnen Kerne, um Kohärenz zu gewährleisten. Dafür ist der Umfang der speicherbaren Werte sehr gering. Zur Auswahl stehen im Wesentlich die Referenz auf die eigene Klasse (Offset -1) und die Länge aller Array-Objekte (Offset 1). Offset -1 wird zu selten aufgerufen, der nutzbare Effekt wäre zu gering (Abbildung 3.5). Die einzig wirklich nutzbringende Variante wäre das Zwischenspeichern der Array-Länge, da Array-Aufrufe sehr häufig stattfinden.

Nimmt man den Mehraufwand durch die Kohärenzlogik in Kauf, steht die Frage, welche Offsets, beziehungsweise Offset-Bereiche, im Cache zu halten sind. Die Auswertung zeigt, dass die Zugriffe auf die Offsets -1 bis 0 relativ gering ausfallen (Abbildung 3.5). Nimmt man dann aber den Bereich -2 bis 1 steigt die Zugriffshäufigkeit enorm. Bei einer weiteren Ausdehnung des Offset-Bereiches auf -3 bis 2 oder -4 bis 3 steigt die Häufigkeit aber nur noch in geringem Maße. Das Zwischenspeichern des Offset-Bereiches -2 bis 1 bzw. nur der Offsets -2 und 1 würde also den höchsten Kosten-Nutzen-Faktor haben. Um einen besseren Einblick zu bekommen, welche Lösung die beste Leistung bringt, sind detailliertere Simulationen notwendig.

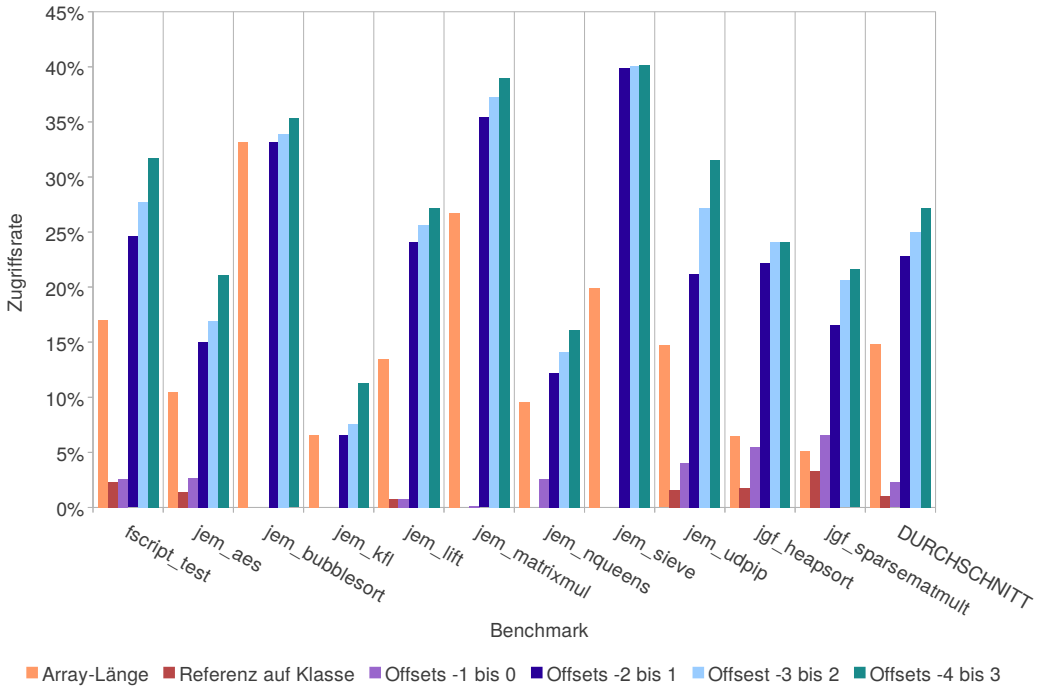


Abbildung 3.5: Vergleich der Zugriffsraten zwischen Konstanten (rot) und Offset-Bereichen (blau)

3.3 Simulation

3.3.1 Voraussetzungen

Aus den obigen Erläuterungen ergibt sich, dass ein kleiner vollassoziativer Cache zu simulieren ist. Zu diesem Zweck wurde ein kleines Simulations-Script geschrieben, dass für verschiedene Cache-Inhalte Hit-Rate (dt. Trefferrate), eingesparte Speicherzugriffe und Speicherzugriffe pro Takt für Cache-Größen von ein bis 16 Cache-Zeilen ausgibt. Zur besseren Auswertung sind die Ergebnisse graphisch dargestellt.

Das Script wurde in mehreren Varianten für verschiedene Caches entwickelt:

- **Array-Längen-Cache**: Es werden nur Array-Längen zwischengespeichert. Diese sind konstant, so dass keine Rücksicht auf Kohärenz genommen werden muss.
- **Offset-Cache**: Es werden die Daten ausgewählter Offsets zwischengespeichert. Hier muss die Kohärenz beachtet werden, indem bei Auftreten des Bytecodes `monitorenter` der Cache geleert wird. Da mit `volatile` markierte Felder in den Trace-Daten nicht mehr zu erkennen sind, konnten diese für die Kohärenz nicht mit berücksichtigt werden.
- **Adress-Cache**: Es werden nur die physikalischen Adressen zu jeder Objekt-Referenz zwischengespeichert. Die Funktionalität entspricht der eines Translation Lookaside Buffers (TLB).

- **TLB inkl. Offset-Cache**: Entspricht vom Aufbau einem TLB, speichert aber außer der physikalischen Adresse auch ausgewählte Offsets in der zugehörigen Cache-Zeile.

3.3.2 Array-Längen-Cache

Wie schon festgestellt, wäre die Array-Länge eine günstige Möglichkeit, wenn man nur konstante Werte zwischenspeichern will. In Abbildung 3.6 ist dargestellt, wie hoch die Trefferrate in Abhängigkeit der Anzahl der Cache-Zeilen wäre, wenn man nur die Array-Länge im Cache halten würde. Es ist gut zu sehen, dass ab sechs Cache-Einträgen kein weiterer Zugewinn zu erwarten ist. Interessanter als die Hit-Rate ist aber, wie viel Prozent der Speicherzugriffe am Ende einspart werden. Abbildung 3.7 zeigt, dass beim Bubblesort-Benchmark ein Drittel eingespart werden kann. Dass heißt ein Drittel aller Speicheranfragen sind Zugriffe auf die Array-Länge. Da zu jedem Zugriff auch das Laden der Referenz und der eigentliche Zugriff auf das Array-Element gehören, scheint der komplette Algorithmus nur auf einem einzigen Array zu rechnen, was sich auch durch den Quelltext bestätigt.

Entsprechend dem Ziel der Arbeit interessiert aber in erster Linie, welche Zugriffsrate nach dem Cache noch bestehen bleibt. Das ist für das Zwischenspeichern der Array-Länge in Abbildung 3.8 zu sehen. Das Ergebnis fällt relativ ernüchternd aus. Die maximale Zugriffsrate sinkt zwar von knapp 0.11 Zugriffen pro Takt auf reichlich 0.9, viele Benchmarks haben aber nur wenige Vorteile von diesem Cache.

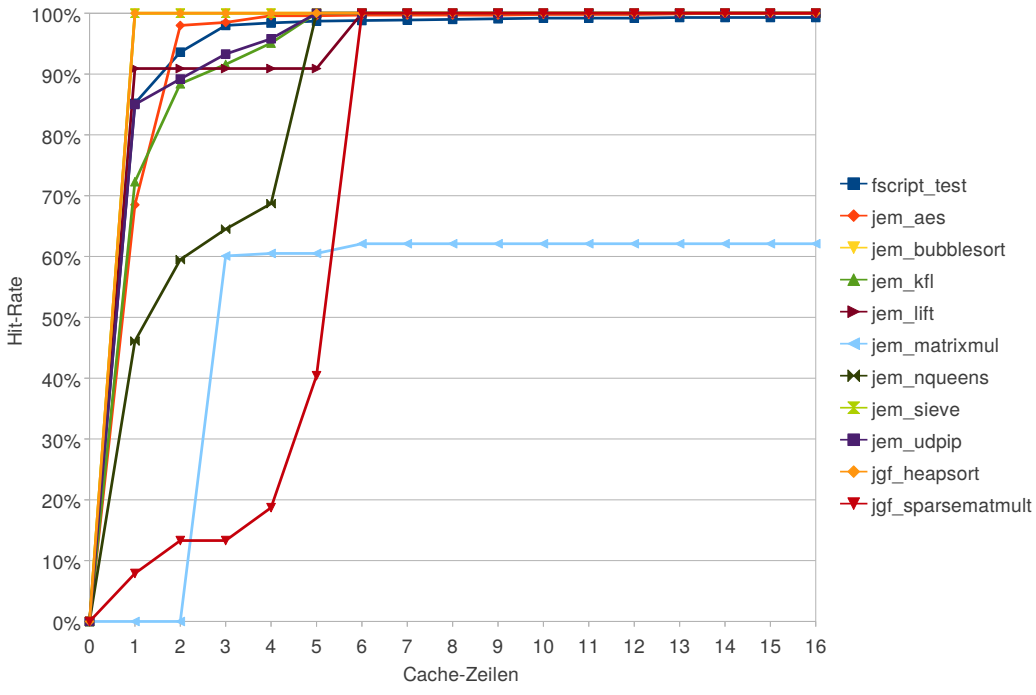


Abbildung 3.6: Hit-Rate eines Array-Längen-Caches

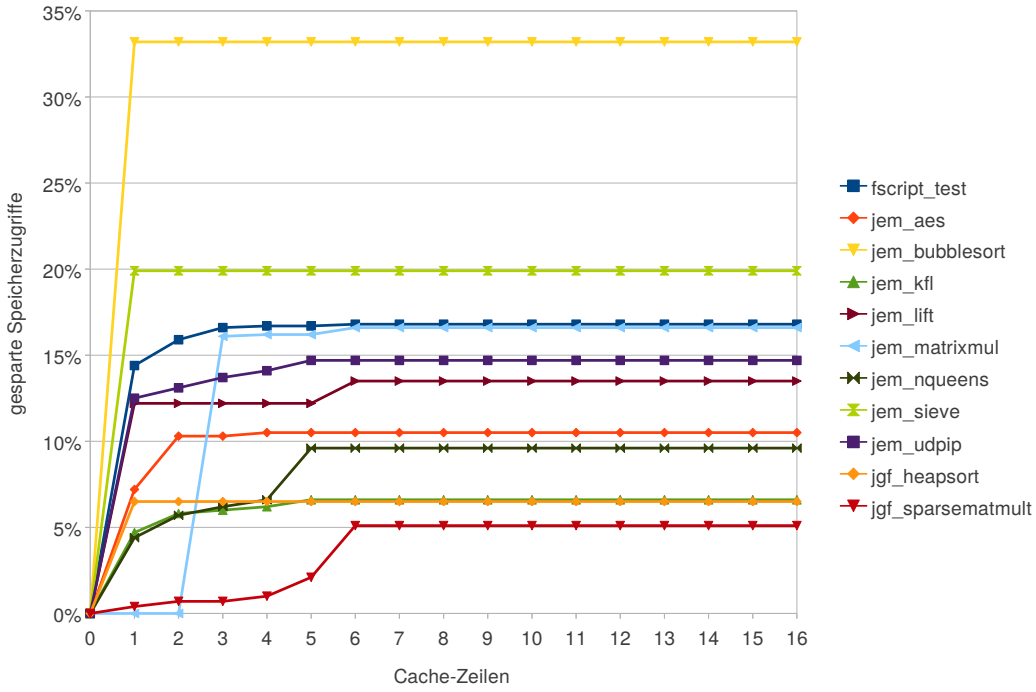


Abbildung 3.7: Eingesparte Speicherzugriffe durch einen Array-Längen-Cache

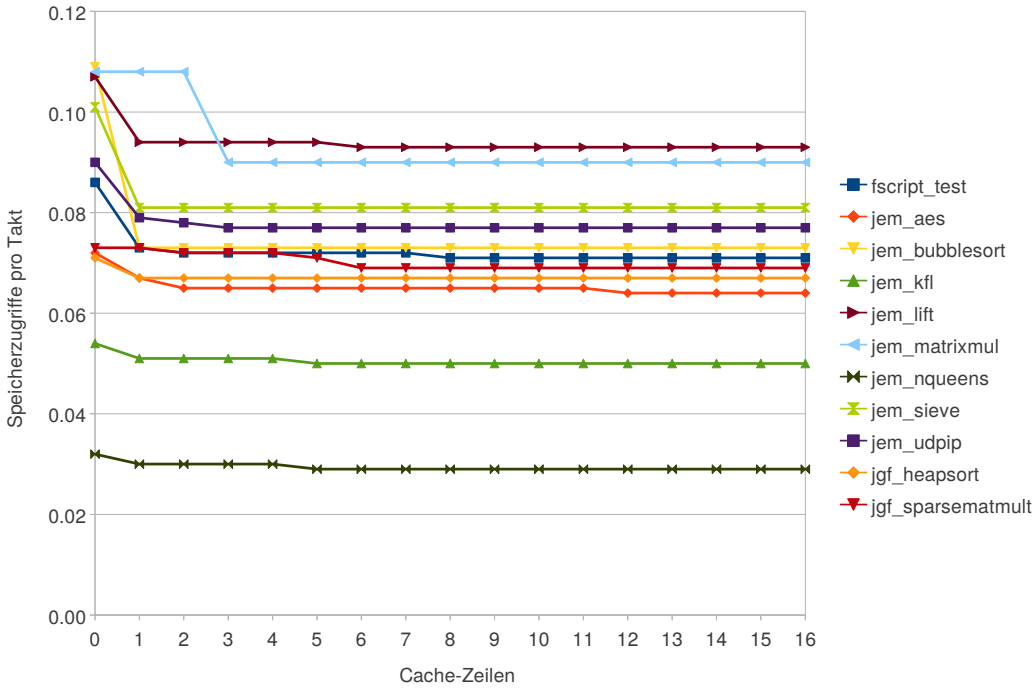


Abbildung 3.8: Speicherzugriffe pro Takt, reduziert durch einen Array-Längen-Cache

RAM	CPU-Takt zu RAM-Takt	Zugriffsdauer in RAM-Takten	Wörter pro Zugriff	Wörter pro CPU-Takt
SRAM	1 : 1	1	1	1
DDR-RAM	1 : 2	16	8	1
Burst-Länge 4	1 : 1			0,5
DDR-RAM	1 : 2	18	16	1,8
Burst-Länge 8	1 : 1			0,9

Tabelle 3.4: Vergleich verschiedener externer Speicher

3.3.3 Offset-Cache

3.3.3.1 DDR-RAM-Anbindung

Da das Kohärenzproblem zwischen den Caches in Java relativ einfach zu beheben ist (Abschnitt 2.2.8), lohnt sich ein Blick auf einen Cache, der mehrere einzelne Offsets oder zusammenhängende Offset-Bereiche zwischenspeichert. Für größere Offset-Bereiche wäre eine DDR-RAM-Anbindung als externer Speicher von Vorteil, da so mit einem Zugriff die komplette Cache-Zeile gefüllt werden kann. Für den SHAP ist eine solche Anbindung mit einer Datenbusbreite von 64 Bit bei einer Burst-Länge von vier oder acht Datensätzen geplant. Nachteilig ist jedoch, dass DDR-RAM hohe Latenzen und Zykluszeiten hat. So bekommt man in 16[1] Takten acht 32-Bit-Wörter bei einer Burst-Länge von 4, bzw. in 18 Takten 16 Wörter bei einer Burst-Länge von 8. Die Zykluszeiten beziehen sich darauf, dass bei jedem Zugriff die komplette Zeile neu angewählt werden muss. Da die Objekte im Speicher weit verteilt liegen können, treffen die hohen Zykluszeiten in vielen Fällen zu.

Die jetzige Speicheranbindung nutzt einen SRAM, der Anfragen innerhalb eines Taktes beantworten kann. Man bekommt also in einem Takt ein 32-Bit-Wort. Aus Tabelle 3.4 ist ersichtlich, dass man mit DDR-RAM nur mit einer Burst-Länge von 8 und einer doppelten RAM-Taktrate gegenüber dem CPU-Takt schneller Daten laden kann als mit SRAM. Das ist aber auch nur dann der Fall, wenn alle Datenwörter eines Bursts wirklich benötigt werden.

Um verschiedene Cache-Konfigurationen vergleichen zu können, sind diese in einem gemeinsamen Diagramm in Abbildung 3.9 dargestellt. Für jede Cache-Größe wurden die Simulationsergebnisse für alle Benchmarks je Konfiguration in einem farbigen Balken zusammengefasst. Dieser Balken gibt die Spanne vom niedrigsten bis zum höchsten Messwert an. Die Box über dem Balken umfasst den Bereich der Standardabweichung σ um den Erwartungswert μ, also $\mu \pm \sigma$, was in etwa zwei Drittel der Messwerte umfasst. Der Erwartungswert μ berechnet sich hier als arithmetisches Mittel über die Werte aller Benchmarks. Ziel ist es also, sowohl das obere Ende des farbigen Balkens als auch die dazu gehörige Box möglichst weit nach unten zu bekommen. Die detaillierten Ergebnisse zu jeder Simulation sind im Anhang A zu finden.

Für eine DDR-RAM-Anbindung sind im Diagramm zwei Konfigurationen über verschiedene

[1] Die Zykluszeiten wurden mir auf Nachfrage von Dr. Martin Zabel genannt

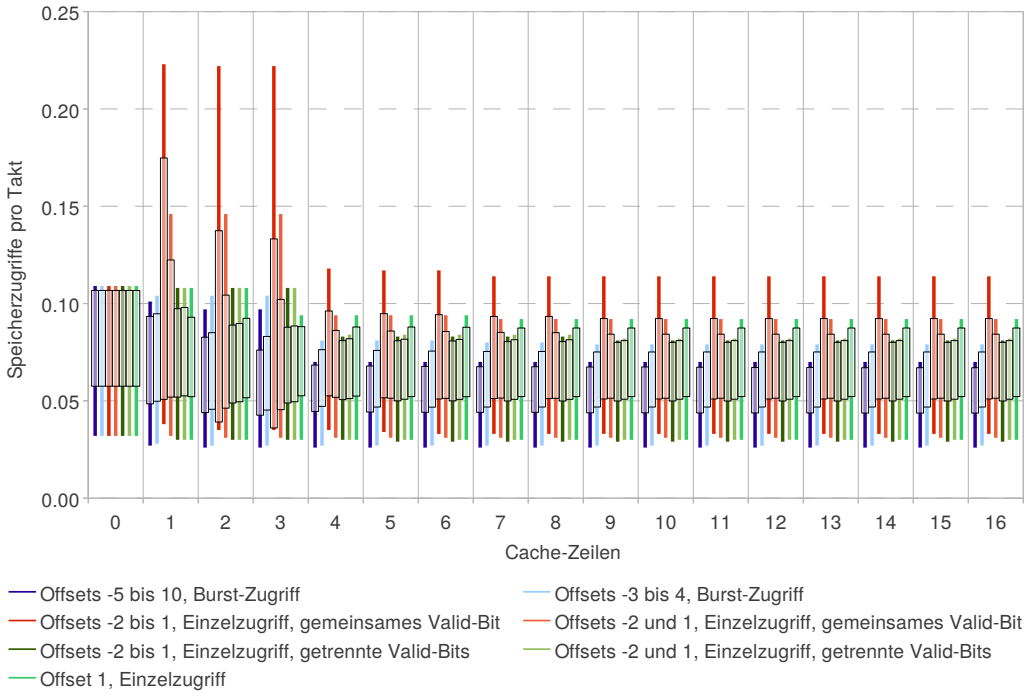


Abbildung 3.9: Speicherzugriffe pro Takt für verschiedene Cache-Konfigurationen

Offset-Bereiche blau dargestellt, einmal über 16 Wörter und einmal über acht Wörter. Diese Ergebnisse zeigen zwar gute Werte, geben aber nur die Zugriffe pro Takt wieder. Um jetzt zu wissen, wie lange die Speicherverbindung belegt ist, muss dieser Wert mit der Anzahl der CPU-Takte der Zugriffsdauer multipliziert werden. Das ist im besten Fall immer noch ein Wert von 8. Auch geht die Simulation davon aus, dass bei einem Speicherzugriff auf den Offset-Bereich des Caches die Speicheranfrage so umgeformt wird, dass genau der passende Offset-Bereich für den Cache geladen wird.

Da der dafür vorausgesetzte DDR-RAM-Controller aber noch nicht vorhanden ist, fällt diese Lösung sowieso von vornherein für eine Implementierung aus. Deshalb muss die günstigste Lösung für den vorhandenen SRAM-Controller gefunden werden.

3.3.3.2 SRAM-Anbindung

Bei einem Zugriff auf den SRAM liegt die Latenz bei einem Takt. Die Anzahl der Zugriffe pro Takt entspricht also auch der Anzahl der Takte, während der die Speicheranbindung belegt ist. Im Gegensatz zum DDR-RAM wird aber bei jedem Speicherzugriff nur ein Wort geladen. Aus diesem Grund ist für das Füllen einer Cache-Zeile auch für jedes Wort ein neuer Speicherzugriff nötig. In Abbildung 3.9 ist die Simulation für den SRAM in rot und grün dargestellt. Für die beiden Simulationen, einmal der Offset-Bereich -2 bis 1 und einmal nur die Offsets -2 und 1, wird die Cache-Zeile beim ersten lesenden Zugriff komplett gefüllt. Durch dieses Verhalten sind aber unter Umständen mehr Speicherzugriffe nötig, als man am Ende einspart. Genau das ist auch in den Ergebnissen (rot) zu sehen, wo teilweise mehr Speicherzugriffe nötig sind, als ohne Cache.

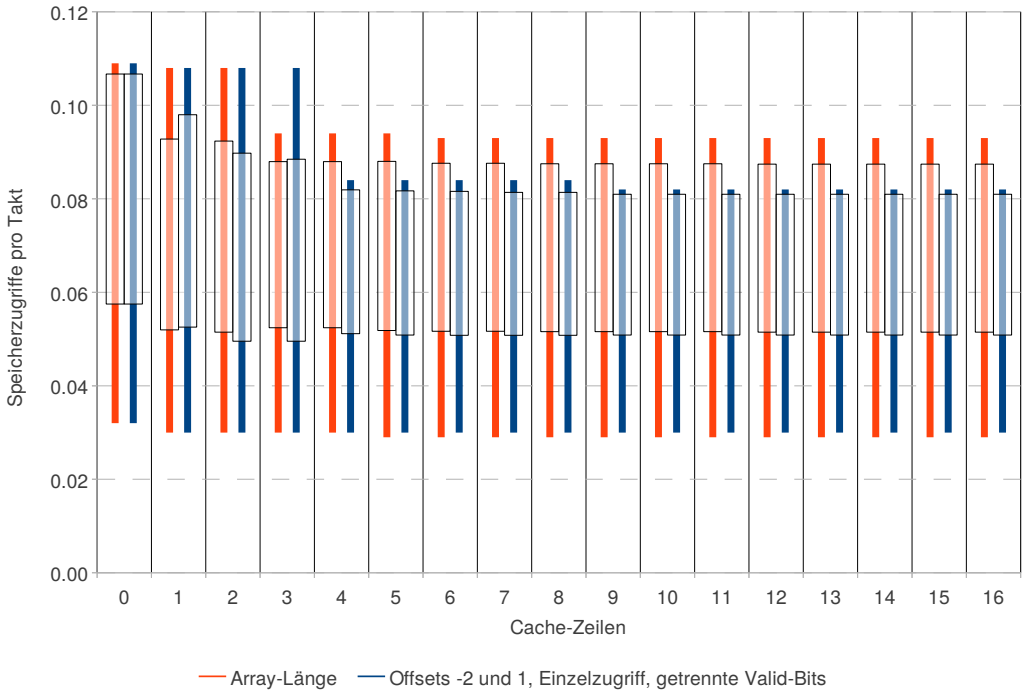


Abbildung 3.10: Vergleich zwischen Array-Längen-Cache und Offset-Cache

Um diese Problem zu umgehen, speichert man nur jedes Offset-Wort einzeln wenn es benötigt wird und markiert mit einem Valid-Bit welches Wort im Cache gültig ist und welches nicht. Die Ergebnisse für diese Lösung sind in der Abbildung 3.9 grün dargestellt. Sowohl die Werte für den Offset-Bereich -2 bis 1 als auch nur die beiden Offsets -2 und 1 zeigen ähnlich gute Werte. Abbildung 3.5 zeigte ja, dass die Erweiterung um weitere Offsets nur wenig Potential bietet. Die Variante, die nur die Offsets -2 und 1 speichert, ist vorzuziehen, da sie fast dieselben Ergebnisse bringt, wie die Variante mit dem kompletten Offset-Bereich, aber nur die Hälfte des Cache-Speichers benötigt. Es werden nur zwei statt vier 32-Bit-Wörter pro Cache-Zeile zwischengespeichert.

