Co-Konfiguration von Hardware- und Systemsoftware-Produktlinien

Dissertation

zur Erlangung des Grades eines

Doktors der Ingenieurwissenschaften

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Matthias Meier

Dortmund

2017

Tag der mündlichen Prüfung:	15. März 2017
Dekan:	Prof. Dr.-Ing. Gernot Fink
Gutachter:	Prof. Dr.-Ing. Olaf Spinczyk
	Prof. Dr.-Ing. Jörg Nolte

Danksagungen

An erster Stelle möchte ich meinem Betreuer Prof. Olaf Spinczyk danken, dass er mir die Gelegenheit gegeben hat, in diesem interessanten und facettenreichen Forschungsgebiet zu arbeiten und mich in den Jahren, in denen diese Arbeit entstanden ist, mit Ratschlägen und Ideen unterstützt hat. Ebenso geht ein großer Dank an Dr. Michael Engel, der mich bereits während meiner Diplomarbeit begleitet und mir damit auch thematisch den Weg zu dieser Arbeit eröffnet hat. Ebenfalls danken möchte ich Prof. Jörg Nolte, dass er sich bereit erklärt hat, als Gutachter für diese Arbeit zur Verfügung zu stehen.
Ein besonderer Dank geht auch an Hendrik Borghorst und Boguslaw Jablkowski für die unzähligen fachlichen Diskussionen, die Unterstützung und die kurzweiligen Gespräche in „Abteilung E09“. Auch den weiteren Kollegen der Arbeitsgruppe *Embedded System Software* Christoph Borchert, Markus Buschhoff, Ulrich Gabor, Alexander Lochmann, Orwa Nassour, Dr. Horst Schirmeier und Jochen Streicher gilt mein herzlicher Dank für die langjährige Unterstützung und die schöne Zeit.
Ebenso möchte ich an dieser Stelle den zahlreichen Studenten herzlich danken, die ihre Abschlussarbeit im Kontext des LavA-Projektes verfasst haben und mit ihrem Einsatz das Projekt bereichert haben. Ein spezieller Dank geht dabei an Mark Breddemann, der das LavA-Projekt beinahe fünf Jahre als studentische Hilfskraft begleitet und zu dessen Gelingen beigetragen hat.
Auch meiner Familie gilt mein herzliches Dankeschön, denn ohne deren stetige Unterstützung und dem mir gegebenen Rückhalt, wäre diese Arbeit nicht möglich gewesen.
Des Weiteren möchte ich der Deutschen Forschungsgemeinschaft (DFG) für die Ermöglichung dieser Arbeit durch die Förderung des Forschungsprojektes „LavA - Laufzeitplattform für anwendungsspezifische verteilte Architekturen“ (Kennzeichen SP 968/4-1 und SP 968/4-2) danken, in dessen Rahmen ein großer Teil dieser Arbeit entstanden ist.

Zusammenfassung

Hardwarearchitekturen im Kontext von Eingebetteten Systemen werden immer komplexer und bewegen sich zukünftig immer häufiger in Richtung von *Multi-* oder *Manycore*-Systemen. Damit diese Systeme ihre optimale Leistungsfähigkeit – für die oftmals speziellen Aufgaben im Kontext von Eingebetteten Systemen – ausspielen können, beschäftigen sich ganze Forschungszweige mit der anwendungsspezifischen Maßschneiderung dieser Systeme. Insbesondere die Popularität von Hardwarebeschreibungssprachen trägt dazu ihren Teil bei. Jedoch ist die Entwicklung von solchen Systemen, selbst bei der Verwendung von Hardwarebeschreibungssprachen und der damit verbundenen höheren Abstraktionsebene, aufwendig und fehleranfällig.

Die Verwendung von Hardwarebeschreibungssprachen lässt allerdings die Grenze zwischen Hard- und Software verschwimmen, denn Hardware kann nun – ähnlich wie auch Software – in textueller Form beschrieben werden. Dies eröffnet Möglichkeiten zur Übertragung von Konzepten aus der Software- auf die Hardwareentwicklung. Ein Konzept um der wachsenden Komplexität im Bereich der Softwareentwicklung zu begegnen, ist die organisierte Wiederverwendung von Komponenten, wie sie in der Produktlinienentwicklung zum Einsatz kommt. Inwieweit sich Produktlinienkonzepte auf Hardwarearchitekturen übertragen lassen und wie Hardware-Produktlinien entworfen werden können, soll in dieser Arbeit detailliert untersucht werden. Die Vorteile der Produktlinientechniken, wie die Möglichkeit zur Wiederverwendung von erprobten und zuverlässigen Komponenten, könnten so auch für Hardwarearchitekturen genutzt werden, um die Entwicklungskomplexität zu reduzieren und so mit erheblich geringerem Aufwand spezifische Hardwarearchitekturen entwickeln zu können. Zudem kann durch die gemeinsame Codebasis einer Produktlinie eine schnellere Markteinführungszeit unter geringeren Entwicklungskosten realisiert werden.

Auf Basis dieser neuen Konzepte beschäftigt sich diese Arbeit zudem mit der Fragestellung, wie zukünftig solche parallelen Systeme programmiert und automatisiert optimiert werden können, um den Entwickler von der Anwendung über die Systemsoftware bis hin zur Hardware mit einer automatisierten Werkzeugkette bei der Umsetzung zu unterstützen. Im Fokus stehen dabei die in dieser Arbeit entworfenen Techniken zur durchgängigen Konfigurierung von Hardware und Systemsoftware. Diese Techniken beruhen im Wesentlichen auf den Programmierschnittstellen zwischen den Schichten, deren Zugriffsmuster sich statisch analysieren lassen. Die so gewonnenen Konfigurationsinformationen lassen sich dann zur automatisierten Maßschneiderung der Systemsoftware- und Hardware-Produktlinie für ein spezifisches Anwendungsszenario nutzen.

Die anwendungsspezifische Optimierung der Systeme wird in dieser Arbeit mittels einer Entwurfsraumexploration durchgeführt. Der Fokus der Entwurfsraumexplo-

ration liegt allerdings nicht allein auf der Hardwarearchitektur, sondern umfasst ebenso die Softwareebene. Denn neben der Maßschneiderung der Systemsoftware, wird auch die auf einer parallelen Programmierschnittstelle aufsetzende Anwendung innerhalb der Entwurfsraumexploration automatisch skaliert, um die Leistungsfähigkeit von *Manycore*-Systemen ausschöpfen zu können.

Publikationen

Teile dieser Dissertation wurden zuvor in Journalen oder in Tagungsbänden von Konferenzen oder Workshops veröffentlicht (in chronologischer Reihenfolge).

1. Matthias Meier. Merkmalbasierte statische Konfigurierung von MPSoCs. Diplomarbeit, Technische Universität Dortmund, May 2009.

2. Matthias Meier, Michael Engel, Matthias Steinkamp, and Olaf Spinczyk. LavA: An open platform for rapid prototyping of MPSoCs. In *Proceedings of the 20th International Conference on Field Programmable Logic and Applications (FPL '10)*, pages 452–457, Milano, Italy, August 2010. IEEE Computer Society Press.

3. Matthias Meier, David Austin, Horst Schirmeier, and Olaf Spinczyk. TMPL: A hardware transactional memory product line. In *Proceedings of the Workshop on Multiprocessor Systems on (Programmable) Chips (MPSoC '11)*, Istanbul, Turkey, July 2011. IEEE Computer Society Press.

4. Matthias Meier and Olaf Spinczyk. LavA: Model-driven development of configurable MPSoC hardware structures for robots. In *Proceedings of the Workshop on Software Language Engineering for Cyber-physical Systems (WS4C '11)*, Lecture Notes in Informatics, Berlin, Germany, October 2011. German Society of Informatics.

5. Matthias Meier, Stefan Hanenberg, and Olaf Spinczyk. AspectVHDL Stage 1: The prototype of an aspect-oriented hardware description language. In *Proceedings of the 2nd Workshop on Modularity in Systems Software (MISS '12)*, pages 3–8, Potsdam, Germany, March 2012. ACM.

6. Matthias Meier, Mark Breddemann, and Olaf Spinczyk. Hardware APIs: A software-centric approach for automated derivation of MPSoC hardware structures based on static code analysis. In *Proceedings of the 27th GI/ITG International Conference on Architecture of Computing Systems (ARCS '14)*, volume 8350 of *Lecture Notes in Computer Science*, pages 111–122, Lübeck, Germany, February 2014. Springer-Verlag.

7. Matthias Meier, Mark Breddemann, and Olaf Spinczyk. Interfacing the hardware API with a feature-based operating system family. *Journal of Systems Architecture*, 61(10):531–538, November 2015.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Inhalt

Eingebettete Systeme nehmen einen immer wichtigeren Stellenwert in Forschung und Entwicklung ein und werden in einer Vielzahl von verschiedenen Anwendungsgebieten eingesetzt. So finden sich Eingebettete Systeme heutzutage vielfach in unserer täglichen Umgebung wieder, wie zum Beispiel in Kraftfahrzeugen, in der Unterhaltungselektronik, in Fertigungsanlagen der Industrie oder in medizinischen Geräten. Neben der steigenden Verbreitung von Eingebetteten Systemen, wächst zudem auch deren Komplexität immer weiter an. Die heutzutage erreichte Integrationsdichte erlaubt nicht nur die Unterbringung von einzelnen Prozessoren auf einem Chip, sondern den Entwurf und die Entwicklung von *Multi-* und *Many-core*-Systemen auf einem einzigen Chip. Die Verwendung solcher Systeme ist auch im Bereich der Eingebetteten Systeme ein immer stärker zu beobachtender Trend. Die OMAP 4 Prozessoren von Texas Instruments werden beispielsweise überwiegend in mobilen Endgeräten eingesetzt und vereinen neben zwei ARM Cortex-A9 und zwei ARM Cortex-M3 Vielzweckprozessoren noch viele spezialisierte Komponenten auf einem Chip, wie zum Beispiel Grafikeinheit oder digitaler Signalprozessor. Durch die gegebene Parallelität sowie Heterogenität dieser Systeme kann nicht nur die Rechenleistung gesteigert werden, sondern wegen der integrierten Vielzweckprozessoren bieten diese Systeme auch die nötige Flexibilität, um bereits bestehende Softwareprojekte zu portieren oder auch neue Funktionalitäten durch einfache Software-Updates nachträglich zu realisieren. Des Weiteren kann durch die Verwendung von *Multi-* und *Manycore*-Systemen auf einem Chip eine höhere Rechenleistung pro Takt erreicht werden, da insbesondere aufgrund von thermalen und energetischen Einschränkungen bei Eingebetteten Systemen, der Takt engeren, physikalischen Grenzen unterlegen ist und nicht beliebig erhöht werden kann.

Zudem verschwimmen die Grenzen zwischen Hard- und Software, denn durch den Einsatz von Hardwarebeschreibungssprachen (engl. *Hardware Description Language*, HDL) wie VHDL [RS13] oder Verilog [TM98] kann Hardware, ähnlich wie Software, über textuelle Beschreibungen formuliert werden. Dabei kann die Hardware nicht nur auf Registertransferebene (engl. *Register Transfer Level*, RTL) beschrieben, verifiziert oder simuliert werden, sondern auch mittels Synthesewerkzeugen auf Gatterebene überführt und letztendlich auf einem CPLD (*Complex Programmable Logic Device*), FPGA (*Field-Programmable Gate Array*) oder ASIC (*Application-Specific Integrated Circuit*) eingesetzt werden. Hochsprachen, wie VHDL oder Verilog, werden zudem immer populärer, denn sie heben die Hardwareentwicklung auf eine höhere Abstraktionsebene, um den aktuellen Trend, der steigenden Komplexität bei digitalen Schaltungen und die immer kürzer werdenden Zeitspannen bis zur Markteinführung, bewältigen zu können.
Neben einfachen digitalen Schaltungen, lassen sich mittels Hardwarebeschreibungssprachen auch komplexe Systeme, wie die zuvor genannten *Multi-* oder *Manycore*-Systeme, entwerfen. Dabei können die Beschreibungen nicht nur die Prozessorkerne selbst umfassen, sondern ein vollständiges System abbilden, welches sowohl Kommunikationsstrukturen als auch Peripheriegeräte umfassen kann. Aufgrund der textuellen Beschreibung des Systems kann dieses wie Software beliebig verändert und an gegebene anwendungsspezifische Anforderungen angepasst werden. Zum Beispiel durch die Variation der Anzahl der Prozessorkerne, dem Austausch der verwendeten Kommunikationsstruktur oder dem Hinzufügen von Peripheriegeräten, wie zum Beispiel einem UART[1]-Controller. Aktuelle High-End-FPGAs wie das Virtex UltraSCALE+ von Xilinx werden bereits mit einer Strukturbreite von 16 nm [Xil15] gefertigt und erlauben so die Integration von Systemen mit weit über 100 Prozessorkernen[2] auf einem Chip.
Dieser hohe Grad an Flexibilität auf der Hardwareebene hat auch Auswirkungen auf die darüberliegende Software. So muss zum Beispiel die Systemsoftware mit solch einer beliebig konfigurierbaren und mit einem potenziell hohen Grad an Parallelität versehenen Hardwareplattform umgehen können. Dabei ziehen sich die Auswirkungen, welche eine konfigurierbare Hardwareplattform hinterlässt, nahezu durch alle Schichten der Systemsoftware. Angefangen bei der Hardwareabstraktionsschicht (engl. *Hardware Abstraction Layer*, HAL), welche bei heterogenen Systemen an die jeweiligen Architekturen angepasst werden muss, über die einzelnen Treiber für optionale Peripheriegeräte, bis hin zu Bibliotheken zur Kommunikation, welche auf die zur Verfügung gestellten Kommunikationsstrukturen abgestimmt werden müssen. Bei Nichtbeachtung dieser Probleme besteht sonst immer die Gefahr von Inkonsistenzen zwischen den Schichten.
Ein weiteres Problem auf Softwareebene stellt im Falle von *Multi-* und *Manycore*-Systemen die große Anzahl von Prozessorkernen dar, welche bei der Entwicklung einer Anwendung auch sinnvoll ausgenutzt werden müssen, damit eine hoch-parallelisierte Hardwareplattform auch den entsprechenden Nutzen einbrin-

[1] *Universal Asynchronous Receiver/Transmitter*
[2] MB-Lite+: `http://ens.ewi.tudelft.nl/~huib/vhdl/mb-lite_plus.php`

gen kann. Beispielsweise müssen Mechanismen zur Synchronisation und Kommunikation zwischen den parallelen Kontrollflüssen zur Verfügung stehen, um solche Systeme effizient und schnell programmieren zu können. Solche Programmiermodelle sind insbesondere aus dem Bereich der Hochleistungsrechner (engl. *High-Performance Computing*, HPC) bekannt, wo die Programmierung von großen Rechenclustern bereits Alltag ist. Dort weit verbreitete Vertreter dieser Programmiermodelle sind MPI [Mes12] (*Message Passing Interface*) und PVM [GBD+94] (*Parallel Virtual Machine*) auf Seite der nachrichtenbasierten Modelle und OpenMP [CDK+01] (*Open Multi-Processing*) auf Seite der Modelle, welche auf gemeinsamem Speicher basieren. Jedoch sind – im Gegensatz zu Anwendungen für Hochleistungsrechner – Anwendungen im Bereich der Eingebetteten Systeme anderen Anforderungen, bezüglich Ressourcen, Energie und Echtzeit, unterworfen. Zudem haben diese oftmals einen eher statischen Charakter, da die meisten, wenn nicht sogar alle, zu bearbeitenden Tasks bereits zur Entwicklung bekannt sind [Mar11]. Auf solch ein statisches Task-Modell setzen auch OSEK/AUTOSAR-konforme [AUT14] Betriebssysteme auf, welche vor allem für echtzeitkritische Systeme in Kraftfahrzeugen eingesetzt werden.

1.1 Motivation

Die Entwicklung von Hardware ist trotz Hardwarebeschreibungssprachen und deren höherer Abstraktionsebene noch immer extrem aufwendig, komplex und fehleranfällig, speziell durch den hohen Grad an gleichzeitig ablaufenden Prozessen in einem Hardwaredesign und der beschwerlichen Suche nach Fehlerquellen in den parallelen Abläufen mittels Simulatoren. Damit ungeachtet dessen, die heutigen, immer kürzer werdenden Produktzyklen eingehalten werden können, ist es von fundamentaler Bedeutung, die Entwicklung mit flexiblen *Workflows* und Methodiken zur Wiederverwendung von Komponenten zu unterstützen. In der Softwareentwicklung wird schon seit längerem das Konzept der Produktlinien angewendet. Die Produktlinienentwicklung wird dabei als „organisierte Wiederverwendung und organisierte Variabilität“ [BKPS04] beschrieben. Dabei beschränkt sich die Verwendung der Produktlinienkonzepte nicht allein auf die Entwicklung von Anwendungssoftware, sondern wird auch für Systemsoftware eingesetzt. Denn gerade bei relativ leichtgewichtigen Betriebssystemen für Eingebettete Systeme [LHSP+09, Mas02] wird das Produktlinienkonzept inzwischen angewandt, um das Betriebssystem mit einem möglichst geringen Overhead bezüglich anwendungs- und hardwarespezifischer Anforderungen maßschneidern zu können.
Da sich mittels Hardwarebeschreibungssprachen auch Hardware textuell beschreiben lässt, ist zu untersuchen, ob sich die bekannten Techniken aus der Software-Produktlinienentwicklung auch auf die Entwicklung von Hardware übertragen lassen und sich so maßgeschneiderte *Multi-* und *Manycore*-Systeme unter Wiederverwendung von erprobten und zuverlässigen Hardwarekomponenten systematisch entwickeln lassen. Neben der erhöhten Zuverlässigkeit der Systeme, könnten Vorteile – welche bereits durch den Einsatz von Produktlinien aus der Softwarewelt

bekannt sind – auch für die Hardwareentwicklung genutzt werden, wie zum Beispiel schnellere Markteinführungszeiten oder geringere Entwicklungskosten durch die gemeinsame Codebasis für verschiedene Ausprägungen der Produktlinie. Des Weiteren kann die Abstraktionsebene zur Repräsentation der Hardware weiter angehoben werden, wenn die wiederzuverwendenden Hardwarekomponenten zum Beispiel auf der Ebene von Prozessoren, Kommunikationsstrukturen oder Peripheriegeräten und deren Eigenschaften angesiedelt sind. So lassen sich nicht nur die Details der Hardwareimplementierung verbergen, sondern es kann die Produktlinie auch völlig losgelöst von Hardwarebeschreibungssprachen verwendet werden. Durch den hohen Abstraktionsgrad und die FPGA-Technologie können *Multi-* bzw. *Manycore*-Systeme so auch von Entwicklern konfiguriert und synthetisiert werden, welche nicht auf das Hardwaredesign spezialisiert sind.
Produktlinien unterstützen zwar den Entwickler bei der raschen Erstellung von maßgeschneiderten und zuverlässigen Systemen, aber die bereits angesprochenen Abstimmungsprobleme zwischen der Systemsoftware- und der Hardwareschicht lassen sich so nicht verlässlich ausschließen. Denn jegliche Fehlkonfigurierung während der Entwicklung kann dazu führen, dass die Schichten nicht mehr harmonieren und das gesamte System, im ungünstigsten Fall, nicht mehr lauffähig ist. So könnte Beispielsweise ein Peripheriegerät eine Unterbrechung auslösen, für welches bei einem Betriebssystem während des Konfigurationsprozesses keine entsprechende Behandlung registriert wurde.

1.2 Ansatz

In dieser Arbeit wird deshalb ein Ansatz mit dem Namen LavA[3] (Laufzeitplattform für anwendungsspezifische verteilte Architekturen) vorgestellt, welcher die automatisierte Co-Konfiguration von Hardware- und Systemsoftware-Produktlinien verfolgt. Dabei handelt es sich um den in Abbildung 1.1 dargestellten softwarezentrierten Konfigurationsprozess. Dieser Konfigurationsprozess konfiguriert und generiert, für eine vom Softwareentwickler vorgegebene Anwendung, automatisiert die passend abgestimmte Systemsoftware sowie Hardware. Die bei diesem Ansatz gewählte Systemsoftware, ist das leichgewichtige und hochkonfigurierbare Betriebssystem CiAO [LHSP+09]. „Softwarezentriert“ beschreibt für diesen Konfigurationsprozess, wo die Informationen zur automatisierten Konfigurierung der Schichten gewonnen werden – hier mittels statischer Codeanalyse aus den Programmierschnittstellen (engl. *Application Programming Interface*, API) zwischen den Schichten. Bei diesem Ansatz wird ausgenutzt, dass die Zugriffsmuster auf eine Programmierschnittstelle oftmals Rückschlüsse auf die benötigten Subsysteme der darunter liegenden Schicht zulassen und die gewonnenen Informationen so zur Maßschneiderung verwendet werden können. Typischerweise können sogar die meisten Merkmale zur Konfigurierung einer Software-

[3] LavA ist ein von der Deutschen Forschungsgemeinschaft (DFG) im Zeitraum von 2009 bis 2015 gefördertes Projekt, in dessen Kontext diese Arbeit entstanden ist.

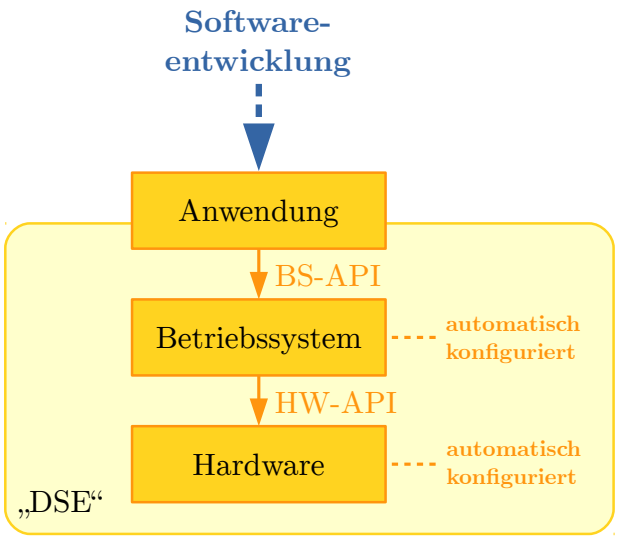


Abbildung 1.1: LavAs softwarezentriertes Modell zur Co-Konfiguration von Betriebssystem und Hardware

Produktlinie aus den Zugriffsmustern abgeleitet werden [SS07]. Dabei ist es jedoch entscheidend, dass der bereits konfigurierte Quelltext aus der oberhalb liegenden Softwareschicht (Softwareinstanz) untersucht wird und nicht der ursprüngliche Quelltext (Produktliniencode). Mit dem aus der jeweiligen Schicht extrahierten Konfigurationswissen, kann dann die Betriebssystem- und Hardwareschicht anwendungsspezifisch maßgeschneidert werden. Das Konfigurationswissen wird so, ausgehend von der Anwendung, durch die einzelnen Schichten gereicht. Dieser durchgängige Konfigurationsprozess sorgt zum einen dafür, dass sowohl Software als auch Hardware einheitlich mit Produktlinientechniken maßgeschneidert werden und zum anderen, dass keine Fehlkonfigurationen zwischen den Schichten auftreten können, da beispielsweise kein kommerzielles Standardbetriebssystem im Nachhinein zwischen den Schichten integriert werden muss.
Merkmale, welche sich nicht anhand von Zugriffsmustern ableiten lassen, wie beispielsweise die Anzahl der Prozessorkerne, die Zuteilung der Tasks zu diesen oder die Wahl der passenden Kommunikationsstruktur, werden über eine multikriterielle Entwurfsraumexploration (engl. *Design Space Exploration*, DSE) variiert. Am Ende des LavA-Prozesses fällt ein anwendungsspezifisch maßgeschneidertes System heraus, welches sich aus Anwendung, Betriebssystem und Hardware zusammensetzt und nach vorgegebenen Kriterien, wie der Menge der zur Verfügung stehenden Hardwareressourcen und zeitlichen Kriterien, optimiert ist.

1.3 Gliederung

In diesem Abschnitt wird ein kurzer Überblick über die Struktur der Arbeit und den Inhalt der folgenden Kapitel gegeben.

Kapitel 2: In diesem Kapitel werden zunächst die benötigten Grundlagen für diese Arbeit vermittelt. Der erste Teil dieses Kapitels beschäftigt sich deshalb mit der Thematik, wie generische Hardwarekomponenten mittels Hardwarebeschreibungssprachen entworfen werden können und wie aus den einfachen textuellen Beschreibungen ein Hardwaredesign entsteht, welches auf einem FPGA genutzt werden kann. Da die begrenzten Ressourcen eines FPGAs ein Hardwaredesign erheblich einschränken können, sind die FPGA-Ressourcen auch im Verlauf dieser Arbeit ein immer wiederkehrendes Thema, weshalb die einzelnen Bausteine eines FPGAs kurz eingeführt werden. Der zweite Teil dieses Kapitels wechselt schließlich von den Hardware verwandten Themen hin zur Software und diskutiert, sowohl die aktuellen Konzepte als auch die Begrifflichkeiten aus der Software-Produktlinienentwicklung.

Kapitel 3: In Kapitel 3 knüpft dann eine Diskussion zum Stand der Forschung im Kontext dieser Arbeit an. Nachdem die diversen Konzepte aus der Produktlinienentwicklung bereits im vorherigen Kapitel präsentiert wurden, rücken in diesem Kapitel vorrangig andere Forschungsansätze zur Generierung von Multiprozessorsystemen und deren Vergleich mit dem in dieser Arbeit angestrebten Ansatz in den Fokus der Betrachtung. Des Weiteren wird in diesem Kapitel untersucht, in welchem Umfang die Systemsoftware in den Konfigurierungsprozessen von anderen Forschungsbeiträgen integriert ist und welche Techniken zur Optimierung der Multiprozessorsysteme in diesen Ansätzen eingesetzt werden.

Kapitel 4: Dieses Kapitel beschäftigt sich mit dem Thema der Hardware-Produktlinien und untersucht dazu im Wesentlichen, inwieweit sich die verschiedenen Konzepte aus der Software-Produktlinien Entwicklung auch auf Hardware übertragen lassen. Darüber hinaus werden in diesem Kapitel Ansätze vorgestellt, um generische und wiederverwendbare Hardwarekomponenten zu implementieren, damit die erforderliche Variabilität für die zahlreichen Varianten einer Produktlinie auch umgesetzt werden kann. Parallel zu diesen Untersuchungen, wird auf Basis der gewonnenen Erkenntnisse und anhand eines typischen Prozesses zur Entwicklung von Software-Produktlinien, die LavA Hardware-Produktlinie vorgestellt und diskutiert.

Kapitel 5: Der Schwerpunkt dieses Kapitels liegt auf dem softwarezentrierten Konfigurationsprozess und erläutert, wie die Gewinnung von Konfigurationsinformationen anhand von Zugriffen auf die Programmierschnittstellen zwischen den Schichten dazu verwendet werden kann, um die jeweils tieferen Schichten automatisiert zu generieren und auf die Anforderungen der oberen Schicht abzustimmen. Hierbei wird insbesondere die Schicht zwischen Systemsoftware und Hardware beleuchtet. Zudem wird diskutiert, welche Vorteile und Möglichkeiten diese enge Kopplung dieser beiden Schichten über die entworfene Schnittstelle mit sich bringt. In dem letzten Teil dieses Kapitels wird zudem eine Programmierschnittstelle vorgestellt, welche, wie bereits in Abbildung 1.1 gezeigt, die Ver-

bindung zwischen der Anwendung und der Systemsoftware herstellt und einem Anwendungsentwickler bei der Entwicklung von komplexen *Multi-* und *Many-core*-Systemen unterstützen soll.

Kapitel 6: Die in den vorherigen Kapiteln beschriebenen Methodiken zur Co-Konfiguration von Systemsoftware und Hardware bilden für dieses Kapitel die Grundlage, um auf Basis einer multikriteriellen Entwurfsraumexploration die Systeme automatisiert sowie anwendungsgetrieben zu optimieren. Neben der Entwurfsraumexploration wird in diesem Kapitel insbesondere diskutiert, wie die von der Anwendung zur Verfügung gestellten Informationen – in Form der Zugriffsmuster auf die Schnittstelle – dazu genutzt werden können, um den Optimierungsprozess zu leiten und damit ein optimiertes, anwendungsspezifisches System, bestehend aus Systemsoftware und Hardware, zu generieren. Darüber hinaus beschäftigt sich dieses Kapitel mit der Fragestellung, wie solche komplexen Systeme bezüglich wichtiger Kriterien, wie der Leistungsfähigkeit oder dem Ressourcenbedarf, bewertet werden können.

Kapitel 7: Dieses Kapitel betrachtet anhand von verschiedenen Anwendungsbeispielen die erzielten Ergebnisse des automatisierten LavA-Optimierungsprozesses. Neben der Qualität der Ergebnisse, wird auch deren Plausibilität beleuchtet und damit gezeigt, dass der vorgestellte Ansatz vielversprechende Ergebnisse liefert.

Kapitel 8: In dem letzten Kapitel wird diese Arbeit schließlich zusammengefasst und ein Ausblick für mögliche Anknüpfungspunkte sowie weitere interessante Forschungsfragen im Kontext der anwendungsspezifischen Co-Konfiguration von Systemsoftware und Hardwarearchitektur aufgeworfen und mögliche Lösungsansätze diskutiert.

1.4 Forschungsbeitrag dieser Arbeit

Der Fokus dieser Arbeit liegt auf der Untersuchung von neuen Methodiken, um die Co-Konfiguration von Hardware und Systemsoftware für komplexe *Multi-* und *Manycore*-Systeme zu erleichtern und den aktuellen Stand der Forschung in dieser Hinsicht zu verbessern. Um dieses Ziel zu erreichen, setzt diese Arbeit an verschiedenen Stellen des Konfigurationsprozesses an und verbindet die diversen vorgestellten Methodiken zu einem durchgängigen Optimierungsprozess, welcher die ansonsten komplizierte, zeitaufwendige und fehleranfällige Entwicklung solcher Systeme unterstützen bzw. automatisieren soll.
Eines der zentralen Paradigmen dieses Ansatzes, ist die Verwendung von statischen Analysen zur unmittelbaren Extraktion von Konfigurationsinformationen aus den Zugriffsmustern der Schnittstellen zwischen den Schichten. Diese Arbeit zeigt, dass es möglich ist, dieses Schema – ausgehend von der Anwendung über

die Systemsoftware bis hin zur Hardware – anzuwenden, um die Schichten unterhalb der Anwendung automatisiert zu konfigurieren, aufeinander abzustimmen und zu optimieren. Damit auch die Hardware anhand von Zugriffsmustern aus der untersten Softwareschicht abgeleitet werden kann, wird in dieser Arbeit eine entsprechende Lösung in Form der sogenannten Hardware-API vorgestellt und diskutiert.

Aufgrund der schrittweisen Konfigurierung von der obersten bis zur untersten Schicht, ist bei dieser Arbeit auch die Systemsoftware explizit in den Konfigurierungsprozess eingebunden und Fehlkonfigurationen zwischen Systemsoftware und Hardware können bereits durch den vorgeschlagenen Prozess selbst vermieden werden. Zudem eröffnet die angestrebte, enge Kopplung von Systemsoftware und Hardware mittels der Hardware-API, eine neue Sicht der Systemsoftware auf die Hardware.

Da in dieser Arbeit Systeme mit einer großen Anzahl von Prozessorkernen im Mittelpunkt stehen, wird darüber hinaus diskutiert, wie solche parallelen und hochkonfigurierbaren Systeme von einem Anwendungsentwickler effektiv programmiert werden können. Um dieses Problem zu lösen wird eine parallele Programmierschnittstelle vorgestellt. Diese Schnittstelle bietet nicht nur eine Verbindung zwischen der Anwendung sowie der Systemsoftware, sondern ermöglicht, wie die Hardware-API, die Extraktion von Konfigurationsinformationen mittels statischer Anwendungsanalysen. Damit stellt die parallele Programmierschnittstelle für einen Entwickler zugleich eine zugängliche Schnittstelle zum Optimierungsprozess zur Verfügung, da der Entwickler die benötigten Informationen nur implizit über die Nutzung von API-Funktionen zur Verfügung stellen muss.

Ein weiterer Schwerpunkt dieser Arbeit ist die Fragestellung, mit welchen Methodiken sich konfigurierbare Familien von Hardwarearchitekturen konzipieren und entwickeln lassen. In der Softwareentwicklung gibt es bereits vielzählige Forschungsbeiträge, welche sich mit dieser Fragestellung im Kontext der Software beschäftigen. In dieser Arbeit wird deshalb untersucht, inwiefern die dort erlangten Erkenntnisse auch für Hardware gelten oder ob andere Methodiken notwendig sind. Im Rahmen dieser Untersuchungen ist unter anderem die Spracherweiterung AspectVHDL entstanden, welche Konzepte aus der Aspektorientierten Programmierung in VHDL bereitstellt.

1.5 Beitrag des Autors zu dieser Dissertation

Gemäß §10 Absatz 2 der „Promotionsordnung der Fakultät für Informatik der Technischen Universität Dortmund vom 29. August 2011“ müssen in einer Dissertation, soweit der Autor wissenschaftliche Ergebnisse verwendet, die in Kooperation mit anderen Wissenschaftlern entstanden sind, die Eigenanteile an diesen Ergebnissen aufgelistet werden. In den folgenden Abschnitten werden diese Anteile kapitelweise beschrieben. Bei allen aufgeführten, begutachteten Publikationen [MESS10, MASS11, MS11, MHS12, MBS14, MBS15] ist der Autor dieser Arbeit

der Erstautor. Prof. Olaf Spinczyk trug zu diesen Publikationen mit technischen Diskussionen, kurzen Textfragmenten und Korrekturlesen bei.

Kapitel 4: Die in diesem Kapitel präsentierte Untersuchung zur Übertragbarkeit von Software-Produktlinientechniken auf Hardwarearchitekturen wurde von dem Autor dieser Arbeit durchgeführt. Das Konzept, zu der als Studie verwendeten transaktionalen Speicherfamilie, stammt von Prof. Olaf Spinczyk und dem Autor. Die zugehörige Implementierung wurde von dem ehemaligen Diplomanden David Austin [Aus10] umgesetzt und vom Autor technisch und inhaltlich begleitet. Die Speicherfamilie wurde im Anschluss von dem Autor dieser Arbeit überarbeitet und evaluiert, woraus die Publikation [MASS11] entstanden ist. Die Mitautoren steuerten Textfragmente zu dieser Publikation bei. Die Plattform der LavA Hardware-Produktlinie, mit ihrer Variabilität und der Möglichkeit zur Generierung von maßgeschneiderten Systemen, wurde von dem Autor dieser Arbeit konzipiert, implementiert und in der Publikation [MESS10] veröffentlicht. Der ehemalige Diplomand Matthias Steinkamp trug zu dieser Publikation mit einer frühen Implementierung zur modellgetriebenen Konfigurierung der LavA Hardware-Produktlinie bei. Die Mitautoren unterstützten zudem mit technischen Ratschlägen und Korrekturlesen. Einige der von dem Autor in die LavA Hardware-Produktlinie integrierten Hardwarekomponenten, wie beispielsweise die Prozessoren, stammen aus Open-Source-Projekten. Die ursprüngliche Vision zur Verwendung von Aspekten in Hardware stammt von Prof. Olaf Spinczyk und Dr. Michael Engel. Diese Idee wurde von dem Autor konkretisiert und die Spracherweiterung AspectVHDL entworfen sowie die zugehörige Erweiterung der VHDL-Grammatik formuliert [MHS12]. Dr. Stefan Hanenberg hat zu dieser Publikation mit seiner Expertise im Bereich der empirischen Studien beigetragen.

Kapitel 5: Die Idee zur anwendungsspezifischen Konfigurierung von Hardware mittels statischer Analysen ist eines der Grundkonzepte des LavA-Projektes und stammt von Prof. Olaf Spinczyk. Eine erste Variante dieses Konzeptes wurde von Matthias Steinkamp im Kontext seiner Diplomarbeit [Ste10] implementiert und von dem Autor dieser Arbeit begleitet. Die folgenden Weiterentwicklungen durch den Autor dieser Arbeit mündeten letztlich in der Hardware-API und wurden in [MBS14] publiziert. Die Konzepte zur Kopplung der Hardware-API mit dem konfigurierbaren Betriebssystem CiAO wurden von dem Autor entwickelt und in der Publikation [MBS15] veröffentlicht. Die Mitautoren der beiden Publikationen [MBS14, MBS15] unterstützten durch Anmerkungen und Korrekturlesen. LavAs paralleles Programmiermodell wurde initial von dem Autor in Kooperation mit dem ehemaligen Bachelorstudenten Sebastian Struwe entworfen [Str13]. Im Anschluss an diese Bachelorarbeit wurde das Programmiermodell von dem Autor dieser Arbeit überarbeitet und in die hier präsentierte Form gebracht. Eine erste prototypische Implementierung dieses Modells wurde in der Bachelorarbeit von Timo Cramer [Cra14] umgesetzt.

Kapitel 6: Der LavA-Optimierungsprozess wurde vollständig von dem Autor dieser Arbeit entworfen und implementiert. Ebenso stammt die Untersuchung der verschiedenen Techniken zur Bewertung der Performanz im Kontext von *Multi-* und *Manycore*-Systemen von dem Autor. Die Veröffentlichung [MS11] von dem Autor dieser Arbeit diskutierte mögliche Anwendungsbereiche der LavA Hardware-Produktlinie und thematisierte zudem die Ressourcenmodelle.

Kapitel 7: Die Evaluierung des LavA-Optimierungsprozesses wurde allein von dem Autor dieser Arbeit durchgeführt.

Kapitel 2

Grundlagen

Inhalt

In diesem Kapitel werden die für den weiteren Verlauf der Arbeit benötigten Grundlagen vermittelt. Zunächst wird in Abschnitt 2.1 diskutiert, wie Hardware mittels Beschreibungssprachen spezifiziert werden kann, welche Methodiken zur Maßschneiderung Hardwarebeschreibungssprachen „von Haus aus“ bieten und welche Möglichkeiten es gibt, um von einer einfachen, textuellen Beschreibung eines Hardwaredesigns zu einem fertigen Chip zu gelangen. In dem folgenden Abschnitt 2.2 werden dann Produktlinientechniken aus der Softwareentwicklung und Forschung näher vorgestellt.

2.1 Konfigurierung von Hardware

Hardwarebeschreibungssprachen sind der „Türöffner“, um Produktlinientechniken aus der Softwareentwicklung auch für die Entwicklung von Hardware einzusetzen. Durch die hier gegebene textuelle Beschreibung der Hardware, lassen sich nahezu beliebige Hardwaredesigns beschreiben, verändern und maßschneidern. Wie aber aus einer einfachen textuellen Beschreibung ein fertiger Chip in Form von CPLDs, FPGAs oder ASICs, konfiguriert bzw. gefertigt werden kann, soll nun in diesem Kapitel behandelt werden.

2.1.1 Generische Designs mit Hardwarebeschreibungssprachen

Die verbreitetsten Hardwarebeschreibungssprachen sind VHDL und Verilog. Verilog ist im amerikanischen Raum sehr dominant, wohingegen sich VHDL in Europa

```
1 entity halfadd is
2     port(x      : in  bit;
3          y      : in  bit;
4          sum    : out bit;
5          carry  : out bit);
6 end halfadd;
```

Abbildung 2.1: Schnittstellenbeschreibung eines Halbaddierers

```
1 architecture behavioral of halfadd is
2 begin
3     sum   <= ((not x) and y) or (x and (not y));
4     carry <= x and y;
5 end behavioral;
```

Abbildung 2.2: Architekturbeschreibung eines Halbaddierers

am stärksten durchsetzen konnte [TH13]. Man geht sogar davon aus, „dass in naher Zukunft über 70% aller Elektronik-Designs mit Hilfe von VHDL beschrieben bzw. definiert werden“ [KM07]. Mit beiden Sprachen lassen sich alle Möglichkeiten, welche konfigurierbare Hardware bietet, auch ausnutzen. Unterschiede gibt es aber zum Beispiel bei der Typisierung oder bei der Kompaktheit der beiden Sprachen. VHDL ist stark getypt und fordert eher ausführlichen Quelltext, wohingegen Verilog genau gegenteilig veranlagt ist. Da die in dieser Arbeit entwickelten und verwendeten Hardwaredesigns auf VHDL basieren, wird dementsprechend der Fokus auf die generische Hardwareentwicklung mittels VHDL gelegt. Andere Sprachen die im Kontext der Hardwarebeschreibung oder der Hardwareverifikation Verwendung finden, wie SystemC [Kes12], SystemVerilog [SDF06] oder „e“ [Pal04], werden in dieser Arbeit nicht im Detail behandelt.
In VHDL sind die Hardwaredesigns aus verschiedenen Komponenten aufgebaut, welche eine hierarchische Struktur aufweisen. Die Hierarchie entsteht dadurch, dass die Komponenten wiederum Subkomponenten instanziieren können und sich so eine baumartige Struktur ergibt. Im Folgenden soll nun erläutert werden, wie in VHDL modulare und generische Komponenten modelliert und beschreiben werden können.

2.1.1.1 Aufbau von Hardwarekomponenten

Die Beschreibung der Hardwarekomponenten in VHDL ist zweigeteilt: zum einen die Schnittstellenbeschreibung und zum anderen die Architekturbeschreibung. Die Schnittstellenbeschreibung definiert die ein- und ausgehenden Signale einer Hardwarekomponente zu ihrer Umgebung. In Abbildung 2.1 ist die Schnittstellenbeschreibung für einen Halbaddierer dargestellt. Dieser Baustein besitzt zwei eingehende und zwei ausgehende Signale vom Typ `bit`, dabei kann der Datentyp `bit` die Werte `0` und `1` annehmen. Die Signale `x` und `y` sollen von dem Bau-

```
1  entity fulladd is
2      port(x       : in  bit;
3           y       : in  bit;
4           carry_i: in  bit;
5           sum     : out bit;
6           carry_o: out bit);
7  end fulladd;
8
9  architecture behavioral of fulladd is
10     signal sum0, c0, c1 : bit;
11 begin
12     ha0: halfadd port map(x=>x,y=>y,sum=>sum0,carry=>c0);
13     ha1: halfadd port map(x=>sum0,y=>carry_i,sum=>sum,carry=>c1);
14     carry_o <= c0 or c1;
15 end behavioral;
```

Abbildung 2.3: Beschreibung eines Volladdierers mittels zweier Halbaddierer

stein addiert und die berechneten Ergebnisse, die Summe (`sum`) und der Übertrag (`carry`), sollen über die ausgehenden Signale weitergeleitet werden. Die Architekturbeschreibung – der zweite Teil einer VHDL Hardwarekomponente – beschreibt das Verhalten der Komponente, im Falle des Halbaddierers also die Berechnung der Summe und des Übertrags für die Eingaben `x` und `y`. Die Architektur des Halbaddierers wird in Abbildung 2.2 gezeigt. Das eigentliche Verhalten des Halbaddieres zur Berechnung ist in den Zeilen 3 und 4 beschrieben. Hier werden die in VHDL zur Verfügung stehenden logischen Operatoren `not`, `and` und `or` verwendet. Dem Signal `sum` wird mittels des Zuweisungsoperators (`<=`) das Ergebnis der Entweder-Oder-Verknüpfung von `x` und `y` zugewiesen und das Signal `carry` erhält den Wert aus der Und-Verknüpfung von `x` und `y`. Das hier beschriebene Verhalten wird dabei nicht sequentiell, Zeile für Zeile abgearbeitet, wie typischerweise bei Sprachen aus der Softwareentwicklung, sondern wird parallel von der Hardware ausgewertet.

2.1.1.2 Wiederverwendung von Hardwarekomponenten

VHDL erlaubt einen modularen Aufbau des Hardwaredesigns, um bereits beschriebene Hardwarebausteine effizient wiederverwenden zu können. Die Funktionsweise soll nun anhand eines Volladdierers demonstriert werden, welcher sich aus zwei Halbaddierern zusammensetzt. Die Schnittstellenbeschreibung eines Volladdierers ähnelt stark der eines Halbaddierers. Die Beschreibung des Volladdierers in Abbildung 2.3 zeigt, dass nur ein weiteres eingehendes Signal zur Schnittstellenbeschreibung eines Volladdierers hinzukommt. Dieses dient zum Einlesen des Übertrags aus der Berechnung des angrenzenden, niederwertigeren Bits. In der Architekturbeschreibung werden in Zeile 10 weitere Signale zur Nutzung innerhalb des Volladdierers deklariert. Der interessante Aspekt bezüglich der Wiederverwendung der Halbaddierer ist in den Zeilen 12 und 13 zu finden,

denn hier werden zwei Halbaddierer instanziiert und verwendet. Bei der Instanziierung wird im Wesentlichen über das *Port-Mapping* festgelegt, wie die ein- und ausgehenden Signale der beiden Halbaddierer mit den Signalen des Volladdierers verknüpft werden sollen. So kann der Halbaddiererbaustein gewissermaßen als Blackbox wiederverwendet werden, ohne dass das Verhalten des Halbaddierers neu beschrieben werden müsste. In Zeile 14 wird abschließend dem ausgehenden Übertrag des Volladdierers das Ergebnis der Oder-Verknüpfung der beiden Überträge der Halbaddierer zugewiesen.
Mit diesem Vorgehen lassen sich auch wesentlich komplexere Bausteine, wie zum Beispiel Peripheriegeräte oder Prozessoren, wiederverwenden, um so ganze *Multi-* oder *Manycore*-Systeme zusammenzufügen. Der Verdrahtungsaufwand, gerade bei komplexeren Bausteinen mit vielen Ein- und Ausgangssignalen, ist jedoch nicht zu unterschätzen und kann, insbesondere bei der Verknüpfung „per Hand", zu Fehlern in dem Hardwaredesign führen. In einem VHDL-Design können neben den selbst implementierten Komponenten auch Fremdkomponenten, sogenannte *IP-Cores* (*Intellectual Property Cores*), eingebunden werden. Diese können dabei entweder als quelloffener VHDL-Quelltext oder als bereits synthetisierte Netzlisten vorliegen.

2.1.1.3 Generische Hardwarekomponenten

Neben der vorgestellten Wiederverwendung von Komponenten können Hardwarekomponenten auch generisch beschrieben werden. Der hierfür zentrale Mechanismus in VHDL heißt *Generic*, mit welchem sich Komponenten parametrisieren lassen. Um diesen Mechanismus zu erläutern, werden die bisher entwickelten Volladdiererbausteine nun zu einem Addierwerk, genauer gesagt zu einem Ripple-Carry-Addierer, verschaltet. Das Addierwerk soll sich jedoch bezüglich der Anzahl der Bit-Stellen parametrisieren lassen, beispielsweise um dieses in verschiedenen Projekten für 16 oder 32 Bit Zahlen verwenden zu können. Das in Abbildung 2.4 dargestellte Addierwerk erlaubt genau so eine Parametrisierung, denn die Schnittstellenbeschreibung verfügt, neben der bereits bekannten Beschreibung der ein- und ausgehenden Signale, über den generischen Parameter `width` (Zeile 2). Der generische Parameter `width` besitzt den Standardwert `4`, welcher aber während der Instanziierung des Ripple-Carry-Addierers durch eine andere Komponente entsprechend überschrieben werden kann. Schon in der Schnittstellenbeschreibung wird der generische Parameter `width` zur Parametrisierung der ein- und ausgehenden Signale verwendet. So wird der Datentyp `bit_vector`, welcher für die Signale `x`, `y` und `sum` verwendet wird, entsprechend der angestrebten Bit-Stellen angepasst. Im Falle des Standardwertes von `4` würden die Bitvektoren also aus einem Bus von vier Signalen bestehen.
Damit analog zu der Anzahl der Bit-Stellen auch die richtige Menge an Volladdierern instanziiert wird, wird in der Architekturbeschreibung das Konstrukt `Generate` verwendet (Zeile 15). `Generate` ermöglicht die iterative Instanziierung von Komponenten mit fester aber beliebiger Anzahl. Dazu werden in den Zei-

```
1  entity ripplecarryadd is
2      generic(width    : integer := 4);
3      port(x           : in  bit_vector(width - 1 downto 0);
4           y           : in  bit_vector(width - 1 downto 0);
5           carry_i     : in  bit;
6           sum         : out bit_vector(width - 1 downto 0);
7           carry_o     : out bit);
8  end ripplecarryadd;
9
10 architecture behavioral of ripplecarryadd is
11     signal carry: bit_vector(width downto 0);
12 begin
13     carry(0) <= carry_i;
14     carry_o  <= carry(width);
15     for i in 0 to width - 1 generate
16         fa: fulladd port map(x=>x(i),y=>y(i),carry_i=>carry(i),
17                              sum=>sum(i),carry_o=>carry(i+1));
18     end generate;
19 end behavioral;
```

Abbildung 2.4: Architekturbeschreibung eines generischen *Ripple-Carry*-Addierers

len 15 bis 18 unter Einsatz der *For*-Schleife entsprechend viele Volladdierer zum Zeitpunkt der Synthese instanziiert und untereinander verdrahtet, bis die durch den generischen Parameter `width` bestimmte Abbruchbedingung erreicht wird. Neben der hier gezeigten Einsatzmöglichkeit lassen sich *Generics* auch dazu verwenden, um zum Beispiel Busbreiten zu konfigurieren oder optionale Beschleuniger in Komponenten zu aktivieren, welche lediglich für spezifische Aufgabenfelder benötigt werden und sonst nur unnötig Ressourcen auf der konfigurierbaren Hardware belegen würden.

2.1.2 FPGA-Technologie

Für den Weg von der Hardwarebeschreibungssprache zum fertigen Chip stehen im Wesentlichen zwei Technologien auf dem Markt zur Verfügung, um anwendungsspezifische *Multi-* oder *Manycore*-Systeme umzusetzen: ASICs und FPGAs. ASICs sind integrierte Schaltungen, welche anwendungsspezifisch gefertigt werden müssen. Nach der Fertigung ist die Funktionsweise dieser nicht mehr änderbar. ASICs eignen sich in erster Linie für die Fertigung von Großserien. Denn je nach Strukturbreite kann allein der Maskensatz zur Fertigung von ASICs oberhalb der Marke von einer Millionen US-Dollar liegen [KB13]. Wegen der folgenden, relativ geringen Stückkosten pro ASIC, können bei einer entsprechend hohen Anzahl an gefertigten Chips die anfänglichen Fixkosten allerdings amortisiert werden. Inzwischen gibt es jedoch auch Anbieter[1], die unterschiedliche Projekte von verschiedenen Kunden auf einem *Wafer* fertigen. So lassen sich die hohen

[1] Circuits Multi-Projects: `http://cmp.imag.fr/`

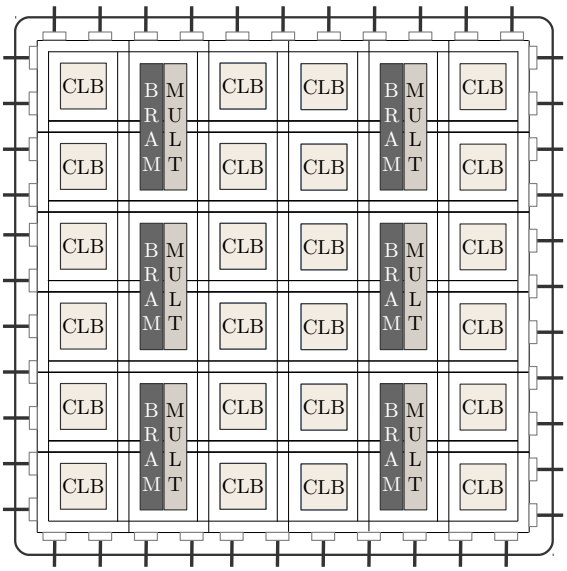


Abbildung 2.5: Grundstruktur eines FPGAs (vereinfacht)

Kosten für Maskensätze auf mehrere Kunden verteilen, wodurch ASICs auch für Kleinserien in Betracht gezogen werden können. Bei FPGAs hingegen handelt es sich prinzipiell um standardisierte, integrierte Schaltungen, welche nicht speziell für den Kunden angepasst werden müssen. Die anwendungsspezifische Anpassung übernimmt bei FPGAs der Anwender selbst, indem er die im FPGA zur Verfügung gestellte konfigurierbare Logik und die konfigurierbaren Verbindungsstrukturen programmiert bzw. konfiguriert. FPGAs haben zwar höhere Stückkosten als ASICs, lassen sich jedoch direkt nach dem Kauf für das jeweilige Vorhaben verwenden. Die meisten FPGAs bieten zudem die Möglichkeit zur Rekonfigurierung der Logik und der Verbindungsstrukturen, sodass das Hardwaredesign an neue Anforderungen angepasst werden kann oder Probleme mit dem Design im Nachhinein behoben werden können. Aufgrund der schnellen Einsatzbereitschaft und der Flexibilität durch die Rekonfigurierung, eignen sich diese insbesondere für die Erstellung von Prototypen, denn gerade bei Prototypen wiegen die negativen Eigenschaften von FPGAs, wie hohe Stückkosten, erhöhte Leistungsaufnahme oder größere Chipfläche, weniger schwer als bei dem Einsatz im letztendlichen Produkt für den Markt. Hardwarebeschreibungen, welche zunächst für die prototypische Implementierung auf FPGAs entwickelt wurden, können allerdings nach entsprechenden manuellen Anpassungen später auch für ASICs verwendet werden [Mar11].

FPGAs lassen sich grob in die Varianten mit flüchtigem und nichtflüchtigem Speicher zur Unterbringung der Konfiguration gruppieren. Exemplare mit flüchtigem Speicher (SRAM-basierende FPGAs) müssen bei jedem Start des FPGAs erneut mit den Konfigurationsinformationen versorgt werden, wenn das FPGA zuvor stromlos war. Dies geschieht zum Beispiel über einen externen oder einen

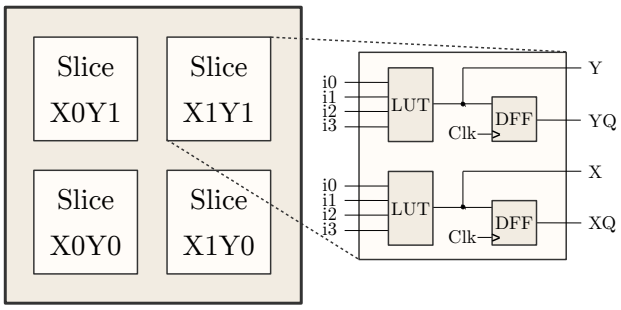


Abbildung 2.6: Aufbau eines CLBs der Spartan-3E Familie (vereinfacht)

auf dem FPGA integrierten *Flash*-Speicher. FPGAs mit nichtflüchtigem Speicher (z. B. *Flash*- oder *antifuse*-basierende FPGAs) halten die Konfigurationsinformationen und sind so direkt nach dem Einschalten einsatzbereit. Im Folgenden soll nun der Aufbau eines FPGAs am Beispiel der auf SRAM-basierenden Spartan-3E Familie [Xil13] von Xilinx vorgestellt werden.

2.1.2.1 Aufbau eines FPGAs

Die Struktur eines FPGAs setzt sich aus verschiedenen Elementen zusammen, welche matrixförmig auf dem Chip angeordnet sind (siehe Abbildung 2.5). Diese Elemente setzen die Schaltungslogik um, die zum Beispiel mit einer Hardwarebeschreibungssprache konfiguriert wird. Die wesentlichen Elemente sind dabei:

- Konfigurierbare Logikblöcke (engl. *Configurable Logic Blocks*, CLBs)
- Konfigurierbare Verbindungsstrukturen
- Ein-/Ausgabeblöcke (engl. *Input/Output Blocks*, IOBs)

Die CLBs bilden das Kernstück des FPGAs, denn diese stellen die Logikressourcen zur Realisierung von synchronen als auch kombinatorischen Schaltungen zur Verfügung. Die Spartan-3E Familie bietet je nach Familienmitglied zwischen 240 und 3.688 der CLBs [Xil13] auf einem FPGA an. Über die konfigurierbaren Verbindungsstrukturen können die CLBs untereinander und mit anderen Elementen auf dem FPGA verknüpft werden. So lassen sich Schaltungen über die gesamte Matrix des FPGAs realisieren. Über Verknüpfungen mit den Ein- und Ausgabeblöcken des FPGAs kann die konfigurierte Schaltung zudem mit den Anschlussstellen des FPGA-Gehäuses verbunden werden. Die CLBs selbst setzen sich für die Spartan-3E Familie, wie in Abbildung 2.6 dargestellt, aus vier *Slices* zusammen. Innerhalb einer *Slice* befinden sich unter anderem zwei LUTs (*Look-Up Tables*) mit jeweils vier Eingängen und zwei Speicherelemente (D-Flipflops bzw. *Latches*). Die auf SRAM-basierenden LUTs können zur Realisierung von beliebigen 4-stelligen Booleschen Funktionen verwendet werden. Für größere Boolesche Funktionen lassen sich die LUTs auch kaskadieren. Zusätzlich kann ein Teil der

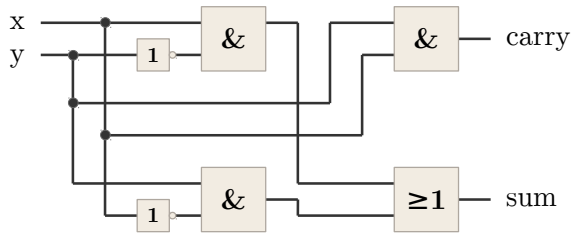


Abbildung 2.7: Darstellung des Halbaddierers auf Gatterebene

LUTs als 16-Bit-Speicher oder als Schieberegister verwendet werden. Mit Hilfe des D-Flipflops am Ausgang der LUT lässt sich die Ausgabe synchronisieren, falls keine kombinatorische Schaltung implementiert werden soll. Neben den bereits genannten elementaren Bestandteilen gibt es noch weitere optional verwendbare Elemente, wie beispielsweise:

- Block-RAMs (BRAMs)
- Multiplizierer
- Digitale Taktmanager (engl. *Digital Clock Manager*)

BRAMs sind kleine Speicher, welche im Falle der Spartan-3E Familie auf SRAM-basieren und 18 Kibit groß sind. Eine Besonderheit von BRAMs ist, dass diese über zwei Schnittstellen verfügen und so zwei gleichzeitige Schreib-/Lesezugriffe auf den Speicher möglich sind. Da die direkte Umsetzung von Multiplikationen in der konfigurierbaren Logik sehr teuer ist, findet man auf einigen FPGAs dedizierte Multiplizierer. Die Multiplizierer der Spartan-3E Familie können das Produkt von zwei 18-Bit-Binärzahlen berechnen. Taktmanager werden zur Synthese von Frequenzen oder zur Phasenverschiebung des Taktes eingesetzt.

2.1.2.2 Von der Hardwarebeschreibungssprache zur Hardware

Der Weg von der Hardwarebeschreibungssprache zum fertigen Hardwaredesign erfolgt über die proprietären Werkzeugketten der verschiedenen FPGA-Hersteller. Der erste Schritt zur Hardware ist die Synthese. Die Synthese setzt die High-Level-Beschreibung der Hardwarebeschreibungssprache in eine Darstellung auf Gatterebene um und erzeugt eine FPGA-spezifische Netzliste. Der graphische Schaltplan des in Unterkapitel 2.1.1.1 beschriebenen Halbaddierers auf hersteller- und technologieunabhängiger Gatterebene ist in Abbildung 2.7 zu sehen. Durch die triviale Funktionsweise des Halbaddierers, ist die in Abbildung 2.2 gezeigte Architekturbeschreibung direkt in dem graphischen Schaltplan wiederzuerkennen. Das Und-Gatter oben rechts in der Abbildung spiegelt die Und-Verknüpfung zur Berechnung des Übertrages wieder und die restlichen Gatter entsprechend die Entweder-Oder-Verknüpfung zur Berechnung der Summe.

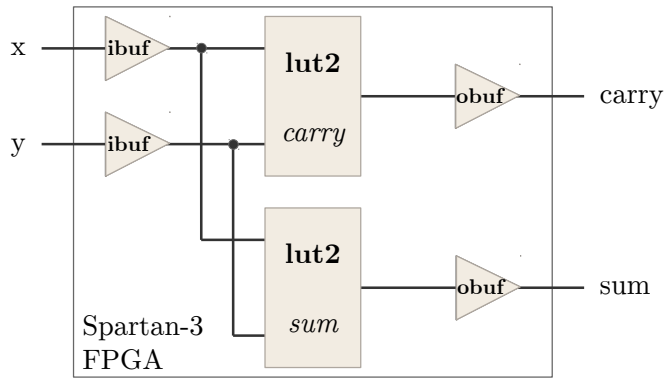


Abbildung 2.8: Technologieabhängige Darstellung (Spartan-3E) des Halbaddierers

In Abbildung 2.8 wird die technologieabhängige Darstellung des Halbaddierers präsentiert. Dementsprechend werden hier FPGA-spezifische Elemente verwendet. Zum einen die schon vorgestellten LUTs und zum anderen Ein- und Ausgabepuffer (`ibuf` bzw. `obuf`) zur Anbindung der internen Schaltungslogik an die IOBs. In diesem Beispiel werden nur zwei der vier verfügbaren Eingänge der beiden LUTs verwendet (für die Signale `x` und `y`), weshalb diese mit `lut2` markiert sind. Die obere LUT berechnet hier den Übertrag und die untere die Summe für den Halbaddierer. Eine LUT wird dazu wie eine Wahrheitstabelle verwendet, in der das Ergebnis für den Ausgang nachgeschlagen werden kann. Prinzipiell sind LUTs mit vier Eingängen kleine 16-Bit-Speicher an die 4-Bit-Adressen angelegt werden können, um das adressierte Bit auslesen zu können. Die obere LUT, welche eine Und-Verknüpfung repräsentiert, legt also genau dann eine Eins an den Ausgang, wenn die Binäradresse aus zwei Einsen besteht. Auf diese Art und Weise lassen sich mit LUTs beliebige logische Schaltungen realisieren.
In dem nächsten Schritt werden die herstellerspezifischen Elemente nach parametrisierbaren Kriterien auf dem FPGA platziert und die Verbindungsstrukturen werden entsprechend gesetzt. Mittels definierter Nebenbedingungen (engl. *Constraints*) kann der Benutzer der Werkzeugkette noch verschiedene Vorgaben setzen, wie zum Beispiel die Verknüpfung von ein- und ausgehenden Signalen mit den gewünschten Anschlüssen am FPGA-Gehäuse. Letztendlich wird in einem letzten Schritt eine Datei mit den Konfigurationsinformationen für das FPGA generiert, mit dem sich dieses schließlich konfigurieren lässt. Der vollständige Durchlauf der Werkzeugkette kann, abhängig von dem gegebenen Design, der angestrebten Optimierung, dem FPGA-Typen und dem verwendeten Rechner, Minuten oder im Falle von großen Hardwaredesigns, wie zum Beispiel *Multi-* oder *Manycore*-Systemen, sogar Stunden in Anspruch nehmen. Insbesondere, wenn das Hardwaredesign, wie bei dem in dieser Arbeit vorgeschlagenen Ansatz, direkt in eine automatisierte Entwurfsraumexploration einbezogen ist, sind die Zeiten der Werkzeugketten erheblich zu lang, um die Auslastung des FPGAs für ein Hardwaredesign zu ermitteln.

2.2 Produktlinien

Ein zentraler Punkt bei dieser Arbeit ist die Untersuchung der Fragestellung, ob sich die Konzepte aus der Produktlinienentwicklung auch auf Hardwarebeschreibungssprachen und somit auf das Hardwaredesign selbst übertragen lassen. So könnten sich sowohl Software-Produktlinien als auch Hardware-Produktlinien in einem gemeinsamen Prozess durchgängig maßschneidern lassen, um Fehlkonfigurationen zwischen den Schichten schon durch den Designprozess selbst auszuschließen. In diesem Kapitel sollen deshalb zunächst die Grundlagen zu Produktlinien vorgestellt und bekannte Methodiken aus der Welt der Software-Produktlinien diskutiert werden.

2.2.1 Einführung in Produktlinien

Die Wiederverwendung von Softwarekomponenten ist bei der heutigen Komplexität von Eingebetteten Systemen sehr wichtig. Gerade im Automobilbereich ist der Trend zu immer komplexerer Software deutlich zu erkennen. Nach anfänglich wenigen Kilobytes an Software, ist man bei modernen Kraftfahrzeugen inzwischen bei Millionen Zeilen von Quelltext angekommen [CMK+11]. Die Wiederverwendung ist dabei ein probates Mittel um der steigenden Komplexität von Software zu begegnen, welche allerdings kein Alleinstellungsmerkmal der Softwareentwicklung ist. Auch zur Fertigung von Automobilen werden zum Beispiel verwandte Konzepte angewendet, wie der Einsatz von Gleichteilen oder die Verwendung von fabrikat- und teilweise herstellerübergreifenden Plattformen. So können Entwicklungs- und Produktionskosten gesenkt und kürzere Zeitspannen bis zur Markteinführung verwirklicht werden. Des Weiteren sorgt die Wiederverwendung für eine höhere Qualität der Komponenten, da sich diese eventuell schon in anderen Produkten bewährt haben oder Schwachstellen längst beseitigt werden konnten.
Produktlinien zeichnet jedoch mehr aus, als die einfache Wiederverwendung von Komponenten. Dazu gehört, dass die Wiederverwendung der Komponenten auf eine gewisse Art und Weise organisiert und die einzelnen Komponenten nach einem definierten Bauplan zu einer Ausprägung der Produktlinie zusammengefügt werden können. Dies erfordert zur Entwicklung einer Produktlinie einen initalen Mehraufwand, da zunächst die wesentlichen Merkmale der Produktlinie sowie ihre Gemeinsamkeiten und Unterschiede ermittelt werden müssen [BKPS04]. Wichtig ist dabei vor allem ein gewisser strategischer Weitblick über das einzelne Produkt hinaus [CN07], da ansonsten die getroffenen Designentscheidungen bei zukünftigen Produkten zu Einschränkungen führen können. Trotzdem dürfen die Grenzen der Produktlinie nicht zu weit gefasst werden, da ansonsten die Kosten für die Produktlinie ausufern können.
In Abbildung 2.9 ist ein Vorschlag für den Entwicklungsprozess einer Software-Produktlinie dargestellt. Der Prozess ist zum einen in die Domänenentwicklung und zum anderen in die Anwendungsentwicklung unterteilt. Die Domänenentwicklung beschäftigt sich dabei mit der Entwicklung der eigentlichen Produktli-

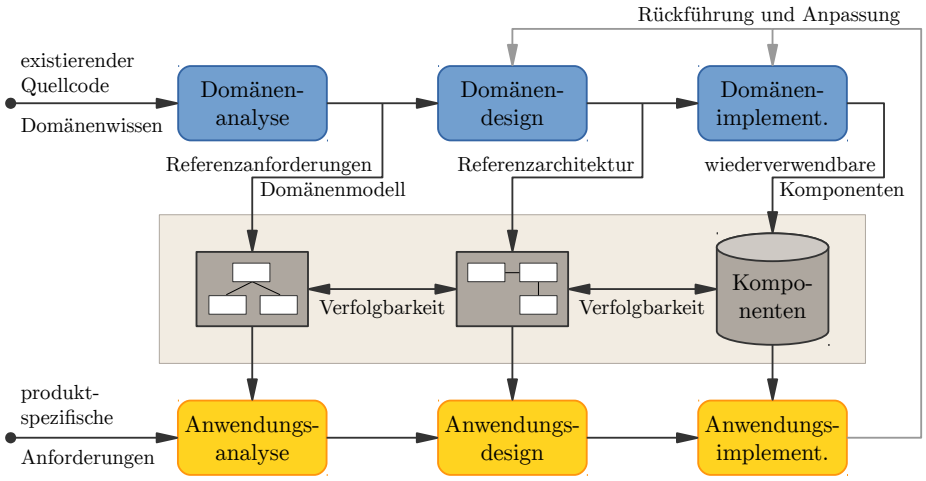


Abbildung 2.9: Software-Produktlinien Referenzprozess nach [vdL02]

nien und deren wiederverwendbaren Komponenten (mittlere Teil der Abbildung), welche variabel konfiguriert und so verschiedene Varianten instantiiert werden können. