Methoden zur applikationsspezifischen Effizienzsteigerung adaptiver Prozessorplattformen

Zur Erlangung des akademischen Grades eines

DOKTOR-INGENIEURS (Dr.-Ing.)

von der Fakultät für
Elektrotechnik und Informationstechnik
am Karlsruher Institut für Technologie (KIT)
genehmigte

DISSERTATION

von

Dipl.-Ing. Carsten Tradowsky

geboren in Ulm

Tag der mündlichen Prüfung:
20. Dezember 2016

Hauptreferent: Prof. Dr.-Ing. Dr. h. c. Jürgen Becker
Korreferent: Prof. Dr.-Ing. Jörg Henkel

Methoden zur applikationsspezifischen Effizienzsteigerung adaptiver Prozessorplattformen

1. Auflage: Dezember 2016

Herausgeber:
Herbie Martin Music
herbiemartinmusic.de

Für meine Familie.

Abstract

*General-Purpose Processor*s (GPPs) are optimized for the average use case, which leads to an unefficient usage of the existing resources. This work examines to which extent a GPP can be adapted to single applications to increase the efficiency of the respective application. In contrast to previous work *Field Programmable Gate Array*s (FPGAs) are only used for prototypical development, so that no FPGA-specific functionality is used for acceleration to allow the developed approaches to be transferred onto *Application Specific Integrated Circuit*s (ASICs).

Adaption can take place dynamically by processor or run-time system on the basis of the respective system parameters, e.g. usability, predictability or real-time capability, to achieve an efficiency increase. A first adaptive processor arises, which not only is cycle-accurately modeled but which has also be prototypically realized in hardware. The adaptive processor architecture is first evaluated in a single processors and then in multi- and many-core systems. It can be shown that this dynamic distribution of existing resources distinctly contributes to the increased efficiency.

In summary it can be shown that the adaptive processor system can be realized with relatively small development effort. The individual adaptive components lead to increased efficiency as the minimal use of additional resources is distinctly exceeded by the performance increase. The system could be scaled from a single core via a multi-core up to a many-core system. Thus the first adaptive processor has been realized that results in increased efficiency without exploiting specific hardware resources such as FPGA resources.

Zusammenfassung

General-Purpose Prozessoren (GPPs) sind für den durchschnittlichen Anwendungsfall optimiert, wodurch vorhandene Ressourcen nicht effizient genutzt werden. In der vorliegenden Arbeit wird untersucht, in wie weit es möglich ist, einen *General-Purpose* Prozessor an einzelne Anwendungen anzupassen und so die Effizienz zu steigern. Im Gegensatz zu vorangegangenen Arbeiten sollen die *Field Programmable Gate Array*s (FPGAs) lediglich zur prototypischen Entwicklung eingesetzt werden, so dass in dieser Arbeit keine FPGA-spezifischen Funktionalitäten zur Beschleunigung verwendet werden, damit die entwickelten Ansätze auch auf *Application Specific Integrated Circuit*s (ASICs) übertragbar sind.

Die Adaption kann zur Laufzeit durch das Prozessor- oder Laufzeitsystem anhand der jeweiligen Systemparameter, wie beispielsweise der Nutzbarkeit, Vorhersagbarkeit oder Echtzeitfähigkeit, erfolgen, um eine Effizienzsteigerung zu erzielen. Es entsteht eine adaptive Prozessorplattform, die nicht nur zyklenakkurat modelliert, sondern auch prototypisch umgesetzt wurde. Die adaptive Prozessorplattform wird zunächst im einzelnen Prozessor und anschließend im Mehr- und Vielkernarchitekturen evaluiert. Es kann gezeigt werden, dass die dynamische Verteilung der vorhandenen Ressourcen deutlich zur Effizienzsteigerung beiträgt.

Zusammenfassend kann gezeigt werden, dass sich ausgehend von der adaptiven Plattform mit relativ geringem Entwicklungsaufwand Prozessorarchitekturen realisieren lassen. Die einzelnen adaptiven Parameter erzeugen eine Effizienzsteigerung, da der Einsatz zusätzlicher Ressourcen von der daraus resultierenden Performanzsteigerung deutlich übertroffen wird. Hierdurch wird der neuartige Prozessor realisiert, der durch das Zusammenspiel von Hardware und Software eine Effizienzsteigerung erzielt, indem Applikation, Laufzeitumgebung, Compiler und auch die Hardware selbst den adaptiven Prozessor unterbrechungsfrei parametrieren.

Vorwort

Die vorliegende Arbeit entstand während meiner Tätigkeit am Institut für Technik der Informationsverarbeitung (ITIV) der Fakultät Elektrotechnik und Informationstechnik (ETIT) am Karlsruher Institut für Technologie (KIT). In dieser Zeit durfte ich mich vielseitigen Themen in der Forschung aber auch der Lehre mit der Betreuung von Vorlesungen, Übungen, Praktika, aber auch studentischen Abschlussarbeiten widmen. Auf diesem Wege möchte ich mich bei allen Kolleginnen und Kollegen des ITIV für die Unterstützung und die spannende, vielseitige Zusammenarbeit bedanken.

Ich möchte meinem Doktorvater Prof. Jürgen Becker für das entgegengebrachte Vertrauen und die Betreuung dieser Arbeit danken. Außerdem möchte ich mich bei Prof. Jörg Henkel für die Zusammenarbeit im *Invasive Computing* (InvasIC) Projekt und die Übernahme des Koreferats bedanken. Des Weiteren danke ich Prof. Ivan Peric, Prof. Michael Heizmann und Prof. Thomas Zwick in ihrer Rolle als Prüfer.

Mein Dank gilt dem gesamten InvasIC Team. Neben den B1 Kollegen möchte ich mich vor allem für die hervorragende teilprojektübergreifende Zusammenarbeit und die daraus entstandenen gemeinsamen Veröffentlichungen bedanken. Eine Bereicherung meines Arbeitsalltags war die Betreuung studentischer Arbeiten. Außerdem möchte ich mich bei den studentischen Hilfskräften bedanken.

Beim besonderer Dank gilt meiner Familie und meinen Freunden.

Karlsruhe, im Dezember 2016
Carsten Tradowsky

Inhaltsverzeichnis

1. Einleitung

1.1. Umfeld und Ableitung der Problemstellung

General-Purpose Prozessoren (GPPs) sind für den durchschnittlichen Anwendungsfall optimiert. Hierdurch sind die vorhandenen Ressourcen der Prozessor Architektur nicht optimal an die Anforderungen der jeweiligen Anwendung angepasst. Hierdurch werden einzelne Anwendungen weniger effizient ausgeführt als andere, die sich eher wie die durchschnittliche Anwendung verhalten.

In der vorliegenden Arbeit wird untersucht, in wie weit es möglich ist, einen vorhandenen GPP an einzelne Anwendungen anzupassen und so die Effizienz bei der Ausführung der jeweiligen Anwendungen zu steigern. Hierbei werden verschiedene Faktoren in die Untersuchung einbezogen:

1. In wie weit ist es möglich eine adaptive Prozessorarchitektur zu entwickeln, die auf einem GPP Instruktionssatz basiert?
2. In wie weit lässt sich eine vorhandene Instruktionssatz Architektur (ISA) erweitern, um die adaptiven Funktionalitäten durch Hardware sowie auch Software zur Laufzeit steuerbar zu machen?
3. In wie weit können kommerzielle Entwicklungstools eingesetzt werden, um den Prozessor sowie auch die entsprechenden Tools, wie beispielsweise Compiler und Assembler zu erstellen?
4. In wie weit lassen sich vorhandene Mikroarchitektur Komponenten der jeweiligen Prozessorarchitektur anpassen und um die adaptiven Funktionalitäten erweitern?
5. In wie weit lassen sich diese Methoden auf aktuelle Mehrkern und zukünftige Vielkern Architekturen anwenden?

Für die Entwicklung des Prozessorsystems kommen *Field Programmable Gate Array*s (FPGAs) zum Einsatz. Im Gegensatz zu vorangegangenen Arbeiten sollen die FPGAs jedoch nur zur Entwicklung eingesetzt werden, so dass keine FPGA-spezifischen Funktionalitäten zur Beschleunigung verwendet werden, damit die entwickelten Ansätze auch auf *Application Specific Integrated Circuit*s (ASICs) anwendbar sind.

Die Adaption soll zur Laufzeit anhand der jeweiligen System Parameter erfolgen. Dies sind insbesondere Funktionale Eigenschaften, wie beispielsweise Nutzbarkeit, Vorhersagbarkeit, Echtzeitfähigkeit, Leistungs- und Energieverbrauch. Anhand dieser Ziele kann das Laufzeitsystem mit dem Prozessorsystem interagieren und dieses Anpassen, um den Prozessor an die jeweilige Anwendung anzupassen und somit für eine Steigerung der Performanz bezogen auf den Flächenverbrauch zu erzielen.

Es können erste Untersuchungen der adaptiven Prozessorplattform durchgeführt werden. Es werden adaptive Erweiterungen auf Basis der bekannten Mikroarchitekturkomponenten Sprungvorhersage, Pipeline (In-Order vs. Out-of-Order) und Cache vorgenommen. Diese Erweiterungen lassen sich aufgrund der hohen Abstraktionsebene vergleichsweise schnell entwickeln und somit zügig in Bezug auf die Anwendbarkeit im gesamten Prozessorsystem evaluieren. Für die weitere Arbeit zeigt sich, dass vor allem die adaptive Sprungvorhersage und der adaptive Cache mit relativ geringem Overhead recht großes Potential zur Effizienzsteigerung besitzen.

Hierzu wird das *Very High Speed Integrated Circuit Hardware Description Language* (VHDL) Modell des Leon3 Prozessors jeweils um die adaptiven Funktionalitäten erweitert. Hierdurch entsteht die adaptive Prozessorplattform, die nicht nur zyklen-akkurat modelliert, sondern auch in Hardware auf einem Prototypen umgesetzt wurde.

1.2. Ziele der Arbeit und eigener Beitrag

Im Kontext dieser Arbeit sollen die Vorteile von *General Purpose* und applikationspezifischen Prozessoren kombiniert werden. Im Bereich der eingebetteten Prozessoren gilt meist ein sehr stark begrenztes Leistungs- und Energiebudget. Daher ist es notwendig die maximale Recheneffizienz bei

begrenztem Budget zu erreichen. Einzelne Faktoren können beispielsweise die Folgenden sein:

- Bekanntheit & Beliebtheit
- Flexibilität
- Programmierbarkeit
- Leistungs- & Energiebudget
- Adaptivität nicht vorhanden
- Entwicklungsaufwand

Daraus ergeben sich folgende Anforderungen und Zielsetzungen für die adaptive Prozessorarchitektur:

- Adaptive Fähigkeiten sollen zur Designzeit zur Verfügung gestellt und zur Laufzeit programmierbar gestaltet werden.
- Vorteile der Laufzeitadaptivität sollen in General-Purpose Prozessorarchitektur gezeigt werden.
- Zusammenhang zwischen Adaptivität und Effizienz soll identifiziert werden.
  - Alle adaptiven Fähigkeiten sollen zuverlässig und vorhersagbar konzeptioniert und umgesetzt werden. Diese sollen somit sind in einer vorhersagbaren Architektur einsetzbar sein.
- Der Fokus soll zunächst auf einem Prozessorkern, dann auf der Erweiterung der Konzepte auf Mehr- und Vielkernprozessorsysteme liegen.
- Es soll eine Verkürzung der Entwicklungszeit erreicht werden.

Es resultiert eine selbstadaptive Prozessorarchitektur, die sich selbst an die Bedürfnisse der Applikation anpasst. Darüber hinaus kann das Laufzeitsystem oder der *Compiler* die Prozessorarchitektur dynamisch die Anforderungen, wie beispielsweise minimale Laufzeit, Flächen- oder Energiebedarf, umsetzen.

Anhand der adaptiven Fähigkeiten, die in dieser Arbeit vorgestellt werden, soll es möglich werden die Effizienz der Prozessorarchitektur sowie auch des gesamten Vielkern- oder Mehrkernprozessorsystems zu steigern. Daher werden den adaptiven Fähigkeiten die oben angeführten Eigenschaften

zugewiesen, die jeweils wie in Tabelle 1.1 dargestellt klassifiziert werden sollen.

Effizienz	Einkern	Mehrkern	Vielkern
a-Core	✓ Performanz ✗ Fläche ✗ Leistung	✓ Performanz ✓ Fläche ✗ Leistung	✓ Performanz ✓ Fläche ✓ Leistung
Gesamt	✗ Effizienz	✓ Effizienz	✓ Effizienz

Tabelle 1.1.: Klassifizierung der adaptiven Fähigkeiten mit Einordnung der bezüglich der jeweils ausschlaggebenden Anforderung.

Im Gegensatz zur reinen Softwareimplementierung in verhaltensorientierten Simulatoren bei bisherigen Arbeiten, erfolgt die Untersuchung anhand einer Hardwareimplementierung auf Basis der LEON3 Prozessorarchitektur. Die Evaluation wird anhand von Benchmarks wie beispielsweise *MiBench* oder *Coremark* sowie auch Applikationsszenarien durchgeführt.

1.3. Aufbau der Arbeit

Die weitere Arbeit ist wie folgt gegliedert: Zunächst werden in Kapitel 2 die Grundlagen für die Arbeit vorgestellt. Diese sind in Mikroprozessoren (Abschnitt 2.1), Reduced Instruction Set Computer (RISC) Prozessor Pipeline (Abschnitt 2.2), Superskalare Prozessoren (Abschnitt 2.3), Sprungvorhersage (Abschnitt 2.4), Cache (Abschnitt 2.5), Scalable Processor ARChitecture (SPARC) & LEON3 (Abschnitt 2.6), *Language for Instruction-Set Architectures* (LISA) & Werkzeugkette (Abschnitt 2.7), Bildverarbeitungsalgorithmen (Abschnitt 2.8) und Nummerische Algorithmen (Abschnitt 2.9) unterteilt. Anschließend wird der Stand der Technik (Kapitel 3) aufgearbeitet und in drei Abschnitten vorgestellt. Zunächst werden Mikroarchitektur Konzepte (Abschnitt 3.1), wie beispielsweise Intel und ARM und Mikroarchitektur Konzepte, wie Sprungvorhersage oder Cache aufgearbeitet. Anschließend folgt ein Überblick über Applikationsspezifische, adaptive

und rekonfigurierbare Prozessoren (Abschnitt 3.2). Des weiteren wird die Mehr- und Vielkern Prozessorarchitekturen (Abschnitt 3.3) im Kontext des Deutsche Forschungsgemeinschaft (DFG) Sonderforschungsbereich (SFB) *Invasive Computing* vorgestellt. Abschließend wird werden aktuelle Architekturbeschreibungssprachen (ADL) (Abschnitt 3.4) präsentiert.

In Kapitel 4 wird *a*-Core: Die adaptive Prozessorarchitektur konzipiert. Dieses Kapitel unterteilt sich in das Design des adaptiven Prozessors (Abschnitt 4.1), die Vorteile des adaptiven Prozessors (Abschnitt 4.2) und die abschließende Realisierung des adaptiven Prozessors (Abschnitt 4.3). Des Weiteren wird kurz die Adaptive Pipeline (Abschnitt 4.4) und die Adaptive Superskalarität (Abschnitt 4.5) vorgestellt.

Als erste Eigenschaft der adaptiven Prozessorarchitektur wird in Kapitel 5 die Selbstadaptivität anhand der adaptiven Sprungvorhersage untersucht. Dieses Kapitel unterteilt sich in das Design der adaptiven Sprungvorhersage (Abschnitt 5.1), die Realisierung der adaptiven Sprungvorhersage (Abschnitt 5.2) und die abschließende Evaluation der adaptiven Sprungvorhersage (Abschnitt 5.3).

Als zweite Eigenschaft wird in Kapitel 6 die Laufzeitadaptivität anhand des adaptiven Cache vorgestellt. Hierzu wird zunächst die Cache- und bandbreitengewahre Programmierung (Abschnitt 6.1) vorgestellt. Anschließend erfolgt die Simulative algorithmische Optimierung der Cacheparametrierung (Abschnitt 6.2). Das Design erfolgt in Abschnitt 6.3 Adaptive Cache Architektur. Anschließend wird es zur Dynamische Cache Reallokation in Mehrkernarchitekturen (Abschnitt 6.4) weiter entwickelt. Abschließend erfolgt die Dynamische Cache Strategieanpassung in Vielkernarchitekturen (Abschnitt 6.5)

Abschließend wird die Methodik dieser Arbeit in Kapitel 7 eingeordnet und mit dem Stand der Technik verglichen. Letztendlich folgt die Schlussfolgerung in Kapitel 8 mit einem Ausblick auf weiterführende Arbeiten.

Zusätzlich erfolgt in Abschnitt A.1 die Vorstellung des Integrierter Entwicklungsfluss für die adaptive Prozessorarchitektur. Hier wird zunächst der *LISA model-based ideal Cortex-M1* (LImbiC) (Unterabschnitt A.1.1) vorgestellt, worauf *a*-Sim: Die adaptiven Simulationen (Unterabschnitt A.1.2) folgen. Abschließend erfolgt eine Evaluation des applikationsspezifischen Prozessormodells (Unterabschnitt A.1.3)

2. Grundlagen

In diesem Kapitel werden die Grundlagen für die vorliegende Arbeit vorgestellt. Zu Beginn wird in das Thema der Mikroprozessoren eingeführt. Anschließend werden die *Reduced Instruction Set Computer* (RISC) Prozessor Pipeline, die *Scalable Processor ARChitecture* (SPARC) und der LEON3 Prozessor vorgestellt.

2.1. Mikroprozessoren

Ein Mikroprozessor, wie er in dieser Arbeit verwendet wird, soll als Zentraleinheit - *Central Processing Unit* (CPU) - einer Datenverarbeitungseinheit verstanden werden. Diese CPU kann Kern eines Mikrocontrollers sein, der mit Peripherie on-Chip realisiert werden kann.

Ein Mikroprozessorsystem ist ein technisches System, das einen Mikroprozessor enthält und von diesem gesteuert wird. Ein Mikrorechner besteht aus mehreren Mikroprozessoren, Speicher, E/A-Einheit und Verbindungssystemen.

2.1.1. Von-Neumann-Prinzip und Aufbau eines Mikroprozessors

Das von-Neumann-Prinzip definiert Grundsätze für die Architektur und die Mikroarchitektur eines Rechners [23].

- Der (Mikro-)Prozessor CPU übernimmt die Ablaufsteuerung und die Ausführung der Befehle. Abbildung 2.1 zeigt die Struktur eines

Mikroprozessors. Er besteht aus einem Steuerwerk (SW) und einem Rechenwerk (RW). Das Steuerwerk arbeitet die Maschinenbefehle unter Berücksichtigung der Statusinformation ab und steuert so die anderen Komponenten. Der Befehlszähler (*Program Counter* (PC)) enthält die Adresse des nächsten Maschinenbefehls, falls kein Sprungbefehl ausgeführt wird. Das Rechenwerk führt alle arithmetisch-logischen Operationen aus, sofern es keine Steuer- oder Eingabe- und Ausgabebefehle sind.

- Das Eingabe- und Ausgabesystem (E/A) stellt für den Prozessor die Schnittstelle nach außen dar, über die auch der Datenaustausch mit externen Komponenten stattfindet.
- Der Hauptspeicher dient zur Ablage von Daten und Programmen. Er ist aus Speicherzellen fester Wortlänge aufgebaut, die einzeln adressiert werden können.
- Die Verbindungseinrichtung dient zur Übertragung der Daten zwischen dem Prozessor und weiteren Speichern sowie der Eingabe- und Ausgabeeinheit.

Code und Daten stehen im selben Speicher. Eine Speicherstelle kann ein Datenwort oder einen Maschinenbefehl (Opcode) enthalten. Die Interpretation hängt nur vom Opcode ab.

Die Ausführung der Befehle erfolgt sequentiell, wobei die Abarbeitung eines Befehls abgeschlossen sein muss, bevor mit dem nächsten Befehl begonnen werden kann. Ein von-Neumann-Rechner arbeitet in zwei Phasen [23]:

- Befehlsbereitstellung - Dekodierphase: Der Befehlszähler adressiert eine Speicherstelle. Das dort gespeicherte Datenwort wird geholt und als Befehl interpretiert.
- Ausführungsphase: Die Inhalte der Speicherzellen werden nun entsprechend des im Datenwort repräsentierten Opcodes abgearbeitet.

Der von-Neumann-Rechner lässt sich mit minimalem Hardwareaufwand realisieren und ist somit optimal für kostenminimale Lösungen geeignet. Die Verbindungseinrichtung erweist sich jedoch als Engpass, da die Daten und Maschinenbefehle transportiert werden müssen. Daher spricht man vom von-Neumann-Flaschenhals. Eine Variante hierzu stellt die Harvard-

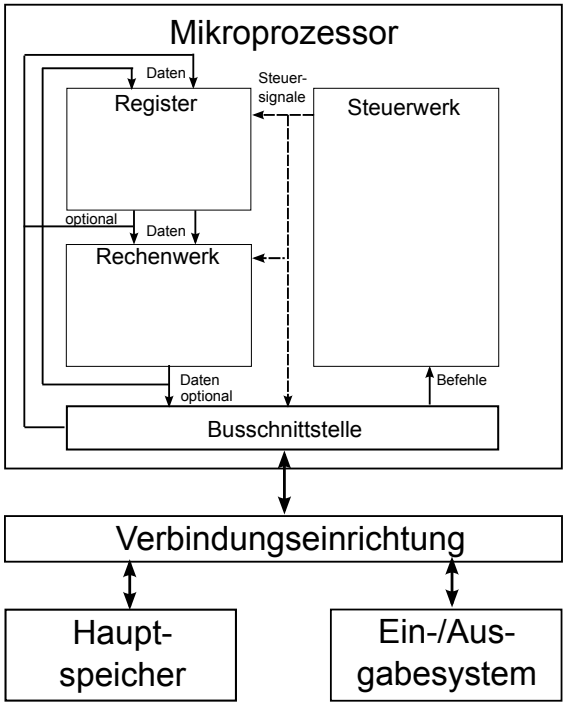


Abbildung 2.1.: Struktur eines von-Neumann-Rechners und eines einfachen Mikroprozessors [23].

Architektur dar. Hierbei gibt es getrennte Speicher und Caches für Daten- und Maschinencode [77].

2.1.2. Prozessorarchitektur und Mikroarchitektur

Eine Prozessorarchitektur definiert die Grenze zwischen Hardware und Software. Diese Befehlssatzarchitektur - Instruktionssatz Architektur (ISA) - ist für den Compiler sichtbar. Sie ist das Programmiermodell des Prozessors. Hierzu gehören neben dem Befehlssatz auch das Befehlsformat, die Adressierungsart, das Speichermodell, bestehend aus Registern, und der

Adressraumaufbau. Die Prozessorarchitektur umfasst jedoch keine Details der Hardware oder der technischen Ausführung des Prozessors.

Eine Mikrorechnerarchitektur bezeichnet die Implementierung der Prozessorarchitektur z.B. als Mikroprozessor. Hierzu gehören die Hardware-Struktur, der Entwurf der Daten- und Kontrollpfade, die Art und Stufen der Pipeline (siehe Abschnitt 2.2), der Grad der Verwendung der Superskalartechnik, die Art und Anzahl der internen Ausführungseinheiten, sowie Einsatz und Organisation von Primär-Cache-Speichern. Optimierende Compiler benötigen auch Kenntnis von diesen Eigenschaften, um einen effizienten Code generieren zu können.

Die Prozessorarchitektur macht Programme von Mikroprozessoren unabhängig. Solange Mikroprozessoren denselben Architekturspezifikationen folgen sind sie binär kompatibel zueinander. Je nach Implementierungstechnik zeigen sich verschiedene Verarbeitungsgeschwindigkeiten. Vorteile der Trennung von Prozessorarchitektur und Mikroarchitektur sind, dass sich z.B. bei Änderung der Technologie Implementierungstechniken ändern. Aus Sicht der Architektur sind nach außen keine Änderungen der Funktion sichtbar. Prozessoren einer Prozessorfamilie haben die gleiche Basisarchitektur. Die neueren, komplexeren erweitern die Architekturspezifikation.

2.1.3. Befehlssatz

Der Befehlssatz (Instruktionssatz Architektur (ISA)) definiert, welche Operationen auf Daten ausgeführt werden können. Er legt die Grundoperationen des Prozessors fest. Folgende Befehlsarten können unterschieden werden [23]:

- Datenbewegungsbefehle übertragen Daten von einer Speicherstelle zur Nächsten. Falls es einen separaten E/A-Adressraum gibt, gehören auch dessen Befehle dazu. Auch Kellerspeicherbefehle (*Push/Pop*) gehören dazu.
- Arithmetisch-logische Befehle (*integer arithmetic and logical*) können Ein-, Zwei- oder Dreioperandenbefehle sein. Prozessoren nutzen verschiedene Befehle für verschiedene Datenformate ihrer Operanden. Der Opcode legt fest, ob Befehle, wie Addition oder Subtraktion, mit oder ohne Vorzeichen oder Carry, ausgeführt werden.

- Schiebe- und Rotationsbefehle schieben Bits nach links oder rechts. Beim Schieben gehen herausgeschobene Bits verloren und werden durch 0 oder 1 ersetzt. Beim Rotieren werden die Bits mit oder ohne Berücksichtigung des Carrys wieder eingefügt.
- Programmsteuerbefehle sind Befehle, die den Programmablauf beeinflussen, wie z.B. Sprung- oder Verzweigungsbefehle.
- Systemsteuerbefehle erlauben es, direkt Einfluss auf Prozessor- oder Systemkomponenten zu nehmen.
- Multimedia-, Gleitkomma und Synchronisationsbefehle sind weitere Befehlsarten.

2.1.4. Befehlsformat

Das Befehlsformat bestimmt die Kodierung der Befehle. Die Befehlscodierung beginnt mit dem Opcode, der den Befehl festlegt. Je nach Opcode werden weitere Felder benötigt, die weiteren Opcode, Adressfelder, Operanden oder Adressen spezifizieren. Je nach Art der Befehle können vier Klassen von Befehlssätzen unterschieden werden [23]:

- Das Dreiadressformat besteht aus dem Opcode, zwei Quelladressen und einer Zieladresse.
- Das Zweiadressformat besteht aus dem Opcode, zwei Quelladressen, wobei eine Quell- der Zieladresse entspricht.
- Das Einadressformat besteht aus dem Opcode und einer Quelladresse, die wiederum Zieladresse ist.
- Das Nulladressformat besteht nur aus dem Opcode.

2.1.5. Adressierungsarten

Die Adressierungsarten legen fest, wie auf Daten zugegriffen wird und wie eine Operanden- oder eine Sprungzieladresse im Prozessor berechnet werden kann. Eine Adressierungsart kann eine im Befehlswort stehende Konstante, ein Register oder einen Speicherplatz im Hauptspeicher spezifizieren.

Es werden explizite und implizite Adressierungsarten unterschieden. Im folgenden werden einige Datenadressierungsarten aufgelistet [23]:

- Registeradressierung: Der Operand steht direkt im Register
- Unmittelbare Adressierung: Der Operand steht als Konstante im Befehlswort
- Direkte oder absolute Adressierung: Die Speicheradresse des Operanden steht im Befehlswort.
- Registerindirekte Adressierung: Die Registeradresse, in der die Speicheradresse des Operanden steht, steht im Befehlswort.
- Befehlszählerindirekter Modus: Der Operand wird im Register adressiert und setzt den Befehlszähler.

2.2. Reduced Instruction Set Computer (RISC) Prozessor Pipeline

Ein interessanter Fokus dieser Arbeit ist die Optimierung der Prozessor Pipeline. In diesem Kapitel wird ein Überblick über die Grundprinzipien des Pipelinings und die damit verbundenen Begrenzungen und Herausforderungen gegeben.

2.2.1. Funktionsweise

Das Grundprinzip des Pipelinings lässt sich am einfachsten anhand von Parallelen aus dem alltäglichen Leben verstehen. Ein Student kann es nicht mehr erwarten seinen Cocktail zu bekommen. Aufgrund des Mangels an StudentInnen ist das Barteam unterbesetzt, wodurch es zu langen Schlangen kommt. Es muss jeweils die Bestellung aufgenommen, abgearbeitet und kassiert werden. Erst anschließend kann der Student seinen Cocktail trinken. An diesem Beispiel ist einfach zu erkennen, dass es den Ablauf optimieren würde, wenn die Abarbeitung, wie in Abbildung 2.2 zu sehen, in mehreren Stufen geschehen würde. Die Bar kann in die Stufen Bestellung,

Getränk mixen und bezahlen unterteilt werden. Jede dieser Aufgaben ist einer StudentIn fest zugeordnet. So wird schnell ersichtlich, dass sobald der erste Student mit der Bestellung fertig ist und auf das Mixen seines Getränkes wartet, bereits der nächste Student seine Bestellung abgeben kann. Auf diese Art und Weise wird, nach Beendigung eines Abarbeitungsschrittes, die Abarbeitung des nächsten Schrittes gestartet. Für den einzelnen durstigen Studenten wird die Abarbeitung nicht schneller, da alle Abarbeitungsstufen durchlaufen werden müssen. Bei vielen Studenten, wie zu erwarten sein wird, wird die Abarbeitung insgesamt schneller. Also mehr beendete Abarbeitungen pro Zeiteinheit, da die Abarbeitungsstufen immer parallel arbeiten und einzelne nicht leer laufen. Hierdurch wird der Durchsatz des Gesamtsystems erhöht. Wird die Anzahl der Aufgaben sehr groß, wird die Zeitersparnis gleich der Anzahl der Pipelinestufen, also nur ein Drittel der Ausführungszeit im studentischen Beispiel.

Auf die gleiche Art und Weise funktioniert auch die Prozessorpipeline. Eine einfache Pipeline lässt sich, wie Abbildung 2.2 zeigt, in vier Stufen unterteilen [122]:

- ***Instruction Fetch* (IF):** Die Instruktion wird aus dem Instruktionsspeicher geholt.
- ***Decode* (DE):** Die Instruktion wird dekodiert und mit den implementierten Befehlen verglichen. Eventuell benötigte Operanden werden aus dem Register geholt.
- ***Execute* (EX):** Die Operation wird ausgeführt oder eine Adresse berechnet
- ***Write Back* (WB):** Der Operand wird in das *Register-File* (RF) oder einen weiter entfernten Speicher geschrieben.

Dieser einfache Aufbau der Pipeline soll zur weiteren Erläuterung für diesen Abschnitt verwendet werden. In Abschnitt 2.6 wird die in dieser Arbeit verwendete SPARC V8 Architektur und deren siebenstufige Pipeline detaillierter eingeführt. Wenn alle Pipelinestufen ausbalanciert sind, also die Abarbeitung der Aufgaben in den einzelnen Stufen gleich lange dauert, kann die obige Diskussion der Ausführungszeit in Gleichung 2.1 gefasst werden [124]:

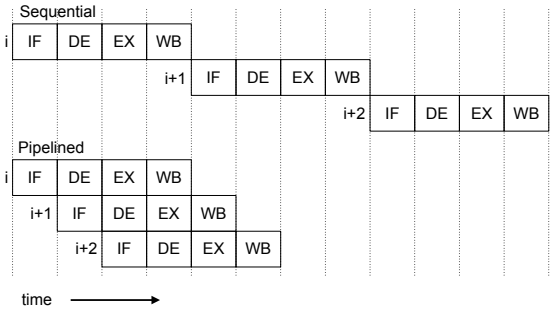


Abbildung 2.2.: Ausführung von Instruktionen sequentiell und mit Pipelining über der Zeit [124].

$$\text{Zeit zwischen Instruktionen}_{pipeline} = \frac{\text{Zeit zwischen Instruktionen}_{nopipeline}}{\text{Anzahl Pipeline-Stufen}} \tag{2.1}$$

In der Realität kann es jedoch der Fall sein, dass die einzelnen Stufen nicht wirklich balanciert sind, da es in jeder der Stufen zu zusätzlichem *Overhead* kommen kann. Daher wird die Verbesserung in einer Implementierung zwar schneller als die Lösung ohne Pipeline sein, aber nicht an den n-fachen Durchsatz heranreichen. Die Pipeline erhöht die Performanz, in dem der Instruktionsdurchsatz erhöht wird. Dies ist eine wichtige Metrik, da in einem komplexen realen Programm Milliarden von Instruktionen nacheinander ausgeführt werden.

Pipelining ist somit eine Technik, um eindimensionalen Parallelismus, den Instruktionslevel Parallelismus (*Instruction Level Parallelism* (ILP)), in eine sequentielle Abarbeitung einzuführen [77]. Um den ILP zu erhöhen kann die Zahl der Ausführungsstufen einer Pipeline, also die Pipelinetiefe erhöht werden. Diese Technik ist im Gegensatz zu anderer Parallelität, z.B. mehreren parallelen Ausführungseinheiten, für den Programmierer unsichtbar und wird ausschließlich durch die Hardware umgesetzt. Pipelining erhöht die Zahl der gleichzeitig ausgeführten Instruktionen und auch die Rate mit der sie gestartet und beendet werden. Die Latenz der Pipeline, das komplette Ausführen einer einzelnen Instruktion, bleibt gleich. Gleichzeitig

wird aber der Durchsatz und damit die Performanz der Pipeline erhöht. Weitere speziellere Pipelinefunktionalitäten, durch welche die räumliche Parallelität eingeführt werden kann sind *Multiple Issue*, *Very Long Instruction Word* (VLIW), *Superskalar*, *Out-of-Order* (OoO) Ausführung, Spekulation, *Reorder Buffer* oder auch *Register Renaming* [77].

2.2.2. Pipeline Konflikte

Es gibt Momente während der Abarbeitung von Programmcodes, in denen das eigentliche Ziel, eine Instruktion pro Taktzyklus aus dem Speicher zu holen und abzuarbeiten, nicht gegeben ist. Tritt dieser Fall auf, wird von einem *Hazard* gesprochen. Es werden drei Arten von *Hazards* unterschieden, auf die im Folgenden einzeln näher eingegangen wird [124].

2.2.2.1. Struktureller Konflikt

Bei einem strukturellen *Hazard* kann die Hardware die sich in der Pipeline befindenden Instruktionen nicht im gleichen Taktzyklus abarbeiten, da die Hardware diese Abarbeitung nicht unterstützt [124]. Ein Beispiel für einen strukturellen *Hazard* kann ein gleichzeitiger Zugriff auf einen Speicher sein. In jedem Taktzyklus wird eine Instruktion aus dem Speicher geholt. Erfolgt im gleichen Taktzyklus ein Zugriff auf Daten in demselben Speicher, tritt dieser strukturelle *Hazard* auf. Dieser kann nur durch einen zweiten Speicher, die Trennung von Instruktions- und Datenspeicher, umgangen werden, ohne dass die Pipeline angehalten werden muss.

2.2.2.2. Daten Konflikt

Ein Daten *Hazard* tritt auf, wenn die Pipeline angehalten werden muss, weil eine Stufe auf die Beendigung der Abarbeitung einer Operation in einer anderen Stufe warten muss [124]. Dies folgt aus der Datenabhängigkeit zweier Instruktionen, wobei sich die bereits abgearbeitete Instruktion noch in der Pipeline befindet und die Operanden, auf die die folgende Instruktion zugreifen will, somit noch nicht wieder im *Register-File* oder

entfernten Speicher zu finden sind. Wenn an dieser Stelle nicht vom Designer eingegriffen werden würde, würde dieser *Hazard* sehr häufig zum eventuell mehrfachen Anhalten der Pipeline und damit zu hohen Kosten, in Form von leerlaufenden Taktzyklen, während der Ausführung von Programmen führen. Zur Lösung dieser *Hazards* könnte man sich zusätzlich auf den Compiler und dessen Instruktionsordnung verlassen. Dies ist bei komplexem Code jedoch sehr ineffizient, so dass sich ein manueller Eingriff im Design durchaus lohnt. Hierbei wird mit Hilfe zusätzlicher Hardware das Ergebnis aus den folgenden Pipelinestufen direkt an die vorigen wartenden Stufen weitergeleitet. Diese Form der Implementierung wird *Forwarding* genannt. Zusätzlich hierzu kann das *Bypassing* angewendet werden, wobei das Ergebnis den Pipelinestufen direkt am *Register-File* vorbei wieder zur Verfügung gestellt wird. Dies verhindert, auf Grund des synchronen halbzyklenhaften Aufbaus gegenwärtiger *Register-File*s - lesen im ersten, schreiben im zweiten Halbzyklus der Taktperiode -, ein zusätzliches Anhalten der Pipeline. Dieses Anhalten der Pipeline wird, wie in Abbildung 2.3 zu erkennen, im englischen als *stall* oder auch als *bubble* bezeichnet [122]. Die einzige nicht durch *Forwarding* oder *Bypassing* lösbare Gefahr für die Pipeline ist der sogenannte *load-use* Datenhazard. Hierbei wird ein Operand aus einem entfernten Speicher angefordert. Dieser kann jedoch aufgrund der Pipelinestruktur erst im nächsten Taktzyklus für die weitere Verarbeitung zur Verfügung stehen, was zum einmaligen Anhalten der Pipeline führt. Die Implementierung von *Forwarding* und *Bypassing* kann effizient mit Hilfe der bereits vorhandenen Pipelineregister erfolgen, so dass durch diese Techniken nur ein minimaler *Hardwareoverhead* im Steuerwerk entsteht.

2.2.2.3. Kontroll Konflikt

Der dritte Typ, der Kontrollhazard, ist dadurch charakterisiert, dass die anstehende Entscheidung auf der zukünftigen Ausführung anderer Instruktionen beruht [124]. Dieser Typ ist auch als Verzweigungshazard bekannt. Die einfachste Art diesen Hazard zu lösen wäre einfach die Pipeline anzuhalten, bis die Ausführung der relevanten Instruktionen beendet ist. Dies ist auf Grund der verschwendeten Taktzyklen die kostspieligste und langsamste Art der Lösung. Wenn in der Dekodierstufe eine Verzweigungsinstruktion erkannt wird, muss schon im nächsten Taktzyklus die

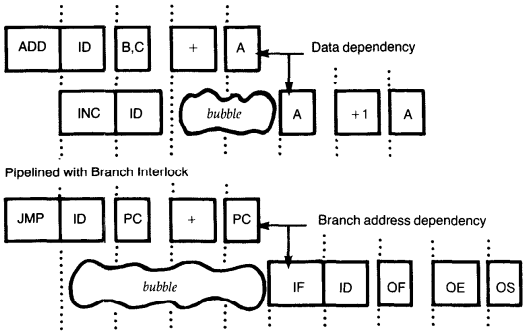


Abbildung 2.3.: Branch und Datenabhängigkeiten führen zu Stall bzw. Bubble in Pipeline [122].

Zielinstruktion aus dem Speicher geholt werden. Dies ist in den allermeisten Fällen nicht möglich, da es sich zumeist um bedingte Sprünge (*conditional branches*) handelt. Daher werden an dieser Stelle Sprungvorhersagen (*branch predictions*) verwendet. Dies bedeutet, dass die Abarbeitung auf Basis dieser Vorhersage fortgesetzt wird. Stellt sich die Vorhersage später als richtig heraus, läuft die Pipeline ohne Verlust weiter. Stellt sich die Vorhersage als falsch heraus, muss das zwischenzeitliche Ergebnis verworfen. Das heißt die falsch abgearbeiteten Stufen müssen aus der Pipeline gespült (*geflushed*) werden, und neu, von der Stelle der Vorhersage an, berechnet werden. Somit werden die Kosten für die gesamte Ausführung minimiert, steigen jedoch mit der Häufigkeit der Falschvorhersagen an. Hierbei werden statische Vorhersagen und dynamische Vorhersagen unterschieden. Diese komplexeren und damit teureren Implementierungen funktionieren mit Hilfe der Sprungausführungsgeschichte (*Branch Prediction Buffer*) optimaler. Die Genauigkeit solcher Vorhersagen kann mit aktuellen Vorhersagemechanismen bei weit über 90% liegen, was das Auftreten dieses *Hazards* deutlich minimiert und dadurch die Performanz der Ausführung optimiert [108], [149].

2.2.2.4. Exceptions & Interrupts

Eine der komplexesten in Prozessorpipelines auftretenden Kontrollstrukturen sind *Exceptions* und *Interrupts* während der Ausführung. Diese sind dazu vorhanden, um abnormale Zustände, wie beispielsweise *Overflows* des Prozessors, abzufangen. Die Gründe dafür können interner (*Overflow*, undefinierte Instruktion) oder externer (I/O *Interrupt*) Natur sein. Der Prozessor muss diese Dysfunktionen abfangen, den Zustand der Pipeline speichern. Damit soll der Prozessor nach dem Aufrufen einiger Subroutinen wieder in den normalen Abarbeitungszustand übergehen zu können, ohne dabei unnötige Kosten in Form von Pipeline *Stalls* oder physikalischen Auswirkungen auf den kritischen Pfad hervorzurufen. Wie bei allen *Hazards* erschweren längere Pipelines deren Lösung und erhöhen vor allem die Kosten von real auftretenden *Hazards*.

2.2.3. Von der Performanz zur Effizienz

In diesem Abschnitt soll einleitend erläutert werden, wie Performanz definiert ist und für diese Arbeit gemessen werden kann. Durch immer kompliziertere Optimierungstechniken, die in moderne Rechenkerne implementiert werden, wird es immer schwieriger aus Kenndaten, wie beispielsweise der Taktgeschwindigkeit (in Hertz (Hz)), auf die eigentliche Performanz der CPU zu schließen. Wenn jedoch eine Auswahl oder Klassifikation zwischen verschiedenen Rechnern erfolgen soll, ist es wichtig die Performanz zu definieren.

Am einfachsten wäre es die Performanz über die Taktgeschwindigkeit zu definieren. Hierbei ist jedoch zu beachten, dass es aus der jeweils angegebenen Taktgeschwindigkeit nicht direkt auf die Ausführungsgeschwindigkeit geschlossen werden kann, da nicht zwangsläufig in jedem Taktzyklus eine Instruktion abgearbeitet wird. Somit ist es sinnvoller die Maßeinheit über die Zeit zu definieren, in der ein Computer eine Aufgabe vollständig abarbeitet. Dadurch kann die *response-time*, auch Ausführungszeit genannt, als Zeit vom Start bis zur Beendigung der Aufgabe definiert werden [124]. In einem größeren Kontext wird es interessant den *Throughput* oder die *Bandwidth*, also die Anzahl der abgearbeiteten Aufgaben in einer bestimmten Zeit zu erhöhen. Daher kann mit *Benchmarks* [65, 165, 166] zwischen

Ausführungszeit für Desktop Computer und Durchsatz für Server unterschieden werden. So ist es sinnvoll, dass die Ausführungszeit zur Steigerung der Performanz möglichst klein werden muss. Dies führt nach [124] auf Gleichung 2.2.

$$\text{Performanz} = \frac{1}{\text{Ausführungszeit}} \tag{2.2}$$

Werden zwei Prozessoren A und B verglichen, wird es sinnvoll n als Faktor der Performanz zwischen CPU A und B, wie in Gleichung 2.3 gegeben, zu definieren [124].

$$\text{n} = \frac{\text{Performanz}_A}{\text{Performanz}_B} = \frac{\text{Ausführungszeit}_B}{\text{Ausführungszeit}_A} \tag{2.3}$$

Eine weitere Größe ist die sogenannte CPU Ausführungszeit in der die CPU für den jeweiligen Benutzer arbeitet, da es bei Designänderung nur interessant ist, wie lange der Nutzer danach, im Vergleich zum vorherigen Design, für die Ausführung seiner Aufgaben benötigt. Damit kann die CPU Ausführungszeit für ein Programm, wie Gleichung 2.4 zu entnehmen, über die Anzahl der CPU Taktzyklen des Programmes und die Taktzykluszeit definiert werden [124].

$$\text{Ausführungszeit} = \#\text{Taktzyklen} \times \text{Taktzykluszeit} \tag{2.4}$$

Hieraus folgt direkt, dass der Designer zwei Möglichkeiten zur Optimierung der Performanz hat. Zum einen kann eine Optimierung der Performanz über weniger Taktzyklen pro ausgeführtes Programm und zum anderen über eine Verringerung der Taktzykluszeit erreicht werden.

In den bisherigen Betrachtungen wurde die Anzahl der Instruktionen noch nicht beachtet. Da der Compiler immer Instruktionen passend zur jeweiligen Zielarchitektur generiert, muss die Anzahl der Instruktionen auch einen Einfluss auf die Performanz haben. Somit kann die Zahl der zur Ausführung eines Programmes benötigten Taktzyklen, wie in Gleichung 2.5 gegeben, über die Instruktionen und die durchschnittliche Taktzyklen pro Instruktion definiert werden [124].

$$\#\text{Taktzyklen} = \#\text{Instruktionen} \times \text{mittlere Taktzyklen pro Instruktion (CPI)} \tag{2.5}$$

Der eigentlich interessante Teil der Formel ist hier die Zahl der mittleren Taktzyklen pro Instruktion (*Average Cycles per Instruction* (CpI)). Somit wird es möglich zwei unterschiedliche Implementierungen der gleichen Architektur zu vergleichen und zu klassifizieren.

Hieraus lässt sich die klassische CPU Performanz Gleichung ableiten [124]:

$$\begin{aligned} &\text{Ausführungszeit} = \\ &= \#\text{Instruktionen} \times \text{mittlere Taktzyklen pro Instruktion (CPI)} \times \text{Taktzykluszeit} \end{aligned} \tag{2.6}$$

Diese Formel gibt Aufschluss über die drei Schlüsselfaktoren zur Definition der Performanz. Somit lassen sich, neben unterschiedlichen Programmen oder Compilern, auch unterschiedliche Implementierungen klassifizieren. Es ist wichtig hervorzuheben, dass die gesamte Rechenzeit weiterhin der Schlüsselfaktor für die Performanz ist, da eine Optimierung einer Größe in der Formel zu einer Verschlechterung einer anderen Größe führen kann, was sich als Verschlechterung auf das Gesamtergebnis auswirken kann.

$$\text{Effizienz} = \frac{\text{Performanz}}{\text{Fläche}} \tag{2.7}$$

Zur Bewertung der Effizienz der Berechnung muss die Performanz, in diesem Fall die Ausführungszeit in Relation zum Aufwand, der für die Erzielung des jeweiligen Ergebnisses notwendig ist, gesetzt werden. Wie in Gleichung 2.7 aufgeschrieben, kann hier die Fläche, die zur Realisierung notwendig ist, ins Verhältnis gesetzt werden.

Um Abschließend die Effizienzsteigerung zu betrachten, indem beispielsweise zwei Realisierungsalternativen verglichen werden, kann Gleichung 2.3 entsprechend erweitert werden:

$$\text{Effizienzsteigerung}_{AB} = \frac{\text{Effizienz}_A}{\text{Effizienz}_B} = \frac{\text{Performanz}_A \times \text{Fläche}_B}{\text{Performanz}_B \times \text{Fläche}_A} \tag{2.8}$$

2.3. Superskalare Prozessoren

Beim Ausnutzen der zeitlichen Parallelität - dem Pipelining - ist der *Instructions per Cycle* (IpC) auf eins begrenzt, wobei er in der Realität meist deutlich kleiner als eins ist. Mit Hilfe der Superskalarität kann die Parallelität innerhalb der Befehlsebene weiter gesteigert werden, und somit auch der IpC größer als eins werden. Superskalare Prozessoren besitzen mehrere parallel angeordnete Funktionseinheiten (*Functional Unit* (FU)), die die parallel zugewiesenen Befehle simultan verarbeiten. Dieses Verfahren wird als Mehrfachzuordnung (*Multiple Issue*) bezeichnet [125]. Bei der Superskalarität handelt es sich um eine erweiterte Pipelining-Technik, die aufgrund einer höheren Parallelität und damit eines höheren IpCs eine verbesserte Performanz erzielen kann.

2.3.1. Aufbau und Funktionsweise

Der Aufbau eines superskalaren Prozessors lässt sich in die drei Bereiche gliedern [125]: Befehlshol- und Befehlszuordnungseinheit, Funktionseinheiten und Freigabeeinheit. Die Befehlshol- und Befehlszuordnungseinheit kann wiederum analog zu Abschnitt 2.2 pipelineartig in Stufen unterteilt werden. Die Decode-Stufe formt aus dem Eingangsbefehlsstrom mit Hilfe der integrierten Konflikterkennungslogik sowie der Sprungvorhersage einen konfliktfreien und spekulativen Befehlsstrom [148]. Zur Konflikterkennung und -lösung werden unterschiedliche Algorithmen und Techniken, wie Scoreboarding, Registerumbennung (*Register Renaming*) oder Tomasulo Algorithmus, benötigt [77]. Die darauf folgende Dispatcheinheit ordnet diese Instruktionen den jeweiligen Funktionseinheiten zu. Die Zuordnung der Befehle an den Funktionseinheiten wird als *Issue* bezeichnet. Zunächst werden diese Befehle in den Zwischenspeicher Reservation Station so lange speichert, bis alle Operanden verfügbar sind und der Befehl ausgeführt werden kann. Die eigentliche Befehlsausführung findet in den

jeweiligen Funktionseinheiten statt. Diese agieren unabhängig voneinander und können ebenfalls unterschiedlich lange Pipelinetiefen und somit auch Ausführungszeiten aufweisen. Ein superskalarer Prozessor ist skalierbar, da sich die Anzahl und die Kombination der parallel arbeitenden Funktionseinheiten nahezu beliebig kombinieren lassen. Bis zu den Reservation Stations erfolgt die schrittweise Ausführung der Befehle in der gleichen Anordnung wie die Programmreihenfolge es vorgibt (*In-Order* (IO) Ausführung). In der *Execute* Stufe können die Instruktionen jedoch außerhalb der Programmreihenfolge ausgeführt werden (Out-of-Order Execution), damit die Auslastung der einzelnen Einheiten gesteigert werden kann. Die Resultate werden der Freigabeeinheit bzw. der *Commit* Stufe zugeführt, die diese zunächst in einem Rückordnungspuffer (*Reorder-Buffer*) an der entsprechenden Stelle zwischenspeichert. Das Ergebnis an der ersten Stelle im Puffer kann nach Bestätigung der Instruktion zurück in den Registersatz geschrieben oder durch *Forwarding* als Operand den Reservation Stations zur Verfügung gestellt werden. In der Freigabeeinheit verlaufen die Prozesse wieder in Programmreihenfolge. Die Befehlsbeendigung und das Schreiben der Ergebnisse ins Zielregister werden als *Completion* bezeichnet.

2.3.2. Konflikte und Abhängigkeiten

Ähnlich wie beim Pipelining können superskalare Prozessoren den theoretisch maximalen IpC aufgrund auftretender Konflikte und Abhängigkeiten nicht erreichen. Durch die Parallelität ergeben sich dieselben Abhängigkeiten und Konflikte wie bei skalaren Prozessoren mit Pipelining (vgl. Unterabschnitt 2.2.2). Durch den Einsatz von *Out-of-Order* (OoO) Ausführung können weitere Datenabhängigkeiten auftreten. Einer dieser drei Konflikte ist der Ressourcenkonflikt, der auftritt, wenn beispielsweise zwei parallel auszuführende Befehle der gleichen Funktionseinheit zugeordnet werden sollen. Zur Lösung dieses Konfliktes muss der zweite Befehl so lange in der Reservation Station zurückgehalten werden, bis die Ressource wieder verfügbar ist. Das Blockieren des abhängigen Befehls wird auch als *Interlock* bezeichnet. Um die Wahrscheinlichkeit eines Ressourcenkonfliktes zu minimieren, können häufig verwendete Funktionseinheiten vervielfacht werden. Eine echte Datenabhängigkeit (*True Data Dependency*) tritt auf, wenn ein Befehl von einem Quelloperanden liest, der zuvor von einem vor-

angehenden Befehl geschrieben wurde [45]. Eine solche Abhängigkeit wird ebenfalls als *Read-After-Write* (RAW) Abhängigkeit bezeichnet. Mögliche Ansätze zur Auflösung dieser Abhängigkeit sind zum einen, den zweiten Befehl zu verzögern, oder zum anderen durch OoO Ausführung einen anderen unabhängigen Befehl vorzuziehen. Dies könnte weitere Datenabhängigkeiten, wie Gegenabhängigkeit (*Antidependence*) und Ausgabeabhängigkeit (*Output Dependence*), hervorrufen. Sprungabhängigkeit (*Procedural Dependence*) besteht, wenn im Befehlsstrom Verzweigungsbefehle auftreten. Bei unbedingten Sprüngen tritt nur eine Verzögerung von einem Takt auf, wenn bereits in der *Decode* Stufe die Sprungadresse berechnet wird. Dagegen wird bei bedingten Sprüngen die Sprungentscheidung erst in der *Execute* Stufe getroffen. Wenn dieser Konflikt durch *Interlocking* gelöst wird, dann können die nächsten Befehlspaare erst nach der *Execute* Stufe geladen werden. Dies würde den Durchsatz der superskalaren Pipeline schmälern. Eine effiziente Sprungvorhersage in der *Decode* Stufe ermöglicht ein Laden des nächsten Befehlspaares bereits nach der *Decode* Stufe, sodass die Performanz durch die spekulative Ausführung gesteigert werden kann. Wird zusätzlich eine *Delay Instruction* durch den Compiler nach der bedingten Verzweigung eingefügt, kann die Performanz weiter verbessert werden.

2.3.3. Kategorisierung von superskalaren Prozessoren

Bei superskalaren Prozessoren gibt es jeweils zwei Strategien für die Befehlszuordnung und Befehlsbeendigung. Die Befehlszuordnung erfolgt entweder in Form von *In-Order* oder *Out-of-Order* Ausführung. Analog dazu findet die Befehlsbeendigung entweder als *In-Order* bzw. *Out-of-Order* Completion statt. Aus den Kombinationen lassen sich superskalare Prozessoren in vier Kategorien einteilen, die sich in Performanz sowie Konflikte und Abhängigkeiten unterscheiden [45].

Als weiterführende Literatur zum Thema Superskalarität sei hier [45, 125, 148] empfohlen.

2.4. Sprungvorhersage

In diesem Abschnitt werden die in der Literatur bekannten Konzepte zur Sprungvorhersage vorgestellt, die für die adaptive Sprungvorhersage wiederverwendet werden. Bei der Sprungvorhersage werden häufig mehrere Konzepte kombiniert, um eine effizientere Ausführung von kontrollflusslastigen Programmen zu erreichen. Bei einer Verzweigung im Programmablauf wählt die Sprungvorhersage auf Basis der Ausführungsgeschichte des bisherigen Programmablaufs entweder statisch oder dynamisch einen Zweig aus. Dieser wird spekulativ ausgeführt bis die entscheidende Instruktion die *Execute* Stufe der Pipeline erreicht. Es wird dann geprüft, ob die Vorhersage korrekt war. Im Falle einer richtigen Vorhersage kann das Programm weiter ausgeführt werden. Bei einer Fehlvorhersage wird die spekulative Ausführung verworfen und es muss der alternative Programmpfad ausgeführt werden. Dies führt zu einer ineffizienteren Auslastung der Pipeline und zu einer längeren Ausführungszeit und sollte somit nach Möglichkeit vermieden werden.

Weiterhin wird zwischen lokalen und globalen Verfahren unterschieden. Es wird hierbei festgelegt, ob das *Branch History Register* (BHR) bzw. das *Pattern History Table* (PHT) für jeden Sprungbefehl einzeln, also pro Adresse, pro Adressbereich, oder für alle Befehle gemeinsam, also global, verwendet werden. Bei Verwendung eines globalen *Branch History Register* kann so die Korrelation zwischen den einzelnen Sprungbefehlen zur Verbesserung der Vorhersagequalität einbezogen werden. Alle Konzepte können für den Aufbau einer Sprungvorhersageeinheit kombiniert werden und durch verschiedene Prädiktoren umgesetzt werden. Es ist allgemein zu beachten, dass die Vorhersagequaliät nicht alleine von der Sprungvorhersage, sondern insbesondere auch vom Aufbau der Anwendung abhängt. Für ein Trefferquote von 90 % oder mehr wird laut McFarling et al. [108] beispielsweise ein *Pattern History Table* in einer Größenordnung von mindestens 30 *Bytes* benötigt.

2.4.1. Prädiktoren

In den folgenden Unterabschnitten werden die gängigsten Prädiktoren mit ihren jeweiligen Besonderheiten vorgestellt.

2.4.1.1. *Always Taken* (AT) & *Always not Taken* (ANT) Prädiktor

Statische Sprungvorhersagen sind sehr einfach realisiert und gehen davon aus, dass sich die Programme sehr ähnlich verhalten. Unter dieser Annahme ist es daher ausreichend ein feste Vorhersage zu festzulegen [23]. Die Variante *Always Taken* sagt vorher, dass immer das Sprungziel genommen wird, was vor allem bei inneren Schleifen sehr häufig zutrifft. Die alternative Variante *Always not Taken* sagt vorher, dass das immer das alternative Sprungziel zutrifft. Diese Vorhersagen erzielen meist keine hohen Trefferquoten ($\approx 60\,\%$) und werden schlechter, sobald der Programmablauf komplexer wird. Dafür lassen sie sich sehr einfach realisieren, da keinerlei Logik für die Aktualisierung und auch keine Prädiktorenfelder benötigt werden. Im Allgemeinen erzielt die *Always Taken* Vorhersage im Vergleich zur *Always not Taken* Vorhersage die besseren Ergebnisse [23]. Sie wird beispielsweise im Sun SuperSPARC eingesetzt, wo hingegen der Intel i486 die *Always not Taken* Vorhersage nutzt.

2.4.1.2. 1-Bit Prädiktor

Die dynamischen Prädiktoren bauen auf den statischen Prädiktoren auf. Eine sehr einfache Erweiterung bietet der 1-Bit Prädiktor. Hierbei werden die beiden statischen Prädiktoren gemeinsam verwendet. Ein einzelnes Bit zeigt, wie in Abbildung 2.4 zu sehen, welcher der beiden statischen Prädiktoren aktuell gewählt ist. Ist beispielsweise der *Always Taken* Prädiktor gewählt, wird dessen Vorhersage so lange genutzt, bis sich eine Vorhersage als falsch herausstellt. In diesem Fall wird auf die *Always not Taken* Vorhersage gewechselt. Diese ist wiederum aktiv bis sich eine Vorhersage als falsch herausstellt. Große Probleme hat der 1-Bit Prädiktor jedoch bei verschachtelten Schleifen, da er bei jeder Iteration der äußeren Schleife zwei Fehlprädiktionen auslöst. Aus diesem Grund wird er auch nur sehr selten, beispielsweise im DEC Alpha 21064 und AMD K5, eingesetzt [23].

2.4.1.3. 2-Bit Prädiktor

Aufgrund der Beschränkungen des 1-Bit Prädiktors wird das vorgestellte Konzept zu einem 2-Bit Prädiktor erweitert. Bei diesem Konzept stehen

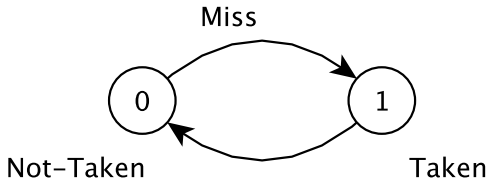
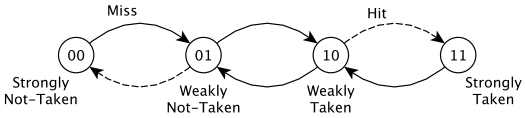


Abbildung 2.4.: 1-Bit Prädiktor [23].

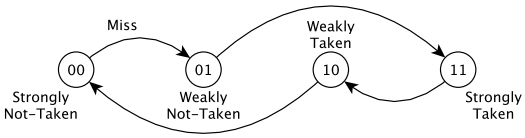
zwei Bits zur Verfügung, um insgesamt vier Zustände abbilden zu können. Somit ist es möglich jeweils einen schwachen und einen starken Zustand für Treffer und Fehltreffer zu definieren. Es gibt zwei Realisierungsalternativen, den Sättigungszähler und den Hysteresezähler, die sich leicht unterscheiden. Der Sättigungszähler ist in Abbildung 2.5a gezeigt. Der Übergang zwischen Treffer und Fehltreffer ist immer schwach, sodass weiterhin das Problem mit den verschachtelten Schleifen bestehen bleibt. Abhilfe schafft hier der Hysteresezähler, der, wie in Abbildung 2.5b gezeigt, nach einem Übergang von stark nach schwach direkt zu stark der Realisierungsalternative wechselt. Somit sind immer zwei falsche Vorhersagen nötig bis der Prädiktor umschaltet. Die vorhergesagte Sprungrichtung wird durch das höchstwertige Bit des Prädiktors angezeigt [23].

Im einfachsten Fall gibt es ein globales *Pattern History Table*. Dies ist jedoch sehr ineffizient, so dass meist 2^n *Pattern History Table* realisiert werden. Es ist hierbei jedoch zu beachten, dass es bei wenigen *Pattern History Table* mit höherer Wahrscheinlichkeit zu *Aliasing* kommt. Dies bedeutet, dass mehrere Adressen das gleiche *Pattern History Table* nutzen, und von unterschiedlichen Sprungadressen aktualisiert werden, die nicht zwangsläufig zur gespeicherten Vorhersage passen. Die Trefferquote steigt stark an, wenn sich möglichst wenig Sprungbefehle den gleichen Prädiktor teilen. Andererseits steigt auch der Realisierungsaufwand deutlich an.

Unter Verwendung einer großen PHT und somit vollständiger Vermeidung des *Aliasing* sind bei einem 2-Bit Prädiktor sogar Erfolgsquoten von bis zu 93,5 % möglich [108]. Aus diesem Grund wird der 2-Bit Prädiktor häufig auch als Basiselement für komplexere Prädiktormodelle verwendet. Im



(a) Sättigungszähler.



(b) Hysteresezähler.

Abbildung 2.5.: Realisierungsalternativen des 2-Bit Prädiktors [23].

Vergleich zum 1- Bit Prädiktor benötigt er jedoch aufgrund seiner zwei Bit pro PHT-Eintrag den doppelten Speicherplatz. Diese Vorhersage wird beispielsweise beim PPC605 und Motorola 68060 eingesetzt.

2.4.1.4. (m,n)-Korrelationsprädiktor

Der (m,n)-Korrelationsprädiktor verwendet die m-Bit eines globalen *Branch History Register*, um damit eine von $2m$ *Pattern History Table* auszuwählen, in welcher dann unter Verwendung der Speicheradresse ein n-Bit Prädiktor selektiert wird. Somit stehen im Idealfall für jeden Sprung $2m$ Prädiktoren zur Verfügung, aus welchen aufgrund der Korrelation zwischen den einzelnen Sprungbefehlen ausgewählt wird. Bei Verwendung zu kleiner *Pattern History Table*s kann es allerdings auch hier zu *Aliasing* kommen. Der Einsatz dieses Prädiktors ist außerdem nur dann sinnvoll, wenn die einzelnen Sprungbefehle über verschiedene Ausführungspfade erreicht werden. Sollten viele Sprünge ähnliche Ausführungspfade aufweisen, kann es sich ergeben, dass große Teile der *Pattern History Table* ungenutzt bleiben und so einen unnötigen zusätzlichen Flächenverbrauch verursachen. Da Versuche gezeigt haben, dass für eine Wahl von $n > 2$ keine Verbesserung der Vorhersage eintritt, werden für die *Pattern History*

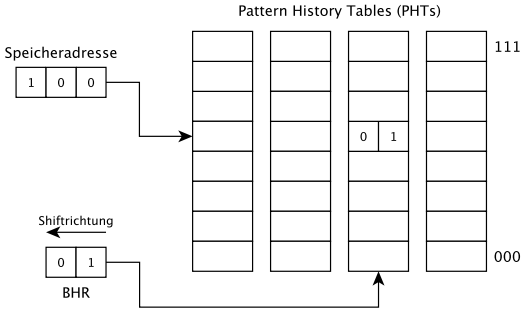


Abbildung 2.6.: (2,2)-Korrelationsprädiktor [120].

*Table*s in der Regel 1- oder 2-Bit Prädiktoren eingesetzt. Die Realisierung eines (2,2)-Korrelationsprädiktors ist in Abbildung 2.6 ersichtlich [120].

2.4.1.5. Zweistufig adaptiver Prädiktor

Im Gegensatz zu den bisher vorgestellten Prädiktoren, wird beim zweistufig adaptiven Prädiktor anstatt bzw. zusätzlich zur Adresse des Sprungbefehls ein *Branch History Register* (BHR) verwendet, um einen Prädiktor, beispielsweise 2-Bit, aus einer oder mehreren *Pattern History Table* zu selektieren [169]. Das *Branch History Register* ist als Schieberegister der Länge n realisiert und enthält dementsprechend die n letzten Sprungrichtungen der jeweiligen Programmausführung. Der Vorteil ist hierbei, dass der Prädiktor nicht nur auf die Adresse der jeweiligen Sprungbefehle festgelegt ist. Anstatt dessen erfolgt die Vorhersage auf Basis des protokollierten Programmverlaufs. Bei zu kurz gewähltem *Branch History Register* tritt jedoch häufig *Aliasing* auf. Vorteilhaft ist jedoch, dass ein Sprungbefehl, sollte dieser auf unterschiedlichen Pfaden verwendet werden, auch unterschiedliche Vorhersagerichtungen speichern kann. Dementsprechend kann, wie in Tabelle 2.1 gezeigt, eine Vielzahl von globalen und lokalen Varianten realisiert werden [170]:

Die Benennung der einzelnen Varianten unterliegt dabei folgendem Schema: Der erste Buchstabe beschreibt die Organisation des *Branch History*

	globale PHT	per-set PHTs	per-address PHTs
globales BHR	GAg	GAs	GAp
per-set BHT	SAg	SAs	SAp
per-address BHT	PAg	PAs	PAp

Tabelle 2.1.: Variationen zweistufig adaptiver Prädiktoren [170].

Register, der letzte Buchstabe die Organisation der *Pattern History Table*s, während der mittlere Buchstabe A für adaptiv steht. Werden mehrere *Branch History Register* verwendet, sind diese zu einer *Branch History Table* (BHT) zusammengefasst.

Die Größe und Anzahl der benötigten *Branch History Table*s bzw. *Pattern History Table*s hängt bei diesem Prädiktor von der globalen bzw. lokalen Auslegung der jeweiligen Tabellen ab. Im Falle lokaler *Pattern History Table*s können bei großen Programmen schnell tausende von Tabellen benötigt werden, welche innerhalb des Prozessors vorgehalten werden müssen. Im Falle des GAg-Prädiktors wird dagegen der Platzbedarf im Vergleich zum 2-Bit Prädiktor nur unwesentlich vergrößert, da dort lediglich ein zusätzliches Schieberegister für die Speicherung des Sprungverlaufs benötigt wird. Die lokalen Varianten dieser Prädiktoren sind in der Lage Trefferquoten von bis zu 97,2 % zu liefern, benötigen dafür jedoch auch sehr große *Branch History Table*s [170]. Das Schema eines *GAg*-Prädiktors mit 3-Bit Branch History ist in Abbildung 2.7 zu sehen. Dieser Prädiktor kommt beispielsweise beim Intel PentiumPro und PII sowie auch AMD K6 zum Einsatz.

2.4.1.6. gShare & gSelect Prädiktor

Die *gShare* und *gSelect* Prädiktoren erweitern das Konzept des zweistufigen Prädiktors, indem sie das *Branch History Register* mit einem gleich großen Teil der Sprungadresse verknüpfen. Hierdurch soll das *Aliasing* der ebenfalls globalen *Pattern History Table* reduziert werden. Der *gShare* Prädiktor verwendet eine *XOR* Verknüpfung, wo hingegen der *gSelect* Prädiktor die jeweils niedrigwertigsten Bits konkateniert. Dies erhöht die Wahrscheinlichkeit Sprünge unterscheiden zu können, die an ähnlichen

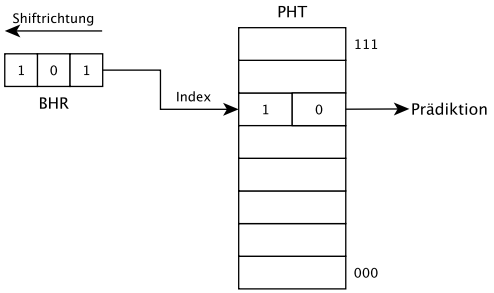


Abbildung 2.7.: Zweistufig adaptiver GAg-Prädiktor mit 3-Bit Branch History [170].

Adressen liegen oder im *Branch History Register* einen ähnlichen Verlauf haben.

Untersuchungen von McFarling et al. [108] haben gezeigt, dass beide Prädiktoren in etwa gleich gute Ergebnisse erzielen und unter Einbeziehung der Korrelation mit lokalem PHT Trefferquoten von bis zu 97 % ermöglichen. Im Vergleich zum im vorherigen Unterkapitel angesprochenen *GAg*-Prädiktor wird nur die zusätzliche Verknüpfung als Logik benötigt, also dementsprechend ressourcensparend global realisiert. Die Trefferquote erhöht sich jedoch signifikant, da *Aliasing* deutlich verringert wird. Typische Vertreter sind hier der Intel PIII und der AMD Athlon.

2.4.1.7. *Dynamic History Length Fitting* (DHLF)

Beim *gShare* Prädiktor ist das *Branch History Register* bitweise mit der Adresse verknüpft. Jedoch können die Sprünge je nach Programmablauf unterschiedlich stark oder gar nicht miteinander korrelieren, so dass die optimale BHR Länge nicht fest vorgegeben sein sollte. *Dynamic History Length Fitting* (DHLF) beschreibt ein Verfahren mit dem es adaptiven Prädiktoren möglich ist die optimale Länge des BHR zur Laufzeit zu ermitteln [95]. Es wird ein *Misprediction Table* erzeugt, deren Anzahl der

Einträge der maximalen Länge des *Branch History Register* entsprechen sollte.

Zunächst muss es eine Lernphase geben, um die optimale Länge des *Branch History Register* zu bestimmen. Für jedes Intervall bleibt die *Branch History Register* Länge fest. Nach Ende des Intervalls, beispielsweise nach einer festen Anzahl getroffener Vorhersagen, wird die BHR Länge neu festgelegt. Wird die BHR Länge geändert, muss dem Prädiktor eine Lernphase gegeben werden, um die PHT an den Programmverlauf anpassen zu können.

Es kann sein, dass die Evaluation der Länge des *Branch History Register*s keine bessere als die aktuelle Länge des BHR findet. Bei einer Verbesserung wird die neu ermittelte Länge genutzt. Die neue Länge des BHR sollte schrittweise angepasst werden, so dass auch mittlere Längen getestet werden und so eventuell noch ein Minimum gefunden werden kann.

2.4.1.8. *Local Minimum Avoidance* (LMA)

Bei der vorgestellten Vorgehensweise kann der Algorithmus bei der Suche nach der optimalen *Branch History Register* Länge in einem lokalen Minimum stecken bleiben. Je nach Ablauf des Programms kann die Suche nach dem globalen Optimum somit negativ beeinflusst werden. So kann es passieren, das mögliche *Branch History Register* Längen nicht in Betracht gezogen werden.

Um dies zu verhindern kann eine *Local Minimum Avoidance* (LMA) eingesetzt werden. Falls eine *Branch History Register* Länge wiederholt auftritt, soll sprunghaft eine andere *Branch History Register* Länge ausgewählt werden. Somit soll bei Bedarf das komplette Spektrum an möglichen *Local Minimum Avoidance* abgedeckt werden können. Bei jedem Einsatz der *Local Minimum Avoidance* wird die Länge zunächst verkürzt und beim nächsten Einsatz wieder verlängert. Falls das *Dynamic History Length Fitting* (DHLF) anschließend selbst wieder die gleiche optimale Einstellung findet, wird die Sprungweite vergrößert. Dies sollte sich so lange wiederholen bis ein neues Optimum gefunden wird.

2.4.1.9. Perceptron Prädiktor

Die meisten Prädiktoren versuchen durch große *Pattern History Table*s oder ausgeklügelte Auswahlverfahren das *Aliasing* so weit wie möglich zu reduzieren, ändern jedoch nichts an der grundlegenden Vorhersagetechnik. Der *Perceptron* Prädiktor verfolgt dagegen einen völlig anderen Ansatz. Er verwendet statt der bisher üblichen 2-Bit Prädiktoren ein *Perceptron* für die Vorhersage eines Sprunges [91]. Jedes *Perceptron* besteht dabei aus so vielen Gewichten, wie Einträge im *Branch History Register* vorhanden sind. Die Gewichte wiederum sind als vorzeichenbehaftete Integer ausgelegt. Außerdem werden die im *Branch History Register* gespeicherten Sprungrichtungen durch die Werte '1' für *Taken* und '-1' für *Not-Taken* repräsentiert. Die Speicherung der *Perceptrons* erfolgt wie bisher in einer Tabelle, welche über einen Teil der Sprungbefehlsadresse indiziert wird. Zur Ermittlung der vorhergesagten Sprungrichtung werden die Gewichte eines *Perceptrons* mit den Einträgen im *Branch History Register* multipliziert und anschließend aufaddiert. Sollte die dabei resultierende Summe ein positives Vorzeichen aufweisen, wird der Sprung als genommen und im Falle eines negativen Vorzeichens als nicht genommen vorausgesagt. Sobald die tatsächliche Sprungrichtung ausgewertet wurde, werden die *Perceptrons* aktualisiert. Dabei werden all die Gewichte, welche zur Vorhersage der korrekten Sprungrichtung beigetragen haben, vergrößert und alle anderen Gewichte verringert. Somit werden Gewichte, welche aufgrund von *Aliasing* falsche Vorhersagen liefern, betragsmäßig minimiert, während korrekt vorhersagende Gewichte betragsmäßig maximiert werden. Dabei ist jedoch zu beachten, dass die Höhe der Gewichte durch die gewählte Integergröße beschränkt ist. Mit zunehmendem Training der *Perceptrons* werden so die Auswirkungen des *Aliasing* immer weiter reduziert. Bei einem Hardwarebudget von 4 Kilobyte liefert der *Perceptron* Prädiktor ein um 10,1 % besseres Ergebnis als ein *gShare* Prädiktor mit gleich großer *Pattern History Table* [91]. Durch die zusätzlich notwendige Logik, die für die Berechnung der Prädiktion sowie die aufwändigere Aktualisierung der Gewichte benötigt wird, weist der *Perceptron* Prädiktor jedoch einen höheren Platzbedarf als ein Prädiktor mit vergleichbarem Hardwarebudget auf. Der Aufbau eines *Perceptron* Prädiktors mit 5-Bit *Branch History* kann in Abbildung 2.8 betrachtet werden.

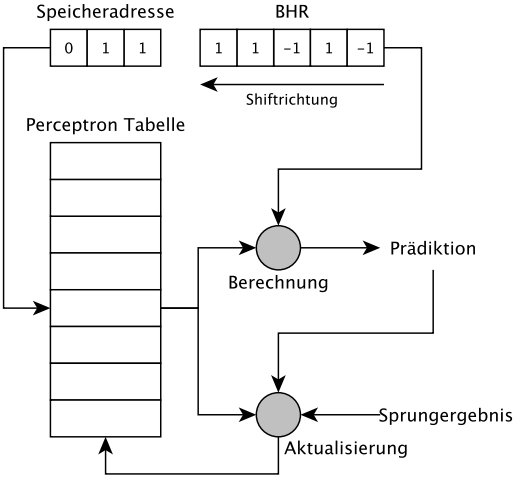


Abbildung 2.8.: *Perceptron* Prädiktor mit 5-Bit *Branch History* [91].

2.4.1.10. *Branch Target Buffer* (BTB)

Eine weitere Möglichkeit, die Auswirkungen der Kontrollflusskonflikte zu minimieren, ist die Verwendung eines *Branch Target Buffer* (BTB). Dieser ist als kleiner Cache-Speicher realisiert, welcher die Sprungziele der letzten n dekodierten Sprungbefehle enthält. Deshalb wird er häufig auch als *Branch Target Address Cache* (BTAC) bezeichnet [147]. Wird ein im *Branch Target Buffer* gespeicherter Sprungbefehl erneut geladen, kann dieser schon in der *Instruction Fetch* Stufe als Sprungbefehl erkannt werden. Zusätzlich wird aufgrund der ebenfalls im *Branch Target Buffer* abgelegten Vorhersagebits entschieden, ob der Sprung genommen wird oder nicht. Falls der Sprung genommen wird, kann so der Befehl an der Sprungzieladresse bereits im nächsten Takt geladen werden.

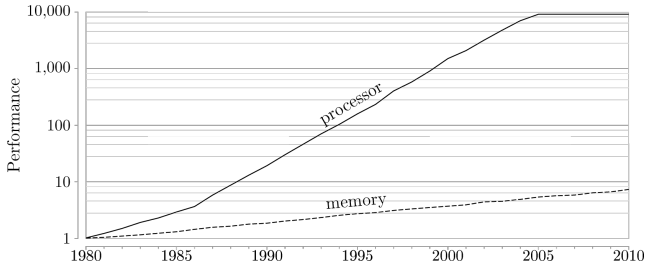
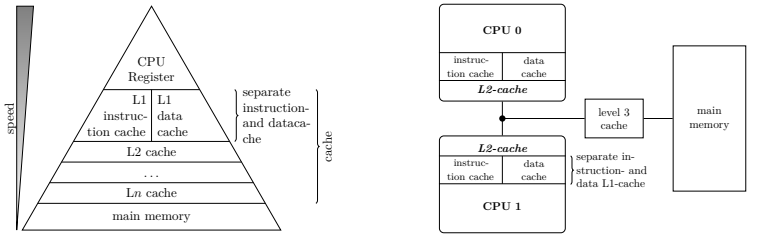


Abbildung 2.9.: Speicherperformanz vs. Prozessor Performanz [77].

2.5. Cache

Die Taktfrequenz der Rechenkerne ist wesentlich höher als die der Periphierie, insbesondere bei Bussen und Speichern. Dadurch entsteht ein Flaschenhals, da es nicht möglich ist die zur Berechnung notwendigen Daten schnell genug zum Rechenkern zu transportieren. Aus diesem Grund wurden im Jahre 1968 von IBM beim System/360 Model 85 erstmal ein Cache eingesetzt, um häufig oder zuletzt verwendete Daten in Prozessorgeschwindigkeit zwischenzuspeichern. Caches sind kleine, schnelle und teure Speicher die verglichen mit langsameren Speichern (z.b. *Double Data Rate* (DDR) mit einem Transistor) sehr viel mehr Fläche (vier bis sechs Transistoren) einnehmen. Abbildung 2.10a zeigt die Speicherpyramide mit n Cacheebenen. Abbildung 2.10b verdeutlicht einen exemplarischen Aufbau mit drei Cacheebenen. Sie werden auf der ersten Ebene direkt am Prozessor angeschlossen. Auf dieser Cacheebene werden für gewöhnlich zwei getrennte Caches für Instruktionen und Daten entsprechend der *Harvard* Architektur verwendet. Weitere Ebenen, beispielsweise über Busse sind denkbar, die zumeist in der *von-Neumann* Architektur realisiert werden. Der *Last Level Cache* (LLC) ist der Cache direkt vor dem Hauptspeicher. Die höheren Cacheebenen sind langsamer dafür aber günstiger realisierbar. Somit hat man jedoch den größten Vorteil nur, wenn die Daten in einer möglichst niedrigen Cacheebene vorgehalten werden können. Dementsprechend kann also der vergleichsweise langsame Zugriff auf höhere Cacheebenen oder sogar den Hautspeicher vermieden werden.



(a) Speicherpyramide. **(b)** Exemplarische Speicherhierarchie.

Abbildung 2.10.: Speicherpyramide (links) und exemplarische Speicherhierarchie (rechts) mit drei Cacheebenen zwischen Rechenkern und Hauptspeicher [77].

Der Cache ist für den Anwender und auch die CPU selbst transparent. Dadurch wird der mittlere Anwendungsfall abgedeckt. Dies hat jedoch zur Folge, dass nicht alle Algorithmen gleich effizient abgearbeitet werden können und für unterschiedliche Cachekonfigurationen optimiert werden müssen [49]. Durch die Transparenz müssen sich die Caches selbst verwalten und bei einer Anfrage von der CPU die zwischengespeicherten Daten weiterleiten. Falls das angeforderte Datum nicht vorhanden ist, muss der Cache es eigenständig aus einer höheren Cacheebene oder dem Hauptspeicher laden.

2.5.1. Modell & Abbildung

Wie bereits angesprochen sind die Caches wesentlich kleiner als die Menge an Daten, die für die Berechnungen benötigt werden. Aus diesem Grund ist ein Cachemodell und eine Abbildung auf dieses Modell notwendig. Der Cache ist grundlegend in Zeilen mit der Zeilenlänge (L) unterteilt. Mehrere dieser Zeilen ergeben einen *Direct Mapped* (DM) Cache, auf den immer die feste Anzahl von Zeilen im Hauptspeicher auf die im Cache vorhandenen Zeilen abgebildet wird, wie in Abbildung 2.11 zu erkennen. Der Hauptspeicher wird also in Regionen in der Größe des Caches aufgeteilt. Um diese Abbildung einfach zu halten wird die Anzahl der Cachezeilen

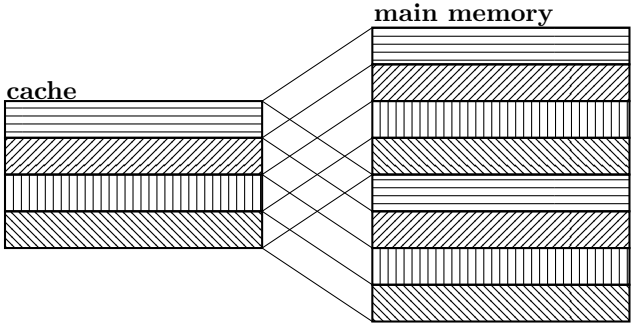


Abbildung 2.11.: Abbildung vom Speicher auf den Cache [124].

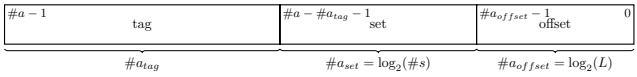


Abbildung 2.12.: Cache Adressschema [124].

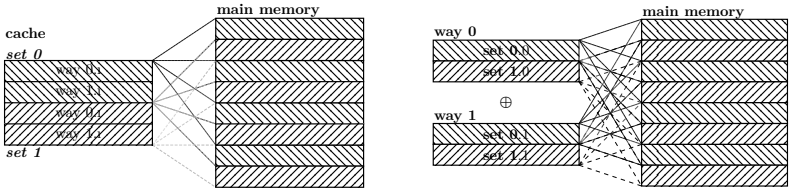
(#l) auf 2^n festgelegt. Dementsprechend ergeben sich $\log_2(\#l)$ Adressbits für die Cachezeilen.

Zur Adressierung werden, wie in Abbildung 2.12 gezeigt, *Tag*, *Index* und *Offset* unterschieden. Die niedrigwertigsten Bits, der *Offset*, zeigen auf die einzelnen Bytes in der Cachezeile. Die darauffolgenden $\log_2(\#s)$ Bits zeigen auf die *Sets*, einer Gruppe von Zeilen, und entsprechen also der Setadresse. Die höchstwertigen Bits entsprechen dem *Tag*, der Adresse im Set an der die eigentlichen Daten im Cache gespeichert werden.

Im Beispiel der *Direct Mapped* Caches entspricht ein *Set* einer Cachezeile. Dies macht die Abbildung ineffizient, da das Laden einer anderen Cachezeile der selben Satzadresse diese Zeile verdrängt, obwohl möglicherweise eine andere Zeile für die Speicherung der neuen Daten zur Verfügung steht. Die Daten müssten also nicht verdrängt werden. Abhilfe schaffen hier assozitive Caches. Hier können mehrere Cachezeilen in einem Satz mit der Satzgröße (s) enthalten sein. Dementsprechend verringert sich die Anzahl der Sätze (#s) bei konstanter Cachegröße. Typische Vertreter sind 2-Wege, 4-Wege und 8-Wege Caches mit jeweils zwei, vier und acht Zeilen pro Satz. Bei vollassoziativen Caches sind alle Cachezeilen in einem Satz

	lines per set s	sets #s	
Direct Mapped	1	#l	Einfache Realisierung, schnell
s-Wege assoziativ	s	$\frac{\#l}{s}$	Kompromiss
Vollassoziativ	#l	1	Höchste Flexibilität

Tabelle 2.2.: Cache mappings.



(a) Satzpartitionierung. **(b)** Wegpartitionierung.

Abbildung 2.13.: Speicherabbildung für einen 2-Wege Cache in Satzpartitionierung (links) und Wegpartitionierung (rechts).

($\#s = 1$ mit $s \leq 1$) enthalten. Tabelle 2.2 gibt einen Überblick über die Cacheabbildungen und der daraus resultierenden Satzgröße (s) und der entsprechenden Anzahl der Sätze (#s).

Abbildung 2.13 zeigt die Abbildung des Cache für einen 2-Wege Cache. In der Literatur wird der Cache als ein Speicher dargestellt (siehe Abbildung 2.13a). Es ist auch eine alternative Darstellung möglich, die die Wege als 2 parallele Cache darstellt (siehe Abbildung 2.13b). Für das Konzept dieser Arbeit wird die zweite Darstellung verwendet. Es muss jedoch beachtet werden, dass jedes Datum nur in einem der beiden parallelen Cachewege stehen kann.

2.5.2. Verdrängungsstrategien

Bei assoziativen Caches muss festgelegt werden welche Zeile aus dem Cache verdrängt werden muss. Es gibt unterschiedliche Strategien, die sich vor allem im Realisierungsaufwand und der Performanz unterscheiden.

Optimaler Algorithmus: Der optimale Algorithmus ersetzt die Daten, die am längsten nicht mehr genutzt wurden []. Dieses multi-objektive Vorgehen ist nicht realisierbar und bietet die Grundlage für alle weiteren Strategien, in welchen die Fehltrefferraten einer alternativen Strategie mit dieser verglichen werden.

Pseudozufall: Dieser Algorithmus lässt sich vergleichsweise einfach durch einen Zähler realisieren, der bei jeder Verdrängung um eins inkrementiert wird und mit der Anzahl der Zeilen ($z \times \mod s$) verrechnet wird. Dieser Wert zeigt auf die Zeile, die ersetzt werden soll.

Round Robin: Diese Strategie arbeitet nach dem *First In – First Out* (FIFO) Prinzip, hat jedoch das Problem der FIFO bzw. *Bélády* Anomalie [19]. Hierbei kann ein größerer Cache zu deutlich mehr Fehltreffern im Cache führen [48]. Die einfachste Realisierung wäre ein Modulo eins Zähler.

***Least-Recently-Used* (LRU):** *Least-Recently-Used* ersetzt die Zeile die am längsten nicht verwendet wurde. Hierzu wird eine Liste angelegt, die die Zugriffe auf die Zeilen protokolliert. Dementsprechend wird die Zeile im letzten Tabelleneintrag ersetzt.

***Least Frequently Used* (LFU):** *Least Frequently Used* ersetzt die Zeile die am wenigsten häufig verwendet wurde. Hierzu wird die Liste entsprechend der Häufigkeit der Zugriffe auf die entsprechende Zeile aktualisiert. Die Zeile am Ende der Liste wird dementsprechend aktualisiert.

Die Realisierungen können entsprechend Damien [35] erfolgen. Hier werden auch noch weitere Strategien neben den hier kurz angesprochenen vorstellt.

2.5.2.1. Schreibstrategien

Es gibt mit *Write Through* (WT) und *Write Back* (WB) zwei Strategien zum Zurückschreiben in den Cache. Diese können jeweils mit oder ohne *Write Allocate* (WA) genutzt werden.

***Write Through* (WT):** *Write Through* aktualisiert den Cache und leitet die Daten direkt an die nächste Hierarchieebene weiter. Dies erleichtert die Cachekohärenz, bringt jedoch das Problem mit sich, dass die Busse und

Kommunikationsnetzwerke vor allem in größeren Architekturen schnell sehr stark überlastet sind.

***Write Back* (WB):** *Write Back* aktualisiert den Cache ohne das geschriebene Datum an die nächste Hierarchieebene weiterzuleiten. Die Daten werden also erst beim Ersetzen einzelner Zeilen oder einem Cache *Flush* synchronisiert. Hierdurch wird der Bus deutlich entlastet. Bei kohärenten Mehrkernarchitekturen muss allerdings entweder durch die Software oder durch die Hardware die Kohärenz, also die Synchronisation der Daten, hergestellt werden.

***Write Allocate* (WA):** Falls eine Adresse bereits gecached ist, müssen die Daten im Cache aktualisiert werden. Wenn die Adresse jedoch noch nicht im Cache vorhanden ist, kann entschieden werden, ob sie direkt an die nächste Hierarchieebene weiter geleitet wird, oder ob eine Zeile im Cache allokiert werden soll. Es ist zu beachten, dass durch das Zurückschreiben einzelner Daten eine deutlicher Mehraufwand im Vergleich zur Aktualisierung ganzer Zeilen entstehen kann.

2.5.3. Formalien

L_W = L / #W Zeilenlänge in Wörtern = Zeilenlänge / Bytes pro Wort
#s = #l / s Anzahl der Sätze = Anzahl der Cachezeilen / Satzgröße
$\#a_{Offset} = \log_2(L_W)$ Anzahl der Offsetbits der Adresse = $\log_2$(Zeilenlänge in Wörtern)
$a_{Satz} = \log_2(\#s)$ Satzadresse = $\log_2$(Anzahl der Sätze)

a_{Tag} = #a - $\#a_{Satz}$ - $\#a_{Offset}$
Tagadresse = Anzahl der Adressbits - Anzahl der Satzbits der Adresse - Anzahl der Offsetbits der Adresse

Tabelle 2.3.: Formale Zusammenhänge bei der Cacheabbildung

2.6. Scalable Processor ARChitecture (SPARC) & LEON3

Bei der SPARC handelt es sich um eine *General Purpose* (GP) 32-bit RISC Prozessor Architektur [2]. SPARC besteht aus einer 32-bit Integer Einheit (*Integer Unit* (IU)), einer Fließkomma Einheit (*Floating-Point Unit* (FPU)) nach *Institute of Electrical and Electronics Engineers* (IEEE)[1] Standard 754 und einer *Co-Processor* (CP) Einheit [2]. Jede dieser Einheiten besitzt einen eigenen Satz Register, so dass jede Einheit für sich konsistent ist. SPARC ist eine 32-bit byteaddressierte Prozessorarchitektur [151]. Diese Architektur kann kosteneffizient und mit hoher Performanz implementiert werden. Im Gegensatz zu Architekturen anderer Hersteller [8], [86], [109], ist die SPARC-Architektur in freier Lizenz verfügbar. SPARC International, Inc. unterstützt mehrere Implementierungen verschiedener Hersteller [56]. Alle Implementierungen haben die gleiche Instruktionssatz Architektur (ISA). In der vorliegenden Arbeit wird die LEON3-Implementierung von Gaisler Research [54], auf die am Ende des Kapitels noch detaillierter eingegangen wird, als Grundlage verwendet.

2.6.1. *Integer Unit* (IU)

SPARC definiert 55 grundlegende Instruktionen [2]. Diese Instruktionen umfassen logische, arithmetische, kontroll-transfer, Speicher und Multiprozessor Instruktionen [2].

2.6.1.1. Register-Fenster

Die SPARC Architektur ist eine registerintensive Architektur mit vielen sich teilweise überlappenden Registern, die als Fenster bezeichnet werden. Die Anzahl dieser Fenster liegt zwischen 2 und 32 und ist abhängig von der jeweiligen Implementierung. Daher kann die Anzahl der Register zwischen 40 (2 Fenster) und 520 (32 Fenster) liegen. Jedes der Fenster besteht aus

[1] Das IEEE ist ein weltweiter Berufsverband von Ingenieuren aus den Bereichen Elektrotechnik und Informatik mit Sitz in New York City.

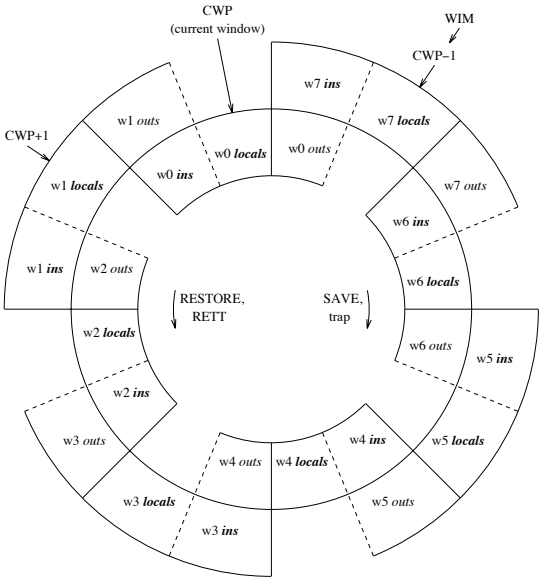


Abbildung 2.14.: Überlappende Register-Fenster (nwindows=8) der SPARC Architektur [151].

32 Registern. Diese 32 Register teilen sich, wie in Abbildung 2.14 zu sehen, in jeweils 8 globale (gemeinsame Register für alle Fenster), Eingabe- (*ins*), Lokale- (*locals*) und Ausgabe- (*outs*) Register [151].

Aufeinanderfolgende Register-Fenster teilen sich 8 Register, wobei die *out*-Register des jetzigen Fensters den *in*-Register des nächsten Fensters entsprechen. Das jeweilige Fenster wird durch den *Current Window Pointer* (CWP) angezeigt. Wird der CWP dekrementiert, wird das nächste Fenster aktiviert. Wird der CWP inkrementiert, wird das vorige Fenster aufgerufen. Das Statusregister *Window Invalid Mask* (WIM) markiert ein oder mehrere Fenster. Wird der CWP de- oder inkrementiert und ein markiertes Fenster aufgerufen, wird der *window under-* oder *overflow Trap* ausgelöst. Sich überlappende Register-Fenster sind ein effizienter Weg, um die Anzahl der Load-Store Instruktionen bei Aufrufen von Sprüngen zu Subroutinen zu minimieren und so auch die Zahl der, je nach der ge-

wählten Implementierung auftretenden, Cache Fehltreffer klein zu halten. Die reduzierte Anzahl der Load-Store Instruktionen bringt vor allem bei Multizyklus-Implementierungen eine erhöhte Performanz.

Die weiteren Status-Register *Processor State Register* (PSR), *Trap Base Register* (TBR), PC & nächster Programzähler (nPC) und *Multiply Step Register* (Y) sind in [151] ausführlich beschrieben.

2.6.1.2. Instruktionen

Alle SPARC Instruktionen sind 32 Bit Instruktionen, die durch eines der drei Formate, wie in Abbildung 2.15 gezeigt, definiert sind.

Format 1 definiert einen PC-relativen Aufruf mit einer 30-bit Verschiebungsvariablen. Somit kann zu jeder Adresse im Adressraum ein Aufruf mit einer Instruktion erfolgen. Verzweigungsinstruktionen und *SETHI* nutzen Format 2. Dieses Format definiert einen 22-Bit Immediate-Wert. Für Verzweigungen ist eine PC-relative Verschiebung um 8 MB möglich. Bei *SETHI* werden die hochwertigen 22 Bit des 32 Bit Wortes gesetzt, wobei die niedrigwertigen 10 Bit auf Null gesetzt werden. Format 3 kodiert die verbleibenden Load-Store-, Multiprozessor-, Arithmetisch-logischen- und Spezial-Instruktionen. Diese sind bei $i = 0$ mit zwei Quellregistern oder bei $i = 1$ mit einem Quellregister und einem Immediate-Wert kodiert.

Nur die Load-Store-Instruktionen können, wie im Allgemeinen bei RISC Architekturen üblich [123], auf das Speichermodell zugreifen. Bei Halbwort-, Wort- oder Doppelwortzugriffen lösen die Instruktionen einen Trap aus, wenn die Speicheradresse nicht richtig ausgerichtet ist. Die Daten sind im *Big-Endian* Format im Speicher abgelegt. Die drei Adressformate des Format 3 machen registerindirekte und -absolute Speicherzugriffe möglich. Die SPARC Architektur unterstützt keine verzögerten Load-Befehle, so dass die geladenen Daten direkt von der folgenden Instruktion weiter benutzt werden können. Allerdings wird die Performanz erhöht, wenn die Instruktion nach dem Load Befehl, wie in Abschnitt 2.2 beschrieben, nicht die geladenen Daten verwendet. Für alle Instruktions- und Datenzugriffe wird die 32-Bit Adresse und ein 8-Bit *Adress Space Identifier* (ASI) übergeben. Dieser wird vom *User/Supervisor* Bit im PSR in Zusammenhang mit dem Daten- und Instruktions-Indikator definiert. Dieser Mechanismus erlaubt es zusätzlich 252 frei wählbare 32-Bit Speicheradressräume hin-

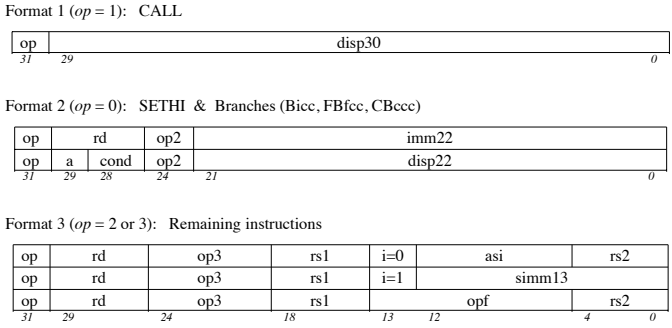


Abbildung 2.15.: SPARC Instruktionsformate [151].

zuzufügen, die über eine Konstante in der Instruktion direkt aufgerufen werden können. Die *Memory Management Unit* (MMU) selbst kann auch in einem solchen alternativen Raum liegen.

Zwei Instruktionen, SWAP und *Load-Store Unsigned Byte* (LDSTUB), unterstützen direkt *Symmetric Multiprocessor* (SMP). SWAP tauscht den Registerinhalt mit einem Wort aus dem adressierten Speicher. LDSTUB holt das adressierte Byte aus dem Speicher in ein Register und setzt die gelesenen Speicherstellen auf FF_{hex}.

Weitere Instruktionen sind die arithmetischen und logischen Instruktionen. Sie nutzen zwei Operanden und schreiben das errechnete Ergebnis in das Zielregister. Bei arithmetischen Instruktionen können zwei verschiedene Typen unterschieden werden. Einige Instruktionen aktualisieren den *Integer Condition Code* (ICC), andere nicht. Es gibt vier ICCs: *Negative* (N), *Zero* (Z), *Overflow* (V) und *Carry* (C). Diese sind im PSR gespeichert.

Es gibt zwei Spezialoperationen SAVE und RESTORE, die den CWP dekrementieren, inkrementieren oder ein Trap auslösen, wenn ein Window over- oder underflow ausgelöst wird. Diese Instruktionen führen gleichzeitig eine Addition aus, so dass der Stackpointer im gleichen Zyklus angepasst und gespeichert werden kann.

Weitere Instruktionen sind die kontroll-transfer Instruktionen wie CALL, BRANCH, JUMP and link und trap on condition code. Um diese Instruktionen effizient ausführen zu können nutzt SPARC das Konzept der

verspäteten Verzweigungen (*delay branches*) [2]. Dies bedeutet, dass nach der Verzweigung eine weitere Instruktion ausgeführt wird, ohne zu berücksichtigen, ob die Verzweigung tatsächlich genommen wurde. Der Compiler versucht zumeist nützliche Instruktionen in diese *delay slots* zu legen, um unnötig viele NOP-Operationen zu vermeiden. Zusätzlich dazu gibt es bei SPARC bedingte Verzweigungen, die ein zusätzliches annul Bit haben. Wird die Verzweigung genommen, wird auch die Instruktion im *delay slot* ausgeführt. Wird die Verzweigung jedoch nicht genommen, so wird die im *delay slot* folgende Instruktion annulliert.

2.6.2. LEON3

Die in dieser Arbeit verwendete Implementierung der SPARC V8 Spezifikation ist der LEON3 von Cobham Gaisler AB. Diese Implementierung wird vor allem im Bereich eingebetteter Systeme für *System-on-Chip* (SOC) Designs verwendet. Gaisler stellt die *Very High Speed Integrated Circuit Hardware Description Language* (VHDL) Beschreibung, inklusive vieler Werkzeuge zur Implementierung, Verifikation und Test, zum kostenfreien Herunterladen zur Verfügung [33]. Die Hauptmerkmale des LEON3-Prozessors sind seine Harvard-Architektur, also getrennte Instruktions- und Datenbusse zum parallelen Speicherzugriff, die siebenstufige Pipeline, getrennte Instruktions- und Datencaches, Hardwaremultiplizierer und Multiprozessorerweiterungen [53]. Die LEON3 Implementierung unterstützt zusätzlich zu allen im SPARC V8 Standard vorgesehenen Instruktionen die CASA Instruktion aus dem SPARC V9 Standard [152].

2.6.2.1. *Integer Unit* (IU)

Die LEON3 Integer Einheit implementiert die SPARC Spezifikation mit einer siebenstufigen Pipeline, die die Harvard-Architektur nutzt [53]. In dieser Pipeline sind getrennte Zugriffe auf Instruktions- und Datencaches möglich. Wie in der Spezifikation vorgesehen, unterstützt der LEON3 Register-Fenster. Die Anzahl der Fenster ist von der gewählten Implementierung abhängig (2-32), der Standardwert ist 8. Zusätzlich sind zur Beschleunigung der Multiplizier- und Dividieroperationen Hardwareeinheiten implementiert, die auch *Multiply-Accumulate* (MAC)-Operationen

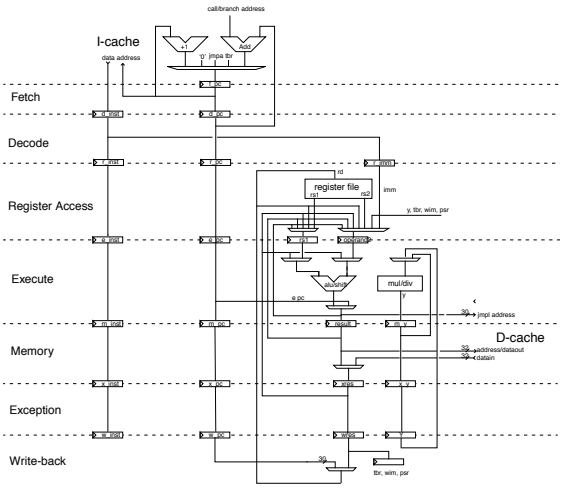


Abbildung 2.16.: LEON3 Integer Einheit Datenpfad Diagramm [32].

unterstützen [1]. Des Weiteren ist ein statische Sprungvorhersage (*branch always*) implementiert. Abbildung 2.16 zeigt das Diagramm der LEON3 Pipeline, das aus den folgenden Stufen besteht [32]:

- ***Instruction Fetch* (FE):** Die Instruktionen werden aus dem Instruktionscache geholt, wenn dieser aktiviert ist. Ansonsten wird direkt auf den *Advanced High-performance Bus* (AHB) zugegriffen. Die Instruktion ist am Ende der Fetch Stufe gültig und wird in der Integer Einheit gespeichert.
- **DE:** Die Instruktionen werden dekodiert und für CALL und BRANCH Instruktionen werden die Zieladressen berechnet.
- ***Register Access* (RA):** Die Operanden werden aus dem RF oder aus internen Nebenleitungen (z.B. bei *Forwarding*) geholt.
- **EX:** Arithmetik-, Logik- und Schiebeoperationen werden ausgeführt. Für Speicheroperationen und Sprünge werden die Adressen berechnet.

- ***Memory* (ME):** Der Datencache wird in dieser Stufe gelesen oder geschrieben.
- ***Exception* (XC):** *Traps* und *Interupts* werden gelöst. Bei Lesezugriffen auf den Speicher werden die Daten ausgerichtet.
- **WB:** Das Ergebnis aus den ALU, logischen, Schiebe- oder Speicheroperationen werden ins *Register-File* zurückgeschrieben.

Einige Instruktionen sind im LEON3 als Multizyklusinstruktion implementiert. Bei einem Reset werden nur die Programmzähler (PC & nPC), das *Supervisor* (S) und *Enable Traps* (ET) Bit im PSR zurückgesetzt. Weitere Informationen zu Details der Implementierung finden sich in [32].

2.6.2.2. Cache Subsystem

In der LEON3-Implementierung wird eine Harvardarchitektur verwendet. Daher kommen getrennte Instruktions- und Datencaches, mit jeweils eigenem Cache Controller, zum Einsatz. Beide Caches können somit getrennt, zu direkt abgebildetem (DM)[2] oder einem multi-set assoziativen[3] Cache von 2-4 Wegen, konfiguriert werden. Die Satzgröße ist von 1 bis 256 kByte aufgeteilt in Lines mit 16 oder 32 Byte Daten. In Multisatz-Konfigurationen kann aus einer der drei Strategien *Least-Recently-Used* (LRU), *Least-Recently-Replaced* (LRR) und (*Pseudo-*) *Random* gewählt werden. Zusätzlich kann eine Cacheline gesperrt werden, um ein Ersetzen zu verhindern. Die Cachebarkeit wird für beide Caches vom AHB-Bus kontrolliert, damit eine Kohärenz mit der aktuellen AHB-Konfiguration erhalten bleibt. Weitere Hintergrundinformationen zu Cache werden in Abschnitt 2.5 dargestellt.

[2] *Direct Mapped* (DM): n = 1, d. h. jeder Block repräsentiert einen eigenen Satz, es gibt also so viele Sätze wie Blöcke. Somit ist für eine gegebene Adresse exakt ein Cacheblock zuständig. Es existiert also eine direkte Abbildung zwischen Hintergrundspeicheradresse und Cacheblöcken [93].

[3] *Set Associative* (SA)): n wird zwischen 2 und 64 gewählt, d. h. die Cacheblöcke sind in Sätzen zu je n Blöcken angeordnet. Hier werden also $\frac{m}{n}$ direkt abgebildete Caches vollassoziativ (also frei) angewählt. Diesen Cache nennt man dann n-fach satzassoziativ oder kurz n-fach assoziativ [84].

2.7. *Language for Instruction-Set Architectures* (LISA) & Werkzeugkette

In diesem Abschnitt wird die *Architecture Description Language* (ADL) *Language for Instruction-Set Architectures* (LISA) und die entsprechende Werkzeug Kette der Firma Synopsys[4] vorgestellt.

2.7.1. *Language for Instruction-Set Architectures* (LISA)

Language for Instruction-Set Architectures (LISA) ermöglicht es dem Prozessor Entwickler sowohl verhaltensbasierte als auch zyklenakkurate Prozessormodelle und ihrer jeweiligen ISA zu erstellen [127]. Diese Modelle erfüllen alle Anforderungen an eine moderne High-Level *Design Space Exploration* (DSE), so dass alle möglichen *Hardware/Software Co-Design* (HSC) Designalternativen, eventuell auch mit Unterstützung durch Co-Simulations-Methoden, evaluiert werden können. Auf diesem Weg ist es möglich die Lücke zwischen der Hardwareentwicklung, beispielsweise mit *Hardware Description Language* (HDL), und der Software oder Compiler-entwicklung zu schließen, da alles aus der *Language for Instruction-Set Architectures* (LISA) Beschreibung abgeleitet bzw. generiert werden kann. Darüber hinaus können verschiedene Designalternativen, die besonders interessant erscheinen, evaluiert werden. Zusätzlich kann auch der Designfluss, der im nächsten Unterabschnitt 2.7.2 vorgestellt wird, evaluiert werden. Der Prozessor sowie auch die Werkzeuge, wie beispielsweise der Compiler, werden auf der höchsten Designebene entworfen. Dadurch findet die Designentscheidung für den weiteren Entwurf auf einer höheren Ebene statt. Anschließend wird der Prozessor auf der Applikationsebene für die jeweiligen Anwendungen passend zusammengestellt. Auf der Instruktionsebene wird eine effiziente Implementierung bezüglich der Performanz aber auch des Ressourcenverbrauchs, speziell für sehr häufig verwendete Instruktionen oder Mikroprogramme, durchgeführt. Dieses *Hardware/Software Co-Design* bietet ausgehend von der Hardware oder der Software jeweils eine Vielzahl an Optimierungsmöglichkeiten [70]. LISA stellt auch die Möglichkeiten zur Beschreibung von Pipelines auf der operativen Ebene

[4] http://www.synopsys.com

zur Verfügung, die die Beschreibung von komplexen Prozessor Pipeline Mechanismen ermöglicht. Eine Instruktion besteht somit aus mehreren Operationen, die als Register Transfer Modell in jeweils einem Zeitschritt beschrieben werden [173]. Je nach Anforderung an die Genauigkeit kann ein Kontrollschritt als eine Instruktion, als ein Zyklus oder als ein phasenakkurates Zeitmodell betrachtet werden.

In dieser Arbeit wird ein umfangreiches Prozessormodell entwickelt, das anschließend um adaptive Fähigkeiten erweitert wird. Auf Basis dieses adaptiven Prozessormodells können alle Werkzeuge für die Hardware- und Softwareentwicklung direkt erstellt werden. Hierdurch wird die Design-, sowie auch die Entwicklungszeit deutlich reduziert.

2.7.2. *Processor Designer* (PD)

Im Folgenden wird ein Überblick über das in der vorliegenden Arbeit verwendete Werkzeug *Processor Designer* (PD) der Firma Synopsys gegeben. Dieses komplexe Werkzeug wurde von einer Gruppe am *Chair for Integrated Signal Processing Systems* (ISS) der Rheinisch-Westfälischen Technische Hochschule (RWTH) Aachen entwickelt, die auch, wie im vorigen Unterabschnitt 2.7.1 beschrieben, maßgeblich für die *Architecture Description Language* (ADL) LISA verantwortlich ist [80]. Das Werkzeug wurde in den 2000er Jahren von der Firma CoWare kommerzialisiert und vermarktet. Am Anfang 2010 wurde CoWare von der Firma Synopsys aquiriert[5].

Mit den heutigen *Time to Market* (TtM) Anforderungen wird es immer schwieriger komplexe neue *Application-Specific Instruction-Set Processor* (ASIP) Architekturen schnell und optimiert in einem Produktzyklus umzusetzen [98]. Von der Konzeption bis zur Implementierung werden verschiedene Gruppen mit unterschiedlichsten Kompetenzen benötigt. Diese Lücke, zwischen *Design Space Exploration* und anschließender Implementierung im *Tool Design*, schließt der *Processor Designer* (PD). Dies ergibt die Möglichkeit alle durch die LISA-Beschreibungen gegebenen Parameter innerhalb einer Werkzeugumgebung umzusetzen und zu verifizieren. Durch die Verbindung der *High-Level*-Beschreibung der Software-Tools und der

[5] http://synopsys.mediaroom.com/index.php?s=43&item=775

Low-Level-Beschreibung der Hardwareimplementierung innerhalb einer LISA-Beschreibung ist es möglich Designänderungen schnell, im bestehenden Modell, umzusetzen. Sie betreffen nur die LISA-Beschreibung und nicht alle Entwicklungswerkzeuge, daher werden die Abhängigkeiten der einzelnen Werkzeuge anhand der neuen Beschreibung aktualisiert. In der Entwicklungsumgebung lassen sich dementsprechend alle, für den kompletten Entwicklungsprozess notwendigen, Werkzeuge automatisch aus der LISA-Beschreibung generieren [139].

2.7.3. *Platform Architect* (PA)

SystemC ist eine Modellierungssprache auf Systemebene und basiert auf der Programmiersprache C++. Durch die zusätzliche C++ Bibliothek wird das Standard C++ um weitere Attribute ergänzt, die Hardwareverhalten, wie Nebenläufigkeit und Timing, beschreiben können. Mit Hilfe von SystemC ist es möglich, sowohl die Hardware als auch Software aller Komponenten auf Systemebene zu verifizieren und zu optimieren. Eine genaue Verifikation auf Systemebene ist notwendig, da ein spät erkannter Performanzverlust zu höheren Entwicklungskosten und einer längeren Entwicklungsdauer führen kann [156]. Die Software *Platform Architect* (PA) der Firma Synopsys[6] ist ein Tool zur Simulation und Analyse von SOC auf Systemebene. Das vom PD generierte SystemC Modell kann in den PA importiert und mit einer Busschnittstelle verbunden werden. Über diese Busschnittstelle erfolgt die Kommunikation mit weiteren Systemkomponenten (Speicher, weitere Prozessoren, etc.), die an den Bus angeschlossenen sind.

2.7.3.1. SystemC Simulation

Die Modellierung auf Systemebene koexistiert mit der Entwicklung auf *Register Transfer Level* (RTL) Ebene als wichtiger Schritt im Systementwurf. Bis heute werden beide Entwurfsmöglichkeiten unabhängig voneinander mit unterschiedlichen Zielen und wenig Schnittstellen zwischen einander genutzt. Ein wichtiger Vertreter der abstrakten System Modellierung ist

[6] http://www.synopsys.com/Prototyping/ArchitectureDesign/Pages/platform-architect.aspx

die C-Bibliothek SystemC, die in der Industrie sowie auch in der Wissenschaft weit verbreitet ist und vor allem für den System Entwurf sowie auch zur Verifikation Anwendung findet. Zunächst wurde SystemC entwickelt, um Nebenläufigkeiten in sequenziellen C-Programmen basierend auf der *Hardware Description Language* (HDL) Beschreibung zu modellieren. Erst spätere Weiterentwicklungen führten zu der eigentlichen Modellierung auf höherer Abstraktionsebene.

Dies wurde insbesondere durch die Einführung des Kommunikationsstandards *Transaction Level Modeling* (TLM) forciert [58]. Mit TLM ist es möglich alle unterschiedlichen Formen von Hardwaremodellen in eine einzige Plattform zu integrieren. Wenn Modelle kein TLM Interface zur Verfügung stellen, können *Wrapper* erstellt werden, mit denen sie in das Modell integriert werden können. Damm et al. [36] beschäftigen sich mit der Verbindung von TLM und *Analog/Mixed Signal* (AMS) Modellen. Jindal et al. [92] verwenden RTL Testumgebungen für SystemC Modelle.

Neben der schnellen Erstellung der Modelle liegt der größte Vorteil in sehr schnellen Simulationsgeschwindigkeiten. Im Gegensatz zu langsameren Simulationszeiten auf RTL Ebene ermöglichen SystemC Simulationen die Verifikation und Evaluation von komplexen Systemen und komplexen Applikationen in kürzerer Zeit. Rissa et al. [132] haben die Simulationszeiten von HDL-RTL Simulationen mit zyklen- bzw. pinakkuraten SystemC Simulationen verglichen, um die Beschleunigung der Simulationen zu belegen. Um die Simulationsgeschwindigkeiten weiter steigern zu können, werden hardwarespezifische Fähigkeiten vernachlässigt und auf Genauigkeit verzichtet. Auf einer abstrakten Ebene beschreibt SystemC nur gemeinsame Schnittstellen und den Simulationskernel, der sich um die zeitliche Synchronisation kümmert. In schnelleren Modellen wird diese Synchronisation in jedem Zyklus gefordert.

2.8. Bildverarbeitungsalgorithmen

In diesem Abschnitt werden die Bildverarbeitungsalgorithmen vorgestellt, die in der Arbeit genutzt werden, um den adaptiven Prozessor (a-Core) zu evaluieren. Zunächst wird eine Faltung präsentiert, die als Filter, beispielsweise zur Glättung, in der Bildverarbeitung verwendet wird. Anschließend

werden die einzelnen Schritte eines Cannyfilters vorgestellt, um Kanten in einem Bild erkennen zu können.

2.8.1. Faltung

Die Faltung wird genutzt, um das Bild für die Kantenerkennung vorzuverarbeiten. Der Faltungsalgorithmus eines Bildes wird mit einer Faltungsmatrix realisiert, um ein Gradientenbild in x- bzw. y-Richtung zu erzeugen. Abbildung 2.17 zeigt den Faltungsalgorithmus einer 3×3 Matrix. Zunächst werden die Variablen initialisiert, die zur Ausführung des Algorithmus benötigt werden. Anschließend werden die Speicheradressen und die Größe des Bildes bestimmt. Hieraus werden die Schleifengrenzen abgeleitet. Die Faltung wird für jeden Punkt in drei Teilen realisiert. Drei Pixel werden aus dem Speicher geladen, berechnet und in das Register gespeichert. Die Ausführung des Algorithmus ist an die Hardware angepasst, in dem nur die acht im *a*-Core vorhandenen Register genutzt werden.

Bei jedem Pixel wird überprüft, ob das Ende der Zeile erreicht ist. Wenn dies der Fall ist, werden die letzten beiden Pixel ausgelassen, ansonsten werden die benachbarten Pixel berechnet. Das Ergebnisbild ist also in jeder Richtung einen Pixel kleiner als das Eingangsbild.

Abbildung 2.18 zeigt die Position des schwarzen Index Pixels und die entsprechende Position der Filter Matrix. Der Algorithmus bricht die Ausführung ab, sobald das Ende der Zeile erreicht ist. Der Algorithmus kann sowohl mit einem Gauss'schen- als auch mit einem Sobelfilter angewendet werden [38].

2.8.2. Cannyfilter

Die Realisierung der Algorithmen für den Cannyfilter basieren auf dem *Integrated Vision Toolkit* (IVT) [12]. Die Ausgabe Bilder der Faltungen mit den Sobelmatrizen S_X und S_Y müssen verfügbar sein um den Absolutwert des Grauwertgradienten M und die Richtung φ berechnen zu können. Eine Approximation des Grauwertes wird mit Gleichung 2.9 berechnet, um rechenintensive Wurzel- und Quadratfunktionen zu vermeiden.

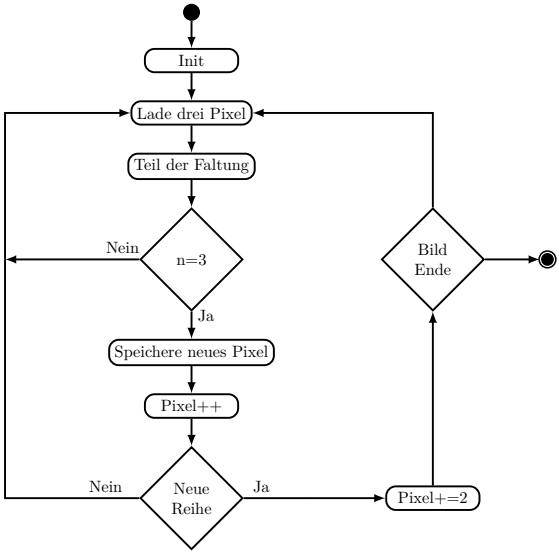


Abbildung 2.17.: *Unified Modeling Language* (UML)-Modell des Faltungsalgorithmus.

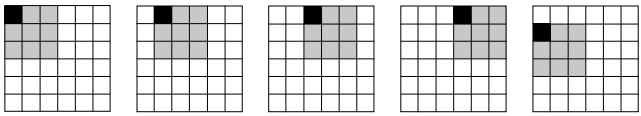


Abbildung 2.18.: Schwarz markiertes Pixel und zur Berechnung benötigte, grau markierte, Pixel des Faltungsalgorithmus.

$$M \approx |S_X| + |S_Y| \tag{2.9}$$

Auch für die Richtung φ wird eine Approximation durchgeführt, die $\arctan(\frac{S_X}{S_Y})$ ersetzt:

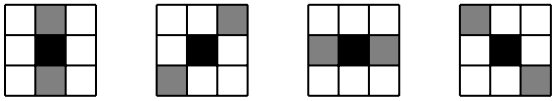


Abbildung 2.19.: Gradient φ und zur Berechnung notwendige benachbarte Pixel.

$$\varphi \approx \begin{cases} 90°, & \text{wenn } |S_X| > 2|S_Y| \\ 135°, & \text{wenn } 2|S_X| > |S_Y| \text{ und } S_X S_Y \geq 0 \\ 45°, & \text{wenn } 2|S_X| > |S_Y| \text{ und } S_X S_Y < 0 \\ 0°, & \text{sonst} \end{cases} \tag{2.10}$$

Die Multiplikation mit zwei kann durch eine bitweise Verschiebung nach links in einem Zyklus realisiert werden. Jede Approximation nutzt aus, dass Gleichung 2.11 immer valide ist:

$$arctan(\frac{1}{2}) = 25.57\,° \approx 22.5\,° \tag{2.11}$$

Es müssen jedoch nur benachbarte Pixel beachtet werden, so dass eine Ungenauigkeit von unter 3 % akzeptabel ist, ohne das Bild dabei verzerrt wird.

Abbildung 2.19 zeigt die vier möglichen Gradienten in Verbindung mit den benachbarten Pixeln wie in Gleichung 2.10 angegeben. Das aktuell schwarz markierte Pixel, dessen Gradienten berechnet werden, liegt in der Mitte. Die grauen Pixel zeigen mögliche Kanten, so dass die Pixel noch mit einer *Non-Maximum Suppression* getestet werden.

Abbildung 2.20 zeigt den Algorithmus der Gradientenberechnung. Die Pixel der *Sobel* S_X und S_Y Faltung werden an der aktuellen Position des Bildes geladen. Der Grauwertgradient M wird verglichen, um festzustellen ob das erkannte Pixel größer als der eingesetzte Schwellenwert T_1 ist. Nur falls dies der Fall ist, wird das Pixel als mögliche Kante deklariert und der Gradient φ berechnet. Andernfalls wird der Algorithmus weiter ausgeführt

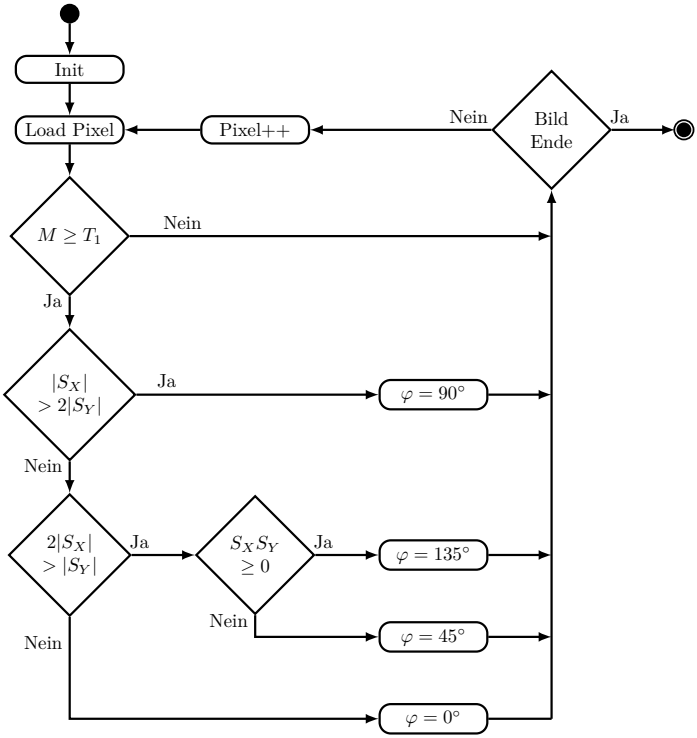


Abbildung 2.20.: UML-Modell des Cannyfilter Gradienten φ Algorithmus.

und das nächste Pixel verglichen. Der Algorithmus bricht die Ausführung ab, sobald alle Pixel durchlaufen wurden.

Die resultierenden Daten werden mit Hilfe der *Non-Maximum Suppression* gefiltert. Dieser Algorithmus ist in Abbildung 2.21 gezeigt. Zunächst wird das zu bearbeitende Pixel gewählt. Falls der Grauwertgradient M größer als ein Schwellenwert T_1 ist, werden die Variablen j und k entsprechend φ gesetzt. j und k repräsentieren die Richtung des Gradienten entsprechend Gleichung 2.10. Die Breite des Bildes wird benötigt um die exakte Position zu bestimmen. Das gewählte Pixel ist ein lokales Minimum, wenn der Grauwertgradient der benachbarten Pixel kleiner ist als der Grauwertgradient des gewählten Pixels. Das aktuelle Pixel wird nur als Kante

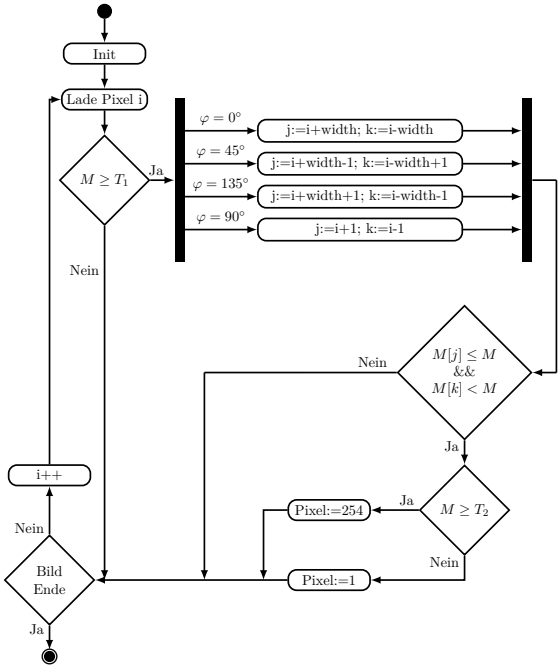


Abbildung 2.21.: UML-Modell des Cannyfilter *Non-Maximum Suppression* Algorithmus.

markiert, wenn der Grauwertgradient größer als ein Schwellenwert T_2 ist. Diese Berechnung wird für jedes Pixel wiederholt. Anschließend wird das komplette Bild durchgegangen und die markierten Pixel werden auf weiß gesetzt. Alle anderen auf schwarz. Das Ergebnis ist ein monochromes Bild, in dem die erkannten Kanten weiß markiert sind.

2.9. Nummerische Algorithmen

Es werden Algorithmen benötigt, die den Cache intensiv nutzen. Somit kommen insbesondere Algorithmen in Betracht, die wiederholt auf Elemente im Cache zugreifen.

2.9.1. Matrix Multiplikation (MMul)

Für eine $m \times n$-Matrix $\mathbf{A} = (a_{ij})$ und eine $n \times p$-Matrix $\mathbf{B} = (b_{jk})$ ergibt sich als Matrixprodukt eine $m \times p$-Matrix

$$\mathbf{C} = (c_{ik}) = \mathbf{AB} \tag{2.12}$$

mit den Elementen

$$c_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk} \,. \tag{2.13}$$

Bei der Berechnung der Matrix $\mathbf{C}$ muss mehrfach auf die Elemente der Matrizen zugegriffen werden. Diese sind in einer bestimmten Reihenfolge im Speicher angeordnet, so dass sie die Anordnung, bzw. die Reihenfolge der Zugriffe positiv auf die Laufzeit auswirkt. Die entsprechenden Optimierungen werden in Abschnitt 6.2 vorgestellt.

Algorithmus 2.1 Matrizenmultiplikation (ohne Optimierungen)

```
    procedure MATRIXPRODUKT(A, B, m, n, p)
2:      C ← NULLMATRIX(m, p)   ▷ Nullmatrix der Dimension m × p erzeugen
        for i ← 1, m do
4:          for k ← 1, p do
                for j ← 1, n do
6:                  (C)_{i,k} ← (C)_{i,k} + (A)_{i,j} (B)_{j,k}
        return C
```

2.9.2. LU-Zerlegung

Als LU-Zerlegung bezeichnet man die Darstellung einer $n \times n$-Matrix $\mathbf{A}$ als Produkt einer unteren Dreiecksmatrix $\mathbf{L}$ und einer oberen Dreiecksmatrix $\mathbf{U}$, wobei per Konvention alle Elemente der Hauptdiagonalen von $\mathbf{L}$ den Wert 1 haben. Die Existenz einer solchen Darstellung (d.h. einer Zerlegung der Form $\mathbf{A} = \mathbf{LU}$) ist nicht garantiert [131].

Bei der LU-Zerlegung mit Pivotisierung ist im Eliminationsprozess zusätzlich ein Vertauschen der Zeilen (bzw. Spalten) von $\mathbf{A}$ erlaubt, wobei durchgeführte Vertauschungen in einer Permutationsmatrix festgehalten werden. Mit diesem Verfahren kann jede quadratische Matrix $\mathbf{A}$ in der Form $\mathbf{PA} = \mathbf{LU}$ dargestellt werden [119], wobei $\mathbf{L}$ eine untere Dreiecksmatrix mit Einsen auf der Hauptdiagonalen, $\mathbf{U}$ eine obere Dreiecksmatrix und $\mathbf{P}$ die oben genannte Permutationsmatrix bezeichnet.

Algorithmus 2.2 LU-Zerlegung mit Pivotisierung (ohne Optimierungen)

```
    procedure LU-Zerlegung(A, n)
 2:     P ← Vektor(n)
        for k ← 1, n do
 4:         (P)_k ← k                                  ▷ keine Permutationen zu Beginn
        for k ← 1, n do
 6:         wähle m ∈ {k, ..., n}, so dass |(A)_{m,k}| ≥ |(A)_{i,k}| ∀_{i∈{k,...,n}}
           if (A)_{m,k} = 0 then
 8:             return                                 ▷ Zusatzbetrachtung nötig
           if m ≥ k then
10:             m-te und k-te Zeile in A vertauschen   ▷ Vertauschung durchführen
                (P)_m und (P)_k vertauschen            ▷ Tausch in P vermerken
12:         for i ← k + 1, n do
                (A)_{i,k} ← (A)_{i,k} / (A)_{k,k}      ▷ Koeffizienten zur Elimination speichern
14:         for j ← k + 1, n do
                for i ← k + 1, n do
16:                 (A)_{i,j} ← (A)_{i,j} − (A)_{i,k} (A)_{k,j}   ▷ übrige Elemente in A
    aktualisieren
        return (P, A)
```

In vielen Fällen kann eine solche LU-Zerlegung mit Pivotisierung durch die in Algorithmus 2.2 gezeigte Implementierung bestimmt werden. In speziellen Fällen wäre während des Prozesses jedoch eine Division durch

Null notwendig, was hier zum Abbruch des Verfahrens führt. Eine Implementierung, die auch in solchen Fällen eine Zerlegung findet, wird im Rahmen dieser Arbeit nicht betrachtet.

Da die Elemente der Hauptdiagonalen von $\mathbf{L}$ auf den Wert 1 festgelegt sind, werden diese nicht explizit berechnet oder gespeichert. So ist es möglich, sowohl $\mathbf{L}$ (bzw. $\mathbf{L} - \mathbf{I}_n$, wobei $\mathbf{I}_n$ die $n \times n$-Einheitsmatrix bezeichnet) als auch $\mathbf{U}$ direkt innerhalb von $\mathbf{A}$ zu speichern. Im Verlauf des Verfahrens werden die Eingabedaten somit überschreiben. Eine weitere Folge dieser Technik ist, dass die betrachteten Elemente nicht wie bei der Matrizenmultiplikation in drei verschiedenen Matrizen organisiert sind. Darüber hinaus werden bei der LU-Zerlegung zwischen den Aktualisierungsprozessen (siehe Zeile 16) noch weitere Schritte, wie etwa die Berechnung von Koeffizienten oder die Pivotisierung, ausgeführt. Der Einfluss dieser Unterschiede auf die Cache-Effizienz soll im Folgenden ebenfalls betrachtet werden.

Anstelle einer $n \times n$-Permutationsmatrix kommt zur Speicherung der Vertauschungen ein Vektor mit n Elementen zum Einsatz.

3. Stand der Technik

In diesem Kapitel wird ein Überblick über den Stand der Technik in der Wissenschaft und in der Industrie gegeben. Zunächst werden bekannte Mikroarchitektur Konzepte aufgegriffen und gegen den Ansatz in dieser Arbeit abgegrenzt. Anschließend folgen Applikationsspezifische, adaptive und rekonfigurierbare Prozessoren. Des Weiteren wird das dieser Arbeit zugrunde liegende Mehrkern-, und Vielkern Architekturkonzept vorgestellt. Abschließend wird eine Überblick über *Architecture Description Language* (ADL) gegeben.

3.1. Mikroarchitektur Konzepte

Im Kontext der vorliegenden Arbeit sind vor allem die Entwicklungen der in den späten 1970er und frühen 1980er Jahren veröffentlichten Arbeiten maßgeblich. Von Patterson et al. [122] wurde in Berkeley der RISC I und von Hennessy et al. [76] die *Very-large-scale Integration* (VLSI) Prozessorarchitektur entwickelt. Die Entwicklungen haben die Kategorie *Reduced Instruction Set Computer* (RISC) geprägt. Bereits seit den 1960er Jahren wurden Konzepte zu den *Complex Instruction Set Computer* (CISC) entwickelt [46]. Die CISC-Rechenkerne zeichnen sich dadurch aus, dass auch komplexe Instruktionen, bei denen die strikte Load-Store-Architektur der *Reduced Instruction Set Computer* (RISC)-Rechner aufgehoben wird, verwendet werden können. Die eigentliche Herausforderung beider Prozessorenarten liegt im Bereich der effizienten und schnellen Speicherverwaltung [123]. Durch den großen kommerziellen Erfolg der Firma Intel[1] und deren x86-Architektur in den 1990er Jahren sind die CISC Rechenkerne

[1] http://www.intel.com

einem breiten Anwenderfeld bekannt. Es wurden jedoch auch verschiedene RISC Prozessoren kommerzialisiert. Die wohl am weitesten verbreitete Weiterentwicklung ist die des RISC I aus Berkeley hin zur *Scalable Processor ARChitecture* (SPARC) Architektur, die von SPARC International Inc.[2] offen lizensiert wird. Die SPARC-Architektur wurde in den 1980er Jahren durchaus konkurrenzfähig zu Intel auf Workstations eingesetzt.

Der Intel Pentium I aus dem Jahre 1993 ist ein historischer Vertreter für einen In-order Issue, In-order Completion superskalaren Prozessor[45]. Es handelt sich um einen zweifach superskalaren Prozessor mit zwei fünfstufigen Integerpipelines (U- und V-Pipelines) [4]. Es können daher maximal zwei simple Instruktionen simultan ausgeführt werden.

Eine weitere weit verbreitete RISC-Architektur ist der *PowerPC* (PPC) [101, 150, 106], der in einem Konsortium der Firmen Apple[3], IBM[4] und Motorola[5] entwickelt wurde und vor allem in Apple-Computern in den 1990er Jahren und Servern verwendet wurde. In den 1990er Jahren schritt die Entwicklung der Prozessoren voran, so dass Erweiterungen, wie beispielsweise *Out-of-Order* (OoO) Ausführung und Spekulation, aber auch Hardwarebeschleuniger wie *Multi Media Extension* (MMX) [128], implementiert wurden. RISC Architekturen konnten sich in den 1990er Jahren nicht kommerziell gegen die x86 Architektur durchsetzen, so dass die heutige Hauptanwendung hauptsächlich im Bereich eingebetteter Systeme liegt. Aus diesem Feld kommt die ARM-Architektur [143]. Mit ressourcensparende Implementierungen bezüglich Fläche und Energie ist es der Firma *Advanced RISC Machines Ltd.* (ARM)[6] in den letzten Jahren gelungen die Marktführung im Bereich eingebetteter Systeme auch auf den Bereich mobiler Geräte, bei welchem ressourceneffiziente Konzepte erfolgsentscheidend sind, zu erlangen [43]. Dies wurde im Kontext des ARM-internen Projektes zum Thema *Dynamic Voltage Scaling* (DVS) *RAZOR* vorangetrieben [27, BMH+13, MBFT13].

Einen bereits kommerzialisierten Ansatz zur Lösung dieser Anforderungen bietet die ARM *big.LITTLE* Technologie [114]. Das Grundprinzip dieser Technologie liegt in der hochintegrierten Kombination eines energiesparen-

[2] http://www.sparc.org
[3] http://www.apple.com
[4] http://www.ibm.com
[5] http://www.motorola.com
[6] http://www.arm.com

den und eines hochperformanten Prozessors. Beide Prozessoren können jeweils ein bis vier Rechenkerne besitzen [88, 89]. Bei weniger rechenintensiven Tasks wird der energiesparende und bei rechenintensiven Tasks der leistungsfähige Prozessor eingesetzt. Beim Samsung[7] Exynos 5 Octa *System-on-Chip* (SOC) können beispielsweise vier Cortex-A7 als *LITTLE* Prozessor und vier Cortex-A15 als *big* Prozessor eingesetzt werden. Srinivasan et al. [153] stellen ein Konzept vor, das über ARM's *big.LITTLE* Ansatz hinaus geht: In Phasen eines hohen Performanzbedarfs soll der Prozessor im *Out-of-Order* Modus betrieben werden.

In aktuellen Prozessoren werden unterschiedliche Sprungvorhersagekonzepte eingesetzt. Leider stellen die Hersteller hierzu wenig Informationen bereit, so dass sich die Ausführungen hier vor allem auf die Untersuchungen von Agner Fog beziehen [47]. Sie reichen von einfachen 2-Bit Prädiktoren mit relativ großen *Pattern History Table*s bis hin zu hybriden *Perceptron* Prädiktoren. Beim VIA Nano kommt ein hybrides Konzept zum Einsatz, das eine Mehrheitsentscheidung aus den kombinierten Prädiktoren trifft [78]. Außerdem zeigt sich ein Trend hin zu immer ausgefeilteren und auch aufwändigeren *Branch Target Buffer* (BTB) Lösungen, welche, wie beispielsweise beim Intel[8] Nehalem auch mehrstufig aufgebaut sein können [171]. Beim AMD[9] *Bulldozer* werden sie sogar von mehreren Prozessoren gemeinsam genutzt [96]. Auch der Einsatz von Schleifenzählern zur exakten Vorhersage der Schleifenwiederholungen ist beispielsweise beim Intel *Nehalem* zu finden. Andererseits ist in einigen Prozessoren, wie beispielsweise der Intel *Sandy Bridge*, ein Rückgang von komplexer Vorhersagelogik zu erkennen. Hierdurch soll die Pipeline Länge und damit auch die *Missprediction Penalty* (*Nehalem* 17 vs. *Sandy Bridge* 15) reduziert werden [47]. Bei aktuellen Intel Prozessoren, wie beispielsweise *Haswell*, *Broadwell* und *Skylake*, nimmt die Komplexität jedoch wieder zu. Es kann davon ausgegangen werden, dass mehrere Prädiktoren realisiert sind, die unterschiedliche Strategien verfolgen.

Intel *Atom* und *Silvermont* Prozessoren haben hingegen eine zweistufigen adaptive Vorhersage mit einem globalen *Pattern History Table* [97]. Die AMD K8 und K10 Prozessoren sind mit dem Instruktionscache verbun-

[7] http://www.samsung.com/semiconductor/products/exynos-solution/application-processor/EXYNOS-5-OCTA-5422
[8] www.intel.com
[9] www.amd.com

den [163]. Des weiteren kommt ein *Branch Target Buffer* zum Einsatz. Beim AMD *Bulldozer*, *Piledrive* und *Steamroller* hat die Sprungvorhersage keine Verbindung mehr zum Instruktionscache. Hier kommt der *Perceptron* Prädiktior als hybrider Prädiktor mit lokalem und globalem Prädiktor zum Einsatz [91]. AMD Bobcat und Jaguar benutzten zwei getrennte Arrays als *Branch Target Buffer*. Die *Missprediction Penalty* beträgt hier 13 Zyklen [29].

Die Performanz von Mehrkernprozessoren ist in starkem Maße von der Performanz des Cachesystems abhängig. Albonesi et al. [3] stellt *Selective Cache Ways* vor, in denen es die Möglichkeit gibt einzelne Cachewege in einem assoziativen Cache während Zeiten geringer Aktivität im Cache zu deaktivieren. Diese Arbeit ist auf die applikations-spezifische Anpassung einzelner Prozessoren fokussiert. In dieser Arbeit sollen einzelne deaktivierte Wege anderen Prozessoren in der Mehrkernarchitektur zur Verfügung gestellt werden können. Malik et al. stellen Wegemanagement in einer Cachearchitektur vor. Es wird auf der Basis von Steuerungsbits entschieden, ob die Wege für Daten oder Instruktionen genutzt werden sollen [105]. Dementsprechend können Teile des Cache als Daten- bzw. Instruktionscache genutzt werden. Tao et al. stellen einen interessanten Ansatz für zur Laufzeit rekonfigurierbare Caches vor. Die Cache Architektur wird im eigens dafür entwickelten Cachesimulator studiert [158]. Weitere Arbeiten von Gordon-Ross et al. [62] und Nowak et al. [116] zu konfigurierbaren Cachearchitekturen nutzen abstrakte Simulationsmodelle, die aus Hardwaresicht bezüglich einer akkuraten Realisierung nicht aussagekräftig sind. Nowak et al. zeigen erste Performanzvorteile und realisieren ihr Cachedesign in einem *PowerPC Field Programmable Gate Array* (FPGA) Makro [116]. Diese Cachearchitektur ist jedoch über den vergleichsweise langsamen peripheren Bus angebunden, was zu Performanzverlusten durch das Busprotokoll führt. In dieser Arbeit wird der Cache tief in die Mikroarchitektur integriert, so dass es möglich wird die Ressourcen zwischen den Prozessoren innerhalb einer Kachel der Mehrkernarchitektur zu partitionieren. Erst hierdurch wird ein Design mit einer vorhersagbaren dynamischen Ressourcennutzung möglich, das die Ziele eines einzelnen Prozessors erreicht und die Effizienz im Gesamtsystem erhöht. Diese Effizienz wird hauptsächlich durch das Verhalten der Applikation und deren Nutzung des Cache bestimmt. Es ist daher wichtig, dass die Applikationsentwickler, aber auch das Laufzeitsystem und der Compiler über Wissen bezüglich

der Prozessorarchitektur und ihrer dynamischen Fähigkeiten verfügen, um die unterschiedlichen Zielsetzungen, wie beispielsweise *Performanz pro Watt*, dynamisch erreichen zu können [61, 130]. Die Ansätze, die von Ji et al. [90] und Nicolaescu et al. [113] vorgestellt werden, passt den Cache vor dem Start der Applikation an die inneren Schleifen der Applikation an. Hierdurch können die Ressourcen der Cachearchitektur sehr effizient genutzt und durch die eingeführte Dynamik weiter verbessert werden. Auf den ersten Blick erscheint es sinnvoll immer die höchste Assoziativität, also nach Möglichkeit einen vollassoziativen Cache, zu nutzen, da die Performanz durch die erhöhte Parallelität gesteigert werden kann. Zhang et al. zeigen jedoch, dass sich dieser Vorteil nur bei bis zu 4- oder maximal 8-Wegen bestätigen lässt, da die Realisierungskosten danach mit größerer Assoziativität sehr stark ansteigen [172]. Damien unterstreicht dies mit den Studien seiner Dissertation [35]. Marty hat in seiner Dissertation gezeigt, dass Verdrängungsstrategien orthogonal zu allen anderen bisher angesprochenen Cachefunktionalitäten ist [107]. Daher liegt der Fokus in dieser Arbeit auf einer Gesamteffizienz für den adaptiven Cache und dessen dynamische Reallokation. Somit sollen die vorhandenen Ressourcen optimal genutzt werden und anhand der Anforderungen der Applikation parametrisiert werden.

Im Bereich der kommerziellen Prozessoren hat *Intel* im Jahr 2015 die *Cache Allocation Technology* (CAT) vorgestellt, die den *Last Level Cache* als gemeinsame verwaltbare Ressource verfügbar macht [87]. Diese ist speziell bei Mehrkernprozessoren mit mehreren gleichzeitigen Programmen von Vorteil und ermöglicht es auf Teile des *Last Level Cache* (LLC) entsprechend der *Class of Service* (COS) zuzugreifen.

3.2. Applikationsspezifische, adaptive und rekonfigurierbare Prozessoren

Aktuelle Entwicklungen im Bereich der Prozessorarchitekturen gehen in die Richtung der optimierten *Application-Specific Instruction-Set Processor* (ASIP) [98]. Durch diesen Ansatz, erweitert durch rekonfigurierbare Instruktionssatz- und Hardwareerweiterungen, kann eine höhere Perfor-

manz pro Leistung und Performanz pro Fläche erreicht werden [73]. Huynh et al. [83] geben im Jahr 2009 einen Überblick über erweiterbare adaptive Prozessoren. Galuzzi et al. [55] geben eine Beschreibung des Problems und einen Überblick über in der Wissenschaft vorhandene Lösungsansätze.

Chueng et al. [30] durchsuchen den möglichen Designraum für eingebettete Anwendungen und geben Hinweise auf effiziente Nutzung der Ressourcen. Ye et al. [168] haben *CHIMAERA* vorgestellt. Hierbei wird ein kleines und schnell rekonfigurierbares FPGA-ähnliches Modul in die Prozessorpipeline eingefügt. Dales stellt in [34] den *Proteus* Prozessor vor. Der rekonfigurierbare Ansatz erweitert den Prozessor um ein *tightly-coupled Fabric*. Lisecky et al. [103, 104] stellen den Warp Prozessor vor. Der Prozessor erkennt laufzeitkritische Aufgaben während der Ausführung. Anschließend wird mit Hilfe von *On-Chip Place & Route* (PAR) eine spezifische Logikrealisierung erstellt, deren Ergebnis zur Ausführung hinzugefügt wird. Eine transparente rekonfigurierbare Beschleunigung wird von Carro et al. [17] präsentiert. Bei diesem Ansatz wird eine eng gekoppelte grobgranulare Architektur für die Laufzeitsynthese genutzt. Hierbei wird im Gegensatz zum Ansatz von Lisecky et al. kein Compiler benötigt. Eine einfachere Laufzeitsynthese, die während der Ausführung der Anwendung durchgeführt wird, wird für die *Adaptive Microinstruction Driven Architecture* (AMIDAR) Klasse von Prozessoren vorgestellt [39]. Die Autoren unterstreichen den Einfluss des Ansatzes auf die Ausführungszeit. Profiling wird während der Laufzeit genutzt, um Kandidaten für die Beschleunigung zu identifizieren.

Die *Rotating Instruction Set Processing Platform* (RISPP), die von Bauer et al. [16] vorgestellt wird, führt neue Spezialinstruktionen in Verbindung mit einem Laufzeitsystem ein, die diese unterstützt. Jede Spezialinstruktion existiert in verschiedenen Alternativen, die von reiner Software bis hin zu verschiedenen Hardwareimplementierungen mit verschiedenen *Trade-Offs* reichen können. Diese Arbeit wurde von Grudnitsky et al. [64] aufgegriffen und um die Verfügbarkeit für alle Kerne einer Mehrkernarchitektur erweitert. Bonzini et al. [20] erstellen einen spezifischen Instruktionssatz für eingebettete Prozessoren auf Basis von grobgranular rekonfigurierbaren Prozessorarrays [6]. *Multi-purpose dynamically Reconfigurable Platform for intensive Heterogeneous processing* (MORPHEUS), das von Thoma et al. [162] vorgestellt wird, ermöglicht die adaptive dynamische Rekonfiguration verschiedener rekonfigurierbarer Architekturen. In diesem Pro-

jekt wurden flexible heterogene Plattformen verschiedener Größe für das *Hardware/Software Co-Design* entwickelt. Im Kontext des *KArlsruhe's Hypermorphic Reconfigurable-Instruction-Set Multi-grained-Array Processor* (KAHRISMA) Projektes haben König et al. [99] einen neuen Architekturansatz vorgestellt. Es enthält grob- und feingranulare, zur Laufzeit rekonfigurierbare, Prozessorarrays, die zur Beschleunigung komplexer Algorithmen verwendet werden können. Darüber hinaus können verschiedene Instruktionssätze auch parallel realisiert werden. Eine aktuelle Herausforderung liegt im Feld von *Dark Silicon*, da davon ausgegangen wird, dass die Energiedichte weiter ansteigt und nicht alle vorhandenen Ressourcen gleichzeitig genutzt werden können [82, 112, 146].

Es gibt bisher nur wenige vielversprechende Arbeiten, bei denen ein *Application-Specific Instruction-Set Processor* (ASIP) mit einer ADL wie beispielsweise *Language for Instruction-Set Architectures* (LISA) entwickelt wurde. Dodani et al. [40] haben einen kleinen Prozessor mit 18 Instruktionen und einer 3-stufigen Pipeline entwickelt. Nohl et al. [115] haben ein Prozessor zur Video Kompression entwickelt.

3.3. Mehr- und Vielkern Prozessorarchitekturen

3.3.1. *Invasive Computing* (InvasIC)

Der Sonderforschungsbereich *Invasive Computing* (InvasIC)[10] ist ein transregionales Verbundprojekt von drei Universitäten: Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Technische Universität München (TUM) und Karlsruher Institut für Technologie (KIT). In diesem Projekt stellt sich die Forschungsfrage wie Computer, insbesondere Mehrkern- und Vielkernarchitekturen mit 1000 oder mehr Kernen im Jahre 2020 aussehen, aus welchen Komponenten sie bestehen, wie sie aufgebaut sind, wie sie kommunizieren und nicht zuletzt auch wie sie programmiert werden [160, 161]. Das Verbundprojekt *Invasive Computing* (InvasIC) leistet dazu einen Beitrag, in dem es im Teilbereich A die grundlegenden Konzepte untersucht und dazu eine Sprache entwickelt, welche die neuen Vorgaben umsetzen bzw.

[10] http://www.invasic.de

weitergeben kann [159]. Der Teilbereich B widmet sich der invasiven Mehrkern Architektur [74]. Hier entsteht die adaptive applikations-spezifische Mikroarchitektur, die aus einer rekonfigurierbaren FPGA Fabric und den Erweiterungen der Prozessorarchitektur besteht, die in dieser Arbeit vorgestellt werden [75]. Außerdem gibt es massiv parallele *Tightly-coupled Processor Alrray*s (TCPAs), die ähnlich einer *Graphics Processing Unit* (GPU) aufgebaut sind [69]. Des Weiteren wird die Leistungseffizienz [144], vor allem im Kontext von *Dark Silicon* [145], sowie auch das Laufzeitmonitoring betrachtet [59, 60]. Eine weitere wichtige Komponente stellt das invasive *Network on Chip* (NoC) zur effizienten Kommunikation in der InvasIC Architektur dar [71]. Der Teilbereich C widmet sich dem Compiler, der approximativen Simulation [133] und der Laufzeit Unterstützung. Hier wird ein eigenes Laufzeitsystem, das *Invasive Run-Time Support System* (iRTSS) OctoPOS [117], eine simulative *Design Space Exploration*, sowie ein eigener Compiler mit Codegenerierung für die InvasIC Architektur entwickelt [21]. Ferner wird auch das Thema Sicherheit in der InvasIC Architektur untersucht [63]. Der Teilbereich D widmet sich den Anwendungen. Hier werden zum einen Anwendungen aus der Robotik verwendet, um die Echtzeitfähigkeit und die Garantien des invasiven Gesamtsystems zu demonstrieren [126]. Zum anderen kommen Anwendungen aus dem *High Performance Computing* Kontext zum Einsatz, um die Möglichkeit der Ausführung von sehr komplexen Anwendungen auf vergleichsweise einfachen eingebetteten *High Performance Computing* (HPC) Architekturen vor allem hinsichtlich der Skalierbarkeit zu unterstreichen [142]. Der Teilbereich Z kümmert sich um die administrativen Tätigkeiten und das Prototyping der neuen invasiven Architektur auf FPGA [18].

3.4. Architekturbeschreibungssprachen (ADL)

Die Entwicklung der *Architecture Description Language* (ADL) begann Anfang der 1990er Jahre. Vorhandene Konzepte waren nicht mehr ausreichend, um einen Prozessor umfassend zu beschreiben. Bis zu diesem Zeitpunkt war es üblich einen Compiler mit einer eigenen Sprache passend zum jeweiligen Instruktionssatz Architektur (ISA) zu entwickeln. Unabhängig davon wurde eine Implementierung in Hardware, oder später in einer *Hardware Description Language* (HDL), entwickelt.

Diese Sprachen werden in der Literatur auch als ADL bezeichnet und lassen sich in drei Kategorien unterteilen [140]:

- **Instruktionssatzspezifische Sprachen:** Diese Sprachen fokussieren sich auf den zu beschreibenden Instruktionssatz und repräsentieren dadurch die Sicht des Programmierers auf die Architektur. Diese Sprachen beschreiben daher hauptsächlich die Instruktionscodierung, die Assemblersyntax und das Verhalten. Beispiele für diese Kategorie von Sprachen sind ISDL [66] und nML [44].

  ISDL wurde am MIT entwickelt und fokussiert sich durch die Verhaltensbeschreibung auf die Architektur-Modellierung sowie die Simulator- und Hardwaregenerierung [66].

  nML wurde an der Technischen Universität Berlin entwickelt und ist auf die Beschreibung des Instruktionssatzes fokussiert [44]. Bei nML Projekten kann sowohl die Software, als auch die Hardware generiert werden. Basierend auf nML wurde ein C-Compiler von der Firma Target Compiler Technologies kommerzialisiert.

- **Architekturspezifische Sprachen:** Diese Sprachen fokussieren sich auf die Struktur der Architektur. Das Verhalten wird durch strukturale Beschreibungen abgebildet und bezieht sich auf die Sicht des Hardwaredesigners. Ein Beispiel ist die Sprache MIMOLA [15].

  MIMOLA wurde an der Universität Dortmund entwickelt und fokussiert sich auf eine hardwarenahe netzlistenähnliche Beschreibung [15]. Es können ein C-Compiler und ein Instruktionssatzsimulator generiert werden.

- **Gemischte instruktionssatzspezifische und architekturspezifische Sprachen:** Dieser letzte Typ von Sprachen bringt die beiden vorher ausgeführten Konzepte, die Struktur und das Verhalten zu beschreiben, in einer ADL zusammen. Beispiele hierfür sind EXPRESSION [67] und die in der Arbeit verwendete Sprache LISA [81]. Diese ADLs sind besonders für den kompletten Designablauf im ASIP-Design geeignet, da sie wie in Abschnitt 2.7.1 beschrieben, alle Bereiche von der *Design Space Exploration* (DSE) über die Werkzeuggenerierung bis hin zur Architekturimplementierung abdecken.

EXPRESSION wurde an der University of California in Irvine entwickelt und beinhaltet die strukturale Sicht sowie auch den Instruktionssatz. Das Hauptziel der Entwicklung war einen zyklenakkuraten Simulator und die dazu gehörige Werkzeugkette zu generieren [67].

LISA wurde an der Rheinisch-Westfälischen Technische Hochschule (RWTH) Aachen entwickelt und vereint die beiden Sichtweisen von Struktur und Verhalten. Mit LISA können somit zyklenakkurate Modelle erstellt werden, aus denen die Werkzeugkette, der Simulator, sowie auch die Hardware generiert werden können [81]. Eine ausführlichere Einführung in LISA findet sich in Abschnitt 2.7.1. LISA wurde als LISAtek aus gegründet und von CoWare kommerzialisiert. 2010 wurde CoWare von Synopsys akquiriert.

Eine weitere ADL wurde aktuell im Rahmen des KAHRISMA[11] Projektes entwickelt [99]. Im Vordergrund stehen hier mehrere Instruktionssets passend zur jeweiligen rekonfigurierbaren Architektur [154]. Der aktuelle Stand der Entwicklung ist das Software-*Framework*, wobei es in einer späteren Entwicklungsphase um die Hardwaregenerierung erweitert werden soll [155]. Im EU Projekt ALMA[12] wurde die Entwicklung weiter geführt, um aus einer abstrakten mathematischen Problembeschreibung, wie beispielsweise SciLab[13], den parallelen Code generieren zu können [24]. Dieser Ansatz wurde im Spin-Off emmtrix Technologies[14] kommerzialisiert.

Weiterführende Informationen zum Thema ADL finden sich in den Büchern [81, 110, 140].

Aufgrund der vielfältigen Möglichkeiten beim verzahnten High-Level Entwurf und der von der Firma Synopsys angebotenen Software, wird in der vorliegenden Arbeit die ADL LISA verwendet. Insbesondere der Ausblick auf LISA 3.0 für adaptive und rekonfigurierbare Prozessoren, sowie auch die Mehrkern Architekturen und parallele Programmierung, lassen LISA sehr interessant erscheinen.

[11] http://www.kahrisma.de
[12] http://www.alma-project.eu
[13] http://www.scilab.org
[14] http://www.emmtrix.com

4. *a*-Core: Die adaptive Prozessorarchitektur

In diesem Kapitel wird das Konzept der adaptiven Prozessorarchitektur vorgestellt. Nach der Fokussierung auf die Verlagerung von bisherigen Designentscheidungen in die Laufzeit und die entsprechend notwendigen Befehlssatzerweiterungen, wird schließlich die zu erwartende Effizienzsteigerung des adaptiven Prozessors (*a*-Cores) diskutiert. Dieses Konzept wurde auf dem Symposium für industrielle eingebettete Systeme (SIES) mit dem *Best Work in Progress* (WIP) *Paper Award* ausgezeichnet [TTHB12b].

4.1. Design des adaptiven Prozessors

Das Konzept der vorliegenden Arbeit legt den Fokus auf die dynamische Anpassung des Prozessors an die Anforderungen der jeweiligen Anwendung. Es wird bewusst von einem adaptiven Prozessor gesprochen, da die adaptiven Erweiterungen über die reine prozessorinterne Mikroarchitektur (μArch) hinaus gehen. Jedoch sollen alle Erweiterungen durch die eindeutige Instruktionssatz Architektur (ISA) repräsentiert werden, damit sie zur Laufzeit selbst durch die Hardware, aber auch von außen durch die Anwendung oder das Laufzeitsystem, gesteuert werden können. Dadurch werden alle Veränderungen, die durch gleichen Prozesse hervorgerufen werden, durch die Verarbeitung in der Prozessor Pipeline gesteuert.

Abbildung 4.1 zeigt die Designalternativen des zu entwerfenden adaptiver Prozessors (*a*-Cores). Es zeigt zum einen die gegebene prozessorinterne Instruktionssatz Architektur (ISA), sowie aber auch die Instruktionssat-

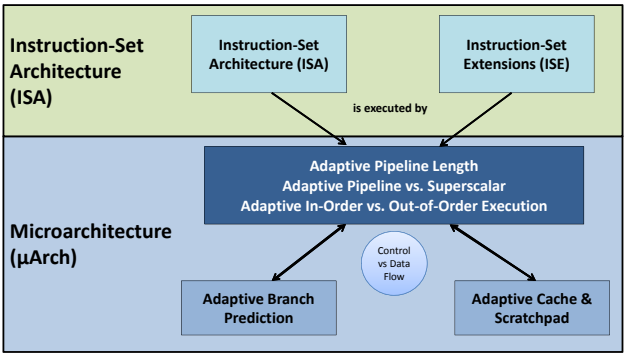


Abbildung 4.1.: Konzept der adaptiven Prozessor Architektur.

zerweiterungen (ISE). Die Instruktionssatzerweiterungen (ISE) gibt die Möglichkeit den *a*-Cores, als Erweiterung der bestehenden Instruktionssatz Architektur (ISA), während der Laufzeit zu adaptieren. Somit kann sowohl der Prozessor selbst, als auch die auf dem Prozessor ausgeführte Software, die Einstellungen zu Laufzeit verändern. Wie in Abbildung 4.1 zu erkennen ist, besteht der *a*-Core aus einer Mikroarchitektur (μArch), die an die jeweilige Anwendung angepasst werden kann. Basierend auf einer einfachen *Branch Prediction* (BP) kann diese dynamisch verändert bis hin zu Konzepten skaliert werden. Dies wird in aktuellen kommerziellen Prozessoren angewendet, falls diese Performanzsteigerung gewünscht ist. Darüber hinaus liegt der Fokus auf der Prozessor Pipeline. Je nachdem ob Kontroll- oder Datenfluss dominante Anwendungen ausgeführt werden, kann diese beispielsweise in der Länge verändert werden und effizient an die jeweiligen Anforderungen angepasst werden.

Des Weiteren kann die Skalarität der Pipeline verändert werden. Eine flexible adaptive superskalare Pipeline entsteht, wenn dynamisch weitere parallele Pipelines mit diversen Ausführungseinheiten hinzugefügt werden. Zur effizienteren Auslastung der superskalaren Pipeline, mit vielen unterschiedlichen Funktionseinheiten, macht es außerdem Sinn dynamisch zwischen *In-Order* (IO) und *Out-of-Order* (OoO) Abarbeitung in der jeweiligen Pipeline umschalten zu können. Hier ist eine signifikante Effizienzsteigerung zu erwarten. Es muss jedoch beachtet werden, dass der

Realisierungsaufwand steigen wird. Auf eine effiziente Ausnutzung der vorhandenen Ressourcen muss dementsprechend geachtet werden. Außerhalb der μArch soll der Cache als weiterer signifikanter Teil des a-Core betrachtet werden. Insbesondere im *Level 1* (L1) Cache werden alle Daten zwischengespeichert, während diese in der Prozessor Pipeline verarbeitet werden. Sobald die Daten größer als der vorhandene Speicherplatz im L1 Cache sind, werden diese verdrängt, was zu einer erhöhten Auslastung auf dem Bus und in höheren Ebenen in der Speicherarchitektur führt. Insbesondere in Mehr und Vielkern Architekturen führt diese erhöhte Auslastung pro Kern zu einer signifikant höheren Auslastung der Busarchitektur. Alle Kerne einer Kachel können meist nur sequentiell auf den lokalen Bus zugreifen, um die Daten mit den höheren Ebenen der Speicherarchitektur, dem *Level 2* (L2) Cache und über das *Network on Chip* (NoC), mit dem entfernten *Double Data Rate* (DDR) Speicher zu synchronisieren. Hier werden in der Arbeit drei Mechanismen zur Effizienzsteigerung betrachtet:

1. Die Eingangsdaten der Applikation und insbesondere die Anordnung der zu verarbeitenden Daten soll an die Parametrisierung des Caches angepasst werden. Alternativ kann hier auch eine Parametrisierung des Caches gewählt werden, die effizienter zu der Datenverarbeitung der jeweiligen Applikation passt.
2. Die bisher zur Designzeit gewählten Parameter des Caches sollen beim a-Core zur Laufzeit veränderbar sein. Dies hat den Vorteil, dass beispielsweise eine erhöhte Assoziativität zur Effizienzsteigerung genutzt werden kann, während bei einer anderen Applikation keine weitere Effizienzsteigerung erwartet wird.
3. Die vorhanden Ressourcen der *Level 1* (L1) Caches sollen effizienter genutzt werden als bisher üblich. Im Gegensatz zum bisher statischen L1 Cache Design sollen die vorhandenen Speicherressourcen von allen a-Cores auf einer Kachel im verteilten Vielkern System genutzt werden können.
4. Diese Parameter sollen zum Erreichen der jeweiligen Optimierungsziele, sowohl von der Software als auch von der Hardware, zur Laufzeit adaptiert werden können. Hier ist es insbesondere wichtig, dass die Optimierungen der Hardware transparent für die Applikation ablaufen. Das Laufzeitsystem muss jedoch zu jedem Zeitpunkt den aktuellen Systemzustand abfragen können, damit die globale Opti-

mierung aus Sicht des Systems nicht durch lokale Optimierungen beispielsweise durch den Prozessor selbst verschlechtert wird.

Das vorgestellte Konzept wurde auch für einen leistungseffizienten *Application-Specific Instruction-Set Processor* (ASIP) Co-Prozessor für Lokalisierungsanwendungen [52, 51, 50] zum adaptiven Lokalisierungsprozessor (SmartLoCore) erweitert [TGB+14].

4.2. Vorteile des adaptiven Prozessors

Im Gegensatz zu anderen Ansätzen in der Literatur, wird in dieser Arbeit ein zyklenakkurates Prozessormodell für den *a*-Core genutzt. Dadurch können die Erweiterungen auch in das generierte *Hardware Description Language* (HDL) Modell übertragen werden, ohne dass Veränderungen oder Erweiterungen vorgenommen werden müssen. Damit unterscheidet sich diese Arbeit wesentlich zu vorangegangenen Arbeiten, die adaptive Prozessoren rein simulativ, und daher meist nur verhaltensbasiert oder zyklenapproximiert, betrachtet haben. In der Realität zeigt sich speziell bei komplexen Prozessoren, dass die simulative Betrachtung nicht ausreichend ist, da die vorhandenen Ressourcen meist nicht gleichmäßig auf der Zielarchitektur angeordnet werden können und somit deutliche Effizienzunterschiede zwischen der Simulation und der synthetisierten Realisierung aufweisen. Es kann in der vorliegenden Arbeit nicht nur eine Abschätzung, sondern eine genaue Aussage, über die zu erwartende Effizienzsteigerung des *a*-Core und seiner einzelnen adaptiven Eigenschaften, getroffen werden. In dieser Arbeit kommt außerdem das eigens hierfür entwickelte Prozessormodell zum Einsatz. Es ist daher möglich alle Erweiterungen tief in der μArch zu verankern und somit die maximale Effizienz bei der Implementierung zu erreichen, da keine Abstraktionsebenen dazwischen liegen. Konkret ist es so möglich, neben der Erweiterung der Pipeline, auch erstmalig den *Level 1* (L1) Cache zu betrachten, da in den vorangegangenen Arbeiten meist nur der *Last Level Cache* (LLC) betrachtet wurde [5, 102, 164, 68, 85]. Da aus dieser sehr genauen Modellierung auch direkt die HDL Beschreibung generiert wird, ist es möglich, neben der reinen Performanz, eine Aussage über den Ressourcenverbrauch zu treffen. Hierdurch kann die Aussage bezüglich der Effizienz über die absolute

Performanz hinaus auch in eine relative Bewertung, insbesondere unter Einfluss der vorhandenen Ressourcen, getroffen werden. Das Ziel der Arbeit ist es die Konzeption und die Entwicklung immer so generisch wie möglich zu gestalten. Zur Realisierung und Evaluierung der Ansätze werden *Field Programmable Gate Array*s (FPGAs) zum Einsatz kommen. Jedoch sollen keine FPGA-spezifischen Eigenschaften, wie beispielsweise die dynamische Rekonfiguration, genutzt werden. Somit ist es möglich alle Konzepte, die im Folgenden vorgestellt werden, auch auf einen *Application Specific Integrated Circuit* (ASIC) zu übertragen. Des Weiteren ist es daher besonders interessant unterschiedliche Realisierungsstrategien, wie beispielsweise *Power, Performance, Area* (PPA), gegeneinander abzuwägen.

Auf Basis der vorangegangenen Überlegungen ist es möglich verschiedene Designentscheidungen für den *a*-Core zu finden, zwischen denen zur Laufzeit umgeschaltet werden kann. Es ist hier sehr wahrscheinlich, dass ein einzelner Registerzugriff wesentlich günstiger realisiert werden kann als ein Zugriff auf den Speicher oder die weitere Hierarchie. Dies ist besonders interessant, da es jetzt mit dem *a*-Core möglich ist Designentscheidungen, die bisher ausschließlich zur Entwurfszeit getroffen wurden, mit dem Konzept dieser Arbeit auch zur Laufzeit zu treffen. Insgesamt sollen alle adaptiven Mechanismen dazu beitragen, dass zum einen der Prozessor weniger oft angehalten wird, da beispielsweise die Daten im Cache sinnvoll angeordnet vorliegen und am Stück geladen werden können, so dass die Verdrängung wesentlich geringer ist. Zum anderen, dass die Pipeline weniger leer läuft bzw. die Auslastung der Pipeline erhöht wird, da die Ausführung weniger häufig unterbrochen wird, weil die adaptive Sprungvorhersage deutlich höhere Trefferquote hat.

Als weiteren interessanten Punkt hat sich im Laufe der Arbeit die Nutzung der bereits angesprochenen verteilten Kachelarchitektur entwickelt, die sowohl über gemeinsam genutzten als auch individuell geteilten Speicher verfügt. Hierdurch ist es möglich, über die reine Hardwareentwicklung hinaus, mit Applikations-, Laufzeitsystem- und Compilerentwicklern Szenarien zur Ausnutzung der Laufzeitfähigkeiten des *a*-Core zu evaluieren. So können Vorteile und Optimierungen in der Entwicklung des Programmes konsequent über alle Entwicklungs- und Abstraktionsebenen bis an die Hardware weitergegeben werden. Umgekehrt ist auch das System zu jedem Zeitpunkt über den jeweiligen Zustand jedes *a*-Cores informiert. Diese Verzahnung verschiebt die Entwicklung des *Hardware/Software Co-Design*

(HSC) von der Designzeit in die Laufzeit und eröffnet neue Möglichkeiten zur Effizienzsteigerung durch den *a*-Core. Hierdurch kann außerdem sichergestellt werden, dass keine einseitige Optimierung durch den Prozessor selbst zu einem lokalen Optimum stattfindet, sondern das globale Optimum aus Systemsicht immer erreicht wird.

4.3. Realisierung des adaptiven Prozessors

In diesem Kapitel wird die Realisierung des *a*-Cores Konzeptes vorgestellt. Hierzu wird ein Entwicklungsfluss auf Basis der *Architecture Description Language* (ADL) *Language for Instruction-Set Architectures* (LISA) und den Werkzeugen *Processor Designer* (PD) und *Platform Architect* (PA) der Firma Synopsys angewendet. Ein besonderer Fokus liegt auf der zyklenakkuraten Modellierung der adaptiven Erweiterungen des Prozessor Modells nach modernsten verzahnten *Hardware/Software Co-Design* (HSC) Designmethoden, wobei einzelne Designentscheidungen aufgrund der durch die Adaptivität neu gewonnenen Flexibilität von der Designzeit in die Laufzeit verlagert werden können. Somit können einzelne Entscheidungen, die aus der *Design Space Exploration* (DSE) des adaptiven Prozessor Modells resultieren, zu einem späteren Zeitpunkt im Prozessor umgesetzt werden, ohne sich für einen mittleren Anwendungsfall zur semieffizienten Ausnutzung der Eigenschaften des Prozessors entscheiden zu müssen.

Weiterführend soll nicht nur der komplette Prozessor, sondern auch die Mehr- und Vielkern Architektur inklusive der Caches, Bussen und möglicher Co-Prozessoren zyklenakkurat modelliert werden. Zur Verkürzung einzelner Testläufe können nach erfolgter Verifikation auch einzelne Teile der Architektur kompiliert, approximiert oder auch nur verhaltensakkurat ausgeführt werden, um die Entwicklung und den Test des *a*-Cores zu beschleunigen. Zur Evaluation muss die gesamte Architektur jedoch wieder in den zyklenakkuraten Modus versetzt werden, um Verfälschungen der Ergebnisse im Vergleich zu einer HDL-Implementierung zu vermeiden, da hier sonst keine zeitlichen Effekte, weder in der Speicherhierarchie noch im Prozessor beachtet würden. Dies ist jedoch besonders wichtig, da der *a*-Core besonders von der Optimierung der Pipeline Mechanismen, insbesondere in der Interaktion mit der Speicherhierarchie, profitiert. Somit

ist nicht nur der *a*-Core, sondern auch die gesamte Architektur, mit dem Prototyping auf einer FPGA Zielarchitektur vergleichbar. Zusätzlich zu der dazu notwendigen HDL Beschreibung, werden auch alle Werkzeuge, wie beispielsweise der C-Compiler, der Assembler, oder auch der Simulator, für den Entwicklungsfluss aus dem zyklenakkuraten Prozessormodell generiert. Beispielsweise wird hierdurch auch eine Co-Simulation mit anderen, sich in der Entwicklung befindenden, Komponenten der Architektur in einem sehr frühen Stadium der Entwicklung möglich, was die Entwicklungszeit, die Kosten und auch die *Time to Market* (TtM) wesentlich verkürzt. Hierbei kann anhand einzelner Testprogramme sichergestellt werden, dass alle Komponenten und Erweiterungen den Anforderungen entsprechen und effizient realisiert sind. Hierdurch kann beispielsweise die Latenz und die *Cycles per Instruction* (CpI) reduziert und somit der Durchsatz im Prozessor erhöht werden. Des Weiteren kann sichergestellt werden, dass der Prozessor, wie das Ausgangsmodell auch, bei der gleichen Zielfrequenz betrieben werden kann. Somit kommt es hier nicht zu unnötigen Einbußen bei der Performanz. Durch diesen ganzheitlichen Ansatz kann sichergestellt werden, dass das Verhalten auf der einen Seite, aber auch die Kompatibilität und die Genauigkeit auf der anderen Seite, garantiert sind. Dies ist aus drei Gründen sinnvoll:

- Veränderungen an der ISA, und damit am gesamten Modell, können einfach umgesetzt und iteriert werden.
- Diese Veränderungen können einfach verifiziert und evaluiert werden, da immer wieder der gleiche Entwicklungsfluss zum Einsatz kommt und alle notwendigen Werkzeug daraus generiert werden.
- Diese Weiterentwicklungen lassen sich einfach an das gesamte Mehr-/ Vielkern System weitergeben, so dass der adaptive Prozessor mit relativ geringer Entwicklungszeit im gesamten Systemzusammenhang betrachtet werden kann.

Vor allem soll nicht nur der gesamte in der *Scalable Processor ARChitecture* (SPARC) V8 Spezifikation spezifizierte Instruktionssatz, sondern auch die von Gaisler Research implementierten Erweiterungen des LEON3 Prozessors, wie beispielsweise Caches, Busse und Hardwarebeschleuniger, modelliert werden. Anstatt einer reinen Implementierung des Instruktionssatzes, was einem *Language for Instruction-Set Architectures* (LISA) Verhaltensmodell zur Simulation des Instruktionssatzes entspricht, soll an

dieser Stelle ein besonderes Augenmerk auf die Modellierung des *a*-Core als zyklenakkurates Modell gelegt werden. Damit soll sichergestellt werden, dass die im weiteren Verlauf dieser Arbeit vorgenommenen Erweiterungen aussagekräftig sind.

Für die Implementierung soll nicht nur der Instruktionssatz, sondern auch die Assemblersyntax mit der entsprechenden Kodierung der Instruktionen modelliert werden. Durch diesen aufwändigeren Ansatz in der Modellierung wird ein umfassendes LISA-Modell des *a*-Cores entstehen, das es ermöglicht alle im *Processor Designer* (PD) vorhandenen Werkzeuge, wie C-Compiler, Assembler und eine Hardwarebeschreibung (HDL), aus dieser einzigen Prozessormodell-Beschreibung zu generieren. Im Weiteren wird bei der Implementierung darauf geachtet, dass das LISA-Modell kompatibel zu der vorhandenen Gaisler-Toolchain bleibt. Somit wird sichergestellt, dass zum einen der aus LISA generierte Compiler, der Assembler und der Linker, aber zum anderen auch der Gaisler Research C-Compiler und damit verifizierter, von Gaisler Research zur Verfügung gestellter Code, verwendet werden kann.

Eine umfangreiche Verifikationsphase soll bereits während der LISA-Modellierung sicherstellen, dass die modellierten Funktionalitäten richtig abgebildet werden. In einer daran anschließenden Testphase soll insbesondere auf die Genauigkeit des Modells, in Bezug auf die Anzahl der Zyklen pro Instruktion bzw. pro Programm zum LEON3, geachtet werden. Durch diesen zweistufigen Ansatz wird zum einen das Verhalten, und zum anderen vor allem die Kompatibilität und Genauigkeit, des zyklenakkuraten Modells bei der Abarbeitung komplexerer Programme sichergestellt. Zur Evaluation, der im Folgenden vorgestellten Erweiterungen, werden Benchmarks aus der MiBench-Suite verwendet [65].

4.4. Adaptive Pipeline

In diesem Abschnitt wird das Konzept der adaptiven Prozessor Pipeline vorgestellt. Es werden Instruktionssatz Erweiterungen vorgestellt, durch die die Pipeline zur Laufzeit verlängerbar oder verkürzbar sein wird. Anschließend wird die Realisierung dieses Konzepts in der *Architecture Description Language* (ADL) *Language for Instruction-Set Architectures* (LISA) erläu-

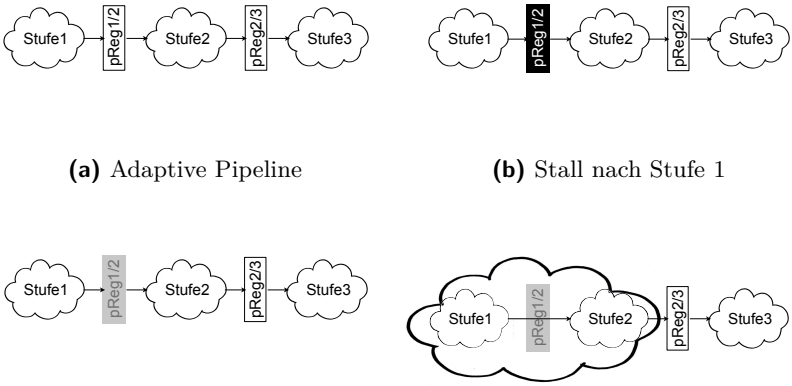
tert. Basierend auf dieser LISA-Modellierung wird das Konzept evaluiert und bewertet. Die Methodik der adaptiven Pipeline wurde auf der *Conference on Reconfigurable Computing and FPGAs* (ReConFig) [HTG+11] und der *Reconfigurable Architectures Workshop* (RAW) [TTHB12a] präsentiert.

4.4.1. Design der adaptiven Pipeline

Die Grundidee besteht darin die Länge der Prozessor Pipeline zur Laufzeit applikationsspezifisch anzupassen. Durch diese Variation der Tiefe der Pipeline soll die Effizienz, je nach vorhandener Arbeitslast, gesteigert werden. Das Neue an diesem Ansatz ist, dass das laufzeitadaptive Umschalten der Pipeline auf Basis der Anforderungen der Software sowohl durch die Hardware, als auch die Software, durchgeführt werden kann.

Efthymiou & Garside [41, 42] haben in ihren Arbeiten diese Effekte bereits am Beispiel einer asynchronen Pipeline untersucht. In [41] wird der Ansatz im Kontext von Leistungsmanagement diskutiert. Hierbei wird die Arbeitslast der Pipeline analysiert. Je nach Bedarf können benachbarte Pipelinestufen fusioniert werden, um so eine Energiereduktion zu erhalten. In [42] wird die vorher genannte Arbeit um den Aspekt der Kontrollflussspekulation erweitert. Sequentielle Ausführungen lasten den Prozessor sehr gut aus. Ist diese Sequenz jedoch unterbrochen, führt dies zum Leerlauf in der Pipeline, welcher keine für die Abarbeitung notwendige Energie verbraucht. Durch die Methode des dynamischen Verbindens und Aufspaltens der Pipelinestufen, kann dieser Effekt minimiert werden, was zu geringerem Energieverbrauch im Gesamtsystem führt.

In der vorliegenden Arbeit ist der Ansatz der asynchronen Pipeline und die Diskussion von Latches jedoch nicht ausreichend. Die Pipeline ist beim LEON3 als synchron siebenstufig definiert. Dies führt zu Begrenzungen der Flexibilität bei der Verwendung des Ansatzes. Das Konzept für diese Arbeit richtet das Hauptaugenmerk somit auf die Verkürzung und Verlängerung der Pipeline. Sollen, wie in Abbildung 4.2a aufgezeigt, zwei benachbarte Pipelinestufen fusioniert werden, ist eine zusätzliche Instruktion notwendig, die vom Compiler oder Laufzeitsystem in den vorhandenen Programmcode eingefügt werden muss. Des Weiteren muss, wie in Abbildung 4.2b dargestellt, die erste Stufe angehalten werden, damit die in der zweiten



(a) Adaptive Pipeline

(b) Stall nach Stufe 1

(c) Transparenz von Pipelineregister 1/2

(d) Verkürzte adaptive Pipeline

Abbildung 4.2.: Adaptieren der Pipeline.

Stufe vorhandenen Daten fertig abgearbeitet werden können. Erst wenn diese Verarbeitung abgeschlossen ist, darf das synchrone Pipelineregister zwischen den Stufen transparent geschaltet werden, siehe Abbildung 4.2c. Transparent bedeutet in diesem Zusammenhang das Umgehen des Registers, um die Daten, wenn die Pipeline Stufen verbunden wurden, ohne Taktflanke direkt weiter an die nächste Stufe übergeben zu können. Die Charakterisierung der Abarbeitung in der Pipeline kann anhand von Token erfolgen. Jeder Instruktion wird ein *Token* zugewiesen, das in jedem Taktzyklus eine Pipelinestufe belegt. In der LEON3 Pipeline sind maximal sieben *Token* vorhanden. Wird die Pipeline verkürzt, fällt für jede verbundene Pipeline Stufe ein Token weg. Dieses muss mit Hilfe des *Stalls* der ersten Pipelinestufe aus der Pipeline herausfallen, bevor die Verbindung der Pipeline Stufen erfolgen darf (vgl. Abbildung 4.2c). Bei der Verlängerung der Pipeline muss dementsprechend ein zusätzliches *Token* eingefügt werden. Das Verbinden von zwei benachbarten Pipelinestufen benötigt demzufolge zwei zusätzliche Zyklen für die zusätzlich einzufügenden Instruktionen und dem einmaligen Anhalten der Pipeline, um das Token aus der Pipe-

line fallen zu lassen. Eine Optimierung ist möglich, da die Diskussion, vor allem des Kontrollflusses, auf Basisblockebene erfolgt. Somit kann beispielsweise ein *Delay Slot*, der ansonsten nur mit einem *No Operation* (NOP) belegt sein würde, für die Instruktion genutzt werden. Analog gilt diese Darstellung auch für das Aufspalten verbundener Pipelinestufen. Es wird eine zusätzliche Instruktion zum Aufspalten benötigt. Ein Anhalten der Pipeline ist nicht notwendig. Das mit der Verlängerung der Pipeline zusätzlich vorhandene Token kann mit der nächsten Instruktion eingefügt werden. Analog ist auch die Compileroptimierung durch das allgemeine Einfügen der Instruktion an Stelle eines NOP Befehls denkbar.

Abschließend lässt sich zusammenfassen, dass das Verbinden und das Aufspalten der Pipelinestufen einmalig drei Zyklen benötigt, mit der vorgestellten Optimierung jedoch nur einen Zyklus. Das Verkürzen der Pipeline um eine Stufe führt dazu, dass jede Instruktion einen Zyklus weniger zum Durchlaufen der Pipeline benötigt. Somit hebt sich der Vorteil der kurzen Pipeline aus ganzheitlicher Sicht, bei immer komplett gefüllter Pipeline, bereits durch das zusätzliche Einfügen der Instruktionen auf. Wichtig ist jedoch anzumerken, dass bei einem Bruch mit der sequentiellen Ausführung auf Basisblockebene, durch Sprünge, Verzweigungen und Funktionsaufrufe, Leerlauf in der Pipeline entsteht. Speziell dieser durch den Kontrollfluss hervorgerufene Pipeline Leerlauf soll mit Hilfe des hier vorgestellten Ansatzes minimiert werden. Dieses Konzept hebt sich somit auch von dem häufig angewendeten und mit weniger Aufwand zu realisierenden Konzept der Sprungvorhersagen ab [100, 108, 149]. Wie der Name schon impliziert, findet dieses Konzept nur bei Verzweigungen Anwendung. Bei *Jumps* und *Calls* kann dieses Konzept nicht greifen, da die Sprungadressen in einer früheren Pipelinestufe noch nicht bekannt sind. Daher kann das Konzept der Pipeline Fusion durchaus als ganzheitliche Erweiterung der vorhandenen Konzepte gesehen werden. Die daraus resultierenden Vor- und Nachteile sollen in dieser Arbeit anhand der erstellten LISA-Modellierung evaluiert werden. Eine vertiefende Diskussion hierzu findet sich im folgenden Abschnitt.

4.4.2. Realisierung der adaptiven Pipeline

In diesem Kapitel wird auf die Realisierung mit besonderem Augenmerk auf die Implementierung des LISA-Modells eingegangen. Zunächst wird der LEON3 Prozessor als LISA-Modell erstellt. Anschließend wird dieses Modell um die, in Unterabschnitt 4.4.1 beschriebenen, konzeptionellen Ansätze erweitert und innerhalb des LISA-Modells evaluiert.

4.4.2.1. Erweiterung des Instruktionssatzes

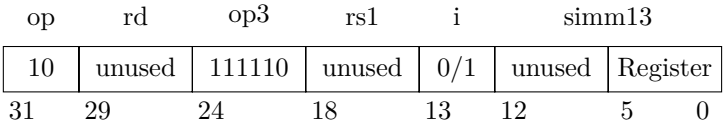


Abbildung 4.3.: Instruktionssatzerweiterungen auf Basis der SPARC V8 ISA mit der die Pipeline Stufen verbunden (i=0) bzw. aufgespalten (i=1) werden können.

Wie in der Konzeption in Unterabschnitt 4.4.1 bereits beschrieben, wird zum Verbinden und zum Aufbrechen von Pipelinestufen jeweils eine neue Instruktion benötigt. Diese Instruktionen müssen dem SPARC Instruktionssatz hinzugefügt werden. Hierzu wird eine freie Instruktion aus dem Format 3 gewählt und so modifiziert, dass eine optimale Implementierung hinsichtlich der Ressourcenminimalität gewährleistet ist. Op3 wird zu $0x3E$ festgelegt. Da insgesamt die beiden Instruktionen Verbinden und Aufspalten benötigt werden, wird das in der SPARC Architektur vorhandene Bit i gewählt, um zwischen Verbinden und Aufspalten zu unterscheiden. Ist i gleich null, handelt es sich um eine Verbinden Instruktion. Ist i gleich eins, handelt es sich um eine Aufspalten Instruktion. Zur weiteren Unterscheidung, um welche Pipeline Register es sich handelt, werden die sechs niederwertigsten Bits der Instruktion gewählt. Jedes Pipeline Register wird von einem Bit repräsentiert. Das erste Pipelineregister (*Instruction Fetch* (FE)/*Decode* (DE)) ist Bit null zugeordnet. Dies geht analog weiter bis hin zu Bit fünf für die letzte Pipelinestufe (*Exception* (XC)/*Write Back*

(WB)) der SPARC Pipeline. Abbildung 4.3 zeigt diese SPARC kompatible Instruktion.

4.4.2.2. Realisierung der Laufzeitadaptivität

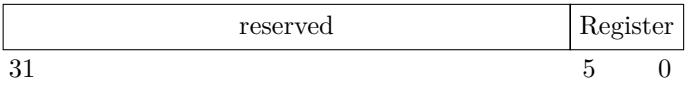


Abbildung 4.4.: Das *Adaptive Configuration Register* das den Zustand der adaptiven Pipeline in den sechs *Least Significant Byte* (LSB) repräsentiert.

Es ist wichtig zur Laufzeit einen Überblick über den Status der Pipeline zu haben. Deswegen wird jede Zustandsänderung durch die zusätzlichen Instruktionen im *Adaptive Configuration Register* (ACR), das analog zu den *Ancillary State Register* (ASR) der SPARC Architektur angelegt ist, abgelegt. Die Vorgehensweise ist analog zu der bei der Instruktion, wo auch jede Pipeline Stufe durch ein Bit repräsentiert wird. Werden zwei Pipelinestufen verbunden, also ein Pipelineregister transparent geschaltet, wird das korrespondierende Bit zu eins gesetzt. Das ACR ist in Abbildung 4.4 gezeigt. Wird eine Verbindung aufgespalten, wird das entsprechende Bit wieder auf null gesetzt. Gleichzeitig sorgt die Instruktion auch für das nötige Anhalten der Pipeline vor dem synchronen Transparentschalten.

Das Transparentschalten lässt sich in der LISA-Modellierung nicht eins zu eins aus der Konzeption übertragen, wie es die physikalische Repräsentation impliziert. Es wurden im Laufe der Arbeit verschiedene Ansätze der Implementierung untersucht, auf die, vor allem auf deren Problematik bei der Umsetzung, im Folgenden kurz eingegangen wird. Ein direktes Umschalten zur Transparenz der Pipelineregister stellt sich schnell als unmöglich heraus, da im LISA Code kein Zugriff auf den Taktzustand der Pipelineregister möglich ist. Ein weiterer Ansatz ist für jeden Zustand der Pipeline eine dezidierte verkürzte Pipeline zu erstellen, zu synchronisieren und zwischen diesen Pipelines zur Laufzeit umzuschalten. Es ist zwar möglich in einem LISA-Modell mehrere Pipelines zu erstellen, allerdings ist es nicht möglich das gleiche Instruktionsset auf mehrere Pipelines abzubilden, da nur ein *Coding Root* definiert werden kann. Auch das Umschalten zwischen Pipelines kann daher nicht in LISA realisiert werden. Auf Grund

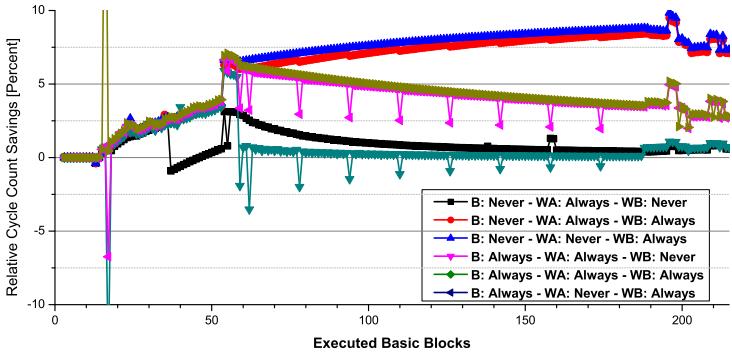


Abbildung 4.5.: Adaptive Pipeline: Cache & Sprungvorhersage Exploration - Relative Zyklenersparnis.

dieser Begrenzungen wird die Implementierung des Konzeptes mit nur einer Pipeline durchgeführt. Als Ausgangspunkt dient die, im vorigen Abschnitt vorgestellte, LEON3 Prozessor Pipeline.

4.4.3. Evaluation anhand der adaptiven Pipeline

Es wird der kürzeste Weg - ein Dijkstra-Algorithmus - mit den optimalen vorher gefundenen Parametern ausgeführt. Das Ergebnis unterstreicht die Wirkung der adaptiven Pipeline.

4.4.3.1. Relative Evaluation mit Quicksort & Dijkstra

Abbildung 4.5 verdeutlicht den Verlauf der relativen Ersparnis der adaptiven Pipeline bezogen auf die jeweilige Konfiguration des *a*-Core. In dieser Grafik ist die relative Ersparnis in Prozent über der zeitlichen Abarbeitung in Form von abgearbeiteten Basisblöcken aufgetragen. Nach der BootRom-Initialisierung, nach Basisblock 12, erfolgt der Sprung in den Hauptspeicher und somit auch die Adaption der Pipeline. Nach etwa 60 Basisblöcken ist ein charakteristischer Verlauf der Initialisierung zu erkennen.

Dies begründet sich damit, dass an dieser Stelle eine Initialisierung des allokierten Speicherbereiches stattfindet. Hierbei wird immer eine Store-Doubleword-Instruktion ausgeführt und die Adresse in jedem Schleifendurchlauf um acht dekrementiert. Die beiden Linien mit der geringen Ersparnis nahe 0% sind die beiden Testläufe mit ineffizienten Cacheeinstellungen bei den beiden statischen Sprungvorhersagestrategien. Die anderen beiden Paare lassen sich ebenso eindeutig zuordnen. Das obere, über die Zeit ansteigende, blaue und rote Paar sind die beiden optimalen Cacheeinstellungen bei der statischen *Branch Never* (BN) Vorhersage. Der Anstieg lässt sich dadurch erklären, dass bei dieser Vorhersage Sprungziele häufiger verworfen werden müssen und somit mehr Pipelineleerlauf entsteht, der durch die adaptive Pipeline minimiert werden kann. Analog dazu handelt es sich bei der, über die Zeit abfallenden, grünen und violetten Linie um die beiden Testläufe mit der statischen *Branch Always* (BA) Vorhersage. Der Abfall über die Zeit lässt sich durch die optimale Sprungvorhersage erklären. Hierdurch entsteht von vornherein weniger Pipelineleerlauf, so dass weniger Spielraum für eine Optimierung durch die adaptive Pipeline besteht. Speziell im charakteristischen Bereich wird deutlich, dass die Sprungvorhersage bereits den Rücksprung an den Beginn der Schleife vorhersagt. Somit ist hier keine Optimierung durch die adaptive Pipeline möglich, wodurch die prozentuale Ersparnis deutlich geringer ausfällt. Wichtig ist an dieser Stelle nochmals hervorzuheben, dass die BA Vorhersage die optimale Lösung darstellt, da insgesamt weniger Zyklen für die Abarbeitung benötigt werden. Somit ist es besser einen kleineren Spielraum für die adaptive Pipeline zu haben, aber insgesamt jedoch ein wesentlich besseres Ergebnis in Form von weniger benötigten Zyklen.

Nachdem das Konzept der adaptiven Pipeline in den vorherigen Testläufen durch hauptsächlich kontrollflusslastige Initialisierungen überprüft wurde, wird in diesem abschließenden Testlauf der kürzeste Weg - ein Dijkstra-Algorithmus [37] - aus der MiBench Suite [65] ausgewählt. Für die direct-mapped Caches werden die optimalen, in den vorigen Tests gefundenen, Einstellungen write-allocate - never und write-back - always Parameter verwendet. Zur Sprungvorhersage wird BA verwendet. Diese Einstellungen ergeben die minimale gesamte Zyklenzahl bei den vorigen Testläufen. Eine zusätzlich durchgeführte Pipelinefusion kann selbst bei diesen optimal gewählten Parametern eine zusätzliche Verbesserung von knapp 4% bringen. Abbildung 4.6 zeigt den zeitlichen Verlauf der Abar-

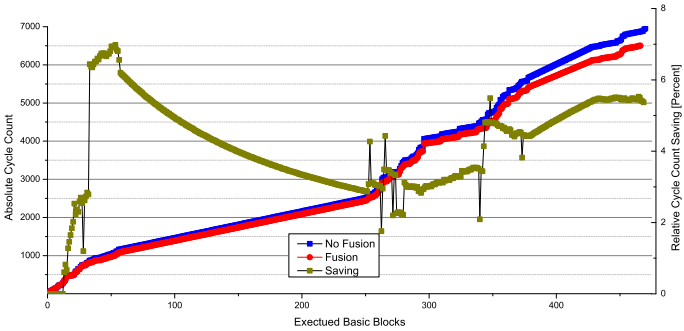


Abbildung 4.6.: Pipeline Fusion: Absolute Zyklenzahl & relative Ersparnis.

beitung in Form von ausgeführten Basisblöcken. Jede Ausführung eines Basisblocks kann theoretisch einen Zyklus einsparen. Allerdings verringert die verbesserte BA Sprungvorhersage das Ergebnis der adaptiven Pipeline, da speziell bei mehrfach wieder aufgerufenen Schleifen ein automatischer Rücksprung durch die Sprungvorhersage erfolgt.

4.4.3.2. HDL Generierung

Aus dem LISA-Modell der adaptiven Prozessor Pipeline kann mit dem Werkzeug PD die HDL Beschreibung generiert werden. Die Generierung funktioniert jedoch nur, wenn die Pipeline Adaptivität statisch modelliert ist, also alle Möglichkeiten der Adaption bereits zur Designzeit vorgegeben sind. Zur Synthese kommt für den Xilinx *Virtex-6* (XC6VLX240T) (ML605) FPGA das Werkzeug Synplify Pro zum Einsatz. Leider wird das *Register-File* (RF) in *Lookup Table* (LUT) und nicht in *BlockRam*s (BRAMs) realisiert. Daher hat die weder die absolute Ausnutzung des FPGAs noch die maximale Frequenz Aussagekraft. Abbildung 4.7 zeigt das Verhältnis des Ressourcenverbrauchs der Pipeline Stufen (links) und der einzelnen Einheiten der Execute Stufe (rechts). Es ist klar zu erkennen, dass die *Execute* Stufe die aufwändigste Stufe ist. Danach folgen die *Decode* und die *Memory* Stufe mit 12,88 % bzw. 10,45 %. Die *Instruction Fetch* und *Write Back* Stufe fallen mit weniger als 1,5 % nur wenig ins Gewicht.

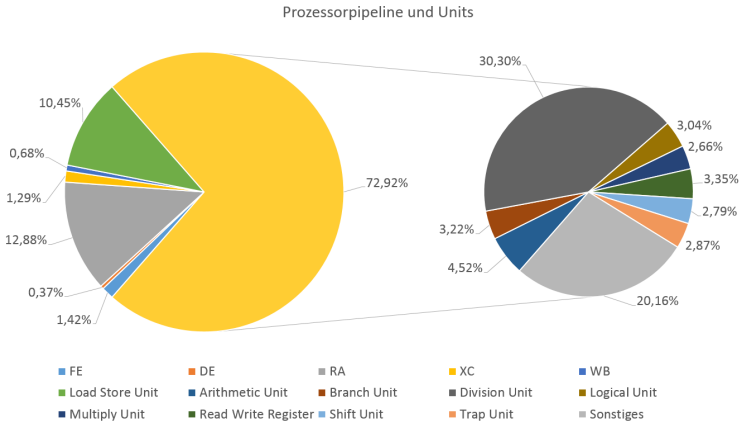


Abbildung 4.7.: Relativer Ressourcenverbrauch der Prozessor Pipeline (links) und der Execute Stufe (rechts).

Innerhalb der *Execute* Stufe fällt sofort die aufwändige Realisierung der Divisionseinheit auf. Im Gegensatz dazu konnte die Multipliziereinheit sehr effizient mit *Digital Signal Processor* (DSP) Slices realisiert werden.

4.5. Adaptive Superskalarität

In diesem Kapitel wird die adaptive Superskalarität vorgestellt. Das vorgestellte Konzept greift die Idee der *Advanced RISC Machines Ltd.* (ARM) big.LITTLE Architektur auf und versucht die konträren Ziele *Low Power* und Performanz in einem Prozessor zu realisieren. Funktionseinheiten können hier jedoch dynamisch hinzu geschaltet werden. Dieses Konzept wird exemplarisch mit LISA modelliert und anschließend mit Benchmarks evaluiert. Dieser Abschnitt basiert auf der Abschlussarbeit von Bao Ngoc An [An14].

4.5.1. Design der adaptiven Superskalarität

Superskalare Prozessoren können, wie in Abschnitt 2.3 beschreiben, in die Klassen *In-Order* (IO) und *Out-of-Order* (OoO) *Issue* bzw. *Completion* unterschieden werden. Da die vorliegende Arbeit auf der SPARC Architektur aufbaut und diese präzise Traps voraussetzt [151], muss die Möglichkeit der *Out-of-Order* (OoO) *Completion* bei der Umsetzung dieser Arbeit ausgeschlossen werden. Das Konzept wird hierdurch jedoch nicht eingeschränkt.

4.5.1.1. Auswahl der Funktionseinheiten

Bei einem superskalaren Prozessor sind mehrere Funktionseinheiten vorhanden, die die Instruktionen parallel und im Falle von OoO *Issue* auch außerhalb der Reihenfolge ausführen können. Jede der Funktionseinheiten ist für eine bestimmte Aufgabe, und dementsprechend auch eine Untermenge an Instruktionen, wie beispielsweise arithmetische, logische oder Load/Store Befehle, optimiert. Durch die Adaptivität soll sowohl die Anzahl der Funktionseinheiten, als auch der von ihnen abgedeckte Aufgabenumfang, skalierbar sein. Eine große Anzahl von Funktionseinheiten verringert die Wahrscheinlichkeit von Ressourcenkonflikten, jedoch steigert es den Flächenverbrauch und die Wahrscheinlichkeit, dass einzelne Funktionseinheiten über längere Zeit leer laufen und sehr ineffizient ausgelastet sind. Ist jedoch eine zu geringe Anzahl an Funktionseinheiten vorhanden, ist es möglich, dass die Auslastung auf Grund von Ressourcenkonflikten und Datenabhängigkeiten sehr gering ist. Es ist somit das Ziel dieses Konzeptes ein optimales Verhältnis zwischen der Anzahl und der Auslastung dieser Funktionseinheiten, also zwischen Performanz und benötigtem Hardware Aufwand, zu finden. Zur Bestimmung der *Issue Rate* und damit der maximal möglichen Parallelität des Prozessors werden auf Basis der Veröffentlichung von Jourdan bzw. Guthaus et al. [94, 65] verschiedene Benchmarks ausgewertet. Besonders interessant sind *SPECint92* und *SPECint2000* von der *Standard Performance Evaluation Corporation* (SPEC), *MiBench* und zusätzlich *Coremark*[1] von der *Embedded Microprocessor Benchmark Consortium* (EEMBC) als aktueller Benchmark, da durch die

[1] http://www.eembc.org/coremark/index.php

Ausführung der relevanten Applikationen auch auf die Ausführungseinheiten zurück geschlossen werden kann. Die Instruktionen werden nach 3 Befehlsgruppen gegliedert: Integeroperationen, Speicheroperationen und Verzweigungsoperationen. Insbesondere beim *Coremark* ist zu erkennen, dass die Verzweigungsbefehle mit 36,11 % aller Befehle einen relativ großen Teil einnehmen, obwohl sie bei den anderen realitätsnäheren Benchmarks unter 20 % liegen. Die Integeroperationen lassen sich in arithmetische, logische und schiebe Befehle unterteilen. Die genauen Verteilungen können in den oben genannten Veröffentlichungen eingesehen werden. Anwendungsabhängig ist es von Vorteil die Anzahl der jeweiligen Funktionseinheiten zu erhöhen, um beispielsweise dem hohen Anteil der Datenverarbeitung beim *Adaptive Differential Pulse Code Modulation* (ADPCM) Rechnung zu tragen.

Die Anzahl an Funktionseinheiten kann mit folgender Gleichung beschrieben werden:

$$N_{FU} = i \times P(I_{FU}) \tag{4.1}$$

Dabei ist N_{FU} die Anzahl an notwendigen Funktionseinheiten eines Typs, i die Zuordnungsrate pro Zyklus und $P(I_{FU})$ die Auftrittshäufigkeit einer Befehlsgruppe, die dieser Funktionseinheit zugeordnet wird.

4.5.2. Realisierung der adaptiven Superskalarität

In diesem Abschnitt wird die Realisierung der adaptiven superskalaren Prozessor Pipeline durch die *Architecture Description Language* (ADL) LISA vorgestellt. Wie bereits angesprochen basiert die Pipeline auf der SPARC Architektur, wodurch die Rahmenbedingen, wie im vorherigen Abschnitt beschrieben, einzuhalten sind. Die Realisierung erfolgt auf Basis der zyklenakkuraten Realisierung der adaptiven Prozessor Pipeline aus Unterabschnitt 4.4.2. Hierbei liegt das Hauptaugenmerk auf der *Integer Unit*. Eine *Floating-Point Unit* oder Co-Prozessoren könnten als zusätzliche funktionale Einheiten realisiert werden.

Wie im Konzept angesprochen, orientiert sich der Aufbau der superskalaren Pipeline am Kodierbaum der SPARC Architektur. Die verschiedenen Instruktionen und deren Formate werden verwendet, um die Instruktionen den jeweiligen Funktionseinheiten zuzuweisen. Um das dynamische Sche-

duling realisieren zu können, muss der Kodierbaum erweitert werden. In der *Instruction Fetch* Stufe müssen mehrere Instruktionen parallel geladen und in der *Decode* Stufe entsprechend parallel dekodiert werden. Dementsprechend muss der LISA *Coding Root* von einer auf mehrere initiale Instruktionen erweitert werden, aus denen alle weiteren Instruktionsformate abgeleitet werden können. Die besondere Komplexität besteht bei dieser Art der Implementierung darin, dass jede Funktionseinheit eine oder mehrere Instruktionen abarbeiten kann. Diese müssen diesen Funktionseinheiten exklusiv zugeteilt werden. Insbesondere muss beachtet werden, dass komplexere Funktionseinheiten mehrere Instruktionen akzeptieren können. Diese können auf anderen Funktionseinheiten eventuell effizienter abgearbeitet werden.

4.5.2.1. Erweiterung des Instruktionssatzes

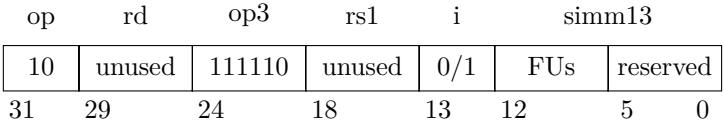


Abbildung 4.8.: Instruktionssatzerweiterungen auf Basis der SPARC V8 ISA mit der die Funktionseinheiten zu Laufzeit hinzu geschaltet werden können.

Wie in der Konzeption in Unterabschnitt 4.5.1 bereits beschrieben, wird zur Steuerung der Adaptivität in der superskalaren Pipeline eine Instruktion benötigt. Hierzu wird die für die adaptive Pipeline gewählte Instruktion entsprechend erweitert. In diesem Fall wird das i-Bit zum Hinzufügen bzw. Abschalten einzelner Funktionseinheiten genutzt. Die freien Bits repräsentieren die einzelnen Funktionalitäten, wie beispielsweise arithmetische oder logische Befehle. Abbildung 4.8 zeigt diese SPARC kompatible Instruktion.

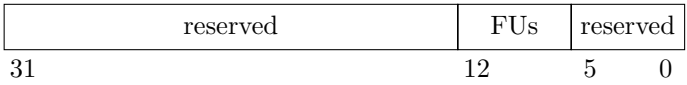


Abbildung 4.9.: Das *Adaptive Configuration Register* das den Zustand der adaptiven Pipeline in sieben Bits repräsentiert.

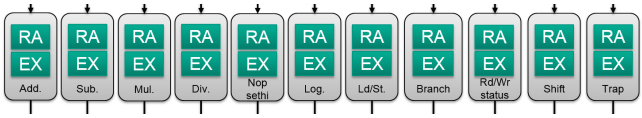
4.5.2.2. Realisierung der Laufzeitadaptivität

Es ist wichtig zur Laufzeit einen Überblick über den Status der aktiven Funktionseinheiten zu haben. Dies ist vor allem für die Laufzeitalgorithmen wichtig, um die Zuweisung der Instruktionen an die Funktionseinheiten richtig einteilen zu können. Deswegen wird jede Zustandsänderung durch die zusätzliche Instruktion analog zum Vorgehen bei der adaptiven Pipeline im ACR abgelegt. Das ACR ist in Abbildung 4.9 gezeigt.

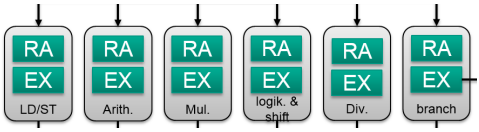
4.5.3. Evaluation anhand der adaptiven Superskalarität

Abschließend wird die adaptive superskalare Prozessor Pipeline bewertet. Zunächst wird mit dem *Processor Generator* (PG) HDL generiert, der für das Xilinx *Virtex-6* (XC6VLX240T) (ML605) FPGA synthetisiert wird. Durch den Vergleich des Logikverbrauchs kann der zusätzliche Aufwand ermittelt werden. Des weiteren kann das SystemC-Modell des Prozessors mit dem *Software Development Package Generator* (SDPG) erstellt werden. SystemC System Simulationen hiermit einfach durchgeführt werden. Bei der Evaluierung wird die Adaption gezeigt, indem zwei superskalare *In-order Issue*, *In-order Completion* Prozessormodelle mit unterschiedlicher Unterteilung der Funktionseinheiten hinsichtlich ihrer Performanz bzw. ihrem Realisierungsaufwand verglichen werden.

Die erste superskalare Prozessor Pipeline besteht aus fein-granular unterteilten Funktionseinheiten mit insgesamt elf Einheiten (vgl. Abbildung 4.10a). Im Gegensatz dazu wird die zweite superskalare Prozessor Pipeline auf nur sechs Funktionseinheiten reduziert, um den Hardwareoverhead zu verringern (vgl. Abbildung 4.10b). Es soll untersucht werden, wie hoch der Performanzunterschied bei der Programmausführung zwischen den



(a) Superskalare Pipeline mit einer feingranularen Unterteilung der Funktionseinheiten. Hier mit den elf Einheiten xyz.



(b) Superskalare Pipeline mit einer grobgranularen Unterteilung der Funktionseinheiten. Hier mit den sechs Einheiten xyz.

Abbildung 4.10.: Superskalare Pipelines mit unterschiedlicher Anzahl von Funktionseinheiten.

beiden Prozessoren ausfällt und ob dieser in Bezug auf den Mehraufwand der Implementierung gerechtfertigt ist.

4.5.3.1. Evaluation mit Bubble-Sort Algorithmus

	Skalare Pipeline	Superskalare Pipeline	
	1 *Functional Unit* (FU)	6 FU	11 FU
Speed Up	Baseline	1,16x	1,18x

Tabelle 4.1.: Performanzsteigerung der beiden adaptiven superskalaren Prozessormodelle im Vergleich zum skalaren Modell des *a*-Core.

Zur Evaluation wird das Prozessormodell mit der skalaren Pipeline mit den beiden Varianten des adaptiven superskalaren Prozessormodells verglichen. Alle drei LISA-Modelle führen den gleichen Bubble-Sort Algorithmus aus.

Mit den beiden superskalaren Modellen lässt sich ein deutlicher Gewinn von 16 % bzw. 18 % im Vergleich zum skalaren Prozessormodell erzielen. Der Vorteil der komplexen, mit mehr Funktionseinheiten aufgebauten, Pro-

zessorpipeline ist mit nur 2 % jedoch gering, so dass sich der Mehraufwand in der Realisierung nicht rechtfertigen lässt, so lange die Garantien nicht die maximale Performanz erfordern. Es ist jedoch zu beachten, dass für diese Evaluation mit dem Bubble Sort Algorithmus ein recht kontrollflusslastiger Algorithmus gewählt wurde, um eine realistische Einschätzung des Ansatzes zu bekommen. Es ist also bei datenflusslastigeren Algorithmen ein besseres Ergebnis zu erwarten. So lässt sich auch der Einsatz von mehr Funktionseinheiten rechtfertigen. Insbesondere spezielle aufwändige Einheiten, wie beispielsweise Multiply oder Division, müssen gut ausgelastet sein, um die Investition zu rechtfertigen. Eine zusätzliche ALU schlägt beispielsweise mit einem 13,78 % höheren LUT Verbrauch zu Buche, was den relativen Speedup auf 2 % minimiert. Somit ist es meist effizienter den einfacheren superskalaren Prozessor einzusetzen.

Eine HDL Generierung des superskalaren Prozessors war an dieser Stelle nicht möglich, da die superskalare Dispatch Stufe zum Zeitpunkt der Evaluation nicht vom *Processor Generator* (PG) unterstützt wurde.

Abschließend wurde mit Hilfe des *Software Development Package Generator*s (SDPGs) aus dem LISA-Modell ein SystemC-Modell erzeugt. Dieses kann generiert und in den PA eingebunden werden. Die Prozessormodelle werden über einen Bus mit einem Speicher verbunden, um den *Bubble Sort* Algorithmus realitätsnah auszuführen und die Ergebnisse der Evaluation auf der Hardware zu bestätigen.

4.6. Zusammenfassung

In diesem Kapitel wird das Konzept für den *a*-Core vorgestellt. Ziel des Konzeptes ist es die Eigenschaften der Prozessorplattform an die jeweilige Anwendung anzupassen. Hierdurch können die jeweiligen Anwendungen effizienter als der mittlere Anwendungsfall, auf den normalerweise optimiert wird, ausgeführt werden. Hierzu wird die Anpassung der Parameter des *a*-Core von der Designzeit in die Laufzeit verschoben, um mit erhöhter Flexibilität auf die unterschiedlichen Anforderungen der Anwendung reagieren und den Prozessor entsprechend anpassen zu können.

Zunächst wird ein Konzept vorgestellt, bei dem die Länge der Pipeline und somit die Verteilung der Last in der Pipeline zur Laufzeit angepasst werden kann. Des Weiteren wird ein Konzept zur adaptiven Anpassung der Superskalarität auf Basis der adaptiven Pipeline vorgestellt. Beide Konzepte werden mit der ADL LISA zyklenakkurat modelliert. Auf diesem Weg ist es möglich in relativ kurzer Entwicklungszeit einen komplexen Prozessor mit sämtlichen, zur Entwicklung notwendigen, Werkzeugen zu entwerfen.

Zur Evaluation kann zum einen das LISA-Modell des Prozessors, sowie auch die daraus generierte HDL Beschreibung, genutzt werden. Zur Evaluation werden Benchmarks aus der MiBench Suite herangezogen. Es kann insbesondere gezeigt werden, dass sich das Umschalten der Hardware deutlich auf die Ausführungszeit auswirkt. Hierbei können mehr als 10 % der zur Ausführung notwendigen Zyklen eingespart werden. Durch Optimierung der Cache und Sprungvorhersage Einstellungen kann der *a*-Core selbst verbessert werden. Die adaptive Pipeline kann jedoch trotzdem fast 3 % der Zyklen einsparen.

Bei der adaptiven superskalaren Pipeline werden 2 unterschiedliche Ausprägungen der Pipeline betrachtet. Einmal sind sechs funktionale Einheiten und einmal elf funktionale Einheiten realisiert. Es kann gezeigt werden, dass die superskalare Pipeline bis zu 18 % effizienter in der Laufzeit ist als die skalare Pipeline. Der Unterschied zwischen den Laufzeiten der superskalaren Realisierungen ist mit nur 2 % nicht groß genug, um den Mehraufwand in der Realisierung der zusätzlichen funktionalen Einheiten zu rechtfertigen.

Bei der Generierung der HDL zeigt sich, dass der *Processor Generator* nicht sehr effizient arbeitet. So wird beispielsweise das Register File mit LUT und nicht mit BRAMs realisiert. Hierdurch wird die maximale Frequenz sehr gering und verliert an Aussagekraft. Die relativen Verhältnisse in der Komplexität der Pipelinestufen und funktionalen Einheiten der *Execute* Stufe können jedoch als Anhaltspunkt für die Bewertung der Effizienz in Bezug auf die verwendete Fläche genutzt werden. Abschließend zeigt sich, dass es nicht möglich ist die Komplexität der adaptiven Modellierung direkt als HDL aus der LISA Beschreibung zu generieren.

5. Selbstadaptivität anhand der adaptiven Sprungvorhersage

Aktuell sind in eingebetteten Systemen vornehmlich einfache Sprungvorhersagen realisiert. Diese entsprechen nicht dem Stand der Technik und sind daher vergleichsweise ineffizient. Die adaptive Sprungvorhersage soll mit verschiedenen Konzepten, wie sie aus dem Stand der Technik bekannt sind, eine effizientere Realisierung bilden. Ziel ist die Selbstadaptivität zur Laufzeit, um zwischen einzelnen Konzepten umschalten zu können und somit eine Effizienzsteigerung zu erzielen.

5.1. Design der adaptiven Sprungvorhersage

Die adaptive Sprungvorhersage soll mehrere Prädiktoren enthalten, zwischen denen zur Laufzeit umgeschaltet werden kann. Es wird ein modulares Konzept unter Verwendung der in Abschnitt 2.4 vorgestellten Prädiktoren erarbeitet. Ein besonderes Augenmerk soll auf der Wiederverwendbarkeit der bereits vorhandenen Prädiktoren liegen. Diese werden so erweitert, dass es möglich ist, einzelne Funktionalitäten zur Laufzeit hinzu zu schalten. Es soll insbesondere möglich sein, auf Basis der statischen Prädiktoren zwischen dynamischen Prädiktoren umschalten zu können. Dies ist nützlich, da komplexere Prädiktoren eine relativ lange Lernphase haben. Während dieser Lernphase kann es zu einer lokalen Verschlechterung kommen. Hierbei ist es meist so, dass einfachere Prädiktoren zu diesem Zeitpunkt eine bessere Vorhersage liefern.

Die Möglichkeit des Umschaltens zur Laufzeit soll sowohl aus der Software, mit Hilfe einer zusätzlichen Instruktion, als auch selbstadaptiv in der Hardware erfolgen können. Somit wird es möglich, dass der Anwendungsentwickler, der Compiler oder das Laufzeitsystem eine Konfiguration der Prädiktoren spezifizieren. Auf Basis dieser Wahl der Prädiktoren kann sich das selbstlernende System sukzessive verbessern und zwischen den Prädiktoren dynamisch umschalten. Es wird hierzu eine Funktion definiert, nach deren Länge der Lernphasen und jeweiliger Priorisierung andere Prädiktoren gewählt werden können. An dieser Stelle sollte sich der Anwendungsentwickler mit Änderungen zurückhalten, da es sehr schwierig bis unmöglich ist, die Laufzeitabhängigkeiten zur Designzeit zu erkennen und aufzulösen. Selbst für den Compiler stellen diese Mechanismen, die tief in der Mikroarchitektur verankert sind, eine große Herausforderung dar. Deshalb wird der Fokus in diesem Kapitel auf die selbstadaptiven Optimierungen zur Laufzeit gelegt.

5.1.1. Erweiterung der Sprungvorhersage

Das modulare Konzept basiert auf den beiden statischen Prädiktoren *Always Taken* und *Always not Taken*. Zunächst muss die vorhandene *Always Taken* Vorhersage um den gegenteiligen Fall erweitert werden. Dies ermöglicht das adaptive Umschalten zwischen beiden statischen Vorhersagen. Es muss die adaptive Pipeline erweitert, die Sprungauswertung sowie auch die Missbehandlung überarbeitet werden. Die Sprungbedingung wird somit korrekt ausgewertet und ein entsprechendes Rückrollen der Befehlsverarbeitung wird in der Pipeline ermöglicht. In den weiteren Schritten werden die dynamischen Prädiktoren hinzugefügt und die adaptive Umschaltlogik so erweitert, dass das Anpassen der gesamten Sprungvorhersage möglich wird.

5.1.1.1. Selektion und Verarbeitung der bedingten Sprungbefehle

Bevor das Design der adaptiven Sprungvorhersage um die weiteren dynamischen Prädiktoren erweitert werden kann, ist es wichtig die bedingten Sprungbefehle in der Pipeline von den unbedingten Sprungbefehlen zu trennen. Optimaler weise kann die Sprungverhersage bereits in der *Instruc-*

tion Fetch Stufe gesteuert werden. Im Falle eines unbedingten Sprunges wird direkt die Sprungadresse berechnet und der Sprung durchgeführt. Im Falle eines bedingten Sprunges muss jedoch, unter Berücksichtigung der jeweils gewählten Prädiktion, entschieden werden, ob wie bei *Taken* das neue Sprungziel berechnet und gesetzt wird, oder wie bei *Not Taken* der Programmablauf fortgesetzt wird.

Die Auswahl eines Prädiktors und das Auslesen der entsprechenden Vorhersage muss dabei bereits beim Laden eines Befehls durchgeführt und die Prädiktion dann zusammen mit dem Befehl in die Pipeline geschrieben werden. Nur so ist sichergestellt, dass für jeden Befehl permanent auf die ihm zugeordnete Vorhersageinformation zugegriffen werden kann, da sich die Prädiktoren aufgrund anderer sich in der Pipeline befindender Sprungbefehle jederzeit ändern können.

5.1.1.2. Umkehr der Sprungauswertung

Für die korrekte Auswertung der Sprungbedingung muss die für einen Sprungbefehl verwendete Prädiktion berücksichtigt werden. Wird beispielsweise die Bedingung eines als genommen vorhergesagten Sprunges (*Taken*) als wahr ausgewertet, wurde der Sprungbefehl korrekt verarbeitet. Wird die Bedingung dagegen als falsch ausgewertet, hätte der Sprung nicht genommen werden dürfen und ein Rückrollen der Befehle wird notwendig. Im Falle eines nicht genommenen Sprunges (*Not Taken*) gilt dies genau umgekehrt: Ist die Bedingung wahr, hätte der Befehl genommen werden müssen und ein Rückrollen ist notwendig. Ist die Bedingung jedoch falsch, wurde der Sprung dagegen korrekt verarbeitet. Eine einfache Lösung ergibt sich durch die Invertierung der ausgewerteten Sprungbedingung, sobald *Not-Taken* vorhergesagt wurde.

5.1.2. Auswahl geeigneter Prädiktoren

Basierend auf den statischen Prädiktoren *Always Taken* und *Always not Taken* können dynamische Prädiktoren ausgewählt werden, die zur Laufzeit zwischen den Zuständen umschalten können und somit eine adaptive Sprungvorhersage ermöglichen. Der adaptiver Prozessor (a-Core) soll die

bestmögliche Performanz unter möglichst geringem Einsatz von Ressourcen für die Realisierung in Anspruch nehmen. Bei der Auswahl der Prädiktoren muss also, vor allem bei diesem eingebetteten System, auf die Effizienz der Implementierung geachtet werden. Dem entgegen steht das Ziel eine möglichst gute Trefferquote von mehr als 90 % zu erreichen. Es müssen bei der Auswahl der Prädiktoren also Kenngrößen, wie der Speicherplatzbedarf für die *Branch History Register* und *Pattern History Table*, sowie auch die Komplexität der Lese- und Aktualisierungslogik beachtet werden.

Die in Abschnitt 2.4 vorgestellten Prädiktoren erreichen ihre bestmöglichen Trefferquoten vor allem wenn die Lokalität der Sprünge berücksichtigt wird. Hierzu sind sehr große *Pattern History Table*s und dementsprechend viel Logik notwendig, um die Laufzeitinformationen erfassen zu können. Die Tabellen können zusammen mit der benötigten Aktualisierungslogik sehr leicht so groß wie der eigentliche Prozessor werden. Aus Effizienzgründen sollen daher im Kontext dieser Arbeit möglichst einfach realisierbare Prädiktoren verwendet werden, die durch die Kombination und Wiederverwendung der Logikressourcen ähnlich gute Trefferquoten erzielen können.

Es werden zunächst einfache 1-Bit und 2-Bit Prädiktoren gewählt, die zumeist die Grundlage für eine aufwändigere Aktualisierungslogik der komplexeren Prädiktoren darstellen. Diese Logik kann daher zu einem späteren Zeitpunkt vollständig wiederverwendet werden. Lediglich die *Pattern History Table*s müssen für die komplexeren Prädiktoren erweitert werden. Bei der Auswahl der 2-Bit Prädiktoren fällt die Wahl auf den Sättigungszähler, der im Vergleich zum Hysteresezähler ein ausgewogeneres Verhalten bei Fehltreffern zeigt, da der Übergang in den entsprechend gegensätzlichen schwachen Zustand erfolgt. Des Weiteren wird der *gShare* Prädiktor ausgewählt, um die Trefferquoten der einfachen dynamischen Prädiktoren weiter zu verbessern. Das *Dynamic History Length Fitting* (DHLF) ist außerdem eine interessante Anpassung des *gShare*, die eine bessere Performanz ermöglichen soll.

Die Verwendung von Hybridprädiktoren ist eine zusätzliche Möglichkeit die Trefferquoten zu verbessern. Hierbei werden die Stärken mehrerer Prädiktoren kombiniert, in dem die aktuellen Trefferquoten ausgewertet werden. Basierend darauf wird zur Laufzeit die Entscheidung getroffen, welcher Prädiktor für die Vorhersage verwendet wird. Dies ist insbesondere

interessant, da das Umschalten zwischen den Prädiktoren adaptiv erfolgen kann.

Für die Auswahl des aktuellen Prädiktors in der adaptiven Sprungvorhersage kommt ein Meta-Prädiktor zum Einsatz. Ähnlich zur *Pattern History Table* des 2-Bit Prädiktors handelt es sich um zwei sättigende Zähler mit vier Zuständen, wobei jeweils zwei für den 2-Bit Prädiktor und *gShare* stehen. Hieraus kann, wie in Tabelle 5.1 zu erkennen ist, abgeleitet werden ob es einer Aktualisierung des Prädiktors bedarf.

2-Bit	gShare	Aktualisierung
Fehltreffer	Fehltreffer	keine
Fehltreffer	Treffer	in Richtung gShare
Treffer	Fehltreffer	in Richtung 2-Bit
Treffer	Treffer	keine

Tabelle 5.1.: Aktualisierung des Meta-Prädiktors.

Da der Anwender zur Designzeit, sowie auch zur Laufzeit, in der Lage sein soll festzulegen, welche statischen bzw. dynamischen Prädiktoren verwendet werden sollen, ist es wichtig den modularen Aufbau so zu gestalten, dass die einfacheren Prädiktoren auch losgelöst von den komplexeren Prädiktoren arbeiten können.

5.1.3. Modulares Sprungvorhersagekonzept

Das modulare Konzept soll es ermöglichen die Prädiktoren in der jeweils einfachsten Variante nutzen zu können. Des Weiteren sollen die bereits vorhandenen Ressourcen wiederverwendet werden und von den jeweils komplexeren Prädiktoren mit genutzt werden, ohne dabei die Funktion des jeweiligen Prädiktors einzuschränken. Hierzu sollen die Prädiktoren als einzelne Module der adaptiven Pipeline hinzugefügt werden. Nach Möglichkeit soll vermieden werden, dass die Funktionalität über mehrere Pipelinestufen hinweg verteilt wird. Vielmehr wird angestrebt, dass die einzelnen Module mit dedizierten Signalen mit den jeweiligen Pipelinestufen kommunizieren, so dass die bisher vorgegebene Struktur der Pipeline wiederverwendet werden kann. Somit ist es auch einfach möglich die Prädiktoren weiter zu

verbessern. Insbesondere die Initialisierung, die jeweiligen Updates und das Auslesen der einzelnen Prädiktoren wird vereinfacht.

Die Initialisierung der *Pattern History Table*s sollte idealerweise zur Laufzeit stattfinden. Auf diese Weise können die ersten Vorhersagen der statischen Prädiktoren genutzt werden, um die dynamischen Prädiktoren zu initialisieren. Somit kann eine relativ lange Lernphase genutzt werden, da die statischen Prädiktoren vor allem am Anfang besser sein werden als die noch nicht initialisierten Prädiktoren, um die dynamischen Prädiktoren zu initialisieren. Sobald die dynamischen Prädiktoren ein besseres Vorhersageergebnis als die statischen Prädiktoren liefern, wird auf diese umgeschaltet. Auf diese Weise kann auch bereits das Wissen zur Laufzeit genutzt werden, um den Arbeitspunkt der dynamischen Vorhersagen besser einstellen zu können. Sobald die Lernphase abgeschlossen ist, also die Trefferquote der dynamischen Vorhersagen besser als die Trefferquote der statischen Vorhersagen ist, kann das automatische Umschalten zur dynamischen Vorhersage erfolgen.

Des Weiteren ist es sinnvoll die Updates für alle Prädiktoren zu synchronisieren. Somit kann sichergestellt werden, dass alle Prädiktoren zum Zeitpunkt eines Umschaltens auf dem gleichen Stand sind. Insbesondere ist es wichtig das Ergebnis der Vorhersage auch an den Meta-Prädiktor weiter zu geben. Hierdurch sind die jeweiligen Trefferquoten am Besten vergleichbar und das automatische Umschalten hat die besten Erfolgsaussichten. Sicherheitshalber muss jedoch überprüft werden, ob das Update bereits bei allen Prädiktoren übernommen wurde, damit keine inkonsistenten Zustände kurz vor oder aufgrund des Umschaltvorgangs hervorgerufen werden, da der zu aktualisierende Prädiktor mit dem verwendeten Sprungbefehl überein stimmt. Dies ist dann nicht der Fall, wenn die beiden Prädiktoren unterschiedliche Vorhersagen treffen oder falls ein Prädiktor aufgrund eines anderen, sich ebenfalls in der Pipeline befindenden, Sprungbefehls bereits aktualisiert wurde.

5.1.4. Adaptives Umschalten

Über das modulare Konzept hinaus muss der adaptiven Sprungvorhersage eine zusätzliche Logik hinzugefügt werden, die das adaptive Umschalten zur Laufzeit ermöglicht. Dies soll auf Basis einer zusätzlichen Instruktion

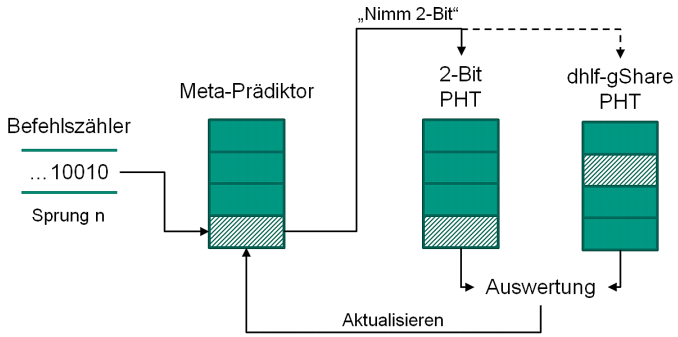


Abbildung 5.1.: Prinzipieller Aufbau eines Hybridprädiktors.

möglich sein, die das *Adaptive Configuration Register* verändert, das von der Auswahllogik gelesen wird. Bei einer automatischen Anpassung der Sprungvorhersage muss das *Adaptive Configuration Register* entsprechend aktualisiert werden, um den jeweils aktuellen Zustand des a-Core für das Laufzeitsystem abzubilden.

Für diese Adaptivität bietet sich insbesondere ein Hybrid-Prädiktor an, der die besten Eigenschaften der vorhandenen Prädiktoren vereint. Abbildung 5.1 zeigt den prinzipiellen Aufbau des Hybrid-Prädiktors bestehend aus dem 2-Bit Prädiktor, dem DHLF *gShare* Prädiktor und dem Metaprädiktor, der auf Basis der Vorhersagen der beiden Prädiktoren den aktuellen Prädiktor auswählt.

Diese Auswahllogik sollte der *Instruction Fetch* Stufe der adaptiven Pipeline hinzugefügt werden. Diese Pipelinestufe wird logisch als letzte ausgeführt, so dass bereits alle Signale aktualisiert zur Verfügung stehen.

5.1.5. Mehrkern- und Vielkern Szenario

Eine Fehlvorhersage ist vor allem dann teuer, wenn aufgrund dieser Vorhersage der Cache aktualisiert wurde und damit die Daten, die für die eigentlich richtige Ausführung benötigt werden, verdrängt wurden und dementsprechend neu geladen werden müssen. Bei der bisher betrachteten Architektur ist der Prozessor zusammen mit dem Speicher direkt am

Bus angeschlossen, was die Zugriffszeiten sehr gering hält. Daher ist eine Fehlvorhersage nicht besonders teuer. Anders verhält sich das, wenn der Prozessor in einer verteilten gekachelten Architektur, wie beispielsweise der in Unterabschnitt 3.3.1 vorgestellten *Invasive Computing* (InvasIC) Architektur, eingesetzt wird.

Liegen die Daten nicht im lokalen *Transaction Level Modeling* (TLM), auf den innerhalb von einem Zyklus zugegriffen werden kann, muss über das *Network on Chip* (NoC) auf den globalen *Double Data Rate* (DDR) Speicher zugegriffen werden. Die Zugriffszeit hängt jetzt maßgeblich von der Entfernung der Kachel zum Speicher ab. An jeder Verzweigung im NoC befindet sich ein *Router*, dessen Latenz auf bis zu 5 Zyklen anwachsen kann [72]. Es ist zu beachten, dass das NoC für nicht priorisierte Anfragen nach dem *Best Effort* Prinzip arbeitet. Die Latenz kann an viel genutzten Knoten deutlich ansteigen, was vor allem im viel genutzten Zentrum der Architektur zu Flächenhälsen führen kann.

Aus diesem Grund ist es wichtig, dass bei der Evaluation mehrere Szenarien betrachtet werden: Neben dem Zugriff innerhalb der Mehrkernkachel, in dieser Realisierung mit vier Kernen, was der einfachen Architektur mit Zugriff über den lokalen Bus entspricht, soll das Szenario auch erweitert werden, um auf Daten benachbarter Kacheln und Daten im globalen Speicher zugreifen zu können.

Es ist zu erwarten, dass die Trefferquote hier einen deutlicheren Einfluss auf die Performanz und auch die Effizienz hat, da das Nachladen durch die teureren Speicherzugriffe wesentlich länger dauert. Dennoch ist zu beachten, dass die Ausführungszeit nicht im gleichen Maße wie die Trefferquote ansteigen wird. Hierbei sind limitierende Faktoren bei der Programmausführung, aber auch die Taktfrequenz, maßgeblich.

5.2. Realisierung der adaptiven Sprungvorhersage

In diesem Unterkapitel wird die Realisierung der adaptiven Sprungvorhersage vorgestellt. Als Grundlage für die Realisierung wird die *Very High Speed*

Integrated Circuit Hardware Description Language (VHDL) Beschreibung des LEON3 Prozessors genutzt. Die bestehenden Strukturen des adaptiven Prozessors (*a*-Cores), der bereits in den vorangegangenen Kapiteln vorgestellt wurde, werden zunächst modularisiert, um anschließend die weiteren Prädiktoren, sowie auch die Auswahl- und Umschaltlogik hinzuzufügen. Weiterführende technische Informationen zur adaptiven Sprungvorhersage sind in den Abschlussarbeiten von Manuel Härdle [Här13], Brian Pachideh [Pac15] und Sven Nitzsche [Nit15] zu finden.

5.2.1. Anpassung des *a*-Core

Zunächst sollen, die im vorigen Abschnitt angesprochenen, notwendigen Änderungen am bestehenden *a*-Core umgesetzt werden. Dies umfasst insbesondere den bestehenden *Always Taken* Prädiktor und dessen Erweiterung auf die gegenteilige statische Vorhersage *Always not Taken*, um die Grundlage für die Realisierung der weiteren dynamischen Prädiktoren vorzubereiten.

Als ersten Schritt ist es wichtig die bedingten von den unbedingten Verzweigungen zu trennen, da nur diese von der Sprungvorhersagelogik bearbeitet werden müssen. Hierzu wird die vorhandene Logik, wie sie auch zur Auswahl der Sprungbefehle genutzt wird, in der *Decode* Stufe erweitert, um die *Condition Codes* innerhalb der jeweiligen Sprungbefehle auf bedingte Sprünge überprüfen zu können.

Im nächsten Schritt muss die Sprungverarbeitung angepasst werden. Es ist jetzt nicht mehr ausreichend nur auf das Sprungziel zu verzweigen. Vielmehr muss jetzt auch im Falle von *Not Taken* eine Verzweigung auf die bereits geladene Instruktion ermöglicht werden. Hierzu muss die entsprechende Fallunterscheidung in der *Instruction Fetch* Stufe erweitert werden. Weicht das Sprungziel vom bereits geladenen *Program Counter* (PC) ab, muss die neue Adresse in den nächster Programzähler (nPC) geladen werden.

Dementsprechend muss auch die Sprungauswertung erweitert werden, damit das Auswertungsergebnis des genommenen *Not Taken* Sprunges interpretiert werden kann. Die Auswertungslogik muss also zusätzlich invertiert werden können, damit die korrekte Auswertung gewährleistet werden kann.

Zusätzlich zur bereits vorhandenen Fehltreffererkennung muss dementsprechend auch eine Treffererkennung hinzugefügt werden, um alle Möglichkeiten des Erfolges oder Misserfolges einer Vorhersage speichern zu können. Dementsprechend müssen auch die jeweils genutzten Prädiktoren aktualisiert werden. Dies ist insbesondere wichtig, um die fehlerfreie Abarbeitung der Instruktion im *Delay Slot* zu gewährleisten. Hierbei muss beachtet werden, dass Instruktionen, die auf einen Sprung folgen, im *Delay Slot* annulliert werden können. Hierzu wird eine Fallunterscheidung auf Basis der Prädiktoren, die dem jeweiligen Sprungbefehl zugeordnet sind, vorgenommen. Diese Unterscheidung muss insbesondere bei einem Treffer erfolgen, da der *Delay Slot* bei einem Fehltreffer immer annulliert werden muss.

Des Weiteren müssen die Befehlszähler *Program Counter* (PC) und nPC durch die Vorhersagen angepasst werden können. Außerdem muss die Möglichkeit bestehen das Sprungziel wieder anpassen zu können, sobald der Sprung in der *Execute* Stufe ausgewertet wurde. Die Prädiktoren werden hierzu ausgewertet und auf Basis der Vorhersage wird der nPC gesetzt.

Einer zusätzlichen gesonderten Behandlung bedürfen die *Delayed Control-Transfer Instructions* (DCTI) Paare. Hierbei treten zwei Verzweigungsinstruktionen direkt nacheinander auf, so dass die Ausführung der zweiten Verzweigung auf dem Ergebnis der Vorhersage der ersten Verzweigung beruht. Es ist hierbei notwendig die Paare sehr früh in der Pipeline zu erkennen, um das Ausführen der zweiten Instruktion im *Delay Slot* rechtzeitig annulieren zu können. Ansonsten wird das Programm an das Ziel der zweiten Instruktion verzweigt, was ein zweifaches Zurückrollen bei einer falschen Vorhersage zur Folge hätte.

5.2.2. DHLF gShare

Die Realisierung des DHLF *gShare* erfolgt als eigenständiges Modul, um welches der *a*-Core erweitert werden muss. Dementsprechend muss auch die Aktualisierungslogik für die adaptiven Prädiktoren erweitert werden, damit sie die Treffer bzw. Fehltreffer weiter verarbeiten kann. Folglich kann jetzt für jeden Treffer eine '1' und für jeden Fehltreffer eine '0' in das *Branch History Register* geschrieben werden.

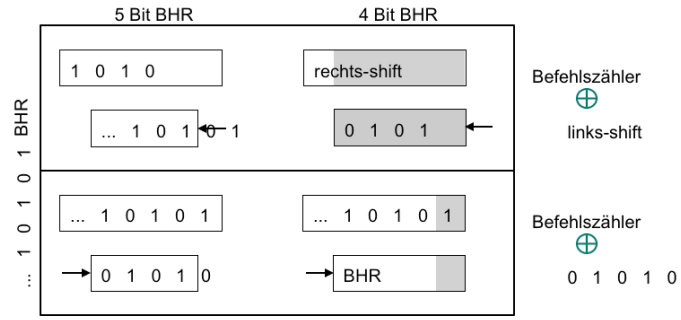


Abbildung 5.2.: Effekt der Schieberichtung des *Branch History Register* auf die *Pattern History Table* Adresse.

Um die dynamische Länge zu realisieren, wird ein Zeiger realisiert, der die zu verwendende *Branch History Register* Länge zwischen $'0'$ und $'N'$ festlegt. Folglich werden nur die höherwertigen Bits des PC mit dem *Branch History Register* verknüpft, da sich diese seltener ändern. So kann auch *Aliasing* vermieden werden.

Dementsprechend muss die Verschiebungsrichtung des *Branch History Register* rechts sein. Falls sich die *Branch History Register* Länge ändert ist es wahrscheinlich, dass einem bereits bekannten Sprungziel die selbe *Pattern History Table* zugewiesen wird. Abbildung 5.2 verdeutlicht diese Problematik, die auch zu stärkerem *Aliasing* führen kann.

Als nächstes kann die Auswertung der Intervalle, wie in den Grundlagen beschrieben, realisiert werden. Zuletzt kann die *Local Minimum Avoidance* (LMA) realisiert werden. Es wird ein Zähler verwendet, der die Anzahl der gleichgewählten *Branch History Register* Längen speichert. Der Zähler wird zurückgesetzt, wenn sich die gewählte *Branch History Register* Länge ändert. Falls der Zähler die gewählte Grenze erreicht wird die *Local Minimum Avoidance* (LMA) aktiviert und zu einer anderen *Branch History Register* Länge gesprungen.

5.2.3. Modulare Sprungvorhersage

Die adaptive Sprungvorhersage setzt sich aus drei Teilen zusammen: i) die Initialisierung, ii) die Aktualisierung und iii) die Auswahl der Prädiktoren. Die Sprungvorhersage wird, wie im vorherigen Konzeptabschnitt angedacht, als eigenständiges Modul in die *Instruction Fetch* Stufe implementiert, auf das aus allen weiteren Pipelinestufen für die entsprechenden Updates zugegriffen werden kann.

Die Initialisierung der Prädiktoren wird als einzelnes Modul realisiert, damit die Trainingsphase der dynamischen Prädiktoren parallel zur Ausführung einfacherer Prädiktoren erfolgen kann. So können die Prädiktoren beispielsweise, je nach Größe des *Pattern History Table*s, auf Treffer für den 1-Bit Prädiktor und schwacher Treffer für den 2-Bit Prädiktor gesetzt werden, falls der jeweilige Prädiktor vorher nicht genutzt wurde. Dies kann zur Laufzeit erfolgen und entspricht bis zur ersten Nutzung des Prädiktors einer statischen *Always Taken* Vorhersage. Die einzelnen *Pattern History Table*s sind voneinander unabhängig, so dass sie individuell, beispielsweise aus energetischen Gründen abgeschaltet werden können. Sie können jedoch auch als Basis für die komplexeren Prädiktoren, wie beispielsweise *gShare* wiederverwendet werden.

Zur Aktualisierung der jeweiligen Prädiktoren bedarf es einer Fallunterscheidung. Diese stellt zunächst fest, ob es sich um einen Treffer oder um einen Fehltreffer handelt. Es muss jeweils überprüft werden, ob die Vorhersage auch eingetreten ist. Dementsprechend muss der Prädiktor aktualisiert werden, im Falle eines 1-Bit Prädiktors jedoch nur im Falle einer falschen Vorhersage. Im Falle des 2-Bit Prädiktors bedarf es auch einer Aktualisierung wenn eine richtige Vorhersage vorliegt: Falls der Prädiktor noch eine schwache Vorhersage gespeichert hat, muss diese zur starken Vorhersage aktualisiert werden. Ansonsten muss die Vorhersage umgedreht und der gegenteilige schwache Zustand gesetzt werden.

Zur Kombination der Prädiktoren muss die Logik in das Prädiktormodul des *a*-Core eingefügt werden. Es wird also eine Tabelle mit 2^N Einträgen mit jeweils 2-Bit für die Aktualisierungs- und Auswahllogik des Meta-Prädiktors benötigt.

Mit Hilfe der Auswahllogik ist es möglich einen der realisierten Prädiktoren auszuwählen. Diese Auswahl des jeweiligen Prädiktors wird durch Fallun-

terscheidung in einem Register realisiert. Dieses Register lässt sich durch eine Instruktion sowie auch durch die adaptive Sprungvorhersage selbst verändern. Dementsprechend muss das Register auch aus der Software lesbar sein, um einen aktualisierten Zustand auswerten zu können. Auf Basis dieses Registers wird die Vorhersage des aktuell gewählten Prädiktors getroffen und an die Pipeline übergeben. Es ist dabei zu beachten, dass die entsprechende Wahl durch Zustandsübergänge des Meta-Prädiktors im Hybrid-Prädiktor überschrieben wird, was zu einer verbesserten Trefferquote im System führen soll.

5.2.4. Erweiterung des Instruktionssatzes

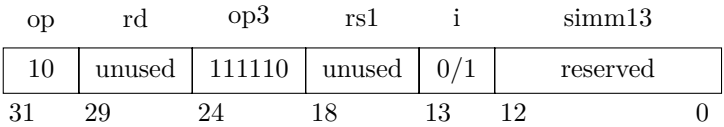


Abbildung 5.3.: Instruktionssatzerweiterungen auf Basis der *Scalable Processor ARChitecture* (SPARC) V8 Instruktionssatz Architektur (ISA) mit der die adaptive Sprungvorhersage konfiguriert werden kann.

Wie in der Konzeption in Abschnitt 5.1 bereits beschrieben, ist die Anpassung der Sprungvorhersage automatisch in der Hardware als auch per Instruktion durch die Software möglich. Dazu muss dem SPARC Instruktionssatz eine Instruktion hinzugefügt werden. Hierzu wird eine freie Instruktion aus dem Format 3 gewählt und so modifiziert, dass eine optimale Implementierung hinsichtlich der Ressourcenminimalität gewährleistet ist. Op3 wird wie in den vorangegangenen Erweiterungen auch zu $0x3E$ festgelegt. Der Unterschied ist hier, dass ein Register genutzt werden muss, um die gewünschte Änderung an die Pipeline zu übergeben, da die 13 verfügbaren Bits des Immediate schon belegt sind. Da die *SETHI* Instruktion genutzt werden soll, wird das 13. Bit als niedrigwertigstes Bit gewählt. Dieses niedrigwertigste Bit 13 repräsentiert die statische Vorhersage *Always Taken* ('1') bzw. *Always not Taken* ('0'). Das darauf folgende Bit 14 repräsentiert die dynamischen 1-Bit ('0') bzw. 2-Bit Vorhersage ('1'). Das nächste Bit 15 selektiert den *gShare* Prädiktor. Das höchstwertige

Bit 16 aktiviert den Hybrid-Prädiktor. Abbildung 5.3 zeigt diese SPARC kompatible Instruktion.

5.2.5. Realisierung der Laufzeitadaptivität

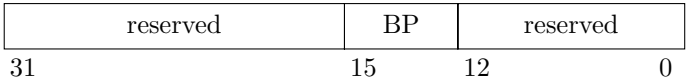


Abbildung 5.4.: Das *Adaptive Configuration Register* das den Zustand der adaptiven Sprungvorhersage repräsentiert.

Es ist wichtig den Status der adaptiven Sprungvorhersage zur Laufzeit abzuspeichern. Hierdurch ist es möglich, dass Änderungen der Konfiguration der Sprungvorhersage, sowohl durch die Software, als auch durch die Hardware vorgenommen werden können. Das niedrigwertigste Bit 12 repräsentiert die statische Vorhersage *Always Taken* ('1') bzw. *Always not Taken* ('0'). Das darauf folgende Bit 13 repräsentiert die dynamischen 1-Bit ('0') bzw. 2-Bit Vorhersage ('1'). Das nächste Bit 14 selektiert den *gShare* Prädiktor. Das höchstwertige Bit 15 aktiviert den Hybrid-Prädiktor.

5.2.6. Laufzeitoptimierung

Zur Verifikation der Realisierung werden einige Tests mit dem *Basicmath Small* Algorithmus aus der *MiBench* Sammlung durchgeführt. Zur Verkürzung der Testläufe wird hier lediglich die kubische Funktion verwendet. Tabelle 5.2 zeigt die Ergebnisse der Testläufe. Es ist zu erkennen, dass der 2-Bit Prädiktor gefolgt von 1-Bit Prädiktor die beste Trefferquote liefert. Es ist jedoch zu beachten, dass die Ausführungszeit für den 1-Bit Prädiktor am Längsten und am zweitlängsten für den 2-Bit Prädiktor ist.

Dies wirft Fragen auf, die sich nur durch die Betrachtung des Verhaltens der Architektur zur Laufzeit erklären lassen. So werden beispielsweise im Falle von Cache Misses lange Wartezeiten für das Laden des Speichers verursacht, in denen die Pipeline angehalten werden muss und keine Verarbeitung erfolgen kann. Daher kann angenommen werden, dass eine Größe

Tabelle 5.2.: Simulationsergebnisse für *Basicmath Small*

Prädiktor	Simulationszeit	Trefferquote
NoBP	2.262.461 ns	-
AT	2.188.824 ns	42 %
ANT	2.204.849 ns	62 %
1-Bit	2.214.749 ns	63 %
2-Bit	2.207.374 ns	65 %

des Datencaches hier auch wesentlich die Laufzeit beeinflusst und nur ein möglichst großer Datencache die eigentliche Bewertung der adaptiven Sprungvorhersage zulässt. Dementsprechend wird in der Evaluation auch die Größe des Datencaches variiert, um neben der Trefferrate der Sprungvorhersage auch eine Bewertung der Auswirkung auf die Laufzeit der Applikation vornehmen zu können.

Andererseits spielt die Konfiguration der Prädiktoren eine entscheidende Rolle für die Effizienz des Gesamtsystems. Wie bereits erwähnt kann das *Aliasing* nur mit sehr großen *Pattern History Table*s verhindert werden. Für die Bewertung der Effizienz soll auch die für die Realisierung verwendete Fläche und nicht nur die reine Performanzsteigerung betrachtet werden. Es ist also sinnvoll die Lokalität zu verbessern, indem beispielsweise 32 (2^5) Prädiktoren gewählt werden, um die letzten 5 Bits der Adresse abbilden zu können. Sollte sich hier bereits eine signifikante Verbesserung zeigen, kann überlegt werden auf Kosten von weiterem Speicherplatz mehr Prädiktoren zur Verfügung zu stellen, um die Performanz weiter steigern zu können. Die Effizienz wird hier jedoch einbrechen, da die Realisierung mehr Speicherplatz benötigt.

5.2.7. Mehrkern- und Vielkernszenario

Es wird zunächst eine minimale Konfiguration genutzt, um den begrenzten Ressourcenbedarf des Xilinx *Virtex-6* (XC6VLX240T) (ML605) *Field Programmable Gate Array* (FPGA) einschätzen zu können. Ein Basisdesign mit zwei Kacheln mit jeweils einem Prozessor benötigt 47 % der verfügbaren *Lookup Table*s (LUTs). Jeder Prozessor verfügt über 8 kB *Level 1* (L1)

Datencache und 16 kB L1 Instruktionscache. Es wird keine *Floating-Point Unit* verwendet.

Mit zwei weiteren Kacheln, also insgesamt vier Kacheln mit jeweils einem Prozessor, sind 95 % der Ressourcen belegt. Leider lässt sich die Realisierung nicht in der gewünschten Größe umsetzen, da der *Virtex-6* (XC6VLX240T) (ML605) nicht genügend *BlockRam* (BRAM) zur Verfügung stellt. Ein zweiter Rechenkern schlägt mit etwa 13 % der Ressourcen zu buche, so dass bei der Evaluation das 2x1 System mit einem Rechenkern pro Kachel zum Einsatz kommt.

5.3. Evaluation der adaptiven Sprungvorhersage

Nach der Realisierung der adaptiven Sprungvorhersage im *a*-Core kann die Bewertung erfolgen. Dies geschieht in mehreren Schritten: Zunächst werden die einzelnen Prädiktoren anhand des *Dhrystone*, des *Coremark* und Benchmarks aus der *MiBench* Suite evaluiert. Anschließend wird die Optimierung der Prädiktoren bewertet. Abschließend erfolgt die Bewertung auf einer Multi-Core Kachel in der Many-Core Architektur, wodurch sich die verbesserte adaptive Sprungvorhersage besonders positiv auf die Ausführungszeit auswirkt.

5.3.1. Bewertung der Prädiktoren

Zunächst werden die Trefferquoten und die damit verbundenen Ausführungszeiten ausgewertet. Auf Grund der relativ langsamen Kommunikation über den *Universal Asynchronous Receiver Transmitter* (UART) werden sämtliche Ausgaben deaktiviert und nur die Trefferquoten bzw. Laufzeiten bei den eigentlichen Berechnungen bewertet. Weiterhin soll der für die Implementierung notwendige zusätzliche Platzbedarf ermittelt werden.

Zur Bewertung kommt ein *Dhrystone* mit einer Million *Dhrystone* Zyklen zum Einsatz. Des weiteren kommt ein *Basicmath Large* zum Einsatz, der eine vierfach verschachtelte Schleife und dementsprechend eine sehr große Anzahl an bedingten Sprungbefehlen abarbeiten muss.

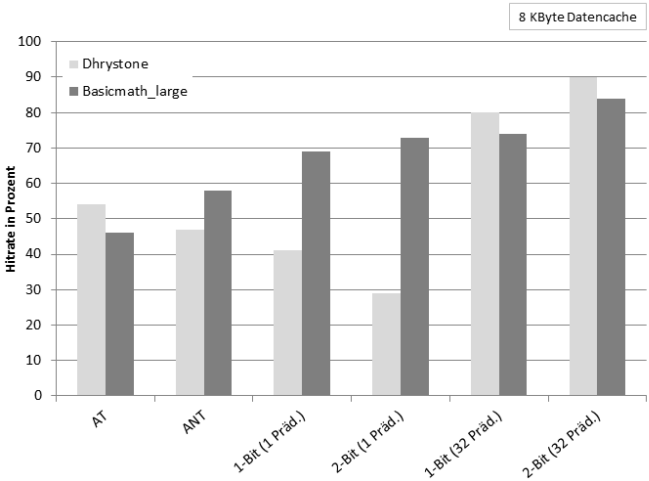


Abbildung 5.5.: Vergleich der Trefferquoten der adaptiven Sprungvorhersage bei 8 kB Datencache.

Es wird ein Xilinx *Virtex-5* (XC5VLX110T) (XUPV5) FPGA zur Bewertung der adaptiven Sprungvorhersage eingesetzt. Es wird eine Kachel der InvasIC Architektur verwendet. Dementsprechend ist nur der Kachel-lokale *Transaction Level Modeling* (TLM) vorhanden. Der Prozessor wird mit 80 MHz Taktfrequenz betrieben. Die Multiplikations- und Divisionseinheit sind vorhanden. Die Fließkommaeinheit ist deaktiviert.

5.3.1.1. Performanz

Abbildung 5.5 zeigt die Trefferquoten für beide *Benchmarks* mit 8 kB Datencache. Es ist zu erkennen, dass *Always Taken* und *Always not Taken* jeweils entgegengesetzt auf die beiden Benchmarks reagieren. Aufgrund seiner Struktur erzeugt der *Dhrystone* bei den einfachen dynamischen Prädiktoren sehr großes *Aliasing*. Der *Basicmath Large* scheint jedoch relativ einfach vorhersagbar zu sein, so dass die Trefferquoten hier bereits auf über 70 % verbessert werden. Sobald 32 Prädiktoren verwendet werden kann die Trefferquote auf bis zu 90 % gesteigert werden.

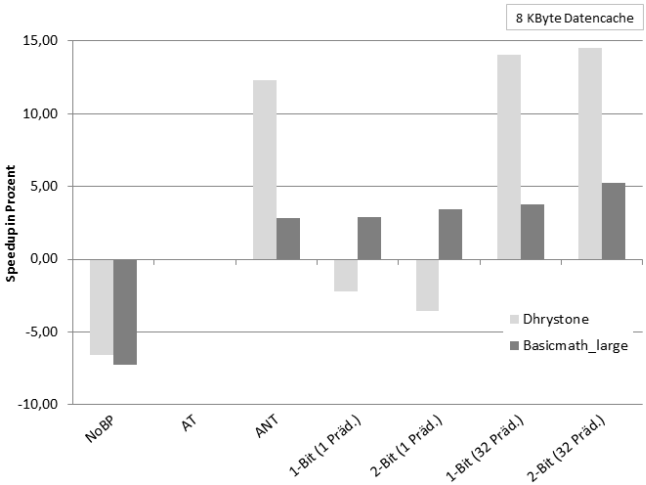
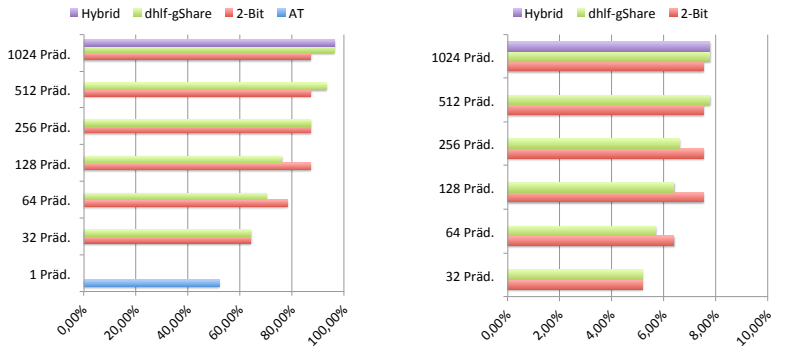


Abbildung 5.6.: Vergleich der Beschleunigung der Ausführungszeit durch die adaptive Sprungvorhersage bei 8 kB Datencache.

Die Ausführungszeiten werden auf Basis der *Always Taken* Vorhersage miteinander verglichen. In Abbildung 5.5 ist zu kennen, dass die *Always not Taken* Vorhersage eine kürzere Ausführungszeit erreicht, obwohl die Trefferquoten beim *Always Taken* leicht besser sind. Ein deutliche Verbesserung der Ausführungszeit wird vor allem mit 32 Prädiktoren bei der Ausführung des *Dhrystone* Benchmarks erreicht.

Im nächsten Testlauf wird der Hybridprädiktor evaluiert. Es werden hierzu der *Dhrystone* und der neuere Coremark Algorithmus eingesetzt. Beim *Dhrystone* ist das gleiche *Aliasing* wie im vorigen Versuch zu erkennen. Der 2-Bit Prädiktor sättigt bei 87 %. Um diese Trefferquote zu erreichen bzw. um diese zu verbessern bedarf es 512 bzw. 1024 Prädiktoren. In dieser Konfiguration ist es möglich 93 % bzw. 97 % Trefferquote zu erreichen. Hier wird jetzt auch der Hybrid-Prädiktor interessant, da er in der Lage ist zum DHLF *gShare* Prädiktor umzuschalten, sobald dieser die höheren Trefferquoten erzielt. Da der Hybrid-Prädiktor die gleiche Trefferquote wie der DHLF *gShare* Prädiktor erreicht, kann davon ausgegangen werden, dass die Auswahl des Meta-Prädiktors sehr früh getroffen wird.



(a) Trefferquote **(b)** Beschleunigung

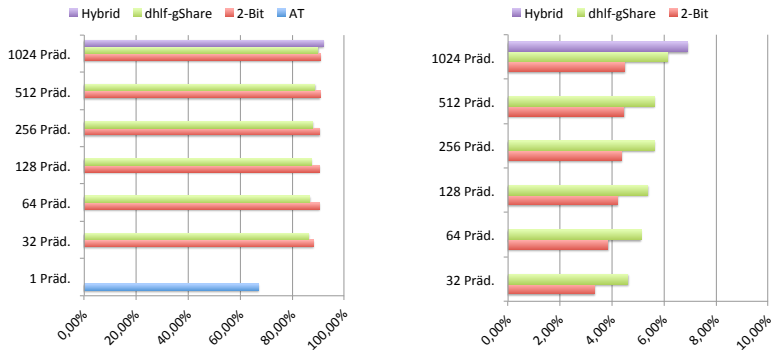
Abbildung 5.7.: Trefferquote (links) und Ausführungszeit (rechts) der adaptiven Sprungvorhersage bei Ausführung des Dhrystone für 2-Bit, DHLF *gShare* und Hybrid-Prädiktor mit verschiedenen Prädiktorzahlen.

Es ist jedoch anzumerken, dass sich die Ausführungszeit nicht im gleichen Maße verkürzt wie sich die Trefferquote verbessert. Dies hat mehrere Gründe: Das Anhalten bzw. Flushen der Pipeline ist nicht teuer, da die Daten sehr schnell aus dem nahen *Static Random-access Memory* (SRAM) geholt werden können. Außerdem ist der Prozessor der einzige Master am Bus, sodass er immer sofort auf den Speicher zugreifen kann. Diese Effekte relativieren sich, sobald der *a*-Core wie in den folgenden Versuchen als Mehrkernprozessor in der gekachelten Vielkernarchitektur mit entferntem Speicher eingesetzt wird.

Bei der Ausführung des Coremark Algorithmus zeigt sich ein anderes Verhalten: Der 2-Bit Prädiktor ist immer leicht besser als der DHLF *gShare* Prädiktor. Besonders interessant ist hier, dass bei geringer Anzahl an Prädiktoren kein *Aliasing* erkennbar ist.

5.3.1.2. Ressourcenverbrauch

Zur Bewertung der Effizienz ist es zunächst wichtig den Platzverbrauch auf den FPGA zu bestimmen. Tabelle 5.3 zeigt die Syntheseergebnisse. Es



(a) Trefferquote **(b)** Beschleunigung

Abbildung 5.8.: Trefferquote (links) und Ausführungszeit (rechts) der adaptiven Sprungvorhersage bei Ausführung des Coremark für 2-Bit, DHLF *gShare* und Hybrid-Prädiktor mit verschiedenen Prädiktorzahlen.

zeigt sich, dass 1-Bit und 2-Bit Prädiktor sehr effizient realisiert werden können. Für das Prädiktormodul mit DHLF *gShare* und 32 Prädiktoren ist ein leicht erhöhter Ressourcenbedarf von etwas mehr als 12 % der benötigten Ressourcen notwendig. Bei 256 bzw. 1024 Prädiktoren wächst der Ressourcenverbrauch, insbesondere der Register, um bis zu 50 % an.

In den vorigen Versuchen war ein unterschiedliches Verhalten für die beiden Benchmarks erkennbar. Während sich beim *Coremark* keine Verbesserung gezeigt hat, profitiert der *Dhrystone* deutlich von mehr Prädiktoren. Bei den ausgewählten Daten steigt die Trefferquote deutlich um 26 % bzw. 52 % im Vergleich zum Design mit 32 Prädiktoren. Der zusätzliche Ressourcenaufwand beträgt 31 % bzw. 51 %. Lediglich der Hybrid-Prädiktor übersteigt das Ressourcenbudget deutlich ohne bei den Ausführungszeiten der Programme einen Vorteil zu bringen. Dies liegt in diesem Fall vor Allem daran, dass *Slice Register* mit sehr viel Ansteuerungslogik anstatt einer sehr viel effizienteren BRAM Realisierung zum Einsatz kommen.

Bei Betrachtung der Ausführungszeit zeigt sich, dass sich der Einsatz der zusätzlichen Ressourcen nicht in jedem Fall lohnt. Speziell in diesem vergleichsweise einfachen Szenario mit nur einem Kern ist es je nach An-

	Prädiktoren PHT Größe	Slice Registers	Slice LUTs	BRAM
a-Core Originaldesign	—	9533	14844	28
Always not Taken (ANT)	—	9563	14746	28
Always Taken (AT)	—	9545	14492	28
2-Bit inkl. stat. Präd.	32	9628	15009	28
gShare inkl. statischen Prädiktoren	32	9960	15237	28
	1024	11041	20764	
Meta-Prädiktor inkl. dynamischen Prädiktoren	32	11168	21778	27
a-Core mit modularen Prädiktoren	32	10355	19301	
	256	11165	23354	
	1024	13597	26303	
a-Core Hybrid-Prädiktor inkl. modularen Prädiktoren	1024	15632	46103	

Tabelle 5.3.: Vergleich der Syntheseergebnisse der adaptiven Sprungvorhersage bei einer konstanten maximalen Frequenz von 120 MHz.

forderungen nicht unbedingt notwendig die Ausführungszeit um etwa eine Sekunde zu verkürzen. Die maximal erreichbare Frequenz bleibt in den verschiedenen Varianten konstant, so dass die zu erwartende Ausführungszeit nicht beeinflusst wird.

5.3.1.3. Optimale Parameter für den DHLF *gShare* Prädiktor

Beim DHLF *gShare* Prädiktor lassen sich drei Parameter frei einstellen. Interessant sind hier vor allem die Größen der Lernphasen und Intervalle, sowie die Wiederholungen der konstanten *Branch History Register* Länge bevor die *Local Minimum Avoidance* aktiviert wird. Je kleiner diese ist, desto häufiger wird das erlernte bzw. die gefundene *Branch History Register* Länge verworfen bzw. an den Programmablauf angepasst. Hierdurch werden auch die erlernten Informationen im *Pattern History Table* öfter verworfen, was wiederum weitere Lernphasen und potentiell mehr falsche Vorhersagen zur Folge hat. Während der Evaluation haben sich folgende Parameter als sinnvoll herausgestellt:

- Intervallgröße: 8.000 bedingte Sprünge
- Lernphase: 8.000 bedingte Sprünge
- *Local Minimum Avoidance*: 1.000 Wiederholungen bzw. 80.000 bedingte Sprünge

5.3.2. Mehrkern- und Vielkernszenario

Zur Evaluation in der gekachelten Architektur kommen unterschiedliche *Benchmarks* zum Einsatz. Auf Basis der Kategorien der *MiBench* Suite [65] wird jeweils ein Vertreter jeder Kategorie zur Evaluation herangezogen:

- **Industrie / Automotive:** Basicmath
- **Consumer Devices:** Mad
- **Office:** Stringsearch
- **Netzwerk:** Dijkstra
- **Sicherheit:** *Advanced Encryption Standard* (AES) (Rijndael)
- **Telekommunikation:** *Fast Fourier Transform* (FFT)
- **System:** Dhrystone, Coremark

Die Laufzeit der *Benchmarks* wird auf maximal eine Minute begrenzt, da eingehende Tests gezeigt haben, dass bereits genügend Zeit für die Lernphasen zur Verfügung steht und somit der zeitliche Rahmen für die Evaluation begrenzt werden kann.

5.3.2.1. Anzahl der Prädiktoren

In den vorigen Szenarien konnte bereits festgestellt werden, dass beim *Dhrystone* die Vorhersagen mit wenig, beispielsweise 32 Prädiktoren, deutliches *Aliasing* zeigen. Es soll zunächst anhand der zusätzlichen Algorithmen festgestellt werden, inwieweit dieses Verhalten auch auf die anderen Algorithmenklassen zutrifft. Abbildung 5.9 zeigt die Trefferquoten für diese Testläufe. Beim Dhrystone ist der Effekt am Stärksten ausgeprägt, was in der Natur des Algorithmus liegt. Beim Dijkstra und Stringsearch ist der Mehrwert des Einsatzes von mehr Prädiktoren nicht sehr deutlich.

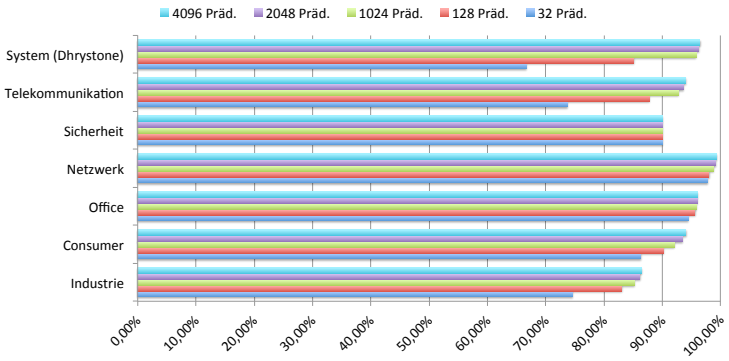


Abbildung 5.9.: Trefferquote der adaptiven Sprungvorhersage *gShare*-Prädiktor mit verschiedenen Prädiktorzahlen und Algorithmen.

Einzig beim AES Algorithmus ist kein *Aliasing* erkennbar. Der Grund hierfür liegt im relativ einfachen Aufbau bzw. sich über die Ausführung des Programms nicht verändernden Ablauf des Kontrollflusses. Der 2-Bit Prädiktor hat, wie in den letzten Versuchen schon festgestellt, eine recht ähnliche Charakteristik und reagiert jedoch nicht so stark auf die Vergrößerung der Prädiktorzahl, so dass bei den komplexeren Kontrollflüssen meist eine größere Anzahl Prädiktoren für das gleiche Vorhersageergebnis benötigt wird.

5.3.2.2. Intervallgröße

Eine weitere wichtige Größe ist die Intervallgröße, bei der sich ein deutlicher Einfluss auf die Genauigkeit der Vorhersage zeigt. Je größer das Intervall ist, desto größer ist der Programmteil, der in die Bewertung einfließt. Es ist jedoch zu beachten, dass Kontextwechsel bei großen Intervallen deutlich weniger gut abgebildet werden. Es ist also bei der Intervallgröße ein *Trade-off* zwischen möglichst großer Abdeckung des Programms und guter Anpassbarkeit anzustreben. Abbildung 5.10 zeigt die Trefferquoten für den *Dhrystone* Algorithmus bei unterschiedlichen Intervallgrößen von

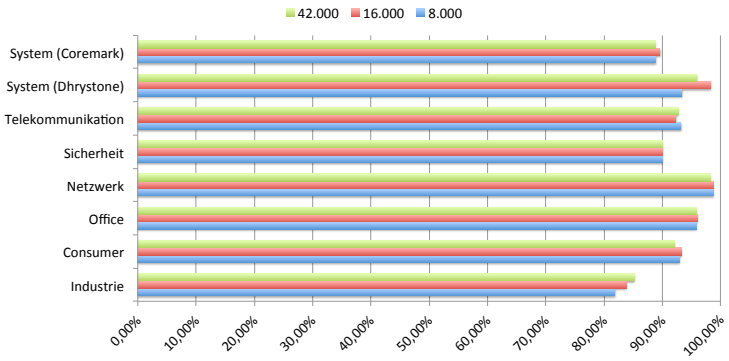


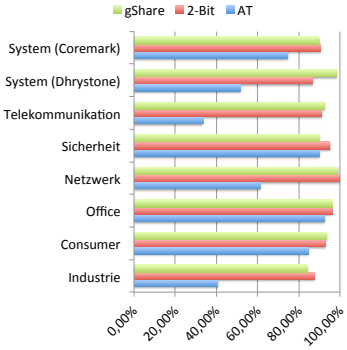
Abbildung 5.10.: Einfluss der Intervallgröße auf die Trefferquote beim *gShare* Prädiktor mit 1024 Prädiktoren bei Ausführung des *Dhrystone* Algorithmus.

8.000, 16.000 und 42.000. Einzig der *Basimath* profitiert vom sehr großen Intervall.

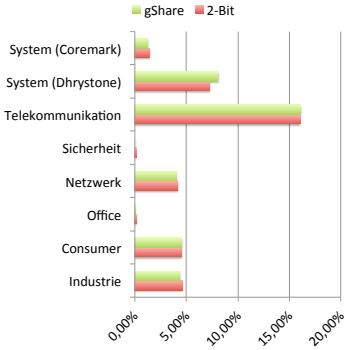
Intervalllänge	Basicmath Trefferquote in %	Dhrystone Trefferquote in %
8.000	81,79	72
16.000	83,76	95,81
42.000	85,2	98,5
128.000	84,66	96,4

Tabelle 5.4.: Vergleich der Trefferquoten für verschiedene Intervallgrößen des *gShare* Prädiktors für den *Basicmath* und den *Dhrystone* Algorithmus.

Tabelle 5.4 verdeutlicht jedoch, dass es selbst für *Basicmath* und *Dhrystone* nicht sinnvoll ist die Intervallgröße weiter zu vergrößern. Nach der starken Verbesserung der Trefferquote von 16.000 auf 42.000 Prädiktoren folgt bei der Vergrößerung des Intervalls auf 128.000 eine leichte Verschlechterung gegenüber der Trefferquote bei 16.000 Prädiktoren, was den Realisierungsaufwand nicht rechtfertigt. Es kann also ein Mittelweg zwischen kurzfristig hoher Trefferquote und Adaptivität gefunden werden, indem 16.000 anstatt bisher 8.000 Prädiktionen als Intervallgröße gewählt wird.



(a) Trefferquote in Prozent.



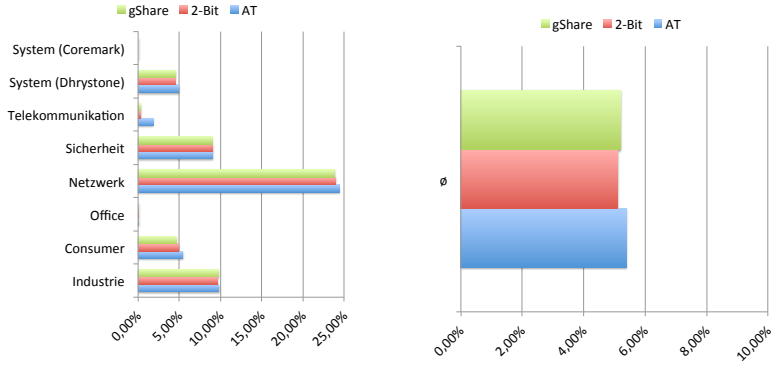
(b) Beschleunigung in Prozent.

Abbildung 5.11.: Trefferquote (links) der adaptiven Sprungvorhersage mit *Always Taken*, 2-Bit und *gShare*-Prädiktor und Performanzsteigerung (rechts) des 2-Bit und *gShare*-Prädiktors im Vergleich zum *Always Taken* Prädiktor.

5.3.2.3. Evaluation mit lokalem Speicher

Die *Always Taken* Vorhersage erreicht im Schnitt Trefferquoten von etwa 60 %. Vereinzelt ist es jedoch auch möglich, dass dieser statische Prädiktor eine ähnliche Trefferquote von teilweise mehr als 90 % wie ein aufwändigerer dynamischer Prädiktor erreicht. Abbildung 5.11 verdeutlicht dies bei einzelnen Algorithmen. Gründe hierfür liegen im einfachen Aufbau der Algorithmen. In diesen speziellen Fällen handelt es sich um *for* Schleifen, die nicht verschachtelt und so sehr einfach vorhersagbar sind. Die Vorteile der adaptiven Sprungvorhersage sind hier auf etwa 0,2 % begrenzt.

Der Netzwerk (Dijkstra) Algorithmus zeigt das erwartete Verhalten. Hier erreicht die *Always Taken* Vorhersage Trefferquoten von etwas mehr als 60 %, während die adaptiven Algorithmen weit über 90 % erreichen. Der Grund hierfür liegt in der hohen zeitlichen und räumlichen Lokalität des Codes. Es ist jedoch zu beachten, dass die zeitliche Beschleunigung der Ausführung vergleichsweise gering ausfällt. Der *Dhrystone* profitiert am Stärksten von der Komplexität der adaptiven Sprungvorhersage. Hier ist auch zu erkennen, dass der *gShare* Prädiktor noch einen Vorteil bei



(a) Relative Änderung der Ausführungszeit.

(b) Durchschnitt aller Algorithmen.

Abbildung 5.12.: Relative Änderung der Ausführungszeit der Algorithmen im Einzelnen (links) und durchschnittliche relative Änderung der Ausführungszeit aller Algorithmen (rechts) bei Zugriff auf globalen DDR im Vergleich zum Zugriff auf den lokalen TLM.

der Trefferquote und bei der Ausführungszeit erzielt. Ähnlich verhält es sich auch beim Industrie- und Telekommunikationsalgorithmus, wobei nur bei der FFT eine deutliche Beschleunigung der Laufzeit von etwa 16 % festzustellen ist. Dies liegt daran, dass die Daten bei vielen Fehlvorhersagen aus dem Speicher nachgeladen werden müssen. Insgesamt kann festgestellt werden, dass die Laufzeit des Algorithmus, wenn er exklusiven Zugriff auf den Speicher hat, nicht im gleichen Maße wie die Trefferquote skaliert. Da ein Prädiktor nicht immer die beste Trefferquote erzielt, unterstreicht dies auch den Mehrwert des Hybridprädiktors, der im letzten Abschnitt evaluiert wurde.

5.3.2.4. Einfluss der Speicherarchitektur

Nachdem die Daten im vorigen Abschnitt noch im TLM abgelegt waren, werden diese jetzt im entfernten globalen DDR abgelegt, auf den über das invasive NoC zugegriffen werden kann. Dies erhöht die Latenz der Zugriffe.

Aufgrund der Begrenzung der Ressourcen auf dem Xilinx ML605 FPGA können jedoch nur zwei Kacheln realisiert werden. Die Taktfrequenz beträgt 50 MHz. Daher kann sich der Anstieg der Latenz nicht wie im großen invasiven System bemerkbar machen. Man bekommt jedoch schnell einen Eindruck, dass es Sinn macht auf effiziente Ausnutzung der vorhandenen Resourcen zu achten, da sich die Ausführungszeit selbst bei exklusivem Zugriff auf die benachbarte Kachel bereits um 5,4 % verlängert.

Abbildung 5.12b zeigt die durchschnittliche Verlängerung der Ausführungszeit für alle in diesem Abschnitt betrachteten Algorithmen. Die Ausführungszeit erhöht sich um etwa 5 %. Es ist außerdem bereits erkennbar, dass der *gShare* im Durchschnitt am effizientesten arbeitet und die wenigsten Speicherzugriffe benötigt. Würde man die invasive Architektur jetzt hoch skalieren und das NoC unter Last testen, würden die Laufzeitunterschiede deutlich stärker ins Gewicht fallen.

Betrachtet man die Algorithmen im Einzelnen, wie in Abbildung 5.12a gezeigt, ist festzustellen, dass die Anzahl der Speicherzugriffe deutlichen Einfluss auf die Laufzeit hat. Wobei bei Ausführung von Office, Coremark und Telekommunikation minimale bis keine Verlangsamung zu erkennen ist, benötigt der Netzwerk Algorithmus (Dijkstra) fast 25 % mehr Ausführungszeit. Beim Coremark liegt dies vor allem daran, dass nur 2 kB Daten verwendet werden, die komplett im Cache abgelegt werden können.

Prädiktor	DDR	1 Router	3 Router	5 Router	7 Router	9 Router
2-Bit	Basis	0,4 %	1,2 %	3,6 %	10,8 %	32,4 %
gShare	Basis	0,6 %	1,8 %	5,4 %	16,2 %	48,6 %

Tabelle 5.5.: Durchschnittliche extrapolierte Performanzsteigerung des 2-Bit und *gShare* Prädiktors bei größerer Entfernung vom Prozessor zum Speicher im Vergleich zur *Always Taken* Vorhersage.

Wie bereits angesprochen konnte für diese Auswertung nur eine 2×1 Architektur realisiert werden. Somit kann nur auf die jeweils benachbarte Kachel zugegriffen werden. Um die Erhöhung der Laufzeit zu verdeutlichen, wird extrapoliert wie sich die Latenz zu einer weiter entfernten Kachel verhalten wird. In Tabelle 5.5 sind die extrapolierten Ergebnisse dargestellt. Es ist zu erkennen, dass die Unterschiede mit der Entfernung des Speichers deutlich anwachsen. Je weiter der Speicher vom Rechenkern entfernt ist,

desto größer wird die Latenz im Netzwerk. Es werden hier fünf Zyklen pro Router angenommen. Dementsprechend verlängert sich die Ausführungszeit mit der Anzahl der Speicherzugriffe auf den entfernten Speicher. Als Ausgangspunkt werden die durchschnittlichen Zahlen dieser Evaluation verwendet. Es ist offensichtlich, dass die effizientere Vorhersage mit größerer Entfernung deutlich Ausführungszeit einsparen kann. Es ist außerdem anzumerken, dass aufgrund der Beschränkungen der Implementierung keine weiteren Algorithmen parallel ausgeführt werden konnten. Wäre dieses Szenario möglich, würden sich die Ausführungszeiten weiter verschlechtern. Je mehr Rechenkerne zur Verfügung stehen und parallele Applikationen im System ablaufen, desto mehr Verkehr entsteht im NoC und die Zugriffe auf den Speicher werden weiter verzögert. Hierbei kann davon ausgegangen werden, dass die Last für jeden weiteren Rechenkern deutlich zunimmt und vor allem die Anfragen am DDR deutlich verzögert werden, was wiederum die Latenz deutlich erhöht. Die Ergebnisse der Exploration sind in Tabelle 5.6 dargestellt.

Prädiktor	DDR	1 Router	3 Router	5 Router	7 Router	9 Router
2-Bit	Basis	3,6 %	11,2 %	21,2 %	38,4 %	77,2 %
gShare	Basis	4,6 %	14,4 %	27,8 %	52,0 %	108,6 %

Tabelle 5.6.: Durchschnittliche extrapolierte Performanzsteigerung des 2-Bit und *gShare* Prädiktors unter Last bei größerer Entfernung vom Prozessor zum Speicher im Vergleich zur *Always Taken* Vorhersage.

5.3.2.5. Bewertung der Effizienz

In den vorangegangenen Abschnitten wurde, neben der Trefferquote zur Bewertung der Prädiktoren, hauptsächlich die Performanz also die Ausführungszeit betrachtet. Für eine umfangreiche Bewertung der adaptiven Sprungvorhersage muss jedoch der zusätzliche Ressourcenverbrauch mit in Betracht gezogen werden. Ergänzend zu den Ausführungen zu den einzelnen Prädiktoren ist anzumerken, dass sich die maximale Frequenz nicht ändert, die Betriebsfrequenz also konstant bleibt.

Im Vergleich zum originalen LEON3 Design nimmt der Platzbedarf pro Prozessor bereits ab 32 Prädiktoren leicht zu (3 %). Hier überwiegt noch

die adaptive Logik, die erst ab 128 Prädiktoren von den *Pattern History Table* Ressourcen überlagert wird (9,59 %). Bei 4096 Prädiktoren ist ein Ressourcenmehraufwand von 41,53 % notwendig.

Im Vergleich zu den Auswertungen unter Last in der InvasIC Architektur ergibt sich ab 1024 Prädiktoren ein leichter Effizienzvorteil, da davon ausgegangen werden kann, dass der Speicher in der verteilten Architektur im Durchschnitt mindestens drei Router entfernt ist. Wenn in der Architektur auf dem NoC sehr viel Last ist, erhöht sich die Effizienz deutlich. Andererseits können auch mehr Prädiktoren unter Erzielung einer positiven Bilanz für die Effizienz eingesetzt werden. Im Falle von 3 Routern und 1024 Prädiktoren eine Effizienzsteigerung von 4,81 %. Im Falle von 5 Routern und 2048 Prädiktoren eine Effizienzsteigerung von 7,56 %. Im Falle von 7 Routern und 4096 Prädiktoren eine Effizienzsteigerung von 10,47 % bzw. 67,07 % mit 9 Routern.

Somit handelt es sich bei der adaptiven Sprungvorhersage um ein effizientes Design, insbesondere für verteilte Mehrkern- und Vielkernarchitekturen mit gemeinsamem Speicher und Kommunikationsressourcen.

5.4. Zusammenfassung

Die adaptive Sprungvorhersage baut auf, aus dem Stand der Technik bekannten, statischen und dynamischen Prädiktoren auf. Die Vorhersagen werden zum ersten Mal überhaupt modular realisiert, da sie bisher teilweise nur konzeptionell oder simulativ veröffentlicht wurden. Insbesondere die effiziente ressourcensparende Kombination der ausgewählten Prädiktoren erweitert die bisher vorgestellten Konzepte. Als Erweiterung zum Stand der Technik ist es weiterhin möglich adaptiv zwischen den Vorhersagen umzuschalten. Der Anwendungsentwickler kann zur Designzeit die Einstellungen vornehmen, die der Compiler und das Laufzeitsystem dynamisch, also zur Laufzeit, anpassen können. Es wird somit erstmals möglich zwischen den Konfigurationsmöglichkeiten umzuschalten. Hierzu wird zur Instruktionssatz Architektur des *a*-Core eine kompatible Instruktion hinzugefügt. Darüber hinaus kann die Sprungvorhersage selbstadaptiv den optimalen Arbeitspunkt, also den Prädiktor mit dem zum jeweiligen Zeitpunkt besten Vorhersageergebnis, zur Laufzeit auswählen. Somit kann

sichergestellt werden, dass die Ressourcen effizient genutzt werden. Dies ist insbesondere der Fall, weil die dynamischen Prädiktoren die beiden statischen Vorhersagen (*Always Taken* und *Always not Taken*) nutzen und vor allem der Meta-Prädiktor die beiden dynamischen Vorhersagen bimodal und *gShare* nutzt. Somit wird zum ersten Mal ein modularer selbstadaptiver Hybridprädiktor realisiert, der das Ergebnis der einzelnen Vorhersagen sogar verbessern kann.

Hinsichtlich der Effizienz kann festgestellt werden, dass sich der Einsatz der komplexeren Prädiktoren, die mit einem teilweise deutlichen Ressourcenmehraufwand realisiert werden müssen, vor allem dann lohnt, wenn Speicherzugriffe aufgrund von Fehlvorhersagen vergleichsweise teuer sind. Daher wird bei der Evaluation der adaptiven Sprungvorhersagen, nachdem zunächst nur eine Kachel betrachtet wurde, die InvasIC Architektur skaliert, so dass der Speicher in einer entfernten Kachel liegt und über das NoC zugegriffen werden muss. Dies führt eine zusätzliche Latenz bei der Ausführung von Anwendungen in einer größeren Architektur ein. Hierbei ist erkennbar, dass sich der Mehraufwand bereits ab einem Abstand von durchschnittlich drei Routern im Netzwerk aufhebt und ab dieser Konfiguration insgesamt eine Effizienzsteigerung festgestellt werden kann. Diese steigt mit dem Abstand des ausführenden Rechenkerns zum globalen Speicher exponentiell an, so dass vor allem entfernte Rechenkerne sehr deutlich von der Effizienzsteigerung profitieren.

6. Laufzeitadaptivität anhand des adaptiven Cache

In diesem Kapitel wird der adaptive Cache vorgestellt. Dieser wird erstmals als Hardwarebeschreibung auf der ersten Cacheebene in der Prozessorarchitektur verankert. Dies gibt den Anwendungsentwicklern die Möglichkeit den *Level 1* (L1) Cache zur Designzeit zu konfigurieren. Der Compiler bzw. das Laufzeitsystem können die Parameter, wie beispielsweise Cachegröße, Assoziativität oder die Cachestrategie, dynamisch anzupassen.

6.1. Cache- und bandbreitengewahre Programmierung

Um den Anforderungen heterogener Architekturen gewachsen zu sein, sollten Anwendungen Informationen zu ihren Anforderungen bereitstellen, um einen verbesserten Durchsatz an Applikationen, eine verbesserte Speicherbandbreite oder auch Energieeffizienz zu erreichen. Unterschiedliche Anwendungsszenarien, wie beispielsweise mehrfach auflösende Bildverarbeitung oder dynamische adaptive hyperbolische Simulationen [13, 57], erfordern ein Umdenken von statischer Ressourcenallokation hin zu dynamischer Ressourcenzuweisung. Durch diese sich zur Laufzeit verändernden Anforderungen der Applikationen, sogenannter Phasen, ist es vorteilhaft unterschiedliche Hardwarekonfigurationen zur Verfügung zu stellen. Da der Speicher und vor allem dessen Hierarchie ein hauptsächlicher Flaschenhals ist, werden diese Eigenschaften anhand der Cachearchitektur, insbesondere

dem L1 Cache, untersucht. Numerische Applikationen, wie beispielsweise die Matrix Multiplikation (MMul), sind üblicherweise

- für eine bestimmte Cachezeilenlänge optimiert,
- nicht in der Lage veränderliche Parameter zu berücksichtigen,
- und nicht in der Lage die Hardware an die jeweiligen Anforderungen anzupassen.

Diese Anforderungsanalyse des adaptiven Caches auf Basis der *Invasive Computing* (InvasIC) Architektur wurde auf der *Applied Reconfigurable Computing* (ARC) vorgestellt [TSV+14].

6.1.1. Applikationsanforderungen

Um die jeweiligen Anforderungen der Applikationen einschätzen zu können, werden aus der Literatur bekannte Algorithmen auf ihre speicherbezogenen Anforderungen untersucht [141]:

Bandbreitenlimitierte Applikationen: Typische Speicherzugriffsmuster für bandbreitenlimitierte Applikationen sind Datenströme, Matrizen und im Allgemeinen relative kleine Berechnungen pro Speicherzugriff (BS) Verhältnisse für aktuelle Architekturen. Für ein Skalarprodukt [118] müssen zwei Vektoren geladen werden, um dann, unter der Annahme, dass die Ergebnisse im Register gespeichert werden, in zwei Multiplikationen berechnet zu werden. Mit der abschließenden Addition zum Wert im Register ergibt sich $BS = 2 : 2 = 1$. Matrizen werden häufig in der Bildverarbeitung für die Kantenerkennung genutzt [28]. Ausgehend von einer 2D Matrixoperation, die zweite Ableitung mit einer Matrixgröße von 3×3 berechnet, müssen fünf Werte geladen werden, wobei jedem Ladevorgang eine Berechnung folgt, wobei der jeweilige Wert im Register zur Verfügung stehen soll. Unter der Annahme, dass die Werte im Cache wieder verwendet werden können, führt dies zu einem Berechnungen pro Speicherzugriff Verhältnis $BS = 6 : 5 \approx 1,2$.

Berechnungsbegrenzte Algorithmen: Hier gibt es beispielsweise die nummerische Quadratur von berechnungsintensiven Funktionen [22]. Mit häufiger Auswertung solcher Funktionen mit sehr vielen Instruktionen ($n >> 1$) mit Formeln zur Quadratur höherer Ordnung sind diese Algorith-

men eindeutig durch die Berechnungen und nicht durch die Speicherzugriffe begrenzt, weil davon ausgegangen werden kann, dass alle zur Berechnung notwendigen Daten in den L1 Cache passen.

Latenzbegrenzte Algorithmen: Unvorhersehbare Speicherzugriffe passieren vor allem mit interaktiven Berechnungen, wie beispielsweise Steuerung, Bildbearbeitung und vor allem räumlichen iterativen Lösern [136]. Da die Speicherzugriffe zufällig auftreten, ist es unwahrscheinlich, dass diese Algorithmen die optimalen Einstellungen des Cache ausnutzen können. Daher hängt das jeweilige Berechnungen pro Speicherzugriff Verhältnis sehr stark vom jeweiligen Speicherzugriffsmuster ab, das nicht vorhergesagt werden kann.

6.1.2. Dynamische Ablaufplanung

Es wird ein neuer Ansatz vorgestellt, der es ermöglicht Einstellungen in der Hardware, insbesondere dem L1 Cache, zur Laufzeit vorzunehmen.

Diese konnten bisher nur statisch, also zur Designzeit, ausgenutzt werden. Es soll untersucht werden inwieweit die Anforderungen der Applikation an den L1 Cache weitergegeben werden können. Interessant erscheint es hier zunächst den Cache vergrößern oder verkleinern zu können. Zusätzlich soll die Möglichkeit bestehen mit höherer oder geringerer Assoziativität zu arbeiten. Die Verdrängungsstrategien werden zu einem späteren Zeitpunkt betrachtet.

Die Daten im Cache sollen zusammen mit den zusätzlichen Informationen in einem Modul gespeichert werden. Hierdurch soll es möglich werden die Größe und die Assoziativität des Cache zu ändern ohne wesentlichen Mehraufwand bei der Verwaltung der Daten hervorzurufen oder die Transparenz in der Verwaltung der Daten aufzulösen. Daneben soll es eine weitere Einheit geben, die sich ausschließlich um die Trefferverwaltung und das entsprechende Nachladen der Daten im Cache kümmert. Cachekohärenz wird interessant sobald der Cache in einer Mehrkernarchitektur zum Einsatz kommt und die Strategie verändert wird. Hierdurch wird auch ein neues Feld bezüglich der Zuordnung der Ressourcen für die jeweiligen Rechenkerne eröffnet. Des Weiteren ist es auch möglich den L1 Cache nicht

mehr ausschließlich privat sondern auch gemeinsam, also geteilt zwischen Rechenkernen zu verwenden.

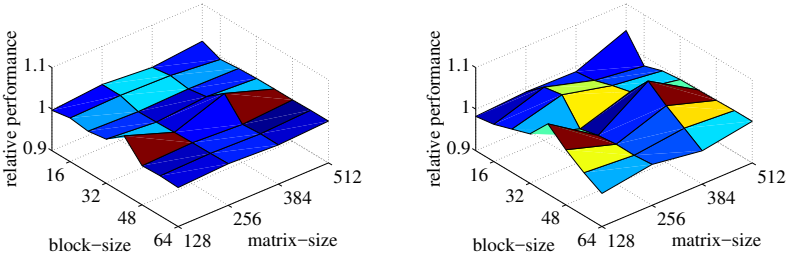
Es wird das invasive Programmierparadigma *invade*, *infect* und *retreat* genutzt und um die Mechanismen zur Invasion der Speicherhierarchie erweitert. Hierdurch können die Parameter der Caches an die applikationsspezifischen Anforderungen angepasst werden. Wichtig ist, dass sich die Anwendung zu jeder Zeit in einem sicheren Zustand befindet und gegebenenfalls Schnittstellen bezüglich der Cachekohärenz für geteilten und globalen Speicher zur Verfügung gestellt werden. So soll auch verhindert werden, dass die Ausführung schlechter, also langsamer als die mittlere Performanz der Applikation werden kann.

6.1.3. Variation der Parameter

In einer ersten Studie wird der Einfluss der Cacheeinstellungen auf die Performanz untersucht. Hierzu wird eine geblockte Matrix Multiplikation (MMul) genutzt. Es kommen zunächst jeweils statische Konfigurationen des adaptiver Prozessor (a-Core) zum Einsatz, die auf dem Xilinx *Virtex-5* (XC5VLX110T) (XUPV5) realisiert werden. Der $16\,kB$ Instruktionscache wird mit zwei Sätzen zu je $8\,kB$ konfiguriert. Der gleichgroße Datencache wird mit vier Sätzen zu je $4\,kB$ realisiert. Somit können die verschiedenen Parameter für diese Studie zur Designzeit definiert werden. Diese Konfiguration stellt zugleich die Basis für die Vergleiche in den folgenden Unterabschnitten dar.

6.1.3.1. Adaptiver Instruktionscache

Die Codegröße ist für eine numerische Berechnung wie die Matrix Multiplikation (MMul) vergleichsweise klein, wobei andere Programme, wie die numerische Quadratur, deutlich größer sein können. Allein hierdurch kann festgestellt werden, dass es vorteilhaft ist die Cachegröße als Parameter des Caches variieren zu können. In einem Versuch wird der Instruktionscache halbiert, in dem ein Satz des Caches deaktiviert wird. Dies hat bei der MMul fast keine Auswirkung ($< 1\,\%$) auf die Ausführungszeit. Dies führt zu weiteren verfügbaren Speicherressourcen auf der ersten Cacheebene, die



(a) Zwei Sätze mit $8\,kB$ pro Satz. **(b)** Ein Satz mit $16\,kB$.

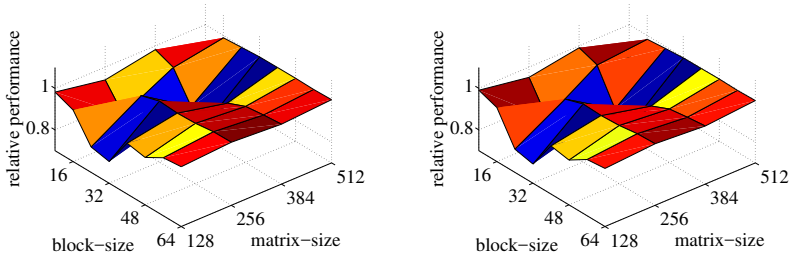
Abbildung 6.1.: Relative Performanz bei Nutzung eines $16\,kB$ Datencaches mit zwei Sätzen mit jeweils $8\,kB$ pro Satz (links) und einem Satz mit $16\,kB$ (rechts).

beispielsweise zum Datencache hinzugefügt oder alternativ zum Sparen von Energie abgeschaltet werden können.

6.1.3.2. Adaptiver Datencache

Applikationen wie die Matrix Multiplikation (MMul) sind darauf ausgelegt die vorhandenen Cache- und Speicherressourcen optimal auszunutzen. Dies wird vor allem erreicht, indem die Cachezugriffe effizient erfolgen ohne auf die jeweilige Konfiguration des Caches zu achten. Zusätzlich zur reinen Ausführungszeit wird auch die Iteration der äußeren Schleife mit der Basiskonfiguration verglichen.

Zunächst wird die Größe des Datencaches mit $16\,kB$ konstant gelassen. Dies ermöglicht zwei Konfigurationen: Die Anzahl der Sätze mit einer dementsprechend veränderlichen Größe der Sätze, wie in Abbildung 6.1 gezeigt. Abbildung 6.1a zeigt die relative Performanz mit zwei Sätzen halber Größe. Es ist eine leichte Performanzsteigerung von ungefähr 3 % zu erkennen. Insbesondere ist zu erkennen, dass einzelne Größen der MMul sehr von der Cachekonfiguration profitieren. In Abbildung 6.1b ist zu erkennen, dass ein einzelner Satz eine um etwa 7 % gesteigerte Performanz aufweist. Auch hier profitieren einzelne Blockgrößen der Matrix besonders von der Anpassung der Cachekonfiguration. Verglichen mit dem



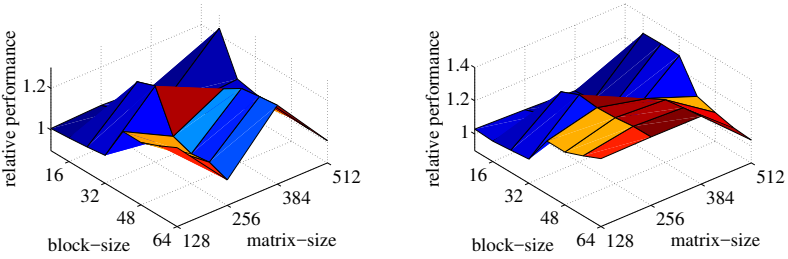
(a) Vier Sätze mit $2\,kB$ pro Satz. **(b)** Zwei Sätze mit $4\,kB$ pro Satz.

Abbildung 6.2.: Relative Performanz bei Nutzung eines $8\,kB$ Datencaches mit vier Sätzen mit jeweils $2\,kB$ pro Satz (links) und zwei Sätzen mit jeweils $4\,kB$ pro Satz (rechts).

vorigen Versuch ist in etwa eine Verdopplung der Performanzsteigerung zu erkennen.

Bei dem zweiten Versuch wird die Cachegröße halbiert. Es stehen jetzt, wie in Abbildung 6.2 zu erkennen, $8\,kB$ Cachespeicher zur Verfügung. Dieser wird in vier Sätze mit jeweils $2\,kB$ also der halben Satzgröße realisiert (siehe Abbildung 6.2a). Alternativ kann die Anzahl der Sätze halbert (zwei Sätze) werden und die Satzgröße entsprechend verdoppelt werden (siehe Abbildung 6.2b). Im Vergleich der relativen Performanz kann jeweils in etwa die gleiche Performanz erreicht werden. Auch im Gegensatz zum ersten Versuch ist hier kein Einfluss auf die Performanz zu erkennen. Daraus kann abgeleitet werden, dass es nicht ausschlaggebend ist, ob bei konstanter Cachegröße die Anzahl der Sätze oder die Größe der Sätze halbiert wird. Dieses Wissen wird bei der Konzeption der Cachearchitektur wieder aufgegriffen. Es ist somit sinnvoll die Anzahl der Wege zu variieren, ohne die Performanz negativ zu beeinflussen. Dies ist insbesondere vielversprechend für größere Applikationen. Dieser Zusammenhang wird im folgenden Abschnitt 6.2 genauer untersucht.

Beim letzten Versuch wird die Anzahl der Sätze konstant gelassen. Jetzt kann die Größe der Sätze variiert werden. Abbildung 6.3 zeigt die Performanzsteigerung der beiden Szenarien. Interessanterweise profitieren verschiedene Blockgrößen unterschiedlich stark von der Vergrößerung des Cache bzw. einige fast gar nicht. Während die Performanzsteigerung bei



(a) Vier Sätze mit $8\,kB$ pro Satz. **(b)** Vier Sätze mit $16\,kB$ pro Satz.

Abbildung 6.3.: Relative Performanz bei Nutzung eines $32\,kB$ Datencaches mit vier Sätzen mit jeweils $8\,kB$ pro Satz (links) und einem $64\,kB$ Datacache mit vier Sätzen mit jeweils $16\,kB$ pro Satz (rechts).

einer Matrixgröße von 256 für eine Blockgröße von 24 und 32 bereits bei Verdopplung des Cache deutlich erkennbar ist (siehe Abbildung 6.3a), muss der Cache nochmals verdoppelt werden, damit größere Blöcke auch von der vierfachen Cachegröße profitieren (siehe Abbildung 6.3b). Dies unterstreicht die applikationsspezifische Abhängigkeit der Performanz und den Bedarf die Cachearchitektur während der Ausführung an die Applikation anzupassen.

6.2. Simulative algorithmische Optimierung der Cacheparametrierung

Nachdem erkannt wurde, dass die Eingabeparameter deutlichen Einfluss auf die Performanz der Algorithmen bei einer bestimmten Parameterwahl des Cache haben, soll zunächst untersucht werden, welche Eingabeparameter für die Algorithmen gewählt werden müssen, um den Algorithmus optimal an die Prozessorarchitektur anzupassen und eine minimale Ausführungszeit zu erreichen. Diese Parametrisierung der Algorithmen wurde zu den *IEEE Computer Architecture Letters* (CAL) eingereicht [**?**].

6.2.1. Räumliche und zeitliche Lokalität

Die Lokalität kann in die räumliche und die zeitliche Lokalität unterteilt werden. Sie beschreibt die Muster der Speicherzugriffe und bildet die Basis für die Wirksamkeit von Caches.

Bei räumlicher Parallelität wird häufig auf benachbarte Speicheradressen zugegriffen. Hier kann eine längere Cachezeile Vorteile bringen. Es ist jedoch zu beachten, dass dies nicht in jedem Szenario funktioniert, da eine längere Cachezeile auch ein längere Ladezeit bedeutet, die bei zu häufiger Verdrängung die Vorteile aufwiegen können. Hier ist der Einsatz unterschiedlicher an die Anwendung angepasster Zeilenlängen sinnvoll.

Die zeitliche Lokalität beschreibt den häufigen Zugriff auf die gleichen Daten. Die ist beispielsweise bei Schleifenkörpern der Fall, da diese im Normalfall sehr häufig ausgeführt werden. Bei der Matrixmultiplikation tritt die zeitliche Lokalität vor allem in der Ergebnismatrix $\mathbf{C}$ auf, da immer auf diese addiert wird.

Diese Lokalitätseigenschaften sollen zunächst ausgenutzt werden, um die Fehltrefferrate im Cache zu senken.

6.2.2. Methoden zur Verringerung der Fehltrefferrate

Es wird ein kurzer Überblick drüber gegeben welche Methoden zur Verringerung der Fehltrefferrate genutzt werden können. Es wird davon ausgegangen, dass die Speicherzugriffe einem bestimmten Muster folgen. Durch die geschickte Anordnung der Daten und der Anpassung des Zugriffsmusters soll die Fehltrefferrate verringert werden, durch die konsequente Anwendung des *Hardware/Software Co-Design* Gedankens in Verbindung mit der Anpassung der Cacheparameter. Weiterführende Informationen sind in einschlägiger Literatur [167, 121, 25, 26] zu finden. Die entsprechenden Codebeispiele sind in der Abschlussarbeit von Tobias Dörr [Dör15] dargestellt.

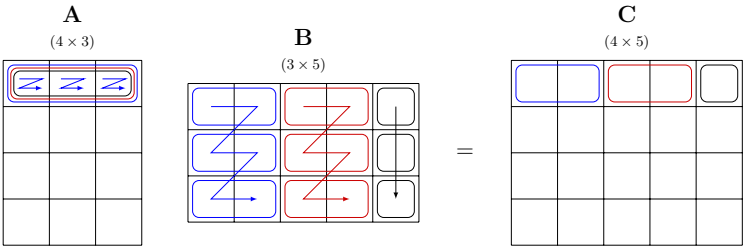


Abbildung 6.4.: Angedeuteter Ablauf einer Matrizenmultiplikation bei normaler Speicherung der Matrizen und eindimensionalen Blöcken in k-Richtung. Hier dargestellt ist eine Blockgröße von $b = 2$. Pfeile kennzeichnen den zeitlichen Ablauf des Zugriffsmusters bei der Bearbeitung eines Blocks. Markierungen in C geben an, von welchen Blöcken das jeweilige Element abhängig ist.

6.2.2.1. Matrix Multiplikation

Durch die Zugriffsreihenfolge auf die Matrizen A (Zeile) und B (Spalte) zur Berechnung der C bietet es sich an die Matrix B als transponierte Matrix B^{T} zu speichern, um auch zeilenweise zugreifen zu können.

$$\mathbf{B} = \begin{pmatrix} b_{11} & \cdots & b_{1p} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{np} \end{pmatrix} \text{und} \mathbf{B}^{\mathrm{T}} = \begin{pmatrix} b_{11} & \cdots & b_{n1} \\ \vdots & \ddots & \vdots \\ b_{1p} & \cdots & b_{np} \end{pmatrix} \tag{6.1}$$

Darüber hinaus wird die Blockbildung auf zwei verschiedene Arten verwendet. Erstens werden Blöcke b innerhalb einer Zeile der Matrix B, wie in Abbildung 6.4 dargestellt, so gebildet, dass das die Länge des Blockes der Zeilenlänge in Wörtern (L_W) entspricht.

Zweitens werden zweidimensionale Blöcke gebildet, so dass auch die Matrix A passend zur Länge der Cachezeile L_W, wie in Abbildung 6.5 dargestellt, geblockt gelesen werden kann, um hier Verdrängung zu vermeiden und räumliche Parallelität zu nutzen.

Des Weiteren sollen, wie bereits angesprochen, die Vorteile der räumlichen Parallelität ausgenutzt werden. Da die Zugriffe auf die Matrizen B und C

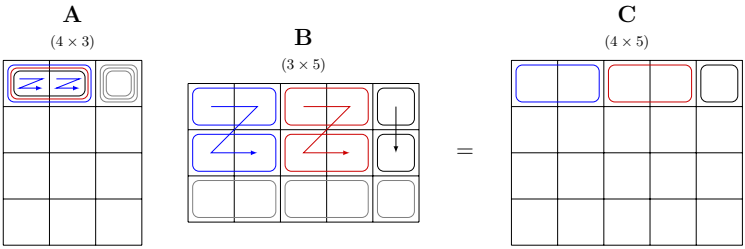


Abbildung 6.5.: Angedeuteter Ablauf einer Matrizenmultiplikation bei normaler Speicherung der Matrizen und zweidimensionalen Blöcken in j- und k-Richtung. Hier dargestellt sind Blöcke mit $b = 2$.

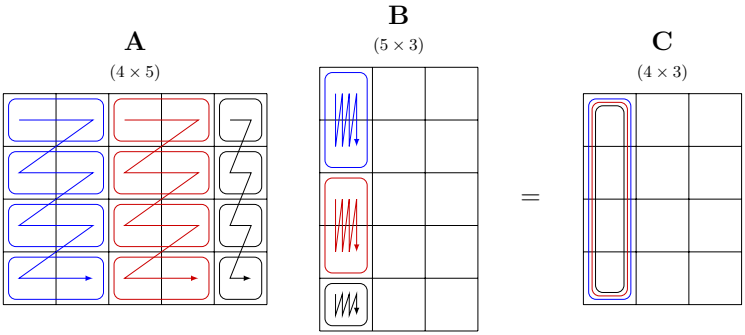


Abbildung 6.6.: Angedeuteter Ablauf einer Matrizenmultiplikation bei Speicherung von B und C als Transponierte sowie eindimensionalen Blöcken in j-Richtung. Hier dargestellt sind Blöcke mit $b = 2$.

jedoch spaltenweise ablaufen, ist es vorteilhaft diese, wie in Abbildung 6.6 dargestellt, als transponierte Matrizen B^{T} und C^{T} zu verwenden. Es ist auffällig, dass die zur Berechnung der Spalte bzw. transponierten Zeile der Matrix C auf jeweils mehrere Zeilen der Matrix A zugreifen muss.

Somit ist es sinnvoll die Berechnung der Zeilen der Matrix C, wie in Abbildung 6.7 dargestellt, auch in Blöcke zu unterteilen. Damit werden weniger Blöcke der Matrix A für die Berechnung benötigt. So kann die Verdrängung aus dem Cache weiter minimiert werden. Dementsprechend kann die jeweilige Blockgröße so gewählt werden, dass die Blockgröße zum

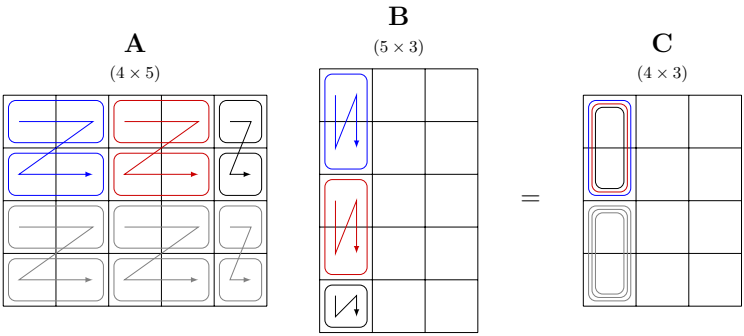


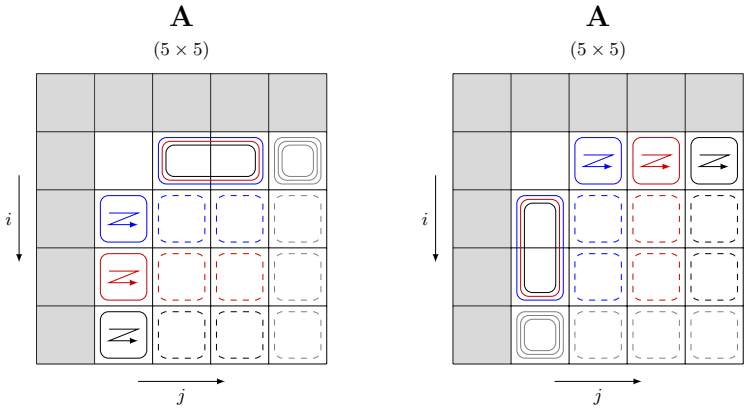
Abbildung 6.7.: Angedeuteter Ablauf einer Matrizenmultiplikation bei Speicherung von B und C als Transponierte sowie zweidimensionalen Blöcken in i- und j-Richtung. Hier dargestellt sind Blöcke mit $b = 2$.

einen der Länge der Cachezeile (L_W) aber zum anderen insgesamt auch der Anzahl der Zeilen im Cache (Anzahl der Cachezeilen (#l)) entspricht, um die Lokalität optimal ausnutzen zu können.

6.2.2.2. LU Zerlegung

Bei der LU Zerlegung soll ein ähnliches Vorgehen die Lokalität ausnutzen. Nach der Berechnung der Elemente der Matrix L werden die Elemente der Untermatrix, die sich durch das Streichen der ersten Spalten und Zeilen der Matrix A ergibt, aktualisiert. Dieser Teil des Algorithmus ist bestimmend für die zeitliche Komplexität, so dass der Fokus auf dessen Optimierung liegt.

Wie in Abbildung 6.8a zu erkennen, werden auch hier Blöcke gebildet, um die zeitliche Lokalität bei der Berechnung auszunutzen. Die Speicherung der einzelnen Elemente findet in der Reihenfolge der Zeilen statt, so dass auch hier räumliche Parallelität ausgenutzt werden kann. Um dementsprechend diese Vorteile auch bei der Umsetzung nutzen zu können, wird die Matrix A als Transponierte gespeichert, um fortlaufend auf die einzelnen Blöcke der Spalten zugreifen zu können.



(a) eindimensionale Blöcke in j-Richtung

(b) eindimensionale Blöcke in i-Richtung

Abbildung 6.8.: Angedeuteter Ablauf der LU-Zerlegung bei normaler Speicherung von A und bei Speicherung von A als Transponierte. Hier dargestellt sind Blöcke mit $b = 2$.

6.2.3. Maß für die Bewertung der Optimierung

Es ist wichtig ein geeignetes Maß für die Bewertung der Optimierung zu finden, da im Sinne des *Hardware/Software Co-Design*s der Algorithmus sowie auch die Hardware angepasst werden. Hierbei ist es dann nicht mehr ausreichend die Fehltrefferrate zu betrachten. Vielmehr muss die absolute Anzahl der Fehltreffer betrachtet und reduziert werden, um die Effizienz des Gesamtsystems steigern zu können. Abbildung 6.9 zeigt einen Versuch mit verschiedenen Blockgrößen und der entsprechenden Anzahl Fehltreffer. Es ist zu erkennen, dass die Fehltrefferrate vor allem durch die Anzahl der Referenzen bestimmt wird. Die Anzahl der Referenzen lässt sich jedoch zumeist nicht direkt auf die Cache Performanz schließen, so dass die Fehltrefferrate hier irreführend ist.

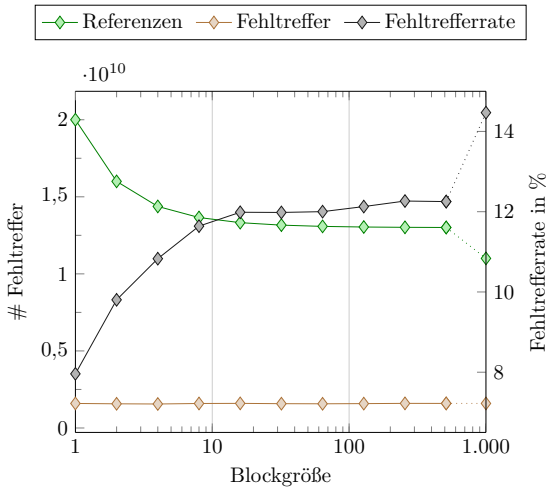


Abbildung 6.9.: Durch Cachegrind simulierte Anzahl der L1-Referenzen und -Misses für eine Matrizenmultiplikation mit $m = n = p = 1000$. Für die Parameter des L1-Datencaches gilt $C = 1024\,\mathrm{B}$, $Satzgrö„e(s) = 2$ sowie $Zeilenlänge(L) = 64\,\mathrm{B}$. Als Algorithmus kommt eine Implementierung mit zweidimensionalen Blöcken und normaler Speicherstruktur zum Einsatz.

Anstatt dessen soll die mittlere Speicherzugriffszeit pro Speicherzugriff, hier exemplarisch mit zwei Cacheebenen, genutzt werden [77]:

$$\underbrace{\text{Hit-Time}_{L1}}_{=:\,T_{H,L1}} + \text{Miss-Rate}_{L1} \cdot \Big(\underbrace{\text{Hit-Time}_{L1}}_{=:\,T_{H,L2}} + \text{Miss-Rate}_{L2} \cdot \underbrace{\text{Miss-Penalty}_{L2}}_{=:\,T_{M,L2}}\Big) \tag{6.2}$$

Es werden alle Speicherzugriffe betrachtet um die Problematik der Missrate in der Bewertung zu umgehen. Hierzu wird die mittlere Speicherzugriffszeit mit der Gesamtzahl der Speicherzugriffe multipliziert, woraus sich die gesamte Speicherzugriffszeit (T_{ges}) ergibt. Es muss auch beachtet werden, dass sich diese Zeit vergrößert, wenn über mehrere Cacheebenen auf den Hauptspeicher zugegriffen wird. Daher werden die Referenzen der jeweiligen Ebene unterschieden R_{Li}. Die Treffer werden als *Hit* (H_{Li}) und Fehltreffer als *Miss* (M_{Li}) abgekürzt:

$$
\begin{aligned}
T_{ges} &= R_{L1}\left(T_{H,L1} + \frac{M_{L1}}{R_{L1}}\left(T_{H,L2} + \frac{M_{L2}}{R_{L2}}\,T_{M,L2}\right)\right) \\
&= R_{L1}\,T_{H,L1} + M_{L1}\left(T_{H,L2} + \frac{M_{L2}}{R_{L2}}\,T_{M,L2}\right) \\
&= R_{L1}\,T_{H,L1} + R_{L2}\,T_{H,L2} + M_{L2}\,T_{M,L2}
\end{aligned}
\tag{6.3}
$$

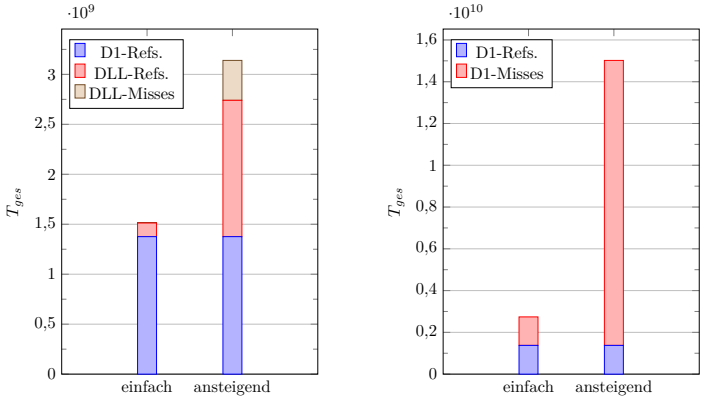
Bei zwei Cacheebenen entsprechen die Fehltreffer im *Level 2* (L2) Cache einem Zugriff im Hauptspeicher (*Main Memory* (MM)), in dem die Daten gespeichert sind. Daher lässt sich T_{ges} mit diesen Annahmen wie folgt darstellen:

$$
T_{ges} = R_{L1}\,T_{H,L1} + R_{L2}\,T_{H,L2} + M_{L2}\,T_{H,MM} \tag{6.4}
$$

Weiterhin muss beachtet werden, dass die Speicherzugriffszeit mit zunehmender Hierarchiestufe wesentlich größer wird. Abbildung 6.10 verdeutlicht die Auswirkung der Latenz auf die gesamte Speicherzugriffszeit. In Abbildung 6.10a ist die gesamte Speicherzugriffszeit für zwei Cacheebenen gegeben. Zunächst werden lediglich alle Cache- bzw. Speicherreferenzen aufgetragen, um diese im zweiten Schritt mit Faktor 5 bzw. 10 zu gewichten, um einen realen Eindruck von gesamte Speicherzugriffszeit zu bekommen. In Abbildung 6.10b wird auf den *Last Level Cache* (LLC) verzichtet, wodurch die gesamte Speicherzugriffszeit ohne Gewichtung unverändert bleibt, die gesamte Speicherzugriffszeit jedoch deutlich zunimmt, da alle Zugriffe auf den L1 Cache, die einen Fehltreffer hervorrufen, an der sehr langsamen Hauptspeicher weitergeleitet werden müssen.

6.2.4. Experimentelle Bestimmung der optimalen Cache Parameter

In diesem Unterabschnitt sollen die Cache Parameter auf Basis der bereits optimierten Algorithmen bestimmt werden. Diese Versuche sollen



(a) Gewichtung der gesamte Speicherzugriffszeit mit LLC.

(b) Gewichtung der gesamte Speicherzugriffszeit ohne LLC.

Abbildung 6.10.: Gesamte Speicherzugriffszeit T_{ges} für den Ablauf einer Matrizenmultiplikation mit $m = n = p = 500$, ungeblockt. Bei einfacher Gewichtung wird jeder Speicherzugriff unabhängig von der Hierarchiestufe, die ihn bedienen kann, mit einer Zeiteinheit gewertet, d.h. $T_{H,L1} = T_{H,L2} = T_{H,MM} = 1$. Bei ansteigender Wertung wird $T_{H,L1} = 1$, $T_{H,L2} = 10$ und $T_{H,MM} = 50$ angenommen. Cache-Parameter sind $C_{L1} = 2048\,\mathrm{B}$, $s_{L1} = 2$ und $C_{LL} = 16\,384\,\mathrm{B}$, $s_{LL} = 8$ bei $L_W = 128\,\mathrm{B}$.

Erkenntnisse liefern wie der Cache aufgebaut werden soll und auf welchen Parametern das Hauptaugenmerk liegen soll.

Alle Versuche werden mit 250×250 Matrizen durchgeführt. In jedem Versuch wird ein Parameter des L1 Caches verändert, während die anderen fest eingestellt bleiben. Für die folgenden Versuche wird das Werkzeug *Cachegrind* [11] auf Basis einer x86 Architektur mit 64 Bit Wortbreite verwendet. Es wird der Compiler der *GNU Compiler Collection* (GCC) mit den Einstellungen *Optimize even more* (-O2) und *Debug Information* (-g) genutzt.

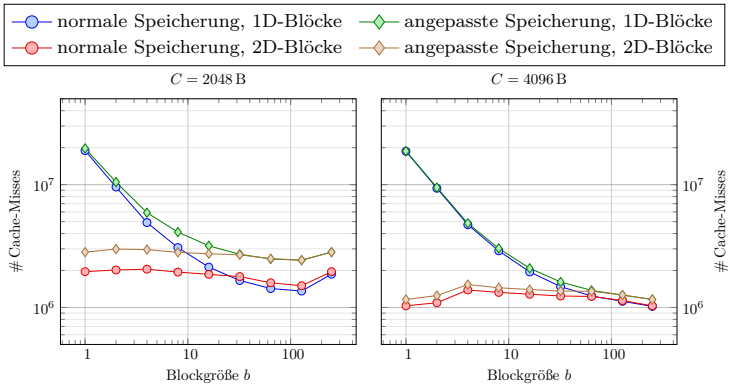


Abbildung 6.11.: Anzahl der Cache-Misses für eine Matrizenmultiplikation mit $n = 250$ und den Cache-Parametern $N = 4, l = 64\,\mathrm{B}$ bei verschiedenen Kapazitäten C.

6.2.4.1. Matrix Multiplikation

Beim ersten Versuch wird die Cachegröße (C) von 2048 Bytes zu 4096 Bytes geändert. Die Assoziativität s bleibt konstant bei 4 Wegen und die Zeilenlänge L_W bei 64 Byte. Der Versuch zeigt, dass die Anzahl der Fehltreffer bei der zweifach geblockten Variante (2D) im Vergleich zur nur einfach geblockten Variante (1D) deutlich weniger abhängig von der Blockgröße ist. Die 1D Variante ist bei der kleineren Cachegröße leicht besser als beim größeren Cache. Es ist außerdem zu erkennen, dass die Anzahl der Misses früher in die Sättigung geht. Es ist außerdem wahrscheinlich, dass die Anzahl der Fehltreffer bei einer Vergrößerung der Blockgröße weiter steigt.

Abbildung 6.12 zeigt die Anzahl des Cache Fehltreffer für zwei verschiedene Assoziativitäten $s = 4$ bzw. $s = 8$. Die Cachegröße ist jetzt fest mit 1024 Byte. Es ist erneut zu erkennen, dass die Anzahl der Fehltreffer bei der zweifach geblockten Variante fast unabhängig von der Blockgröße ist. Durch die Erhöhung der Assoziativität kann die Anzahl der Fehltreffer nochmals deutlich reduziert werden. Variante zwei hat hier leichte Vorteile.

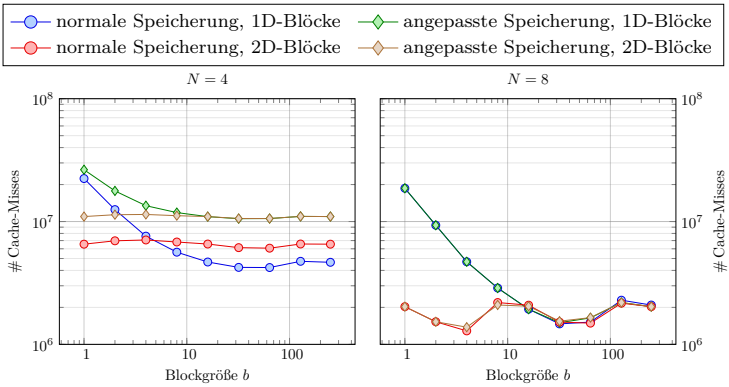


Abbildung 6.12.: Anzahl der Cache-Misses für eine Matrizenmultiplikation mit $n = 250$ und den Cache-Parametern $C = 1024\,\mathrm{B}, l = 64\,\mathrm{B}$ bei verschiedenen Assoziativitäten N.

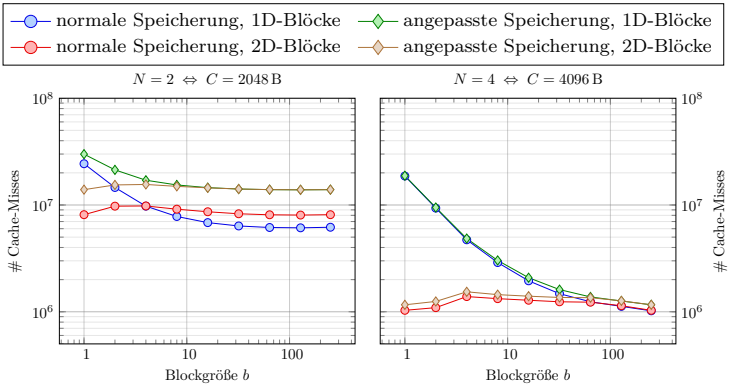


Abbildung 6.13.: Anzahl der Cache-Misses für eine Matrizenmultiplikation mit $n = 250$ und den Cache-Parametern $C/N = 1024\,\mathrm{B}, l = 64\,\mathrm{B}$, d.h. mit einer festen Kapazität pro Weg.

Beim letzten Versuch werden Cachegröße und Assoziativität so verändert, dass deren Verhältnis, in diesem Versuch $\frac{C}{s} = 1024\,Byte$, konstant bleibt. Wie in Abbildung 6.13 zu erkennen ist haben die zweidimensional

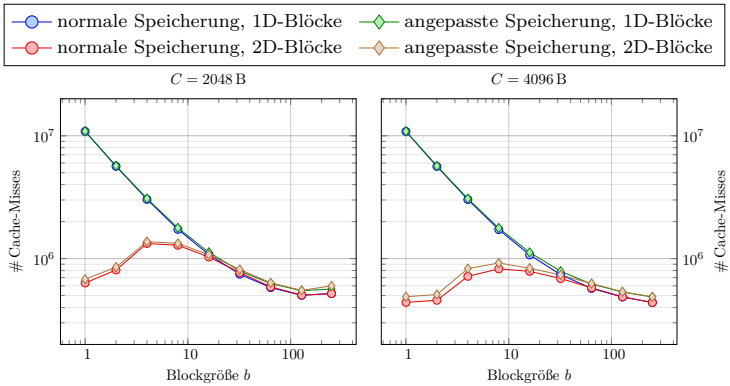


Abbildung 6.14.: Anzahl der Cache-Misses für eine LU-Zerlegung mit $n = 250$ und den Cache-Parametern $N = 4, l = 64\,\mathrm{B}$ bei verschiedenen Kapazitäten C.

geblockten Varianten wiederum die wenigsten Fehltreffer. Während die eindimensionale Variante sehr stark von der Blockgröße abhängt, zeigt die zweidimensionale Variante fast keine Abhängigkeit. Insgesamt hat auch hier die zweite Variante, ohne die spezielle Anordnung im Speicher, die geringste Anzahl Fehltreffer.

6.2.4.2. LU-Zerlegung

Bei der LU-Zerlegung wird die gleiche 250×250 Matrix wie bei der Matrix Multiplikation genutzt. Auch hier werden die gleichen drei Versuche durchgeführt. Zunächst wird die Cachegröße, wie in Abbildung 6.14 gezeigt, von 2048 Bytes zu 4096 Bytes verändert. Es ist zu erkennen, dass die zweifach geblockte Variante deutlich weniger Fehltreffer auslöst als die einfach geblockte Variante. Es ist jedoch interessant, dass Blockgrößen zwischen 4 und 128 bei einer Cachegröße von 4096 Byte in der zweifach geblockten Variante mehr Fehltreffer hervorrufen als bei einer Blockgröße von 2. Bei der einfach geblockten Variante fällt hingegen die Anzahl der Fehltreffer gleichmäßig ab. Möglicherweise kann hier durch eine Erhöhung der Blockgröße die Anzahl der Fehltreffer weiter reduziert werden. Bei

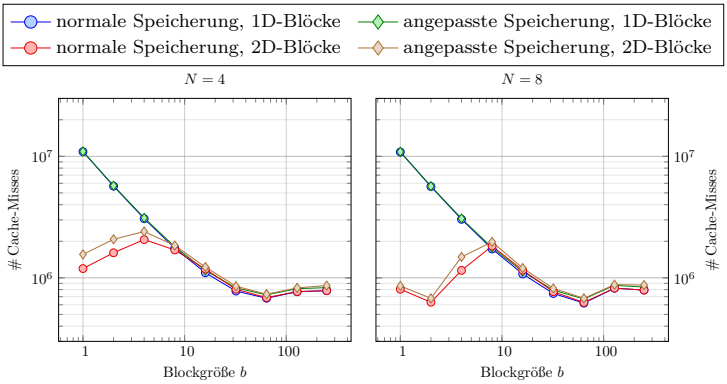


Abbildung 6.15.: Anzahl der Cache-Misses für eine LU-Zerlegung mit $n = 250$ und den Cache-Parametern $C = 1024\,\mathrm{B}, l = 64\,\mathrm{B}$ bei verschiedenen Assoziativitäten N.

einer Cachegröße von 2048 Byte ist dieser Effekt auch zu erkennen, wobei bereits ab einer Blockgröße von 64 weniger Fehltreffer auftreten.

Im zweiten Versuch wird die Assoziativität wieder bei einer festen Cachegröße von 1024 Byte von $s = 4$ zu $s = 8$ verändert. Wie in in Abbildung 6.15 zu erkennen, hat die zweifach geblockte Variante der LU-Zerlegung bei einer Blockgröße von weniger als $b = 8$ die geringere Anzahl an Fehltreffern. Jedoch ist auch hier zu erkennen, dass Blockgrößen zwischen $b = 4$ und $b = 64$ wiederum mehr Fehltreffer hervorrufen. Die minimale Anzahl der Fehltreffer ist für beide Assoziativitäten fast gleich.

Abschließend wird das Verhältnis von Cachegröße und Satzgröße in diesem Versuch konstant gelassen ($\frac{C}{s} = 1024\,\mathrm{Byte}$). Es ist interessant, dass für $s = 2$ die zweifach geblockte Variante bis zu einer Blockgröße von $b = 8$ weniger Fehltreffer hervorruft. Auch bei $s = 4$ ist zu erkennen, dass die zweifach geblockte Variante bis zu einer Blockgröße von $b = 32$ weniger Fehltreffer hervorruft. Insgesamt ist festzuhalten, dass die absolute Anzahl an Fehltreffern, die für $s = 4$ bereits bei einer Blockgröße von $b = 2$ erreicht ist, für $s = 2$ erst bei einer Blockgröße von $b > 32$ erreicht ist.

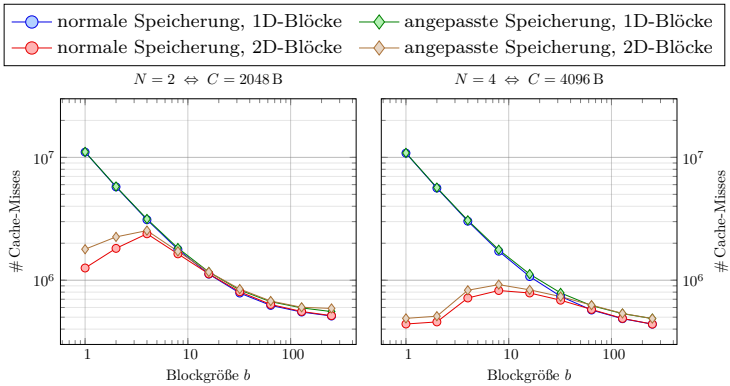


Abbildung 6.16.: Anzahl der Cache-Misses für eine LU-Zerlegung mit $n = 250$ und den Cache-Parametern $C/N = 1024\,\mathrm{B}, l = 64\,\mathrm{B}$, d.h. mit einer festen Kapazität pro Weg.

6.2.5. Schlussfolgerung

Insgesamt ist die zweifach geblockte Variante der Matrix Multiplikation und der LU-Zerlegung deutlich weniger abhängig von der gewählten Blockgröße. Es ist deutlich zu erkennen, dass die neben der zeitlichen Parallelität auch die räumliche Parallelität ausgenutzt wird. Die einfach geblockte Variante der Algorithmen erreicht die minimale Anzahl der Fehltreffer erst für eine relativ große Blockgröße, mit mindestens $b = 8$, zumeist jedoch $b > 32$.

Für die Konzeption des adaptiven Cache ist es vor allem interessant, dass die gleichzeitige Vergrößerung der Cachegröße (C) und der Satzgröße (s) eine deutliche Reduktion der Fehltreffer erzeugt, was bei der Veränderung einzelner Parameter nicht in dem Maße festgestellt werden kann. Dieses Ergebnis unterstreicht den Ansatz den Cache anstatt in Sätzen in Wegen zu interpretieren, um hier die Parallelschaltung der Wege ausnutzen zu können. So wird direkt die Assoziativität und durch die Verwendung der zusätzlichen Ressourcen auch die Cachegröße (C) verändert.

Dieser Ansatz wird im folgenden Unterkapitel vorgestellt.

6.3. Adaptive Cache Architektur

Die adaptive Cache Architektur soll generisch entwickelt werden, damit sie auf jeder Cacheebene in der adaptiven Prozessorarchitektur eingesetzt werden kann. Es wird angestrebt die Architektur plattformunabhängig zu entwickeln. Hierdurch soll verhindert werden, dass implementierungsspezifische Komponenten genutzt werden, wie beispielsweise die dynamische Rekonfiguration in *Field Programmable Gate Array*s (FPGAs). Damit kann die adaptive Cachearchitektur auch unabhängig vom *a*-Core in einer Prozessorarchitektur eingesetzt werden. Hierbei können die generischen Elemente durch realisierungsspezifische Komponenten, wie Speicher oder Kommunikationsnetzwerke, genutzt werden.

Da die Anforderungen an einen L1 Cache bezüglich der zeitlichen aber auch physichen Realisierung am Höchsten sind, wird in diesem Kapitel von einem L1 Cache gesprochen. Insbesondere die Ladezeit von einem Zyklus bei vorhandenen Daten, also einem Treffer im Cache, soll ermöglicht werden, um keine Performanz gegenüber gängigen Architekturen einzubüßen. Auf allen anderen Cacheebenen sind die Anforderungen an den Cache weniger hoch, so dass hier auch langsamer gearbeitet werden könnte. Deswegen wird der adaptive Cache mit den Anforderungen an einen L1 Cache entwickelt, der selbstverständlich auch auf höheren Cacheebenen eingesetzt werden kann. Die adaptive Cache Architektur wurde auf der *Architecture of Computing Systems* (ARCS) [TCO+16a] vorgestellt.

6.3.1. Modularisierung

Der adaptive Cache wird, wie in Abbildung 6.17 dargestellt, in mehrere Module unterteilt: Die Cachesteuerung, den Cachespeicher, die Speichersteuerung, den *Transceiver* und die Verteilungssteuerung. Damit sich dieser selbst verwalten kann sind die minimal notwendigen Verbindungen gezeigt, um die Parallelität des Caches voll ausnutzen zu können. Halbduplex Verbindungen können Lesen oder Schrieben, während Vollduplex Verbindungen gleichzeitig Lesen und Schreiben können. Während die Verbindungen zwischen Cache und der *Central Processing Unit* (CPU), dem Speicher bzw. dem Bus im halbduplex Betrieb ausgelegt werden können, werden alle cacheinternen Verbindungen vollduplex ausgelegt, damit die Module in

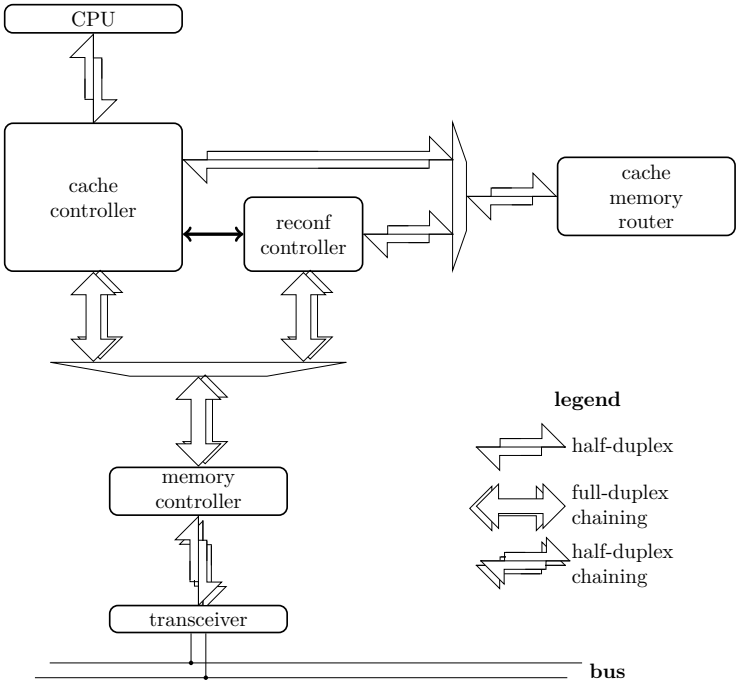


Abbildung 6.17.: Übersicht der einzelnen Module des adaptiven Caches: Cachesteuerung, Cachespeicher, Verteilungssteuerung, und *Transceiver*.

CPU Geschwindigkeit autarke Verwaltungsaufgaben übernehmen können. Im Folgenden werden die einzelnen Module vorgestellt:

Cachesteuerung: Die Cachesteuerung ist das zentrale Modul des adaptiven Caches. Sie koordiniert alle vorhandenen Module und synchronisiert die Abläufe. Neben den Standardabläufen werden alle Verwaltungstätigkeiten Richtung Speicher und Bus gesteuert. Außerdem wird die Adaption durchgeführt, die von der getrennten Steuerungseinheit verwaltet wird. Neben dem Ruhezustand sind diese konkreten Zustände möglich: Lesen, Schreiben, Spülen, Verwerfen und Anpassen. Insbesondere die Adaptionssequenz wird im Folgenden im Detail vorgestellt.

Cachespeicher: Im Cachespeicher werden die Cachezeilen, die Daten-, *Tag*- und Steuerbits basierend auf der jeweiligen Verdrängungsstrategie gespeichert. Der Cachespeicher besteht aus zwei Teilen: Ein Teil generiert die Informationen und Steuersignale die in den Cacheblöcken, dem eigentlichen Speicher, abgelegt werden.

Speichersteuerung: Die Speichersteuerung koordiniert die Zugriffsprioritäten und realisiert die Schreibpuffer. Auf diesem Weg muss sich die Speichersteuerung nicht um die Serialisierung des Datenstroms kümmern.

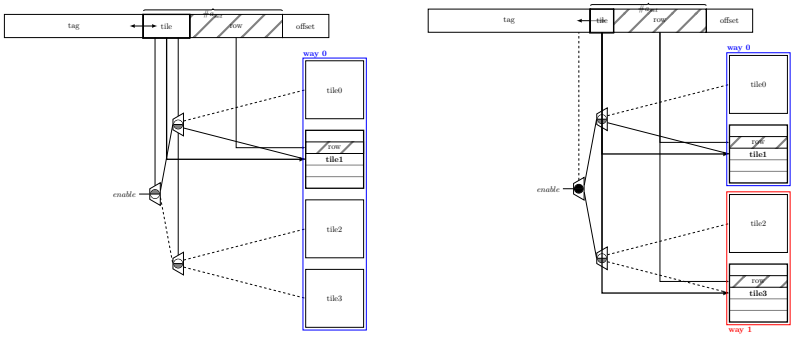
Transceiver: Der *Transceiver* realisiert das Busprotokoll und agiert als Schnittstelle zum Bus je nach Realisierung als *Advanced High-performance Bus* (AHB) oder *Advanced eXtensible Interface Bus* (AXI) Master.

Adaptionssteuerung: Die Adaptionssteuerung wird in einem einzelnen Modul realisiert. Dies ermöglicht es Änderung an der Parametrisierung des Caches vorzunehmen, ohne den Cachespeicher anpassen zu müssen. Konkret bedeutet das, dass die Veränderungen über die Cachesteuerung vorgenommen werden sollen, damit unnötige Zugriffe zur Sortierung der Daten im Cachespeicher vermieden werden können.

6.3.1.1. Cachespeicher

Im adaptiven Cache werden assoziative s-Wege Caches als s parallele Cachespeicher betrachtet. Dementsprechend kann jede Speicherstelle nur einmal pro Speicher vergeben werden. Anhand der Assoziativität wird dementsprechend ausgesucht in welchen Cachespeicher das Datum geschrieben wird. Sobald kein vollassoziativer Cache genutzt wird, können Adressbits verwendet werden, um die verschiedenen Speicher zu adressieren bis ein *Direct Mapped* Cache übrig bleibt. Diese s parallelen Cachespeicher werden im Folgenden Cacheblöcke oder kurz Blöcke genannt.

Abbildung 6.18 zeigt wie die Satzadresse (a_{Satz}) aufgeteilt ist. Diese besteht aus zwei Teilen: Einem statischen Teil, der Zeilenadresse (a_{Zeile}), die die Zeile in einem Cacheblock identifiziert, und einem dynamischen Teil, der Blockadresse (a_{Block}), die den entsprechenden Cacheblock im Weg adressiert. Die Multiplexer zur Ansteuerung der Cacheblöcke werden von der Blockadresse (a_{Block}) gesteuert. Wenn die Assoziativität, wie in Abbildung 6.18b dargestellt, erhöht wird, wird der Multiplexer auf der ers-



(a) *Direct Mapped* Cache. **(b)** 2-Wege Cache.

Abbildung 6.18.: Cachespeicher Adressierung und Netzwerk: Die schwarzen Pfeile zeigen die Abbildung der Adresse, die darunter liegenden Multiplexer die entsprechende Realisierung

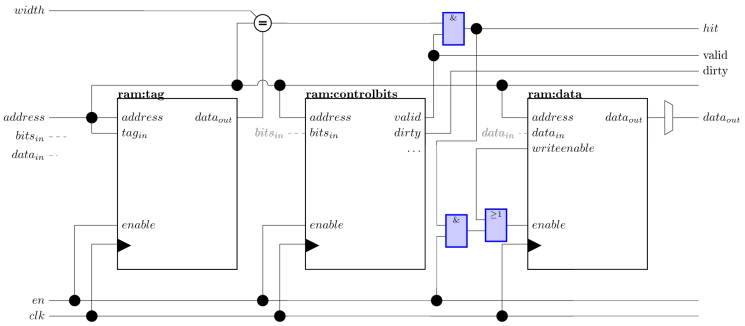


Abbildung 6.19.: Cacheblock: Tag-, Steuerbit- und Datenspeicher sind in individuellen Speichern realisiert.

ten Ebene durch die Reduktion der Blockadresse (a_{Block}) abgeschaltet, so dass beide Multiplexer der tieferen Ebene parallel angesteuert werden. Mit erhöhter Assoziativität kann dementsprechend auf ein Bit der Blockadresse (a_{Block}) verzichtet werden. Beim vollassoziativen Cache kann komplett auf die Blockadresse (a_{Block}) verzichtet werden. Dem entsprechend verkürzt sich auch die Satzadresse (a_{Satz}).

Abbildung 6.20.: *Cache Configuration Register* (CCR) bestehend aus der Satzgröße, der Zeilenlänge, der Anzahl der verfügbaren Cacheblöcke, den *Flags* und der Verdrängungsstrategie.

Es werden, wie in Abbildung 6.19 dargestellt, drei unabhängige Speicherblöcke für den Tag-, den Steuerbit- und den Datenspeicher genutzt. Der Tag- und der Datenspeicher enthalten jeweils Tag und Daten. Der Steuerbitspeicher enthält die *Valid* und *Dirty* Bits und die jeweilige Information über die Zuordnung der Cachezeile passend zur Verdrängungsstrategie.

Durch diese Realisierung können Zeileninhalte erneuert werden, ohne dass auf den Tagspeicher zugegriffen werden muss, da nur die Steuerbits neu geschrieben werden müssen. Falls besonders kleine Leistungs- oder Energiebudgets eingehalten werden müssen und leichte Performanzeinbußen tragbar sind, können die Zugriffe auf die drei Speicher serialisiert werden. Dies wird vor allem bei der partiellen Veränderung der Cachestrategie in Abschnitt 6.5 ausgenutzt.

6.3.2. Adaption

Der Cache besitzt folgende Möglichkeiten zur Adaption: Satzgröße (s), Cachegröße (C)=Satzgröße (s)·Anzahl der Sätze (#s), die Zeilenlänge (L) und die Verdrängungsstrategie.

6.3.2.1. *Cache Configuration Register* (CCR):

Alle Informationen über die Parametrisierung werden im *Cache Configuration Register* (CCR) gespeichert, das Teil der Cachesteuerung ist. Vor einer Adaption sollte der aktuelle Zustand gelesen werden, um sicher zu gehen, dass sich die Zielparametrisierung ausgehend von der aktuellen Parametrisierung realisieren lässt. Hier ist die Anzahl der verfügbaren Cacheblöcke beispielsweise begrenzend für die maximal mögliche Satzgröße. Theoretisch wäre es möglich jede natürliche Zahl abzubilden. Zur einfacheren Ansteuerung und Realisierung werden binäre Bäume verwendet, so dass

bei der Umsetzung des *Cache Configuration Register* (CCR) nur Zweierpotenzen beachtet werden müssen. Das CCR ist in Abbildung 6.20 gezeigt. Es besteht aus dem Logarithmus Dualis der Satzgröße und der Zeilenlänge. Anschließend folgt die Anzahl der verfügbaren Cacheblöcke in binärer Kodierung. Die gezeigten *Flags* beziehen sich hier auf *Write Allocate*, *Write Through*, *Write Back*. Abschließend wird die Verdrängungsstrategie angegeben.

6.3.2.2. Cachespeicher:

Mit den durch das CCR einstellbaren Parametern können, wie in Abschnitt 2.5 beschrieben, alle weiteren Parameter für den adaptiven Cache berechnet werden. Zum einfacheren Verständnis des Vorgehens bei der Adaption des Cachespeichers werden im Folgenden die Fälle der Veränderung einzelner Parameter besprochen. Es ist offensichtlich, dass im späteren Betrieb mehrere Parameter gleichzeitig verändert werden können. Abbildung 6.21 gibt einen Überblick über die veränderbaren Parameter, die Einfluss auf die Aufteilung des Cachespeichers haben und die jeweiligen Randbedingungen unter denen die Adaption erfolgen darf und die dafür notwendigen Kosten. Ziel ist es immer einen kostspieliges Spülen des Caches zu vermeiden.

Satzgröße (s): Die Satzgröße wird durch das Konzept der Cacheblöcke bestimmt. Bei einem *Direct Mapped* Cache können die Daten nur in einem Cacheblock stehen; bei assoziativen Caches in mehreren parallelen Cacheblöcken. Hieraus wird ersichtlich, dass Daten durch die Adaption in verbotenen Cacheblöcken stehen können. Dies muss bei der Adaption entsprechend berücksichtigt werden, woraus sich folgende Möglichkeiten ergeben:

Reduktion der Satzgröße (s): Bei einer Satzgröße von 2^n stehen 2^n valide Speicherstellen zur Verfügung. Wenn die Satzgröße von 2^n auf 2^m reduziert wird, ergeben sich nach der Reduktion 2^m valide Speicherstellen, so dass 2^{n-m} Speicherstellen nicht mehr valide sind. Da die 2^m Speicherstellen eine Teilmenge der Ausgangsparametrisierung ist, müssen die Daten nicht angefasst werden. Die anderen $2^n - m$ Daten müssen entweder an freie Speicherstellen im verbliebenen Speicher vorschoben oder in die nächste

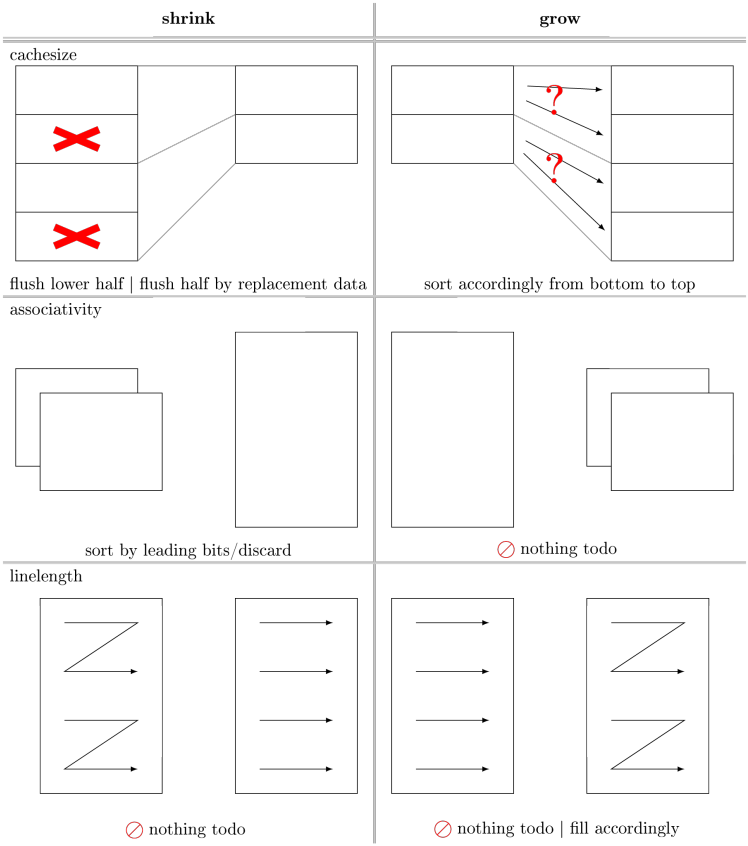


Abbildung 6.21.: Adaptionsmöglichkeiten.

Speicherhierarchieebene synchronisiert werden, um Datenverlust im Cache zu vermeiden.

Erhöhung der Satzgröße (s): Entsprechend der Argumentation bei der Reduktion, ergeben sich durch die Erhöhung der Satzgröße auf 2^m Speicherstellen 2^{m-n} neue Speicherstellen. Diese stellen zusätzliche valide Speicherstellen dar, die vorher nicht bearbeitet werden müssen, so dass hier bei der Verwaltung kein Mehraufwand entsteht.

Cachegröße (C): Die Cachegröße wird durch die Anzahl der verfügbaren Cacheblöcke bestimmt. Somit kann die Anzahl der Cachebloöcke beispielsweise reduziert werden um Energie zu sparen. Andererseits können auch mehr Cacheblöcke hinzugeschaltet werden, um beispielsweise die Assoziativität, die Cachegröße und damit letztendlich auch die Performanz zu erhöhen.

Auch hier spielt die Betrachtungsweise eines s-Wege assoziativen Cache als Cachedie zentrale Rolle, da dieser aus s parallelen Caches besteht. Konsequenterweise sollte die Ab- oder Hinzuschaltung von $s \pm m$ Caches auch in der Ab- bzw. Hinzuschaltung von $s \pm m$ Wegen resultieren. Sollte nach dem Prinzip von Albonesi et al. [3] nur die Vergrößerung des Speichers, nicht aber die Erhöhung der Assoziativität gewollt sein, muss darauf geachtet werden, dass 2^{m-s} Cacheblöcke benötigt werden, damit die Assoziativität konstant bleibt. Dies wird im Folgenden genauer betrachtet:

Reduktion der Wege: Es muss darauf geachtet werden, dass die Daten in einer validen Speicherstelle stehen. Wenn Wege abgeschaltet werden, müssen die Daten in den nicht mehr genutzten bzw. umgewidmeten Cacheblöcken umsortiert oder mit der höheren Ebene der Speicherhierarchie synchronisiert werden. Wenn der Cache allerdings im *Write Through* Modus arbeitet, kann dieser Schritt übersprungen werden, da die Daten bereits synchronisiert sind.

Reduktion der Sätze: Wenn die Anzahl der Sätze reduziert wird, entspricht das einer Reduktion der Assoziativität, da weniger valide Speicherstellen pro Weg zur Verfügung stehen. Dies entspricht dem Abschalten einzelner Cacheblöcke deren Daten, wie oben beschrieben zunächst synchronisiert werden müssen.

Erhöhung der Wege: Das Hinzufügen neuer Cacheblöcke als Wege bedarf keiner besonderen Behandlung, da davon ausgegangen werden kann, dass die Cacheblöcke leer sind, bzw. die Daten nicht valide sind, da diese zuvor, wie oben beschrieben, synchronisiert werden mussten.

Erhöhung der Sätze: Wie bei der Reduktion der Sätze beschrieben, kann auch hier davon ausgegangen werden, dass die Erhöhung der Wege zu einer Erhöhung der Assoziativität führt. Die Erhöhung der Sätze würde also eine Reduktion der Wege nach sich ziehen, wodurch, wie oben erklärt, sichergestellt werden muss, dass die Daten zunächst synchronisiert werden.

Zeilenlänge (L): Die Veränderung der Zeilenlänge funktioniert auf Basis der definierten Zeilenlänge, indem mehrere Zeilen zu einer virtuellen Zeile zusammengefasst werden, um beispielsweise längere *Burst* Zugriffe über den Bus zu ermöglichen. Die maximale virtuelle Zeilenlänge sollte kleiner sein als die Satzgröße, um unnötigen Steuerungsaufwand und potentiell auch Verdrängung über Satzgrenzen hinweg zu vermeiden. Dementsprechend muss bei der Nutzung dieser Fähigkeit nur auf die Einhaltung der minimalen Zeilenlänge geachtet werden.

Weiterführende technische Hintergründe sind in den Abschlussarbeiten von Maximilian Braun [Bra12] und Malte Vesper [Ves13] zu finden.

6.3.3. Bewertung der adaptiven Cache Architektur

Auf Basis der vorgestellten Parameter kann der adaptive Cache zur Laufzeit an die Anforderungen der Anwendung angepasst werden. Es muss jedoch vom Laufzeitsystem beachtet werden, dass bestimmte Einstellungen auf Basis der aktuellen Parametrierung günstiger oder teurer zu realisieren sind als andere. Dieses Kosten/Nutzen Verhältnis soll in den folgenden Versuchen bestimmt werden.

Es werden hierzu zunächst drei Hauptaspekte untersucht. Zuerst werden die für die jeweiligen Parametrisierungen notwendigen Ressourcen bestimmt. Dies ist insbesondere interessant, sobald die Cacheblöcke innerhalb einer Kachel der InvasIC Architektur geteilt werden können. Diese Betrachtungen werden im folgenden Abschnitt 6.4 vertieft. Anschließend wird der Einfluss des adaptiven Caches auf die Applikations- aber auch Cacheperformanz überprüft. Abschließend wird die Performanz und der Ressourcenverbrauch der adaptiven Cache Architektur in einem Fallbeispiel mit zwei Applikationen auf einer Kachel untersucht. Es wird ein *Virtex-5* (XC5VLX110T) (XUPV5) FPGA verwendet.

6.3.3.1. Ressourcenverbrauch

Zur Bestimmung der benötigten Ressourcen wird ein 8-Wege Cache mit 256, 512 und 1024 Byte pro Weg verwendet. Abbildung 6.22 zeigt den Logikverbrauch in *Slices* für den *Virtex-5* (XC5VLX110T) (XUPV5) FPGA.

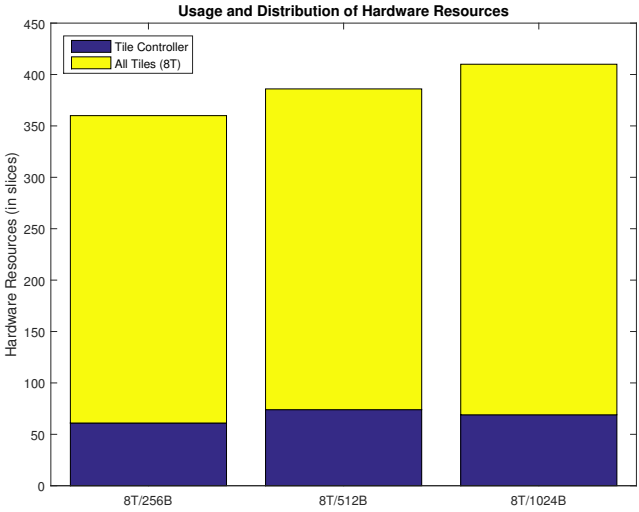


Abbildung 6.22.: Nutzung und Verteilung der Ressourcen für die Speichersteuerung und alle Cacheblöcke für einen 8-Wege Cache mit drei unterschiedlichen Cachegrößen (2048, 4096 und 8192 Byte).

Es ist bemerkenswert, dass der Logikverbrauch mit etwas über 50 Slices auch bei Vergrößerung des Cache weniger als 20 % der benötigten Ressourcen verbraucht. Unter der Vorgabe, dass die Cachegröße verdoppelt bzw. vervierfacht wird, steigt der Ressourcenverbrauch für die Speichersteuerung um nicht mal 10 Slices an. Auch bei der Betrachtung des gesamten Ressourcenverbrauchs fällt auf, dass die Ressourcen zur Ansteuerung von doppelt bzw. viermal so vielen Cacheblöcken lediglich 4 % bzw. 9 % mehr Slices benötigt werden.

Neben der Steuerlogik muss auch der eigentliche Cachespeicher realisiert werden. Dieser wird auf dem XUPV5 durch *BlockRam* (BRAM) realisiert. Wie im vorigen Kapitel erläutert, werden für jeden Cacheblock drei BRAMs zur Realisierung benötigt. Bei 8 Wegen ergibt das insgesamt 24 BRAMs. Dies ist notwendig, um die parallelen Zugriffe zur internen Verwaltung der Daten, die innerhalb des Caches im Vollduplexmodus benötigt werden, realisieren zu können. Darüber hinaus kann auch angenommen werden, dass unter der Maßgabe, dass die BRAMs 18 kB bzw. 36 kB groß sind, dass

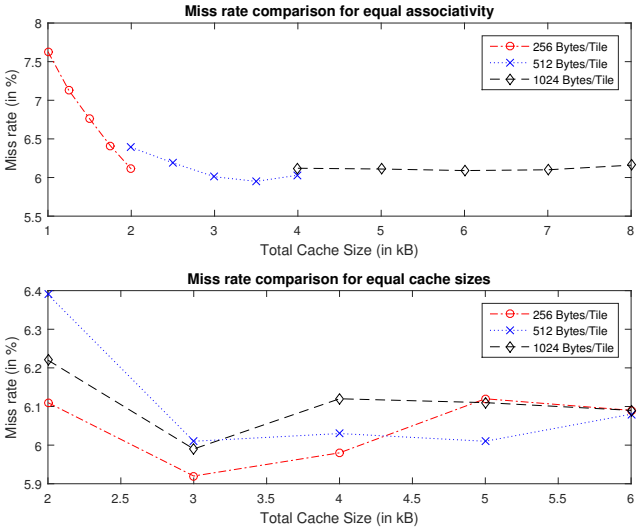


Abbildung 6.23.: Fehltrefferrate im adaptiven Cache bei verschiedenen Cachegrößen bei gleicher Assoziativität (oben) bzw. bei gleicher Cachegröße (unten).

auch größere Cache unter Verwendung der gleichen Anzahl an Ressourcen realisiert werden können. Diese Annahme wird im nächsten Unterabschnitt genauer untersucht.

6.3.3.2. Bewertung der Performanz

Im zweiten Schritt soll die Performanz der adaptiven Cachearchitektur bewertet werden. Es wird zunächst die Performanz des Caches und anschließend die Performanz der Applikation also die Missrate bzw. die Laufzeit bewertet. Als Applikation wird der *Coremark* mit 2000 Iterationen, also einer Laufzeit von etwa 35 Sekunden auf dem *a*-Core verwendet. Es werden hierzu die drei Parametrisierungen auf dem XUPV5 FPGA aus dem vorigen Abschnitt verwendet.

Cacheperformanz: Es wird die Anzahl der verfügbaren Cacheblöcke variiert. Es werden in diesem Abschnitt zwei Versuche durchgeführt: Hier kann die Cachegröße sowie auch die Assoziativität verändert werden. Zunächst soll die Assoziativität konstant bleiben. Es werden zu den vier ursprünglich vorhandenen Cacheblöcken einzelne Cacheblöcke hinzugeschaltet bis acht Cacheblöcke verfügbar sind. Durch die konstante Assoziativität ergibt sich für die kleinste Blockgröße in der größten Konfiguration die gleiche Cachegröße wie für die nächstgrößere Blockgröße in der kleinsten Konfiguration. Für die Vollständigkeit des Versuchs sind hier auch Cachegrößen aufgetragen, die sich aus fünf bzw. sieben Cacheblöcken ergeben, was im CCR aktuell nicht dargestellt werden kann. Daher erfolgt anschließend ein weiterer Versuch, in dem gleiche Cachegrößen verglichen werden.

Die Ergebnisse sind in Abbildung 6.23 (oben) aufgetragen. Es ist zu erwarten, dass die Fehltrefferrate mit steigender Cachegröße abnimmt. Es ist jedoch interessant, dass die Konfiguration mit sieben Cacheblöcken bei $512\,kB$ pro Weg, also $3,5\,kB$ Cachegröße die geringste Fehltrefferrate erreicht.

Für den zweiten Versuch wird die Cachegröße von $2\,kB$ bis $6\,kB$ in $1\,kB$ Schritten variiert. Der Vorteil in diesem Versuch ist, dass der Einfluss der Größe der Wege, in Verbindung mit der Anzahl der notwendigen Cacheblöcke und der daraus resultierenden Cachegröße, jeweils direkt abgelesen werden kann. Es werden insgesamt 24 Cacheblöcke, also 72 BRAMs, vorgehalten, um alle Cachegrößen auch mit der kleinsten Einstellung pro Weg realisieren zu können.

Abbildung 6.23 (unten) zeigt die Fehltrefferraten für die unterschiedlichen Cachegrößen. Ausgehend von den oberen Ergebnissen ist es sehr interessant, dass die kleinere Wegeinstellung bei $3\,kB$ Cachegröße mit deutlich mehr Ressourcenaufwand ca. $0,1\,\%$ besser ist. Es werden jedoch doppelt so viele BRAMs benötigt, wodurch diese Konfiguration aus Effizienzsicht nicht sinnvoll ist. Nach Möglichkeit sollte hier immer mit möglichst großen Wegen gearbeitet werden, um Ressourcen zu sparen.

Applikationsperformanz: Bei diesem Versuch wird die Performanz, also die Ausführungszeit des *Coremark* für die verschiedenen Cacheparameter betrachtet. Abbildung 6.24 (oben) zeigt, dass die Ausführung bis zu einer gewissen Cachegröße von mehr Ressourcen profitiert. Ab einer Cachegröße von $3\,kB$ zeigt ich kein Vorteil durch die Vergrößerung des Caches. Nach

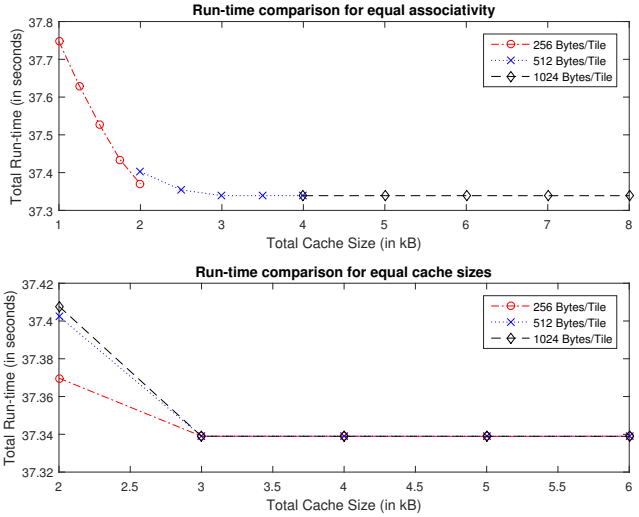


Abbildung 6.24.: Ausführungszeit des Coremark bei verschiedenen Cachegrößen bei gleicher Assoziativität (oben) bzw. bei gleicher Cachegröße (unten).

den Ergebnissen bei der Missrate muss jedoch festgestellt werden, dass diese, wie in Abbildung 6.24 (unten) zu erkennen, ab $3\,kB$ Cache keine Auswirkung auf die Ausführungszeit hat. Hier spielt es auch keine Rolle wie groß die Wege sind.

Hieraus lässt sich jedoch auch ableiten, dass es vorteilhaft sein kann diese Ressourcen, da sie im aktuellen Szenario keinen Vorteil bringen, einem anderen Rechenkern auf der gleichen Kachel zur Verfügung zu stellen. Ein Fallbeispiel soll diese Idee im nächsten Unterabschnitt untersuchen.

6.3.3.3. Fallbeispiel des adaptiven Cache in einem Mehrkernprozessor

Nachdem im letzten Versuch festgestellt wurde, dass ein größerer Cache nicht immer eine Steigerung der Performanz bringt, soll überprüft werden, ob die Idee des adaptiven Caches auch für Mehrkernprozessoren Anwendung finden kann. Basierend auf der Parametrisierung kann ein

Tabelle 6.1.: Testsystem bestehend aus einer Kachel mit zwei Rechenkernen und den dargestellten Parametern für den adaptiven Cache.

Parameter	Wert
Anzahl CPUs	2
Taktfrequenz	80 MHz
I-Cachegröße	8 kB
D-Cache Anzahl Cacheblöcke	8
D-Cache Blockgröße	512 Byte
D-Cachegröße	4 kB

Rechenkern von der Abschaltung von Ressourcen bei einem anderen Prozessor profitieren, so dass hier zeitweise mehr Ressourcen zur Verfügung stehen könnten.

Zur Validierung dieser Idee soll ein Fallbeispiel genutzt werden, bei dem zunächst ein *Coremark* auf zwei Prozessoren ausgeführt wird. In einem zweiten Schritt soll dann ein *Adaptive Differential Pulse Code Modulation* (ADPCM) ausgeführt werden. Dieser profitiert im Vergleich zum *Coremark* auch bei einem kleinen Cache nicht von mehr Ressourcen. Daher sollen dem *Coremark* in einem zweiten Durchlauf mehr Ressourcen zur Verfügung gestellt werden. Konkret wird der Versuch mit acht nativen Cacheblöcken mit jeweils 512 *Byte* durchgeführt. Diese werden zunächst gleich (4 : 4) zwischen den beiden Rechenkernen aufgeteilt. Bei einem zweiten Durchlauf werden die Cacheblöcke dann ungleichmäßig (2:6) zugunsten des *Coremark* aufgeteilt. Die Zeit, die für die Adaption benötigt wird, wird in das Ergebnis mit einberechnet, damit es in Bezug auf das Szenario insgesamt möglichst aussagekräftig ist.

Tabelle 6.2.: Ausführungszeit der beiden *Benchmarks* mit der jeweils notwendigen Zeit für die Adaption.

	Benchmark	Cacheblöcke	Adaptionszeit	Ausführungszeit
CPU 1	MiBench	4	$0\,ns$	$13,31\,ms$
CPU 2	Coremark	4	$0\,ns$	$11,60\,ms$
CPU 1	MiBench	2	$75\,ns$	$13,38\,ms$
CPU 2	Coremark	6	$880\,ns$	$11,27\,ms$

Tabelle 6.2 zeigt einen Überblick über die Ergebnisse der Laufzeiten der Ausführungen mit den beiden Konfigurationen. Bei der statischen Ausführung ist keine Adaption notwendig, so dass diese Zeit nur einmal vor dem zweiten Durchlauf mit veränderter Parametrisierung auftritt. Dementsprechend verändern sich auch die Ergebnisse des zweiten Durchlaufs. Es ist zu erkennen, dass der ADPCM mit weniger Cache fast gar nicht ($< 1\,\%$) langsamer wird als in der ursprünglichen Konfiguration. Andererseits profitiert der Coremark aber deutlich ($2,8\,\%$) von der Vergrößerung des Caches.

Tabelle 6.3.: Relative Ausführungszeit und Performanzsteigerung in der Mehrkernarchitektur.

	Benchmark	relative Ausführungszeit	Performanz- und Effizienzsteigerung
CPU 1	MiBench	$+67\,\mu s$	$-0.5\,\%$
CPU 2	Coremark	$-329\,\mu s$	$+2.8\,\%$

Tabelle 6.3 fasst die Ergebnisse zusammen. Hierzu wird die Adaptionszeit von $880\,ns$ zur Gesamtlaufzeit hinzugerechnet. Die macht in diesem Fall bei einem Betrag im zweistelligen Millisekunden Bereich weniger als $0,01\,\%$ der Laufzeit aus, so dass insgesamt eine Performanz- und aufgrund des gleichbleibenden Ressourceneinsatzes auch eine Effizienzsteigerung von $2,8\,\%$ entsteht.

6.4. Dynamische Cache Reallokation in Mehrkernarchitekturen

Nachdem in den vorangegangenen Abschnitten gezeigt werden konnte, dass bei unterschiedlicher Parametrisierung des adaptiven Cache die Algorithmen unterschiedlich stark vom erhöhten Ressourceneinsatz profitieren, soll in diesem Abschnitt untersucht werden, wie vorhandene Ressourcen dynamisch anderen Prozessorkernen auf einer Kachel im Mehrkernsystem zur Verfügung gestellt werden können. Die Reallokation muss mit sehr geringem bzw. ohne zusätzlichen Ressourcen- und vor allem Zeitaufwand durchgeführt werden können. Trotzdem müssen die Anforderungen an der

L1 Cache eingehalten werden, so dass beispielsweise kein Spülen des Caches getriggert werden sollte. So wird in diesem Abschnitt nach der konzeptionellen Entwicklung der Reallokation vor allem der Ressourcenmehraufwand betrachtet, um die Effizienz bewerten zu können. Die dynamische Cache Reallokation wurde auf der ARC [TCO+16b] vorgestellt.

6.4.1. Design der dynamischen Cache Reallokation

Um die dynamische Cache Reallokation durchführen zu können, muss der adaptive Cache erweitert werden, Diese Erweiterungen werden in den folgenden Unterabschnitten vorgestellt.

6.4.1.1. Anforderungen

Auch bei den Erweiterungen zur dynamischen Cache Reallokation sollen keine realisierungsspezifischen Komponenten verwendet werden. Somit soll es möglich sein, dass Konzept mit relativ geringem Entwicklungsaufwand auf FPGAs und *Application Specific Integrated Circuit*s (ASICs) zu realisieren, ohne das Gesamtsystem an dieser Stelle konzeptionell einzuschränken. Eine weitere wichtige Anforderung ist die Einhaltung der Latenz von einem Zyklus, damit der adaptive Cache weiterhin als L1 Cache eingesetzt werden kann. Im Gegensatz zu bisher bekannter Literatur, wo ähnliche Konzepte im LLC realisiert wurden, soll die dynamische Cache Reallokation tief in der Mikroarchitektur des Rechenkerns als L1 Cache realisiert werden. Hierdurch wird die Performanz des Cachesystems zum Hauptfokus, wobei die effiziente Nutzung von Ressourcen mit möglichst geringem Ressourcenmehraufwand nicht minder wichtig ist.

6.4.1.2. Benötigte Module

Die für die dynamische Cache Reallokation benötigten Module sind in Abbildung 6.25 gezeigt. Zunächst werden die im vorigen Abschnitt angesprochenen Cacheblöcke als generisch verfügbare Cacheblöcke interpretiert, die den einzelnen Rechenkernen zugewiesen werden können. Die Anforderung kann beispielsweise durch die Software zur Laufzeit an die Cache-

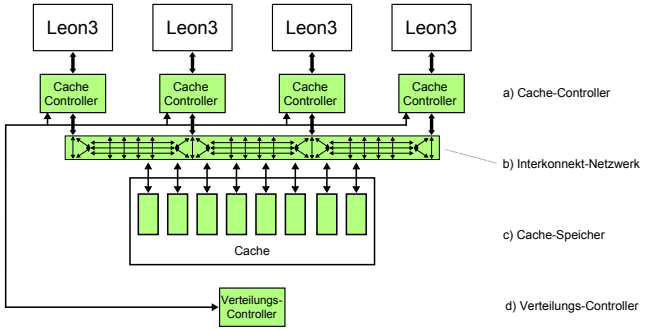


Abbildung 6.25.: Übersicht der Module des adaptiven Caches: Alle für die dynamische Reallokation notwendigen Module sind farbig hinterlegt.

bzw. Verteilungssteuerung erfolgen. Diese steuert das Verbindungsnetzwerk, über das der Cachespeicher der Cachesteuerung des jeweiligen Prozessors zugeordnet werden kann. Es fällt hierbei auf, dass das Verbindungsnetzwerk einen weiteren Freiheitsgrad in die Architektur bringt. Dieser darf die Flexibilität nicht auf Kosten des Zeitverhaltens steigern, da sonst die Anforderungen an den L1 Cache nicht mehr erfüllt sind und die Performanz und dementsprechend auch die Effizienz einbrechen würde. Die individuelle Cachesteuerung soll für jeden Prozessor beibehalten werden, um in der Lage zu sein eine individuelle Parametrisierung bezüglich Zeilenlänge, Assoziativität, Cachegröße oder auch Verdrängungsstrategie für jeden Rechenkern zu realisieren. Aus Effizienzgründen soll nur eine globale Verteilungssteuerung eingesetzt werden.

Cachespeicher: Der Cachespeicher besteht aus einzelnen Cacheblöcken. Als zusätzliche Eigenschaft sollen ungenutzte Cacheblöcke deaktiviert werden können. Dies ist speziell bei *Dark Silicon* Anforderungen an die Architektur interessant, um dynamisch auf das zur Verfügung stehende Energiebudget reagieren zu können. Das Verbindungsnetzwerk muss in der Lage sein im Falle eines Treffers die Daten innerhalb eines Zyklus aus dem Cachespeicher an den jeweiligen Rechenkern zu liefern. Jeder einzelne Cacheblock kann als einfacher *Direct Mapped* Cache gesehen werden. Dementsprechend können n Cacheblöcke als n-fach vergrößerter *Direct Mapped* Cache oder als n-fach assoziativer Cache mit n parallelen Cacheblöcken angesehen werden. Selbstverständlich sind auch alle Mittelwege der

Realisierung mit $n - m$-facher Cachegröße und $m - n$-facher Assoziativität realisierbar. Dementsprechend sollten für Assoziativität und Cachegröße optimale Parameter für die jeweilige Applikation gewählt werden.

Verbindungsnetzwerk: Das Verbindungsnetzwerk ist das Kernstück der dynamischen Cache Reallokation. Es ermöglicht die dynamische Zuweisung der Cacheblöcke an die Rechenkerne und den Transport der Daten für bis zu n parallele Anfragen der n vorhandenen Rechenkerne. Dies würde bedeuten, dass es möglich sein müsste jeden beliebigen Cacheblock mit jedem vorhandenen Prozessor zu verbinden. Wie bereits angesprochen ist die Performanz des Verbindungsnetzwerks jedoch maßgebend für die Gesamtperformanz des der Cachearchitektur. Dementsprechend darf die Latenz nicht erhöht und der Durchsatz dadurch minimiert werden. Trotzdem sollte der Ressourcenmehraufwand so gering wie möglich sein, um eine effiziente Realisierung zu gewährleisten. Dadurch sind dedizierte Verbindungen zwischen Cachesteuerung und Cachespeicher nicht möglich. Konkret müssen folgende Anforderungen erfüllt sein:

- Ein Lesezugriff vom Prozessor bzw. von der Cachesteuerung zum Cachespeicher muss in einem Zyklus erfolgen.
- Die maximale Frequenz darf durch die Realisierung nicht gesenkt werden.

Es werden hierzu zwei Metriken verglichen:

- Anzahl der zur Realisierung benötigten Multiplexer
- Anzahl der zur Realisierung benötigten Verbindungen

Zur besseren Nachvollziehbarkeit werden lediglich $2:1$ Multiplexer eingesetzt, die bei der Realisierung durch die Werkzeuge weiter zusammengefasst und optimiert werden können. Der Verbindungsaufwand ist im Vorfeld sehr schwer einzuschätzen, da vor allem die Leitungslängen einen deutlichen Einfluss auf die maximale Frequenz haben. Diese sind jedoch zur Designzeit nicht bekannt, da sie ausschließlich von der zur Realisierung gewählten Zielarchitektur und der Platzierung in dieser abhängen. Daher werden folgende Annahmen getroffen:

- Die Leitungslänge zwischen CPU und Cacheblock wird als 1 angenommen.
- Alle Leitungen sind gleich lang.

- Alle Verzweigungspunkte der Verbindungsleitungen sind gleich weit voneinander entfernt.
- Die Anzahl der Leitungen ist die Summe der Leitungen zwischen diesen Punkten.
- N als Anzahl der CPUs mit $2^n = N, n \in \mathbb{Z}$
- M als Anzahl der Cacheblöcke mit $2^m = M, m \in \mathbb{Z}$

Gemeinsamer Bus: Eine klassische Realisierungsmöglichkeit ist ein gemeinsamer Bus für alle Rechenkerne und Cacheblöcke. Dementsprechend haben alle Prozessoren Zugriff auf die gleichen Daten und der Cache wird, wie in Abbildung 6.26a gezeigt, implizit geteilt. Hier müsste die Prozessor Cache Kommunikation serialisiert werden, falls die Kommunikation geteilt werden würde. Alternativ müsste mindestens der vierfache Ressourcenaufwand betrieben werden, um parallele Kommunikation zu ermöglichen. Dies macht aus Effizienzsicht keinen Sinn. Dieses Konzept könnte in einer höheren Cacheebene angewandt werden, wie beispielsweise im *Last Level Cache* (LLC). Die serielle Kommunikation ist jedoch durch die Erhöhung der Latenz nicht unter Einhaltung der genannten Anforderungen für einen L1 Cache möglich.

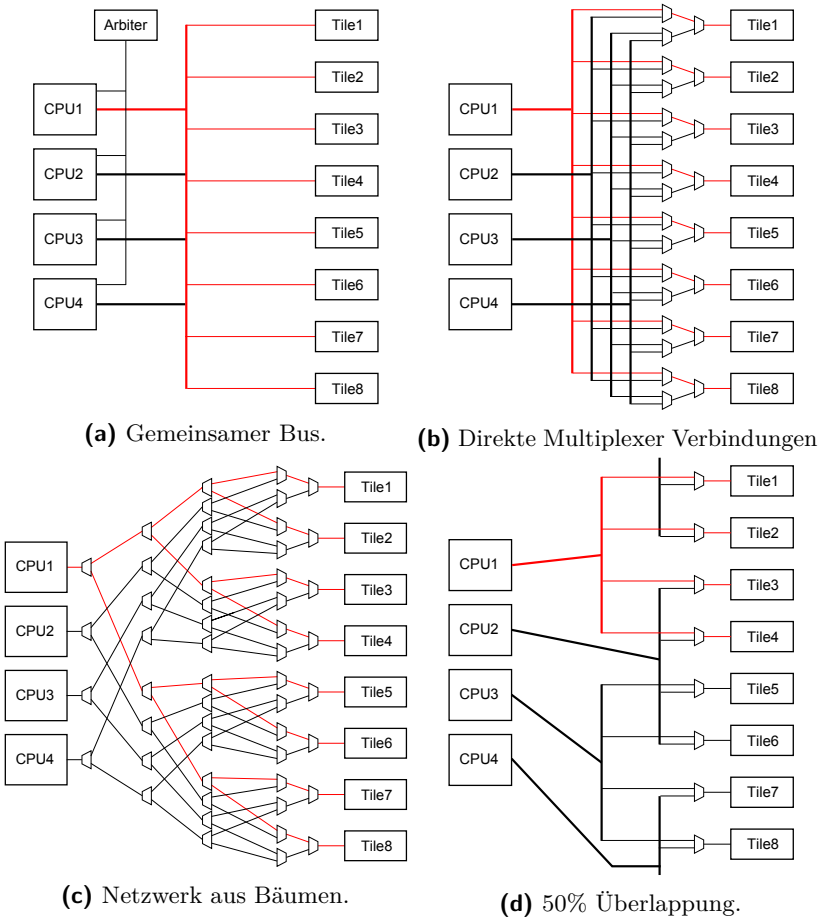
Direkte Multiplexer Verbindungen: Abbildung 6.26b zeigt ein erweitertes Verbindungsnetzwerk, das mit Multiplexern realisiert wird. N Prozessoren sind mit M Cacheblöcken mit $N:1$ Multiplexern verbunden. Da hier $2:1$ Multiplexer genutzt werden um die Kosten der Realisierung abzuschätzen, verlängert sich der kritische Pfad dementsprechend mit erhöhter Latenz.

Die Anzahl der Multiplexer beträgt

$$P_D = \sum_{i=1}^{\log_2(N)} \frac{N \cdot M}{2^i}. \tag{6.5}$$

Die Anzahl der Verbindungen beträgt

$$B_D = \sum_{i=1}^{\log_2(N)+1} 2^i \cdot M + N, \tag{6.6}$$



(a) Gemeinsamer Bus.

(b) Direkte Multiplexer Verbindungen.

(c) Netzwerk aus Bäumen.

(d) 50% Überlappung.

Abbildung 6.26.: Übersicht der verglichenen Methoden zur Realisierung des Verbindungsnetzwerks.

mit der Anzahl Ebenen

$$S_D = \log_2(N) + 2. \tag{6.7}$$

Es ergibt sich folgender Gesamtaufwand:

$$L_D = \frac{B_D}{S_D} = \frac{\sum_{i=1}^{\log_2(N)+1} 2^i \cdot M + N}{\log_2(N) + 2} \tag{6.8}$$

Netzwerk aus Bäumen: Das Netzwerk aus Bäumen verbindet Rechenkerne und Cacheblöcke, wie in Abbildung 6.26c gezeigt, mit einem binären Baum. Auf diese Art kann eine dedizierte Verbindung von jedem Prozessor zu jedem Cacheblock aufgebaut werden und die serielle Buskomminukation vermieden werden.

Die Anzahl der Multiplexer beträgt

$$P_{NaB} = \sum_{i=0}^{\log_2(M)-1} 2^i \cdot N + \sum_{i=1}^{\log_2(N)} \frac{N \cdot M}{2^i}. \tag{6.9}$$

Die Anzahl der Verbindungen beträgt

$$B_{NaB} = \sum_{i=0}^{\log_2(M)-\mathbf{2}} 2^i \cdot N + \sum_{i=1}^{\log_2(N)} \frac{N \cdot M}{2^i}, \tag{6.10}$$

mit der Anzahl Ebenen

$$S_{NaB} = \log_2(M) + \log_2(N) + 1. \tag{6.11}$$

Es ergibt sich folgender Gesamtaufwand:

$$L_{NaB} = \frac{B_{NaB}}{S_{NaB}} = \frac{\sum_{i=0}^{\log_2(M)-\mathbf{2}} 2^i \cdot N + \sum_{i=1}^{\log_2(N)} \frac{N \cdot M}{2^i}}{\log_2(M) + \log_2(N) + 1} \tag{6.12}$$

50% Überlappung: Ein anderer Ansatz ist die Reduktion der Komplexität des Designs, um den vergleichsweise hohen Realisierungsaufwand einzugrenzen. Hierbei muss nicht jeder Prozessor mit jedem Cacheblock verbunden werden. Dieser Ansatz ist in Abbildung 6.26d gezeigt. Es wird

festgelegt, dass mindestens zwei Cacheblöcke pro Prozessor vorhanden sein müssen. Diese könnten individuell parametrisiert oder auch bzgl. *Dark Silicon* abgeschaltet sein. Es wird jedoch nicht mehr möglich sein beliebige Verbindungen über andere Verbindungen hinweg zu realisieren, sondern sich es muss sich an der benachbarten Realisierung orientieren. Dementsprechend soll es möglich sein 50 % der Ressourcen der benachbarten Prozessoren reallokieren zu können. Dementsprechend ist nur eine Multiplexerstufe nötig und die Länge und die Komplexität des Verbindungsnetzwerks kann deutlich reduziert werden. Außerdem kann der Ressourcenaufwand deutlich reduziert werden und die geringe Latenz ermöglicht es das Design im L1 Cache einzusetzen.

Die Anzahl der Multiplexer beträgt

$$P_{50\,\%} = M. \tag{6.13}$$

Die Anzahl der Verbindungen beträgt

$$B_{50\,\%} = M + 2 \cdot M + N, \tag{6.14}$$

mit der Anzahl Ebenen

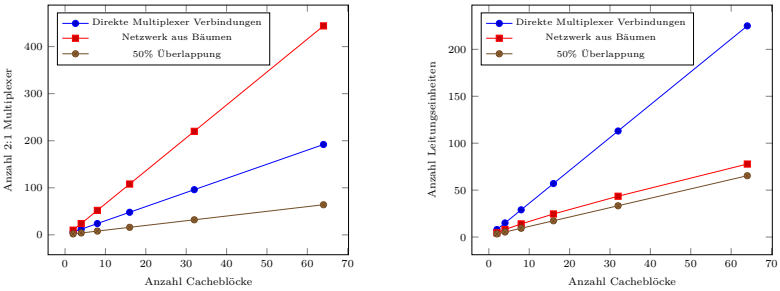
$$S_{50\,\%} = 3. \tag{6.15}$$

Es ergibt sich folgender Gesamtaufwand:

$$L_{50\,\%} = \frac{B_{50\,\%}}{S_{50\,\%}} = \frac{M + 2 \cdot M + N}{3} \tag{6.16}$$

Auswahl der Designmethode: Durch die Abhängigkeit der Performanz der ausgeführten Applikation von den gewählten Parametern kann gefolgert werden, dass die Anzahl der Fehltreffer und dementsprechend auch die Fehltrefferrate (FTR) abhängig von der Cachegröße (C) ist [79, 134, 137]:

- $FTR \propto m \cdot C$
- $FTR \propto \log_n C$
- $FTR \propto \sqrt[n]{C}$



(a) Kosten der Multiplexer. **(b)** Kosten der Verdrahtung.

Abbildung 6.27.: Kostenvergleich der Multiplexer und der Verdrahtungskomplexität für die verschiedenen Methoden zur Realisierung des Verbindungsnetzwerks.

Die Fehltrefferraten sind proportional zur n-ten Wurzel oder zum n-ten Logarithmus der Cachegröße, so dass extreme Zuordnungen der Cacheblöcke dementsprechend nicht als sinnvoll erscheinen. Dies würde beispielsweise zutreffen, wenn ein Prozessor sehr viele oder fast alle Cacheblöcke belegen würde und die anderen Prozessoren lediglich die minimale Konfiguration hätten. Somit ist es möglich 50 % Überlappung zu realisieren, ohne notwendige Flexibilität bei der Ressourcenverteilung einzubüßen.

Abbildung 6.27 zeigt die Kosten im Vergleich der Realisierungsalternativen für die benötigten $2:1$ Multiplexer (Abbildung 6.27a) und die Anzahl der Leitungseinheiten (Abbildung 6.27b). Auf Basis dieser Analyse kann 50 % Überlappung als Variante mit dem geringsten Realisierungsaufwand und relativ geringem Einfluss auf das Gesamtkonzept der dynamischen Cache Reallokation gewählt werden. Da die Ausführungszeit der Applikationen durch die Einschränkungen der 50 % Überlappung Methode laut Literatur nicht beeinflusst wird, kann diese Variante ausgewählt werden. Die Realisierung wird im nächsten Unterabschnitt vorgestellt.

6.4.2. Realisierung

Im Folgenden wird die Realisierung der dynamischen Cache Reallokation vorgestellt. Ein besonderes Augenmerk liegt hier auf der Cache-, Speicher- und der Verteilungssteuerung.

Entsprechend der *Harvard* Architektur haben Instruktions- und Datencache jeweils eine individuelle Cachesteuerung. Die Steuerung des Instruktionscache kann einfacher realisiert werden, da nur Lesezugriffe bearbeitet werden müssen. Daher wird im Folgenden der komplexer realisierte Datencache betrachtet. Beim Instruktionscache können die entsprechenden Funktionalitäten wiederverwendet oder auch weggelassen werden. Es ist somit möglich, dass Instruktions- und Datencache individuell parametrisiert werden können. Rein konzeptionell ist es auch denkbar Cacheblöcke auch zwischen Instruktions- und Datencache dynamisch zu reallokieren. Beim aktuellen Entwicklungsstand ist dies jedoch nur zur Designzeit möglich, um den daraus resultierenden Ressourcenmehraufwand bei der Ansteuerung zu vermeiden.

6.4.2.1. Cachesteuerung

Die Cachesteuerung ist der wichtigste Teil des Cachesystems. Abbildung 6.28 zeigt den schematischen Aufbau. Die Cachesteuerung muss unterschiedliche zentrale Aufgaben erfüllen und ist daher an alle Module angebunden. Zuerst steht die Kommunikation mit dem Rechenkern im Vordergrund. Dies bedeutet, dass die Speicherinstruktionen (Load/Store) bearbeitet werden müssen, wobei es je nach Instruktion zu unterschiedlich breiten Speicherzugriffen (8, 16, 32, oder 64 Bit) kommen kann. Im selben Maße wichtig ist die gleichzeitige Kommunikation mit dem eigentlichen Cachespeicher, der in den Cacheblöcken realisiert ist. Die Kommunikation erfolgt über das bereits vorgestellte Verbindungsnetzwerk. Damit die Cachesteuerung die dynamische Reallokation durchführen kann, wird neben dem Lesen und dem Schreiben der Zustand Adaption definiert. Die Adaption wird mit erhöhter Priorität durchgeführt, so dass diese immer so schnell wie möglich ausgeführt, wird um undefinierte Zustände zu vermeiden, da der nachfolgende Lese- oder Schreibzugriff nicht vorgezogen werden kann. Somit kann die optimale Konfiguration für die angefragte Parametrisie-

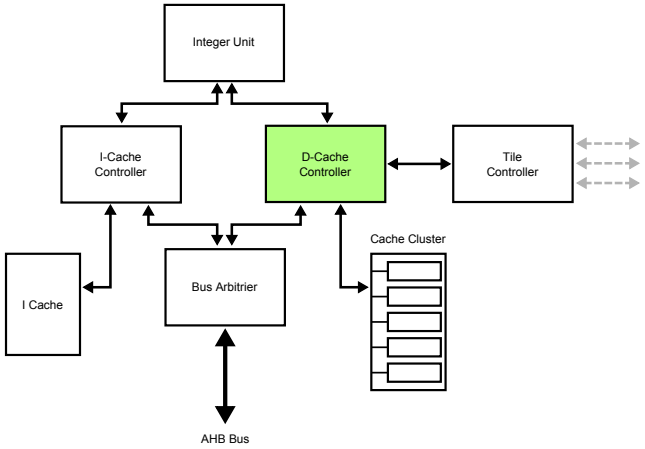


Abbildung 6.28.: Übersicht über die Cachesteuerung und angebundenen Module.

rung realisiert und damit für alle nachfolgenden Instruktionen garantiert werden. Es muss jedoch beachtet werden, dass die Verarbeitung von bereits angefragten Lese- oder Schreiboperationen noch im Hintergrund ablaufen könnte. Die kann insbesondere bei Schreiboperationen auftreten, da eine dreistufige Schreibhierarchie in Richtung der Bussteuerung realisiert ist, bevor das Cachesystem und damit auch der Prozessor angehalten werden muss.

Bei einer Leseoperation wird zunächst überprüft, ob an der angefragten Adresse Daten im Cache liegen, also ein Treffer vorliegt. Ist dies der Fall, werden die Daten an die Cachesteuerung gegeben und an die Prozessorpipeline weitergeleitet. Ist dies bei einem Fehltreffer nicht der Fall, müssen zwei Aktionen gestartet werden: Zunächst muss die Prozessorpipeline so lange angehalten werden bis die Daten geladen sind. Ansonsten könnten falsche Daten in die Pipeline geladen werden, da der Ladevorgang über eine oder mehrere weitere Cachehierarchien oder auch den Hauptspeicher mehrere Zyklen benötigt. Die Cachesteuerung leitet die Anfrage an die Speicherhierarchie weiter und wartet bis die Daten zurückgegeben werden, die direkt an die Pipeline weitergeleitet werden. Parallel werden die geladenen Daten im Cache abgelegt, um für weitere Berechnungen nahe am

Prozessor vorhanden zu sein. Daraufhin wird die restliche Cachezeile ab der angefragten Adresse proaktiv geladen, da die Wahrscheinlichkeit sehr hoch ist, dass mit fortlaufenden Daten gearbeitet wird. Somit muss die Pipeline bei der darauffolgenden Anfrage nicht angehalten werden, wenn die Daten bereits in der Cachesteuerung angekommen sind.

Der aktuelle Status der Cachesteuerung wird im jeweiligen *Cache Configuration Register* (CCR) abgelegt, das durch die Software sowie auch die Hardware gelesen bzw. geschrieben werden kann. Somit wäre auch in diesem Konzept eine Selbstadaption denkbar. Hierbei muss jedoch darauf geachtet werden, dass die Parameter für den jeweiligen Prozessor nicht konträr zu den Parametern der anderen Prozessoren und der globalen Optimierung sind, also beispielsweise genügend Cacheblöcke zur Verfügung stehen. Daher ist es ratsam ausgehend vom globalen Ziel die Parametrisierung der einzelnen Cachesteuerung der Prozessoren abzuleiten und dynamisch beispielsweise durch die Systemsoftware vorzunehmen.

6.4.2.2. Verteilungssteuerung

Die Verteilungssteuerung ist für die Verdrahtung zwischen den Cacheblöcken und den Prozessoren über das Verbindungsnetzwerk zuständig. Somit ist es auch denkbar, dass einzelne Cacheblöcke nicht nur exklusiv einem Prozessor zur Verfügung stehen, sondern auch als gemeinsamer Speicher genutzt werden zu können, was vor allem auf höheren Hierarchieebenen Sinn macht. Außerdem beinhaltet diese Steuerung auch die Parametrisierungen aller Prozessoren der jeweiligen Kachel. Für jedes Auslösen einer Reallokation wird der aktuelle Zustand mit dem angefragten Zustand verglichen und die Machbarkeit überprüft. Somit können unmögliche oder undefinierte Zustände, beispielsweise aufgrund von unbedacht genutzter Selbstadaption, vermieden werden. Bei einer Reallokation werden zunächst so viele Cacheblöcke wie möglich freigegeben. Anschließend werden so viele Cacheblöcke wie nötig zum Cache hinzugefügt und über das Verbindungsnetzwerk verbunden. Somit steht die neue Parametrisierung zur Verfügung.

Die Reallokation benötigt mehrere Kommunikationsschritte zwischen der Cache- und der Verteilungssteuerung. Um die maximale Cacheperformanz während der Reallokation zu gewährleisten wird der Cache nicht angehalten

bis die neue Konfiguration hergestellt werden kann, also alle benötigten Ressourcen vorhanden und alle Hintergrundprozesse abgeschlossen sind. Je nach der gewählten Zielparametrisierung müssen einige Regeln eingehalten werden, um undefinierte Zustände zu vermeiden, wie beispielsweise $n \neq \mathbb{N}^2$ Cacheblöcke bezüglich der Assoziativität. Zuerst werden die Cacheblöcke zugeordnet. Abschließend wird die Anzahl der Cachezeilen festgelegt, was die Burstlänge auf dem Bus bestimmt. Weitere technische Hintergründe sind in der Abschlussarbeit von Christoph Orsinger [Ors14] zu finden.

6.4.3. Bewertung der dynamischen Cache Reallokation

Die Bewertung der dynamischen Cache Reallokation erfolgt vor allem im Hinblick auf die zusätzlich notwendigen Ressourcen, um die Reallokation zur Laufzeit durchführen zu können. Dementsprechend werden zunächst die für die Realisierung notwendigen Ressourcen für die Cacheblöcke, die Cache- und Verteilungssteuerung ermittelt. Anschließend wird das Design mit einem LEON3 Prozessor verglichen, um die Vor- bzw. Nachteile der dynamischen Cache Reallokation zu identifizieren.

Es wird wiederum der Xilinx XUPV5 FPGA genutzt. Es werden zwei Rechenkerne auf einer Kachel der InvasIC Architektur realisiert. Entsprechend der Evaluation der adaptiven Cache Architektur, wie in Unterunterabschnitt 6.3.3.3 beschrieben, kommt auch hier ein 2-Wege Cache mit zwei Cacheblöcken, ein 4-Wege Cache mit vier Cacheblöcken und ein 8-Wege Cache mit acht Cacheblöcken im Datencache zum Einsatz. Durch die Veränderung der Größe der Cacheblöcke kann die Gesamtgröße des Caches verändert werden, während die Anzahl der verwendeten Wege konstant bleibt.

6.4.3.1. Gesamter Ressourcenaufwand

Zunächst wird der gesamte Ressourcenaufwand zur Realisierung der dynamischen Cache Reallokation auf Basis der adaptiven Cachearchitektur ermittelt. Die Referenz ist der LEON3 Prozessor, auf dem das Design basiert. Es werden die notwendigen logischen, die Speicher und *DSP48E* Ressourcen verglichen. Durch Ausnutzung der adaptiven Cachearchitek-

Tabelle 6.4.: Insgesamt notwendiger Ressourcenaufwand zur Realisierung eines Zweikernprozessors mit unterschiedlichen Cachegrößen im Vergleich zum LEON3 Prozessor als Referenz.

(a) 2-Wege Cache mit vier Cacheblöcken.

	Referenz 2 Cores 2 W	2 W 1 kB 4 T 256 B/T	2 W 2 kB 4 T 512 B/T	2 W 4 kB 4 T 1024 B/T
Slices	7932	7413	7612	7576
BRAM	21	27	27	27
DSP48Es	8	8	8	8

(b) 4-Wege Cache mit acht Cacheblöcken.

	Referenz 2 Cores 4 W	4 W 2 kB 8 T 256 B/T	4 W 4 kB 8 T 512 B/T	4 W 8 kB 8 T 1024 B/T
Slices	8084	7676	7772	7806
BRAM	29	39	38	39
DSP48Es	8	8	8	8

(c) 8-Wege Cache mit sechzehn Cacheblöcken.

	Referenz 2 Cores 4 W	8 W 4 kB 16 T 256 B/T	8 W 8 kB 16 T 512 B/T	8 W 16 kB 16 T 1024 B/T
Slices	Konfig.	8455	8466	8274
BRAM	nicht	63	62	63
DSP48Es	möglich	8	8	8

tur kann die Größe der Cacheblöcke zwischen 256 *Byte*, 512 *Byte* und 1024 *Byte* variiert werden, so dass sich die Cachegröße jeweils verdoppelt, wobei die Anzahl der vorhandenen Wege konstant gehalten wird. Beim Referenzdesign kann zur Designzeit die Anzahl der Wege verändert werden, wobei sich gezeigt hat, dass nur die 4-Wege Konfiguration fehlerfrei funktioniert und die 8-Wege Konfiguration nicht realisiert werden kann. Die Cachegröße ergibt sich hier jeweils aus der Satzgröße von 1 kB.

Tabelle 6.4 zeigt die Ergebnisse der Versuche bei drei verschiedenen Cachekonfigurationen. Insgesamt zeigt sich, dass der Logikverbrauch etwas geringer als im Referenzdesign ist. Durch die effiziente Ausnutzung der Speicherressourcen kann ein Anstieg der Anzahl der benötigten BRAMs vermieden werden. Hintergrund ist hier, dass pro Cacheblock drei BRAMs

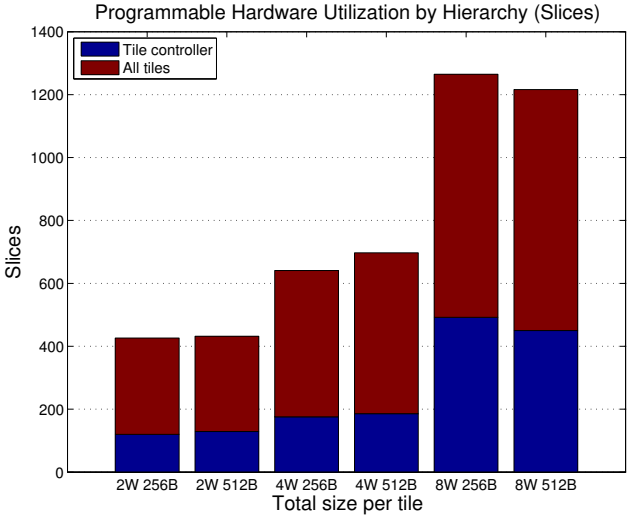
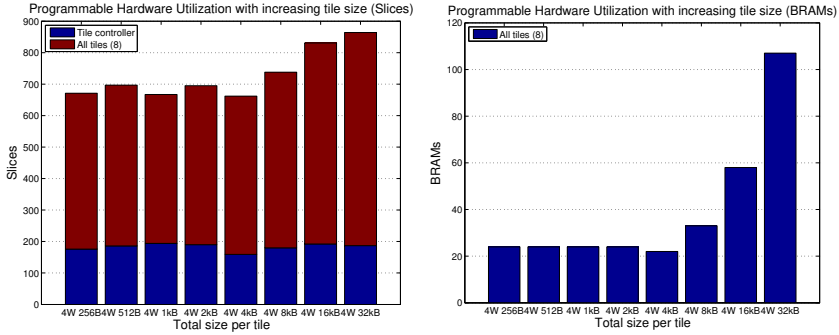


Abbildung 6.29.: XUPV5 slice utilization by hierarchy mit einem (*Direct Mapped* (DM)), zwei und vier Wegen.

initialisiert werden, die Daten, Tag und die Flags speichern. Dadurch ist es möglich die Größe der Cacheblöcke zu skalieren, ohne dass weitere BRAMs benötigt werden. Bemerkenswert ist auch, dass die *Slices* in Tabelle 6.5a für den größten Cache mit 1024 *Byte* pro Cacheblock geringer sind als bei 512 *Byte* pro Cacheblock. Gleiches zeigt sich auch beim 8-Wege Cache (Tabelle 6.5c). Des Weiteren ist interessant, dass beim 4-Wege Cache (Tabelle 6.5b) bzw. 8-Wege Cache (Tabelle 6.5c) bei 512 *Byte* pro Cacheblock ein BRAM eingespart werden kann. Die Anzahl der für die Adressberechnung notwendigen *DSP48E* Ressourcen bleibt konstant.

6.4.3.2. Ressourcenaufwand der einzelnen Hierarchieebenen

Im zweiten Versuch werden die Logikressourcen nach den Hierarchieebenen des Designs analysiert, um die Verteilung der Ressourcen zwischen Cacheblöcken und der Ansteuerung des Cachespeichers unterscheiden zu können. Abbildung 6.29 zeigt das Ergebnis für einen *Direct Mapped*, 2- und 4-Wege



(a) Logikressourcenaufwand (*Slices*). **(b)** Logikressourcenaufwand (BRAMs).

Abbildung 6.30.: *Slice* und BRAM Ressourcenaufwand mit steigender Cacheblockgröße.

Cache. Wie bereits im vorigen Versuch angesprochen bleibt der Logikverbrauch bei der Verdopplung des Cachegröße bei gleicher Wegkonfiguration in etwa konstant ($\approx \pm 4\,\%$) Es ist zu erkennen, dass der Logikverbrauch für die Verteilungssteuerung mit der Anzahl der Wege anwächst. Dies ist nachvollziehbar, da parallele Anfragen an die als Wege vorhandenen Cacheblöcke gestellt werden und anschließend verglichen bzw. die Daten weiter geleitet werden müssen. Interessant ist, dass sich die benötigten Ressourcen jedoch nicht sofort verdoppeln, sondern zunächst um etwa 50 % und im zweiten Schritt um 100 % ansteigen. Dies gilt insbesondere für die Steuerungslogik, deren Komplexität im gleichen Maße wie die innerhalb der Cacheblöcke ansteigt.

6.4.3.3. Ressourcenaufwand bei steigender Cacheblockgröße

Im letzten Versuch werden Logik und BRAM Verbrauch bei steigender Cachegröße untersucht. Es wird hierzu ein 2-Wege Cache verwendet. Dies soll insbesondere dazu dienen den Einfluss größerer Cachegrößen von bis zu $64\,kB$ mit $32\,kB$ pro Weg auf den Ressourcenverbrauch zu untersuchen. Abbildung 6.30a zeigt den Logikverbrauch in *Slices* wie bei den vorigen Versuchen. Es ist zu erkennen, dass der Logikverbrauch bis zu

einer Cachegröße von bis zu $8\,kB$ ($4\,kB$ pro Weg) fast konstant bleibt. Bei Cachegrößen von mehr als $8\,kB$ steigt der Ressourcenverbrauch um bis zu 20 % an, um einen bis zu $64\,kB$ großen Cache zu realisieren. Dies macht auf L1 Ebene wenig Sinn, sollte aber für höhere Cacheebenen durchaus in Betracht gezogen werden, da keine deutlichen Mehrkosten verursacht werden.

Im nächsten Schritt muss überprüft werden, wie viele BRAMs für die Realisierung der Cacheblöcke benötigt werden. Der vorige Versuch hatte angedeutet, dass auch größere Caches mit konstantem Ressourcenaufwand realisiert werden können. Abbildung 6.30b zeigt das Ergebnis. Auch hier ist zu erkennen, dass Caches mit bis zu $8\,kB$ ($4\,kB$ pro Weg) mit konstantem Einsatz von 24 BRAMs realisiert werden können. Für größere Caches werden für jeden weiteren Cacheblock drei zusätzliche BRAMs benötigt. Dies liegt an der hohen Parallelität innerhalb des Cache, um die Latenz von einem Zyklus gewährleisten zu können. Andererseits kann ein $4\,kB$ *Direct Mapped* Cache mit ähnlich vielen Ressourcen wie ein 256 Byte Cache realisiert werden. Dies ermöglicht eine sehr große Flexibilität bei der dynamischen Cache Reallokation ohne die Kosten für den Ressourcenaufwand in die Höhe zu treiben, wodurch die Effizienz des Gesamtsystems bei konstanten Kosten erhöht werden kann.

6.5. Dynamische Cache Strategieanpassung in Vielkernarchitekturen

In aktuell in der Forschung eingesetzten Mehrkernarchitekturen ist die Cachekohärenz nur innerhalb einzelner Kacheln sichergestellt, um die Skalierbarkeit der Architektur bei höherer Auslastung sicherstellen zu können. Daher müssen Daten, die zwischen den Kacheln ausgetauscht werden, über den gemeinsamen globalen Speicher kopiert werden. Dies bedeutet, dass durch den Kopiervorgang Daten im Cache stehen, die in einem für diese Kachel verbotenen Adressbereich stehen.

Zur einfacheren Handhabung dieser Problematik werden in diesem Abschnitt Konzepte vorgestellt mit denen Caches partiell invalidiert und auch die Schreibstrategie geändert werden kann. In diesem Abschnitt wird das

Konzept zur Veränderung der Cachestrategie vorgestellt, umgesetzt und bewertet. Der Fokus liegt hier auf der partiellen Invalidierung des Caches, so dass ein komplettes Spülen des Caches vermieden werden kann. Dies soll die Effizienz des Gesamtsystems erhöhen, da ein Spülen des Caches relativ viel Zeit in Anspruch nimmt, da hier viele in diesem Moment nicht benötigte Daten synchronisiert werden. Abschließend werden die Vorteile und der Realisierungsaufwand des Konzeptes bewertet. Die partielle dynamische Veränderung der Cachestrategien ist zur *Design, Automation and Test in Europe* (DATE) angenommen [MT17].

6.5.1. Daten Transfer Szenario

Abbildung 6.31 zeigt den schematischen Aufbau einer gekachelten Mehrkernarchitektur. Es sind exemplarisch zwei Kacheln mit jeweils vier Rechenkernen gezeigt. Auf jeder Kachel existiert der *Transaction Level Modeling* (TLM) als kachellokaler Speicher, über den Daten per *Message Passing* ausgetauscht werden können. Innerhalb dieser Kachel wird die Cachekohärenz über den kachellokalen Bus sichergestellt. In der in dieser Arbeit verwendeten InvasIC Architektur auf Basis des LEON3 Prozessors wird die Cachekohärenz über die *Write Through* Strategie umgesetzt. Der L2 Cache ist der Abschluss der Kachel, der die Anfragen an den Netzwerkadapter und das *Network on Chip* (NoC) weiterleitet und gleichzeitig die Last vom *Network on Chip* (NoC) nimmt. Somit besteht keine Cachekohärenz zwischen den Kacheln. Zur Partitionierung des globalen Speichers wird der *Partitioned Global Address Space* (PGAS) genutzt. Die Daten müssen dementsprechend von der Software, genauer gesagt vom Laufzeitsystem, speziell dem innerhalb des InvasIC Projektes entwickelten X10 [138] Compilers, verwaltet werden [21, 111].

Es gibt verschiedene Ansätze, um die Datenstruktur vom Adressbereich der ersten Kachel in den Adressbereich der zweiten Kachel zu kopieren. Der klassische Ansatz ist *Message Passing* zu benutzen. Hierbei müssen die Daten zunächst innerhalb der eigenen Speicherpartition serialisiert werden. Anschließend können die serialisierten Daten in einer oder mehreren Nachrichten an die fremde Speicherpartition gesendet werden. Dort müssen sie in die ursprüngliche Form gebracht werden, was innerhalb der fremden Partition abläuft. Das Problem aus Hardwaresicht ist hierbei, dass dieser

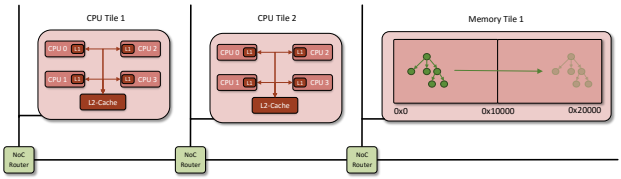


Abbildung 6.31.: Übersicht der InvasIC Architektur mit zwei Kacheln und jeweils vier Prozessoren. Der globale Speicher ist in zwei partitionierte Adressbereiche unterteilt.

Kopiervorgang von einem Prozessor der eigenen Partition ausgeführt wird. Dies führt dazu, dass innerhalb dieser Kachel Daten, die aus dem eigenen in den fremden Adressbereich kopiert werden zum einen noch mit den veralteten eigenen Adressen im Cache liegen und zum anderen jetzt fremde Adressbereiche im Cache liegen, die innerhalb dieser Kachel verboten sind. Diese müssen zum Abschluss des Kopiervorgangs aus dem Cache gespült werden, damit die Daten konsistent innerhalb der jeweiligen Partition vorhanden sind und keine veralteten Daten mehr in einer der Cachehierarchien abgelegt sind. Es laufen aktuell Arbeiten, um den Kopiervorgang vor allem hinsichtlich des Speicherverbrauchs zu optimieren. Diese Konzepte, wie beispielsweise *Message Passing via Shared Memory* oder auch *Cloning* beschleunigen den Kopiervorgang wesentlich, ohne dabei jedoch die Problematik der veralteten Daten in fremden Partitionen umgehen zu können.

Daher wird im Folgenden ein Konzept vorgestellt, mit dem Daten gezielt aus dem Cache gespült werden können, in dem nur diese Daten invalidiert werden können. Dies bedeutet konkret, dass es dann nicht mehr nötig ist den kompletten Cache zu spülen. Statt dessen können konkret die Daten, die beispielsweise an fremden Adressen gespeichert werden, invalidiert werden. Zunächst muss mittels eines Synchronisationspunktes festgestellt werden, dass alle benötigten Daten aus dem Cache in den Speicher kopiert wurden, damit durch das Invalidieren keine Daten im Cache verloren gehen. Dieser Vorgang ist in jedem Fall bei jedem der genannten Vorgänge wichtig. Das partielle Invalidieren von Adressbereichen kann dann gezielt eingesetzt werden, um diesen Vorgang zügig abzuschließen.

6.5.2. Konzept zur partiellen Invalidierung

Die partielle Invalidierung des Caches soll auf der Basis von Adressbereichen erfolgen, die in der Software spezifiziert werden. Zur einfachen Steuerung der Funktionalität soll es möglich sein die Startadresse und die Länge des zu invalidierenden Bereichs festzulegen. Diese Parameter lösen eine Invalidierung innerhalb der adaptiven Cachearchitektur aus. Hierzu wird ausgenutzt, dass es innerhalb der Cacheblöcke einen einzelnen Speicher gibt, in dem die *Valid* Bits zusammen mit anderen Informationen gespeichert sind. Laut der Vorgaben der adaptiven Cachearchitektur sollen zumindest ganze Cachezeilen invalidiert werden, um mit einem *Valid* Bit pro Cachezeile arbeiten zu können. So muss innerhalb des Caches berechnet werden in welcher Zeile die Adressen stehen bzw. muss bei assoziativen Caches bedacht werden, dass die Daten in parallelen Cacheblöcken gespeichert sein könnten. Das reine Invalidieren bezieht sich hier nur auf die Validbits, da in dem vorgestellten Szenario davon ausgegangen werden kann, dass alle benötigten Daten zuvor synchronisiert wurden. Zur Invalidierung einer Cachezeile müssen konkret die folgenden Schritte durchgeführt werden:

- Laden der Flags der adressierten Cachezeile aus dem entsprechenden Cacheblock.
- Bei Bedarf Prüfung, ob die Voraussetzung für eine Invalidierung erfüllt sind, beispielsweise die vorangegangene Synchronisation der Daten abgeschlossen ist.
- Zurücksetzen des der Cachezeile entsprechenden Validbits.

Im Vergleich zu einem Spülen des Caches, bei dem zunächst alle Daten im Cache synchronisiert werden müssen, benötigt das Invalidieren lediglich einen Zyklus pro Cachezeile. Unter der Annahme, dass ein Teil m der n insgesamt verfügbaren Cachezeilen invalidiert werden soll, stehen die Daten in $n - m$ Cachezeilen im Gegensatz zum Spülen auch weiterhin zur Verfügung. In diesem Falle müssten die herausgespülten Daten wieder geladen werden. Damit ergibt sich die Laufzeit des Spülens zu:

$$t = \left(\frac{n}{\#Cacheblöcke} \right) T_{clk} + (n - m)\, T_{load} \tag{6.17}$$

Auch bei der partiellen Invalidierung müsste im schlechtesten Fall der komplette Cache durchlaufen werden. Somit kann die Gesamtlaufzeit wie folgt angenommen werden:

$$t = \left(\frac{2n}{\#Cacheblöcke} \right) T_{clk} \tag{6.18}$$

In der durchschnittlichen Anwendung wird angenommen, dass $m << n$ ist. Des Weiteren kann davon ausgegangen werden, dass die Daten nicht mehr benötigt werden. Daher kann die Invalidierung für den Prozessor bereits nach einen Taktzyklus als abgeschlossen betrachtet werden, während die Cachearchitektur intern noch die Validbits zurücksetzt. Es muss hierzu lediglich der Adressbereich, der durch die Software festgelegt wurde, hinterlegt werden, damit weitere Anfragen in diesem Adressbereich bereits als invalidiert behandelt werden. Diese Modul arbeitet für den Cache selbst transparent im Hintergrund, so dass die Updates erfolgen sobald keine weiteren Datentransfers im Cache zu bearbeiten sind.

6.5.2.1. Partielle Veränderung der Cachestrategie

Die InvasIC Architektur arbeitet im L1 Cache mit der *Write Through* Strategie. Dies erzeugt eine relativ hohe Last auf dem Bus der Kachel und sorgt dementsprechend auch für relativ große Verdrängung im L2 Cache. Als Erweiterung zur partiellen Invalidierung ist es auch denkbar die Cachestrategie partiell zu verändern. Es ist in diesem Fall wichtig die Cachestrategie nicht für den kompletten Cache zu verändern, da die Systemsoftware auf die Cachekohärenz angewiesen ist, die über die *Write Through* Strategie realisiert wird. Dementsprechend wäre es jedoch interessant den Adressbereich, in dem private Applikationsdaten stehen, partiell auf die *Write Back* Strategie anzupassen und die Daten zum Ende der Verarbeitung einmalig mit den höheren Speicherhierarchien zu synchronisieren. Auch hier soll das adressbereichsbasierte Konzept verwendet werden, so dass die Schnittstellen und Module der partiellen Invaliderung wiederverwendet bzw. erweitert werden können.

6.5.3. Realisierung der partiellen Invalidierung

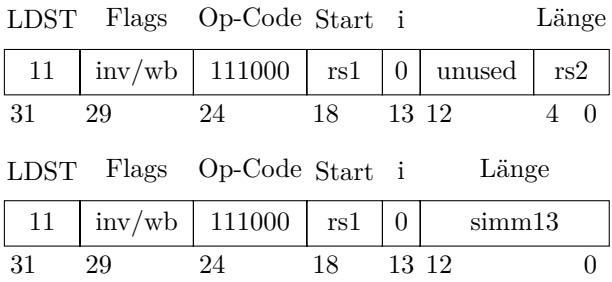


Abbildung 6.32.: Instruktionssatzerweiterungen auf Basis der *Scalable Processor ARChitecture* (SPARC) V8 Instruktionssatz Architektur (ISA) mit der die partielle Invalidierung durchgeführt wird. Die Startadresse wird im Register *rs1*, die Länge im Register *rs2* oder mit der Konstanten *simm13* übergeben.

Für die partielle Invalidierung wird der Instruktionssatz des *a*-Core erneut erweitert. Die zusätzliche Instruktion ermöglicht es die Startadresse und die Länge des zu invalidierenden Adressbereiches zur Laufzeit an die adaptive Cachearchitektur zu übergeben. Es wird das gleiche Instruktionsformat wie bei den anderen Instruktionssatzerweiterungen verwendet. Da bei dieser Instruktion keine Berechnung durchgeführt wird, kann das Ergebnisregister umgewidmet werden, um mit einem Flag anzuzeigen, ob es sich um eine Invalidierung oder eine Veränderung der Strategie handelt. Das Register *rs1* gibt die Startadresse des zu invaliderenden Bereiches an. Da immer ganze Cachezeilen invalidiert werden sollen, wird die Adresse mit der Anzahl der *Offset* Bits ($1, 2, 4, 8$ Wörter pro Zeile) maskiert, um die Startadresse am Anfang der Zeile zu berechnen. Die Länge des Bereichs kann entweder über das Register *rs2* oder die Konstante *simm13* direkt an die Instruktion übergeben werden. Der Op-Code wird zu 111000 definiert, um kompatibel zur SPARC Architektur und den restlichen Erweiterungen des *a*-Core und InvasIC Projektes zu sein.

Passend zu dieser Instruktion wird auch der *Assembler* (siehe LISA-Code 6.1) und *Dissassembler* erweitert, um mit dem neuen Instruktionsformat umgehen zu können.

```
1 ...
2 const struct sparc_opcode sparc_opcodes[] = {
3 ...
4 { "pwbinvd", F3(3, 0x38, 0), F3(~3, ~0x38, ~0), "1,2,&", 0, v8},
5 { "pwbinvd", F3(3, 0x38, 1), F3(~3, ~0x38, ~1), "1,i,&", 0, v8},
6 ...
7 }
8 ...
```

LISA-Code 6.1: Definition der pwbinvd-Instruktion im Quellcode des Assemblers

Es wird ein zusätzliches Register im Cache hinzugefügt, das den Status der Invalidierung anzeigt. Falls zwei Invalidierungen in aufeinanderfolgenden Instruktionen aufgerufen werden, muss der Prozessor angehalten werden, bis die erste Invalidierung abgeschlossen ist. *INV* zeigt die laufende Invaliderung an und *CNT* zeigt die Anzahl der verbleibenden Cacheoperationen bis die Invalidierung abgeschlossen ist. Das Register ist in Abbildung 6.33 gezeigt:

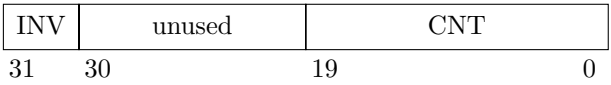


Abbildung 6.33.: Register, das den Status der aktuell laufenden Invalidierung anzeigt.

Im adaptiven Cache wird der Cachespeichersteuerung jeweils die Funktionalität zur Steuerung der Invalidierung hinzugefügt. In einem späteren Schritt könnte diese Funktionalität zur weiteren Effizienzsteigerung in die Cachesteuerung verlagert werden.

In der aktuellen Realisierung wird der Realisierungsaufwand vergleichsweise hoch, da viele Ressourcen mehrere Ports benötigen und diese zusätzlich vorgehalten werden müssen und dadurch die Flexibilität auf Kosten der Realisierung beim Speicherzugriff erhöht.

Ein großer Vorteil der adressbasierten Invalidierung ist der geringe Ressourcenaufwand. Dadurch wird die Zeit für die Invalidierung jedoch abhängig von der Länge des zu invaliderenden Adressbereichs. Die Zeit für eine

Invalidierung eines Adressbereichs mit der Länge von n Bytes ergibt sich zu:

$$t = \left(\frac{n}{\#BytesProZeile} + \#Cachetreffer \right) \cdot T_{clk} \tag{6.19}$$

Hierbei werden die Cachezeilen sequentiell durchlaufen, was zu einem vorhersagbaren Zeitaufwand für die Invalidierung führt.

Falls innerhalb der Cachezeilen noch Vergleiche durchgeführt werden, um zu überprüfen, ob die Invalidierung durchgeführt werden kann, werden zwei weitere Zugriffe auf die Cachezeile benötigt. Damit steigt der Zeitaufwand für die Invalidierung auf:

$$t = \left(2 \cdot \frac{\#Cachezeilen}{\#Cacheblöcke} \right) \cdot T_{clk} \tag{6.20}$$

Diese Laufzeit ist jetzt direkt von der Anzahl der zu invalidierenden Cachezeilen abhängig, was vor allem bei großen Adressbereichen ein großer Vorteil bei den Vorhersagbarkeit der Invalidierung ist.

6.5.3.1. Realisierung der partiellen Veränderung der Cachestrategie

Die partielle Veränderung der Cachestrategie wird in vier Schritten realisiert. Zunächst wird die gleiche Softwareschnittstelle konfiguriert, die auch für die Invalidierung genutzt wird. Des Weiteren wird die Cachesteuerung angepasst, um die unterschiedlichen Schreibstrategien umsetzten zu können. Anschließend muss die Ersetzung von Cachezeilen realisiert werden. Abschließend muss die Synchronisation mit dem Speicher bzw. mit der höheren Hierarchieebene sichergestellt werden.

Dementsprechend muss auch die Cachesteuerung erweitert werden, die bei jeder Speicheranfrage prüfen muss, ob die Daten im Cache liegen und diese aus dem Cache geladen bzw. in den Cache gespeichert werden müssen.

Bei der Synchronisation mit dem Speicher kann gewählt werden, ob die Daten gleichzeitig mit der Synchronisation auch invalidiert werden sollen, wie es beispielsweise in dem eingangs vorgestellten Szenario genutzt wird.

Dementsprechend werden folgende *Flags* für die in Abbildung 6.32 vorgestellte Instruktion wie folgt definiert:

Tabelle 6.6.: Definition der *Flags* für die Instruktion.

Flag	Funktion
0	*No Operation* (NOP)
1	Partielle Invalidierung
2	Partielle Cachestrategie
3	Partielle Invalidierung und Cachestrategie
4	Status (*Debug only*)

Weitere technische Informationen sind in der Abschlussarbeit von Michael Mechler [Mec16] zu finden.

6.5.4. Bewertung der partiellen Invalidierung

In diesem Abschnitt wird die Effizienz der zusätzlichen Fähigkeiten des adaptiven Caches bewertet. Zunächst wird der Ressourcenaufwand für die Realisierung der partiellen Invalidierung und der Veränderung der Cachestrategie untersucht. Anschließend wird auf Basis des eingangs vorgestellten Szenarios eine Bewertung der Effizienzsteigerung vorgenommen.

6.5.4.1. Bewertung des Ressourcenaufwands

Für die Realisierung werden, wie im vorigen Abschnitt beschrieben, sehr viele Logikressourcen benötigt, da die Realisierung zu Gunsten der Flexibilität individuell nahe den Cacheblöcken in der Speichersteuerung realisiert wird. Tabelle 6.7 zeigt die Ergebnisse der Realisierung auf dem Xilinx XUPV5 FPGA. Für die Realisierung werden knapp 3000 zusätzliche Logikelemente benötigt. Durch die Zwischenspeicherung des Status in der Cachesteuerung werden knapp 1250 zusätzliche Register benötigt. Die zusätzlichen BRAMs werden zur Speicherung der zusätzlichen *Flags* genutzt.

Da diese Realisierung in der InvasIC Architektur eingesetzt wird, wird die Zielfrequenz auf lediglich 25 MHz heruntergesetzt, weil die Architektur auf dem Protoypensystem mit sechs High-end *Virtex-5* FPGAs mit insgesamt

Tabelle 6.7.: Ressourcenaufwand für die Realisierung.

	Ressourcenaufwand	
	zusätzlich	relativ
Slices	1489	+15,2 %
Register	623	+14,6 %
LUT	1491	+15,0 %
BRAM	1	+4,9 %

mehr als 50.000 *Slices* (CHIPit) bei dieser Frequenz betrieben wird. Somit wird sichergestellt, dass alle vorhandenen Komponenten, wie NoC und *Double Data Rate* (DDR) Speicher fehlerfrei funktionieren. Das Cachesystem erreicht nach *Place & Route* eine maximale Frequenz von etwa 30 Mhz. Es kann davon ausgegangen werden, dass diese durch Verschiebung der zusätzlichen Funktionalität in der zentralen Cachesteuerung und die Anpassung des Verbindungsnetzwerks wieder auf über 50 MHz gesteigert werden kann. Aktuell liegt der kritische Pfad dementsprechend aus der *Memory* Pipelinestufe über die Cachesteuerung und Verbindungsnetzwerk bis in die Cacheblöcke.

6.5.4.2. Bewertung der Effizienz

In diesem Abschnitt soll die Effizienzsteigerung durch die Nutzung der partiellen Invalidierung im Vergleich zum bisher verwendeten Spülen des Caches verwendet werden. Als Grundlage für die Berechnung dient das eingangs vorgestellte Szenario. Es muss beachtet werden, dass ein Spülen des Caches den kompletten Cache invalidiert, also auch Daten, die später noch benötigt werden, verdrängt werden. Diese müssen zum späteren Zeitpunkt wieder geladen werden, was einen zusätzlichen Vorteil der partiellen Invalidierung darstellt, der in diesem Szenario nicht gemessen wird.

Folgende Formeln werden zur Berechnung der Zeiten zum Spülen bzw. partiellen Invalidieren des Caches verwendet:

Beim Spülen des Caches ergeben sich die acht Zyklen aus dem Durchlaufen der Pipeline und dem Schreiben des *Cache Configuration Register*, um das Spülen zu starten. Dementsprechend werden beim partiellen Invalidieren nur sechs Instruktionen zum Starten der Cachesteuerung benötigt. Falls

Tabelle 6.8.: Gegenüberstellung der Laufzeit in Taktzyklen für das Spülen bzw. partielle Invalidieren des Caches bei einer bzw. mehreren aufeinanderfolgenden Anfragen.

Spülen	partielles Invalidieren
$t = 8 + \frac{\#Cachezeilen}{\#Cacheblöcke}$	$t = 6$
$t = 8 + \frac{\#Cachezeilen}{\#Cacheblöcke}$	$t = 6 + \frac{\#CachezeilenproSatz}{\#Cacheblöcke} + 1$

der zu invalidierende Adressbereich innerhalb einer Instruktion übergeben werden kann und die partielle Invalidierung damit abgeschlossen ist, kann diese im Hintergrund durchgeführt werden, so dass der Prozessor nicht angehalten werden muss.

Es werden zwei Kacheln mit jeweils vier Rechenkernen realisiert. Jeder Kern hat $16\,kB$ 2-Wege Instruktions- und $8\,kB$ 2-Wege Datencache realisiert. Hinzu kommt ein L2 4-Wege Cache mit $64\,kB$ pro Kachel und lokaler TLM mit $8\,MB$ *Static Random-access Memory* (SRAM) Speicher. Das InvasIC NoC verbindet die beiden Kacheln mit dem globalen DDR Speicher.

Für den adaptiven Cache werden in diesem Szenario folgende Ressourcen benötigt:

Tabelle 6.9.: Ressourcenaufwand für die Realisierung der adaptiven Caches.

	adaptiver Cache		
	$16\,kB$ L1I$	$8\,kB$ L1D$	$64\,kB$ L2$
BRAMs	33	25	117
BRAMs pro Kachel	132	100	117
BRAMs gesamt	264	200	234
Slices pro Kachel	23844		833
Slices gesamt	47688		1666

Hieraus folgt, dass ein Spülen des gesamten $64\,kB$ L2 Cache ausgeführt werden muss. Für die partielle Invalidierung werden Datengrößen von $2^3 = 16\,\text{Byte}$ bis $2^{18} = 256\,kB$ betrachtet.

Abbildung 6.34 zeigt das Ergebnis des Versuchs. Es ist zu erkennen, dass das Spülen des $64\,kB$ Caches 1032 Zyklen benötigt. Danach wird davon

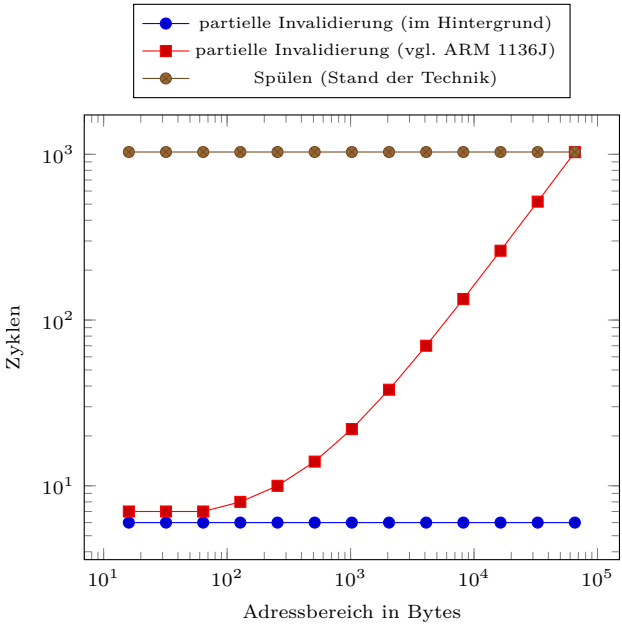


Abbildung 6.34.: Vergleich der benötigten Zyklen für das Spülen bzw. partielle Invalidieren des $64\,kB$ L2 Caches.

ausgegangen, dass der Cache mehrfach gespült werden muss, um keinen Datenverlust zu erleiden. Daher muss der Prozessor für die gezeigte Anzahl der Zyklen angehalten werden. Es ist zu beachten, dass die Daten, die nach dem Speichern in die fremde Partition in der eigenen Partition nicht mehr benötigt werden bzw. nicht mehr erlaubt sind, da sie im fremden Speicherbereich liegen. Daher ist nach zunächst erfolgter Synchronisation der Daten ein Invalidieren der Daten ausreichend. Hierbei soll die partielle Invalidierung genutzt werden. Es wird davon ausgegangen, dass bei Adressbereichen, die größer als der Cache sind die gleiche Zyklenzahl wie bei einem Flush benötigt wird. Sollten die Daten aus dem zuerst synchronierten Adressbereich nicht mehr benötigt werden, steigt die Zyklenzahl proportional zur Anzahl der Cachezeilen. Es ist offensichtlich, dass das Konzept insbesondere für kleine Adressbereiche interessant ist. Es können

hierbei etwa 99 % der benötigten Zyklen eingespart werden. Aber auch bei Adressbereichen, die größer als der Cache sind, ergibt sich eine Zyklenersparnis von 50, 22 % bei 128 kB bzw. 25, 15 % bei 256 kB Cachegröße. Es ist also insgesamt eine deutliche Effizienzsteigerung zu erwarten, da lediglich etwa 15 % mehr Ressourcen eingesetzt werden müssen, um diese Performanzsteigerung zu erzielen. Bei deutlich größeren Speicherbereichen halbiert sich die zu erwartende Effizienzsteigerung bis bei 512 kB Cachegröße der *Break Even* Punkt erreicht ist. Es kann jedoch aus Performanzgründen sinnvoll sein die partielle Invalidierung einzusetzen, da diese mit der Anzahl der zu invalidierenden Cachezeilen als Faktor vorhersagbar ist und daher für das Einhalten von *Deadlines* eingesetzt werden kann.

6.6. Zusammenfassung

Die adaptive Cache Architektur ermöglicht es die Parameter während der Laufzeit flexibel einzustellen. Veränderungen, die bisher ausschließlich zur Designzeit vorgenommen werden konnten, können in der adaptiven Cache Architektur zur Laufzeit vorgenommen werden. Dies gibt Anwendungsentwicklern, aber auch dem Laufzeitsystem, neue Möglichkeiten den Cache an die Anforderungen der Applikation anzupassen. Diese Anforderungen können von den nicht-funktionalen Eigenschaften des Systems dynamisch an die Cache Architektur gestellt werden. Typische Beispiele sind Energiebudgets im Kontext von *Dark Silicon* oder vorhersagbare *Deadlines* in Steuerungsanwendungen.

Die adaptive Cache Architektur wird unabhängig von der Zielarchitektur generisch konzipiert und erstmals im L1 Cache als Hardwarebeschreibung umgesetzt. Einerseits müssen die Fähigkeiten den Anforderungen von maximal einem Zyklus Latenz im L1 Cache genügen. Andererseits muss die Realisierung ressourcenschonend erfolgen, damit nicht nur Performanz-, sondern auch Effizienzvorteile erzielt werden können.

Zunächst werden die Parameter der Anwendungen optimiert, die für die Evaluation des adaptiven Caches genutzt werden. Somit können die Performanzvorteile einzelner Parameter der adaptiven Cache Architektur genau bestimmt werden. Hieraus werden die Parameter abgeleitet die dynamisch veränderbar sein sollen: Es wird definiert, dass die Veränderung der Asso-

ziativität, die in dieser Cache Architektur als Parallelschaltung von Wegen bei assoziativen Caches interpretiert wird, dementsprechend auch eine Veränderung der Cachegröße bewirken kann. Neben diesen Parametern ist die Zeilenlänge in ein Vielfaches der Basiszeilenlänge einstellbar. Darüber hinaus soll auch die Strategie zur Laufzeit geändert werden können. Es kann hier gezeigt werden, dass sich die Cachearchitektur sehr effizient ohne nennenswerten Einsatz zusätzlicher Ressourcen realisieren lässt. Die neu gewonnene Flexibilität kann genutzt werden, um den Cache zur Laufzeit an die Applikation anzupassen. Hier zeigt sich, dass nicht immer die größte Assoziativität oder Cachegröße das beste Ergebnis liefert. Dementsprechend können Ressourcen freigegeben werden, die nicht zur Performanzsteigerung beitragen. So kann der Energiebedarf verringert werden und die Effizienz weiter gesteigert werden. Die Adaption selbst macht weniger als $0,1\,\%$ der Ausführungszeit aus. Da diese somit fast keinen Einfluss auf die Performanz hat und weitere Ressourcen nicht genutzt werden, kann überlegt werden die verfügbaren Ressourcen des Cache den anderen Prozessoren auf der Kachel der Mehrkernarchitektur zuzuordnen.

Es wird ein Konzept entwickelt, bei dem es möglich wird die Cacheressourcen zur Laufzeit den benachbarten Rechenkernen auf einer Kachel der InvasIC Architektur zur Verfügung zu stellen. Weiterhin wird gezeigt, dass sich dieses Konzept der dynamischen Ressourcen Allokation sehr effizient als Erweiterung der adaptiven Cache Architektur realisieren lässt. Es werden im Vergleich zum Einkernprozessor zwischen $5\,\%$ und $10\,\%$ mehr Logikressourcen benötigt, die jedoch nur einmal pro Kachel anfallen, also durch das Verbindungsnetzwerk zwischen den Rechenkernen geteilt werden. Die Realisierung des Speichers basiert auf den nativen Speicherblöcken der Zielarchitektur. Die hoch parallele Realisierung erlaubt die Skalierung der Cachegröße bis hin zu einem 2-Wege $8\,kB$ Cache ohne zusätzliche Ressourcen zu benötigen. Erst bei einer Vergrößerung des Caches hin zu beispielsweise $64\,kB$ werden etwa viermal so viele Speicherressourcen benötigt, während die Ansteurungslogik um lediglich $20\,\%$ ansteigt.

Abschließend wird untersucht wie die Strategie des Caches dynamisch verändert werden kann. Es zeigt sich, dass es vorteilhaft sein kann die Strategie des Caches in einzelnen Bereichen, beispielsweise für den User- und nicht den Systembereich, partiell dynamisch anzupassen. Wenn auf der Basis dieser Daten gearbeitet wird, müssen diese synchronisiert und verworfen werden. Hierzu wird das Konzept der partiellen Invalidierung ein-

zelner Speicherbereiche entworfen. Es dient insbesondere dazu das Spülen des gesamten Caches zu vermeiden, um nicht Systemdaten, die weiterhin benötigt werden, aus dem Cache zu verdrängen. Hier würde nicht nur Zeit für das Spülen, sondern auch zusätzliche Zeit für das erneute Laden benötigt werden. Da bereits synchronisierte und anschließend nicht mehr benötigte Daten aus dem Cache verdrängt werden können, ist es hier ausreichend diese zu invalidieren. Hier kann die hochparallel entworfene Cache Architektur ausgenutzt werden. Diese ermöglicht es die einzelnen Cachezeilen in einem Taktzyklus zu invalidieren. Hierdurch ist auch dieses Konzept vorhersagbar. Die partielle Invalidierung kann aufgrund der autarken Realisierung im Hintergrund durchgeführt werden. Dies bedeutet, dass in diesem Fall unabhängig von der Größe des Bereichs lediglich 6 Taktzyklen für die Invalidierung benötigt werden. Dies bedeutet eine Ersparnis von 99 % der zum Vergleich der für das Spülen des Cache benötigten Taktzyklen. Bei einem Realisierungsmehraufwand von etwa 15 % kann folglich vor allem bei großen Caches eine Effizienzsteigerung von bis zu 84 % erreicht werden.

7. Einordnung der Methodik

Dieses Kapitel gibt einen Überblick über die in dieser Arbeit vorgestellte Methodik der adaptiven Prozessorplattform. Es soll herausgearbeitet werden, in wie fern besondere Mechanismen dieser adaptiven Prozessorplattform mit welchem Mehrwert in welcher Konfiguration genutzt werden können. Hierbei wird insbesondere zwischen Einkern, Mehrkern- und Vielkernarchitekturen unterschieden und dabei die jeweiligen Vor- und Nachteile beim Einsatz der jeweiligen Architekturspezifika herausgearbeitet. Weiterhin werden die Anforderungen spezifiziert, die je nach Szenario verändert werden können. Es wird hier auch zwischen der Kern- bzw. der Systemsicht unterschieden, um so auf Basis des lokalen Optimums das jeweilige globale Optimum zu finden.

7.1. Ansatz für die Bewertung der adaptiven Prozessorplattform

Die adaptive Prozessorplattform ist ein flexibles Framework, das die Anpassung der Eigenschaften der Prozessorarchitektur an die Applikation zur Optimierung der jeweiligen Parameter für die Erzielung eines Optimums, bezüglich der Effizienz aus Systemsicht, ermöglicht. Diese Parameter können sowohl von der Software, der Applikation bzw. dem Laufzeitsystem, als auch von der Hardware, also der Prozessorarchitektur selbst, dynamisch angepasst werden. Hierbei besteht die Gefahr, dass einzelne Parameter, wie die eingangs dargestellten Optimierungsziele, die aus Sicht des Prozessors eine Verbesserung des lokalen Optimierungsziels darstellen das globale Optimierungsziel nicht erfüllen. Es kann jedoch sein, dass diese Ziele im

einzelnen Prozessor nicht zu erreichen sind, weil beispielsweise die benötigten Ressourcen aus Applikation- oder Systemsicht anderweitig effizienter genutzt werden können. Daher ist es unerlässlich den einzelnen Prozessor als Teil des Mehr- und Vielkernsystems zu verstehen, das die globalen Optimierungsziele erreichen soll. Diese werden vom Laufzeitsystem auf Basis der jeweiligen Applikation dynamisch festgelegt.

Die Tabelle 7.1 greift die in der Einleitung vorgestellte Klassifizierung auf und verfeinert diese in den folgenden Kapiteln anhand der einzelnen in dieser Arbeit vorgestellten Eigenschaften, wobei Performanz- und Flächeneffizienz quantitativ bewertet und der Leistungs- bzw. Energiebedarf auf Basis der durchgeführten Untersuchungen qualitativ abgeleitet werden.

Effizienz	Einkern	Mehrkern	Vielkern
a-Core	+ Performanz - Fläche - Leistung	++ Performanz - Fläche - Leistung	+++ Performanz - Fläche - Leistung
Gesamt	- Effizienz	+ Effizienz	++ Effizienz

Tabelle 7.1.: Klassifizierung der adaptiven Fähigkeit mit Einordnung der auf Grund der Anforderungen gesteigerten Effizienz im Mehr- und Vielkernprozessor.

Auf Basis dieser Klassifikation wird eine Einordnung möglich, in wie weit die einzelne Eigenschaft der Effizienz für den Einkern-, Mehrkern- oder Vielkernprozessor vorteilhaft ist. So wird es beispielsweise Fähigkeiten geben, die die Performanz des Einkernprozessors deutlich steigern. Dies geht jedoch mit einem deutlich erhöhten Einsatz an Ressourcen einher. Daher ist auch zu erwarten, dass die Leistungs- und Energieeffizienz sinkt. Daraus folgt bei einer deutlichen Performanzsteigerung eine leichte Effizienzsteigerung im Einkernprozessor. Da diese Eigenschaft für jeden Prozessor individuell realisiert werden muss, ist der Ressourcenmehrverbrauch sehr deutlich, so dass diese Eigenschaft nur in einem einzelnen Prozessor genutzt werden sollte, da für das Einkern- und Vielkernprozessorsystem keine Effizienzsteigerung mehr zu erwarten ist.

Andere Eigenschaften zeigen ihren Vorteil erst im Mehrkernsystem, da der Ressourcenmehraufwand für einen einzelnen Prozessor zu hoch ist, um

eine Effizienzsteigerung zu erzielen. Durch die intelligente Nutzung der zur Verfügung stehenden Eigenschaft, beispielsweise durch das Laufzeitsystem, kann auch die Effizienz des gesamten Vielkernprozessorsystems deutlich gesteigert werden, wenn insbesondere der Speicherflaschenhals adressiert wird und so Last vom *Network on Chip* (NoC) genommen werden oder Datenlokalität und speichernahe Verarbeitung ausgenutzt werden kann.

7.2. Adaptive Pipeline

Die adaptive Pipeline wurde als *Architecture Description Language* (ADL) Modell mit *Language for Instruction-Set Architectures* (LISA) umgesetzt. Anschließend wurde aus diesem Modell *Very High Speed Integrated Circuit Hardware Description Language* (VHDL) Code generiert, um diese Implementierung auf einem *Field Programmable Gate Array* (FPGA) zu realisieren. Im weiteren Verlauf der Arbeit hat sich herausgestellt, dass sich dieser Ansatz sehr gut eignet, um die einzelne Funktionalität zügig mit relativ geringer Entwicklungszeit einzuordnen. Obwohl die Modellierung als zyklenakkurates Modell realisiert wird, ist jedoch zu beachten, dass dies keine optimale Realisierung im Vergleich zu einer individuell optimierten VHDL Realisierung darstellt. Daher wird zunächst lediglich eine relative Betrachtung des Ressourcenverbrauchs durchgeführt, die im Weiteren um die absoluten Zahlen für eine Realisierung auf einem Xilinx *Virtex-6* (XC6VLX240T) (ML605) FPGA ergänzt wird. Diese Zahlen belegen einen deutlichen Realisierungsmehraufwand, da die Pipeline eines jedes einzelnen Prozessors erweitert werden muss. Durch den erhöhten Einsatz an Ressourcen ist es in diesem Fall auch möglich den Leistung-, und insbesondere den Energieverbrauch zu verringern, da Pipeline Register zwischen zwei Pipeline Stufen, beispielsweise durch *Zig Zag Power Gating* [BMH+13, MBFT13] ausgeschaltet werden können.

Diese Erkenntnisse sind in Tabelle 7.2 zusammengefasst. Für den Einkernprozessor wird also eine gesteigerte Performanz bei gesteigertem Ressourceneinsatz in Verbindung mit geringerer Leistungsaufnahme möglich, was zu einer leichten Effizienzsteigerung führt. Für den Mehrkern- und speziell den Vielkernprozessor ist keine weitere Effizienzsteigerung möglich, da die Fläche zur Realisierung mehrfach in jedem Prozessor benötigt wird. Hier-

Effizienz	Einkern	Mehrkern	Vielkern
a-Core	+ Performanz - Fläche + Leistung	+ Performanz - - Fläche + Leistung	+ Performanz - - Fläche + Leistung
Gesamt	+ Effizienz	- Effizienz	- Effizienz

Tabelle 7.2.: Klassifizierung der adaptiven Pipeline mit Einordnung der auf Grund der Anforderungen gesteigerten Effizienz im Einkernprozessor.

durch ist es auch nicht mehr möglich die Leistungsaufnahme zu verringern, so dass die Gesamteffizienz weiter sinkt.

7.3. Adaptive Superskalarität

Als Erweiterung der adaptiven Pipeline werden zwei Varianten einer superskalaren Prozessorpipeline modelliert. Auch hier kommt LISA zum Einsatz. Es wird eine grob- und eine feingranulare Realisierung untersucht. Es zeigt sich, dass die Performanz für einen *Bubblesort* Algorithmus um 16 % bzw. 18 % gesteigert werden kann. Gleichzeitig steigt der Ressourcenaufwand für die Realisierung einer *Arithmetic Logic Unit* (ALU) um 13,87 % an. Gleichzeitig ist auf Basis der einschlägigen Literatur ebenso eine Steigerung des Leistungsaufnahme zu erwarten. Je nach Komplexität der Algorithmen, die für eine *Out-of-Order* (OoO) Ausführung benötigt werden, kann der Leistungs- und Ressourcenverbrauch weiterhin deutlich ansteigen.

Effizienz	Einkern	Mehrkern	Vielkern
a-Core	++ Performanz - Fläche - Leistung	++ Performanz - - Fläche - - Leistung	++ Performanz - - Fläche - - Leistung
Gesamt	+ Effizienz	- Effizienz	- Effizienz

Tabelle 7.3.: Klassifizierung der adaptiven Superskalarität mit Einordnung der auf Grund der Anforderungen gesteigerten Effizienz im Einkernprozessor.

Zusammenfassend sind die Erkenntnisse bezüglich der superskalaren Pipeline in Tabelle 7.3 dargestellt. Ausgehend von der Performanzsteigerung im Einkernprozessor kann bei einer leichten Zunahme des Ressourcenaufwands und dementsprechender Leistungszunahme eine leichte Effizienzsteigerung erreicht werden. Diese Ressourcen müssen jedoch für jeden Prozessor individuell investiert werden, was mit einer Steigerung der Leistungsaufnahme einher geht. Es kann somit nur eine Effizienzsteigerung erreicht werden, wenn die Algorithmen die zur Verfügung stehenden Ressourcen optimal ausnutzen. Es ist daher im durchschnittlichen Fall mit einer leichten Verringerung der Effizienz im Mehrkern- bzw. im Vielkernfall zu rechnen.

7.4. Adaptive Sprungvorhersage

Die adaptive Sprungvorhersage ist ein interessanter Ansatz, da hier erstmals Methoden, die sich seit Jahrzehnten in Prozessoren bewährt haben, aufgegriffen werden und für die Einsatz in Mehrkern- und Vielkernprozessoren weiterentwickelt werden. Hierbei wird lediglich das grundlegende Design, welches aus der Literatur bekannt ist, und vor allem in kommerziellen Prozessoren zum Einsatz kam (vgl. Abschnitt 3.1) aufgegriffen und für den Einsatz in effizienten Systemen genutzt.

Hierzu werden etablierte Sprungvorhersagetechniken aufgegriffen und so erweitert, dass mit möglichst wenig Einsatz zusätzlicher Ressourcen ein effizientes Gesamtergebnis erzielt werden kann. Es entsteht ein modularer Metaprädiktor, der dynamisch, auf Basis des zur Laufzeit erlernten, umschalten kann, um eine verbesserte Trefferquote zu erreichen. Ein weiteres Ziel ist es insbesondere die Trefferrate von McFarling et al. [108] von 97 % zu erreichen, ohne hier individuelle Tabelleneinträge zu nutzen, was zu relativ hohem Ressourcenverbrauch führt.

Durch die konsequente Verfolgung dieser Ziele entsteht eine modulare Sprungvorhersage, bei der einzelne Eigenschaften, die zur Verbesserung der Trefferrate beitragen, individuell hinzugeschaltet werden können. Somit kann spezifisch auf die Klassen der Algorithmen bzw. deren Eigenschaften eingegangen werden, da es einige gibt, bei denen auch bei Einsatz von zusätzlichen Ressourcen keine weitere Performanzsteigerung zu erwarten ist. Bei anderen lohnt sich bereits die Investition von relativ wenig zusätzlichen

Ressourcen, beispielsweise in Form zusätzlicher globaler Tabelleneinträge, um zu einer Performanzsteigerung beizutragen.

Des Weiteren wird der Ansatz der Selbstadaptivität verfolgt, da es für den Anwendungsentwickler sowie für das Laufzeitsystem sehr schwierig ist, zur Laufzeit die richtigen Entscheidungen zur Parametrisierung der adaptiven Sprungvorhersage zu treffen. Daher wird ein leichtgewichtiger selbst lernender Algorithmus entwickelt und in der Sprungvorhersage umgesetzt, so dass die einzelnen Module des Metaprädiktors der Sprungvorhersage dynamisch hinzu geschaltet werden können, um eine höhere Trefferquote zu erzielen. Weiterhin ist der Algorithmus in der Lage den Arbeitspunkt, insbesondere den des *Dynamic History Length Fitting* (DHLF) zu verbessern, oder im Fall keiner weiteren Verbesserung diesen auch zu verwerfen.

Somit entsteht der erste Metaprädiktor, der sehr leichtgewichtig realisiert und selbstständig zwischen den einzelnen Prädiktoren (2-Bit, gShare, DHLF) umschalten kann, um den jeweils optimalen Arbeitspunkt zu erzielen. Im Kontext der Mehrkernprozessoren können Speicherressourcen geteilt werden. Dies bedeutet, dass eine definierte Anzahl an möglichen Tabelleneinträgen für alle Kerne des Mehrkernsystems zur Verfügung steht. Der Prozessor mit dem größten Bedarf sichert sich den exklusiven Zugriff auf einen Teil der Ressourcen, bzw. bekommt diesen mit dem Ziel der optimalen Verteilung zugewiesen. Hierdurch kann mit geringem Einsatz zusätzlicher Ressourcen im Vergleich zum Einkernprozessor eine Performanzsteigerung erzielt werden. Dies ist auch insbesondere interessant, da sich die somit erhöhte Trefferquote sehr positiv auf die Auslastung der Speicher und speziell der Kommunikationsverbindungen auswirkt. Hierdurch kann die Verdrängung im *Level 1* (L1) Cache minimiert und somit auch die Auslastung des kachellokalen Speicherbusses minimiert werden. Konsequenterweise wird auch der *Level 2* (L2) Cache effizient genutzt und hierdurch die Last auf dem NoC deutlich minimiert. Weiterhin werden die Zugriffe auf den *Double Data Rate* (DDR) Speicher minimiert, was deutlich zur Effizienzsteigerung beiträgt.

Konkret werden die Untersuchungen der Sprungvorhersage mit einem *Dhrystone*, einem *Coremark* und exemplarischen Algorithmen für jede Kategorie der *MiBench* Suite durchgeführt. Ziel ist es hierbei herauszufinden, mit welcher Kombination der einzelnen Vorhersagen die bestmögliche Effizienz bei der Erreichung des Ziel erhöhter Performanz erreicht werden kann. So

kann beispielsweise gezeigt werden, dass eine statische, sehr effizient realisierbare 2-Bit Vorhersage besser funktioniert, also eine höhere Trefferquote bei geringerem Ressourceneinsatz realisiert (1 - 256 Prädiktoren). Erst ab einem Einsatz von 512 Prädiktoren bei der dynamischen Vorhersage ist eine Performanzesteigerung gegenüber der statischen Vorhersage, jedoch unter erhöhtem Ressourceneinsatz zu erkennen. Bei *CoreMark* zeigt sich auch, dass die Kombination der realisierten Vorhersagen, also der Hybridprädiktor eine Erhöhung der Trefferquote sowie auch der Performanz mit sich bringt. Dies wird wie bereits beschrieben mit relativ geringem zusätzlichen Ressourceneinsatz realisiert, wodurch trotzdem Trefferquoten jenseits der 90 % erzielt werden können (vgl. Unterunterabschnitt 5.3.1.1).

Dies wird bei einem genaueren Blick auf Unterunterabschnitt 5.3.1.2 deutlich. Hier sind die einzelnen Aufwände zur Realisierung der Prädiktoren dargestellt. Es ist deutlich zu erkennen, dass die komplexeren Prädiktoren auf den einfacheren aufbauen und deren Ressourcen wieder verwenden. So benötigt der *gShare* mit 1024 Prädiktoren in etwa so viele Ressourcen wie der Metaprädiktor mit 32 Prädiktoren. Die Zusammenfassung zum Hybridprädiktor steigert die benötigen Ressourcen um lediglich 30 %.

Resultierend aus diesen Untersuchungen können die optimalen Parameter für die Lernphase und die Intervalle, sowie die *Branch History Register* (BHR) Länge bzw. Wiederholungen festgelegt werden bevor die *Local Minimum Avoidance* (LMA) aktiviert wird.

Abschließend kann gezeigt werden, dass einzelne Algorithmen, namentlich Sicherheit (AES) und Office (Dijkstra), nur mäßig von den investierten Ressourcen profitieren, während der Industriealgorithmus (Basicmath) und der *Dhrystone* fast 10 % und der Telekommunikationsalgorithmus (FFT) fast 25 % Performanz gewinnen. Dementsprechend steigt auch die Effizienz des Mehrkern- und vor allem auch des Vielkernsystems deutlich an. Diese Effizienzsteigerung hängt hierbei insbesondere von der Entfernung der Kachel zum globalen Speicher ab und kann über 30 % (2-Bit) bzw. über 40 % (gShare) betragen. Sobald das System unter Last ist, also die Laufzeit hauptsächlich von der Verfügbarkeit der Kommunikationsressourcen abhängt, kann die extrapolierte Effizienzsteigerung schnell 50 % übersteigen (vgl. Tabelle 5.6).

Diese Bewertung ist in Tabelle 7.4 zusammengefasst. Hierbei ist nochmals zu unterstreichen, dass die Ressourcen, die im Einkernprozessor eingesetzt

Effizienz	Einkern	Mehrkern	Vielkern
a-Core	+ Performanz - Fläche - Leistung	++ Performanz - Fläche - Leistung	+++ Performanz - Fläche - Leistung
Gesamt	+ Effizienz	++ Effizienz	+++ Effizienz

Tabelle 7.4.: Klassifizierung der adaptiven Sprungvorhersage mit Einordnung der auf Grund der Anforderungen gesteigerten Effizienz insbesondere im Mehr- und Vielkernprozessor.

werden leicht dem Mehrkernprozessor zur Verfügung gestellt werden können. Dies ist insbesondere vorteilhaft da diese Effizienzsteigerung dem gesamten Mehrkernsystem, vor allem wenn dies stark ausgelastet ist, zu Gute kommt, da die Last von den Kommunikationsressourcen genommen wird.

7.5. Adaptiver Cache

Beim adaptiven Cache werden zwei Kerngedanken verfolgt:

1. Die Konfigurierbarkeit zur Laufzeit aus der Anwendung sowie auch durch das Laufzeitsystem.
2. Die effiziente Nutzung und Wiederverwendung der bereits vorhandenen Ressourcen.

Im ersten Schritt wurden die Parameter bestimmt, durch die der Cache vollständig beschrieben wird. Die dynamische Anpassung erfolgt alleine durch die Veränderung dieser folgenden Parameter:

- Zeilenlänge in Wörtern (L_W)
- Satzgröße (s)
- Cachegröße (C) = Satzgröße (s) × Anzahl der Sätze (#s)

Alle Parameter werden im *Cache Configuration Register* (CCR) gespeichert und sind so für die Software lesbar, um den aktuellen Status des Systems abfragen zu können. Gleichzeitig wird die Anpassung durch eine Veränderung der Parameter in dem jeweiligen Register angestoßen.

Im nächsten Schritt erfolgt eine Einordnung der Parameter anhand der zwei numerischen Algorithmen Matrix Multiplikation (MMul) und LU-Zerlegung. Es kann gezeigt werden, dass sowohl die Anordnung der Daten durch den Programmierer als auch die Parametrisierung des Caches deutlichen Einfluss auf die Performanz der Applikation haben. Außerdem kann belegt werden, dass es neben der Betrachtung der Fehltrefferrate auch wichtig ist die absolute Anzahl an Fehltreffern zu minimieren. Während dies im Einkernprozessor nicht sofort offensichtlich ist, zeigt sich im Mehrkern- und vor allem im Vielkernprozessorsystem eine deutliche Zunahme der Zeit für die Kommunikation, die gegenüber der Zeit, die zum Rechnen verwendet wird, deutlich überwiegt.

Für den adaptiven Cache ist vor allem interessant, dass die gleichzeitige Vergrößerung der Cachegröße (C) und der Satzgröße (s) eine deutliche Reduktion der Fehltreffer erzeugt, was bei der Veränderung einzelner Parameter nicht in dem Maße festgestellt werden kann. Dieses Ergebnis unterstreicht den Ansatz den Cache anstatt in Sätzen in Wegen zu interpretieren, um für die Veränderung die Parallelschaltung der Wege ausnutzen zu können, die auch direkt die Assoziativität und durch die Verwendung der zusätzlichen Ressourcen auch die Cachegröße (C) verändert.

Bei der Anpassung der Parametrisierung der Cachearchitektur muss beachtet werden, dass bei der Verkleinerung bzw. bei der Vergrößerung Daten umsortiert oder synchronisiert werden müssen, damit die Transparenz des Caches zu jeder Zeit erhalten bleibt. Es ist offensichtlich, dass die Vergrößerung des Cache ohne vorherige Maßnahmen erfolgen kann. Es ist jedoch sinnvoll die Daten anschließend neu zu sortieren, um schnelle Zugriffe auf vorhandene Daten zu ermöglichen. Bei einer Verkleinerung des Caches muss ein partieller *Flush*, also ein partielles Zurückschreiben mit anschließendem Invalidieren der Cachezeilen erfolgen. Sobald dieser Vorgang abgeschlossen ist, kann der entsprechende Cacheblock abgeschaltet werden, um Energie zu sparen. Als Vorgriff auf die weiteren in dieser Arbeit vorgestellten Möglichkeiten sei angemerkt, dass der abgeschaltete Cacheblock den anderen Prozessoren auf der jeweiligen Kachel zur Verfügung gestellt werden kann. Auch der partielle *Flush*, der aus Zurückschreiben und Invalidieren der Cachezeilen besteht, wird im weiteren Verlauf der Arbeit nochmals aufgegriffen und dem Laufzeitsystem als individueller Befehl zur Cachesteuerung zur Verfügung gestellt.

Bei der Bewertung der Cachearchitektur zeigt sich, dass der Cachecontroller sehr effizient realisiert werden kann. Weiterhin zeigt sich, dass die verwendeten Speicherressourcen effizient genutzt werden und Raum für die Wiederverwendung der Ressourcen bieten. Eine erste Fallstudie belegt den Vorteil der dynamischen Cache Reallokation, die im Folgenden vorgestellt wird. Es kann gezeigt werden, dass durch die veränderte Zuordnung der insgesamt konstant vorhandenen Cacheressourcen eine Performanzsteigerung für einen Algorithmus erzielt werden kann, ohne die Performanz des anderen Algorithmus deutlich zu beeinflussen ($< 1\,\%$). Der Mehraufwand für die erneute bzw. veränderte Anordnung der Cacheressourcen ist hier bereits eingerechnet. Da die verwendeten Ressourcen konstant bleiben entspricht die Performanzsteigerung auch einer Effizienzsteigerung des Mehrkern Prozessorsystems durch die adaptive Cachearchitektur.

Bei der dynamischen Cache Reallokation wird die Cachesteuerung jeweils dem Prozessor zugeordnet. Der Cachespeicher wird jetzt zentral verwaltet und durch ein Verbindungsnetzwerk mit der Cachesteuerung verbunden. Dies ermöglicht es die Cacheresourcen dynamisch den einzelnen Prozessoren zuzuordnen. Hierbei ist jedoch zu beachten, dass dies einen enormen Freiheitsgrad in die Architektur bringt, die mit einem deutlichen Mehraufwand bei der Realisierung einhergeht. Daher wird festgelegt, dass nur benachbarte Prozessoren sich jeweils die Hälfte der vorhandenen Ressourcen teilen können. Dies verringert den Aufwand für die Realisierung deutlich und deckt eine Vielzahl an Optimierungsmöglichkeiten ab.

So kann beispielsweise zwischen 2-, 4- und 8-Wegen gewählt werden. Außerdem kann die Größe der Cacheblöcke, beispielsweise 256, 512 oder 1024 Byte pro Cacheblock gewählt werden. Dieser Parameter ist insbesondere abhängig von der gewählten Realisierung, da es direkt die minimalen vorhandenen Speicherressourcen der gewählten Realisierung abbildet. Hierbei ist es wichtig festzuhalten, dass die physisch vorhandenen Speicherressoucen genutzt werden. So ist beispielsweise die Anzahl der verwendeten *BlockRam* (BRAM) Blöcke zwischen $0,25\,kB$ und $4\,kB$ konstant. Erst darüber hinaus werden zusätzliche Speicherressourcen für die Realisierung benötigt. Dies ermöglicht eine sehr große Flexibilität bei der dynamischen Cache Reallokation ohne die Kosten für den Ressourcenaufwand in die Höhe zu treiben, wodurch die Effizienz des Gesamtsystems bei konstanten Kosten erhöht werden kann.

Wie zuvor angesprochen soll die Flexibilität der adaptiven Cachearchitektur neben dem Programmierer auch insbesondere dem Laufzeitsystem zugänglich gemacht werden. Hierdurch ergibt sich die Möglichkeit die Cachearchitektur neben den kachellokalen Optimierungen auch systemweit anzupassen, um so die Effizienz, insbesondere im Hinblick auf die Kommunikation, zu steigern. Hierzu werden die Funktionalitäten des partiellen *Flushes* aufgegriffen. Diese lassen sich in ein partielles *Write-Back*, also die Synchronisation von Daten mit der nächsten Ebene der Speicherhierarchie, und partielles Invalidieren, dem Verwerfen der bisher gespeicherten Daten unterteilen. Die Option des partiellen *Write-back* eröffnet die Möglichkeit Teile des Cache, beispielsweise einen Bereich, in dem Daten einer bestimmten Applikation liegen, am Ende einer Berechnung zu synchronisieren. So wird vermieden, dass Zwischenergebnisse die nur lokal benötigt werden, unnötig das Bussystem auslasten. Dies erhöht die Effizienz der einzelnen Applikation, aber insbesondere auch die Effizienz der einzelnen Kachel im Mehrkernsystem, da der Bus für die Kommunikation der relevanten Daten genutzt werden kann und nicht durch nicht mehr benötigte Daten verstopft wird. Dies ist vorteilhaft, da die Kommunikation über das NoC deutlich teurer ist und mit dem Abstand zum Hauptspeicher weiter ansteigt.

Im nächsten Schritt ist erkennbar, dass Zwischenergebnisse gespeichert werden, die ab einem bestimmten Punkt der Berechnung nicht mehr benötigt werden. Diese belegen jedoch Platz im Cache, was nach einiger Zeit zu Verdrängung führt, um Speicherplatz für die neueren Daten zu schaffen. Da diese Daten jedoch im weiteren Verlauf nicht mehr benötigt werden, ist es sinnvoll diese noch im L1 Cache zu invalidieren, um den begrenzten Speicherplatz des Caches effizient zu nutzen.

Eine weitere neue Eigenschaft der adaptiven Cachearchitektur ist es die oben beschriebenen Fähigkeiten im Hintergrund umzusetzen. Dies bedeutet, dass die Invalidierung von der Software angestoßen werden kann und startet, sobald die Instruktion aus der Pipeline an die Cachesteuerung übergeben wurde. Es wird ein Daten Transfer Szenario untersucht: Hierbei werden Daten zwischen Kacheln, also über nicht cachekohärente Bereiche hinweg, kopiert, also geklont. Beim Klonen werden die Daten über die Speicherhierarchie kopiert, so dass in der Partition nicht gültige Daten stehen. Eine detaillierte Beschreibung des Szenarios ist in Unterabschnitt 6.5.1 und [MT17] zu finden. Hierdurch wird es notwendig die Daten nach dem

Abschluss des Klonens zu invalidieren, damit diese nicht mehr in der ehemaligen Partition zur Verfügung stehen.

Im Gegensatz zu bisherigen Ansätzen bei ARM Prozessoren, insbesondere dem ARM1136J(F)-S [7], muss die Prozessorpipeline nicht angehalten werden. Dementsprechend kann die Abarbeitung fortgesetzt werden, während die Cachesteuerung den neuen Zustand bereits kennt und die Invalidierung im Hintergrund zeilenweise vornimmt. Dies erhöht die Performanz deutlich. In Abhängigkeit von der Cachegröße ist es möglich bis zu 99 % der Zyklen einzusparen, in denen der Prozessor bisher angehalten wurde. Es werden etwa 15 % Ressourcen mehr für die Cachesteuerung benötigt, was zu einer deutlichen Effizienzsteigerung im Bezug auf das Mehrkern- und Vielkernsystem führt. Es gibt weitere Arbeiten, in denen bereichsbasierte Cache Operationen vorgeschlagen werden [31, 129, 135]. Auf Basis der durchgeführten Recherchen wird in dieser Arbeit erstmals eine adaptive Cache Architektur prototypisch umgesetzt, die in der Lage ist die bereichsbasierten Instruktionen für die Software zur Verfügung zu stellen.

Zusammenfassend lässt sich die adaptive, dynamische Cachearchitektur, wie in Tabelle 7.5 gezeigt, einordnen.

Effizienz	Einkern	Mehrkern	Vielkern
a-Core	+ Performanz - Fläche + Leistung	++ Performanz + Fläche + Leistung	+++ Performanz ++ Fläche ++ Leistung
Gesamt	+ Effizienz	++ Effizienz	+++ Effizienz

Tabelle 7.5.: Klassifizierung des adaptiven Cache mit Einordnung der auf Grund der Anforderungen gesteigerten Effizienz insbesondere im Mehr- und Vielkernprozessor.

Durch die Erhöhung der Performanz im Einkernfall, einhergehend mit relativ geringerem Ressourceneinsatz, kann eine sehr effiziente Realisierung erzielt werden. Dies ist umso mehr der Fall, wenn die vorhandenen Ressourcen zwischen den Rechenkernen im Mehrkernprozessor auf einer Kachel geteilt werden. Am effizientesten ist der Ansatz, wenn er im Vielkernprozessor die Kommunikationsressourcen entlastet und somit zur Effizienz des Gesamtsystems beiträgt.

8. Schlussfolgerung

In der vorliegenden Arbeit wird eine adaptive Prozessorplattform entworfen und prototypisch umgesetzt. Auf Basis des Prototypen wird eine Bewertung der einzelnen Parameter der jeweiligen adaptiven Prozessorarchitektur vorgenommen. Durch die Erweiterung einer *General-Purpose* Prozessor (GPP)-Architektur können die Vorteile bestehender Prozessorarchitekturen, wie beispielsweise Bekanntheit, Beliebtheit, Flexibilität oder Programmierbarkeit erhalten werden. Bisherige Nachteile, wie die Optimierung auf den mittleren Anwendungsfall, werden durch die adaptiven Fähigkeiten adressiert. Das Leistungs- und Energiebudget, das insbesondere in eingebetteten Systemen eine Hauptanforderung darstellt, wird durch die Wiederverwendbarkeit einzelner Ressourcen verbessert. Hierdurch ergibt sich ein weiterer Vorteil beim Einsatz der adaptiven Prozessorplattform in Mehrkernarchitekturen, da die Ressourcen nur einmal pro Kachel aufgewendet werden müssen. Dies hat auch positive Auswirkungen auf die Nutzung mehrerer Kacheln in einem Vielkernsystem, da die effiziente Nutzung der vorhandenen Ressourcen die Kommunikationsinfrastruktur entlastet und so Kosten für die Realisierung sowie auch Leistung und Energie im Betrieb eingespart werden können.

Damit die adaptive Prozessorarchitekturplattform für den Programmierer, sowie auch für das Laufzeitsystem parametrierbar ist, werden Erweiterungen des Instruktionssatzes konzipiert, deren Werte transparent in Statusregistern abgelegt werden. Dies hat den Vorteil, dass die jeweilige Prozessorarchitektur auch selbstadaptiv Anpassungen vornehmen kann, die für das System im Register sichtbar werden. Somit kann das Laufzeitsystem auf Basis der lokalen Optimierung im Prozessor eine systemweite Optimierung vornehmen, um die Effizienz des Gesamtsystems weiter zu verbessern. Es entsteht hierbei eine neuartige adaptive Prozessorarchitektur, die selbstadaptive, sowie auch laufzeit-adaptive Mechanismen der Software

umsetzen kann. Dies ist insbesondere die prototypische Realisierung auf rekonfigurierbarer *Field Programmable Gate Array* (FPGA) Hardware, so dass hier nicht nur eine Bewertung der Performanz, sondern auch eine Bewertung der Effizienz treffen lässt, die sich aus der Performanz, dem Realisierungsaufwand und der Energieeffizienz zusammensetzt. Dies ist insbesondere vorteilhaft, da bei der abschließenden Klassifizierung zwischen Einkern-, Mehrkern-, und Vielkernarchitektur unterschieden werden kann. Somit kann gezeigt werden, dass Eigenschaften, die der adaptiven Prozessorarchitektur hinzugefügt werden, und im Einkern Prozessor nicht effizient realisiert werden können, insbesondere im Mehrkern- und Vielkernsystem ihre Stärken ausspielen. Hierbei spielt die dynamische Wiederverwendung bzw. Umverteilung der Ressourcen zwischen den einzelnen Prozessoren eine wichtige Rolle. Auch die feingranularere Aufteilung der einzelnen Instruktionen, um dem System einen detaillierten Zugriff auf Parameter des Prozessors zu geben, erweist sich insbesondere im Mehrkernsystem als sehr vorteilhaft.

Angefangen bei der adaptiven Pipeline zeigt sich, dass der Realisierungsaufwand im einzelnen Prozessor relativ groß ist. Andererseits birgt das dynamische Verkürzen in langen Pipelines das Potential zur Einsparung von Energie. Diese Idee wird bei der adaptiven Superskalarität aufgegriffen. Hier werden funktionale Einheiten – vom Addierer bis hin zur voll umfänglichen ALU – statisch zur Verfügung gestellt und dynamisch zugeordnet, so dass zum einen der funktionale Umfang der einzelnen Pipeline, sowie zum anderen auch der Grad der Parallelität dynamisch gewählt werden kann. Dieses in der Realisierung aufwändige Konzept spielt seinen Vorteil insbesondere bei stark unterschiedlichen Lasten aus, so dass die vorhandenen Ressourcen effizient aufgeteilt und vollständig ausgelastet werden können, was zu einer leichten Effizienzsteigerung des Gesamtsystems führt.

Schnelle Bewertungen auf hoher Abstraktionsebene sind der Schlüssel um den Herausforderungen der steigenden Komplexität und benötigten Effizienz durch adaptive Prozessoren in Mehr- und Vielkern Systemen gerecht zu werden. Hierfür wird zur Entwicklung des adaptiven Prozessors (a-Cores) auf Basis der *Architecture Description Language* (ADL) *Language for Instruction-Set Architectures* (LISA) ein Entwicklungsfluss vorgestellt und bewertet. Daraus können die Softwareentwicklungswerkzeuge, die Simulationsmodelle und die synthetisierbaren *Hardware Description Language* (HDL)-Modelle generiert werden. Der präsentierte integrierte

Entwicklungsfluss erweitert den Stand der Technik im Systementwurf mit der Vorstellung der adaptiven Simulationen (*a*-Sims), um den Anforderungen an die Entwicklung adaptiver Prozessoren gerecht zu werden. In den meisten Fällen ist es wünschenswert den Prozessor direkt aus dem Modell heraus zu generieren. Hierdurch kann das Modell iterativ verbessert werden und es wird ein wichtiger Kreislauf im Entwicklungsfluss zur Verbesserung adaptiver Prozessoren hergestellt, da die Änderungen direkt in das Modell fließen, das anschließend für die Evaluation generiert wird.

Die ADL LISA eignet sich gut um einen adaptiven Prozessor mit wenigen applikationsspezifischen Erweiterungen zu entwickeln. Hierdurch können neue Ideen, speziell das Hinzufügen neuer Instruktionen in einen bestehenden Instruktionssatz, in relativ kurzer Zeit evaluiert werden. Das LISA selbst sowie auch die daraus generierten Werkzeuge und insbesondere das SystemC-Modell sind sehr gut geeignet, um den Prozessor und dessen applikationsspezifische Erweiterungen zu evaluieren. Die Modelle können daher sehr gut als Basis für weitere Entwicklungen eines optimierten Prozessors genutzt werden. Die Weiterentwicklungen der HDL Beschreibungen auf Basis der Erkenntnisse dieser Kapitel werden in den folgenden Kapiteln nicht mehr in das LISA-Modell eingefügt, da die feingliedrigen Details nicht im LISA-Modell abgebildet werden können.

Bei der adaptiven Sprungvorhersage wird ein modularer Aufbau realisiert, so dass die Prädiktoren die Elemente der jeweils einfacheren Vorhersage wiederverwenden. Außerdem wird ein leichtgewichtiger selbst lernender Algorithmus realisiert, der in der Lage ist die Vorhersage zur Laufzeit zu verbessern. Dies trägt neben einer Verbesserung der Trefferrate vor allem zur Reduktion der absoluten Anzahl an Fehltreffern bei. Jede einzelne richtige Vorhersage verbessert die Ausnutzung der vorhandenen Daten im Cache, da beispielsweise kein Kontextwechsel erfolgt, der eine Allokation im Cache notwendig macht. Dies ist wertvoll, da sich die Trefferrate mit steigendem Abstand zum Hauptspeicher stark auf die Ausführungszeit auswirkt, hauptsächlich wegen der Auslastung des *Network on Chip*s (NoCs). Somit wird vor allem im Mehrkernsystem durch das Teilen vorhandener Ressourcen zum Anlegen der Prädiktoren ein Beitrag zur Effizienzsteigerung geleistet, was sich auch im Vielkernsystem durch Verringerung der Last auf dem NoC positiv auf auswirkt.

Der adaptive Cache wird als neuartige modulare Architektur entwickelt, die ein flexibles Parametrieren dieser Architektur ermöglicht. Hierbei können durch Ausnutzung der räumlichen- und zeitlichen Parallelität optimierte Applikationen effizient ausgeführt werden, um die Fehltrefferrate und insbesondere die absolute Anzahl an Fehltreffern zu minimieren. Des Weiteren wird die Cachearchitektur für die Mehrkern Prozessorarchitektur erweitert. Dies eröffnet neue Möglichkeiten der Parametrierung durch die effiziente Wiederverwendung der vorhandenen Speicherressourcen. Es kann insbesondere gezeigt werden, dass die bereits vorhandenen Ressourcen effizient zugeordnet und verwendet werden, da hier für die Performanzsteigerung kein weiterer Ressourceneinsatz nötig ist. Abschließend kann in einem Daten Transfer Szenario gezeigt werden, dass sich der Einsatz der neuen feingranularen Instruktionen lohnt, um beispielsweise den Cache teilweise zu invalidieren anstatt diesen wie bisher üblich vollständig zu flushen. Hierbei ist es insbesondere vorteilhaft, dass die modulare Cachearchitektur die Verwaltung der Invaliderung selbst übernehmen kann. Dadurch ist also keine Unterbrechung der Abarbeitung notwendig, wodurch die Effizienz des Gesamtsystems deutlich gesteigert werden kann.

Zusammenfassend kann gezeigt werden, dass sich ein adaptives Prozessorsystem mit relativ geringem Entwicklungsaufwand realisieren lässt. Die einzelnen adaptiven Komponenten erzeugen eine Steigerung der Effizienz, da der Einsatz zusätzlicher Ressourcen von der Ersparnis der Laufzeit deutlich übertroffen wird. Das System konnte von einem Einkern- über einen Mehrkern- bis hin zu einem Vielkernsystem skaliert werden, wobei die Basis hierfür mit einem zyklenakkuraten High-Level Simulationsmodell gelegt wurde, welches anschließend in eine HDL Beschreibung übertragen wurde. Hierdurch wird der neuartige Prozessor realisiert, der durch das Zusammenspiel von Hardware und Software eine Effizienzsteigerung erzielt, indem Applikation, Laufzeitumgebung, Compiler und auch die Hardware selbst den adaptiven Prozessor unterbrechungsfrei parametrieren.

A. Anhang

A.1. Integrierter Entwicklungsfluss für die adaptive Prozessorarchitektur

Auf Basis der Erkenntnisse aus dem vorangegangenen Kapitel, soll in diesem Kapitel evaluiert werden, in wie weit sich das Werkzeug *Processor Designer* (PD) für die Entwicklung von adaptiven Prozessor Architekturen eignet. Wie in Abschnitt 3.2 beschrieben, gibt es nur wenige Veröffentlichungen, welche sich mit der Entwicklung von adaptiven Prozessoren auf hoher Entwurfsebene beschäftigen. Diese wirft einige Fragen auf, die bisher noch nicht beantwortet wurden:

- Ist es möglich einen adaptiven Prozessor (a-Core) auf hoher Abstraktionsebene in einer einzigen abstrakten Beschreibung zu entwickeln und die benötigten Werkzeuge aus dieser hohen Entwurfsebene zu generieren?
- Können automatisch generierte Werkzeuge wie beispielsweise Simulatoren, Compiler und auch die *Hardware Description Language* (HDL) Beschreibung für den kompletten Entwurfszyklus des adaptiver Prozessor (a-Core) verwendet werden?
- Ist es möglich, falls die oben genannten Punkte zutreffen, den Prozessor um die jeweiligen anwendungsspezifischen Fähigkeiten zu erweitern?

Komplexe Aufgaben sollen aus der Software in die Hardware verlagert werden, um eine effizientere Ausführung einer spezifischen Anwendung zu ermöglichen. Dies eröffnet neue Möglichkeiten, die aufgrund des teuren

und langen Entwicklungszyklus und damit auch zu langen *Time to Market* (TtM) in kommerziellen Prozessoren bisher nicht denkbar gewesen wären.

Die Methodik wurde auf dem *Symposium on VLSI* (ISVLSI) [THDB13] und in dem daraus resultierenden Buchkapitel [THMB16] veröffentlicht.

A.1.1. *LISA model-based ideal Cortex-M1* (LImbiC)

Dieses Kapitel stellt das Design, die applikationsspezifischen Erweiterungen sowie die Realisierung des *LISA model-based ideal Cortex-M1* (LImbiC) mittels der *Language for Instruction-Set Architectures* (LISA) vor.

A.1.1.1. Design des applikationsspezifischen Prozessormodells

Dieser Abschnitt beschreibt das Design des *Language for Instruction-Set Architectures* (LISA)-Modells, welches die grundlegenden Funktionalitäten auf Basis des *Advanced RISC Machines Ltd.* (ARM) *Cortex-M1* und seiner *ARMv6* Instruktionssatz Architektur (ISA) realisiert. *LISA model-based ideal Cortex-M1* (LImbiC) wird mit 2 zusätzlichen Instruktionen erweitert, dessen Syntax und Kodierung als Erweiterung der bestehenden ISA definiert wird. Das Design ermöglicht den Vergleich von drei Prozessormodellen in Unterabschnitt A.1.3:

- Basic-LImbiC - ohne applikationsspezifische Erweiterungen
- LImbiC - mit applikationsspezifischen Erweiterungen
- Small-LImbiC - reduzierter *Application-Specific Instruction-Set Processor* (ASIP) zur Kantenerkennung mit Faltung und Cannyfilter

LImbiC Prozessor Architektur LImbiC wird als 32 Bit Harvard Architektur entworfen. Ein *Read-only Memory* (ROM) wird genutzt, um das Programm zu speichern. Ein *Random-access Memory* (RAM) wird genutzt, um die Daten zu speichern. Der Datenspeicher ist für die Anwendung der Bildverarbeitung angepasst, in dem er 2 Byte Speicherzugriffe erlaubt, so dass einzelne Pixel adressiert werden können. Im Gegensatz zum ARM spezifischen *little endian* format, kommt das *big endian* Format zum Einsatz.

Das Bild ist Pixel für Pixel im Speicher abgelegt, so dass der Applikationsentwickler die Bildgröße kennen muss, um die gespeicherten Daten korrekt verarbeiten zu können. Die ISA des ARM *Cortex-M1* wird als Basis und Leitlinie für den Entwurf des LImbiC genutzt. Im Gegensatz zum ARM Prozessor muss der LImbiC für diese Applikation keine Interrupts oder *Exceptions* unterstützen, da hier kein Betriebssystem zum Einsatz kommt. LImbiC besteht demnach entsprechend zu [9] aus 2 Instruktionsklassen:

- Daten Transport und Verzweigungsinstruktionen
- Arithmetischen, logischen und bitmanipulativen Instruktionen

Da der *Cortex-M1* und damit auch der LImbiC die ARM *Thumb* Instruktionssatz Architektur nutzt, werden sowohl 16 Bit als auch 32 Bit Instruktionen unterstützt. Alle Instruktionen werden von der *pre-UAL*[1] unterstützt, so dass es möglich ist die existierenden ARM Werkzeuge, wie beispielsweise den Compiler, für die Softwareentwicklung zu nutzen. Ziel ist es kompatibel zum Maschinencode des *Cortex-M1* zu bleiben.

LImbiC hat 13 32 Bit *General Purpose* Register, sowie einen 32 Bit *Stack Pointer* (SP), ein *Link Register* (LR) und einen *Program Counter* (PC). Das Programm Status Register wurde zu einem applikationsspezifischen Statusregister (ASSR) herunter skaliert, das nur den Status der *Arithmetic Logic Unit* (ALU) und den Modus des Prozessors abbildet. Zudem wurden keine weiteren Register hinzugefügt, um die Kompatibilität der vorhandenen Werkzeuge nicht einzuschränken.

LImbiC nutzt eine einfache dreistufige Pipeline, die aus *Instruction Fetch* (FE), *Decode* (DE) und *Execute* (EX) besteht. Dementsprechend gibt es 2 Pipeline Register, die jeweils den Programmzähler und die Instruktion von Stufe zu Stufe weitergeben. In der *Instruction Fetch* Stufe werden 32 Bit Wörter aus dem Speicher geholt. Wenn ein Wort nicht nur eine 32 Bit Instruktion sondern zwei 16 Bit Instruktionen beinhaltet, werden diese nacheinander in getrennten Zyklen an die *Decode* Stufe weitergeleitet. Abschließend wird die Instruktion in der *Execute* Stufe abgearbeitet. Hier erfolgen auch die Speicherzugriffe, da keine eigene *Memory* Stufe existiert.

[1] *pre-UAL* ist eine frühe *Thumb* Syntax, die nur auf die *Thumb* ISA, nicht auf die *Thumb-2* ISA passt [14].

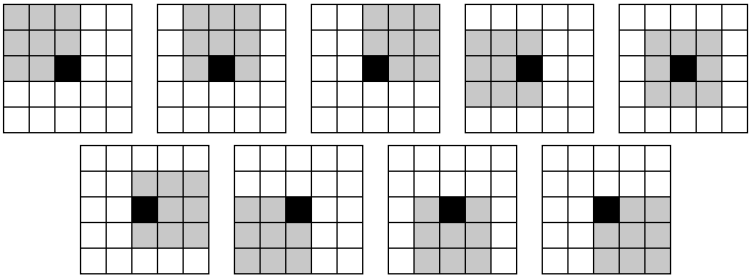


Abbildung A.1.: Benachbarte Pixel, die für eine Berechnung der Index Pixel notwendig sind: Faltung mit einem berechneten Pixel.

A.1.1.2. Erweiterung des applikationsspezifischen Prozessormodells

Dieser Abschnitt stellt die applikationsspezifischen Erweiterungen des LImbiC mit den Bildverarbeitungsalgorithmen aus Abschnitt 2.8 vor. Das Hauptziel ist hierbei die Reduktion der Zyklenzahl und somit der Ausführungszeit. Hierbei spielt die Anzahl der Speicherzugriffe eine große Rolle. Eine Bewertung erfolgt abschließend in Unterabschnitt A.1.3.

Faltung Für die Berechnung einer Faltung müssen alle Pixel eines Bildes neunmal aus dem Speicher geladen werden. Abbildung A.1 illustriert wie die graue Faltungsmatrix durch das 25 Pixel Bild wandert. Das schwarz markierte Pixel muss neunmal aus dem Speicher geladen werden. Auch wenn der Algorithmus das Pixel im Stack speichern würde, würde die Anzahl der für die Berechnung notwendigen Zyklen nicht signifikant verringert werden, da es gibt nicht genügend *Low* Register, um alle neun Pixel zwischenzuspeichern, die für die Berechnung benötigt werden. Die einzige Möglichkeit ist es also drei Pixel einer Zeile im Stack zwischen zu speichern. Zwei der Pixel werden auch für die Berechnung des nächsten Pixels benötigt. Hierzu werden die *Push* und *Pop* Instruktionen der ARM ISA genutzt. Dieser Weg ist insbesondere für große Matrizen sinnvoll.

Durch die Bestimmung mehrerer Pixel in einem Taktzyklus kann die Anzahl der Speicherzugriffe signifikant reduziert werden. Abbildung A.2 verdeutlicht diesen Ansatz für die gleichzeitige Berechnung von fünf Pixeln.

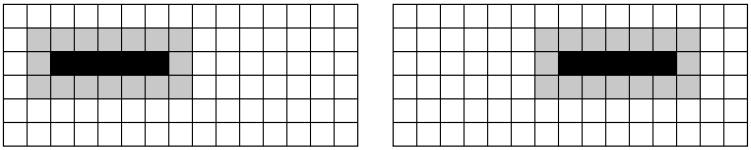


Abbildung A.2.: Benachbarte Pixel, die für eine Berechnung der Index Pixel notwendig sind: Faltung mit fünf berechneten Pixeln.

Opcode	Typ	Faltung
00	-	*reserviert*
01	SX	Horizontaler *Sobel* Operator (S_X)
10	SY	Vertical *Sobel* Operator (S_Y)
11	GA	*Gauss'scher* Filter

Tabelle A.1.: Definierte Varianten der Faltung.

Es werden so nur 21 anstatt der 45 notwendigen Speicherzugriffe benötigt. Im Allgemeinen sind $3(n+2)$ anstatt $9n$ Speicherzugriffe notwendig.

Durch die Latenz beim Speicherzugriff, welche durch die Speicherhierarchie und die Caches hervorgerufen wird, kann davon ausgegangen werden, dass sich der Speicherzugriff in der Realität weiter verkürzt.

Es ist jedoch ein zusätzlicher Aufwand von sechs Speicherzugriffen pro Pixel Cluster notwendig, wenn benachbarte Pixel berechnet werden. Die klassische *Load-Store* Architektur der ARM Architektur wird hierbei nicht verändert sondern lediglich effizienter genutzt, da der Anwendungsprogrammierer die Anordnung der Pixel kennt und effizient auf die Pixel zugreift.

Die Kodierung der neuen Instruktion ist kompatibel zur *ARMv6-M* ISA. Eine bislang undefinierte Instruktion wird herangezogen und auf deren Basis eine neue Kodierung erstellt. Die vier niedrigwertigsten Bits sind laut der Architekturspezifikation Null [9]. So können die niedrigwertigsten 2 Bits zur Definition der Varianten der Faltung verwendet werden. Tabelle A.1 zeigt die definierten Varianten. Die Kodierung '00' ist bereits reserviert. Es gibt sechs weitere Bits neben der Typdefinition, die die Startadresse des Eingabe bzw. Ausgabe Bildes definieren. Die resultierende Instruktion mit

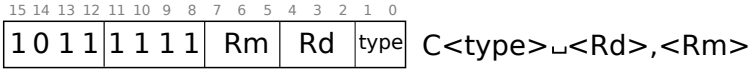


Abbildung A.3.: Faltung Instruktion: Kodierung und Syntax.

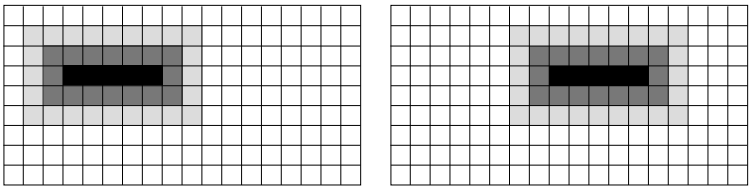


Abbildung A.4.: Benachbarte Pixels, die für eine Berechnung der Index Pixel notwendig sind: Cannyfilter Berechnung mit fünf Pixeln.

deren Kodierung und Syntax ist in Abbildung A.3 gezeigt. Die Faltung wird durch das C für *Convolution* beschrieben. Als Typen stehen *Sobel* in x-Richtung (S_X), *Sobel* in y-Richtung S_Y und *Gauss'scher* (GA) zur Verfügung.

Eine weitere applikationsspezifische Anpassung ist die automatische Inkrementierung der Registerwerte ähnlich zu den *LDM* bzw. *STM* Instruktionen. Die Register werden automatisch um n Pixel inkrementiert, sobald diese berechnet wurden. Somit sind durch die automatische Inkrementierung keine zusätzlichen *ADD* Instruktionen notwendig, um die Faltung auszuführen.

Cannyfilter Wie bei der vorangegangenen Faltung auch, soll die applikationsspezifische Implementierung des Cannyfilters die Ausführungszeit minimieren, in dem die Speicher Zugriffe und Verarbeitungszyklen minimiert werden. Auch diese Instruktion wird so implementiert, dass mehrere Pixel gleichzeitig verarbeitet werden. Wie in Abbildung A.4 gezeigt, benötigt der Cannyfilter mehr als nur die direkt benachbarten Pixel. Um die Pixel zu berechnen, muss zunächst die Richtung φ des Gradienten berechnet werden. Hierzu werden die Grauwertgradienten M der benachbarten dunklen grauen Pixel berechnet. In diesem Fall werden die hellgrauen Pixel benötigt. Sie müssen zusätzlich geladen werden, so dass $5(n+4)$ Pixel benötigt werden um n Pixel eines Bildes zu berechnen. Wenn jedes

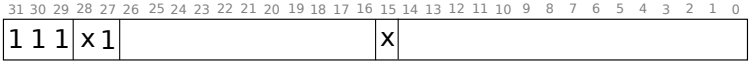
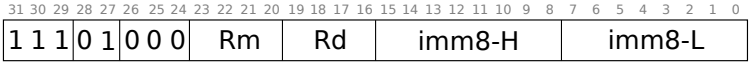


Abbildung A.5.: Nicht implementierte Instruktion Kodierung entsprechend [9].



CAN␣<Rd>,<Rm>,#<imm8-L>,#<imm8-H>

Abbildung A.6.: Cannyfilter Instruktion: Kodierung und Syntax.

Pixel einzeln berechnet werden würde, müssten 25 Pixel aus dem Speicher geladen werden, was $25 \times n$ Pixel ergibt. Dementsprechend, wie in Abbildung A.4 gezeigt, wird die Anzahl der Speicherzugriffe von 125 auf 45 für die gleichzeitige Berechnung von fünf Pixeln reduziert. Wie in dem Fall der Faltung, ergeben sich hier ein Mehraufwand von 15 Zugriffen.

Eine Instruktion mit entsprechender Kodierung und Syntax wird aus den nicht benutzten Instruktionen der *ARMv6-M* ISA gewählt. Zusätzlich zu den beiden Speicheradressregistern des Eingangs- und Ausgangsbildes müssen die zwei Schwellenwerte T_1 und T_2 in der Instruktion abgebildet werden. Selbst wenn Register genutzt werden würden, um die Schwellenwerte zu speichern, wären 12 Bits notwendig um vier Register Adressen abzubilden. Da in der *ARMv6-M* ISA keine ungenutzte 16 Bit Instruktion verfügbar ist, wird eine 32 Bit Instruktion gewählt.

Abbildung A.5 zeigt die 32 Bit Instruktion mit Kodierung und Syntax. *ARMv6-M* Instruktionen beginnen immer mit einer führenden '111' [9]. Die darauffolgenden Bits dürfen nicht '00' sein, um eine Verwechselung mit Verzweigungsinstruktionen zu vermeiden.

Abbildung A.6 zeigt die Adaptierung der Instruktionskodierung für den Cannyfilter mit der Kodierung '01'. Jeder Schwellenwert muss die Werte von 0 bis 255 beinhalten, was eine 8 Bit *Immediate* notwendig macht. Die verfügbare Kodierung ermöglicht feingranulare Schritte des Farbspektrums, jedoch werden für diese Anwendung nur 8 Bit Graustufen Bilder verarbeitet. Im Gegensatz zur Faltung handelt es sich bei beiden Registern um *High Register*.

A.1.1.3. Realisierung des applikationsspezifischen Prozessormodells

Die *Architecture Description Language* (ADL) LISA wird verwendet um LImbiC mit seinen Erweiterungen, wie im vorigen Abschnitt beschrieben, für die Beschleunigung von Bildverarbeitungsalgorithmen für beispielsweise eine Faltung und einen Cannyfilter zu modellieren. Die Evaluation des Prozessors und der applikationsspezifischen Erweiterungen wird im folgenden Abschnitt präsentiert. Weiterführende technische Informationen sind in der Abschlussarbeit von Tanja Harbaum [Har12] zu finden.

```
1  OPERATION convolution IN pipe.EX{
2   DECLARE{ GROUP addr_in,addr_out = { regL }; LABEL c0,c1;}
3   CODING { 0b10111111 addr_in addr_out c1=0bx[1] c0=0bx[1]}
4   SYNTAX { "C" }
5   IF ( (c1==1)&&(c0==1) ) THEN { SYNTAX {"GA "}
6    BEHAVIOR{
7     for(i=0;i<4;i++){
8      img[i]=(int)dat[i]+((int)dat[i+1]<<1)+(int)dat[2]
9      +((int)dat[i+6]<<1)+((int)dat[i+7]<<2)+((int)dat[i+8]<<1)
10     +(int)dat[i+12]+((int)dat[i+13]<<1)+(int)dat[i+14];
11    img[i]=img[i]>>4;
12    }}}
13  IF ( (c1==0)&&(c0==1) ) THEN { SYNTAX {"SX "}
14   BEHAVIOR{
15    for(i=0;i<4;i++){
16     img[i]=-(int)dat[i]+(int)dat[2]
17        -((int)dat[i+6]<<1)+((int)dat[i+8]<<1)
18        -(int)dat[i+12]+(int)dat[i+14];
19    if (img[i]<0)img[i]=-img[i];
20    img[i]=img[i]>>2;
21    }}}
22  IF ( (c1==1)&&(c0==0) ) THEN { SYNTAX {"SY "}
23   BEHAVIOR{
24    for(i=0;i<4;i++){
25     img[i]=-(int)dat[i]-((int)dat[i+1]<<1)-(int)dat[2]
26       +(int)dat[i+12]+((int)dat[i+13]<<1)+(int)dat[i+14];
27    if (img[i]<0)img[i]=-img[i];
28    img[i]=img[i]>>2;
29    }}}
30  SYNTAX {adress_out "," adress_in}
31  BEHAVIOR{SRAM[address]=img[i];R[addr_in]+=4; R[addr_out]+=4;}}
```

LISA-Code A.1: LISA Realisierung der Faltung

Faltung Zunächst wurde der Gauss'sche- und der Sobelfilter wie in LISA-Code A.1 realisiert, um den Cannyfilter zu ermöglichen. Diese Realisierung geht von einem 256×256 Pixel Eingabebild aus. Aus dem Register, das die Startadresse enthält, werden alle benötigten Pixel für die Faltung indexiert und aus dem Speicher geladen. Für den Sobelfilter wird ein 254×254 Pixel Eingabebild erwartet. Entsprechend der Zeilenlänge werden die benötigten Pixel aus dem Speicher geladen. Nachdem alle Daten aus dem Speicher geladen wurden, wird die Faltung mit einer entsprechenden Faltungsmatrix gestartet. Der Absolutwert wird berechnet und für den Sobelfilter durch vier geteilt. Dies wird getan um ein 8 Bit Grauwert Bild zu erhalten. Somit wird das Spektrum von 1024 auf 256 Grauwerte reduziert.

Anschließend werden die vier berechneten Pixel in den Speicher geschrieben. Die Startadresse des Ausgabebildes wird hierzu aus dem zweiten Register gelesen. Abschließend werden die Register *Rd* und *Rn* geschrieben. Alle Bilddaten liegen in einem eindimensionalen Array im LISA-Code vor, da der *Processor Designer* zweidimensionale Arrays nicht zur HDL Code Generierung unterstützt.

Cannyfilter Um den Befehl für den Cannyfilter zu realisieren, werden zunächst zwei *High Register* und zwei 8-Bit *Immediates* deklariert. Anschließend werden die Speicheradressen des Eingabe- und des Ausgabe Bildes geladen. Dementsprechend müssen zwei Pixel pro Zeile mehr geladen werden als dann nach der Bearbeitung gespeichert werden. Die entsprechende Realisierung ist in LISA-Code A.2 gezeigt.

Es werden Arrays für die Zwischenergebnisse des Sobelfilters in S_X und in S_Y Richtung, für den Absolutwert des Grauwertgradienten M und die Richtung φ des Gradienten sowie für die als Endergebnis berechneten Pixel benötigt. Zunächst muss das Ausgangsbild geladen und der *Sobel* Filter darauf angewendet werden. Es wird sowohl das absolute Ergebnis als auch die Summe der Absolutwerte und des Grauwert Gradienten gespeichert. Mit dieser Information kann die Richtung des Gradienten berechnet und gespeichert werden. Anschließend wird die *Non-maximum Suppression* genutzt um die detektierten Kanten zu einer Linienstärke von eins zu reduzieren. Abschließend werden die berechneten Pixel gespeichert und die Register werden inkrementiert, so dass die nächsten Pixel berechnet werden können.

```
1  OPERATION canny_filter IN pipe.EX{
2   DECLARE { GROUP src,dest = {regH}; GROUP highT,lowT={imm8}; }
3   CODING {0b01000 src dest highT lowT}
4   SYNTAX {"CAN " dest "," src "," lowT "," highT}
5   BEHAVIOR{
6    /*calculate SobelX and SobelY*/
7    for(i=0;i<21;i++){
8     if(i==7)j+=2;
9     if(i==14)j+=2;
10    sx[i]=-(int)dat[i+j]+(int)dat[i+2+j]
11      -((int)dat[i+j+9]<<1)+((int)dat[i+j+11]<<1)
12      -(int)dat[i+j+18]+(int)dat[i+j+20];
13    sx[i]=sx[i]>>2;//max 4*255
14    if (sx[i]<0)asx[i]=-sx[i]; else asx[i]=sx[i];
15
16    sy[i]=-(int)dat[i+j]-((int)dat[i+j+1]<<1)-(int)dat[i+j+2]
17       +(int)dat[i+j+18]+((int)dat[i+j+19]<<1)+(int)dat[i+j+20];
18    sy[i]=sy[i]>>2;//max 4*255
19    if (sy[i]<0)asy[i]=-sy[i]; else asy[i]=sy[i];
20    s[i]=(asx[i]+asy[i])>>1;//max 2*255}
21   /*calculate cases*/
22   for(i=0;i<5;i++){
23    if (s[i+8]>=lowT){
24     if (asx[i+8] > (asy[i+8]<<1)) c[i] = 0;
25      else if ((asx[i+8]<<1) > asy[i+8]) c[i] =
26       = sx[i+8] * sy[i+8]>=0 ? 2 : 3;
27       else c[i] = 1;};}
28   img[0]=s[8]; img[1]=s[9]; img[2]=s[10]; img[3]=s[11]; img[4]=s[12];
29   /*Non-Maximum Suppression*/
30   for(i=0;i<5;i++){
31    if (img[i] >= lowT){
32     if(c[i]==0) if (s[i+9]<=img[i] && s[i+7]<img[i])
33      img[i] = img[i]>=highT ? 254 : 1;
34     if(c[i]==1) if (s[i+15]<=img[i] && s[i+1]<img[i])
35      img[i] = img[i]>=highT ? 254 : 1;
36     if(c[i]==2) if (s[i+16]<=img[i] && s[i]<img[i])
37      img[i] = img[i]>=highT ? 254 : 1;
38     if(c[i]==3) if (s[i+14]<=img[i] && s[i+2]<img[i])
39      img[i] = img[i]>=highT ? 254 : 1;};}
40   for(i=0;i<5;i++){
41    if (img[i]==254)img[i]=255;else img[i]=0;}
42   SRAM[out_img]=img[0]; SRAM[out_img+1]=img[1]; SRAM[out_img+2]=img[2];
43   SRAM[out_img+3]=img[3]; SRAM[out_img+4]=img[4]; R[src]+=5; R[dest]+=5;}}
```

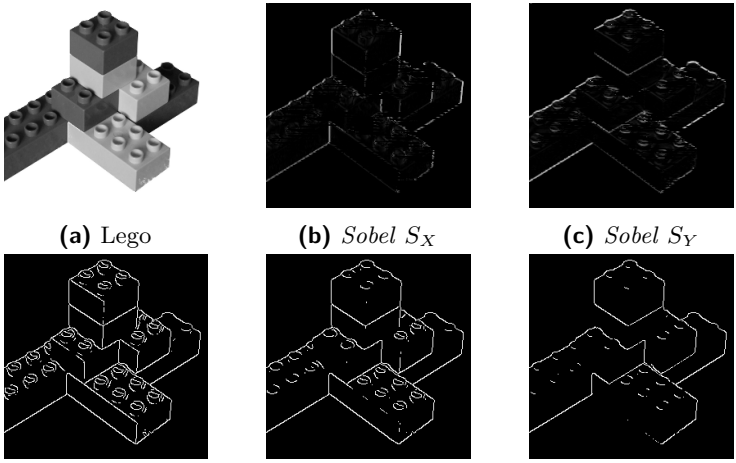
LISA-Code A.2: LISA Realisierung des Cannyfilter

Evaluation der applikationsspezifischen Erweiterungen Der Instruktionssatz von LImbiC wird mit dem *ARM C Compiler* (ARMcc) getestet, indem Assembler und C-Programme für die *ARMv6-M* Architektur übersetzt werden. Diese werden anschließend in der aus der LISA Beschreibung erzeugten Werkzeugkette, insbesondere im *Processor Designer*, ausgeführt. Somit ist die Syntax, die Kodierung und die Zyklenzeit der Instruktionen, Funktionen und Programme verifiziert.

Anhand von Abbildung A.7 lässt sich das Vorgehen und die Funktionsweise nachvollziehen. Abbildung A.7a zeigt das Eingabe Bild. Das Ausgabe Bild nach einer Faltung ist mit dem Sobelfilter in x-Richtung bzw. y-Richtung in Abbildung A.7b bzw. Abbildung A.7c zu sehen. Die im Eingabe Bild erkannten Kanten werden durch helle Pixel angezeigt. Die Helligkeit des Pixels ist durch die Wahrscheinlichkeit, dass es Teil einer Kante ist, bestimmt. Die Kanten der Legosteine sind jeweils gut zu erkennen. Jedoch sind die Kanten aufgrund des einfachen Vorgehens des Algorithmus verschwommen.

Der Cannyfilter extrahiert die Kanten und ergibt eine klare Repräsentation der Kanten der Legosteine. In der unteren Reihe von Abbildung A.7 sind die Ergebnisse für Cannyfilter mit unterschiedlichen Schwellenwerten gezeigt. Die Parameter für die Schwellenwerte werden variiert um den Einfluss der Schwellenwerte auf die Qualität des Ausgabebildes zu bestimmen. Das Eingabebild ist das Ergebnis der Faltung, das in Abbildung A.7b bzw. Abbildung A.7c gezeigt ist. In Abbildung A.7d werden die Schwellwerte zu $T_1 = 10$ und $T_2 = 20$ gesetzt. In Abbildung A.7e bzw. Abbildung A.7f werden die Schwellwerte zu $T_1 = 20$ und $T_2 = 30$ respektive $T_1 = 20$ und $T_2 = 50$ gesetzt.

Wie zu erwarten ist, reduziert die Erhöhung des Schwellwertes T_2 die Anzahl der erkannten Kanten. Die Ausgabebilder zeigen somit mehr Details bei der Wahl eines geringeren Schwellenwertes T_2. Abschließend lässt sich sagen, dass die zusätzlichen Instruktionen zusammen mit der *ARMv6-M* ISA funktionieren und es somit möglich ist mit den ausgewählten Algorithmen Kanten zu erkennen. Ein Bewertung bezüglich des Realisierungsaufwands der zusätzlichen Instruktionen erfolgt in Unterabschnitt A.1.3.



(a) Lego **(b)** *Sobel* S_X **(c)** *Sobel* S_Y

(d) Cannyfilter mit $T_1 = 10$ und $T_2 = 20$ **(e)** Cannyfilter mit $T_1 = 20$ und $T_2 = 30$ **(f)** Cannyfilter mit $T_1 = 20$ und $T_2 = 50$

Abbildung A.7.: Lego Eingabe Bild und die Ausgabe Bilder nach einer Faltung mit S_X bzw. S_Y und einem Cannyfilter mit unterschiedlichen Schwellwerten.

A.1.2. *a*-Sim: Die adaptiven Simulationen

Frühe Evaluation mit akkuraten *High Level* Simulationsmodellen, wie beispielsweise LImbiC, sind der Schlüssel zu einem effizienten Entwicklungsprozess für komplexe Mehr-/Vielkern Systeme. Um *Time to Market* zu verkürzen, werden Hardware und Software eng verzahnt parallel zueinander entwickelt. Moderne *Hardware/Software Co-Design* ermöglicht es den Entwicklungsprozess mit dem Evaluationsprozess zu synchronisieren.

A.1.2.1. Design der adaptiven Simulationen

Insbesondere bei adaptiven Prozessoren ist eine frühzeitige Evaluation der zusätzlichen Fähigkeiten relevant. Im Kontext von heterogenen Plattformen ist es wichtig Wege zu finden, um unterschiedliche Ausprägungen der Architektur realisieren und parallel auch simulieren zu können, um ihre

Abhängigkeiten aufzuzeigen, so dass die Software sobald wie möglich ausgeführt werden kann. Dieser Abschnitt präsentiert einen Absatz um die folgenden Ziele zu erreichen:

- Die frühzeitige Evaluation der adaptiven Hardware, so dass anschließend nur die erfolgversprechendsten Erweiterungen realisiert werden.
- Bereitstellung von Funktionalitäten, damit Anwendungsentwickler ihre Programme frühzeitig realitätsnah für die applikationsspezifischen Erweiterungen optimieren können.
- Ausnutzen von schnellen und akkuraten Simulationen, auch im Vergleich zu hardwarebasierten, wie beispielsweise *Field Programmable Gate Array* (FPGA), Entwicklungsplattformen. Insbesondere im Kontext eingebetteter Systeme ist eine schnelle Evaluation auf *High Performance Computing* (HPC) Systemen möglich.

ADLs und System Sprachen ermöglichen Simulationsgeschwindigkeiten, die für die Bewertung komplexer Applikationen notwendig sind. Außerdem vereinfachen sie die Entwicklung und die Verfügbarkeit von Hardwaremodellen für die entsprechenden Plattformen. Um die Kosten und auch den zeitlichen Aufwand in Grenzen zu halten, werden die existierenden Werkzeuge basierend auf der ADL LISA wieder verwendet und für den gewünschten Zweck erweitert. Dies erhöht die Nutzbarkeit sowie auch die Wiederverwertbarkeit des Ansatzes. Die Möglichkeit einzelne Komponenten eines Systems auszutauschen, die Plattform jedoch intakt zu belassen ist ein wichtiger Schritt für die Evaluation von adaptiven Prozessoren mit Hilfe von SystemC *Transaction Level Modeling* (TLM).

Das Konzept ist wie folgt gegliedert: Zunächst wird der Basisfluss vorgestellt, der die Schritte für einen schnellen Entwicklungszyklus von frühen funktionalen Modellen von applikationsspezifischen Prozessoren vorstellt. Die weiterführenden Schritte befassen sich mit adaptiven Simulationen und Abwägungen zwischen Simulationsgeschwindigkeit und Genauigkeit der Simulation. Das Ergebnis der Kombination des Basisflusses und der adaptiven Simulationen ist eine komplette integrierte Werkzeugkette, wie sie in Abbildung A.8 gezeigt ist.

Basisfluss Der Basisfluss wird begonnen, in dem man eine LISA Beschreibung des a-Core erstellt. Dies kann erreicht werden, in dem man

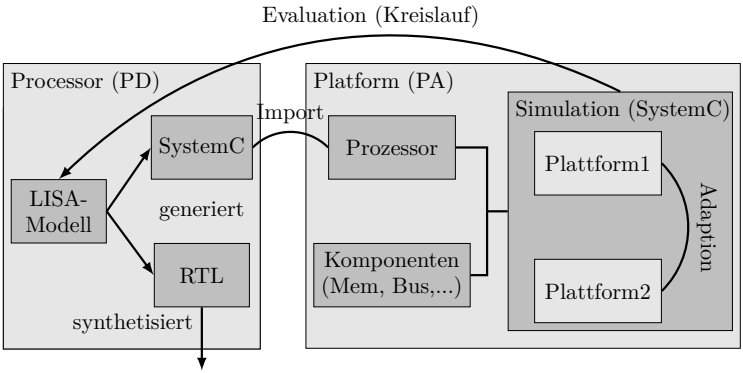


Abbildung A.8.: Integrierter Werkzeugfluss bestehend aus Basisfluss, in dem ein SystemC-Modell mit Hilfe von *Processor Designer* aus einem LISA-Modell generiert wird. Dieses Modell wird anschließend in *Platform Architect* importiert, wo vorhandene Komponenten aus der Bibliothek verwendet werden, um eine Simulationsplattform aufzubauen. Diese Plattformen werden um die Fähigkeit der adaptiven Simulationen (a-Sims) erweitert, deren Ergebnisse anschließend wieder in das LISA einfließen und den Prozessor entsprechend der Anforderungen iterativ verbessert.

ein neues Modell entwickelt, wie beispielsweise LImbiC in diesem Kapitel. Der Prozessor kann dann entwickelt, simuliert und verifiziert werden, in dem eine Anwendung ausgeführt wird. Sobald das LISA-Modell erstellt ist, kann ein SystemC-Modell automatisch aus der Beschreibung generiert werden.

Diese SystemC Beschreibung kann dann mit anderen SystemC Beschreibungen verknüpft werden, so dass eine komplette Plattform erstellt werden kann. Viele Basiskomponenten werden sich nicht weiter verändern, so dass vordefinierte und bereits über längere Zeit verifizierte Modelle in die Plattform eingebaut werden können. Zu SystemC inkompatible Verbindungen benötigen Adapter, die Teil der jeweiligen Modelle sind. Da die komplette Plattform somit aus SystemC-Modellen und damit C/C++ Code besteht, kann die Plattform mit einem C Compiler in eine ausführbare Datei übersetzt werden.

Adaptive Simulationen Der Basisfluss, der im vorigen Unterabschnitt beschrieben ist, zeigt einen Weg eine neu entwickelte Plattform zu erstellen und zu simulieren. Somit können jetzt die Vorteile von SystemC ausgenutzt und die Modellierung und die Simulation der Architektur erweitert werden. Die Möglichkeit unterschiedliche Abstraktionsebenen zu simulieren und die schnelle Erstellung und Veränderung von neuen Modellen erlaubt es die veränderliche Natur von adaptiven Prozessoren in die Simulation zu transferieren.

Es wird ein integrierter Werkzeugfluss erstellt, der es erlaubt Plattformen mit zur Laufzeit veränderlichen Parametern oder sogar unterschiedlichen Anforderungen zu erstellen und zu evaluieren. Schlussendlich kann der Softwareentwickler einen Funktionsaufruf in die Applikation einfügen, die das simulierte Hardwaremodell verändert, wenn diese Funktion aufgerufen wird. Dies ermöglicht es dem Softwareentwickler den Prozessor mit dem Ziel der effizienten Ausführung der Applikation zu verändern. Der integrierte Werkzeugfluss besteht aus zwei Teilen: Zum einen der Entwicklerseite und zum anderen der Anwenderseite. Ein Nutzer erhält für die Evaluation Zugriff auf alle vorher integrierten Plattformen und ihre Parameter; während der Entwickler in der Lage ist diese Plattformen zu erweitern, indem vorhandene Modelle modifiziert oder neue Modelle hinzugefügt werden.

Die Adaption kann dabei sowohl hinsichtlich der Granularität und dem Ausmaß der Veränderung variieren. Die Granularität bezieht sich hierbei auf den Maßstab der Adaptionen, die den Applikationscode betreffen. Im Grobgranulieren kann ein Veränderung aus ganzen Funktionen bestehen, die große Teile der Programmlogik implementieren, die manuell für bestimmte Architekturen oder bestimmte Eigenschaften der Architektur zulassen während feingranulare Adaptionen auf Schleifen- oder Methodenebene erlaubt. Das Ausmaß beschreibt die Ebene der Veränderung in einer Plattform. Dies reicht von kleinen Adaptionen, wie die Veränderung der Cache Verdrängungsstrategie, bis hin zu großen Veränderungen, durch die komplett andere Architekturen zum Einsatz kommen können. Um ein Hardwarearchitekturmodell durch eine Sprache auf Systemebene zu verändern, kann ein umfangreiches Modell der Hardware bereits integrierten Lösungen erstellt werden. Dieses Modell kann das mittels Konfigurationsdateien oder Datenstrukturen dynamisch entscheiden welche Version genutzt werden soll. Alternativ kann auch eine neue Plattform erstellt werden, in dem der LISA Code erneut übersetzt wird. Um den aktuellen Zustand der Simula-

tion wieder herstellen zu können, muss dieser aus dem aktuellen Modell extrahiert, gespeichert und evtl. auch übertragen bzw. auf das neue Modell übersetzt werden. Bereits eingebaute Veränderungen sind vorteilhaft für kleine Veränderungen. Hingegen ermöglicht das erneute Übersetzen von neuen Simulationen einen tiefergehenden modularen Ansatz. Im neuen integrierten Werkzeugfluss für die *a*-Sim kann eine Kombination aus beiden Varianten genutzt werden. Eingebaute Veränderungen sind in einigen Modellen bereits vorhanden, speziell um bestimmte Konfigurationen der realen Hardware wieder zu spiegeln, die durch spezielle Instruktionen, wie sie im vorigen Kapitel vorgestellt wurden, ausgelöst werden können.

Automation Ein essentieller Teil der *a*-Sims ist deren Automation. Dies ist notwendig um die Komplexität der Realisierung vor dem Anwender zu verbergen, wenn dieser beispielsweise einen Algorithmus und dessen Optimierungen auf der Zielhardware evaluiert. Zusätzlich ermöglicht die Automatisierung weniger Interaktion mit dem Nutzer und dadurch schnellere, komplexere Simulationen, die im Hintergrund beispielsweise auf HPC Maschinen abgearbeitet werden können.

Das Werkzeug für die *a*-Sims ist dafür zuständig die Simulationen entsprechend den Anforderungen automatisch zu verändern und jeweils entsprechend zu initialisieren. Die Anfrage für eine Adaption erfolgt durch eine spezielle Instruktion, die mit einem Funktionsaufruf in der Anwendung aufgerufen wird. Dieser Aufruf wird vom Werkzeug ausgewertet und in einen *Breakpoint* für die Simulation übersetzt. Hierdurch wird die Simulation zum gewünschten Zeitpunkt angehalten und entsprechend den Anforderungen angepasst. Um dem Anwender eine vielfältige Auswahl an Komponenten und Modellen für die Adaption bereitzuhalten wird eine Auswahl an Architekturen und übersetzten Modellen aufgebaut. Es ist anschließend möglich basierend auf der vorgestellten Werkzeugkette neue Komponenten zu integrieren oder bestehende Komponenten zu verändern bzw. zu erweitern. Der herausfordernde Teil der Erweiterungen ist die Übersetzung des jeweiligen Zustandes, so dass die Komponenten transparent in den adaptiven Simulationen eingesetzt werden können.

Simulationszeit gegenüber Abstraktionsebenen Die gleiche Methode wie zur Adaptierung der Simulationen zu unterschiedlichen Hardware-

modellen kann auch bei der Adaption des Abstraktionsebenen eingesetzt werden. Hierbei wird die Abstraktion eines Modells verändert anstatt die Eigenschaften des Modells selbst zu verändern. Somit ist es möglich einzelne Teile der Plattform, die für die aktuelle Evaluation nicht im Detail interessant oder zeitkritisch sind, auf einer höheren Abstraktionsebene schneller zu simulieren ohne dabei Genauigkeit für die relevanten Modelle zu verlieren. Des Weiteren ist es möglich die Simulation auch in der Zeitdomäne zu unterteilen. So kann eine Applikation bis zu einem Zeitpunkt sehr schnell simuliert werden, wie beispielsweise ein Bootprozess oder bereits validierte Teile einer Anwendung, die keiner genaueren Betrachtung bedürfen. Anschließend kann die Simulation in einen akkuraten Modus geschaltet werden, damit eine bestimmte Komponente der Hardware zu einem bestimmten Zeitpunkt der Anwendung im Detail evaluiert werden kann.

A.1.2.2. Realisierung der adaptiven Simulationen

Die Realisierung basiert auf der ADL LISA, der daraus generierten SystemC Beschreibung und entsprechenden Werkzeugen der Firma Synopsys *Processor Designer* (PD) und *Platform Architect* (PA). Diese Werkzeuge greifen ineinander, wodurch die Integration von LISA-Modellen in die Plattformen vereinfacht wird, da eine Vielzahl an Standard Komponenten zur Verfügung steht, die für eine größere Plattform benötigt werden. Weiterführende technische Informationen sind in der Abschlussarbeit von Leonard Masing [Mas13] zu finden.

Adaptierungen Da alle SystemC-Modelle auf C/C++ Code basieren ist es möglich auf deren Struktur zuzugreifen und diese zu verändern. Um dies zu erreichen wird ein spezielles *Application Programming Interface* (API) definiert und in die Modelle integriert. Diese API ermöglicht es den Status eines Modells zur Laufzeit zu verändern, indem das Modell an einen *Debugger* angeschlossen wird. Dieser *Debugger* kann manuell oder durch einen Algorithmus aufgerufen werden, um die Simulation anzuhalten und den Status der Eigenschaften des Modells abzufragen. Durch den Zugriff auf Speicher und Register der Plattform wird der aktuelle Zustand der Simulation in eine *Intermediate representation* (IR)

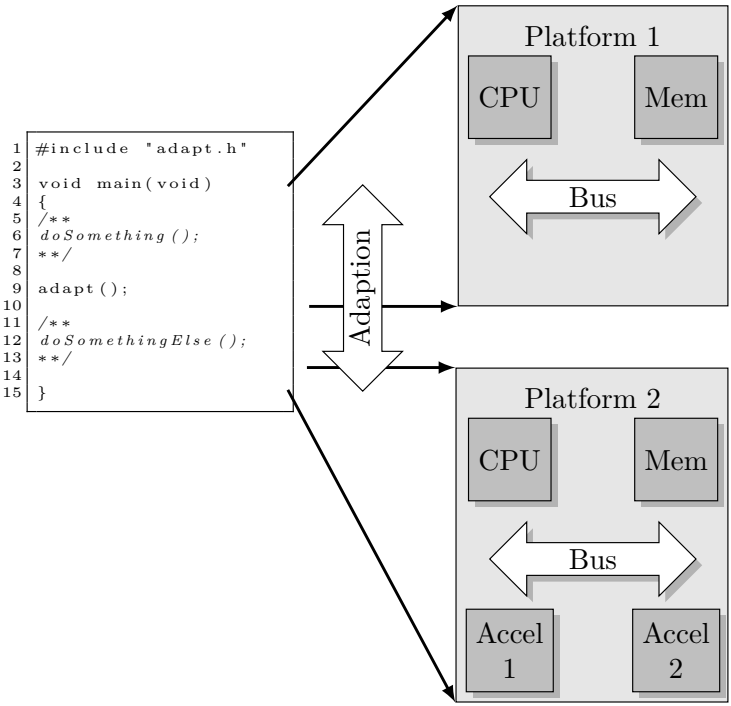


Abbildung A.9.: Die Adaptierung einer Hardwareplattform, die durch eine Funktion (*adapt();*) in der Software zur Laufzeit der *a*-Sims aufgerufen wird.

gespeichert. Die gespeicherte IR wird anschließend genutzt, um den Zustand in der neuen Simulation wieder herzustellen. Falls sich die Plattform während der Simulation geändert hat, wenn beispielsweise eine Veränderung angestoßen wurde, wird eine Übersetzung der IR an die Parameter der neuen Plattform notwendig.

Die Möglichkeit zur Veränderung der Hardware - nur durch das Einfügen eines Aufrufs in die Anwendung - unterstreicht die Trennung zwischen einer Entwickler- und einer Anwendersicht bei diesem Ansatz. Dieses Merkmal und die generelle Idee des Ansatzes der Adaptierungen ist in Abbildung A.9 gezeigt. Eine Anwendung wird auf dem Modell der Hardwareplattform

ausgeführt. Der Funktionsaufruf *adapt();* verändert die Architektur der Plattform. Dies wird auf realer Hardware durch das einfügen spezieller Instruktionen, wie im vorigen Kapitel vorgestellt, erreicht.

Das Werkzeug, das diese Funktionalitäten ausführt, wird zur Designzeit mit der Simulation verlinkt. Zu Beginn wird der Anwendungscode, der die Adaption ausführt, im Speicher abgelegt. Anschließend wird die Simulation ausgeführt. Das Werkzeug ist somit auch für die Unterbrechung der Simulation und die entsprechenden *SAVE* und *RESTORE* Operationen verantwortlich.

A.1.2.3. Evaluation der adaptiven Simulationen

Um die Anwendbarkeit des vorgestellten Ansatzes zu zeigen, wurden Experimente mit *a*-Sim durchgeführt. Zur Durchführung der Experimente wird das LISA Prozessormodell des LImbiC verwendet.

Hierzu wird mit dem Werkzeug *Processor Designer* SystemC der kompatible Plattformcode aus der LISA Beschreibung des Prozessors generiert. Das generierte Modell wird anschließend der zu evaluierenden Plattform im *Platform Architect* hinzugefügt.

Als Experiment werden die adaptiven Simulationen evaluiert, in dem das Busmodell mit Hilfe des vorgestellten Ansatzes adaptiert wird. Die Plattform ist wie folgt aufgebaut: Sie besteht aus zwei LImbiC Prozessoren, die über einen Bus mit einem Speicher und weiteren Peripheriekomponenten verbunden sind. Dieser Bus kann auf zwei unterschiedliche Arten modelliert werden: Entweder als generisches TLM-basiertes *Loosely Timed* Modell oder als akkurates *Advanced High-performance Bus* (AHB)-Modell. Es wird wieder der *Quicksort* Algorithmus verwendet. Der Algorithmus wird so angepasst, dass die Adaption an einem bestimmten Punkt aufgerufen werden kann. Der Aufruf kann an einer beliebigen Stelle während der Ausführung des Sortierens erfolgen. Somit ist es dem Entwickler möglich eine Adaption feingranular, auch zu verschiedenen Zeitpunkten während der Evaluation durchzuführen. In diesem Beispiel wird die Applikation zunächst mit dem TLM basierten Bus ausgeführt. Anschließend wird die Ausführung in der Mitte unterbrochen und es wird auf den akkuraten AHB Bus gewechselt.

Busmodell	Einkern	Mehrkern
TLM LT	0,513 s	0,511 s
AHB	0,513 s	0,711 s

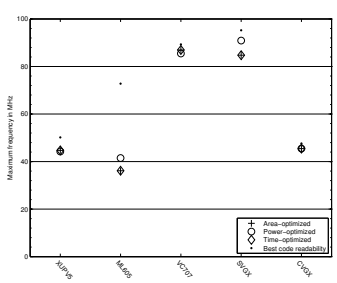
Tabelle A.2.: Konkurrenzsituation auf einem Bus im Mehrkernsystem, die nur mit einem akkuraten AHB Simulationsmodell zu sehen ist.

Die Simulationsergebnisse sind in Tabelle A.2 gegeben. Die Tabelle zeigt die Zeiten für eine Ausführung des *Quicksort* Algorithmus auf der Mehrkernplattform und auf der Einkernplattform im Vergleich. Aus diesen Ergebnissen wird klar ersichtlich, dass die Ausführungszeit deutlich ansteigt, sobald der Bus als akkurates AHB-Modell simuliert wird. Dies kommt daher, dass der TLM-Bus nur eine feste Verzögerung für jeden Zugriff errechnet, wobei der AHB-Bus das komplette Buszugriffsmodell realisiert. Die Zeiten beziehen sich insbesondere auf die Ausführung nach der Adaption, so dass der zeitliche Mehraufwand für die Simulation klar ersichtlich ist.

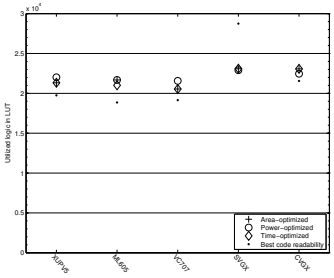
A.1.3. Evaluation des applikationsspezifischen Prozessormodells

In diesem Abschnitt wird die Evaluation des applikationsspezifischen Prozessors LImbiC vorgestellt. Hierzu wird der HDL Code für die verschiedenen Versionen des LImbiC Prozessors generiert, wie sie in Unterabschnitt A.1.1 vorgestellt wurden. Die drei unterschiedlichen Varianten des LImbiC werden anschließend für unterschiedliche Entwicklungsboards der Firmen Altera und Xilinx synthetisiert. Hierdurch können die einzelnen applikations-spezifischen Eigenschaften der Prozessoren bewertet werden. Zusätzlich werden die Ergebnisse der Synthese basierend auf der automatischen Generierung des HDL Code auf Basis des LISA-Modells verglichen, um eine Einordnung zu Codequalität und Nutzbarkeit geben zu können.

Die drei Varianten des LImbiC werden mit Synplify Pro der Firma Synopsys synthetisiert. Das *Place & Route* (PAR) wird anschließend mit dem jeweiligen Werkzeug des Herstellers für die Plattform durchgeführt. Die synthetisierbare *Verilog* Beschreibung wird aus dem LISA-Modell generiert.



(a) Maximale Frequenz (MHz)



(b) Logik Nutzung (*Lookup Table* (LUT))

Abbildung A.10.: Maximale Frequenz (links) und Logik Nutzung (rechts) des Basis-LImbiC synthetisiert für Flächen-, Zeit-, Leistungsoptimierung und Beste Code Lesbarkeit.

Somit kann eine *Design Space Exploration* (DSE) durchgeführt werden, um die verschiedenen Versionen des generierten HDL Codes bewerten zu können. Hierdurch können verschiedene Varianten des LImbiC anhand ihrer Effizienz, also der Performanz und dem dazu benötigten Flächenverbrauch, bewertet werden.

A.1.3.1. Basis-LImbiC & Vergleich der Einstellungen

In diesem Experiment wird der Basis-LImbiC für fünf verschiedene FPGA Plattformen synthetisiert.

- Altera *Stratix V* (5SGXEA7K2F40C2N) (SVGX)
- Altera *Cyclone V* (5CGXFC7D6F31C7NES) (CVGX)
- Xilinx *Virtex-5* (XC5VLX110T) (XUPV5)
- Xilinx *Virtex-6* (XC6VLX240T) (ML605)
- Xilinx *Virtex-7* (XC7VX485T) (VC707)

Hierbei werden die unterschiedlichen Einstellungen des *Processor Generator* (PG) exploriert. Die Standardeinstellung ist 'Beste Code Lesbarkeit'. Weitere Optionen sind: flächenoptimiert, zeitoptimiert, leistungsoptimiert,

die den jeweils optimalen HDL Code für das jeweilige Optimierungsziel generieren sollen.

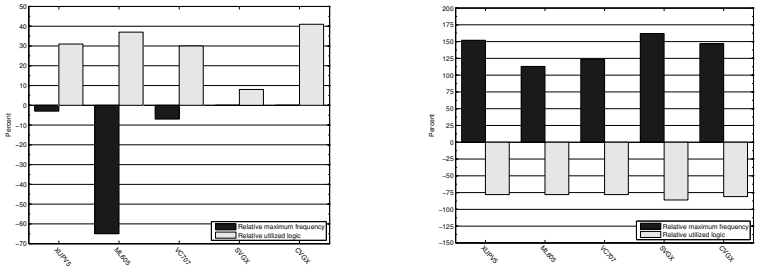
Abbildung A.10 zeigt die resultierenden Implementierungen mit den unterschiedlichen Eigenschaften. Die maximale Frequenz (Abbildung A.10a) und die benötigte Anzahl Logikzellen (Abbildung A.10b) werden für die vier Explorationseinstellungen und die fünf FPGAs aufgetragen. Insgesamt weichen die maximalen Frequenzen abhängig von den unterschiedlichen Einstellungen kaum voneinander ab. Einzige Ausnahme ist hier der Xilinx *Virtex-6* (XC6VLX240T) (ML605). Der ML605 ist der neueste und größte FPGA, der die Xilinx Instruktionssatzerweiterungen (ISE) für *Place & Route* (PAR) nutzt. Es kann angenommen werden, dass die ISE dabei Probleme hat die Logik gleichmäßig im relativ großen ML605 FPGA zu platzieren, so dass die maximale Frequenz hier einbricht. Im Vergleich dazu hat der VC707 deutlich weniger Varianz, da hier schon das neue Werkzeug *Vivado* zum Einsatz kommt.

Die Einstellungen für die Flächen- und Zeitoptimierung scheinen immer die gleichen Ergebnisse zu liefern. Einzig die leistungsoptimierte Einstellung variiert indem eine leicht bessere Effizienz auf *Xilinx* FPGAs und eine leichte verschlechterte Effizienz bei *Altera* FPGAs zu beobachten ist. Der Einfluss auf den Leistungsverbrauch im Betrieb der Designs wird an dieser Stelle nicht weiter untersucht. Die Einstellung 'Beste Code Lesbarkeit' erreicht die beste Performanz pro Fläche. Einzig auf dem *Altera* SVGX werden bei dieser Einstellung deutlich mehr LUTs benötigt, während der Logikverbrauch bei allen anderen FPGAs um ca. 10 % sinkt.

In Bezug auf die genutzten Logik Elemente ist die leistungsoptimierte Einstellung leicht besser als die flächen- und zeitoptimierte Einstellung. Nur die Einstellung für 'Beste Code Lesbarkeit' schafft ein besseres Ergebnis. Daher wird die Einstellung 'Beste Code Lesbarkeit' für alle weiteren Experimente zur Evaluation der applikationsspezifischen Erweiterungen gewählt.

A.1.3.2. LImbiC - Basis-LImbiC mit Erweiterungen

LImbiC besteht aus dem Basis-LImbiC und den applikationsspezifischen Erweiterungen für die Faltung und den Cannyfilter. Abbildung A.11a zeigt



(a) LImbiC **(b)** Small-LImbiC

Abbildung A.11.: Maximale Frequenz (MHz) and Logik Nutzung (LUT) des LImbiC relativ zum Basis-LImbiC (links) and des Small-LImbiC relativ zum Basis-LImbiC (rechts).

den relativen Logik Verbrauch und die relative maximale Frequenz im Vergleich zum Basis-LImbiC.

Für die meisten FPGAs bedeutet dies auf den ersten Blick eine Verschlechterung der Effizienz, da die Instruktionen bis zu 40 % mehr Fläche belegen. Es ist jedoch wichtig, dass diese zusätzliche Fläche für neue Instruktionen genutzt wird, die in der Lage sind die Bildverarbeitungsalgorithmen deutlich effizienter auszuführen als die einzelnen Instruktionen, die bisher zur Verfügung stehen, da die Anzahl der benötigten Zyklen signifikant reduziert wird. Ein 254×254 Pixel Bild hat 64.516 Pixel, die in der Schleife von 35 Instruktionen berechnet werden müssen.

Dies resultiert in 2,26 Millionen Instruktionen, die für die Berechnung der Faltung benötigt werden. Im Gegensatz dazu benötigt die applikationsspezifische Lösung nur fünf Instruktionen pro Schleife um vier Pixel zu berechnen, was insgesamt 81.650 Instruktionen ergibt. Dies ist eine Reduktion von ca. 98 % der benötigten Zyklen. Dies reduziert dementsprechend auch die Ausführungszeit, wobei der Performanzzuwachs hier durch den Flaschenhals der Speicherhierarchie limitiert ist, da die Load/Store Befehle weiterhin sequentiell abgearbeitet werden müssen. Dies ergibt dementsprechend trotz des Ressourcenmehraufwands insgesamt eine deutliche Effizienzsteigerung.

A.1.3.3. Small-LImbiC - Der optimierte Bildverarbeitungsprozessor

Das Design des LImbiC zeigt das Potential der Verwendung von zusätzlichen Instruktionen in der bestehenden ISA. Auf der anderen Seite werden bestehende Instruktionen damit obsolet, so lange die angedachte Zielanwendung ausgeführt werden soll. Somit kann die ISA in diesem Fall deutlich auf eine minimale Anzahl von Instruktionen, die für die Bildverarbeitung benötigt werden, reduziert werden. Hieraus resultiert der Small-LImbiC, dessen ISA nur aus den applikationsspezifischen Instruktionen *CGA, CSX, CSY, CAN* und Verzweigungsinstruktionen, sowie auch den arithmetischen Instruktionen *ADD, LSL und CMP* besteht.

Abbildung A.11b zeigt die Resultate für die Fläche und die maximale Frequenz relativ zum Basis-LImbiC. Die maximal erreichbare Frequenz wird mehr als verdoppelt und die Fläche wird um mehr als 75 % reduziert, da hauptsächlich der Realisierungsaufwand für die applikationsspezifischen Instruktionen bleibt. Die Effizienz steigt in dieser Konsequenz um den Faktor zehn im Vergleich zum Basis-LImbiC. Zusätzlich wird auch die Ausführungszeit deutlich verkürzt.

A.1.3.4. Prototyping von LImbiC

Die ADL LISA enthält einige Sprachkonstrukte, die speziell für die Entwicklung von Modellen gedacht ist, aus denen der HDL Code generiert werden soll. Für Modelle, die nur für die Simulation und die Werkzeugkette gedacht sind, ist es nicht nötig das Design mit diesen zusätzlichen Konstrukten zu modellieren. Wenn das Design jedoch prototypisch realisiert werden soll, gibt es einige Richtlinien und Eigenschaften, die das Design deutlich verbessern. Im Speziellen sei darauf verwiesen, dass das Teilen der Ressourcen für die optimale Realisierung genutzt werden sollte. Ansonsten kann es passieren, dass bei der Generierung Ressourcen, wie beispielsweise Teile der ALU, doppelt angelegt werden, die unnötig Fläche belegen. Im Fall der Zugriffe auf das *Register-File* können so unnötige Multiplexer vermieden werden [157]. Dies kann auch dazu führen, dass mehr Speicher mit mehr parallelen Anschlüssen als eigentlich notwendig vorgesehen wird, da nicht erkannt wird, dass die Zugriffe mehrerer Instruktionen über einen Lese-/Schreibanschluss der *Memory* Stufe erfolgen.

Zunächst wurde LImbiC als Simulationsmodell entwickelt, um *a*-Sim zu evaluieren. Hierdurch kann ein Vergleich zwischen der Realisierung des Simulationsmodells und dem für die Synthese optimierten Modells des Basis-LImbiC erfolgen. Das für die Synthese optimierte Modell wurde erstellt, um mehrere Zugriffe auf externe Speicher verwalten zu können, wie dies von den zusätzlich hinzugefügten Instruktionen genutzt wird.

Die für die Synthese optimierte Variante des Basis-LImbiC wurde entlang der Vorgaben der Firma Synopsys neu erstellt [157]. Alle Zugriffe auf Register wurden in eine Operation verschoben, um das Teilen der Ressourcen zwischen allen Instruktionen mit einem optimierten *Forwarding* zu realisieren. ALU Operationen wurden soweit wie möglich gruppiert um die Dekodierlogik in der *Decode* Stufe zu vereinfachen. Zusätzlich wurde eine zweite Pipeline nur für Speicherzugriffe hinzugefügt, um mehrere Zugriffe über einen Speicheranschluss zu verwalten und auch der Latenz von Speicher und Bus Rechnung zu tragen.

Die Simulations- als auch die Synthesevariante des Basis-LImbiC wurden für den Xilinx XUPV5 synthetisiert. Beide Modelle realisieren also die gleiche ISA und werden ohne *Debug Unit* synthetisiert. Zum Vergleich werden die Zahlen des ARM *Cortex-M1* von der Hersteller Webseite übernommen [10]. Tabelle A.3 zeigt die Ergebnisse der Synthesen. Für die bessere Vergleichbarkeit der Ergebnisse wurde nur der Prozessor selbst ohne externe Busse oder Speicher synthetisiert.

Wie aus den Ergebnissen zu erkennen ist, wurde der Flächenbedarf durch die Optimierungen für die Synthese um den Faktor 2 verringert. Die Anzahl der Register konnte dementsprechend etwas erhöht werden, um die Hierarchien, insbesondere bei den Speicherzugriffen, besser abbilden zu können. Alle Anpassungen haben jedoch nur einen minimalen Einfluss auf die maximale Frequenz.

Im Vergleich zum ARM *Cortex-M1* haben die aus der LISA Beschreibung generierten Modelle einen höheren Flächenverbrauch und erreichen die 100 MHz nicht. Es ist jedoch zu bedenken, dass es sich beim ARM *Cortex-M1* im Gegensatz zum LISA-Modell um eine stark optimierte *Verilog*-Realisierung handelt, welches die Werkzeuge sehr effizient auf FPGAs abbilden können. Somit ist das für die Synthese optimierte Modell des Basis-LImbiC nur 10 % größer als das ARM Design.

	Basis-LImbiC		ARM
	Sim.-opt.	Syn.-opt.	*Cortex-M1*
Maximale Frequenz (MHz)	87.5	87.3	100
Genutzte Register	612	968	—
Genutzte Logik (LUT)	6587	3211	2900

Tabelle A.3.: Vergleich von Basis-LImbiC und der für die Synthese optimierten Variante und dem ARM *Cortex-M1*.

A.1.3.5. Beschränkungen des *a*-Core Entwicklungsflusses

Während der Konzeption und der Umsetzung des adaptiven Prozessors (*a*-Cores) konnten einige bemerkenswerte Einschränkungen des LISA-basierten Entwicklungsflusses identifiziert werden. Einige der Herausforderungen hängen direkt mit der ADL LISA zusammen. So lassen sich keine Eingabeparameter für LISA Operationen definieren. Anstatt dessen müssen globale Variablen genutzt werden, um Daten an eine LISA Operation zu übergeben. Dies steigert die Codegröße und erschwert die Fehlerbeseitigung. Des Weiteren unterstützt LISA keine C-typischen Funktionen. Diese müssen stattdessen als C Prä-Prozessor Makros definiert werden. Dies erschwert die Lesbarkeit des Codes und die Fehlerbeseitigung.

Eine weitere Beschränkung ist, dass sich die Programmierstile zwischen dem *Processor Generator* und *Processor Designer* unterscheiden, was die Portierbarkeit der Modelle deutlich erschwert. Ein großer Teil des Codes musste neu geschrieben werden, um eine effiziente HDL Implementierung basierend auf der LISA-Modellierung zu erhalten, obwohl die Simulation bereits fehlerfrei funktionierte und kompatibel zum LISA-basierten Entwicklungsfluss war. Im Gegensatz zum *Processor Designer* unterstützt der *Processor Generator* keine arithmetischen Funktionen innerhalb der Kodierung, obwohl diese explizit in der LISA Referenz definiert sind [157].

Abschließend haben die Ergebnisse der Evaluation gezeigt, dass die zeit- bzw. flächenoptimierten Einstellungen des *Processor Generator*s nach der Erstellung der *Bitfiles* im gleichen Flächenverbrauch bzw. der gleichen maximalen Frequenz resultieren. Dadurch wird die *Design Space Exploration* stark einschränkt ohne den generierten Code bearbeiten zu müssen.

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

H_{Li} *Hit*

L_W Zeilenlänge in Wörtern

MM *Main Memory*

M_{Li} *Miss*

T_{ges} gesamte Speicherzugriffszeit

$\#a_{Offset}$ Anzahl der Offsetbits der Adresse

$\#a_{Satz}$ Anzahl der Satzbits der Adresse

μ**Arch** Mikroarchitektur

a_{Block} Blockadresse

a_{Satz} Satzadresse

a_{Tag} Tagadresse

a_{Zeile} Zeilenadresse

-O2 *Optimize even more*

-g *Debug Information*

#W Bytes pro Wort

#a Anzahl der Adressbits

#l Anzahl der Cachezeilen

#s Anzahl der Sätze

***a*-Core** adaptiven Prozessor

***a*-Core** adaptiver Prozessor

***a*-Sim** adaptive Simulationen

ACR *Adaptive Configuration Register*

ADL *Architecture Description Language*

ADPCM *Adaptive Differential Pulse Code Modulation*

AES *Advanced Encryption Standard*

AHB *Advanced High-performance Bus*

ALU *Arithmetic Logic Unit*

AMIDAR *Adaptive Microinstruction Driven Architecture*

AMS *Analog/Mixed Signal*

ANT *Always not Taken*

API *Application Programming Interface*

ARC *Applied Reconfigurable Computing*

ARCS *Architecture of Computing Systems*

ARM *Advanced RISC Machines Ltd.*

ARMcc *ARM C Compiler*

ASI *Adress Space Identifier*

ASIC *Application Specific Integrated Circuit*

ASIP *Application-Specific Instruction-Set Processor*

ASR *Ancillary State Register*

ASSR applikationsspezifischen Statusregister

AT *Always Taken*

AXI *Advanced eXtensible Interface Bus*

BA *Branch Always*

BHR *Branch History Register*

BHT *Branch History Table*

BN *Branch Never*

BP *Branch Prediction*

BRAM *BlockRam*

BS Berechnungen pro Speicherzugriff

BTAC *Branch Target Address Cache*

BTB *Branch Target Buffer*

C Cachegröße

C *Carry*

CAL *IEEE Computer Architecture Letters*

CAT *Cache Allocation Technology*

CCR *Cache Configuration Register*

CHIPit Protoypensystem mit sechs High-end *Virtex-5* FPGAs mit insgesamt mehr als 50.000 *Slices*

CISC *Complex Instruction Set Computer*

COS *Class of Service*

CP *Co-Processor*

CpI *Cycles per Instruction*

CPU *Central Processing Unit*

CVGX *Cyclone V* (5CGXFC7D6F31C7NES)

CWP *Current Window Pointer*

DATE *Design, Automation and Test in Europe*

DCTI *Delayed Control-Transfer Instructions*

DDR *Double Data Rate*

DE *Decode*

DFG Deutsche Forschungsgemeinschaft

DHLF *Dynamic History Length Fitting*

DM *Direct Mapped*

DSE *Design Space Exploration*

DSP *Digital Signal Processor*

DVS *Dynamic Voltage Scaling*

E/A Eingabe- und Ausgabesystem
EEMBC *Embedded Microprocessor Benchmark Consortium*
ET *Enable Traps*
ETIT Elektrotechnik und Informationstechnik
EX *Execute*

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg
FE *Instruction Fetch*
FFT *Fast Fourier Transform*
FIFO *First In – First Out*
FPGA *Field Programmable Gate Array*
FPU *Floating-Point Unit*
FTR Fehltrefferrate
FU *Functional Unit*

GCC *GNU Compiler Collection*
GP *General Purpose*
GPP *General-Purpose Processor*
GPP *General-Purpose* Prozessor
GPU *Graphics Processing Unit*

HDL *Hardware Description Language*
HPC *High Performance Computing*
HSC *Hardware/Software Co-Design*
Hz Hertz

ICC *Integer Condition Code*
IEEE *Institute of Electrical and Electronics Engineers*
IF *Instruction Fetch*
ILP *Instruction Level Parallelism*

InvasIC *Invasive Computing*

IO *In-Order*

IpC *Instructions per Cycle*

IR *Intermediate representation*

iRTSS *Invasive Run-Time Support System*

ISA Instruktionssatz Architektur

ISE Instruktionssatzerweiterungen

ISS *Chair for Integrated Signal Processing Systems*

ISVLSI *Symposium on VLSI*

ITIV Institut für Technik der Informationsverarbeitung

IU *Integer Unit*

IVT *Integrated Vision Toolkit*

KAHRISMA *KArlsruhe's Hypermorphic Reconfigurable-Instruction-Set Multi-grained-Array Processor*

KIT Karlsruher Institut für Technologie

L Zeilenlänge

L1 *Level 1*

L2 *Level 2*

LDSTUB *Load-Store Unsigned Byte*

LFU *Least Frequently Used*

LImbiC *LISA model-based ideal Cortex-M1*

LISA *Language for Instruction-Set Architectures*

LLC *Last Level Cache*

LMA *Local Minimum Avoidance*

LR *Link Register*

LRR *Least-Recently-Replaced*

LRU *Least-Recently-Used*

LSB *Least Significant Byte*

LUT *Lookup Table*

MAC *Multiply-Accumulate*

ME *Memory*

MIT *Massachusetts Institute of Technology*

ML605 *Virtex-6* (XC6VLX240T)

MMU *Memory Management Unit*

MMul Matrix Multiplikation

MMX *Multi Media Extension*

MORPHEUS *Multi-purpose dynamically Reconfigurable Platform for intensive Heterogeneous processing*

N *Negative*

NoC *Network on Chip*

NOP *No Operation*

nPC nächster Programzähler

OoO *Out-of-Order*

PA *Platform Architect*

PAR *Place & Route*

PC *Program Counter*

PD *Processor Designer*

PG *Processor Generator*

PGAS *Partitioned Global Address Space*

PHT *Pattern History Table*

PPA *Power, Performance, Area*

PPC *PowerPC*

PSR *Processor State Register*

RA *Register Access*
RAM *Random-access Memory*
RAW *Reconfigurable Architectures Workshop*
RAW *Read-After-Write*
ReConFig *Conference on Reconfigurable Computing and FPGAs*
RF *Register-File*
RISC *Reduced Instruction Set Computer*
RISPP *Rotating Instruction Set Processing Platform*
ROM *Read-only Memory*
RTL *Register Transfer Level*
RW Rechenwerk
RWTH Rheinisch-Westfälischen Technische Hochschule

S *Supervisor*
s Satzgröße
SA *Set Associative*
SDPG *Software Development Package Generator*
SFB Sonderforschungsbereich
SIES Symposium für industrielle eingebettete Systeme
SmartLoCore adaptiven Lokalisierungsprozessor
SMP *Symmetric Multiprocessor*
SOC *System-on-Chip*
SP *Stack Pointer*
SPARC *Scalable Processor ARChitecture*
SPEC *Standard Performance Evaluation Corporation*
SRAM *Static Random-access Memory*
SVGX *Stratix V* (5SGXEA7K2F40C2N)
SW Steuerwerk

TBR *Trap Base Register*
TCPA *Tightly-coupled Processor Alrray*
TLM *Transaction Level Modeling*
TtM *Time to Market*
TUM Technische Universität München

UART *Universal Asynchronous Receiver Transmitter*
UML *Unified Modeling Language*

V *Overflow*
VC707 *Virtex-7* (XC7VX485T)
VHDL *Very High Speed Integrated Circuit Hardware Description Language*
VLIW *Very Long Instruction Word*
VLSI *Very-large-scale Integration*

WA *Write Allocate*
WB *Write Back*
WIM *Window Invalid Mask*
WIP *Work in Progress*
WT *Write Through*

XC *Exception*
XUPV5 *Virtex-5* (XC5VLX110T)

Y *Multiply Step Register*

Z *Zero*

Literaturverzeichnis

[1] Abdelgawad, A. ; Bayoumi, M.: High Speed and Area-Efficient Multiply Accumulate (MAC) Unit for Digital Signal Prossing Applications. In: *IEEE International Symposium on Circuits and Systems*, 2007, S. 3199–3202

[2] Agrawal, Anant ; Garner, Robert B.: SPARC: A scalable processor architecture. In: *Future Generation Computer Systems* 7 (1992), S. 303–309

[3] Albonesi, D. H.: Selective cache ways: on-demand cache resource allocation. In: *32nd Annual International Symposium on Microarchitecture*, 1999, S. 248–259

[4] Alpert, D. ; Avnon, D.: Architecture of the Pentium microprocessor. In: *IEEE Micro* 13 (1993), Nr. 3, S. 11–21

[5] An, Deukhyeon ; Kim, Jeehong ; Han, JungHyun ; Eom, Young I.: Reducing Last Level Cache Pollution in NUMA Multicore Systems for Improving Cache Performance. In: *International Conference on Computational Science and Its Applications*, Springer, 2012, S. 272–282

[6] Ansaloni, G. ; Bonzini, P. ; Pozzi, L.: Heterogeneous coarse-grained processing elements: A template architecture for embedded processing acceleration. In: *Design, Automation Test in Europe Conference Exhibition*, 2009, S. 542–547

[7] ARM: *ARM1136J-S technical reference manual.* r1p5, 2009

[8] ARM Holdings: *Investor Relations - Company Overview.* `http://ir.arm.com/phoenix.zhtml?c=197211&p=irol-homeprofile`

[9] ARM Limited (Hrsg.): *ARMv6–M Architecture Reference Manual.* CB1 9NJ: ARM Limited, 2010

[10] ARM Ltd.: *Cortex-M1 Processor - Performance.* online. `http://www.arm.com/products/processors/cortex-m/cortex-m1.php`. Version: Juni 2016

[11] Armour-Brown, Cerion ; Borntraeger, Christian ; Fitzhardinge, Jeremy ; Hughes, Tom ; Jovanovic, Petar ; Jevtic, Dejan ; Krohm, Florian ; Love, Carl ; Johnson, Maynard ; Mackerras, Paul ; Mueller, Dirk ; Nethercote, Nicholas ; Pavlu, Petr ; Raisr, Ivo ; Seward, Julian ; Assche, Bart V. ; Walsh, Robert ; Waroquiers, Philippe ; Weidendorfer, Josef: *Valgrind User Manual Cachegrind: a cache and branch-prediction profiler.* http://valgrind.org/docs/manual/cg-manual.html: Valgrind, 2015

[12] Azad, Pedram ; Gockel, Tilo ; Dillmann, Rüdiger: *Computer Vision: Das Praxisbuch.* Elektor-Verlag, 2007

[13] Bader, Michael ; Bungartz, Hans-Joachim ; Gerndt, Michael ; Hollmann, Andreas ; Weidendorfer, Josef: Invasive Programming as a Concept for HPC. In: *Proc. of the 10th IASTED Int. Conf. on Parallel and Distr. Comp. and Netw.*, 2011

[14] Bai, Ying: *Practical Microcontroller Engineering with ARM Technology.* John Wiley & Sons

[15] Bashford, Steven ; Bieker, Ulrich ; Harking, Berthold ; Leupers, Rainer ; Marwedel, Peter ; Neumann, Andreas ; Voggenauer, Dietmar: The mimola language. In: *University of Dortmund* (1994)

[16] Bauer, Lars ; Shafique, Muhammad ; Kramer, Simon ; Henkel, Jörg: RISPP: Rotating Instruction Set Processing Platform. In: *Proceedings of the 44th Annual Design Automation Conference*, 2007, S. 791–796

[17] Beck, Antonio Carlos S. ; Rutzig, Mateus B. ; Gaydadjiev, Georgi ; Carro, Luigi: Transparent Reconfigurable Acceleration for Heterogeneous Embedded Applications. In: *Proceedings of the Conference on Design, Automation and Test in Europe*, 2008, S. 1208–1213

[18] Becker, J. ; Friederich, S. ; Heisswolf, J. ; Koenig, R. ; May, D.: Hardware prototyping of novel invasive multicore architectures. In: *17th Asia and South Pacific Design Automation Conference*, 2012, S. 201–206

[19] Belady, L. A. ; Nelson, R. A. ; Shedler, G. S.: An Anomaly in Space-time Characteristics of Certain Programs Running in a Paging Machine. In: *Commun. ACM* 12 (1969), Nr. 6, S. 349–353

[20] Bonzini, P. ; Pozzi, L.: Recurrence-Aware Instruction Set Selection for Extensible Embedded Processors. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 16 (2008), Nr. 10, S. 1259–1267

[21] Braun, Matthias ; Buchwald, Sebastian ; Mohr, Manuel ; Zwinkau, Andreas: *An X10 compiler for invasive architectures*. KIT, Fakultät für Informatik, 2012

[22] Brechmann, Eike C. ; Schepsmeier, Ulf: Modeling dependence with C-and D-vine copulas: The R-package CDVine. In: *Journal of Statistical Software* 52 (2013), Nr. 3, S. 1–27

[23] Brinkschulte, Uwe ; Ungerer, Theo: *Mikrocontroller und Mikroprozessoren*. Springer, 2010

[24] Bruckschloegl, T. ; Oey, O. ; Rückauer, M. ; Stripf, T. ; Becker, J.: A Hierarchical Architecture Description for Flexible Multicore System Simulation. In: *IEEE International Symposium on Parallel and Distributed Processing with Applications*, 2014, S. 190–196

[25] Bryant, Randal ; David Richard, O'Hallaron: *Computer systems: a programmer's perspective*. Bd. 281. Prentice Hall Upper Saddle River, 2003

[26] Bryant, Randal E. ; O'Hallaron, David R.: CS:APP2e Web Aside MEM:BLOCKING: Using Blocking to Increase Temporal Locality / Carnegie Mellon University. 2012. – Forschungsbericht

[27] Bull, D. ; Das, S. ; Shivashankar, K. ; Dasika, G. S. ; Flautner, K. ; Blaauw, D.: A Power-Efficient 32 bit ARM Processor Using Timing-Error Detection and Correction for Transient-Error Tolerance and Adaptation to PVT Variation. In: *IEEE Journal of Solid-State Circuits* 46 (2011), Nr. 1, S. 18–31

[28] Bungartz, Hans-Joachim ; Riesinger, Christoph ; Schreiber, Martin ; Snelting, Gregor ; Zwinkau, Andreas: Invasive Computing in HPC with X10. In: *Proceedings of the Third ACM SIGPLAN X10*

Workshop, 2013, S. 12–19

[29] Burgess, B. ; Cohen, B. ; Denman, M. ; Dundas, J. ; Kaplan, D. ; Rupley, J.: Bobcat: AMD's Low-Power x86 Processor. In: *IEEE Micro* 31 (2011), Nr. 2, S. 16–25

[30] Cheung, Newton ; Parameswaran, Sri ; Henkel, J.: INSIDE: INstruction Selection/Identification amp; Design Exploration for extensible processors. In: *International Conference on Computer Aided Design (ICCAD)*, 2003, S. 291–297

[31] Christgau, Steffen ; Schnor, Bettina: Software-managed Cache Coherence for Fast One-Sided Communication. In: *PMAM*, 2016. – ISBN 978–1–4503–4196–7, S. 69–77

[32] COBHAM: *GRLIB IP Core User's Manual*. 1.5.0. 2016

[33] COBHAM: *GRLIB IP Library User's Manual*. 1.5.0. 2016

[34] Dales, M.: Managing a reconfigurable processor in a general purpose workstation environment. In: *Design, Automation and Test in Europe*, 2003, S. 980–985

[35] Damien, Gille: *Study of different cache line replacement algorithms in embedded systems*, KTH, Diss., 2007

[36] Damm, M. ; Grimm, C. ; Haas, J. ; Herrholz, A. ; Nebel, W.: Connecting SystemC-AMS models with OSCI TLM 2.0 models using temporal decoupling. In: *Forum on Specification, Verification and Design Languages*, 2008, S. 25–30

[37] Dijkstra, E. W.: A note on two problems in connexion with graphs. In: *Numerische Mathematik* (1959)

[38] Dillmann, Rüdiger: *Kognitive Systeme*. Vorlesungsskript,

[39] Döbrich, S. ; Hochberger, C.: Effects of Simplistic Online Synthesis for AMIDAR Processors. In: *International Conference on Reconfigurable Computing and FPGAs*, 2009, S. 433–438

[40] Dodani, V. R. ; Kumar, Nikhil ; Nanda, Umakanta ; Mahapatra, K.: Optimization of an Application Specific Instruction Set Processor using Application Description Language. In: *5th International Conference on Industrial and Information Systems*, 2010, S. 325–328

[41] Efthymiou, A. ; Garside, J. D.: Adaptive pipeline depth control for processor power-management. In: *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 2002, S. 454–457

[42] Efthymiou, A. ; Garside, J. D.: Adaptive pipeline structures for speculation control. In: *Ninth International Symposium on Asynchronous Circuits and Systems*, 2003, S. 46–55

[43] Ernst, D. ; Kim, Nam S. ; Das, S. ; Pant, S. ; Rao, R. ; Pham, Toan ; Ziesler, C. ; Blaauw, D. ; Austin, T. ; Flautner, K. ; Mudge, T.: Razor: a low-power pipeline based on circuit-level timing speculation. In: *36th Annual IEEE/ACM International Symposium on Microarchitecture*, 2003, S. 7–18

[44] Fauth, A. ; Praet, J. V. ; Freericks, M.: Describing instruction set processors using nML. In: *European Design and Test Conference*, 1995, S. 503–507

[45] Flik, Thomas ; 2005 (Hrsg.): *Mikroprozessortechnik und Rechnerstrukturen.* Springer, 2005

[46] Flynn, M. J. ; Mitchell, C. L. ; Mulder, J. M.: And Now a Case for More Complex Instruction Sets. In: *Computer* 20 (1987), Nr. 9, S. 71–83

[47] Fog, Agner: The microarchitecture of Intel, AMD and VIA CPUs / Technical University of Denmark. 2016. – Forschungsbericht

[48] Fornai, Peter ; Iványi, Antal: FIFO anomaly is unbounded. In: *CoRR* abs/1003.1336 (2010)

[49] Frigo, Matteo ; Leiserson, Charles E. ; Prokop, Harald ; Ramachandran, Sridhar: Cache-Oblivious Algorithms. In: *ACM Trans. Algorithms* 8 (2012), Nr. 1, S. 4:1–4:22

[50] Gädeke, T. ; Johnson, M. ; Hedley, M. ; Stork, W.: Fusion of wireless ranging and inertial sensors for precise and scalable indoor localization. In: *2014 IEEE International Conference on Communications Workshops (ICC)*, 2014, S. 138–143

[51] Gädeke, T. ; Schmid, J. ; Krüger, M. ; Jany, J. ; Stork, W. ; Müller-Glaser, K. D.: A bi-modal ad-hoc localization scheme for wireless networks based on RSS and ToF fusion. In: *10th Workshop*

on Positioning Navigation and Communication, 2013, S. 1–6

[52] Gädeke, T. ; Schmid, J. ; Zahnlecker, M. ; Stork, W. ; Müller-Glaser, K. D.: Smartphone pedestrian navigation by foot-IMU sensor fusion. In: *Ubiquitous Positioning, Indoor Navigation, and Location Based Service (UPINLBS)*, 2012, S. 1–8

[53] Gaisler: *Leon3/GrLib SOC IP Library.* 2011

[54] Gaisler, J.: A portable and fault-tolerant microprocessor based on the SPARC v8 architecture. In: *International Conference on Dependable Systems and Networks*, 2002, S. 409–415

[55] Galuzzi, Carlo ; Bertels, Koen: The Instruction-Set Extension Problem: A Survey. In: *ACM Trans. Reconfigurable Technol. Syst.* 4 (2011), Nr. 2, S. 18:1–18:28

[56] Garner, R. B. ; Agrawal, A. ; Briggs, F. ; Brown, E. W. ; Hough, D. ; Joy, B. ; Kleiman, S. ; Muchnick, S. ; Namjoo, M. ; Patterson, D. ; Pendleton, J. ; Tuck, R.: The scalable processor architecture (SPARC). In: *Thirty-Third IEEE Computer Society International Conference, Digest of Papers Compcon Spring '88.*, 1988, S. 278–283

[57] Gerndt, M. ; Hollmann, A. ; Meyer, M. ; Schreiber, M. ; Weidendorfer, J.: Invasive computing with iOMP. In: *Forum on Specification and Design Languages (FDL)*, 2012, S. 225–231

[58] Ghenassia, Frank (Hrsg.): *Transaction Level Modeling with SystemC:TLM Concepts and Applications for Embedded Systems.* 2005

[59] Glocker, E. ; Schmitt-Landsiedel, D.: Modeling of temperature scenarios in a multicore processor system. In: *Advances in Radio Science* 11 (2013), S. 219–225

[60] Glocker, Elisabeth ; Chen, Qingqing ; Zaidi, Asheque M. ; Schlichtmann, Ulf ; Schmitt-Landsiedel, Doris: Emulated ASIC Power and Temperature Monitor System for FPGA Prototyping of an Invasive MPSoC Computing Architecture. In: *CoRR* (2014)

[61] Gordon-Ross, Ann ; Lau, Jeremy ; Calder, Brad: Phase-based Cache Reconfiguration for a Highly-configurable Two-level Cache Hierarchy. In: *Proceedings of the 18th ACM Great Lakes Symposium*

on VLSI, 2008, S. 379–382

[62] Gordon-Ross, Ann ; Vahid, Frank: A Self-tuning Configurable Cache. In: *Proceedings of the 44th Annual Design Automation Conference*, 2007, S. 234–237

[63] Götzfried, Johannes ; Müller, Tilo ; Clercq, Ruan de ; Maene, Pieter ; Freiling, Felix ; Verbauwhede, Ingrid: Soteria: Offline Software Protection Within Low-cost Embedded Devices. In: *Proceedings of the 31st Annual Computer Security Applications Conference*, 2015, S. 241–250

[64] Grudnitsky, A. ; Bauer, L. ; Henkel, J.: COREFAB: Concurrent reconfigurable fabric utilization in heterogeneous multi-core systems. In: *2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2014, S. 1–10

[65] Guthaus, M. R. ; Ringenberg, J. S. ; Ernst, D. ; Austin, T. M. ; Mudge, T. ; Brown, R. B.: MiBench: A free, commercially representative embedded benchmark suite. In: *IEEE International Workshop on Workload Characterization*, 2001, S. 3–14

[66] Hadjiyiannis, George ; Hanono, Silvina ; Devadas, Srinivas: ISDL: An Instruction Set Description Language for Retargetability. In: *Proceedings of the 34th Annual Design Automation Conference*, 1997, S. 299–302

[67] Halambi, A. ; Grun, P. ; Ganesh, V. ; Khare, A. ; Dutt, N. ; Nicolau, A.: EXPRESSION: a language for architecture exploration through compiler/simulator retargetability. In: *Design, Automation and Test in Europe*, 1999, S. 485–490

[68] Hameed, Fazal ; Bauer, Lars ; Henkel, Jörg: Reducing Latency in an SRAM/DRAM Cache Hierarchy via a Novel Tag-Cache Architecture. In: *Proceedings of the 51st Annual Design Automation Conference*, 2014, S. 37:1–37:6

[69] Hannig, Frank ; Lari, Vahid ; Boppu, Srinivas ; Tanase, Alexandru ; Reiche, Oliver: Invasive Tightly-Coupled Processor Arrays: A Domain-Specific Architecture/Compiler Co-Design Approach. In: *ACM Trans. Embed. Comput. Syst.* 13 (2014), Nr. 4s, S. 133:1–133:29

[70] Haubelt, Christian ; Teich, Jürgen (Hrsg.): *Digitale Hardware/Software-Systeme: Spezifikation und Verifikation*. 2010

[71] Heisswolf, J. ; Zaib, A. ; Weichslgartner, A. ; Karle, M. ; Singh, M. ; Wild, T. ; Teich, J. ; Herkersdorf, A. ; Becker, J.: The Invasive Network on Chip - A Multi-Objective Many-Core Communication Infrastructure. In: *27th International Conference on Architecture of Computing Systems (ARCS)*, 2014, S. 1–8

[72] Heisswolf, Jan: *A Scalable and Adaptive Network on Chip for Many-Core Architectures*, Karlsruher Institut für Technologie (KIT), Dissertation, 2014

[73] Henkel, J.: Closing the SoC design gap. In: *Computer* 36 (2003), Nr. 9, S. 119–121

[74] Henkel, J. ; Herkersdorf, A. ; Bauer, L. ; Wild, T. ; Hübner, M. ; Pujari, R. K. ; Grudnitsky, A. ; Heisswolf, J. ; Zaib, A. ; Vogel, B. ; Lari, V. ; Kobbe, S.: Invasive manycore architectures. In: *17th Asia and South Pacific Design Automation Conference*, 2012, S. 193–200

[75] Henkel, Jörg ; Bauer, Lars ; Hübner, Michael ; Grudnitsky, Artjom: i-Core: A run-time adaptive processor for embedded multi-core systems. In: *International Conference on Engineering of Reconfigurable Systems and Algorithms*, 2011

[76] Hennessy, J. L.: VLSI Processor Architecture. In: *IEEE Transactions on Computers* C-33 (1984), Nr. 12, S. 1221–1246

[77] Hennessy, John L. ; Patterson, David A.: *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th. Morgan Kaufmann Publishers Inc., 2011

[78] Henry, G. G.: The VIA Isaiah Architecture / Centaur Technology, Inc. Version: 2008. `http://www.centtech.com/wp-content/uploads/2014/09/WP2-080124Isaiah-architecture-brief.pdf`. 2008. – Forschungsbericht

[79] Hill, M. D. ; Smith, A. J.: Evaluating associativity in CPU caches. In: *IEEE Transactions on Computers* 38 (1989), Nr. 12, S. 1612–1630

[80] Hoffmann, A. ; Kogel, T. ; Nohl, A. ; Braun, G. ; Schliebusch, O. ; Wahlen, O. ; Wieferink, A. ; Meyr, H.: A novel

methodology for the design of application-specific instruction-set processors (ASIPs) using a machine description language. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20 (2001), Nr. 11, S. 1338–1354

[81] Hoffmann, Andreas ; Meyr, Heinrich ; Leupers, Rainer: *Architecture Exploration for Embedded Processors with LISA*. 2002

[82] Huynh, Huynh P. ; Mitra, Tulika: Instruction-set Customization for Real-time Embedded Systems. In: *Proceedings of the Conference on Design, Automation and Test in Europe*, 2007, S. 1472–1477

[83] Huynh, Huynh P. ; Mitra, Tulika: Runtime Adaptive Extensible Embedded Processors – A Survey. In: *Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, 2009, S. 215–225

[84] Inoue, K. ; Ishihara, T. ; Murakami, K.: Way-predicting set-associative cache for high performance and low energy consumption. In: *International Symposium on Low Power Electronics and Design*, 1999, S. 273–275

[85] Intel: Improving Performance by Utilizing Cache Allocation / Intel. 2015 (331843-001US). – White Paper

[86] Intel Corporation: *About Intel - Investor Relations*. `http://www.intc.com/`

[87] Intel Corporation: Improving Real-Time Performance by Utilizing Cache Allocation Technology: Enhancing Performance via Allocation of the Processor's Caches / Intel Corporation. 2015 (331843-001US). – White Paper

[88] Jeff, Brian: Advances in big. LITTLE Technology for Power and Energy Savings (Improving Energy Efficiency in Mobile Devices) / ARM. 2012. – White Paper

[89] Jeff, Brian: big.LITTLE Technology Moves Towards Fully Heterogeneous Global Task Scheduling (Improving Energy Efficiency and Performance in Mobile Devices) / ARM. 2013. – White Paper

[90] *Kapitel* Compiler-Directed Cache Assist Adaptivity. In: Ji, Xiaomei ; Nicolaescu, Dan ; Veidenbaum, Alexander ; Nicolau, Alexandru ; Gupta, Rajesh: *High Performance Computing: Third International*

Symposium. Springer Berlin Heidelberg, 2000, S. 88–104

[91] Jimenez, D. A. ; Lin, C.: Dynamic branch prediction with perceptrons. In: *The Seventh International Symposium on High-Performance Computer Architecture*, 2001, S. 197–206

[92] Jindal, R. ; Jain, K.: Verification of transaction-level SystemC models using RTL testbenches. In: *First ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, 2003, S. 199–203

[93] Jouppi, Norman P.: Improving Direct-mapped Cache Performance by the Addition of a Small Fully-associative Cache Prefetch Buffers. In: *25 Years of the International Symposia on Computer Architecture*, 1998, S. 388–397

[94] Jourdan, S. ; Sainrat, P. ; Litaize, D.: Exploring configurations of functional units in an out-of-order superscalar processor. In: *22nd Annual International Symposium on Computer Architecture*, 1995, S. 117–125

[95] Juan, Toni ; Sanjeevan, Sanji ; Navarro, Juan J.: Dynamic History-length Fitting: A Third Level of Adaptivity for Branch Prediction. In: *SIGARCH Comput. Archit. News* 26 (1998), Nr. 3, S. 155–166

[96] Kanter, David: AMD's Bulldozer Microarchitecture / Real World Tech. Version: 2010. `http://www.realworldtech.com/bulldozer/`. 2010. – Forschungsbericht

[97] Kanter, David: Silvermont, Intel's Low Power Architectures / Real World Tech. Version: 2013. `http://www.realworldtech.com/silvermont/`. 2013. – Forschungsbericht

[98] Keutzer, K. ; Malik, S. ; Newton, A. R.: From ASIC to ASIP: the next design discontinuity. In: *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 2002, S. 84–90

[99] Koenig, R. ; Bauer, L. ; Stripf, T. ; Shafique, M. ; Ahmed, W. ; Becker, J. ; Henkel, J.: KAHRISMA: A Novel Hypermorphic Reconfigurable-Instruction-Set Multi-grained-Array Architecture. In: *Design, Automation Test in Europe*, 2010, S. 819–824

[100] Lee, J. K. F. ; Smith, A. J.: Branch Prediction Strategies and Branch Target Buffer Design. In: *Computer* 17 (1984), Nr. 1, S. 6–22

[101] Levitan, D. ; Thomas, T. ; Tu, P.: The PowerPC 620 microprocessor: a high performance superscalar RISC microprocessor. In: *Compcon '95.'Technologies for the Information Superhighway', Digest of Papers.*, 1995, S. 285–291

[102] *Kapitel* Euro-Par. In: Lodde, Mario ; Flich, Jose ; Acacio, Manuel E.: *Dynamic Last-Level Cache Allocation to Reduce Area and Power Overhead in Directory Coherence Protocols.* Springer, 2012, S. 206–218

[103] Lysecky, Roman ; Stitt, Greg ; Vahid, Frank: Warp Processors. In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 11 (2004), Nr. 3, S. 659–681

[104] Lysecky, Roman ; Vahid, Frank: Design and Implementation of a MicroBlaze-based Warp Processor. In: *ACM Transactions on Embedded Computing Systems (TECS)* 8 (2009), Nr. 3, S. 22:1–22:22

[105] Malik, A. ; Moyer, B. ; Cermak, D.: A low power unified cache architecture providing power and performance flexibility. In: *Proceedings of the 2000 International Symposium on Low Power Electronics and Design*, 2000, S. 241–243

[106] Marsala, A. ; Kanawati, B.: PowerPC processors. In: *Proceedings of the 26th Southeastern Symposium on System Theory*, 1994, S. 550–556

[107] Marty, Michael R.: *Cache coherence techniques for multicore processors*, University of Wisconsin - Madison, Diss., 2008

[108] McFarling, S. ; Hennesey, J.: Reducing the Cost of Branches. In: *SIGARCH Computer Architecture News* 14 (1986), S. 396–403

[109] MIPS Technologies, Inc.: *Company - About Us.* `http://www.mips.com/company/about-us/`

[110] Mishra, Prabhat ; Dutt, Nikil: *Processor Description Languages.* Morgan Kaufmann Publishers Inc., 2008

[111] Mohr, Manuel ; Buchwald, Sebastian ; Zwinkau, Andreas ; Erhardt, Christoph ; Oechslein, Benjamin ; Schedel, Jens ; Lohmann, Daniel: Cutting out the Middleman: OS-level Support for

x10 Activities. In: *Proceedings of the ACM SIGPLAN Workshop on X10*, 2015 (X10 2015), S. 13–18

[112] Muthukaruppan, Thannirmalai S. ; Pricopi, Mihai ; Venkataramani, Vanchinathan ; Mitra, Tulika ; Vishin, Sanjay: Hierarchical Power Management for Asymmetric Multi-core in Dark Silicon Era. In: *Proceedings of the 50th Annual Design Automation Conference*, 2013, S. 174:1–174:9

[113] *Kapitel* Compiler-Directed Cache Line Size Adaptivity. In: Nicolaescu, Dan ; Ji, Xiaomei ; Veidenbaum, Alexander ; Nicolau, Alexandru ; Gupta, Rajesh: *Intelligent Memory Systems: Second International Workshop*. Springer Berlin Heidelberg, 2001, S. 183–187

[114] Nikov, K. ; Nunez-Yanez, J. L. ; Horsnell, M.: Evaluation of Hybrid Run-Time Power Models for the ARM Big.LITTLE Architecture. In: *13th International Conference on Embedded and Ubiquitous Computing (EUC)*, 2015, S. 205–210

[115] Nohl, Achim ; Schirrmeister, Frank ; Taussig, Drew: Application Specific Processor Design Architectures, Design Methods and Tools. In: *Proceedings of the International Conference on Computer-Aided Design*, IEEE Press, 2010, S. 349–352

[116] Nowak, Fabian ; Buchty, Rainer ; Karl, Wolfgang: A Run-time Reconfigurable Cache Architecture. In: *International Conference on Parallel Computing: Architectures, Algorithms and Applications*, 2007

[117] Oechslein, Benjamin ; Schedel, Jens ; Kleinöder, Jürgen ; Bauer, Lars ; Henkel, Jörg ; Lohmann, Daniel ; Schröder-Preikschat, Wolfgang: OctoPOS: A parallel operating system for invasive computing. In: *Proceedings of the International Workshop on Systems for Future Multi-Core Architectures*, 2011, S. 9–14

[118] Ogita, Takeshi ; Rump, Siegfried M. ; Oishi, Shin'ichi: Accurate Sum and Dot Product. In: *SIAM Journal on Scientific Computing* 26 (2005), Nr. 6, S. 1955–1988

[119] Okunev, P. ; Johnson, C. R.: Necessary And Sufficient Conditions For Existence of the LU Factorization of an Arbitrary Matrix. In: *ArXiv Mathematics e-prints* (2005)

[120] Pan, Shien-Tai ; So, Kimming ; Rahmeh, Joseph T.: Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation. In: *SIGPLAN Not.* 27 (1992), Nr. 9, S. 76–84

[121] Park, N. ; Hong, B. ; Prasanna, V. K.: Tiling, block data layout, and memory hierarchy performance. In: *IEEE Transactions on Parallel and Distributed Systems* 14 (2003), Nr. 7, S. 640–654

[122] Patterson, David A.: Reduced Instruction Set Computers. In: *Commun. ACM* 28 (1985), S. 8–21

[123] Patterson, David A. ; Ditzel, David R.: The Case for the Reduced Instruction Set Computer. In: *SIGARCH Computer Architecture News* 8 (1980), Nr. 6, S. 25–33

[124] Patterson, David A. ; Hennessy, John L.: *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface*. 4th. Morgan Kaufmann Publishers Inc., 2008

[125] Patterson, David A. ; Hennessy, John L.: *Rechnerorganisation und Rechnerentwurf : die Hardware/Software-Schnittstelle*. Oldenbourg, 2011

[126] Paul, J. ; Stechele, W. ; Kröhnert, M. ; Asfour, T. ; Dillmann, R.: Invasive Computing for robotic vision. In: *17th Asia and South Pacific Design Automation Conference*, 2012, S. 207–212

[127] Pees, Stefan ; Hoffmann, Andreas ; Zivojnovic, Vojin ; Meyr, Heinrich: LISA—Machine Description Language for Cycle-accurate Models of Programmable DSP Architectures. In: *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, 1999, S. 933–938

[128] Peleg, A. ; Weiser, U.: MMX technology extension to the Intel architecture. In: *IEEE Micro* 16 (1996), Nr. 4, S. 42–50

[129] Prescher, Thomas ; Rotta, Randolf ; Nolte, Jörg: Flexible Sharing and Replication Mechanisms for Hybrid Memory Architectures. In: *MARC* Bd. 55, 2011, S. 67–72

[130] Ranganathan, Parthasarathy ; Adve, Sarita ; Jouppi, Norman P.: Reconfigurable Caches and Their Application to Media Processing. In: *SIGARCH Comput. Archit. News* 28 (2000), Nr. 2, S. 214–224

[131] Reid, Matthew W.: Pivoting for LU Factorization / University of Puget Sound. 2014. – Forschungsbericht

[132] Rissa, Tero ; Donlin, Adam ; Luk, Wayne: Evaluation of SystemC Modelling of Reconfigurable Embedded Systems. In: *Proceedings of the Conference on Design, Automation and Test in Europe*, 2005, S. 253–258

[133] Roloff, S. ; Hannig, F. ; Teich, J.: Approximate time functional simulation of resource-aware programming concepts for heterogeneous MPSoCs. In: *17th Asia and South Pacific Design Automation Conference*, 2012

[134] Rothberg, Edward ; Singh, Jaswinder P. ; Gupta, Anoop: Working Sets, Cache Sizes, and Node Granularity Issues for Large-scale Multiprocessors. In: *SIGARCH Comput. Archit. News* 21 (1993), Nr. 2, S. 14–26

[135] Rotta, Randolf ; Prescher, Thomas ; Traue, Jana ; Nolte, Jörg: Data Sharing Mechanisms for Parallel Graph Algorithms on the Intel SCC. In: *MARC*, 2012, S. 13–18

[136] Rüde, U.: *Mathematical and Computational Techniques for Multilevel Adaptive Methods*. Society for Industrial and Applied Mathematics, 1993

[137] Sair, Suleyman ; Charney, Mark: Memory behavior of the SPEC2000 benchmark suite / IBM TJ Watson Research Center. 2000. – Forschungsbericht

[138] Saraswat, Vijay ; Bloom, Bard ; Peshansky, Igor ; Tardieu, Olivier ; Grove, David: X10 language specification / IBM. 2014 (2.5). – Forschungsbericht

[139] Schliebusch, O. ; Hoffmann, A. ; Nohl, A. ; Braun, G. ; Meyr, H.: Architecture implementation using the machine description language LISA. In: *7th Asia and South Pacific and the 15th International Conference on VLSI Design*, 2002, S. 239–244

[140] Schliebusch, Oliver ; Meyr, Heinrich ; Leupers, Rainer: *Optimized ASIP Synthesis from Architecture Description Language Models*. Springer, 2007

[141] Schreiber, Martin: *Cluster-Based Parallelization of Simulations on Dynamically Adaptive Grids and Dynamic Resource Management*, Technische Universität München, Diss., 2014

[142] Schreiber, Martin ; Riesinger, Christoph ; Neckel, Tobias ; Bungartz, Hans-Joachim ; Breuer, Alexander: Invasive Compute Balancing for Applications with Shared and Hybrid Parallelization. In: *International Journal of Parallel Programming* 43 (2015), Nr. 6, S. 1004–1027

[143] Seal, David: *ARM architecture reference manual*. Pearson Education, 2001

[144] Shafique, M. ; Vogel, B. ; Henkel, J.: Self-adaptive hybrid Dynamic Power Management for many-core systems. In: *Design, Automation Test in Europe Conference Exhibition*, 2013, S. 51–56

[145] Shafique, Muhammad ; Garg, Siddharth ; Henkel, Jörg ; Marculescu, Diana: The EDA Challenges in the Dark Silicon Era: Temperature, Reliability, and Variability Perspectives. In: *Proceedings of the 51st Annual Design Automation Conference*, 2014, S. 185:1–185:6

[146] Shafique, Muhammad ; Garg, Siddharth ; Mitra, Tulika ; Parameswaran, Sri ; Henkel, Jörg: Dark Silicon As a Challenge for Hardware/Software Co-design: Invited Special Session Paper. In: *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*, 2014, S. 13:1–13:10

[147] Silc, Jurij ; Robic, Borut ; Ungerer, Theo: *Processor architecture: from dataflow to superscalar and beyond*. Springer Science & Business Media, 2012

[148] Smith, J. E. ; Sohi, G. S.: The microarchitecture of superscalar processors. In: *Proceedings of the IEEE* 83 (1995), Nr. 12, S. 1609–1624

[149] Smith, James E.: A Study of Branch Prediction Strategies. In: *25 Years of the International Symposia on Computer Architecture (Selected Papers)*, 1998, S. 202–215

[150] Song, S. P. ; Denman, M. ; Chang, J.: The PowerPC 604 RISC microprocessor. In: *IEEE Micro* 14 (1994), Nr. 5, S. 8–

[151] SPARC International, Inc.: *The SPARC Architecture Manual Version 8*. 1992

[152] SPARC International, Inc.: *The SPARC Architecture Manual Version 9*. 1994

[153] Srinivasan, S. ; Rodrigues, R. ; Annamalai, A. ; Koren, I. ; Kundu, S.: A study on polymorphing superscalar processor dynamically to improve power efficiency. In: *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2013, S. 46–51

[154] Stripf, T. ; König, R. ; Becker, J.: A novel ADL-based compiler-centric software framework for reconfigurable mixed-ISA processors. In: *International Conference on Embedded Computer Systems (SAMOS)*, 2011, S. 157–164

[155] Stripf, T. ; König, R. ; Becker, J.: A cycle-approximate, mixed-ISA simulator for the KAHRISMA architecture. In: *Design, Automation Test in Europe*, 2012, S. 21–26

[156] Synopsys: *Platform Architect Datasheet*, 2010

[157] Synopsys: *Processor Designer Product Family: Processor Design Guide*, 2010

[158] Tao, Jie ; Kunze, Marcel ; Nowak, Fabian ; Buchty, Rainer ; Karl, Wolfgang: Performance Advantage of Reconfigurable Cache Design on Multicore Processor Systems. In: *International Journal of Parallel Programming* 36 (2008), Nr. 3, S. 347–360

[159] Teich, J. ; Weichslgartner, A. ; Oechslein, B. ; Schröder-Preikschat, W.: Invasive computing - Concepts and overheads. In: *Forum on Specification and Design Languages*, 2012, S. 217–224

[160] Teich, Jürgen: Invasive Algorithms and Architectures Invasive Algorithmen und Architekturen. In: *it - Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik* 50 (2008), Nr. 5, S. 300–310

[161] Teich, Jürgen ; Henkel, Jörg ; Herkersdorf, Andreas ; Schmitt-Landsiedel, Doris ; Schröder-Preikschat, Wolfgang ; Snelting, Gregor ; Hübner, Michael (Hrsg.) ; Becker, Jürgen (Hrsg.): *Invasive Computing: An Overview*. 2011. – 241–268 S.

[162] Thoma, F. ; Kuhnle, M. ; Bonnot, P. ; Panainte, E. M. ; Bertels, K. ; Goller, S. ; Schneider, A. ; Guyetant, S. ; Schuler, E. ; Muller-Glaser, K. D. ; Becker, J.: MORPHEUS: Heterogeneous Reconfigurable Computing. In: *2007 International Conference on Field Programmable Logic and Applications*, 2007, S. 409–414

[163] Vries, Hans de: Understanding the detailed Architecture of AMD's 64 bit Core / Chip Architect. Version: 2003. `http://www.chip-architect.com/news/2003_09_21_Detailed_Architecture_of_AMDs_64bit_Core.html`. 2003. – Forschungsbericht

[164] *Kapitel* Architecture of Computing Systems. In: Warrier, Tripti S. ; Anupama, B. ; Mutyam, Madhu: *An Application-Aware Cache Replacement Policy for Last-Level Caches*. Springer, 2013, S. 207–219

[165] Weicker, Reinhold P.: Dhrystone: a synthetic systems programming benchmark. In: *Communications of the ACM* 27 (1984), Nr. 10, S. 1013–1030

[166] Weicker, Reinhold P.: An overview of common benchmarks. In: *Computer* 23 (1990), Nr. 12, S. 65–75

[167] Wolf, Michael E. ; Lam, Monica S.: A Data Locality Optimizing Algorithm. In: *SIGPLAN Not.* 26 (1991), Nr. 6, S. 30–44

[168] Ye, Zhi A. ; Moshovos, Andreas ; Hauck, Scott ; Banerjee, Prithviraj: CHIMAERA: A High-performance Architecture with a Tightly-coupled Reconfigurable Functional Unit. In: *SIGARCH Computer Architecture News* 28 (2000), Nr. 2, S. 225–235

[169] Yeh, Tse-Yu ; Patt, Yale N.: Alternative Implementations of Two-level Adaptive Branch Prediction. In: *SIGARCH Comput. Archit. News* 20 (1992), Nr. 2, S. 124–134

[170] Yeh, Tse-Yu ; Patt, Yale N.: A Comparison of Dynamic Branch Predictors That Use Two Levels of Branch History. In: *SIGARCH Comput. Archit. News* 21 (1993), Nr. 2, S. 257–266

[171] Yeh, T.Y. ; Sharangpani, H.P.: *Method and apparatus for branch prediction using first and second level branch prediction tables*. 2001

[172] Zhang, C. ; Vahid, F. ; Najjar, W.: A highly configurable cache architecture for embedded systems. In: *30th Annual International*

Symposium on Computer Architecture, 2003, S. 136–146

[173] Zivojnovic, V. ; Pees, S. ; Meyr, H.: LISA-machine description language and generic machine model for HW/SW co-design. In: *Workshop on VLSI Signal Processing*, 1996, S. 127–136

Betreute studentische Arbeiten

[An14] An, Bao N.: *Design und Modellierung einer adaptiven superskalaren Prozessorpipeline auf Basis des LEON3*, Karlsruher Institut für Technologie, Masterarbeit Nr. 1823, Mai 2014

[Bra12] Braun, Maximilian: *Design und Implementierung von adaptiven Cache-Strukturen*, Karlsruher Institut für Technologie, Bachelorarbeit Nr. 1618, Oktober 2012

[Cor12] Cordero, Enrique: *Bestimmung von Temperaturgradienten auf rekonfigurierbarer Hardware*, Karlsruher Institut für Technologie, Bachelorarbeit Nr. 1567, Mai 2012

[Cor14] Cordero, Enrique: *Emulation von Multi-Level Leistungsumrichtern*, Karlsruher Institut für Technologie, Masterarbeit Nr. 1861, September 2014

[Dec14] Decock, Laurent: *Design und Modellierung des SmartLoCore ASIP-Konzeptes für Lokalisierungsalgorithmen*, Karlsruher Institut für Technologie, Bachelorarbeit Nr. 1828, April 2014

[Dör15] Dörr, Tobias: *Evaluierung einer adaptiven Cache-Architektur anhand von invasiven Anwendungsszenarien*, Karlsruher Institut für Technologie, Bachelorarbeit Nr. 2012, Oktober 2015

[Gei15] Geiger, Michael: *Automatische Laufzeitgenerierung von Schleifen-Beschleunigern mit partieller dynamischer Rekonfiguration innerhalb der invasiven Mikroarchitektur*, Karlsruher Institut für Technologie, Masterarbeit Nr. 1963, Mai 2015

[Har12] Harbaum, Tanja: *Konzeptionierung, Modellierung und anwendungsspezifische Optimierung eines Low-Power Prozessors auf Basis der ARM Architektur*, Karlsruher Institut für Technologie, Diplomarbeit Nr. 1594, November 2012

[Här13] Härdle, Manuel: *Evaluation, Implementierung und Optimierung von adaptiven Sprungvorhersagen*, Karlsruher Institut für Technologie, Diplomarbeit Nr. 1643, Januar 2013

[Hel13] Held, Felix: *Vergleich von modellbasierten high-level Beschreibungen für FPGA Zielarchitekturen anhand von Audio-Signalverarbeitungsalgorithmen*, Karlsruher Institut für Technologie, Bachelorarbeit Nr. 1710, Mai 2013

[Hel16] Held, Felix: *Ermöglichen der Software-Basierten Laufzeit-Parametrisierung der adaptiven Cache Architektur in einem Shared Memory Multi-/Many Core System*, Karlsruher Institut für Technologie, Masterarbeit Nr. 2117, Juni 2016

[Knj13] Knjasev, Viktor: *Design und Implementierung einer laufzeitadaptiven balancierten Prozessor-Pipeline auf Basis des LEON3*, Karlsruher Institut für Technologie, Bachelorarbeit Nr. 1668, April 2013

[LM13] Lucio Martinez, Josè A.: *Adaptive digitale Audiosignalverarbeitung auf partiell und dynamisch rekonfigurierbarer Hardware*, Karlsruher Institut für Technologie, Masterarbeit Nr. 1762, November 2013

[Mas13] Masing, Leonard: *Modellierung und Evaluation der invasiven Mikroarchitektur*, Karlsruher Institut für Technologie, Diplomarbeit Nr. 1671, April 2013

[Mec13] Mechler, Michael: *Erweiterung der Co-Simulation um Online und Offline Debugging Methoden*, Karlsruher Institut für Technologie, Bachelorarbeit Nr. 1670, März 2013

[Mec16] Mechler, Michael: *Flexibles, partielles Parametrisieren von Software-definierten Bereichen des adaptiven Caches zur Laufzeit in einem Shared Memory Multi-/Many Core System*, Karlsruher Institut für Technologie, Masterarbeit Nr. 2116, Juni 2016

[Nit15] Nitzsche, Sven: *Evaluation der adaptiven Sprungvorhersage des i-Cores in der InvasIC Architektur*, Karlsruher Institut für Technologie, Bachelorarbeit Nr. 1985, Juli 2015

[Nug14] Nugroho, Louis: *Evaluation der adaptiven Sprungvorhersage des i-Cores zur Anwendbarkeit in invasiven Applikationsszenarien*,

Karlsruher Institut für Technologie, Bachelorarbeit Nr. 1810, April 2014

[Ors14] Orsinger, Christoph: *Dynamische Reallokation von rekonfigurierbaren adaptiven Multi-Core Cache-Strukturen*, Karlsruher Institut für Technologie, Diplomarbeit Nr. 1825, Mai 2014

[Pac15] Pachideh, Brian: *Optimierung der adaptiven Sprungvorhersage des i-Cores in der InvasIC Architektur*, Karlsruher Institut für Technologie, Bachelorarbeit Nr. 1986, Juli 2015

[Sch14] Schmitt, Eric: *Validation of Mechatronic Systems*, Karlsruher Institut für Technologie, Masterarbeit Hector School, November 2014

[Sei12] Seidenspinner, Erik: *Design und Implementierung von Signalverarbeitungsalgorithmen auf unterschiedlichen Zielarchitekturen*, Karlsruher Institut für Technologie, Bachelorarbeit Nr. 1615, Oktober 2012

[Tsa13] Tsague, Jean P.: *Entwurf und Modellierung eines online Hardware-Monitors für die LEON3 Prozessorpipeline*, Karlsruher Institut für Technologie, Diplomarbeit Nr. 1651, Februar 2013

[Ves13] Vesper, Malte: *Design und Implementierung von rekonfigurierbaren adaptiven Cache-Strukturen auf Xilinx FPGAs*, Karlsruher Institut für Technologie, Masterarbeit Nr. 1721, September 2013

Eigene Veröffentlichungen

Journale & Buchkapitel

[BGT+15] Beuth, T. ; Gaedeke, T. ; Tradowsky, C. ; Becker, J. E. ; Klimm, A. ; Sander, O.: The Road to ÏTIV Labs- an Integrated Concept for Project-Oriented Systems Engineering Education. In: *International Journal of Information and Education Technology* 5 (2015), Nr. 4, S. 250–254. – Best Paper of the Session Award from the Committee of ICMEI 2014

[DBF+13] Delicia, G. Shalina P. ; Bruckschloegl, Thomas ; Figuli, Peter ; Tradowsky, Carsten ; Almeida, Gabriel M. ; Becker, Juergen: Bringing Accuracy to Open Virtual Platforms (OVP): A Safari from High-Level Tools to Low-Level Microarchitectures. In: *International Journal of Computer Applications* International Conference on Innovations In Intelligent Instrumentation, Optimization And Signal Processing (ICIIIOSP) (2013), Nr. 10, S. 22–27

[FTM+15] Figuli, Peter ; Tradowsky, Carsten ; Martinez, Jose ; Sidiropoulos, Harry ; Siozios, Kostas ; Stenschke, Holger ; Soudris, Dimitrios ; Becker, Jürgen: A Novel Concept for Adaptive Signal Processing on Reconfigurable Hardware. In: *Applied Reconfigurable Computing: 11th International Symposium, ARC*. Springer International Publishing, 2015, S. 311–320

[SFS+15] Siozios, Kostas ; Figuli, Peter ; Sidiropoulos, Harry ; Tradowsky, Carsten ; Diamantopoulos, Dionysios ; Maragos,

Konstantinos ; Delicia, Shalina P. ; Soudris, Dimitrios ; Becker, Jürgen: TEAChER: TEach AdvanCEd Reconfigurable Architectures and Tools. In: *Applied Reconfigurable Computing: 11th International Symposium, ARC*. Springer International Publishing, 2015, S. 103–114. – Nominated as Best Paper Candidate

[TCO+16a] Tradowsky, Carsten ; Cordero, Enrique ; Orsinger, Christoph ; Vesper, Malte ; Becker, Jürgen: Adaptive Cache Structures. In: *Architecture of Computing Systems – ARCS*. Springer International Publishing, 2016, S. 87–99

[TCO+16b] Tradowsky, Carsten ; Cordero, Enrique ; Orsinger, Christoph ; Vesper, Malte ; Becker, Jürgen: A Dynamic Cache Architecture for Efficient Memory Resource Allocation in Many-Core Systems. In: *Applied Reconfigurable Computing: 12th International Symposium, ARC*. Springer International Publishing, 2016, S. 343–351

[THMB16] Tradowsky, C. ; Harbaum, T. ; Masing, L. ; Becker, J.: A Novel ADL-based Approach to Design Adaptive Application-Specific Processors. In: *Best of ISVLSI*. Springer International Publishing, 2016. – Forthcoming

[TLW+15] Tradowsky, Carsten ; Lauber, Andreas ; Werner, Stephan ; Beuth, Thorsten ; Müller-Glaser, Klaus D. ; Sax, Eric: Porter for the ITIV LABS – Objective-Related Engineering Education in an Undergraduate Laboratory. In: *Journal of Teaching and Education* Bd. 4. 2015, S. 45–58

[TSV+14] Tradowsky, Carsten ; Schreiber, Martin ; Vesper, Malte ; Domladovec, Ivan ; Braun, Maximilian ; Bungartz, Hans-Joachim ; Becker, Jürgen: Towards Dynamic Cache and Bandwidth Invasion. In: *Reconfigurable Computing: Architectures, Tools, and Applications: 10th International Symposium, ARC*. 2014

Konferenz- & Workshopbeiträge

[FTGB13] Figuli, P. ; Tradowsky, C. ; Gaertner, N. ; Becker, J.: ViSA: A highly efficient slot architecture enabling multi-objective ASIP cores. In: *International Symposium on System on Chip (SoC)*, 2013, S. 1–8

[HGT+12] Hübner, M. ; Goehringer, D. ; Tradowsky, C. ; Henkel, J. ; Becker, J.: Adaptive processor architecture - invited paper. In: *International Conference on Embedded Computer Systems (SAMOS)*, 2012, S. 244–251

[HTG+11] Hübner, M. ; Tradowsky, C. ; Gohringer, D. ; Braun, L. ; Thoma, F. ; Henkel, J. ; Becker, J.: Dynamic Processor Reconfiguration. In: *2011 International Conference on Reconfigurable Computing and FPGAs*, 2011, S. 123–128

[MT17] Mohr, Manuel ; Tradowsky, Carsten: Pegasus: Efficient Data Transfers for PGAS Languages on Non-Cache-Coherent Many-Cores. In: *Design, Automation Test in Europe*, 2017. – Submitted

[TCD+12] Tradowsky, C. ; Cordero, E. ; Deuser, T. ; Hübner, M. ; Becker, J.: Determination of on-chip temperature gradients on reconfigurable hardware. In: *International Conference on Reconfigurable Computing and FPGAs*, 2012, S. 1–8

[TFS+13] Tradowsky, Carsten ; Figuli, Peter ; Seidenspinner, Erik ; Held, Felix ; Becker, Jürgen: A New Approach to Model-Based Development for Audio Signal Processing. In: *Audio Engineering Society Convention 134*, 2013

[TGB+14] Tradowsky, C. ; Gädeke, T. ; Bruckschlögl, T. ; Stork, W. ; Müller-Glaser, K. D. ; Becker, J.: SmartLoCore: A Concept for an Adaptive Power-Aware Localization Processor. In: *22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2014, S. 478–481

[THDB13] Tradowsky, C. ; Harbaum, T. ; Deyerle, S. ; Becker, J.: LImbiC: An adaptable architecture description language model for developing an application-specific image processor. In: *IEEE Computer Society Annual Symposium on VLSI*

(ISVLSI), 2013, S. 34–39

[TTHB12a] Tradowsky, C. ; Thoma, F. ; Hübner, M. ; Becker, J.: On Dynamic Run-time Processor Pipeline Reconfiguration. In: *IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW)*, 2012, S. 419–424

[TTHB12b] Tradowsky, Carsten ; Thoma, Florian ; Hübner, Michael ; Becker, Jürgen: LISPARC: Using an architecture description language approach for modelling an adaptive processor microarchitecture (Best Work-in-Progress (WiP) Paper Award). In: *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*, 2012, S. 279–282. – Best Work-in-Progress (WiP) Paper Award

Patente

[BMH+13] Biggs, John P. ; Myers, James E. ; Howard, David W. ; Flynn, David W. ; Tradowsky, Carsten: *Integrated circuit, method of generating a layout of an integrated circuit using standard cells, and a standard cell library providing such standard cells.* May 2013

[MBFT13] Myers, James Edward ; Biggs, John Philip ; Flynn, David Walter ; Tradowsky, Carsten: *Apparatus for storing a data value in a retention mode.* May 2013