In Abbildung 3.10 ist der Vergleich zwischen dem Array-Längen-Cache und dem Offset-Cache mit den Offsets -2 und 1 dargestellt. Der Offset-Cache zeigt gegenüber dem Konstanten-Cache bessere Ergebnisse. Ein Array-Längen-Cache benötigt 14 Bit für den Datenteil pro Cache-Zeile, der Offset-Cache 64 Bit. Der Rest des Caches wäre identisch. Der Offset-Cache ist also teurer, da mehr Speicher benötigt wird, dafür aber flexibler.

3.3.4 Adress-Cache

3.3.4.1 Alleinstehender TLB

Ein weiterer großer Bereich von Speicheranfragen, der sich eignet um durch einen Cache abgefangen zu werden, ist das Laden der physikalischen Adresse einer Objektreferenz. Dazu wird mit Hilfe eines Translation Lookaside Buffers (TLB) die Umsetzung von virtuellen in reale

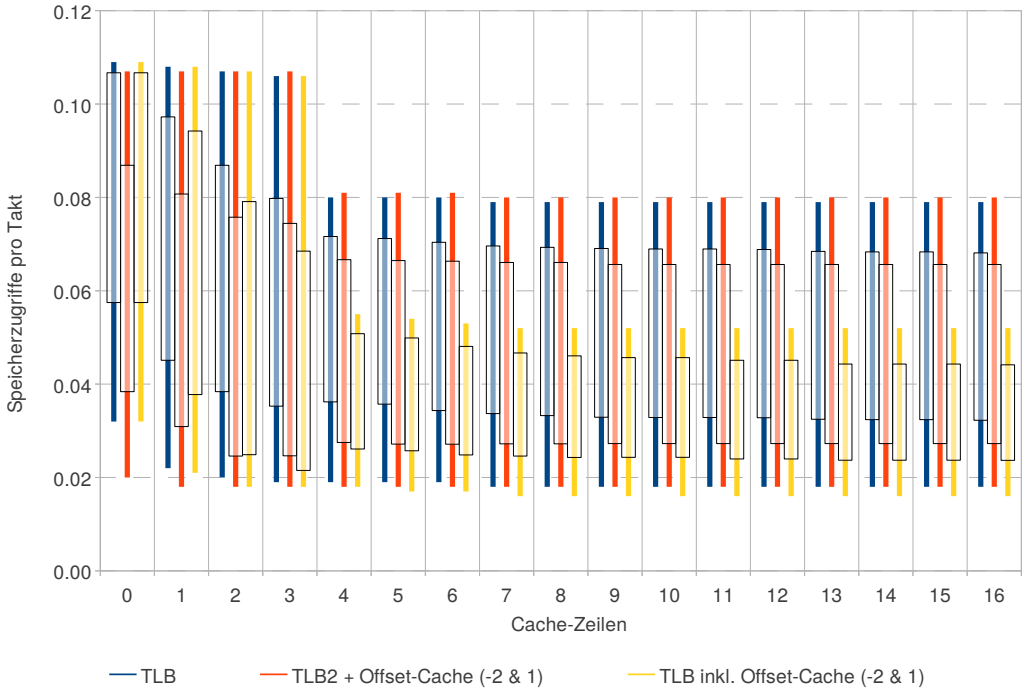


Abbildung 3.11: Vergleich von Lösungen mit einem TLB

Adressen zwischengespeichert. Dieser Cache speichert im SHAP sowohl die Basisadresse als auch den Bias-Wert zu den jeweiligen Objektreferenzen. Die Ergebnisse der Simulation eines reinen TLBs sind in Abbildung 3.11 blau dargestellt. Im bestehenden System gibt es schon einen TLB mit zwei Einträgen. Dieser hat aber auf manche Benchmarks nur wenig Einfluss. Im Diagramm ist zu sehen, dass sich der Maximalwert zwischen einem TLB mit keinen oder nur zwei Einträgen kaum ändert. Erst ab vier Einträgen zeigt ein TLB seine volle Wirkung.

3.3.4.2 TLB in Kombination mit Offset-Cache

Zum Vergleich dazu ist in Rot eine Variante dargestellt, die zu dem bestehenden TLB mit zwei Einträgen den Offset-Cache beinhaltet. Diese Lösung zeigt zwar etwas bessere Werte als ein reiner Offset-Cache, aber nur minimal bessere Werte als ein reiner TLB. Zudem wären für jeden der beiden Caches getrennte Tag-Speicher und Tag-Vergleicher notwendig.

Deswegen gibt es den Ansatz, die Tag-Einheit für beide Cache-Lösungen gemeinsam zu nutzen. Man hat also nur eine Tag-Einheit und einen kombinierten Speicher, der in jeder Cache-Zeile sowohl die Informationen für die physikalische Adresse des Objektes als auch die Offsets enthält. Diese Lösung, im Diagramm gelb dargestellt, zeigt die besten Ergebnisse. Die Anzahl der Speicherzugriffe kann so von durchschnittlich 0.063 Zugriffen bei einem TLB mit zwei Einträgen auf 0.038 Zugriffe bei einem kombinierten Cache mit vier Einträgen gesenkt werden. Im Durchschnitt können so die Speicherzugriffe um ca. 40% reduziert werden.

3.4 Schlussfolgerung

Ein Offset-Cache benötigt zwar mehr Cache-Speicher als ein Array-Längen-Cache, bietet aber das größere Einsparpotential bezüglich der Speicherzugriffe pro Takt. Zudem ist er auch flexibler hinsichtlich Größe und Inhalt der Cache-Zeilen. Aus diesem Grund ist er dem Array-Längen-Cache vorzuziehen.

Im Vergleich zwischen einem Offset-Cache und einem TLB hat eine Kombination von Beiden das beste Kosten-Nutzen-Verhältnis, da nur eine Tag-Einheit benötigt wird, die gemeinsam genutzt wird. Pro Cache-Zeile werden so 64 Bit für die Offsets benötigt, dazu kommen 13 Bit für die Basisadresse und acht Bit für den Bias-Wert. Nur durch das Hinzufügen von 21 Bit pro Cache-Zeile für die physikalische Adresse kann die Effektivität des Offset-Caches noch einmal deutlich gesteigert werden. Im Diagramm (Abbildung 3.11) ist auch zu sehen, dass schon ab vier Cache-Zeilen das Optimum fast erreicht ist.

4 Prototypische Implementation

4.1 Entwurf eines Objekt-Caches

4.1.1 Zielstellung und Modularisierung

Aus Kapitel 3 ergibt sich folgende Zielstellung: Der Entwurf eines kleinen vollassoziativen Caches mit wenigstens vier Cache-Zeilen, der neben den Offsets -2 und 1 auch die Basisadresse und den Bias-Wert zu jedem Objekt speichert. Das entspricht quasi einem TLB mit integriertem Offset-Cache. Das Ganze sollte nach Möglichkeit so flexibel implementiert sein, dass man ohne größeren Aufwand sowohl die Anzahl der Cache-Zeilen (n) als auch die Anzahl und Nummern der Offsets (m) verändern kann. Dabei soll die Taktfrequenz von 80 MHz erhalten werden. Die Implementation erfolgt in der Hardwarebeschreibungssprache VHDL.

Die Adressierung von Datenwörtern ist in zwei nacheinander ablaufende Teile gegliedert, erst das Aktivieren der Referenz und dann das nachfolgende Laden der Offset-Nummer. Es ergibt sich also, dass zwei Vergleicherblöcke notwendig sind, einer für die Referenzen und einer für die Offsets. Der Cache gliedert sich somit in drei Teile. Die Tag-Einheit für das Vergleichen der Referenzen und das Auswählen der Cache-Zeile, der Offset-Teil, der auswertet, ob der angeforderte Offset im Cache liegt bzw. im Cache gespeichert werden kann. Und zu Letzt ist natürlich auch noch der eigentliche Cache-Speicher notwendig.

4.1.2 Tag-Einheit

4.1.2.1 Grundstruktur

Hauptbestandteil der Tag-Einheit (Abbildung 4.1) ist ein Tag-Register für jede der n Cache-Zeilen, jeweils kombiniert mit einem Vergleicher, der feststellt, ob ein neu angeforderter Tag in dieser Cache-Zeile liegt. Ein Tag entspricht hier der Referenz eines Objektes. Die Vergleicher generieren jeweils zwei Signale, das Hit-Signal und ein Index-Signal mit der entsprechenden Bit-Breite, um die Indexnummer der Zeile darstellen zu können. Ist das Ergebnis des Vergleiches positiv, spricht man von einem Cache-Hit, im anderen Fall von einem Cache-Miss. Bei einem Cache-Miss sind das Hit- und das Index-Signal eines Vergleichers 0. Über ODER-Gatter werden die Signale aller Vergleicher zu einem Signal zusammengefasst. Ist das resultierende Hit-Signal 1, ist auch der Index-Wert gültig, ansonsten nicht.

Da der Cache die physischen Adressen der Objekte zwischenspeichert, ist bei jedem Zugriff entweder anzugeben, in welcher Cache-Zeile diese Daten liegen, oder in welche Zeile die neu

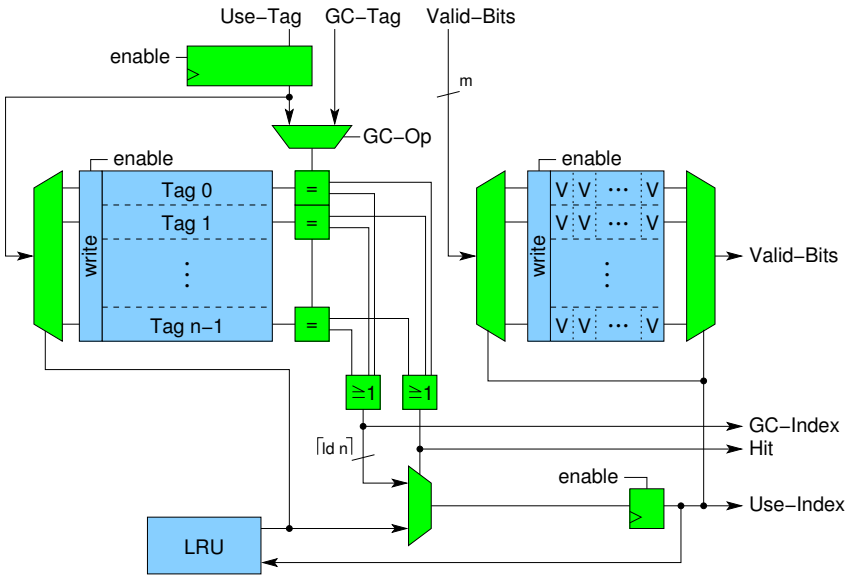
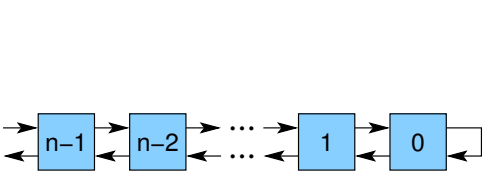


Abbildung 4.1: Schematischer Aufbau der Tag-Einheit

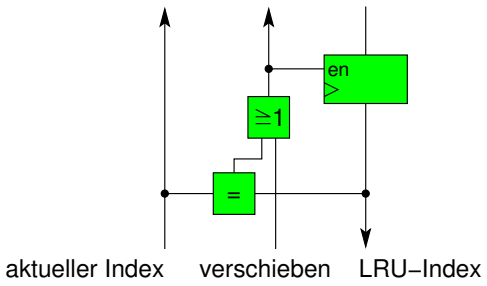
zu lesenden Daten geschrieben werden sollen. Aus diesem Grund muss die Tag-Einheit auch einen Zeilen-Index für die neuen Daten ausgeben, wenn kein Hit festgestellt wurde. Es ist also eine Ersetzungsstrategie zu implementieren, welche die zu ersetzende Zeile auswählt. Um die wenigen Cache-Zeilen bestmöglich zu nutzen, wurde hier als Ersetzungsstrategie Least-Recently-Used (LRU) gewählt.

4.1.2.2 Least-Recently-Used-Logik

Für eine Ersetzungsstrategie nach dem LRU-Prinzip gibt es die unterschiedlichsten Ansätze, wie zum Beispiel quadratische Matrizen, Schieberegister, Zähler, Link-Listen und viele weitere. Als eine recht kostengünstige Lösung, die auch für höhere Assoziativitäten noch gut skaliert, hat sich eine Variante mit einem eindimensionalen systolischen Array erwiesen [SMV04, Gro02]. Dieses Array (Abbildung 4.2a) ist eine Reihe von aneinander geketteten Knoten, wobei es pro Cache-Zeile genau einen Knoten gibt. Ein solcher Knoten (Abbildung 4.2b) enthält ein Register



(a) Aufbau des eindimensionalen systolischen Arrays



(b) Knoten des systolischen Arrays

Abbildung 4.2: Schematischer Aufbau der LRU-Logik

für die Indexnummer einer Cache-Zeile, einen Vergleicher mit der Bus-Breite einer Indexnummer und ein ODER-Gatter.

Wird am vorderen Ende dieser Kette der aktuell genutzte Index angelegt, wandert dieser an allen Knoten vorbei und wird in jedem Knoten mit dem jeweiligen Registerinhalt verglichen. Entspricht der Inhalt eines Registers dem neuen Index, wird er aus diesem entfernt. Das geschieht, indem dieser und alle weiteren Knoten das Signal erhalten, den Registerinhalt des jeweils nächsten Knotens zu übernehmen. Der aktuelle Index wird in das Register des hintersten Knotens kopiert. Auf diese Weise entsteht eine selbstorganisierende Liste mit der Reihenfolge der Zugriffe auf die Indizes. Am vorderen Ende der Liste kann so jederzeit der Index der am Längsten nicht mehr genutzten Cache-Zeile abgelesen werden.

Das Array enthält einen sehr langen kombinatorischen Pfad, der bei einem langen Array das Zeitverhalten negativ beeinflussen könnte. Dieser Pfad ist das Signal, dass allen weiteren Knoten anzeigt, ihren Register-Inhalt aus dem jeweils nächsten Knoten zu holen. Da dieses Signal in jedem Knoten durch ein ODER-Gatter muss, kann es zum kritischen Pfad werden und die zu erreichende Taktfrequenz reduzieren. Deshalb ist in [Gro02] angegeben, dass der neue Index und das kritische Signal zum Verschieben der Array-Elemente in jedem Knoten mit einer Register-Stufe um einen Takt verzögert wird. Diese zusätzlichen Register haben keinen zeitlichen Einfluss auf die Berechnung des LRU-Index, da sich das Array auch zeitverzögert intern organisieren kann. Aber die Register sind im SHAP nicht zwangsweise notwendig, da nicht so hohe Taktraten erreicht werden. Um dem Problem für sehr lange Arrays oder höhere Taktraten trotzdem gerecht zu werden, ist eine Möglichkeit implementiert, diese Register-Stufe in regelmäßigen Abständen in das Array einzufügen und es somit in Blöcke kürzerer kombinatorischer Pfade zu teilen.

4.1.2.3 Verbindung zum Garbage Collector

Mit den Informationen zur physikalischen Adresse der Objekte enthält der Cache Daten, die vom Garbage Collector jederzeit verändert werden können, indem dieser Objekte löscht oder verschiebt. Tritt dieser Fall ein, wird vom GC für einen Takt angezeigt, welches Objekt betroffen ist und ob es gelöscht bzw. wo es hin verschoben wurde. Aus diesem Grund hat die Tag-Einheit einen weiteren Port für den GC, der den Inhalt der Tag-Register bei einem Cache-Miss nicht verändert, aber anzeigt, ob und in welcher Zeile das gesuchte Objekt liegt. Wurde das Objekt gelöscht, wird die Referenz aus dem Tag-Register entfernt. Wenn das Objekt nur verschoben wurde, wird dem Cache-Speicher mitgeteilt, in welcher Zeile die Basisadresse zu aktualisieren ist. Um Ressourcen zu sparen, teilen sich der Port für die normale Cache-Funktion und der Port für die GC-Informationen die Tag-Vergleicher. Werden beide Ports zur gleichen Zeit angesprochen, hat der GC-Port die höhere Priorität und die Anfrage am Port des Kerns wird um einen Takt-Zyklus nach hinten verschoben.

4.1.2.4 Valid-Bits

Die Basisadresse und der Bias-Wert werden immer als gültig angesehen, da diese direkt nach dem Erzeugen einer neuen Cache-Zeile aus dem Hauptspeicher in den Cache geladen werden und dort vom Garbage Collector auch aktuell gehalten werden. Sollte dennoch einmal eine neue Cache-Zeile aktiviert werden müssen, bevor zur alten Zeile die Adressdaten geladen wurden, wird in dieser die Objekt-Referenz aus dem Tag-Register gelöscht und damit die Zeile als ungültig markiert. Dadurch werden für die Basisadresse und den Bias-Wert keine extra Valid-Bits benötigt, die anzeigen, ob die Daten gültig sind.

Im Gegensatz dazu wird für jeden der m Offsets, die im Cache abgelegt werden können, ein Valid-Bit geführt. Diese Valid-Bits werden auch in der Tag-Einheit verwaltet, als jeweils ein Satz pro Cache-Zeile. Die Valid-Bits sind als kompletter Satz nach außen geführt und können auch nur als kompletter Satz geändert werden. Welcher Satz bzw. welche Cache-Zeile die aktuelle ist, wird durch den aktuellen Zeilen-Index ausgewählt.

4.1.3 Offset-Vergleicher

Für jeden Offset der im Cache zwischengespeichert werden soll, wird ein Vergleicher angelegt. Mit dem Ergebnis dieser Vergleicher und den dazugehörigen Valid-Bits werden Signale generiert, die angeben, ob der aktuell angeforderte Offset zwischengespeichert werden kann oder schon im Cache liegt. Da die Vergleicher den Offset jeweils nur mit einer Konstanten vergleichen, können diese Signale kostengünstig aus der kombinatorischen Verknüpfung der einzelnen Bits des Offsets generiert werden. Zudem wird ein Offset-Index-Signal generiert, mit dessen Hilfe das richtige Offset-Wort im Cache adressiert wird.

4.1.4 Cache-Speicher

Der Cache-Speicher besteht aus drei Teilen (Abbildung 4.3). Ein Speicher für die Basisadressen, einer für die Bias-Werte und einer für die Daten-Wörter (Offset-Werte). Für die beiden Adressspeicher ist es notwendig, dass diese im Write-First-Modus arbeiten, da ein zu schreibender Wert unter Umständen sofort im nächsten Takt gebraucht werden kann. Für den Speicher der Basisadressen ist der Write-First-Modus durch zusätzliche Bypass-Register implementiert, da der Dual-Port-Speicher diesen Schreibmodus hier nicht unterstützt. Die Adressierung der beiden Adressspeicher erfolgt über den Tag-Index, den die Tag-Einheit generiert.

Der Speicher für die Basisadresse hat zwei Schreib-Ports, einen für das normale Zwischenspeichern der geladenen Adresse und einer für den Garbage Collector, um die geänderten Adressen verschobener Objekte aktualisieren zu können. Da im SHAP kein Speichermodul mit zwei Schreib-Ports zur Verfügung steht, wurde ein Dual-Port-Speicher verwendet, der getrennte Schreib- und Lese-Ports hat. Damit ist es möglich, auf getrennten Adressen zu schreiben und zu lesen. Die beiden Schreib-Eingänge werden per Multiplexer auf den Schreib-Port geführt. Kommen über den normalen und den GC-Port gleichzeitig Schreibaufforderungen, hat der GC-Port Vorrang und die Verarbeitung am normalen Port wird um einen Takt verzögert. Durch die

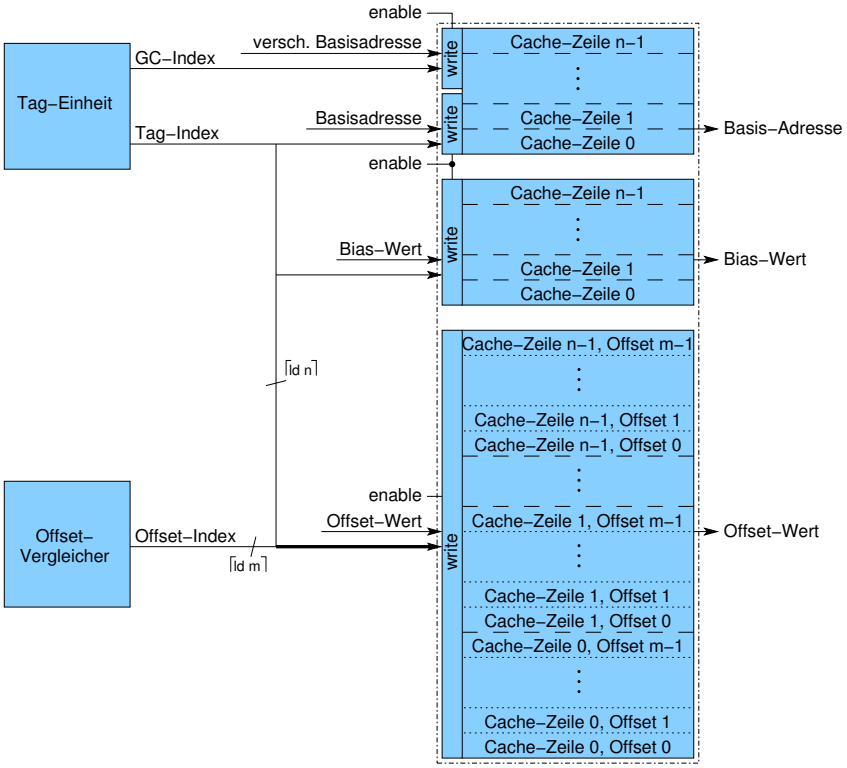


Abbildung 4.3: Schematischer Aufbau des Cache-Speichers

getrennten Schreib- und Lese-Ports können nicht aktive Cache-Zeilen geändert werden, ohne den aktuell gelesenen Wert zu verändern.

Im Offset-Speicher werden die einzelnen Offset-Wörter übereinander gestapelt, da immer nur ein Offset-Wort zur selben Zeit gebraucht wird. So wird vermieden, dass das Design einen Speicher mit sehr hoher Datenbreite aber nur sehr geringer Adresstiefe bekommt, was vor allem in FPGAs zu einem ungünstigen Ressourcenverbrauch führen würde. In einem FPGA würden dann viele Blöcke mit geringerer Datenbreite aber höherer Adresstiefe parallel geschalten, deren Speicherplatz dann kaum ausgenutzt würde. Aus diesem Grund werden der Tag-Index und der Offset-Index zu einer Cache-Adresse verkettet.

