Effiziente Mehrkernarchitektur für eingebettete Java-Bytecode-Prozessoren

Dissertation

zur Erlangung des akademischen Grades Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von
Dipl.-Inf. Martin Zabel
geboren am 03.01.1981 in Dessau

eingereicht am 26.07.2011
verteidigt am 16.12.2011

Betreuer:	Prof. Dr.-Ing. habil. Rainer G. Spallek, TU Dresden
Zweitgutachter:	Prof. Dr.-Ing. habil. Djamshid Tavangarian, Uni Rostock
Fachreferent:	Prof. Dr.-Ing. habil. Martin Wollschlaeger, TU Dresden

Selbstständigkeitserklärung

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

Dresden,

Dipl.-Inf. Martin Zabel

Vorwort

Die Java-Plattform bietet eine Reihe von Vorteilen für die schnelle Entwicklung komplexer Software, darunter: durchgängige Objektorientierung, automatische Speicherverwaltung, Portabilität und Sicherheit durch Definition einer virtuellen Maschine (JVM) sowie insbesondere eine umfangreiche Unterstützung für die Parallelisierung von Programmen auf Thread-Ebene. Die Ausführung des Java-Bytecodes auf einer konkreten Zielplattform ist häufig entweder durch eine relativ langsame Interpretation oder durch eine ressourcenintensive Übersetzung in einen prozessorspezifischen Maschinencode gekennzeichnet (z. B. Just-in-Time-Compiler). Eine Alternative für eingebettete Systeme mit begrenzten Ressourcen stellen Java-(Bytecode-)Prozessoren dar, die den Bytecode als nativen Befehlssatz unterstützen.

Die Weiterentwicklung klassischer Prozessoren hat im Allgemeinen gezeigt, dass durch Integration mehrerer Prozessorkerne auf einem Chip die Verarbeitungsleistung erheblich gesteigert werden kann. Die zentrale Fragestellung dieser Arbeit lautet daher:

> Wie ist die Architektur eines Java-Mehrkernprozessors zu gestalten,
> damit die auf Thread-Ebene ohnehin vorhandene Parallelität eines
> Java-Programms effizient zur Leistungssteigerung genutzt werden kann?

In diesem Zusammenhang sind auch Schnittstellen für Test und Diagnose zu diskutieren.

Im Detail wird die Realisierbarkeit von Mehrkern-Java-Bytecode-Architekturen im Allgemeinen analysiert. Zu berücksichtigen sind dabei speziell die Eigenschaften der JVM sowie die Anforderungen an Test und Diagnose. Daran anschließend wird ein Java-Mehrkernprozessor auf Basis der SHAP-Plattform diskutiert. Der gewählte Lösungsansatz wird anhand einer prototypischen Implementierung hinsichtlich Skalierbarkeit, Verarbeitungsleistung, Effizienz und Funktionalität bewertet. Dabei kommt insbesondere eine Trace-Architektur als Test- und Diagnosewerkzeug zum Einsatz.

Es wird nachgewiesen, dass eine hohe Leistungssteigerung bei nur geringem zusätzlichen Hardwareaufwand erzielt werden kann. Der gewählte Ansatz weist zudem eine gute Skalierbarkeit und Effizienz auf. Ebenso wird eine höhere Leistung gegenüber anderen bekannten Arbeiten nachgewiesen. Damit wird die zentrale Fragestellung positiv beantwortet.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

µP Mikroprozessor

AGU engl.: address generation unit

API engl.: application programming interface

CC-NUMA engl.: cache coherent non-uniform memory access

CDC engl.: Connected Device Configuration

CLDC engl.: Connected Limited Device Configuration

DMA engl.: direct memory access

FPGA engl.: field programmable gate array

GC engl.: garbage collector

JavaEE engl.: Java Platform, Enterprise Edition

JavaME engl.: Java Platform, Micro Edition

JavaSE engl.: Java Platform, Standard Edition

JDK engl.: Java Development Kit

JPUs engl.: Java Processing Units

JRE engl.: Java Runtime Environment

JVM engl.: Java Virtual Machine

LE engl.: Logic Element

MFLOPS engl.: million floating-point operations per second

MIMD engl.: multiple instruction multiple data

MIPS engl.: million instructions per second

MISD engl.: multiple instruction single data

NoRMA engl.: no-remote memory access

NUMA engl.: non-uniform memory access

RISC engl.: reduced instruction-set processor

RTSJ engl.: Real-Time Specification for JAVA

SIMD engl.: single instruction multiple data

SISD engl.: single instruction single data

TLB engl.: translation look-aside buffer

UMA engl.: uniform memory access

VLIW engl.: very long instruction word

1 Einleitung

1.1 Motivation

Die Java-Plattform ist sehr gut zur Entwicklung komplexer Software geeignet. Sie bietet eine Reihe von Vorteilen:

- Die durchgehende Objektorientierung kapselt Daten und Operationen auf diesen Daten. Sie stellt auch die Techniken der Vererbung, Polymorphie und dynamischen Bindung zur Verfügung.
- Java-Programme werden in einen plattformunabhängigen Java-Bytecode für eine virtuelle Maschine übersetzt. Diese Maschine kann für viele Zielplattformen implementiert werden, sodass überall ein und derselbe Bytecode ausgeführt werden kann.
- Die automatische Speicherverwaltung verhindert Speicherlecks und die fälschliche Freigabe von Objekten.
- Typische Sicherheitsrisiken wie Pufferüberläufe können nicht auftreten, da Stack- und Heap-Speicher nicht frei adressierbar sind.
- Die Sprache bietet zusammen mit der Klassenbibliothek eine umfangreiche und einfach zu erlernende Unterstützung für die Parallelisierung von Programmen auf Thread-Ebene.
- Die Java-Plattform ist in verschiedenen Abstufungen standardisiert, die jeweils auf andere Geräteklassen — vom Handy bis zum Server — ausgerichtet und optimiert sind.

Die Ausführung des Java-Bytecodes auf einer konkreten Zielplattform ist häufig entweder durch eine relativ langsame Interpretation oder durch eine ressourcenintensive Übersetzung in einen prozessorspezifischen Maschinencode gekennzeichnet (z. B. Just-in-Time-Compiler). Eine Alternative für eingebettete Systeme mit begrenzten Ressourcen stellen Java-(Bytecode-)Prozessoren dar, die den Bytecode als nativen Befehlssatz unterstützen [Sch05]. Diese weisen außerdem weitere Vorteile auf:

- Die Ausführung von Java-Programmen kommt ohne einem darunter liegendem Betriebssystem aus.

- Der kompakte Java-Bytecode erlaubt kostengünstige Dimensionierungen des Programmspeichers.
- Mit Wegfall des Betriebsystems vereinfacht sich die Laufzeitanalyse für Echtzeitanwendungen.
- Eine Wiederverwendung von Java-Prozessoren für das Framework „ .Net “ von Microsoft ist ebenfalls denkbar.

Die Forschung an Java-Prozessoren begann circa 1997 mit der Vorstellung des „picoJava-I“ von Sun Microelectronics [OT97]. Seitdem wurde ein gutes Dutzend Java-Prozessoren mit einem Prozessorkern für allgemeine sowie spezielle Anwendungsfelder entwickelt. Mit diesen Prozessoren wurden verschiedene Ansätze zur Gestaltung der Mikroarchitektur für Java-Bytecode untersucht. Dies umfasst insbesondere die Nutzung von Parallelität auf Bit- und Befehlsebene. Ebenso betrachtet wurde die Umsetzung objektorientierter Konzepte wie z. B. die automatische Speicherverwaltung.

Die Weiterentwicklung klassischer Prozessoren hat i. Allg. gezeigt, dass eine Steigerung der Verarbeitungsleistung nur durch Erhöhung der Taktfrequenz praktisch nicht realisierbar ist. Daher geht der Trend aktuell in Richtung mehrerer Prozessorkerne auf einem Chip. Damit kann einerseits die Verarbeitungsleistung durch Parallelität von Programmen auf Thread-Ebene wesentlich gesteigert werden. Andererseits kann auch die Reaktionsfähigkeit von Echtzeitsystemen verbessert werden: Ein oder mehrere gleichzeitig eintreffende Ereignisse können nun parallel zueinander bearbeitet werden.

Seit 2007 kam es zu mehreren Parallelentwicklungen von Java-Prozessoren mit mehreren Prozessorkernen, die diesen Forschungsbereich unabhängig voneinander vorantrieben. Erste brauchbare Entwicklungen erschienen fast zeitgleich 2008 und 2009 und wurden in den folgenden Jahren verfeinert. Die hier vorgestellte Arbeit wurde ebenfalls 2007 begonnen. Erfahrungen aus dem Universalprozessorbereich in Kombination mit Erkenntnissen zur dynamischen Häufigkeit einzelner Java-Bytecodes (u. a. [EKEL00]) lassen auch für Java-Mehrkernprozessoren eine wesentliche Steigerung der Verarbeitungsleistung erwarten. Die zentrale Fragestellung dieser Arbeit lautet daher:

> Wie ist die Architektur eines Java-Mehrkernprozessors zu gestalten,
> damit die auf Thread-Ebene ohnehin vorhandene Parallelität eines
> Java-Programms effizient zur Leistungssteigerung genutzt werden kann?

In diesem Zusammenhang sind auch Schnittstellen für Test und Diagnose zu diskutieren. Eine Diagnose von Fehlern aufgrund paralleler Zugriffe auf gemeinsam genutzte Ressourcen erfordert Methoden, die keinen Einfluss auf das Laufzeitverhalten des Programms nehmen. Anstatt wie bisher üblich die Software, muss nun die Hardware instrumentiert werden. Das in dieser Arbeit eingesetzte Werkzeug besteht in der Verbindung

des Prozessors mit einer universellen Trace-Architektur, die eine Aufzeichnung von Prozessorzuständen parallel zur unmodifizierten Ausführung des Programms erlaubt. Eine universelle Trace-Architektur kann u. a. für die Laufzeitanalyse in Echtzeitsystemen oder für die Analyse eines eingebetteten Systems in realer Umgebung genutzt werden.

1.2 Zielstellung

Die Zielstellung dieser Arbeit umfasst im Einzelnen:

- Analyse von Mehrkernprozessoren, speziell auch von Java-Prozessoren,
- Untersuchungen zur Realisierbarkeit von Mehrkern-Java-Bytecode-Architekturen (Java-Mehrkernprozessor),
- Analyse des Funktionsverhaltens, speziell der Speicherarchitektur eines konkreten Ansatzes,
- Implementierung eines Prototypen auf der Basis von SHAP sowie dessen Optimierung und Parametrierung,
- Test und Bewertung der Funktionalität des Prototypen,
- Leistungs- und Effizienzbewertung des gewählten Ansatzes auch im Vergleich zu anderen bekannten Ansätzen.

Im Vordergrund stehen dabei neben einer möglichst hohen Steigerung der Verarbeitungsleistung durch Parallelität, eine sehr gute Skalierbarkeit, eine hohe Effizienz sowie eine bessere Leistung gegenüber bekannten alternativen Ansätzen. Mögliche Einsatzgebiete der Zielarchitektur sind vor allem die eingebetteten Systeme, Echtzeitanwendungen und kleinere objektorientierte Rechnerknoten.

1.3 Herangehensweise

Der Ausgangspunkt für die vorliegende Arbeit ist die SHAP-Plattform mit einem Prozessorkern [ZPRS07a], die voll kompatibel zur standardisierten „Connected Limited Device Configuration“ [Sun03] der „Java Platform, Micro Edition“ ist. Als Startpunkt diente der Arbeitsstand aus dem Frühjahr 2007, der gemeinsam von Thomas B. Preußer und mir am Lehrstuhl für VLSI-Entwurfssysteme, Diagnostik und Architektur an der Technischen Universität Dresden ab Anfang 2006 entwickelt wurde. Die Aufgabenteilung bis 2007 stellte sich dabei wie folgt dar: Herr Preußer hat für diesen Java-Prozessor insbesondere das Stack-Modul mit integriertem Thread-Management implementiert. Er hat ebenso maßgeblich die SHAP-Klassenbibliothek entwickelt. Ich implementierte für den

Einkernprozessor vor allem die Mikroarchitektur und einen einfachen Methoden-Cache. Ich habe ebenso den Mikrocode-Assembler entwickelt. In gemeinsamer Arbeit entstanden die automatische Speicherverwaltung des Prozessors inklusive nebenläufigem Hardware-Garbage-Collector, der Mikrocode und der SHAP-Linker.

Herr Preußer hat in seiner Arbeit ab 2007 die Weiterentwicklung des Einkernprozessors vorangetrieben. Ich habe mich stattdessen auf den Mehrkernprozessor konzentriert. Die jeweils neu implementierten SHAP-Komponenten stehen natürlich dem jeweils anderen Projekt zur Verfügung.

Im zweiten Kapitel wird der Stand der Forschung und Technik erläutert und dabei insbesondere die Java-Plattform und Java-Prozessoren näher betrachtet. Im Detail wird außerdem der verwendete Arbeitsstand der SHAP-Plattform beschrieben.

In Kapitel 3 werden verschiedene, mögliche Lösungsvarianten für die Ausgestaltung einer Mehrkernarchitektur für Java-Prozessoren im Allgemeinen aufgezeigt. Zu berücksichtigen sind dabei insbesondere die Eigenschaften, die sich speziell aus der JVM-Spezifikation ergeben. Des Weiteren werden Schnittstellen für Test und Diagnose erörtert. Im Anschluss daran konzentriere ich mich auf eine Lösung. Ich analysiere sie anhand der SHAP-Plattform und vergleiche sie mit anderen vorgeschlagenen Lösungsvarianten. Eine zentrale Rolle spielt hierfür die Analyse der Speicherbandbreite für die Ausführung von Java-Programmen auf SHAP.

Im vierten Kapitel wird auf die prototypische Implementierung des SHAP-Mehrkernprozessors und deren Parametrierung eingegangen. Diese wird anschließend einer Leistungs- und Effizienzbewertung unterzogen und mit den Ergebnissen anderer Arbeiten verglichen. Abschließend wird die korrekte Funktion des Prototypen mittels Test überprüft und bewertet.

Im letzten Kapitel werden noch einmal die wesentlichen Ergebnisse zusammengefasst und im Sinne der Zielstellung bewertet. Abschließend wird ein Ausblick auf mögliche zukünftige Arbeiten gegeben.

1.4 Danksagung

Zuerst möchte ich meinem Betreuer Prof. Spallek für die intensive Betreuung meiner Dissertation danken. Insbesondere die vielen kritischen Anmerkungen zu den Arbeitsständen und den begleitenden Publikationen führten immer wieder zu neuen Ideen. Bei meinem Zweitgutachter möchte ich mich für die kontroverse Diskussion zur Einreichung und die Begleitung der Verteidigung bedanken. Besonderer Dank gilt auch Herrn Preußer für die sehr gute Zusammenarbeit im SHAP-Projekt. Auch wenn seit 2007 jeder sein eigenes Thema weiterverfolgt hat, so trug doch der Erfahrungsaustausch und die gemeinsame Nutzung neuer Komponenten zum Fortschritt meiner Arbeit bei. Ebenfalls Danke sagen

möchte ich den Studenten, die im Rahmen von Beleg- und Diplomarbeiten zusammen mit mir Untersuchungen für SHAP durchgeführt und neue Komponenten implementiert haben. Den weiteren Kollegen am Lehrstuhl möchte ich für den einen oder anderen Tipp sowie Entlastung bei administrativen Belangen danken. Zu guter Letzt möchte ich mich außerdem bei meiner Schwester, meinen Eltern, meinen Großeltern und diversen Bekannten für die moralische Unterstützung insbesondere in schwierigen Zeiten bedanken.

2 Stand der Forschung und Technik

2.1 Java-Plattform

2.1.1 Begriffe

Die folgenden Begriffsdefinitionen rund um die Java-Plattform sind, sofern nicht anders angegeben, dem Buch von Heinisch et. al. [HMHG07] entlehnt:

Rechner-Plattform: „Unter Rechnerplattform wird die Kombination von Betriebssystem und zugehöriger Rechnerhardware verstanden."

Java-Plattform: Zu einer Java-Plattform gehören: die Programmiersprache Java, verschiedene Werkzeuge (Java-Compiler u. a.), die „Java Virtual Machine" und eine Klassenbibliothek.

Die Oracle Corp.[1] (kurz Oracle) unterscheidet zwischen verschiedenen Ausführungen (engl. Bezeichnungen):

- „Java Platform, Standard Edition" (JavaSE) für Desktop-PCs,
- „Java Platform, Enterprise Edition" (JavaEE) für Server,
- „Java Platform, Micro Edition" (JavaME) für mobile Endgeräte (s. Abschn. 2.1.3).

Die einzelnen Ausführungen unterscheiden sich im Wesentlichen durch Art und Umfang der Klassenbibliothek und durch die verfügbaren Werkzeuge. Es wird jeweils ein angepasstes „Java Development Kit" und „Java Runtime Environment" bereit gestellt.

„Java Development Kit" (JDK): Das JDK bezeichnet die konkrete *Implementierung* einer Java-Plattform mit einem für die jeweilige Rechnerplattform spezifischen „Java Virtual Machine". Es enthält alle Komponenten für die *Entwicklung* eigener Java-Programme.

Bekannte Beispiele für Implementierungen der JavaSE-Spezifikation sind:

- Java SE JDK von Oracle[2],

[1] Ehemals Sun Microsystems, Inc.

[2] http://www.oracle.com/technetwork/java/javase/downloads/index.html

- OpenJDK unter Leitung von Oracle[3],
- Apache Harmony JDK von Apache[4].

„Java Runtime Environment“ (JRE): Das JRE umfasst nur diejenigen Bestandteile eines JDK, die für die *Ausführung* von Java-Programmen notwendig sind: „Java Virtual Machine“ und Klassenbibliothek.

„Java Virtual Machine“ (JVM): Die JVM führt den Bytecode des Java-Programms aus. Die Implementierung ist an die jeweilige Rechnerplattform angepasst. Weiteres hierzu s. Abschn. 2.1.2.

Klassenbibliothek: Eine Klassenbibliothek stellt das API (engl.: application programming interface) bereit. Sie umfasst Klassen und Schnittstellen, welche die Funktionalität der jeweiligen Java-Plattform offerieren.

2.1.2 Java Virtual Machine

Wie bereits genannt, führt die JVM den Bytecode des Java-Programms aus. Die JVM kann dazu auf verschiedene Weisen implementiert werden [Sch05]:

Interpreter: Bei dieser Variante ist die JVM ein Softwareprogramm, das den Bytecode interpretiert.

Just-in-Time-Compilation: Dies ist eine Erweiterung des Interpreters, bei der zur Laufzeit die Methoden (auch) in den Maschinencode der Rechnerplattform übersetzt werden, um die Ausführung zu beschleunigen.

Batch-Compilation: Hier wird der Java-Quellcode direkt in den Maschinencode der Rechnerplattform übersetzt, statt in den Bytecode der JVM.

Java-Prozessor: Dies ist eine Implementierung der JVM in Hardware. Der native Befehlssatz des Java-Prozessors ist der Bytecode der JVM.

Die JVM bildet die Laufzeitumgebung für das auszuführende Programm. Sie übernimmt Aufgaben wie [HMHG07]:

- Laden der Klassen des Java-Programms und des API,
- Ausführung des Bytecodes nebst Prüfung des Kontextes,
- Speicherverwaltung inklusive GC (engl.: garbage collector),
- Ein- / Ausgabe-Operationen,
- Verwaltung und Synchronisation von Threads.

[3] http://openjdk.java.net/
[4] http://harmony.apache.org/

Die JVM-Spezifikation legt u. a. fest [LY99]:

- Semantik des Java-Bytecodes, nebst Datentypen, gemeinsamer Oberklasse (Class Object) und einer Klasse für Unicode-Zeichenketten (Class String)
- Getrennte Adressräume für Bytecode, den Operanden-Stack und den Objekt-Heap
- Eine automatische Speicherverwaltung des Heaps: Der von Objekten belegte Speicherbereich wird ausschließlich automatisch wieder freigegeben. Für die Bestimmung und Freigabe der nicht mehr benötigten Objekte ist der GC zuständig.
- Synchronisation zwischen mehreren Threads: Dafür sind sogenannte Monitore und dazu passend Bytecodes vorgesehen. Im Gegensatz zu Sperrvariablen (engl.: spin locks) können damit abgesicherte kritische Abschnitte von ein und demselben Thread mehrmals betreten werden. Unterschiedliche Threads schließen sich jedoch gegenseitig aus. Jedes Java-Objekt kann als Monitor dienen, d. h. die Synchronisation erfolgt auf Objekte (z. B. die Liste, die atomar verändert werden soll).

2.1.3 Java Platform, Micro Edition

2.1.3.1 Konfigurationen

Die vorliegende Arbeit beschränkt sich auf JavaME, da die anderen Editionen (JavaSE und JavaEE) nicht für den Einsatz auf Geräten mit begrenzten Ressourcen, wie mobile und eingebettete Systeme, konzipiert sind. Es wird bei JavaME zwischen zwei Konfigurationen unterschieden [Sun00]:

Die „Connected Device Configuration“ (CDC) ist für eingebettete Systeme mit moderaten Ressourcen bzgl. Hauptspeicher und Netzwerk vorgesehen.

Die „Connected Limited Device Configuration“ (CLDC) zielt auf Endgeräte mit einfachen Benutzerschnittstellen, geringem Hauptspeicherausbau und langsamen Netzwerkbandbreiten ab. Neben dem Basisprofil steht für die CLDC noch optional das „Mobile Information Device Profile“ zur Verfügung, das zusätzliche Funktionalität wie dauerhafte Datenspeicherung, grafische Ausgabe per LCD, erweiterte Netzwerkfunktionen u. a. bietet.

Die vorliegende Arbeit beschränkt sich auf die CLDC, da bereits diese Konfiguration die Ausführung mehrerer Threads und deren Synchronisation unterstützt. Weitergehende Informationen finden sich im Handbuch von Sun Microsystems Inc. [Sun03].

2.2 SHAP-Plattform

2.2.1 Allgemeines

Die SHAP-Plattform ist eine Hard- und Softwareumgebung für die Ausführung von Java-Programmen in eingebetteten Systemen unter Nutzung eines Java-Prozessors. SHAP vereint damit Rechner- und Java-Plattform, benötigt aber kein anderes unterliegendes Betriebssystem. SHAP ist kompatibel zur JavaME in der CLDC.

SHAP umfasst im Einzelnen:

- Implementierung der JVM als Mikroprozessor (μP),
- angepasste Java-Klassenbibliothek,
- Linker für die Erzeugung der ausführbaren SHAP-Datei,
- Assembler für den Mikrocode des Prozessors,
- Modell der Prozessorarchitektur für taktgenaue Simulation.

Es wurden keine neuen Elemente zur Programmiersprache Java hinzugefügt. Damit wird kein spezieller Java-Compiler für SHAP benötigt. Als Standard wurde der Eclipse Java-Compiler[5] festgelegt.

In den folgenden Abschnitten soll kurz der Arbeitsstand beschrieben werden, der als Grundlage für die Entwicklung einer SHAP-Mehrkernarchitektur diente. Ausführliche Informationen zu dieser Grundlage sind in vorhandenen Publikationen zu finden [ZPRS07a, ZPRS07b, PZR07].

Die wesentlichen Besonderheiten der SHAP-Einkernarchitektur sind:

- Implementierung der JVM als echtzeitfähiger Java-Prozessor:
  - native Ausführung des Java-Bytecodes ohne unterliegendes Betriebssystem,
  - integriertes Stack- und Thread-Management,
  - nebenläufige Ausführung des GC ohne Blockierung des Programms,
  - unabhängige Monitore für die Synchronisation von Threads,
  - preemptives Thread-Scheduling.
- Unterstützung objektorientierter Konzepte:
  - automatische Speicherverwaltung,
  - strukturierte Ausnahmebehandlung,
  - Mehrfachvererbung mittels Interfaces.
- Unterstützung des CLDC-API in der Version 1.1.

[5] http://www.eclipse.org

2.2.2 SHAP-Klassenbibliothek

Das API von SHAP ist kompatibel zum Basisprofil des CLDC-API in der Version 1.1. Als Grundlage für die SHAP-Klassenbibliothek diente die CLDC-Implementierung des phoneME-Projekts[6]. Anzupassen waren all jene Funktionen, die von einer konkreten JVM-Implementierung abhängen. Dies betraf:

- Laden der Klassen eines Programms,
- Verwaltung und Synchronisation von Java-Threads,
- Ein-/Ausgabe-Operationen (`System.in`, `out` und `err`),
- Aufruf und Interaktion mit dem GC,
- Werfen von Ausnahmen (engl.: exceptions) entsprechend der JVM-Spezifikation wie bspw. Zugriff auf eine Null-Referenz (s. u.),
- Softwareimplementierung für Operationen auf Gleitkommazahlen,
- Spezialfunktionen für das beschleunigte Kopieren von Objekten und Datenfeldern.

Die Anpassung erfolgte durch zusätzlichen Java-Code und zusätzliche Spezial-Bytecodes. Letzteres war immer dann notwendig, wenn auf Funktionen des SHAP-µP zurückgegriffen werden musste.

Dem Basisprofil der CLDC wurden Klassen für zusätzliche Funktionalitäten hinzugefügt:

- Grafische Benutzerschnittstellen mit Tastatursteuerung (Pakete `ite.conio` und `ite.graphx`),
- Netzwerkkommunikation über UDP/IP (Paket `ite.io` u. a.).

Entsprechend der JVM-Spezifikation werden in folgenden Fällen Exceptions vom Mikrocode erzeugt und geworfen:

- Auflösen der Null-Referenz,
- Division durch Null,
- Erzeugen eines Arrays mit negativer Größe oder zu vielen Elementen,
- Zugriff auf einen Index außerhalb der Array-Grenzen,
- Schreiben eines Objekts mit falschem Typ in ein Objekt-Array,
- Fehlerhafte Typkonvertierung,
- Verbotene Objektinstanziierungen.

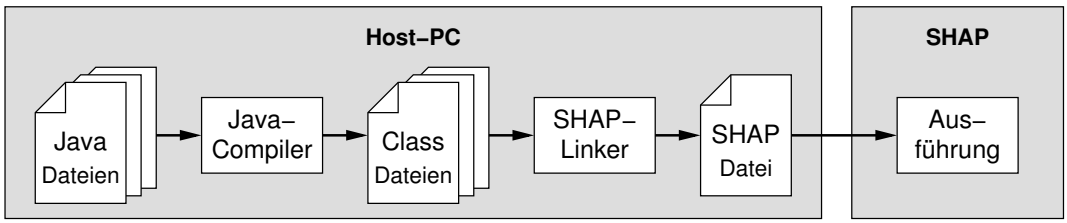

Abbildung 2.1: Position des SHAP-Linkers im Entwurfsfluss

2.2.3 SHAP-Linker

Der SHAP-Linker transformiert die vom Java-Compiler erzeugten Class-Dateien des Programms und der Klassenbibliothek in eine SHAP-Datei, die vom SHAP-µP direkt ausgeführt wird, siehe Abb. 2.1. Diese Transformation erfüllt mehrere Zwecke:

- Die Klassen werden mitsamt ihren Klassenvariablen und Methoden in die interne Struktur überführt, die auch später zur Laufzeit verwendet wird. Für jede Klasse wird eine separate Instanz im Heap-Speicher von SHAP angelegt (im Gegensatz zu einer großen statischen Struktur für alle Klassen), um somit Klassen weiterer Programme dynamisch zur Laufzeit nachladen zu können [Ebe10].
- Die auf Zeichenketten basierenden Verweise im Konstantenpool einer Klasse werden durch die entsprechenden Indizes in die interne Klassenstruktur ersetzt. Textkonstanten werden während des Ladens der SHAP-Datei als Instanzen vom Typ String alloziert, sodass sie anschließend wie zur Laufzeit erzeugte Strings behandelt werden können.
- Die `native`-Methoden, die zum Aufruf von JVM-internen Funktionen wie bspw. Erzeugung eines Threads dienen, werden durch neu definierte Spezial-Bytecodes ersetzt, um die entsprechende Funktion im SHAP-µP aufzurufen.

Während der Transformation können gleichzeitig Optimierungen am Bytecode durchgeführt werden. Dieser Aspekt ist Bestandteil der Dissertation von Hrn. Preußer [Pre11].

2.2.4 SHAP-Mikroprozessor

Der SHAP-Einkernprozessor besteht aus folgenden Komponenten (Abb. 2.2):

Der Prozessorkern führt den vom SHAP-Linker modifizierten Java-Bytecode ohne Unterstützung durch ein unterliegendes Betriebssystem aus.

Für die Ausführung der Bytecodes wird ein **Mikroprogrammsteuerwerk** mit einer 4-stufigen Pipeline verwendet. Der Mikrocode besteht aus 65 Befehlen zu je 9 Bit, wobei die kürzeste Mikroprogrammsequenz aus einem Befehl besteht. Der

[6] https://phoneme.dev.java.net/

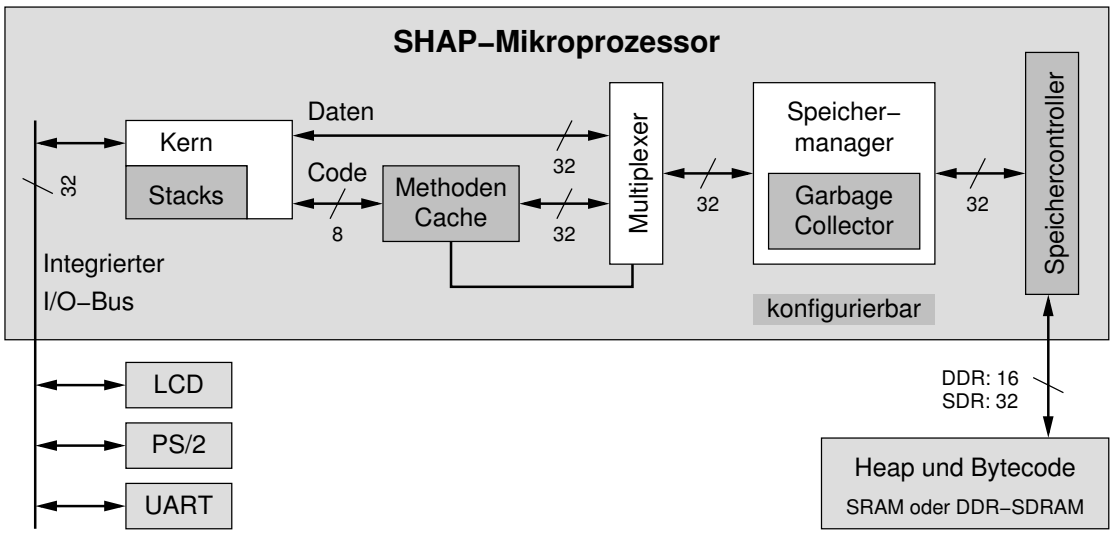

Abbildung 2.2: SHAP-µP in der Fassung von 2007

Wechsel zwischen den Sequenzen benötigt keine zusätzlichen Taktzyklen. Befehle und Daten des Mikroprogramms sind in getrennten Speichern[7] abgelegt und über separate Busse zugreifbar.

Das Mikroprogramm implementiert auch den **Scheduler**. Die Strategie ist auf Round-Robin festgelegt, um die zur Verfügung stehende Rechenzeit gleichmäßig auf die Threads zu verteilen. Die Ausführung der Threads wird nach Ablauf ihrer Zeitscheibe preemptiv unterbrochen, wobei die Ausführung des aktuellen Bytecodes noch beendet wird. Wenn ein Thread auf ein Betriebsmittel oder ein Ereignis wartet, kann er selbstständig die Ausführung abgeben.

Das **Rechenwerk** beinhaltet eine 32-Bit-ALU (engl.: arithmetic logic unit) für Ganzzahlarithmetik. Gleitkommaoperation werden in Software emuliert.

Für den Zugriff auf Stack, Bytecode, Heap und I/O stehen getrennte **Busse** zur Verfügung.

Der Stack ist als On-Chip-Speicher realisiert und direkt an den Prozessorkern angeschlossen. Diese Komponente unterstützt „multi-threading“, indem sie mehrere dynamisch wachsende Stackbereiche für entsprechend mehrere Threads bereitstellt. Der Thread-Wechsel wird durch den Scheduler ausgelöst.

Der Methoden-Cache speichert die aktuell ausgeführte Java-Methode zwischen, um ein Lesen des Bytecodes mit geringer Latenz zu gewährleisten. In der ersten Realisierung war nur die Speicherung einer Methode vorgesehen. Eine Weiterentwicklung, die natürlich auch dem SHAP-Mehrkernprozessor zur Verfügung steht, erfolgte durch Thomas B. Preußer [PZS07].

[7] 2048×9 Bit für Befehle, 64×32 Bit für Daten

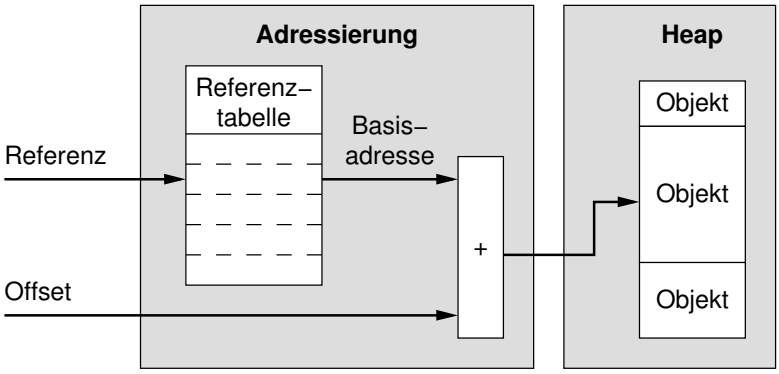

Abbildung 2.3: Indirekte Adressierung

Der Speichermanager verwaltet automatisch die Objekte auf dem Java Heap. Die Lokalisierung der Objekte im Speicher erfolgt über eine indirekte Adressierung (s. Abb. 2.3), damit das Verschieben von Objekten im Speicher zwecks Kompaktierung des belegten Speichers auf einfache Art und Weise ermöglicht wird. Die Zuordnung zwischen Referenz und Basisadresse ist in einer Referenztabelle am Anfang des externen Speichers hinterlegt. Zur Beschleunigung der Adressberechnung ist ein TLB (engl.: translation look-aside buffer) mit einem Eintrag vorgesehen.

