Eine adaptive Architekturbeschreibung für eingebettete Multicoresysteme

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.-Inform. Thomas Bruckschlögl

geboren in Neuburg an der Donau

Tag der mündlichen Prüfung:
20.07.2017

Hauptreferent: Prof. Dr.-Ing. Dr. h. c. Jürgen Becker
Korreferent: Prof. Dr. sc.techn. Andreas Herkersdorf

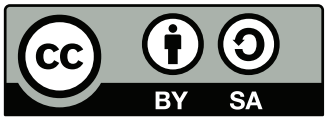
Version: 1.0.0

Stand: 24. August 2017

Eine adaptive Architekturbeschreibung für eingebettete Multicoresysteme

1. Auflage: August 2017

Herausgeber: Thomas Bruckschlögl



Für meine Familie

Abstract

The growing complexity of embedded systems demands for new methodologies for handling the software-view as well as the architecture-view on the system. Regarding development goals, in most cases these two views are not compatible with each other. As a result, architecture description languages and system simulations are used for abstract system description and the evaluation of the system against performance, power and energy consumption or system utilization.

This work presents a new architecture description language that has been developed having in mind the demanding requirements of automated software parallelisation for embedded multicore systems. With its features and supporting tools, the language is able to combine the architectural view with the software-view of embedded systems.

Based on the description a simulation framework has been developed, which implements a novel and adaptive methodology for handling different abstraction layers in simulation and system evaluation.

Zusammenfassung

Die zunehmende Komplexität eingebetteter Systeme verlangt nach neuen Methoden zur Handhabung der vorherrschenden Sichtweisen auf das System. Die Softwaresicht eines Anwendungsentwicklers und die Hardwaresicht der Architektur eines eingebetteten Systems stehen hinsichtlich der Entwicklungsziele oftmals orthogonal zueinander. Architekturbeschreibungssprachen und Systemsimulationen dienen hierbei der abstrakten Beschreibung und der Evaluation der Leistung, des Energieverbrauchs oder der Systemauslastung.

Diese Arbeit stellt eine Architekturbeschreibungssprache vor, welche für die besonderen Anforderungen einer automatisierten Softwareparallelisierung für eingebettete Multicoresysteme konzipiert worden ist. Sie stellt damit ein Bindeglied zwischen den beiden Sichtweisen dar und wird durch eine Reihe von Werkzeugen unterstützt.

Basierend auf dieser Architekturbeschreibungssprache wurde ein Simulationsframework entworfen und implementiert, welches sich durch eine neuartige Adaptivität in Bezug auf unterschiedliche Abstraktionsebenen innerhalb der Simulation als auch der Systemevaluation auszeichnet.

Vorwort

Die vorliegende Arbeit entstand während meiner Zeit am Institut für Technik der Informationsverarbeitung (ITIV) und am FZI Forschungszentrum Informatik. Diese Kombination erlaubte es mir wichtige Einblicke in Grundlagenforschung und Lehre als auch in anwendungsorientierte Aufgaben zu erhalten.

Für diese Möglichkeit möchte ich mich bei meinem Doktorvater Herr Prof. Jürgen Becker bedanken, der zunächst am ITIV und später als Direktor am FZI die Betreuung meiner Arbeit übernommen hat. Darüber hinaus bedanke ich mich bei Herr Prof. Andreas Herkersdorf für die Übernahme des Korreferats und das hilfreiche wissenschaftliche Feedback.

Danken möchte ich auch allen Kollegen am ITIV und am FZI für die gute Zeit, die gute Zusammenarbeit und geführten Diskussionen. Ein besonderer Dank gilt meinen Freunden und meiner Familie, die während der gesamten Zeit immer hinter mir standen, mich unterstützt und motiviert haben.

Karlsruhe, im August 2017

Thomas Bruckschlögl

Inhaltsverzeichnis

Kapitel 1

Einleitung

Während bereits 1927 die ersten Patente zu Transistoren von Julius Edgar Lilienfeld [Lil27] veröffentlicht wurden, dauerte es noch viele Jahre bis es 1958 zur Erfindung des ersten Integrierten Schaltkreises [Kil76] kam. Doch schon kurz darauf veröffentlichte Gordon E. Moore seine These zur Verdopplung der Komplexität integrierter Schaltkreise [Moo65] welche bis heute unter dem Namen "Moore's Law"bekannt ist und bis heute besteht, wenn auch nicht in seiner ursprünglichen Form.

Auch wenn Moore selbst, seine ursprüngliche Prognose einer Verdopplung der Komponenten auf einem integrierten Schaltkreis innerhalb eines Jahres selbst, auf eine Verdopplung alle zwei Jahre angepasst hat, zeigt die Entwicklung der letzten 40 Jahre, dass diese Gesetzmäßigkeit weiterhin gültig ist.

Der in Abbildung 1.1 dargestellte Verlauf der Prozessorentwicklung zeigt deutlich, dass die Anzahl an Transistoren entsprechend Moore's Law kontinuierlich gestiegen ist.

Über die Jahre ist diese Entwicklung allerdings an eine physikalische Grenze gestoßen. Die sogenannte Power-Wall [PH08, Seite 39] verhindert eine weitere Steigerung der Taktfrequenz eines Prozessors, da der Energieverbrauch exponentiell mit der Taktfrequenz wächst. Durch immer kleinere Strukturgrößen konnte dieser Effekt lange Zeit überbrückt werden und dennoch wird keine Steigerung der Taktfrequenz mehr erreicht. Die Leistung von Prozessoren, bis dahin weitestgehend aus einem einzigen Kern bestehend, konnte weiterhin für einige Zeit durch Architekturverbesserungen gesteigert werden. Dennoch ist ersichtlich, dass diese Steigerung deutlich abgenommen hat und deshalb andere Lösungen gesucht wurden.

Mit der Verfügbarkeit vieler Transistoren wurden Prozessoren entwickelt, welche mehrere Prozessorkerne auf einem Chip vereinen und damit eine enorme Leistungssteigerung bei geringfügig höherem Energieverbrauch bieten. Kommerzielle Serverprozessoren bieten bereits heute deutlich über 10 Prozessorkerne an und besitzen damit ein hohes Leistungspotenzial. Selbst aktuelle Smart-

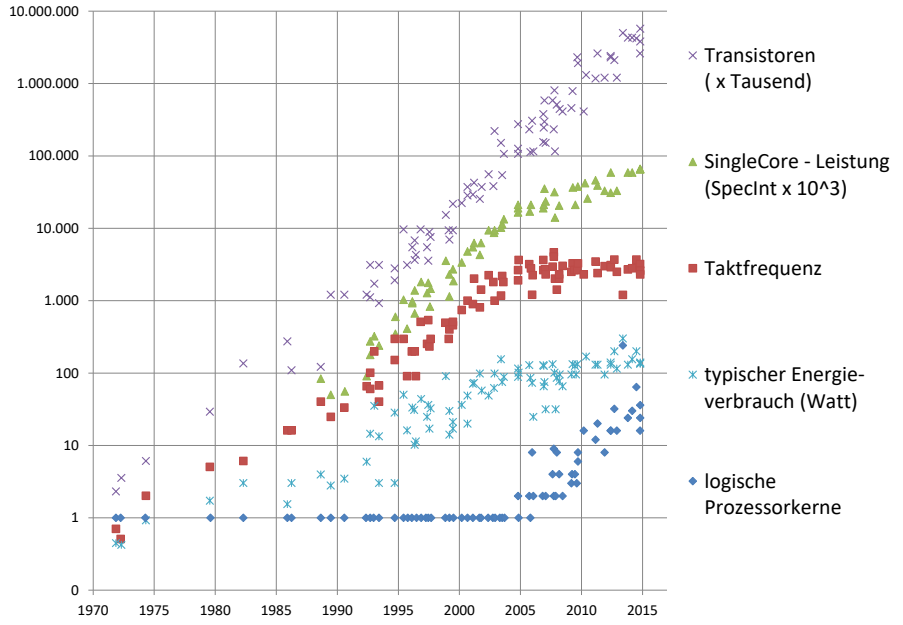


Abbildung 1.1: Darstellung der Entwicklung von Prozessoren von 1975 bis 2015 [Rup16, Moo11]

phones besitzen Prozessoren, welche mit bis zu 8 oder mehr Prozessorkernen ausgestattet sind.

Und auch im Bereich der eingebetteten Systeme, werden immer mehr Mehrkernprozessoren eingesetzt, da sie über ein exzellentes Energie/Leistungsverhältnis verfügen.

Das große Problem, welches auch über 10 Jahre nach der Einführung erster kommerzieller Mehrkernprozessoren besteht ist deren Programmierbarkeit um die zur Verfügung stehenden Leistungsreserven effektiv nutzen zu können.

Während einerseits unterschiedliche Programmiersprachen und Programmiermodelle für die Parallelverarbeitung entwickelt wurden, wuchs andererseits

die Komplexität der Hardwarearchitektur enorm um auf jedem einzelnen Prozessorkern, als auch auf allen Kernen gemeinsam, soviel Leistung wie möglich bereitstellen zu können.

Das Ergebnis ist, dass beide Entwicklungsrichtungen die parallele Softwareentwicklung auf eine Komplexitätsebene heben, welche kaum mehr handhabbar ist. Betrachtet man die Architektur und die Entwicklung paralleler Anwendungen für eingebettete Systeme sowohl aus Hardware als auch aus Nutzer- bzw. Entwicklersicht ergeben sich grundsätzliche Unterschiede.

Während Nutzer einen Algorithmus zur Lösung eines Problems entwickeln möchten stehen auf Hardwareseite eine Vielzahl möglicherweise heterogener Komponenten zur Verfügung.

Während Nutzer eine mathematische Beschreibung und mächtige Hochsprachen zur Softwareentwicklung bevorzugen, müssen auf Hardwareseite Programmiermodelle, Kommunikationsstrukturen, verfügbare Datentypen und Spezialinstruktionen sowie weitere Hardwareeigenschaften berücksichtigt werden.

Während ein Entwickler seine Software möglichst unabhängig von der Art des Parallelismus und der Verarbeitung beschreiben möchte, implementieren Prozessoren und Mehrkernprozessoren verschiedenste sequentielle als auch parallele Architekturen wie RISC, CISC, VLIW, SIMD, MIMD, MISD.

Die heutige Compilertechnik ist zwar schon sehr weit fortgeschritten, erfordert es aber für nahezu jede Architektur einen eigenen dedizierten Compiler zu entwickeln, der die spezifischen Hardwareeigenschaften kapselt und dem Softwareentwickler zur Verfügung stellt.

Dabei den Überblick über die schier unzähligen Realisierungsalternativen zu behalten wird für Anwendungsentwickler dedizierter eingebetteter Systeme immer schwieriger.

Aus diesem Grund wurde das EU Forschungsprojekt ALgorithm parallelization for Multicore Architectures (ALMA) [BSO+12] ins Leben gerufen, welches sich zum Ziel gesetzt hat die Programmierung von Mehrkernprozessoren in eingebetteten Systemen zu vereinfachen. Dazu sollte ein Framework entwickelt werden, welches eine automatisierte Parallelisierung mathematischer Algorithmen aus der Sprache Scilab [Sci13] für Mehrkernprozessoren anbietet.

Um die Lücke der oben beschriebenen Sichtweisen eines Algorithmenentwicklers und einer Hardwarearchitektur zu schließen sollte eine Architecture Description Language (ADL) zum Einsatz kommen, welche in der Lage ist, die für die Parallelisierung notwendigen Hardwareeigenschaften aufzunehmen und den Werkzeugen des ALMA Framework zur Verfügung zu stellen.

Dadurch wird es einerseits dem Entwickler ermöglicht sich auf die Entwicklung des Algorithmus zu konzentrieren ohne sich größere Gedanken über eine mögliche Hardwarearchitektur machen zu müssen. Andererseits erlaubt die Verwendung einer ADL die Entwicklung hardwareunabhängiger Parallelisierungswerkzeuge, welche für eine Vielzahl an unterschiedlichen Komponenten eingesetzt werden können.

Als weiteres Kernelement des Frameworks soll ein Architektur- und Softwaresimulator eingesetzt werden, der eine direkte Evaluation des parallelisierten Algorithmus ermöglicht um das Ergebnis iterativ optimieren zu können.

1.1 Ziele und Beitrag der Arbeit

Aus diesem Umfeld lassen sich die grundsätzlichen Ziele dieser Arbeit ableiten, welche den Maßstab für die zu entwickelnde Architekturbeschreibungssprache und die zugehörigen Werkzeuge bieten:

- Das Hauptziel der ADL ist das Schließen der Lücke zwischen Nutzer- und Hardwaresicht, in dem notwendige Hardwareeigenschaften modelliert und den Parallelisierungswerkzeugen zur Verfügung gestellt werden.
- Die ADL muss die notwendige Flexibilität bieten, um einen durchgängigen Prozess von Systembeschreibung, Parallelisierung, Simulation, Evaluation und Optimierung darstellen zu können.
- Mit dem Hintergrund der automatisierten Parallelisierung ist es notwendig einerseits strukturelle Systeminformationen als auch Verhaltensinformationen der Hardwarearchitektur beschreiben zu können.
- Um die iterative Optimierung der Parallelisierungswerkzeuge zu unterstützen müssen Simulationszeiten so kurz und die gewonnenen Informationen so genau wie möglich erhoben werden.

In dieser Arbeit wird deshalb eine flexible Architekturbeschreibungssprache vorgestellt, welche es erlaubt sowohl Struktur- als auch Verhaltensinformationen komplexer heterogener Multicorearchitekturen zu modellieren, welche sich sowohl an der Nutzersichtweise als auch an der Hardwaresichtweise orientiert. Dazu wurde ein ADL-Compiler entwickelt, der die beschriebene Architektur analysiert, die zur Verfügung gestellten Informationen extrahiert und diese an die Parallelisierungswerkzeuge weitergibt. Des Weiteren erzeugt der ADL-Compiler eine, auf einer Komponentenbibliothek und einem Simulationsframework basierende, Simulationsumgebung zur Simulation, Evaluation und parametrierbaren Entwurfsraumexploration von Multicoresystemen.

Um die durchgängige Flexibilität des Frameworks zu ermöglichen wurde für die ADL ein spezielles Hierarchiekonzept entwickelt, welches sowohl die Beschreibung von alternativen Implementierungen, hierarchischen Implementierungen als auch die Beschreibung unterschiedlicher Abstraktionsebenen einer Komponente zulässt. Besonders durch die Verwendung der hierarchischen

Modellierung unterschiedlicher Abstraktionsebenen kann dabei eine deutliche Reduktion der Simulationszeiten für iterative Parallelisierungsansätze erzielt werden ohne nennenswerte Verluste in der Qualität der parallelisierten Anwendung hinnehmen zu müssen.

Evaluiert wurde deshalb die Fähigkeit zum Einsatz in der Entwurfsraumexploration sowie die adaptive Simulation mit Komponenten auf unterschiedlichen Abstraktions- und Genauigkeitsebenen, sowie deren Auswirkungen sowohl auf die Entwurfsraumexploration als auch auf die automatisierte iterative Parallelisierung von Algorithmen.

1.2 Aufbau der Arbeit

Diese Arbeit verfolgt eine Thematische Gliederung der Kapitel, so dass Grundlagen und weitere thematisch verwandte Arbeiten in die entsprechenden Kapitel integriert wurden.

Kapitel 2 gibt zunächst einen Einblick in das EU-Forschungsprojekt ALMA, in dessen Rahmen diese Arbeit entstanden ist.

Kapitel 3 beschreibt die Anforderungen und Ziele des Frameworks und gibt einen Überblick über die einzelnen Komponenten und der Zusammenhänge untereinander. Dazu wird jeweils eine kurze Zusammenfassung der Beschreibungssprache und des zugehörigen Compilers, der Simulationsumgebung sowie des Profilers wiedergegeben.

In Kapitel 4 wird somit die Umsetzung und der Aufbau der Architekturbeschreibungssprache beschrieben, sowie die in der Sprache realisierten Hierarchiekonzepte, die Sichtweisen auf Hardware und Software innerhalb der ADL und parametrierbaren Beschreibungsmöglichkeiten erläutert.

Das Kapitel 5 beschreibt die Realisierung des ADL-Compilers zum Übersetzen der Architekturbeschreibungssprache in eine Zwischendarstellung zur

Verwendung im Simulator. Dabei wird auch der vollständige Ablauf sowie die implementierten Funktionen im Übersetzungsvorgang detailliert beschrieben.

Kapitel 6 beschreibt nun die Simulationsumgebung, die implementierten Simulationsklassen und Konfigurationsmöglichkeiten. Dazu beschreibt es außerdem die Methodik zur dynamischen Erstellung einer Simulationsumgebung aus der Architekturbeschreibungssprache.

Als nächstes beschreibt Kapitel 7 die Laufzeitanalysefunktionen von Instrumentierungsfunktionen über die Profiling Klassen und deren Integration in das Simulationsframework.

Anschließend gibt Kapitel 8 einen Überblick über die durch den Einsatz der Beschreibungssprache erzielten Ergebnisse im Rahmen der Entwurfsraumexploration, der Performanzanalyse, sowie der Spracheigenschaften im Hinblick auf die hierarchische Beschreibung und die damit verbundenen Möglichkeiten zur Steuerung des Simulationsframework.

Das Kapitel 9 fasst die Arbeit abschließend zusammen.

Kapitel 2

ALMA

Dieses Kapitel beschreibt das EU Forschungsprojekt ALMA: ALgorithm parallelization for Multicore Architectures, welches das Ziel verfolgt hat Algorithmen aus Hochsprachen automatisiert zu parallelisieren und direkt für eine eingebettete Multicorearchitektur zu kompilieren. Es werden die Ziele des Projekts sowie der Lösungsansatz zur automatisierten Parallelisierung dargestellt. Dazu gehört eine kurze Beschreibung der Komponenten und der verwendeten Kahrisma Architektur.

Das ALMA [GAV+12] [SOB+13] Projekt befasst sich mit einem neuartigen Ansatz für den Entwicklungsprozess für Hardware und Softwaresysteme. Das Ziel des Projektes ist es eine Entwicklungsumgebung für die automatisierte Parallelisierung sequentieller Algorithmen bereitzustellen, welche die Komplexität eines zugrunde liegenden Multiprozessorsystems vor dem Endnutzer versteckt, indem Algorithmen, welche in SciLab [Sci13] geschrieben sind, kompiliert und für rekonfigurierbare Multiprozessorsysteme optimiert werden. Als Ergebnis erzeugt die ALMA-Toolchain ausführbare Dateien für eine Multicorehardware als auch eine Simulationsumgebung.

Eines der Designziele des ALMA Projekts ist die Hardwareunabhängigkeit der Toolchain, welche nicht dediziert für eine Zielarchitektur implementiert werden, sondern eine Vielzahl an Architekturen unterstützen soll. Obwohl die Toolchain generisch ist, muss das Ergebnis, also die parallelisierte Anwendung, für die zugrundeliegende Architektur optimiert werden, weshalb eine detaillierte Kenntnis über die Hardwarearchitektur zwingend notwendig ist.

Eine Schlüsselkomponente für eine effektive Parallelisierung ist daher eine abstrakte Beschreibung der Architektur sowie eine Simulationsumgebung mit der die erstellte Anwendung evaluiert werden kann. Nur so können iterative Verbesserungen an der Anwendung durchgeführt, sowie eine leistungsfähige und effiziente Parallelisierung gefunden werden.

Innerhalb des ALMA Projekts wurden zwei Beispielarchitekturen verwendet. Eine der beiden ist die KArlsruhe's Hypermorphic Reconfigurable-Instruction-Set Multi-grained-Array (KAHRISMA) Architektur [RKB+10], die zweite ist eine Multicore Architektur der niederländischen Firma Recore [Rec17].

Das Ergebnis dieses Forschungsprojektes wird im StartUp emmtrix Technologies [SRO+16] weiter verfolgt.

2.1 ALMA Software Framework

Das ALMA Software Framework besteht aus mehreren Gruppen an Werkzeugen und Software-Bibliotheken, wie sie in Abbildung 2.1 dargestellt sind. Der linke Teil dieses Diagramms beschreibt dabei die Compiler-Toolchain mit ihren Verarbeitungsschritten um aus einer SciLab Anwendung eine für Multicore Architekturen optimierte ausführbare Datei zu erstellen. Diese kann anschließend direkt auf der Hardware als auch innerhalb des Systemsimulators ausgeführt werden.

Dabei wird zunächst die SciLab Anwendung durch eine Vorverarbeitung in eine Zwischendarstellung in C überführt. In diesem Schritt werden bereits hilfreiche Informationen extrahiert, die für die spätere Parallelisierung der Anwendung genutzt werden können. Dazu gehören beispielsweise Größen- und Datentypinformationen von Matrizen und Variablen oder Lebensdauer und Sichtbarkeiten von Variablen. Anschließend wird das C-Programm mit Hilfe des Generic Compiler Suite (GeCoS) Framework [FYEM+13] in einen Hierarchical Task Graph (HTG) überführt. Auf diesem lassen sich feingranulare als auch grobgranulare Transformationen zur Parallelisierung durchführen um einerseits eine optimierte Code-Ausführung innerhalb einer Verarbeitungseinheit (feingranular) als auch eine Verteilung des Programms auf mehrere Prozessoren (grobgranular) zu erreichen.

Das Ergebnis ist wiederum ein C-Programm, welches nun parallelisiert wurde und von einem architekturspezifischen Compiler in eine Ausführbare Datei übersetzt wird. Für die KAHRISMA Architektur baut dieser Compiler auf dem Low Level Virtual Machine (LLVM) [Lat13b] Compiler Framework auf und verwendet darüber hinaus angepasste Werkzeuge aus der Gnu Compiler Collection [C. 10]. Dieser wurde in der Arbeit von Timo Stripf [Str13] entwickelt und erforscht. Dabei wurden die notwendigen Funktionen zur Unterstützung der Laufzeitrekonfiguration auf unterschiedliche Instruction Set Architectures (ISAs) in das Back-End des LLVM Compiler Framework implementiert. Der

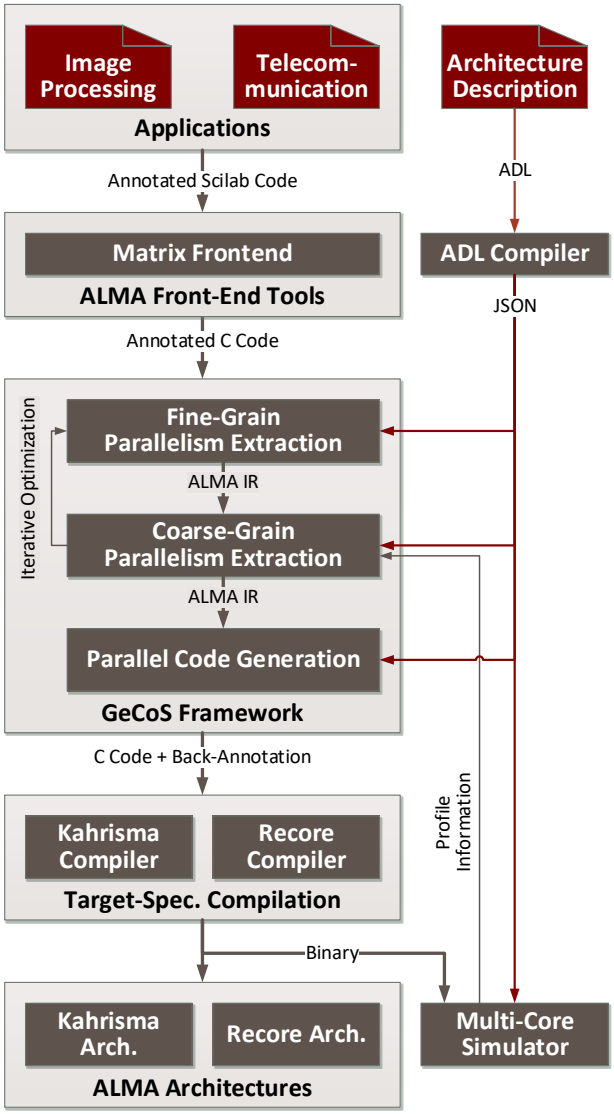


Abbildung 2.1: Übersicht über die ALMA Toolchain [BSO+12]

KAHRISMA Compiler erzeugt ausführbare Dateien im Executable and Linkable Format (ELF) [Com95] Dateiformat, welche mehrere ISAs aus KAHRISMA unterstützen und damit die Laufzeitrekonfiguration unterstützen.

Um die architekturspezifischen Eigenschaften der Hardware optimal im Parallelisierungsprozess nutzen zu können, wird in der ALMA-Werkzeugkette eine ADL eingesetzt. Diese Sprache muss in der Lage sein, die notwendigen und hilfreichen Informationen der Architektur abzubilden, gleichzeitig aber die Flexibilität bieten um in verschiedenen Werkzeugen sinnvoll eingesetzt zu werden. Nur durch den Einsatz einer derartigen Architekturbeschreibungssprache kann das gesamte Parallelisierungsframework architekturunabhängig, erweiterbar und flexibel gestaltet werden um auch zukünftigen Anforderungen und Architekturen gerecht zu werden.

Die ALMA Werkzeugkette verwendet nun einen iterativen Ansatz mit dem die Parallelisierung der Anwendung schrittweise optimiert werden kann. Deswegen muss die Architekturbeschreibungssprache die Grundlage für eine Systemweite Simulationsumgebung darstellen, mit deren ein Ergebnis der Softwareparallelisierung evaluiert und iterativ optimiert werden kann. Hier spielt der Trade-Off zwischen feingranularer und grobgranularer Parallelisierung eine für die Performanz der Anwendung entscheidende Rolle. Dazu wird ein flexibles Simulationsframework benötigt, welches zusammen mit der ADL eine ausführbare Simulationsumgebung für die parallelisierte Anwendung bereitstellt. In dieser Umgebung müssen Evaluationswerkzeuge wie Applikationsprofiling oder Architekturtracing zur Verfügung stehen, welche auch die besonderen Anforderungen an die Evaluation der einzelnen Parallelisierungsschritte ermöglicht. Die genaue Abbildung der Compilertransformationen auf die Profilingfunktionen ist daher zwingend erforderlich um diese Informationen mit einer Feedback-Schleife auch wieder verarbeiten zu können.

2.2 Die KAHRISMA Hardwarearchitektur

KAHRISMA ist eine flexible Very Long Instruction Word (VLIW) Architektur, welche aus einer Menge an Ausführungseinheiten bzw. Prozessorelementen, genannt Encapsulated Datapath Element (EDPE), besteht. Diese sind zur Laufzeit rekonfigurierbar und können so verschieden komplexe Architekturen oder Spezialinstruktionen realisieren. Jede der möglichen Architekturen bietet seine Vorteile für bestimmte Anwendungen, Applikationen und Einsatzzwecke. Ein KAHRISMA System, wie es in Abbildung 2.2 dargestellt ist, kann aus mehreren Dutzend dieser EDPEs bestehen, wodurch es zu einem komplexen Multicoresystem wird.

EDPEs können zu Konfigurationen zusammengefasst werden um als eine größere einzelne Ausführungseinheit zu arbeiten, um den Instruction Level Parallelism (ILP) innerhalb einer Anwendung besser ausnutzen zu können. Eine derartige Konfiguration besteht aus bis zu vier KAHRISMA Prozessorkernen welche auf dem Run-Time Scalable Issue-Width (RSIW) Konzept basieren. Jeder Prozessorkern implementiert einen zweifach-VLIW (RSIW2) Prozessor. Bei einer Kombination von vier Prozessoren (RSIW2222) wird also ein achtfach-VLIW Prozessor erstellt [KSJB11]. Eine weitere Möglichkeit der Konfiguration besteht darin aus den EDPEs Spezialinstruktionen zu konstruieren um die Ausführung von dedizierten Anwendungen zu beschleunigen.

Zur Nutzung des ILP kann die Architektur mit Ihrer Vielzahl an Prozessoren auch als klassische Multicorearchitektur genutzt werden. Dadurch können Anwendungen auch vom Thread Level Parallelism (TLP) profitieren, der die parallele Ausführung mehrerer unabhängiger oder zusammengehöriger Anwendungen erlaubt.

Dadurch kann die Architektur auf aktuelle Anforderungen und Belastungen reagieren oder auf spezielle Ziele, wie kurze Reaktionszeiten oder einen maximierten Datendurchsatz hin optimiert werden.

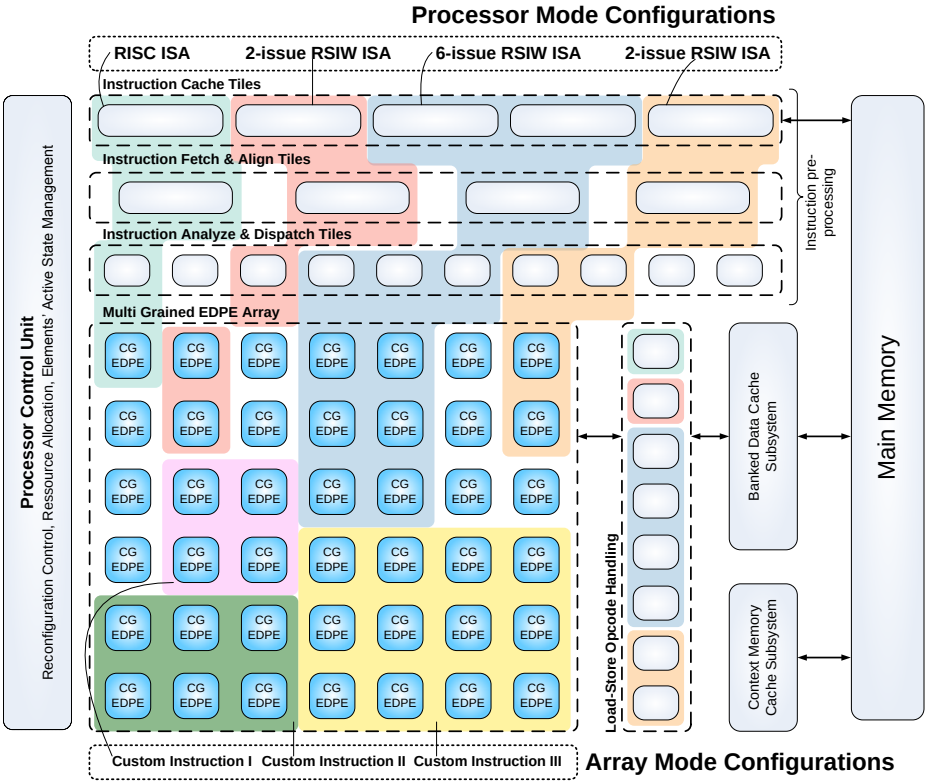


Abbildung 2.2: Übersicht über die KAHRISMA Architektur in der unterschiedliche Konfigurationen aktiv sind. Die Ausführungseinheiten sind in der linken unteren Ecke zu sehen. Bildquelle [RKB+10]

Kapitel 3

Ein Framework für die Evaluation eingebetteter Systeme

Dieses Kapitel beschreibt den Aufbau und das Zusammenspiel der einzelnen Komponenten des in dieser Arbeit entwickelten Frameworks. Dabei wird jeweils ein Überblick über die Ziele des Frameworks und die Anforderungen an das Framework definiert.

3.1 Ziele

Der grundlegende Zweck des Beschreibungsframeworks ist die Unterstützung unterschiedlicher Werkzeuge für die Softwareentwicklung. Ein besonderer Bereich hierbei ist die Entwicklung von parallelen Anwendungen. Dabei ist eine sehr genaue Kenntnis der zugrundeliegenden Hardware notwendig um eine effiziente Berechnung und Ausführung zu erreichen.

Daraus ergibt sich das primäre Ziel der Architekturbeschreibung eine abstrakte Informationsquelle über die Hardwarearchitektur zur Verfügung zu stellen um den Entwicklungsprozess paralleler Programme sowie die automatisierte Parallelisierung sequentieller Algorithmen für eingebettete Systeme zu unterstützen.

Die im Rahmen des ALMA Projektes [BSO+12] entwickelten Softwarewerkzeuge zur automatisierten Parallelisierung von Algorithmen nutzen diese Möglichkeit um hardwareunabhängig arbeiten zu können. Die über die Hardwarearchitektur notwendigen Kenntnisse können so aus der Architekturbeschreibung extrahiert und in den Entwicklungsprozess eingebunden werden. Somit kann eine Vielzahl unterschiedlicher Hardwarearchitekturen unterstützt werden, was allerdings eine hohe Flexibilität und Adaptivität des Frameworks erfordert.

Ein weiteres Ziel des Frameworks ergibt sich aus der allgemeinen Anforderung an Parallelisierungswerkzeuge möglichst effizienten und schnell ausführbaren Code zu erzeugen. Um dieses Ziel erreichen zu können sind Messungen der Leistung des Systems und des erzeugten Quellcodes notwendig. Deshalb ist ein Ziel dieses Framework die Integration einer Simulationsumgebung mit der das beschriebene System als auch die erzeugte parallele Software simuliert und evaluiert werden kann.

Mit der Verknüpfung der ADL mit Architekturanalysen und einer Simulationsumgebung mit integriertem Profiler ist es wichtig, dass die ADL ein durchgän-

giges Konzept verfolgt um eine in allen Werkzeugen konsistente und gemeinsame Beschreibung des Systems zu ermöglichen.

3.2 Anforderungen

Ausgehend von den Zielen, ergeben sich die Anforderungen an das Beschreibungsframework.

Analysierbarkeit Die Architekturbeschreibung muss formalisiert und analysierbar sein, damit Sie für die Verwendung in Entwicklungswerkzeugen für heterogene parallele Systeme eingesetzt werden kann. Andernfalls können Compiler und Scheduler keine Entscheidungen treffen, die maßgeblich für die Effizienz eines parallelen heterogenen Systems verantwortlich sind. Darüber hinaus wäre damit keine Systemerzeugung in Simulationssystemen möglich.

Erweiterbarkeit und Adaptierbarkeit Um der ständigen Weiterentwicklung eingebetteter Systeme Rechnung tragen zu können, muss eine Architekturbeschreibungssprache entsprechend erweiterbar sein. Dies betrifft sowohl die Unterstützung neuster Technologien und Komponenten als auch die Beschreibungsmuster an sich. Die Adaption der einzelnen Werkzeuge auf ein beschriebenes System spielt dabei eine sehr wichtige Rolle in der Konsistenz des Frameworks.

Rekonfigurierbarkeit Da Rekonfigurierbare Systeme einen besonderen Stellenwert bei der Entwicklung heterogener Systeme genießen und manche davon die Eigenschaft besitzen ihre Funktion, Struktur und ihr Verhalten zur Laufzeit zu ändern, soll die Architekturbeschreibung sowie das Framework diese Möglichkeiten schon in der Design-Phase mit einbeziehen.

Reduzierung von Komplexität Im Vergleich zu anderen Beschreibungssprachen, soll diese Sprache die Möglichkeit bieten auf einfache Art und Weise, reguläre als auch irreguläre Systeme zu beschreiben um deren Komplexität handhabbar zu gestalten.

Mixed-Level-Beschreibung Die Beschreibung soll in der Lage sein, sowohl Verhaltens- als auch Strukturinformationen zu beinhalten. Diese Informationen sollen hierarchisch beschrieben und verwendbar sein. Dies soll eine Flexibilität bei der Auswahl verschiedener Parameter bei Beschreibung, Analyse und Simulation ermöglichen. Besonders die Simulation soll die Ausführung unterschiedlicher Abstraktionsebenen unterstützen.

3.3 Frameworks für die Entwurfsraumexploration

SESAM [VGS+10, PEP06] ist ein Entwurfsraumexplorationsframework, welches es erlaubt Architekturen als auch Anwendungen sowie Performance Modellierungen zu erstellen. Allerdings werden Architektur und Anwendung getrennt voneinander simuliert, da für die Architektur eine rein strukturelle Beschreibung verwendet wird. Dadurch ist eine binärkompatible Simulation der Anwendung nicht möglich, was SESAM für den Einsatz innerhalb eines iterativen Parallelisierungsansatz ungeeignet macht.

Hermes Multi Processor System (HeMPS) [CDCM09] ist ein Framework zur Beschreibung von homogenen Multiprozessorsystemen mit Network-on-Chip Kommunikationssystemen. Zur Verfügung steht eine NoC Infrastruktur und auf MIPS Plasma basierende Prozessorsimulatoren. Die Beschreibung des Systems steht allerdings keine Architekturbeschreibungssprache zur Verfügung sondern ein graphischer Editor mit der die Größe eines HeMPS Array definiert und Software-Tasks in Form von Task Graphen modelliert werden können. HeMPS unterstützt nun die dynamische Allokation von Tasks auf das

Multiprozessorsystem und deren Simulation mit Hilfe von zyklenakkuraten SystemC Modellen.

Angiolini et. al. [ACL+06] vereinen nun die CAD Werkzeuge von LISATek, die LISA Architekturbeschreibungssprache mit MPARM, einer Multiprozessorumgebung auf SystemC Basis. Beide Sprachen bzw. Systeme arbeiten auf Bit- und Signal-Akkuraten Simulationsklassen, welche entweder, im Fall von MPARM, direkt in SystemC beschrieben sind, oder die Beschreibung von Application-Specific Instruction Set Processors (ASIPs) in LISA erlauben, aus denen sich SystemC Modelle erzeugen lassen. Weitere Abstraktionsebenen sind hier allerdings nicht möglich. Die Beschreibung mit LISA wird im Abschnitt 4.3 beschrieben.

ALMA ähnliche Parallelisierungswerkzeuge sind MAPS [CCS+08] oder MPA [BBWA09]. Der Unterschied zwischen ALMA, MAPS und MPA besteht darin, dass MAPS und MPA eine Applikation in C-Code verwenden und daher deutlich eingeschränkter in der Anwendungsprogrammierung sind, als wenn ein SciLab Algorithmus verwendet wird. Des Weiteren verwendet MAPS analytische Verfahren für die Performance Evaluation anstelle einer Simulation.

Mit der Problematik von ILP und Cache-Optimierungen, welche oftmals gegensätzliche Auswirkungen auf die Leistungssteigerung eines Systems besitzen, befasst sich die Arbeit von Barthou et. al. [BDC+07]. Vorgeschlagen wird eine feingranulare Analyse des Programmcodes, um automatisch gute Kompromisse zwischen beiden Optimierungsmöglichkeiten zu erzielen und damit die Gesamtleistung des Systems nochmals zu steigern. Es fehlen allerdings grobgranulare Ansätze, welche Anwendungen auf Multicoreprozessoren verteilen können.

In [DZK+13] hingegen wird eine Lösung vorgestellt, welche sich besonders um das Mapping von Threads auf Multicore Systeme und damit die Ausführung auf verschiedenen Kernen kümmert. Dabei wird ein neuartiger Ansatz verfolgt, welcher eine effektive Wiederverwendung von Daten innerhalb einer geteilten Cache Struktur eines Multicoreprozessors ermöglicht und damit

die Belastung der Kommunikations- und Speichersysteme verringert. Die Beschränkung hierbei liegt darin, dass nur Schleifen ohne weitere Abhängigkeiten im Daten- als auch Kontrollfluss betrachtet werden können und dies somit für den Einsatz spezialisierter eingebetteter Systeme kaum in Frage kommt.

Um derartige Abhängigkeiten zu betrachten, wird in [HJB+13] ein dynamisches Framework beschrieben, welches zur Laufzeit eines Programms eine Abhängigkeitsanalyse durchführt und einen dynamischen Scheduler für das aufrufen weiterer Schleifeniterationen aufbaut. Basierend auf einem Basisscheduler, welcher dem Programm als Laufzeitsystem hinzugefügt werden muss, können so dynamisch individuelle Schedules ermittelt werden, welche sich den aktuellen Gegebenheiten anpassen um das Programm weiter zu beschleunigen und die Parallelität auf mehreren Kernen besser auszunutzen. Auch hier liegt die Beschränkung darin, dass nur Schleifeniterationen betrachtet werden können und im Vergleich zu ALMA keine tiefgreifende Analyse und Partitionierung allgemeiner Algorithmen stattfindet.

3.4 Aufbau und Komponenten

Das Framework besteht aus mehreren Komponenten, die jeweils aufeinander Aufbauen oder unterschiedliche Teilergebnisse erzeugen. Die Grundlage stellt dabei die Architekturbeschreibungssprache dar, deren Zusammenspiel mit den anderen Komponenten in Abbildung 3.1 dargestellt ist.

Des Weiteren existiert ein ADL-Compiler um die Architekturbeschreibung zu übersetzen und weiteren Werkzeugen zur Verfügung zu stellen. Als weitere Werkzeuge stehen Analysefunktionen zur Verfügung die in den Compiler integriert sind und auf dessen Ergebnissen ausgeführt werden können. Die Ergebnisse der Analysewerkzeuge können dann wiederum Entwicklungswerkzeugen wie der ALMA-Toolchain oder GeCoS übergeben werden. Direkt auf

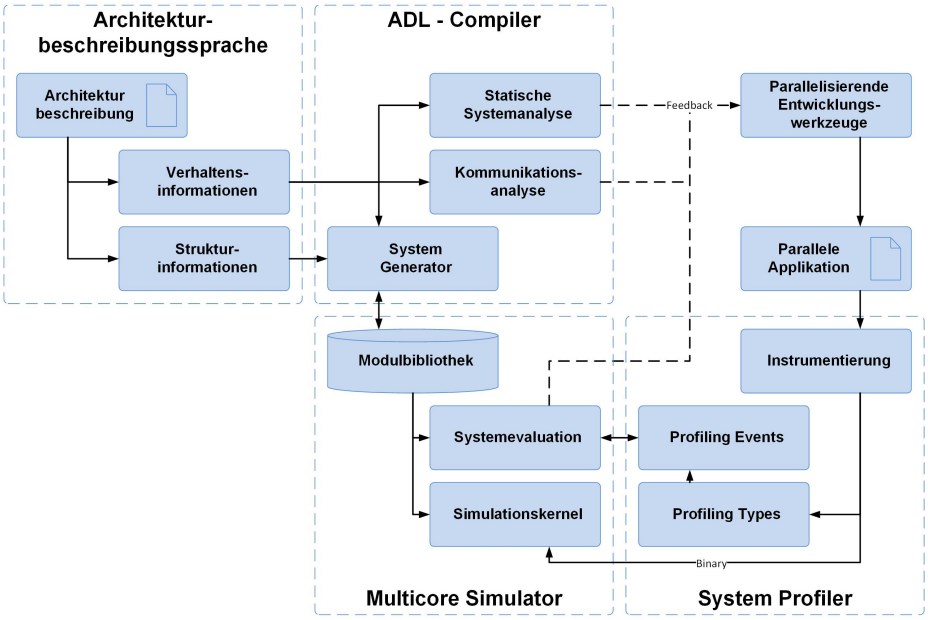


Abbildung 3.1: Komponenten und Werkzeuge im Framework und deren Zusammenhang mit der ADL

den Ergebnissen des Compilers arbeitet auch das Simulationsframework. Dieses generiert aus einer Bibliothek an Simulationsklassen und der Architekturbeschreibung eine simulierbare Systemumgebung zur Ausführung paralleler Software.

Das Simulationsframework unterstützt dabei eine Vielzahl an unterschiedlichen Simulationsmethoden von funktional bis zyklenakkurat oder von abstrakter Softwaresimulation bis zur binärkompatiblen Simulation. Welche Simulationsmethode zum Einsatz kommt wird durch die Simulationsklasse bestimmt. Das Simulationsframework gibt den Rahmen vor und steuert die Simulation der einzelnen Module.

Als weiteres wichtiges Werkzeug wurde für das Simulationsframework ein flexibler und adaptiver Profiler entworfen, dessen Funktionen als Bibliotheksklassen dem Simulationsframework zur Verfügung gestellt werden. Durch die Modellierung eines Systems in der ADL kann das Simulationsframework anschließend die notwendigen Klassen entsprechend der instanziierten Module auswählen und zur Verfügung stellen. Dadurch kann sichergestellt werden, dass Profilingfunktionen je nach instanziiertem Modul verfügbar sind.

3.4.1 Architekturbeschreibungssprache

Die ADL soll die Beschreibung von Hardwarearchitekturen ermöglichen, einfach erlernbar, flexibel und leicht erweiterbar sein. Aus diesem Grund bedient sich die ADL den aus typischen Hardwarebeschreibungssprachen, wie Very High Speed Integrated Circuit Hardware Description Language (VHDL) [IEE09] oder Verilog [IEE06], bekannten Konzepten aus Modulen und Instanzen, Ports und Interfaces, sowie den Verbindungen daraus.

Die aus der Hardwarebeschreibung bekannten Elemente bilden das strukturelle Gerüst der Architekturbeschreibung. Diese wird wiederum mit Verhaltensbeschreibungen annotiert. Innerhalb des in Abbildung 3.2 dargestellten Y-Diagramms nach Gajski und Kuhn befindet sich die Architekturbeschreibungssprache deshalb auf den oberen markierten Ebenen.

Das grundlegende Element der Architekturbeschreibungssprache stellt das Modul dar. Ein Modul modelliert eine für sich eigenständige und abgeschlossene Komponente einer Hardwarearchitektur wie beispielsweise einen Prozessor oder einen Speicher. Als Strukturkomponente beschreibt es den Aufbau einer solchen Systemkomponente. Ports geben die Kommunikationspunkte, also Eingabe und Ausgabemöglichkeiten an. Ports werden mit einem Interface verknüpft, wodurch ein Verbindungstyp definiert wird. Um ein System

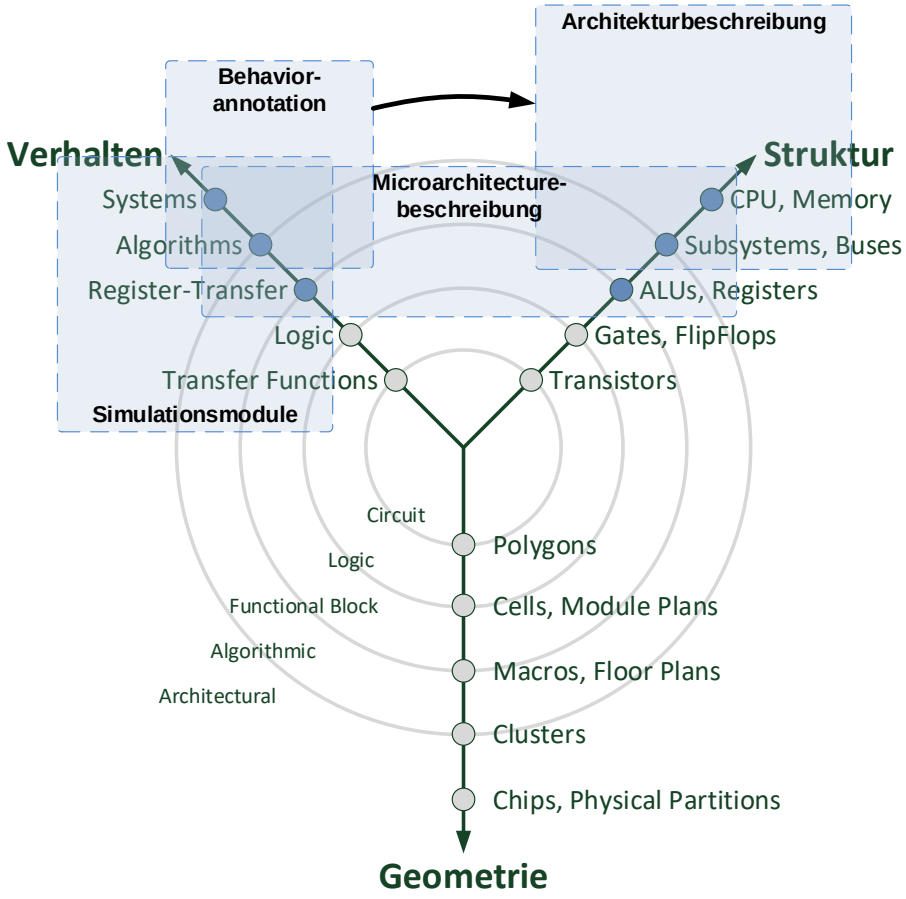


Abbildung 3.2: Einordnung der ADL im Y-Diagramm nach Gaisky und Kuhn

zu beschreiben werden nun Module instanziiert und diese Instanzen anschließend miteinander verbunden. Die Verbindung der Ports impliziert die Verwendung eines Interfaces, welches im Modulport referenziert wurde. So können direkt inkompatible Module, bzw. Ports erkannt werden, wenn die Instanzports unterschiedliche Interfaces verwenden. Die Modulbeschreibung unter-

stützt die Verwendung von Hierarchieebenen um Module aus Teilkomponenten aufzubauen oder um Simulationsalternativen zu beschreiben. In diesem Hierarchiekonzept liegt die Besonderheit darin, dass jede Ebene in sich abgeschlossen ist und damit auch jede Hierarchieebene eine Abstraktionsebene darstellen kann.

Als weiteres grundlegendes Element dieser Architekturbeschreibungssprache wird eine Verhaltensbeschreibung, genannt **Behavior**, verwendet. Ein **Behavior** annotiert Module und Interfaces, wodurch die Verhaltensinformationen auch auf die Instanzen übertragen werden können. Um eine klare Analysierbarkeit der Sprache zu erlauben werden Verhaltensbeschreibungen nicht in einer Programmiersprache verfasst, sondern als klar definierte Eigenschaften innerhalb der ADL angegeben. Um das funktionale Verhalten eines Moduls zu beschreiben, können die Simulationsklassen und damit eine Programmiersprache verwendet werden. Diese können entsprechend dem Hierarchiekonzept auf mehreren Abstraktionsebenen implementiert werden, um so das Verhalten auf dem gewünschten Detailgrad zu implementieren.

Um den Einsatz in Compiler-Werkzeugen und Parallelisierungswerkzeugen zu ermöglichen umfasst die Architekturbeschreibungssprache ein spezielles umfangreicheres **Behavior**, mit dem Prozessorinformationen abbildbar sind. Dazu gehören die vorhandenen internen Ressourcen und Ausführungseinheiten, welche nicht als eigenständige Module modelliert werden müssen. Aber auch Befehlssatzinformationen inklusive einer semantischen Beschreibung der Instruktionen welche auf den passenden Compiler abgebildet werden können sind Teil dieser Beschreibung. Besonders wichtig für feingranulare Parallelisierungsschritte sind Performanzinformationen, Belegungsinformationen von Ressourcen, Latenz- und Befehlsdurchsatzinformationen der Ausführungseinheiten. Auf grobgranularer Ebene ist die Struktur eines Systems sowie die Kommunikationsinfrastruktur entscheidend. Struktur- als auch Verhaltensinformationen können extrahiert und analysiert werden. Dadurch können detaillierte Informationen über Nachrichten und Kommunikationswege zwischen Prozessorkernen ermittelt werden.

Als weiteres Element werden Konfigurationen eingesetzt. Eine Konfiguration beschreibt dabei eine Untermenge der in einer Systembeschreibung verfügbaren Instanzen und Verbindungen. Konfigurationen können dafür verwendet werden rekonfigurierbare Systeme zu beschreiben. Innerhalb eines Systems können mehrere sich überlappende Konfigurationen beschrieben werden. Allerdings muss sichergestellt sein, dass diese Konfigurationen nicht gleichzeitig geladen werden. Auf Analyse- und Beschreibungsebene kann dies durch den Compiler abgedeckt werden. Innerhalb der Simulation müssen dafür entsprechende Laufzeitumgebungen oder Betriebssystemfunktionen in die Simulationsmodule implementiert werden.

3.4.2 ADL-Compiler

Um die ADL effektiv verarbeiten zu können, beinhaltet das Framework einen Compiler, welcher die Sprache in eine Zwischendarstellung übersetzt. Innerhalb des Simulationsframework wird aus dieser Zwischendarstellung die Simulationsumgebung erzeugt.

Der Compiler wird als Multi-Pass Compiler [ALSU06] implementiert. Obwohl nicht direkt nach jedem Pass Ausgabedateien erzeugt werden, ist es dennoch nicht möglich die ADL mit einem einzigen Compiler-Pass zu übersetzten. Dies liegt unter anderem an der notwendigen Weiterleitung von Verbindungsinformationen bei der Verwendung von Implementierungen. Diese lassen sich nur iterativ auflösen, nachdem alle Module zunächst einmalig geparst worden sind. In diesem Fall wird durch die einzelnen Compilerpasses ein Ergebnis erzeugt, welches für weitere Analysen als auch für die Simulationsumgebung zur Verfügung gestellt wird.

Direkt im Compiler werden auch statische und generische Analysewerkzeuge als Compiler-Pass integriert. Die Analyse-Passes folgen nun den Compiler-Passes und fügen dem Compiler-Ergebnis jeweils mehr Informationen hinzu.

Durch das Pass Konzept kann sichergestellt werden, dass die Analysefunktionen auch durchgeführt werden können, da alle auf der gleichen Datenbasis durchgeführt werden. Jede Analysefunktion kann dabei einen eigenen Ausgabedatensatz definieren. Die Ausgabe kann an den Compiler übergeben werden, so dass ein gleichmäßiges Ausgabeformat verwendet wird. Dies erleichtert die anschließende Weiterverarbeitung der Analysedaten.

Die Analysewerkzeuge werden primär dafür verwendet, die für die Softwareparallelisierung notwendigen Informationen aus der ADL zu extrahieren. Dazu zählen einerseits High-Level Informationen über die globale Systemarchitektur, welche in grobgranularen Code-Transformationen eingesetzt werden können. Für die Parallelisierung werden zudem Speicherhierarchien und Speicherarchitekturen evaluiert sowie die Kommunikationsinfrastruktur hinsichtlich ihrer Kommunikationskosten zwischen Prozessoren analysiert. Auf der anderen Seite werden Low-Level Informationen über Prozessorarchitekturen, Befehlssatzinformationen und Ressourcenverbrauch extrahiert, die für die feingranulare Parallelisierung und die Code-Generierung verwendet werden.

Der Aufbau sowie die Funktionsweise des ADL-Compiler sind in Kapitel 5 näher beschrieben.

3.4.3 Multicore Simulator

Der Multicore Simulator dient primär der Evaluation eingebetteter Systeme bestehend aus Hardwarekomponenten sowie sequentieller als auch paralleler Anwendungen und Softwarefunktionen. Für die Verwendung in der ALMA Toolchain soll einerseits sichergestellt werden, dass das parallelisierte Programm auch tatsächlich ausführbar ist und kann deshalb simulativen Tests unterzogen werden. Des Weiteren kann nach einer erfolgreichen Parallelisierung noch keine genaue Aussage über die tatsächlich erreichte Beschleunigung

des Programms oder die Qualität der Parallelisierung hinsichtlich Ressourcennutzung und Auslastung des Systems getroffen werden. Um diese Merkmale zu evaluieren bietet sich eine Gesamtsystemsimulation der Anwendung als auch der Hardwarekomponenten an. Nur dadurch lässt sich das Zusammenspiel aus Hardware und Software sinnvoll evaluieren.

Um dies zu ermöglichen wurde der Simulator mit einer Binärkompatibilität entworfen. Dies bedeutet, dass die zu simulierende Architektur Programme ausführen kann, welche auch auf einer realen Hardware ausführbar sind. Bei der Anwendung handelt es sich damit nicht mehr um ein Modell.

Als Eingabedaten verwendet der Simulator eine kompilierte ausführbare parallelisierte Anwendung sowie eine Architekturbeschreibung in ADL. Analog zum Aufbau der ADL-Module verwendet der Simulator einen Bibliotheksansatz, bestehend aus den Simulationsmodulen sowie den Kommunikationsinterfaces. Diese basieren auf SystemC [Sys12] und werden vom Simulator entsprechend der ADL Beschreibung zu einer Simulationsumgebung verbunden. Damit erhält der Simulator die Flexibilität um grundsätzlich jegliche in der ADL beschreibbare Architektur auch simulieren zu können.

Die Nutzung dieses Bibliotheksansatzes mit Simulationsmodulen bietet dabei mehrere für den Entwurf eingebetteter Systeme wichtige Merkmale. Zum einen lässt sich das Hierarchiekonzept der ADL damit vollständig umsetzen, wodurch sich umfangreiche Möglichkeiten zur Entwurfsraumexploration ergeben. Die Simulationsmodule können dabei entweder als Teilkomponenten realisiert werden, so dass mehrere Simulationsmodule ein hierarchisch höher angesiedeltes Simulationsmodul ersetzen können, oder ein Simulationsmodul stellt eine Implementierung auf einer niedrigeren Abstraktionsebene mit höherer Simulationsgenauigkeit dar. Aus diesem Ansatz lässt sich somit auch ein Trade-Off modellieren, mit dem Simulationsgeschwindigkeit gegenüber der Simulationsgenauigkeit abgewogen werden kann. Durch Auswahl der Hierarchieebene innerhalb der ADL, lässt sich die Simulationsumgebung verändern,

wodurch sich abhängig von den gewählten Simulationsmodulen unterschiedliche Simulationseigenschaften ergeben.

Zum anderen kann damit sichergestellt werden, dass die Konsistenz zwischen der ADL, der Simulationsumgebung sowie dem Systemprofiler stets gewahrt bleibt.

3.4.4 Adaptiver System Profiler

Der System Profiler ist ein wichtiges Werkzeug um die Leistungsfähigkeit einer Applikation in Verbindung mit der zugrundeliegenden Hardwarearchitektur zu messen. Er wird deshalb direkt in das Framework und den Simulator integriert und dient der Performanzanalyse parallelisierter Anwendungen. Der Profiler stellt dafür ein generisches Framework zur Verfügung, welches der Hierarchiebeschreibung der ADL sowie dem Simulationsframework gerecht wird. Die Herausforderung liegt dabei in der Bereitstellung eines konsistenten Frameworks, während sich die verfügbaren Simulationsmodule stark voneinander unterscheiden können.

Aus diesem Grund stellt das Framework die notwendigen Funktionen als Application Programming Interface (API) bereit um in jedes Simulationsmodul integriert werden zu können. Es stehen dafür Instrumentierungsfunktionen, Profiling Events und Profiling Types zur Verfügung. Eine Instrumentierungsfunktion dient als Einsprungfunktion innerhalb einer parallelen Anwendung und wirft bei Ausführung sogenannte Profiling Events im Simulator. Durch die Simulation kann die Ausführung der Anwendung unterbrochen werden, wobei nur ein minimaler Unterschied zu einer nicht instrumentierten Anwendung entsteht.

Ein Event kann nun mehrere Profiling Types ansprechen. Das bedeutet, dass die durch ein Event generierten Informationen an mehrere Types weitergegeben werden und dort weiterverarbeitet oder analysiert werden können.

Wird ein Profiling Event direkt in einem Simulationsmodul gebunden, kann er auch ohne Instrumentierungsfunktion geworfen werden. Dadurch kann für das Profiling eine Simulationsunterbrechung durchgeführt werden und hat damit keinen Einfluss auf die simulierte Anwendung.

Die Trennung der Funktionen auf Events und Types ermöglicht es der hierarchischen Flexibilität der ADL und des Simulationsframeworks gerecht zu werden. Ein Event reagiert immer auf genau eine Instrumentierungsfunktion und wird deshalb nur erzeugt, wenn die Funktion, und damit das zugehörige Modul, auch verfügbar ist. So könnte eine Anwendung auf einem Modul simuliert werden, welches mehr oder weniger Profiling Funktionen zur Verfügung stellt als ein anderes Simulationsmodul.

Die Evaluationsfunktionen innerhalb eines Types werden nun vom Event angestoßen. Stehen also weniger Funktionen zur Verfügung, so werden auch weniger Events generiert. Die damit angestoßenen Evaluationsfunktionen werden damit selektiert und deshalb nur ausgeführt, wenn alle notwendigen Informationen zur Verfügung stehen. Auf diese Weise kann ein konsistentes Ergebnis erzielt werden, in dem die Evaluationsmethoden direkt von den verwendeten Simulationsklassen abhängen.

Da die Evaluationsfunktion innerhalb eines Profiling Types implementiert wird, bestimmt die Instrumentierungsfunktion noch nicht welche Evaluationsfunktion tatsächlich ausgeführt wird. Um Daten nicht mehrfach erheben zu müssen kann ein Event die Informationen auch an mehrere Profiling Types weitergeben. Innerhalb des Profiling Types lassen sich somit beliebige Evaluationsfunktionen von Funktionsprofiling über Kommunikationsprofiling, bis hin zu Prozessor-, Speicher-, oder Instruktionstracing implementieren.

Des Weiteren können Simulationsmodule auch eigene spezifische Methoden zum Profiling oder Tracing der Architektur bereitstellen. Um diese Funktionen nutzbar zu machen, kann die API des Profilers dazu verwendet werden, diese Informationen zu sammeln und zu visualisieren. Dies geschieht durch die

Implementierung von Wrapper-Funktionen innerhalb des Simulationsmoduls, welches auch den eigentlichen Simulator kapselt. Alternativ erlaubt das Simulationsframework die Weitergabe von Kommandozeilenparametern an das Simulationsmodul, so dass weiterhin eigene Ausgaben erzeugt werden können.

Kapitel 4

Eine modulare und hierarchische Architekturbeschreibung

Dieses Kapitel beschreibt die Architekturbeschreibungssprache, ihren Aufbau und die Funktionsweise. Darüber hinaus gibt dieses Kapitel eine Einordnung von Architekturbeschreibungssprachen im Vergleich zu anderen Sprachen wieder.

4.1 Einordnung

Für die Beschreibung von Spezifikationen, Systemen und Systemeigenschaften, Software oder Hardwarearchitekturen werden unterschiedliche Sprachen verwendet. Allen voran stehen natürliche Sprachen, wie Deutsch oder Englisch, welche in ihrer Ausdrucksstärke sehr mächtig sind, allerdings auch schwer interpretierbar und daher nicht automatisiert zu verarbeiten sind. Eine natürliche Sprache für die Beschreibung einer Multicorearchitektur zur verwenden ist daher keine geeignete Wahl.

Formale Sprachen wurden entworfen um mit Hilfe mathematischer Konstrukte Verifikationen durchführen zu können. Mit einer Formalen Sprache können Spezifikationen eindeutig beschrieben und automatisiert analysiert sowie verifiziert werden. Für die Entwicklung sowie die abstrakte Beschreibung eines Multicoresystems sind auch formale Sprachen nicht geeignet, da sich abstrakte Eigenschaften und Systemstrukturen nur unzureichend mit mathematischen Konstrukten beschreiben lassen.

Eine weitere Klasse an Sprachen sind Modellierungssprachen wie beispielsweise die Unified Modelling Language (UML) [Obj15]. Sie erlauben die Modellierung der Struktur und auch der Architektur. Modellierungssprachen ermöglichen zwar auch die Beschreibung von Funktionsabläufen, Eigenschaften und Zusammenhängen, allerdings lassen sich konkrete Verhaltensweisen und Funktionsimplementierungen nur unzureichend beschreiben. So ist es nicht möglich Compiler oder Simulationsumgebungen aus einer Modellierungssprache heraus zu generieren.

Für die Funktionsbeschreibung eignen sich Programmiersprachen, wie C [KR88], C++ [Str00] oder auch Java [Ora15], sehr gut. Auch Architekturmodellierungen lassen sich mit Programmiersprachen gut umsetzten. Die Problematik bei Programmiersprachen besteht auf Grund hoher Freiheitsgrade in der Wahl der Implementierung darin, eine automatisierte Analyse von Architektureigenschaften anzubieten.

Eine Hardware Description Language (HDL), welche eine synthetisierbare Beschreibung verfolgt, stellt Architekturinformationen auf einer sehr detaillierten Ebene dar. Die Erzeugung von Hardwarestrukturen ist eine Eigenschaft die Formale Sprachen oder Programmiersprachen nicht abbilden können. Im Gegenzug sind HDLs deutlich eingeschränkter was die Beschreibungsfähigkeiten angeht, weshalb HDLs als eigene Sprachgruppe gesehen werden können. Wie in Abbildung 4.1 zu erkennen ist, haben alle Sprachen einen gemeinsamen Kern, besitzen aber auch jeweils Eigenschaften die die Sprachen voneinander abgrenzen können.

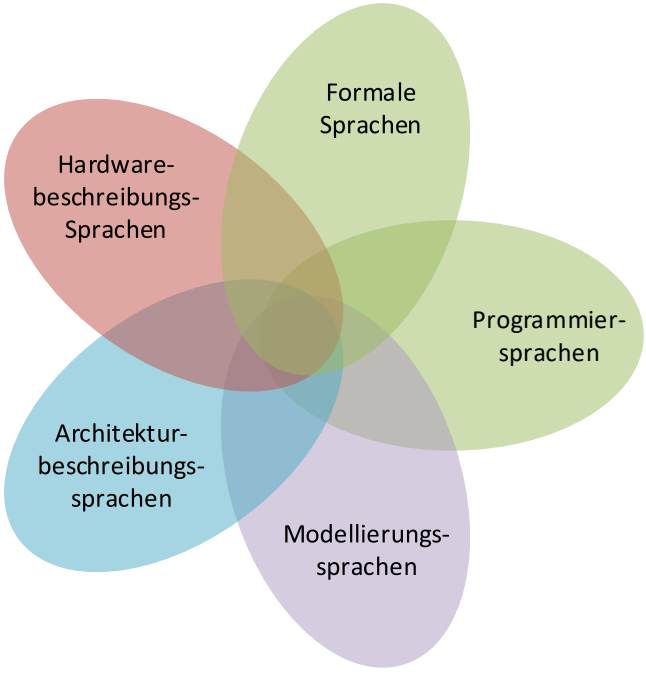


Abbildung 4.1: Abgrenzung einer ADL von anderen Sprachen nach Mishra et. al. [MD08]

Da sich nun viele Sprachtypen für die Beschreibung von Hardware- oder Softwarearchitekturen verwenden lassen, ist eine dedizierte Abgrenzung, ob eine Sprache eine ADL ist, nicht eindeutig durchführbar. Erkennbar ist allerdings, dass ADLs oftmals gemeinsame Eigenschaften, wie eine formale Beschreibung, abstrakte Informationsdarstellung, einfache Modellierungen und Möglichkeiten zur automatisierten Analyse bieten.

4.2 Klassifikation

Das Hauptziel einer ADL ist eine formale Beschreibung von Systemen, Systemkomponenten oder einzelnen Funktionsblöcken sowie deren Verbindungen untereinander. Dabei können zum einen die Strukturbeschreibung als auch die Verhaltensbeschreibung ein Ziel der ADL sein. Weitere Ziele die von ADLs verfolgt werden sind Hardwareentwicklung, Systemverifikation oder Simulator und Compiler Generierung.

Nach Mishra et. al. [MD05] lassen sich ADLs dadurch in folgende Beschreibungsklassen einteilen und anschließend in verschiedene Zielrichtungen klassifizieren:

Strukturelle ADLs werden dazu verwendet die Struktur und den Aufbau von Systemen, Systemkomponenten oder Prozessoren zu beschreiben.

Verhaltensbasierte ADLs bieten die Möglichkeit das Verhalten einer Komponente zu modellieren und zu beschreiben.

gemischte ADLs sind Sprachen die sowohl Struktur, als auch Verhaltensinformationen abbilden. Durch die Kombination der beiden Komponenten, sind diese im Regelfall aber komplexer als reine Sprachen aus nur einem der beiden Bereiche.

partielle ADLs sind Beschreibungssprachen, die nur Teile einer der vorhergegangenen Bereiche abdecken. Sie können sowohl strukturell, verhaltensbasiert oder auch gemischte Sprachen sein. Ihre Beschreibungsmöglichkeiten sind allerdings auf dedizierte Anwendungsfälle hin reduziert und erlauben keine allgemeine Systembeschreibung.

In Abbildung 4.2 sind diese zusätzlich entsprechend ihrer Zielsetzung unterteilt. Es ist zu erkennen, dass nicht alle von einer ADL verfolgten Ziele von der Beschreibungsklasse abhängen.

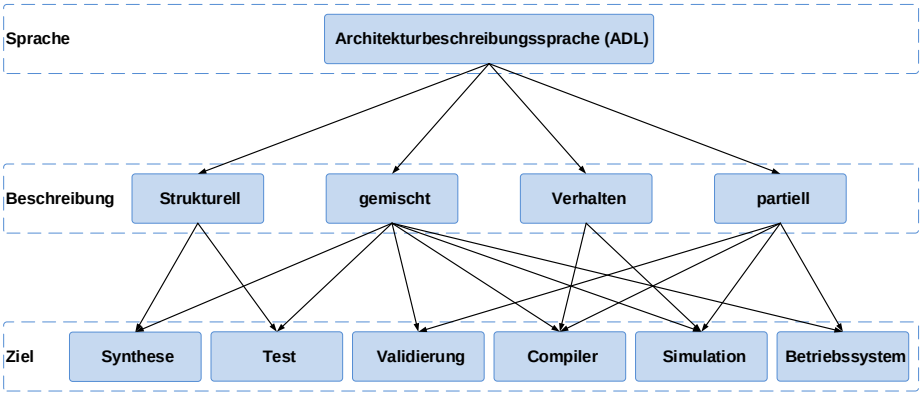


Abbildung 4.2: Klassifikation von Architekturbeschreibungssprachen (abgeleitet aus [MD05])

4.3 ADLs und Sprachen

Dieser Abschnitt beschreibt bekannte Architekturbeschreibungssprachen und Arbeiten im Bereich der Architekturbeschreibung und Analyse. Dazu werden jeweils die Kernpunkte einer Arbeit beschrieben und mögliche Schwachpunkte für die Verwendung in dieser Arbeit genannt.

IP-XACT [Wkh10] ist ein Standard zur Beschreibung von Komponenten, Systemen, Kommunikationsinfrastruktur und zugehörigen Interfaces mit XML. Der Standard beschreibt dazu mehrere Meta-Beschreibungen (XML-Schemas) aus der XML Beschreibungen von Hardwarekomponenten und deren Eigenschaften abgeleitet werden können. Hauptsächliches Ziel ist es vor allem Electronic Design Automation (EDA) Werkzeuge zur automatisierten Generierung, Packaging, Bestückung von Platinen usw. zu unterstützen, deren Systembeschreibung auf RTL-Ebene stattfindet und somit eine synthetisierbare Systembeschreibung erstellt. Diese detaillierte Beschreibung macht IP-XACT für High-Level Analysen und Simulationsumgebungen uninteressant.

Als Erweiterung zu IP-XACT wird in [Mra09] ein Modellbasierter Ansatz vorgestellt, welcher es erlaubt neben der in IP-XACT verfügbaren RTL-Ebene eine sogenannte Transaction-Accurate Ebene zu modellieren aus der sich SystemC TLM Code generieren lässt um die Simulation eingebetteter, mit IP-XACT beschriebener, Systeme zu beschleunigen und die Wiederverwendbarkeit von Modulen zu erhöhen. Dabei ist bisher eine gleichzeitige Verwendung unterschiedlicher Abstraktionsebene noch nicht möglich.

Die Computer Architecture Description Language (CADL) [BVL09] ist eine Struktur orientierte Architekturbeschreibungssprache basierend auf XML. Das Ziel von CADL ist die effiziente Beschreibung von Prozessoren, Instruktionssatz und Pipelinestufen zur Erzeugung von zyklenakkuraten multithreaded Simulatoren. Im Gegensatz zu SystemC, welches einen singlethreaded Kernel besitzt, erlaubt ein multithreaded Simulator deutlich schnellere Ausführungszeiten der Simulation durch eine parallele Ausführung des Simulationscodes. Die Beschreibung erlaubt dazu allerdings nur die Modellierung von Prozessoren und deren internen Komponenten, wie der Pipeline oder dem Registersatz. Die Beschreibung eines Multicore Systems ist in CADL daher noch nicht möglich.

Die Coarse-Grained Array Description Language (CGADL) [FMSR09] ist eine gemischte Architekturbeschreibungssprache zur Beschreibung von grobgranu-

lar rekonfigurierbaren Systemen, welche aus einem Array an Prozessoreinheiten sowie einem Verbindungsnetzwerk bestehen. Eine Prozessoreinheit wird dabei von CGADL auf detaillierter struktureller Ebene beschrieben, und kann Verhaltensbeschreibungen auf der Ebene einer Hardwarebeschreibungssprache enthalten um daraus eine Hardwaresynthese anzustoßen. Alternativ kann damit eine SystemC Simulation erzeugt werden. Im Vergleich zu anderen ADLs fokussiert CGADL einfache Prozessoren, welche mit RISC oder CISC Prozessoren in typischen Multiprozessorsystemen nicht vergleichbar sind. CGADL stellt damit einen Spezialbereich dar, der für die automatisierte Parallelisierung von Hochsprachen-Algorithmen nicht geeignet ist.

Harmless [KBB+12] (Hardware ARchitecture Modeling Language for Embedded Software Simulation) ist eine Architekturbeschreibungssprache, welche Ihren Fokus auf die Beschreibung von flexiblen Instruktionssätzen legt, um Echtzeitsysteme schnell und effizient simulieren zu können. Dazu wird die Beschreibung der Mikroarchitektur von der Beschreibung des Instruktionssatz getrennt um eine Flexibilität hinsichtlich der Wiederverwendung eines Instruktionssatzes auf mehreren Mikroarchitekturen zu erlauben. Des Weiteren wird eine Mikroarchitektur für rein funktionale Simulatoren nicht benötigt, was die Simulationsgeschwindigkeit erhöhen kann. Der Fokus auf Echtzeitsysteme legt nahe, das Multicoreprozessoren nicht unterstützt werden, zumal keine Kommunikationsstrukturen beschreibbar sind.

Die in [LBV13] vorgeschlagene Architekturbeschreibungssprache im Framework PolyCuSP (Politechnique Customizable Soft Processor) ist eine auf XML basierende Sprache, welche die Lücke zwischen syntheseorientierten ADLs, wie bspw. LISA [SHN+02] und einer vollständigen ASIC Entwicklung schließen möchte. Deshalb setzt PolyCuSP auf parametrierbare Prozessortemplates und der Beschreibung von Spezialbefehlen, welche zusammen eine beschleunigte Entwicklung von ASIPs ermöglichen soll. Auf Seiten der ADL, ist diese auf die Beschreibung von Instruktionen und Instruktionssätzen ausgelegt und daher nicht für Multicorearchitekturen anwendbar. Für die Simulation einer mit PolyCuSP beschriebenen Architektur lassen sich zyklenakkurate RTL

Modelle generieren, welche auch in synthetisierbaren HDL Code transformiert werden können. Daneben stehen nur Matlab Modelle für die funktionale Validierung der Instruktionen bereit.

Eine weitere Sprache ist die Embedded Architecture Description Language (EADL) [LPXL10]. EADL ist eine Beschreibungssprache für eingebettete Hardware-Software Systeme, welche auf einem Komponentenmodell, bestehend aus HW-Komponenten und SW-Komponenten, aufbaut. Das Ziel der Sprache ist die Unterstützung von und Integration in Co-Design, Co-Simulation Co-Verifikation und Co-Synthese Methoden. EADL setzt dabei auf die Verwendung einer Formalen Sprache, wodurch eine formale Verifikation einer Architektur sowie der modellierten Funktionalitäten ermöglicht wird. Das Grundmodell innerhalb der EADL ist ein abstraktes eingebettetes System, bei dem Softwarekomponenten innerhalb eines Betriebssystems ausgeführt werden, und die Hardwarekomponenten als Application-Specific Integrated Circuit (ASIC) Prozessor zur Verfügung stehen. Dazwischen stehen sogenannte Bridge-Komponenten, welche einen Adapter zwischen Softwaresemantik und Hardwareevents darstellen. Während eine rein strukturelle Beschreibung den Aufbau des Systems beschreibt, dient die formale Beschreibung nun zur Definition der Funktionalität über Eigenschaften (Properties), die innerhalb der strukturellen Komponenten deklariert werden. Von dieser formalen Beschreibung des Systems lässt sich nun auch simulierbarer und synthetisierbarer HDL-Code generieren. Um die Wiederverwendbarkeit der einzelnen Komponenten zu erhöhen lassen sich Templates und Patterns beschreiben.

Für die Unterstützung von rekonfigurierbaren Architekturen durch Architekturbeschreibungssprachen wird in [LAR11] eine Beschreibungssprache vorgeschlagen, welche XML als Datenbasis verwendet. Auf struktureller Ebene beschreibt die Sprache physikalische Blöcke und Verbindungen. Dabei können primitive Blöcke, wie LookUp Tables (LUTs), Speicher oder Flip-Flops beschrieben werden. Die Verbindungen sind FPGA-typisch als direkte Verbindungen, Crossbar-Verbindungen oder Multiplexer-Verbindungen beschreibbar. Um eine hierarchische Beschreibung zu ermöglichen, können komplexe Blocke auch

aus Teilkomponenten, wie weiteren Komplexen Blöcken oder primitiven Blöcken bestehen. Die Beschreibungssprache dient dabei hauptsächlich der Synthese von komplexeren logischen Blöcken innerhalb von heterogenen FPGA Architekturen und arbeitet deshalb eng mit Netzlistenbeschreibungen zusammen. Eine Hochsprachensimulation, wie mit SystemC oder C++, zur Entwurfsraumexploration ist dadurch nicht möglich.

In [SBP12] wird eine Plattformbeschreibungssprache (PDL) vorgeschlagen um die Programmierung von heterogenen Multicoresystemen zu vereinfachen. Dazu werden die Komponenten eines heterogenen Multicoresystems durch eine generische hierarchische Struktur beschrieben, welche aus einer oder mehreren Master-Komponenten und Worker-Komponenten besteht. Dazwischen existieren sogenannte hybride Komponenten, welche sowohl als Master als auch als Worker in Erscheinung treten können. Das Basisziel der PDL sind Task-orientierte Programme bei der ein Master die Aufgaben zur Laufzeit an die Worker verteilt. Das Konzept ist damit vergleichbar zu Programmiermodellen wie OpenCL [SGS10] oder Cuda von Nvidia [NBGS08], orientiert sich damit an dynamischen Multicoresystemen und versucht die unterschiedlichen Programmiermodelle für Multicoresysteme zu abstrahieren.

EXPRESSION [HGG+99, GHKG98] ist eine gemischte ADL welche das Ziel verfolgt Komponenten und Prozessoren von System-On-Chip (SoC) Architekturen zu beschreiben. Mit EXPRESSION lassen sich detaillierte Beschreibungen von Instruktionssätzen für ASIPs, sowie individuelle Speicherhierarchien in SoC Architekturen erstellen. Aus der Beschreibung lassen sich automatisch, Compiler, Simulator und Reservierungstabellen für das beschriebene System generieren. Da die Sprache für die Entwicklung von Singlecoreprozessoren entworfen wurde sind Interprozessorkommunikation und Multicorebeschreibungen nicht möglich.

LISA [SHN+02] ist eine Architekturbeschreibungssprache zur Beschreibung des Verhaltens eines Prozessors in Form des Instruktionssatzes. Allerdings besitzt LISA auch eine strukturelle Komponente, welche die Beschreibung von

Pipelinestufen ermöglicht. Deshalb wurde LISA in Hardware Ressourcen und ausführbare Operationen unterteilt. Dies ermöglicht es durch die verfügbaren Werkzeuge zyklenakkurate Simulatoren der beschriebenen Architekturen zu generieren sowie Compiler für Spezialinstruktionen zu erstellen, welche ausführbare Dateien erzeugen können.

In [RAC13] wird eine mögliche Anwendung von LISA für die High-Level-Synthese von ASIPs in Multicoreumgebungen vorgestellt. Im Allgemeinen ist LISA auf Grund des Fokus auf Pipelinestufen und Spezialinstruktionen weniger gut für Multicorearchitekturen geeignet.

4.3.1 ArchC

ArchC [ARB+05] ist ein Erweiterung von SystemC und ist damit keine klassische ADL sondern stellt spezialisierten C++ Code dar. Für die Systembeschreibung stellt SystemC zwar die entsprechenden Funktionen zur Verfügung, deren Beschreibung ist aber in dieser Form nicht standardisiert. ArchC versucht genau diese Lücke zu schließen indem es Makros, Templates und Schlüsselwörter mit einen standardisierten Format innerhalb von SystemC für die Systembeschreibung bereitstellt. Daneben existiert ein auf Eclipse basierendes Werkzeug, der Platform Designer [AGB+05], zur Systemmodellierung. Das damit verfolgte Ziel von ArchC ist eine bessere Wiederverwendbarkeit von Systemkomponenten einer Systembeschreibung und einer damit verbundenen Beschleunigung des Entwicklungsprozesses.

Der Fokus der in [RABA04] beschriebenen ArchC ADL liegt auf Prozessorarchitekturen und der Beschreibung von Instruktionssätzen, sowie die automatisierte Generierung von Werkzeugen für die beschriebenen Architekturen. Dadurch, dass ArchC sowie SystemC auf C++ basieren, lassen sich sowohl Struktur, als auch Verhaltensinformationen durch die Programmiersprache beschreiben. Dies macht ArchC zu einer Beschreibungssprache gemischten Types. Die

in ArchC verfügbaren Makros dienen hauptsächlich der Beschreibung struktureller Informationen, da die Makros für die Standardisierung als auch die Interoperabilität gedacht sind. Schlüsselwörter können anschließend dazu verwendet werden um Systemdetails zu beschreiben sowie für die Überarbeitung und Verfeinerung der Beschreibung.

In seiner Basisversion stellt ArchC zwei Makros für die Systembeschreibung zur Verfügung `AC_ARCH` und `AC_ISA`.

`AC_ARCH` beinhaltet dabei die Ressourcen einer Architektur während `AC_ISA` den Befehlssatz des Prozessors spezifiziert.

Um nun eine korrekte, vollständige und funktionale ArchC Systembeschreibung zu erhalten sind allerdings weitere Verhaltensbeschreibungen notwendig die über Standard SystemC Code bereitgestellt werden müssen. Dabei ist es möglich mit einer einfachen und rein funktionalen Systembeschreibung zu beginnen, welche Schrittweise verbessert wird, bis eine vollständige zyklenakkurate Beschreibung erreicht ist.

Sollen unterschiedliche Abstraktionsebenen miteinander verknüpft werden, so ist dies innerhalb von SystemC möglich. In [SRAA11] ist eine Speicherbeschreibung basierend auf SystemC Transaction-level Modeling (TLM) beschrieben, welche es ermöglicht sowohl ein funktionales als auch zyklenakkurates Modell zu verwenden.

Um auch Multiprozessorsysteme beschreiben zu können, wird in [ABAA05] eine Multi-Processor System-on-Chip (MPSoC) Erweiterung vorgestellt. diese erlaubt es mit ArchC auch komplexere homogene oder heterogenen Multiprozessor Systeme zu beschreiben. Darüber hinaus ist es auch möglich Synchronisations- und Kommunikationsmechanismen innerhalb eines MPSoC zu beschreiben.

Dazu wird ein neues Makro `AC_SYSTEM` eingeführt, welches grob mit dem `AC_ARCH` Makro, für die Beschreibung von Einkernprozessoren, vergleichbar ist.

4.3.2 SoCLib

Die SocLib Plattform [SoC11] ist ein auf SystemC basiertes Simulationsframework, welches aus einer Bibliothek an Simulationsmodulen eine Simulationsumgebung in SystemC generiert. Dafür stellt SoCLib eine Bibliothek an Prozessor Intellectual Property (IP)-Cores sowie Systemkomponenten eines MPSoC bereit. Die Bibliothekskomponenten nutzen ein gemeinsames als auch miteinander kompatibles Format, das als Basis für die Erstellung eines Rapid Prototyping Systems sowie weiterer Schritte innerhalb einer Design Space Exploration (DSE) dient.

Um die Interoperabilität der verschiedenen Komponenten sicherzustellen wird jede Komponente, genannt Module, mit Metadaten beschrieben. Analog zu dieser Metadatenbeschreibung wird auch ein System durch eine solche Metadatenbeschreibung realisiert.

Sowohl die Komponentenmetadaten als auch die Systemmetadaten werden in einer Beschreibungsdatei beschrieben, welche auf dem Syntax der Python Programmiersprache [Pyt13] aufbaut. Dadurch lassen sich für den Parsing Vorgang, die für die Programmiersprache Python verfügbaren Werkzeuge verwenden.

In dieser Systembeschreibung werden nun Informationen über die mit den Modules assoziierten Quelldaten eingefügt. Dies sind die C++ Klassennamen sowie die Template-Bezeichner, welche für die SystemC Instanziierung, die Spezifikation von Modul-Parametern, die Verbindungsinformationen sowie für Parameter weiterer Werkzeuge benötigt werden.

Die Modulbeschreibung erfolgt dabei rein strukturell, und lässt nur wenige weitere Architekturanalysen zu, da die Parameter stark auf C++ Templates aufbauen. SocLib dient dabei als Simulationsplattform, welche durch die Metadatenbeschreibung erzeugt wird. Diese bestimmt nun auch die Verwendung sowie die Ziele einer SoCLib Plattform.

SoCLib Modules können auf zwei unterschiedliche Abstraktionsebene beschrieben werden: dem Zyklenakkuraten und Bitakkuraten (Cycle-Accurate / Bit-Accurate (CABA)) Level sowie einem speziellen SystemC TLM Level, genannt TLM-Distributed Time (DT), welches im Abschnitt 6.2.4 näher erläutert wird.

Damit mehrere Module mit jeweils unterschiedlichen Abstraktionsebenen verwendet werden können, muss ein sogenanntes Transactor-Component [GM10] beschrieben und verwendet werden, welches zwischen CABA und TLM-DT übersetzen kann. Jede Modulbeschreibung definiert dabei den Typ der Beschreibung, womit das verwendete Abstraktionslevel spezifiziert wird. Die Verwendung unterschiedlicher Modulbeschreibungen und Beschreibungstypen wird damit erlaubt und ermöglicht die Simulation auf mehreren Abstraktionsebene. Allerdings sind die Abstraktionsebenen auf die beiden beschriebenen beschränkt und es lassen sich keine weiteren Ebenen dazwischen dynamisch generieren.

4.3.3 Kahrisma Core ADL

Das Kahrisma Projekt bestand aus grundsätzlich zwei Komponenten, erstens der Entwicklung einer flexiblen VLIW Architektur [RKB+10] und zweitens einem adaptiven Compiler [SKRB12], der die Entwicklung von Anwendungen für diese Architektur ermöglicht. Um das Softwareframework [SKB11], bestehend aus Compiler, Assembler und Linker der Flexibilität der Architektur anzupassen, wurde von Stripf [Str13] die Kahrisma-Core-ADL entworfen. Dabei handelt es sich um eine Prozessorbeschreibungssprache welche der Beschreibung des Instruktionssatz, der Register und der Operanden dient. Sie unterstützt dabei die Beschreibung des Verhaltens der Instruktionen, welches anschließend zur Generierung des Compilers verwendet wird.

Die Besonderheit dieser Beschreibungssprache ist die Möglichkeit sogenannte Mixed-ISA Architekturen zu beschreiben. Die Kahrisma Architektur ist eine

solche Architektur, welche die Ausführung unterschiedlicher Instruktionssätze erlaubt und sich selbst zur Laufzeit rekonfigurieren, d.h. den Instruktionssatz wechseln, kann. Ein Instruktionssatz der Kahrisma Architektur wird durch eine Konfiguration dargestellt, die aus einem oder mehreren Prozessorkernen bestehen kann. Jeder Kahrisma Kern beschreibt eine zwei-fach VLIW Architektur, welche durch Verwendung mehrerer Prozessorkerne bis zu einer acht-fach VLIW Architektur ausgebaut werden kann.

Der Compiler kann die Informationen aus der Architekturbeschreibungssprache nutzen, um Anwendungen zu erzeugen, die zur Laufzeit rekonfiguriert werden können, um die Kahrisma Architektur effektiv zu nutzen. Darüber hinaus dient die ADL der Erzeugung eines Simulators, welcher in Form eines Basisframeworks die Ausführung von generischen Prozessorinstruktionen erlaubt. Durch das Einbinden der ADL in dieses Basisframework wird der Simulator befähigt diese beschriebenen Instruktionssätze zu simulieren und kann dadurch flexibel auf die Anforderungen der Architektur reagieren.

Die Kahrisma-Core-ADL basiert auf einer Datenbeschreibungssprache welche auch für die Architekturbeschreibungssprache dieser Arbeit für Multicore-Architekturen verwendet wird.

4.4 Die ALMA Architekturbeschreibungssprache

Das grundsätzliche Ziel der Beschreibungssprache ist die Modellierung von Hardwarearchitekturen für die Verwendung in parallelisierenden Softwareentwicklungswerkzeugen und damit die Sicherstellung der Hardwareunabhängigkeit dieser Werkzeuge. Sie bedient sich dabei, wie in Abschnitt 3.4.1 beschrieben, sowohl struktureller als auch verhaltensbasierter Elemente.

Die Zielsetzung der Sprache ist damit die Unterstützung von Compilern sowie die Erstellung einer Simulationsumgebung für Multicorearchitekturen. Beson-

deres Augenmerk liegt auch auf der Unterstützung parallelisierender Werkzeuge wie der ALMA Toolchain. Daher kann die Sprache wie folgt als gemischte ADL mit der Zielrichtung Compiler-orientiert, Simulationsorientiert aber auch Parallelisierungsorientiert eingeordnet werden:

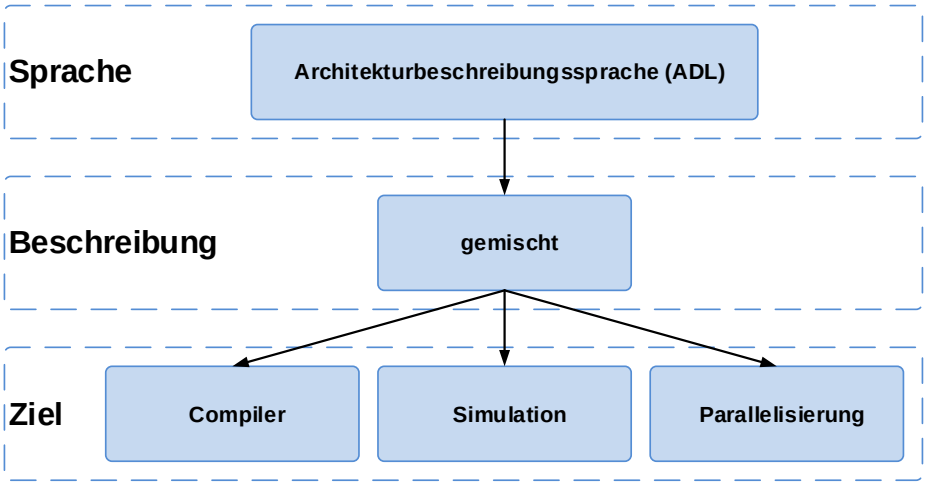


Abbildung 4.3: Klassifikation der ALMA-ADL

Die ADL besteht aus grundsätzlich zwei voneinander unabhängigen Komponenten. Einerseits basiert die Sprache auf einer Data Description Language (DDL) [Str13], welche in ihrer Funktion ähnlich und kompatibel zu bekannten Markup-Sprachen wie der Extensible Markup Language (XML) [Wor08] oder der JavaScript Object Notation (JSON) [ECM11] ist. Darauf aufgesetzt ist die Architekturbeschreibungssprache, welche Elemente innerhalb der DDL definiert. Dazu existiert eine META-Sprachbeschreibung welche die Regeln für die Architekturbeschreibungssprache festlegt und die verwendbaren Elemente und Konstrukte definiert, sowie gültige Wertebereiche spezifiziert.

4.4.1 Datenbeschreibung

Die Datenbeschreibungssprache DDL ermöglicht es Daten in einer Baumstruktur abzulegen. Diese Struktur besitzt zwei unterschiedlichen Elementtypen: Knoten und Blätter. Die Blätter des Baumes stellen nicht weiter verzweigbare Elemente dar und repräsentieren skalare Datentypen. Ein Knoten besitzt mindestens ein weiteres Kind-Element und wird in der Datenbeschreibungssprache durch einen sogenannten Container dargestellt. Ein Container kann wiederum Container-Elemente besitzen wodurch die Baumstruktur aufgespannt wird.

Als mögliche Datentypen stehen Ganzzahl (integer) und Gleitkommazahlen (float) sowie Zeichenketten (string) zur Verfügung. Darüber hinaus können Werte als Boolean (bool) oder undefiniert (undef) deklariert werden. Ganzzahlen und Zeichenketten werden auch als Schlüssel verwendet welche einen Knoten identifizieren.

Um kompakte Beschreibungen zu ermöglichen unterstützt die Sprache auch Variablen, Kontrollstrukturen und Schleifen, sowie Funktionen, welche auf den Daten ausgeführt werden können. Dadurch wird die Sprache flexibel und weit einsetzbar und behält durch die Baumstruktur eine mögliche Kompatibilität zu bekannten Markup-Sprachen wie XML oder JSON. Nach auflösen der Variablen und Sprachstrukturen kann die Datenbeschreibungssprache in eine reine Baumstruktur überführt und so in XML oder JSON übersetzt werden.

Weitere Informationen zur Spezifikation der Sprache, ihrer Funktionen und Operatoren sind im Anhang A beschrieben.

4.4.2 Architekturbeschreibung

Aufbauend auf der Datenbeschreibungssprache wurde die ADL als META-Sprachdefinition beschrieben, welche die Regeln für eine Architekturbeschreibungssprache festlegt. Dabei basieren sowohl die META Sprachbeschreibung

als auch die Architekturbeschreibung auf der gleichen Datenbeschreibungssprache. Dies ermöglicht sowohl eine einfache Erweiterbarkeit der Sprache als auch die Definition des Regelwerkes, so dass die Sprache analysierbar gehalten werden kann.

Die in dieser META-Beschreibung definierten Objekte erlauben anschließend eine Verifikation von Architekturbeschreibungen hinsichtlich ihrer syntaktischen Korrektheit.

Die Architekturbeschreibung selbst ist aufgebaut aus **Sections**, welche die einzelnen Abschnitte für die Beschreibung von Modulen oder Instanzen darstellen. Die Beschreibung definiert hier grundsätzlich sechs unterschiedliche **Sections**, welche auf der obersten Ebene der Architekturbeschreibungssprache angesiedelt sind (Veröffentlicht in [SOB+12a]):

Global Allgemeine Informationen zum System

Interfaces Interface-Bibliothek

Modules Modul-Bibliothek

TopLevel Wurzelmodul einer Architekturbeschreibung

Microarchitectures Instruktionssatz und Ressourceninformation

Configurations Konfigurationsalternativen des TopLevel Moduls

Neben den **Sections** wurden für die Architekturbeschreibung sogenannte **Architecture Description Blocks** oder auch kurz **Blocks** definiert, welche wiederkehrende Elemente in der Architekturbeschreibung beschreiben, die innerhalb unterschiedlicher **Sections** die gleiche Funktion erfüllen und damit nicht auf der Wurzelebene der Datenbeschreibungssprache verwendet werden können. Dazu zählen Verhaltensbeschreibungen, Simulationsparameter, Port und Porttypen sowie die Beschreibung von Implementierungen.

Der Zusammenhang der einzelnen Komponenten der ADL ist in Abbildung 4.4 als UML-Klassendiagramm dargestellt, obwohl die Komponenten in dieser Form keine Klassen darstellen, lassen sich damit die Relationen der Komponenten untereinander sehr gut darstellen. **Sections** sind in Blau dargestellt und **Blocks** in Rot.

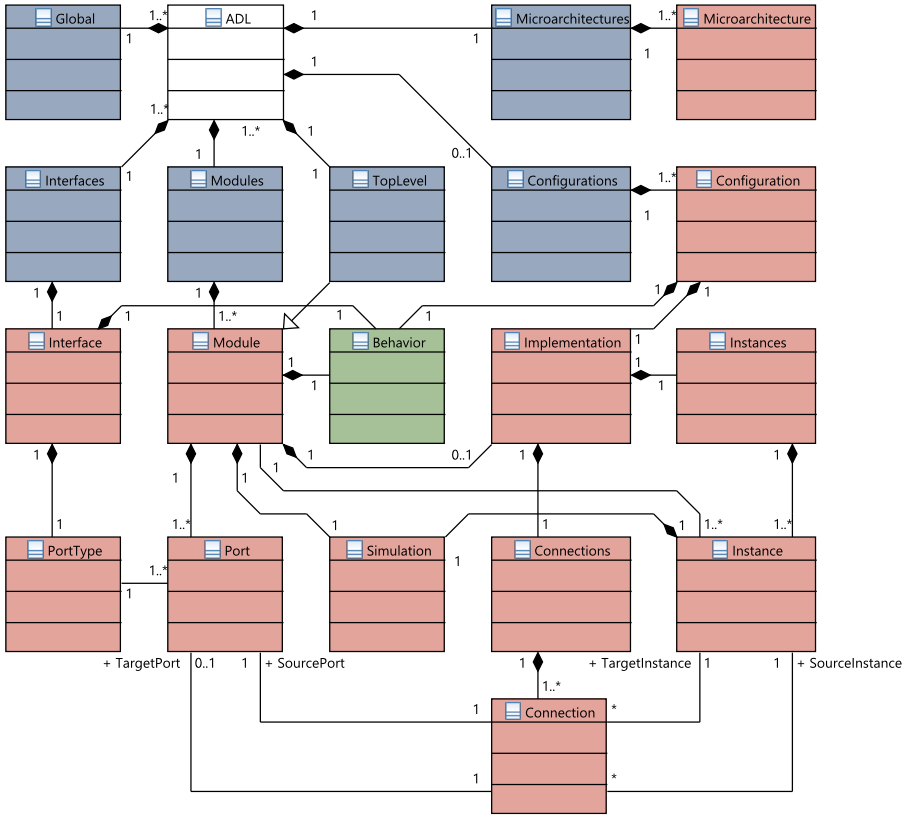


Abbildung 4.4: Darstellung der ADL Komponenten als UML Klassendiagramm

Im Folgenden wird ein Überblick über die **Sections** sowie der Verhaltensbeschreibung gegeben. Detaillierte Informationen zur formalen Definition und

der Implementierung der **Sections** und der **Blocks** sind im Anhang B aufgeführt.

4.4.2.1 Global

Im Global Abschnitt der Architekturbeschreibungssprache werden allgemeine und systemweite Parameter beschrieben. Für die Hardwarebeschreibung sind dies einerseits die Basisfrequenz des Systems, welche für jedes instanziierte Module angenommen wird, solange die Basisfrequenz nicht individuell überschrieben wird. Andererseits existiert der Parameter `BootConfiguration`. Dieser zeigt an, welche Konfiguration bei Start des Systems ausgewählt und konfiguriert werden soll, wenn Konfigurationen in der Systembeschreibung vorhanden sind.

Der Aufbau des `Global` Abschnitts ist in Abbildung B.1 dargestellt, während Quellcode B.1 ein Beispiel mit einer Basisfrequenz von 500Mhz und der Startkonfiguration mit dem Namen RSIW_1 aus dem Konfigurationsbereich der KAHRISMA-Prozessor-Architektur darstellt.

4.4.2.2 Interfaces

`Interfaces` stellen die Typen einer Verbindung zwischen Systemkomponenten und Instanzen dar.

Ein Interface beschreibt einen Verbindungstyp, der die Verbindung zwischen zwei Modul-Instanzen beschreibt. Jedes Interface besitzt einen Namen, mit dem es eindeutig gekennzeichnet wird. Darüber hinaus besitzt jedes Interfaces das Element PortType, welches im Abschnitt B.4.1 näher beschrieben ist, sowie eine Verhaltensbeschreibung Behavior, siehe auch Abschnitt B.9, in der zusätzliche High-Level Informationen gespeichert werden können. Jedem Interface wird außerdem ein `Simulation` Abschnitt zugeordnet in der die zugehörige

Simulationsklasse angegeben wird. Wird keine Simulationsklasse angegeben, so wird für eine Simulation ein Standard-SystemC Interface angenommen.

Da ein `Interface` jeweils an Ports von Modulen gekoppelt wird, besitzt das Interface keinen eigenständigen Port, sondern nur den sogenannten PortType. Dieser definiert die Rolle, die das an diesem PortType angeschlossene Modul in der Verbindung einnimmt. Dazu lässt sich für angegebene PortTypes eine Multiplizität angeben. Diese definiert die Anzahl der gültigen Verbindungen von Ports mit diesem PortType an das Interface. Die Definition der Multiplizität entspricht dabei weitestgehend der Kardinalität in Klassendiagrammen und kann als Ganze Zahl oder Intervall angegeben werden. Die Definition eines `Interface` ist im Anhang in Abschnitt B.4 angegeben.

4.4.2.3 Modules

Als Umsetzung eines Bibliotheksansatzes zur Wiederverwendung von Komponentenbeschreibungen stellen die Module Systemkomponenten dar, welche innerhalb einer Systembeschreibung instanziiert werden können. In Modulen sind die Verbindungsmöglichkeiten, sowie die Grundlegenden Verhaltensbeschreibungen vorhanden.

Die Verwendung von Ports, PortTypes und Interfaces erlaubt die Strukturelle Integritätsprüfung sowie statische und generische Kommunikationsanalysen, sofern Verhaltensinformationen verfügbar sind.

Darüber hinaus kann ein Modul eine **Implementierung** besitzen, welche einerseits die Instanziierung der Module vornimmt (siehe Abbildung 4.5) und andererseits eine Unterstruktur beschreibt, wiederum bestehend aus Modulen und Verbindungen (Siehe Abbildung 4.9).

Mit dieser Implementierung wird das in Abschnitt 3.4.1 vorgestellte Hierarchiekonzept realisiert. Im Verlauf dieser Arbeit wird in Anschnitt 4.7.3 näher

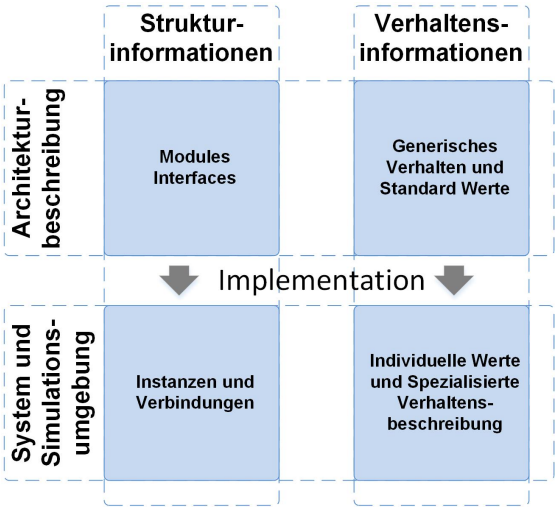


Abbildung 4.5: Darstellung der Implementierung als Werkzeug zur Instanziierung und Individualisierung von Modulen

auf die Hierarchie der ADL, ihre Möglichkeiten und Verwendung im Framework eingegangen.

Dazu verfügt ein Modul über eine Verhaltensbeschreibung, mit der es als Speicher, Prozessor oder Kommunikationskomponente, aber auch als eine Kombination aus mehreren Elementen identifiziert werden kann.

Ein weiteres Element eines Moduls sind Simulationsinformationen, die im Multicore Simulator, siehe Kapitel 6, verwendet und an die Simulationsklassen weitergegeben werden. Darüber hinaus wird in diesem Abschnitt die hierarchische Systemgenerierung gesteuert, welche Auswirkungen auf das zu simulierende System, zur Verfügung stehende Profiling Informationen und weitere Architekturanalysen hat.

Die zwei zwingend notwendigen Angaben sind die Referenz auf die SystemC Simulationsklasse und die Angabe über den Zustand des Moduls in der Hierarchie als aktiv oder inaktiv. Ist ein Modul aktiv, so wird die Hierarchie nicht

weiter aufgelöst, und die hier angegeben Simulationsklasse im Fall einer Simulation verwendet. Ist das Modul inaktiv so wird dessen Implementierung evaluiert. Die Angabe der Simulationsklasse kann durch einen leeren String ersetzt werden, wenn es sich nur um rein analytische Beschreibungen handelt.

Zusätzlich zu den notwendigen Simulationsparametern ist die Definition von klassenspezifischen Simulationsparametern möglich. Deren Verfügbarkeit hängt von der jeweiligen Simulationsklasse ab und erlaubt es damit Parameterwerte zu setzen, welche nur für die Simulationsklasse, nicht jedoch für eine Hardwareimplementierung, notwendig sind.

Während die Angabe der Ports und der Simulationseigenschaften eines Moduls zwingend notwendig ist, um einerseits die Verbindung zu anderen Modulen und andererseits die Simulation als auch die Handhabung der Hierarchie im Simulationsabschnitt zu ermöglichen, sind die Angaben zu Verhalten und Implementierung optional.

4.4.2.4 TopLevel

Das `TopLevel` ist ein besonders Modul, in welchem die eigentliche Systembeschreibung vorgenommen wird. Hierzu werden die in `Modules` beschriebenen Module als Instanzen implementiert und über Ihre Ports verbunden.

Das TopLevel der Architekturbeschreibungssprache ist der Ausgangspunkt der Systemerzeugung. Während die Abschnitte Interfaces und Modules als Komponentenbibliothek dienen, ist das TopLevel die erste erzeuge Instanz im System. Im Grunde handelt es sich um ein automatisch instanziiertes Modul und folgt daher syntaktisch und semantisch einer Modulbeschreibung mit geringen Abweichungen. Die Definition von Modulports ist nicht zwingend erforderlich. demgegenüber müssen die Abschnitte Simulation und Implementation analog zu Modulen definiert werden.

Sollte für ein TopLevel eine Simulationsklasse vorhanden sein, so kann diese auch verwendet werden. Im Allgemeinen ist das TopLevel allerdings inaktiv und dient nur der Beschreibung der höchsten Hierarchieebene.

4.4.2.5 Configurations

Die **Section** `Configurations` ist eine optionale Beschreibung für rekonfigurierbare Systeme welche die Funktionalität einer Instanz oder einer Gruppe von Instanzen ändern kann. Dazu beschreibt eine Konfiguration eine Untermenge der im Toplevel beschriebenen Instanzen und Verbindungen.

Während der Aufbau und Informationsgehalt der Implementierung innerhalb einer Konfiguration mit der eines Moduls identisch ist, wird die Implementierung hier anderweitig verwendet. Die hier genannten Instanzen und Verbindungen stellen eine Anforderungsebene dar um diese Konfiguration abzubilden. D.h. die durch die Instanzen referenzierten Module müssen auch in der beschriebenen Systemarchitektur im TopLevel, und darunterliegenden Hierarchieebenen, instanziiert worden sein und die Verbindungen zur Verfügung stellen. Ist dies der Fall, so kann diese Konfiguration auf das System angewandt werden und beispielsweise zusätzliche Verhaltensbeschreibungen oder eine andere Mikroarchitektur bereitstellen.

Eine Besonderheit dieser Konfigurationen stellt dabei die Fähigkeit dar auf mehrere Instanzen eines Moduls angewendet zu werden. Der ADL Compiler prüft bei Anwendung einer Konfiguration alle vorhandenen Instanzen und Verbindungen auf das in der Konfiguration angegebene Muster.

Für derart beschriebene rekonfigurierbare Architekturen kann eine dynamische Änderung der Konfiguration während der Simulation durchgeführt werden. Dazu ermittelt der ADL Compiler zu Beginn der Simulation alle anwendbaren Konfigurationsmöglichkeiten auf das erzeugte System und führt Protokoll über deren Anwendbarkeit. Die dynamische Rekonfiguration zur Laufzeit übernimmt anschließend ein Laufzeitsystem im Multicore Simulator.

4.4.2.6 Microarchitectures

Der Abschnitt `Microarchitectures` ist im Grunde eine Verhaltensbeschreibung eines Mikroprozessors, welche detaillierte Informationen über den Befehlssatz und die Prozessorressourcen enthält. Die Verwendung von `Microarchitectures` ist in Abschnitt 4.5 näher beschrieben.

4.4.2.7 Verhaltensbeschreibung mittels Behavior

Dieser Abschnitt beschreibt die in der Architekturbeschreibungssprache definierten Verhaltensparameter, welche in einem Modul, einer Konfiguration oder in einem Interface verwendet werden können. Die Verhaltensbeschreibungen sind immer für das Modul gültig in dem sie definiert werden und können beliebig kombiniert werden. Dies ermöglicht es in hierarchischen Beschreibungen Verhaltenskombinationen zu beschreiben, die in einer tieferen Ebene durch mehrere einzelne Module repräsentiert werden können.

Abbildung 4.6 gibt eine Übersicht über die Verfügbaren Verhaltensbeschreibungen der ADL.

Die Definitionen zu den einzelnen Verhaltensbeschreibungen und die zugehörigen Parameter sind im Anhang in Abschnitt B.9 und in Tabelle B.5 beschrieben.

4.5 Architekturbeschreibung aus Softwaresicht

Eines der Hauptkonzepte dieser Architekturbeschreibung ist es, die Architektur nicht nur aus Hardwaresichtweise zu beschreiben. Das bedeutet, dass die Architekturbeschreibung nicht nur aus strukturellen Elementen besteht, sondern auch Verhaltensbeschreibungen enthalten kann. Darüber hinaus wird die

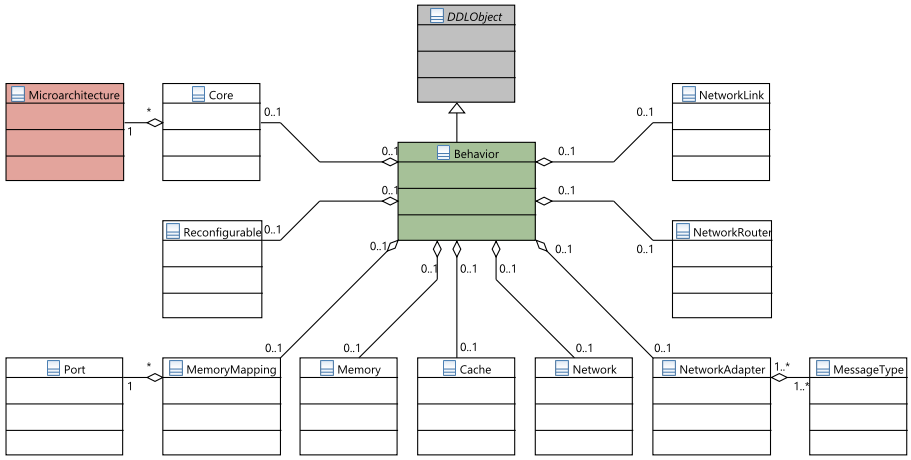


Abbildung 4.6: Darstellung der verfügbaren Verhaltensbeschreibungen innerhalb der ADL als Klassendiagramm

Hardware derart beschrieben, dass ein Softwareentwicklungswerkzeug optimalen Nutzen aus der Beschreibung ziehen kann. Dies ist daran erkennbar, dass Komponenten wie beispielsweise der Instruktionssatz nicht nur im Maschinencode sondern auch mit einer semantischen Deklaration beschrieben werden. Dies ermöglicht es die Hardwarebeschreibung direkt in Compiler und Parallelisierungswekzeuge zu integrieren, die direkt von der semantischen Beschreibung profitieren können. Andererseits erlaubt die Beschreibung im Maschinencode und die zugehörige Ressourcenverwendung auch eine Ableitung der Hardwarearchitektur auf mehreren Ebenen. Dies umfasst sowohl die Systemebene als auch die Komponentenebene der Strukturachse im Y-Diagramm (siehe Abbildung 3.2).

Eine wichtige Komponente der ADL ist dafür die Beschreibung von Mikroarchitekturen als Verhaltensbeschreibung von Prozessoren und Prozessorkernen. Mit dieser Verhaltensbeschreibung werden die strukturellen Informationen einer Multicorearchitektur mit detaillierten Informationen einer Mikroarchitek-

tur verknüpft. Dabei spielt für die Gesamtsystemarchitektur der interne Aufbau eines Prozessors eine untergeordnete Rolle. Allerdings ist die Angabe der verfügbaren Ausführungseinheiten und Ressourcen sowie die Definition des Instruktionssatzes eine notwendige Information für die Parallelisierung von Anwendungen auf feingranularer Ebene.

Um die Komplexität der Mikroarchitektur vor dem Softwareentwickler zu verbergen, sie aber dennoch den Werkzeugen zur Verfügung stellen zu können, wurde hier dieser Zwischenweg gewählt. Jede Mikroarchitektur speichert, wie in Abbildung 4.7 dargestellt, die von dieser Architektur unterstützten Datentypen, die verfügbaren Ressourcen, wie Ausführungseinheiten oder Register, eine Instruktionssatzbeschreibung sowie Compiler-spezifische Parameter.

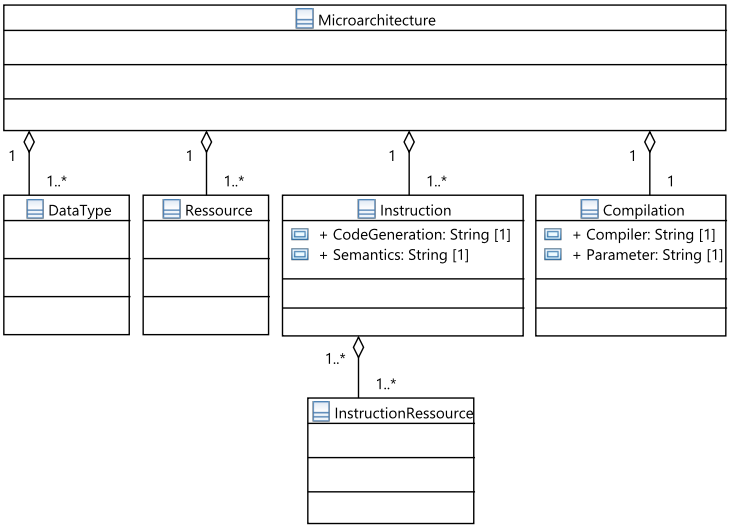


Abbildung 4.7: Klassendiagramm der Microarchitecures-Section

4.6 Flexibilität und Heterogenität

Ein weiterer Vorteil den die Architekturbeschreibungssprache bietet ist deren Flexibilität. Durch die Wahl einer Datenbasissprache mit Baumstruktur ist die Architekturbeschreibungssprache grundsätzlich Domänenunabhängig. Erst durch die Erstellung einer META-Beschreibung und der Entwicklung des Compilers wird die Architekturbeschreibungssprache auf die Domäne der eingebetteten Multicoresysteme zugeschnitten. Dennoch behält die Architekturbeschreibungssprache die Flexibilität auch in anderen Domänen oder zur Beschreibung anderer Systeme verwendet zu werden.

Darüber hinaus besitzt die Sprache durch die Basis der Datenbeschreibung eine Flexibilität die in anderen Sprachen nicht vorhanden ist. Variablen, Schleifen und konditionale Codeausführung sind Konstrukte welche aus Programmiersprachen entnommen sind, und in den bekannten MarkUp Sprachen, wie XML [Wor08] oder JSON [ECM11] nicht ohne weiteres verwendbar oder überhaupt nicht verfügbar sind. Dadurch sind reguläre Strukturen einfach zu beschreiben und die ADL erlaubt es damit flexibel große oder komplexe Systeme zu beschreiben. Durch die bedingte Code Ausführung und die Variablen lassen sich auch dynamisch unterschiedlichste Systeme aus einer ADL heraus erzeugen, was zur Flexibilität der Sprache beiträgt.

Durch die Trennung der Beschreibung von der Simulationsumgebung sowie durch die Entwicklung eines komplexen Compilers, welcher auf mehreren Phasen aufbaut, ist es sehr schnell möglich auch die Beschreibungsmöglichkeiten innerhalb der ADL zu erweitern. Dadurch kann die ADL auch auf zukünftige Entwicklungen im Bereich der eingebetteten Systeme reagieren und die verschiedensten Komponenten beschreiben.

Dies macht sich auch in der Anzahl der beschreibbaren Komponenten und der Verbindungsstrukturen bemerkbar. Im Gegensatz zu manch bestehenden Frameworks ist diese Architekturbeschreibung nicht auf ein Kommunikationsmodell festgelegt.

Es lassen sich durch die Module der Architekturbeschreibung sowohl Bus-Architekturen, direkte Verbindungen oder Netzwerkarchitekturen simulieren. Die Verbindungstopologie eines Netzwerks kann innerhalb eins Netzwerkmoduls funktional, oder durch die Verbindungen der Module untereinander strukturell modelliert werden.

Um der zukünftigen Entwicklung Rechnung zu tragen, ist es mit dieser Architekturbeschreibung auch möglich unterschiedlichste Module zu beschreiben. Jedes Modul kann dabei unterschiedliche Typen, wie Prozessoren, Beschleuniger, Speicher oder Kommunikationsinfrastrukturelemente, darstellen. Damit lassen sich auch heterogene Architekturen sowie System-on-Chip Architekturen mit beliebigen Komponenten beschreiben.

Diese Beschreibungsmöglichkeit ist notwendig um als Basis für eine effektive und effiziente Parallelisierung für eingebettete Systeme eingesetzt zu werden. Spielt heutzutage der Energieverbrauch eine immer größere Rolle im Entwicklungsprozess, so lassen sich leistungsfähige und energieeffiziente Systeme nur durch die Implementierung anwendungsspezifischer Prozessoren realisieren, die in einem heterogenen System eine für den Anwendungszweck ideale Kombination darstellen.

Architekturbeschreibungssprachen, welche wie LISA [SHN+02] oder CASL [WMP+08] die Beschreibung von Pipelinestufen und Spezialbefehlen ermöglichen und daraus automatisch Compiler und Simulatoren generieren können, vermissen den Vorteil hierin auch einen “Standard“ Prozessor integrieren zu können. Auch hier profitiert das Framework von der Trennung der Architekturbeschreibung von der eigentlichen Simulation des Systems, da sich ein anwendungsspezifisch generierter Prozessor auch in das Framework integrieren und durch die ADL beschreiben lässt. So können unterschiedlichste Technologien kombiniert und integriert werden.

4.7 Hierarchische Beschreibung und Abstraktionsebenen

Hierarchie beschreibt grundsätzlich den Aufbau von Komponenten durch untergeordnete Komponenten, so dass ein übergeordnetes Element durch Teilelemente dargestellt werden kann. Dabei stehen die Teilelemente auf einer niedrigeren Hierarchieebene als das Gesamtelement. Durch die Verwendung von Hierarchien lassen sich in Teilen auch Details beschreiben, welche auch für das Gesamtelement gelten, sie werden auf höherer Ebene aber ausgeblendet.

Im Vergleich dazu gibt es Abstraktionsebenen. Eine Abstraktion eines Elementes ist die Darstellung des Elements auf einer höheren Ebene, wobei Details weggelassen werden. Bei einer Modellbildung spricht man von einer Abstraktion, wenn das Modell die Realität widerspiegelt. Dabei kann das Modell aber in seiner Komplexität reduziert werden um nur bestimmte Aspekte einer Realität darzustellen oder um Details wegzulassen, die auf der höheren Abstraktionsebene nicht mehr notwendig sind, oder die Komplexität des Modells unnötig erhöhen würden.

In der klassischen Abstraktion [Jan04] von Modellen spricht man von mehreren Modellierungs- oder Abstraktionsebenen. Die für diese Arbeit relevanten Ebenen sind dabei die "Realität", also die existierende oder zu entwickelnde Hardware oder Software, welche ausgeführt beziehungsweise verwendet werden soll um eine Anwendung oder Funktion auszuführen und zu realisieren. Daneben spielt das Modell die wichtige Rolle. So stellt die Beschreibung einer Architektur mit Hilfe einer Architekturbeschreibungssprache ein Modell der Realität dar, welches die notwendigen Eigenschaften darstellt, die zur Erfüllung der Aufgaben: Simulation, Parallelisierung, Analyse und Entwurfsraumexploration, benötigt werden. Darüber hinaus wird auch ein Meta-Modell realisiert. Dies entspricht einer Anleitung sowie einem Regelwerk zum Erstellen von Modellen, also Architekturbeschreibungssprachen. Dazu gehört nicht

die Syntax der Sprache, sondern die einzelnen dafür verwendeten Komponenten. Konkret handelt es sich um den Aufbau der Module, Instanzen oder Verhaltensbeschreibungen sowie alle weiteren Komponenten der Architekturbeschreibungssprache.

Der Unterschied zu einer klassischen Hierarchie besteht nun in der Behandlung von Information und Details. Sowohl Abstraktionsebenen als auch Hierarchieebenen können dazu verwendet werden um ein System oder in diesem Fall eine Hardwarearchitektur auf unterschiedlichen Ebenen zu modellieren. Bei der Abstraktion werden nun beim Wechsel in eine höhere Abstraktionsebene Details entfernt oder vereinfacht. Das Modell ist allerdings in Bezug auf seine Ebene vollständig. Bei einer Hierarchie gehen immer alle Ebenen in das Gesamtmodell ein. So dass eine niedrigere Hierarchieebene zwar mehr Details darstellt, das Gesamtsystem aber aus den Komponenten einer niedrigeren Ebene aufgebaut ist. Dadurch sind alle Informationen und Details zu jedem Zeitpunkt vorhanden. Ein Modell auf einer definierten Abstraktionsebene kann also hierarchische Elemente aufweisen.

Das besondere Merkmal dieser Architekturbeschreibungssprache ist es nun, sowohl ein hierarchisches Modell zu verwenden, als auch die Möglichkeit zu bieten unterschiedliche Abstraktionsebenen zu beschreiben. Durch die Verwendung hierarchischer Module, können Systeme aus Teilkomponenten aufgebaut werden. Dies vereinfacht die Modellierung von Systemen enorm. Als Beispiel sei ein Mikroprozessor genannt, welcher aus mehreren Komponenten wie Ausführungseinheiten, Zwischenspeichern oder Peripherieeinheiten bestehen kann und bildet an sich ein komplexes System. Als Komponente eines Multiprozessorsystems können nun durch das Instanziieren eines Prozessormoduls die Teilkomponenten des Prozessors automatisch instanziiert werden. Dadurch müssen die Teilkomponenten nicht jeweils einzeln beschrieben werden. Große komplexe Systeme können also in kleinere Teilsysteme aufgeteilt und hierarchisch modelliert werden.

Parallel dazu soll es in der ADL auch sehr einfach möglich sein, unterschiedliche Abstraktionsebenen zu beschreiben. Dies erleichtert die Verwendung der ADL hinsichtlich unterschiedlicher Ziele sowie in den verschiedensten Entwicklungsschritten eines Multiprozessorsystems. Diese Abstraktion wird dadurch erreicht, dass ein hierarchisch modelliertes Modul einerseits aus Teilkomponenten aufgebaut wird, andererseits davon aber auch ersetzt werden kann. Das bedeutet, durch die Verwendung einer höheren Ebene stehen die Informationen aus der niedrigeren Ebene nicht mehr zur Verfügung.

Eine weitere Möglichkeit zur Behandlung unterschiedlicher Abstraktionsebenen stellt die Trennung von Beschreibung und Nutzung dar. Wird die Architekturbeschreibung im Simulator verwendet um eine Simulationsumgebung zu erzeugen, so sind die Simulationsmodule von den ADL Modulen getrennt implementiert. Lediglich ein Verweis in der ADL gibt an, welches Simulationsmodul verwendet werden soll. Damit ist es möglich über die hierarchische Modellierung eines Moduls mehrere Simulationsmodule an ein ADL Modul zu binden. Dabei kann jedes Simulationsmodul eine andere Abstraktionsebene implementieren, beispielsweise eine funktionale Simulation oder eine zyklenakkurate Simulationsimplementierung.

Um dies zu ermöglichen und dennoch eine konsistente Simulationsumgebung erzeugen zu können, muss die hierarchische Beschreibung sehr strikt ausgelegt werden. Während reine hierarchische Beschreibungen damit umgehen können, wenn Verweise oder Verbindungen über mehrere Ebenen hinweg angelegt werden, ist dies hier nicht mehr der Fall. Für die Auswertung der Abstraktionsebene über die hierarchische Beschreibung ergibt sich die strikte Anforderung, dass Verbindungen zwischen ADL Modulen immer auf der gleichen Ebene definiert werden und die Grenzen des Elternmoduls nicht überschreiten dürfen. Dadurch dürfen Teilkomponenten nur innerhalb derselben Systemkomponente oder mit ihrer Elternkomponente verbunden werden.

Dieses Hierarchiekonzept gibt dabei keine Begrenzung der maximalen Ebenen an. Da jedes Modul durch Teilkomponenten dargestellt werden kann, welche

wiederum aus Teilkomponenten bestehen kann, können damit Hierarchieketten abgebildet werden die sehr viele Ebenen beinhalten.

Ein Vorteil der sich aus diesem Konzept ergibt ist die Möglichkeit gleiche Module auf unterschiedlichen Abstraktionsebenen innerhalb derselben Simulationsumgebung zu verwenden und zu simulieren. Damit kann beispielsweise in einem Multiprozessorsystem ein einzelner Prozessor zyklenakkurat und mit sehr vielen Architekturdetails simuliert werden, während die anderen Prozessoren mit einem weniger detaillierten aber wesentlich schnelleren Simulationsmodul simuliert werden.

Auf der anderen Seite benötigt ein derartiges Konzept eine erweiterte komplexere Werkzeugunterstützung. Um Abstraktionsebenen auszuwählen, müssen entsprechende Mechanismen in der ADL vorgesehen werden. Beim Ersetzen eines Moduls durch dessen Teilkomponenten müssen, auf Grund der strikten Hierarchieregeln, die Verbindungen zu den Elternmodulen richtig aufgelöst werden. Dieser Prozess wird als Hierarchieauflösung innerhalb des ADL Compilers implementiert.

Aus diesem Hierarchiemodell, kombiniert mit unterschiedlichen Abstraktionsebenen, lassen sich zwei Konzepte ableiten die in Abbildung 4.8 dargestellt sind. Das erste Konzept stellt die Vertikale Hierarchie dar, während das zweite Konzept horizontale Hierarchie genannt wird.

Beide Konzepte, veröffentlicht in [BOR+14], werden in den nächsten beiden Abschnitten genauer beschrieben.

4.7.1 Vertikale Hierarchie

Die Vertikale Hierarchie entsteht aus der Beschreibung eines Hierarchischen Modells mit mehreren Abstraktionsebenen. Dabei beschreibt jede Hierarchieebene, das Selbe Modell auf einer anderen Abstraktionsebene. Für die Architekturbeschreibung bedeutet dies, dass jeweils nur ein Modul vorhanden ist,

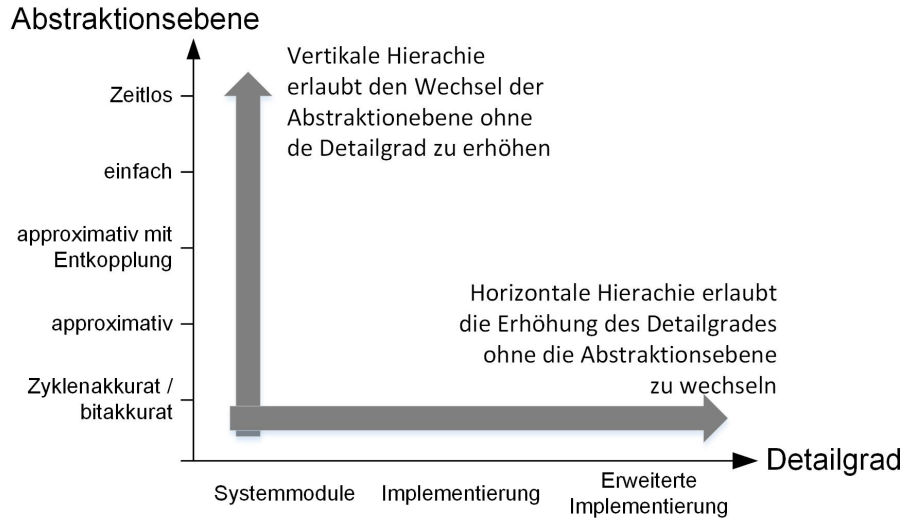


Abbildung 4.8: Darstellung der beiden unterstützten Hierarchiekonzepte innerhalb einer Modul Implementierung der ADL

welches jeweils die gleiche Funktion erfüllt. In jedem Modul können allerdings unterschiedliche Verhaltenseigenschaften implementiert und beschrieben sein. Dadurch kann je nach verwendeter Hierarchieebene der Fokus der Beschreibung verändert werden, beispielsweise auf ein genaueres Berechnungsmodell interner Instruktionen oder auf eine genauere Kommunikationsmodellierung. Das Konzept wird erweitert durch die Verlinkung der Simulationsklassen je nach Abstraktionsebene. So können bei gleicher Funktionalität auch genauere Simulationsmodelle verwendet werden. Abbildung 4.8 stellt diese vertikale Achse der ADL-Hierarchie dar.

4.7.2 Horizontale Hierarchie

Auf der anderen Seite stellt die horizontale Hierarchie die X-Achse des Hierarchieschaubilds dar, wie es in Abbildung 4.8 zu sehen ist. Hierbei werden nun

Module durch eine Implementierung ersetzt, die aus mehreren Modulen besteht. Damit werden dem Modell weitere Details hinzugefügt, deren Funktionalität aber schon auf höherer Ebene beschrieben sind. Die Implementierung stellt dabei beispielsweise das Ersetzen eines Prozessormoduls mit einer Implementierung aus Prozessorkern, internem Speicher und Peripherieeinheiten dar. Auf der horizontalen Ebene wird dem Modell keine Funktionalität hinzugefügt, sondern nur vorhandene Elemente detaillierter dargestellt. Dabei können allerdings weitere Einstellungsmöglichkeiten einstehen, wie beispielsweise die Größe eines internen Speichers, die auf der höheren Ebene nicht zur Verfügung standen.

Das Prinzip dabei ist nun, eine Entwurfsraumexploration möglichst effizient durchzuführen. Anstatt zu Beginn schon hunderte Eigenschaften zu evaluieren, können grobe Modelle für eine schnelle Eingrenzung des Entwurfsraums verwendet werden. Eine zweite Stufe der Exploration kann anschließend mit detaillierteren Modellen in einem kleineren Entwurfsraum durchgeführt werden.

Die ADL erlaubt es dennoch sowohl horizontale als auch vertikale Hierarchiekonzepte zu vereinen. So könnte beispielsweise ein Prozessormodell zur Verfügung stehen, welches rein funktional beschrieben und simuliert werden kann. Eine mögliche Implementierung könnte dabei aus mehreren Modulen mit jeweils zyklenakkurater Simulation bestehen. Dabei gehen allerdings mögliche Zwischenstufen verloren.

4.7.3 ADL Modul Implementation

Die `Implementation` stellt einen ADL-Block dar, der in Modulen oder Konfigurationen sowie dem TopLevel verwendet werden kann.

Über die Implementierung von Modulen, also der Realisierung eines Moduls durch Teilkomponenten, werden die Hierarchieebenen innerhalb der ADL auf-

gespannt. Welche Hierarchieebene für die Simulation verwendet wird, wird über die Simulationsparameter einer Instanz gesteuert.

Die Implementierung beschreibt einen Satz an Modulinstanzen und deren Verbindungen untereinander. Dabei werden Instanzen angelegt, welche mit Hilfe ihrer Ports miteinander verbunden werden können. Eine Implementierung eines Moduls beschreibt damit eine Hierarchieebene innerhalb des Moduls, da jede Instanz wiederum einem Modul zugeordnet ist.

Daher gilt für die Verbindungen eine strikte Einhaltung der Hierarchischen Ordnung. Instanzen können nur mit Instanzen auf der gleichen Ebene, d.h. innerhalb derselben Implementierung oder mit ihrem Elternmodul verbunden werden. Nur so lassen sich die Unterschiedlichen Hierarchieebenen effektiv und korrekt auflösen um eine simulierbare und analysierbare Umgebung zu erhalten.

Abbildung 4.9 stellt die in der ADL verfügbaren Elemente zur Beschreibung der Hierarchie dar. Aus der Verknüpfung zwischen einem Modul und einer Instanz ist es möglich, dass mehrere Hierarchieebenen beschrieben werden was in Abbildung 4.9 in grün dargestellt ist. Eine Instanz kann wiederum, mit der Verknüpfung zum Modul, eine Implementierung besitzen.

4.7.3.1 Instanzen

In diesem Abschnitt der Implementierung werden Instanzen von Modulen definiert. Eine Instanz stellt dabei ein konkretes Objekt eines Moduls dar. Im Vergleich zu objektorientierten Programmiersprachen beschreibt ein Modul eine Klasse, während eine Instanz ein Objekt dieser Klasse darstellt. Jede Instanz besitzt daher eine Referenz zu dem Modul von dem die Instanz abgeleitet wird. Darüber hinaus kann eine Instanz einen Simulationsblock enthalten, der die im Modul beschriebenen Simulationsframework überschreibt oder erweitert. Damit können Instanzen individualisiert und angepasst werden. Über den speziellen ADL-Operator **SubInstance** lassen sich über die Implementierung eines

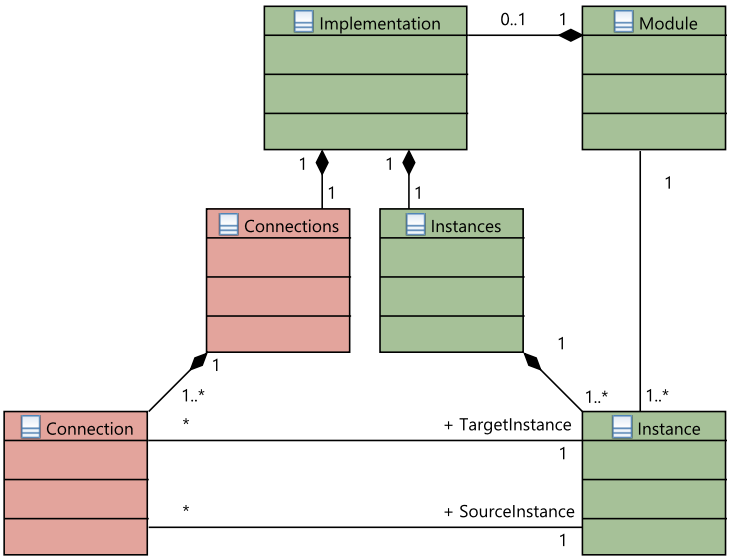


Abbildung 4.9: Darstellung der Hierarchiemodellierung innerhalb der ADL als Klassendiagramm

Moduls die darin erzeugten Instanzen auf den unteren Hierarchieebenen aufrufen und einzelne Parameter überschreiben.

Parameter die nicht direkt von Instanzen oder über **SubInstance** angesprochen werden, werden vom Elternmodul übernommen, so dass das Ergebnis der Vereinigungsmenge entspricht:

$$Instance_{Eigenschaften} = Module_{Eigenschaften} \cup SubInstance_{Eigenschaften}$$

4.7.3.2 Connections

Dieser Abschnitt beinhaltet alle innerhalb dieser Implementierung auftretenden Verbindungen zwischen Instanzen. Verbindungen werden durch die voll-

ständige Angabe der Quellinstanz, des Quellports, des Zielports und der Zielinstanz definiert. Die Angabe des zu verwendenden Interfaces ist nicht notwendig, da sie implizit aus der Portdefinition im Modul abgeleitet werden kann. Verbindungen zwischen Instanzen werden innerhalb der ADL immer bidirektional propagiert. Soll also eine unidirektionale Verbindung modelliert werden, so geschieht dies über die Definition des Interfaces und deren PortTypes, welche über die Modulports referenziert werden.

Kapitel 5

Ein flexibler ADL-Compiler

Dieses Kapitel beschreibt den Aufbau und die Funktionen des ADL Compilers. Dabei wird auf die einzelnen Funktionen eingegangen, welche für die Überführung der ADL in die Zwischendarstellung benötigt werden und welche Transformation auf der ADL durch die jeweiligen Funktionen durchgeführt werden.

Der ADL-Compiler hat die Aufgabe die Architekturbeschreibung in eine Zwischenrepräsentation zu überführen, welche dann effektiv weiterverarbeitet werden kann. Dazu gehören einerseits das Parsen der Datenbeschreibungssprache sowie die Übersetzung der ADL-Beschreibung in eine Zwischendarstellung. Auf der Basis dieser Zwischendarstellung werden Struktur- und Verhaltensinformationen durch Architekturanalysefunktionen extrahiert, so dass diese zur Parallelisierung einer Anwendung verwendet werden können.

Die Zwischendarstellung wird zudem innerhalb des Simulators weiterverwendet um daraus die Simulationsumgebung zu erzeugen ([BOR+14]). Zu den Aufgaben des ADL-Compilers gehört deshalb auch die Hierarchieauflösung entsprechend der ADL-Parameter, sowie die dadurch notwendige Reparatur aufgebrochener Instanzverbindungen.

5.1 Software-Architektur

Die Architektur des Compilers folgt den Prinzipien aktueller Multi-Pass Compiler Frameworks. Die einzelnen Funktionen werden modular und unabhängig implementiert, so dass sie nur auf den zuvor erzeugten Ergebnissen arbeiten und damit austauschbar und aktualisierbar bleiben. Dadurch wird auch sichergestellt, dass der Compiler entsprechend anpassbar und erweiterbar ist, sollten neue Funktionen in der Architekturbeschreibungssprache umgesetzt werden.

Für die Architektur des Compilers existieren deshalb Container-Klassen für die einzelnen ADL-Objekte. Jedes ADL-Element wird nun durch ein Objekt innerhalb eines Containers dargestellt, in dem die Parameter, Variablen und Funktionen abgelegt werden.

Die Container Klassen sind im Klassendiagramm (Abbildung 5.1) blau hinterlegt. Die einzelnen Passes des Compilers sind als Funktionen in die CSADL Klasse integriert.

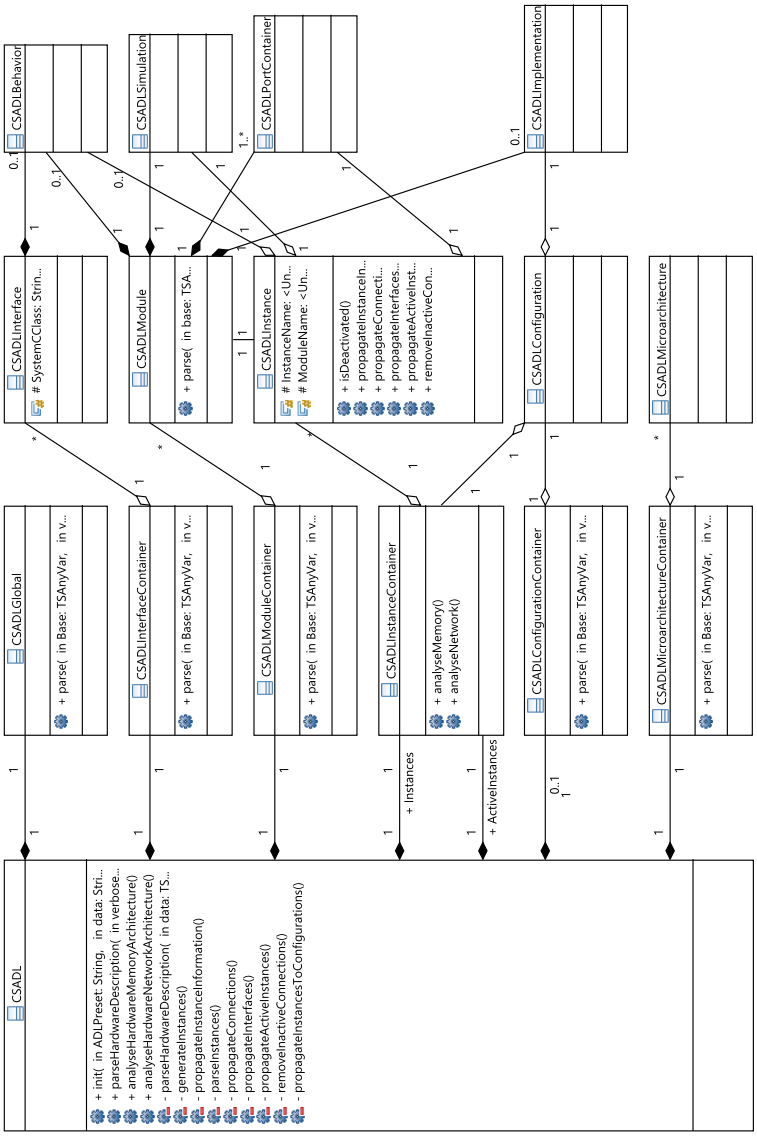


Abbildung 5.1: Klassendiagramm des Compilers und des internen Aufbaus

5.2 Analysephase

Die Analysephase eines Compilers bezeichnet man üblicherweise als Frontend, in dem lexikalische, syntaktische und semantische Analysen des Quellcodes durchgeführt werden. Diese Schritte wendet auch der ADL-Compiler auf den ADL-Quellcode an. Über die Angabe einer META Beschreibung können zusätzliche sprachbedingte semantische Analysen ausgeführt werden.

5.2.1 Lexikalische und syntaktische Prüfung

In diesem Pass werden die Eingabedateien auf ihre lexikalische und syntaktische Korrektheit geprüft. Dazu kommt die Bibliothek der Datensprache DDL zum Einsatz. D.h. der Compiler überprüft hiermit die Syntax der Datenbeschreibungssprache auf Fehler. Des Weiteren werden in diesem Schritt Variablen und Kontrollstrukturen aufgelöst, so dass Schleifen vollständig ausgerollt und bedingte Ausführungen aufgelöst wurden. Alle Variablen werden hierbei durch ihre Werte ersetzt sowie ADL-Funktionen ausgeführt und evaluiert.

Das Ergebnis ist also eine vollständig analysierte ADL-Beschreibung ohne Variablen oder Kontrollstrukturen.

5.2.2 Semantische Prüfung

Die META-Beschreibung wird in derselben Datenbeschreibungssprache angegeben wie eine Architekturbeschreibung selbst. Dabei gibt die META-Beschreibung den Rahmen vor, welche Inhalte eine Architekturbeschreibung beinhalten muss und wie deren Aufbau gestaltet ist, sowie welche Daten eingetragen werden.

Diese Beschreibung ist unerlässlich, wenn eine Analysierbarkeit der Eingabedaten garantiert werden soll.

Da die META-Beschreibung in DDL verfasst ist, durchläuft auch die META-Beschreibung die Syntaxprüfung mittels DDL und wird dann als zweite Eingabedatei vom Compiler verarbeitet. Die semantische Prüfung der Architekturbeschreibungssprache evaluiert nun die strukturelle Korrektheit einer Architekturbeschreibung indem Sie die ADL-Eingabedatei auf die Einhaltung der in der META-Beschreibung definierten Regeln hin prüft. Dieser Pass ändert die ADL-Eingabedatei nicht ab, sondern erlaubt nur deren weitere Verarbeitung im Compiler.

In Abbildung 5.2 ist ein Beispiel einer Meta-Beschreibung angegeben, welches einige der Beschreibungsmöglichkeiten darstellt.

```
['SingleData']['NetworkAdapter']['Hash'] = {
  ['Order'] = ('Latency', 'MessageTypes');
  ['MayExists'] = ['MustExists'] = set(['Order']);
  ['SingleData']['Latency']['Int'] = true;       // cycles
  ['SingleData']['MessageTypes']['Hash'] = {
    ['Data']['Hash'] = {
      ['Order'] = ('DataSize', 'DataDirection', 'SendOverhead', '
          ReceiveOverhead');
      ['MayExists'] = set(['Order']);
      ['MustExists'] = set('DataSize', 'DataDirection');
      ['SingleData']['DataSize']['Int'] = true; // bits
      ['SingleData']['DataSize']['Vector'] = {
        ['Data']['Int'] = true;
      };
      ['SingleData']['DataDirection']['String'] = true;
      ['SingleData']['SendOverhead']['Int'] = true; // bits
      ['SingleData']['ReceiveOverhead']['Int'] = true;// bits
    };
  };
};
```

Abbildung 5.2: ADL META Beschreibung der `NetworkAdapter`-Verhaltensbeschreibung mit Unterstützung mehrerer Nachrichtentypen.

Die dazugehörige logische Struktur der Meta-Beschreibung wird durch die Baumstruktur in Abbildung 5.3 realisiert.

Einige semantische Prüfungen, wie beispielsweise die korrekte Verbindung von Interfaces, oder bestimmter Eigenschaften, wie `SendOverhead` und `ReceiveOverhead`, können nicht durch die Meta-Prüfung abgedeckt werden.

Diese beiden Eigenschaften haben weiterreichende Abhängigkeiten von anderen Eigenschaften, hier `DataDirection`, welche die Verwendung der beiden vorherigen Eigenschaften definiert. Diese semantischen Prüfungen werden in den nachfolgenden Compilerpasses durchgeführt, sobald die notwendigen Informationen verarbeitet wurden und zur Verfügung stehen.

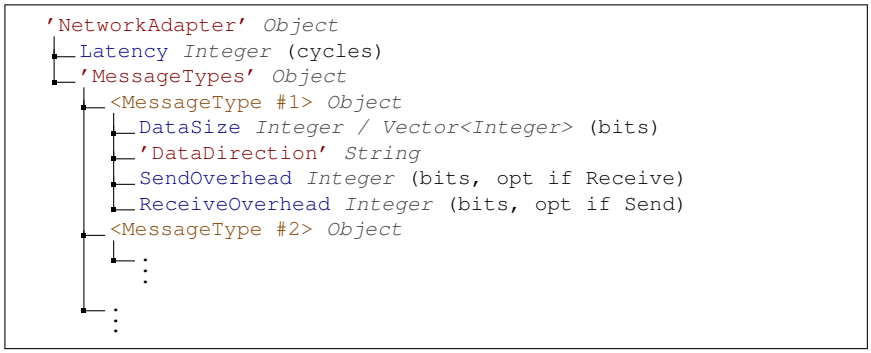


Abbildung 5.3: Struktur der `NetworkAdapter`-Verhaltensbeschreibung.

Handelt es sich hierbei um klar ausführbare Prüfungen, welche direkt nach dem Parsen der Informationen durchgeführt werden, so werden sie in diesen Schritt integriert. Andernfalls werden komplexere Funktionen im Anschluss an den Parsing Prozess implementiert und ausgeführt.

5.3 Sprachevaluation

Das Backend des ADL Compilers ist im Gegensatz zu Programmiersprachen-Compilern nicht in der Lage ausführbaren Code zu erzeugen. Die Funktion

ist aber einem Multi-Pass Compiler ähnlich. Das Backend des ADL-Compilers erzeugt eine in-memory Repräsentation, die an weitere Werkzeuge weitergegeben werden kann und auf der Architekturanalysen ausgeführt werden.

Die Passes sind als Funktionen der Klasse CSADL realisiert und arbeiten auf den darin referenzierten Container-Klassen für Module, Instanzen usw., siehe Abbildung 5.1. Im Folgenden werden die einzelnen Passes des ADL-Compilers beschrieben, die in der angegebenen Reihenfolge aufgerufen werden.

Module Parsing In diesem Pass werden die in der ADL beschriebenen Module vollständig geparst. D.h. die in der ADL angegebenen Werte einzelner Parameter werden in die Variablen der Modul-Objekte geschrieben. Für jedes Beschriebene ADL-Modul wird ein eigenes Objekt im Modul-Container erstellt. Dieser Schritt übernimmt nur die Rohdaten und führt noch keinerlei weitere Transformationen oder Analysen aus. Dies ermöglicht eine einfachere und effizientere Implementierung der Individualisierung einzelner Instanzen, da die durch SubInstance zur Verfügung gestellten Werte direkt mit den Rohdaten kombiniert werden können.

Generate Instances Aus den zuvor generierten Modul-Objekten werden nun Instanzen erzeugt. Es wird ein weiterer Container für Instanzen verwendet, in den nun jeweils ein Objekt pro ADL-Instanz erzeugt wird. Auch hier sind nun alle Instanzen eindeutig, können aber von demselben Modul-Objekt abstammen. Durch diese Kombination kann die Referenz zum Modul-Objekt erhalten werden, was spätere Analysefunktionen befähigt auf Modulebene zu arbeiten.

Jedes Instanzobjekt kopiert dabei das Modul aus dem es erzeugt wurde, besitzt darüber hinaus aber auch die Verbindungsinformationen welche in den Implementierungsabschnitten der ADL-Beschreibung modelliert sind. Hierbei erzeugt der Compiler, alle Instanzen die in allen modellierten Hierarchieebene beschrieben werden. Allerdings nur solche Instan-

zen, die sich aus der initialen Instanz des TopLevel und dessen Implementierung ableiten lassen.

Propagate Instance Information Nachdem alle Instanzen erzeugt wurden, werden Verhaltensbeschreibungen, SubInstance Informationen und Implementierungen der Module an die Instanzen propagiert. hierbei werden die Rohdaten übertragen und mit den Rohdaten der Instanzen kombiniert. Dabei können Parameter überschrieben oder vollständige Objekte hinzugefügt werden. Die Daten liegen hier noch immer als Rohdaten vor. Für die korrekte Auflösung der Hierarchie sind die Implementierungsinformationen notwendig. So kann eine Instanz feststellen, welche Verbindungen zwischen den Hierarchieebenen, erlaubt sind direkte Verbindungen zur Eltern-Instanz und zu Kind-Instanzen, existieren. Dabei kann auch eine rekursive Verteilung vorgenommen werden um alle Hierarchieebenen korrekt zu erfassen.

Parse Instances Sind alle Ebenen vollständig erzeugt und die Informationen an alle Instanzen korrekt propagiert worden, können die einzelnen Rohdaten in den Instanzen geparst werden. Dazu wird über alle Instanz-Objekte iteriert und die Rohdaten in Variablen oder C++-Structs überführt.

Propagate Connections Dieser Pass ermittelt die korrekten Verbindungen zwischen Eltern einer Instanz und Kindern einer Instanz. Dadurch, dass bei Auswahl einer tieferen Hierarchieebene eine Instanz entfernt wird, müssen die Kinder der entfernten Instanz wissen, welche Instanzen mit dieser Instanz verbunden sind, damit diese Verbindungen korrekt aufgelöst werden können. Dazu werden alle Verbindungen entsprechend an Kinder und Eltern propagiert. Dies erleichtert im Nachhinein die Reparatur von Verbindungen.

Propagate Interfaces Da nun alle Verbindungen vollständig propagiert wurden, können diesen die entsprechenden Interfaces zugeordnet werden. Dadurch erhalten die Verbindungen auch Verhaltensinformationen

und Typinformationen. In diesem Pass wird daher auch die Konsistenz der Instanzverbindungen geprüft.

Propagate ActiveInstances Die Auswahl der Hierarchieebenen im Simulation Block einer Modul- oder Instanzbeschreibung wird in diesem Pass ausgewertet. Der ADL-Compiler erstellt einen neuen Instanz-Container und kopiert in diesen nur die aktiven Instanzen, welche zur Analyse oder Simulation ausgewählt wurden. Anschließend wird nur noch auf diesem Container gearbeitet. Der vollständige Instanz-Container ist weiterhin verfügbar und kann zur Rekonfiguration der Simulationsumgebung verwendet werden.

Remove Inactive Connections Auf den aktiven Instanzen wird nun eine Bereinigung und Reparatur durchgeführt. Verbindungen zu nicht mehr existierenden Instanzen werden gelöscht und durch die alternativen Verbindungen ersetzt um eine Hierarchieebene zu überspringen. Der ADL-Compiler führt zu diesem Zweck eine Tiefensuche auf den aktiven Instanzen durch und beginnt mit der Ersetzung bei denjenigen Instanzen die keine Kind-Verbindungen mehr besitzen. Dadurch können die nun daran anzuschließenden Instanzen, Verbindungen sowie alternative Verbindungen, gleichermaßen aktualisiert werden. Dadurch wird es möglich Verbindungen mehrerer fehlender Hierarchieebenen korrekt wiederherzustellen.

Kapitel 6

ADL basierte Multicore System Simulation

Dieses Kapitel beschreibt den Aufbau und die Funktionsweise des Multicore Simulators, sowie eine kurze Beschreibung von SystemC, welches als Grundlage für die Simulation verwendet wird. Für den Multicoresimulator wird der vollständige Toolflow beschrieben, um aus der ADL eine simulierbare Umgebung zu erzeugen, sowie die Erstellung von Simulationsstatistiken.

6.1 Ziele

Das Simulationsframework verfolgt mehrere Ziele, welche aus dem EU-Projekt ALMA entstanden sind. Dazu zählt als eines der Hauptziele die Evaluation paralleler Programme welche mit Hilfe der ALMA Toolchain erstellt worden sind. Da diese Werkzeuge architekturunabhängig entworfen worden sind, erhalten sie die architekturspezifischen Informationen aus der ADL. Eine Aufgabe des Simulators ist es nun eine Simulationsumgebung für die in der ADL beschriebenen Architektur bereitzustellen und die Evaluation des parallelen Programms für diese Architektur zu ermöglichen.

Dementsprechend wird das Ziel der Flexibilität des Frameworks davon abgeleitet. Ein fest definierter Simulator wäre kontraproduktiv für die Entwicklung architekturunabhängiger Parallelisierungswerkzeuge. Deshalb soll der Simulator sowohl auf die zu simulierende Architektur, als auch auf die Abstraktionsebene der Architekturbeschreibung anpassbar sein und mehrere Simulationsmodi adaptiv unterstützen.

Für die Evaluation von sowohl Architektur als auch Anwendung ist es unerlässlich entsprechende Funktionen mit in den Simulator zu integrieren. Dies geschieht durch den System Profiler, welcher in der Lage ist sowohl Tracing als auch Profiling Funktionen zu unterstützen.

Die Ergebnisse der Evaluation sollen anschließend in geeigneter Form für die weitere Verarbeitung zur Verfügung gestellt werden. Ein besonderes Ziel ist dabei die Rückführung dieser Ergebnisse in die Eingabebeschreibung der Architektur. Dadurch lassen sich iterative Optimierungen sowohl bei der Verwendung in Entwicklungswerkzeugen, wie der ALMA-Toolchain, als auch in der Simulation selbst realisieren.

6.2 Hardware und Software Simulation

Ähnlich zu einem realen Computer System ist das zentrale Element eines MPSoC Simulators die Prozessorsimulationseinheit. Dabei kann in mehrere Klassen von Simulatoren unterschieden werden. Abhängig vom Ziel der Simulation ergeben sich einerseits Hardwaresimulatoren, welche das Ziel verfolgen die physikalischen Vorgänge eines Prozessors nachzubilden und zu simulieren. Auf der anderen Seite gibt es funktionale Simulatoren, die die Funktionsweise eines Prozessors nachempfinden. Diese können physikalische Effekte mit einbeziehen, sind aber in der Regel nicht darauf ausgelegt.

Innerhalb der funktionalen Simulatoren gibt es wiederum mehrere Unterteilungen. Es gibt reine Funktionssimulatoren, in denen die Struktur, also der Aufbau des Prozessors, keine Rolle spielt. Das Verhalten des Prozessors, also dessen Funktion, wird dabei über mathematische Funktionen abgebildet. Je nach Detailgrad können auf einem solchen System Softwareanwendungen auch ausgeführt werden, ist allerdings eher selten der Fall, da diese direkt in die mathematische Beschreibung integriert sind. Auf Grund der mathematischen Simulation sind rein funktionale Simulatoren sehr schnell in der Ausführung und sind daher für eine Entwurfsraumexploration für Softwarefunktionen sehr gut geeignet.

Eine weitere Ebene sind strukturelle Simulatoren. Diese bilden den inneren Aufbau eines Prozessors ab, wodurch eine sehr genaue Simulation der inneren Abläufe eines Prozessors entsteht. Der Fokus solcher Simulatoren liegt dabei nicht auf der Ausführung von Softwareanwendungen, sondern eher dem Low-Level Entwurf neuer Prozessorarchitekturen. Durch die strukturelle Simulation lassen sich Hardwaretests durchführen und Ausführungsfehler aufdecken. Eine solche Simulation erfordert auf Grund ihrer Komplexität hohe Rechenkapazitäten und kann im Regelfall nicht mit der realen Geschwindigkeit des zu simulierenden Prozessors ausgeführt werden. Eine Simulation hat daher eine

sehr lange Laufzeit und ist deswegen nicht für die Exploration von MPSoC Systemen geeignet.

Eine sich dazwischen bewegende Klasse sind sogenannte Instruktionssatzsimulatoren (ISSs). Sie simulieren einen Prozessor indem sie sich auf funktionaler Ebene wie ein Prozessor verhalten und dabei seinen Instruktionssatz (ISA) simulieren. Der Instruktionssatz stellt die ausführbaren Befehle eines Prozessors dar. Je nach Ausführung des ISS kann es sich nun um rein funktionale oder rein strukturelle Simulatoren handeln. Oftmals werden aber auch beide Sichtweisen miteinander verknüpft um die Vorteile beider Seiten zu verwenden.

Da ein ISS jeweils nur einen Prozessor simuliert, werden diese Simulatoren mit weiteren Simulationskomponenten kombiniert um eine vollständige Systemsimulation zu erhalten. Eine Systemsimulation enthält neben dem Prozessor auch weitere Elemente wie Speicher, Caches, Bus- und Kommunikationssysteme oder Input/Output (I/O) Controller.

Da ISS oftmals strukturelle als auch funktionale Elemente miteinander verbinden, kann eine Systemsimulation auf verschiedenen Abstraktionsebenen realisiert werden. Dadurch lassen sich unterschiedlichste Ziele hinsichtlich der Performanz einer Simulation erfüllen. Dies kann beispielsweise eine möglichst schnelle oder eine möglichst genaue Simulation sein. Da sich die Ziele Genauigkeit und Geschwindigkeit orthogonal zueinander verhalten, ist es oftmals so, dass eine sehr genaue Simulation auch höchstwahrscheinlich eine sehr langsame Simulation ist und umgekehrt.

Zu den genauesten Abstraktionsebenen der Simulatoren zähl die Register-Transfer Level RTL Simulation, wie sie in Hardwarebeschreibungssprachen wie VHDL oder Verilog, aber auch in SystemC eingesetzt wird. Für die Ausführung von Prozessorinstruktionen, d.h. bei einer Verwendung als ISS können damit Geschwindigkeiten bis wenige hundert Hertz erzielt werden. Heutzutage finden Prozessoren in diesen Frequenzbereichen nur noch in Spezialbereichen Anwendung, weshalb eine Simulation in den seltensten Fällen für die

Simulation von Softwarefunktionen verwendet werden kann. Allerdings lassen sich durch die Strukturelle Genauigkeit einer Register-Transfer Level (RTL) Simulation, die internen Abläufe eines Prozessors präzise nachempfinden, testen und optimieren. Befehle, Speicherzugriffe oder Kommunikationsoperationen lassen sich mit Hilfe einer RTL Simulation zyklengenau, und teilweise auch sub-zyklengenau, simulieren. Für die Ausführung einer Softwareanwendung sind diese jedoch nicht geeignet. Wohl aber um dedizierte Funktionen in Hardware zu beschreiben und diese zu simulieren.

In Abbildung 6.1 werden nun bekannte Simulationstechnologien hinsichtlich ihrer Genauigkeit und ihrer Simulationsgeschwindigkeit gegenübergestellt. Ziel dieser Arbeit ist es ein Framework, bereitzustellen, dass sich unterschiedlicher Technologien bedienen kann um dadurch ein möglichst breites Spektrum abzubilden.

Um nun eine hohe Simulationsgeschwindigkeit zu erreichen ist es notwendig diese strukturelle Genauigkeit aufzugeben. Rein funktionale Simulatoren ohne Zeitmanagement, wie beispielsweise OVP [Imp08], können Geschwindigkeiten bis hoch in einstellige Gigahertzbereiche erreichen, die im eingebetteten Bereich bereits schneller simulieren können, als ein reales System für die Ausführung benötigt. So können auf sogenannten virtuellen Plattformen Systemsimulationen mit laufenden Betriebssystemen und Anwendungen simuliert werden. Die Kosten einer solchen Simulation gehen auf die Genauigkeit sowie die Messfunktionen. Ohne jegliches Zyklen- oder Zeitmanagement kann eine Software nur auf seine Funktion simuliert werden. Die Verwendung eines zeitlich genauen Profilers, beispielsweise für eine automatisierte Parallelisierung, ist auf dieser Ebene nicht mehr möglich.

Zwischen einer RTL Simulation und einer vollständig Zeitlosen funktionalen Simulation liegen etliche Kombinationen und Verfeinerungsschritte wie sie in Abbildung 6.2 dargestellt sind. Des Weiteren besitzen beide Ebenen sehr unterschiedliche Anforderungen an die Beschreibung des Systems. Daraus ergibt

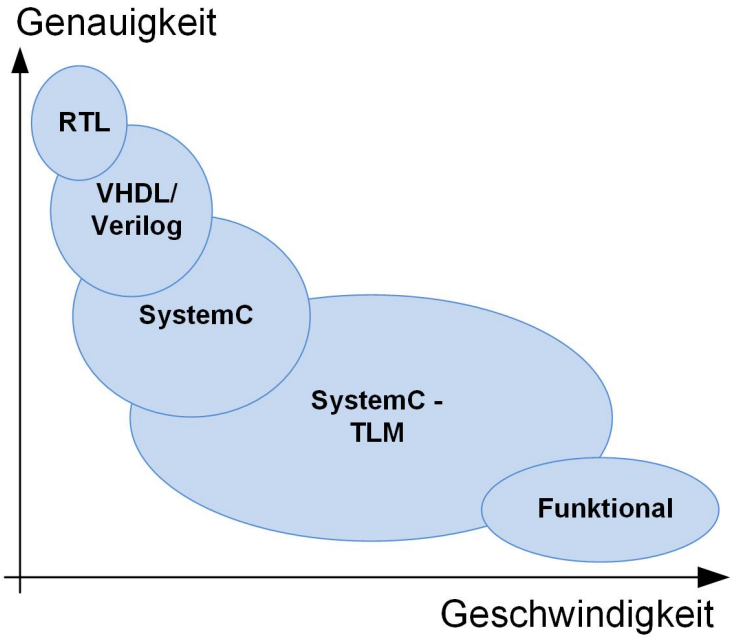


Abbildung 6.1: Verhalten unterschiedlicher Abstraktionsebenen und Simulationstechnologien hinsichtlich Genauigkeit und Geschwindigkeit einer Simulation

sich die Diskrepanz, dass analog zu den ADLs, auch hier Simulatoren für eine dedizierte Kombination des Abstraktionslevels entstanden sind. Für jeden Simulator sind eigene Simulationsmodule, eigene Beschreibungen sowie Implementierungen notwendig, die jeweils verfeinert werden müssen, wenn sie auf einen andere Abstraktionsebene gebracht werden sollen.

Gajski und Cai [CG03] haben dafür jeweils für die Domänen Computation (Berechnung) und Communication (Kommunikation) drei Basisklassen definiert von denen sich unterschiedliche Abstraktionsmodelle ableiten lassen. Die Kategorien sind in Abbildung 6.2 dargestellt, überlagert von Simulationstech-

nologien und Beschreibungssprachen, die die jeweiligen Abstraktionsmodelle abdecken. Die Modelle sind:

A "Specification model": zeitlose funktionale Modelle.

B "Component-assembly model": zeitliche funktionale Modelle.

C "Bus-arbitration model": Transaktionsmodelle

D "Bus-functional model": Kommunikations- und Verhaltensmodelle

E "Cycle-accurate computation model": akkurate Berechnungsmodelle

F "Implementation model": Register-Transfer-Modelle

Diese fehlende Flexibilität soll der MPSoC System Simulator ausgleichen, indem eine ADL verwendet wird, die die notwendige Struktur als auch Verhaltensbeschreibung für sowohl zyklenakkurate als auch funktionale Simulationssysteme bietet. Kombiniert wird dies mit einem Basissimulatorsystem, welches ein Framework für die Kombination unterschiedlichster Simulationsmodule bereitstellt, in dem mehrere Abstraktionsebenen kombiniert und mit einer gemeinsamen ADL gesteuert werden kann. So kann in allen Verfeinerungsschritten auf dieselbe Basis an Simulationsmodulen sowie auf eine gemeinsame Beschreibung zurückgegriffen werden um die Simulationsumgebung an die aktuellen Bedürfnisse anzupassen.

6.2.1 SystemC

Für die Systemsimulation mit unterschiedlichen Detailgraden hat die Open SystemC Initiative (OSCI) (seit 2011 mit Accellera zur Accellera Systems Initiative [Acc11] fusioniert) SystemC als Standard etabliert. SystemC stellt ein Framework für die Modellierung, die Simulation und die Analyse von digitalen Systemkomponenten oder ganzen Systemen dar. Implementiert als Quellcodebibliothek besteht SystemC aus einer Menge an C++ Klassen, Interfaces

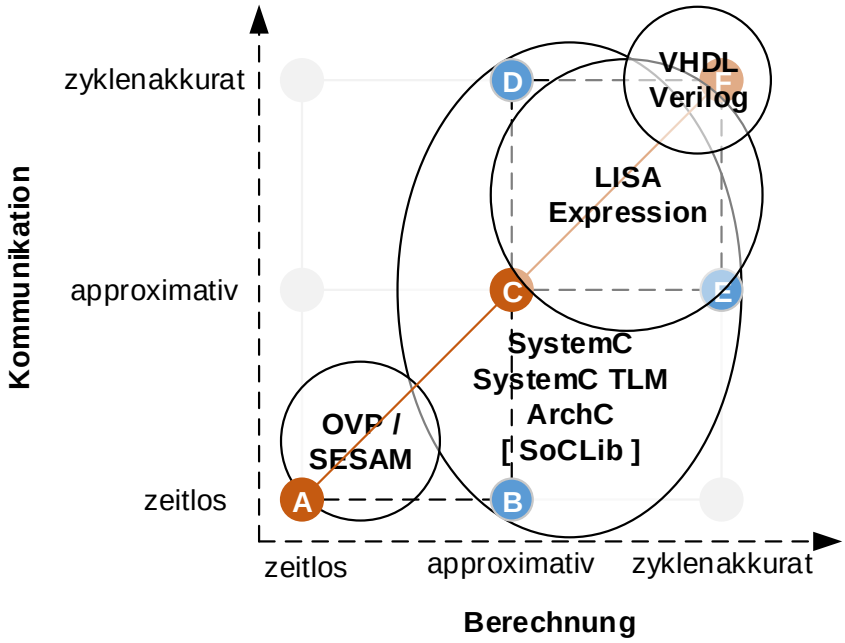


Abbildung 6.2: Darstellung der Verfeinerungsschritte einer Simulation mit Technologien und Beschreibungssprachen, nach [CG03]

sowie Makros die für die Beschreibung und Modellierung eines Systems verwendet werden. Dazu implementiert SystemC einen Simulationskernel, der die Simulation eines in SystemC beschriebenen Systems und damit weitreichende Analysen ermöglicht.

Die primäre Domäne von SystemC ist die Implementierung eines Simulationsmodells, kann aber unter Einschränkungen auch als ADL interpretiert werden um digitale Systeme zu beschreiben. In diesem Fall sind die Beschreibung als auch das Simulationsmodell miteinander vermischt. Da SystemC auf C++ aufbaut ist es primär eine Programmiersprache, so dass eine Systembeschrei-

bung immer in das Simulationsmodell integriert und somit implementiert sein muss.

Für die Simulation stellt SystemC Mechanismen und Interfaces auf verschiedenen Detailebenen bereit, die sowohl für die Simulation als auch die Systembeschreibung verwendet werden. Die möglichen Detailgrade umfassen dabei rein funktionale Beschreibungen bis hin zu zyklenakkuraten Simulationsmodellen, welche ähnlich einer Hardwarebeschreibungssprache auf Bitänderungen und Signalflankenänderungen innerhalb eines Taktzyklus reagieren können. Dabei steigt der Rechenaufwand für die Simulation mit wachsendem Detailgrad. Während rein funktionale Modelle hohe Simulationsgeschwindigkeiten erreichen, bewegen sich bitakkurate und zyklenakkurate Modelle in den Simulationsgeschwindigkeiten einer Hardwaresimulation wie beispielsweise VHDL.

Über die Implementierung von sogenannten Wrappern und Adaptern ist es möglich Übersetzungen von und in andere Detailgrade oder Berechnungsmodelle zu implementieren, wodurch diese innerhalb einer Simulation miteinander kombiniert werden können.

Die Natur der Programmiersprache C++ ermöglicht zwar die Wiederverwendung von Klassen und damit von Systemkomponenten, allerdings lassen sich Modifikationen nur im vorgegebenen Modell realisieren. Soll der Detailgrad eines Modells oder einer Systemkomponente verändert werden, so muss eine Implementierung erfolgen. Dies macht SystemC während der Design-Phase sehr flexibel verliert diese Flexibilität aber in der Implementierung eines Simulationsmodells.

6.2.1.1 Architektur Modellierung

Für die Architekturmodellierung verwendet SystemC [Sys12, Gro02] sogenannte Modules für die Beschreibung von Komponenten sowie Ports und Interfaces für die Kommunikation. Mit Hilfe von Prozessen wird das Verhalten eines Moduls beschrieben.

Modules Das Basiselement der strukturellen Systembeschreibung in SystemC ist ein Modul. Module können mit Hilfe des SCMODULE Makro oder über C++ Objektdefinitionen erzeugt werden. Das Modul erlaubt die Kapselung und die Gruppierung von Komponentenfunktionen um die Systemkomponenten zu organisieren.

Ports, Interfaces und Channels Module kommunizieren miteinander über **Ports**, über die Informationen von einem Modul zum anderen Modul übertragen werden können. Um die Kompatibilität zwischen Modulen und der Zielfunktionalität herstellen zu können, nutzt eine Verbindung ein **Interface**. Damit SystemC mit dem C++ Standard konform bleibt, stellen Interfaces abstrakte Elemente dar, welche nur Funktionen deklarieren ohne eine Implementierung bereitzustellen. Ein **Channel** wird nun von einem Interface abgeleitet und implementiert die darin deklarierten Funktionen. Ein **Port** eines Moduls wird nun durch die Implementierung eines Interfaces an einen Channel gebunden. Durch die Vererbungsmöglichkeiten in C++ ist es auch möglich mehrere Channels zu gruppieren und somit hierarchisches Channels zu erzeugen. ein **Primitive Channel** hingegen implementiert nur einen direkten Port.

Prozesse Da Hardware in der Lage ist Befehle und Operatoren auch parallel und gleichzeitig abzuarbeiten muss SystemC dieses Verhalten simulieren. Dieses Ziel wird erreicht in dem Aktivitäten als **Processes** modelliert werden, welche anschließend, ähnlich zu Hardwaresimulationssprachen wie VHDL nebenläufig bzw. im Zeitmultiplexverfahren ausgeführt werden. Die Ausführungsreihenfolge der Prozesse hängt von der Implementierung des Simulationskernels ab.

6.2.1.2 Simulationskernel

Der Simulationskernel von SystemC wurde Event-basiert entworfen und macht sich dabei eine breite Interpretation des Begriffs Event zu nutze. Sys-

temC Events können dabei entweder zeitliche Events, beispielsweise eines Timers, oder Signalevents wie die steigende flanke eines Taktsignals darstellen. Es werden aber auch SystemC interne Mechanismen bereitgestellt, um eigene Events zu definieren auf die der Simulationskernel reagieren kann. Da dadurch die Anzahl auftretender Events sehr groß werden kann, wird eine Simulation die auf alle Events reagiert entsprechend langsam. Daher wurde für SystemC ein alternatives Konzept entwickelt: SystemC-TLM [Ope09]. SystemC-TLM arbeitet dabei auf einer funktionalen Ebene und ignoriert dabei gewisse hardwarespezifische Details. Dadurch verliert SystemC-TLM zwar auch an Genauigkeit während der Simulation, gewinnt allerdings auch an Geschwindigkeit. Um SystemC und SystemC-TLM gegenüberzustellen, sei ein Speicherzugriff als Beispiel genannt. Während in SystemC alle Bit-Wechsel auf dem Kommunikationsbus berechnet und simuliert werden müssen, gibt ein Funktionsaufruf, die sogenannte Transaction, in SystemC-TLM nur das Ergebnis des Speicherzugriffs zurück. Zyklenzahlen und Ausführungszeiten können dabei berechnet, geschätzt oder vollständig ignoriert werden. Damit verschiebt sich auch der Einsatzzweck und der Use-Case von Prozessorarchitekturen und Hardwareschaltkreisen hin zu Funktionsblöcken und ganzen Systemen oder sogenannten Virtuellen Plattformen in der mehrere IP Blöcke modelliert und simuliert werden können.

SystemC und SystemC-TLM erlauben die Modellierung und Simulation von Systemen auf unterschiedlichen Genauigkeitsebenen (im SystemC Standard [Sys12] auch **Coding Styles** genannt). Dazu zählen:

Untimed Dies ist das schnellste aber auch ungenaueste Simulationslevel. Ein System welches als **untimed** modelliert wird stellt dabei keine bis kaum Zeitinformationen zur Verfügung, deren Genauigkeit wiederum nicht definiert ist. **Untimed** Modelle können daher nur für die schnelle funktionale Verifikation von Software oder Hardware verwendet werden.

Loosely-Timed In loosely-timed Systemen werden Transaktionen zwar blockierend ausgeführt, ein sogenanntes **Temporal Decoupling** ermöglicht allerdings die Beschleunigung der Simulation. Die Zeitliche Entkopplung sorgt dafür, dass Simulationsmodule bis zu einem gewissen Zeitquantum der globalen Simulationszeit vorauslaufen können um unnötige Synchronisationsintervalle zu vermeiden und damit den Switching Overhead in komplexeren Systemen zu reduzieren.

Approximately-timed Diese Modellierungsebene basiert auf nicht-blockierenden Transaktionen welche allerdings nicht von einer zeitlichen Entkopplung profitieren können. Dadurch wird eine höhere zeitliche Genauigkeit erreicht als bei einer **loosely-timed** Simulation.

Cycle-accurate auf der genauesten aber damit langsamsten Modellierungsebene stehen sowohl blockierende als auch nicht-blockierende Funktionsaufrufe zur Verfügung. Diese richten sich allerdings strikt nach dem SystemC Standard und erlauben keine Verwendung von TLM.

SystemC erlaubt die Verwendung mehrerer Coding-Styles innerhalb einer Simulation, wodurch die Verfeinerung eines Moduls in ein genaueres Modell modular und schrittweise bei gleichzeitiger Verifikation der Module untereinander erfolgen kann.

6.2.2 Simulationstechniken

Für die eigentliche Simulation wurden bereits mehrere Frameworks vorgeschlagen. Während viele Frameworks auf SystemC aufbauen [BBB+05] [MKB+11], basieren andere auf eigens entwickelten C-basierten Simulationen [CGH+08] oder Khan Prozess Netzwerken um die Anwendung zu modellieren [HBLH09] [CHB09].

In [ANMD07] wird ein Simulationsansatz basierend auf TLM Unterebenen vorgeschlagen um eine Performanceabschätzung durchzuführen. Allerdings gibt

es keine Aussagen über System bzw. Architekturbeschreibungsmöglichkeiten oder über eine iterative Applikationsoptimierung.

Um einen Kompromiss zwischen Geschwindigkeit und Genauigkeit innerhalb von TLM Modellen basierend auf SystemC 2.0 zu erzielen wird in [BSS+06] ein Multi-Level Ansatz vorgeschlagen. Dabei wird für die Kommunikation ein Konzept implementiert, welches mehrere Abstraktionsebenen vorhält und durch spezielle Komponenten innerhalb des SystemC Channels Entscheidungsmöglichkeiten zur Wahl der Implementierung zur Verfügung stellt. Eine abstrakte Beschreibung des Systems existiert nicht, weshalb alle Alternativen direkt mit in das SystemC Modell implementiert werden. Dadurch wird die Beschreibung des Systems extrem Komplex, was eine automatisierte Architekturanalyse auf Strukturebene erheblich erschwert.

Mit ReSP schlagen Beltrame et. al [BFS09, BBF+08, MKB+11] einen Ansatz vor, der es erlaubt mit Hilfe der Skriptsprache Python Architekturen zu beschreiben, welche an einen SystemC Simulator gekoppelt werden. Durch eine automatische Integration von Python in C++, welches die Basis für SystemC darstellt, lassen sich Simulatorinstanzen unkompliziert in die Simulationsumgebung integrieren, ohne dass Wrapper-Klassen für den Simulator notwendig sind. Dazu wird ein SystemC IP Core mit Hilfe eines Python Skriptes analysiert und die Funktionen anschließend in C++ bereitgestellt, woraus eine SystemC main Datei zur Simulation erstellt wird. Dieses Vorgehen ist besonders für automatisierte simulationsbasierte Entwurfsraumexplorationen hilfreich, bei der auch Softwarekomponenten in SystemC modelliert werden.

In [FQS08] beschreiben Fummi et al. ein auf SystemC TLM basierendes Framework zur Netzwerksimulation. Der Fokus des SCNSL genannten Simulationsframeworks liegt auf der Modellierung von vernetzten eingebetteten Systemen, wodurch sich beispielsweise auch kabellose Sensornetzwerke simulieren lassen. Die Architektur von SCNSL besteht aus einem Simulationskernel basierend auf SystemC für die Netzwerksimulation und der Beschreibung von Nodes, welche eingebettete Systeme, Systemkomponenten oder Prozessoren dar-

stellen können. Dazu kommen Ports und Channels, welche die Nodes miteinander verbinden und damit das eigentliche Netzwerk darstellen, sowie Packets um die Dateneinheit zu modellieren, welche zwischen Nodes ausgetauscht wird. Um die Netzwerksimulation zu beschleunigen wird die Beschreibung der Node vom Netzwerkframework entkoppelt, wodurch große Synchronisationsintervalle erzielt werden können. Innerhalb des Frameworks ist auch die Kombination von RTL oder TLM Modellen möglich. Eingeschränkt wird das Framework durch den Fokus auf Netzwerkkomponenten, was eine Systemweite Analyse des Systems inklusive der Verarbeitungseinheiten erschwert.

Virtuelle Plattformen, welche wie OVP [Imp08] oder HySim [GKK+08] auf nativ kompilierten Code zurückgreifen, erreichen hohe Simulationsgeschwindigkeiten. Dabei ist allerdings zu beachten, dass die Synchronisation zwischen dem nativ ausgeführten Simulationscode und dem simulierten Prozessor zu starken Genauigkeitsverlusten führen kann. Durch Entkopplung des nativen Codes und des spezifischen ISS lässt sich zwar die funktionale Korrektheit [MEJ+12] der Simulation verbessern, allerdings sind kaum Timing-Informationen über die internen Strukturen des MPSoC Systems verfügbar. Somit lassen sich parallele Anwendungen hervorragend funktional testen, aber für die Systemanalyse zur automatisierten Parallelisierung sind diese weniger geeignet.

Zur weiteren Beschleunigung von Simulationen werden verschiedene Ansätze vorgeschlagen. Der in [QD11] angewandte Ansatz verwendet Cuda [NBGS08] zur Beschreibung von RTL Tasks um diese parallel auf Graphics Processing Units (GPUs) auszuführen. Die Ausgangsbasis dafür stellt ein HDL Code in Verilog [IEE06] dar, welcher auf CUDA Funktionen gemappt wird. Der Simulator wird anschließend vollständig in CUDA implementiert und führt damit die HDL Simulation durch. Eine derartige Simulation erlaubt damit allerdings nur die Verwendung von RTL Code, weshalb die Flexibilität zum Wechseln des Abstraktionslevels nicht gegeben ist. Darüber hinaus lässt sich damit keine binärkompatible Simulation durchführen, wie sie für die Parallelisierung notwendig ist.

6.2.3 ArchC

Eine ArchC Systembeschreibung kann dazu verwendet werden um automatisiert einen ISS sowie die zugehörige Toolchain, bestehend aus Assembler, Linker und C++ Compiler Backend, zu erzeugen. Das Zielsystem kann dabei entweder auf funktionaler Ebene oder auf zyklenakkurater Ebene erzeugt werden und können, wie in [MKB+11] beschrieben, auch kombiniert werden. Allerdings sind keine weiteren Simulationsebenen außer funktional oder zyklenakkurat definiert.

ArchC beinhaltet dafür den Interpretierenden Simulatorgenerator `acsim`, welcher die Verhaltensbeschreibung einer ArchC Architekturbeschreibung verwendet um einen Simulator zu generieren. Eine schnellere Variante, `accsim`, kompiliert direkt den ArchC C++ Quellcode und setzt dabei auf Präprozessoren um die Simulation zu beschleunigen.

Um Rapid-Prototyping mithilfe von ArchC zu ermöglichen steht das Platform Designer Framework [AGB+05] zur Verfügung, welches Werkzeuge für die Entwurfsraumexploration bereithält. Zu den Werkzeugen zählen die beiden SystemC Simulatoren, Werkzeuge für die Modellierung mit ArchC sowie Analytische Plugins für die Eclipse Integrated Development Environment (IDE) [Ecl13] sowie eine Komponentenbibliothek.

6.2.4 SoClib und DSX

Das Ziel von SoClib ist die Bereitstellung einer umfangreichen Menge an IP Cores um MPSoC Systeme zu erstellen und zu testen. Werkzeuge rund um SoClib bestehen aus einem zyklen- und bitakkuraten Simulationskernel (SystemCASS [BG07, BG10]) und dem Design Space Explorer (DSX [Des11, Mwm09]). Das zentrale Element der SoClib Plattform ist eine Bibliothek an Central Processing Unit (CPU) IP Cores jeweils mit einem ISS für die Prozessorarchitektur. neben den Prozessorkernen stellt SoClib auch Simulationsklassen

für Speicher, Kommunikationselemente, Coprozessoren oder diverse Controller zur Verfügung. Die Simulationsklassen verwenden dabei zwei unterschiedliche Abstraktionsebenen:

CABA (Cycle-Accurate / Bit-Accurate) Herkömmliche zyklenakkurate SystemC Modelle erlauben keine Simulation von Signaländerungen innerhalb eines Taktzyklus. Das CABA Modell aus SoClib beachtet dabei auch Events die innerhalb eines Taktzyklus auftreten und macht dadurch auch Bitänderungen sichtbar.

TLM-DT (TLM with Distributed Time) Dieses spezielle Modell von SystemC-TLM ist für eine verteilte sowie Multi-Threaded Simulation [MMGP10, PMGP10] von MPSoC Systemen konzipiert worden.

Ein weiteres wichtiges Werkzeug der SoClib Plattform ist **soclib-cc**, welches es erlaubt Systemkomponenten für SoClib zu erstellen oder Informationen aus den Metadaten der Module auszulesen. Der SoClib Design Space Explorer (DSX) nutzt nun die SoClib Bibliothek sowie die Modulbeschreibungen um Systeme zu simulieren und Anwendungen auf die Komponenten zu platzieren.

6.2.5 Kahrisma Simulator

Die Funktionalität des Kahrisma MPSoC Simulators innerhalb des ALMA Framework ist getrennt in die Aufgaben Systemgenerierung und Systemsimulation. Abbildung 2.1 zeigt die beiden Bestandteile: den ADL Compiler (siehe Kapitel 5) und den Multi-Core Simulator, welche beide im Rahmen dieser Arbeit beschrieben werden.

Die Basis für diese Simulation und Beschreibungssprache stellt ein für KAHRISMA entwickelter Mehrkernsimulator [San11] dar, der es ermöglicht, aus einer einfachen Systembeschreibung mehrere KAHRISMA Simulatorinstanzen zu generieren und als Multicore-System zu simulieren

Der KAHRISMA Prozessorkernsimulator selbst stellt einen zyklenapproximativen ISS [SKB12] für die Architektur zur Verfügung, welcher aus der KAHRISMA-ADL konfiguriert werden kann.

Er unterstützt dabei die Simulation der KAHRISMA-mixed-ISA Simulation bereits innerhalb eines KAHRISMA Prozessorsimulator, so dass dafür kein eigener Simulator entwickelt werden muss. Für eine tatsächliche Multicoresimulation muss hierfür aber noch ein Ressourcenmanagement und Konfigurationsmanagement entworfen werden, da dieses nur rein funktional zur Verfügung steht und bei größeren Arrays für das Mapping von Anwendungen auf dem MultiCore Array zu Problemen führt.

Um ein System simulieren zu können, welches auch Softwareanwendungen ausführen kann, werden allerdings weitere Module wie Speicher und Kommunikationsinfrastruktur benötigt.

6.3 Aufbau des MultiCore Simulators

Im Simulator wird eine Modul-Bibliothek implementiert, welche die unterschiedlichen Simulationsklassen aufnimmt. Eine Basisklasse sorgt für die Systemgenerierung und Erzeugung, sowie die Anbindung des System Profilers. Des Weiteren baut der Simulator auf SystemC auf. Der Simulationskernel von SystemC wird dazu verendet die Simulation zu steuern und zu kontrollieren. Das Zeitmanagement der Simulation übernimmt SystemC. Insgesamt besteht das Framework aus den folgenden Komponenten deren Zusammenspiel in Abbildung 6.3 dargestellt ist.

MultiCore Simulator Der Simulator integriert Basisklassen zur Systemerzeugung und zur Verbindung der einzelnen Module sowie die SystemC-Laufzeitumgebung. In der Basisklasse werden außerdem alle Simulationsoptionen und Steuerungsfunktionen implementiert. Dazu gehört

auch die Integration des ADL-Compilers für die Verarbeitung der Architekturbeschreibung.

Simulationsklassenbibliothek Diese Bibliothek umfasst alle Simulationsklassen der einzelnen Module. Diese Klassen implementieren SystemC Schnittstellen sowie die Simulatoren der einzelnen Module.

SystemC Simulationskernel Der Simulationskernel steuert des Zeitverhalten des Systemsimulators und dient der Synchronisation der einzelnen Simulationsklassen.

ADL-Compiler Der ADL-Compiler, siehe Kapitel 5, verarbeitet die Architekturbeschreibung und stellt dem Simulationsframework eine Zwischendarstellung der ADL zur Verfügung.

System Profiler Der Profiler, siehe Kapitel 7, wird direkt in den Simulator integriert. Dadurch können die Profiling-Funktionen aus den Simulationsklassen heraus angesprochen werden.

Die klare Trennung der Compiler Funktionalität, welche im ADL-Compiler implementiert wurde, sowie die Funktionen, welche für die Simulation eines MPSoC System benötigt werden, führt zu einem kleinen Simulationskernel der von der Vorverarbeitung des Systemgenerators abhängt. Das Simulationsmodul für den KAHRISMA Prozessor wurde als Wrapper um den KAHRISMA Prozessorarchitektursimulator implementiert und erhielt dazu ein Laufzeitsimulationsmodul. Während des Startup der Simulation, instanziiert der Simulator alle Simulationsmodule, wie Verarbeitungseinheiten, Speicher, Netzwerke etc., basierend auf den Ergebnissen des ADL-Compiler. Die einzelnen EDPE Prozessoreinheiten der KAHRISMA Architektur werden von dessen Laufzeitmodul verwaltet, wodurch die ADL Konfigurationen und damit die Rekonfigurierbarkeit des Systems auch während der Simulation unterstützt wird. Entsprechend der Verbindungsinformationen der ADL werden auch die Simulationsmodule über SystemC Bindings verbunden. Die von der ADL angebotene

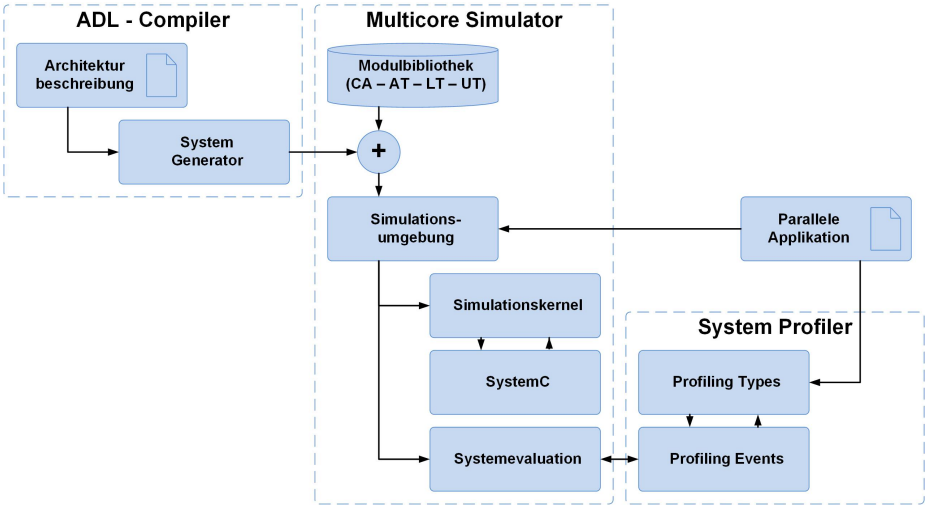


Abbildung 6.3: Aufbau des System Simulators

Flexibilität macht es dabei notwendig SystemC Module Ports zu deaktivieren, wenn sie in einer Systembeschreibung nicht aufgeführt sind.

Die SystemC Simulationsklassen des Simulators können vollständigen Gebrauch aller in der ADL bereitgestellten Parameter machen und sogar eigene Parameter definieren, welche als Modulparameter in der `Simulation` Beschreibung der ADL zugänglich gemacht werden können.

Dies wird ermöglicht durch die Einbindung der ADL-Objekte aus dem ADL Compiler in die Simulationsumgebung. Das in Abbildung 6.4 dargestellte Klassendiagramm zeigt die Assoziationen zwischen der rot dargestellten ADL und der SysSim Basisklasse sowie einer ADLInstance und dem jeweiligen SysSim-Module in blau, von welchem die systemspezifischen Simulationsmodule abgeleitet werden.

Durch diese flexible und umfassende Parametrierbarkeit der Simulationsmodule können diese auch als Instanzen über die ADL stark individualisiert wer-

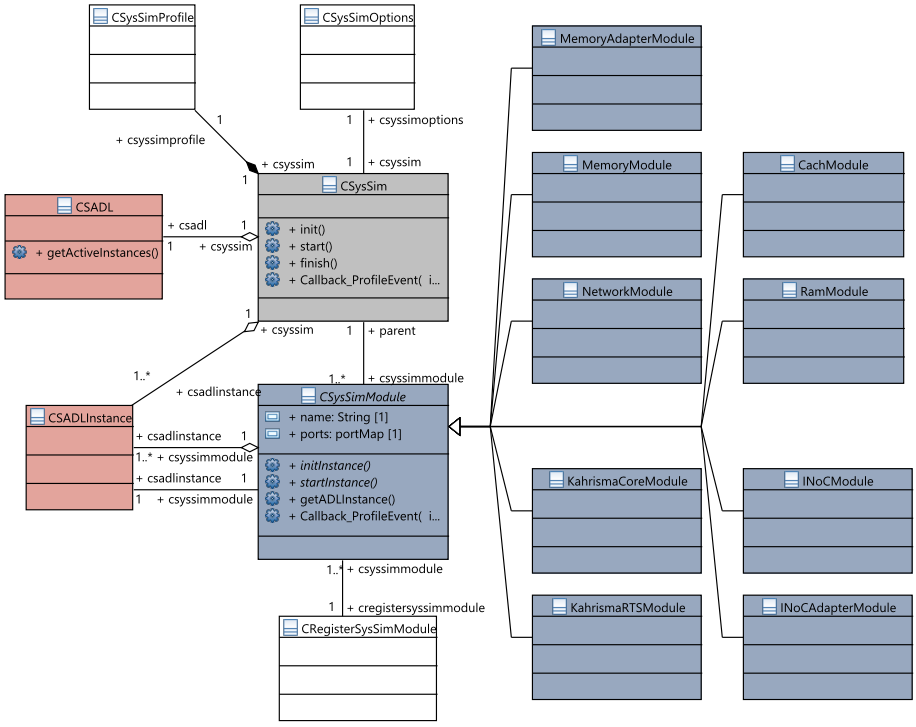


Abbildung 6.4: Vereinfachtes Klassendiagramm des SystemSimulators

den, wodurch der Entwicklungsaufwand für Simulationsmodule stark reduziert wird.

6.4 Multicore Simulator mit Simulationskernel

Innerhalb des Multicore Simulators wird SystemC verwendet und die Verbindungen zwischen den Simulationsklassen zu realisieren sowie um die Simulation durch den Simulationskernel von SystemC zu steuern. SystemC implementiert einen Event-basierten Simulationskernel als C++ Bibliothek, welche in

C++ Projekte eingebunden und somit in C++ Anwendungen verwendet werden kann. Dadurch ist es mit SystemC möglich sowohl Hardware als auch Softwarekomponenten oder eine beliebige Kombination daraus zu simulierten. Die Simulation ist aufgebaut mit SystemC-Modulen, Ports, Interfaces und Channels welche die Systemkomponenten und deren Kommunikation beschreiben. Jedes dieser Elemente kann unterschiedliche Genauigkeitsstufen von bit- und zyklenakkurater Simulation und Berechnung bis hin zu vollständig zeitloser Simulation implementieren. Darüber hinaus können für die Kommunikation Signale, vergleichbar mit Hardwareleitungen, und Funktionsaufrufe mit gekapselter Funktionalität implementiert werden.

In allen Implementierungsvarianten, außer der vollständig zeitlosen Simulation, verwendet SystemC eine globale Zeit zur Synchronisation der einzelnen Module. Diese Zeit wird durch die Simulation-Events gesteuert und unterstützt auch die Ausführung von Delta-Cycles, als sich direkt auswirkende Änderungen von Werten auf Grund von Nebenläufigkeiten der simulierten Hardware

Der Simulationsablauf in SystemC besteht aus einer Initialisierungsphase sowie aus einer sich wiederholenden Schleife aus einer Evaluationsphase, einer Aktualisierungsphase sowie einer Zeitfortschreibungsphase. Innerhalb der Initialisierung übernimmt SystemC die Aufgaben Module zu instanziieren und Laufzeitobjekte zu erstellen. Darüber hinaus werden die Ports und Interface-Objekte miteinander verknüpft sowie initiale Variablenwerte gesetzt.

Die Simulationsschleife beginnt jeweils mit einer Evaluationsphase in der die Signal- und Variablenwerte der Module ausgewertet und die dadurch anzustoßenden Prozesse bestimmt werden. Diese Prozesse werden in der Updatephase ausgeführt und bedingen Wertänderungen auf Signalen und Variablen. Anschließend erhöht SystemC die globale Simulationszeit. Eine Ausnahme davon stellen sogenannte Delta-Cylces dar, welche eine Wiederholung der Evaluations- und Updatephase darstellen ohne die globale Simulationszeit zu

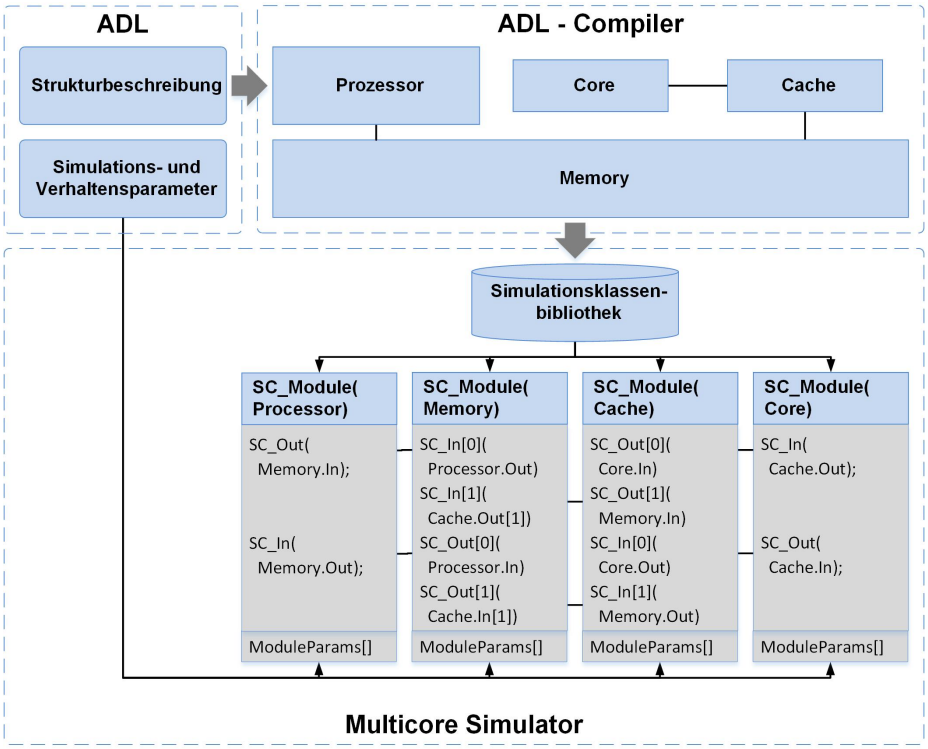


Abbildung 6.5: Aufbau der SystemC Simulationsklassen aus der ADL Beschreibung und Übergabe der Verhaltensparameter

erhöhen. Dies kommt vor, wenn durch eine Updatephase Signalwerte oder Variablen verändert werden auf die ein SystemC Prozess sensitiv reagiert. Dieser wird nun erneut ausgeführt ohne die Simulationszeit zu erhöhen und bedingt dadurch eine erneute Evaluations- und Updatephase.

6.5 Simulationsablauf

Der allgemeine Simulationsablauf des Multicoresimulators ist unterteilt in sieben einzelne Schritte, die nacheinander ausgeführt werden. Dieser Abschnitt beschreibt die einzelnen Simulationsschritte welche in Abbildung 6.6 dargestellt sind.

6.5.1 Framework Instanziierung

In dieser ersten Phase wird durch die Hauptklasse des Frameworks die Simulationsklasse SysSim initialisiert. Dabei werden die Simulationsoptionen, welche über Command-Line-Optionen oder Konfigurationsdateien übergeben werden geparst und in der Simulationsklasse als eigenständiges Objekt abgelegt. In einem zweiten Schritt wird der ADL-Compiler aufgerufen und erhält die in den Optionen angegebene ADL-Datei als Eingabeparameter. Durch den Kompilierprozess wird eine Speicherrepräsentation der Architekturbeschreibung erstellt und der Simulationsumgebung zur Verfügung gestellt. Der dritte Schritt der Framework Initialisierung instanziiert den SystemProfiler. Die zur Verfügung stehenden Profiling-Typen und Profiling-Events werden im Profiler registriert und aktiviert.

6.5.2 Simulationsmodulinstanziierung

Mit Hilfe der ADL Speicherrepräsentation führt das Simulationsframework eine Instanziierung der Simulationsmodule aus. Die in der ADL angegebenen aktiven Simulationsmodule werden durch das Simulationsframework ausgelesen und dazu verwendet die notwendigen SystemC Klassen aus der Simulationsbibliothek zu instanziieren. Sind die SystemC Klassen instanziiert leitet das Framework die Modul und Instanzspezifischen Simulationsoptionen, siehe B.3.2, an die SystemC Klasse weiter.

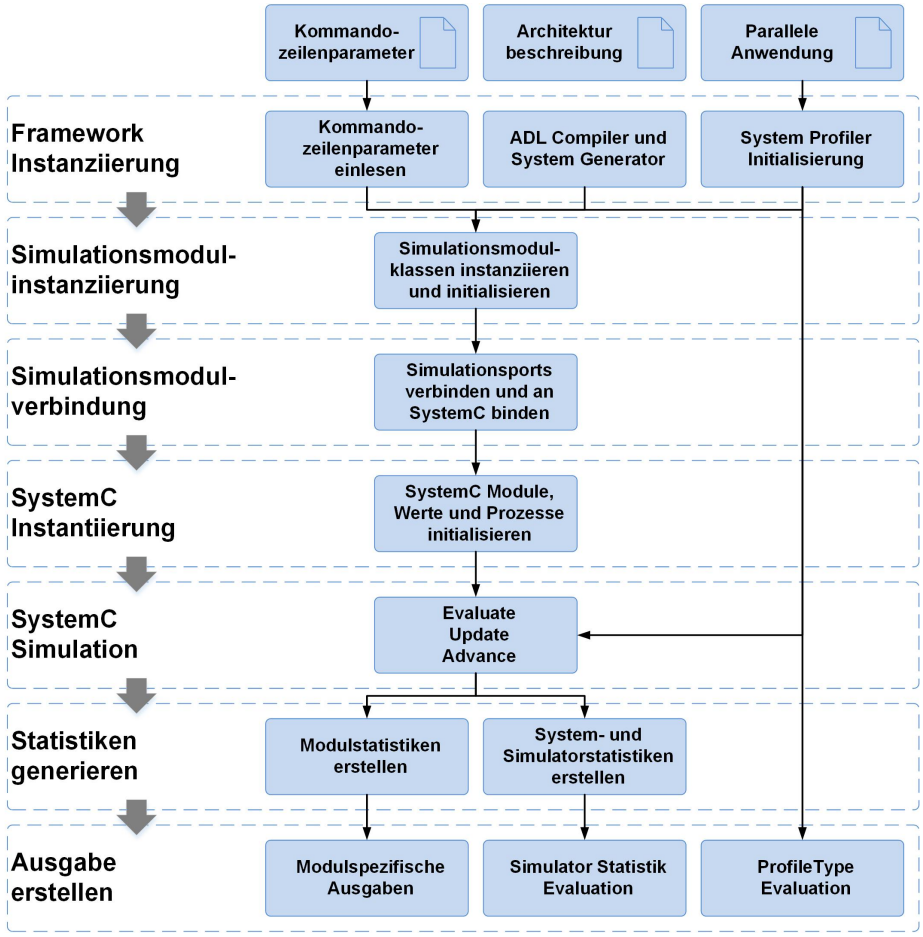


Abbildung 6.6: Ablauf einer Simulation

6.5.3 Simulationsmodulverbindung

Nachdem die SystemC Klassen instanziiert wurden, können die Ports der SystemC Module miteinander verbunden werden. Dazu werden die Architekturverbindungen, sowie die zugehörigen Ports, aus der ADL ausgelesen. Um eine

korrekte SystemC Simulation zu ermöglichen müssen diese Ports an SystemC Bibliotheksfunktionen gebunden werden, die die Kommunikationssimulation übernehmen. Alternativ können hierfür eigene Interfaceklassen in SystemC implementiert werden, welche die in der ADL beschriebenen Eigenschaften abbilden. In diesem Schritt können Inkonsistenzen auftreten, falls die Simulationsklassen nicht miteinander kompatibel sind. Die SystemC Laufzeitumgebung stellt diese Fehler fest, so dass sie korrigiert werden können.

6.5.4 SystemC Instanziierung

Sind alle SystemC Simulationsklassen korrekt miteinander verbunden, übernimmt die Simulationsklasse SysSim den Simulationsstart indem es den SystemC Simulationskernel anstößt. In dieser Phase instanziiert dann der Simulationskernel die notwendigen SystemC Module, Ports, Interfaces sowie Variablen und Signale um die eigentliche Systemsimulation vorzubereiten.

6.5.5 SystemC Simulation

Nun beginnt die eigentliche System Simulation, die vom SystemC Kernel ausgeführt wird. Dieser beginnt mit der Ausführung der Simulationsschleife aus Evaluation und Update wie in Abschnitt 6.4 beschrieben. Jedes Simulationsmodul implementiert dabei einen SystemC Prozess, sodass jedes Modul durch den SystemC Scheduler entsprechend seiner Sensitivität aufgerufen wird. Die Prozesse sind so implementiert, dass sie auf die Ports des SystemC Moduls und damit auf die Ports des ADL Moduls sensitiv reagieren.

Wurde ein Modul samt seinem Prozess ausgeführt, so wird es in einen Wartezustand gesetzt, aus dem es durch den Scheduler wieder aktiviert wird, sobald sich ein entsprechender Port-Wert geändert hat. Die durch die Ausführung des

Prozesses ermittelten Variablen und Signalwerte, welche an die Ports des Moduls angelegt werden, stehen nach der Updatephase auch den anderen Simulationsmodulen zur Verfügung.

Wie zuvor beschrieben kann es hierbei auch zur Ausführung von Delta-Cycles kommen, wenn Module direkt sensitiv auf diese Änderungen reagieren. Wurden alle Simulationsmodule ausgeführt und findet sich kein weiterer aktiver Prozess mehr, so erhöht SystemC die Simulationszeit und stößt die nächste Simulationsrunde an.

Während der Simulation, kann ein Simulationsmodul einen Profiling-Callback in das Simulationsframework ausführen. Da SystemC die Ausführung von regulären C++ Code während eines Simulationszyklus erlaubt, hat der Funktionsaufruf in den Profiler des Frameworks keine Auswirkung auf die eigentliche Systemsimulation oder die Simulationszeit und erlaubt dadurch die Sammlung von Modul- und Anwendungsinformation zur Laufzeit.

Die Simulation kann durch reguläre als auch irreguläre Gründe gestoppt werden. Eine erfolgreiche und regulär beendete Simulation wird dadurch erreicht, dass alle Simulationsmodule korrekt ausgeführt und beendet werden ohne den Simulationskernel über einen Fehler zu benachrichtigen. Eine Alternative ist ein expliziter Stop-Aufruf durch eines der Module um den Simulationskernel zu beenden. Durch einen solchen Stop-Aufruf wird die Simulation direkt im Kernel angehalten, so dass direkt auf den letzten Zeitstempel des Simulationskernels zugegriffen werden kann, wodurch die Simulationszeit bestimmt werden kann. Daher wurde der Kernel derart implementiert diesen Zeitstempel auch bei vollständiger Ausführung der Module zu erstellen und zu ermitteln. Wird kein Stop-Aufruf durchgeführt überprüft der Simulationskernel die einzelnen Module auf Aktivität und die Erstellung von Simulationsevents, welche Wertänderungen beinhalten können. Sind keine Events mehr vorhanden, so prüft der Kernel über eine definierbare Wartezeit hinweg die Simulation auf neue Events und erklärt nach deren Ablauf die Simulation für beendet.

Dies kann allerdings auch auftreten wenn sich die Simulationsmodule in einer Deadlock Schleife befinden. Um dies zu erkennen müssen die Simulationsmodule eigene Mechanismen implementieren um den Simulationskernel bei Beendigung der Simulation über den Fehlerzustand oder noch ausstehende Benachrichtigungen auf die ein Modul wartet zu unterrichten. Andernfalls ist der Kernel nicht in der Lage einen Deadlock zu erkennen. Eine weitere Möglichkeit ist das Ausführen eines Stop-Aufrufs durch ein Modul, sobald ein Fehler oder Deadlock erkannt wurde. Der Fehlercode kann dann an den Simulationskernel übergeben werden.

6.5.6 Statistiken generieren

nachdem die Simulation beendet wurde übernimmt das Framework wieder die Steuerung über den weiteren Verlauf. Dieser besteht darin, die Simulationsergebnisse der einzelnen Module zu sammeln und Simulationsstatistiken zu generieren. Dazu gehören mehrerer Funktionen innerhalb des Frameworks. Eine Callback Funktion des Frameworks ruft dazu die einzelnen Simulationsklassen auf und fragt die darin generierten Statistikdaten, wie die letzte aktive Simulationszeit, die Anzahl ausgeführter Prozessorbefehle oder den Fehlerzustand des Moduls ab. In Anschluss daran werden auch weitere registrierte Statistikfunktionen aufgerufen um modul- oder instanzspezifische Information abzufragen. Bei einer Netzwerksimulation können hier beispielsweise dessen Auslastung, durchschnittliche Paketlänge usw. vom Simulationsklassenentwickler implementiert und dem Framework zur Verfügung gestellt werden.

6.5.7 Ausgabe erstellen

Die im vorherigen Schritt generierten und vom Framework gesammelten Informationen werden in einer Baumstruktur abgelegt und im JSON Format abgespeichert. Dadurch stehen sie nach der Simulation weiteren Werkzeugen wie

der automatisierten Parallelisierung oder auch dem Anwendungs- und Systementwickler zur Analyse bereit. Des Weiteren werden in dieser Simulationsphase die Profiling Daten, ebenfalls im JSON Format, erstellt. Dazu ruft das Simulationsframework den System Profiler auf. Dieser ruft die registrierten Profiling Types nacheinander auf, wodurch jeweils eine Ausgabedatei mit den Typ-Spezifischen Daten erzeugt wird. Die Möglichkeiten des System Profilers sind in Kapitel 7 genauer beschrieben.

6.6 Hierarchie innerhalb des Simulators

Aus der Sicht des Multicore Simulators spielt die hierarchische Struktur der Architekturbeschreibung nur eine untergeordnete Rolle, da die Hierarchie schon durch den ADL-Compiler aufgebrochen wurde. Der Simulator erhält eine flache, hierarchielose Beschreibung um die Simulationsumgebung zu erstellen.

Diese Beschreibung besteht nur aus den aktiven Instanzen der Architekturbeschreibung und deren zugehörigen SystemC Simulationsklassen.

Einzig die erstellten Modul- und Instanznamen lassen einen Rückschluss auf die hierarchischen Beziehungen zu. Die Instanzen sind allerdings vollständig evaluiert und für die Simulation vorbereitet worden.

Auch die Verbindungen werden vom ADL-Compiler entsprechend angepasst und im Auflösungsprozess der hierarchischen Struktur repariert indem Verbindungen ersetzt werden. welche zuvor zwischen deaktivierten Modulen und Kind-Instanzen existiert haben. Am Ende des hierarchischen Auflösungsprozesses existieren somit nur noch Verbindungen zwischen Instanzen, welche in einer SystemC Klasse verfügbar sind.

6.7 Timing Parameter

Die Architekturbeschreibung erlaubt die Angabe von Laufzeitparametern und System- bzw. Modulfrequenzen. Diese werden dazu verwendet die Systemgeschwindigkeit zu ermitteln und können mit der globalen SystemC Simulationszeit kombiniert werden. Unterschiedliche Modulfrequenzen werden zudem zur Synchronisation der Simulationsklassen und Simulationsmodule verwendet.

Da die globale SystemC Zeit unabhängig von realen Zeiten ist, muss über diese Frequenzen die im System simulierte Zeit berechnet werden. Primär werden zunächst die Taktzyklen der einzelnen Module über die globale SystemC Zeit ermittelt und anschließend die Simulationszeit mit Hilfe der in der Architekturbeschreibung angegebenen Modulfrequenzen berechnet.

Um eine korrekte Synchronisation zwischen Modulen mit unterschiedlichen Frequenzen während der Simulation zu erreichen ermitteln Module mit von der Systemfrequenz abweichenden Frequenzen das Frequenzverhältnis und fügen falls notwendig zusätzliche Wartezyklen ein, damit die Reaktionszeiten mit der globalen Systemfrequenz berechnet werden können. Insgesamt arbeitet das SysSim Modul, welches die SystemC Basisklasse der Simulation darstellt, nach dem Approximately-Timed Simulationsmodell von SystemC TLM.

Dieses Berechnungsmodell erlaubt die Definition modulspezifischer Timing-Parameter, welche anschließend über TLM synchronisiert werden. Dadurch ist es möglich auch unterschiedliche Timing-Modelle innerhalb einer Simulation zu implementieren und zu verwenden, wodurch eine Mixed-Level Simulation erreicht wird.

Der Befehlssatz-Simulator der KAHRISMA Architektur verwendet ein Approximatives Zeitmodell (Approximately-Timed (AT)), welches sehr eng mit dem approximativen TLM-AT-Level (siehe Abschnitt 6.2.1.2) verbunden ist. Um nun die Simulationsgeschwindigkeit adaptiv anpassen zu können, unterstützt

der KAHRISMA-Simulator eine sogenannte *yield-time*. Diese erlaubt es einer Kernsimulator-Instanz gegenüber anderen Simulationsinstanzen in der Zeit voranzuschreiten ohne eine Synchronisation durchführen zu müssen. Mit dieser *yield-time* wird der Parameter *run-ahead-cycle* definiert. Durch die Veränderung der *yield-time* wird das Synchronisationsverhältnis zwischen den ISS-Modulen verändert. Wird die *yield-time* verlängert, so verlängert sich das Synchronisationsintervall und damit verringert sich der Switching Overhead während der Simulation. Dadurch kann die Simulation deutlich beschleunigt werden, verliert aber auch an Genauigkeit, da Taktzyklen weniger akkurat approximiert werden können. Für die Modellierung dieses Parameters wird in der ADL der `CyclesPerRun` Parameter dem `Simulation` Block eines Moduls als modulspezifischer Parameter hinzugefügt. Nun kann innerhalb der Architekturbeschreibung dieser Parameter für eine adaptive Simulation innerhalb einer extensiven Entwurfsraumexploration verwendet werden um beispielsweise aussichtsreiche Kandidaten genauer und nicht so aussichtsreiche Kandidaten schneller zu simulieren.

Kapitel 7

ADL-adaptives Profiling und Systemevaluation

Dieses Kapitel beschreibt den SystemProfiler des Frameworks. Dabei wird auf dessen Aufbau und die Komponenten, sowie die Funktionsweise eingegangen.

7.1 Evaluation und Analyse

Für die Evaluierung und Messung von Software gibt es verschiedene Verfahren und Techniken für unterschiedliche Ziele und Aufgaben. Dieses Kapitel soll einen kurzen Überblick darüber verschaffen, welche Analyseverfahren für welche Aufgabenbereiche geeignet sind und welchen Einsatzzweck sie erfüllen. Dadurch lassen sich die Funktionen des SystemProfilers einordnen und klassifizieren.

7.1.1 Debugging

Diese wohlbekannte Technik ist das Grundgerüst der Softwareanalyse und dient hauptsächlich zur Fehlersuche sowie Fehleranalyse. Für eine genaue Analyse werden beim Kompilieren des Programms sogenannte Debug-Informationen im Code belassen. Sie dienen der Lesbarkeit des Codes und erfüllen im übersetzten Programm eigentlich keine Funktion mehr. Mit Hilfe dieser Informationen lassen sich allerdings Verknüpfungen zwischen kompiliertem Code und Quelldateien erschaffen, da sie Angaben darüber machen, zu welchem Programmteil im Quellcode der vorliegende kompilierte Code gehört. Mit der schrittweisen Ausführung des Quellcodes können nun interne Abläufe analysiert und auf ihre Korrektheit hin überprüft werden.

7.1.2 Profiling eingebetteter Systeme

Profiling beschreibt die Erstellung eines definierten Profils eines Systems oder einer Software. Ein solches Profil kann die Ausführung von Funktionen und die durch Funktionen benötigte Zeit oder Ressourcen beschreiben. Es kann sich dabei aber auch um ein Speichernutzungsprofil [HSW07], Cache-Zugriffsstatistiken [Sew04], ein Energieprofil [Hüb09] des Systems oder andere Profile handeln.

Profiling im Bereich der Ausführung von Software oder der Performanz- und Zeitmessung dient der Aufdeckung von sogenannten Hot-Spots oder auch Schwachstellen innerhalb einer Software.

Dadurch können aber auch Rückschlüsse über das gesamte Verhalten einer Software gezogen werden. Es wird somit erkennbar, welche Funktion wie oft von einer anderen Funktion aufgerufen wird. Was kann man nun aus solchen Erkenntnissen folgern? Die Antwort hierauf ist, dass die gewonnenen Informationen aufzeigen können, an welcher Stelle ein Programm mit möglichst wenig Aufwand optimiert werden kann. Dabei ist es wichtig die Kombination aus Ausführungszeit und Programmverhalten zu betrachten.

Für die Softwareparallelisierung sind diese Informationen unerlässlich um einerseits eine möglichst performante und damit schnelle Softwareausführung zu erhalten, aber auch um die Auslastung der zur Verfügung stehenden Prozessoren zu maximieren.

Für das Profiling werden oftmals zwei unterschiedliche Verfahren angewandt. Je nach Verfügbarkeit von Systemressourcen und Zugriffsmöglichkeiten besitzen sie unterschiedliche Vor- und Nachteile.

Einer der ersten Profiler ist gprof [GKM82, FS88] welcher sowohl auf Sampling aufsetzt, allerdings auch eine betriebssystemabhängige Instrumentierung erlaubt. gprof ermittelt ein Ausführungsprofil, welches angibt, wie viel Zeit die einzelnen Funktionen eines Programms benötigen. Dabei wird zwischen exklusiver Zeit, also Zeit, in der Instruktionen dieser Funktion ausgeführt werden, und inklusiver Zeit, in der Instruktionen von aufgerufenen Funktionen ausgeführt werden unterschieden. Viele neuere Profiler [Sle11, Ver11, NS03, NS07] bauen auf den gleichen Prinzipien auf, erlauben aber mehr Einstellungsmöglichkeiten oder die Integration von Trace-Points, d.h. betriebssysteminterne Schnittstellen, neuere Betriebssysteme, graphische Darstellungen der Ergebnisse oder eine besonders leichtgewichtige Implementierung und Integration in die Anwendung.

7.1.2.1 Sampling

Das Sampling beschreibt eine einfache und schnelle Möglichkeit ein Profiling durchzuführen. Um ein Profiling zu erstellen, nimmt der Profiler in einem regelmäßigen Intervall Stichproben vom Programmablauf, indem er beispielsweise den Callstack eines Prozessors oder einen Speicherbereich ausliest. Ein Callstack Sample stellt eine Liste von Funktionen in folgender Reihenfolge dar. An oberster Stelle steht die Funktion, die zum Zeitpunkt der Stichprobe in Bearbeitung war. Die darunter aufgelisteten Funktionen geben den Aufrufverlauf an. Durch dieses Verfahren ist es dem Profiler möglich ein Profiling zu erstellen ohne den Programmcode verändern zu müssen. Der Programmablauf wird im Regelfall also nicht beeinflusst. Allerdings hat die Intervallgröße einen massiven Einfluss auf die Genauigkeit und die Leistung des Gesamtsystems. Ein zu kleines Intervall kann dazu führen, dass ein Prozessor zu viel Zeit damit verbringt Samples zu generieren, so dass keine Ressourcen für die eigentliche Programmausführung zur Verfügung stehen. Ein zu groß gewähltes Intervall birgt dagegen die Gefahr, dass kleinere Funktionen nicht von einem Sample getroffen werden. Die Folgen sind fehlende Funktionen oder eine falsche Verteilung von Ausführungszeiten.

Beim Sampling wird die Ausführungszeit nicht direkt gemessen, sondern es werden ausschließlich Stichproben zu bestimmten Zeitpunkten genommen. Die Ausführungszeit einzelner Funktionen muss daher mit statistischen Methoden ermittelt werden, welche die tatsächliche Ausführungszeit möglichst genau approximieren sollen.

Fehlende Funktionsaufrufe oder Codeblöcke würden innerhalb einer automatisierten Parallelisierung zu großen Ungenauigkeiten und damit schlechten Parallelisierungsergebnissen führen.

7.1.2.2 Instrumentierung

Der Begriff Instrumentieren kommt von der Anbindung von Instrumenten zur Messung der Laufzeiten oder Ausführungshäufigkeiten. Die Instrumente sind in diesem Fall zusätzliche Monitorfunktionen, die einen Funktionsaufruf oder einen Rücksprung an den Profiler weitergeben. Eine andere Art von Instrumenten sind Zähler, die im Programmcode untergebracht werden um die Ausführungshäufigkeit von Funktionen aufzuzeichnen. Das Ergebnis ist daher sehr detailliert und gibt Informationen darüber, welche Funktionen wie oft aufgerufen wurden. Darüber hinaus gibt das Ergebnis die aufrufenden Funktionen und die aufgerufenen Funktionen aller instrumentierten Funktionen an. Durch die Verwendung von Zeitmarken lassen sich auch sehr genaue Zeitmessungen erstellen, um herauszufinden, wie viel Zeit eine Funktion oder ein Codeblock tatsächlich zur Ausführung benötigt. Mit Instrumentierungsfunktionen ist es darüber hinaus auch möglich exakt zu protokollieren, wie viel Zeit auf Ein- und Ausgabe, Sperrmechanismen oder andere Konstrukte und Systemkomponenten gewartet werden muss. Man unterscheidet im Allgemeinen zwischen einer manuellen und einer automatischen Instrumentierung. Bei der manuellen Instrumentierung fügt der Programmierer die zusätzlichen Funktionen selbst in den Code ein. In diesem Fall ist der Programmierer der Profiler selbst. Im Normalfall übernimmt er dann auch die Auswertung der Ergebnisse. Von automatischer Instrumentierung spricht man, wenn ein Profiler das Programm ohne den Programmierer ändert. Das heißt, der Profiler fügt die zusätzlichen Funktionen und Zähler selbstständig in den Programmcode ein und wertet die Ergebnisse nach der Programmausführung aus. Wie zu Beginn beschrieben bedeutet Instrumentieren auf jeden Fall einen Eingriff in den Programmcode. Dies ist der große Nachteil von instrumentiertem Code: Er entspricht nicht mehr der ursprünglichen Software, die ausgeführt werden sollte.

7.1.3 Tracing

Ganz andere Ziele verfolgt das Tracing. Als ein Test, der bewusst auf das interne Verhalten abzielt, ermöglicht das Tracing eine genaue Verfolgung des Programmablaufs oder internen Routinen und Hardwarefunktionen. Ein Tracing kann grundsätzlich auf zwei Ebenen durchgeführt werden. Die erste Ebene ist die Daten- oder Instruktionsebene. Dabei werden ausgeführte Befehle oder Datenzugriffe zum Zeitpunkt und in der Reihenfolge ihrer Ausführung gespeichert oder ausgegeben. Dies kann zum Erkennen von Fehlern verwendet werden, wenn die richtige Ausführung des zu testenden Programms bekannt ist.

Die Protokollierung aller Befehle ist allerdings sehr speicherlastig und rechenintensiv. In vielen Fällen reicht es aus ein Tracing nur auf Kontrollflussebene durchzuführen. Das Programm wird dazu in einen Kontrollfluss überführt, der aus einzelnen Basisblöcken aufgebaut ist. Dadurch ist es nicht mehr nötig jeden einzelnen Befehl zu protokollieren.

Bei komplexen Programmen sind oftmals mehrere Tausend Basisblöcke nötig um den gesamten Programmablauf darzustellen. Daher wird ein Tracing oftmals nicht von Hand, wie das Debugging, sondern von Tools ausgeführt, die eine automatisierte Programmablaufanalyse erstellen.

7.1.4 Performanzanalyse

Für die Ermittlung der Performance werden in [CHB09] strukturelle Performanzmodelle verwendet. Diese werden innerhalb des Kompiliervorgangs von GCC [C. 10] durch Verwendung einer Maschinenbeschreibung erzeugt. Im Normalfall werden Maschinenbeschreibungen innerhalb von GCC verwendet um architekturspezifischen Code zu generieren und spezifische Codeoptimierungen durchzuführen. Diese Maschinenbeschreibung wird ausgetauscht und dazu verwendet SystemC Simulationsmodelle zu erzeugen, mit denen eine

Performanzermittlung durchgeführt werden kann. Das erzielte Ergebnis ist zwar sehr genau, bietet allerdings nur eine statische Abschätzung der Ausführungszeit. Somit können dynamische Effekte bei der automatisierten Parallelisierung nur schwer abgeschätzt werden.

In [CMH+13] wird eine kombinierte Technik zwischen Sampling und Profiling vorgeschlagen, welche einen interessanten Ansatz darstellt. Um den Overhead der Instrumentierung möglichst gering zu halten, werden keine statischen Instrumentierungsfunktionen in den Anwendungscode eingebracht, sondern eine sogenannte Dynamische Binärinstrumentierung verwendet, die es ermöglicht, den ausgeführten Code zur Laufzeit zu instrumentieren. Im Allgemeinen übernimmt eine Laufzeitumgebung die Kontrolle über die Ausführung der Anwendung, um so die Instrumentierung dynamisch einfügen zu können, wodurch der Overhead gering bleibt. Um diesen Overhead noch weiter zu verringern, wird ein Start und Stopp Signal eingefügt, welches in regelmäßigen Intervallen, analog zu einem Sampling, die Kontrolle an die Anwendung zurückgibt oder sich die Kontrolle nimmt. So werden Sampling und Instrumentierung kombiniert, was den Overhead verringert und dennoch nur zu moderaten Genauigkeitsverlusten führt. Hierbei können weitere Kompromisse eingegangen werden um die Genauigkeit zu erhöhen oder den Overhead zu vermindern. Die wichtigsten Parameter hierfür sind das Samplingintervall und die Dauer der laufenden Instrumentierung pro Samplingintervall.

Moseley et. al. [MSR+07] verfolgen einen Schattenprofiling genannten Ansatz, um den Einfluss der Instrumentierung auf die auszuführende Applikation zu minimieren. Dazu wird ein Schattenprozess erzeugt, welcher als Klon zu der zu instrumentierenden Applikation erzeugt wird und an Stelle dieser instrumentiert wird. Dieser Ansatz kann auf Multicoreprozessoren angewandt werden, wenn genügend Rechenressourcen zur Verfügung stehen. Um den Overhead für das Gesamtsystem zu minimieren ist es möglich Schattenprozesse dynamisch zu erzeugen, welche im System auf unterschiedliche Prozessoren geschedult werden können.

Ein ähnlicher Ansatz wird in [WH07] verwendet. Analog zum Schattenprofiling werden hier Schattenprozesse einer nicht-instrumentierten Applikation erzeugt und mit der Dynamischen Binärinstrumentierung, vgl. [CMH+13], instrumentiert. Die Kombination des Schattenprofiling mit der dynamischen Instrumentierung erlaubt es nicht nur die Applikation, sondern auch daran angebundene Bibliotheken zu instrumentieren und dabei den Einfluss auf die Applikation gering zu halten.

Beide Ansätze erfordern allerdings einen dynamischen Scheduler innerhalb des Systems, welcher nicht in allen eingebetteten Systemen verfügbar ist.

7.2 System Profiler

Die oben beschriebenen Möglichkeiten sind grundsätzlich auch in einem Simulationssystem einsetzbar. Auch die Anbindung externer Profiler, wie gprof [GKM82] oder andere, ist möglich. Allerdings geht der Einsatz eines instrumentierenden Profilers immer zu Lasten der Ausführungszeit einer parallelen Anwendung. Viele externe Profiler setzen deshalb auch auf das Sampling von Zustandsdaten. Dieses muss in einer Simulationsumgebung allerdings explizit unterstützt werden, oder es muss ein Betriebssystem mit der Anwendung simuliert werden worüber diese Daten erhoben werden können. Da für die automatisierte Parallelisierung in ALMA außerdem stellenweise sehr feingranulare Messungen vorgenommen werden müssen, ist der Einsatz von Instrumentierungsfunktionen aus Qualitätsgründen vorzuziehen. Die Verfälschung der Simulationsergebnisse kann dadurch reduziert werden, dass Instrumentierungsfunktionen vom Simulator abgefangen werden können und direkt als Simulatorcode ausgeführt werden.

Die Basis des System Profilers stellen deshalb Instrumentierungsfunktionen dar, welche in den Applikationscode eingefügt und im Simulator registriert werden müssen. Während der Simulation der Applikation achtet der Profiler

auf die Ausführung dieser Instrumentierungsfunktionen. Dabei kann jedes Simulationsmodul auf die in den Basisklassen implementierten Funktionen zurückgreifen um die Instrumentierungsfunktionen abzufangen. Grundsätzlich werden Instrumentierungsfunktionen als Paar eingefügt. Eine *start*-Funktion leitet eine Messung ein, indem ein Zeitstempel während der Simulation erzeugt wird. Eine darauf folgende *end*-Funktion erzeugt wiederum einen Zeitstempel, welcher mit dem Start-Zeitstempel verrechnet wird ([BBO+14]).

Jeder Instrumentierungsfunktion muss im Applikationscode eine eindeutige ID zugewiesen werden, womit *start*- und *end*-Funktion identifiziert werden können. Darüber hinaus fügt das Framework entsprechende Event und Type Bezeichner hinzu, welche eine vollständige systemweite Eindeutigkeit garantieren. Zusätzlich können weitere Parameter innerhalb von Modulen hinzugefügt werden, welche anschließend zur Verarbeitung der gesammelten Informationen genutzt werden können.

Das Funktionsformat einer start-Instrumentierungsfunktion ist in Quellcode 7.1 dargestellt, wobei `<TYPE>` durch einen der verfügbaren Profile-Types ersetzt werden muss (vgl. Abschnitt 7.4):

```
1 start_<TYPE>_profiling(uint32 task_id, uint32
      processingElement_id);
```

Quellcode 7.1: System Profiler Funktionsformat zur Instrumentierung

Es stehen drei allgemeine Profiling-Types zur Verfügung, welche insgesamt sechs Instrumentierungsfunktionen bereitstellen. Diese sind in Tabelle 7.1 aufgelistet.

Instrumentierungsfunktionen für unterschiedliche Profiling Types können überlappend sowie umschließend in den Code integriert werden, da sie sich nicht beeinflussen. Für Instrumentierungsfunktionen welche demselben ProfilingType zugeordnet werden sind nur umschließende Funktionsaufrufe möglich. Eine Überlappung ist nicht erlaubt, da sonst Zeitstempel nicht mehr eindeutig zugeordnet werden können und anschließende Berechnungen fehler-

Funktion	Beschreibung
`start_task_profiling`	Startmarkierung für eine TASK Profiling Messung, welche ein durch Start und Ende definiertes Quellcodesegment vermisst.
`end_task_profiling`	Endmarkierung für eine TASK Profiling Messung.
`start_send_profiling`	Startmarkierung für eine SEND Profiling Messung um Kommunikationsfunktionen wie beispielsweise MPI_Ssend zu profilen.
`end_send_profiling`	Endmarkierung für eine SEND Profiling Messung.
`start_recv_profiling`	Startmarkierung für eine RECIEVE Profiling Messung um Kommunikationsfunktionen wie beispielsweise MPI_recv zu profilen.
`end_recv_profiling`	Endmarkierung für eine RECIEVE Profiling Messung.

Tabelle 7.1: Verfügbare allgemeine Profilingfunktionen

haft sind. Quellcode 7.2 gibt dabei ein Beispiel für eine fehlerhafte Integration von Instrumentierungsfunktionen an.

```
1 //fehlerhafte Überlappung von Instrumentierungsfunktionen
2 //basierend auf demselben Profiling Type und derselben
3 //Ausführungseinheit
4 start_task_profiling(1, 1);
5 start_task_profiling(2, 1);
6 end_task_profiling(1, 1);
7 end_task_profiling(2, 1);
```

Quellcode 7.2: Fehlerhafte Überlappung von Instrumentierungsfunktionen

7.2.1 Aufbau des SystemProfiler

Der Aufbau des Profilers ist in Abbildung 7.1 abgebildet, anhand derer auch die Funktionsweise und Zusammenhänge erläutert werden. In Grün dargestellt ist die Basisklasse des Profilers, welche als Objekt im Systemsimulator

erzeugt und initialisiert wird. Die vier Funktionen stellen die API des Profilers dar, um Profiling Events und Profiling Types zu registrieren und miteinander zu verknüpfen. Die Funktion `profileEvents(...)` dienst dabei als Einsprung in die Rot dargestellten Event-Objekte, welche wiederum die einzelnen Profiling Types aufrufen und die gesammelten Daten weitergeben. In Blau sind dargestellt die Basisklasse für Profiling Types und die für das ALMA Parallelisierungswerkzeug implementierten, davon abgeleiteten Types, welche aktiv im Framework eingesetzt werden.

7.3 Profiling Events

Ein Profiling Event ist ein bei der Initialisierung des System Profiler generiertes Objekt, welches auf die Instrumentierungsfunktionen hört. Jede Instrumentierungsfunktion besitzt eine eindeutige ID, welche auf einem Profiling Event registriert werden kann. Anschließend wird das Event ausgeführt sobald eine registrierte Instrumentierungsfunktion ausgeführt wird. Um zu verhindern, dass es zu Interferenzen zwischen mehreren Events kommt, ist eine Registrierung einer Instrumentierungs ID nur auf genau einem Event möglich. Um dennoch eine größtmögliche Komplexität bereitstellen zu können ist es möglich einen Event mit mehreren Profiling Types zu kombinieren. Dadurch können die Informationen, wie der aktuelle Taktzyklus oder die Task ID, welche durch das Event gesammelt werden mehreren Auswertealgorithmen zur Verfügung gestellt werden.

Das System Simulator Framework stellt dafür Funktionen zur Registrierung von Profile Events im System Profiler, als auch der IDs zu den Events zur Verfügung. Darüber hinaus werden die Funktionen bereitgestellt, welche es erlauben die Profile Events mit den Profile Types zu verknüpfen.

Die RegisterEvent Funktion erzeugt dabei ein Profiling Event und weißt ihm die übergebene Instrumentierungs ID zu. Die Funktion gibt einen Fehlerwert

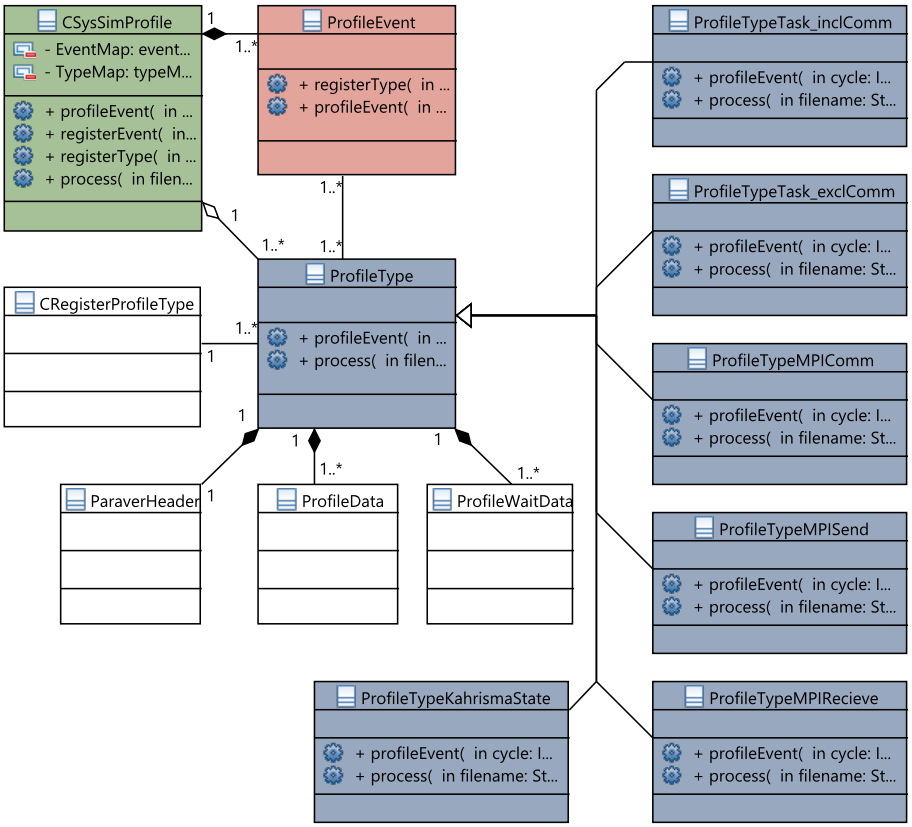


Abbildung 7.1: Klassendiagramm des SystemProfilers

zurück, der angibt ob die ID registriert werden konnte (`true`), oder ob schon ein Event mit dieser ID existiert (`false`).

```
1 bool profile_register_pEvent(uint8_t eID, CTSString name);
```

Quellcode 7.3: Profiling Event Registrierung

Die Type Mapping Funktion verknüpft ein existierendes Profiling Type Objekt mit einem Profiling Event. Das Event speichert alle mit ihm verknüpften Types ab und ruft eine Execute-Funktion innerhalb der Types auf, jedes Mal, wenn das Event aufgerufen wird. Auch diese Funktion gibt Fehlerzuständen zurück, je nachdem ob ein Type schon auf diesem Event registriert wurde (`false`), wenn der Type nicht existiert (`false`), wenn des Event nicht existiert (`false`) oder wenn eine erfolgreiche Verknüpfung durchgeführt wurde (`true`).

```
1 bool profile_map_pEvent_pType(uint8_t eID, CTSString
      TypeName);
```

Quellcode 7.4: Verknüpfung von Profiling Type und Event

7.4 Profiling Types

Ein Profiling Type stellt eine Analyse und Auswertefunktion dar, welche auf den Daten der auftretenden Profiling Events ausgeführt wird. Dabei kann ein Profiling Type auf einen oder auch mehrere Events angewiesen sein. Um diese Daten zu erhalten reicht es aus, den Profiling Type über die in Quellcode 7.4 dargestellte Funktion mit dem Profiling Event zu verknüpfen. Wie oben beschrieben, wird nun bei jedem auftretenden Event eine Execute Funktion aufgerufen, welche vom Profiling Type Entwickler individuell implementiert werden kann. Je nachdem, welche Events von einem Simulationsmodul zur Verfügung gestellt werden, können damit modulspezifische Profilingfunktionen implementiert werden. Auf diese Weise lassen sich aber auch alle weiteren auftretenden Profiling Events innerhalb einer Simulationsumgebung miteinander kombinieren. Die Registrierfunktionen der Event-Klasse prüfen dabei ob ein Profiling Type während einer Simulation verwendet werden kann oder nicht.

Im nächsten Abschnitt sind ProfilingTypes als Beispiele beschrieben, wie sie im ALMA Projekt verwendet wurden.

7.5 Ausgabe

Für ein allgemeines Profiling erlaubt der SystemProfiler die Verwendung von drei vordefinierten Profiling Types die jeweils eine spezifische Profiling Ausgabe erzeugen. Diese drei Types definieren ein TASK Profiling welches die Ausführungszeiten und Ausführungsinformationen von Quellcodeabschnitten sammelt. Des Weiteren wird ein SEND und ein RECIEVE Profiling implementiert um die Kommunikationszeiten von zu sendenden Daten und zu empfangenen Daten zu ermitteln. Darüber hinaus steht ein Profiling Type für das Statustracing eines Prozessors zur Verfügung.

7.5.1 Task ID Profiling

Die Profiling Informationen des TASK Profiling zielen auf die Ermittlung von Ausführungszeiten und Ausführungsinformationen von sogenannten TASKS. Als Task wird ein Quellcodeabschnitt bezeichnet, welcher sich aus einem oder mehreren Basisblöcken zusammensetzen kann. Die Profiling Informationen werden in Java Script Object Notation (JSON) abgelegt womit eine Weiterverarbeitung der Informationen in Werkzeugen zur Codegenerierung als auch zur Softwareparallelisierung wie der ALMA Toolchain ermöglicht. Welche Informationen durch das TASK Profiling gesammelt werden ist in Tabelle 7.2 abgebildet.

Der Profiler besitzt eine grundlegende Fehlerbehandlung. Das bedeutet er stellt fest, wenn durch falsche Instrumentierung Unregelmäßigkeiten innerhalb der ProfilingIDs auftreten. Dadurch werden ungültige Profilinginformationen erzeugt, die der Profiler ignoriert.

Die Profiling Informationen werden pro Instrumentation ID und pro Ausführungseinheit, beispielsweise ein Prozessor in einem Multicoresystem, erzeugt.

Wert	Beschreibung
minimum	Die minimale Ausführungszeit einer Ausführung des instrumentierten Codesegments in Cycles.
maximum	Die maximale Ausführungszeit einer Ausführung des instrumentierten Codesegments in Cycles.
average	Die mittlere Ausführungszeit einer Ausführung des instrumentierten Codesegments in Cycles.
waiting	Die durchschnittliche Zeit einer Ausführung des instrumentierten Codesegments in Cycles, in der der simulierte Prozessorkern keine Instruktionen ausführt. Beispielsweise, wenn auf Kommunikation gewartet wird. Dies ist nur ermittelbar, wenn der Simulator die Wartezeiten ermitteln kann.
execute	Die Gesamtanzahl der Ausführungen des instrumentierten Codesegments in Cycles.
ignored	Die Anzahl der Ausführungen, welche durch den Profiler ignoriert werden. Dies kann durch interne Fehler oder falsche Instrumentierung auftreten. Bei ignorierten Ausführungen werden die gemessenen Zyklen nicht zur Berechnung von Minimum, Maximum, Average oder Waiting herangezogen.

Tabelle 7.2: TASK Profiling Ausgabeinformationen

Das Ergebnis wird sortiert und in einer Baumstruktur im JSON Format abgebildet. Die Sortierreihenfolge ist wie folgt. Ein Beispiel dafür ist in Quellcode 7.5 angegeben.

1. Profiling Type Name
2. Ausführungseinheiten ID
3. Instrumentation ID (Task ID)

Zur Visualisierung der Profiling Ergebnisse wurde der Profiler mit einer Funktion ausgestattet, die Profiling Informationen in Form einer Paraver Trace [PLCG95, Bar14] zu schreiben. Das Paraver Trace Format stammt aus dem High-Performance Computing Bereich und wurde für das Profiling und Tracing von verteilten Anwendungen basierend auf MPI auf Serversystemen und

```
{
  "ALMA_TASK":  {          //ProfilingType Name
    "0":  {                //ID der Ausführungseinheit
      "1":  {              //ID der
          Instrumentierungsfunktion
        "minimum":  17,
        "maximum":  17,
        "average":  17,
        "waiting":  0,
        "execute":  100,
        "ignored":  0
      }
    },
  },
  "ALMA_SEND":  {
    "1":  {
      "2":  {
        "minimum":  6,
        "maximum":  6,
        "average":  6,
        "waiting":  0,
        "execute":  1,
        "ignored":  0
      }
    }
  },
}
```

Quellcode 7.5: Beispiel eines ALMA Task Profiling Eintrags

Rechenzentren entwickelt. Es visualisiert ein System sowie Applikationszustände in zeitlicher Abfolge. Dazu werden sogenannte State Traces verwendet, die einer Zeitspanne einen Status zuweisen. Dieser Status wird pro Thread ermittelt. Dabei kann ein System aus mehreren Nodes mit jeweils mehreren Prozessoren bestehen. Dazu können Anwendungen auf dem Serversystem definiert werden. Die Definition einer Anwendung beinhaltet die Aufteilung der Anwendung in Tasks, welche einer Node zugewiesen werden. Eine Task besteht wiederum aus mehreren Threads die dann auf den Prozessoren ausge-

führt werden. Darüber hinaus können zusätzliche Kommunikationsevents eingefügt werden, welche beispielsweise auf MPI Send und MPI Recieve Funktionen gemappt werden.

Paraver ist ein Werkzeug in einer Reihe von Analysewerkzeugen die auch für weitere Analysen der Applikation verwendet werden können.

Der Systemsimulator verwendet nun die Eigenschaften des Paraver Tracing Werkzeugs um die ermittelten Traces und Profiling Informationen zu visualisieren. Dazu ermittelt der Simulator die Anzahl der Verwendeten Prozessorsimulatoren bzw. Prozessoren und die Gesamtausführungszeit der Applikation in Zyklen. Diese wird definiert durch das Maximum an simulierten Zyklen der Prozessoren innerhalb der Architekturbeschreibung.

Ein Paraver State Trace Eintrag wird mit dem Format aus Quellcocde 7.6 erzeugt und beinhaltet die Laufzeit die eine Codesegment in einem definierten Zustand verbringt.

```
1 1:<core>:<app>:<task>:<thread>:<starting time>:<ending
      time>:<state value>
```

Quellcode 7.6: Paraver State Record Eintrag

Paraver State Trace Einträge werden unterschieden durch die Ausführungseinheit, die Applikation, die Tasks der Applikation sowie der Threads innerhalb eines Tasks. Diese Eigenschaften werden nun spezifisch für den Multicore Simulator genutzt um die Profiling Informationen abzubilden. Für jedes Paar an passenden Instrumentierungsfunktionen, jeweils start- und end-Funktion, wird für das allgemeine Profiling ein Eintrag mit dem Status 1 erzeugt um anzuzeigen, dass das Codesegment in diesem Zeitintervall ausgeführt wurde.

Innerhalb des Simulators werden die Ebenen core, app, task und thread dazu verwendet hierarchische Codesegmente zu visualisieren, die sich gegenseitig

beinhalten. Dies ist durch die Verwendung innerhalb der ALMA Toolchain entstanden, welche für die Parallelisierung einen HTG verwendet, um die Applikation zu strukturieren. Dieser hierarchisch aufgebaute Baum beinhaltet hierarchische Knoten und Blattknoten. Ein Blattknoten ist im Regelfall ein Basisblock während ein hierarchischer Knoten mehrere Basisblöcke als auch Kommunikation beinhalten kann. Ein Knoten wird dabei als Task bezeichnet, wenn es sich um einen Blattknoten handelt oder als HTask für einen hierarchischen Knoten.

Im Folgenden sind die Ebenen und deren Verwendung im System Profiler beschrieben:

Processing Cores werden für die Visualisierung des Prozessors bzw. Simulationsmodul verwendet auf dem die Instrumentierungsfunktion ausgeführt wurde.

Application wird für Level 1 Profiling IDs verwendet. Dies ist entweder eine TaskID, welche in keinem HTask integriert ist, oder um den obersten HTask innerhalb der Struktur.

Task wird für Level 2 Profiling IDs verwendet. Dies ist entweder eine TaskID, welche in genau einem HTask integriert ist, oder um den zweithöchsten HTask innerhalb der Baumstruktur.

Thread wird für Level $2 + i$ Profiling IDs verwendet. Dies ist immer der letzte Task, also Blattknoten, einer hierarchischen Struktur mit zwei oder mehr hierarchischen Knoten. HTasks welche an dritter oder weiteren Stelle in der Hierarchie stehen werden nicht dargestellt, da Paraver nur die Verwendung von drei Ebenen zulässt.

Für jeden Profiling Type wird eine eigene Tracing Datei erzeugt. Diese wird unter `<app>.profiling.<taskName>.prv` abgespeichert.

7.5.2 Kahrisma Spezifisches Prozessor Tracing

Für das Kahrisma Simulationsmodul wurde ein spezielles Statustracing implementiert, welches während einer Simulation durchgeführt werden kann. Dazu wurden die Instrumentierungsfunktionen direkt in das Simulationsmodul integriert, so dass keine Änderung der Anwendung notwendig ist. Das Prozessortracing ermittelt dabei den Wechsel der Zustände des Kahrisma-Prozessors. Die Zustände sind in Tabelle 7.3 aufgeführt und beschrieben. Der Prozessorstatus wird dabei von der Art der ausgeführten Instruktionen bestimmt, und kann mit Hilfe von Kommunikationsvorgängen bestimmt werden.

Die Informationen werden analog zum allgemeinen Profiling in Paraver State Trace Einträge geschrieben mit dem Unterschied, dass je nach Zustand auch ein anderer Wert für den Eintrag gesetzt wird.

Statuscode	**Name**	**Beschreibung**
0	unused	Der Prozessorsimulator wurde nicht konfiguriert und deswegen nicht verwendet.
1	idle	Der Prozessorsimulator wurde konfiguriert, führt aber derzeit keine Instruktionen aus.
2	running	Der Prozessorsimulator wurde gestartet und führt Recheninstruktionen aus.
3	waiting send	Der Prozessorsimulator wartet auf einen anderen Prozessor bevor er eine sendende Kommunikation startet.
4	sending	Der Prozessorsimulator führt Instruktionen zum Senden von Daten aus.
5	waiting recv	Der Prozessorsimulator wartet aber auf einen anderen Prozessor um Daten zu empfangen.
6	recieving	Der Prozessorsimulator empfängt Daten von einem anderen Kern.

Tabelle 7.3: Mögliche Kahrisma Prozessor Statuscodes mit Beschreibung

Für das Tracing sind nicht alle Ebenen des Paraver Eintrags notwendig. Deshalb schreibt der SystemProfiler die Kahrisma ProzessorID als core und als

thread in den Paraver State Eintrag. Durch das Einfügen der ID als Thread werden Aggregationsfehler verhindert, welche bei der Verwendung des Paraver Werkzeugs auftreten können, wenn der Eintrag nicht geschrieben wurde. Die Datei wird mit dem Namen `<app>.paraverCoreTrace.prv` abgelegt.

Kapitel 8

Framework Evaluation

Dieses Kapitel beschreibt die mit der ADL und dem Simulationsframework durchgeführten Experimente und stellt deren Ergebnisse vor. Es wurden Untersuchungen hinsichtlich der Anwendbarkeit der Hierarchischen Beschreibung zur Beschleunigung der Simulation, der Eignung zur Entwurfsraumexploration sowie zur Verwendung innerhalb iterativer Parallelisierungswerkzeugen durchgeführt.

Einer der Hauptabwägungen eines Simulators ist die Bilanz zwischen der Performanz oder Geschwindigkeit einer Simulation und der Genauigkeit der Simulation und ihrer Ergebnisse. Die hierarchische Struktur der ADL erlaubt es diese Abwägung zu treffen, dynamisch zu verändern sowie die Vorteile unterschiedlicher Auslegungen von Performanz und Geschwindigkeit effektiv nutzbar zu machen. Während in anderen Beispielen oftmals die Ebenen von TLM zur Differenzierung dieser Abwägung verwendet werden [WDK+04] erlaubt die ADL die Modellierung genauigkeits-sensitiver Parameter, welche sowohl eine grobgranulare Simulation als auch eine feingranulare Modellierung erlauben. Durch die Verwendung mehrerer Hierarchieebenen eines einzelnen Moduls können Leistungsprobleme spezifischer Moduleigenschaften oder von Teilkomponenten eines Moduls ermittelt werden während andere Bereiche eines Systems auf funktionaler Ebene ohne nennenswerte Geschwindigkeitsnachteile simuliert werden können. Dies kann durch die ADL soweit gesteuert werden, dass auch unterschiedliche Instanzen desgleichen Moduls unterschiedlich simuliert werden können.

In den folgenden Abschnitten werden einige der Möglichkeiten der ADL evaluiert und analysiert um das Geschwindigkeits- und Genauigkeitsverhalten der Simulation zu bewerten.

Dazu zunächst einige Definitionen, welche im Verlauf dieses Kapitels verwendet werden:

Instruction Eine innerhalb eines simulierten Prozessors vollständig ausgeführte Operation der simulierten ISA wird *Instruction* genannt. Die Anzahl der benötigten Instruktionen für eine Simulation ist unabhängig vom Host-Computer auf dem die Simulation ausgeführt wird.

Clock Cycle Ein einzelner Ausführungsschritt der Simulation, welcher an die Taktfrequenz des simulierten Prozessors gebunden ist, ist ein *Clock Cycle* und führt zu einem Fortschreiten der Simulationszeit. Eine *Instruction* kann auch mehrere *Clock Cylces* benötigen um ausgeführt zu

werden. Analog zur Anzahl der Instruktionen hängt auch die Anzahl der Taktzyklen nicht vom Host-Computer ab.

Simulation Time Die Simulationszeit ist die reale Zeit, die der Simulator benötigt um die Simulation auszuführen. Diese Zeit hängt stark von der Leistung des Host-Computer ab.

Simulated Time Im Vergleich zur Simulationszeit, ist die simulierte Zeit die Zeit, welche die simulierte Anwendung bei realer Ausführung benötigt hätte. Sie wird über die Synchronisation der Taktfrequenzen und den ausgeführten Taktzyklen ermittelt.

Wall Clock Die sogenannte *Wall Clock* ist eine globale systemweite Zeit, gegenüber der alle Module synchronisiert werden.

Simulation Speed Die Simulationsgeschwindigkeit ist eine abgeleitete Größe die das Verhältnis:

$$Speed = \frac{\#Instructions}{SimulationTime}$$

beschreibt. Da die Simulationszeit in die Berechnung mit einfließt ist auch dieser Wert stark vom Host-Computer abhängig.

Run-ahead Cycle Die Anzahl an Taktzyklen, die eine Simulationsinstanz gegenüber anderen Instanzen vorweg laufen kann ohne synchronisiert werden zu müssen, wird als *Run-ahead Cycles* definiert.

Simulationsgenauigkeit Die Genauigkeit der Simulation wird immer als relativer Wert verglichen mit einem Referenzsystem angegeben. Das Referenzsystem beschreibt dabei ein System, bei dem die maximale Anzahl an Simulationsmodulen mit jeweils maximaler interner Genauigkeit, vgl. zeitlose Simulation gegenüber RTL-Simulation sowie ein minimales Synchronisationsintervall verwendet werden.

8.1 Zielsetzung der Evaluation

Der Fokus des Evaluationsvorgangs ist die Beschreibung sowie die Analyse des Einflusses verschiedenster Parameter und deren Änderungen innerhalb der ADL auf die erzeugte Simulationsumgebung und die Simulation selbst. Einen hohen Stellenwert nehmen dabei die Effekte der hierarchischen Beschreibung im Simulator ein, sowie der Einfluss des *run-ahead* einzelner Prozessorinstanzen. Andere Systemparameter werden verwendet um die Beschreibung samt Simulation im Hinblick auf den Einsatz in einer Entwurfsraumexploration hin zu untersuchen.

Ein weiteres Ziel dieser Evaluation ist die Feststellung der Eignung des Simulationssystems für iterative Parallelisierungswerkzeuge und der Einfluss der Simulationsgenauigkeit auf die erzielbare Qualität einer automatisierten Parallelisierung. Durch den Zusammenhang der Parameter, des Hierarchiekonzepts der ADL und des Simulators mit der Simulationsgenauigkeit kann hier ein indirekter Zusammenhang zur Ergebnisqualität hergestellt werden.

Für die Evaluation stehen die folgenden Module mit ihren Implementierungen zur Verfügung:

Modul	Implementierung
EDPE	EDPE_Core
	L1 cache
	Local Memory
Memory	L2 cache
	Global memory
Network	NetworkAdapter
	iNoC

Tabelle 8.1: Verwendete Hierarchieebenen und Module innerhalb der Evaluation

Die Architekturbeschreibungssprache erlaubt es durch hierarchische Modellierung mehrere Ebenen mit unterschiedlichen Detailgraden zu definieren. Für

diesen Zweck wurde eine Speicherhierarchie sowie eine hierarchische Netzwerkkomponente aufgebaut, welche in Tabelle 8.1 dargestellt wird. Ein **MemoryAdapter** wird dabei auf allen Ebenen implementiert, übernimmt allerdings jeweils andere Funktionen um die verfügbaren Speicherkomponenten korrekt ansprechen zu können.

Sowohl der ADL-Compiler als auch der Multicore Simulator lassen sich über Kommandozeilenparameter steuern, siehe auch Anhang C. Dadurch lassen sich Skripte implementieren die automatisiert über Simulationsparameter und Architekturparameter iterieren und die resultierenden Systeme evaluieren. Der Simulator erstellt mit Hilfe des System Profiler Laufzeitstatistiken, welche anschließend skriptbasiert ausgewertet werden können. Durch diese Entkopplung der Simulation von der endgültigen Auswertung können umfassende Daten gesammelt werden, welche nicht für jede Analyse benötigt werden. Im Nachhinein kann dadurch aber jeweils ausgewählt werden welche Daten betrachtet werden sollen ohne Simulationen wiederholen zu müssen.

8.2 Hierarchische Simulation und Entwurfsraumexploration

Für eine umfassende Entwurfsraumexploration werden nun die Modul- und Parameterkombinationen simuliert, analysiert und evaluiert, welche der Speicherhierarchie des Systems zuzuordnen sind. Um eine Systemweite Aussage über Simulationszeiten und Leistungsparameter zu erhalten werden die Statistiken der einzelnen Prozessoren gegenübergestellt und entweder gemittelt, das Maximum oder das Minimum bestimmt. Je nach Konfiguration, übernimmt ein dedizierter externer Prozessor das Laufzeitmanagement des KAHRISMA-Arrays oder ein Prozessor des Arrays übernimmt diese Aufgabe zu Beginn. Beide Konfigurationen sind auch in realen Multicore-Prozessoren denkbar. Für

eine reine Leistungsbetrachtung der Anwendungsausführung können im ersten Fall statistische Informationen und Evaluationsdaten des Management-Prozessors ignoriert werden, während im zweiten Fall eine Overhead Betrachtung durchgeführt werden muss.

8.2.1 Anwendung

Die Applikation mit der eine erste Evaluation durchgeführt wird, ist eine Matrix Multiplikation [Lös12], welche auf eine Vielzahl von Prozessoren verteilt werden kann. Für die Berechnung wird eine parametrierbare Anzahl verteilter Prozessoren verwendet, welche Daten und Nachrichten entsprechend dem Message Passing Interface (MPI)-Standard [Mes09, WD96] kommunizieren. Zunächst wird ein *Control-Task* erstellt, welcher die *Worker*-Instanzen erzeugt. Jede dieser Instanzen wird auf einen EDPE Prozessor verteilt, wodurch die gesamte Anwendung parallel ausgeführt wird.

Diese spezielle Version der Matrix Multiplikation unterstützt neben unterschiedlichen Eingangsgrößen alle KAHRISMA Zielarchitekturen sowie unterschiedliche Prozessorkonfigurationen. Für diese Evaluation wird zunächst nur die Zielarchitektur **RSIW2**, also ein zwei-fach VLIW Kahrisma Prozessor, verwendet, während die Schlüsselparameter der Applikation die Anzahl der verwendeten MPI Prozesse (n) und die Matrixgröße (M) als Eingangsgröße sind.

Matrizen wachsen in ihrer Größe, d.h. Anzahl an Elementen, exponentiell mit der Größe des Parameters m wie folgt:

$$M = 2^m \times 2^m, m \in \mathbb{N} \tag{8.1}$$

Die Anzahl der verwendbaren Prozesse, die für die Berechnung verwendet werden können verwendet daher auch die Basis Zwei. Je nach Konfiguration müssen daher:

$$i = \begin{cases} n, \text{ohne dedizierter Laufzeitmanagementinstanz} \\ n+1, \text{mit dedizierter Laufzeitmanagementinstanz} \end{cases} \tag{8.2}$$

Instanzen für die Ausführungseinheiten zur Verfügung stehen um die Applikation ausführen zu können.

8.2.2 Grundlegende Parameter

Während des Evaluationsprozess werden unterschiedlichste Simulationsparameter verändert, während andere unverändert bleiben. In Tabelle 8.2 werden alle dynamischen sowie alle statischen Parameter, deren Wertebereich sowie deren Standardwert aufgelistet. Sollten in einer Abbildung keine Änderungen eines Parameterwertes explizit aufgeführt sein, so kann angenommen werden, dass das System mit dem Standardwert des Parameters erzeugt und simuliert wurde. Die Standardwerte für die Cache- und Speicherparameter des KAHRISMA Systems sind aus der Arbeit von Stripf [SKB12] entnommen.

Die Anzahl der inaktiven EDPEs eines Multicore Systems haben während der Simulation keinen messbaren Einfluss auf die Ergebnisse. Dies kann gezeigt werden, indem Systeme mit deutlich unterschiedlich vielen Prozessorkernen, zwischen 2 und 100, verwendet werden um eine Anwendung zu simulieren, welche nur einen Prozess verwendet. Ein messbarer Unterschied liegt nur in der Instanziierung des Simulationssystems und in der Anlaufzeit des Simulators, da diese mit der Anzahl an verfügbaren simulierten Prozessoren steigt. Die Laufzeitergebnisse werden davon allerdings nicht beeinflusst, da die Messungen erst mit dem Start der eigentlichen Simulation nach dem Initiierungsprozess beginnen.

Um die allgemeine Laufzeit gering zu halten bietet sich ein System mit 16 bzw. 20 EDPEs an, da je nachdem welche MPI-Konfiguration gewählt ist entweder 16 bzw. 17 Prozessoren für die MPI-Prozesse zur Verfügung stehen müssen. Im

Module/Parameter	Standard	Wertebereich	Einheit
EDPEs	20	-	
System Frequenz	500	-	MHz
Speicherfrequenz	500	200, 500	MHz
MPI Prozesse	1	2, 4, 8, 16	
Matrix Größe	9	7, 8, 9, 14, 16, 18	$2^x \times 2^x$
Run-ahead Zyklen	1	10, 100, 1000, 10000	Cycles
Lokale Speicherverzögerung	18	-	Cycles
L1 hit Verzögerung	3	-	Cycles
L1 Größe	2	128, 256, 512, 1024	KB
L2 hit Verzögerung	6	-	Cycles
L2 Größe	256	64, 1024, 2048	KB
Globale Speicherverzögerung	18	-	Cycles

Tabelle 8.2: Evaluationsparameter mit Wertebereichen und Standardwerten [BOR+14]

Folgenden wurde für die Kachelstruktur des KAHRISMA-Evaluationssystem eine 4×5 Anordnung ausgewählt um 20 EDPEs zu realisieren. Die Anordnung wirkt sich auf die Kommunikationsinfrastruktur aus, weshalb eine möglichst quadratische Anordnung gewählt wurde. Für maximal 16 Prozesse ist daher eine 4×4 Anordnung gut geeignet.

8.2.3 Ausgangsbasis der Simulation

Die Ausgangssituation der Simulatorperformanz ist in Abbildung 8.1 dargestellt und zeigt die *Simulation Time* eines Standardsystems, vgl. Tabelle 8.2, wobei unterschiedlich große Prozessorarrays für die Berechnung der Matrix-Multiplikation verwendet wurden. Die Anzahl an MPI-Prozessen bewegt sich von minimal 1 Prozess bis hin zu 16 Prozessen als das Maximum.

Da diese Ergebnisse die Basis für weitere Analysen und Vergleiche darstellen, wurde das System auf dem niedrigsten verfügbaren Abstraktionslevel simu-

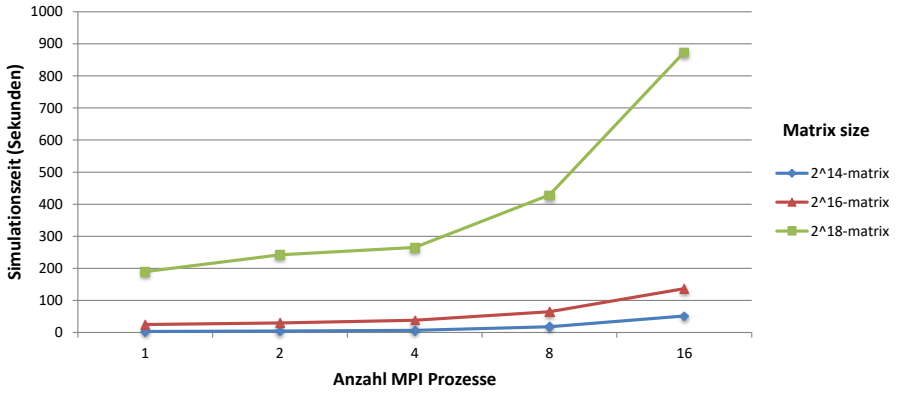


Abbildung 8.1: Simulation Time unterschiedlicher Matrixgrößen in Bezug auf die verwendeten MPI-Prozesse.

liert. Für den run-ahead-Parameter wurde der niedrigstmögliche Wert 1 verwendet, was den Simulator dazu zwingt, jeden Prozessor nach einem berechneten Zyklus zu synchronisieren. Um Parameteränderungen vergleichen zu können wird dieses System im Hinblick auf Genauigkeit und Geschwindigkeit als Referenzsystem verwendet.

Die Laufzeit der Anwendung korreliert sehr stark mit der Matrizengröße, da die Anzahl der Elemente nach Formel 8.1 exponentiell mit der Größe wächst. Somit steigt die Komplexität der Berechnung und damit erhöht sich auch die Ausführungszeit.

Ein anderes Ergebnis ist die längere Simulation Time bei einer Erhöhung der MPI-Prozesse, da der allgemeine Aufwand der Simulation aber auch der Switching Overhead innerhalb der Simulation mit der wachsenden Anzahl an aktiven Prozessorinstanzen wächst.

Bei kleineren Matrizen ist der prozentuale Anstieg des Aufwands größer als bei größeren Matrizen, da der Kommunikationsaufwand deutlich steigt. Für Matrizen einer Größe von $2^14 \times 2^14$ benötigt die Simulation mit 16 MPI-Prozessen

14.6 Mal länger als bei Verwendung von nur einem MPI-Prozess. Bei einer Größe von $2^16 \times 2^16$ dauert die Simulation nur 5.6 Mal länger und bei der größten getesteten Matrix mit $2^18 \times 2^18$ liegt dieser Faktor nur noch bei 4.6.

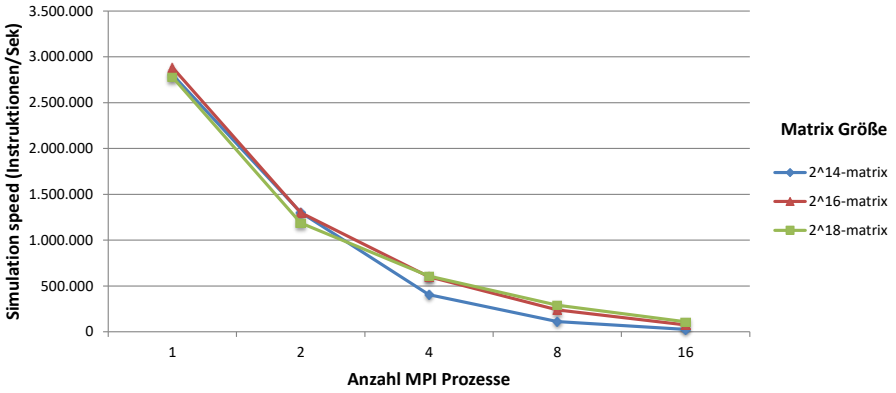


Abbildung 8.2: Simulation Speed der Ausführung von Matrixmultiplikationen mit jeweils unterschiedlicher Matrizengröße mit Bezug auf die Anzahl der MPI Prozesse. Grundlage bildet das KAHRISMA Systeme mit Standardeinstellungen und Hierarchieevaluation.

Die Simulationszeit verlängert sich mit steigender Anzahl an Prozessen, analog dazu sinkt die Simulationsgeschwindigkeit. Abbildung 8.2 zeigt die resultierende Simulationsgeschwindigkeit aus den Simulationszeiten aus Abbildung 8.1. Aus den Simulationsgeschwindigkeiten ist klar ersichtlich, dass die Geschwindigkeit des Simulators einzig von der Anzahl der Prozesse abhängt und nicht von der Eingabegröße, hier der Größe der Matrizen. Die Reduktion der Geschwindigkeit der einzelnen Simulationsdurchläufe ist in Tabelle 8.3 dargestellt.

Die Gesamtperformance des Simulators bei einer Simulation von 16 MPI Prozessen erreicht maximal 1/3 der möglichen Performance einer Simulation mit 1 MPI Prozess. Bei steigender Anzahl der MPI Prozesse wächst der Overhead

für das Management der Prozesse enorm, da SystemC nur einen realen Prozessorkern unterstützt.

		MPI Prozesse				
Matrixgröße	Ergebnisbereich	1	2	4	8	16
$2^7 \times 2^7$	Instanz	100	42.7	18.1	4.3	1.1
	System	100	85.3	72.5	34.5	17.8
$2^8 \times 2^8$	Instanz	100	43.6	19.2	6.8	2.1
	System	100	87.1	76.7	54.2	34.3
$2^9 \times 2^9$	Instanz	100	40.5	19.6	6.8	2
	System	100	81.1	78.6	54.2	32.1

Tabelle 8.3: Reduktion des Simulation Speed verglichen mit dem Referenzsystem bei eingeschalteter Hierarchiesimulation in Prozent.

8.2.4 Einfluss von Simulationsparametern

Die folgenden Ergebnisse werden zur Referenzleistung des Simulators relativ in Bezug gesetzt. Dabei wird der Einfluss des *run-ahead* Parameters auf die Systemsimulation gemessen und bewertet. Ein weiterer Betrachteter Faktor ist die Simulation auf unterschiedlichen horizontalen Hierarchieebenen. Das bedeutet, dass Systeme mit unterschiedlichen Detailgraden simuliert werden und die Leistung des Simulators evaluiert wird. Dabei kann davon ausgegangen werden, dass ein System auf niedrigerer Hierarchieebene, also mit mehr Details, auch ein genaueres Ergebnis für die Anwendungssimulation erzeugt, da Speichereffekte, Kommunikationseffekte oder sonstige im System auftretende Ereignisse berücksichtigt werden können.

Das Ziel dieser Simulation und Parameterevaluation ist es einen sinnvollen Trade-Off zwischen Detailgrad (Hierarchie), run-ahead, Ergebnisgenauigkeit und Simulationsgeschwindigkeit zu erreichen. Dabei muss trotz allem die Korrektheit der Ergebnisse eingehalten werden.

Die Genauigkeit einer Simulation kann nun durch die *Differenz der Taktzyklen* zur Referenzsimulation definiert werden. Die höchste Genauigkeit der Referenzsimulation wird erreicht, wenn alle Informationen aller Module zur Verfügung stehen. Das bedeutet in diesem Fall, dass ein maximal detailliertes Modell mit Evaluation aller horizontalen Hierarchieebenen verwendet wird. Entsprechend wird hier ein minimaler run-ahead mit dem Wert 1 eingesetzt, welcher die KAHRISMA Prozessorkerne zwingt, nach jeder ausgeführten Instruktion eine Synchronisation mit anderen Modulen durchzuführen sowie die verwendeten Taktzyklen zu berechnen.

Dieses Referenzsystem wird in den folgenden Abbildungen als 100% Genauigkeit festgelegt, wohl wissend, dass die Simulation nicht vollständig einer realen Ausführung entspricht.

Die Simulatorperformanz kann des Weiteren von der Simulationszeit abgeleitet werden. Eine kürzere Simulationszeit korreliert mit einer höheren Leistung des Simulators. Die absoluten Simulationszeiten, welche in Abbildung 8.1 dargestellt sind, werden hier als Referenz verwendet, um die weiteren Simulationen daran zu messen.

Analog zur Anzahl der simulierten Taktzyklen, wird die Referenz der Simulationszeiten mit maximaler horizontaler Hierarchieevaluation bei minimalem run-ahead bestimmt.

Um den Einfluss der beiden beschriebenen Parameter zu bestimmen wurde eine große Matrixmultiplikation mit unterschiedlichen Systemkonfigurationen simuliert. In Abbildung 8.3 werden die Ergebnisse der einzelnen Simulationen mit 2 MPI Prozessen bei unterschiedlich belegtem run-ahead Parameter dargestellt. Die gestrichelten Linien stellen dabei ein Simulationssystem ohne Hierarchieevaluation dar, während die durchgezogenen Linien ein System mit vollständiger Hierarchie abbilden. Die Roten Linien geben den Verlauf der Simulationszeit wieder, während die blauen Linien die simulierten Taktzyklen angeben.

Den Referenzpunkt stellt in dieser Evaluation das voll hierarchische Modell bei einem run-ahead von 1 dar und entspricht dem Datenpunkt der durchgezogenen Linien bei run-ahead 1 und 100% relativer Genauigkeit.

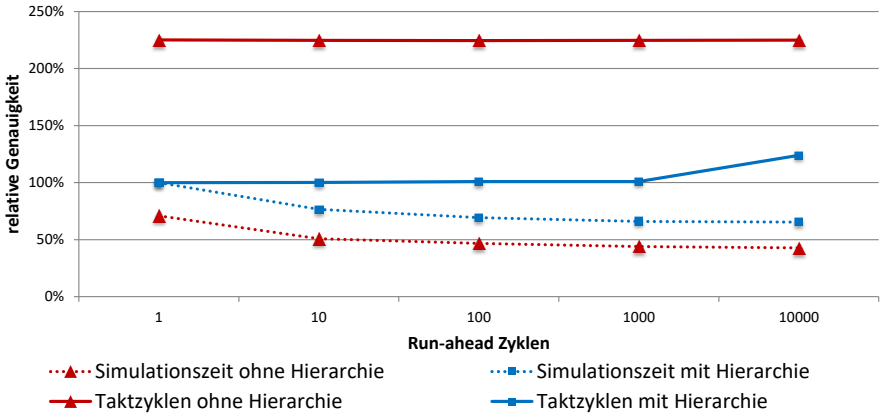


Abbildung 8.3: Simulation Time und Simulationsgenauigkeit einer Simulation einer $2^9 \times 2^9$ Matrix Multiplikation ausgeführt auf einem System mit 2 MPI Prozessen bei Standardeinstellungen. [BOR+14]

Anhand der Ergebnisse lässt sich erkennen, dass die Simulationsgenauigkeit bei vollständiger Hierarchie auch bei größeren Synchronisationsintervallen von bis zu 1000 run-ahead Zyklen sehr stabil bleibt und erst bei noch größeren Intervallen stark ansteigt. Bei deaktivierter Hierarchieevaluation ist zu erkennen, dass die zusätzlichen Module der hierarchischen Simulation einen deutlichen Vorteil bringen, da die berechneten Taktzyklen der Simulation deutlich höher liegen, während der run-ahead Parameter hier kaum einen Einfluss zu haben scheint.

Verglichen zur Genauigkeit des Simulationssystems lässt sich bei der Simulationszeit eine deutliche Verbesserung bei größeren Synchronisationsintervallen feststellen. Der größere run-ahead verringert den Synchronisationsaufwand sowohl bei hierarchischen also auch bei nicht-hierarchischen Simulationsumgebungen.

Es ist allerdings auch zu erkennen, dass hierarchische Modelle von einem größeren run-ahead stärker profitieren als nicht-hierarchische Modelle. Insgesamt zeigen diese Ergebnisse, dass schon bei einem Wechsel von 1 auf 10 run-ahead Zyklen die Simulationszeit in beiden Simulationsumgebungen um etwa 25% reduziert werden kann ohne große Einbußen bei der Genauigkeit. Einen guten Trade-Off bietet hier der Arbeitspunkt bei 1000 run-ahead Zyklen bei einer Reduktion der Simulationszeit von etwa 50%.

Ein ähnliches Bild zeichnet die Evaluation bei der Verwendung von 8 MPI Prozessen, wie in Abbildung 8.4 dargestellt. Deutliche Unterschiede gibt es hier bei der Simulation des nicht hierarchischen Modells, dessen Taktzyklenberechnung von der Verwendung mehrerer Prozesse profitiert. Dies lässt sich durch die Anzahl verfügbarer Quellen erklären, da in diesem Fall 8 Simulationsmodule verwendet werden, welche jeweils eine Taktzyklenberechnung vornehmen, als auch dadurch, dass die Anwendung auf 8 Prozessoren verteilt jeweils kleinere Anteile auf einem einzelnen Prozessor benötigt. Ein einzelnes Simulationsmodul, muss also einen kleineren Anteil ausführen, weshalb der erzeugte Fehler durch fehlende Speichereffekte usw. geringer wird.

Ähnlich zur Evaluation mit zwei Prozessen verhält sich die Hierarchische Simulation weitgehend stabil, erzeugt aber einen früheren Drift bereits bei einem run-ahead von 1000, was sich durch die höhere Anzahl an Prozessen und damit mehr Synchronisationsaufwand erklären lässt.

Damit ist feststellbar, dass bei steigender Prozessanzahl zwar der Synchronisationsaufwand steigt, aber auch dessen Notwendigkeit wächst um genaue Simulationsergebnisse erzielen zu können.

Um von einer möglichst hohen Simulationsgeschwindigkeit zu profitieren verschiebt sich der optimale Arbeitspunkt des Systemsimulators auf einen run-ahead Wert von 100, da schon hier bei geringen Qualitätsverlust eine Beschleunigung um fast 50% bei Verwendung einer hierarchischen Simulation erzielt werden kann.

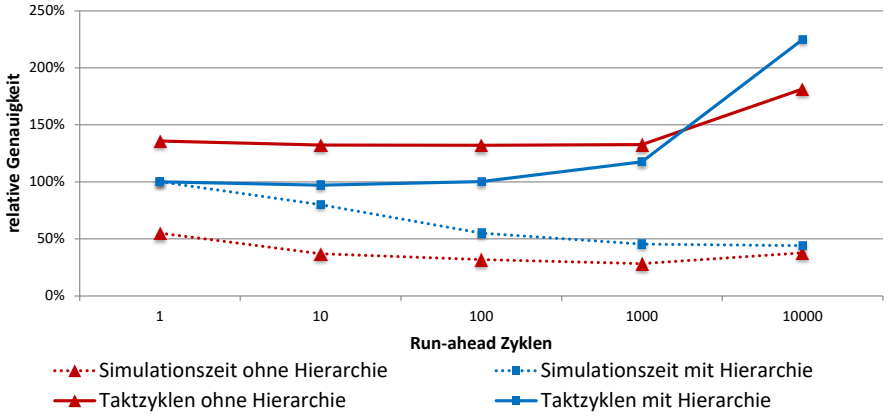


Abbildung 8.4: Simulation Time und Simulationsgenauigkeit einer Simulation einer $2^9 \times 2^9$ Matrix Multiplikation ausgeführt auf einem System mit 8 MPI Prozessen bei Standardeinstellungen. [BOR+14]

Bei der Verwendung von 16 MPI Prozessen in Abbildung 8.5 lassen sich die Trends bestätigen, welche sich schon bei 8 Prozessen abgezeichnet haben. Die nicht-hierarchische Simulation wird relativ gesehen genauer, da Speichereffekte weitaus weniger ins Gewicht fallen. Kommunikation fällt bei dieser Matrix-Multiplikation kaum ins Gewicht, da die Anwendung zu Beginn aufgeteilt, anschließend weitestgehend individuell berechnet und zum Schluss wieder zusammengefügt wird.

Interessant sind die Arbeitspunkte bei sehr großen run-ahead Werten. Bei 10000 run-ahead Zyklen profitiert der Simulator nicht mehr von der Reduzierung des Synchronisationsaufwands, sondern leidet stark unter der folgenden Ungenauigkeit und der fehlerhaften Ausführung von nicht synchronisierten Instruktionen. Dadurch steigt sowohl der generierte Fehler stark an, als auch die Simulationszeit, selbst bei nicht-hierarchischen Modellen.

Tabelle 8.4 stellt nun die Simulationsgeschwindigkeit der einzelnen Simulationsumgebungen bei einem run-ahead von 1000 gegenüber. Im Vergleich zur

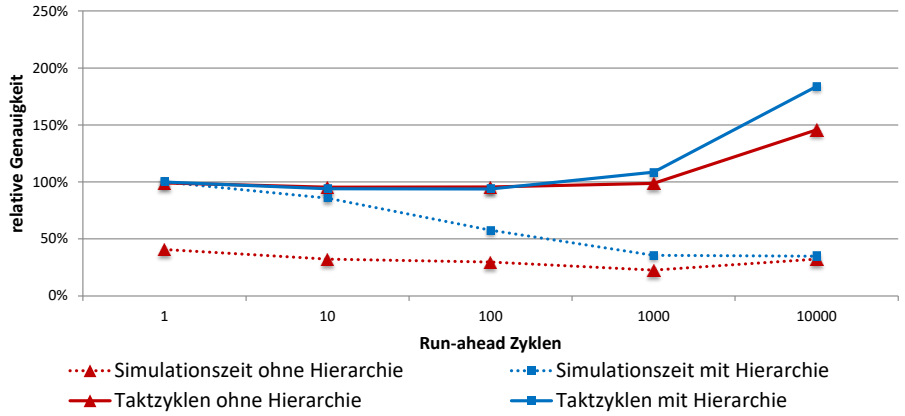


Abbildung 8.5: Simulation Time und Simulationsgenauigkeit einer Simulation einer $2^9 \times 2^9$ Matrix Multiplikation ausgeführt auf einem System mit 16 MPI Prozessen bei Standardeinstellungen. [BOR+14]

Referenztabelle 8.3 ist eine deutliche Verbesserung zu sehen, da die Simulationszeiten nicht schwächer sinken, als bei Verwendung von 1 run-ahead Zyklus. Erkennbar ist, das erstens kleinere Eingangsgrößen weniger stark von einem größeren run-ahead Wert profitieren als größere Programme und zweitens die Simulatorleistung bei einer höheren Anzahl an Prozessen durch einem größeren run-ahead stark verbessert werden kann.

Um die bisherigen Ergebnisse zusammenfassen, zeigt Abbildung 8.6 das Ergebnis einer Entwurfsraumexploration. Die dafür verwendeten Parameter sind als Anwendung die Matrixmultiplikation mit einer Matrizengröße von $2^7 \times 2^7$ bei einer Verwendung von 8 MPI Prozessen für die Ausführung. Der Level 1 Cache besitzt eine Größe von 2KB, während der Level 2 Cache 64KB groß ist. Jeder Punkt in Abbildung 8.6 stellt dabei ein Ergebnis dar, welches sich aus der ermittelten Simulationszeit sowie dem erzeugten Fehler relativ zum Referenzmodell zusammensetzt. Das Referenzmodell verwendet dabei eine vollständige hierarchische Evaluation der Systembeschreibung sowie ein Synchronisationsintervall von 1 Zyklus.

		MPI Prozesse				
Matrizengröße	Ergebnisbereich	1	2	4	8	16
$2^7 \times 2^7$	Instanz	100	46.5	14.4	4	0.9
	System	100	92.9	57.8	32.2	15.1
$2^8 \times 2^8$	Instanz	100	45	20.8	8.3	2.6
	System	100	90.1	83.3	66.4	41.8
$2^9 \times 2^9$	Instanz	100	42.8	21.9	10.4	3.9
	System	100	85.6	87.5	83.3	62.8

Tabelle 8.4: Reduktion des Simulation Speed bei einem run-ahead von 1000 Zyklen bei eingeschalteter Hierarchiesimulation in Prozent.

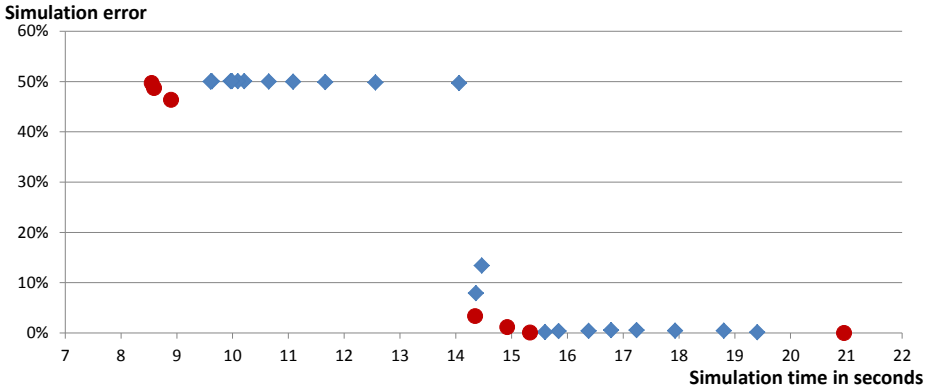


Abbildung 8.6: Ergebnisse einer Entwurfsraumexploration, welche den Kompromiss zwischen Genauigkeit und Simulationszeit verdeutlicht. In Rot sind Pareto-optimale Arbeitspunkte dargestellt. [BOR+14]

Das Ergebnis zeigt, dass sich aus den unterschiedlichen Kombinationen zwischen Detailgrad und Synchronisationsintervall unterschiedliche mögliche Arbeitspunkte ergeben, welche jeweils pareto-optimale Lösungen darstellen können.

8.2.5 Matrizengröße

Im vorherigen Abschnitt wurde beschrieben, welchen Einfluss die Simulationsparameter der ADL auf die tatsächliche Simulation ausüben. Besonders der Hierarchieparameter ist dabei auch für die Größe der zu simulierenden Anwendung interessant.

In diesem Abschnitt soll daher ermittelt werden wie die Simulationsparameter mit der zu simulierenden Last auf dem Simulator zusammenhängen.

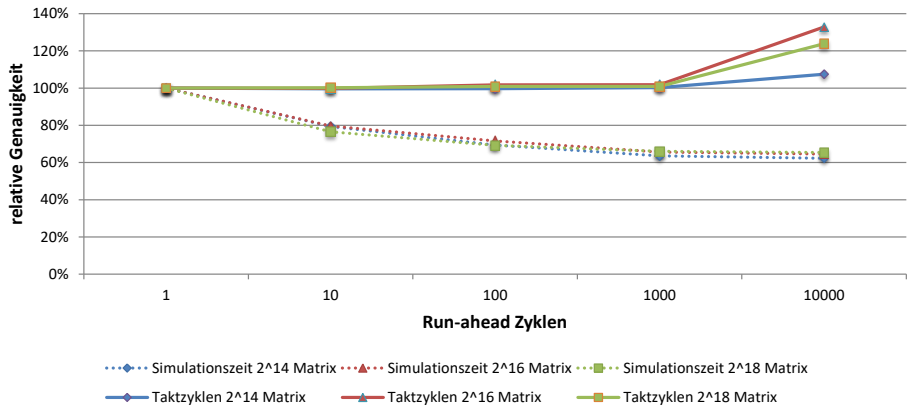


Abbildung 8.7: Simulation Time und simulierte Taktzyklen einer Simulation unterschiedlich großer Matrix Multiplikationen ausgeführt auf einem System mit 2 MPI Prozessen in Bezug auf den run-ahead bei Standardeinstellungen mit Hierarchieevaluation.

Abbildung 8.7 zeigt eine Simulationsumgebung mit zwei MPI Prozessen, welche für die Multiplikation von großen Matrizen verwendet wird.

Bis zu 1000 run-ahead Zyklen zeigt sich, dass sowohl Genauigkeit als auch die Simulationszeit für alle drei Matrizengrößen nur wenig voneinander abweicht. Erst bei einem run-ahead von 10000 zeigen sich wie in Abbildung 8.3 entsprechende Abweichungen bei der Genauigkeit der Simulation, während die Simulationszeiten nahe beieinander liegen. Dies lässt den Schluss zu, dass kleine

Simulationssysteme mit wenigen Prozessoren bzw. Prozessen, von der Größe der Anwendung kaum beeinflusst werden.

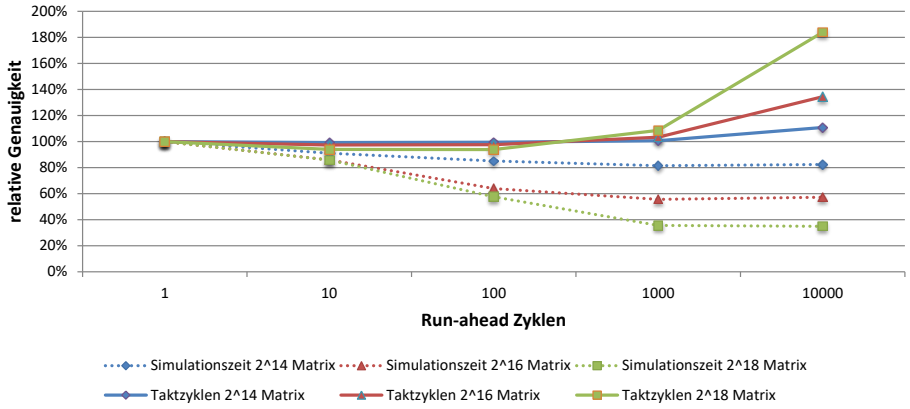


Abbildung 8.8: Simulation Time und simulierte Taktzyklen einer Simulation unterschiedlich großer Matrix Multiplikationen ausgeführt auf einem System mit 16 MPI Prozessen in Bezug auf den run-ahead bei Standardeinstellungen mit Hierarchieevaluation.

Bei größeren Simulationssystemen, beispielsweise mit 16 MPI Prozessen in Abbildung 8.8, ist dieser Einfluss stärker messbar. Schon bei einem run-ahead von 10 Zyklen hat die Eingangsgröße einen sichtbaren Einfluss auf die Simulationszeit. Bei größeren Lasten, kann der Simulator besser von den Verbesserungen durch den run-ahead profitieren als bei kleineren Matrizen. Im Gegenzug leidet allerdings die Qualität bei größeren Matrizen stärker und erreicht bei kleinen Matrizengrößen eine bessere Genauigkeit.

8.2.6 Speicherfrequenzen und Cachegrößen

Während dieser Simulation wurden bisher alle Module bei gleicher simulierter Taktfrequenz, 500Mhz, evaluiert. In vielen modernen heterogenen SoC Syste-

men ist es allerdings oftmals so, dass unterschiedliche Komponenten auch mit unterschiedlichen Taktfrequenzen betrieben werden.

Deshalb wird in diesem Abschnitt der Zusammenhang zwischen unterschiedlichen Modulfrequenzen im Simulator ermittelt. Dies geschieht im Rahmen einer Entwurfsraumexploration bei der eine Cache-Hierarchie bestehend aus unterschiedlichen First Level (L1) und Second Level (L2) Cache Größen für die Matrixmultiplikation evaluiert wird. Die Exploration erfolgt mit Hilfe der ADL, indem Parameter entsprechend modifiziert werden.

In dieser Evaluation wurde alle Simulationsumgebungen mit Hierarchieevaluation erstellt, um die Cache-Module mit in die Simulationsumgebung zu integrieren. Dadurch befinden sich alle Simulationsumgebungen auf dem gleichen Abstraktions- und Detaillevel. Die Anzahl der simulierten Taktzyklen beschreibt deswegen nicht mehr die Qualität der Simulation an sich, sondern die Performanz, bzw. Ausführungszeit der zu simulierenden Anwendung. Der Systemsimulator rechnet dabei mit einer abstrakten Anzahl an Taktzyklen, so dass unterschiedliche Modulfrequenzen ausgeglichen werden. Eine geringere Anzahl an Taktzyklen spiegelt demnach eine höhere Performanz des zu simulierenden Systems wider.

Zusammenfassend kann vorweggenommen werden, dass die Speicherfrequenzen einen großen Einfluss auf das Gesamtsystem haben, sofern die lokalen Arbeitsdaten größer sind als die einzelnen Caches.

Abbildung 8.9 zeigt nun die Auswirkungen unterschiedlicher Cache Größen auf die Anzahl der simulierten Taktzyklen. Da jeder Prozessor einen eigenen L1 Cache besitzt ist dessen Größe dominant gegenüber kleinen L2 Caches. Die benötigten Taktzyklen der Anwendung können deutlich reduziert werden, wenn die Arbeitsdaten vollständig in den L1 Cache passen. Dadurch ist keine Auslagerung auf den Hauptspeicher mehr notwendig, was in der Exploration bei einer Cachegröße von 1024 KB erreicht wird (siehe grüne Linien in Abbildung 8.9).

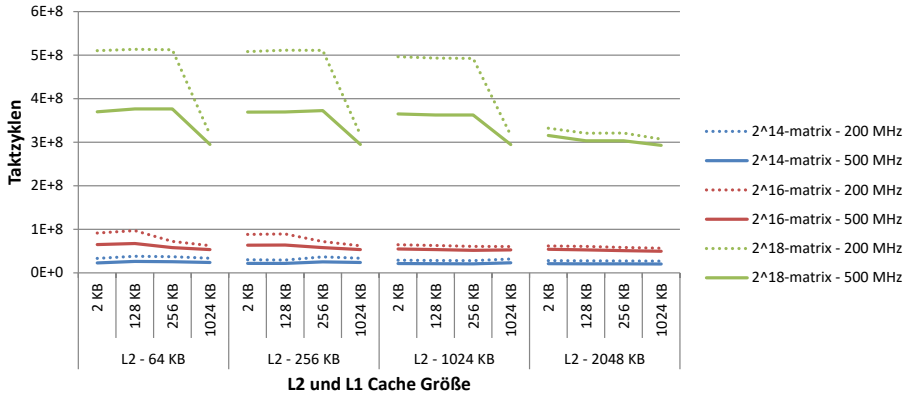


Abbildung 8.9: Einfluss der Speicherfrequenz und der Cachegrößen auf die simulierten Taktzyklen für unterschiedliche Matrizengrößen bei Verwendung von zwei MPI Prozessen. [Brä13]

Die Arbeitsdaten eines MPI Prozesses hängen von der Größe der Matrizen sowie vom verwendeten Datentyp ab. Bei 32Bit Datentypen erreichen die Arbeitsdaten folgende in Tabelle 8.5 dargestellten Werte.

Matrizengröße	Anzahl Elemente	Elementgröße	Arbeitsdaten
$2^7 \times 2^7$	16384	32 bit/4 byte	64 KB
$2^8 \times 2^8$	65536	32 bit/4 byte	256 KB
$2^9 \times 2^9$	262144	32 bit/4 byte	1024 KB

Tabelle 8.5: Größe eines Arbeitsdatensatzes eines MPI Prozesses bei unterschiedlichen Matrizengrößen

Da jeder MPI Prozess eine eigene Kopie der Daten erhält, müssen die Arbeitsdaten entsprechend mit der Anzahl der MPI Prozesse multipliziert werden.

Bei zwei MPI Prozessen ergeben sich so Systemarbeitsdaten von 128KB, 512KB und 2MB. Diese Daten sind wichtig für die Größe des L2 Cache.

Betrachtet man die in Abbildung 8.9 dargestellten Ergebnisse, lassen sich für unterschiedlich große Matrizen die folgenden Schlussfolgerungen ableiten.

Kleine Matrizen, blaue Linien, können von einem größeren L1 Cache nicht nur profitieren. Wächst der L1 Cache, während der L2 Cache zu klein ist, um die Systemarbeitsdaten aufzunehmen, so erhöht sich auch die Ausführungszeit der Matrix Multiplikation. Bestenfalls lässt sich eine Beschleunigung der Anwendung um 11% gegenüber den Standardwerten der KAHRISMA Architektur, siehe Tabelle 8.2, erzielen.

Bei mittelgroßen Matrizen, rote Linien, zeigt sich, dass diese besser auf einen größeren L1 reagieren, wobei auch hier bei kleinen L2 Größen zunächst eine Stagnation der Leistung eintritt und erst mit ausreichender Cachegröße eine signifikante Beschleunigung erreicht werden kann. Werden schnellere Speicher verwendet, so sind die Auswirkungen der Cachegrößen nicht mehr so ausschlaggebend für die Performanz des Systems.

Bei großen Matrizen, grüne Linien, haben die einzelnen Cachegrößen erst dann deutliche Auswirkungen, sobald die Arbeitsdaten eines Prozesses in den L1 Cache oder die Gesamtsystemdaten in den L2 Cache passen. Ist eines von beiden der Fall so hat die Speicherfrequenz kaum Einfluss auf die Leistung des Systems. Hier zeigt sich auch der Einfluss des L2 Cache besonders, dass bei ausreichender Größe des L2 Cache, der L1 Cache in den Hintergrund rückt. Bei den beiden anderen Matrizengrößen zeigt sich dieses Verhalten schon bei einer L2 Cache Größe von 1024 KB.

8.2.7 Einfluss der MPI Effekte auf die Simulatorleistung

Die Anzahl der Verwendeten MPI Prozesse hat zudem einen erheblichen Einfluss auf die Performanz eines Multicoresystems. Abbildung 8.10 zeigt die benötigten Taktzyklen einer MPI Instanz welche für die Berechnung unterschiedlicher Matrizengrößen benötigt werden. Kleine und mittlere Matrizen profitieren entweder nicht oder nur marginal von der Parallelisierung da der Aufwand

für die Synchronisation der MPI Prozesse zu groß ist. Erst bei den größeren Matrizen ist eine massive Beschleunigung bis zu vier Prozessen sichtbar.

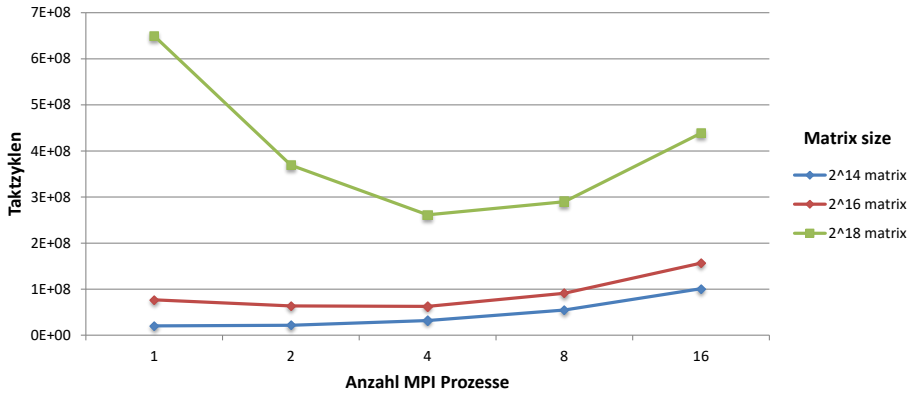


Abbildung 8.10: Simulierte Taktzyklen einer Systemsimulation bei maximaler Genauigkeit (mit Hierarchieevaluation und minimalem runahead) in Bezug auf die verwendeten MPI Prozesse. [Brä13]

8.3 Hierarchische Netzwerkanalyse für Multicore Systeme

Während im vorherigen Abschnitt schon gezeigt wurde, dass die hierarchische Beschreibung durch die ADL in Verbindung mit dem flexiblen Multicoresimulator auf Prozessorebene einsetzbar ist um Speicherhierarchien zu evaluieren und zu explorieren, wird in diesem Abschnitt der Fokus auf die Implementierung einer hierarchischen Netzwerkkommunikation gelegt. Dabei wird wiederum das KAHRISMA Multicore System verwendet. Allerdings mit dem Unterschied, dass das Netzwerkmodul, welches für die Kommunikation zwischen den Prozessoren verwendet wird auf hierarchische Weise beschrieben wird, mit dem Ziel mehrere Abstraktionsebenen zur Verfügung zu stellen.

Für die Beschreibung und die Simulation stehen nun zwei unterschiedliche Module zur Verfügung. Einerseits ist dies ein Funktionales Netzwerkmodul, basierend auf einer direkten FIFO Kommunikation zwischen den Prozessorkernen und andererseits auf einer zyklenakkuraten Netzwerksimulation des invasive Network-on-Chip (iNoC) basierend auf der Arbeit von Heißwolf [Hei15]. Dazu wurden ein Netzwerkadapter entwickelt, welcher die Synchronisation zwischen den approximativen Simulationsmodulen der Kahrisma Architektur und dem zyklenakkuraten Netzwerkmodul übernimmt.

Ziel dieser Evaluation ist es darzustellen welche Möglichkeiten durch die hierarchische Beschreibung auch für Kommunikationskomponenten zur Verfügung stehen, sowie die Untersuchung der Übertragungsmöglichkeiten und Anforderungen an die Applikation.

8.3.1 Anwendung

Für die Evaluation der Netzwerksimulation kommt eine einfache PingPong Anwendung zum Einsatz, da diese ausreichend ist um die Auswirkungen der Kommunikationsparameter auf die Simulationsgenauigkeit zu zeigen. Da die Ausführung der Anwendung fast ausschließlich durch Kommunikation zwischen den Prozessoren bestimmt ist, lassen sich die einzelnen Verarbeitungsschritte innerhalb der Simulation sehr gut abbilden.

8.3.2 Grundlegende Parameter

Die Netzwerksimulation besteht aus zwei Komponenten, einer funktionalen und einer zyklenakkuraten Kommunikationssimulation. Dabei werden folgende wesentlichen Parameter eingesetzt: Die Array bzw. Netzwerkgröße wird bestimmt durch die Anzahl an Prozessoren, EDPEs, im Multicoresystem. Das Verbindungsdelay, also die Zeit, welche ein Datum von einem Kern zum anderen

Kern benötigt, ist bei der funktionalen Simulation auf 20 Zyklen festgelegt. Bei der zyklenakkuraten Simulation wird diese Übertragungszeit durch die einzelnen Komponenten im System und im Netzwerk, wie beispielsweise Router, bestimmt und jeweils für jedes Datenpaket berechnet.

Daneben gibt es noch die Paketgröße, welche in der funktionalen Simulation nicht zum Einsatz kommt, da die Daten direkt über FIFO-Verbindungen an den Empfängerprozessor übertragen werden. Neben weiteren Parametern, die von der zyklenakkuraten Netzwerksimulation des iNoC [Hei15], bereitgestellt werden, ist die Paketgröße allerdings ein interessanter Parameter für die Parallelisierung von Anwendungen. Unterschiedliche Einstellungen der Paketgröße sind in Tabelle 8.6 aufgelistet.

Name	**Pakete**	**Delay**	**Beschreibung**	**Applikation**
Blackbox	1	20 Zyklen	funktionales Netzwerk	standard
iNoC Pkg2	2	variabel	iNoC Netzwerk mit Paketgröße 2, entspricht Blackbox Funktion	standard
iNoC Pkg8	8	variabel	iNoC Netzwerk mit maximaler Paketgröße 8	erweitert
iNoC Pkg16	16	variabel	iNoC Netzwerk mit maximaler Paketgröße 16	erweitert

Tabelle 8.6: Simulationsvarianten der Netzwerksimulation

8.3.3 Ausgangsbasis der Simulation

Die Funktionale Netzwerksimulation besteht aus Kahrisma Prozessorkernsimulationen an deren Ports ein Netzwerksimulationsmodul direkt angeschlossen ist. Dieses besteht aus direkten Verbindungen zwischen zwei Ports, welche

dynamisch bei Bedarf eines Kerns hergestellt werden. Eine Verbindung wird durch einen FIFO-Channel realisiert, dessen Reaktionszeit, respektive Latenz in Zyklen, durch die ADL angepasst werden kann. Die Verbindung unterstützt blockierendes und nicht-blockierendes Senden und Empfangen, sowie auch Kombinationen davon. Die durch den FIFO-Channel entstehende, bzw. programmierbare, zeitliche Verzögerung wird als Wartezyklus der Verarbeitungszeit des jeweiligen Kernsimulators zugeordnet.

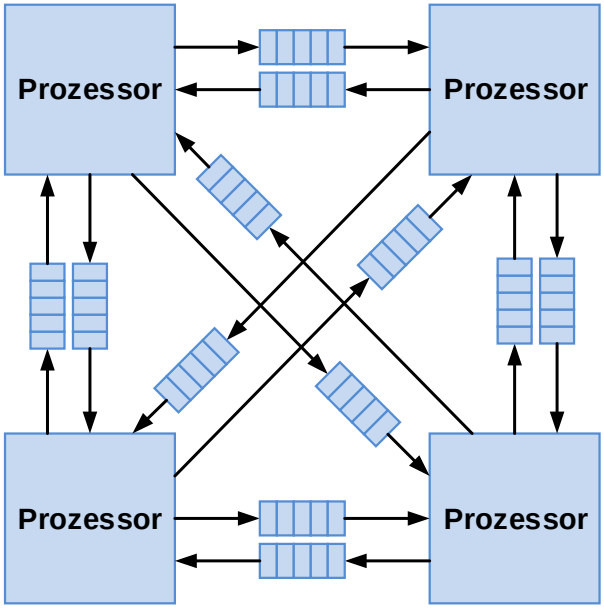


Abbildung 8.11: Aufbau der funktionalen Simulation

Dadurch lässt sich eine Funktionale Netzwerksimulation erzeugen und simulieren. Die FIFOs erlauben den mehrfachen Aufbau von Verbindungen zum

selben Simulationsmodul, welches sich in diesem Fall selbstständig um das Handling der Verbindungen kümmern muss.

Die Ungenauigkeit des Simulationsmodells entsteht dabei durch die festgelegten Latenzen des Netzwerkmoduls, welche auf alle Verbindungen angewandt wird allerdings keine Auslastungsmetriken oder weitere Verbindungseigenschaften wie die Länge der Verbindung, Anzahl der Hops, Zwischenspeicher oder Verarbeitungszeiten in Infrastrukturkomponenten berücksichtigt.

8.3.4 Netzwerksimulation

Demgegenüber erlaubt es die hierarchische Beschreibung der ADL auch alternative Simulationsmodule einzusetzen. Der Aufbau des detaillierteren zur Verfügung stehenden Modells besteht aus einem Netzwerkmodul sowie zusätzlichen Netzwerkadapter Modulen, die nun die Knotenpunkte zwischen den Prozessorsimulationsmodulen und dem Netzwerkmodul übernehmen.

Dazu stellen diese entsprechende Puffer-Speicher zur Verfügung und übernehmen sowohl Strukturen als auch Aufgaben zum Verbindungsauf- und Abbau sowie dem Verbindungsmanagement.

Das Netzwerkmodul beinhaltet dabei Netzwerkrouter, welche interne Verbindungskomponenten wie Buffer, Ports und die eigentlichen Netzwerklinks implementieren. Darüber lassen sich nun detaillierte Analysen und Kommunikationsprofile erstellen und die Ergebnisse daraus in die automatisierte Parallelisierung von Software für eingebettete Multicoresysteme einbringen. Die Modellierung des iNoC Netzwerkmoduls wird durch die hierarchische Implementierung des funktionalen Netzwerkmoduls innerhalb der ADL realisiert.

Eine erneute Beschreibung ist dadurch nicht notwendig. Die zyklenakkurate Simulation zusammen mit den Netzwerkadaptern kann durch die Deaktivierung des funktionalen Modells in der ADL der Simulationsumgebung hinzugefügt werden.

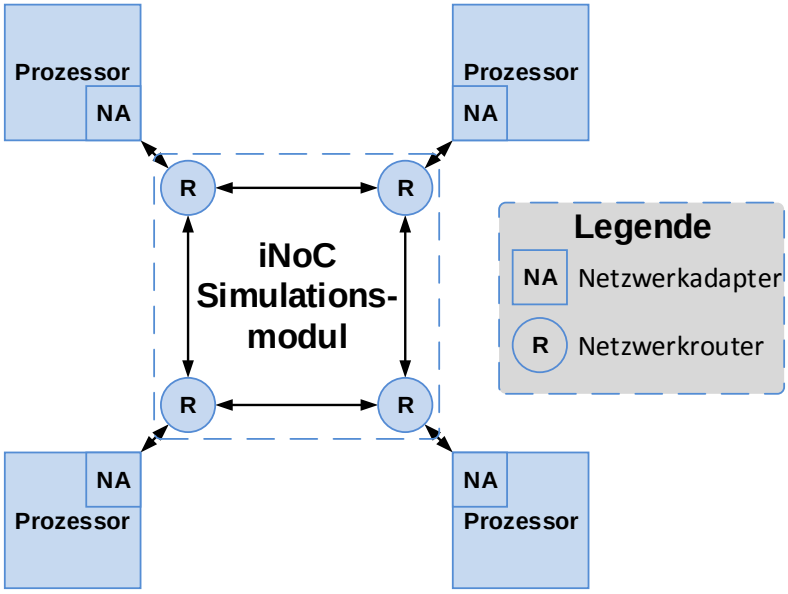


Abbildung 8.12: Aufbau der zyklenakkuraten Netzwerksimulation

Die zusätzlichen Netzwerkadaptermodule übernehmen dabei die zusätzliche Aufgabe der zeitlichen Synchronisation zwischen den zyklenapproximativen Prozessorsimulatoren und der zyklenakkuraten Netzwerksimulation. Die Genauigkeit wird nun dadurch erhöht, dass einerseits eine zyklenakkurate Kommunikationssimulation verwendet wird und andererseits die genaueren Kommunikationswerte in die approximativen Simulationsmodelle übertragen werden können, indem die berechneten Zyklen an den Prozessor als Wartezeiten individuell für jede einzelne Verbindung übergeben werden. Zur Kommunikationsanalyse wurden darüber hinaus unterschiedliche Features des iNoC implementiert. So unterstützt das iNoC auch unterschiedliche Paketgrößen, welche Einfluss auf die Auslastung des Netzwerks und damit die Performanz einer

parallelen Applikation haben können. Diese Eigenschaften stehen dem funktionalen Modell nicht zur Verfügung.

8.3.5 Netzwerkanalyse und Simulationsergebnisse

Vergleicht man die Simulationszeit einer einfachen PingPong Anwendungen mit unterschiedlich großen Datenpaketen, siehe Abbildung 8.13, so wird deutlich, dass die zyklenakkurate Netzwerksimulation sehr viel langsamer abläuft als eine rein funktionale Implementierung.

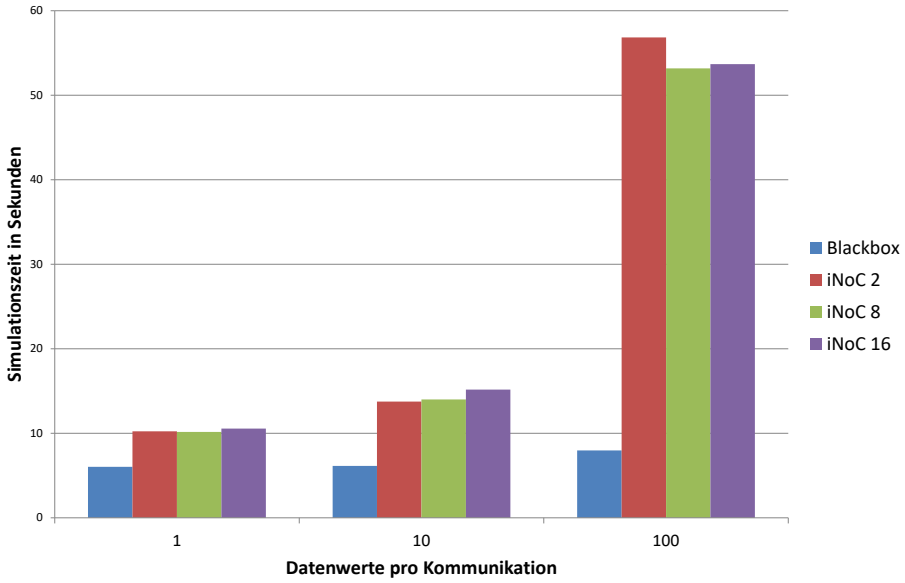


Abbildung 8.13: Darstellung der Simulationszeit in Sekunden unterschiedlicher Simulationskonfigurationen des Netzwerkmoduls. [PB14]

Doch lassen sich mit der zyklenakkuraten Netzwerksimulation deutlich genauere Analysen über das Kommunikations- und Zeitverhalten des Multico-

resystems durchführen. Dies ist in den Abbildungen 8.14 und 8.15 dargestellt, in denen die durchschnittlich benötigten Zyklen pro Paket insgesamt und pro Nutzdatenwort berechnet und dargestellt werden.

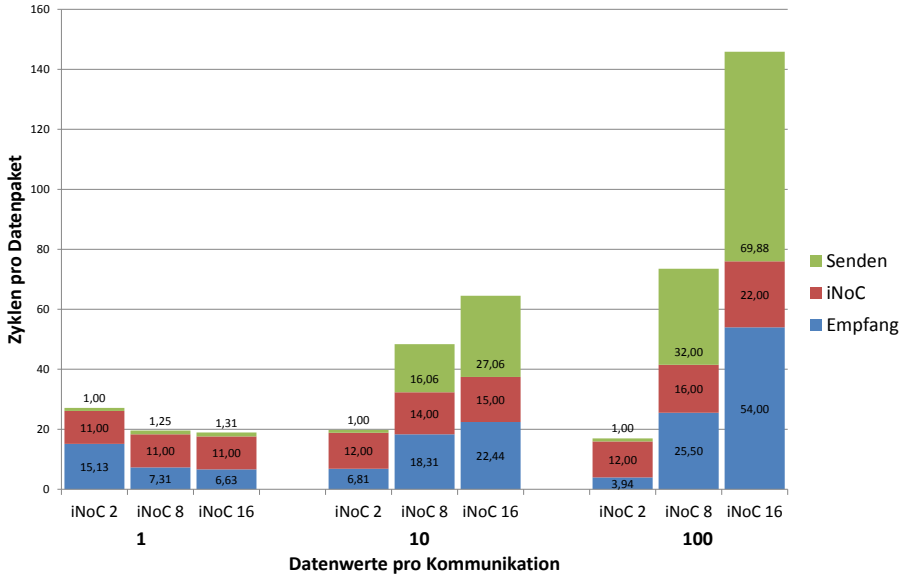


Abbildung 8.14: Durchschnittlich benötigte Anzahl an Zyklen pro übertragenem Paket bei unterschiedlicher Paketgröße je nach zu übertragendem Datensatz. [PB16]

Schon durch die Ausführung des einfachen PingPong Benchmark zwischen zwei Prozessoren lässt sich feststellen, dass die Kommunikation zwischen Prozessoren stark Anwendungsabhängig ist. Die durchschnittlichen Transport- und Verarbeitungszeiten zwischen zwei Prozessorinstanzen hängt stark von der Größe der zu übertragenden Daten, als auch der zurückzulegenden Strecke ab. Ein Detailparameter, den das iNoC Simulationsmodul über die ADL-Beschreibung nutzbar macht ist die Möglichkeit eine maximale Paketgröße zu definieren. Tabelle 8.7 stellt die Evaluationsergebnisse der effektiv genutzten

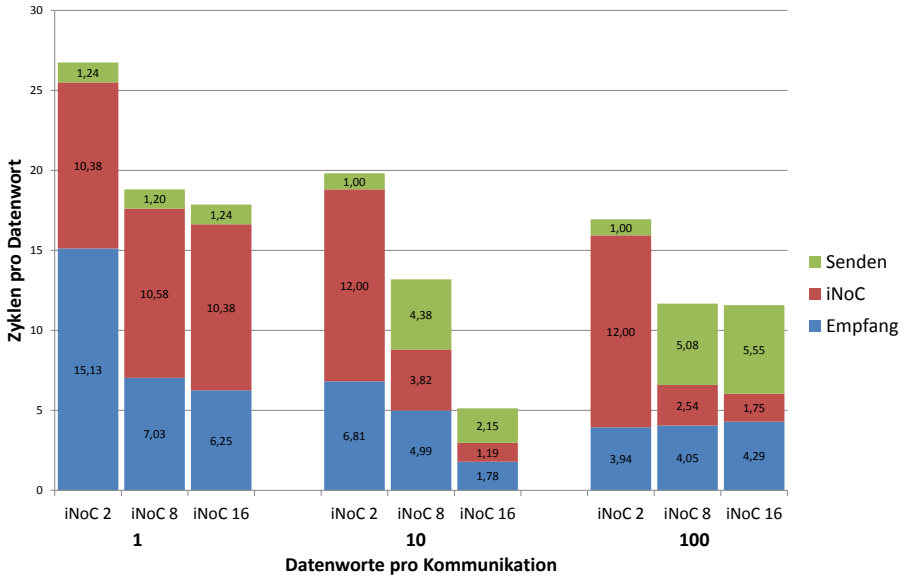


Abbildung 8.15: Durchschnittlich benötigte Anzahl an Zyklen pro übertragenem Datenwort bei unterschiedlichen Paketgrößen je nach zu übertragendem Datensatz. [PB16]

Paketgröße für unterschiedliche Datensätze an, sowie die durchschnittlich zur Übertragung benötigten Zyklen.

Dabei ist erkennbar, dass größere Datensätze auch von größeren Paketgrößen profitieren, diese allerdings auch in eine Sättigung laufen. In diesem Beispiel sind zwei unterschiedliche pareto-optimale Arbeitspunkte zu erkennen. Für große Datensätze ist eine Konfiguration mit einer Paketgröße von 8 Datenworten nur minimal langsamer als eine Konfiguration mit 16 Datenworten, bietet allerdings eine bessere durchschnittliche Paketauslastung. Bei einer Realisierung eines Netzwerks mit einer Paketgröße von maximal 8 Datenworten könnten darüber hinaus Zwischenspeicher eingespart und Verarbeitungseinheiten weniger komplex ausfallen. Ein anderes Ergebnis stellt der Arbeitspunkt für

Tabelle 8.7: Durchschnittliche effektive iNoC Paketgröße sowie resultierende durchschnittliche Kommunikationszeiten

Maximale Paketgröße	Datensatzgröße klein (1)	mittel (10)	groß (100)
effektive durchschnittliche Paketgröße, in Klammer Paketnutzung			
2 words	2,00 (100%)	2,00 (100%)	2,00 (100%)
8 words	2,04 (25%)	4,67 (58%)	7,30 (91%)
16 words	2,06 (13%)	6,50 (40%)	13,60 (85%)
durchschnittliche Kommunikationsszyklen pro Datenwort			
2 words	27,0	19,0	16,0
8 words	18,3	12,5	10,8
16 words	17,0	4,8	10,7

mittlere Datensatzgrößen dar. Da dieser vollständig in ein Netzwerkpaket der Größe 16 passt, ist hier eine deutlich kürzere Kommunikationszeit zu ermitteln als bei allen anderen Konfigurationen.

Damit wird deutlich, dass die maximale Paketgröße eine applikationsspezifische Größe darstellt, welche im Umfeld eingebetteter Multicoreprozessoren von Applikation zu Applikation variiert. Daher ist diese ein gutes Beispiel dafür, wie die Implementierung von ADL-Modulen dazu verwendet werden kann, um Einfluss auf die Komplexität und den Umfang einer Entwurfsraumexploration zu nehmen.

8.4 Adaptive Multi-Level-Simulation

Für eine effektive Verwendung von Parallelisierungswerkzeugen im Entwicklungsprozess müssen die Verfahren zur Parallelisierung sowohl hinsichtlich ihrer Qualität als auch hinsichtlich Ihrer Geschwindigkeit und Performanz zur Lösungsfindung hin möglichst optimale Ergebnisse liefern. Diese Optimierungsziele lassen sich daraus auch auf Simulationssysteme ableiten. Einerseits erfordert eine hohe Parallelisierungsqualität eine möglichst genaue Messung

der Performanz und Optimierungsparameter der Hardware als auch der parallelisierten Anwendung. Auf der anderen Seite bedingt eine genaue und akkurate Simulation auch eine lange Ausführungszeit des Simulationssystems. Wird die Simulation auf Geschwindigkeit hin optimiert, so müssen oftmals Details verborgen werden, wodurch Ungenauigkeiten entstehen können, welche sich negativ auf das Parallelisierungsergebnis auswirken werden. Auf Basis der in dieser Arbeit vorgestellten ADL mit ihrem horizontalen als auch vertikalen Hierarchiekonzept, kombiniert mit dem flexiblem Simulationsframework wurde eine adaptive Evaluationsumgebung entwickelt, die in der Lage ist, sich den jeweiligen Anforderungen der Applikation, des Hardwaresystems als auch der Parallelisierungsqualität anzupassen.

Dadurch wird der Einsatz eines Simulationssystems für iterative Parallelisierungsansätze ermöglicht, welche sonst durch lange Simulationslaufzeiten nicht anwendbar wären oder durch zu ungenaue Simulationen unzureichende Ergebnisse erzielen würden.

Das maßgebliche Ziel einer Parallelisierung ist die Beschleunigung der Ausführung bei gleichzeitig relativer Verringerung des Energieverbrauchs. Dabei spielen verschiedenste Faktoren eine Rolle um Prozessoren nicht über Gebühr zu belasten sowie um Kommunikationskosten zu minimieren.

Betrachtet man nun die Beschleunigung als primäre Optimierungsfunktion der Parallelisierung, so kann diese als das Verhältnis zwischen sequentieller Ausführungszeit und der Summe von Paralleler Ausführungszeit und Kommunikationskosten bestimmt werden. Damit ergibt sich eine Beschleunigung wie folgt:

$$Speedup = \frac{t_{sequentiell}}{t_{parallel} + t_{Kommunikation}} \tag{8.3}$$

Für die Parallelisierung einer gesamten Anwendung kann diese Formel auch auf einzelne Tasks abgeleitet werden, so dass auf jeden Task einzeln eine Paral-

lelisierung angewandt werden kann wenn die entsprechenden Ressourcen zur Verfügung stehen und wenn gilt:

$$t_{parallel} + t_{Kommunikation} < t_{sequentiell} \tag{8.4}$$

Die einzelnen Ausführungszeiten und Kommunikationszeiten können dabei von Anwendung zu Anwendung, von Hardware zu Hardware und von Scheduling zu Scheduling stark unterschiedlich sein.

Um die Einhaltung dieser Gleichung zu überwachen ist daher eine möglichst genaue, aber auch sehr rechenaufwändige, Evaluation der Performanzparameter notwendig.

Dies macht die Qualität einer automatisierten Parallelisierung stark abhängig von der Qualität der Evaluationsergebnisse. Der SystemProfiler des Simulators ermöglicht nun eine taskspezifische Messung der Performanzparameter die der Adaptivität des Simulationssystems in nichts nachsteht. Dies wird durch die Kombination akkurater Simulationen mit hoher Genauigkeit und funktionalen Simulationen mit hoher Geschwindigkeit ermöglicht.

Implementiert wurde nun eine iterative selbst-adaptierende Simulationsumgebung, die Evaluationsergebnisse in die Architekturbeschreibung selbst zurückführt, dadurch Simulations- und Verhaltensparameter anwendungsfallspezifisch adaptiert und damit die Genauigkeit nachfolgender Simulationen verbessert, während auf die höheren Geschwindigkeiten einer schnelleren Simulation gesetzt werden kann.

Das Ergebnis ist eine Feedback-Schleife in die die Parallelisierungswerkzeuge aktiv eingebettet wurden, wie sie in Abbildung 8.16 dargestellt ist.

8.4.1 ADL Update Loop

Der erste Schritt dieser iterativen Optimierung ist die Ausführung einer Tiefensuche entlang der Vertikalen Implementierung der Simulationsmodule um

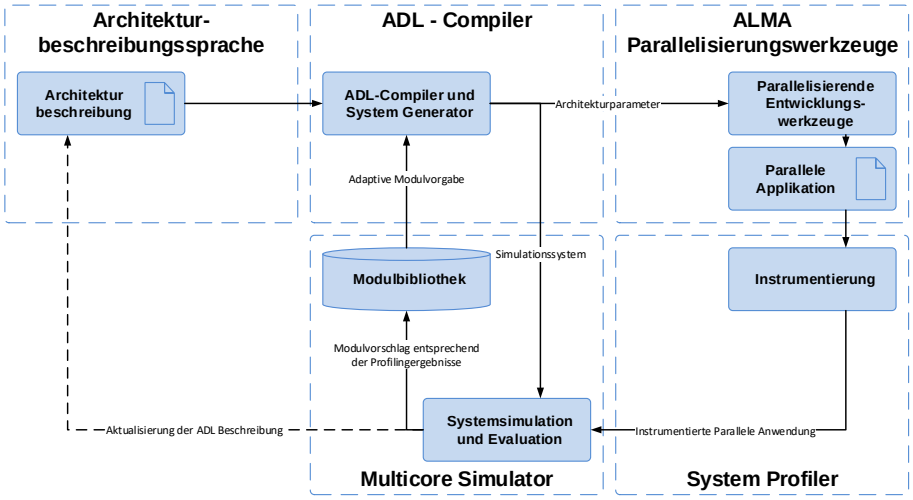


Abbildung 8.16: Feedback-Schleife im Framework von ADL über Simulation zurück zur ADL

die genauesten Simulationsklassen und damit das geringste Abstraktionslevel eines Moduls zu finden.

Auf diese Weise wird eine Simulationsumgebung mit maximal verfügbarer Genauigkeit, dem höchsten Detailgrad und dem niedrigstmöglichen Abstraktionslevel über alle Module hinweg erzeugt. Das bedeutet, es gibt keine Kombination an Simulationsmodulen innerhalb der Simulationsbibliothek, welche die modellierten Systemmodule genauer simulieren und evaluieren kann. Diese Simulationsumgebung wird nun verwendet um eine initiale Simulation des Systems und der parallelisierten Anwendung durchzuführen. Der generierte parallele Programmcode, sowie nach Verfügbarkeit auch die Simulationsmodule, werden instrumentiert um ein Applikationsprofiling, sowie ein Statustracing der Module auszuführen wodurch Performance Parameter, Kommunikationskosten, Auslastungs- und Nutzungsstatistiken erzeugt werden. Dieses Set an gesammelten Informationen stellt die genauesten Profiling Informationen dar, welche über das Simulationssystem generiert werden können. Sie wer-

den nun über die Feedback-Schleife der ADL aggregiert und in die höheren Abstraktionsebenen und Modelle zurückgeführt. Eine Aggregation kann beispielsweise dadurch implementiert werden, dass zunächst über eine dedizierte Verbindung über das Netzwerk eine Summe der Übertragungszeit ermittelt wird. Eine zweite Aggregation kann anschließend die Durchschnittswerte für Kommunikationszeiten im gesamten Netzwerk ermitteln, welche an eine funktionale Netzwerksimulation übergeben werden. Die Art der Aggregation ist abhängig davon, welche Verhaltensparameter auf der Elternebene verfügbar sind. Da ein Modul nur seine direkten Eltern- und Kind-Module kennt müssen Aggregationen über mehrere Ebenen hinweg mit mehreren Aggregationsstufen implementiert werden.

8.4.2 Evaluation der Feedback-Schleife

Um dieses Ansatz zu evaluieren wurde ein Multicoresystem, bestehend aus einem Array an Verarbeitungseinheiten sowie einem Verbindungsnetzwerk verwendet um parallele Anwendungen zu simulieren. Jede Verarbeitungseinheit wird dabei durch einen Prozessor aus dem Kahrisma System repräsentiert und über einen zyklenapproximativen Instruktionssatzsimulator simuliert.

Diese Simulatoren wurden mit Hilfe eines SystemC Wrappers (SysSimModule) implementiert deren Kommunikation untereinander auf TLM Funktionen basiert. Mit Hilfe der horizontalen Implementierung dieses Arrays und der Individualisierungsmöglichkeiten der ADL, könnten jedem Prozessor auf diese Weise individuelle Parameterwerte für beispielsweise Taktfrequenz, Speicherhierarchie, Cache-Verhalten etc. zugewiesen werden.

Verschiedenste Module dieser Systembeschreibung erlauben außerdem die Verwendung der vertikalen Implementierung. Das verwendete Prozessormodul erlaubt beispielsweise die Integration eines Level-1 Cache, sowie eines dedizierten lokalen Speichers wenn die zweite Abstraktionsebene gewählt wird.

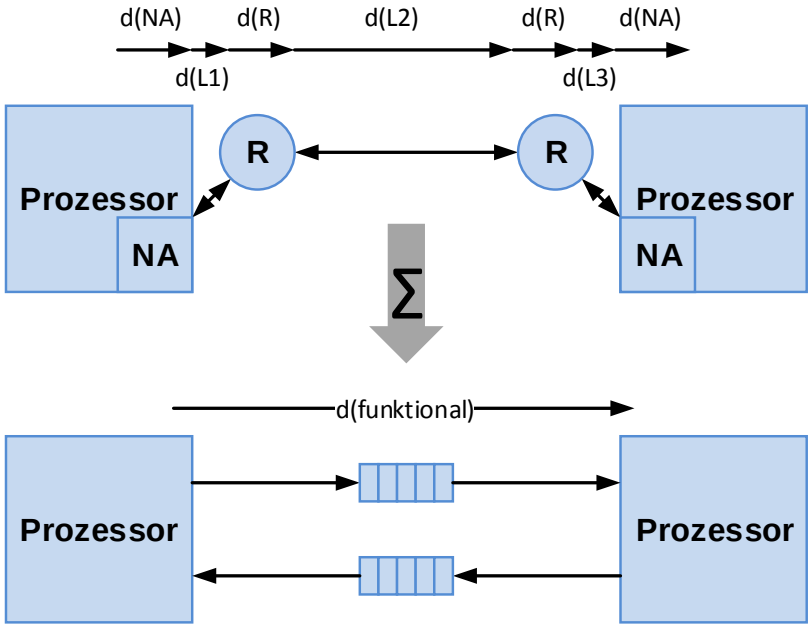


Abbildung 8.17: Darstellung einer einfachen Summen-Aggregatfunktion beim Wechsel von Abstraktionsebenen in einer Netzwerksimulation über die Transferzeitfunktion d(x)

Ein weiteres Beispiel ist das Speichermodul des gemeinsamen geteilten Systemspeichers, welches eine Implementierung besitzt in der ein Level-2 Cache Module implementiert werden kann. Während das high-level Speichermodul nur eine durchschnittliche Speicherzugriffszeit ermittelt, ermöglicht ein Cache-Modul die Ermittlung von Hit- und Miss Raten, Cache Effizienzanalysen sowie individuell berechenbaren zyklenapproximativen Speicherzugriffszeiten.

Um die Kommunikation zwischen den Prozessoren zu ermöglichen, wurde ein Netzwerkmodul implementiert, welches auf hoher Ebene nur eine funktionale First In First Out (FIFO) basierte Kommunikation zwischen je zwei Prozessorkernen ermöglicht. Die detailliertere Variante des Netzwerkmoduls verwendet eine zyklenakkurate SystemC Simulation des iNoC [Hei15] das eine 2D-Mesh Topologie implementiert. Dieses Simulationsmodul nutzt eine RTL-Simulation zur Berechnung der Komponenten, als auch die dafür notwendigen Signal-Channels in SystemC. Dadurch bietet dieses Simulationsmodul eine deutlich höhere Genauigkeit, benötigt dafür aber auch deutlich mehr Rechenzeit, wodurch die Simulationsgeschwindigkeit sinkt.

Als Schnittstelle zwischen dem Prozessorsimulationsmodul und dem Netzwerkmodul wurde ein Netzwerkadapter entworfen, welcher zwei grundlegende Aufgaben übernimmt. Einerseits wird die Funktionalität eines Netzwerkadapters realisiert, der eine korrekte Anbindung und Kommunikation zwischen Prozessoren über das iNoC ermöglicht und andererseits implementiert der Netzwerkadapter die notwendigen Synchronisationsmethoden um zyklenapproximative Simulationsmodule und zyklenakkurate Simulationsmodule miteinander zu verbinden.

Durch die derart verfügbaren Simulationsmodule lassen sich zu Evaluationszwecken vier unterschiedliche Abstraktionsebenen mit Hilfe der vertikalen Implementierung darstellen, welche in Tabelle 8.8 aufgelistet sind.

A – High Accuracy verwendet eine detaillierte aber zyklenapproximative Prozessorsimulation inklusive Cache Hierarchie mit einem sehr kleinen internen Synchronisationsintervall (vgl. Absatz 8.2) kombiniert mit einem zyklenakkuraten Netzwerkmodul, bestehend aus Netzwerkadaptern und Routerimplementierung.

B – Mid-High Accuracy verwendet eine weniger detaillierte Prozessorsimulation bei größerem internen Synchronisationsintervall kombiniert mit einem zyklen-akkuraten Netzwerkmodul, bestehend aus Netzwerkadaptern und Routerimplementierung.

Tabelle 8.8: Übersicht über die Genauigkeiten unterschiedlicher Abstraktionsebenen

Ebene	**Prozessormodell Genauigkeit**	**Netzwerkmodell Genauigkeit**	**Allgemeine Genauigkeit**
A – High	detailliert, Sync = 1 +	RTL ++	 +++
B – Mid-High	standard, Sync = 100 o	RTL ++	 ++
C – Mid-Low	detailliert, Sync = 1 +	FIFO o	 +
D – Low	standard, Sync = 100 o	FIFO o	 o

C – Mid-Low Accuracy verwendet eine detaillierte aber zyklen-approximative Prozessorsimulation inklusive Cache Hierarchie mit einem sehr kleinen internen Synchronisationsintervall kombiniert mit einem funktionalen FIFO basierten Netzwerkmodul.

D – Low Accuracy verwendet eine weniger detaillierte Prozessorsimulation bei größerem internen Synchronisationsintervall kombiniert mit einem funktionalen FIFO basierten Netzwerkmodul.

8.4.3 Simulationsgeschwindigkeit

Um die Update-Schleife zu evaluieren wurde wieder eine Matrixmultiplikation verwendet. Im Vergleich zur Analyse der hierarchischen Beschreibung, siehe Abschnitt 8.2, wurde nun keine bereits parallelisierte Implementierung verwendet, sondern das Parallelisierungswerkzeug aus dem ALMA Projekt dazu verwendet, eine in Scilab geschriebene sequentielle Matrixmultiplikation automatisiert zu parallelisieren.

Schon durch die Verwendung einer Matrixmultiplikation wird die Notwendigkeit genauer aber auch schneller Simulationen deutlich.

Table 8.9 zeigt nun die benötigte Zeit um das System und die Anwendung auf den jeweiligen Abstraktionsebenen *High-Accuracy*, *Mid-High Accuracy*, *Mid-Low Accuracy* und *Low-Accuracy* zu simulieren. Verwendet wurde dafür ein Intel Core i5 Prozessor mit einer Frequenz von 2.6 GHz. Wie zu erwarten war, benötigt das genaueste Simulationsmodell bestehend aus dem detaillierten Prozessormodul und dem zyklenakkuraten iNoC Modul die meiste Zeit, gefolgt von den anderen Abstraktionsebenen in der Reihenfolge ihrer Definition.

Tabelle 8.9: Simulationszeiten für eine Simulation einer 64x64 Matrixmultiplikation mit 4 Verarbeitungseinheiten

Abstraktionsebene	**Simulationszeit in Sekunden**
A - High	360.059
B - Mid-High	323.214
C - Mid-Low	25.348
D - Low	17.474

Um nun zu ermitteln, welcher Speedup für die Ausführung eines iterativen Optimierungsansatzes möglich ist wurde hier zunächst der maximale Speedup ermittelt:

$$S = \frac{t_{sim(A)}}{t_{sim(D)}} = \frac{360.059}{17.474} = 20.6 \tag{8.5}$$

Da die Parallelisierungswerkzeuge aus dem ALMA Projekt einen iterativen Optimierungsansatz verwenden, um die Qualität der Parallelisierung schrittweise zu verbessern werden mehrere Simulationsdurchläufe benötigt um das jeweilige Ergebnis zu evaluieren. Die Gesamtsimulationszeit ergibt sich damit aus dem Produkt der Simulationszeit mit der Anzahl der Iterationsschritte:

$$t_{total} = n * t_{sim} \tag{8.6}$$

Dabei können anhand der Abstraktionsebenen sowohl eine maximale Simulationszeit, als auch eine minimale Simulationszeit abgeleitet werden:

$$t_{max} = n * t_{sim(A)} \;\; und \;\; t_{min} = n * t_{sim(D)} \tag{8.7}$$

Der maximalen Speedup für den iterativen Ansatz lässt sich dadurch wie folgt definieren und ist gleich dem maximalen Speedup bei einer einzelnen Simulation:

$$S_{max} = \frac{t_{max}}{t_{min}} = \frac{n * t_{sim(A)}}{n * t_{sim(D)}} = \frac{t_{sim(A)}}{t_{sim(D)}} = S \tag{8.8}$$

Soll der maximale Speedup erzielt werden, so müssen in jedem Iterationsschritt High-Level Simulationsmodule verwendet werden. Das Ergebnis ist allerdings eine inakkurate Performance Evaluation, welche sich negativ auf die Parallelisierungsqualität auswirkt. Um anwendungsspezifische Performanceparameter und Kommunikationszeiten zu ermitteln, sollte mindestens eine vollständig detaillierte Simulation mit hoher Genauigkeit ausgeführt werden. Typischerweise wird eine solche Simulation dem ersten Iterationsschritt zugeordnet, wenn keine gravierenden Änderungen an der Parallelisierten Anwendung zu erwarten sind. Andernfalls müsste eine erneute detaillierte Simulation durchgeführt werden. Die so als typisch definierte iterative Simulation benötigt nun folgende Laufzeit:

$$t_{typ} = t_{max} + n - 1 * t_{min} \tag{8.9}$$

In diesem Evaluationsbeispiel kann der typische Speedup über die Ausführung einer Simulation auf Ebene A und den restlichen Simulationen auf Ebene D derart definiert werden:

$$S_{typ} = \frac{t_{max}}{t_{typ}} = \frac{n * t_{sim(A)}}{t_{sim(A)} + (n-1) * t_{sim(D)}} \tag{8.10}$$

Mit wachsender Anzahl an Iterationen innerhalb des Parallelisierungswerkzeug nähert sich der Speedup dem Wert $S = 20.6$, welcher durch die einzelnen Abstraktionsebenen bedingt ist. Hierbei wird aber auch ersichtlich, dass die Beschleunigung des Gesamtprozesses mit jeder weiteren Iteration langsamer wächst.

8.4.4 Auswirkungen auf die Parallelisierungsqualität

Als Ausgangsbasis für die Evaluation der erreichbaren Qualität einer automatisierten Parallelisierung wird eine einfache Matrixmultiplikation verwendet. Allerdings nicht in C Code sondern es wird hier eine sequentielle Implementierung in SciLab [Sci13] den ALMA Werkzeugen zur Verfügung gestellt. Das System auf das die Anwendung parallelisiert wird ist ein KAHRISMA Multicoreprozessor mit vier Prozessorkernen und der oben beschriebenen Architektur.

Zur Evaluation werden zwei unterschiedliche Parallelisierungsansätze verwendet um die unterschiedlichen Anforderungen an die Kommunikationssimulation aufzuzeigen.

Für jede Parallelisierung werden vier Iterationsschritte verwendet, wobei drei unterschiedliche Simulationskonfigurationen zur Verfügung stehen.

(1) high: *High Accuracy* Simulation für alle Iterationen, entspricht Modell A (siehe Tabelle 8.8

(2) low: *Low Accuracy* Simulation für alle Iterationen, entspricht Modell D

(3) hybrid: Die erste Iteration nutzt eine *High Accuracy* Simulation, nach A, während alle weiteren Iterationen nach Simulationsmodell D durchgeführt werden.

Statuscode	Name	Farbe
0	unused	schwarz
1	idle	weiß
2	running	grün
3	waiting send	türkis
4	sending	blau
5	waiting recieve	gelb
6	recieving	rot

Tabelle 8.10: Durch Tracing ermittelte Prozessorzustände während der Simulation

Die Ergebnisse sind als Status Tracing dargestellt, und wurden mit dem SystemProfiler erstellt und mit dem Analysewerkzeug Paraver [PLCG95, Bar14] visualisiert. Tabelle 8.10 stellt die Prozessorzustände aus Tabelle 7.3 und deren farbliche Zuordnung dar.

Der erste verwendete Parallelisierungsansatz ist ein sogenannter scatter-gather Ansatz, bei dem eine Anwendung in unabhängige möglichst gleich große Teile aufgesplittet und verteilt (scatter) wird, welche dann von jeweils einem Prozessorkern ausgeführt werden. Sind aller Teilergebnisse berechnet, werden diese auf einem dedizierten Kern zusammengeführt (gather) und das Endergebnis ermittelt.

Die Ergebnisse aus 8.18 zeigen deutlich, dass eine genauere Simulation der Kommunikationsinfrastruktur bei vier durchgeführten Iterationen nur marginale Auswirkungen auf die Qualität der Parallelisierung hat. Der Vergleich der Ausführungszeiten zeigt zwar kleine Unterschiede, diese sind aber nicht ausschlaggebend, wenn keine Echtzeitgrenzen gefordert werden.

Der zweite Parallelisierungsansatz verfolgt eine Pipeline Verarbeitung. Dabei wird eine Anwendung in deutlich kleinere voneinander abhängige Prozesse aufgeteilt, welche jeweils von einem zum nächsten Prozessor weitergegeben werden. Dieser Ansatz eignet sich besonders für Streaming-Anwendungen, bei denen ein Prozessor die Eingangsdaten verarbeitet und ein anderer dedizierte

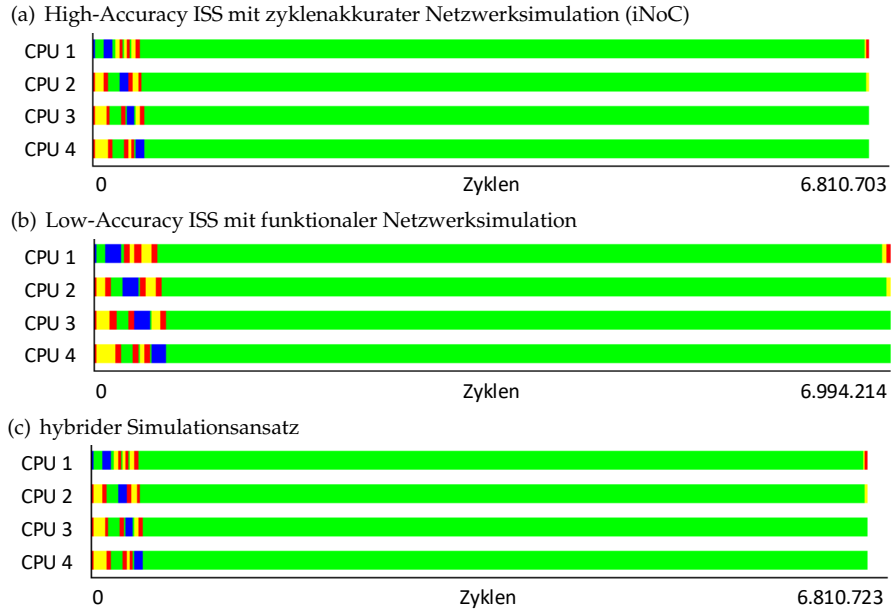


Abbildung 8.18: Qualität der erzielten Parallelisierung bei Verwendung eines Scatter-Gather Ansatzes

Prozessor das Endergebnis erzeugt, oder aber für heterogene Architekturen bei denen die einzelnen Prozessoren Spezialfunktionen bereitstellen.

Für die Parallelisierung bedeutet dies, dass eine deutlich genauere Kommunikationsanalyse durchgeführt werden muss, da das Zusammenspiel der einzelnen Prozessoren stark von der Kommunikation untereinander abhängt. Wird nun die Matrixmultiplikation nach dieser Technik parallelisiert, zeigt sich in Abbildung 8.19, dass hierbei die Genauigkeit der Simulation ausschlaggebend sein kann, um ein gutes Endergebnis zu erhalten.

Die erste Abbildung 8.19(a) zeigt dabei zum Vergleich das Ergebnis der initialen Parallelisierung, welche als Ausgangsbasis für den Iterativen Prozess aus dem sequentiellen Code generiert wurde.

Abbildung 8.19: Qualität der erzielten Parallelisierung bei Verwendung eines Pipeline Ansatzes

Anschließend werden wieder die drei unterschiedlichen Simulationsmodelle verwendet um den iterativen Parallelisierungsprozess durchzuführen. Klar ersichtlich ist, dass der akkurate Ansatz 8.19(b) gute Ergebnisse liefert, während der funktionale Simulationsansatz 8.19(c) eine gute Pipeline Ausführung nur erahnen lässt. Mit einem hybriden Ansatz 8.19(d) können nun sehr gute Parallelisierungsergebnisse erzielt werden, dessen Ungenauigkeit nicht mehr Maßgeblich für die allgemeine Parallelisierung ist.

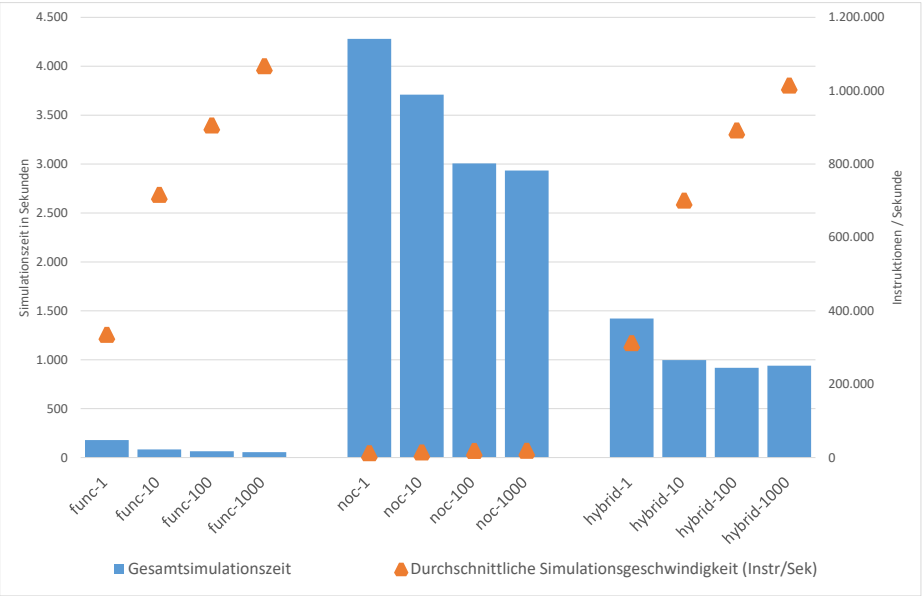


Abbildung 8.20: Darstellung der Gesamtsimulationszeit und der durchschnittlichen Simulationsgeschwindigkeit über den Parallelisierungsprozess

Durch den hybriden Simulationsansatz kann dadurch gezeigt werden, dass Simulationszeiten signifikant verringert werden können. Dies wird in Abbildung 8.20 deutlich, in der die Simulationszeiten mit unterschiedlichen Synchronisationsintervallen und die daraus resultierende durchschnittliche Simulationsgeschwindigkeit während des Prallelisierungsprozesses dargestellt werden. Mit zunehmenden Synchronisationsintervall kann die Simulationszeit nochmals verringert werden.

Die Qualität der erreichten Parallelisierungsergebnisse mit dem hybriden Simulationsansatz zeigt, dass zwar Genauigkeitsverluste vorhanden sind, diese aber keinen relevanten Einfluss auf die allgemeine Qualität des Ergebnisses haben.

Kapitel 9

Zusammenfassung

Ausgehend von der wachsenden Komplexität eingebetteter und zunehmend heterogener Systeme bestehend aus mehreren Prozessoren, Beschleunigern und Spezialkomponenten für die optimale Erfüllung einer gegebenen Aufgabe, sind neue Methoden zur Programmierung dieser Systeme notwendig. Das Forschungsprojekt ALMA hat dabei mit einer automatisierten Parallelisierung einen wichtigen Schritt in diese Richtung vollführt. Um die Komplexität eines hoch parallelen und zugleich eingebetteten Systems handhaben zu können wurden in dieser Arbeit eine Architekturbeschreibungssprache lexikalisch und syntaktisch entworfen und eine entsprechende Werkzeugunterstützung umgesetzt.

Daraus ist ein Framework zur Beschreibung, Analyse und Simulation von eingebetteten Multicore Systemen entstanden, welches durch seine Flexibilität den Anforderungen an eine Verwendung in Parallelisierungswerkzeugen gewachsen ist.

Für die Beschreibung des Systems wurden daher die zwei höchst unterschiedlichen Sichtweisen des Anwendungsentwicklers und der zugrundeliegenden Hardwarearchitektur vereint um eine ideale Beschreibungsgrundlage von sowohl strukturellen Informationen des Systems als auch Verhaltensinformationen abbilden zu können.

Die Verknüpfung dieser beiden Sichtweisen erlaubt nun eine einfache, flexible und effiziente Beschreibung komplexer Multicore Systeme und deren Einbindung in Parallelisierungswerkzeuge um eben diese Komplexität des Systems dem Anwendungsentwickler bei der Parallelisierung auf verständliche Weise bereitzustellen.

Die Architekturbeschreibungssprache bedient sich dafür sowohl bekannten Konzepten aus Hardwarebeschreibungssprachen, wie der Verwendung von Modulen, Ports und Interfaces zur strukturellen Beschreibung, aber auch semantischer Methoden, welche üblicherweise in Compilern eingesetzt werden um Verhalten und Instruktionen zu beschreiben.

Des Weiteren wurde die Sprache mit einer Hierarchischen Beschreibung, genannt Implementierung, versehen, die es erlaubt sowohl hierarchische Aspekte, genannt horizontale Hierarchie, als auch unterschiedliche Abstraktionsebenen, genannt vertikale Hierarchie, eines Systems oder einer Systemkomponente zu beschreiben.

Dazu wurde ein Compiler entwickelt, welcher die entworfene Architekturbeschreibungssprache übersetzt, das beschriebene System analysieren kann, und die Informationen an eine flexible Simulationsumgebung weitergibt. Während dieses Übersetzungsvorgangs, werden die beiden Hierarchiekonzepte vom ADL-Compiler aufgelöst und als flache Systembeschreibung einer Simulationsumgebung übergeben. Diese erlaubt die Simulation und das Profiling des Systems gemeinsam mit der parallelen Anwendung und kann dazu verwendet werden sowohl Werkzeugen als auch dem Anwendungsentwickler das notwendige Feedback zu geben um die Parallelisierung zu optimieren.

Die Evaluation hinsichtlich der Hierarchieebene und der Simulationsparameter des Frameworks zeigt die Vielseitigkeit und die Flexibilität welche durch die Verwendung der ADL in Kombination mit dem Systemsimulator erreicht werden kann. Sie zeigen besonders die Fähigkeit des Frameworks sich adaptiv auf die jeweiligen Anforderungen hinsichtlich Simulationsgeschwindigkeit und Simulationsgenauigkeit hin anzupassen um sowohl für grobgranulare Optimierungen als auch für feingranulare Parallelisierungsentscheidungen eingesetzt werden zu können.

Hervorzuheben ist auch die Fähigkeit des Frameworks das gesamte Spektrum an Arbeitspunkten hinsichtlich Detailgrad, Abstraktionsebene, Genauigkeit und Simulationsgeschwindigkeit in einer gemeinsamen Werkzeugübergreifenden ADL zu kapseln. Diese Fähigkeit wird eingesetzt um die Auswirkungen unterschiedlicher Abstraktionsebenen und Genauigkeitsstufen auf den Parallelisierungsprozess zu evaluieren. Es wurde eine Feedbackschleife zwischen mehreren Simulationsdurchläufen implementiert mit der hochdetaillierte Informationen auf höhere Abstraktionsebenen übertragen und damit die

Gesamtsimulationszeit deutlich reduziert werden kann. Dieser hybride Simulationsansatz erlaubt es nun die Genauigkeit über den gesamten Parallelisierungsprozess hoch und die Simulationszeit gering zu halten, was zu einer im allgemeinen sehr guten Parallelisierungsqualität führt, welche mit den Ergebnissen einer dauerhaft detaillierten und akkuraten Simulation vergleichbar sind.

Anhang A

Spezifikation Datenbeschreibungssprache

Dieser Anhang beschreibt die Spezifikation der Datenbeschreibungssprache und ihrer Elemente. Es handelt sich hier um einen Ausschnitt der Spezifikation um die Grundlegenden Funktionen zu erläutern und um das Verständnis für den Aufbau der ADL zu vervollständigen.

Die Datenbeschreibungssprache wurde von Timo Stripf [Str13] während seiner Forschungsarbeit für die Kahrisma-ADL entwickelt. Durch die enge Verknüpfung zum ALMA Projekt wurde diese als Basis für die Architekturbeschreibungssprache verwendet. Die hier angegebene Spezifikation dient daher nur dem Verständnis einiger Begrifflichkeiten die in dieser Arbeit verwendet worden sind.

A.1 Datentypen

Die Datenbeschreibungssprache unterscheidet skalare und assoziative Datentypen. Skalare Datentypen stellen dabei Werte dar und können einen der folgenden Typen aufweisen:

undef ein existierender, aber nicht weiter spezifizierter Wert

bool ein boolescher Datentyp, kann **wahr** oder **falsch** annehmen

string Zeichenkettendatentyp

integer Ganzzahldatentyp

float Gleitkommatyp mit einfacher Genauigkeit

Die assoziativen Datentypen, auch Container genannt, stellen Knoten des Datenbaumes dar und besitzen Kind-Elemente, entweder ein skalares Kind-Element oder eine unbegrenzte Anzahl an assoziativen Kind-Elementen. Jeder Container wird durch einen Schlüssel identifiziert, der entweder vom Typ **string** oder vom Typ **integer** sein muss. Handelt es sich beim Schlüssel um eine **string**, so ist der Container ein Objekt-Container. Bei **integer** Schlüsseln handelt es sich um Vektor-Container. Die beiden Container unterscheiden sich in den Zugriffsmöglichkeiten durch Element-Zugriff oder Index-Zugriff, sowie durch die interne Datenhaltung.

A.2 Lexikalische Elemente

Die Datenbeschreibungssprache bedient sich ähnlichen lexikalischen Eigenschaften, wie sie auch aus Programmiersprachen bekannt sind.

Kommentare Die Sprache unterstützt Zeilenweise Kommentare sowie mehrzeilige Kommentare die mit "//" eingeleitet bzw. mit "/*" begonnen und mit "*/" abgeschlossen werden können.

Bezeichner Werden für konstante Variablen oder Funktionen verwendet. Sie müssen mit einem Buchstaben oder dem Zeichen "_" beginnen.

Literale sind die kleinstmöglichen Terme bestehend aus einer Zahl oder einem einzelnen Zeichen.

Trennzeichen sind Klammern, Komma und Semikolon.

Füllzeichen Leerzeichen und Zeilenumbruch trennen Bezeichner voneinander ab, werden aber darüber hinaus, ähnlich zu Kommentaren, im Kompilierprozess nicht weiter betrachtet.

Operatoren Die Sprache unterstützt die üblichen Operatoren für logische, arithmetische und Vergleichsfunktionen, welche an C++ angelehnt sind. Komma, Semikolon und eckige Klammern werden als spezielle Element-Zugriffsoperatoren verwendet.

A.3 Syntax

Um die Darstellung von Zahlen und Bezeichnern zu differenzieren müssen Zahlen immer mit einem Dezimalzeichen beginnen, während Bezeichner immer mit einem Buchstaben oder dem Sonderzeichen "_" beginnen müssen. Die Zahlen können binär oder hexadezimal mit Präfix (**0b** oder **0h**) sowie dezimal

als Ganzzahl oder Gleitkommazahl angegeben werden und jeweils mit einem Vorzeichen versehen werden.

Zeichenketten und Bezeichner werden hingegen grundsätzlich mit einfachen Hochkommata ohne Unterstützung für Sonderzeichen oder doppelten Hochkommata mit Escape-Zeichen für Sonderzeichen definiert. Eine weitere Möglichkeit stellt der Quote-Operator *q{...}* dar, der es erlaubt formatierten Text als Zeichenkette auch über mehrere Zeilen hinweg abzulegen.

A.4 Konstanten

Die Datenbeschreibungssprache kennt drei definierte Konstanten. Dies sind die Werte **true** und **false** für die Wertzuweisung von booleschen Datentypen mit wahr bzw. falsch, sowie der Wert **undef** für die Zuweisung eines undefinierten Wertes.

A.5 Operatoren

Als Operatoren stehen die aus Programmiersprachen bekannten logischen und arithmetischen Operatoren sowie Vergleichsoperatoren zur Verfügung. Eine Besonderheit stellen die Elementzugriffsoperatoren dar, welche in der Übersicht in Tabelle A.1 beschrieben sind.

A.6 Spezialoperatoren

Neben den üblichen Operatoren unterstützt die Sprache weitere Operatoren für den Elementzugriff. Dazu gehört der Elementzugriff "[ ]", welcher auf ein

Operator	Beschreibung	Priorität	Richtung
()	Gruppierung		—
[]	Elementzugriff		—
{}	Block Operator		—
	leerer Operator	1	von Links
!	logisches NICHT	2	von Rechts
~	Bitweises Komplement (bitweises NICHT)	2	von Rechts
+	Unäres Plus	2	von Rechts
-	Unäres Minus	2	von Rechts
*	Multiplikation	3	von Links
/	Division	3	von Links
%	Modulo	3	von Links
+	Addition	4	von Links
-	Subtraktion	4	von Links
«	Bitweises Linksschieben	5	von Links
»	Bitweises Rechtsschieben	5	von Links
<	Kleiner als	6	von Links
<=	Kleiner oder Gleich	6	von Links
>	Größer als	6	von Links
>=	Größer oder Gleich	6	von Links
==	Gleichheit	7	von Links
!=	Ungleichheit	7	von Links
&	Bitweises UND	8	von Links
^	Bitweises exklusives ODER (XOR)	9	von Links
|	Bitweises ODER	10	von Links
&&	Logisches UND	11	von Links
||	Logisches ODER	12	von Links
,	Vektor Operator	13	von Links
=	Zuweisung	14	von Rechts
+=	Addition mit Zuweisung	14	von Rechts
-=	Subtraktion mit Zuweisung	14	von Rechts
*=	Multiplikation mit Zuweisung	14	von Rechts
/=	Division mit Zuweisung	14	von Rechts
%=	Modulo-Division mit Zuweisung	14	von Rechts
&=	Bitweises UND mit Zuweisung	14	von Rechts
^=	Bitweises exklusives ODER mit Zuweisung	14	von Rechts
|=	Bitweises ODER mit Zuweisung	14	von Rechts
«=	Linksschieben mit Zuweisung	14	von Rechts
»=	Rechtsschieben mit Zuweisung	14	von Rechts
;	Befehls Operator	15	von Links

Tabelle A.1: Operatoren

existierendes Element zugreifen und dieses Überschreiben können, oder falls es nicht vorhanden ist ein neues Element erstellen. Der Zugriff erfolgt entsprechend dem im Operator angegebenen Datentyp: Strings erzeugen Objektelemente während Ganzzahlen Vektorindexzugriffe durchführen. Wird ein Vektor angegeben, so werden mehrere Zugriffe kaskadiert.

A.7 Variablen

Im Gegensatz zu anderen Markup-Sprachen wie XML oder JSON unterstützt die Datenbeschreibungssprache die direkte Verwendung von Variablen. Variablen werden über das Schlüsselzeichen "$" gefolgt von einem Bezeichner definiert. Die Sprache unterstützt allerdings keine Zugriffsberechtigungen, so dass Variablen immer gelesen oder Geschrieben werden können, der Gültigkeitsbereich einer Variable ist aber auf den aktuellen Block beschränkt. In Unterblöcken werden Kopien verwendet, welche keine Änderung der zuvor definierten Variable erzeugen.

Für jede Datei der Datenbeschreibungssprache existiert **$VARIABLES** welches als Objekt alle Variablen dieser Datei beinhaltet und bei Einbindung weiterer Dateien an diese übergeben werden kann.

A.8 Kontrollflussstrukturen

Die Datenbeschreibungssprache unterstützt die Kontrollflussstrukturen *If – elif – else* für bedingte Ausführungen, *for* für gezählte Schleifenausführung und *foreach* für Operationen die auf allen Elementen eines Objekts oder Vektors ausgeführt werden sollen.

A.9 Funktionen

Für die Erstellung einiger Datenstrukturen gibt es neben den Operatoren auch Funktionen die diese Aufgabe übernehmen können. Andere Funktionen unterstützen die Verwendung der Sprache. Hier werden die für die Architekturbeschreibung notwendigen Funktionen beschrieben.

A.9.1 vector(...) or v(...)

Diese Funktion erzeugt einen neuen Vektor und muss verwendet werden, wenn leere Vektoren oder Vektoren mit nur einem Element erzeugt werden sollen.

A.9.2 replacevars(string) or r(string)

Die Ersetzen-Funktion wird benötigt um Elemente aus Variablen heraus zu erzeugen. Soll ein Variablenwert mit in den Elementschlüssel integriert werden so kann innerhalb dieser Funktion der Bezeichner der Variable durch den aktuellen Wert ersetzt werden.

A.9.3 isset($var)

Die Funktion *isset()* überprüft ob eine Variable innerhalb dieser Datei, oder einem Block, schon definiert worden ist und gibt wahr oder falsch zurück. Dies wird dazu verwendet um konsistentes Einbinden von Dateien zu ermöglichen.

A.10 Backus-Naur Form

Quellcode A.1 stellt nun die vollständige Backus-Naur Definition [Int96] der Datenbeschreibungssprache dar. Mit dieser Definition wird die Syntax der Sprache vollständig definiert.

```
1  Block      ::= Statements|RValue|Empty;
2  Statements ::= {Statement,';'}-;
3  Statement  ::= Assignment|Selection|Iteration;
4  Assignment ::= {LValue,AssignOp}-,RValue;
5  Selection  ::= 'if','(',RValue,')',RValue,{'elif','(',RValue,')',
6                 RValue},['else',RValue];
7  Iteration  ::= 'for','(',RValue,';',RValue,';',RValue,')',RValue;
8  LValue     ::= Access|Variable,[Access];
9  Access     ::= {'['RValue']'}-;
10 RValue     ::= Scalar|LValue|Function|'{',Block,'}'|Vector|Op;
11 Scalar     ::= String|Integer|Float|'undef';
12 Op         ::= RValue,BinaryOp,RValue|UnaryOp,RValue|'(',RValue,')';
13 Vector     ::= RValue{','RValue}-;
14 Variable   ::= '$',Identifier;
15 Function   ::= Identifier,'(',[RValue],')'
16 AssignOp   ::= '='|'+='|'-='|'*='|'/='|'%='|
17                '<<='|'>>='|'|='|'&='|'^=';
18 BinaryOp   ::= '+'|'-'|'*'|'/'|'%'|'&'|'|'|'^'|'&&'|'||'|
19                '<<'|'>>'|'>'|'>='|'<'|'<='|'!='|'==';
20 UnaryOp    ::= '+'|'-'|'!'|'~';
21 Undef      ::= 'undef';
22 Boolean    ::= 'true'|'false';
23 Integer    ::= {Digit}-|'0x',{HexDigit}-|'0b',{BinDigit}-;
24 String     ::= ?Datensprachenzeichenketten?;
25 Float      ::= {Digit}-,'.',{Digit}-;
26 Identifier ::= Letter,{'_'|Digit|Letter};
27 BinDigit   ::= '0'|'1';
28 Digit      ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9';
29 HexDigit   ::= Digit|'a'|'b'|'c'|'d'|'e'|'f'|'A'|'B'|'C'|'D'|'E'|'F';
30 Letter     ::= 'a'|'b'|'c'|'d'|'e'|'f'|'g'|'h'|'i'|'j'|'k'|
31                'l'|'m'|'n'|'o'|'p'|'q'|'r'|'s'|'t'|'u'|'v'|
32                'w'|'x'|'y'|'z'|'A'|'B'|'C'|'D'|'E'|'F'|'G'|
33                'H'|'I'|'J'|'K'|'L'|'M'|'N'|'O'|'P'|'Q'|'R'|
34                'S'|'T'|'U'|'V'|'W'|'X'|'Y'|'Z';
```

Quellcode A.1: ADL Sprachdefinition in Backus-Naur-Form

Anhang B

Spezifikation Architekturbeschreibungssprache

Dieser Anhang beschreibt die Spezifikation der Architekturbeschreibungssprache und ihrer Elemente. Dabei beschreibt dieses Kapitel den Aufbau sowie die Struktur der einzelnen Abschnitte auf technischer Ebene sehr detailliert.

B.1 Schlüsselwörter

Schlüsselwörter der ADL sind die in der Datenbeschreibungssprache beschriebenen Funktionen aber auch die Abschnittsbezeichnungen, wie `Module`, und die Verhaltensbeschreibungen. Des Weiteren werden die Zeichen "." und ":" verwendet um die Instanznamen im Kompilierprozess systemweit eindeutig zu gestalten. Der Punkt trennt dabei die Hierarchieebenen auf, wodurch der Instanzname mit den Elterninstanzen verknüpft wird. Der ":" wird verwendet um die Portnamen eines Modules, bzw. einer Instanz, mit dem Instanznamen zu verknüpfen und um festzulegen, dass es sich bei diesem Namen um einen Port handelt.

Schlüsselwörter dürfen nicht als Eigennamen oder Bezeichner verwendet werden.

B.2 Global

Im `Global` Abschnitt werden allgemeine und systemweite Parameter beschrieben. Derzeit verfügbar sind die Basisfrequenz `Frequency` eines Systems sowie der Parameter `BootConfiguration`, der die initiale Konfiguration eines Systems definiert.

```
'Global' Object
  ├─'BootConfiguration' String (opt)
  └─Frequency Integer (Hz)
```

Abbildung B.1: Struktur der Global-Section

Der Aufbau des `Global` Abschnitts ist in Abbildung B.1 dargestellt, während Quellcode B.1 ein Beispiel mit einer Basisfrequenz von 500Mhz und der Startkonfiguration mit dem Namen RSIW_1 aus dem Konfigurationsbereich der KAHRISMA-Prozessorarchitektur darstellt.

```
['Global'] = {
  ['BootConfiguration'] = 'RSIW_1';
  ['Frequency'] = 500*1000*1000;
};
```

Quellcode B.1: ADL Beispielcode für die Global-Section einer KAHRISMA Architekturbeschreibung

B.3 Module

Module werden innerhalb des `Modules` Abschnitts beschrieben und über einen Namen eindeutig identifiziert. Ein Modul benötigt dabei eine `Port` Beschreibung mit Interfaces, sowie die Beschreibung der Simulationseigenschaften. Optionale Angaben sind die Verhalten und eine hierarchische Implementierung.

B.3.1 Ports

Ein Port beschreibt einen Anknüpfungspunkt an ein Modul. Jedem Port wird ein Interface aus der Architekturbeschreibung und ein Porttype dieses Interface zugewiesen. Dadurch wird der Typ des Ports über das Interface festgelegt. Als Schlüssel wird der Portname angegeben und als Wert das Interface mit Porttype, getrennt durch "." wie in Quellcode B.3 zu sehen ist.

B.3.2 Simulation

Dieser Abschnitt dient der Beschreibung von notwendigen und zusätzlichen Simulationsparametern, die im Multicore Simulator, siehe Kapitel 6, verwendet und an die Simulationsklassen weitergegeben werden.

Die zwei zwingend notwendigen Angaben sind die Referenz auf die SystemC Simulationsklasse und die Angabe über den Zustand des Moduls als Aktive

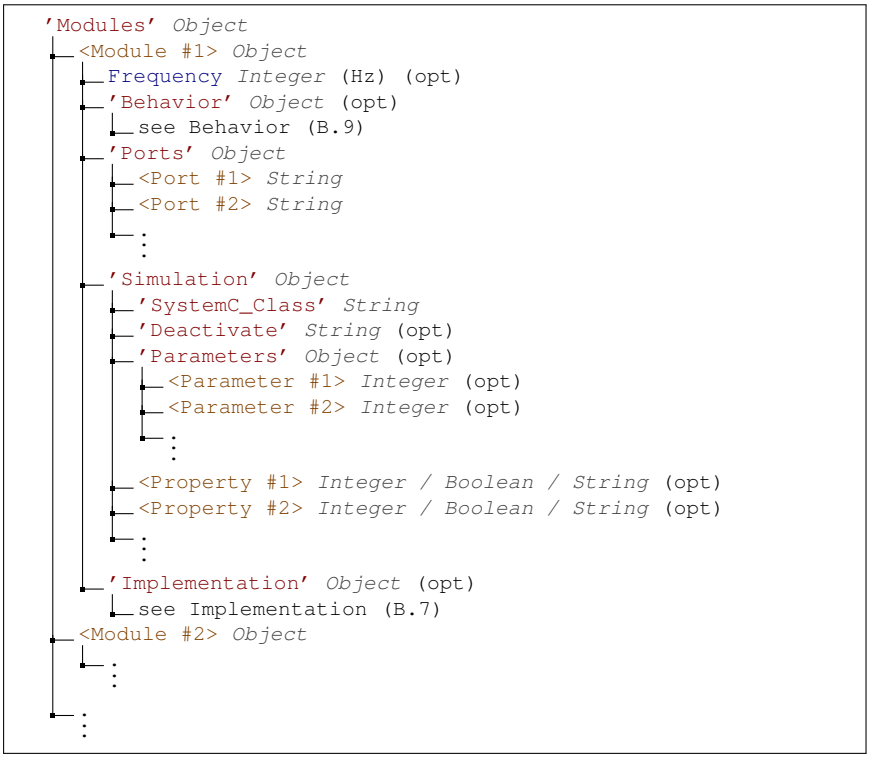


Abbildung B.2: ADL-Struktur der Modul Beschreibung

```
['Modules'] = {
  ['Core_Module'] = {
    ['Ports'] = {
      /* Portname -> Interfacename */
      ['MemInOut'] = 'GlobalMemoryConnect.Master';
      ['NetInOut'] = 'Noc_Link_RouterCore';
      ['RTSOut'] = 'RTS.Master';
    };

    ['Behavior'] = {
      ['Core'] = 'RSIW22';
    };

    ['Simulation'] = {
      ['SystemC_Class'] = 'CKahrismaEDPEModule';

      ['Parameters'] = {
        /* Parametername -> Value */
        ['StackSize'] = 256 * 1024;
        ['CyclesPerRun'] = 100;
        ['NetInPorts'] = 4;
        ['NetOutPorts'] = 4;
        ['NetInBufferSize'] = 2000;
      };
    };
  };
```

Quellcode B.2: ADL Code Beispiel für eine Definition eines Moduls

```
['Ports'] = {
    ['Port_Name'] = 'InterfaceName.PortType';
    ...
  };
```

Quellcode B.3: Portdefinition innerhalb von Modulen

oder Inaktiv für die Steuerung der Hierarchie Die Angabe der Simulationsklasse kann durch einen leeren String ersetzt werden, wenn es sich nur um rein analytische Beschreibungen handelt.

Zusätzlich zu den notwendigen Angaben ist die Definition von Klassenspezifischen Simulationsparametern möglich. Deren Verfügbarkeit hängt von der jeweiligen Simulationsklasse ab und erlaubt es damit Parameterwerte zu setzen, welche nur für die Simulationsklasse, nicht jedoch für eine Hardwareimplementierung, notwendig sind.

Die Struktur der Simulationsparameterbeschreibung ist in Abbildung B.3 dargestellt.

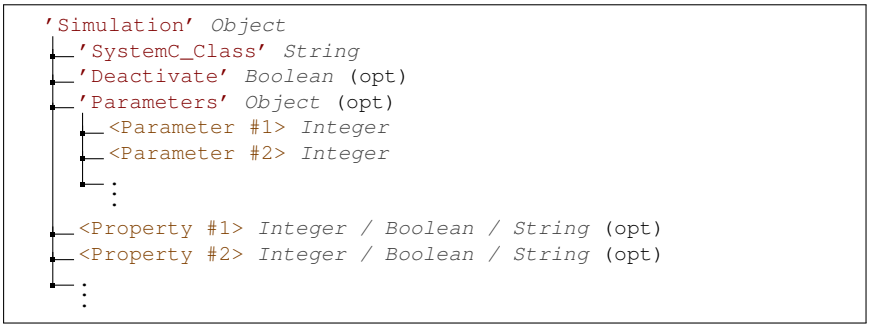


Abbildung B.3: Struktur des ADL-Blocks: Simulation

B.4 Interfaces

Jedes `Interface` besitzt einen Namen, mit dem es eindeutig gekennzeichnet wird. Darüber hinaus besitzt jedes Interfaces das Element `PortType`, welches im Abschnitt B.4.1 näher beschrieben ist, sowie eine Verhaltensbeschreibung `Behavior`, siehe auch Abschnitt B.9, in der zusätzliche High-Level Informationen gespeichert werden können. Jedem Interface wird außerdem ein

`Simulation`-Abschnitt zugeordnet in der die zugehörige Simulationsklasse angegeben wird. Wird keine Simulationsklasse angegebene wird für eine Simulation ein Standard SystemC Interface angenommen.

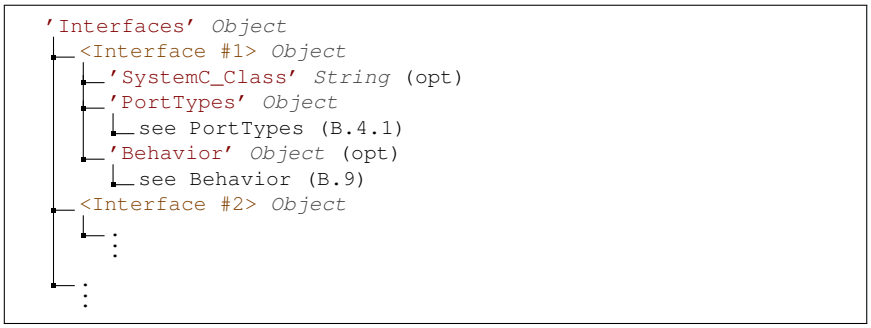


Abbildung B.4: ADL-Struktur der Interface Beschreibung

```
['Interfaces'] = {
  ['Interface_Name'] = {
    ['PortType'] = {
      ['PortName']['Multiplicity'] = (min, max);
    };
    ['Behavior'] = {
      ['Behavior_Name'] = {
        ['Property_Name'] = ...;
        ['Property_Name'] = ...;
      };
    };
  };
  ...
};
```

Quellcode B.4: ADL Code Beispiel für eine Definition eines Interfaces

B.4.1 PortTypes

Da ein `Interface` jeweils an Ports von Modulen gekoppelt wird, besitzt das Interface keinen eigenständigen Port, sondern einen sogenannten PortTyp. Dieser definiert die Rolle, die das an diesem Porttyp angeschlossene Modul in der

Multiplizität	Beschreibung
$undef$	Null bis unendliche Anzahl an Verbindungen
$(0, undef)$	entspricht undef
i	genau i Verbindungen sind gefordert, i ist Ganzzahl
(i, i)	entspricht i
(i, j)	mindestens i aber maximal j Verbindungen

Tabelle B.1: Übersicht über die möglichen Angaben der Multiplizität

Verbindung einnimmt. Dazu lässt sich für angegebene PortTypes eine Multiplizität angeben. Diese definiert die Anzahl der gültigen Verbindungen von Ports mit diesem Porttyp an das Interface. Tabelle B.1 stellt die möglichen Angaben der Multiplizität dar, während Abbildung B.5 die ADL Struktur einer PortType Definition darstellt. Jeder Porttyp wird über einen Namen innerhalb des Interface eindeutig definiert. Der Schlüssel kann auch durch "['.']" ersetzt werden. Dies entspricht einem Default PortType, der jedem Modulport zugewiesen wird, wenn bei dessen Definition nur das Interface ohne spezifischen Typ angegeben wird.

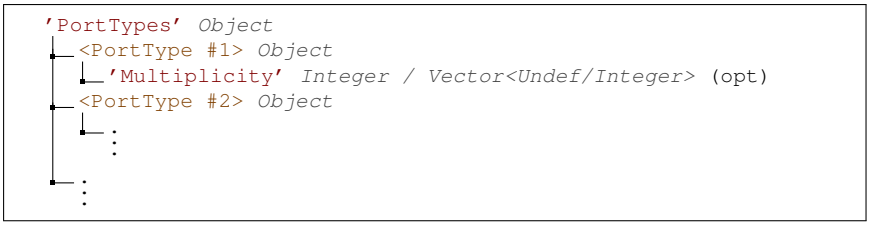


Abbildung B.5: ADL Struktur von PortTypes

B.5 Toplevel

Bei dem Toplevel der ADL handelt es sich um ein automatisch instanziiertes Modul. Dieses folgt daher syntaktisch und semantisch einer Modulbeschreibung mit geringen Abweichungen. Die Definiten von Modulports ist nicht

zwingend erforderlich. Demgegenüber müssen die Abschnitte Simulation und Implementation analog zu Modulen definiert werden.

Sollte für ein TopLevel eine Simulationsklasse vorhanden sein, so kann diese auch verwendet werden. Im Allgemeinen ist das TopLevel allerdings inaktiv und dient nur der Beschreibung der höchsten Hierarchieebene.

```
'TopLevel' Object
├─'Ports' Object (opt)
├─'Simulation' Object
│ ├─'SystemC_Class' String
│ └─'Deactivate' Boolean
└─'Implementation' Object
  ├─'Instances' Object
  │ └─see Instances (B.7.1)
  └─'Connections' Object (opt)
    └─see Connections (B.7.2)
```

Abbildung B.6: ADL Struktur von TopLevel

B.6 Configurations

Die `Configuration` Section erlaub es der ADL auch rekonfigurierbare Systeme zu modellieren. Eine Konfiguration wird definiert mit einem Namen und einer Implementierung, siehe B.7. zusätzlich kann für eine Konfiguration ein spezielles verhalten definiert werden.

B.7 Implementation

Eine Implementierung eines Moduls beschreibt eine Hierarchieebene innerhalb des Moduls und besteht aus Instanzen und Verbindungen

Instanzen können nur mit Instanzen auf der Gleichen Ebene, d.h. innerhalb derselben Implementierung oder mit ihrem Elternmodul verbunden werden.

```
['Configurations'] = {
  ['Configuration_Name1'] = {
    ['Implementation'] = {
      ['Instances'] = {
        ['Instance_Name1'] = {
          ['Module'] = 'Name_Module';
        };
        ...
      };
      ['Connection'] = {
        ['Instance_Name1'] = {
          ['Port_Name'] = v('Target_Module', 'Target_Port');
          ...
        };
        ...
      };
    };
  };
    ['Behavior'] = {
      ['Core'] = {
        ['Microarchitecture'] = 'Microarchitecture_Name';
      };
    };
  };
  ['Configuration_Name2'] = {
  ...
  };
  ...
}; // Configurations
```

Quellcode B.5: ADL Code Beispiel für eine Definition einer Configuration

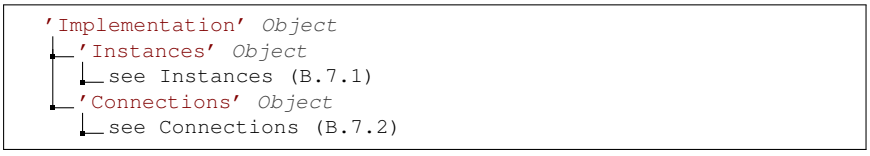


Abbildung B.7: Struktur des ADL-Blocks: Implementation

B.7.1 Instances

Jede Instanz wird, wie in Abbildung B.8 dargestellt, durch ein DDL Objekt beschrieben, dessen Schlüssel der Instanzname ist. Eine Instanz besitzt eine Referenz zum dem Modul von dem die Instanz abgeleitet wird. Instanzen können durch Angabe von Modulelementen, siehe Abschnitt B.3, individualisiert werden. Weiterhin kann zur Individualisierung über Hierarchieebenen hinweg der optionale Block `SubInstance` verwendet werden.

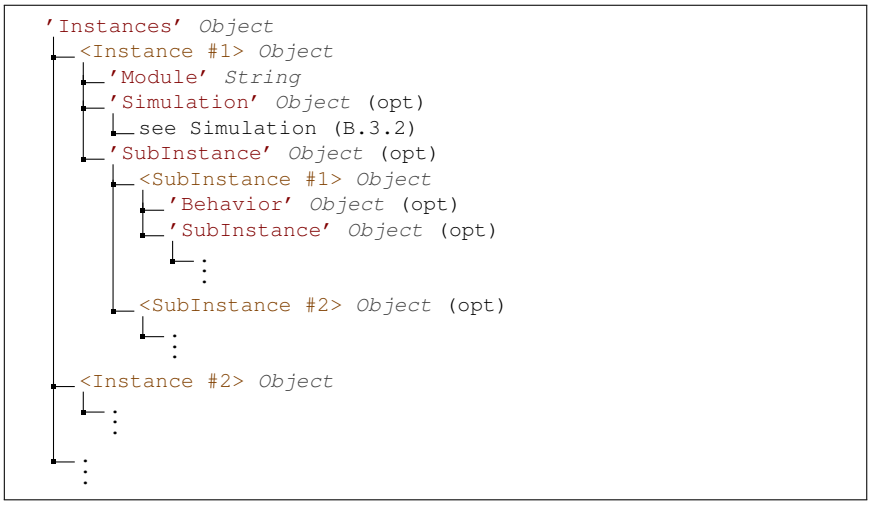


Abbildung B.8: Struktur des ADL-Blocks: Instances

Änderungen mit Hilfe von `SubInstance` wirken sich nur auf die konkret angegebenen Instanzen aus, nicht aber auf das Modul, was eine Globale Änderung aller Instanzen zur Folge hätte, oder auf andere Instanzen des Moduls. Quellcode B.6 zeigt hierbei ein Beispiel wie Verhaltensbeschreibungen in der ADL individualisiert werden können.

Eine über SubInstance initiierte Individualisierung muss dabei nicht alle Eigenschaften des Referenzmoduls beinhalten. Die Beiden Beschreibungen werden

```
1 ['SubInstance']['Inst1']['Behavior']['BehaviorName']['Property'] =
      {}
2 ['SubInstance']['Inst1']['SubInstance']['Inst2']['Behavior']['
    BehaviorName']['Property'] = {}
3 ['SubInstance']['Inst3']['Behavior']['BehaviorName']['Property'] =
      {}
```

Quellcode B.6: ADL Code Beispiel für das Überschreiben von Modulparametern mittels SubInstance

derart miteinander verknüpft, das im Modul bestehende Parameter überschrieben und nicht vorhandene Parameter ergänzt werden.

Der Befehl SubInstance kann auch dazu verwendet werden, um die aktuelle Instanz selbst zu individualisieren. Sollte eine angegebene Instanz nicht gefunden werden, so wird der Block vom ADL Compiler ignoriert, was durch das Aktivieren oder Deaktivieren von Instanzen oder Modulen vorkommen kann. Sofern die Implementierungshierarchie bekannt ist, kann SubInstance auch geschachtelt verwendet werden. dadurch können auch tiefer gelegene Instanzebenen individualisiert werden.

B.7.2 Connections

Jede Verbindung wird durch ein DDL-Objekt beschrieben, deren Schlüssel der Name der Quellinstanz ist. innerhalb dieses Blocks werden die Ports der Instanz aufgelistet und deren Wert mit einem Tupel festgelegt. Das Tupel besteht aus der Zielinstanz und dem Zielport. Die Angabe des zu verwendenden Interfaces ist nicht notwendig, da sie implizit aus der Portdefinition im Modul abgeleitet werden kann.

Verbindungen zwischen Instanzen werden innerhalb der ADL immer auf in die entgegengesetzte Richtung propagiert.

nicht verwendete Instanzports müssen mit **open** gekennzeichnet werden um die Fehlerhafte Modellierungen zu vermeiden.

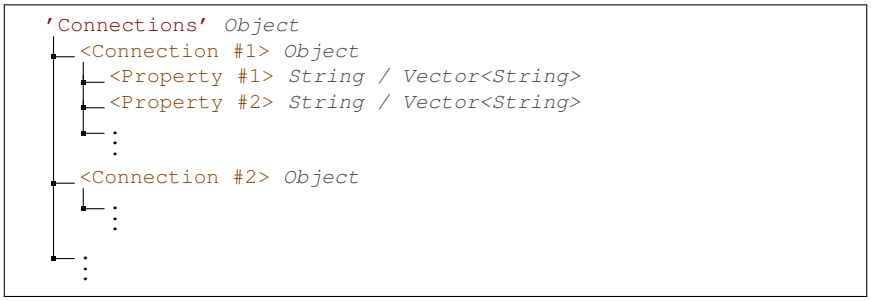


Abbildung B.9: Struktur des ADL-Blocks: Connections

B.8 Microarchitectures

Diese **section** beinhaltet detaillierte Informationen über Mikroarchitekturen, welche einem Prozessorelement zugeordnet werden können. Jede Mikroarchitektur wird über Ihren Namen eindeutig identifiziert und kann so in Modulen über dessen `Behavior` referenziert werden.

Jede Mikroarchitektur speichert dabei die von dieser Architektur unterstützten Datentypen, die Verfügbaren Ressourcen, wie Ausführungseinheiten oder Register, eine Instruktionssatzbeschreibung sowie Compiler spezifische Parameter. Abbildung B.10 gibt den Strukturellen Aufbau einer Mikroarchitekturbeschreibung an. Ein Beispiel ist in Quellcode B.7 abgebildet. Die einzelnen Komponenten sind im Anschluss genauer beschrieben.

B.8.1 DataTypes

Die Datentypen, die in diesem Abschnitt angegeben werden, geben an, in welcher Form sie von der Mikroarchitektur unterstützt werden. Datentypen können als **NATIVE** oder **EMULATED** definiert werden, wobei ein **NATIVE** Datentyp direkt von der Architektur unterstützt wird, während ein **EMULATED**

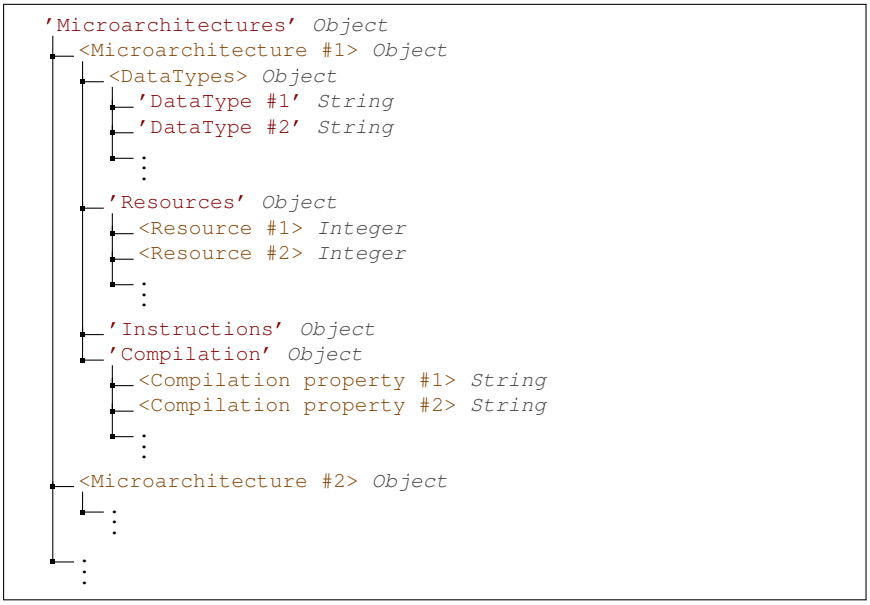


Abbildung B.10: Struktur des ADL-Blocks: Microarchitecture

```
['Microarchitectures'] = {
  ['RSIW2'] = { // Microarchitecture Name
    ['DataTypes'] = {
      ['Int<64>'] = "EMULATED";
      ['Int<32>'] = "NATIVE";
      ['2xInt<16>'] = "NATIVE";
      ['4xInt<8>'] = "NATIVE";
    };
    ['Resources'] = {
      ['ALU'] = 2;
      ['MUL'] = 1;
      ['LS'] = 1;
      ['BR'] = 1;
      ['RegFile'] = 1;
    };
    ['Instructions'] = {
      ['Add'] = {
        ['DataType'] {
          ['*'] = ('Int<8>', 'Int<16>', 'Int<32>');
        };
        ['Semantic'] = v('set', 'DDst', ('add', 'DSrc1', 'DSrc2'))
            ;
        ['CodeGeneration'] = '(%DSrc1 + %DSrc2)';
        ['Resources'] = {
          [] = {
            ['ALU'] = 1;
          };
        };
      };
      ...
    };
    ['Compilation'] = {
      ['Compiler'] = 'kahrisma-gcc';
      ['Parameter'] = "-target RSIW2";
    };
  };
};
```

Quellcode B.7: ADL Code Beispiel für eine Definition einer Microarchitecture

Datentyp nur durch zusätzliche Architekturfunktionen bereitgestellt wird. Nicht angegebene Datentypen sind in der Architektur nicht verfügbar. Für Single Instruction Multiple Data (SIMD) Befehle lassen sich die entsprechenden Datentypen durch Angabe der maximal parallel innerhalb einer Ressource verarbeitbaren Befehle mit der angegeben Bitbreite erweitern.

Die derzeit verfügbaren Datentypen sind Ganzzahl (**Int<>**) und Gleitkommazahlen (**Float<>**) da deren Hardwareunterstützung im Bereich der eingebetteten Systeme stark von der Architektur abhängt. Ein Multiplikator vor dem Datentyp gibt dabei die SIMD Parallelität an, während die Bitbreite des Datentyps nach dem Datentyp angegeben wird.

B.8.2 Resources

Der Ressourcen Block listet die in der Architektur verfügbaren Ausführungseinheiten auf, welche für ein klassisches Scheduling auf Prozessoren notwendig sind. Dabei wird grundsätzlich in Arithmetisch Logische Einheiten (ALU), Multiplikationseinheiten (MUL), Speicherzugriffseinheiten (LoadStore (LS)), Verzweigungsvorhersage (BR) und Registersätze (RegFile) unterschieden. Angegeben werden diese Typen als Schlüssel und deren Wert gibt die Anzahl an Verfügbaren Einheiten an. Eine Angabe mit einem Boolean Wert, **true** entspricht der Angabe **1**.

B.8.3 Instructions

Der **Block** `Instructions` beinhaltet eine Liste der in der Architektur verfügbaren Mikrobefehle und entspricht in der Namensgebung einer Assembler Beschreibung des Befehlssatzes. Jeder Befehl wird dabei durch die Angabe der kompatiblen Datentypen, einer semantischen Beschreibung, einer C-Code Repräsentation sowie den Benötigten Ressourcen und deren Ausführungszeit definiert.

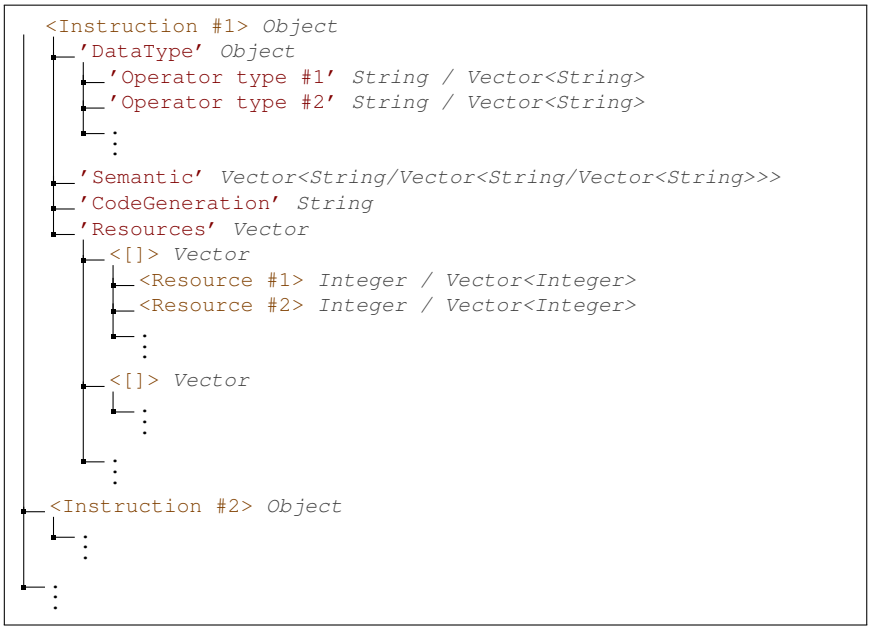


Abbildung B.11: Struktur des ADL-Blocks: Instruction

B.8.3.1 DataType

Datentypen, die innerhalb einer Instruktion angegeben werden, spiegeln die Anforderungen zum Ausführen des Befehls wieder. Dabei kann für jeden Operanden ein unterschiedlicher Datentyp angegeben werden. So ist das Ergebnis einer Multiplikation von zwei 32*Bit* breiten Operanden im Regelfall 64*Bit* breit, weshalb unterschiedliche Datentypen notwendig werden, sofern keine besonderen Behandlungen am Ergebnis vorgenommen werden. Die Angabe von"['*']" definiert die Verwendung eines Datentypen auf alle Operanden. Welchen Datentyp das Ergebnis besitzt ist vom Befehl abhängig und kann den Tabellen B.2, B.3 und B.4 entnommen werden.

B.8.3.2 Semantics

Für jeden Befehl wird eine semantische Beschreibung in Baumstruktur definiert, welche in der automatisierten Übersetzung verwendet werden kann. Diese semantische Beschreibung ist daher angelehnt an das LLVM Compilerframework [Lat13a, WB13] und unterstützt unäre sowie binäre Funktionen. Einen Spezialfall stellen Entscheidungs- und Vergleichsfunktionen dar, welche drei Operanden benötigen.

B.8.3.2.1 Unäre Operationen unäre Operationen besitzen in der Beschreibung zwei Parameter. Der erste Parameter beschreibt den Operationsnamen und der Zweite Parameter den Operanden auf dem der Befehl ausgeführt wird. Ein Übersicht der Verfügbaren unären Befehle ist in Tabelle B.2 aufgeführt. Das Ergebnis der Operation muss dabei mit der Operation *set*, siehe Abschnitt B.8.3.2.2, in ein Zielregister geschrieben werden.

Semantik	Beschreibung	Rückgabe-datentyp
negate, op1	Berechnet das Einerkomplement eines Registerwertes	entspricht op1
not, op1	Falls $op1 = 0$, dann ist das Ergebnis 1, andernfalls 0	
load op1	lädt den Wert an der Adresse, welche über op1 angegeben ist	

Tabelle B.2: Semantische Beschreibungen unärer Operationen

B.8.3.2.2 Binäre Operationen Binäre Operationen besitzen in der Beschreibung drei Parameter. Der erste Parameter beschreibt wiederum den Operationsnamen und der Zweite, sowie dritte Parameter die Operanden auf denen der Befehl ausgeführt wird. Ein Übersicht der Verfügbaren binären Befehle ist in Tabelle B.3 aufgeführt. Das Ergebnis der Operation muss dabei mit der binären Operation *set* in ein Zielregister geschrieben werden.

B.8.3.2.3 Vergleichs- und Auswahloperationen Diese Operationen besitzen in der Beschreibung vier Parameter. Der erste Parameter beschreibt auch hier den Operationsnamen während bei Vergleichsoperationen der zweite Parameter den Vergleichstyp beinhaltet und die beiden letzten Parameter die Operanden darstellen. bei einer Auswahlfunktion beinhaltet der zweite Parameter den zu prüfenden Operanden und der dritte sowie vierte Parameter die Ergebniswerte. Dabei kann eine Vergleichsoperation dazu verwendet werden den Prüfoperanden für die Auswahloperation zu ermitteln. Das Ergebnis der Auswahl Operation muss anschließend mit der binären Operation *set*, siehe B.8.3.2.2, in ein Zielregister geschrieben werden.

Quellcode B.8 zeigt nun eine Semantische Beschreibung einer Vergleichsoperation als Beispiel an. Die erste Zeile gibt dabei die Konstruktion Vergleichsoperation an und die untere Zeile eine konkrete Umsetzung eines Vergleichs auf Gleichheit.

Semantik	Beschreibung	Rückgabedatentyp
set, r, op1	setzt r auf den Rückgabewert von op1	entspricht op1
add, op1, op2	Addition von op1 und op2	Beide Operanden müssen einen identischen Datentypen ausweisen. Dann entspricht der Rückgabe-datentyp diesem Datentyp. Bei Vektor(SIMD)-Operationen kann op2 auch ein Skalar sein. Dann entspricht der Rückgabe-Datentyp dem Datentyp von op1
addusat, op1, op2	gesättigte vorzeichenlose Addition	
addssat, op1, op2	gesättigte vorzeichenbehaftete Addition	
sub, op1, op2	Subtraktion von op1 und op2	
subusat, op1, op2	gesättigte vorzeichenlose Subtraktion	
subssat, op1, op2	gesättigte vorzeichenbehaftete Subtraktion	
mul, op1, op2	Multiplikation von op1 und op2	
mulusat, op1, op2	gesättigte vorzeichenlose Multiplikation	
mulssat, op1, op2	gesättigte vorzeichenbehaftete Multiplikation	
divs, op1, op2	vorzeichenbehaftete Division von op2 durch op1 gerundet zur Null	
divu, op1, op2	vorzeichenlose Division	
rems, op1, op2	vorzeichenbehaftete Modulo Operation von op2 durch op1 (Vorzeichen entspricht op1)	
remu, op1, op2	vorzeichenlose Modulo Operation	
shl, op1, op2	logische Verschiebung nach links von op1 um op2 Stellen	
shru, op1, op2	logische Verschiebung nach rechts von op1 um op2 Stellen aufgefüllt mit Null	
shrs, op1, op2	arithmetische Verschiebung nach rechts von op1 um op2 Stellen, MSB von op1 wird mit dem Vorzeichen beschrieben	
and, op1, op2	Logische Bitweise UND Verknüpfung	
or, op1, op2	Logische Bitweise ODER Verknüpfung	
xor, op1, op2	Logische Bitweise Exklusive ODER Verknüpfung	
store, op1, op2	Schreibt den Wert von op1 an die Adresse welche durch op2 angegeben wird	—

Tabelle B.3: Semantische Beschreibungen binärer Operationen

Semantik	Beschreibung	Rückgabe-datentyp
sel, op1, falseValue, trueValue	Wenn op1 **falsch** oder Null ist, so wird **falseValue** zurückgegeben, andernfalls **trueValue**	
cmp, cond, op1, op2	Vergleich der beiden Operanden mit dem operater **cond**: Der Vergleichsoperator kann **eq**: gleich; **ne**: ungleich; **gts/gtu**: echt größer; **ges/geu**: größer oder gleich; **lts/ltu**: echt kleiner; **les/leu**: kleiner oder gleich sein. Für die letzten vier existieren Varianten mit (s) und ohne (u) Vorzeichen	Int<1>

Tabelle B.4: Semantische Beschreibung der Vergleichs- und Auswahloperationen

```
1 ['Semantic'] = v('set', 'target operand', ('select',('cmp', '
      comparison type', 'source one', 'source two'), 'false value',
      'true value'));
2
3 ['Semantic'] = v('set', 'DDst', ('select',('cmp', 'eq', 'DSrc1', '
      DSrc2'),'00000000', 'FFFFFFFF'));
```

Quellcode B.8: ADL Code Beispiel für eine Definition einer Vergleichsoperation mittels semantischer Beschreibung

B.8.3.3 Resources

Im **Ressource Block** einer Instruktion wird die Nutzung von Ressourcen durch die Instruktion beschrieben. Hierbei werden mehrere Ebenen verwendet. Auf oberster Ebene werden Ressourcenanforderungen als Liste UND-Verknüpft. Auf der zweiten Ebene werden die Ressourcen ODER-Verknüpft.

Elemente aus der Anforderungsliste können entweder sequentiell hintereinander als auch parallel zueinander ausgeführt werden, ein Beispiel hierzu ist in Quellcode B.9 zu sehen.

jedes Ressourcenelement auf der unteren Ebene wird nun mit einem Tupel-Wert definiert, der sowohl die Befehlsausführungsrate, als auch die Latenz der Befehlsausführung auf dieser Ressource beschreibt. Im Allgemeinen werden diese Ausführungszeiten in Takten angegeben. Die tatsächliche Ausführungszeit lässt sich anschließend über die Taktfrequenz berechnen.

Die Angabe von Ressourcen bestimmt allerdings nicht die dedizierte Ausführungseinheit, falls mehrere des gleichen Typs vorhanden sind. Ist dies gewünscht, so kann eine Ressource auch als deklarierter Typ angelegt werden. Ein Beispiel zwei unterschiedlicher Ressourcenallokationen ist in Abbildung B.9 dargestellt.

```
1  // Eine so definierte Instruktion benötigt zwei ALUs für jeweils 1
       Taktzyklus.
2  dies kann die gleiche ALU sequentiell in zwei aufeinanderfolgenden
       Takten sein, oder zwei unterschiedliche ALUs mit paralleler
     Ausführung
3  ['Resources'] = {
4    [] = {
5      ['ALU'] = 1;
6    };
7    [] = {
8      ['ALU'] = 1;
9    };
10 };
11 // Eine auf diese Weise definierte Instruktion benötigt eine ALU
     für 2 aufeinanderfolgende Zyklen.
12 Dies kann eine ALU sein, oder zwei ALUs die eine zwingend
     sequentielle Ausführung besitzen, so dass dieser Befehl auf
     jeden Fall 2 Taktzyklen benötigt
13 ['Resources'] = {
14   [] = {
15     ['ALU'] = 2;
16   };
17 };
```

Quellcode B.9: ADL Code Beispiel für eine Ressourcenallokation von Instruktionen

B.8.3.4 CodeGeneration

Der **Block** `CodeGeneration` dient der automatischen Quellcode Generierung durch externe Tools. Dabei wird hier ein C-Code Template als Wert angegeben. Funktionsoperanden können dabei durch einen Escapecharacter "%" definiert werden wodurch diese vom ADL Compiler ignoriert und für die weitere Ersetzung in externen Tools zur Verfügung steht.

Durch die Angabe einer C-Code Repräsentation in Kombination mit der semantischen Compilernahen und programmiersprachenunabhängigen Instruktionsbeschreibung können Code Analysen als auch Codegenerierung effizient implementiert werden und in beide Richtungen verwendet werden.

B.8.3.5 Compilation

Dieser Abschnitt enthält Informationen über den für diese Architektur zu verwendenden Compiler und entsprechende Flags, welche für eine erfolgreiche Kompilierung gesetzt werden müssen.

B.9 Behavior

Dieser Abschnitt beschreibt die in der Architekturbeschreibungssprache definierten Verhaltensparameter, welche in einem Modul, einer Konfiguration an einem Interface verwendet werden können. Verhalten werden als Objekt verwendet und besitzen einen definierten Namen sowie klar definierte Eigenschaften, welche mit Werten belegt werden können.

Die Verhaltensbeschreibungen sind immer für das Modul gültig in dem sie definiert werden und können beliebig kombiniert werden. Dies ermöglicht es in Hierarchischen Beschreibungen Verhaltenskombinationen zu beschreiben, die

in einer tieferen Ebene durch mehrere einzelne Module repräsentiert werden können.

Tabelle B.5 gibt eine Übersicht über die Verfügbaren Verhaltensbeschreibungen und ihre Eigenschaften. Durch die META-Sprachdefinition wird festgelegt, dass nur diese Verhaltensbeschreibungen erlaubt sind, wodurch die Analysierbarkeit der Sprache garantiert wird.

Verhalten	**Eigenschaften**	**Verwendung**
Reconfigurable	—	Module
MemoryMapping	Mappings	Module
Cache	HitDelay, Size, LineSize, Associativity, WriteStrategy ReplacementStrategy,	Module
Memory	Delay, Size	Module
NetworkAdapter	Latency, MessageTypes	Module
Network	Latency, Method, RoutingProtocol	Module
NetworkRouter	Latency	Module
Link	Latency, Throughput, Direction, Sender	Interface
Core	Microarchitecture	Module

Tabelle B.5: Übersicht über die definierten Verhaltensbeschreibungen in ADL

Um die Individualisierbarkeit einzelner Instanzen zu erlauben, können mit `SubInstance`, siehe B.7.1, Verhaltenseigenschaften überschrieben oder ganze Verhaltensbeschreibungen ergänzt werden.

B.9.1 Reconfigurable

Dieser Wert kann auf wahr oder falsch gesetzt werden und sagt aus, dass ein Modul rekonfigurierbar ist. Rekonfigurierbare Module müssen zwangsläufig auch in einer Konfiguration verwendet werden.

B.9.2 MemoryMapping

Mit Hilfe eines `MemoryMapping` können Zugriffsadressen auf Speicherbereiche berechnet werden, und dient dazu lokale Speicheradressen in entfernte

Speicheradressen umzurechnen. Speicherzuordnungen können über mehrere Module hinweg kaskadiert werden. Die Berechnung erfolgt anschließend rekursiv. Speicherzuordnung gelten jeweils für den angegebenen Port eines Moduls. Die anzugebenden Adressen sind jeweils die lokale und die entfernte Startadresse des Bereichs, welcher durch die Größe definiert wird.

MappingName	Modulweit eindeutiger Name.
Port	Modulport an welchen der Speicherbereich angeschlossen wird.
LocalAddress	Lokale Startadresse des Speicherbereichs.
TargetAddress	Entfernte Startadresse des Speicherbereichs.
Size	Größe des Adressraums in Bytes.

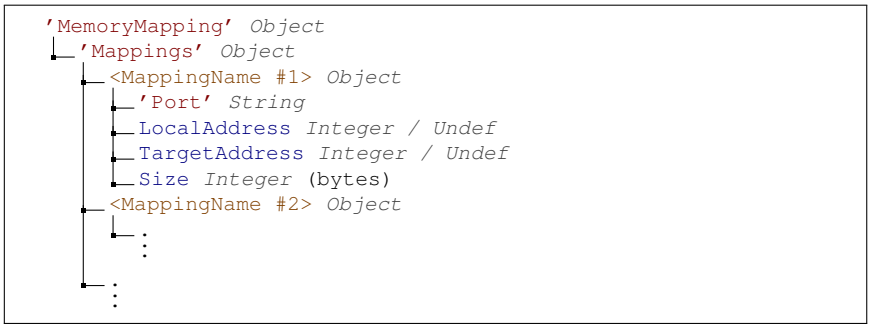


Abbildung B.12: Aufbau der MemoryMapping Verhaltensbeschreibung

Sind entweder `LocalAdress` oder `TargetAdress` innerhalb eines Moduls nicht definiert, so muss Sie über `SubInstance` bei der Instanziierung mit angegeben werden. Alternativ kann eine Simulationsklasse diese Aufgabe übernehmen. In diesem Fall ist allerdings keine statische Analyse mehr möglich.

B.9.3 Cache

Die Beschreibung von `Cache` Eigenschaften beinhaltet einige Parameter, welche die Struktur eines Caches sowie die zeitlichen Eigenschaften beschreibt:

HitDelay Reaktionszeit in Takten, wenn die angefragten Daten im Cache liegen.

Size Größe des Caches in Bytes.

LineSize Größe eine Cache-Zeile in Bytes.

Associativity Beschreibt den Faktor der Assoziativität des Caches.

ReplacementStrategy definiert die Ersetzungsstrategie des Caches: Derzeit werden zuletzt verwendet (`LRU`), am wenigsten Verwendet (`LFU`) und Zufällig (`Random`) unterstützt.

WriteStrategy Definiert die Rückschreibstrategie: Erlaubt sind Durchschreiben (`WriteThrough`) oder Zurückschreiben (`WriteBack`).

```
'Cache' Object
  HitDelay Integer (cycles)
  Size Integer (bytes)
  LineSize Integer (bytes)
  Associativity Integer
  'ReplacementStrategy' String
  'WriteStrategy' String
```

Abbildung B.13: Aufbau der Cache Verhaltensbeschreibung

B.9.4 Memory

Dieses Verhalten beschreibt einfache und allgemeine Speicherparameter für Speicher mit zufälligem Lese und Schreibzugriff.

Delay Reaktionszeit in Takten bis zur Verfügbarkeit der Daten.

Size Größe des Speichers in Bytes.

Abbildung B.14: Aufbau der Memory Verhaltensbeschreibung

B.9.5 NetworkAdatper

Die Verhaltensbeschreibung Netzwerkadapter wird dazu verwendet das Kommunikationsverhalten einer Verbindung auf Prozessorseite zu beschreiben. Diese Beschreibung nutzt die Applikationsebene (Schicht 7 im ISO-OSI Modell) und definiert Nachrichten, welche durch diese Verbindung übertragen werden können.

Das Verhalten `NetworkAdapter` besitzt dabei folgende Struktur:

Latency Beschreibt die Verarbeitungszeit einer Nachricht in Zyklen im Netzwerkadapter.

MessageTypes Objekt, welche alle Nachrichtentypen beinhaltet.

MessageType Name Objekt, welches einen Nachrichtentyp definiert.

DataSize Gibt die Nutzdatenmenge einer Nachricht als Vektor mit Minimum und Maximum an. Sind Minimum und Maximum identisch, so kann auch der skalare Wert angegeben werden.

DataDirection Beschreibt die Übertragungsrichtung einer Nachricht. Die Richtung kann als sendend (`S`), empfangend (`R`) oder bidirektional (`SR`) definiert werden.

SendOverhead Notwendige Zusatzinformationen in Bits beim Senden einer Nachricht, die zur Nutzdatenmenge addiert werden.

ReceiveOverhead Notwendige Informationen in Bits die nach Empfang einer Nachricht als Antwort verschickt werden.

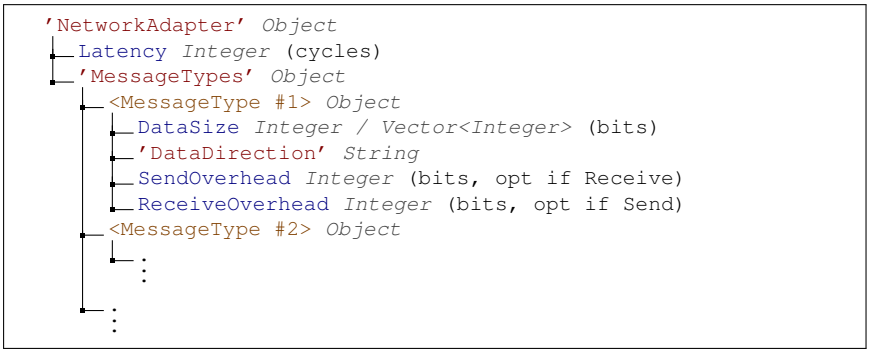


Abbildung B.15: Aufbau der NetworkAdapter Verhaltensbeschreibung

Die Übertragungsrichtung gibt an, in welche Richtung Datenpakete verschickt werden können. D.h. Sie definiert, ob der Netzwerkempfänger mit diesem Nachrichtentyp Daten senden, empfangen oder beides kann. Eine Nachricht kann dadurch als einfache Nachricht definiert werden, in dem dieser Wert auf sendend oder empfangend gesetzt wird. Wird eine Bidirektionale Nachricht

definiert, so erhält Sie ein Transaktionsverhalten. Die Größe der generierten Antwort, um eine Transaktion als abgeschlossen zu markieren, wird durch das Element `ReceiveOverhead` beschrieben.

B.9.6 Network

Das `Netzwerk`-Verhalten beschreibt die grundlegenden Eigenschaften, die benötigt werden um ein Kommunikationsnetzwerk auf hoher Ebene zu beschreiben. Dabei ist dieses Verhalten ausreichend, wenn nur eine sehr abstrakte Ebene analysiert oder simuliert wird. Werden darunterliegende Ebenen verwendet, so kann dieses Verhalten durch mehrere Instanzen ersetzt werden, bzw. kann bspw. in einem Router ein genaueres Verhalten beschreiben.

Latency — Durchschnittliche Übertragungszeit im gesamten Netzwerk.

Method — Art des Verbindungsaufbaus: Packetvermittelt (`Packet`) oder Leitungsvermittelt (`Circuit`).

RoutingProtocol — Beschreibt das verwendetet Routingprotokoll.

```
'Network' Object
  Latency Integer (cycles)
  'Method' String
  'RoutingProtocol' String
```

Abbildung B.16: Aufbau der Network Verhaltensbeschreibung

B.9.7 NetworkRouter

Der Netzwerkrouter ist eine Komponente eines Netzwerks und wird in unteren Hierarchieebenen verwendet. Das Verhalten definiert eine detailliertere

Beschreibung eines Netzwerks, wenn es mit `NetworkLinks` verwendet wird. Dieses Verhalten kann nicht gleichzeitig mit `Network` verwendet werden, bzw. überschreibt dieses.

Latency Verarbeitungszeit von Paketen im Router.

```
'NetworkRouter' Object
└─Latency Integer (cycles)
```

Abbildung B.17: Aufbau der NetworkRouter Verhaltensbeschreibung

B.9.8 NetworkLink

Der `NetworkLink` beschreibt das Verhalten eines Links und wird üblicherweise innerhalb eines `Interface` verwendet. Das verhalten definiert eine Übertragungszeit sowie den reinen Datendurchsatz und die Übertragungsrichtung. Diese kann als Simplex – streng Unidirektional, HalfDuplex – abwechselnd bidirektional oder FullDuplex – bidirektional definiert werden. Dazu definiert dieses Verhalten den PortType des Interface auf den dieses Verhalten angewandt werden soll. Dazu ist es möglich auf unterschiedlichen Porttypen je ein anderes Verhalten zu beschreiben.

Latency Übertragungszeit in Takten für den in Throughput angegeben Bit-Wert.

Throughput Datendurchsatz je Richtung in Bits.

Direction Übertragungsrichtung: Simplex, HalfDuplex oder FullDuplex.

Sender PortType des Interface.

Um die Übertragungsgeschwindigkeit zu bestimmen muss der Durchsatz durch die Latenz geteilt werden. Wird eine Fullduplex Verbindung verwendet so kann der Durchsatz Wert verdoppelt werden, da zwei Übertragungswege gleichzeitig verwendet werden können.

```
'Link' Object
  ├─ Latency Integer (cycles)
  ├─ Throughput Integer (bits)
  ├─ 'Direction' String
  └─ 'Sender' String (opt if !Simplex)
```

Abbildung B.18: Aufbau der Link Verhaltensbeschreibung

B.9.9 Core

Das Verhalten `core` wird dazu verwendet ein Modul oder eine Instanz als Verarbeitungseinheit zu kennzeichnen. Dabei kann es sich um einen einfachen Addierer handeln oder um komplexe Prozessoren. Diese Komplexität wird in einer Mikroarchitektur beschrieben. Diese wird als Referenz angegeben und beschreibt die notwendigen Details (siehe B.8).

Microarchitecture Referenz zur Mikroarchitekturbeschreibung.

```
'Core' Object
  └─ 'Microarchitecture' String
```

Abbildung B.19: Aufbau der Core Verhaltensbeschreibung

Anhang C

Simulator Anwendung und Steuerung

Dieser Anhang beschreibt die Steuerungsbefehle des Simulators und dessen Kommandozeilenparameter. Dabei wird das Befehlsformat und die verfügbaren Befehle samt ihrer Auswirkungen auf den Simulator beschrieben. Des Weiteren erlaubt das Framework die Verwendung von Konfigurationsdateien um den Simulator zu konfigurieren. Der Aufbau der Konfigurationsdateien wird hier ebenso beschrieben, wie die spezifischen Kommandozeilenparameter des Kahrisma Simulationsmoduls.

Der Multicore Simulator wird als Anwendung mit Kommandozeileninterface (Command Line Interface (CLI)) kompiliert und kann damit anderen Werkzeugen einfach zur Verfügung gestellt werden. Außerdem lässt er sich damit auch als eigenständiges Werkzeug zur Simulation nutzen.

Das CLI erlaubt durch die Angabe von Befehlen und Argumenten die Steuerung des Simulators wodurch er auch Skriptbasiert in Entwurfsraumexplorationswerkzeugen oder der ALMA-Parallelisierungstoolchain effektiv eingesetzt werden kann.

Die zur Verfügung stehenden Befehle lassen sich in Globale Befehle und Spezifische Befehle unterteilen. Zu letzteren gehören sowohl ADL-Modul, als auch ADL-Instanz-spezifische Befehle, die direkten Einfluss auf die ADL-Implementierung der Module und Instanzen und damit auf die Simulationsklassen nehmen können.

In Quellcode C.1 ist nun das Kommandozeilenformat für den Aufruf des SystemSimulators angegeben. Spitze Klammern geben dabei Argumente an, während optionale Kommandozeilenkomponenten mit eckigen Klammern beschrieben werden.

```
1 Systemsim.exe --sadl <ADL-File> [#Module <elf-file> --cmd
      <cmd_arg>]
```

Quellcode C.1: Kommandozeilenformat des System Simulators

C.1 Globale Kommandozeilenbefehle

Die Befehle der Kommandozeilenanwendung die in die Kategorie der Globalen Befehle gehören, steuern die Konfiguration des Frameworks und des Simulators an sich. Als Bezeichner für Befehle wird der "- -" Operator verwendet der direkt vor dem Befehl steht Nach dem Befehl werden die Argumente des Befehls getrennt von Leerzeichen in einer Liste aneinander gereiht.

```
1 --Befehl <Befehlsargument>
```

Quellcode C.2: Globales Befehlsformat des System Simulators

Das Befehlsformat ergibt sich damit wie folgt:

Tabelle C.1 zeigt eine Übersicht der Verfügbaren Globalen Kommandozeilenbefehle an:

C.2 Modul Spezifische Kommandozeilenparameter

Diese Kommandozeilenbefehle werden vom Simulator als Text erfasst und anschließend unverändert an die Simulationsmodule weitergereicht. Dadurch wird es einem Simulationsmodulentwickler ermöglicht eigene Kommandozeilenbefehle zu definieren um das Modul bei Simulationsstart konfigurieren zu können oder vom Entwickler definierte Optionen auszuwählen. Dabei muss das Simulationsframework nicht angepasst werden und kann so auf die dynamische Systemerzeugung aus der ADL-Datei heraus reagieren.

Als Bezeichner wird "#" in Kombination mit einem ADL-Modulnamen verwendet. Der Bezeichner kann mehrfach verwendet werden um mehrere Module anzusprechen, muss aber hinter allen Globalen Befehlen stehen, da alle Zeichen nach dem Modul-Bezeichner bis zu einem weiteren Modul-Bezeichner oder einem Instanz-Bezeichner als Text interpretiert werden.

```
1 #Module_Name [arguments]
```

Quellcode C.3: CLI Trennzeichen für Module

Das Format der Modul-Spezifischen Kommandozeilenbefehle und der Argumentliste wird in dem jeweiligen SystemC Modul definiert. Die Argumente werden als Text an das SystemC Modul übergeben. Das SystemC Modul muss

Befehl	Argument Beschreibung
`--sadl`	`<ADL-file>` gibt einen Pfad zu und den Namen einer ADL-Datei an, welche für die System Simulation verwendet werden soll. Dieser Befehl muss immer angegeben werden, direkt oder über Konfigurationsdateien.
`--cfg`	`<config-files>` gibt einen Pfad zu und den Namen einer Konfigurationsdatei an, welche weitere Befehle und Argumente Global oder Spezifischer Art beinhalten kann.
`--set`	`<string>` Erlaubt die Angabe von ADL-Quellcode der an die ADL Datei angehängt und mit kompiliert wird.
`--quiet`	 schaltet die Ausgaben des Simulators während der Simulation aus.
`--ssim-profile`	`<out-prefix>` aktiviert das Profiling mit Instrumentierten Anwendungen. Die Ausgabedatei wird benannt mit dem angegebenen Prefix sowie dem Namen des Profiling Types. Die Ausgabe erfolgt im JSON Format.
`--ssim-stats`	`<out-prefix>` aktiviert die Erzeugung einer Statistikdatei des Systemsimulators, welche interne Statistiken des Simulators, sowie soweit Implementiert auch der Module erzeugt. Die Ausgabedatei wird benannt mit dem angegebenen Präfix sowie einer Statistikbezeichnung. Die Ausgabe erfolgt im JSON Format.
`--ssim-trace`	`<out-prefix> [ProfilingTypeName=[<Paraver Config File>]]` aktiviert das Tracing interner Profiling Funktionen oder instrumentierter Anwendungen. IM Gegensatz zu ssim-profile werden die Ausgaben in das Paraver Trace File Format transformiert und können mit einer Paraver-Konfigurationsdatei für die Visualisierung konfiguriert werden. Die Ausgabedateien pro Profiling Type werden benannt mit dem angegebenen Präfix sowie dem Namen des Profiling Types. Die Ausgabe erfolgt im Paraver State Trace Format.

Tabelle C.1: Global Befehle für den System Simulator

die Befehlsinterpretation innerhalb der Initialisierung des Moduls implementieren.

Der Argumenttext wird dabei an alle Instanzen des ADL-Moduls, also auch an alle SystemC-Instanzen dieses Moduls weitergeleitet. Diese Weiterleitung wird durch den Befehlsinterpreter sichergestellt, welcher in der Simulationsbasisklasse implementiert ist. Die allgemeine Simulationsmodulklasse, von der alle Simulationsmodule abgeleitet werden müssen beinhaltet die Funktionen zur Annahme und Speicherung des Textarguments und stehen somit immer zur Verfügung.

Der Modul-Bezeichner wird gleichzeitig als Trennzeichen interpretiert, weshalb die Verwendung von Modul-Spezifischen Argumenten welche "#" im Namen tragen unzulässig ist.

Der Befehlsinterpreter erlaubt allerdings die Angabe eines Escape-Zeichens. Wird innerhalb der Kommandozeile ein doppelter Modul-Bezeichner aufgeführt, also "##", so wird ein Bezeichner entfernt und der zweite Bezeichner als Text an das Modul weitergegeben.

Im Folgenden ist ein Beispiel zur Anwendung des Bezeichners innerhalb eines Arguments angegeben.

```
1 #Module_Name ##argument [arguments]
```

Quellcode C.4: Escape Zeichen für Modul-Bezeichner

C.3 Instanz Spezifische Kommandozeilenbefehle

Kommandozeilenbefehle, welche für spezifische Module bereitstehen können auch auf einzelne Instanzen angewandt werden. Dies hat zur Folge, dass die Befehle vom Simulator nur an die ausgewählte Instanz und nicht an das Simulationsmodul weiterleitet werden. Dadurch werden alle anderen Instanzen

eines Moduls unverändert zur Simulation verwendet und nur eine einzelne Instanz konfiguriert. Dies kann zur weiteren Individualisierung einzelner Instanzen verwendet werden.

Analog zum Bezeichner für Module, dem "#", wird hierfür ein "@" als Bezeichner und "@@" als Escape-Zeichen verwendet.

```
1 @Instance_Name [arguments]
```

Quellcode C.5: CLI Trennzeichen für Instanzen

```
1 @Instance_Name @@argument [arguments]
```

Quellcode C.6: CLI Escape Zeichen für Instanznamen

C.4 Konfigurationsdateien

Innerhalb der Kommandozeilenparameter ist es möglich eine Konfigurationsdatei anzugeben, welche weitere Befehle an den Simulator enthalten kann. Da der Simulator die Angabe einer ADL-Datei erfordert, kann diese als Kommandozeilenargument oder innerhalb einer Konfigurationsdatei angegeben werden. Es ist auch möglich mehrere Konfigurationsdateien anzugeben. die in den Dateien angegebenen Befehle werden zu einem Befehlsstrom zusammengefasst und anschließend verarbeitet. Dabei ist die Reihenfolge der Konfigurationsdateien ausschlaggebend dafür, welche Werte verwendet werden. Grundsätzlich gilt, dass importierte Konfigurationsdateien bestehende Werte überschreiben. Eine Ausnahme bilden Befehle die über die Kommandozeile angegeben werden. Diese werden immer mit höherer Priorität gegenüber allen Konfigurationsdateien behandelt.

Die Konfigurationsdateien basieren, wie die ADL auch, auf der Syntax der Datenbeschreibungssprache. Innerhalb einer Konfigurationsdatei werden, analog zu den Kommandozeilenparametern, drei Bereiche definiert um die Funktionen globaler als auch Modul- und Instanz-spezifischer Befehle abzubilden. Der

Bereich `Global` beinhaltet alle befehle die zur Steuerung des Simulators und des Frameworks zur Verfügung stehen. Jeder Befehl wird dabei als Schlüssel ohne den Befehlsbezeichner "--" verwendet und die zugehörigen Argumente als Werte, zum Beispiel als Zeichenkette, Vektor oder Zahlenwert, dem Schlüssel zugeordnet. Ein Beispiel einer Konfigurationsdatei ist in Abbildung C.7 dargestellt.

Modul- und Instanzspezifische Befehle werden unter den Blöcken `Modules` und `Instances` zusammengefasst. Unterblöcke mit dem jeweiligen Modul oder Instanznamen übernehmen dabei die Funktion der entsprechenden Bezeichner "#" und "@", welche dadurch entfallen können.

Auch hier werden Befehle als Schlüssel verwendet und deren Argumente als Werte. Sind mehrere Argumente vorhanden können diese als Vektor oder als Zeichenkette angegeben werden. Vektoren werden vom Framework automatisch aufgelöst. Im Gegensatz zum globalen Befehlen müssen Befehle und Argumente aber so angegeben werden wie sie vom Modulentwickler vorgegeben wurden und vom Modul verarbeitet werden können.

```
1  [Global] = {
2    [sadl] = <ADL-FILE>;
3    [cfg] = v(config-file-1, ..., config-file-n);
4  };
5  [Modules] = {
6    [Module_Name] = {
7      //Module Specific options
8    };
9  };
10 [Instances] = {
11   [Instance_Name] = {
12     //Instance-specific options
13   };
14 };
```

Quellcode C.7: Aufbau einer Konfigurationsdatei für den System Simulator

C.5 Software und Anwendungen für die Simulation

Die Verwendung von Anwendungsdateien und Applikationen, welche auf der Hardware Architektur simuliert werden sollen, ist stark vom verwendeten Simulationsmodul abhängig. Das Applikationsformat als auch der Bereitstellung von ausführbaren Dateien muss über Modul- oder Instanzspezifische Befehle behandelt werden. Das Simulationsframework unterstützt nativ keine Ausführung von Softwaredateien, sondern benötigt einen Instruktionssatzsimulatormodul, welches die Dateien ausführt und simuliert.

C.6 Kahrisma Spezifische Kommandozeilenparameter

Da der Kahrisma Prozessor als Anwendungsfall dieser Architekturbeschreibung dient und auch für die Evaluation verwendet wird sollen hier die für diesen Prozessorsimulator zur Verfügung stehenden Kommandozeilenparameter beschrieben und erläutert werden. Dieses Kapitel dient daher auch als Beispiel für eine Implementierung und Umsetzung des zuvor beschriebenen Beschreibungs- und Simulationsframework. Es zeigt außerdem auf, wie ein Simulator, welcher nicht auf SystemC basiert und das Framework eingebunden werden kann.

Da Kahrisma ein natives Multicoresystem mit ISA Runtime Rekonfigurierbarkeit darstellt, wurde für die Simulation ein dediziertes Laufzeitsystem (Run Time System - RTS) implementiert, welches die Konfigurierbarkeit des KahrismaSystems übernimmt. Alle Kommandozeilenparameter werden daher nur an das RTS weitergeleitet, welches anschließend die Befehle wiederum an die entsprechenden Prozessorsimulatoren übergibt.

Das RTS Modul unterstützt die im Kahrisma Prozessor definierten befehle, welche als Argumentliste an das Modul übergeben werden. Mindestens eines dieser Argumente muss ein Pfad zu einer Ausführbaren Datei im ELF-Format sein, welche die zu simulierende Applikation beinhaltet. Es wird empfohlen diesen Parameter zu Beginn der Argumentliste zu schreiben, da er ohne vorausgehenden Befehl angegeben wird. Er kann aber auch an anderen Stellen definiert werden, wenn alle anderen Befehle und deren Argumentlisten beachtet werden um zu verhindern, dass die Applikation als Argument eines anderen Befehls missgedeutet wird.

Alle Kahrisma.spezifischen Befehle starten mit "- -" als Bezeichner, welcher direkt vom Befehlsnamen gefolgt wird. Befehlsargumente werden mit Leerzeichen getrennt jeweils hinter dem Befehl beschrieben. Um Befehle und Argumente an die zu simulierende Anwendung weiterzuleiten existiert der Befehl "- -args". Dieser übergibt alle hinter dem Befehl beschriebenen Argumente an die Applikation und wird nur noch durch die Frameworkkommandos "#" und "@" begrenzt. Daher müssen alle Simulatoroptionen für das Kahrismamodul vor dem "- -args" platziert werden.

Quellcode C.8 gib t nun das Format und Reihenfolge der Kahrismabefehle innerhalb der Kommandozeile an.

```
1 #Kahrisma_RTS_Module <elf-file> [--opt <opt_args>] [--args
      <arguments>]
```

Quellcode C.8: Kommandozeilenformat für die Kahrisma Simulationsmodule

Tabelle C.2 Stellt eine Übersicht über die verfügbaren Kommandozeilenparameter für das Kahrisma RTS bereit:

Befehl	Beschreibung
`<elf-file>`	gibt einen Pfad zu einer Ausführbaren Datei mit der zu simulierenden Applikation an.
`--trace <filename>`	schaltet das vollständige Applikationstracing an. Der Simulator generiert ein Instruktionstracing der ausgeführten Applikation und speichert dies unter dem angegebenen Namen ('<filename>.trace') ab.
`--simpletrace <filename>`	Schaltet ein einfaches Applikationstracing an, welches lesbare Instruktionsinformationen sammelt aber weniger Informationen als das vollständige Tracing beinhaltet.
`--alltrace <filename>`	schaltet alle verfügbaren Tracing Methoden des Simulators an.
`--stats <filename>`	Erzeugt eine Statistikausgabe des Simulators über die ausgeführte Applikation.
`--asmstats <filename>`	Erzeugt eine Statistikausgabe des Assemblers über die ausgeführte Applikation.
`--funcstats <filename>`	Erzeugt eine Statistikausgabe über Funktionen und Funktionsaufrufe der ausgeführten Applikation.
`--allstats <filename>`	Erzeugt alle vorgenannten Statistikausgaben.
`--crashdump <filename>`	Erstellt einen vollständiges Abbild im Falle eines Absturzes oder unkontrollierter Beendigung der Simulation.
`--callgraph <filename>`	Erzeugt einen Callgraph der Applikation.
`--quiet`	unterdrückt alle Ausgaben des Simulators während der Simulation
`-`	leitet alle Ausgaben der Simulation an std:out weiter.
`| <programm>`	Die 'pipe' arbeitet wie die Linux Pipe und öffnet das angegebene Programm und leitet alle Ausgaben an dieses weiter.
`| gzip --o <filename>.gz`	Dies ist ein Beispiel für die Funktionsweise der Pipe: Es öffnet gZip und leitet alle Ausgaben daran weiter um eine komprimierte Ergebnisdatei zu erzeugen.

Tabelle C.2: Kahrisma-spezifische Kommandozeilenbefehle

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Abkürzungsverzeichnis

ADL	Architecture Description Language.
ALMA	ALgorithm parallelization for Multicore Architectures.
API	Application Programming Interface.
ASIC	Application-Specific Integrated Circuit.
ASIP	Application-Specific Instruction Set Processor.
AT	Approximately-Timed.
CABA	Cycle-Accurate / Bit-Accurate.
CADL	Computer Architecture Description Language.
CGADL	Coarse-Grained Array Description Language.
CISC	Complex Instruction Set Computer.
CLI	Command Line Interface.
CPU	Central Processing Unit.
DDL	Data Description Language.
DSE	Design Space Exploration.
DSX	SoClib Design Space Explorer.
DT	Distributed Time.

EADL	Embedded Architecture Description Language.
EDA	Electronic Design Automation.
EDPE	Encapsulated Datapath Element.
ELF	Executable and Linkable Format.
FIFO	First In First Out.
GeCoS	Generic Compiler Suite.
GPU	Graphics Processing Unit.
HDL	Hardware Description Language.
HeMPS	Hermes Multi Processor System.
HTG	Hierarchical Task Graph.
I/O	Input/Output.
IDE	Integrated Development Environment.
ILP	Instruction Level Parallelism.
iNoC	invasive Network-on-Chip.
IP	Intellectual Property.
ISA	Instruction Set Architecture.
ISS	Instruction Set Simulator.
JSON	JavaScript Object Notation.
KAHRISMA	KArlsruhe's Hypermorphic Reconfigurable-Instruction-Set Multi-grained-Array.
L1	First Level.
L2	Second Level.
LLVM	Low Level Virtual Machine.

LUT	LookUp Table.
MIMD	Multiple Instruction Multiple Data.
MISD	Multiple Instruction Single Data.
MPI	Message Passing Interface.
MPSoC	Multi-Processor System-on-Chip.
OSCI	Open SystemC Initiative.
RISC	Reduced Instruction Set Computer.
RSIW	Run-Time Scalable Issue-Width.
RTL	Register-Transfer Level.
SIMD	Single Instruction Multiple Data.
SoC	System-On-Chip.
TLM	Transaction-level Modeling.
TLP	Thread Level Parallelism.
UML	Unified Modelling Language.
VHDL	Very High Speed Integrated Circuit Hardware Description Language.
VLIW	Very Long Instruction Word.
XML	Extensible Markup Language.

Literaturverzeichnis

[ABAA05] ARAUJO, Cristiano ; BARROS, Edna ; AZEVEDO, Rodolfo ; ARAUJO, Guido: Processor Centric Specification and Modeling of MPSoCs Using ArchC. In: *Forum on specification and Design Languages*, 2005, S. 303–315

[Acc11] ACCELLERA SYSTEMS INITIATIVE: *Accellera and Open SystemC Initiative (OSCI) Approve Merger, Unite to Form Accellera Systems Initiative*. `http://www.accellera.org/news/pr/view?item{_}key=f27cc80b296f97da1746984c3615407770f4c442`. Version: 2011

[ACL+06] ANGIOLINI, Federico ; CENG, Jianjiang ; LEUPERS, Reiner ; FERRARI, Federico ; FERRI, Cesare ; BENINI, Luca: An Integrated Open Framework for Heterogeneous MPSoC Design Space Exploration. In: *Proceedings of the Design Automation and Test in Europe Conference* (2006), 1–6. `http://dx.doi.org/10.1109/DATE.2006.244000`. – DOI 10.1109/DATE.2006.244000. – ISBN 3–9810801–1–4

[AGB+05] ARAUJO, Cristiano ; GOMES, Millena ; BARROS, Edna ; RIGO, Sandro ; AZEVEDO, Rodolfo ; ARAUJO, Guido: Platform designer: An approach for modeling multiprocessor platforms based on SystemC. In: *Design Automation for Embedded Systems* 10 (2005), Nr. 4, 253–283. http://dx.doi.org/10.1007/s10617-006-0654-9. – DOI 10.1007/s10617–006–0654–9. – ISSN 09295585

[ALSU06] AHO, Alfred V. ; LAM, Monica S. ; SETHI, Ravi ; ULLMAN, Jeffrey D.: *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2006. – ISBN 0321486811

[ANMD07] ATITALLAH, Rabie B. ; NIAR, Smail ; MEFTALI, Samy ; DEKEYSER, Jean L.: An MPSoC performance estimation framework using transaction level modeling. In: *Proceedings - 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2007*, 2007. – ISBN 0769529755, 525–533

[ARB+05] AZEVEDO, Rodolfo ; RIGO, Sandro ; BARTHOLOMEU, Marcus ; ARAUJO, Guido ; ARAUJO, Cristiano ; BARROS, Edna: The ArchC architecture description language and tools. In: *International Journal of Parallel Programming* 33 (2005), Nr. 5, 453–484. http://dx.doi.org/10.1007/s10766-005-7301-0. – DOI 10.1007/s10766–005–7301–0. – ISBN 1076600573

[Bar14] BARCELONA SUPERCOMPUTING CENTER: *Paraver: a flexible performance analysis tool*. 2014

[BBB+05] BENINI, Luca ; BERTOZZI, Davide ; BOGLIOLO, Alessandro ; MENICHELLI, Francesco ; OLIVIERI, Mauro: MPARM: Exploring the multi-processor SoC design space with systemC. In: *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology* 41 (2005), Nr. 2, S. 169–182. http://dx.doi.org/10.

1007/s11265-005-6648-1. – DOI 10.1007/s11265–005–6648–1. – ISSN 13875485

[BBF+08] BELTRAME, Giovanni ; BOLCHINI, Cristiana ; FOSSATI, Luca ; MIELE, Antonio ; SCIUTO, Donatella: ReSP: A Non-Intrusive Transaction-Level Re ective MPSoC Simulation Platform for Design Space Exploration. In: *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC*, 2008. – ISBN 978–1–4244–1921–0, S. 673–678

[BBWA09] BAERT, R. ; BROCKMEYER, E. ; WUYTACK, S. ; ASHBY, T.J.: Exploring parallelizations of applications for MPSoC platforms using MPA. In: *2009 Design, Automation & Test in Europe Conference & Exhibition*, 2009. – ISBN 978–1–4244–3781–8, 1148–1153

[BDC+07] BARTHOU, Denis ; DONADIO, Sebastien ; CARRIBAULT, Patrick ; DUCHATEAU, Alexandre ; JALBY, William: Loop optimization using hierarchical compilation and kernel decomposition. In: *International Symposium on Code Generation and Optimization, CGO 2007* (2007), S. 170–182. http://dx.doi.org/10.1109/CGO.2007.22. – DOI 10.1109/CGO.2007.22. ISBN 0769527647

[BFS09] BELTRAME, Giovanni ; FOSSATI, Luca ; SCIUTO, Donatella: ReSP: A nonintrusive transaction-level reflective MPSoC simulation platform for design space exploration. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28 (2009), Nr. 12, 1857–1869. http://dx.doi.org/10.1109/TCAD.2009.2030268. – DOI 10.1109/TCAD.2009.2030268. – ISBN 9781424419227

[BG07] BUCHMANN, Richard ; GREINER, Alain: A fully static scheduling approach for fast cycle accurate SystemC simulation of MPSoCs. In: *Proceedings of the International Conference on Microelectronics, ICM*, Institute of Electrical and Electronics Engineers, 2007. – ISBN 9781424418473, 101–104

[BG10] BUCHMANN, Richard ; GREINER, Alain: *SystemCASS*. https://www-asim.lip6.fr/trac/systemcass. Version: 2010

[BSO+12] BECKER, Jürgen ; STRIPF, Timo ; OEY, Oliver ; HUEBNER, Michael ; DERRIEN, Steven ; MENARD, Daniel ; SENTIEYS, Olivier ; RAUWERDA, Gerard ; SUNESEN, Kim ; KAVVADIAS, Nikolaos ; MASSELOS, Kostas ; GOULAS, George ; ALEFRAGIS, Panayiotis ; VOROS, Nikolaos S. ; KRITHARIDIS, Dimitrios ; MITAS, Nikolaos ; GOEHRINGER, Diana: From Scilab to high performance embedded multicore systems - The ALMA approach. In: *Proceedings - 15th Euromicro Conference on Digital System Design, DSD 2012* (2012), 114–121. http://dx.doi.org/10.1109/DSD.2012.65. – DOI 10.1109/DSD.2012.65. ISBN 9780769547985

[BSS+06] BELTRAME, G. ; SCIUTO, D. ; SILVANO, C. ; LYONNARD, D. ; PILKINGTON, C.: Exploiting TLM and Object Introspection for System-Level Simulation. In: *Proceedings of the Design Automation & Test in Europe Conference* 1 (2006). http://dx.doi.org/10.1109/DATE.2006.244004. – DOI 10.1109/DATE.2006.244004. – ISBN 3–9810801–1–4

[BVL09] BARNES, Christopher ; VAIDYA, Pranav ; LEE, Jaehwan J.: An XML-based ADL framework for automatic generation of multithreaded computer architecture simulators. In: *IEEE Computer Architecture Letters* 8 (2009), Nr. 1, S. 13–16. http://dx.doi.org/10.1109/L-CA.2009.2. – DOI 10.1109/L–CA.2009.2. – ISSN 15566056

[C. 10] C. JUNG AND THE GCC TEAM: *GCC, the GNU Compiler Collection, 4.5 Contribution*. http://gcc.gnu.org/. Version: nov 2010

[CCS+08] CENG, J. ; CASTRILLON, J. ; SHENG, W. ; SCHARWÄCHTER, H. ; LEUPERS, R. ; ASCHEID, Gerd ; MEYR, H. ; ISSHIKI, T. ; KUNIEDA, H.: Maps. In: *Proceedings of the 45th annual conference on Design automation - DAC '08*, 2008. – ISBN 9781605581156, 754

[CDCM09] CARARA, Everton A. ; DE OLIVEIRA, Roberto P. ; CALAZANS, N. L V. ; MORAES, Fernando G.: HeMPS - A framework for NoC-based MPSoC generation. In: *Proceedings - IEEE International Symposium on Circuits and Systems* (2009), 1345–1348. http://dx.doi.org/10.1109/ISCAS.2009.5118013. – DOI 10.1109/ISCAS.2009.5118013. – ISBN 9781424438280

[CG03] CAI, Lukai ; GAJSKI, Daniel D.: Transaction level modeling: an overview. In: *First IEEE/ACM/IFIP International Conference on Hardware/ Software Codesign and Systems Synthesis* (2003), 19–24. http://dx.doi.org/10.1109/CODESS.2003.1275250. – DOI 10.1109/CODESS.2003.1275250. ISBN 1–58113–742–7

[CGH+08] CONG, Jason ; GURURAJ, Karthik ; HAN, Guoling ; KAPLAN, Adam ; NAIK, Mishali ; REINMAN, Glenn: MC-sim: An efficient simulation tool for MPSoC designs. In: *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD* (2008), nov, 364–371. http://dx.doi.org/10.1109/ICCAD.2008.4681599. – DOI 10.1109/ICCAD.2008.4681599. – ISBN 9781424428205

[CHB09] CHEUNG, Eric ; HSIEH, Harry ; BALARIN, Felice: Fast and accurate performance simulation of embedded software for MPSoC. In: *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC* (2009), 552–557. http://dx.doi.org/10.1109/ASPDAC.2009.4796538. – DOI 10.1109/ASPDAC.2009.4796538. ISBN 9781424427482

[CMH+13] CHO, Hyoun K. ; MOSELEY, Tipp ; HANK, Richard ; BRUENING, Derek ; MAHLKE, Scott: Instant profiling: Instrumentation sampling for profiling datacenter applications. In: *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013* (2013). http://dx.doi.org/10.1109/

CGO.2013.6494982. – DOI 10.1109/CGO.2013.6494982. ISBN 9781467355254

[Com95] COMMITTEE, T I S.: *Tool Interface Standard ({TIS}) Executable and Linking Format ({ELF}) Specification Version 1.2*. Version: may 1995. http://pdos.csail.mit.edu/6.828/2012/readings/elf.pdf

[Des11] DESIGN SPACE EXPLORER: *Design Space Explorer*. https://www-asim.lip6.fr/trac/dsx. Version: 2011

[DZK+13] DING, Wei ; ZHANG, Yuanrui ; KANDEMIR, Mahmut ; SRINIVAS, Jithendra ; YEDLAPALLI, Praveen: Locality-aware mapping and scheduling for multicores. In: *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013* (2013), 1–12. http://dx.doi.org/10.1109/CGO.2013.6495009. – DOI 10.1109/CGO.2013.6495009. ISBN 9781467355254

[Ecl13] ECLIPSE FOUNDATION: *Eclipse Project*. http://www.eclipse.org. Version: 2013

[ECM11] ECMA INTERNATIONAL: *Standard ECMA-262: ECMAScript Language Specification*. http://www.ecma-international.org/publications/files/drafts/tc39-2009-025.pdfhttp://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf. Version: 2011

[FMSR09] FILHO, Julio O. ; MASEKOWSKY, Stephan ; SCHWEIZER, Thomas ; ROSENSTIEL, Wolfgang: CGADL: An architecture description language for coarse-grained reconfigurable arrays. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 17 (2009), Nr. 9, 1247–1259. http://dx.doi.org/10.1109/TVLSI.2008.2002429. – DOI 10.1109/TVLSI.2008.2002429. – ISSN 10638210

[FQS08] FUMMI, F. ; QUAGLIA, D. ; STEFANNI, F.: A systemC-based framework for modeling and simulation of networked embedded systems. In: *Proceedings - 2008 Forum on Specification, Verification and Design Languages, FDL'08* (2008), 49–54. http://dx.doi.org/10.1109/FDL.2008.4641420. – DOI 10.1109/FDL.2008.4641420. ISBN 9781424422654

[FS88] FENLASON, Jay ; STALLMAN, Richard: *GNU gprof*. 1988

[FYEM+13] FLOC'H, Antoine ; YUKI, Tomofumi ; EL-MOUSSAWI, Ali ; MORVAN, Antoine ; MARTIN, Kevin ; NAULLET, Maxime ; ALLE, Mythri ; L'HOURS, Ludovic ; SIMON, Nicolas ; DERRIEN, Steven ; CHAROT, François ; WOLINSKI, Christophe ; SENTIEYS, Olivier: GeCoS: A framework for prototyping custom hardware design flows. In: *IEEE 13th International Working Conference on Source Code Analysis and Manipulation, SCAM 2013* (2013), S. 100–105. http://dx.doi.org/10.1109/SCAM.2013.6648190. – DOI 10.1109/SCAM.2013.6648190. ISBN 9781467357395

[GAV+12] GOULAS, George ; ALEFRAGIS, Panayiotis ; VOROS, Nikolaos S. ; VALOUXIS, Christos ; GOGOS, Christos ; KAVVADIAS, Nikolaos ; DIMITROULAKOS, Grigoris ; MASSELOS, Kostas ; GOEHRINGER, Diana ; DERRIEN, Steven ; MENARD, Daniel ; SENTIEYS, Olivier ; HUEBNER, Michael ; STRIPF, Timo ; OEY, Oliver ; BECKER, Juergen ; RAUWERDA, Gerard ; SUNESEN, Kim ; KRITHARIDIS, Dimitrios ; MITAS, Nikolaos: From Scilab to multicore embedded systems: Algorithms and methodologies. In: *Proceedings - 2012 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, IC-SAMOS 2012* (2012), S. 268–275. http://dx.doi.org/10.1109/SAMOS.2012.6404184. – DOI 10.1109/SAMOS.2012.6404184. ISBN 9781467322973

[GHKG98] GRUN, Peter ; HALAMBI, A ; KHARE, A ; GANESH, V: EXPRESSION: An ADL for system level design ex-

ploration. In: *... of Information and ...* (1998). http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.36.8524{&}rep=rep1{&}type=pdf

[GKK+08] GAO, L. ; KARURI, K. ; KRAEMER, S. ; LEUPERS, R. ; ASCHEID, Gerd ; MEYR, H.: Multiprocessor performance estimation using hybrid simulation. In: *Proceedings - Design Automation Conference* (2008), S. 325–330. http://dx.doi.org/10.1109/DAC.2008.4555832. – DOI 10.1109/DAC.2008.4555832. – ISBN 9781605581156

[GKM82] GRAHAM, Susan L. ; KESSLER, Peter B. ; MCKUSICK, Marshall K.: Gprof: A call graph execution profiler. In: *ACM Sigplan Notices*, 1982, S. 120–126

[GM10] GREINER, Alain ; MELLO, Aline V.: *CABA/TLM-DT Transactors for the SoCLib virtual prototyping platform*. http://www.soclib.fr/trac/dev/wiki/WritingRules/Transactors. Version: 2010

[Gro02] GROTKER, Thorsten: *System Design with SystemC*. Norwell, MA, USA : Kluwer Academic Publishers, 2002. – ISBN 1402070721

[HBLH09] HUANG, Kai ; BACIVAROV, Iuliana ; LIU, Jun ; HAID, Wolfgang: A modular fast simulation framework for stream-oriented MPSoC. In: *Proceedings - 2009 IEEE International Symposium on Industrial Embedded Systems, SIES 2009*, 2009. – ISBN 9781424441105, 74–81

[Hei15] HEISSWOLF, Jan: *A Scalable and Adaptive Network on Chip for Many-Core Architectures*, Karlsuher Institut für Technologie, Dissertation, 2015

[HGG+99] HALAMBI, Ashok ; GRUN, Peter ; GANESH, Vijay ; KHARE, Asheesh ; DUTT, Nikil ; NICOLAU, Alex ; A. HALAMBI ET AL.: EX-

PRESSION: a language for architecture exploration through compiler/simulator retargetability. In: *Proceedings of Design, Automation and Test in Europe Conf. and Exhibition, 1999.* Dordrecht : IEEE Comput. Soc, 1999. – ISBN 0–7695–0078–1, 485–490

[HJB+13] HUANG, Jialu ; JABLIN, Thomas B. ; BEARD, Stephen R. ; JOHNSON, Nick P. ; AUGUST, David I.: Automatically exploiting cross-invocation parallelism using runtime information. In: *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013* (2013), 1–11. http://dx.doi.org/10.1109/CGO.2013.6495001. – DOI 10.1109/CGO.2013.6495001. ISBN 9781467355254

[HSW07] HUBERT, Heik ; STABERNACK, Benno ; WELS, Kai-Immo: Performance and memory profiling for embedded system design. In: *SIES'07. International Symposium on Industrial Embedded Systems, 2007.*, 2007, S. 94–101

[Hüb09] HÜBERT, Heiko: *MEMTRACE: A memory, performance and energy profiler targeting RISC-based embedded systems for data-intensive applications*, TU Berlin, Diss., 2009

[IEE06] IEEE: IEEE Standard for Verilog Hardware Description Language. In: *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (2006), S. 0,1–560. http://dx.doi.org/10.1109/IEEESTD.2006.99495. – DOI 10.1109/IEEESTD.2006.99495

[IEE09] IEEE: IEEE Standard VHDL Language Reference Manual. In: *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)* (2009), jan, S. c1–626. http://dx.doi.org/10.1109/IEEESTD.2009.4772740. – DOI 10.1109/IEEESTD.2009.4772740

[Imp08] IMPERAS SOFTWARE: *Open Virtual Platforms*. 2008

[Int96] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION: Information Technology - Syntactic Metalanguage - Extended BNF. In: *ISO/IEC 14977:1996* (1996)

[Jan04] JANTSCH, A.: *Modeling Embedded Systems and SoC's: Concurrency and Time in Models of Computation*. Morgan Kaufmann Publishers Inc., 2004. – ISBN 9781558609259

[KBB+12] KASSEM, Rola ; BRIDAY, Mikaël ; BÉCHENNEC, Jean L. ; SAVATON, Guillaume ; TRINQUET, Yvon: Harmless, a hardware architecture description language dedicated to real-time embedded system simulation. In: *Journal of Systems Architecture* 58 (2012), Nr. 8, 318–337. http://dx.doi.org/10.1016/j.sysarc.2012.05.001. – DOI 10.1016/j.sysarc.2012.05.001. – ISSN 13837621

[Kil76] KILBY, J.S.: Invention of the integrated circuit. In: *IEEE Transactions on Electron Devices* 23 (1976), jul, Nr. 7, S. 648–654. http://dx.doi.org/10.1109/T-ED.1976.18467. – DOI 10.1109/T–ED.1976.18467. – ISBN 0018–9383 VO – 23

[KR88] KERNIGHAN, Brian W. ; RITCHIE, Dennis M.: *The C Programming Language*. 2nd. Prentice Hall Professional Technical Reference, 1988. – ISBN 0131103709

[KSJB11] KÖNIG, Ralf ; STRIPF, Timo ; JAN, Heisswolf ; BECKER, Jürgen: A scalable microarchitecture design that enables dynamic code execution for variable-issue clustered processors. In: *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, 2011. – ISBN 9780769543857, S. 150–157

[LAR11] LUU, Jason ; ANDERSON, Jason H. ; ROSE, Jonathan S.: Architecture description and packing for logic blocks with hierarchy, modes and complex interconnect. In: *Proceedings of the 19th ACM/-SIGDA international symposium on Field programmable gate arrays - FPGA '11*, 2011. – ISBN 9781450305549, 227

[Lat13a] LATTNER, Chris: *LLVM Language Reference Manual*. http://llvm.org/docs/LangRef.html. Version: 2013

[Lat13b] LATTNER, Chris: *The LLVM Compiler Infrastructure*. Version: 2013. http://www.llvm.org

[LBV13] LANGLOIS, J.M. P. ; BOIS, Guy ; VAKILI, Shervin: Customised soft processor design: a compromise between architecture description languages and parameterisable processors. In: *IET Computers & Digital Techniques* 7 (2013), Nr. 3, 122–131. http://dx.doi.org/10.1049/iet-cdt.2012.0088. – DOI 10.1049/iet–cdt.2012.0088. – ISSN 1751–8601

[Lil27] LILIENFELD, Julius E.: *Electric Current Control Mechanism*. http://worldwide.espacenet.com/publicationDetails/biblio?CC=CA{&}NR=272437A{&}KC=A{&}FT=D{&}ND=1{&}date=19270719{&}DB={&}locale=de{_}EP. Version: 1927

[Lös12] LÖSCH, Achim: Untersuchung von Anwendungen auf Parallelität auf Thread- und Befehlssatzebenen für die Kahrisma Architektur. In: *Karlsruher Institut für Technologie* (2012)

[LPXL10] LI, Juncao ; PILKINGTON, Nicholas T. ; XIE, Fei ; LIU, Qiang: Embedded architecture description language. In: *Journal of Systems and Software* 83 (2010), Nr. 2, 235–252. http://dx.doi.org/10.1016/j.jss.2009.09.043. – DOI 10.1016/j.jss.2009.09.043. – ISBN 0164–1212

[MD05] MISHRA, P ; DUTT, N: Architecture description languages for programmable embedded systems. In: *IEE Proceedings-Computers and Digital Techniques*, 2005, 285–297

[MD08] MISHRA, Prabhat ; DUTT, Nikil: *Processor Description Languages*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2008. – ISBN 9780080558370, 9780123742872

[MEJ+12] MURILLO, Luis G. ; EUSSE, Juan ; JOVIC, Jovana ; YAKOUSHKIN, Sergey ; LEUPERS, Rainer ; ASCHEID, Gerd: Synchronization for hybrid MPSoC full-system simulation. In: *Proceedings of the 49th Annual Design Automation Conference on - DAC '12* (2012), 121. http://dx.doi.org/10.1145/2228360.2228383. – DOI 10.1145/2228360.2228383. – ISBN 9781450311991

[Mes09] MESSAGE PASSING INTERFACE FORUM: *{MPI}: A Message-Passing Interface Standard Version 2.2*. Version: sep 2009. http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf

[MKB+11] MOREIRA, João ; KLEIN, Felipe ; BALDASSIN, Alexandro ; CENTODUCATTE, Paulo ; AZEVEDO, Rodolfo ; RIGO, Sandro: Using multiple abstraction levels to speedup an MPSoC virtual platform simulator. In: *Proceedings of the International Workshop on Rapid System Prototyping* (2011), S. 99–105. http://dx.doi.org/10.1109/RSP.2011.5929982. – DOI 10.1109/RSP.2011.5929982. – ISBN 9781457706585

[MMGP10] MELLO, A ; MAIA, I ; GREINER, A ; PECHEUX, F: Parallel simulation of systemC TLM 2.0 compliant MPSoC on SMP workstations. In: *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, 2010. – ISSN 1530–1591, S. 606–609

[Moo65] MOORE, Gorden E.: Cramming More Components onto Integrated Circuits. In: *Electronics* 38 (1965), apr, Nr. 8

[Moo11] MOORE, Chuck: Data processing in exascale-class computer systems. In: *The Salishan Conference on High Speed Computing*, 2011

[Mra09] MRABTI, Amin E.: Extending IP-XACT to support an MDE based approach for SoC design. In: *Design, Automation & Test ...* (2009), 586–589. http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=5090733. – ISBN 9783981080155

[MSR+07] MOSELEY, Tipp ; SHYE, Alex ; REDDI, Vijay J. ; GRUNWALD, Dirk ; PERI, Ramesh: Shadow profiling: Hiding instrumentation costs with parallelism. In: *International Symposium on Code Generation and Optimization, CGO 2007* (2007), S. 198–208. http://dx.doi.org/10.1109/CGO.2007.35. – DOI 10.1109/CGO.2007.35. ISBN 0769527647

[Mwm09] MWMRCOPROCCREATION: *MwmrCoprocCreation*. https://www-asim.lip6.fr/trac/dsx/wiki/MwmrCoprocCreation. Version: 2009

[NBGS08] NICKOLLS, John ; BUCK, Ian ; GARLAND, Michael ; SKADRON, Kevin: Scalable Parallel Programming with CUDA. In: *Queue* 6 (2008), März, Nr. 2, 40–53. http://dx.doi.org/10.1145/1365490.1365500. – DOI 10.1145/1365490.1365500. – ISSN 1542–7730

[NS03] NETHERCOTE, Nicholas ; SEWARD, Julian: Valgrind: A program supervision framework. In: *Electronic notes in theoretical computer science* 89 (2003), Nr. 2, S. 44–66

[NS07] NETHERCOTE, Nicholas ; SEWARD, Julian: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: *ACM Sigplan notices* 42 (2007), Nr. 6, S. 89–100

[Obj15] OBJECT MANAGEMENT GROUP (OMP): *Unified Modeling Language (UML) Version 2.5*. 2015

[Ope09] OPEN SYSTEMC INITIATIVE: *TLM-2.0 Language Reference Manual*. 2009

[Ora15] ORACLE: *Java Language Specification: Java SE 8 Edition*. feb 2015

[PEP06] PIMENTEL, Andy D. ; ERBAS, Cagkan ; POLSTRA, Simon: A systematic approach to exploring embedded system architectures at

multiple abstraction levels. In: *IEEE Transactions on Computers* 55 (2006), Nr. 2, S. 99–111. http://dx.doi.org/10.1109/TC.2006.16. – DOI 10.1109/TC.2006.16. – ISBN 0018–9340

[PH08] PATTERSON, David A. ; HENNESSY, John L.: *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. 4th. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2008 http://www.google.com/books?id=3b63x-0P3{_}UC{&}printsec=frontcover{%}255Cnhttp://portal.acm.org/citation.cfm?id=1502247. – ISBN 0123744938, 9780123744937

[PLCG95] PILLET, Vincent ; LABARTA, Jes{\'u}s ; CORTES, Toni ; GIRONA, Sergi: Paraver: A tool to visualize and analyze parallel code. In: *Proceedings of WoTUG-18: transputer and occam developments*, 1995, S. 17–31

[PMGP10] PESSOA, Isaac M. ; MELLO, Aline ; GREINER, Alain ; PÊCHEUX, François: Parallel TLM simulation of MPSoC on SMP workstations: Influence of communication locality. In: *Proceedings of the International Conference on Microelectronics, ICM*, 2010. – ISBN 9781612841519, S. 359–362

[Pyt13] PYTHON SOFTWARE FOUNDATION: *Python*. http://www.python.org. Version: 2013

[QD11] QIAN, Hao ; DENG, Yangdong: Accelerating RTL simulation with GPUs. In: *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD* (2011), 687–693. http://dx.doi.org/10.1109/ICCAD.2011.6105404. – DOI 10.1109/ICCAD.2011.6105404. – ISBN 9781457713989

[RABA04] RIGO, Sandro ; ARAUJO, Guido ; BARTHOLOMEU, Marcus ; AZEVEDO, Rodolfo: ArchC: A SystemC-based architecture description language. In: *16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2004)*, 2004. – ISBN 0769522408, S. 66–73

[RAC13] RAKOSSY, Zoltan E. ; APONTE, Axel A. ; CHATTOPADHYAY, Anupam: Exploiting architecture description language for diverse IP synthesis in heterogeneous MPSoC. In: *2013 International Conference on Reconfigurable Computing and FPGAs, ReConFig 2013* (2013), 1–6. http://dx.doi.org/10.1109/ReConFig.2013.6732287. – DOI 10.1109/ReConFig.2013.6732287. – ISBN 9781479920792

[Rec17] RECORE SYSTEMS: *Xentium architecture*. http://www.recoresystems.com/products/reliable-data-processing-in-space/xentium-vliw-dsp-ip-core/, 2017

[RKB+10] R. KOENIG ET AL. ; 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: *2010 Design, Automation & Test in Europe Conf. & Exhibition (DATE 2010)*, Ieee, 2010. – ISBN 978–3–9810801–6–2, 819–824

[Rup16] RUPP, Karl: *40 Years of Microprocessor Trend Data*. https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/. https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/. Version: jun 2016

[San11] SANDMANN, Timo: KAHRISMA - Entwicklung eines ADL-basierten systemweiten mehrprozessor Simulators. In: *Karlsruher Institut für Technologie* (2011)

[SBP12] SANDRIESER, Martin ; BENKNER, Siegfried ; PLLANA, Sabri: Using explicit platform descriptions to support programming of heterogeneous many-core systems. In: *Parallel Computing* 38 (2012), Nr. 1-2, 52–65. http://dx.doi.org/10.1016/j.parco.2011.10.008. – DOI 10.1016/j.parco.2011.10.008. – ISBN 9781450305778

[Sci13] SCILAB ENTERPRISES: *Scilab*. http://www.scilab.org. Version: 2013

[Sew04] SEWARD, J AND NETHERCOTE, N AND FITZHARDINGE, J: *Cachegrind: a cache-miss profiler*. 2004

[SGS10] STONE, John E. ; GOHARA, David ; SHI, Guochun: OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. In: *IEEE Des. Test* 12 (2010), Mai, Nr. 3, 66–73. http://dx.doi.org/10.1109/MCSE.2010.69. – DOI 10.1109/MCSE.2010.69. – ISSN 0740–7475

[SHN+02] SCHLIEBUSCH, O. ; HOFFMANN, A. ; NOHL, A. ; BRAUN, G. ; MEYR, H.: Architecture implementation using the machine description language LISA. In: *Proceedings of ASP-DAC/VLSI Design 2002. 7th Asia and South Pacific Design Automation Conference and 15h International Conference on VLSI Design*, 2002. – ISBN 0–7695–1441–3, 239–244

[SKB11] STRIPF, Timo ; KOENIG, Ralf ; BECKER, Juergen: A novel ADL-based compiler-centric software framework for reconfigurable mixed-ISA processors. In: *Proceedings - 2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, IC-SAMOS 2011*, 2011. – ISBN 9781457708008, S. 157–164

[SKB12] STRIPF, Timo ; KÖNIG, Ralf ; BECKER, Jürgen: A cycle-approximate, mixed-ISA simulator for the KAHRISMA architecture. In: *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2012. – ISBN 978–1–4577–2145–8, 21–26

[SKRB12] STRIPF, Timo ; KÖNIG, Ralf ; RIEDER, Patrick ; BECKER, Jürgen: A compiler back-end for reconfigurable, mixed-ISA processors with clustered register files. In: *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2012*, 2012. – ISBN 9780769546766, S. 462–469

[Sle11] SLEEPY: *Sleepy*. 2011

[SoC11] SOCLIB: *SoClib*. `http://www.soclib.fr`. Version: 2011

[SRAA11] SANTOS, Luiz ; RIGO, Sandro ; AZEVEDO, Rodolfo ; ARAUJO, Guido: *Electronic System Level Design*. Springer-Verlag, 2011. – 3–10 S. `http://dx.doi.org/10.1007/978-1-4020-9940-3`. `http://dx.doi.org/10.1007/978-1-4020-9940-3`. – ISBN 978–1–4020–9939–7

[SRO+16] STRIPF, Timo ; RIAR, Frederik ; OEY, Oliver ; RÜCKAUER, Michael ; BECKER, Jürgen: *emmtrix*. `http://www.emmtrix.com/`. Version: 2016

[Str00] STROUSTRUP, Bjarne: *The C++ Programming Language*. 3rd. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2000. – ISBN 0201700735

[Str13] STRIPF, Timo: *Softwareframework für Prozessoren mit variablen Befehlssatzarchitekturen*, Karlsruher Institut für Technologie, Diss., 2013. `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000043813`

[Sys12] SYSTEM C STANDARDIZATION WORKING GROUP: *1666-2011 - {IEEE Standard for Standard SystemC Language Reference Manual}*. Version: 2012. `http://standards.ieee.org/getieee/1666/download/1666-2011.pdf`

[Ver11] VERYSLEEPY: *VerySleepy*. 2011

[VGS+10] VENTROUX, N. ; GUERRE, A. ; SASSOLAS, T. ; MOUTAOUKIL, L. ; BLANC, G. ; BECHARA, C. ; DAVID, R.: SESAM: An MPSoC simulation environment for dynamic application processing. In: *Proceedings - 10th IEEE International Conference on Computer and Information Technology, CIT-2010, 7th IEEE International Conference on Embedded Software and Systems, ICESS-2010, ScalCom-2010*, Ieee, 2010. – ISBN 9780769541082, 1880–1886

[WB13] WOO, Mason ; BRUKMAN, Misha: *Writing an {LLVM} Compiler Backend*. Version: 2013. http://www.llvm.org/docs/WritingAnLLVMBackend.html

[WD96] WALKER, DW ; DONGARRA, JJ: MPI: a standard message passing interface. In: *Supercomputer* 12 (1996), 56–68. http://users.cs.cf.ac.uk/David.W.Walker/papers/supercomputer96.ps. – ISBN 0168–7875

[WDK+04] WIEFERINK, A. ; DOERPER, M. ; KOGEL, T. ; LEUPERS, R. ; ASCHEID, Gerd ; MEYR, H.: Early iss integration into network-on-chip designs. In: *Lecture notes in computer science* (2004), 443–452. http://www.springerlink.com/index/EKQD5YYLJP9LELC4.pdf. – ISBN 3–540–22377–0

[WH07] WALLACE, Steven ; HAZELWOOD, Kim: SuperPin: Parallelizing dynamic instrumentation for real-time performance. In: *International Symposium on Code Generation and Optimization, CGO 2007* (2007), S. 209–217. http://dx.doi.org/10.1109/CGO.2007.37. – DOI 10.1109/CGO.2007.37. ISBN 0769527647

[Wkh10] WKH, D Q G.: *IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tools Flows*. IEEE, 2010. – C1–360 S. http://dx.doi.org/10.1109/IEEESTD.2010.5417309. http://dx.doi.org/10.1109/IEEESTD.2010.5417309. – ISBN 9780738161594

[WMP+08] WALTERS, Edward K. ; MOSS, J. E B. ; PALMER, Trek ; RICHARDS, Timothy ; WEEMS, Charles C.: CASL: A rapid-prototyping language for modern micro-architectures. In: *Computer Languages, Systems and Structures* 34 (2008), Nr. 4, S. 195–211. http://dx.doi.org/10.1016/j.cl.2007.06.001. – DOI 10.1016/j.cl.2007.06.001. – ISBN 1477–8424

[Wor08] WORLD WIDE WEB CONSORTIUM: *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. Version: nov 2008. http://www.w3.org/TR/2008/REC-xml-20081126/

Eigene Veröffentlichungen

[ALB+13] ALMEIDA, Gabriel M. ; LONGHI, Oliver B. ; BRUCKSCHLÖGL, Thomas ; HÜBNER, Michael ; HESSEL, Fabiano ; BECKER, Jürgen: Simplify: A framework for enabling fast functional/behavioral validation of multiprocessor architectures in the cloud. In: *Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium Workshops and PhD Forum, IPDPSW 2013* (2013), S. 2200–2205

[BBO+14] BECKER, Jürgen ; BRUCKSCHLÖGL, Thomas ; OEY, Oliver ; STRIPF, Timo ; GOULAS, George ; RAPTIS, Nick ; VALOUXIS, Christos ; ALEFRAGIS, Panayiotis ; VOROS, Nikolaos S. ; GOGOS, Christos: Profile-Guided compilation of scilAb algorithms for multiprocessor systems. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* Bd. 8405 LNCS. 2014, S. 330–336

[BOR+14] BRUCKSCHLÖGL, Thomas ; OEY, Oliver ; RÜCKAUER, Michael ; STRIPF, Timo ; BECKER, Jürgen: A hierarchical architecture description for flexible multicore system simulation. In: *Proceedings - 2014 IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA 2014* (2014), S. 190–196

[DBF+13] DELICIA, G Shalina P. ; BRUCKSCHLÖGL, Thomas ; FIGULI, Peter ; TRADOWSKY, Carsten ; MARCHESAN, Gabriel ; BECKER, Jürgen: Bringing Accuracy to Open Virtual Platforms (OVP): A Safari from High-Level Tools to Low-Level Microarchitectures. (2013), S. 22–27

[FHG+11] FIGULI, Peter ; HÜBNER, Michael ; GIRARDEY, Romuald ; BAPP, Falco ; BRUCKSCHLÖGL, Thomas ; THOMA, Florian ; HENKEL, Jörg ; BECKER, Jürgen: A heterogeneous SoC architecture with embedded virtual FPGA cores and runtime Core Fusion. In: *Proceedings of the 2011 NASA/ESA Conference on Adaptive Hardware and Systems, AHS 2011* (2011), S. 96–103

[GVA+13] GOULAS, George ; VALOUXIS, Christos ; ALEFRAGIS, Panayiotis ; VOROS, Nikolaos S. ; GOGOS, Christos ; OEY, Oliver ; STRIPF, Timo ; BRUCKSCHLÖGL, Thomas ; BECKER, Jürgen ; EL MOUSSAWI, Ali ; NAULLET, Maxime ; YUKI, Tomofumi: Coarse-grain optimization and code generation for embedded multicore systems. In: *Proceedings - 16th Euromicro Conference on Digital System Design, DSD 2013* (2013), S. 379–386

[SOB+12a] STRIPF, Timo ; OEY, Oliver ; BRUCKSCHLÖGL, Thomas ; KOENIG, Ralf ; BECKER, Jürgen ; GOULAS, George ; ALEFRAGIS, Panayiotis ; VOROS, Nicolaos ; POTMAN, Jordy ; SUNESEN, Kim ; DERRIEN, Steven ; SENTIEYS, Olivier: A Compilation-and Simulation-Oriented Architecture Description Language for Multicore Systems. In: *2012 IEEE 15th International Conference on Computational Science and Engineering (CSE)* (2012), S. 383–390

[SOB+12b] STRIPF, Timo ; OEY, Oliver ; BRUCKSCHLÖGL, Thomas ; KÖNIG, Ralf ; HÜBNER, Michael ; BECKER, Jürgen ; RAUWERDA, Gerard ; SUNESEN, Kim ; KAVVADIAS, Nikolaos ; DIMITROULAKOS, Grigoris ; MASSELOS, Kostas ; KRITHARIDIS, Dimitrios ; MITAS, Nikolaos ; GOULAS, George ; ALEFRAGIS, Panayiotis ; VOROS, Nikolaos S. ; DERRIEN, Steven ; MENARD, Daniel ; SENTIEYS, Olivier ;

GOEHRINGER, Diana ; PERSCHKE, Thomas: A flexible approach for compiling scilab to reconfigurable multi-core embedded systems. In: *ReCoSoC 2012 - 7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip, Proceedings* (2012)

[SOB+13] STRIPF, Timo ; OEY, Oliver ; BRUCKSCHLÖGL, Thomas ; BECKER, Jürgen ; RAUWERDA, Gerard ; SUNESEN, Kim ; GOULAS, George ; ALEFRAGIS, Panayiotis ; VOROS, Nikolaos S. ; DERRIEN, Steven ; SENTIEYS, Olivier ; KAVVADIAS, Nikolaos ; DIMITROULAKOS, Grigoris ; MASSELOS, Kostas ; KRITHARIDIS, Dimitrios ; MITAS, Nikolaos ; PERSCHKE, Thomas: Compiling Scilab to high performance embedded multicore systems. In: *Microprocessors and Microsystems* 37 (2013), S. 1033–1049

[TGB+14] TRADOWSKY, Carsten ; GÄDEKE, Tobias ; BRUCKSCHLÖGL, Thomas ; STORK, Wilhelm ; MÜLLER-GLASER, Klaus D. ; BECKER, Jürgen: SmartLoCore: A concept for an adaptive power-aware localization processor. In: *Proceedings - 2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2014* (2014), S. 478–481

Betreute studentische Arbeiten

[BHK16] BEN HAJ KHALIFA, Saifeddine: *PRP-Kommunikationsplattform auf XMOS-Controller*, Karlsruher Institut für Technologie, Masterarbeit Nr. 2079, Januar 2016

[Brä13] BRÄHLER, Stefan: *Entwurf und Implementierung eines parametrierbaren MPSoC Simulators auf Basis einer ADL*, Karlsruher Institut für Technologie, Diplomarbeit Nr. 1695, Juli 2013

[GS16] GUIAKAM SITSO, Adeline: *Modellierung und Analyse eingebetteter CPU/GPU Architekturen*, Karlsruher Institut für Technologie, Masterarbeit, November 2016

[Hus17] HUSSEIN, Ahmad: *Hardwaremodellierung und Systemiteration für die Entwurfsraumexploration eingebetteter Systeme*, Karlsruher Institut für Technologie, Bachelorarbeit Nr. 2236, May 2017

[Kem14] KEMPF, Fabian: *Erweiterung und Evaluierung eines LEON3 Simulators für Multicore Simulationen*, Karlsruher Institut für Technologie, Bachelorarbeit Nr. 1878, August 2014

[PB14] PAZMINO BETANCOURT, Victor H.: *Integration eines Network-on-Chip in das ALMA-Simulationsframework*, Karlsruher Institut für Technologie, Bachelorarbeit Nr. 1891, November 2014

[PB16] PAZMINO BETANCOURT, Victor H.: *Modellierung eingebetteter Software Funktionen für heterogene eingebettete Systeme*, Karlsruher Institut für Technologie, Masterarbeit Nr. 2177, November 2016

[Sch16] SCHMIDT, Michael: *Modellierung und Implementierung von Sicherheitsmaßnahmen in OPC UA Netzwerken*, Karlsruher Institut für Technologie, Masterarbeit Nr. 2131, Juni 2016