4.1.5 Maßnahmen zur Erhaltung der Kohärenz

Gemäß den Grundlagen aus Kapitel 2 gibt es nur wenige Fälle, in denen die Kohärenz herzustellen ist.

Das ist zum einen beim Betreten von `synchronized`-Abschnitten, zu erkennen am Bytecode `monitorenter`. In diesem Fall werden alle Valid-Bits des eigenen Caches zurückgesetzt, um bei Bedarf die aktuellen Werte aus dem Hauptspeicher zu lesen. Die dortigen Werte werden immer aktuell gehalten, indem der Cache im Write-Through-Modus arbeitet, d.h. alle Schreibvorgänge werden an den Hauptspeicher weitergegeben. Ein weiterer Vorteil dieses Schreibmodus ist es auch, dass der Garbage Collector nicht jeden Cache nach eventuellen neuen oder bereits überschriebenen Objektreferenzen durchsuchen muss.

Ein weiterer Fall, bei dem die aktuellen Werte aus dem Hauptspeicher gelesen werden müssen, sind Zugriffe auf als `volatile` gekennzeichnete Datenfelder. Diese Felder sind aber nicht mehr als solche am Bytecode zu erkennen. Aus diesem Grund wurde ein neuer Bytecode und ein zugehöriger neuer Microcode im System hinzugefügt. Dieser Bytecode wird vom SHAP-Linker vor jedem Lesen eines als `volatile` gekennzeichneten Datenfeldes in den Bytecode-Stream eingefügt. Dieser neue Bytecode sorgt dann dafür, dass alle Valid-Bits zurückgesetzt werden. Die Spezifikation der JVM gibt an, dass nur der Zugriff auf das eigentliche `volatile`-Feld zum bzw. vom Hauptspeicher durchzuführen ist [LY99]. Das heißt bei einem als `volatile` markierten Array werden nur Zugriffe auf die Array-Referenz, nicht auf dessen Elemente, am Cache vorbei geleitet. Viele andere Autoren und Programmierer behaupten aber, dass alle Elemente [Ull06] als `volatile` agieren, bzw. dass dieses Verhalten durch das Überschreiben der Array-Referenz mit sich selbst [Cof12a, Cof12b, Man12] emuliert werden kann, da man davon ausgeht, dass dann der komplette Cache mit dem Hauptspeicher synchronisiert wird. Um dem gerecht zu werden, werden beim Auftreten von `volatile`-Zugriffen alle Valid-Bits zurückgesetzt.

Speziell im SHAP gibt es noch den Fall des Thread-Wechsels, in dem sich der Cache mit dem Hauptspeicher synchronisieren muss. Da im Thread-Objekt Datenfelder genutzt werden, die auf die anderen Thread-Objekte verweisen muss sichergestellt werden, dass der Kern nach Anlegen eines neuen Threads diesen auch in seiner Liste findet, um diesen zu starten. Werden aus dem Cache nur alte Referenzen auf die schon bestehenden Threads gelesen, kann es vorkommen, dass ein neuer Thread nicht startet. Deswegen wird auch vor jedem Thread-Wechsel der Microcode ausgeführt, der den Cache-Inhalt ungültig macht, indem die Valid-Bits zurückgesetzt werden (analog `volatile`).

Die Cache-Zeilen und die damit verbunden Daten zur physischen Adresse des jeweiligen Objektes sind davon nicht abhängig und bleiben weiterhin im Cache gültig, bis sie überschrieben werden. Es werden in allen drei oben beschriebenen Fällen nur die Valid-Bits für den Offset-Teil des Caches zurückgesetzt, in dessen zugehörigen Kern das entsprechende Ereignis eingetreten ist.

4.2 Auswertung der implementierten Lösung

4.2.1 Leistungssteigerung

Als Messwert für die Leistungssteigerung bei Mehrkernprozessoren wird hier der Speed-Up herangezogen. Dieser berechnet sich als Quotient der Ausführungszeit T eines Benchmarks auf einem Mehrkernprozessor mit p Kernen im Vergleich zu einem Einkernprozessor. Die Ausführungszeit hängt aber auch von der Anzahl I der durchlaufenen Iterationen ab, die sich bei manchen Benchmarks nach der Anzahl der Kerne richtet. Deswegen kann der Speed-Up auch in Abhängigkeit der Rechenleistung L in Iterationen pro Sekunde berechnet werden:

$$L_p = \frac{I_p}{T_p} \tag{4.1}$$

$$S_p = \frac{L_p}{L_1} = \frac{T_1}{T_p}\Big|_{I_1=I_p} \tag{4.2}$$

Die Messung der Ergebnisse erfolgte auf einem FPGA-Board XUPV5 mit einem Virtex-5 von Xilinx als FPGA. Der FPGA hat theoretisch genug Ressourcen um SHAP-Konfigurationen mit bis zu 18 Kernen aufzunehmen. Die Möglichkeiten des Routings begrenzen aber sinnvolle Ergebnisse auf 16 Kerne, da ab 17 Kernen eine Taktfrequenz von 80 MHz nicht mehr garantiert werden kann. Die gemessene Rechenleistung, die der Berechnung der Speed-Up-Werte zugrunde liegt, ist in Anhang B zu finden.

Aus den Erkenntnissen aus Kapitel 3 ist zu vermuten, dass schon vier Cache-Zeilen genügen, um eine deutlich Leistungssteigerung zu erreichen. Abbildung 4.4 bestätigt diese Annahme für den Lift-Benchmark. Wie in der Simulation werden auch hier neben den Daten zur physikalischen Adresse nur die Offsets -2 und 1 zwischengespeichert. Der Speed-Up für 16 Kerne bei vier Cache-Zeilen kann von bisher 7,9 auf 12,0 gesteigert werden. Eine Verdoppelung auf acht Cache-Zeilen hat nur noch geringe Verbesserungen auf 12,4 zur Folge. Insgesamt ergibt sich bei acht Cache-Zeilen ein Plus von 57%.

Anders sieht die Situation beim JGF Matrixmultiplizierer für dünnbesetzte Matrizen (SparseMatMult) aus. In Abbildung 4.5 ist zu sehen, dass mit vier Cache-Zeilen der Speed-Up bei 16 Kernen von 7,6 auf nur 9,2 gesteigert werden kann. Einen größeren Leistungsgewinn erhält man erst mit acht Cache-Zeilen. Der Speed-Up kann so bei 16 Kernen auf 14,1 gesteigert werden. Dies entspricht einem Plus von 86%.

Um festzustellen, ob wirklich acht Cache-Zeilen notwendig sind, wurden die Benchmarks

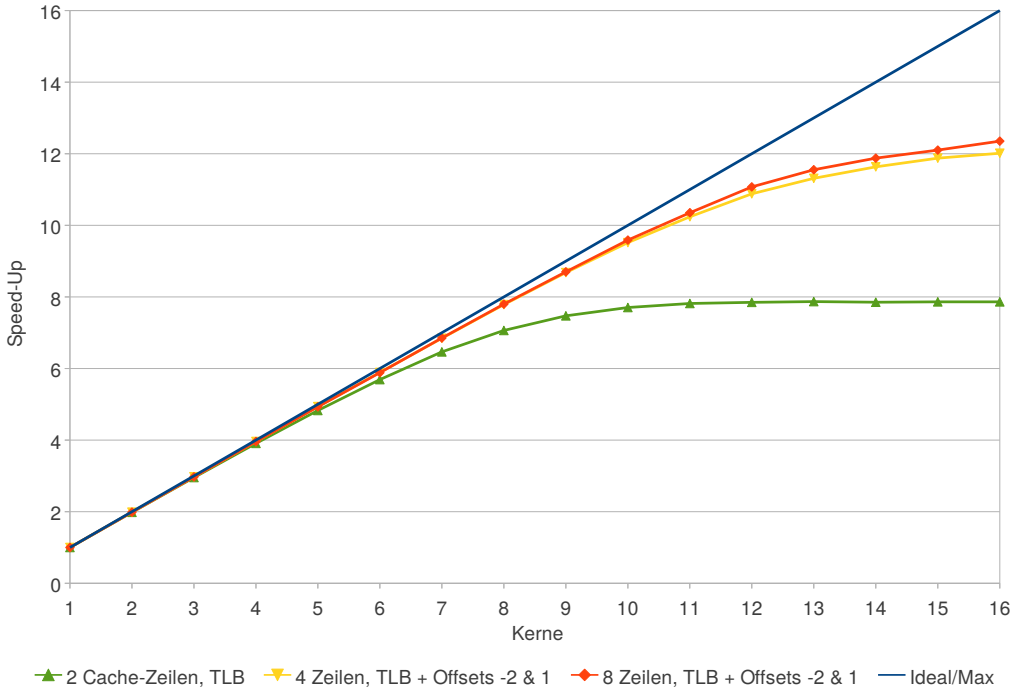


Abbildung 4.4: Speed-Up für den Benchmark JEM Lift

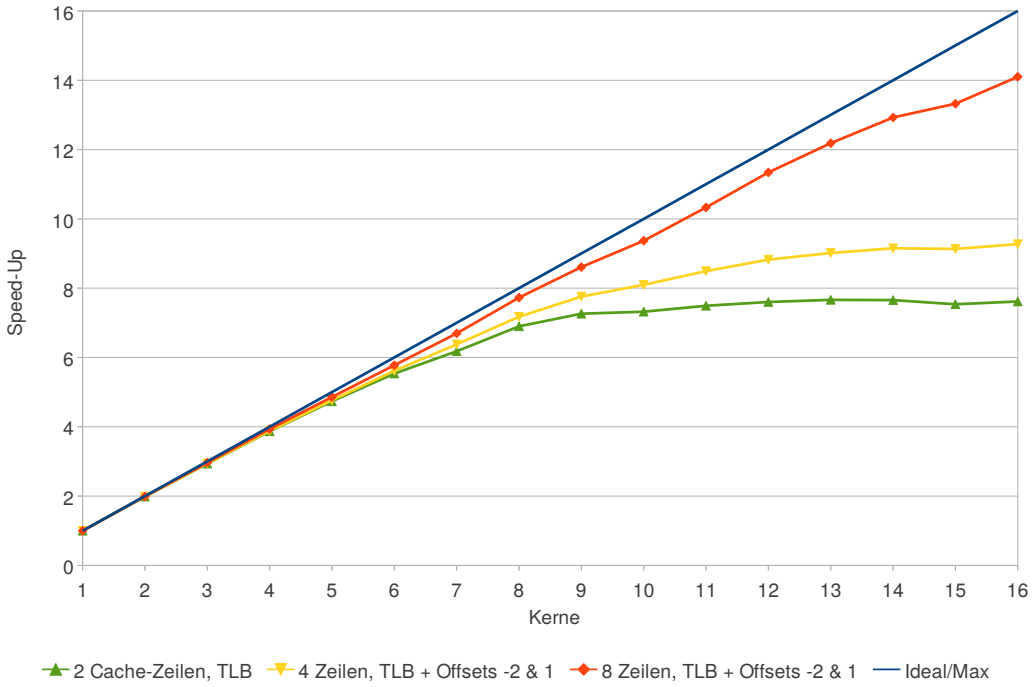


Abbildung 4.5: Speed-Up für den Benchmark JGF SparseMatMult

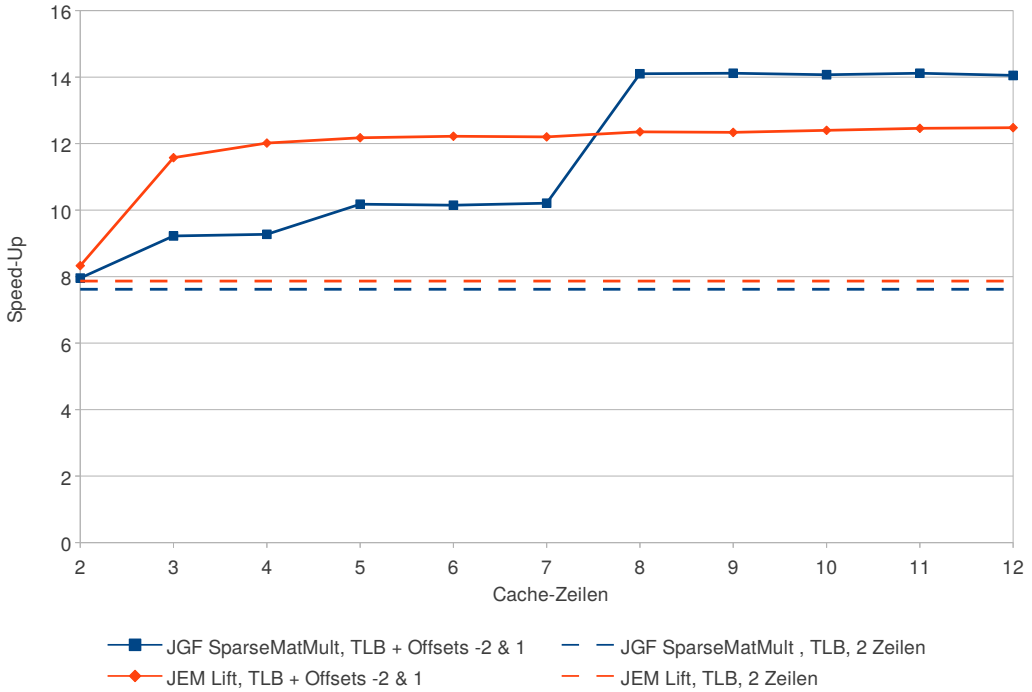


Abbildung 4.6: Speed-Up bei 16 Kernen in Abhängigkeit der Anzahl der Cache-Zeilen

Lift und SparseMatMult in Abhängigkeit der Anzahl der Cache-Zeilen ausgeführt. Der resultierende Speed-Up für 16 Kerne ist in Abbildung 4.6 dargestellt, wobei die bisherigen Werte gestrichelt eingezeichnet sind. Man sieht nun, dass Lift auch schon mit drei Cache-Zeilen gute Ergebnisse bringt, der Matrixmultiplizierer aber wirklich acht Zeilen braucht, um auf hohe Speed-Up-Werte zu kommen. Wie viele Cache-Zeilen benötigt werden, hängt von der Anzahl der Objekte ab, die in den Berechnungsschleifen angesprochen werden. Werden mehr Objekte referenziert, als Cache-Zeilen vorhanden sind, verdrängen sich die Objekte fortwährend gegenseitig aus dem Cache.

Für den Benchmark FScript hat der Objekt-Cache nur geringen Einfluss, der größere limitierende Faktor ist hier das Allokieren neuer Objekte. Aufgrund des begrenzten Heap-Speichers muss der Garbage Collector sehr häufig aktiv werden. Hat der GC nur die standardmäßige Priorität gegenüber anderen Speicherzugriffen, sinkt die Leistung mit steigender Kernanzahl wieder, da nicht schnell genug Platz für neue Objekte geschaffen werden kann (Abbildung 4.7). Setzt man den Garbage Collector auf eine hohe Priorität erreicht FScript etwas bessere Leistungswerte (Abbildung 4.8). Der Speed-Up kann bei 16 Kernen und acht Cache-Zeilen trotzdem nur von 8,0 auf 9,5 gesteigert werden.

4.2.2 Eingesparte Hauptspeicherzugriffe

Ziel des Objekt-Caches war, Speicherzugriffe einzusparen. In Abbildung 4.9 ist zu sehen, dass die durchschnittliche Anzahl der Speicherzugriffe pro Takt von 0,071 auf 0,044 bei vier Cache-

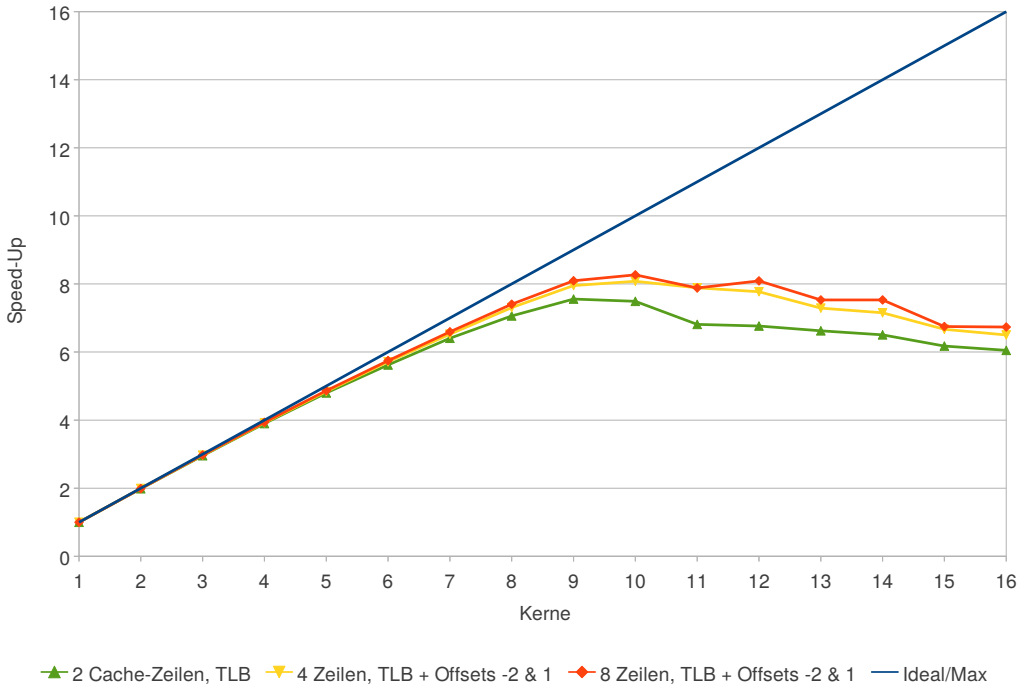


Abbildung 4.7: Speed-Up für den Benchmark FScript, Standard GC-Priorität

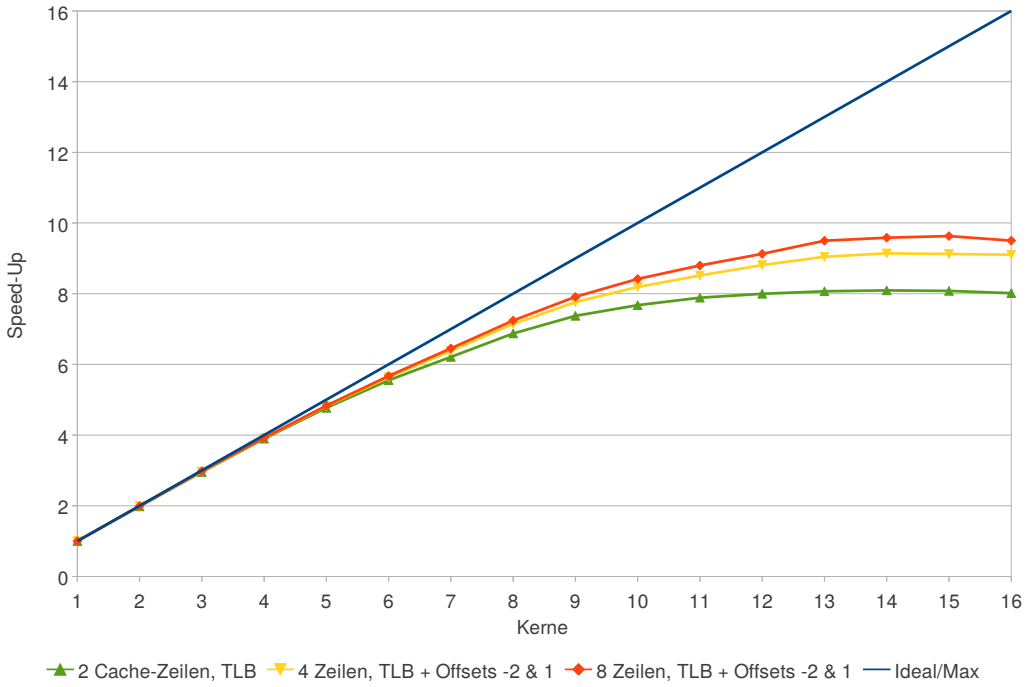


Abbildung 4.8: Speed-Up für den Benchmark FScript, hohe GC-Priorität

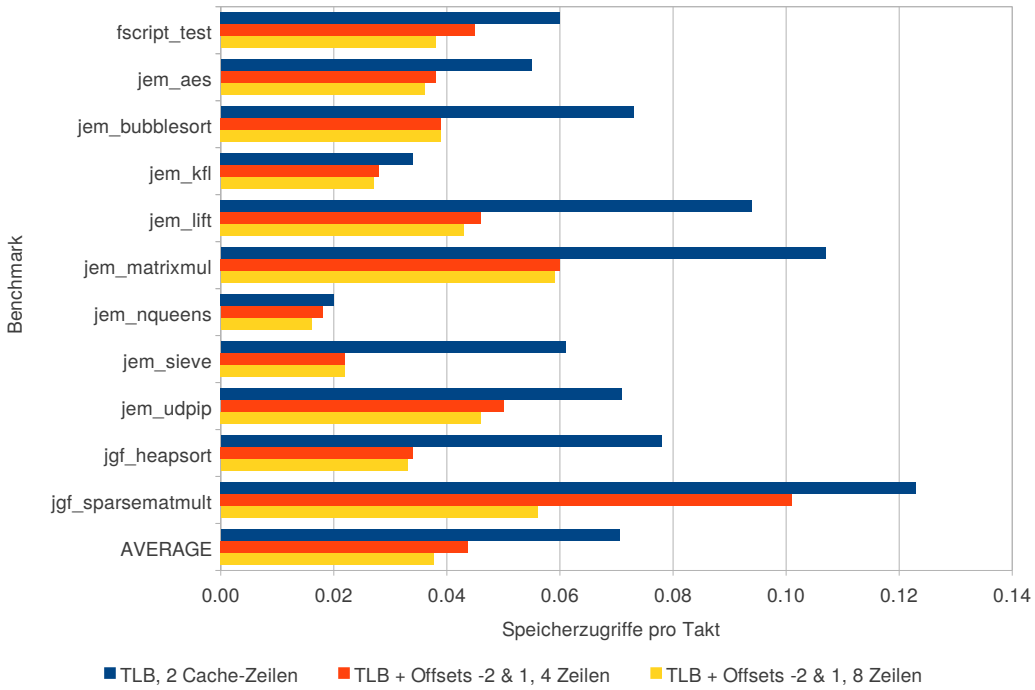


Abbildung 4.9: Vergleich Speicherzugriffe pro Takt mit und ohne Objekt-Cache

Zeilen, bzw. auf 0,038 bei acht Cache-Zeilen, reduziert werden kann. Dies entspricht einer Einsparung von 38% bei vier Cache-Zeilen und 46,5% bei acht Cache-Zeilen. Dieser Wert deckt sich mit den Simulationsergebnissen, die ca. 40% Einsparung ergaben. Es ist auch zu sehen, dass nur der Benchmark JGF SparseMatMult von acht statt vier Cache-Zeilen profitiert.

4.2.3 FPGA-Ressourcenbedarf

Mit dem neuen Objekt-Cache erhöht sich auch der Ressourcenbedarf. Dieser ist hängt vom FPGA-Board ab, auf dem SHAP implementiert wird. Die folgenden Werte beziehen sich auf einen Virtex-5-FPGA von Xilinx auf einem XUPV5-Board. Wichtige Eckdaten des Ressourcenverbrauches sind die Menge der Lookup-Tabellen (LUTs), Register (Regs) und Block-RAMs der Größen 18 kBit (18k-BRAMs) und 36 kBit (36k-BRAMs). Die Werte stammen aus der Ausgabe der Synthese-Tools. Der Bedarf pro Kern berechnet sich aus dem durchschnittlichen Zuwachs für jeden weiteren Kern. Der von der Kernanzahl unabhängige Grundbedarf an Ressourcen berechnet sich aus der Differenz zwischen dem Bedarf für die Einkernvariante und dem durchschnittlichen Bedarf pro Kern.

Der bisherige Ressourcenbedarf berechnet sich in Abhängigkeit der Kernanzahl p wie folgt:

$$\begin{aligned} \text{LUTs}(p) &\approx 2816 + p \cdot 2835 && (4.3)\\ \text{Regs}(p) &\approx 1973 + p \cdot 1447 && (4.4)\\ \text{18k-BRAMs}(p) &= 1 + p \cdot 2 && (4.5)\\ \text{36k-BRAMs}(p) &= 1 + p \cdot 3 && (4.6) \end{aligned}$$

Inklusive einem Objekt-Cache mit acht Cache-Zeilen sieht der Ressourcenbedarf wie folgt aus:

$$\begin{aligned} \text{LUTs}(p) &\approx 2816 + p \cdot 3026 && (4.7)\\ \text{Regs}(p) &\approx 1973 + p \cdot 1557 && (4.8)\\ \text{18k-BRAMs}(p) &= 1 + p \cdot 4 && (4.9)\\ \text{36k-BRAMs}(p) &= 1 + p \cdot 4 && (4.10) \end{aligned}$$

Der neue Objekt-Cache kostet demnach zusätzlich pro Kern etwa 191 LUTs und 110 Register. Zudem fällt der höhere Bedarf an Block-RAM-Modulen auf. Für den Offset-Speicher wird pro Kern ein 36-kBit-Modul verwendet und für die Basisadressen und die Bias-Werte je ein 18-kBit-Modul. Der größere 36-kBit-Speicher für die Offsets begründet sich mit der Daten-Bus-Breite von 32 Bit, die von den kleineren 18-kBit-Speichern nicht zur Verfügung gestellt werden kann. Die Speicher-Module werden aber nur schwach ausgelastet, da diese Module jeweils für mindestens 1024 Datensätze Platz haben [Xil10]. Benötigt wird aber nur ein Datensatz je Cache-Zeile für die Adressspeicher und ein Datensatz für jeden Offset-Wert pro Cache-Zeile im Offset-Speicher.

Xilinx bietet zwei Varianten für Speicher-Module im FPGA an. Den hier genutzten Block-RAM und kleinere Speicherelemente, die „Distributed RAM“ genannt werden. Diese nutzen

mehrere LUTs als Speicher und können mindestens 32 Datensätze speichern [Xil10]. Dem Synthese-Tool wurde hier die Wahl gelassen, anhand der Speichergröße und den vorhanden Ressourcen selbst zu entscheiden, welcher Speicher-Typ verwendet wird. Vom Tool wurde für alle Speicher-Module der Block-RAM ausgewählt.

Da die BRAM-Module für wesentlich mehr Cache-Zeilen Platz bieten als hier betrachtet, ist deren Anzahl für die hier betrachtete Anzahl von Cache-Zeilen konstant. Einzig der Bedarf an LUTs und Registern ist abhängig von der Anzahl der Zeilen. Im Vergleich zum SHAP ohne Objekt-Cache ergibt sich folgender Mehrbedarf pro Kern für n Cache-Zeilen:

$$\text{LUTs}(n) \approx 79 + n \cdot 15 \tag{4.11}$$

$$\text{Regs}(n) \approx -44 + n \cdot 20 \tag{4.12}$$

Diese Werte sind aber nur eine grobe lineare Näherung, da der reale Bedarf nicht linear ist, sondern mit der Anzahl der Index-Bits für die Adressierung der Cache-Zeilen auch eine logarithmische Komponente enthält. Diese ist hier aber sehr gering, da die linear steigende Anzahl der Register und LUTs für die Tag-Speicher und -Vergleicher sowie die Valid-Bits überwiegt.

Auffallend ist, dass bis einschließlich zwei Cache-Zeilen weniger Register gebraucht werden, als bisher. Die Ursache ist die Verlagerung der bisherigen TLB-Speicher von Registern in Block-RAM-Module.

4.2.4 Weitere Auswertungen

4.2.4.1 Mehr Offsets

Der Objekt-Cache ist nicht nur hinsichtlich der Anzahl der Cache-Zeilen konfigurierbar, auch wie viele und welche Offsets zwischengespeichert werden sollen, kann eingestellt werden. Abbildung 4.10 zeigt für acht Cache-Zeilen den Vergleich zwischen einem Cache für die Offsets -2 und 1 und einem Cache mit den acht häufigsten Offsets, die sich aus der Analyse in Kapitel 3 ergeben. Das sind die Offsets 1, -2, -5, 4, 2, -4, -1 und 8.

Im Ergebnis ist festzustellen, dass sich für den Matrixmultiplizierer (SparseMatMult) keine nennenswerten Verbesserungen ergeben. Einzig der Benchmark Lift zeigt leichte Verbesserungen. Der Speed-Up bei 16 Kernen steigt von 12,4 auf 13,7. Die Ursache liegt im Offset -5, da Lift maßgeblich dafür verantwortlich ist, dass -5 der dritthäufigste Offset ist. Weitere Offsets im Cache zu halten ist also hinsichtlich des Leistungsgewinnes sehr anwendungsspezifisch.

Da der Block-RAM genügend Speicherplatz für weitere Offsets bietet, kostet es keine zusätzlichen Speicher-Module. Im Gegensatz dazu erhöht sich aber der Bedarf an LUTs und Registern, da mehr Valid-Bits benötigt werden und die Offsets mit noch mehr Konstanten verglichen werden müssen. Der neue Ressourcenbedarf ergibt sich wie folgt:

$$\text{LUTs}(p) \approx 2816 + p \cdot 3053 \tag{4.13}$$

$$\text{Regs}(p) \approx 1973 + p \cdot 1607 \tag{4.14}$$

Im Vergleich zu den Gleichungen 4.7 und 4.8 kostet die Erweiterung von zwei auf acht Offsets bei acht Cache-Zeilen pro Kern zusätzlich 27 LUTs und 50 Register.

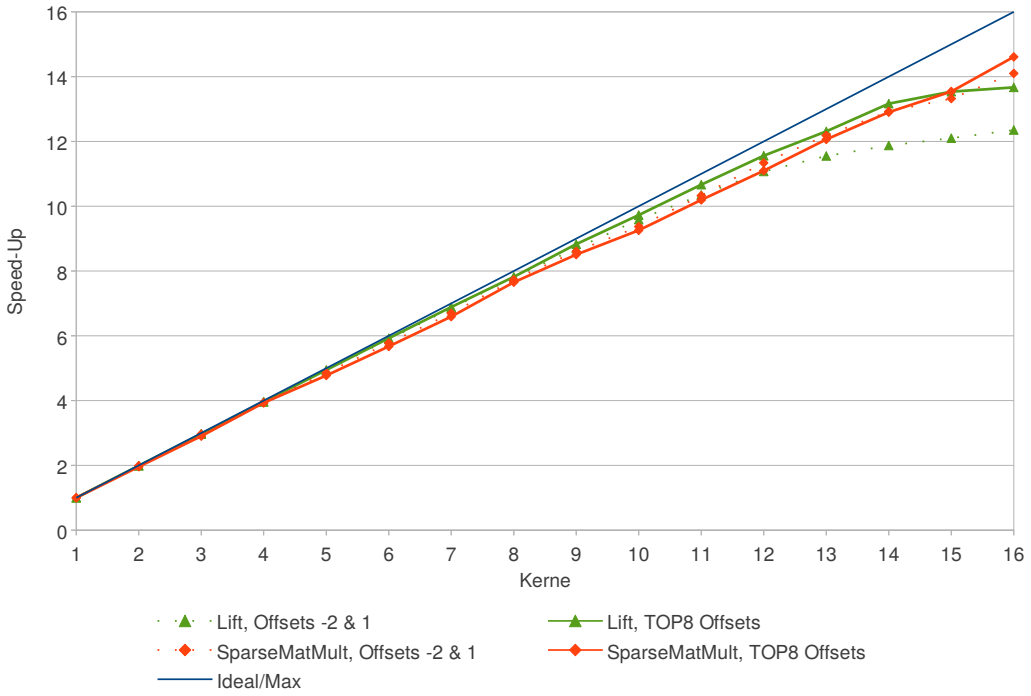


Abbildung 4.10: Vergleich Speed-Up mit Offsets -2 & 1 und den 8 häufigsten Offsets

4.2.4.2 Distributed RAM

Um zu vermeiden, dass das Synthese-Tool fast leere Block-RAMs als Speicher verwendet, kann angewiesen werden, dass Distributed RAM zu verwenden ist. In diesem Fall werden im Vergleich zum alten System ohne Objekt-Cache keine weiteren Block-RAM-Module benötigt. Dafür erhöht sich der allgemeine Bedarf an Registern und LUTs:

$$\text{LUTs}(p) \approx 2816 + p \cdot 3078 \tag{4.15}$$

$$\text{Regs}(p) \approx 1973 + p \cdot 1580 \tag{4.16}$$

Distributed RAM kostet demnach im Vergleich zur Block-RAM-Lösung in 4.7 und 4.8 zusätzlich 52 LUTs und 23 Register pro Kern. Dieser Speicher stellt in der Konfiguration mindestens 32 Speicherplätze zur Verfügung. Ein Cache mit acht Cache-Zeilen und zwei Offsets benötigt für den Adressspeicher nur acht Speicherplätze und für den Datenspeicher 16 Speicherplätze.

4.2.4.3 Leistungssteigerung des Einkernprozessors

Auch auf dem Einkernprozessor konnte mit acht Cache-Zeilen eine leichte Leistungssteigerung erzielt werden (Abbildung 4.11). Die bisherige Rechenleistung ist im Diagramm gestrichelt dargestellt. So konnte zum Beispiel Lift von 17.550 Iterationen pro Sekunde (IpS) um 12% auf 19.700 IpS gesteigert werden. SparseMatMult erreicht sogar eine Steigerung um 20% (von 19,1 IpS auf 23 IpS).

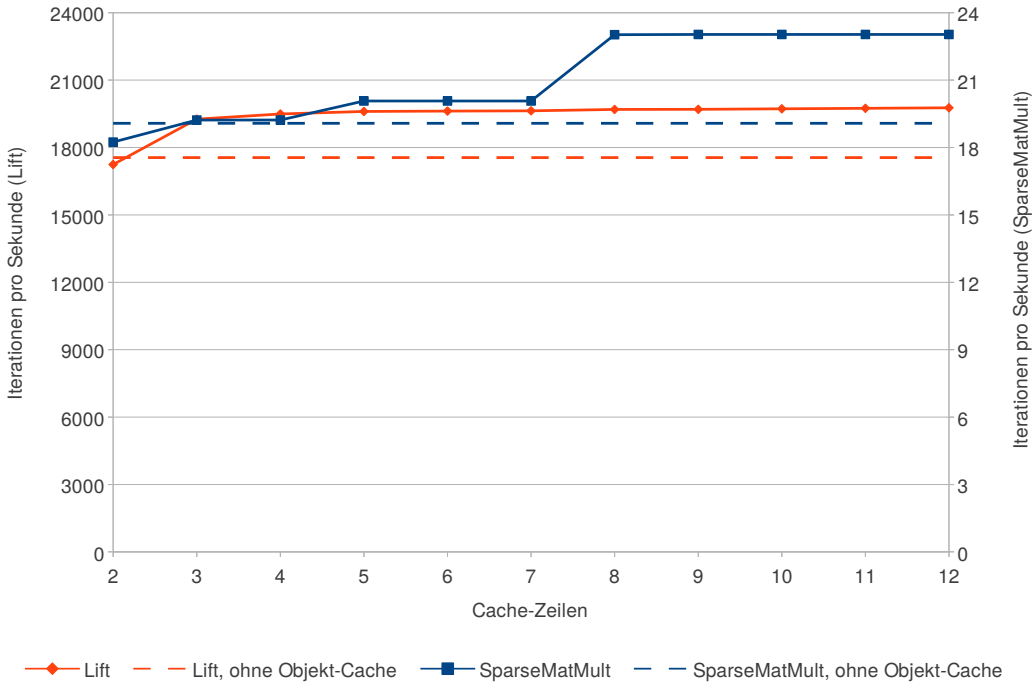


Abbildung 4.11: Rechenleistung auf einem Kern in Abhängigkeit der Anzahl der Cache-Zeilen

Die Ursache für die Leistungssteigerung liegt hier nicht an den eingesparten Speicherzugriffen, sondern zum einen in einer höheren Zahl an zwischengespeicherten physikalischen Adressen, die jetzt nicht mehr aus dem Hauptspeicher geladen werden müssen. Zum anderen werden aber auch Offset-Werte, die im Cache liegen, innerhalb eines Taktes an den Kern weitergegeben.

Es braucht aber mindestens drei Cache-Zeilen, um schneller als bisher zu sein. Das liegt daran, dass der Speicherzugriff bei einem Cache-Miss beim Aktivieren einer Objekt-Referenz jetzt einen Takt länger dauert. Die Tag-Einheit benötigt durch die vielen Vergleicher diesen zusätzlichen Takt.

Zusammen mit dem gesteigerten Speed-Up von 14,1 erhöht sich bei 16 Kernen die absolute Rechenleistung vom Benchmark SparseMatMult von 145,3 IpS auf 324,7 IpS. Das entspricht einem Plus von 123%. Lift erreicht hier eine Steigerung von immerhin noch 76% (von 138.026 Ips auf 243.309 IpS).

5 Zusammenfassung und Ausblick

In dieser Arbeit wurde eine Möglichkeit vorgestellt mit Hilfe eines objektorientierten Caches die Leistung der Speicheranbindung des SHAP-Mehrkernprozessors zu verbessern. Der implementierte Objekt-Cache ist speziell auf den Daten-Port der SHAP-Architektur angepasst und berücksichtigt die Besonderheiten des objektorientierten Speicherzugriffes. Dabei werden nicht nur die eigentlichen Nutzdaten zwischengespeichert, sondern auch die physikalischen Adressen der intern virtuell adressierten Objekte.

Für den Entwurf wurde das bestehende System analysiert und mit Hilfe der SHAP-eigenen Trace-Architektur die Vorgänge am Daten-Port eines Rechenkerns ausgewertet. Mit diesen Daten wurden für verschiedene Cache-Konfigurationen Simulationen durchgeführt und bewertet. Die verschiedenen Konfigurationen unterscheiden sich hinsichtlich der Größe des Caches und Umfangs der Daten, die zwischengespeichert werden. Auf Basis der Simulationsergebnisse erfolgte die Konzeption eines Objekt-Caches.

Entworfen wurde ein kleiner vollassoziativer Cache mit einer Ersetzungsstrategie nach dem Least-Recently-Used-Prinzip. Der Cache beachtet dabei alle Belange der Synchronisation zwischen einzelnen Threads auf verschiedenen Kernen, um die Kohärenz zu erhalten. Es werden auch die Nachrichten des Garbage Collectors über gelöschte oder verschobene Objekte ausgewertet, um damit die zwischengespeicherten Daten bei Bedarf zu aktualisieren. Die Implementation des Caches erfolgte in der Hardwarebeschreibungssprache VHDL und ist statisch parametrierbar. So ist zum Beispiel die Anzahl der Cache-Zeilen frei einstellbar. Genauso kann auch die Liste der Offsets beliebig angepasst werden, die vom Cache zwischengespeichert werden können.

Die prototypische Implementation wurde auf einem FPGA-Board mit verschiedenen Benchmarks getestet und die Rechenleistung ausgewertet. Vor allem für eine höhere Anzahl von Kernen konnte die Rechenleistung gegenüber der bisherigen Lösung deutlich gesteigert werden. Mit dem Cache können durchschnittlich 46,5% der Speicherzugriffe abgefangen werden. Dadurch steigt der maximal mögliche Speed-Up, da jetzt deutlich mehr Threads auf mehr Kernen parallel ausgeführt werden können, bevor die Bandbreite ausgelastet ist. So kann der Speed-Up bei 16 Kernen teilweise von 7,6 auf 14,1 gesteigert werden, was einem Plus von 86% entspricht. Auch auf dem Einkernprozessor konnte die Leistung um bis zu 20% gesteigert werden. Zusammen ergibt das dann bei 16 Kernen sogar eine Steigerung der absoluten Rechenleistung um 123%.

Die Cache-Architektur ist darauf ausgelegt, dass als externer Speicher ein SRAM verwendet wird, der pro Speicherzugriff ein Datenwort liefert. Anders sieht es bei einem externen DDR-RAM aus, bei dem durch den Burst-Zugriff mehrere Datenwörter in einem Zug gelie-

fert werden. Der Nachteil dieser Speicheranbindung sind die hohen Zykluszeiten, die für jeden Burst-Zugriff benötigt werden. Dies ist auch für die Architektur des Caches zu beachten. Die Analyse zeigt aber, dass es wenig sinnvoll ist, alle Daten eines Burst-Zugriffes in den jeweiligen Caches der einzelnen Kerne zu halten. Viel mehr wäre zu untersuchen, ob nicht ein gemeinsamer Level-2-Cache am Speicher-Controller sinnvoll ist, um Zugriffs- und Zykluszeiten des DDR-RAM zu verdecken.

Die vorgestellte Implementation beschränkt sich auf die Offsets -2 und 1. Die Analyse zeigt aber, dass man durch weitere Offsets gezielt einzelne Anwendungen beschleunigen kann. Da sich die weiteren Offsets aber je nach ausgeführter Anwendung unterscheiden, wäre ein Ansatz zur weiteren Beschleunigung eine adaptive oder rekonfigurierbare Lösung. So könnte durch eine gezielte Laufzeitanalyse für jede Anwendung die optimale Konfiguration der zwischenspeicherbaren Offsets eingestellt werden. Ziel wäre eine dynamische Parametrierung für eine gegebene Anwendung ohne Änderung oder Neusynthese der Hardware.

Anhang A

Simulationsergebnisse

A.1 Offsets -5 bis 10, Burst-Zugriff

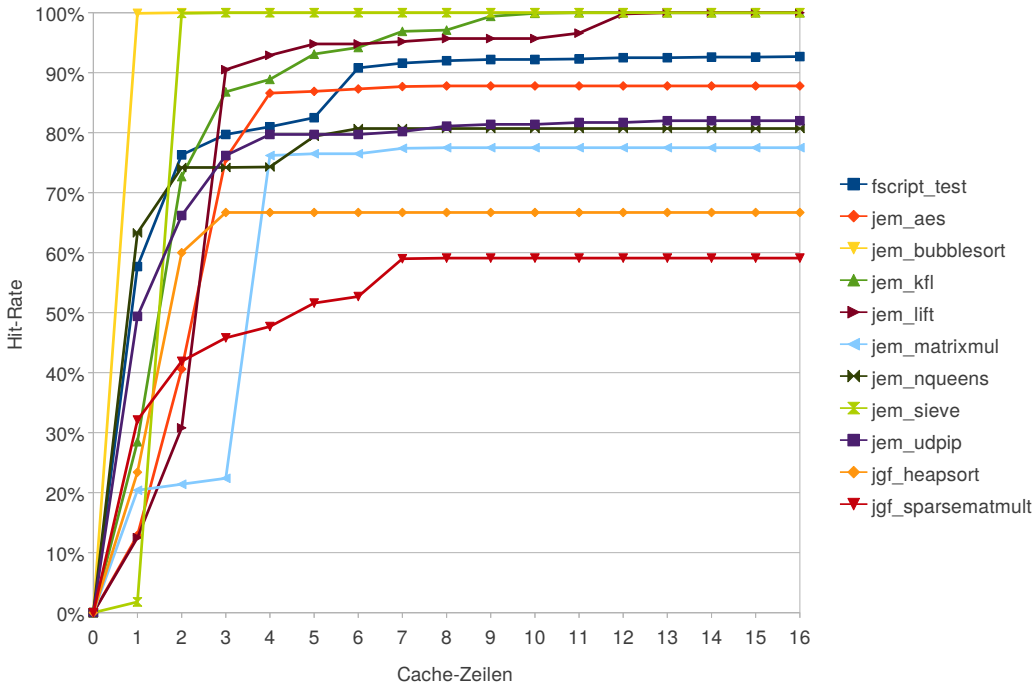


Abbildung A.1: Offsets -5 bis 10, Burst-Zugriff - Hit-Rate

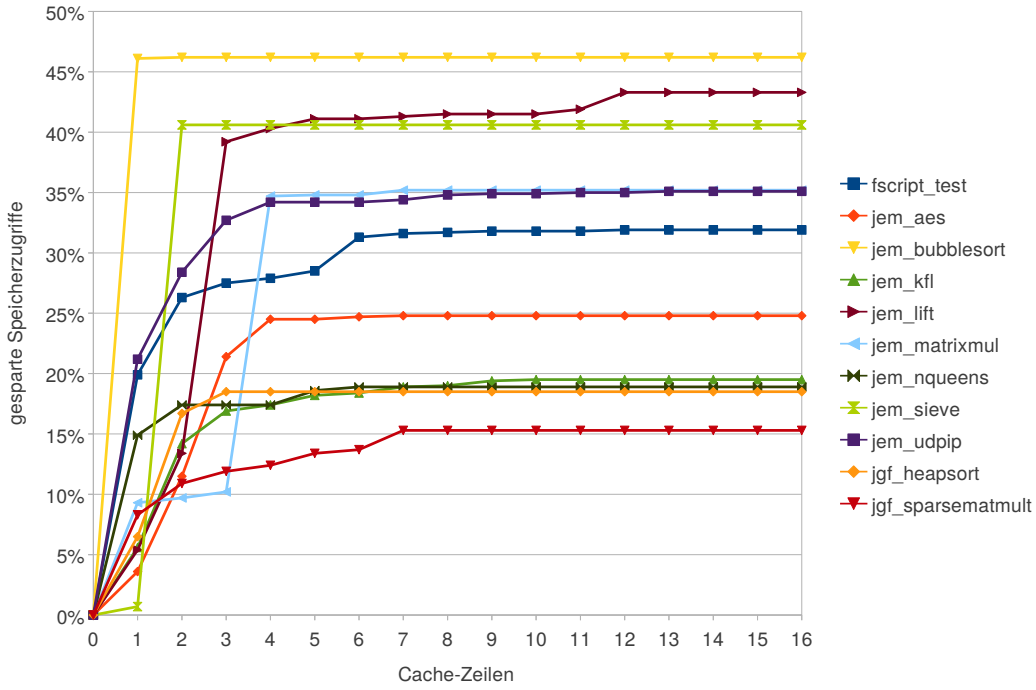


Abbildung A.2: Offsets -5 bis 10, Burst-Zugriff - Eingesparte Speicherzugriffe

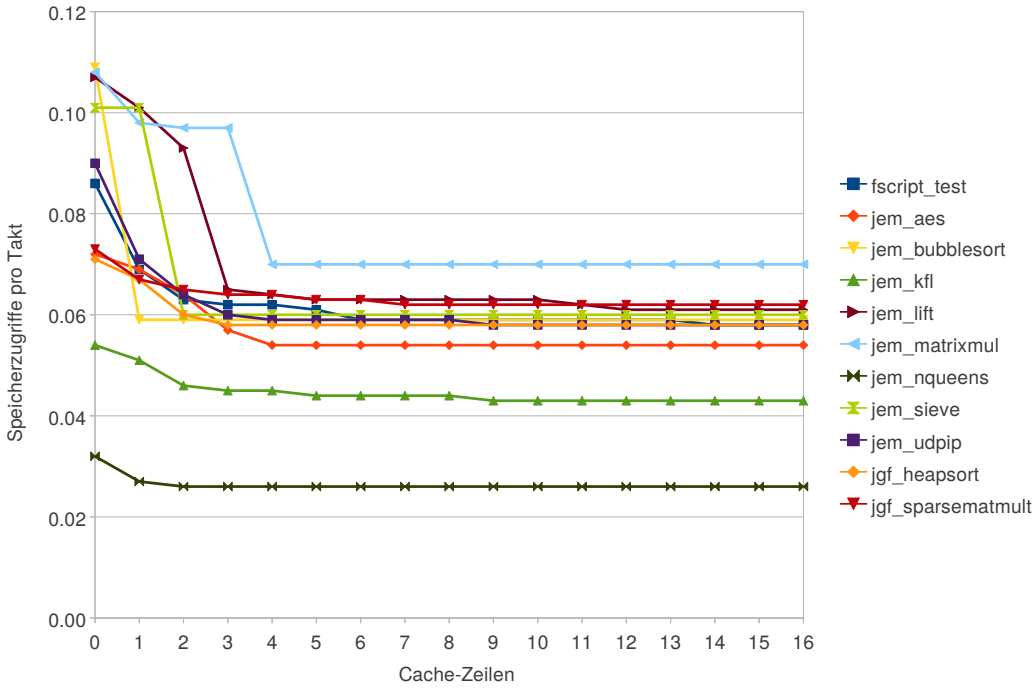


Abbildung A.3: Offsets -5 bis 10, Burst-Zugriff - Speicherzugriffe pro Takt

A.2 Offsets -3 bis 4, Burst-Zugriff

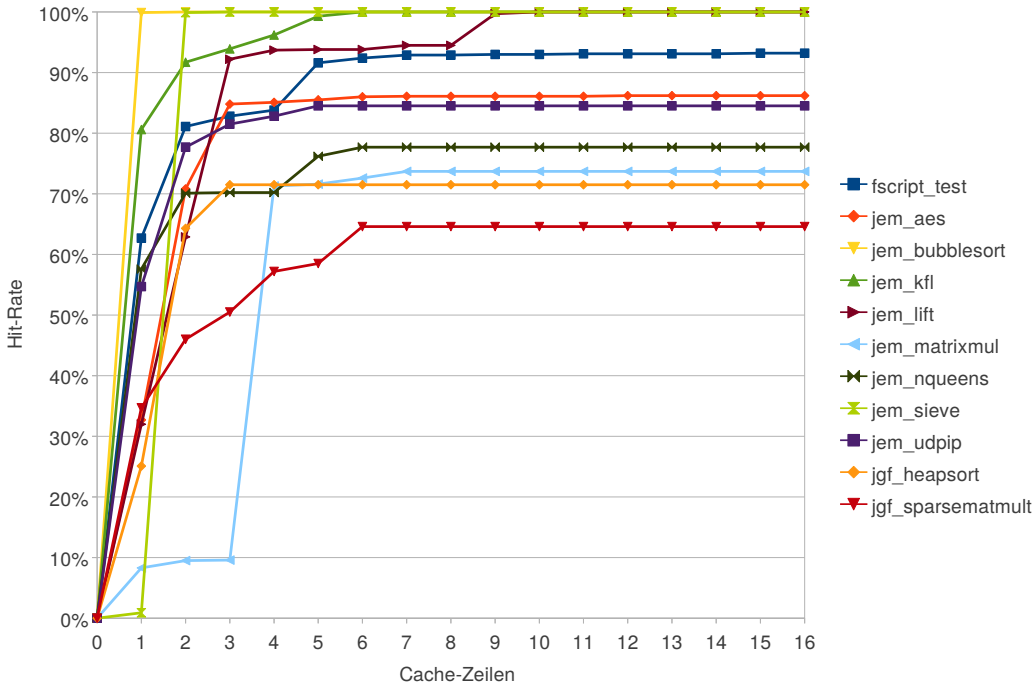


Abbildung A.4: Offsets -3 bis 4, Burst-Zugriff - Hit-Rate

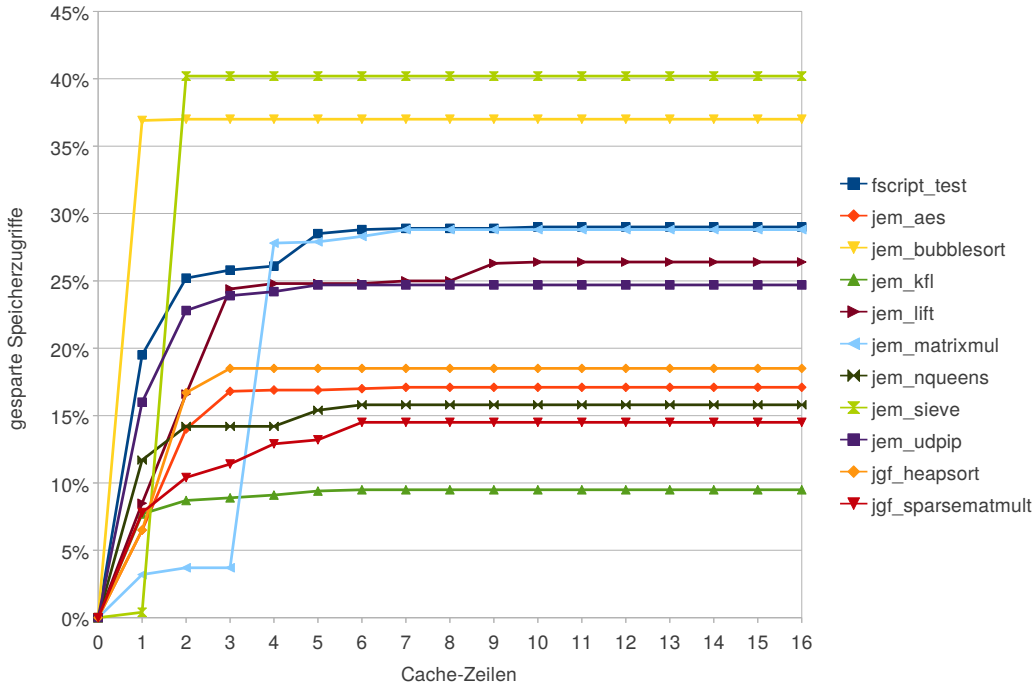


Abbildung A.5: Offsets -3 bis 4, Burst-Zugriff - Eingesparte Speicherzugriffe

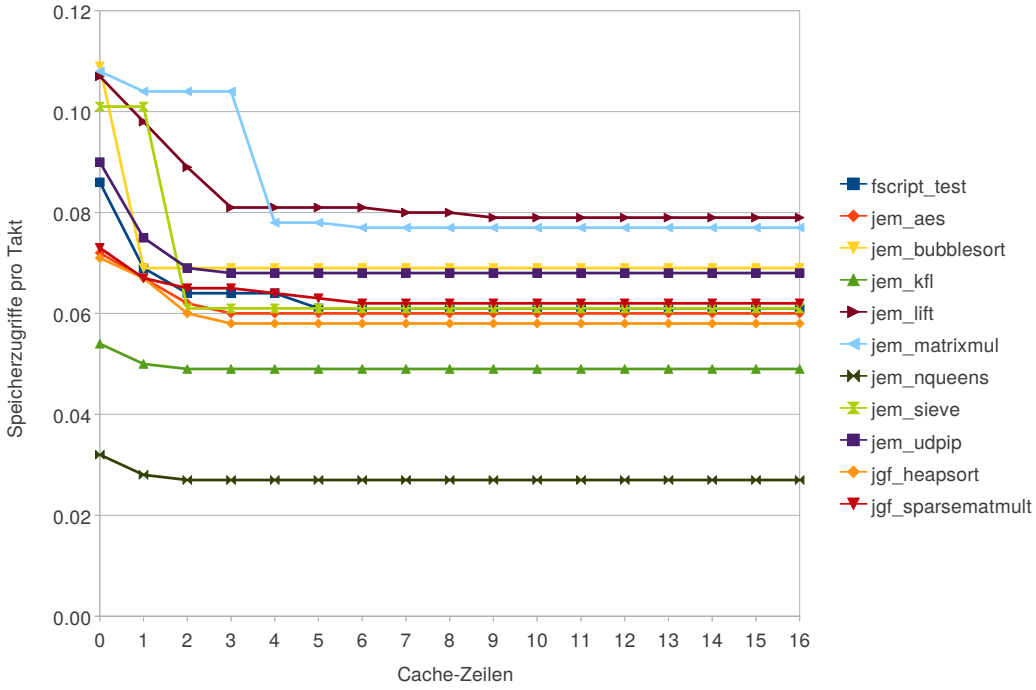


Abbildung A.6: Offsets -3 bis 4, Burst-Zugriff - Speicherzugriffe pro Takt

A.3 Offsets -2 bis 1, Einzel-Zugriff, volle Cache-Zeile

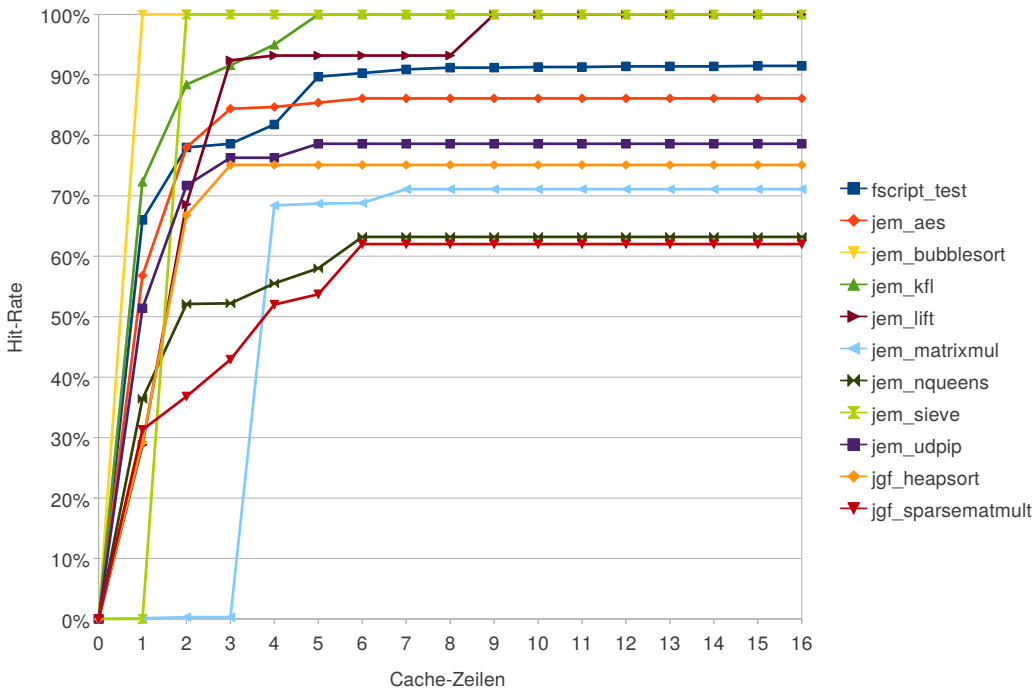


Abbildung A.7: Offsets -2 bis 1, Einzel-Zugriff, volle Cache-Zeile - Hit-Rate

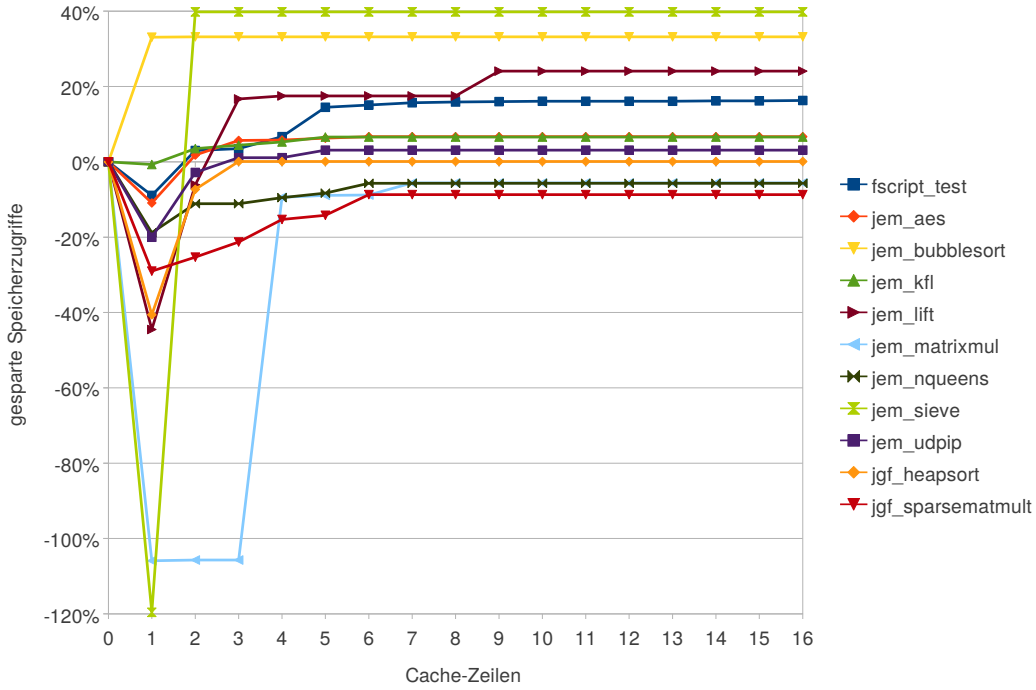


Abbildung A.8: Offsets -2 bis 1, Einzel-Zugriff, volle Cache-Zeile - Eingesparte Speicherzugriffe

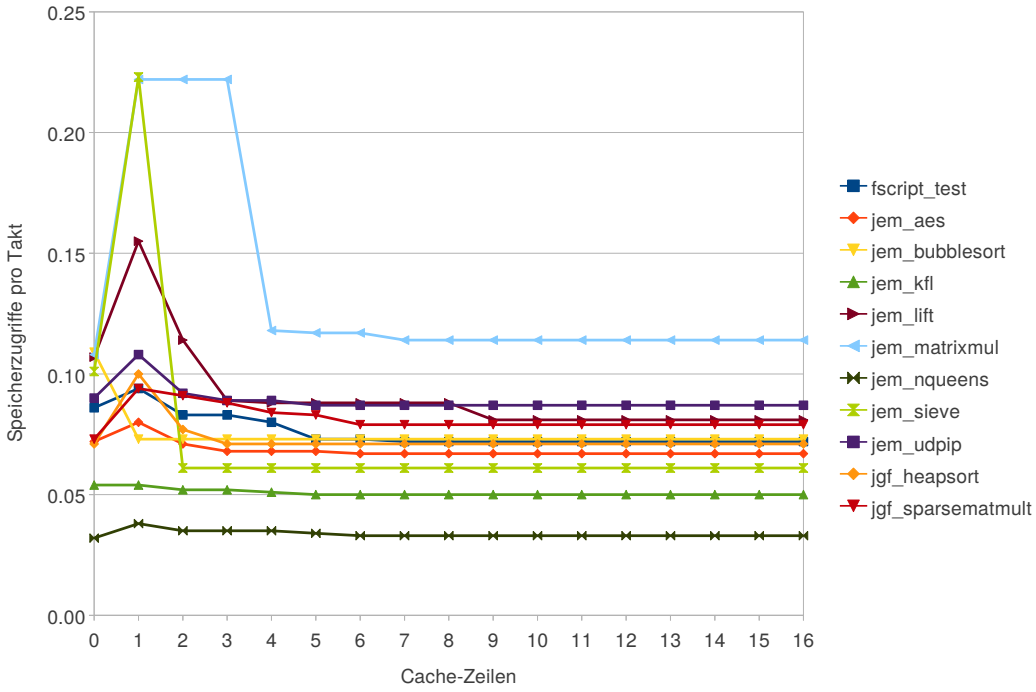


Abbildung A.9: Offsets -2 bis 1, Einzel-Zugriff, volle Cache-Zeile - Speicherzugriffe pro Takt

A.4 Offsets -2 und 1, Einzel-Zugriff, volle Cache-Zeile

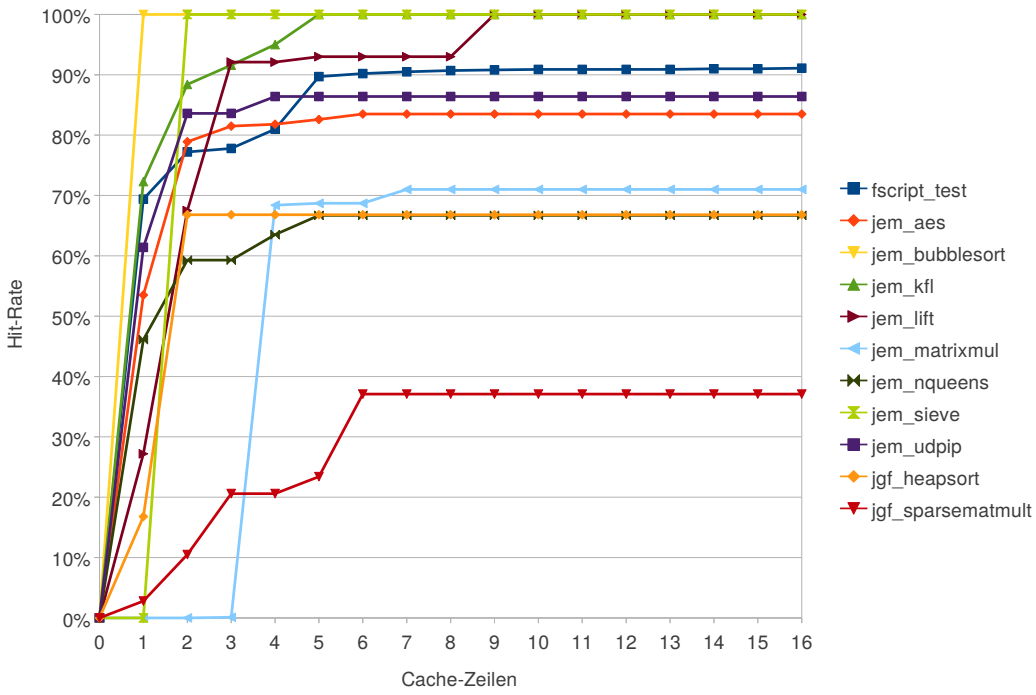


Abbildung A.10: Offsets -2 und 1, Einzel-Zugriff, volle Cache-Zeile - Hit-Rate

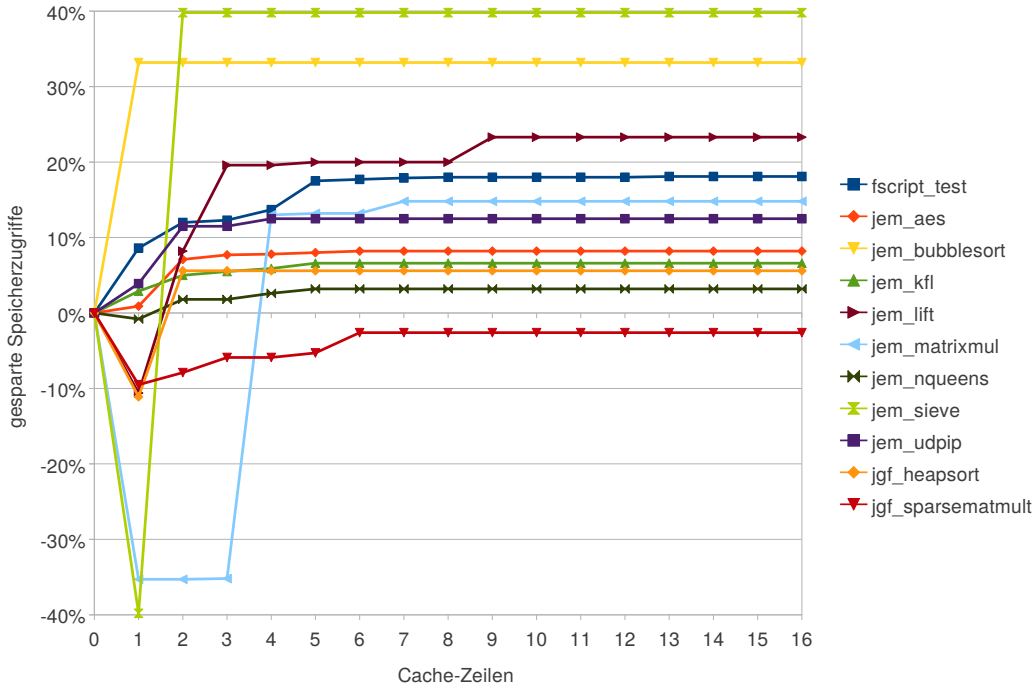


Abbildung A.11: Offsets -2 und 1, Einzel-Zugriff, volle Cache-Zeile - Eingesparte Speicherzugriffe

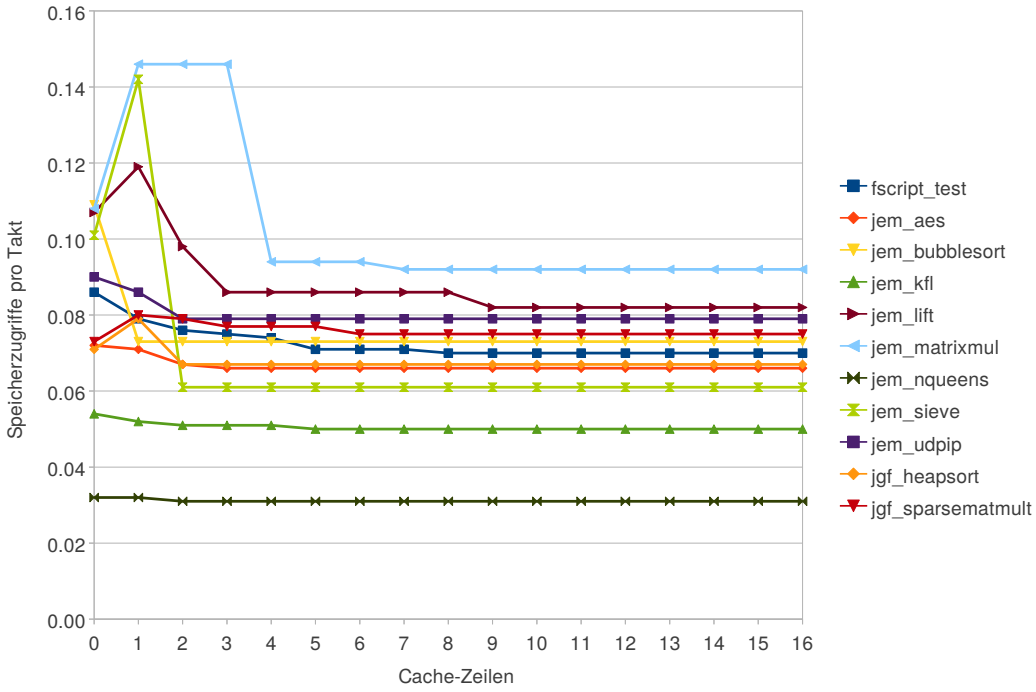


Abbildung A.12: Offsets -2 und 1, Einzel-Zugriff, volle Cache-Zeile - Speicherzugriffe pro Takt

A.5 Offsets -2 bis 1, Einzel-Zugriff, getrennte Valid-Bits

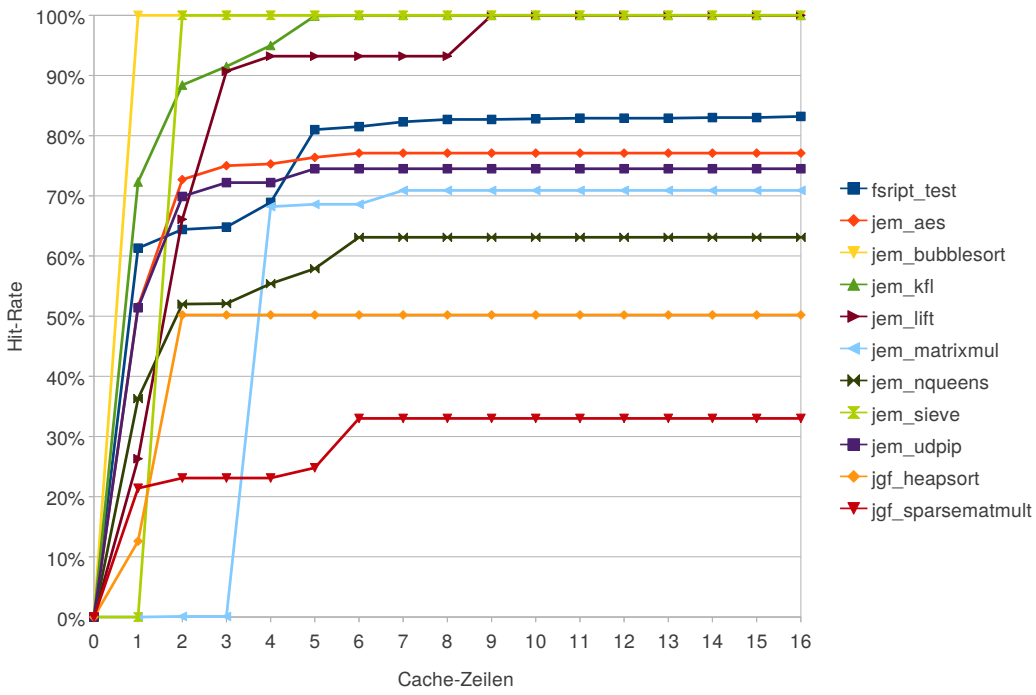


Abbildung A.13: Offsets -2 bis 1, Einzel-Zugriff, getrennte Valid-Bits - Hit-Rate

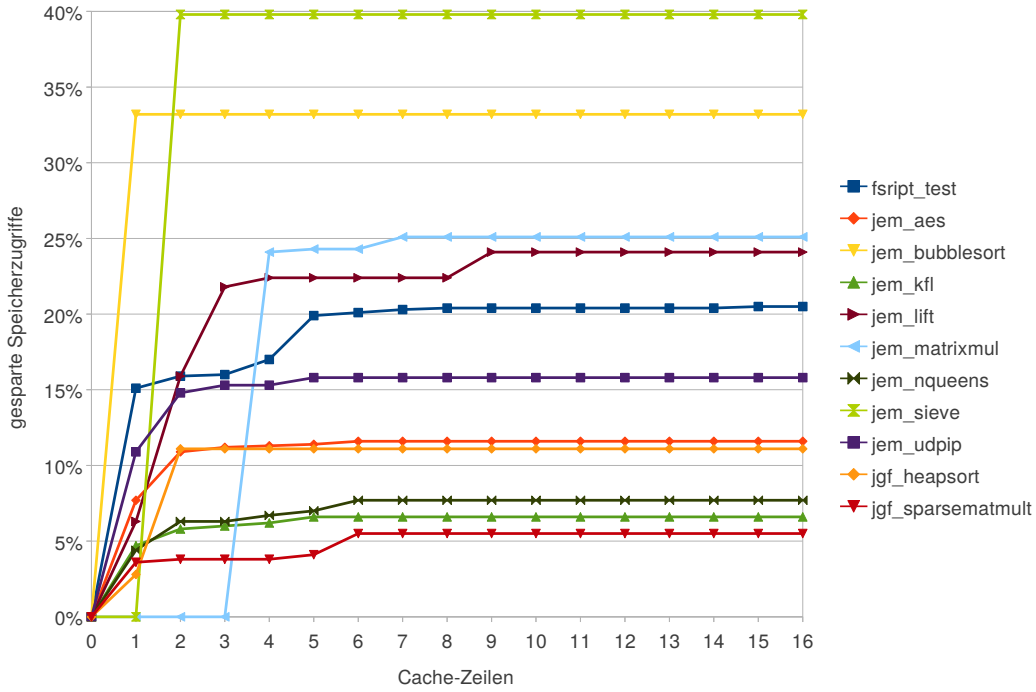


Abbildung A.14: Offsets -2 bis 1, Einzel-Zugriff, getrennte Valid-Bits - Eingesparte Speicherzugriffe

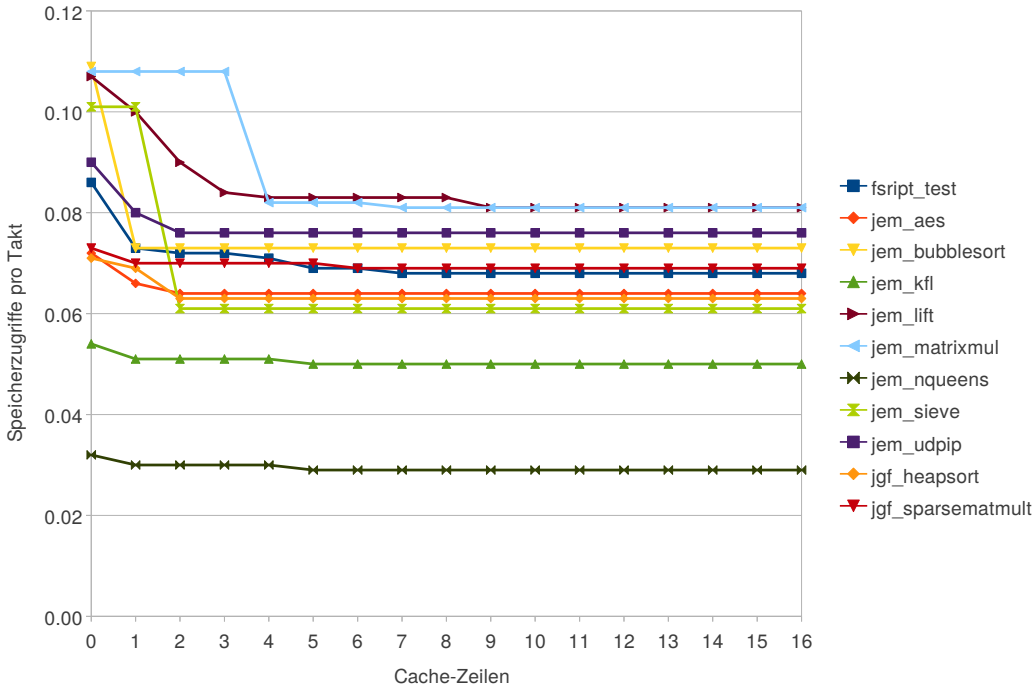


Abbildung A.15: Offsets -2 bis 1, Einzel-Zugriff, getrennte Valid-Bits - Speicherzugriffe pro Takt

A.6 Offsets -2 und 1, Einzel-Zugriff, getrennte Valid-Bits

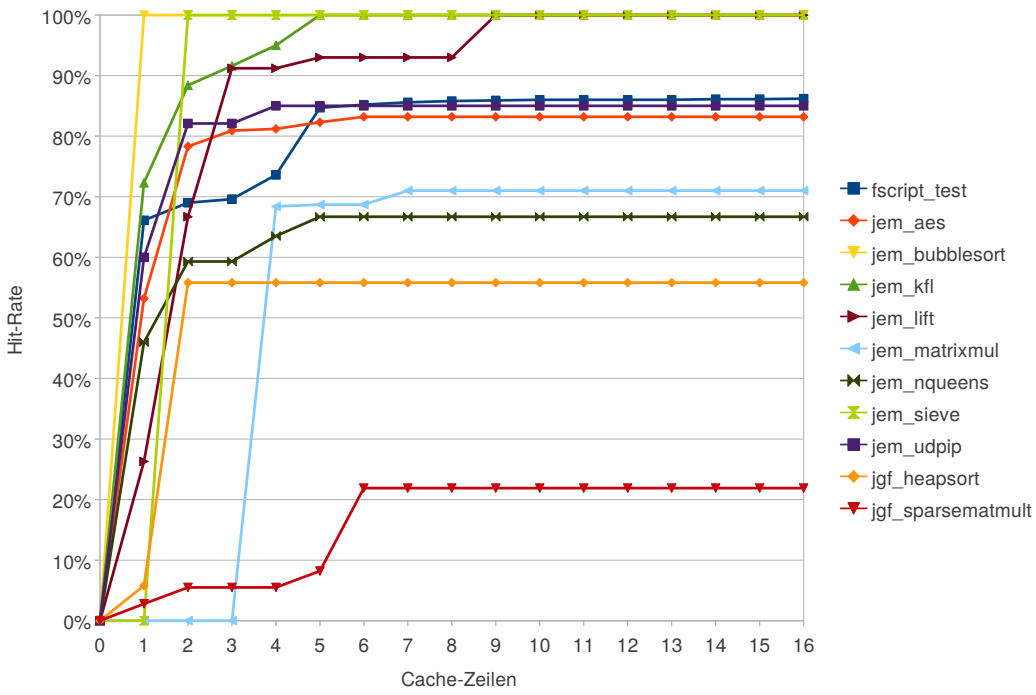


Abbildung A.16: Offsets -2 und 1, Einzel-Zugriff, getrennte Valid-Bits - Hit-Rate

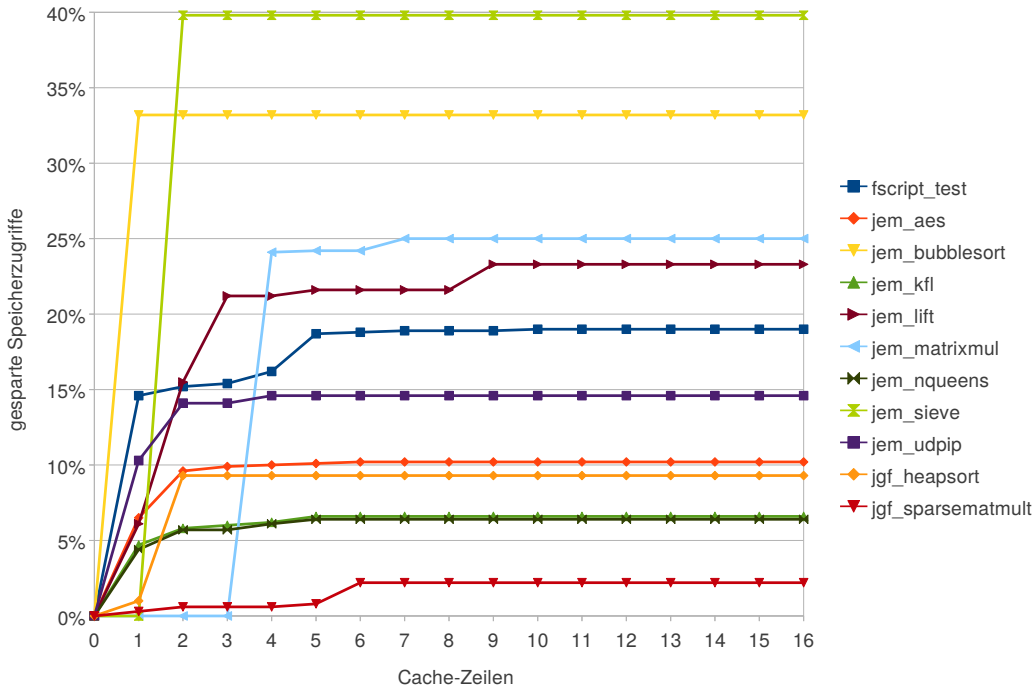


Abbildung A.17: Offsets -2 und 1, Einzel-Zugriff, getrennte Valid-Bits - Eingesparte Speicherzugriffe

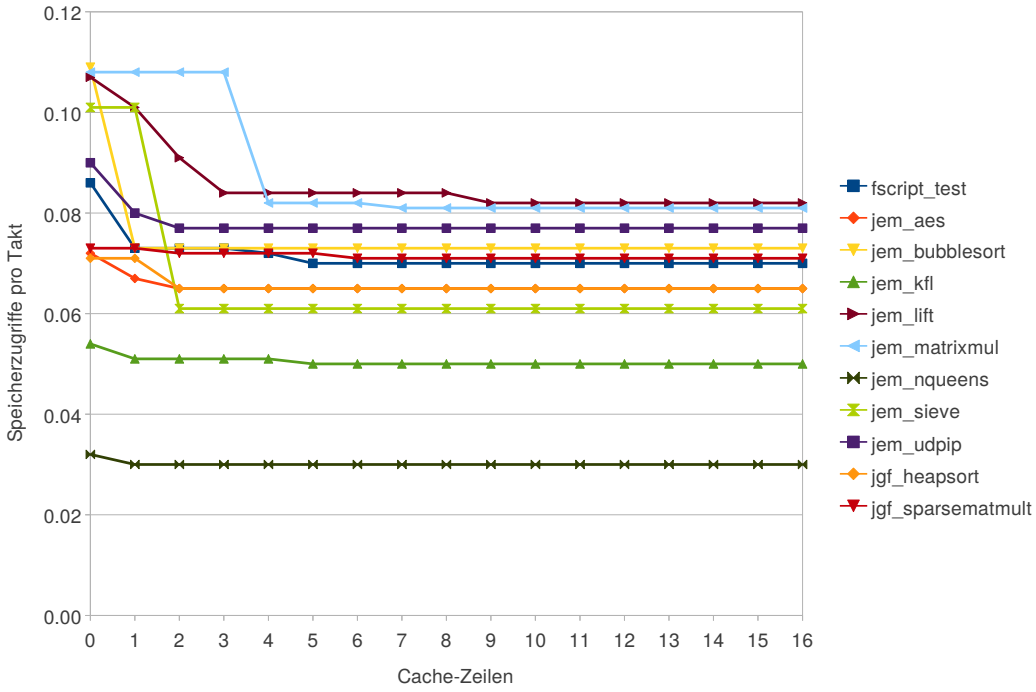


Abbildung A.18: Offsets -2 und 1, Einzel-Zugriff, getrennte Valid-Bits - Speicherzugriffe pro Takt

A.7 Offset 1, Einzel-Zugriff, getrennte Valid-Bits

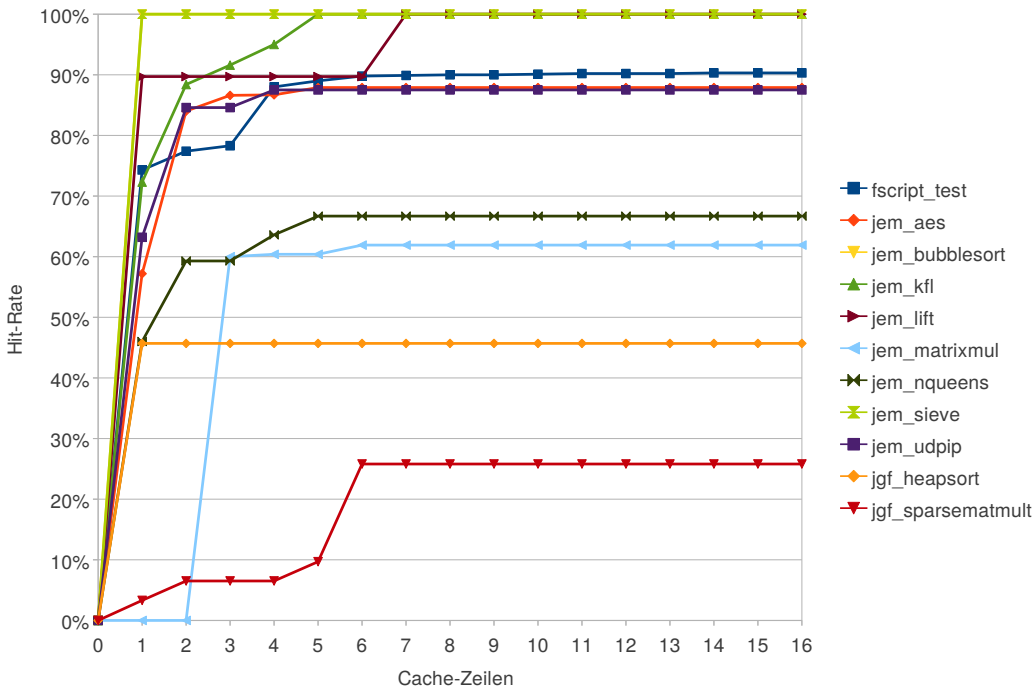


Abbildung A.19: Offset 1, Einzel-Zugriff, getrennte Valid-Bits - Hit-Rate

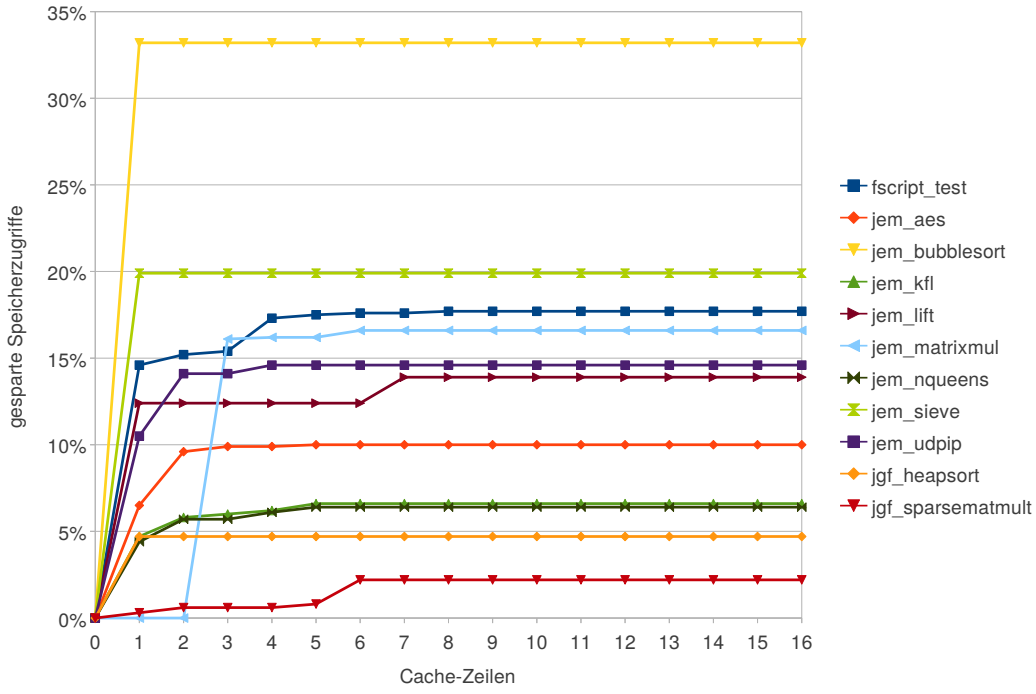


Abbildung A.20: Offset 1, Einzel-Zugriff, getrennte Valid-Bits - Eingesparte Speicherzugriffe

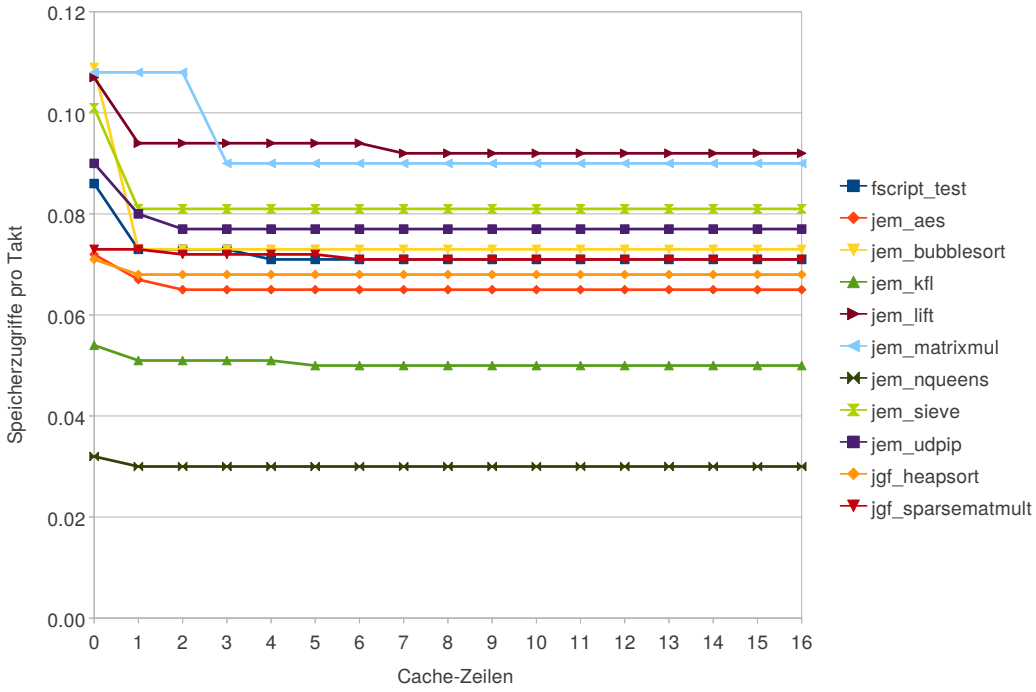


Abbildung A.21: Offset 1, Einzel-Zugriff, getrennte Valid-Bits - Speicherzugriffe pro Takt

A.8 Translation Lookaside Buffer

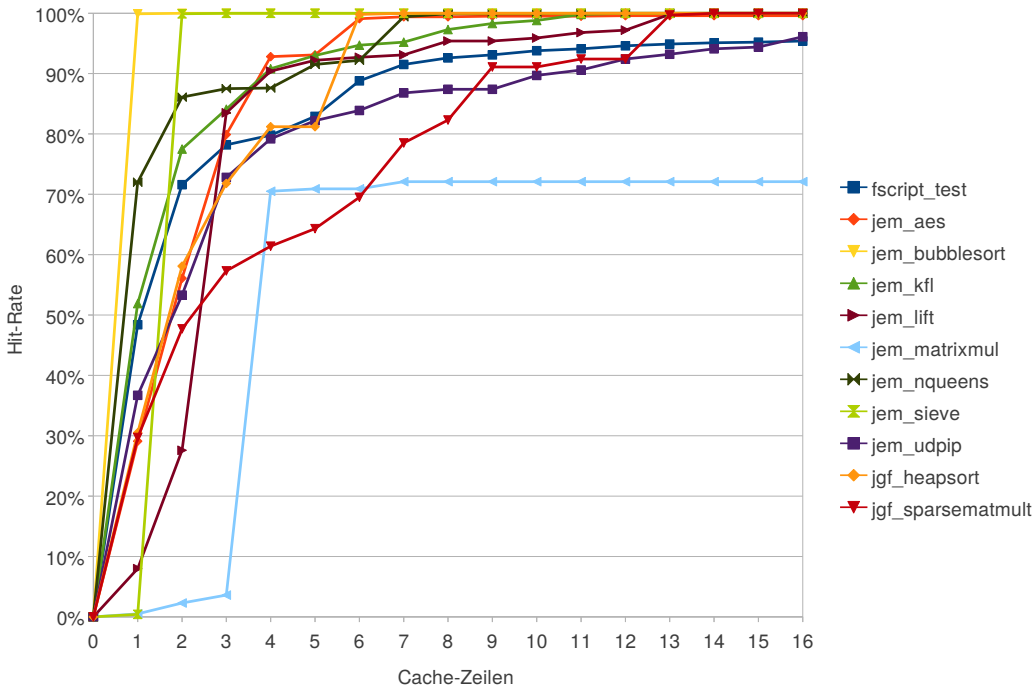


Abbildung A.22: Translation Lookaside Buffer - Hit-Rate

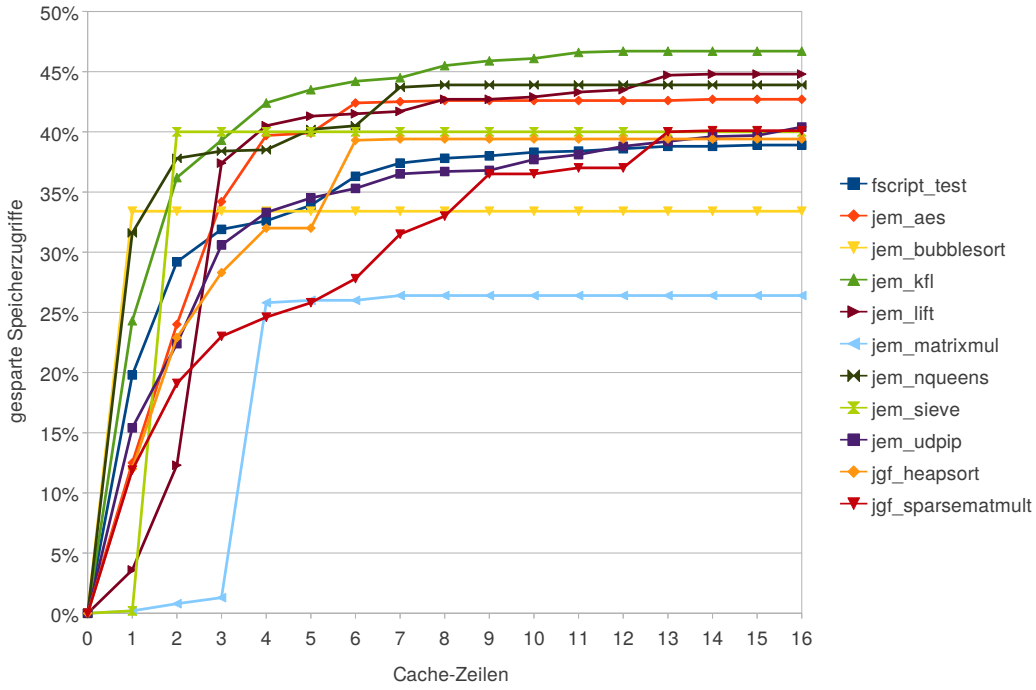


Abbildung A.23: Translation Lookaside Buffer - Eingesparte Speicherzugriffe

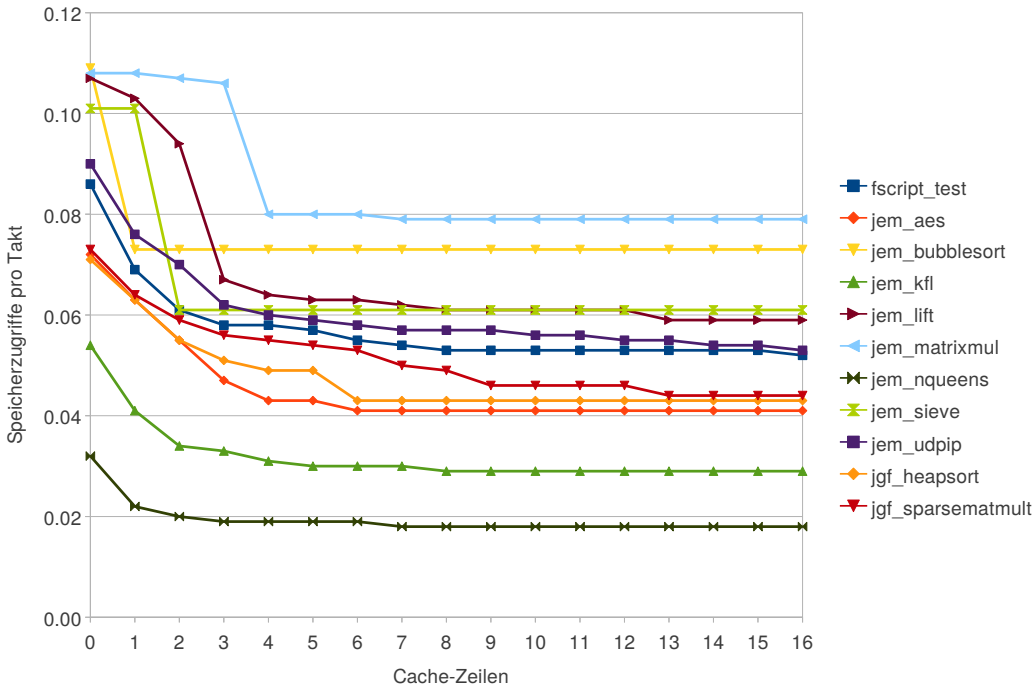


Abbildung A.24: Translation Lookaside Buffer - Speicherzugriffe pro Takt

A.9 TLB + Offsets -2 und 1, getrennte Valid-Bits

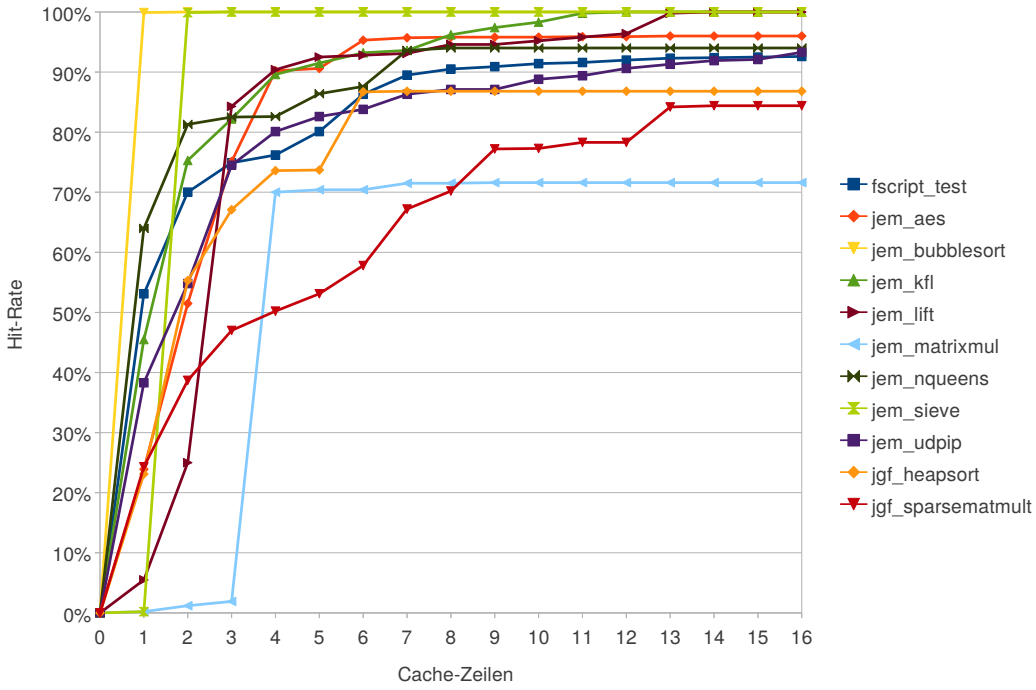


Abbildung A.25: TLB + Offsets -2 und 1, getrennte Valid-Bits - Hit-Rate

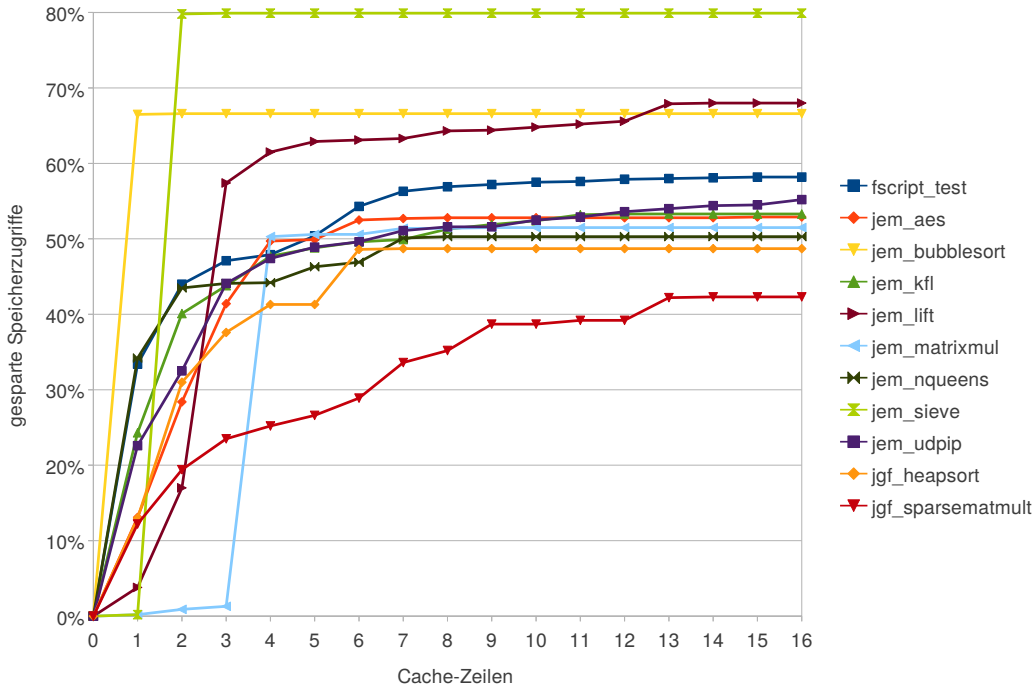


Abbildung A.26: TLB + Offsets -2 und 1, getrennte Valid-Bits - Eingesparte Speicherzugriffe

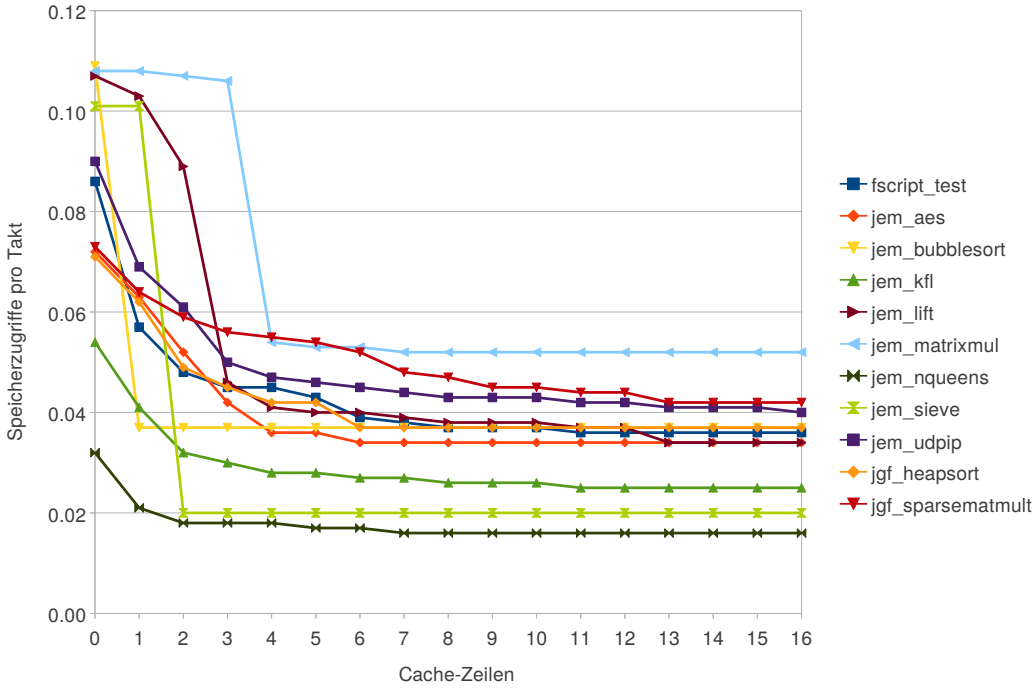


Abbildung A.27: TLB + Offsets -2 und 1, getrennte Valid-Bits - Speicherzugriffe pro Takt

Anhang B

Messergebnisse

B.1 Rechenleistung

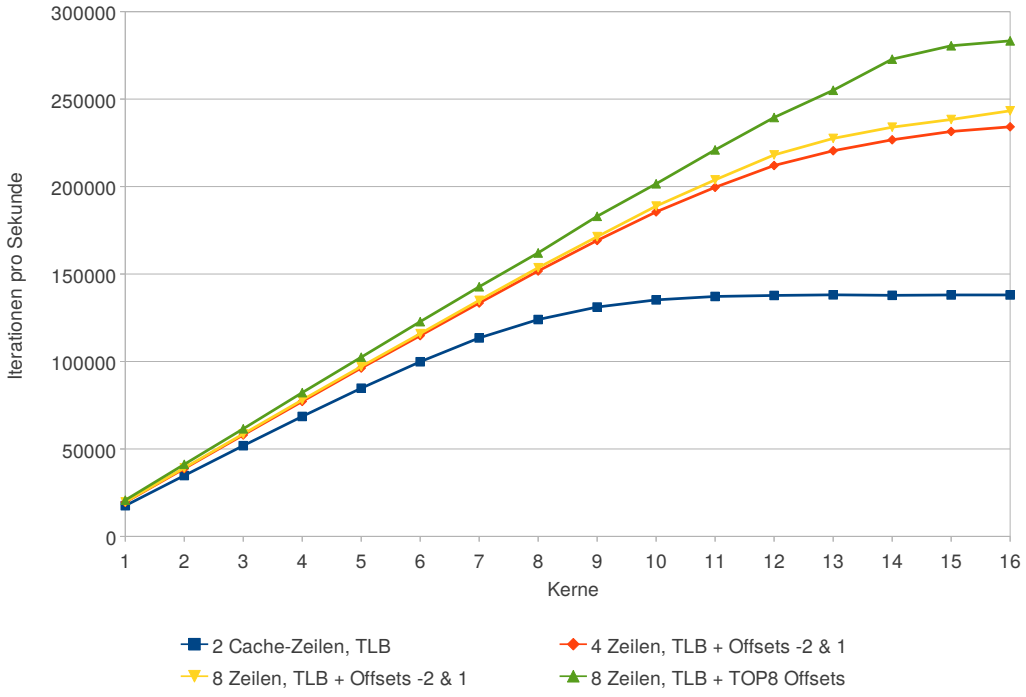


Abbildung B.1: Rechenleistung für den Benchmark JEM Lift

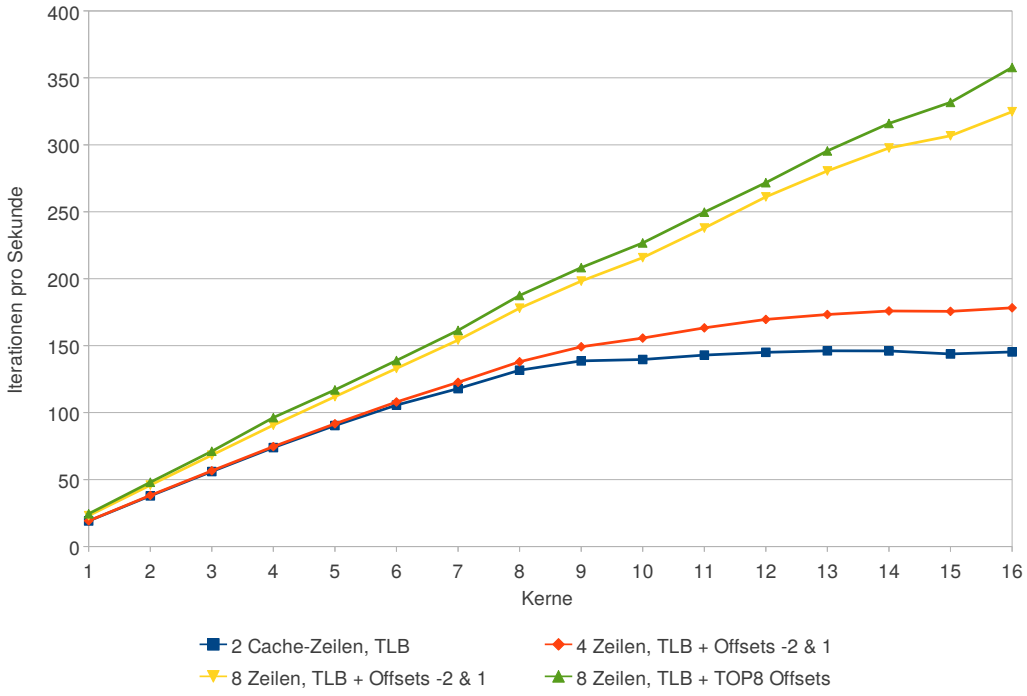


Abbildung B.2: Rechenleistung für den Benchmark JGF SparseMatMult

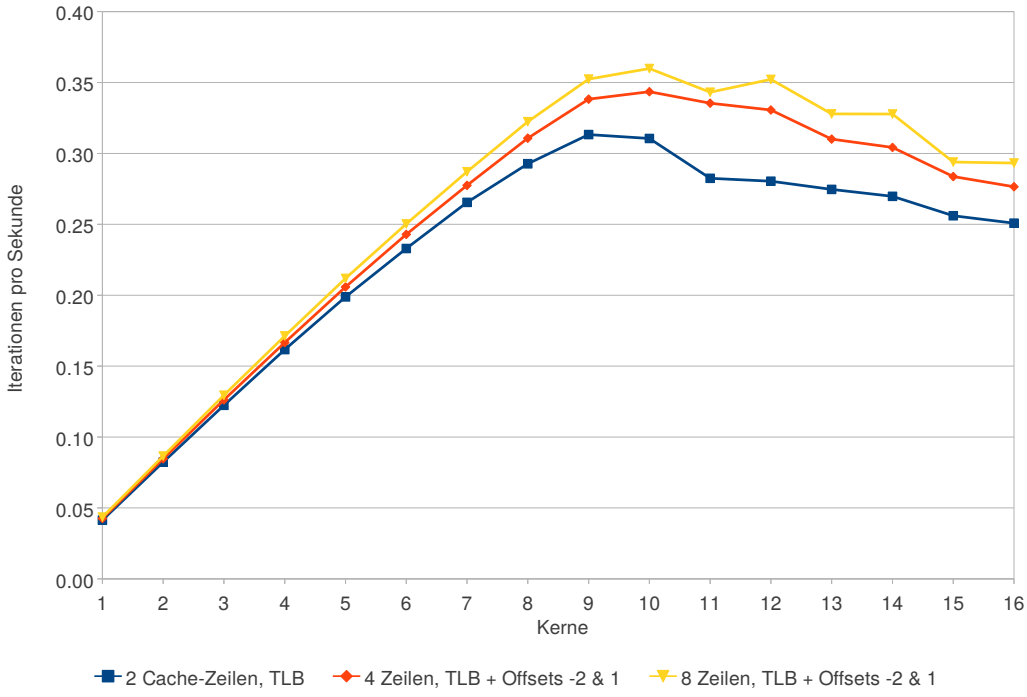


Abbildung B.3: Rechenleistung für den Benchmark FScript, normale GC-Priorität

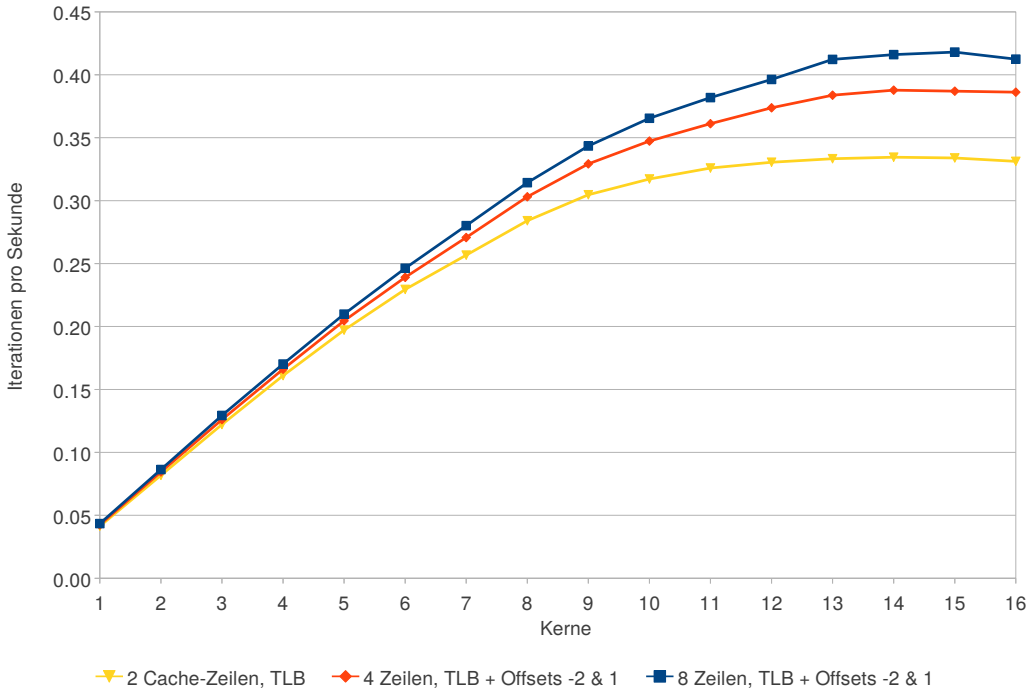


Abbildung B.4: Rechenleistung für den Benchmark FScript, hohe GC-Priorität

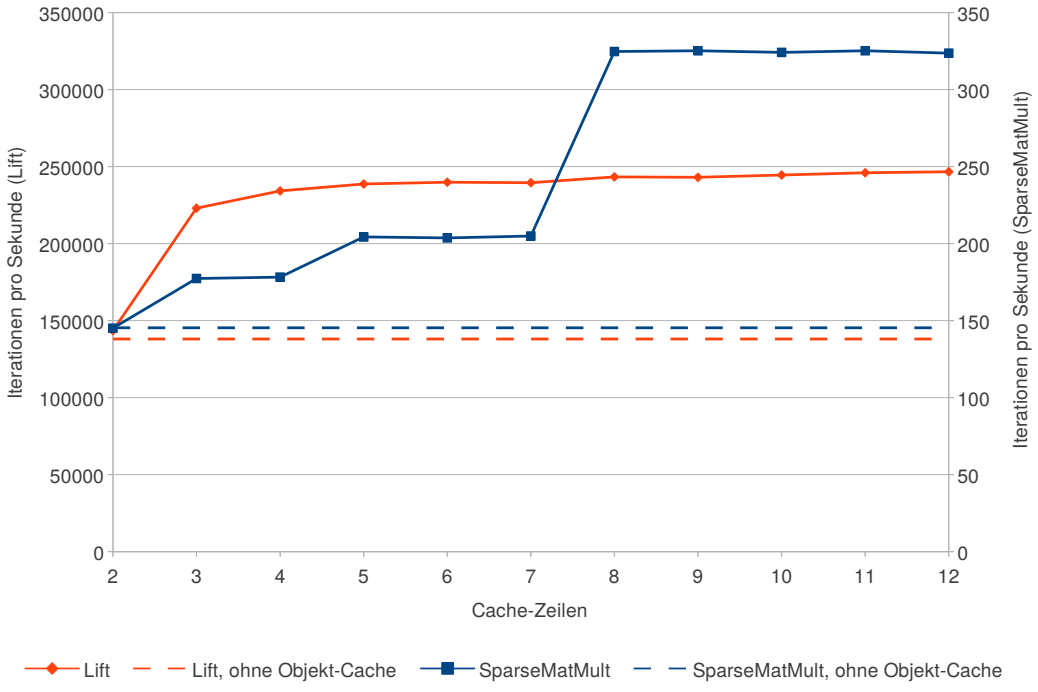


Abbildung B.5: Rechenleistung auf 16 Kernen in Abhängigkeit der Anzahl der Cache-Zeilen

B.2 Ressourcenbedarf

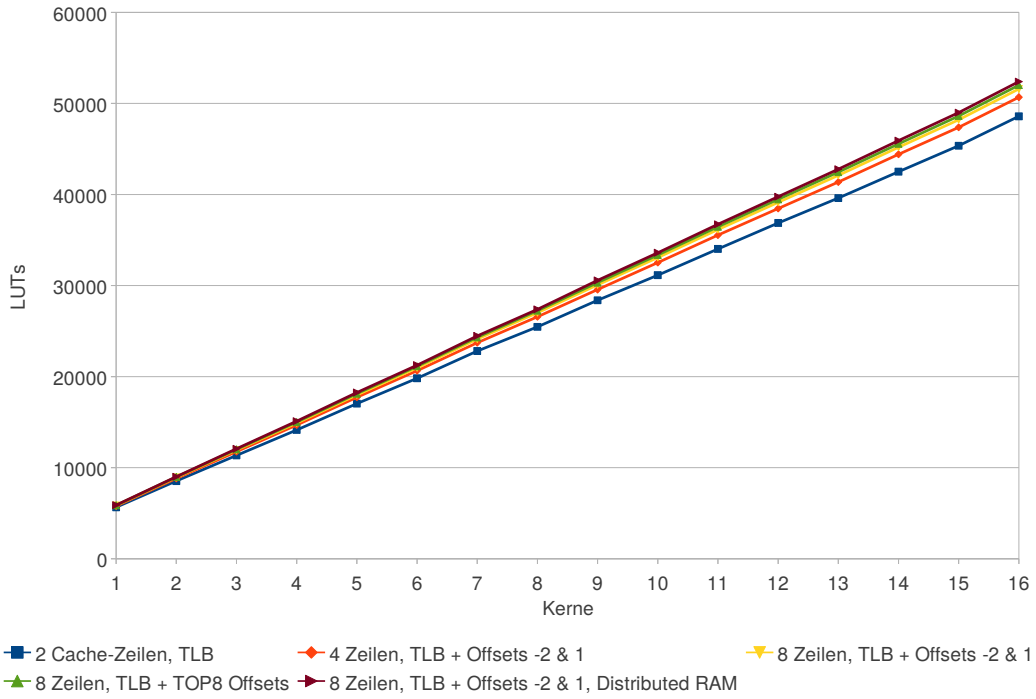


Abbildung B.6: Bedarf an Lookup-Tabellen in Abhängigkeit der Anzahl der Kerne

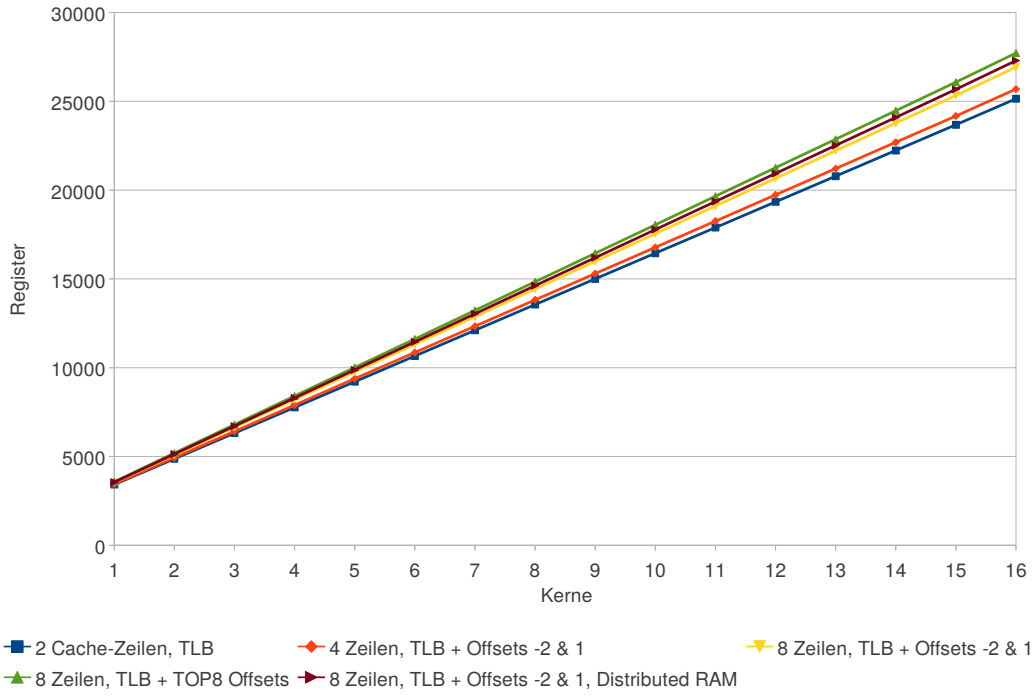


Abbildung B.7: Bedarf an Registern in Abhängigkeit der Anzahl der Kerne

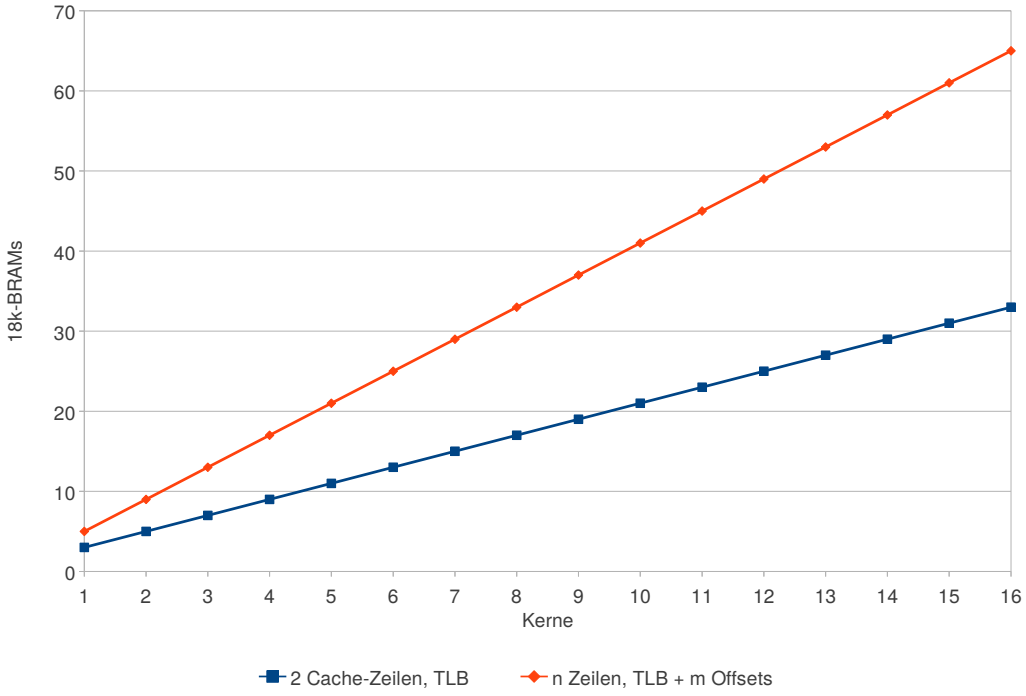


Abbildung B.8: Bedarf an 18kBit-Block-RAM in Abhängigkeit der Anzahl der Kerne

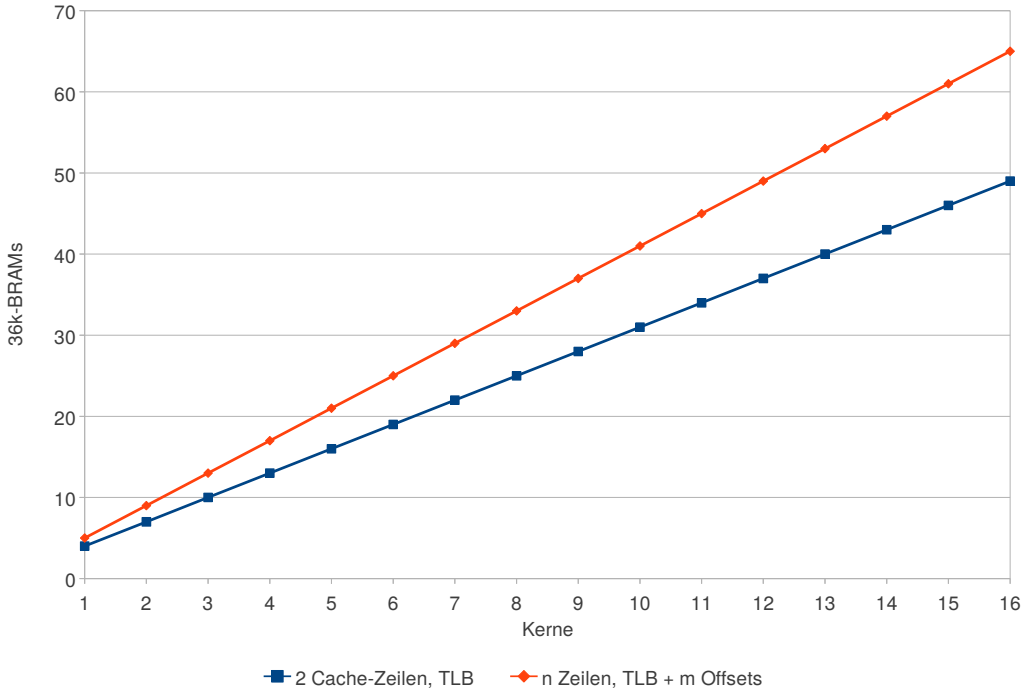


Abbildung B.9: Bedarf an 36kBit-Block-RAM in Abhängigkeit der Anzahl der Kerne

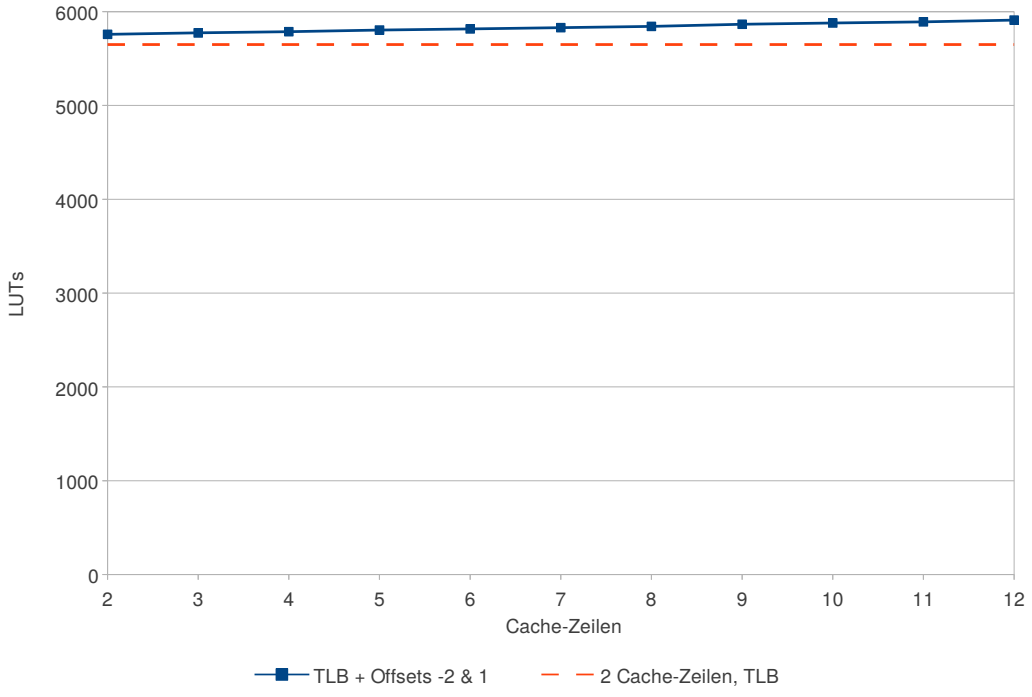


Abbildung B.10: Bedarf an Lookup-Tabellen in Abhängigkeit der Anzahl der Cache-Zeilen

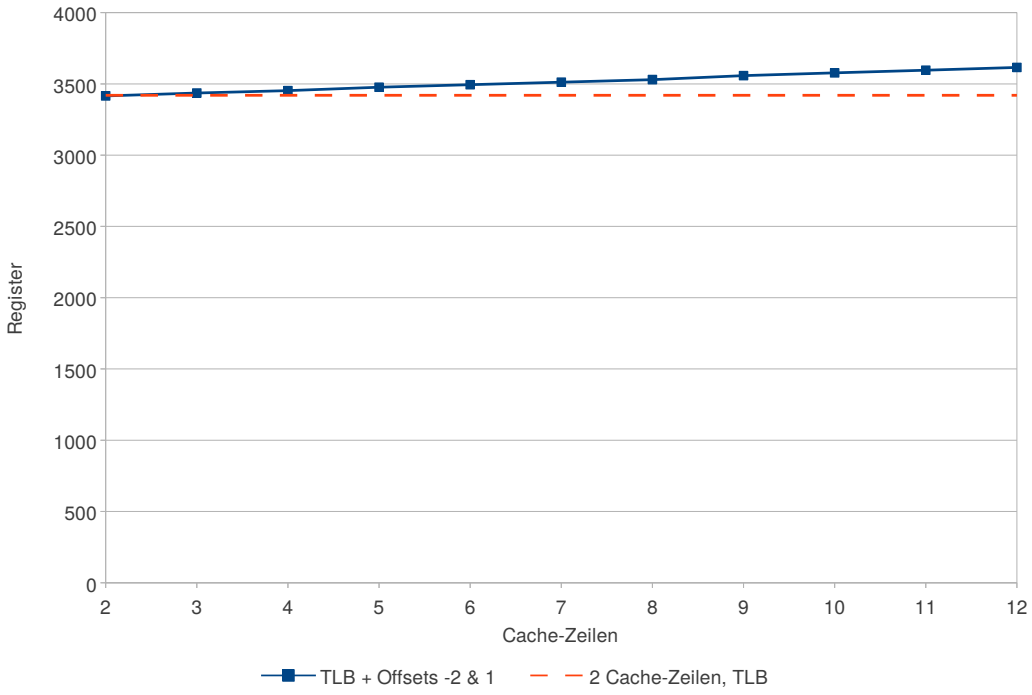


Abbildung B.11: Bedarf an Registern in Abhängigkeit der Anzahl der Cache-Zeilen

Literaturverzeichnis

[Ale09] ALEX, Stefan: *Anforderungsanalyse für Daten-Caches für die SHAP-Mikroarchitektur*, Technische Universität Dresden, Großer Beleg, Feb. 2009

[Ale10] ALEX, Stefan: *Entwurf und Implementierung einer parametrierbaren Trace-Hardware am Beispiel der SHAP-Mikroarchitektur*, Technische Universität Dresden, Diplomarbeit, Jan. 2010

[BSW+99] BULL, J. M. ; SMITH, L. A. ; WESTHEAD, M. D. ; HENTY, D. S. ; DAVEY, R. A.: A methodology for benchmarking Java Grande applications. In: *Proceedings of the ACM 1999 conference on Java Grande (JAVA 1999)*. New York, NY, USA : ACM, 1999. – ISBN 1–58113–161–5, S. 81–88

[CG93] CHANG, J.M. ; GEHRINGER, E.F.: Evaluation of an object-caching coprocessor design for object-oriented systems. In: *IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD 1993)*, 1993, S. 132–139

[Cof12a] COFFEY, Neil: *Volatile arrays in Java*. Jan. 2012. – http://www.javamex.com/tutorials/volatile_arrays.shtml

[Cof12b] COFFEY, Neil: *The volatile keyword in Java 5*. Jan. 2012. – http://www.javamex.com/tutorials/synchronization_volatile_java_5.shtml

[FSc12] *FScript*. Jan. 2012. – http://fscript.sourceforge.net/

[Gro02] GROSSMAN, J. P.: A Systolic Array for Implementing LRU Replacement. In: *Project Aries Technical Memo ARIES-TM-18*. Cambridge, MA, USA : AI Lab, M.I.T., Mrz. 2002

[HCT10] HU, Guang ; CHAI, Zhilei ; TU, Shiliang: Memory access mechanism in embedded real-time Java processor. In: *The 2nd International Conference on Computer and Automation Engineering (ICCAE 2010)* Bd. 5, 2010, S. 786–790

[HP07] HENNESSY, John L. ; PATTERSON, David A.: *Computer Architecture: A Quantitative Approach*. 4th edition. San Francisco, CA 94111, USA : Morgan Kaufmann Publishers, 2007. – ISBN 978–0–12–370490–0

[HPS10] HUBER, Benedikt ; PUFFITSCH, Wolfgang ; SCHOEBERL, Martin: WCET Driven Design Space Exploration of an Object Cache. In: *Proceedings of the 8th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2010)*, ACM, 2010, S. 26–35

[JEM12] *JemBench*. Jan. 2012. – http://sourceforge.net/projects/jembench/

[JGF12] *The Java Grande Forum Benchmark Suite*. Jan. 2012. – http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html

[JSQ07] JI, Weixing ; SHI, Feng ; QIAO, Baojun: A self-maintained memory module supporting DMM. In: *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems (CASES 2007)*. New York, NY, USA : ACM, 2007. – ISBN 978–1–59593–826–8, S. 189–197

[LJW+09] LIU, Mengxiao ; JI, Weixing ; WANG, Zuo ; LI, Jiaxin ; PU, Xing: High Performance Memory Management for a Multi-core Architecture. In: *Ninth IEEE International Conference on Computer and Information Technology (CIT 2009)*. Xiamen, China : IEEE, Okt. 2009, S. 63–68

[LSK04] LIU, C. ; SIVASUBRAMANIAM, Anand ; KANDEMIR, M.: Organizing the last line of defense before hitting the memory wall for CMPs. In: *Software, IEE Proceedings*, 2004. – ISSN 1530–0897, S. 176–185

[LY99] LINDHOLM, Tim ; YELLIN, Frank: *The Java(TM) Virtual Machine Specification*. 2. Addison-Wesley Professional, Apr. 1999. – ISBN 201432943

[Man12] MANSON, Jeremy: *Volatile Arrays in Java*. Jan. 2012. – http://jeremymanson.blogspot.com/2009/06/volatile-arrays-in-java.html

[Olu09] OLUNCZEK, Andrej: *ASIC-Synthese der SHAP-Mikroarchitektur*, Technische Universität Dresden, Großer Beleg, Mrz. 2009

[PH09] PATTERSON, David A. ; HENNESSY, John L.: *Computer Organization and Design: The Hardware/Software Interface*. 4th edition. Burlington, MA 01803, USA : Morgan Kaufmann Publishers, 2009. – ISBN 978–0–12–374493–7

[PZR07] PREUSSER, Thomas B. ; ZABEL, Martin ; REICHEL, Peter: The SHAP Microarchitecture and Java Virtual Machine / Fakultät Informatik, Technische Universität Dresden. 2007 (TUD-FI07-0). – Forschungsbericht. – ISSN 1430–211X

[PZS07] PREUSSER, Thomas B. ; ZABEL, Martin ; SPALLEK, Rainer G.: Bump-pointer method caching for embedded Java processors. In: *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems (JTRES 2007)*. New York, NY, USA : ACM, 2007. – ISBN 978–1–59593–813–8, S. 206–210

[Rei07] REICHEL, Peter: *Entwurf und Implementierung verschiedener Garbage-Collector-Strategien für die Java-Plattform SHAP*, Technische Universität Dresden, Großer Beleg, Okt. 2007

[Rei08] REICHEL, Peter: *Realisierung einer integrierten Speicherverwaltung mit der Unterstützung schwacher Referenzen für den Prozessor SHAP*, Technische Universität Dresden, Diplomarbeit, Aug. 2008

[Sch04] SCHOEBERL, Martin: A Time Predictable Instruction Cache for a Java Processor. In: *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)* Bd. 3292, Springer, 2004, S. 371–382

[Sch09] SCHOEBERL, Martin: Time-predictable Cache Organization. In: *Proceedings of the First International Workshop on Software Technologies for Future Dependable Distributed Systems (STFSSD 2009)*, IEEE Computer Society, 2009, S. 11–16

[Sch11] SCHOEBERL, Martin: A Time-predictable Object Cache. In: *Proceedings of the 14th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2011)*, IEEE Computer Society, 2011, S. 99–105

[SLF04] SHING, Yu W. ; LI, Richard ; FONG, Anthony S.: Hardware concurrent garbage collection for short-lived objects in an object-oriented processor. In: *International Conference on Electrical, Electronic and Computer Engineering (ICEEC 2004)*, 2004, S. 285–288

[SMV04] SUDARSHAN, T. S. B. ; MIR, Rahil A. ; VIJAYALAKSHMI, S.: Highly Efficient LRU Implementations for High Associativity Cache Memory. In: *Proceedings of 12th IEEE International Conference on Advanced Computing and Communications*, 2004, S. 87–95

[SP05] SANTTI, T. ; PLOSILA, J.: Architecture for an advanced Java coprocessor. In: *International Symposium on Signals, Circuits and Systems (ISSCS 2005)* Bd. 2, 2005, S. 501–504

[SPH09] SCHOEBERL, Martin ; PUFFITSCH, Wolfgang ; HUBER, Benedikt: Towards Time-Predictable Data Caches for Chip-Multiprocessors. In: *Software Technologies for Embedded and Ubiquitous Systems, 7th IFIP WG 10.2 International Workshop (SEUS 2009)*, 2009, S. 180–191

[SPU10] SCHOEBERL, Martin ; PREUSSER, Thomas B. ; UHRIG, Sascha: The embedded Java benchmark suite JemBench. In: *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2010)*. New York, NY, USA : ACM, Aug. 2010. – ISBN 978–1–4503–0122–0, S. 120–127

[TSP10] TYYSTJÄRVI, J. ; SÄNTTI, T. ; PLOSILA, J.: Heap access optimizations for a hardware-accelerated Java virtual machine. In: *International Symposium on System on Chip (SoC)*, 2010, S. 125–128

[Ull06] ULLENBOOM, Christian: *Java ist auch eine Insel*. 5., aktualisierte und erweiterte Auflage. Bonn : Galileo Computing, 2006. – ISBN 3–89842–747–1

[UW07] UHRIG, Sascha ; WIESE, Jörg: jamuth: an IP processor core for embedded Java real-time systems. In: *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems (JTRES 2007)*. New York, NY, USA : ACM, 2007. – ISBN 978–1–59593–813–8, S. 230–237

[Wik12] WIKIPEDIA: *Cache algorithms*. Jan. 2012. – http://en.wikipedia.org/wiki/Cache_algorithms

[Xil10] Xilinx: *Virtex-5 FPGA User Guide*. Mai 2010

[ZRS08] ZABEL, Martin ; REICHEL, Peter ; SPALLEK, Rainer G.: Multi-Port-Speichermanager für die Java-Plattform SHAP. In: *Dresdner Arbeitstagung Schaltungs- und Systementwurf (DASS'2008)*. Zeunerstr. 38, D-01069 Dresden : Fraunhofer-Institut für Integrierte Schaltungen, Institutst, May. 2008. – ISBN 3–9810287–2–4, S. 143–147

[ZS09] ZABEL, Martin ; SPALLEK, Rainer G.: SHAP - Scalable Multi-Core Java Bytecode Processor / Fakultät Informatik, Technische Universität Dresden. 2009 (TUD-FI09-13). – Forschungsbericht. – ISSN 1430–211X