Der Speichermanager besitzt lediglich einen Kommando-Port für alle Aktionen. Der Zugriff von Prozessorkern und Methoden-Cache erfolgt im Multiplex-Betrieb. Die Steuerung des Multiplexers erfolgt durch den Methoden-Cache, wobei er sich während des Cachings exklusiven Zugriff erteilt.

Der Garbage-Collector ist in den Speichermanager integriert. Er arbeitet parallel zum Prozessorkern. Er bestimmt automatisch die nicht mehr referenzierten Objekte und gibt diese frei. Er ist auch für die Kompaktierung des freien Speichers zuständig.

Auch das Verschieben von Objekten im Heap-Speicher zum Zwecke der Kompaktierung erfolgt nebenläufig zu den Lese- und Schreibzugriffen des Prozessorkerns. Ebenso kann auf Objekte zugegriffen werden, während sie verschoben werden. Dazu müssen Speicherzugriffe auf den bereits verschobenen Objektteil auf die neue Position umgelenkt werden. Die Adressberechnung prüft wie in Abb. 2.4 dargestellt:

1. Stimmt die Objektreferenz des Lese- / Schreibzugriffs mit der der Objektverschiebung überein?
2. Wenn ja, wird auf einen Offset nach der aktuellen Verschiebeposition zugegriffen?
3. Wenn ja, addiere Offset zur neuen statt aktuellen Basisadresse.

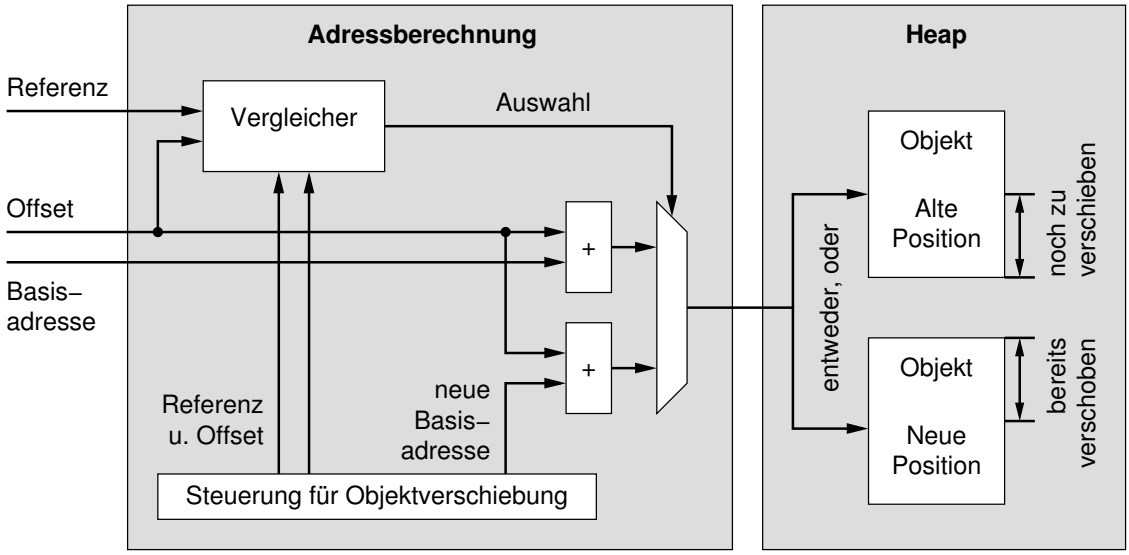

Abbildung 2.4: Adressberechnung bei nebenläufiger Verschiebung von Objekten

Der integrierte Speichercontroller bindet den Speichermanager direkt an einen externen Speicher auf SRAM- oder DDR-SDRAM-Basis an. Der Zweck besteht in kurzen Latenzen beim Speicherzugriff.

Der integrierte I/O-Bus gewährt Zugriff auf verschiedene I/O-Geräte für die Kommunikation mit der Außenwelt. Der Prozessorkern ist der Bus-Master. Die I/O-Geräte können mittels `native`-Methoden (und damit speziellen Bytecodes) angesprochen werden.

2.2.5 Laden einer SHAP-Datei

Das Laden einer SHAP-Datei auf den SHAP-µP erfolgt in 4 Schritten:

1. Der Mikrocode lädt den in Java implementierte Bootloader in den Heap-Speicher.
2. Der Bootloader lädt die Java-Klassen und Strings des Programms und der Klassenbibliothek.
3. Die statischen Initialisierungsmethoden werden ausgeführt.
4. Die Ausführung beginnt dann wie üblich mit der `main`-Methode.

2.2.6 Mikrocode-Assembler

Für die Assemblierung des Mikrocodes wird das „XASM — The eXtensible Assembler and Generic Assembler Framework“ eingesetzt, welches am Lehrstuhl für VLSI-

Entwurfssysteme, Diagnostik und Architektur an der Technischen Universität Dresden von Marco Kaufmann und mir entwickelt wurde.

Ausschlaggebend für das XASM waren die einfache Spezifikation des Befehlssatzes über eine Textdatei. SHAP-Spezifika wie die Sprungtabelle des Mikrocodes werden über ein Plugin realisiert.

2.2.7 Modell der Prozessorarchitektur

Als Alternative zur Simulation der gesamten — in VHDL vorliegenden — Hardwarebeschreibung wurde auch ein Modell des SHAP-µP für den Prozessorsimulator DITO [Pre03] erstellt. Mit diesem kann der Mikrocode taktgenau ausgeführt und der Stack- und Heapzustand beobachtet werden. Es steht sowohl eine Einzelschrittausführung als auch eine kontinuierliche Ausführung bis zum Eintreten einer Bedingung zur Verfügung.

Ebenso ist es mit dem Modell möglich den Bytecode des Programms oder des API zu debuggen. Jedoch fehlen DITO aktuell erweiterte Features eines Software-Debuggers wie automatische Zuordnung der Quellcodezeile zum aktuell ausgeführten Bytecode.

2.3 Klassifikation von Parallelarchitekturen

2.3.1 Ebenen der Parallelität

Die Verarbeitungsleistung von Prozessorarchitekturen kann mittels Parallelität auf 4 verschiedenen Ebenen gesteigert werden [SW07, S. 124],[HP07]:

Bitebene: (engl.: bit-level parallelism) Mehrere Bits eines Datenworts werden parallel verarbeitet. Auf der erweiterten Bitebene werden mehrere Datenwörter eines Vektors mit ein und demselben Befehl verarbeitet (SIMD, s.u.).

Befehlsebene: (engl.: instruction-level parallelism) Es werden mehrere Befehle parallel ausgeführt (VLIW, engl.: very long instruction word) und / oder die Befehlsausführung mittels Pipelining verschachtelt.

Thread-Ebene: (engl.: thread-level parallelism) Auf Thread-Ebene werden die einzelnen Fäden (Threads) eines Programms parallel ausgeführt.

Programmebene: (engl.: application-level parallelism) Im Rahmen einer Multi-Task-Umgebung werden mehrere Anwendungen oder Prozesse parallel ausgeführt. Die mögliche Trennung in 2 separate Ebenen (für Anwendung und Prozess) ist hier nicht relevant.

2.3.2 Klassifikation nach Flynn

Die Art und Weise der Parallelität in Prozessorarchitekturen lässt sich weiterhin nach Flynn klassifizieren. Dabei wird nach der Anzahl der Befehlsströme und der Anzahl der Datenströme unterschieden, sodass sich folgende 4 Klassen ergeben [SW07, S. 125]:

SISD (engl.: single instruction single data): Ein Befehlsstrom arbeitet auf einem Datenstrom.

MISD (engl.: multiple instruction single data): Leere Klasse.

SIMD (engl.: single instruction multiple data): Ein Befehlsstrom arbeitet auf mehreren Datenströmen.

MIMD (engl.: multiple instruction multiple data): Mehrere Befehlsströme arbeiten auf mehreren Datenströmen.

2.3.3 Mehrkernprozessoren

Die vorliegende Arbeit befasst sich ausschließlich mit Parallelität auf Thread- und Programmebene. Die im Folgenden vorgestellte Prozessorarchitektur fällt damit in die Klasse MIMD. Unabhängig davon werden im Mikroprogrammsteuerwerk von SHAP Befehlsebenen- (Pipelining) und Bitebenenparallelität angewendet.

Die Integration mehrerer Prozessoren (eines MIMD-Systems) auf einem Chip wird heute als Mehrkernprozessor (engl.: multi-core processor) bezeichnet. Die einzelnen Kerne umfassen jeweils eigene Steuer- und Rechenwerke und ggf. Caches. Alle Kerne eines Prozessors teilen sich dagegen typischerweise Speicher- und I/O-Busse sowie ggf. weitere Cache-Ebenen [HP07, S. 198].

2.3.4 Speicher und Adressräume

MIMD-Systeme lassen sich weiterhin nach der Speicheranbindung ihrer Prozessoren klassifizieren. Es sind zu unterscheiden [HP07, PH05]:

- Die physische Anordnung des Speichers:

  Zentraler / Gemeinsamer Speicher: Alle Prozessoren nutzen einen zentralen gemeinsamen Speicher und besitzen damit einen gemeinsamen Adressraum.

  Verteilter Speicher: Der Speicher ist physisch auf die einzelnen Prozessoren verteilt.

- Die logische Organisation des Adressraums:

Gemeinsamer Adressraum: (engl.: shared address space) Der gesamte Speicher kann von den Prozessoren über einen gemeinsamen Adressraum angesprochen werden.

In Abhängigkeit von der physischen Anordnung erhält man einen zentralen gemeinsamen Speicher (engl.: central shared-memory) oder einen verteilten gemeinsamen Speicher (engl.: distributed shared-memory). Der Datenaustausch zwischen mehreren Threads kann implizit über gemeinsam genutzte Variablen im Speicher erfolgen.

Private Adressräume: (engl.: private address space) Jeder Speicher bildet einen separaten Adressraum, sodass jeder Prozessor nur auf seinen Speicher zugreifen kann.

Der Datenaustausch zwischen Threads auf verschiedenen Prozessoren muss explizit über Nachrichten erfolgen (engl.: message-passing).

- Die resultierenden Eigenschaften beim Speicherzugriff:

  UMA (engl.: uniform memory access): Die Zeit für einen Speicherzugriff im gemeinsamen Adressraum ist *unabhängig* von der (physischen) Adresse. Das ist bei einem zentralen Speicher üblich.

  NUMA (engl.: non-uniform memory access): Die Zeit für einen Speicherzugriff im gemeinsamen Adressraum ist *abhängig* von der (physischen) Adresse. Das ist bei einem verteilten Speicher üblich.

  Liegt außerdem Cache-Kohärenz vor, nennt man dies Cache-kohärenter NUMA (CC-NUMA, engl.: cache coherent non-uniform memory access).

  NoRMA (engl.: no-remote memory access): Zugriff auf den Speicher anderer Prozessoren ist nicht möglich [SW07, S. 127].

Ein MIMD-System mit zentralem gemeinsamem Speicher wird auch als symmetrisches Mehrprozessorsystem (engl.: symmetric (shared-memory) multiprocessor) bezeichnet, da dort das UMA-Prinzip gilt. Der Speicher kann dabei in mehrere Bänke unterteilt sein [HP07, S. 200]. Im Allgemeinen wird Kohärenz zwischen den Caches verschiedener Prozessoren sichergestellt [HP07, S. 205].

Die Aufteilung des gemeinsamen Speichers in mehrere Bänke zwecks Erhöhung der verfügbaren Speicherbandbreite stellt eine Mischform zwischen zentralem und verteiltem Speicher dar. Das UMA-Prinzip gilt weiterhin [HP07, S. 216].

Für die Sicherstellung der Cache-Kohärenz in einem Mehrprozessorsystem muss die gemeinsame Nutzung von Datenblöcken im physischen Speicher verfolgt werden. Dazu stehen zwei Klassen von Protokollen zur Verfügung [HP07, S. 208]:

Verzeichnis-basiert: Der Status bzgl. der gemeinsamen Nutzung eines Datenblocks wird nur einmal in einem (logisch gesehen) zentralem Verzeichnis festgehalten.

Snooping: Der Status bzgl. der gemeinsamen Nutzung eines Datenblocks wird dezentral in jedem Cache gespeichert, der auch eine Kopie dieses Datenblocks besitzt. Die Caches verschiedener Prozessoren sind über ein Netzwerk verbunden, auf dem sie die Anforderung von Datenblöcken aller Prozessoren beobachten können (engl.: snoop).

Die im Folgenden vorgestellte Mehrkernvariante von SHAP realisiert die Prinzipien:

- zentraler gemeinsamer Speicher für den Java Heap und Bytecode,
- verteilter Speicher mit privaten Adressräumen für den Stack,
- (Methoden-)Cache für den Bytecode.

2.4 Verifikation und Test

Verifikation und Test haben ein korrekt funktionierendes System zum Ziel. Sie unterscheiden sich jedoch prinzipiell in der Herangehensweise.

2.4.1 Test

2.4.1.1 Allgemeines

Nach G. Kemnitz ist ein Test „ein Funktionsablauf, bei dem das Testobjekt mit Eingaben stimuliert wird und seine Ausgaben auf Richtigkeit überwacht werden“ [Kem07, S. 33]. Die gesamte Folge der Eingaben wird dabei als Testsatz und eine Teilfolge als Testschritt bezeichnet. Ein Test spürt Fehler auf, kann aber nicht garantieren, dass das Testobjekt frei von Fehlern ist.

Die grundsätzlichen Strategien für die Auswahl der Testschritte sind [Kem07, S. 40ff]:

Funktionsorientierte Testauswahl: Hierbei wird die Funktion des Testobjekts in die Auswahl einbezogen, um den Testaufwand auf die wichtigsten und sicherheitskritischsten Funktionen zu fokussieren.

Strukturorientierte Testauswahl: Hier wird die Struktur des Systems ausgenutzt. Mögliche Ansätze sind der Zellentest und die modellfehlerorientierte Testauswahl.

Zufällige Testauswahl: Aus den möglichen Eingaben für das Testobjekt wird eine zufällige Auswahl getroffen. Die erforderliche Anzahl der Testschritte „hängt in erster Linie von der Struktur des Testobjekts und nicht von seiner Funktion ab.“ [Kem07, S. 45]

Bei einem Schaltkreis ist zwischen Entwurfsfehlern und physischen Fehlern zu unterscheiden [ABF94]:

- Entwurfsfehler entstehen während des Schaltkreisentwurfs. Der Test auf Entwurfsfehler wird im folgenden Abschnitt näher betrachtet.
- Physische Fehler sind bedingt durch die Fertigung des Schaltkreises sowie dem anschließendem Einsatz und der damit verbundenen Strahlungsbelastung und Alterung. Sie sind im Rahmen der vorliegenden Arbeit nicht von Bedeutung, da kein Schaltkreis gefertigt werden soll und die Entwurfswerkzeuge als korrekt angenommen werden.

Zum Teil wird in der Literatur unter dem Begriff Test nur der Test auf physische Fehler verstanden. Der Test auf Entwurfsfehler wird dort der Verifikation zugeordnet (vgl. [Ber06]).

2.4.1.2 Test auf Entwurfsfehler

Der Test auf Entwurfsfehler besteht „aus der Ausführung oder der Simulation der unmodifizierten Systembeschreibung mit einer hinreichend langen zufälligen Eingabefolge und der Überwachung der Ausgaben.“ [Kem07, S. 236].

Eine automatisierte Berechnung der Soll-Ausgaben erfordert eine zweite, fehlerfreie Entwurfsbeschreibung oder ein Golden Device. Stehen diese nicht zur Verfügung, muss auf andere Kontrollverfahren zurückgegriffen werden [Kem07, S. 236]:

Die Inspektion der Testausgaben erfolgt durch manuelle Kontrolle.

Bei Verdopplung und Vergleich werden die Ausgaben mit denen eines Vergleichssystems verglichen. Im Gegensatz zu Soll-Ausgaben sind die Vergleichswerte jedoch „etwa mit derselben Häufigkeit falsch wie die zu überwachenden Daten.“ [Kem07, S. 169]

Die Probe prüft, ob das Ergebnis einem hinreichenden Kriterium genügt. Eine spezielle Form ist die Rückgewinnung der Eingabe aus der Ausgabe, mit Hilfe der inversen Funktion des Testobjekts.

Der Plausibilitätstest prüft, ob der Ausgabewert im zulässigen Wertebereich liegt. Bei einem Syntaxtest wird zusätzlich geprüft, ob die Folgen von Werten ein Wort einer vereinbarten Sprache sind.

2.4.2 Verifikation

2.4.2.1 Allgemeines

Aufgabe der Verifikation ist allgemein „die Überprüfung der Korrektheit eines Schaltungsentwurfs [gegen die Spezifikation] in Bezug auf die funktionalen, zeitlichen und sonstigen Anforderungen“ [SS07].

Das Einhaltung vorgegebener Zeitschranken für den Schaltkreis wird hier durch die verwendeten Entwurfswerkzeuge garantiert. Hier soll daher die Verifikation des funktionalen Verhaltens näher betrachtet werden.

Unter Verifikation wird teilweise nur der formale Nachweis verstanden (z. B. [RP06]). Je nach persönlicher Interpretation werden aber auch simulationsbasierte Ansätze zur Verifikation hinzugezählt, insbesondere, wenn mit Test nur der Test auf Fertigungsfehler assoziiert wird (z. B. [Ber06]).

Der Nachweis der korrekten Funktion wird auch als funktionale Verifikation bezeichnet. Der Begriff wird jedoch unterschiedlich angewendet:

- Einerseits dient er als Oberbegriff für die (funktionale) formale und die (funktionale) simulationsbasierte Verifikation [Lam05, PH09].
- Andererseits wird er als Gegenstück zur formalen Verifikation verstanden [SS07, Ber06].

2.4.2.2 Formale Verifikation

Die Verfahren der formalen Verifikation nutzen formale Methoden für den Nachweis der gewünschten Eigenschaft. Im Gegensatz zum Entwurfstest und zur simulationsbasierten Verifikation wird insbesondere kein Satz von Testschritten benötigt. Die Verfahren lassen sich in zwei Kategorien einteilen [Lam05, Ber06]:

Equivalence-Checking: Hierbei werden zwei Schaltungsbeschreibungen (Entwurfszustände) der selben Spezifikation auf Äquivalenz überprüft.

Property- / Model-Checking: Bei diesen Verfahren werden aus der Spezifikation Eigenschaften (engl.: properties) abgeleitet, die eine Schaltungsbeschreibung erfüllen muss. Der Nachweis erfolgt, indem entweder der gesamte Zustandsraum auf Einhaltung der Eigenschaft geprüft wird oder indem Beweise mittels deduktiver Methoden geführt werden (engl.: theorem proving).

Die Verfahren der formalen Verifikation werden im Rahmen der vorliegenden Arbeit aus Komplexitätsgründen nicht angewendet.

2.4.2.3 Simulationsbasierte Verifikation

Bei der simulationsbasierten Verifikation wird die Schaltungsbeschreibung in eine Testbench eingebettet, die die Umgebung des Schaltkreises modelliert [Ber06]. Die Testbench erzeugt die Eingaben (Stimuli) und verarbeitet ggf. die Ausgaben zu neuen Eingaben weiter. Die Simulation kann nur die Anwesenheit von Fehlern feststellen, jedoch nicht deren Abwesenheit.

Die Auswahl der Stimuli ist entweder zufällig oder wird entsprechend der zu testenden Funktionalität ausgewählt [Lam05]. Für die Verifikation von Verbindungsstrukturen wird auch folgende Mischform eingesetzt [BGA05]: Die Zugriffe werden zufällig ausgewählt, jedoch werden für jeden Zugriff die passenden (und damit ausgewählte) Stimuli angelegt.

Neben der manuellen Überprüfung, können sowohl die Ausgaben als auch die internen Zustände mit Hilfe von Zusicherungen (engl.: assertions) [Ber06] überwacht werden. Assertions sind spezielle Anweisungen, die in die Testbench und / oder in die Schaltungsbeschreibung eingefügt werden und vom Simulator auf Gültigkeit geprüft werden. Die simulationsbasierte Verifikation wird daher auch als „Assertion-based Verification“ bezeichnet [Bor09].

Die Bewertung der simulationsbasierten Verifikation erfolgt mit den Metriken [Ber06]:

Codeüberdeckung: (engl.: code coverage) Diese gibt an, wie viel Prozent

- der Anweisungen (engl.: statement coverage),
- der Kontrollpfade (engl.: path coverage),
- der Ausdrücke (engl.: expression coverage) und
- der Zustände der endlichen Automaten (engl.: finite-state machine coverage)

in der Schaltungsbeschreibung durch die Simulation ausgeführt wurden.

Eine Codeüberdeckung von 100% gibt daher nur an, ob die Schaltungsbeschreibung vollständig abgearbeitet wurde. Insbesondere kleine Prozentzahlen geben an, dass die Verifikation noch nicht abgeschlossen ist.

Funktionsüberdeckung: (engl.: function coverage) Diese Metrik gibt an, wie viel Prozent der in der Spezifikation festgehaltenen Funktion ausgeführt wurden, z. B. wie viel Prozent der Prozessorbefehle oder der möglichen Füllstände einer Warteschlange simuliert wurden.

Mit der Simulation sind folgende Verfahren verwandt [SS07, BGA05]:

Bei der hardwarebeschleunigten Simulation werden die nicht zu untersuchenden Schaltungsteile in Hardware ausgelagert, um deren Funktion dort schneller zu berechnen.

Bei der Emulation wird die Schaltung auf speziellen hardwarebasierten Emulatoren implementiert und damit die Funktion gegenüber der Simulation viel schneller berechnet. Der Kern besteht häufig aus programmierbaren Schaltkreisen, die mit der gewünschten Funktion geladen werden und damit das Verhalten emulieren. Die Eingaben werden von Funktionsgeneratoren erzeugt. Es können wie bei der Simulation sowohl die Ausgaben als auch interne Zustände beobachtet werden.

Zusätzlich kann der Emulator an die reale Umgebung angeschlossen werden, in der später auch der Schaltkreis eingesetzt wird. Somit können Daten mit Komponenten ausgetauscht werden, für die gar keine Simulationsmodelle vorliegen.

Beim FPGA-Prototyping (FPGA = engl.: field programmable gate array) wird die Funktion des Schaltkreis ebenfalls auf programmierbare Schaltkreise geladen. Diese sind an die reale Umgebung angeschlossen. Im Unterschied zur Emulation können jedoch standardmäßig keine internen Zustände beobachtet werden.

2.4.3 Abgrenzung

Wie bereits angeführt wurde, ist die Abgrenzung zwischen Test und Verifikation in der Literatur nicht einheitlich. Nach meiner Auffassung kann der Test auf Entwurfsfehler auch durch simulationsbasierte Verifikation erfolgen. Ich habe mich daher im Folgenden auf den Begriff Test festgelegt.

2.4.4 Anwendung

Die Systembeschreibung der Mehrkernvariante von SHAP wurde sowohl mittels Simulation als auch mittels FPGA-Prototyping getestet. Um beim FPGA-Prototyping zumindest einen Teil der internen Zustände sichtbar zu machen, kommt eine von mir entwickelte und implementierte Trace-Architektur zum Einsatz. Die Testauswahl erfolgt nach Funktion, Struktur und auch Zufall. Als Kontrollverfahren kommen die Inspektion der Testausgaben und Proben (= Assertions) zum Einsatz. Die Bewertung erfolgt mit den Metriken Codeüberdeckung (nur Simulation) und Funktionsüberdeckung.

Physische Fehler und formale Verifikation werden wie gesagt nicht weiter betrachtet.

2.5 Leistungsbewertung

Die Leistung eines Prozessors kann anhand verschiedener Metriken bewertet werden. Die folgenden Ausführungen sind, sofern nicht anders angegeben, dem Buch von H. G. Kruse [Kru09] entnommen. Der dabei verwendete Begriff „Last“ umfasst die auszuführende Aufgabe, z. B. ein komplettes Programm oder auch nur die Bearbeitung eines einzelnen Auftrags / Datenpakets.

2.5.1 Ausführungszeit und CPI

Ein erster Ansatz ist der Vergleich auf Basis der Ausführungszeiten t_{EXE} für eine gegebene Last:

$$t_{EXE} = N_I \cdot CPI \cdot T \ . \tag{2.1}$$

Dabei bezeichnet:

- N_I die Anzahl der ausgeführten Instruktion,
- CPI die Anzahl der Taktzyklen pro Befehl und
- $T = 1/f$ die Periodendauer eines Taktzyklus bei einer Taktfrequenz von f.

Der CPI-Wert als statistischer Mittelwert ist in der Regel unterschiedlich für verschiedene Befehlsgruppen k aus der Menge aller Befehlsgruppen K und wird mit CPI_k bezeichnet. Der CPI-Wert des betrachteten Programms kann über die dynamischen Befehlshäufigkeiten p_k der einzelnen Befehlsgruppen bestimmt werden:

$$CPI = \sum_{k \in K} p_k \cdot CPI_k \, . \tag{2.2}$$

2.5.2 MIPS, MFLOPS und Durchsatz

Leistung im Sinne von Arbeit pro Zeit kann anhand der Maßzahlen MIPS, MFLOPS und Durchsatz beurteilt werden.

Das gängige Maß MIPS (engl.: million instructions per second) gibt die Anzahl der ausgeführten Instruktionen pro Sekunde in Millionen an:

$$MIPS = \frac{N_I}{t_{EXE}} \cdot 10^{-6} = \frac{f}{CPI} \cdot 10^{-6} \, . \tag{2.3}$$

Es eignet sich jedoch nur bedingt zum Vergleich von verschiedenen Prozessoren aufgrund der Abhängigkeit vom CPI-Wert. MIPS ist sowohl von der Last als auch vom Compiler abhängig, da beides Einfluss auf die dynamischen Befehlshäufigkeiten p_k hat. Dies kann insbesondere dazu führen, dass der MIPS-Wert sich invers zur gemessen Leistung (z. B. der Ausführungszeit) verhält, wie das Beispiel in [Kru09] anschaulich zeigt.

Ein weiteres gängiges Maß ist MFLOPS (engl.: million floating-point operations per second), das analog zu MIPS definiert ist:

$$MFLOPS = \frac{N_{FLOPS}}{t_{EXE}} \cdot 10^{-6} \, . \tag{2.4}$$

Hierbei gibt N_{FLOPS} die Anzahl der Gleitkommaoperation der Last an. MFLOPS ist ebenso lastabhängig, denn die so gemessene Leistung hat nur wenig Aussagekraft, wenn die Last keine oder nur wenige Gleitkommaoperationen erfordert.

Standard-Last für MIPS und MFLOPS ist der sogenannte Linpack-Benchmark[8]. Er wird insbesondere beim Vergleich von Supercomputern und der Aufnahme in die Top-500-Liste[9] verwendet. Aufgrund der enormen Leistungsfähigkeit heutiger Systeme erfolgt dort die Leistungsangabe jedoch in Tera-FLOPS und Peta-FLOPS.

[8] http://www.top500.org/project/linpack
[9] www.top500.org

Der Durchsatz D als drittes gängiges Maß gibt die (durchschnittliche) Anzahl von erledigten Aufträgen pro Zeiteinheit an. Dazu wird die Anzahl erledigter Aufträge N_{erl} im Messintervall t_M bestimmt:

$$D = \frac{N_{erl}}{t_M} \; . \tag{2.5}$$

Er ist ebenso lastabhängig. Die Last kann zudem mit jedem Auftrag (z. B. von einem Client an einen Server) variieren, sodass Durschnittswerte untersucht werden müssen.

2.5.3 Antwortzeit und Latenz

Die Antwortzeit oder auch Verweilzeit gibt die (durchschnittliche) Dauer für die Bearbeitung eines Auftrags an. Sie umfasst neben der Ausführungszeit ebenso die Wartezeit in Zwischenpuffern. Diese Puffer speichern Aufträge zwischen, solange die Ausführungseinheit beschäftigt ist.

2.5.4 Leistungssteigerung bei Nutzung von Parallelität

Die Leistungssteigerung bei der Ausführung auf n parallelen Prozessoren oder Verarbeitungseinheiten (einer Pipeline) kann mit den Gesetzen von Amdahl und Gustafson abgeschätzt oder berechnet werden [Amd67, Gus88].

Amdahl betrachtet die Verkürzung der Ausführungszeit unter Einsatz von n Prozessoren für eine gegebene, konstante Last [Amd67]. Die Verkürzung wird als Beschleunigungsfaktor (engl.: speed-up) $S_A(n)$ angegeben. Er hängt im allgemeinen Fall nur vom parallelisierbaren Zeitanteil α des Programms ab und berechnet sich aus den Ausführungszeiten $t_{EXE}(i)$ auf i Prozessoren:

$$S_A(n) = \frac{t_{EXE}(1)}{t_{EXE}(n)} = \frac{1}{(1-\alpha)+\alpha/n} \; . \tag{2.6}$$

Der Verlauf des Speed-Ups ist für große n asymptotisch:

$$\lim_{n\to\infty} S_A(n) = 1/(1-\alpha) \; . \tag{2.7}$$

Im Gegensatz zu Amdahl lässt Gustafson die Ausführungszeit konstant und skaliert stattdessen die Last [Gus88]. Der Speed-Up $S_G(n)$ gibt nun an, um welchen Faktor die Problemgröße oder die Anzahl der Aufträge erhöht werden kann. Dazu wird die Last auf einem Prozessor in einen seriellen $(1-\beta)$ und einen parallelen Anteil β zerlegt. Im allgemeinen Fall ergibt sich aus den Durchsätzen $D(i)$ auf i Prozessoren (im selben Messintervall t_M):

$$S_G(n) = \frac{D(n)}{D(1)} = \frac{N_{erl}(n)}{N_{erl}(1)} = (1-\beta)+n\cdot\beta \; . \tag{2.8}$$

Der Speed-Up nach Gustafson ist demnach eine lineare Funktion.

Die theoretisch mögliche Leistung von n Prozessoren ist n-mal so hoch wie die eines einzelnen Prozessors. Die Effizienz mit der dies erreicht wird, kann entsprechend Amdahl und Gustafson berechnet werden zu:

$$E_A(n) = \frac{S_A(n)}{n} = \frac{1}{(1-\alpha)n+\alpha}\,, \tag{2.9}$$

$$E_G(n) = \frac{S_G(n)}{n} = \frac{1-\beta}{n}+\beta\,. \tag{2.10}$$

Dabei ist anzumerken, dass für den Regelfall $\alpha < 1$ die Effizienz nach Amdahl bei großer Prozessoranzahl gegen Null geht:

$$\lim_{n\to\infty} E_A(n) = \begin{cases} 0 & \text{für } \alpha < 1 \\ 1 & \text{für } \alpha = 1 \end{cases}\,. \tag{2.11}$$

2.5.5 Benchmarks

Für die Leistungsbewertung von Java-Laufzeitumgebungen stehen eine Reihe von Benchmarks zur Verfügung. Eine Auswahl an bekannteren Benchmarks ist in Tab. 2.1 gegenüber gestellt. Von Interesse sind dabei die Kriterien:

- Anforderungen an das API,
- benötige Menge an Heap-Speicher und
- Nutzung von Parallelität auf Thread-Ebene.

Neben den zum Teil synthetischen Benchmarks können auch echte Programme zur Leistungsbewertung herangezogen werden.

Im Rahmen dieser Arbeit können nur Programme eingesetzt werden, die lediglich CLDC und weniger als 1 MiB Heap-Speicher erfordern. Zudem sollten sie die Parallelität auf Thread-Ebene bewerten. Damit kommen von den bekannteren Benchmarks nur die Suite des Java Grande Forum und Teile des „JemBench“ in Frage. Zusätzlich werden auch eigene Benchmarks evaluiert.

2.6 Bisherige Arbeiten

Die vorliegende Arbeit zielt ausschließlich auf Mehrkernprozessoren, die nativ Java-Bytecode ausführen. Nicht Gegenstand sind Softwareimplementierungen der JVM für Mehrkernprozessoren mit anderen Befehlssätzen. Damit werden auch keine Prozessoren betrachtet, die lediglich spezielle Befehle anbieten, um Interpretation oder Kompilierung zu vereinfachen (z. B. Jazelle RCT [Por05b], MAJC [TCC+00]).

Tabelle 2.1: Eigenschaften bekannterer Java Benchmarks

Benchmark	API	Heap-Speicher	Teilprogramme mit Parallelität auf Thread-Ebene
SPECjvm2008[a]	JavaSE	ca. 1 MiB	diverse
SPECjbb2005 und SPECjAppServer[b]	JavaEE	$\geq$ 512 MiB	diverse
Linpack für Java[c]	CLDC	$\leq$ 1 MiB	nicht parallelisiert
Grinderbench[d]	CLDC	unbekannt	nur in der Sektion „Benutzerinteraktion“
DaCapo Benchmarks [BGH+06]	JavaSE	unbekannt	keine
Benchmark-Suite des Java Grande Forum[e]	CLDC	Problemgröße anpassbar	Matrixmultiplikation IDEA-Verschlüsselung Raytracing Monte-Carlo-Simulation
JemBench [SPU10]	CLDC	$\leq$ 1 MiB	Raytracer, Matrixmultiplikation Lösung des N-Damen-Problems

[a] http://www.spec.org/jvm2008/docs/benchmarks/index.html
[b] http://www.spec.org/jbb2005/ und http://www.spec.org/jAppServer2004/
[c] http://www.netlib.org/benchmark/linpackjava/
[d] http://www.grinderbench.com/about.html
[e] http://www.epcc.ed.ac.uk/research/java-grande/

Für die Einordnung der in dieser Arbeit verwendeten SHAP-Plattform (s. Abschn. 2.2) werden im folgenden Abschnitt zunächst andere Java-Prozessoren kurz vorgestellt. Im Anschluss daran werden diejenigen Prozessoren näher betrachtet, die zusätzlich mehrere Kerne auf einem Chip integrieren und damit in direktem Bezug zu dieser Arbeit stehen.

2.6.1 Einordnung von SHAP

Es wurde bereits eine Reihe von Ansätzen zur effizienten Ausführung von Java-Bytecode direkt in der Hardware von eingebetteten Systemen erforscht. Dazu gehört auch die Abbildung der JVM-Konzepte (s. Abschn. 2.1.2) auf Hardware- und Softwarekomponenten, um eine schnelle Ausführung auch ohne die Hilfe eines Betriebssystems zu erreichen. Zu den Java-Prozessoren zählen:

- FemtoJava [ICJ01],
- picoJava-I und -II [MO98, OT97, Sun99b],
- jamuth (ehemals Komodo) [UW07],

- JOP und JopCMP [Sch05, Sch06, PS08],
- JEM2 alias aJile-100 [Har01],
- die Erweiterung „Jazelle DBX“ der ARM-Prozessoren [Por05a],
- REALJava [TSP08],
- BlueJEP [GW07]
- ein rekonfigurierbarer Java-Prozessor (R-Java)[KKT+00],
- ein asynchroner Java-Prozessor [YCM+04],
- ein Java-Prozessor für Multimedia-Anwendungen [BKP99],
- ein VLIW Java-Prozessor (nur als Simulation) [BC04],
- AMIDAR (nur als Simulation) [GH05],
- SHAP (Abschn. 2.2).

In Tab. 2.2 sind die Eigenschaften ausgewählter Prozessoren und deren Realisierung wichtiger JVM-Konzepte gegenüber gestellt. Nicht aufgeführt sind Prozessoren, zu denen die Literatur keine ausreichenden Informationen bereitstellt oder die nicht in Hardware umgesetzt wurden.

Die Java-Prozessoren unterscheiden sich hauptsächlich in den Punkten:

Stack: Er wird typischerweise in einem lokalen, internen Speicher oder als Registerbank bereitgestellt.

Mikroarchitektur: Insbesondere bei einer Stack-Realisierung als Registerbank bietet sich zusätzlich die Technik „Bytecode-Folding“ an, bei der mehrere Bytecodes zu einem internen RISC-ähnlichen Befehl (RISC = engl.: reduced instruction-set processor) zusammengefasst werden.

Für die Ausführung komplexer Bytecodes werden Mikrocode und / oder Software-Traps eingesetzt. Bei letzterem wird eine Softwareroutine aufgerufen, die die gewünschte Funktionalität (z. B. Gleitkommaarithmetik) implementiert. Die Softwareroutinen liegen wiederum als Java Bytecode vor. Eine Ausnahme bilden REALJava und ARM Jazelle DBX, die mehrere Befehlssätze unterstützen.

Threads: Die meisten Prozessoren unterstützen die parallele Ausführung mehrerer Threads, bei manchen ist die Anzahl jedoch statisch festgelegt: bei jamuth die Anzahl der Echtzeit-Threads, bei JOP die Anzahl aller Threads. Der nötige Scheduler ist direkt in Hardware, im Mikrocode oder in Software implementiert. Die Monitore für die Synchronisation zwischen Threads sind zumeist den Objekten zugeordnet. Zum Teil steht jedoch nur ein einzelner, globaler Monitor zur Verfügung (z. B. JOP).

Tabelle 2.2: Eigenschaften ausgewählter Java-Prozessoren

	FemtoJava [ICJ01]	PicoJava-II [Sun99b]	Berekovic et.al. [BKP99]	jamuth (Komodo) [UW07]	JOP [Sch05]	JEM2 (aJ-100) [Har01]	REALJava [TSP08]	SHAP (Abschn. 2.2)
Pipeline-Stufen	5	6	?	5	4	?	?	4
Stack-Realisierung	e	r	r	i	i	r	i	i
Bytecode-Folding	-	+	+	-	-	?	+	-
Mikrocode	-	+	+	+	+	+	-	+
Software-Traps	-	+	?	+	+	+	+	+
Allokation von Objekte	-	+	+	+	+	+	+	+
Mehrere Threads	-	+	-	(+)[a]	(+)[b]	+	+	+
Thread HW-Support	-	-	-	+	-	+	?	+
Scheduler	-	?	-	H	S	M	S	M
Synchronisation	-	+	-	?	(+)[c]	+	?	+
Echtzeit-Threads	-	?	-	+	+	+	?	+
Garbage-Collector	-	S	S	S	S	S	-	H
GC für Echtzeit-Threads	-	?	-	+	-	-	-	+

H	Hardware	M	Mikrocode	S	Software
i	Interner Speicher	r	Register	e	Externer Speicher
+	unterstützt	-	nicht u.	?	unbekannt

[a] nur statische Anzahl von Echtzeit-Threads
[b] nur statisch allozierte Threads
[c] nur einzelner, globaler Monitor

Speicherverwaltung: Der GC ist, sofern vorhanden, entweder in Hardware implementiert oder wird als Java-Thread in Software ausgeführt. Zu unterscheiden ist weiterhin, ob wie bei SHAP und jamuth der GC auch für Threads mit Echtzeit-Priorität zur Verfügung steht oder ob stattdessen die restriktivere „Real-Time Specification for JAVA“ (RTSJ) [Sun99a] zur Anwendung kommt. Die RTSJ sieht für Echtzeit-Threads spezielle Teil-Heaps vor, deren Objekte erst nach der Beendigung des betreffenden Threads freigegeben werden.

2.6.2 Java-Mehrkernprozessoren

Die Beschleunigung von Java-Prozessoren durch Nutzung von Parallelität auf Befehlsebene wurde bereits vielfältig erforscht. Zum einen wurden Konzepte zum Bytecode-Folding (s.o.) untersucht, (z. B. [RTJ00, EKEL01, SEP06]). Zum anderen wurden für Multimediaanwendungen auch die Gruppierung von Java-Bytecodes zu VLIW-Befehlen [BC04] sowie Parallelität auf der erweiterten Bitebene analysiert [BKP99]. Auch SHAP nutzt bereits Parallelität auf Bitebene und Befehlsebene (Pipelining).

Die vorliegende Arbeit befasst sich daher ausschließlich mit Prozessoren, die Java-Bytecode nativ ausführen und gleichzeitig Parallelität auf Thread-Ebene nutzen. Dazu werden im Folgenden bisherige bzw. parallel zu meiner Arbeit entwickelte Ansätze vorgestellt.

2.6.2.1 JopCMP

Das JopCMP-System verbindet mehrere JOP-Kerne über den SimpCon-Bus mit dem zentralem gemeinsamen Speicher und den I/O-Komponenten [PS08]. Für den SimpCon-Bus wurden für die Arbitrierung neben dem ursprünglich verwendeten Round-Robin noch andere Strategien untersucht [PS10]. Der SimpCon-Bus stellt möglicherweise einen Flaschenhals dar, da kein Pipelining von Speicherzugriffen möglich ist.

Der gemeinsame Speicher ist als externer Speicher realisiert. Er beinhaltet den Java Heap, die Stacks der (inaktiven) Threads sowie den Bytecode des Programms. Objekte werden indirekt über eine Referenz adressiert. Der Speichercontroller ist in den JopCMP integriert.

Jeder JOP-Kern besitzt einen eigenen Befehls-Cache sowie einen Stack-Cache, in dem der Stack des gerade aktiven Threads zwischengespeichert wird.

Ein Objekt-Cache wurde lediglich auf Basis eines Verhaltensmodells analysiert [HPS10]: Er wird als Objekt-Cache bezeichnet, da jede Cache-Zeile nur ein Objekt zwischenspeichert. Ebenso erstreckt sich ein Objekt nur über eine Cache-Zeile. Da die Länge einer Cache-Zeile begrenzt ist, wird der darüber hinausgehende Objektteil nicht zwischengespeichert. Die Identifizierung der Cache-Zeile (Tag) erfolgt über die Objektreferenz, sodass bei einem Cache-Hit auch die Auflösung der Objektreferenz in eine Basisadresse entfällt. Beim Schreiben wird die Strategie „write-through“ in Kombination mit „no-write allocate“ verwendet. Zusammen mit der Invalidierung des Caches beim Betreten eines `synchronized`-Blocks oder Zugriff auf eine `volatile`-Variable in Java wurde ein einfaches Cache-Kohärenz-Protokoll umgesetzt.

Für die Synchronisation von Threads steht nur ein globaler Monitor zur Verfügung. Voneinander unabhängige kritische Abschnitte können daher nicht gleichzeitig betreten werden.

Der GC ist in Java implementiert. Ein Zugriff auf den Inhalt des Heaps und somit auch der Stacks ist über spezielle `native`-Methoden möglich. Während der Ausführung des GC werden alle anderen Threads angehalten. Der GC wird auf einem designierten Kern ausgeführt [PS07].

2.6.2.2 jamuth-Mehrkernprozessor

Sascha Uhrig evaluierte einen Mehrkernprozessor auf der Basis von „jamuth“ [Uhr09]. Ein Avalon-Bus verbindet mehrere Kerne mit dem zentralen gemeinsamen Speicher und bietet ein Pipelining von Speicherzugriffen verschiedener Kerne an. Die Anbindung der I/O-Komponenten geschieht über einen separaten Avalon-Bus.

Auch hier besitzt jeder Kern einen eigenen Befehls-Cache sowie einen eigenen Stack-Cache. Ein Heap-Cache ist nicht vorgesehen.

Jeder jamuth-Kern unterstützt die quasi-parallele Ausführung von bis zu 4 Threads. Das Laden und Dekodieren der Befehle erfolgt zunächst parallel. Ein integrierter Hardware-Scheduler weist dann einem dieser Threads die gemeinsam benutzten Ausführungseinheiten (Rechenwerk, Buscontroller) zu.

Der verteilte Mark-And-Sweep-GC läuft parallel auf allen Kernen und ist in Java implementiert [UU09]: Auf jedem Kern wird ein GC-Thread gestartet und dieser parallel zu den anderen Threads (des Programms) ausgeführt. Eine Synchronisation zwischen den GC-Threads ist zum Start eines GC-Zyklus und nach Abschluss der Mark-Phase notwendig. Eine Synchronisation zwischen GC und Programm ist nur bei Allokation und Freigabe von Objekten notwendig. Ferner wird der Speicher in Segmente unterteilt. Ein Segment wird von je einem GC-Thread verwaltet. Kompaktierung des freigegebenen Speichers ist nicht vorgesehen. Objekte im Speicher werden direkt ohne Umweg über eine Referenz adressiert.

2.6.2.3 REALJava

Die REALJava-Architektur verbindet einen RISC-Prozessor mit mehreren „Java Processing Units“ (JPUs) über einen gemeinsamen Bus [TSP08]. Die JPUs führen nur einfache Java-Bytecodes in Hardware aus und sind mit einem lokalen Speicher für den Stack und den Methoden-Code bestückt. Der Java Heap liegt im zentralen gemeinsamen Speicher. Jedes Mal wenn ein komplexer Bytecode ausgeführt werden soll, wird ein Interrupt bei dem RISC-Prozessor ausgelöst. Dieser interpretiert dann den Bytecode in Software. Ein GC ist nicht vorgesehen.

Es stehen zwei Konfigurationen zur Auswahl [TSP10], die beide auf den „Processor Local Bus“ von Xilinx setzen: 1.) PowerPC 405 plus bis zu 3 JPUs oder 2.) Microblaze plus bis zu 8 JPUs. Im Gegensatz zu den anderen Projekten, setzt REALJava ein Betriebssystem (Linux) auf dem RISC-Prozessor ein.

2.6.2.4 ARM Jazelle DBX

Ab der Prozessorarchitektur „ARMv5“ steht ARM-Prozessoren optional die Erweiterung „Jazelle DBX“ zur Verfügung [Por05a]. Sie ist bspw. im Mehrkernprozessor Cortex-A9[10] integriert.

Die Decode-Pipelinestufe wird um einen Zustandsautomaten erweitert, der Java-Bytecodes dekodiert. Ein spezieller Befehl schaltet zwischen den verschiedenen Befehlssätzen um. Im Jazelle-Modus werden die 4 obersten Stack-Elemente, lokale Variablen, der Stackzeiger und der Befehlszähler im Registersatz gespeichert. Stack-Elemente werden automatisch zwischen Hauptspeicher und Registersatz umkopiert (engl.: stack spill). Für die Bytecode-Ausführung werden bereits vorhandene Caches wiederverwendet.

Jazelle DBX unterteilt die Bytecodes in 3 Klassen:

- direkte Ausführung
- Emulation durch eine kurze Folge von ARM-Befehlen
- „undefinierte“ Befehle: Hierbei wird ein Software-Trap aufgerufen, der wiederum die gewünschte Funktion als ARM-Code implementiert.

Welche Bytecodes welcher Klasse zugeordnet werden, variiert je nach Jazelle-Implementierung in einem konkreten Prozessor.

2.6.3 Vergleich mit klassischen Mehrkernprozessoren

In diesem Abschnitt soll kurz ein Vergleich mit klassischen Mehrkernprozessoren hergestellt werden, die nicht direkt Java-Bytecode als Befehlssatz unterstützen. Ich beschränke mich dabei auf aktuelle Prozessoren, die von den Herstellern für eingebettete Systeme empfohlen werden.

In Tab. 2.3 sind die Anzahl der Prozessorkerne und die Anbindung des/r Speicher inklusive Caches gegenüber gestellt. Zwischen den Gruppen der Java- und der klassischen Prozessoren sind folgende Gemeinsamkeiten und Unterschiede erkennbar:

Kernanzahl: Diese ist bei Java-Prozessoren tendenziell höher.

Threads/Kern: In der Regel steht je Kern nur ein Steuerwerk zur Verfügung. Damit können nur die Befehle eines Threads gleichzeitig dekodiert werden. Eine quasiparallele, abwechselnde Ausführung mehrerer Threads ist in der Regel immer möglich.

Speicher und Caches: Beide Gruppen verwenden vorwiegend einen zentralen gemeinsamen Speicher für Befehle und Daten (Heap und Stack). Zugriffe auf diesen

[10] http://www.arm.com

werden durch Caches beschleunigt, wobei klassische Prozessoren mehrere Cache-Ebenen einsetzen.

Abweichend von dieser Regel werden auch lokale Speicher pro Kern eingesetzt:

- bei SHAP für den Stack,
- bei REALJava für Stack und Befehle.

Speicherbandbreite: Klassische Prozessoren erhöhen die Speicherbandbreite durch breitere Datenbusse und / oder mehrerer Speicherkanäle.

Tabelle 2.3: Eigenschaften ausgewählter Mehrkernprozessoren

Name	Kerne	Threads/Kern	L1-Cache	L2-Cache	L3-Cache	Speicherkanäle
Java-Prozessoren						
jamuth-Mehrkernproz.	1 – 3	1 – 3	I+S, P	-	-	1x 32-bit Avalon
JopCMP	1, 2, 4, 8	1	I+S, P	-	-	1x 32-bit SimpCon
REALJava	1 – 8	1	L, P	-	-	1x 32-bit PLB
SHAP-Mehrkernproz.	1 – 18	1	I+L, P	-	-	1x 32-bit
Klassische Prozessoren						
ARM Cortex-A9	1 – 4	1	I+D, P	opt.	-	1x 64-bit AMBA
ARM Cortex-A15	1 – 4	1	I+D, P	U, G	-	1x 128-bit AMBA
AMD G-Series	1 – 2	1	I+D, P	U, P	-	1x 64-bit DDR3
AMD Opteron 4100	4, 6	1	I+D, P	U, P	U, G	2x 64-bit DDR3
AMD Opteron 6100	8	1	I+D, P	U, P	U, G	4x 64-bit DDR3
Freescale MPC8640	2	1	I+D, P	U, P	-	2x 64-bit DDR2
Freescale P5020	2	1	I+D, P	U, P	U, G	2x 64-bit DDR3
Intel Core-i3 2120	2	2	I+D, P	U, P	U, G	2x 64-bit DDR3
Intel Core-i5 2400	4	1	I+D, P	U, P	U, G	2x 64-bit DDR3
Intel Core-i7 2600	4	2	I+D, P	U, P	U, G	2x 64-bit DDR3
IBM PowerPC 476	1 – ?	1	I+D, P	U, ?	-	2x 128-bit

I	Befehls-Cache	D	Daten-Cache		
S	Stack-Cache	U	vereinigter Befehls- / Daten-Cache		
L	lokale(r) Speicher pro Kern (siehe Text)				
P	privater Cache pro Kern	G	gemeinsamer Cache für alle Kerne		
opt.	optional	-	nicht vorhanden	?	unbekannt

3 Java-Mehrkernprozessor

3.1 Zielstellung

Den Ausgangspunkt für diese Arbeit stellt die SHAP-Plattform von 2007 mit einem Prozessorkern dar (Abschn. 2.2). Auf Basis dieser Plattform soll ein Java-Mehrkernprozessor entwickelt werden, der folgende Ziele erreicht:

- Integration mehrerer Prozessorkerne auf einem Chip,
- Steigerung der Verarbeitungsleistung durch Nutzung von Parallelität auf Thread-Ebene,
- höhere Leistung als vergleichbare Ansätze,
- sehr gute Skalierbarkeit und Effizienz,
- effektive Ausnutzung der Speicherbandbreite,
- Kompatibilität zur bisherigen SHAP-Plattform,
- Beobachtbarkeit mit Hardware-Instrumentierung.

Diese Ziele sollen durch folgende, eigene Ansätze erreicht werden:

1. Effiziente Speicherarchitektur bestehend aus:
   - lokalem On-Chip-Speicher für den Stack (je Kern),
   - einem zentralen gemeinsamen Speicher für Heap und Bytecode,
   - einem Mehrport-Speichermanager mit optimierter objektorientierter Adressierung, Pipelining und einem Vollduplex-Speicherbus,
   - schnelle atomare Operationen für Thread-Synchronisation im integriertem Speichercontroller,
   - einem GC mit verteiltem Stack-Scan.
2. Zweistufiges verteiltes preemptives Thread-Scheduling.
3. Trace-Architektur für:
   - Test und Diagnose sowie
   - Evaluation des Systems.

Für die Entwicklung des Mehrkernprozessors werden in diesem Kapitel zunächst die Grundlagen untersucht und der Entwurfsraum abgesteckt. Darauf aufbauend werden anhand von Analysen sinnvolle Varianten für die SHAP-Plattform ausgewählt und ein Prototyp entwickelt. Die Funktionalität, Leistungsfähigkeit und Effizienz wird im nächsten Kapitel bewertet.

3.2 Entwurfsraum

Beim Entwurf eines Java-Mehrkernprozessors stellen sich im Wesentlichen 6 Fragen:

- Welche Alternativen stehen für die physische Anordnung der einzelnen Speicher für Stacks, Heap und Bytecode zur Verfügung?
- Welche Auswirkungen hat die Speicheranordnung auf die automatische Speicherverwaltung?
- Wie sollen I/O-Komponenten angebunden werden?
- Welche Auswirkungen ergeben sich für das Thread-Scheduling?
- Welche Möglichkeiten bieten sich für die Thread-Synchronisation an?
- Welche Schnittstellen stehen für Test und Diagnose zur Verfügung?

3.2.1 Speicheranordnung

Entsprechend der JVM-Spezifikation (s. Abschn. 2.1.2) besitzt jedes Java-Programm — ein oder mehreren Threads, die sich einen Heap teilen — drei Typen von Adressräumen:

- getrennte, private Stack-Adressräume für jeden Thread,
- ein gemeinsamer Heap-Adressraum für alle Threads,
- ein gemeinsamer Bytecode-Adressraum für alle Threads.

In den folgenden Abschnitten werden getrennt für jeden Typ die möglichen physischen Anordnungen der Speicher erläutert. Anschließend wird auf hybride Varianten und auf die parallele Ausführung mehrerer Programme eingegangen.

3.2.1.1 Stack

Die privaten Stack-Adressräume können in mehreren verteilten oder einem zentralen Speicher abgelegt werden. Mögliche Varianten sind wie in Abb. 3.1 dargestellt:

a) Der Speicher ist auf die Prozessorkerne verteilt, wobei jeder Kern exklusiven Zugriff auf seinen lokalen Speicher erhält. In jedem lokalen Speicher ist genau ein Stack abgelegt.

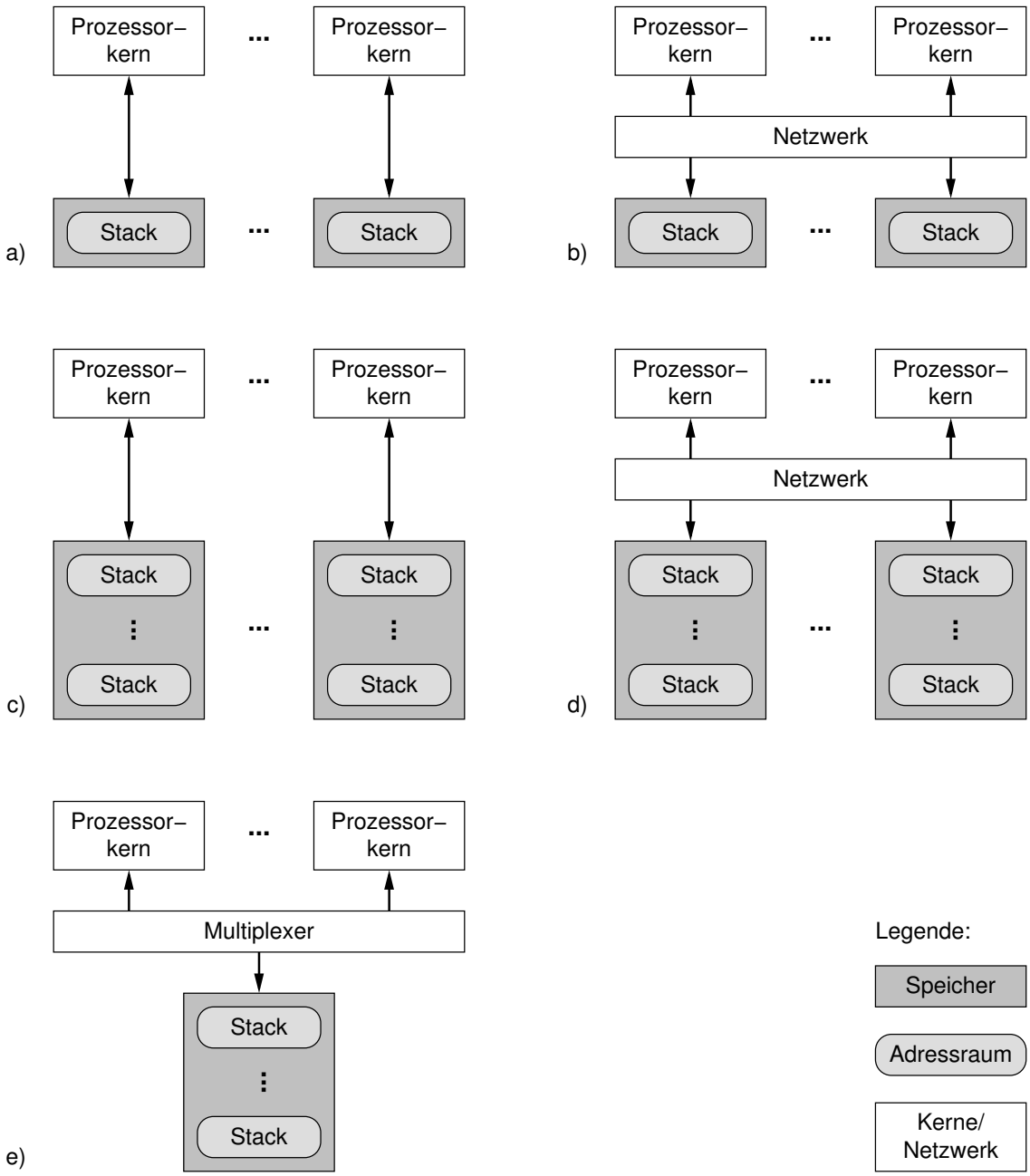

Abbildung 3.1: Mögliche Anordnungen (a–e) der Stack-Adressräume

b) Der Speicher ist auf die Prozessorkerne verteilt. Über ein Verbindungsnetzwerk kann jedoch jeder Kern auf jeden Speicher zugreifen. In jedem lokalen Speicher ist genau ein Stack abgelegt.

c) Der Speicher ist wie in a) verteilt. In jedem Speicher werden aber mehrere Stacks verwaltet, sodass jeder Kern die Ausführung zwischen verschiedenen Threads wechseln kann.

d) Der Speicher ist wie in b) verteilt. In jedem Speicher werden aber mehrere Stacks verwaltet, sodass die Ausführung zwischen mehr Threads wechseln kann als Kerne vorhanden sind.

e) Alle Stacks sind in einem zentralen Speicher abgelegt. Die Anzahl der Stacks muss größer oder gleich der Kernanzahl n sein, damit auch eine n-fach parallele Ausführung möglich ist.

Da fast jeder Java-Bytecode (außer z. B. `goto`) auf den Stack zugreift, ist eine kurze Latenz beim Speicherzugriff anzustreben. In den Varianten a) und c) sollte eine lokale Platzierung von On-Chip-Speichern ausreichen. In den übrigen ist zusätzlich der Einsatz von Caches — ggf. in mehreren Ebenen — zu prüfen:

- In Variante b) führt das Verbindungsnetzwerk typischerweise zu zusätzlichen Latenzen.
- Zusätzlich zu den Latenzen des Netzwerks, müssen sich in Variante d) ggf. mehrere Kerne die Bandbreite zu einem Speicher teilen, falls die gerade von ihnen verwendeten Stacks in einem Speicher abgelegt sind.
- In Variante e) müssen sich prinzipbedingt alle Kerne die Bandbreite zum zentralen Speicher teilen.

Bei Einsatz von Stack-Caches ist die Kohärenz nur abgeschwächt sicherzustellen, da niemals mehrere Threads auf den gleichen Stack zugreifen können. Wird jedoch die Ausführung eines Threads auf einem anderen Prozessorkern fortgesetzt, so ist lediglich sicherzustellen, dass der Cache zunächst in den Speicher zurückgeschrieben wird. Anschließend kann die Ausführung mit den aktuellen Daten auf einem anderen Prozessorkern fortgesetzt werden.

Die Varianten a) und c) sind aus Sicht des Chipflächen- und Energiebedarfs vorteilhaft, da einerseits das Verbindungsnetzwerk entfällt und andererseits kein Cache zu erwarten ist. Gleichzeitig ist aber im Gegensatz zu den anderen Varianten das Thread-Scheduling eingeschränkt: Je Kern kann der Scheduler nur über eine Teilmenge der Threads verfügen.

Die Varianten a) und b) beschränken die Stack- und damit auch die Threadanzahl stark. Zum Teil werden für die Verwaltung von Systemressourcen zusätzlich zum Programm ein oder mehrere Threads benötigt, die jedoch erfahrungsgemäß nur kurzzeitig Rechenzeit

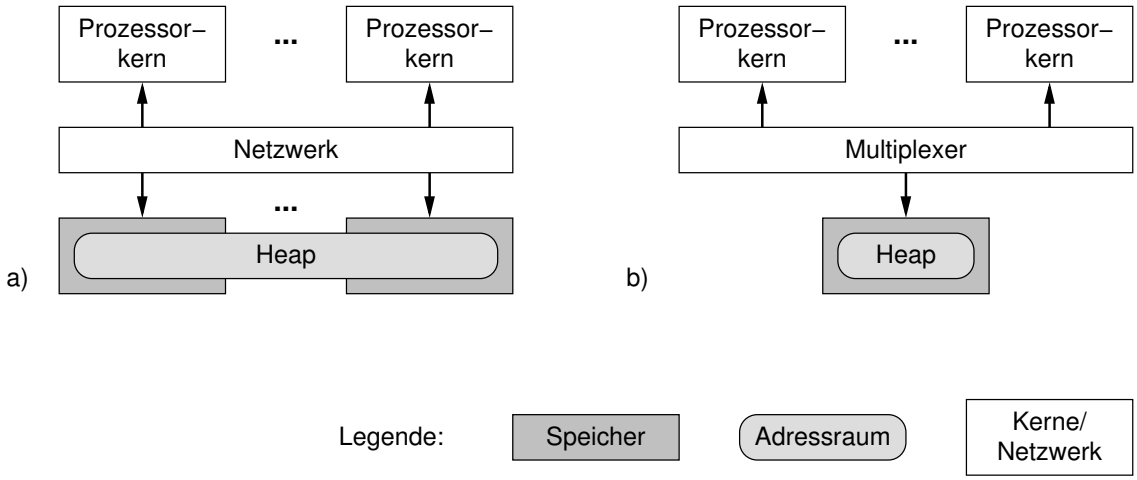

Abbildung 3.2: Mögliche Anordnungen (a und b) des Heap-Adressraums

beanspruchen. Da jedoch keine weiteren Stacks zur Verfügung stehen, liegen die freien Rechenkapazitäten somit brach. Abhilfe schaffen sowohl die Varianten c) und d) als auch Variante e), wenn mehr Stacks als Kerne verwaltet werden können.

Variante e) hat den großen Nachteil, dass insbesondere bei hoher Kernanzahl nur ein Bruchteil der Speicherbandbreite der anderen Varianten zur Verfügung steht.

Aus dieser Abwägung der Vor- und Nachteil stechen die Varianten c) und d) positiv hervor. Konfiguration c) bedarf dabei weniger Chipfläche und Energie, d) bietet stattdessen mehr Freiheiten beim Scheduling.

3.2.1.2 Heap

Der gemeinsame Heap-Adressraum kann in mehreren verteilten oder einem zentralen Speicher abgelegt werden. Mögliche Varianten sind wie in Abb. 3.2 dargestellt:

a) Der Speicher ist auf die Prozessorkerne verteilt. Da jeder Kern auf den gesamten Adressraum zugreifen können muss, ist ein Verbindungsnetzwerk nötig.

b) Der Heap ist in einem zentralen Speicher abgelegt.

Im Vergleich zum Stack sind Zugriffe auf den Heap seltener, da nur ein Teil der Bytecodes darauf zugreift. Ob Heap-Caches notwendig sind, muss anhand einer konkreten Prozessorarchitektur bewertet werden.

Bei lokalen Heap-Caches —z. B. einem pro Kern— ist deren Kohärenz sicherzustellen. Dazu können einerseits die klassischen Protokolle auf Basis von Verzeichnissen oder Snooping eingesetzt werden (s. Abschn. 2.3.4). Die JVM-Spezifikation lässt aber auch die in JopCMP umgesetzte Alternative zu (vgl. Abschn. 2.6.2.1): bei Betreten eines kritischen

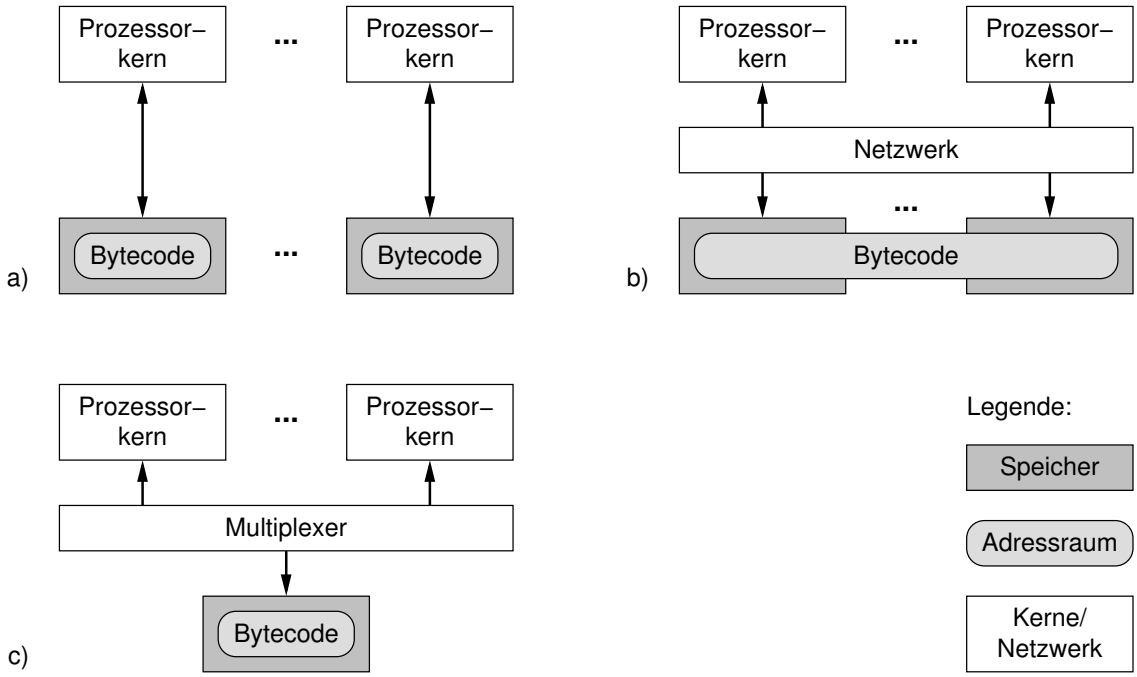

Abbildung 3.3: Mögliche Anordnungen (a–c) des Bytecode-Adressraums

Abschnitts (durch den Bytecode `monitorenter`) oder Zugriff auf eine `volatile`-Variable invalidiert der auslösende Kern seinen eigenen Write-Through-Cache.

Variante a) bietet prinzipiell eine höhere Speicherbandbreite als b). Werden die Speicher jedoch — z. B. aufgrund ihrer Größe — als externe Speicher realisiert, dann ist der zusätzliche Bedarf an Chipfläche und Energie für mehrere Speichercontroller und vielen I/O-Pins von Nachteil.

Wie viel Speicherbandbreite tatsächlich benötigt wird und ob Caches notwendig sind, müssen entsprechende Analysen anhand einer konkreten Prozessorarchitektur ermitteln.

3.2.1.3 Bytecode

Der gemeinsame Bytecode-Adressraum kann in mehreren verteilten oder einem zentralen Speicher abgelegt werden. Mögliche Varianten sind wie in Abb. 3.3 dargestellt:

a) Da auf den Bytecode nur lesend zugegriffen wird, besteht die Möglichkeit, je Kern ein Duplikat des Bytecodes in einem lokalen Speicher vorzuhalten.

b) Der Speicher ist auf die Prozessorkerne verteilt. Da jeder Kern auf den gesamten Adressraum zugreifen können muss, ist ein Verbindungsnetzwerk nötig.

c) Der Bytecode ist in einem zentralen Speicher abgelegt.

Da bei jeder Befehlsausführung auf den Bytecode zugegriffen werden muss, ist eine kurze Latenz beim Speicherzugriff essentiell. Caches sind (ggf. in mehreren Ebenen) empfehlenswert:

- in Variante a), sofern externe Speicher eingesetzt werden, deren Speichercontroller zusätzliche Latenzen bedingen,
- in Variante b), um die Latenzen des Verbindungsnetzwerks zu verstecken sowie
- in Variante c), da sich dort mehrere Kerne die Speicherbandbreite teilen müssen.

Da Bytecode nur gelesen werden kann, existiert kein Kohärenzproblem bei Bytecode-Caches.

Variante a) erfordert insbesondere bei hoher Kernanzahl viel mehr Chipfläche und Energie als die anderen beiden Konfigurationen. Sie ist daher zu vermeiden. Variante b) bietet potentiell mehr Speicherbandbreite als c). Dem Gegenüber stehen jedoch die schon beim Heap angeführten Nachteile beim Einsatz von externem Speicher.

3.2.1.4 Hybride Varianten

Die bisherigen Erläuterungen betrachteten zunächst den Fall, dass für jeden Adressraumtyp separate Speicher vorgesehen sind: Entweder enthielt ein Speicher nur Stacks oder ein Speicher war nur dem Bytecode oder nur dem Heap zugeordnet. Ein Speicher kann darüber hinaus natürlich auch Adressräume verschiedenen Typs enthalten.

In Abb. 3.4 sind 2 Varianten dargestellt, die alle 3 Adressraumtypen integrieren. Sie sind von den o. g. Konfigurationen abgeleitet:

a) Der Speicher ist auf die Kerne verteilt. Heap- und Bytecode-Adressraum erstrecken sich über alle Speicher. Jeder Speicher enthält einen oder mehrere Stacks. Als Basis dienten die Varianten Stack-d), Heap-a) und Bytecode-b).

b) Es wird ein zentraler Speicher verwendet. Dieser enthält einen Bytecode-, einen Heap und mehrere Stack-Adressräume. Als Basis dienten die Varianten Stack-e), Heap-b) und Bytecode-c).

Analog sind weitere Varianten denkbar, die lediglich 2 Adressraumtypen integrieren.

Die hybriden Varianten vereinen die bereits besprochenen Vor- und Nachteile der Basisvarianten. Der Vorteil von hybriden Varianten gegenüber getrennten Speichern besteht darin, dass den Adressräumen je nach Bedarf dynamisch Speicher zugewiesen werden kann. Der SHAP-µP macht bspw. davon Gebrauch, indem Heap und Bytecode zusammen in einem Speicher verwaltet werden.

Auch bei den hybriden Varianten ist der Einsatz von Caches (ggf. in mehreren Ebenen) zu erwägen. Neben getrennten Caches für Stack, Bytecode und Heap sind auch gemeinsame Befehls- und Daten-Caches denkbar. Vorteilhaft sind gemeinsame Caches für externe

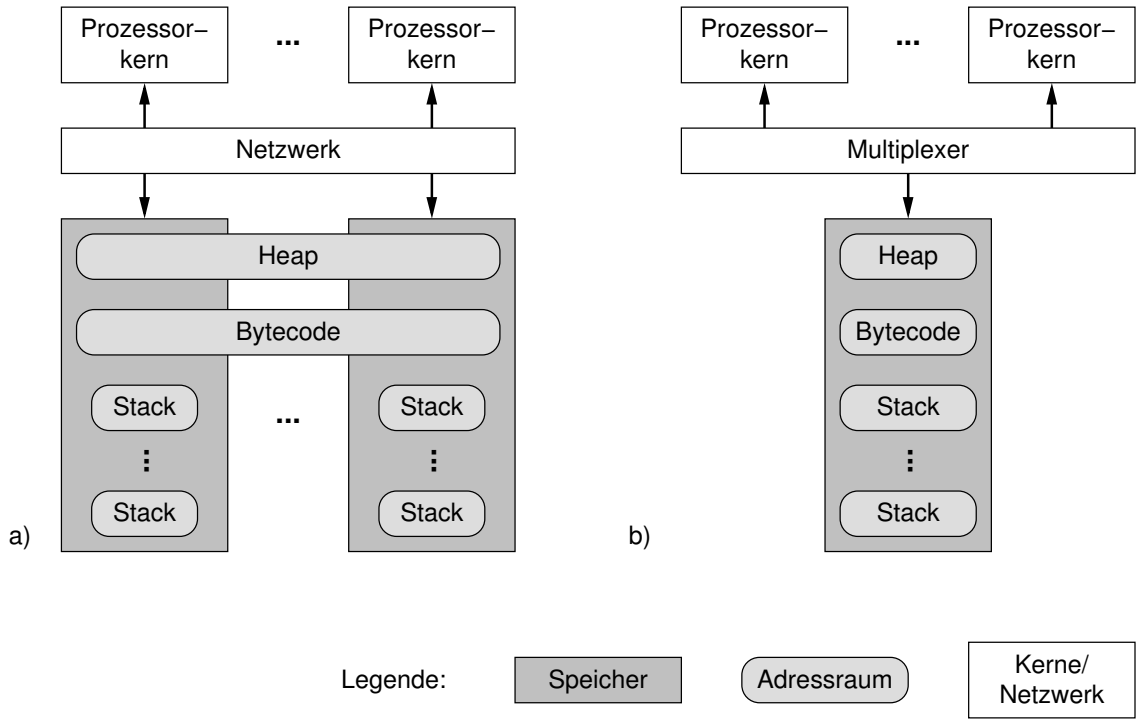

Abbildung 3.4: Ausgewählte hybride Anordnungen (a und b) der Adressräume

Speicher auf DRAM-Basis, um die Speicherzugriffslatenz sowohl der Speicher als auch des Speichercontrollers zu verdecken.

3.2.1.5 Parallele Ausführung von Java-Programmen

Sollen mehrere Java-Programme parallel ausgeführt werden, so gelten die bisherigen Überlegungen weiterhin:

- Jeder Thread hat nach wie vor seinen privaten Stack.
- Jedes Programm hat seinen eigenen Heap und damit einen eigenen Satz von Objekten. Die JVM-Spezifikation (s. Abschn. 2.1.2) gestattet lediglich Zugriff auf die eigenen Objekte und verbietet einen Speicherzugriff auf beliebige Adressen. Damit können die Objekte verschiedener Programme auf einem gemeinsamen Heap alloziert werden, da ein Austausch von Objektreferenzen zwischen Programmen laut JVM-Spezifikation nicht möglich ist.
- Jedes Programm besitzt seinen eigenen Bytecode. Das API ist jedoch für alle gleich. Die JVM-Spezifikation gestattet wiederum nur Zugriff auf den eigenen Bytecode oder das API. Damit kann analog der Bytecode verschiedener Programme in einem Bytecode-Adressraum verwaltet werden. Dies erspart zudem die mehrfache, getrennte Speicherung des API-Bytecodes für jedes Programm.

3.2.2 Speicherverwaltung

3.2.2.1 Garbage-Collection

Die Anordnung der Adressräume für die Stacks und den Heap wirkt sich auch auf die Leistungsfähigkeit der automatischen Speicherverwaltung mittels Garbage-Collection aus.

Verteilte Speicher für Stack und / oder Heap bieten im Gegensatz zum zentralen Speicher die Möglichkeit diese parallel nach Objektreferenzen zu durchsuchen. Ein nennenswerter positiver Effekt ist jedoch nur zu erwarten, wenn die Belegung der verteilten Speicher ausgewogen ist. Für das parallele Durchsuchen wird zudem mehr Chipfläche und / oder Energie für die Steuerung benötigt:

- Bei einer Hardwarerealisierung des Zustandsautomaten ist ein Schaltwerk pro verteiltem Speicher erforderlich.
- Bei einer Softwarerealisierung ist analog ein Thread pro verteiltem Speicher erforderlich. Diese Threads müssen parallel auf allen Prozessorkernen[1] ausgeführt werden.

Verteilter Speicher für den Heap ermöglicht außerdem die parallele Objektfreigabe und Kompaktierung des freien Speichers. Eine nennenswerte Beschleunigung ist aber wiederum nur bei Ausgewogenheit zu erwarten. Außerdem werden analog oben mehr Chipfläche und / oder Energie benötigt.

3.2.2.2 Objektbezogene Adressierung

Der Zugriff auf den Heap erfolgt laut JVM-Spezifikation (s. Abschn. 2.1.2) immer objektbezogen. Adressgeneratoren (AGU, engl.: address generation unit) müssen dazu die Basisadresse des Objekts mit dem Offset der Objektvariable verrechnen.

Bei einem verteilten Heap-Speicher (Variante a) ist eine AGU pro Prozessorkern notwendig, um den zuständigen Speicher zu bestimmen. Bei einem zentralen Heap-Speicher (Variante b) kann auch eine zentrale AGU eingesetzt werden.

Für den Fall, dass für die Adressberechnung zunächst die Basisadresse anhand der Objektreferenz nachgeschlagen werden muss (Bsp. SHAP-µP), ist zusätzlich der Einsatz eines TLB zu erwägen. Ein TLB je Prozessorkern erscheint vorteilhaft, da zu erwarten ist, dass die Threads größtenteils auf verschiedenen Objekten operieren. Außerdem ist die Kohärenz der TLBs sicherzustellen, falls der GC Objekte im Speicher verschieben kann.

[1] Die Anzahl an verteilten Speichern stimmt i. Allg. mit der Anzahl der Prozessorkerne überein.

3.2.3 I/O-Anbindung

Aufgrund der JVM-Spezifikation kann die Anbindung von I/O-Komponenten unabhängig von der der Programm- und Datenspeicher betrachtet werden. Der Zugriff auf I/O-Komponenten obliegt einzig der konkreten JRE und wird typischerweise über `native`-Methoden bereitgestellt. Alternativ wäre aber auch ein Zugriff über spezielle Variablen (z. B. Datenfelder) denkbar.

Bei der Ausgestaltung der I/O-Anbindung muss abgewogen werden zwischen Chipflächenbedarf und Energieaufwand für die Verbindungsstruktur einerseits und Flexibilität für den Scheduler andererseits:

- Der kleinste Aufwand ist zu erwarten, wenn eine I/O-Komponente nur an einen Prozessorkern angeschlossen ist, da dadurch Signalleitungen lokal (und somit kurz) realisierbar sind und keine Verbindungen zu anderen Kernen anfallen. In diesem Fall ist jedoch das Scheduling eingeschränkt, da der entsprechende Verwalter-Thread für die Komponenten nur auf diesem einen Kern ausgeführt werden kann.
- Im Sinne eines flexiblen Schedulings ist es zweckmäßiger, wenn alle Kerne gleichberechtigt auf die I/O-Komponenten zugreifen können. Die Verbindungsstruktur sollte anhand der Anforderungen an Latenz und Durchsatz der I/O-Operationen ausgewählt werden.

Für I/O-Komponenten mit hohem Datendurchsatz sollte die direkte Anbindung an den Heap-Speicher per DMA (engl.: direct memory access) erfolgen, auch um die Prozessorkerne zu entlasten. Damit reduzieren sich auch die Anforderungen an die Verbindungsstruktur (sofern vorhanden).

3.2.4 Thread-Scheduling

Für das Scheduling können die allgemein bekannten Strategien eingesetzt werden. Von Interesse sind hier nur die Auswirkungen der Anbindung der Stack-Speicher und I/O-Komponenten.

Werden die verteilten Stack-Speicher jeweils nur lokal an einen Prozessorkern angebunden (Varianten a und c), dann erfolgt das Scheduling zweistufig:

1. Beim Start des Threads muss zunächst ein geeigneter Kern ausgewählt werden. Dieser wird bis zum Beenden des Threads beibehalten.
2. Je Prozessorkern erfolgt das Scheduling zwischen „seinen“ Threads mit einer geeigneten Strategie. Diese Stufe entfällt bei Variante Stack-a).

Ist eine I/O-Komponente nur an einen oder an ausgewählte Prozessorkerne angeschlossen, so muss dies in der Scheduling-Strategie berücksichtigt werden.

3.2.5 Thread-Synchronisation

Die Verwaltung der Monitore für die Absicherung der kritischen Abschnitte kann wahlweise per Software oder über eine dedizierte Hardwarekomponente erfolgen:

- Bei der Softwareimplementierung werden die Zustandsinformationen des Monitors sinnvollerweise im Speicherbereich der Objekte abgelegt. Damit wirkt sich hier aber ebenso die Ausgestaltung der Speicheranbindung aus.

  Die konkret verwendeten Verbindungsnetzwerke zwischen den Prozessorkernen und den Speichern müssen die atomaren Operationen Read-Modify-Write oder Compare-And-Swap unterstützen, damit der Zustand eines Monitors nur von einem Thread zur gleichen Zeit geändert werden kann und somit dieser allein den kritischen Abschnitt (wiederholt) betritt. Diese atomaren Operationen sind ebenfalls bei der Implementierung der Heap-Caches (sofern vorhanden) zu berücksichtigen.

- Wird nur eine kleine Anzahl von Objekten auch als Monitor verwendet, ist auch eine Verwaltung der Zustandsinformationen in einer dedizierten Hardwarekomponente eine Überlegung wert. Die maximale Anzahl gleichzeitig belegter Monitore ist dabei nur durch die zugestandene Chipfläche und Verlustleistung begrenzt.

  Im einfachsten Fall werden nur Zustandsinformationen für einen einzigen, globalen Monitor verwaltet. Nachteilig ist jedoch, dass in diesem Fall das Betreten verschiedener, voneinander unabhängiger kritischer Abschnitte nicht möglich ist. Die Verarbeitungsleistung sinkt, da mehrere Threads verschiedene (unabhängige) Betriebsmittel trotzdem nur nacheinander akquirieren können. Nach der Freigabe des Monitors kann natürlich auf einen anderen umgeschaltet werden.

Da Synchronisation zwangsweise zu einer seriellen statt parallelen Abarbeitung des Programms führt, ist sie zu vermeiden. Chipfläche und Verlustleistung für seltene Synchronisationen zu investieren ist damit ineffizient. Einer Softwareimplementierung sollte daher der Vorzug gegeben werden.

3.2.6 Schnittstellen für Test und Diagnose

3.2.6.1 Klassenbibliothek

Parallel zur Erweiterung des Java-Prozessors auf mehrere Kerne sind auch Anpassungen der Klassenbibliothek zu erwarten: z. B. geänderter Start des Systems, erweiterter Einfluss auf das Thread-Scheduling. Diese Anpassungen sind zu testen.

Da die Anpassungen erfahrungsgemäß nur von geringem Umfang sind, sollen hier nur einfache Verfahren diskutiert werden:

Die strukturierte Ausnahmebehandlung ist Bestandteil der Programmiersprache Java und steht damit jederzeit — sowohl im Prototyp als auch im finalen Produkt — zur Verfügung. Sie ermöglicht die zuverlässige Überprüfung des Zustands einer Teilkomponente zur Laufzeit. Sie eignet sich jedoch kaum zur Ermittlung der Ursache im Fehlerfall.

Der Software-Debugger ist besser für die Suche nach der Fehlerursache geeignet. Er erfordert zusätzliche Hardwareschnittstellen um bspw. das Programm im Einzelschritt zu durchlaufen oder den Speicherinhalt anzuzeigen. Dafür kann er auch für die spätere Programmentwicklung genutzt werden.

Der Test des Anwenderprogramms ist nicht Gegenstand dieser Arbeit.

3.2.6.2 Mehrkernprozessor

Für den Test des Mehrkernprozessors stehen die Verfahren aus Abschn. 2.4 zur Verfügung. Da kein Schaltkreis gefertigt werden soll und die Entwurfswerkzeuge als korrekt angenommen werden, sind nur Entwurfsfehler Gegenstand dieser Arbeit.

Getestet werden soll eine synthesefähige Hardwarebeschreibung des Prozessors. Diese berücksichtigt im Gegensatz zu einem abstrakten Prozessormodell auch das zeitliche Verhalten (u. a. Taktfrequenz) und stellt damit eine realistische Basis für die Leistungsbewertung dar. Im Fokus steht insbesondere das Gesamtsystem, da angenommen wird, dass die aus dem Einkernprozessor übernommenen Teilkomponenten bereits getestet wurden.

Der Test der Hardwarebeschreibung mittels Simulation ist zeitlich intensiv und damit unpraktikabel. Dafür können zeitgleich alle internen Signale überwacht und deren Verlauf aufgezeichnet werden.

Im Gegensatz dazu bietet das FPGA-Prototyping eine Ausführung des Testprogramms in Echtzeit und die Einbeziehung der realen Umgebung. Jedoch ist die Beobachtbarkeit eingeschränkt. Die Genauigkeit ist vergleichbar zur Simulation. Für die Beobachtung und Überprüfung stehen beim FPGA-Prototyping folgende Verfahren bereit:

Fehleranzeigen: Die internen Zustände können zur Laufzeit durch integrierte Vergleicherkomponenten überwacht werden. Eine Ausnahme bilden lediglich die nicht initialisierten Speicherbereiche.

Ein erkannter Fehler wird bspw. in einem Flip-Flop dauerhaft zwischengespeichert und mittels externer Anzeigen (z. B. LEDs) sichtbar gemacht. Die Fehlerspeicher können gleichzeitig als Triggerimpulse für Logikanalysatoren und Trace-Architekturen dienen, die im Folgenden aufgeführt sind.

Externer Logikanalysator: Für die Aufzeichnung von Signalverläufen können externe Logikanalysatoren angeschlossen werden. Dieses Verfahren eignet sich jedoch

nur begrenzt für Mehrkernprozessoren, da häufig nur wenige freie Pins am FPGA zur Verfügung stehen, um interne Signale zu beobachten.

Interner Logikanalysator: Wird der Logikanalysator in den Schaltkreis integriert, können theoretisch alle Signale beobachtet werden. Praktisch ist jedoch die Aufzeichnungskapazität begrenzt:

- Wird der Signalverlauf zunächst in einem internen Speicher abgelegt, so ist der Beobachtungszeitraum jedoch durch dessen Größe stark begrenzt. Erst nach Übertragung der Daten zum PC kann eine neue Beobachtung gestartet werden.
- Werden die aufgezeichneten Daten unverzüglich zum PC (o. ä.) übertragen (Trace-Architektur), dann begrenzt die Bandbreite der dafür notwendigen Hochgeschwindigkeitsschnittstelle die Anzahl der gleichzeitig überwachbaren Signale. Aufgrund der kontinuierlichen Übertragung können dafür auch lange Zeiträume protokolliert werden.

Nach der Übertragung der aufgezeichneten Daten zum PC, können diese dort überprüft werden.

3.3 SHAP-Mehrkernarchitektur

In diesem Abschnitt wird eine Mehrkernarchitektur auf Basis der SHAP-Plattform (mit einem Kern) entwickelt. Für jeden Aspekt des oben besprochenen Entwurfsraums wird eine für SHAP geeignete Variante ausgewählt und die getroffenen Designentscheidungen anhand von SHAP-spezifischen Analysen begründet. In diesem Zusammenhang wird auf neue und geänderte Komponenten der SHAP-Plattform eingegangen.

3.3.1 Speicherarchitektur

3.3.1.1 Stack

Innerhalb des SHAP-µP ist der Stack-Speicher direkt an den Prozessorkern gekoppelt. Er verwaltet zudem mehrere Stacks (s. Abschn. 2.2.4). In Fortführung dieses Konzeptes kommen somit die Varianten c) und d) aus dem Stack-Entwurfsraum (s. Abschn. 3.2.1.1) in Frage.

Die Entscheidung fiel zugunsten der Variante c) aus, d. h. jeder Kern erhält seinen eigenen, lokalen Stack-Speicher mit jeweils mehreren Stacks. Ausschlaggebend war der Vorteil gegenüber Variante d), dass kein Verbindungsnetzwerk und keine Stack-Caches erforderlich sind und der damit verbundene Bedarf an Chipfläche und Energie entfällt.

Der Speicher enthält die Stacks immer komplett, sodass beim Stack-Überlauf / -Unterlauf oder beim Thread-Wechsel keine Daten von / zum Heap übertragen werden müssen.

Der Nachteil der Variante c) bestand jedoch darin, dass das Scheduling eingeschränkt ist, da jeder Kern nur auf seine Stacks zugreifen kann. Für SHAP bietet sich prinzipiell folgender Ausweg an: Der Stack-Speicher ist aufgrund des GC bereits als Dual-Port-Speicher mit 2 getrennten Schnittstellen realisiert. Über die GC-Schnittstelle wäre der Transfer eines Stacks von einem Stack-Speicher zu einem anderen möglich, um die Ausführung des Threads auf einem anderen Kern fortzusetzen. Dieser theoretische Nachteil zeigte in der praktischen Erprobung der SHAP-Mehrkernarchitektur jedoch keine Relevanz, sodass ein Stack-Transfer nicht weiter verfolgt wurde.

An den Stack-Speichern und deren Anbindung an die Prozessorkerne sind somit keine Anpassungen notwendig. Auf die Änderungen am Scheduling wird in Abschn. 3.3.3 näher eingegangen.

3.3.1.2 Bytecode

Der SHAP-µP sieht keinen getrennten Speicher für den Bytecode vor. Der Bytecode ist stattdessen in speziellen Objekten auf dem Heap gespeichert (s. Abschn. 2.2.4). Dieses Konzept soll nicht geändert werden. Damit ist die Anordnung des Bytecode-Speichers an die des Heap-Speichers geknüpft, die im folgenden Abschnitt diskutiert wird.

Der schnelle Zugriff auf den Bytecode mit kurzer Latenz wird nach wie vor durch einen Methoden-Cache bereitgestellt. Dazu wird je Kern ein eigener Cache eingesetzt, analog dem L1-Cache klassischer Mehrkernprozessoren (vgl. Abschn. 2.6.3).

Die Weiterentwicklung des Methoden-Caches erfolgte durch Hrn. Preußer ([PZS07]). Die dort ermittelten Hit-Raten der favorisierten Strategie „BumpRetInv“ wurden als geeignet für die Mehrkernarchitektur erachtet. Andere Strategien werden daher nicht verfolgt.

3.3.1.3 Heap

Für die SHAP-Mehrkernarchitektur stehen prinzipiell beide im Rahmen des Entwurfsraums (s. Abschn. 3.2.1.2) diskutierten Varianten der physischen Heap-Anordnung zur Auswahl: a) verteilter oder b) zentraler Heap-Speicher. Die Entscheidung zugunsten einer Variante sowie dem Einsatz von Caches soll anhand der benötigten Speicherbandbreite getroffen werden. Dazu wird im Folgenden die benötigte Speicherbandbreite auf dem SHAP-Einkernprozessor für verschiedene Programme analysiert.

Methodik der Analyse Die Analyse der Speicherbandbreite umfasst nur Datenzugriffe oder Allokationen. Es wird ein effizientes Methoden-Caching vorausgesetzt, welches nur einer geringen Bandbreite bedarf.

Anstatt die benötigte Speicherbandbreite direkt zu messen, wird folgende andere Herangehensweise genutzt. Sie hat den Vorteil, dass auch theoretische Formen der Speicheranbindung analysiert werden können.

Der Prozessorkern — und zukünftig auch die anderen Kerne — sowie der Speichermanager inklusive Speicherbus werden mit derselben Taktfrequenz betrieben. Somit ist die Auslastung der verfügbaren Speicherbandbreite u definiert als:

$$u = m_{\text{Total}} / e_{\text{Total}} \ . \tag{3.1}$$

Dabei bezeichnet:

- e_{Total} die Summe von Taktzyklen für die Ausführung des gesamten Programms und
- m_{Total} die Summe der Taktzyklen, die der Speicherbus für die Bearbeitung der Speicherzugriffe beschäftigt ist, sodass er keinen neuen Zugriff akzeptiert.

Die Summen werden berechnet, indem für jeden Bytecode k aus der Menge aller Bytecodes K die Taktzyklen für Ausführung e_k und Speicherzugriffe m_k aufaddiert werden:

$$\begin{aligned} m_{\text{Total}} &= \sum_{k \in K} n_k \cdot m_k \ , \\ e_{\text{Total}} &= \sum_{k \in K} n_k \cdot e_k \ . \end{aligned}$$

Die Werte e_k und m_k können durch Analyse des SHAP-Mikrocodes mittels Zählung der Taktzyklen bestimmt werden. Sie haben entweder einen festen Wert (für jeden Bytecode) oder hängen vom konkreten Heap-Speicher und dessen Anbindung (Speicherbus, Speichercontroller, Caches) ab. Dazu werden im übernächsten Abschnitt verschiedene Systemkonfigurationen untersucht. Allgemein ist festzustellen:

- e_k hängt von der Leselatenz des Heap-Speichers und seiner Anbindung ab, da die Ausführung des Programm-Threads solange angehalten wird, bis das Datenwort aus dem Speicher gelesen ist.
- m_k hängt vom Durchsatz des Heap-Speichers, dessen Speichercontroller und dem Speicherbus ab. Unterstützen alle 3 Komponenten Pipelining, so geht jeder Speicherzugriff nur mit einem Takt in die Berechnung ein, da jede Pipelinestufe entsprechend nur für einen Takt belegt ist. Ist dagegen kein Pipelining verfügbar, geht in die Berechnung die gesamte Dauer der Lese- respektive Schreibzugriffe ein.

Die Ausführungszeiten einzelner Bytecodes hängen zusätzlich von deren Operanden ab, z. B. Größe des zu allozierenden Objektes. Dies wurde in der Zählung der Taktzyklen entsprechend berücksichtigt. Die fraglichen Operanden konnten durch eine Quelltextanalyse der Programme im Voraus bestimmt werden.

Der Wert n_k gibt an, wie oft der Bytecode k ausgeführt wird und kann ersetzt werden durch seine dynamische Befehlshäufigkeit $p_k = n_k / n_{\text{Total}}$. Die Einführung der Befehlshäufigkeit p_k ermöglicht einen einfacheren Vergleich von Programmen untereinander und mit theoretischen Bytecode-Verteilungen. Abschließend kann Gl. 3.1 umgeformt werden zu:

$$u = \frac{\sum_{k \in K} p_k \cdot m_k}{\sum_{k \in K} p_k \cdot e_k} \,. \tag{3.2}$$

Dynamische Befehlshäufigkeiten Die dynamischen Befehlshäufigkeiten p_k wurden auf dem SHAP-µP mit einem Kern für verschiedene Programme bestimmt:

- **Queens** sucht nach allen Lösungen des N-Queens-Problem und ist Teil der Benchmark-Suite „JemBench“ [SPU10].
- **Lift** ist eine industrielle Applikation und ebenfalls Teil von JemBench.
- **FScriptME** ist ein Skript-Interpreter der selbst in Java programmiert ist[2]. Das hier verwendete Skript berechnet die Länge der Collatz-Reihe für die ersten 43 Fibonacci-Zahlen.
- **SparseMatmultInt** (SMMI) multipliziert eine dünn besetzte Matrix mit einem Vektor. Hierfür wurde der Benchmark „SparseMatmult“ aus der „Multi-threaded“ Benchmark-Suite des Java Grande Forum[3] angepasst, siehe Abschn. 4.2.1. Im Gegensatz zu den anderen Programmen wird hier nur die Multiplikationsschleife betrachtet.

Die Tab. 3.1 listet die Häufigkeiten entsprechend ihrer Befehlskategorie auf, wie sie von El-Kharashi u. a. definiert sind [EKEL00]. Ein Vergleich mit den Ergebnissen von El-Kharashi (ebenso in der Tabelle enthalten) zeigt, dass Lift und FScriptME ähnliche Bytecodehäufigkeiten aufweisen wie typische Java-Programme.

Systemkonfigurationen Zwecks Bestimmung der Werte für e_k und m_k werden nun verschiedene Systemkonfigurationen untersucht. Die resultierende Auslastung der verfügbaren Speicherbandbreite u ist ebenfalls in Tab. 3.1 enthalten.

Zuerst wurde die konkrete Implementierung des SHAP-µP (mit einem Prozessorkern) auf dem XUPV5-Entwicklungsboard der Firma Xilinx evaluiert. Dieses Board bietet einen ZBT-SRAM mit einem 32 Bit breiten Datenbus, sodass sich ein Durchsatz von einem Speicherzugriff pro Takt ergibt. Die Leselatenz von 3 Takten beeinflusst nur den Wert e_k. Es wurde der bereits vorhandene TLB mit nur einem Eintrag berücksichtigt.

[2] http://fscript.sourceforge.net/

[3] http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/threads.html

Tabelle 3.1: Dynamische Befehlshäufigkeiten und belegte Speicherbandbreite

	Programm				Mix
	Queens	Lift	FScriptME	SMMI	El-Kharashi
Dyn. Befehlshäufigkeit n. Kategorie					
Scalar Data Transfer	46.85%	33.91%	51.56%	34.48%	43.34%
Stack	12.42%	16.60%	2.81%	3.45%	10.11%
ALU	26.66%	16.50%	9.26%	10.34%	10.70%
Control Flow	8.09%	13.34%	18.30%	3.45%	12.58%
Object: Method Handling	4.43%	1.43%	1.66%	0.00%	5.25%
Object: Field Access	0.78%	12.32%	6.55%	24.14%	10.18%
Object: Object Creation	0.00%	0.00%	0.09%	0.00%	0.12%
Object: Type Checking	0.00%	0.00%	0.03%	0.00%	0.11%
Object: Array Specific	0.69%	5.89%	8.00%	24.14%	7.43%
Advanced	0.08%	0.00%	1.22%	0.00%	0.16%
Benutzte Speicherbandbreite (1 Thread)					
XUPV5 mit einem Kern	2.18%	9.89%	7.26%	12.70%	$\approx$ 10%
ohne Pipelining des Speicherzugriffs	5.69%	28.26%	19.76%	37.33%	$\approx$ 29%
mit großem TLB (100% Hit-Rate)	1.96%	6.12%	5.73%	8.94%	$\approx$ 7%
mit Objekt-Cache (50% Hit-Rate)	1.71%	4.06%	4.08%	—	$\approx$ 5%
mit Objekt-Cache (90% Hit-Rate)	1.47%	2.19%	2.55%	—	$\approx$ 3%
DE2 mit einem Kern	6.50%	22.12%	17.74%	30.43%	$\approx$ 21%

Das Queens-Programm benutzt fast nur den Stack für die Berechnung. Nur ein kleiner Anteil der verfügbaren Speicherbandbreite (2%) wird benötigt, sodass ein zentraler Speicher ausreichend ist. Die anderen Programme verwenden 7% bis 13% der Bandbreite. Selbst der Applikations-Mix von El-Kharashi benötigt nur etwa 10%. Die Werte für diesen Mix konnten nur näherungsweise berechnet werden, da Befehlshäufigkeiten nur für die häufigsten sowie die langsamsten Bytecodes in der Literatur angegeben sind.

Zum Vergleich: Würde der Speicher auf dem XUPV5 kein Pipelining unterstützen, wäre der Speichercontroller bei einem Lesezugriff entsprechend der Latenz für einen Lesezugriff, also 3 statt einem Takt, beschäftigt. Dadurch wird die Auslastung der Speicherbandbreite effektiv um den Faktor 3 erhöht. (Schreibzugriffe sind nicht betroffen, dafür seltener.)

Die indirekte Adressierung der Objekte auf der SHAP-Plattform benötigt jeweils einen zusätzlichen Speicherzugriff, um die Basisadresse aus der Objekttabelle im Speicher zu laden (vgl. Abschn. 2.2). Dieser Zugriff kann durch große TLBs (mit vielen Einträgen) abgefangen werden. Die Matrixmultiplikation benötigt bspw. mindestens 8 Einträge. Für die Lift- und FScriptME-Benchmarks wären noch mehr Einträge zu erwarten. Der Einsatz eines großen TLBs würde die Auslastung der Speicherbandbreite auf 6–9% reduzieren.

Die benötigte Speicherbandbreite kann zusätzlich zum großen TLB mittels eines Objekt-Caches für jeden Prozessorkern reduziert werden, jedoch mit den bereits in Ab-

schn. 3.2.1.2 genannten Nachteilen. Werden durchschnittliche Hit-Raten von 50% erzielt, reduziert sich die Auslastung auf 4–5%, bei einer Hit-Rate von 90% sogar auf 2–3%. Diese Hit-Raten erscheinen zu optimistisch für die Matrixmultiplikation, daher ist kein Wert in der Tabelle angegeben.

Analysiert wurde ebenfalls die konkrete Implementierung des SHAP-µP auf dem DE2 Development-Board von der Firma Altera. Der SRAM-Speicher auf diesem Board besitzt nur einen 16 Bit breiten Datenbus, sodass jeder Speicherzugriff den Speicherbus für 2 Takte belegt. Folglich sind die Werte m_k doppelt so groß, sodass auch die Auslastung der Speicherbandbreite sich gegenüber dem XUPV5 etwa verdoppelt.

Die Zählung der Taktzyklen führte noch zu einem sekundären Ergebnis. Bei Kopieroperationen zwischen 2 Objekten, z. B. bei Operationen auf Zeichenketten, ist ein TLB mit nur einem Eintrag nutzlos. Bei jedem Kopierschritt verdrängen sich die Basisadressen der Objekte jeweils gegenseitig. Daher wird im Folgenden ein TLB mit 2 Einträgen und der Strategie „Least-Recently Used“ eingesetzt.

Fazit Zusammenfassend kann gesagt werden, dass ein zentraler Heap-Speicher für den SHAP-Mehrkernprozessor ausreichend ist, sofern Speicherbus, Speichercontroller sowie der Speicher selbst Pipelining unterstützen. Sowohl Objekt-Caches als auch große TLBs sollten für bis zu 10 Kerne nicht notwendig sein. Es wurde daher Variante b) für die physische Anordnung des Heap-Speichers gewählt.

3.3.1.4 Mehrport-Speichermanager

Wie bereits in Abschn. 2.2 beschrieben, besaß der Speichermanager des SHAP-µP von 2007 nur einen Kommando-Port. Der Zugriff darauf erfolgte im Multiplex-Betrieb. Die Fortführung dieses Prinzips für die Anbindung vieler Komponenten, wie Prozessorkerne und DMA-Geräte, ist jedoch nicht effizient. Zwei gravierende Nachteile wären präsent:

- Die Arbitrierung zwischen den Komponenten müsste atomare Operationen aus Objektaktivierung und ggf. mehrfachem Lese- und Schreibzugriff vorsehen, da nur eine Referenz aktiviert werden kann. Dies führt jedoch zu einer ineffizienten Ausnutzung der Speicherbandbreite, da kein Pipelining der Lese- und Schreibzugriffe verschiedener Komponenten möglich wäre.
- Außerdem wären Einsparungen bei mehrfacher Verwendung der gleichen Referenz durch einen Prozessorkern nur noch selten möglich. Durch den Wechsel zwischen den Komponenten müssten ständig Referenzen zu Basisadressen aufgelöst werden, da jede Komponente typischerweise mit einem anderen Objekt arbeitet.

Eine alternative Lösung besteht daher darin, den Speichermanager mit mehreren Kommando-Ports, einem pro zugreifender Komponente, zu versehen. Jeder Port kann nun

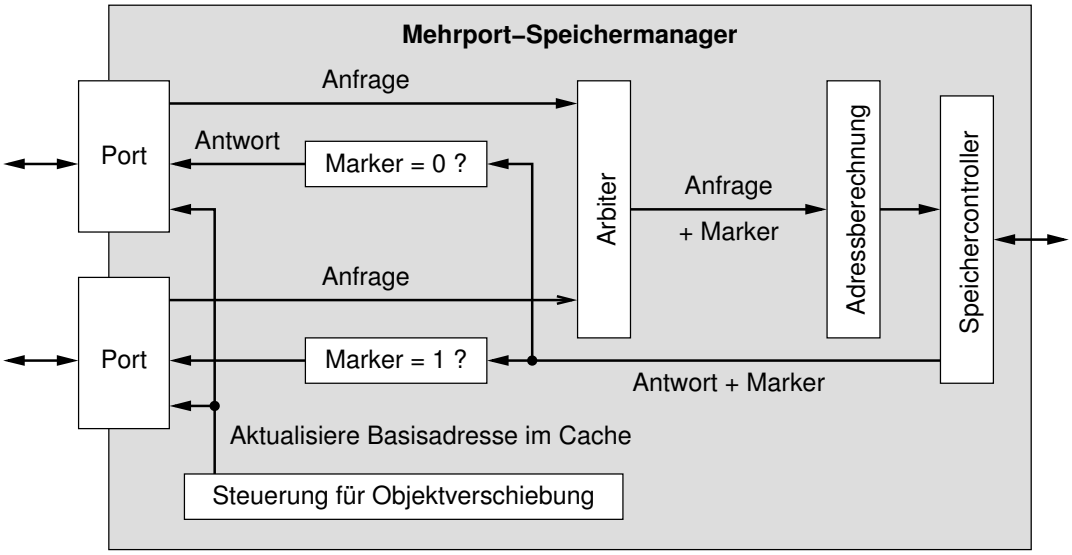

Abbildung 3.5: Schematischer Aufbau des Mehrport-Speichermanagers
(Es sind nur die für Lese- / Schreibzugriffe relevanten Komponenten dargestellt.)

seine eigene Referenz aktivieren und halten. Die Arbitrierung erfolgt erst auf der Ebene der Low-Level-Kommandos:

- Ermitteln der Basisadresse,
- Lesen und / oder Schreiben,
- Monitor betreten (s. Abschn. 3.3.4),
- Allokation von Objekten.

Der schematische Aufbau hierzu ist in Abb. 3.5 dargestellt. Der Heap-Speicher ist über einen Vollduplex-Speicherbus angebunden: In Hinrichtung übermittelt er das vom Arbiter selektierte Kommando sowie die Schreibdaten (*Anfrage*) an den Speicher und in Rückrichtung transferiert er gleichzeitig andere Lesedaten (*Antwort*) zurück an den Port. Zwecks Zuordnung von Lesedaten zu Kommando/Port wird ein *Marker* eingesetzt. Für die Allokation von Objekten sprechen die Ports direkt die Objektverwaltung des Speichermanagers an, wobei ein separater Arbiter zum Einsatz kommt.

Somit ergibt sich ein Pipelining der Speicherzugriffe verschiedener Ports und damit eine effiziente Ausnutzung der Speicherbandbreite. Sofern der Speicher ebenfalls Pipelining unterstützt, kann eine maximale Bandbreite von einem 32-Bit-Speicherzugriff pro Takt erzielt werden. Jegliche Bandbreite die nicht von den Ports genutzt wird, steht nach wie vor automatisch dem GC für das Scannen und Verschieben von Objekten zur Verfügung.

Da die Basisadressen in den einzelnen Ports zwischengespeichert werden, müssen sie nach dem Verschieben eines Objektes ggf. aktualisiert werden. Dafür ist ein weiterer Ka-

nal verantwortlich, der die betreffende Referenz und die neue Basisadresse übermittelt. Jeder Port bestimmt nun selbstständig, ob er seine zwischengespeicherte Basisadresse aktualisieren muss.

Jeder Prozessorkern ist jetzt über zwei Ports an den Speichermanager angebunden. Einer für Objektzugriffe (d. h. Allokation und Datenzugriffe) und ein zweiter für den Methoden-Cache des Kerns. Damit kann der Cache parallel zur Ausführung des Java-Bytecodes befüllt werden. Für jeden weiteren Kern werden entsprechend zwei weitere Ports hinzugefügt. Ein weiterer Port steht für DMA-Operationen bereit. Eine Beispielkonfiguration mit n Kernen ist in Abb. 3.6 auf S. 61 dargestellt.

Jeder Port erhält seinen eigenen TLB. Für Objektzugriffe wird ein TLB mit 2 Einträgen eingesetzt. Für Methoden-Cache und DMA genügt ein TLB mit einem Eintrag.

Für die Anbindung des zentralen Speichers wird somit ein gemeinsam genutzter Speicherbus eingesetzt, der Pipelining von Transaktionen (engl: pipelined transactions) unterstützt. Für die Arbitrierung wird eine zweistufige Strategie eingesetzt:

1. Je Prozessorkern wird separat zwischen Datenzugriffe und Zugriffe des Methoden-Caches arbitriert. Datenzugriffe werden priorisiert, damit während des Cachings der Bytecode weiter ausgeführt werden kann.
2. Zwischen den Prozessorkernen sowie dem Port für DMA-Operationen wird Round-Robin eingesetzt, um den einzelnen Prozessorkernen und dem DMA-Kanal einen gleichberechtigten, fairen Zugriff zu bieten.

3.3.1.5 Garbage-Collector

Verteilter GC Die auf die Kerne verteilten Stack-Speicher bilden ebenso die Basis für einen teilweise verteilten GC. Für jeden Kern wird eine eigene Scan-Logik integriert nebst einer eigenen Markierungstabelle, die für jede auf dem lokalen Stack gefundene Objektreferenz ein Markierungsbit speichert. Der sogenannte Root-Scan kann daher auf allen Kernen parallel erfolgen und somit die notwendige Zeit minimiert werden.

Eine gedachte bitweise ODER-Verknüpfung aller lokalen Markierungstabellen ergibt dann das Root-Set für den GC. Dies wird mit folgenden Schritten erreicht. Zunächst sind die Kerne und der GC über einen designierten GC-Bus mit einer Daisy-Chain-Struktur verbunden [Rei08]. Diese Struktur ist günstiger für die Verdrahtung der notwendigen Signalleitung als eine sternförmige Anbindung. Über eine Bus-Nachricht sammelt der GC in 16-Bit-Blöcken die lokalen Markierungsbits von allen Kernen ein. Die ODER-Verknüpfung wird realisiert, indem eine Nachricht losgeschickt wird, in der kein Bit gesetzt ist. Nachdem ein Kern die Nachricht empfangen hat, setzt er (aber löscht nicht) ein oder mehrere Bits entsprechend seiner lokalen Tabelle und sendet dann die modifizierte Nachricht an den nächsten Kern. Der letzte Kern leitet die fertige Nachricht zurück

an den GC. Mit mehreren solcher Nachrichten nacheinander wird die gesamte Tabelle übertragen.

Der GC besitzt ebenfalls eine eigene Markierungstabelle. Darin werden alle Markierungen von den Kernen gesammelt, sowie die Objektreferenzen markiert, die beim Scannen des Heaps gefunden wurden. Der Heap-Scan ist nicht parallelisiert, da ein zentraler Heap-Speicher eingesetzt wird.

Software-GC als Alternative Die Implementierung des GC als Hardwarekomponente kann prinzipiell so erhalten bleiben. Eine Ausführung des GC-Algorithmus auf einem SHAP-Kern als Java-Thread(s) ist denkbar, wurde aber nicht praktisch geprüft. Die zu erwartende Einsparung an Chipfläche ist eher gering, da Folgendes zu bedenken ist:

- Die Speicherverwaltung muss bereits während des Uploads des Programms samt API funktionieren. Da der Code für die Java-Klassen ebenfalls in Objekten gespeichert wird, muss deren Allokation mittels Hardwarekomponenten erfolgen.
- Das Scannen der Objekte auf dem Heap erfordert eine Vielzahl von Speicherzugriffen. Dieser Vorgang sollte weiterhin mittels einer Hardwarekomponente beschleunigt werden.
- Erst die Realisierung des Stack-Scan in Software und damit der Wegfall der separaten Speicher für Markierungstabellen bringt einen nennenswerten Gewinn an Chipfläche. Die Größe der Tabelle beträgt aktuell 1 KByte pro Kern.

Priorität Der GC hat auch in der überarbeiteten Form die kleinste Priorität beim Speicherzugriff. Er nutzt daher nur die von den Kernen nicht belegte Speicherbandbreite. Es gibt lediglich zwei Ausnahmen. Die Freigabe von Objekten erfolgt mit hoher Priorität, damit schnell wieder Heap-Speicher zur Verfügung steht. Außerdem werden Objekte immer in Teilen zu 32 Bytes verschoben. Nachdem die Verschiebeoperation mit niedriger Priorität begonnen wurde, muss sie aber atomar und damit mit hoher Priorität abgeschlossen werden.

3.3.2 I/O-Architektur

Der integrierte I/O-Bus der SHAP-Plattform von 2007 unterstützt lediglich einen, aber nicht mehrere Bus-Master (Prozessorkerne), sodass ein Wechsel zu einem anderen Bussystem notwendig ist. Bei der Auswahl eines Bussystems steht aber nicht dessen Leistungsfähigkeit, sondern vielmehr das Angebot an IP-Cores für I/O-Komponenten im Vordergrund. Für hohen Datendurchsatz sollte stattdessen DMA durch direkten Anschluss des I/O-Gerätes an einen Port des Speichermanagers genutzt werden, wie folgende Abschätzung aufzeigt.

Der Lese-/Schreibzugriff auf I/O-Komponenten vom Java-Programm aus erfolgt beim SHAP-µP durch die `native`-Methoden `ioRead()` und `ioWrite()` (vgl. Abschn. 2.2.4). Ist ein Datenpaket zwecks späterer Weiterverarbeitung in einem Puffer zwischenzuspeichern, so wird typischerweise die Programmschleife

```
for(int i=0; i<laenge; i++) {puffer[i] = ioRead(adresse);}
```

verwendet. Jede Schleifeniteration liest maximal 32 Bit in 40 Takten. Das Schreiben von 32 Bit über die Programmschleife

```
for(int i=0; i<laenge; i++) { ioWrite(puffer[i],adresse); }
```

benötigt sogar 43 Takte. Selbst wenn die Schleifen ausgerollt werden, können lediglich jeweils 6 Takte eingespart werden. Im Gegensatz zu den `native`-Methoden bietet der DMA über den Speichermanager einen Durchsatz von 32 Bit pro Takt und somit eine ca. 40-mal höhere Datentransferrate.

Für den Mehrkernprozessor wurde der Wishbone-Bus der OpenCores-Initiative ausgewählt [Ope02]. Für diesen Bus sind nicht nur eine Vielzahl von I/O-Geräten verfügbar [4], er besitzt auch ein einfach zu implementierendes Busprotokoll. Das ist vorteilhaft, denn auch eine Reihe von SHAP-spezifischen I/O-Komponenten muss an diesen Bus angeschlossen werden: der Speichermanager für die Konfiguration und Statusabfrage des GC, der Timer für die globale Systemzeit, das Config-Device für die Abfrage der Kernanzahl zur Laufzeit des Systems sowie das I/O-Gerät für den Upload des Java-Programms (UART, USB-Controller, Flash-Controller o. a.). Letzteres muss SHAP-spezifische Kommandos anbieten, damit der Bootloader unabhängig vom I/O-Gerät implementiert werden kann. Bei einem Wechsel auf ein anderes Bussystem müssten diese I/O-Komponenten entweder angepasst oder über eine Bus-Bridge angebunden werden.

Der Wishbone-Bus wird in der Variante „Shared Bus“ eingesetzt, da nur der Anschluss von I/O-Geräten mit geringer Datenrate vorgesehen ist. Außerdem benötigt der Bus so nur wenig Chipfläche.

3.3.3 Thread-Scheduling

Jeder Prozessorkern kann Java Bytecode ausführen und auf denselben, gemeinsamen Heap-Speicher (der auch den Bytecode der Klassen enthält) zugreifen. Aus dieser Sicht kann der Scheduler prinzipiell jeden Thread auf einem beliebigen Kern einplanen.

Zu beachten ist jedoch, dass die Stacks nur lokal vom jeweiligen Kern zugreifbar sind. Im aktuellen Prototypen ist ein Transfer von einem lokalen Stack-Speicher in den eines anderen Kerns jedoch nicht implementiert, sodass die Ausführung eines Threads an den

[4] opencores.org

Kern gebunden ist, auf dem er auch gestartet wurde (vgl. Abschn. 3.3.1.1). Das Scheduling besteht folglich aus 2 Stufen:

1. Die Ausführung eines Threads wird auf dem Kern gestartet, der aktuell die geringste Anzahl Threads ausführt. Bei Gleichstand wird der Kern mit der höchsten Identifikationsnummer ausgewählt.
2. Die Threads, die jeweils auf demselben Kern gestartet wurden, werden dort per Round-Robin eingeplant, wie es auch für den ursprünglichen SHAP-µP von 2007 der Fall ist (vgl. Abschn. 2.2).

Für beide Teile sind auch komplexere Strategien denkbar. Untersuchungen dazu waren aber nicht Gegenstand der vorliegenden Arbeit.

3.3.4 Thread-Synchronisation

Die Mehrkernarchitektur kann die Softwareimplementierung der Monitor-Verwaltung weitgehend von der SHAP-Plattform mit nur einem Prozessorkern übernehmen. Bei dem ursprünglichen SHAP-µP von 2007 hat dazu der Mikrocode des Bytecodes `monitorenter` den alten Zustand des Monitors aus dem Speicher gelesen, anschließend geprüft und ggf. geändert sowie den neuen Wert wieder zurückgeschrieben. Da zwischendurch kein Thread-Wechsel stattfinden konnte, war kein atomarer Speicherzugriff erforderlich.

Wie bereits in Abschn. 3.2.5 ausgeführt, erfordert die Zustandsänderung des Monitors jetzt eine atomare Lese-Schreib-Operation auf die betreffenden Objekte im Heap-Speicher. Wird jedoch das obige Schema auch für den Mehrkernprozessor angewendet, müsste mit Beginn des Lesebefehls der Speicherbus für alle anderen gesperrt und mit Abschluss des Schreibbefehls wieder freigegeben werden. Eine Alternative eröffnet der Einsatz eines zentralen Heap-Speichers: Die atomare Read-Modify-Write-Operation wird für diesen Zweck direkt im Speichercontroller statt in einem Prozessorkern ausgeführt. Auf dem XUPV5-Entwicklungsboard (wie in Abschn. 3.3.1.3) verkürzt sich dadurch die Dauer des Bytecodes `monitorenter` von 17 Takten auf 6 Takte.

Der Bytecode `monitorexit` kann nur von dem Thread ausgeführt werden, der auch den Monitor besitzt. Damit ist kein atomarer Speicherzugriff für die Zustandsänderung nötig.

3.3.5 Test des Mehrkernprozessors

Für den Nachweis der korrekten Funktion des Mehrkernprozessors müssen die Änderungen an Speichermanager, Speichercontroller, GC und I/O-Anbindung getestet werden.

Die Simulation eignet sich sehr gut für den Test

- einzelner Speicherzugriffe,
- einzelner und konkurrierender[5] atomarer Operationen auf Monitoren für die Thread-Synchronisation,
- von I/O-Zugriffen (die immer sequentiell ausgeführt werden) sowie
- der Arbiter für Speicherzugriffe, Allokation und I/O-Bus.

Für die Simulation kommen zum einen die zuständigen Komponenten Speichermanager, Speichercontroller, I/O-Controller und Arbiter in Frage. Zum anderen ist auch das Gesamtsystem zu simulieren, um die Interaktion mit den Prozessorkernen zu überprüfen.

Anders verhält es sich für den Test

- paralleler Speicherzugriffe, da diese verschachtelt in einer Pipeline ausgeführt werden, sowie
- von GC-Operationen, da diese parallel zu den Speicherzugriffen und Allokationen des Programms ausgeführt werden.

Für beide Aspekte ist eine sehr große Anzahl von Testschritten zu betrachten, sodass nur eine zufällige Testauswahl in Frage kommt. Die Simulation vieler Testschritte ist praktisch ineffizient, da sie sehr viel Zeit beansprucht: ModelSim (von Mentor Graphics) benötigt auf einem Mittelklasse-PC (Intel Core 2 Duo mit 2,67 GHz) etwa 6 Stunden, um die VHDL-Beschreibung des SHAP-µP mit einem Kern für eine 1 Milliarde Taktzyklen (Testsätze) zu simulieren. Im Vergleich dazu benötigt der FPGA-Prototyp nur 12,5 Sekunden (bei 80 MHz), sodass hier dieser verwendet werden soll.

Von den im Abschn. 3.2.6.2 vorgestellten Verfahren sind interne Logikanalysatoren in Form von Trace-Architekturen vorteilhaft:

- Integrierte Vergleicher können einfache Zusicherungen überprüfen sowie die Aufzeichnung von relevanten Signalen steuern.
- Eine lange Aufzeichnungsdauer ermöglicht die Prüfung von zeitlich weit auseinander liegender Aktionen, wie Allokation und Freigabe von Objekten.
- Die Aufzeichnung ist non-intrusive, d. h. sie beeinflusst nicht die Ausführung der Testschritte. Diese Eigenschaft ist notwendig, um Fehler aufgrund paralleler Zugriffe auf gemeinsam genutzte Ressourcen zu diagnostizieren.
- Die taktgenaue und ereignisgesteuerte Aufzeichnung ist sehr gut geeignet für die Suche nach der Fehlerursache. Eine Aufzeichnung ist auch in einem bestimmten Zeitintervall vor einem beobachteten Fehler möglich.

[5] Atomare Operationen setzen per Definition die Pipeline im Speicherbus außer Kraft und verhalten sich daher wie einzelne Zugriffe.

Konkret wird die Trace-Architektur von Stefan Alex [Ale10] eingesetzt. Sie bietet folgende Vorteile:

- parallele, taktgenaue Überwachung vieler Komponenten wie mehrere Prozessorkerne und Speichermanager,
- flexible Generierung von Aufzeichnungsmodulen (Tracer) und Ereignissteuerung (Trigger),
- benutzerdefinierte Nachrichtenformate u. a. für den Speichermanager, da hier die für Prozessorkerne üblichen Programm- und Daten-Traces nicht geeignet sind,
- hohe Datentransferrate dank Gigabit-Ethernet-Schnittstelle und verlustfreier Kompression,
- geringer Bedarf an On-Chip-Speichern auf dem FPGA für die kurzzeitige Zwischenpufferung der aufgezeichneten Signalwerte.

3.3.6 Laden einer SHAP-Datei

Der Ladeprozess einer SHAP-Datei (s. Abschn. 2.2.5) muss um eine Initialisierung der zusätzlichen Prozessorkerne erweitert werden:

- Schritt 1 — Mikrocode lädt den Bootloader in den Heap-Speicher — wird nur von Prozessorkern 0 ausgeführt. Die anderen Kerne warten zunächst auf ein spezielles Datenwort, dass ihnen Kern 0 schickt.
- Die Schritte 2 bis 4 bleiben prinzipiell unverändert.

  Die `Thread`-Klasse erhält zusätzlich einen statischen Initialisierer, der ganz normal während Schritt 3 ausgeführt wird. Der Initialisierer erzeugt je Prozessorkern einen speziellen Kontroll-Thread und sendet die Objektreferenz des Threads an den entsprechenden Kern als Datenwort. Jeder Kern startet daraufhin seinen Kontroll-Thread, der eine Warteschlange überwacht. Jeder Kern kann in die Warteschlange eines anderen Kern schreiben, um dort im Rahmen des Scheduling einen neuen Thread zu starten.

3.3.7 SHAP-Klassenbibliothek

Folgende Funktionen sind für die parallele Ausführung mehrere Threads auf dem Mehrkernprozessor zusätzlich sinnvoll:

- `Runtime.availableProcessors()` liefert die Anzahl der Prozessorkerne.
- `Thread.startOn(int core)` startet den Thread auf einem gewünschten Prozessorkern, anstatt die Auswahl dem Scheduler zu überlassen.

Des Weiteren muss die Klassenbibliothek intern an das geänderte Thread-Scheduling und das Laden der SHAP-Datei angepasst werden. Die Methoden für die Thread-Synchronisation können unverändert vom Einkernprozessor übernommen werden, da lediglich die Implementierung der zugehörigen Bytecodes angepasst werden muss.

Für den Test der geänderten und neuen Funktionen steht unverändert die strukturierte Ausnahmebehandlung der SHAP-Plattform von 2007 zur Verfügung. Ein Software-Debugger wurde aufgrund der Einfachheit der betroffenen Funktionen nicht implementiert.

3.3.8 Unveränderte Komponenten

An folgenden Komponenten der SHAP-Plattform von 2007 sind keine Änderungen für die Mehrkernarchitektur notwendig:

- SHAP-Linker,
- Assembler für den Mikrocode des Prozessors.

Das DITO-Modell der Einkernprozessorarchitektur wurde nicht auf mehrere Kerne erweitert. Der Einsatzzweck — Debugging des Mikrocode — erforderte kein Mehrkernmodell, da der geänderte Mikrocode für `monitorenter` auch mit einem Kern überprüft werden kann.

3.3.9 Zusammenfassung

Der Aufbau eines Systems mit der SHAP-Mehrkernarchitektur ist nochmals in Abb. 3.6 dargestellt. Es besteht aus folgenden Hardwarekomponenten:

- Dem SHAP-Mehrkernprozessor mit n Kernen, die jeweils einen lokalen Stack und einen eigenen Methoden-Cache besitzen. Die Anbindung an den zentralen Heap-Speicher erfolgt über den Mehrport-Speichermanager, dessen Speicherbus Pipelining von Transaktionen unterstützt. Der GC ist größtenteils im Speichermanager enthalten, aber auch auf die Stack-Speicher verteilt. Die Ansteuerung des externen Speicher erfolgt nach wie vor über einen integrierten Speichercontroller. Je Kern stehen getrennte Busse für den Zugriff auf Stack, Heap, Bytecode und I/O zur Verfügung.
- Dem externen, zentralen Heap-Speicher, der auch den Bytecode enthält.
- Den I/O-Komponenten für die Kommunikation mit der Außenwelt. Diese sind über einen gemeinsam genutzten Wishbone-Bus mit mehreren Bus-Mastern angebunden. Wahlweise können die I/O-Geräte auch per DMA an den Speichermanager angeschlossen werden.

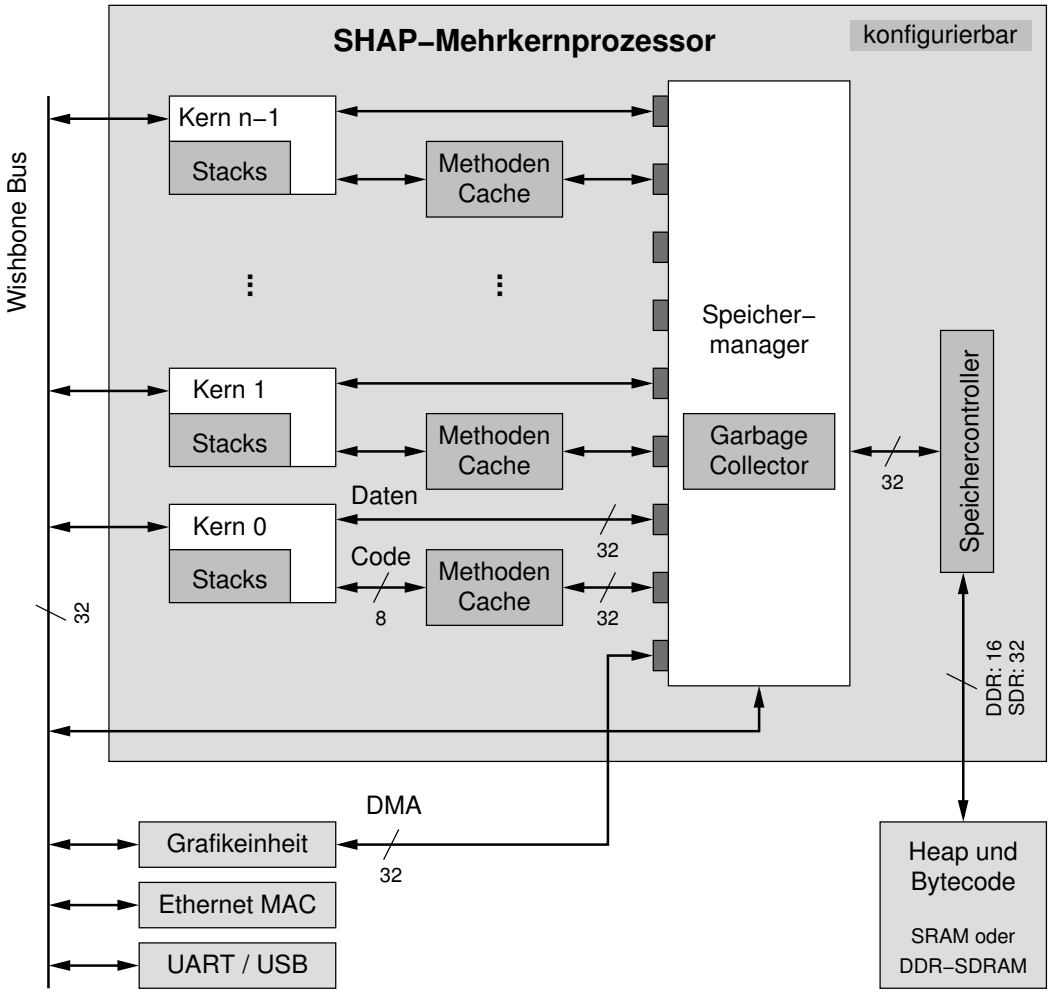

Abbildung 3.6: Aufbau eines Systems mit SHAP-Mehrkernprozessors

Die Eigenschaften der SHAP-Mehrkernarchitektur sind:

- native Ausführung von Java Bytecode ohne unterliegendem Betriebssystem,
- variable Anzahl von Kernen mit jeweils lokalem On-Chip-Stack und Methoden-Cache
- in der Größe konfigurierbarer Stack,
- in der Größe und Strategie konfigurierbarer Methoden-Cache,
- zentraler Heap-Speicher, der auch den Bytecode enthält,
- Mehrport-Speichermanager mit einem Vollduplex-Speicherbus und Pipelining von Transaktionen,
- separate, kleine TLBs für jeden Speichermanager-Port mit einem (Methoden-Cache und DMA) bzw. zwei Einträgen (Objektzugriffe),
- schnelle atomare Operationen für voneinander unabhängige Monitore zur Thread-Synchronisation,
- nebenläufiger, nicht blockierender GC mit verteiltem Stack-Scan,
- 2-stufiges preemptives Thread-Scheduling mittels einer Round-Robin-Strategie,
- Anbindung der I/O-Komponenten über einen gemeinsam genutzten, Multi-Master-fähigen Bus an die Prozessorkerne,
- Anbindung der I/O-Komponenten optional auch per DMA an den Speichermanager für hohe Datentransferraten,
- Kompatibilität zur SHAP-Plattform und damit Unterstützung des CLDC-API,
- objektorientierte Konzepte analog der SHAP-Einkernarchitektur:
  - automatische Speicherverwaltung,
  - strukturierte Ausnahmebehandlung,
  - Mehrfachvererbung mittels Interfaces,
- optionaler Anschluss einer Trace-Architektur für Test und Diagnose sowie Evaluation der Mehrkernarchitektur.

4 Ergebnisse

4.1 Prototypische Implementierung

4.1.1 Allgemeines

Die Implementierung eines Prototypen und die Auswertung der SHAP-Mehrkernarchitektur erfolgte unter Verwendung von programmierbaren Schaltkreisen, hier FPGAs, und dazu vom FPGA-Hersteller gefertigte Entwicklungsboards. Als Ausgangsbasis stand

- die VHDL-Beschreibung des SHAP-Einkernprozessors,
- der Assemblercode für den Mikrocode sowie
- der Java-Quellcode des SHAP-API

jeweils mit Stand von 2007 zur Verfügung. Diese drei Quelltexte waren auch für die SHAP-Mehrkernarchitektur entsprechend Abschn. 3.3 anzupassen und zu erweitern.

Der Prototyp des SHAP-Mehrkernprozessors verwendet außerdem eine neue GC-Implementierung mit Unterstützung für „Weak References“. Dieser GC wurde von Thomas B. Preußer im Rahmen seiner Dissertation am Lehrstuhl für VLSI-Entwurfssysteme, Diagnostik und Architektur an der Technischen Universität Dresden parallel zu dieser Arbeit entwickelt. Kernpunkt des neuen GC ist der Einsatz des Mikroprozessors ZPU dessen Firmware mittels C programmiert wird. Die ZPU steuert diverse Hardwarekomponenten u. a. für den Zugriff auf Stack- und Heapspeicher, GC-Bus und Markierungstabellen. Die für den SHAP-Mehrkernprozessor benötigten, von mir vorgenommenen, Änderungen wurden bereits in Abschn. 3.3.1.5 erläutert.

Verwendet wird außerdem eine optimierte Version des SHAP-Linkers, die Thomas B. Preußer ebenfalls im Rahmen seiner Dissertation untersuchte.

Die VHDL-Beschreibung des SHAP-Mehrkernprozessors ist weitestgehend von einem spezifischen FPGA unabhängig. Anzupassen sind lediglich die Instanziierung der auf dem FPGA vorhandenen On-Chip-Speichermodule für alle integrierten Speicher (Stack, Mikrocode, GC-Firmware und Caches).

Für ein konkretes FPGA-Entwicklungsboard sind weiterhin der integrierte Speichercontroller sowie die I/O-Geräte anzupassen. Prototypische Implementierungen stehen für folgende Boards bereit (englische Bezeichnungen):

- Virtex-5 ML505 Prototyping Board der Firma Xilinx,
- Virtex-5 XUPV5 University Board der Firma Xilinx,
- Spartan-3 Starter Kit Board der Firma Xilinx,
- Spartan-3E Starter Kit Board der Firma Xilinx,
- Cyclone-II DE2 Development Board der Firma Altera.

Die VHDL-Beschreibung kann in folgenden Punkten konfiguriert werden:

- Anzahl der Prozessorkerne n (ohne ZPU),
- Größe des Stack-Speichers über die Anzahl und Größe der Stack-Blöcke,
- Größe und Strategie des Methoden-Caches,
- maximale Größe eines Objekts und damit auch die Größe eines Segments im externen Speicher,
- maximale Anzahl (gleichzeitig) allozierbarer Objekte,
- Größe des externen Speichers in Abhängigkeit vom eingesetzten Entwicklungsboard.

Im Gegensatz zur VHDL-Beschreibung sind

- der modifizierte Assemblercode des Mikrocodes,
- der erweiterte Java-Quellcode des SHAP-API und
- der C-Quellcode der GC-Firmware

unabhängig vom spezifischen FPGA. Assemblercode und Java-Quellcode müssen ebenfalls nicht für eine konkrete SHAP-Konfiguration angepasst werden. Die GC-Firmware kann analog der VHDL-Beschreibung in den Punkten

- maximale Größe eines Objekts und damit auch die Größe eines Segments im externen Speicher,
- maximale Anzahl (gleichzeitig) allozierbarer Objekte und
- Größe des externen Speichers

konfiguriert werden.

In den folgenden Abschnitten werden die konkreten SHAP-Konfigurationen beschrieben, wie sie für die Leistungsbewertung verwendet wurden.

4.1.2 XUPV5-Entwicklungsboard

4.1.2.1 Allgemeine Eigenschaften

Für die Leistungsbewertung wurde primär das „XUPV5 University Board“ der Firma Xilinx verwendet, da nur dieses folgende 2 Merkmale gleichzeitig bietet:

- Die Größe des FPGAs ermöglicht eine Implementierung des SHAP-µP mit bis zu 18 Kernen.
- Der externe Speicher besitzt einen 32-Bit-Datenbus und unterstützt genauso wie der interne Speicherbus ein Pipelining der Speicherzugriffe. Die Speicherbandbreite ist damit mit 1 32-Bit-Zugriff/Takt maximal.

Das XUPV5-Entwicklungsboard umfasst einen Virtex-5 XC5VLX110T-1 FPGA und eine Auswahl an externen Schnittstellen und Speichern. Neben dem FPGA wurde für die Leistungsbewertung der externe ZBT-SRAM, die UART für die Kommunikation mit dem PC und die LEDs / LCD-Anzeige für Statusanzeigen verwendet. Dieses Board ist bis auf den FPGA identisch mit dem „ML505 Prototyping Board“ der Firma Xilinx. Letzteres bietet jedoch nur einen XC5VLX50T FPGA mit etwa halb so viel Ressourcen.

4.1.2.2 Konfiguration

Es wurde folgende Basiskonfiguration von SHAP verwendet:

- je Kern 8 KiB Stack-Speicher zu 32 Blöcken je 64 32-Bit-Wörtern,
- Objekte bis zu einer Größe von 16383 32-Bit-Wörtern,
- maximal 8191 (gleichzeitig) allozierbare Objekte,
- 1 MiB externer Speicher aufgeteilt in 16 Segmente zu je 64 KiB.

Die Stackgröße hat nur einen Einfluss auf die benötigten FPGA-Ressourcen und damit indirekt auch auf die erzielbare Taktfrequenz. Auf die Ausführungszeit eines Programms wirkt sie sich jedoch nicht aus. Die minimale Blockgröße von 64 Worten ist notwendig für einzelne API-Funktionen. Für die untersuchten Benchmarks wären mindestens 8 Blöcke zu je 64 Worten und damit 2 KiB Stack nötig.

Die Aufteilung des externen Speichers in 16 Segmente ist ein Kompromiss. Einerseits erfordern große Objekte (Klassenobjekte, Datenfelder) ebenso große Segmente. Andererseits erfordert eine effiziente Garbage-Collection eine große Anzahl von Segmenten. Die später noch vorgestellten Programme SparseMatmultInt und Crypt benötigen Datenfelder mit 16000 Wörtern (64000 Byte zzgl. 12 Byte Verwaltungsdaten).

Weiterhin ist es wünschenswert, dass so viele Objekte wie möglich zu einem Zeitpunkt gleichzeitig alloziert sein können. Aufgrund des Aufbaus der Speicherverwaltung können

jedoch maximal $2^{14} - 1$ Objekte konfiguriert werden. Zu beachten ist jedoch, dass die Referenztabelle (vgl. Abschn. 2.2.4) einen Teil des externen Speichers belegt. Die Wahl fiel hier deshalb auf $2^{13} - 1 = 8191$ Objekte. Die dafür notwendige Objekttabelle belegt 64 KiB Speicher und damit eines der 16 Segmente. Die anderen Segmente stehen für den Heap (inklusive Bytecode) zur Verfügung.

Ausgehend von der Basiskonfiguration wurden folgende Parameter variiert, wobei alle Parameterkombinationen untersucht wurden:

- Anzahl der Prozessorkerne: $n = 1 \ldots 18$ (ohne ZPU);
- Größe des Methoden-Cache:
  a) je Kern 2 KiB Methoden-Cache (Minimum) mit der Strategie „BumpRetInv“,
  b) je Kern 4 KiB Methoden-Cache zu 2 Blöcken je 2 KiB mit der Strategie „BumpRetInv“.

4.1.2.3 Ressourcenbedarf

Im Folgenden wird der Bedarf an FPGA-Ressourcen aufgeschlüsselt. Dazu wurde eine Synthese der VHDL-Beschreibung sowie die Platzierung und Verdrahtung der benötigten FPGA-Ressourcen unter Verwendung der Software „Xilinx ISE“ in der Version 12.1 durchgeführt. Alle folgenden Messwerte zum Ressourcenbedarf wurden aus dem „Map Report“ entnommen. Die Einhaltung der von mir vorgegebenen (zu erzielenden) Taktfrequenz wurde im „Timing Analyzer Report“ überprüft.

In der o. g. Konfiguration können für alle Parameterkombinationen bis zu 18 Prozessorkerne bei einer Taktfrequenz für das Gesamtsystem von 80 MHz auf dem FPGA implementiert werden. Dies entspricht der maximalen Taktfrequenz des ursprünglichen SHAP-µP von 2007 mit einem Kern.

Bei 18 Kernen und 4 KB Methoden-Cache sind bereits 90% der Slice-LUTs und 41% der Slice-Register belegt. Da LUTs und Register nicht immer paarweise in eine Slice platziert wurden, sind sogar 98% der verfügbaren Slices belegt und der FPGA somit voll ausgelastet. Eine Erhöhung auf 19 Kerne ist nur bei gleichzeitiger Reduktion der Taktfrequenz auf 72 MHz möglich. Die Auslastung der LUTs steigt auf 95% und die der Slices auf 99%. Daher wurden nur Konfigurationen bis zu 18 Kernen weiter untersucht.

In Tab. 4.1 ist der Bedarf an Slice-Register und Slice-LUTs in Abhängigkeit von der Anzahl der Prozessorkerne für beide Methoden-Cache-Größen aufgelistet und in Abb. 4.1 grafisch dargestellt. Darin ist für jede Messreihe (Tabellenspalte) ein linearer Zusammenhang für $n = 1 \ldots 16$ Kerne zu erkennen. In diesem Bereich können daher die Messwerte durch die Funktion

$$f(x) = K_1 + K_2 \cdot n \tag{4.1}$$

Tabelle 4.1: Ressourcenbedarf an Virtex-5 Slice-Register / LUTs auf dem XUPV5-Entwicklungsboard

Anzahl Prozessorkerne	4 KiB Methoden-Cache		2 KiB Methoden-Cache	
	Slice Regs	Slice LUTs	Slice Regs	Slice LUTs
1	3420	5660	3299	5427
2	4871	8512	4629	8080
3	6317	11303	5954	10657
4	7766	14102	7282	13231
5	9212	16965	8607	15899
6	10658	19771	9932	18481
7	12105	22801	11258	21226
8	13548	25538	12587	23679
9	14999	28273	13911	26304
10	16445	31075	15235	28816
11	17892	33950	16561	31460
12	19339	36821	17887	34089
13	20783	39554	19210	36582
14	22229	42439	20535	39242
15	23676	45315	21861	41895
16	25144	48516	23223	44829
17	27098	59528	25057	55601
18	28574	62706	26396	58599
Konstanter Anteil K_1	1974,75	2783,60	1973,45	2833,07
Linearer Faktor K_2	1447,12	2838,66	1326,44	2607,12
Maximale Abweichung für $n \leq 16$	15,33	313,84	26,51	282,01

approximiert werden. Die Konstanten K_1 und K_2 wurden getrennt für jede Messreihe mit Hilfe des Programms „gnuplot“ in der Version 4.4 ermittelt, das zur Lösung der Approximationsaufgabe den Levenberg-Marquardt-Algorithmus verwendet [C+10]. Anschließend wurde die betragsmäßig maximale Abweichung der Approximation zu den Messwerten (für $n \leq 16$) selbst berechnet. Die Konstanten sowie die maximale Abweichung sind ebenfalls in Tab. 4.1 enthalten. Die Approximation ist zum Vergleich außerdem in Abb. 4.1 grafisch dargestellt.

Die Approximation der Slice-LUTs ist nur für bis zu 16 Prozessorkerne gültig. Bei 2 KiB Methoden-Cache liegt die LUT-Anzahl für 17 Kerne mit 55601 um ca. 8450 höher als erwartet. Bei 18 Kernen werden sogar ca. 8800 LUTs mehr benötigt. Für 4 KiB

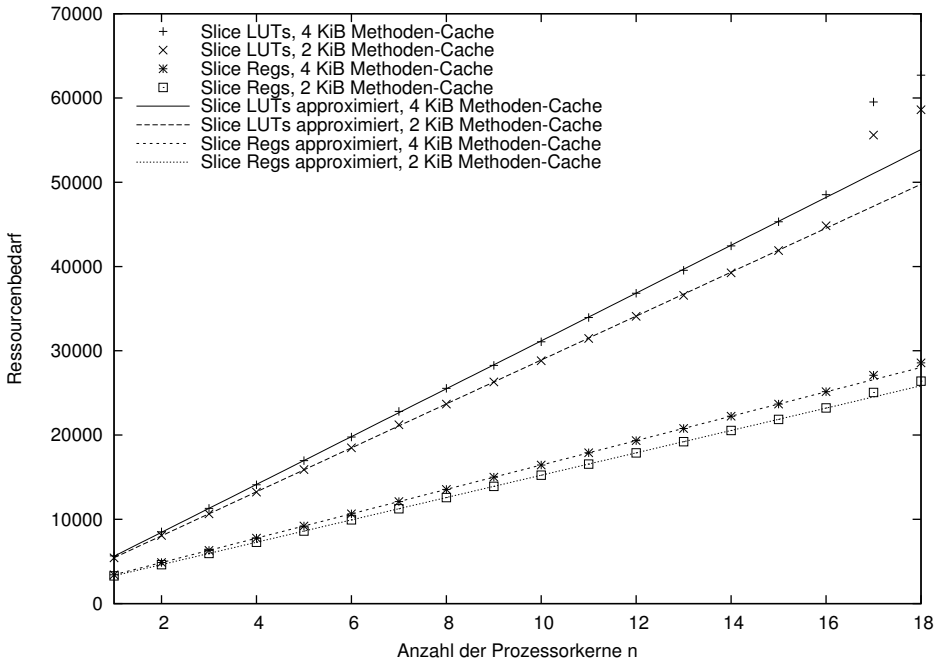

Abbildung 4.1: Ressourcenbedarf an Virtex-5 Slice-Register / LUTs auf dem XUPV5-Entwicklungsboard

Methoden-Cache gilt dies gleichermaßen (ca. 8500 und 8800 LUTs). Warum dies so ist, ist nicht nachvollziehbar. Wird nämlich für die Synthese probehalber ein XC5VLX155T FPGA mit mehr LUT-Ressourcen verwendet, dann entspricht die Anzahl der Slice-LUTs wieder in etwa den erwarteten Werten. Gleiches gilt auf für die Anzahl der Slice-Register, wobei hier die gemessenen Werte um ca. 500 Register (für 17 und 18 Kerne) höher liegen als erwartet.

Die Anzahl der RAM-Blöcke variiert nur mit der Anzahl der Kerne:

$$\mathrm{BRAM}_{36\,\mathrm{kBit}}(n) = 1 + 3 \cdot n \tag{4.2}$$

$$\mathrm{BRAM}_{18\,\mathrm{kBit}}(n) = 1 + 2 \cdot n \tag{4.3}$$

Der GC benötigt davon einen 36-kBit-BRAM für den Programm- und Datenspeicher der ZPU und einen 18-kBit-BRAM für die globale Markierungstabelle. Pro Kern werden zwei 36-kBit-BRAMs für den Stack, ein 36-kBit-BRAM für den Methoden-Cache, ein 18-kBit-BRAM für den Mikrocode und ein 18-kBit-BRAM für die lokale Markierungstabelle benötigt.

Für nur 2 KiB Methoden-Cache wäre jeweils ein 18-kBit-BRAM statt eines 36-kBit-BRAM zu erwarten. Jedoch benötigt der Methoden-Cache einen Dual-Port-RAM mit zwei 32 Bit breiten Datenbussen. Diese RAM-Konfiguration steht auf dem Virtex-5 FPGA aber erst beim Zusammenschluss zweier 18-kBit-BRAMs zu einem 36-kBit-BRAM zur

Verfügung. Bei 2 KiB Methoden-Cache wird lediglich nur die untere Hälfte des Speichers genutzt.

Die 32×32-Bit-Multiplizierer (Prozessorkern) und 32×16-Bit-Multiplizierer (ZPU) werden durch die DSP48E-Blöcke bereitgestellt. Die Anzahl ist unabhängig von der Methoden-Cache-Größe:

$$\text{DSP48E}(n) = 2 + 3 \cdot n \,. \tag{4.4}$$

Für den integrierten Speichercontroller wird eine feste Anzahl von Registern in den I/O-Blöcken des FPGAs benötigt:

$$\text{IOB\_FF}(n) = 87 \,. \tag{4.5}$$

Zusammenfassend ist festzustellen, dass der Hardwareaufwand in Form von FPGA-Ressourcen linear mit der Anzahl der Prozessorkerne ansteigt. Dies gilt für bis zu 16 Kerne und wäre auch für 17 und 18 Kerne zu erwarten.

4.1.3 DE2-Entwicklungsboard

Sekundär wurde auch das „DE2 Development Board“ der Firma Altera für die Leistungsbewertung herangezogen. Es ermöglicht einen Vergleich mit anderen Systemen auf ein- und derselben Hardwareplattform.

4.1.3.1 Konfiguration

Auf dem DE2-Entwicklungsboard kommt der Cyclone-2 FPGA EP2C35-C6 zum Einsatz. Für die Leistungsbewertung wurde weiterhin der externe SRAM, die UART für die Kommunikation mit dem PC und die LEDs / 7-Segment-Anzeige für Statusausgaben verwendet.

Da der Cyclone-2 FPGA viel weniger Speicher integriert als der Virtex-5 des XUPV5-Boards, mussten Stack-Speicher und Methoden-Cache in ihrer Größe angepasst werden. Es wurde daher folgende, abgewandelte Basis-Konfiguration verwendet:

- je Kern 1 KiB Stack-Speicher zu 8 Blöcken je 32 32-Bit-Wörtern,
- je Kern 2 KiB Methoden-Cache (Minimum) mit der Strategie „BumpRetInv“,
- Objekte bis zu einer Größe von 16383 32-Bit-Wörtern,
- maximal 8191 gleichzeitig allozierbare Objekte,
- 1 MiB externer Speicher aufgeteilt in 16 Segmente.

Da mit dieser Konfiguration später nur der Lift-Benchmark evaluiert wird, konnte die Größe eines Stack-Blocks auf 32 Wörter reduziert werden. Es werden aber weiterhin 8 Blöcke und damit 1 KiB Stack benötigt.

Der externe Speicher besitzt außerdem nur einen 16 Bit breiten Datenbus. Der integrierte Speichercontroller wurde so implementiert, dass jeder Speicherzugriff (32 Bit) den Speicherbus für 2 Takte belegt, wobei die Latenz beim Lesezugriff aber insgesamt 3 Takte beträgt (teilweises Pipelining).

Ausgehend von der Basiskonfiguration wurde diesmal nur die Anzahl der Prozessorkerne (ohne ZPU) variiert: $n = 1 \ldots 6$. Für einen späteren Vergleich mit JopCMP wurde zudem der Ressourcenbedarf ohne GC ermittelt.

4.1.3.2 Ressourcenbedarf

Für die Synthese der VHDL-Beschreibung sowie die Platzierung und Verdrahtung der benötigten FPGA-Ressourcen wurde diesmal die zum FPGA passende Software „Altera Quartus II“ in der Version 10.0 verwendet. Der Bedarf an FPGA-Ressourcen wurde aus dem „Fitter Report“ entnommen. Die Einhaltung der vorgegebenen Taktfrequenz wurde mit dem „TimeQuest Timing Analyzer“ überprüft.

In der Konfiguration mit GC können bis zu 6 Prozessorkerne bei einer Taktfrequenz für das Gesamtsystem von 60 MHz auf dem FPGA implementiert werden. Bei 6 Kernen sind bereits 78% der LE-LUTs, 89% der LEs und 75% der verfügbaren Speicher belegt (LE = engl.: Logic Element).

Der Bedarf an LE-LUTs und -Register ist in Tab. 4.2 in Abhängigkeit von der Anzahl der Prozessorkerne aufgelistet und grafisch in Abb. 4.2 dargestellt. Wiederum ist ein linearer Zusammenhang erkennbar, diesmal über den gesamten Bereich von $n = 1 \ldots 6$. Die Approximation der Messwerte und die Berechnung der maximalen Abweichung wurde analog Abschn. 4.1.2.3 durchgeführt. Die zugehörigen Ergebnisse sind ebenfalls in Tab. 4.2 enthalten. Abbildung 4.2 visualisiert zudem die Approximation. Der Verlauf für die Konfiguration ohne GC wurde in der Grafik der Übersicht halber weggelassen, da diese Konfiguration ohnehin eine Ausnahme darstellt.

Der restliche Hardwareaufwand ergibt sich nach folgenden Gleichungen:

$$\text{Memory\_Bits}(n) = \begin{cases} 49152 + 52024 \cdot n & \text{mit GC} \\ 43832 \cdot n & \text{ohne GC} \end{cases} \tag{4.6}$$

$$\text{Multiplier\_Elements}(n) = \begin{cases} 4 + 6 \cdot n & \text{mit GC} \\ 6 \cdot n & \text{ohne GC} \end{cases} \tag{4.7}$$

$$\text{I/O\_Register}(n) = 67 \tag{4.8}$$

Die LE-LUTs und Register werden von der Altera-Software in LEs zusammen gepackt. Die resultierende LE-Anzahl ist aber wenig aussagekräftig, da sie erfahrungsgemäß nicht nur von der Logiksynthese sondern auch von der zu erreichenden Taktfrequenz und dem Auslastungsgrad des FPGA abhängt — siehe dazu auch das Beispiel zu 19 Kernen auf

Tabelle 4.2: Ressourcenbedarf an Cyclone-2 LEs / LUTs / Register auf dem DE2-Entwicklungsboard

Anzahl	mit GC			ohne GC		
Prozessorkerne	LE Regs	LE LUTs	LEs	LE Regs	LE LUTs	LEs
1	6219	9014	11321	3792	5456	6860
2	7458	12438	15094	4887	8597	10189
3	8692	15850	18071	5976	11689	13507
4	9929	19284	22363	7067	14774	16934
5	11163	22661	25997	8155	17887	20299
6	12397	26054	29477	9243	20994	23616
Konst. Anteil K_1	4985,47	5619,87	7609,07	2705,00	2368,33	3480,47
Lin. Faktor K_2	1235,49	3408,66	3650,89	1900,00	3104,14	3358,20
Max. Abweichung	1,96	29,49	490,74	3,00	20,38	48,07

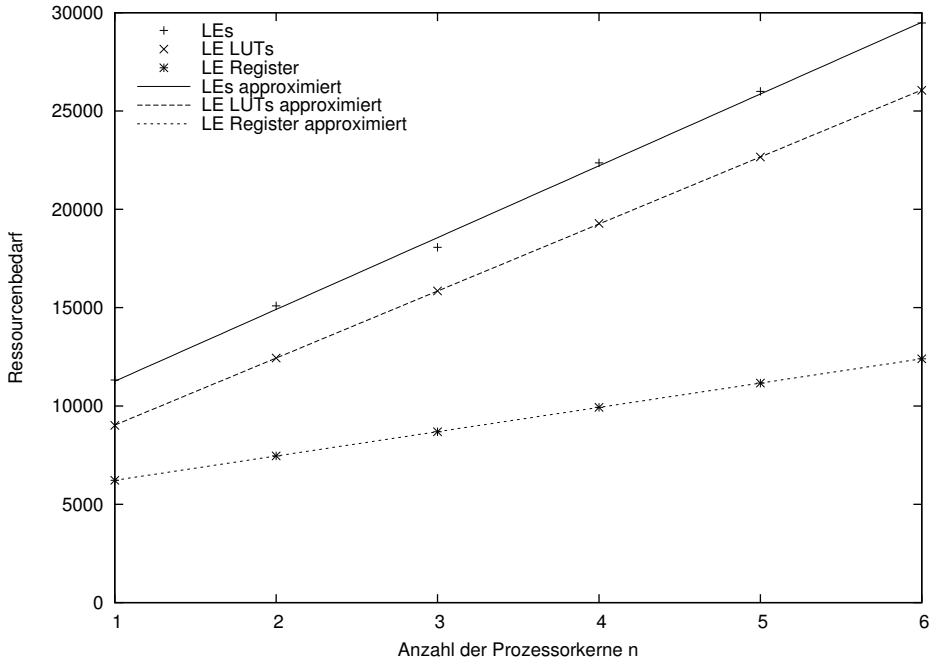

Abbildung 4.2: Ressourcenbedarf an Cyclone-2 LEs / LUTs / Register auf dem DE2-Entwicklungsboard

dem XUPV5-Board in Abschn. 4.1.2. Sie dient vielmehr dem Vergleich mit anderen Arbeiten. LE-Anzahl, Konstanten für die lineare Approximation sowie die maximale Abweichung sind in Tab. 4.2 enthalten.

Zusammenfassend gesagt, steigt der Hardwareaufwand auf dem DE2-Board linear mit der Anzahl der Prozessorkerne. Diesmal ist kein unerwarteter Bedarf an LE-Register und LE-LUTs bei zunehmender Auslastung des FPGAs zu verzeichnen.

4.2 Leistungsbewertung

In diesem Abschnitt werden zunächst die zur Leistungsbewertung verwendeten Benchmarks vorgestellt. Anhand dieser wird dann die reale Steigerung der Verarbeitungsleistung ausgewertet und anschließend in Relation zur theoretisch möglichen gesetzt. Abschließend wird der SHAP-Mehrkernprozessor mit bisherigen Arbeiten verglichen.

4.2.1 Benchmarks

4.2.1.1 Allgemeines

Die im Folgenden vorgestellten Benchmarks wurden genauso wie das SHAP-API standardmäßig mit dem Eclipse Java-Compiler übersetzt (vgl. Abschn. 2.2.1). Eine Ausnahme bildet der Lift-Benchmark (s. u.). Der SHAP-Linker wurde in der Standardkonfiguration eingesetzt.

4.2.1.2 Primäre Benchmarks

Für die Bewertung wurden primär Algorithmen verwendet, die sich in Teilaufgaben zerlegen lassen und damit auf mehrere Prozessorkerne verteilbar sind. Die Berechnung der Teilaufgaben wird parallel mit einem Thread pro Prozessorkern ausgeführt. Die verwendeten Benchmarks werden im Folgenden kurz charakterisiert:

Queens sucht nach allen Lösungen des N-Damen-Problems[1] für eine gegebene Schachbrettgröße und ist Teil der Benchmark-Suite „JemBench“ [SPU10] in der Version 2.0[2]. Um eine gleichmäßige Auslastung vieler Prozessorkerne zu erreichen, wird hier abweichend ein Schachbrett mit 13×13 Feldern verwendet. Auf diesem sind entsprechend 13 Damen so zu platzieren sind, dass sie sich nicht gegenseitig bedrohen. Für die Parallelisierung auf Thread-Ebene wird die Berechnung in 31100 Teilprobleme zerlegt, in dem für jedes Teilproblem bereits 5 Damen vorplatziert sind.

[1] http://de.wikipedia.org/wiki/Damenproblem
[2] http://jembench.sourceforge.net/

Die Verteilung der Teilprobleme auf die Prozessorkerne / Threads erfolgt dynamisch, da die Zeit für die Berechnung eines Teilproblems stark variiert. Zu Beginn und nach Abschluss der Berechnung eines Teilproblems holt sich jeder Thread das nächste Teilproblem aus einer Art Aufgabenliste. Diese Liste wird jedoch nicht im Speicher vorgehalten, sondern jeder Eintrag auf Anforderung erzeugt. Ist die Liste leer, dann ist die Berechnung des Gesamtproblems beendet.

Der Zugriff auf die Aufgabenliste muss synchronisiert werden, ebenso wie die Addition der Teilergebnisse. Diese Phasen sind jedoch im Vergleich zur Berechnung sehr kurz, sodass kaum Auswirkungen auf den erzielbaren Speed-Up zu erwarten sind.

SparseMatmultInt multipliziert eine dünn besetzte Matrix mit einem Vektor. Als Grundlage diente der Benchmark „SparseMatmult“ aus der Suite „Multi-threaded Benchmarks“ des Java Grande Forum[3].

Es wurde die aktuelle Version 1.0 der Suite verwendet und der Datentyp der Matrix- / Vektorelemente von Gleitkomma auf Ganzzahl geändert. Dies war notwendig, da Gleitkommaoperationen auf SHAP aktuell nur per Software emuliert werden und damit die Ausführung überwiegend aus Arithmetik bestanden hätte.

Ebenso wurde die Matrix verkleinert, da auf SHAP ein Integer-Feld maximal 16380 Elemente umfassen darf. Es wurde eine Matrix der Größe 4000×4000 verwendet, wobei nur 16000 Elemente verschieden von 0 sind.

Die Berechnung wird in n (nahezu) gleich große Teile zerlegt, wobei n der Anzahl der Prozessorkerne entspricht. Die Ausführungszeiten der Teilberechnung sind somit (nahezu) identisch, sodass eine statische Verteilung genügt. Eine Synchronisation ist damit insofern nur notwendig, um auf den Abschluss aller Teilberechnungen zu warten.

Wie vorgegeben umfasst die Zeitmessung lediglich die Matrixmultiplikation, jedoch nicht die vorbereitenden Maßnahmen für die Erzeugung der Matrix. Um die Messgenauigkeit zu erhöhen, wird dabei jede Teilberechnung vom zuständigen Thread mehrmals hintereinander ausgeführt.

Crypt entstammt ebenfalls aus der obigen Benchmark-Suite des Java Grande Forum. Dieser Benchmark ver- / entschlüsselt eine Nachricht mit dem IDEA-Algorithmus.

Es musste lediglich die Länge der Nachricht auf 16376 Zeichen angepasst werden. SHAP erlaubt nur ein Byte-Feld mit maximal 16380 Elementen; für den IDEA-Algorithmus muss die Nachrichtenlänge zudem durch 8 teilbar sein.

[3] http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/threads.html

Die Nachricht wird für die Parallelisierung in n (nahezu) gleich große Teile aufgeteilt. Es genügt damit eine statische Verteilung wie bei SparseMatmultInt.

Wie vorgegeben umfasst die Zeitmessung lediglich die Ver- und Entschlüsselungsroutine, jedoch nicht Nachrichtenerzeugung und abschließende Überprüfung.

Minimum ist ein selbst entwickelter, synthetischer Benchmark, der den minimalen Speed-Up unter folgenden Randbedingungen ermittelt:

- Es ist keine Thread-Synchronisation notwendig.
- Methoden-Caching und die dafür notwendige Speicherbandbreite ist nur zum Beginn des Benchmarks erforderlich.
- Der GC ist kein limitierender Faktor.

Ohne diese Bedingungen ließe sich immer ein Benchmark konstruieren, dass maximal einen Speed-Up von 1 erreicht.

Die wiederholte Ausführung des Bytecodes `putfield` lastet die zur Verfügung stehende Speicherbandbreite am stärksten aus. Außerdem ist ein Zugriffsmuster zu wählen, sodass der TLB der Adressberechnung keine Treffer erzielt. Es wurde daher folgender Algorithmus eingesetzt (Java-Code):

```
int i = 100000;
Data data0 = new Data();
Data data1 = new Data();
Data data2 = new Data();

while(i-->0) {
  data0.x = 0;
  data1.x = 0;
  data2.x = 0;
  data0.x = 0;
  data1.x = 0;
  data2.x = 0;
  // weitere 31 mal, damit Overhead für die Schleife
  // nicht ins Gewicht fällt
  ...
}
```

Entsprechend Abschn. 2.5.5 kämen noch weitere Benchmarks aus der JemBench-Suite in Frage. Diese wurden jedoch nicht evaluiert, da zu wenige Teilprobleme generiert werden, um eine gleichmäßige Verteilung auf viele Prozessorkerne zu erreichen.

4.2.1.3 Sekundäre Benchmarks

Für die Evaluation sollen ebenso 2 Algorithmen herangezogen werden (individuelle Gründe s. u.), die sich nicht in Teilaufgaben zerlegen lassen und damit nicht auf Thread-Ebene parallelisierbar sind. Stattdessen werden mehrere Instanzen des gleichen Algorithmus in einem Prozess gleichzeitig ausgeführt. Damit wird der Anwendungsfall emuliert, dass das System mehrere Anfragen zur gleichen Zeit bearbeiten soll.

Die Anzahl der Anfragen und damit Instanzen sei konstant. Je nach Anzahl der Prozessorkerne entfallen ggf. mehrere Instanzen auf einen Kern.

Folgende Programme wurden verwendet:

Lift ist eine industrielle Anwendung und Teil der Benchmark-Suite „JemBench“ [SPU10] in der Version 2.0[4]. Sie ist auch Bestandteil der Suite „JavaBench-Embedded“ in der aktuellen Entwicklerversion und zusammen mit den Quellen für JopCMP erhältlich[5].

Dieses Anwendung wurde für einen Vergleich mit JopCMP ausgewählt. Zwecks Vergleichbarkeit der Ergebnisse wurde dieser Benchmark abweichend mit dem Java-Compiler des JavaSE JDK von Oracle übersetzt.

Für die Messung wurden mit einer selbst entwickelten Programmschleife 200000 Anfragen (= Ausführungen) gleichmäßig auf n Threads (= Kernanzahl) entspricht. Die Zahl wurde so groß gewählt, damit sich für $n = 1 \ldots 18$ Prozessorkerne eine annähernd gleiche Verteilung ergibt. Die Zeitmessung beginnt mit dem Starten der einzelnen Threads und endet direkt nachdem alle Threads ihre zugewiesenen Anfragen beendet haben. Synchronisation ist nur für das Warten auf den Abschluss der Berechnungen notwendig.

FScriptME ist ein Skript-Interpreter der selbst in Java programmiert ist[6]. Das hier verwendete Skript berechnet zunächst die 2. bis 43. Fibonacci-Zahl und für diese dann die Länge der Collatz-Reihe[7].

Dieses Programm wurde ausgewählt, da es im Gegensatz zu den anderen auch zur Laufzeit häufig neue Objekte alloziert: Wird eine Interpreter-Instanz auf dem SHAP-µP ausgeführt, so alloziert diese im Mittel 20 Objekte/ms bei einer Taktfrequenz von 80 MHz. Die Allokationen sind etwa gleichmäßig über die gesamte Laufzeit von 24 s verteilt, wie in der Häufigkeitsverteilung in Abb. 4.3 ersichtlich

[4] http://jembench.sourceforge.net/
[5] http://jopdesign.com/
[6] http://fscript.sourceforge.net/
[7] Die erste Fibonacci-Zahl entspricht der zweiten. Die nullte Fibonacci-Zahl ist 0 und dafür ist die Collatz-Reihe nicht definiert.

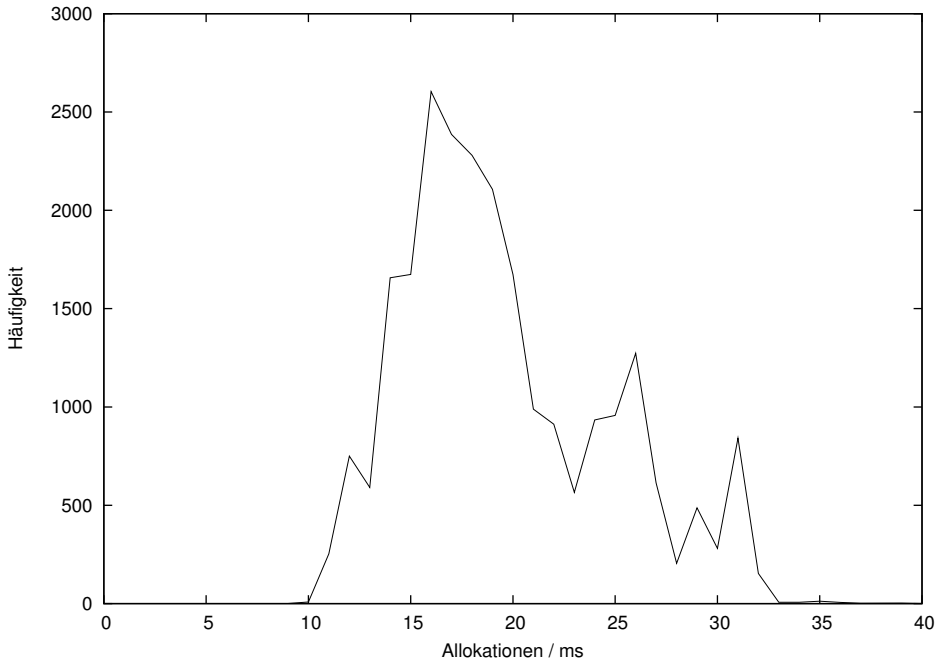

Abbildung 4.3: Allokationsrate von einer FScriptME-Instanz

ist. Die Lebensdauer dieser Objekte ist jedoch gering, sodass sie nach kurzer Zeit wieder vom GC freigegeben werden können.

Da die Laufzeit des FScriptME-Benchmarks für das gewählte Skript relativ lang ist ($\geq 24\,\mathrm{s}$), wurde die Zeitmessung abweichend vorgenommen: Statt eine große Anzahl von Anfragen auf alle Prozessorkerne gleichmäßig zu verteilen, wurde lediglich eine Anfrage pro Prozessorkern berechnet. Gemessen wurde die Zeit $t_{EXE}(n)$ vom Starten der Threads bis zum Abschluss der Berechnung. Für die Berechnung des Speed-Ups hätten jedoch nun n Anfragen auf dem Einkernprozessor ausgeführt werden müssen. Die dafür benötigte Zeit entspricht hierbei $t'_{EXE}(1) = n \cdot t_{EXE}(1)$ und wird in Gl. 2.6 an Stelle von $t_{EXE}(1)$ eingesetzt:

$$S_{A,FScriptME}(n) = \frac{n \cdot t_{EXE}(1)}{t_{EXE}(n)} \,. \tag{4.9}$$

4.2.2 Speed-Up

4.2.2.1 Allgemeines

Die Leistungsfähigkeit des SHAP-Mehrkernprozessors soll hier anhand der Beschleunigung der Programmausführung bei gleicher Last beurteilt werden. Dazu wird der Speed-Up nach Amdahl entsprechend Abschn. 2.5.4 bestimmt.

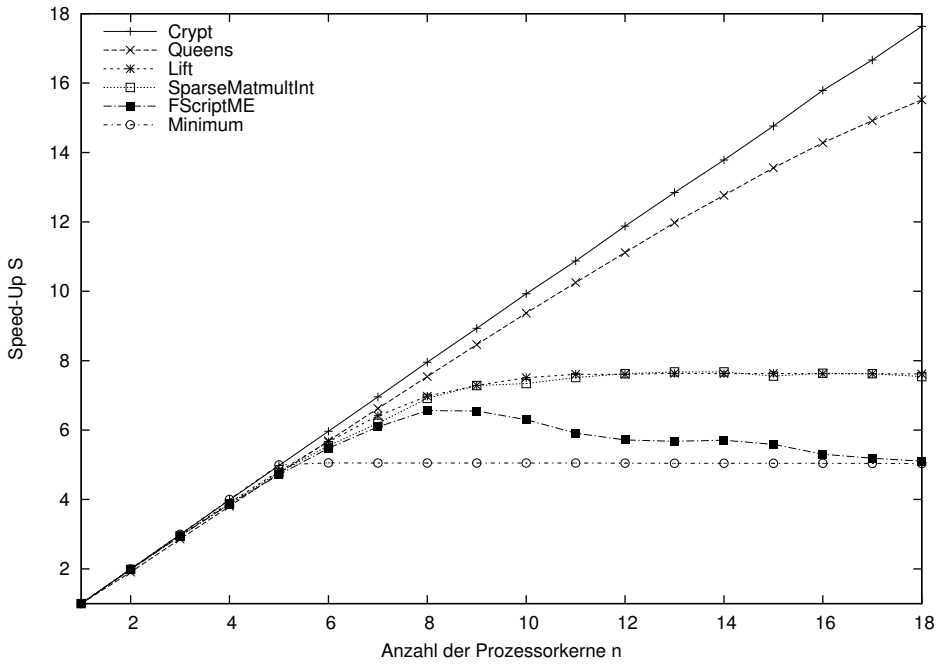

Abbildung 4.4: Speed-Up auf dem XUPV5-Board bei 2 KiB Methoden-Cache

Die hier gewählten Benchmarks erfordern nur wenig (Queens) oder gar keine Synchronisation während der Berechnung. Gemäß Amdahls Gesetz wäre der parallele Anteil α nahezu 1 und demnach folgender idealer Speed-Up zu erwarten:

$$S_{A,\text{ideal}}(n) = \frac{1}{(1-\alpha)+\alpha/n} \overset{\alpha\to 1}{\to} n\,. \tag{4.10}$$

Der ideale Speed-Up berücksichtigt jedoch weder die begrenzte Bandbreite des gemeinsamen Speichers noch die Performanceeinschränkung durch den GC. Für die Bestimmung der real erzielbaren Speed-Ups wurde, wie bereits in Abschn. 4.1.2 angesprochen, das XUPV5-Entwicklungsboard verwendet.

4.2.2.2 Primäre Benchmarks und Lift

Zunächst wurde die Parameterkombinationen mit 2 KiB Methoden-Cache und $n = 1 \ldots 18$ Prozessorkernen untersucht. Die real erzielten Speed-Ups in Abhängigkeit von der Anzahl der Prozessorkerne n sind in Abb. 4.4 dargestellt. Folgende Ergebnisse sind auszumachen:

- Bei dem Crypt-Benchmark steigt der Speed-Up fast linear bis $n = 18$ an. Die Abweichung bei $n = 18$ zum idealen Speed-Up (Gl. 4.10) beträgt lediglich 2%.
- Der Queens-Benchmark erzielt ebenso nahezu einen linearen Speed-Up. Die Abweichung zum idealen Speed-Up ist zwar größer, aber bedingt durch die Synchronisation auf die gemeinsame Aufgabenliste. Abhilfe schafft hier die Aufteilung in

weniger, aber größere Teilprobleme. Bei nur 4 vorplatzierten Damen und damit entsprechend 6404 Teilproblemen, beträgt der Speed-Up 17,7 auf 18 Prozessorkernen. Die Ausführungszeit auf einem Kern bleibt dabei unverändert.

- Ein nahezu linearer Anstieg des Speed-Ups ist bei SparseMatmultInt und Lift nur bis etwa $n = 8$ Prozessorkerne zu verzeichnen. Für größere n nähert sich der Speed-Up bei SparseMatmultInt und Lift dem Grenzwert 7,6 an, da nun die verfügbare Speicherbandbreite erschöpft ist.
- Der Speed-Up des Minimum-Benchmarks steigt zunächst linear bis $n = 5$ auf den Wert $S_5 = 5$ an und bleibt dann konstant. Das heißt: Ein Programm, das den Bedingungen aus Abschn. 4.2.1.2 genügt, erreicht
  - den idealen Speed-Up für $n \leq 5$ sowie
  - einen minimalen Speed-Up von 5 für $n > 5$.

Der Einsatz eines 4 KiB- statt eines 2 KiB-Methoden-Caches verkürzt die Ausführungszeiten von Crypt, Queens und Lift nur um weniger als 5% und ist daher nicht zu empfehlen. Für SparseMatmultInt ist gar keine Verbesserung zu verzeichnen.

4.2.2.3 FScriptME

Ein komplexeres Bild ergibt sich für den FScriptME-Benchmark. Hier wirkt nicht nur die Speicherbandbreite als begrenzender Faktor sondern auch die Performanceeinschränkung durch den GC. Aus einer Statusanzeige des FPGA-Prototypen ist ersichtlich, dass immer wieder für kurze Zeiträume keine freien Referenzen für die Allokation neuer Objekte zur Verfügung stehen, da der GC die unreferenzierten nicht wieder schnell genug freigibt. In diesen Zeiträumen werden die Threads, die neue Objekte allozieren möchten, angehalten.

Die Speed-Ups von FScriptME für Parameterkombinationen mit 2 KiB Methoden-Cache sind ebenfalls in Abb. 4.4 dargestellt. Wie dort ersichtlich ist, erreicht der Speed-Up bei $S_8 = S_9 = 6,6$ einen Scheitelpunkt. Dass er nach $n = 9$ wieder abfällt statt einen Grenzwert zu erreichen, ergibt sich aus folgender Überlegung: Je mehr Prozessorkerne zur Verfügung stehen, desto mehr FScriptME-Threads werden parallel ausgeführt. Damit steigt auch die Belegung der Speicherbandbreite durch die Prozessorkerne, sodass weniger Bandbreite für den GC übrig bleibt, um den Heap zu durchsuchen oder den Speicher zu kompaktieren. Der GC wird somit ausgebremst und die GC-Performance sinkt.

Wird sogar die komplette Speicherbandbreite durch die Threads belegt, dann pausiert die Durchsuchung des Heaps als auch die Kompaktierung des Speichers[8]. Werden weiterhin Objekte alloziert, so stehen über kurz oder lang keine freien Referenzen und / oder kein freier Speicher mehr zur Verfügung. Sobald dies eintritt, wird ein Thread angehalten,

[8] Freigabe von Objekten ist nicht betroffen

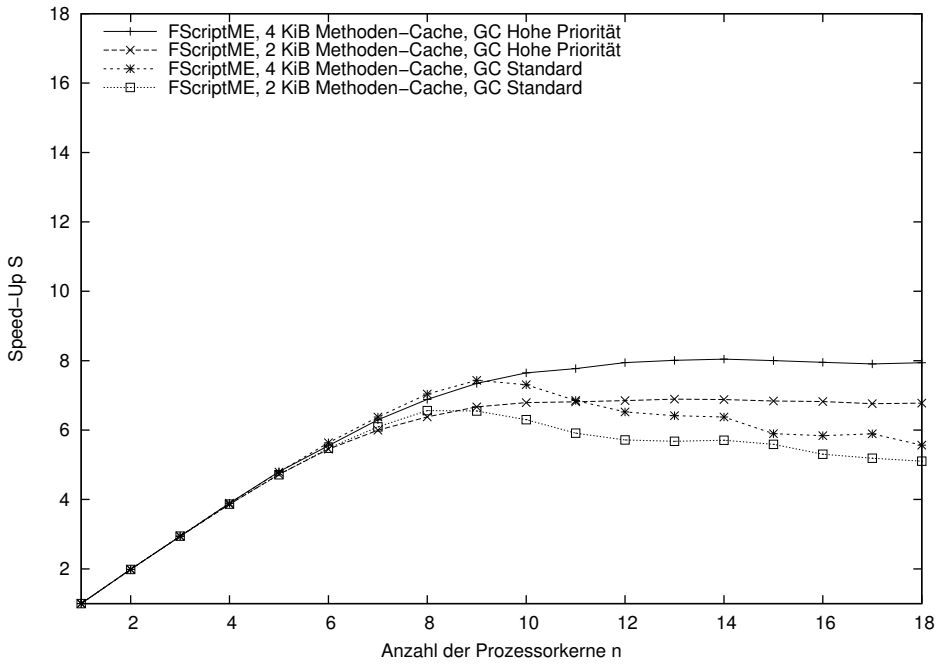

Abbildung 4.5: Speed-Up von FScriptME

sobald er versucht erneut ein Objekt zu allozieren. Sobald genügend Threads angehalten wurden, wird wieder Speicherbandbreite frei, sodass der GC obige Aktionen fortsetzen kann. Sobald wieder freie Referenzen und freier Speicher zur Verfügung stehen, wird die Ausführung der Threads wieder fortgesetzt.

Die Ursache in der Blockierung des GC liegt darin, dass dessen Zugriffe auf den Speicher am niedrigsten priorisiert sind. Daher wurde alternativ auch eine Änderung auf höchste Priorität geprüft und somit dem GC Speicherbandbreite zugesichert:

- *Fall 2 KiB Methoden-Cache*:

  Die Ausführungszeiten für einen Kern sind für beide GC-Prioritäten fast identisch: 25,115 s ohne und 25,192 s mit Priorisierung. Der Speed-Up fällt aber mit Priorisierung für $n > 9$ nicht mehr ab, sondern nähert sich dem Maximum von 6,8 an wie in Abb. 4.5 dargestellt. Basis für die Speed-Up-Berechnung ist die Zeit $t_{EXE}(1)$ ohne Priorisierung.

  Ebenfalls in der Abbildung ersichtlich: Der Speed-Up auf 8 Kernen mit Priorisierung (6,4) ist etwas kleiner als ohne Priorisierung (6,6). Dies liegt darin begründet, dass bereits hier der GC einen Anteil von 4,5% der Speicherbandbreite benötigt und durch die Priorisierung auch zugeteilt bekommt. Den FScriptME-Threads steht daher weniger Bandbreite zur Verfügung.

- *Fall 4 KiB Methoden-Cache*:

  Wird der Methoden-Cache auf 4 KiB erweitert, verkürzt sich zunächst die Ausführungszeit auf einem Kern um 4% von 25,115 s auf 24,221 s, jeweils ohne Priorisierung. Außerdem sind die erzielbaren Speed-Ups generell größer als bei 2 KiB Methoden-Cache, wie ebenfalls in Abb. 4.5 zu sehen ist. Basis für die Speed-Up-Berechnung ist diesmal die Zeit $t_{EXE}(1)$ ohne Priorisierung, aber mit 4 KiB Methoden-Cache.

  Ohne Priorisierung erreicht der Speed-Up einen Scheitelwert bei $S_9 = 7,5$ und fällt nach $n = 9$ wieder merklich ab. Abhilfe schafft auch hier die Priorisierung. Der Speed-Up nähert sich dann dem Maximum von 7,9 an. Die Ausführungszeit auf einem Kern ist wiederum fast identisch: 24,221 s ohne Priorisierung und 24,292 s mit Priorisierung.

Die Priorisierung des GC bietet keinen Vorteil. Einerseits liegt bei beiden Methoden-Cache-Größen das Speed-Up-Maximum mit Priorisierung nur unwesentlich höher als ohne Priorisierung. Andererseits werden mit Priorisierung bei kleineren Kernanzahlen zum Teil schlechtere Speed-Ups erzielt, obwohl für $n \leq 9$ noch genügend Speicherbandbreite zwischen den Speicherzugriffen der Kerne übrig ist. Im Sinne einer deterministischen Laufzeit ist so oder so eine Konfiguration zu wählen, bei der der GC nicht überlastet ist.

4.2.3 Effizienz

Die Effizienz der SHAP-Mehrkernarchitektur wird hier auf Basis der Verarbeitungsleistung beurteilt. Die Verarbeitungsleistung eines Prozessorkerns wird dazu separat für jeden Benchmark auf den Wert 1 normiert. Auf n Kernen ist dann theoretisch die n-fache Verarbeitungsleistung verfügbar. Die Effizienz entspricht damit der realen Nutzung / Auslastung der theoretischen Verarbeitungsleistung. Für die Berechnung der Effizienz wird der Speed-Up nach Amdahl entsprechend Gl. 2.9 aus Abschn. 2.5.4 verwendet.

Die Effizienz für die einzelnen Benchmarks ist in Abb. 4.6 dargestellt. Für die Abbildung sowie die folgenden Erläuterungen wurde die Konfigurationen mit 2 KiB Methoden-Cache verwendet.

- Der Crypt-Benchmark erreicht eine sehr gute Effizienz von 98% bei $n = 18$ Kernen.
- Die Effizienz des Queens-Benchmarks ist bedingt durch die notwendige Synchronisation etwas geringer, aber immer noch gut: $E_A(18) = 86\%$. Der Einbruch auf 95% bei 2 Kernen ist durch die Synchronisation bedingt, die bei einem Kern noch nicht notwendig ist. Mit Vergrößerung der Queens-Teilprobleme auf nur noch 4 vorplatzierte Damen kann die Effizienz auf $E_A(18) = 99\%$ gesteigert werden.

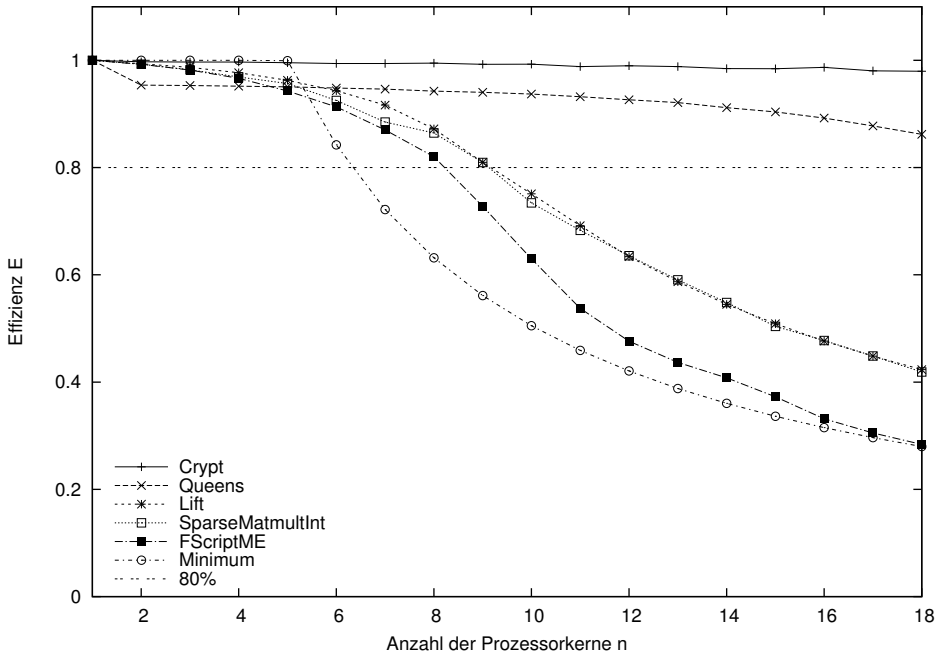

Abbildung 4.6: Effizienz auf Basis der Verarbeitungsleistung
2 KiB Methoden-Cache

- Die Effizienz der Programme Lift und SparseMatmultInt liegt nur für $n \leq 9$ Kernen über einem akzeptablen Wert von 80%. Für größere n fällt die Auslastung der Prozessorkerne merklich auf Werte um die 40% ab.
- Die 80%-Schranke hält der FScriptME-Benchmark nur für $n \leq 8$. Danach sinkt die Effizienz sogar auf 28% ab. Wird der GC priorisiert, ergibt sich qualitativ derselbe Verlauf. Die Effizienzen betragen nun: $E_A(8) = 80\%$ und $E_A(18) = 38\%$.
- Der Minimum-Benchmark zeigt, dass unter den in Abschn. 4.2.1.2 genannten Randbedingungen eine sehr gute Effizienz von $\geq 80\%$ für bis zu 6 Kerne erzielt werden kann. Danach fällt die Effizienz ab.

Fazit: Für typische parallelisierbare Programme ist der Einsatz von 8 bis 9 Kernen im SHAP-Mehrkernprozessor sinnvoll.

4.2.4 Vergleich

4.2.4.1 Allgemeines

Ein Vergleich zu anderen Java-Mehrkernprozessoren ist nur auf Basis von absoluter Performance aussagekräftig, denn wenn die Verarbeitungsleistung eines einzelnen Kerns reduziert wird, sinkt auch die benötigte Speicherbandbreite, womit potentiell höhere Speed-Ups möglich sind.

4.2.4.2 JopCMP

Für einen direkten Vergleich stehen nur die Messwerte des Prozessors JopCMP (vgl. Abschn. 2.6.2.1) zur Verfügung [PS08]. Der dort verwendete Lift-Benchmark ist identisch mit dem hier verwendeten und wurde dort auf dem Altera DE2-Board evaluiert. Für einen fairen Vergleich habe ich ebenfalls das DE2-Board herangezogen, denn dieses Board besitzt nur einen 16-Bit-Datenbus zum externen Speicher und damit eine deutlich reduzierte Speicherbandbreite (vgl. Abschn. 4.1.3).

Nach den Erkenntnissen aus Abschn. 3.3.1.3 belegt ein Lift-Benchmark-Thread auf SHAP 22% der verfügbaren Speicherbandbreite des DE2-Boards. Somit sollte eine parallele Ausführung von bis zu 4 Instanzen ohne Einbußen möglich sein.

Das JopCMP-System wird mit einer höheren Taktfrequenz als der SHAP-Mehrkernprozessor betrieben (90 MHz statt 60 MHz). Jedoch belegt jeder 32-Bit-Speicherzugriff den gemeinsam genutzten Speicherbus für 4 statt 2 Taktzyklen. Daher belegt der Lift-Benchmark auf dem JopCMP-System auch mehr Speicherbandbreite: 40% auf JopCMP mit 1 KiB Methoden-Cache und 32% auf JopCMP mit 2 oder 4 KiB Methoden-Cache.

Die Abb. 4.7 stellt SHAP und JopCMP gegenüber und zwar auf Basis von Lift-Ausführungen pro Millisekunde. Auf dem DE2-Board ist die Verarbeitungsleistung eines SHAP-Kerns trotz der wesentlich niedrigeren Taktfrequenz nur 1% geringer als die eines JopCMP-Kerns mit 2 KiB Methoden-Cache: 14000 gegenüber 14150 Ausführungen/s. Ab diesem Punkt skaliert die Verarbeitungsleistung etwa linear für bis zu 4 SHAP-Kerne ($S_A(4) = 3.2$) aber nur für bis zu 2 JopCMP-Kerne ($S_A(2) = 1.7$). Insbesondere ist die Leistung von 3 SHAP-Kernen größer als die von 4 oder 8 JopCMP-Kernen. Diese Performance-Lücke ist nicht durch den GC bedingt, denn der Lift-Benchmark alloziert Objekte nur während der Initialisierung.

Wie in Abb. 4.7 zu sehen, erreicht SHAP auf dem DE2-Board ein Maximum von 47710 Ausführungen/s. JopCMP erzielt hingegen nur 35690 bei 2 KiB Methoden-Cache und 27540 bei 1 KiB. Zum Vergleich ist auch die absolute Performance für bis zu 8 SHAP-Kerne auf dem XUPV5-Board angegeben. Die Verarbeitungsleistung von SHAP auf dem XUPV5-Board ist immer absolut größer als auf dem DE2-Board. Ursache ist die höhere Taktfrequenz und die größere Speicherbandbreite.

Ein weiterer Vergleich der Systeme ist auf Basis der eingesetzten FPGA-Ressourcen möglich. Hier ist zunächst festzustellen, dass JopCMP viel weniger LEs als der SHAP-µP benötigt, da dort u. a. der GC in Software statt in Hardware implementiert ist: JopCMP benötigt jeweils ≈3000 LEs und 45 kBit RAM pro Kern [PS08]. Da der Lift-Benchmark jedoch keinen GC erfordert, wird im Folgenden neben dem Standard „SHAP mit GC“ auch eine Variante ohne GC betrachtet (Ressourcenbedarf s. Abschn. 4.1.3.2).

In Abb. 4.8 ist die LE-Effizienz, d. h. die pro LE erzielte Verarbeitungsleistung, dargestellt. Zu sehen ist, dass nur im Bereich 4 . . . 8 Prozessorkerne der SHAP-µP in etwa

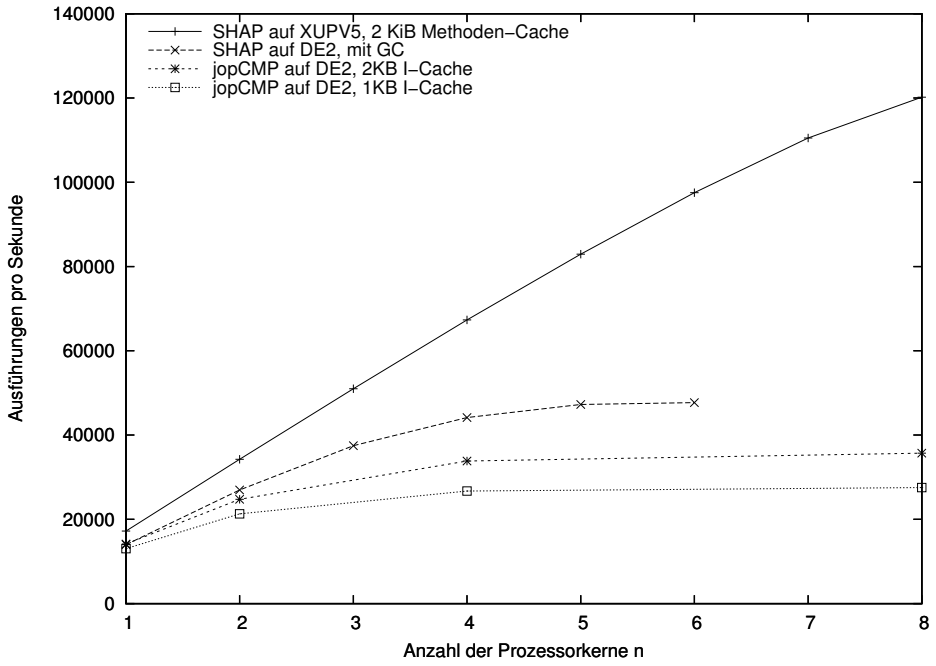

Abbildung 4.7: Vergleich der Performance zwischen SHAP und JopCMP

die gleiche Effizienz aufweist wie JopCMP. Die SHAP-Variante ohne GC schneidet dabei noch ca. 30% besser als die Variante mit GC ab. Für den Bereich $1 \ldots 3$ Kerne ist die LE-Effizienz von SHAP schlechter.

Im Bedarf an On-Chip-Speichern schneidet SHAP ohne GC mit 43832 Bit/Kern sogar geringfügig besser ab als JopCMP (45 kBit/Kern). SHAP mit GC benötigt jedoch wieder mehr Speicher.

4.2.4.3 REALJava

Für den Vergleich mit dem REALJava-Prozessor ist die dortige Multiplikation einer Matrix mit einem Vektor vergleichbar zu dem hier verwendeten SparseMatmultInt-Benchmark. Erstere kombiniert Zugriffe auf den Heap und lokalen Speicher und erreicht einen guten Speed-Up von 3.75 auf 4 JPUs [TSP10] was einer sehr guten Effizienz $E_A(4) = 94\%$ entspricht. Jedoch profitiert die Multiplikation nicht von noch mehr JPUs, stattdessen sinkt der Speed-Up sogar auf 3.63 auf 8 JPUs. Dies mag einerseits bedingt sein durch den stetig steigenden Bedarf an Speicherbandbreite. Andererseits nutzen die JPUs aber auch lokalen Daten-Speicher.

Für einen direkten Vergleich mit SHAP fehlen detaillierte Angaben zum Benchmark und zur absoluten Verarbeitungsleistung.

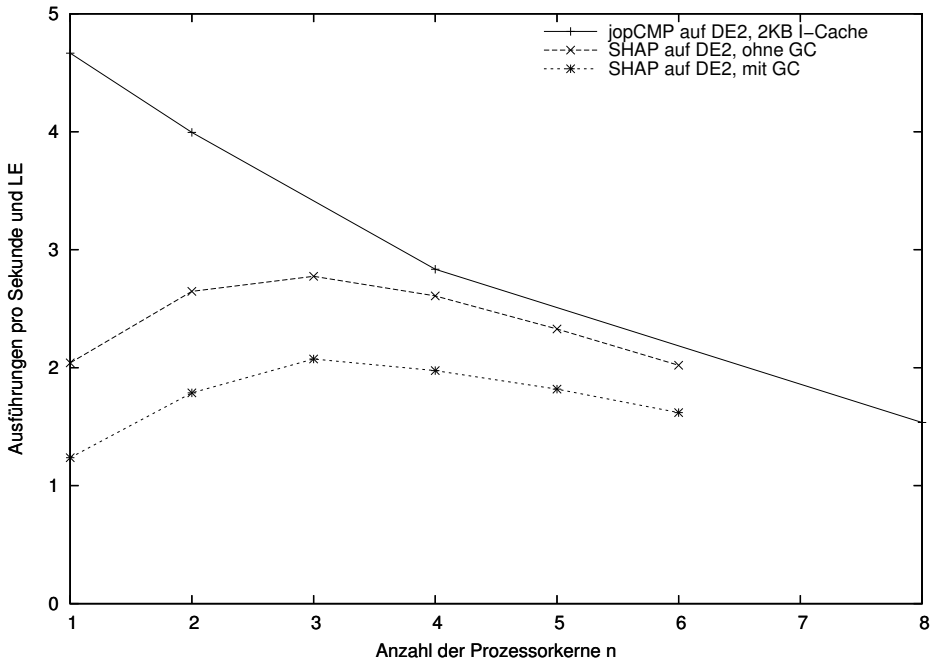

Abbildung 4.8: Vergleich der LE-Effizienz zwischen SHAP und JopCMP

4.2.4.4 jamuth-Mehrkernprozessor

Für die Leistungsbewertung von „jamuth“ mit mehreren Kernen hat S. Uhrig die Programme Sieve, Kfl und UdpIp der Benchmark-Suite JavaBenchEmbedded (in der Version 1.0) auf bis zu 3 Kernen ausgeführt [Uhr09]. Die Messungen wurden dort auf dem Entwicklungsboard „DBC3C40“ von Devboards durchgeführt.

Wenn nur 1 Benchmark-Thread pro Kern ausgeführt wird, beträgt der Speed-Up 1,8 für 2 Kerne und 2,6 für 3 Kerne. Dies entspricht Effizienzen von $E_A(2) = 90\%$ und $E_A(3) = 87\%$. Unter Nutzung der Mehrfädigkeit des Prozessors steigt der Speed-Up jedoch nur gering: z. B. 3,4 für 6 Threads (3 Kerne mit je 2 Threads).

Für einen direkten Vergleich mit SHAP fehlen Angaben zur absoluten Verarbeitungsleistung. Außerdem sind keine Speed-Ups für mehr als 3 Kerne bekannt.

4.3 Test

Dieser Abschnitt wertet die Durchführung der simulationsbasierten und der praktischen Tests an den Teilkomponenten sowie am gesamten SHAP-Mehrkernprozessor aus. Von Interesse sind dabei die an der ursprünglichen SHAP-Architektur vorgenommenen Änderungen. Die Bytecode-Ausführung eines einzelnen Prozessorkerns wird als korrekt angenommen.

4.3.1 Durchführung

4.3.1.1 Simulationsbasierter Test

Die VHDL-Beschreibung des SHAP-Mehrkernprozessors kann mit Hilfe des VHDL-Simulators „ModelSim“ von Mentor Graphics simuliert werden. Dieses Programm kann während der Simulation auch gleichzeitig Statistiken für die Beurteilung der Codeüberdeckung erstellen. Für die Simulation einzelner Komponenten werden wie allgemein üblich Testbenches eingesetzt.

Auch für die Simulation des Gesamtsystems steht eine Testbench bereit, die neben dem Mehrkernprozessor auch eine Beschreibung für den extern angebundenen Speicher bereitstellt. Für das schnelle Laden der SHAP-Datei wird ein I/O-Gerät eingesetzt, das direkt aus der Datei liest. Dafür waren keine Anpassungen am Mikrocode oder am SHAP-API notwendig.

Die Systemsimulation beginnt mit der Ausführung des Bootloaders im Mikrocode und endet (spätestens) mit dem Ende des Java-Programms. Es können all diejenigen Programme simuliert werden, die nicht auf externe Ereignisse (z. B. Tastatureingaben) angewiesen sind. Ausgaben sind möglich, wenn sie keine Bestätigungsnachrichten erfordern. Textausgaben der Java-Programme werden zur besseren Lesbarkeit in einer Textdatei gespeichert.

Im bestehenden VHDL-Code wurden Assert-Anweisungen ergänzt, die lokal den Systemzustand auf Gültigkeit prüfen. Detektiert wird:

- Lesen von einem nicht initialisierten Speicherplatz,
- ungültige Zustände der Objektreferenzen bei Allokation und Freigabe,
- ungültige Basisadressen bei Lese- und Schreibzugriffen auf Objekte.

4.3.1.2 Praktische Tests

Die praktischen Tests wurden auf dem XUPV5-Entwicklungsboard durchgeführt. Je nach Test wurde der SHAP-Mehrkernprozessor mit oder ohne Trace-Architektur synthetisiert und dann auf den FPGA programmiert.

4.3.2 Busarbitrierung

Der Zugriff mehrerer Prozessorkerne auf gemeinsam genutzte Busse wird mittels mehrerer Instanzen derselben Round-Robin-Arbiter-Komponente geregelt. Der Arbiter kann zur Kompilierzeit für zwei verschiedene Betriebsmodi konfiguriert werden:

- Im Modus „Single-Cycle Request“ wird der Bus jeweils nur für einen Takt zugeteilt. Dieser Modus wird für die Arbitrierung der Lese-/Schreibzugriffe der Speichermanager-Ports verwendet, die in einer Pipeline ausgeführt werden.

- Im Modus „Multi-Cycle Request“ erstreckt sich jeder Zugriff auf mehrere Takte. Er wird für die Arbitrierung des I/O-Busses sowie der Zugriffe auf die Objektverwaltung (Allokation von Objekten) genutzt.

In beiden Modi kennzeichnet ein Token, welcher Master zur Zeit die höchste Priorität besitzt. Die Priorität der anderen Master nimmt reihum mit steigendem Index ab. Bei mehreren gleichzeitigen Zugriffsversuchen erhält der Master mit der höchsten Priorität den Zugriff auf den Bus. Das Token wird an den Nachfolger (aufsteigender Index) weitergereicht. Erfolgt in einem Taktzyklus kein Buszugriff, dann verbleibt das Token an Ort und Stelle. Zu Beginn (Reset) erhält der Master mit dem kleinsten Index das Token.

Der Round-Robin-Arbiter wurde mit Hilfe einer Testbench für jeden Modus simulativ getestet. Die Ausgaben und inneren Zustände wurden manuell überprüft.

Die Codeüberdeckung setzt sich für beide Modi wie folgt zusammen:

- Der Arbiter besteht aus wenigen Anweisungen die zu 100% (= Statement-Coverage) abgedeckt wurden.
- Die Position des Token wird in einem Zustandsregister zwischengespeichert. Dieses wird über zwei Kontrollpfade nebst zugehörigen Ausdrücken geladen: erstens Reset und zweitens Wechsel zum nächsten Master. Die Path- und Expression-Coverage betrugen jeweils 100%.
- Der Arbiter enthält keinen endlichen Automaten.

Für die Funktionsüberdeckung ist zunächst festzuhalten, dass für jeden Master immer dieselben Verknüpfungen auszuführen sind:

1. Besitzt der Master das Token oder erhält er die Zugriffserlaubnis vom Vorgänger:
   a) Möchte er einen Zugriff ausführen, so erhält er auch Zugriff auf den Bus. Das Token erhält der nachfolgende Master.
   b) Möchte er keinen Zugriff ausführen, dann erhält der Nachfolger die Zugriffserlaubnis. Die Weitergabe stoppt, sobald wieder Master erreicht ist, der aktuell das Token hält.
2. Besitzt der Master keine Zugriffserlaubnis, dann kann er diese auch nicht weiterreichen.

Alle diese Fälle wurden geprüft und aufgetretene Fehler korrigiert.

Für das Multiplexing der Buszugriffe entsprechend der Arbitrierung wurde die allgemein bekannte VHDL-Anweisung

```
signal <= array_of_signals(integer);
```

verwendet. Dieses wurde nicht gesondert getestet.

4.3.3 Speicherzugriffe und Thread-Synchronisation

Die Überprüfung der einzelnen und konkurrierenden Speicherzugriffe unterscheidet sich in der Durchführung. Die atomaren Lese-Schreib-Operationen für die Thread-Synchronisation wurden wie die einzelnen Speicherzugriffe behandelt.

4.3.3.1 Einzelne und atomare Speicherzugriffe

Einzelne und atomare Speicherzugriffe wurden simulationsbasiert getestet. Dazu wurden Testbenches für

- die Ports des Speichermanagers[9] und
- den Speichercontroller am Beispiel des SRAM-Controllers

erstellt.

Die Ports sowie der Speichercontroller müssen das Protokoll des Speicherbusses implementieren und enthalten dazu jeweils einen Zustandsautomaten. Alle möglichen Kontrollpfade in den Automaten wurden überprüft und alle Zustände erreicht. Damit wurden auch alle VHDL-Anweisungen mindestens einmal ausgeführt. Bei Verzweigungsbedingungen wurden jeweils die einzelnen Teilausdrücke berücksichtigt. Damit beträgt die Codeüberdeckung in allen 4 Teilbereichen 100%.

Bei der Funktionsüberdeckung ergibt sich folgendes Bild:

- Es wurden alle zulässigen Befehle des Speichermanager-Ports getestet.
- Es wurden alle zulässigen Befehle des Speichercontrollers getestet.
- Bei Anforderung eines Monitors wurden folgende Zustände geprüft: „Monitor noch nicht belegt“, „Monitor durch denselben Thread belegt“ und „Monitor durch anderen Thread belegt“.

Der Speicherbus wurde nicht gesondert getestet, da dieser nur aus Pipelineregister, Multiplexern und der aus dem Einkernprozessor übernommen Adressberechnung besteht. Die Arbitrierung wurde bereits im Abschn. 4.3.2 behandelt.

Das korrekte Zusammenspiel aller Komponenten wurde anhand einer Simulation des Gesamtsystems manuell überprüft und aufgetretene Fehler korrigiert.

4.3.3.2 Konkurrierende Speicherzugriffe

Aufgrund der Vielzahl an möglichen Kombinationen von gleichzeitigen und aufeinander folgenden Speicherzugriffen wurde ein Test mit zufallsorientierter Auswahl der Testschritte durchgeführt.

[9] Die Ports sind eigene VHDL-Entitäten.

Für den Test wurde der Benchmark „SparseMatmultInt“ (s. a. Abschn. 4.2.1) verwendet, da er folgende vorteilhafte Eigenschaften aufweist:

- Das Testprogramm wird mit Hilfe mehrerer Threads auf mehreren Prozessorkernen ausgeführt. Es führt zu einer hohe Auslastung der Speicherbandbreite und führt damit zu konkurrierenden Speicherzugriffen.
- Jeder Thread führt zwar den gleichen Algorithmus aus, die Threads werden aber zeitversetzt gestartet.
- Jeder Thread arbeitet auf seinem eigenen Datenbereich. Damit ist keine Synchronisation notwendig und es wird auf verschiedene Adressen zugegriffen.
- Der Algorithmus greift auf 8 verschiedene Objekte zu und überprüft damit auch den TLB für die Adressberechnung.
- Die zeitlichen Abstände zwischen den Zugriffen sind unterschiedlich.
- Die Datenwerte sind pseudozufällig.
- Das Ergebnis der Matrixmultiplikation und damit auch die Speicherzugriffe werden abschließend durch Berechnung einer Checksumme geprüft.
- Es werden nur während der Initialisierung Objekte alloziert und damit kein GC benötigt. Der GC soll in diesem Test nicht geprüft werden und wurde daher deaktiviert.

Ein kleiner Nachteil des Benchmarks ist jedoch, dass Methoden nur während der Initialisierungsphase des Benchmarks und damit selten zwischengespeichert werden müssen. Die dafür zuständigen Ports am Speichermanager werden aber genauso behandelt wie die Ports für die Datenzugriffe.

Der Benchmark „FScriptME“ bietet sich zwar auch an, da dort ebenfalls die Speicherzugriffe pseudozufällig verteilt sind. Er alloziert jedoch sehr viele Objekte: Ohne GC ist nur ein Test mit wenigen Testschritten d. h. einer kurzen Laufzeit von ca. 40 ms möglich.

Zur Messung der exakten Anzahl der vom Benchmark ausgeführten Speicherzugriffe wurde der SHAP-µP mit der Trace-Architektur gekoppelt und für das XUPV5-Entwicklungsboard synthetisiert. Der SHAP-µP wurde mit einem Kern für den Referenzwert sowie mit 8, 12 und 16 Kernen konfiguriert, um den Bereich mit der maximalen Speicherauslastung abzudecken. Bei einer Synthese mit 18 Kernen und Trace-Architektur wird die gewünschte Taktfrequenz von 80 MHz nicht mehr erreicht.

Für jede SHAP-µP-Konfiguration wurde ein Lauf durchgeführt. Die Messung erstreckte sich nur über die parallel ausgeführte Matrixmultiplikation, nicht aber über die serielle Initialisierung und Checksummenberechnung. Die gemessene Anzahl der Speicherzugriffe und die benötigte Laufzeit sind in Tab. 4.3 aufgelistet. Daraus wurden die durchschnittliche Anzahl der Speicherzugriffe pro ms und die durchschnittliche Auslastung

Tabelle 4.3: Anzahl der Speicherzugriffe und Auslastung der Speicherbandbreite für SparseMatmultInt

Durchlauf	Kerne	Laufzeit in ms	Speicherzugriffe absolut	Speicherzugriffe pro ms	Auslastung d. Bandbreite
1	1	10517	106631479	10139	12.7%
2	8	1519	107057757	70479	88.1%
3	12	1379	109415873	79344	99.2%
4	16	1377	110118961	79970	100.0%

der Speicherbandbreite berechnet. Bei 80 MHz Takt sind maximal 80000 Speicherzugriffe pro Millisekunde möglich. Wie zu sehen ist, wird bei 16 Kernen nahezu die gesamte Speicherbandbreite ausgenutzt. Daraus kann man eine hohe Zahl konkurrierender Speicherzugriffe ableiten, zumal ab 8 Kernen der Speed-Up nur noch geringfügig ansteigt (vgl. Abschn. 4.2.2.2).

Die Berechnung der Matrixmultiplikation war in jedem Durchlauf korrekt. Der Test wurde somit erfolgreich absolviert.

4.3.4 Speicherverwaltung

Hier ist zwischen dem Test einzelner Komponenten und dem Test der Speicherverwaltung insgesamt zu unterscheiden.

4.3.4.1 Einzelne Komponenten

In die Ausführung von Objektallokationen und GC-Operationen sind eine Reihe von Hardwarekomponenten der Speicherverwaltung sowie die Firmware des GC involviert.

Die Verwaltung der Speichersegmente über die GC-Firmware kann jedoch praktisch nicht simulativ getestet werden. Die in C geschriebene Firmware greift häufig auf Daten zurück, die außerhalb der ZPU (z. B. in den GC-Markierungstabellen, in den Objekten auf dem Heap) gespeichert sind. Eine Simulation würde daher erfordern, das restliche SHAP-System ebenfalls nachzubilden. Daher wurde lediglich eine statische Codeanalyse durchgeführt: Mit Hilfe des Werkzeugs „CMBC“ der University of Oxford wurde geprüft, ob die Zugriffe auf die (in der Firmware deklarierte) Tabelle mit den Zustandsinformationen für die Speichersegmente alle gültig sind. Auf die Tabelle wird ausschließlich indiziert zugegriffen, sodass jeweils ein Bounds-Check für die einzelnen Funktionen der Segmentverwaltung durchgeführt wurde. Im einzelnen wurden folgende Funktionen mit den Optionen `-bounds-check` und `-all-claims` überprüft:

- Segmentliste initialisieren (`seg_initialize`)

- Allokationssegment austauschen (`seg_fetsch_aseg` und `seg_push_aseg`)
- Segmentzustand aktualisieren (`seg_update_d`): Der Quellcode dieser Funktion musste angepasst werden, um verschiedene reale Situationen zu simulieren.
- Auswahl von Segmenten (`seg_get_new_target_seg` und `seg_get_first_used`)

Alle Überprüfungen waren erfolgreich.

Auf einen Test einzelner Teilkomponenten der Speicherverwaltung wurde verzichtet. Stattdessen wurde deren Zusammenspiel und damit die Ausführung ganzer Objektallokationen und GC-Operationen im folgenden Abschnitt getestet. Zu den potenziellen Komponenten gehören:

- Verwaltung des Allokationssegments,
- Zugriff auf den GC-Bus,
- Steuerung des Stack-Scans,
- die einzelnen Operationen der GC-Markierungstabellen,
- die einzelnen Funktionen der Objektverwaltung wie das Anlegen, Durchsuchen, Verschieben und Freigeben von Objekten — dies umfasst nur den Zustandsautomaten, der Rest besteht aus Busmultiplexern und einfachen Adressgeneratoren.

4.3.4.2 Gesamtheit

Der separate Test einzelner Komponenten ist nur wenig aussagekräftig. Von Interesse ist vielmehr das Zusammenspiel der einzelnen Komponenten der Speicherverwaltung während der Objektallokation und Abarbeitung der GC-Operationen. Erschwerend kommt hinzu, dass während eines GC-Zyklus die Kerne weiterhin Objekte allozieren können.

Der Test der Speicherverwaltung wurde praktisch am FPGA-Prototypen durchgeführt. Zum Einsatz kam die Trace-Architektur mit der folgende Kommandos nebst deren Parameter aufgezeichnet wurden:

- von den Prozessorkernen angeforderte Operationen: Objekt anlegen, beleben und auf Weak-Reference prüfen;
- vom GC ausgeführte Operationen: Referenzzustand lesen und schreiben, Objekt nach Referenzen durchsuchen, Objekt verschieben sowie freigegeben;
- von der Objektverwaltung delegierte Operationen: Platz im Allokationssegment prüfen sowie für ein Objekt reservieren;
- auf dem GC-Bus übertragene Nachrichten.

In einem ersten Schritt wurde der GC immer nur manuell gestartet. Die Heap-Zugriffe des Programms waren damit zeitlich von denen des GC getrennt. Mithilfe eines spezialisierten Programms wurden nacheinander folgende Teilaspekte geprüft:

1. Anlegen einer Reihe von Objekten unterschiedlicher Größe und Überprüfung der Position im Speicher.
2. Überprüfung der Verschiebung vom Allokationssegment in ein GC-Zielsegment.
3. Freigabe einzelner Objekte, sodass die Auslastung der GC-Zielsegmente unter eine bestimmte Grenze fällt. Daraufhin werden die Objekte in ein neues Zielsegment verschoben und damit die Kompaktierung überprüft.
4. Löschen aller Referenzen auf die angelegten Objekte und damit Prüfung der Freigabe aller dieser Objekte.

Ebenso wurde überprüft, ob die Informationen aus den GC-Markierungstabellen korrekt übertragen werden.

In einem zweiten Schritt wurde der GC wieder in seiner Standardkonfiguration und damit nebenläufig zum Java-Programm ausgeführt. Ziel war es, etwaige Race-Conditions aufzudecken. Dazu wurde für jede Referenz protokolliert, wann sie angelegt, gescannt, verschoben und wieder freigegeben wurde. Ein weiteres Programm hat dann auf dem Desktop-PC den Trace auf folgende Fehler hin überprüft:

- Freigabe einer Referenz, die im selben GC-Zyklus beim Scan gefunden wurde,
- Freigabe einer Referenz, die noch gar nicht alloziert wurde,
- Verschiebung einer noch nie alloziierten oder einer bereits freigegebenen Referenz,
- Freigabe einer bereits freigegebenen Referenz und damit auch die mehrfache Freigabe derselben Referenz hintereinander,
- fehlerhaftes Lesen und Schreiben von Referenzzuständen während des Verschiebens eines Objekts.
- Beim Anlegen eines Objekts wird nicht der Kopf der Liste der freien Referenzen verwendet. Dies schließt die Allokation einer bereits allozierten Referenz ein.

Als Testprogramm wurde der FScriptME-Benchmark verwendet, da dessen Speicherzugriffe aus Sicht der Speicherverwaltung pseudozufällig sind. Somit ist eine gute Funktionsüberdeckung zu erwarten. FScriptME wurde zudem in der Konfiguration für 1, 4 und 8 Kerne ausgeführt.

Als weiterer Zufallstest kam ein spezialisiertes Testprogramm zum Einsatz, dass ohne Pause Objekte zufälliger Größe unter Verwendung mehrerer Kerne alloziert. Die Referenzen neuer Objekte werden an eine zufällige Position in einer Tabelle eingetragen. Die

dort vorher gespeicherte Referenz wird damit gelöscht, sodass der GC das Objekt wieder freigeben kann. Somit wird der GC auch bei maximaler Speicherauslastung getestet.

Alle gefundenen Fehler wurden korrigiert. Nach erneuter Ausführung aller Teiltests wurden diese erfolgreich absolviert.

4.3.5 Gesamtsystem

Der Test des Gesamtsystems erfolgte über die Ausführung verschiedener Testprogramme und die manuelle Überprüfung der Ausgaben. Der Test wurde mit dem XUPV5-Entwicklungsboard und $n = 1 \ldots 18$ Prozessorkernen durchgeführt.

Queens: Die von dem Benchmark Queens berechnete Lösungsanzahl wurde überprüft. Für den Test wurde die Konfiguration 13×13 Felder mit 5 vorplatzierten Damen genutzt (73712 Lösungen).

Dieses Testprogramm deckt die Busarbitrierung, die Thread-Synchronisation und das Thread-Scheduling ab.

FScriptME: Dieser Benchmark wurde so geändert, dass die Länge der Collatz-Reihe für die 2. bis 43. Fibonacci-Zahl auf der seriellen Schnittstelle ausgegeben wird. Die Ausgabe wurde mit den Werten verglichen, die die Ausführung des Java-Programms auf einem PC liefert: 0, 1, 7, 5, 3, 9, 7, 13, 112, 30, 23, 83, 63, 38, 36, 122, 102, 33, 137, 161, 79, 103, 145, 112, 180, 129, 246, 102, 131, 186, 135, 128, 245, 119, 112, 273, 240, 127, 244, 299, 191, 290.

Dieses Testprogramm deckt die Busarbitrierung, die Speicherverwaltung und das Thread-Scheduling ab.

SparseMatmultInt: Die Matrixelemente werden mit Pseudozufallszahlen initialisiert. Neben dem bereits vorgegebenen Startwert, wurde der Zufallszahlengenerator noch mit 2 weiteren Startwerten initialisiert. Zum Abschluss bildet das Programm bereits eine Prüfsumme und vergleicht diese mit einem bekannten Wert, der durch Ausführung des Java-Programms auf einem PC ermittelt wurde.

Startwert	Prüfsumme
10101010	1410034512
20202020	-1439223032
30303030	-216845024

Dieses Testprogramm deckt die Busarbitrierung, die Thread-Synchronisation und das Thread-Scheduling ab.

Alle Testprogramme lieferten ein positives Ergebnis. Somit war der Test des Gesamtsystems erfolgreich.

5 Zusammenfassung und Ausblick

5.1 Zusammenfassung

Die vorliegende Arbeit ordnet sich in die Nutzung von Parallelität auf Thread-Ebene unter Verwendung eines Mehrkernprozessors ein. Im Speziellen werden Konzepte für Java-Prozessoren untersucht, welche nativ Java-Bytecode ausführen können. Dazu wird zuerst der Stand der Forschung und Technik auf diesem Gebiet studiert. Es folgen Analysen zur Realisierbarkeit von Mehrkern-Java-Bytecode-Architekturen sowohl im Allgemeinen als auch speziell für die SHAP-Plattform. Abschließend wird der selbst gewählte Ansatz anhand einer prototypischen Implementierung hinsichtlich Skalierbarkeit, Verarbeitungsleistung, Effizienz und Funktionalität bewertet.

Alle Ziele dieser Arbeit wurden erreicht. Die im Einzelnen erzielten Ergebnisse seien hier kurz zusammengefasst und bewertet:

- Die detaillierte Analyse von Java-Mehrkernprozessoren zeigt, dass für Bytecode und Heap zentrale gemeinsame Speicher eingesetzt werden. Die einzelnen Ansätze unterscheiden sich jedoch u. a. hinsichtlich Stack-Speicher, Mikroarchitektur, Verbindungsnetzwerken und der Speicherverwaltung. Ein zentraler gemeinsamer Speicher ist auch bei eingebetteten klassischen Mehrkernprozessoren zu finden, die jedoch über andere Speicherhierarchien angebunden sind.

  Die Analyse zeigte außerdem die Aktualität des Forschungsbereichs. Für eingebettete Systeme ist deutlich der Trend zu Mehrkernprozessoren erkennbar.

- Die eigenen Untersuchungen zeigen vielfältige Lösungsvarianten für die Realisierung von Mehrkern-Java-Bytecode-Architekturen auf. Während für den Stack prinzipiell verteilte Speicher vorzuziehen sind, muss für Bytecode und Heap zwischen Vor- und Nachteilen abgewogen werden. Verteilte Speicher bieten zudem die Möglichkeit den GC zu beschleunigen. I/O-Komponenten sollten an alle Prozessorkerne angeschlossen werden, Komponenten mit hohem Durchsatz außerdem per DMA an den Heap-Speicher. Das Thread-Scheduling ist auf die Anbindung der Stack-Speicher und I/O-Komponenten abzustimmen. Für die Thread-Synchronisation ist eine Verwaltung der Monitore in Software ausreichend. Als Schnittstelle für Test und Diagnose ist eine Trace-Architektur zweckmäßig.

Die Untersuchungen berücksichtigten alle relevanten Aspekte der JVM-Spezifikation. Trotz sorgfältiger Abwägung von Vor- und Nachteilen kann die Entscheidung zugunsten einer Lösungsvariante aber erst anhand eines konkreten Java-Prozessors nebst seiner Plattform getroffen werden.

- Der Lösungsansatz für die SHAP-Plattform umfasst einen zentralen gemeinsamen Speicher für Heap und Bytecode, dem Einsatz von Pipelining und eines Vollduplex-Speicherbusses, verteilten Stack-Speichern, einem teilweise verteilten GC und ein 2-stufiges Thread-Scheduling. Thread-Synchronisation, I/O und die Schnittstellen für Test und Diagnose wurden entsprechend den obigen Vorschlägen implementiert.

  Es wurde ein neuartiger Ansatz für eine Mehrkern-Java-Bytecode-Architektur vorgestellt. Die Auswahl der Lösungsvariante berücksichtigte dabei die speziellen Eigenschaften der SHAP-Plattform. Eine Analyse der belegten Speicherbandbreite auf dem Einkernprozessor weist dazu Reserven für bis zu 10 Kerne aus. Die Kombination von Java-Prozessor und Trace-Architektur ermöglicht Detailuntersuchungen an Hard- und Software, ohne die Ausführung von Java-Programmen zu beeinflussen.

- Es stehen optimierte prototypische Implementierungen des Mehrkernprozessors für verschiedene FPGA-Entwicklungsboards bereit. Die Implementierungen können in verschiedenen Punkten, u. a. der Anzahl der Prozessorkerne konfiguriert werden. Bei der Parametrierung der Prototypen wurde die Kompatibilität zur SHAP-Plattform (und damit auch zur standardisierten CLDC der JavaME) gewahrt.

  Es konnte nachgewiesen werden, dass bis zu 18 Kerne mit der gleichen maximalen Taktfrequenz des Einkernprozessors auf einem FPGA integriert werden können. Weiterhin wächst der Bedarf an FPGA-Ressourcen nur linear mit der Kernanzahl. Damit ergibt sich eine sehr gute Skalierbarkeit des gewählten Ansatzes.

- Die neu implementierten und modifizierten Komponenten wurden getestet. Ebenso wurde das Gesamtsystem ausführlich mithilfe von Benchmarks und der Trace-Architektur überprüft. Dabei erkannte Fehler wurden entsprechend der Spezifikation korrigiert.

  Mit den durchgeführten Maßnahmen wurde die korrekte Funktion entsprechend der JVM-Spezifikation nachgewiesen. Mir ist bewusst, dass der Test nur die Anwesenheit von Fehlern zeigt, nicht deren Abwesenheit.

- Eine anschließende Leistungsbewertung unter Verwendung verschiedener, repräsentativer Benchmarks zeigt, dass selbst für Programme mit überdurchschnittlich häufigen Speicherzugriffen ein Speed-Up von bis zu 8 erzielt wird und damit eine sehr gute Effizienz bei bis zu 9 Kernen erreicht wird. Selbst bei dichtester Abfol-

ge von Objektzugriffen ist noch ein Speed-Up von 5 erzielbar. Ebenso wird eine deutlich höhere Leistung gegenüber anderen bekannten Arbeiten nachgewiesen.

Die gewählte Speicheraufteilung und -anbindung hat sich bewährt. Neben den obligatorischen Verbindungsnetzwerken ist kein zusätzlicher Hardwareaufwand (wie Objekt-Caches) vonnöten, um eine hohe Effizienz bei vielen Kernen zu erzielen.

Die vorliegende Arbeit erläutert detailliert die Gestaltung eines Java-Mehrkernprozessors zur effizienten Nutzung der auf Thread-Ebene ohnehin vorhandenen Parallelität eines Java-Programms. Die zentrale Fragestellung wird damit positiv beantwortet.

5.2 Ausblick

Im Laufe der Bearbeitung des Themas traten neue Fragestellungen auf, die sich für zukünftige Arbeiten anbieten:

- Die Integration sehr vieler Kerne (Stichwort „Many Core“) erfordert neben einer breitbandigeren Speicheranbindung auch Maßnahmen zur Entlastung der Heap-Speicherschnittstelle. Ein mögliches Arbeitsthema hierzu wäre, neben den klassischen Verfahren (Daten-Cache, TLB) auch spezielle Ansätze zu untersuchen, die sich Eigenschaften der JVM zunutze machen. Die Betrachtungen sollten dabei auch das Kosten-Nutzen-Verhältnis und die Verlustleistung berücksichtigen.
- Der Einsatz des ZPU-Mikroprozessors für den GC führt aktuell zu einem heterogenen System. Effizienter wäre jedoch ein homogenes System mit einem zusätzlichen SHAP-Kern anstelle der ZPU. Die Verarbeitungsleistung des zusätzlichen Prozessorkerns kann dann adaptiv für den GC oder Berechnungen des Programms genutzt werden. Die Probleme einer Verlagerung des GC auf einen SHAP-Kern wurden in der vorliegenden Arbeit bereits angesprochen. Zukünftige Arbeiten sollten daher die Untersuchungen vertiefen und auch die Initialisierungsphase mit einbeziehen.
- Das Ziel des aktuellen Schedulers der SHAP-Plattform besteht darin, die verfügbare Rechenleistung gleichmäßig auf alle Threads zu verteilen. Insbesondere für Echtzeitsysteme ist jedoch mehr Flexibilität wünschenswert. Die Frage wäre, wie die Echtzeitanforderungen unter den Randbedingungen des SHAP-Mehrkernprozessors erfüllt werden können.

Literaturverzeichnis

[ABF94] ABRAMOVICI, Miron ; BREUER, Melvin A. ; FRIEDMAN, Arthur D.: *Digital Systems Testing and Testable Design*. Revised printing. New York : Wiley-IEEE Press, 1994. – ISBN 978–0–780–31062–9

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

[Amd67] AMDAHL, Gene: Validity of the single processor approach to achieving large-scale computing capabilities. In: *Proc. 1967 Spring Joint Computer Conf. (AFIPS'67, Spring)* Bd. 30, 1967, S. 483–485

[BC04] BECK, Antonio C. S. ; CARRO, Luigi: A VLIW low power Java processor for embedded applications. In: *Proc. 17th Symp. Integrated Circuits and System Design (SBCCI'04)*, ACM, 2004. – ISBN 1–58113–947–0, S. 157–162

[Ber06] BERGERON, Janick: *Writing Testbenches Using System Verilog*. New York : Springer Science+Business Media, 2006. – ISBN 978–0–387–29221–2

[BGA05] BAILEY, Brian ; GRANT, Martin ; ANDERSON, Thomas: *Taxonomies for the Development and Verification of Digital Systems*. New York : Springer Science+Business Media, 2005. – ISBN 978–0–387–24019–0

[BGH+06] BLACKBURN, Stephen M. ; GARNER, Robin ; HOFFMANN, Chris ; KHANG, Asjad M. ; MCKINLEY, Kathryn S. ; BENTZUR, Rotem ; DIWAN, Amer ; FEINBERG, Daniel ; FRAMPTON, Daniel ; GUYER, Samuel Z. ; HIRZEL, Martin ; HOSKING, Antony ; JUMP, Maria ; LEE, Han ; MOSS, J. Eliot B. ; MOSS, B. ; PHANSALKAR, Aashish ; STEFANOVI'C, Darko ; VANDRUNEN, Thomas ; DINCKLAGE, Daniel von ; WIEDERMANN, Ben: The DaCapo benchmarks: Java benchmarking development and analysis. In: *Proc. 21st ACM SIGPLAN Conf. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'06)*, ACM, 2006. – ISBN 1–59593–348–4, S. 169–190

[BKP99] BEREKOVIC, Mladen ; KLOOS, Helge ; PIRSCH, Peter: Hardware realization of a Java Virtual Machine for high performance multimedia applications. In: *J. VLSI Signal Process. Syst.* 22 (1999), Nr. 1, S. 31–43. – ISSN 0922–5773

[Bol07] BOLLELLA, Greg (Hrsg.): *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems, JTRES 2007, Institute of Computer Engineering, Vienna University of Technology, 26–28 Sept. 2007, Vienna, Austria*. ACM, 2007 . – ISBN 978–1–59593–813–8

[Bor09] BORMANN, Jörg: *Vollständige funktionale Verifikation*, Kaiserslautern, Universität, Diss., 2009

[C+10] CRAWFORD, Dick u. a.: *gnuplot 4.4 : An Interactive Plotting Program*, März 2010. http://www.gnuplot.info, Abruf: 30. Mai 2011

[Ebe10] EBERT, Peter: *Entwurf und Implementierung einer Applikationsverwaltung für den SHAP-Bytecodeprozessor*, Dresden, Technische Universität, Großer Beleg, 2010

[EKEL00] EL-KHARASHI, M. W. ; ELGUIBALY, F. ; LI, K. F.: A quantitative study for Java microprocessor architectural requirements. Part II: high-level language support. In: *Microprocessors and Microsystems* 24 (2000), Nr. 5, S. 237–250. – ISSN 0141–9331

[EKEL01] EL-KHARASHI, M. W. ; ELGUIBALY, Fayez ; LI, Kin F.: Adapting Tomasulo's algorithm for bytecode folding based Java processors. In: *SIGARCH Comput. Archit. News* 29 (2001), Nr. 5, S. 1–8. – ISSN 0163–5964

[GH05] GATZKA, Stephan ; HOCHBERGER, Christian: The AMIDAR class of reconfigurable processors. In: *J. Supercomput.* 32 (2005), Nr. 2, S. 163–181. – ISSN 0920–8542

[Gus88] GUSTAFSON, John L.: Reevaluating Amdahl's law. In: *Commun. ACM* 31 (1988), S. 532–533. – ISSN 0001–0782

[GW07] GRUIAN, Flavius ; WESTMIJZE, Mark: BlueJEP: A flexible and high-performance Java embedded processor. In: [Bol07], S. 222–229

[Har01] HARDIN, David S.: Real-time objects on the bare metal: An efficient hardware realization of the Java virtual machine. In: *Proc. 4th Int'l Symp. Object-Oriented Real-Time Distributed Computing (ISORC'01)*, IEEE Computer Society Press, 2001. – ISBN 0–7695–1089–2, S. 53

[HMHG07] HEINISCH, Cornelia ; MÜLLER-HOFMANN, Frank ; GOLL, Joachim: *Java als erste Programmiersprache*. 5., überarb. u. erw. Aufl. Wiesbaden : B.G. Teubner Verlag, 2007. – ISBN 978–3–8351–0147–0

[HP07] HENNESSY, John L. ; PATTERSON, David A.: *Computer Architecture : A Quantitative Approach*. 4th edition. San Francisco : 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: [KV10], S. 26–35

[ICJ01] ITO, Sérgio A. ; CARRO, Luigi ; JACOBI, Ricardo P.: Making Java work for microcontroller applications. In: *IEEE Design Test of Comp.* 18 (2001), Nr. 5, S. 100–110. – ISSN 0740–7475

[Kem07] KEMNITZ, Günter: *Test und Verlässlichkeit von Rechnern*. Berlin : Springer-Verlag, 2007. – ISBN 978–3–540–45963–7

[KKT+00] KIMURA, Shinji ; KIDA, Hiroyuki ; TAKAGI, Kazuyoshi ; ABEMATSU, Tatsumori ; WATANABE, Katsumasa: An application specific Java processor with reconfigurabilities. In: *Proc. 2000 Conf. Asia South Pacific Design Automation (ASP-DAC'00)*, ACM, 2000. – ISBN 0–7803–5974–7, S. 25–26

[Kru09] KRUSE, Hans G.: *Leistungsbewertung bei Computersystemen*. Berlin : Springer-Verlag, 2009. – ISBN 978–3–540–71053–0

[KV10] KALIBERA, Tomás (Hrsg.) ; VITEK, Jan (Hrsg.): *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems, JTRES 2010, Prague, Czech Republic, August 19–21, 2010*. ACM, 2010 . – ISBN 978–1–4503–0122–0

[Lam05] LAM, William K.: *Hardware Design Verification : Simulation and Formal Method-Based Approaches*. Prentice Hall Computer, 2005. – ISBN 978–0–130–22561–0

[LY99] LINDHOLM, Tim ; YELLIN, Frank: *The Java(TM) Virtual Machine Specification*. 2nd edition. Amsterdam : Addison-Wesley Longman, 1999. – ISBN 978–0201432947

[MO98] MCGHAN, Harlan ; O'CONNOR, Mike: PicoJava: A direct execution engine for Java bytecode. In: *Computer* 31 (1998), Nr. 10, S. 22–30. – ISSN 0018–9162

[Ope02] OPENCORES ORGANIZATION (Hrsg.): *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*. Rev. B.3. World Wide Web: OpenCores Organization, 2002

[OT97] O'CONNER, J. M. ; TREMBLAY, Marc: picoJava-I: The Java Virtual Machine in hardware. In: *IEEE Micro* 17 (1997), Nr. 2, S. 45–53. – ISSN 0272–1732

[PH05] PATTERSON, David A. ; HENNESSY, John L.: *Rechnerorganisation und -entwurf*. 3. Aufl. München : Spektrum Akademischer Verlag, 2005. – ISBN 3–8274–1595–0

[PH09] PRADHAN, Dhiraj K. ; HARRIS, Ian G.: *Practical Design Verification*. 1st edition. Cambridge University Press, 2009. – ISBN 978–0–521–85972–1

[Por05a] PORTHOUSE, Chris: High performance Java on embedded devices / ARM Ltd. Cambridge, 2005. – Firmenschrift. – http://www.arm.com/documentation/White_Papers/index.html

[Por05b] PORTHOUSE, Chris: Jazelle for Execution Environments / ARM Ltd. Cambridge, 2005. – Firmenschrift. – http://www.arm.com/documentation/White_Papers/index.html

[Pre03] PREUSSER, Thomas B.: *Entwicklung eines Prozessorsimulators unter besonderer Berücksichtigung des Organic Computing*, Dresden, Technische Universität, Diplomarbeit, 2003

[Pre11] PREUSSER, Thomas B.: *Increasing the Performance and Predictability of the Code Execution on an Embedded Java Platform*, Dresden, Technische Universität, Diss., 2011

[PS07] PITTER, Christof ; SCHOEBERL, Martin: Towards a Java multiprocessor. In: [Bol07], S. 144–151

[PS08] PITTER, Christof ; SCHOEBERL, Martin: Performance evaluation of a Java chip-multiprocessor. In: *Proc. 3rd Int'l Symp. Industrial Embedded Systems (SIES'08)*, IEEE Press, 2008, S. 34–42

[PS10] PITTER, Christof ; SCHOEBERL, Martin: A real-time Java chip-multiprocessor. In: *ACM Trans. Embed. Comput. Syst.* 10 (2010), Nr. 1, S. 9:1–34. – ISSN 1539–9087

[PZR07] PREUSSER, Thomas B. ; ZABEL, Martin ; REICHEL, Peter: The SHAP Microarchitecture and Java Virtual Machine / Technische Universität Dresden, Fakultät Informatik. 2007 (TUD-FI07-02). – Forschungsbericht. – ISSN 1430–211X. – ftp://ftp.inf.tu-dresden.de/pub/berichte/tud07-02.pdf

[PZS07] PREUSSER, Thomas B. ; ZABEL, Martin ; SPALLEK, Rainer G.: Bump-pointer method caching for embedded Java processors. In: [Bol07], S. 206–210

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

[RP06] RECHENBERG, Peter ; POMBERGER, Gustav: *Informatik-Handbuch*. 4. Aufl. München : Carl Hanser Verlag, 2006. – ISBN 978–3–446–40185–3

[RTJ00] RADHAKRISHNAN, Ramesh ; TALLA, Deependra ; JOHN, Lizy K.: Allowing for ILP in an embedded Java processor. In: *Proc. 27th Int'l Symp. Computer Architecture (ISCA'00)*, ACM, 2000. – ISBN 1–58113–232–8, S. 294–305

[Sch05] SCHOEBERL, Martin: *JOP: A Java Optimized Processor for Embedded Real-Time Systems*, Wien, Universität, Diss., 2005

[Sch06] SCHOEBERL, Martin: A time predictable Java processor. In: *Proc. Conf. Design, Automation and Test in Europe (DATE'06)*. Leuven : European Design and Automation Association, 2006. – ISBN 3–9810801–0–6, S. 800–805

[SEP06] SIDERIS, Isidoros ; ECONOMAKOS, George ; PEKMESTZI, Kiamal: A cache based stack folding technique for high performance Java processors. In: RICHARD-FOY, Marc (Hrsg.): *Proc. 4th Int'l Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES'06)*, ACM, 2006. – ISBN 1–59593–544–4, S. 48–57

[SPU10] SCHOEBERL, Martin ; PREUSSER, Thomas B. ; UHRIG, Sascha: The embedded Java benchmark suite JemBench. In: [KV10], S. 120–127

[SS07] SIEMERS, Christian ; SIKORA, Axel: *Taschenbuch Digitaltechnik*. 2. Aufl. München : Carl Hanser Verlag, 2007. – ISBN 978–3–446–40903–3

[Sun99a] SUN MICROSYSTEMS, INC. (Hrsg.): *JSR-100, Real-Time Specification for Java*. Version 1.0.2. World Wide Web: Sun Microsystems, Inc., 1999. – http://java.sun.com/javase/technologies/realtime.jsp

[Sun99b] SUN MICROSYSTEMS, INC. (Hrsg.): *picoJava-II Processor Core Description*. Palo Alto: Sun Microsystems, Inc., 1999

[Sun00] SUN MICROSYSTEMS, INC. (Hrsg.): *J2ME Building Blocks for Mobile Devices*. Palo Alto: Sun Microsystems, Inc., 2000

[Sun03] SUN MICROSYSTEMS, INC. (Hrsg.): *Connected Limited Device Configuration : Specification*. Version 1.1. Santa Clara: Sun Microsystems, Inc., 2003

[SW07] SCHNEIDER, Uwe ; WERNER, Dieter: *Taschenbuch der Informatik*. 6. Aufl. München : Carl Hanser Verlag, 2007. – ISBN 978–3–446–40754–1

[TCC+00] TREMBLAY, M. ; CHAN, J. ; CHAUDHRY, S. ; CONIGLIAM, A.W. ; TSE, S.S.: The MAJC architecture: A synthesis of parallelism and scalability. In: *IEEE Micro* 20 (2000), Nr. 6, S. 12–25. – ISSN 0272–1732

[TSP08] TYYSTJÄRVI, Joonas ; SÄNTII, Tero ; PLOSILA, Juha: Instruction set enhancements for high-performance multicore execution on the REALJava plat-

form. In: *Proc. 26th Norchip Conference*, IEEE Press, 2008. – ISBN 978–1–4244–2492–4, S. 190–193

[TSP10] TYYSTJÄRVI, Joonas ; SÄNTII, Tero ; PLOSILA, Juha: Parallel performance evaluation of a multicore Java co-processor system. In: *Poster Session of DATE'10 Workshop: "Designing for Embedded Parallel Computing Platforms: Architectures, Design Tools, and Applications"*, 2010

[Uhr09] UHRIG, Sascha: Evaluation of different multithreaded and multicore processor configurations for SoPC. In: BERTELS, Koen (Hrsg.) ; DIMOPOULOS, Nikitas J. (Hrsg.) ; SILVANO, Cristina (Hrsg.) ; WONG, Stephan (Hrsg.): *Proc. 9th Int'l Workshop Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS'09)*, Springer-Verlag, 2009 (LNCS 5657). – ISBN 978–3–642–03137–3, S. 68–77

[UU09] UHRIG, Sascha ; UNGERER, Theo: A garbage collection technique for embedded multithreaded multicore processors. In: BEREKOVIC, Mladen (Hrsg.) ; MÜLLER-SCHLOER, Christian (Hrsg.) ; HOCHBERGER, Christian (Hrsg.) ; WONG, Stephan (Hrsg.): *Proc. 22nd Int'l Conf. Architecture of Computing Systems (ARCS'09)*, Springer-Verlag, 2009 (LNCS 5455). – ISBN 978–3–642–00453–7, S. 207–218

[UW07] UHRIG, Sascha ; WIESE, Jörg: Jamuth: an IP processor core for embedded Java real-time systems. In: [Bol07], S. 230–237

[YCM+04] YU, Chun-Pong ; CHOY, Chiu-Sing ; MIN, Hao ; CHAN, Cheong-Fat ; PUN, Kong-Pang: A low power asynchronous Java processor for contactless smart card. In: *Proc. 2004 Conf. Asia South Pacific Design Automation (ASP-DAC'04)*, IEEE Press, 2004. – ISBN 0–7803–8175–0, S. 553–554

[ZPRS07a] ZABEL, Martin ; PREUSSER, Thomas B. ; REICHEL, Peter ; SPALLEK, Rainer G.: Secure, real-time and multi-threaded general-purpose embedded Java microarchitecture. In: *Proc. 10th Euromicro Conf. Digital System Design Architectures, Methods and Tools (DSD'07)*, IEEE Computer Society Press, 2007. – ISBN 978–0–7695–2978–3, S. 59–62

[ZPRS07b] ZABEL, Martin ; PREUSSER, Thomas B. ; REICHEL, Peter ; SPALLEK, Rainer G.: SHAP — Secure Hardware Agent Platform. In: *Dresdner Arbeitstagung Schaltungs- und Systementwurf (DASS'07)*. Dresden : TUDpress, Verlag der Wissenschaften GmbH, 2007. – ISBN 978–3–940046–28–4, S. 119–126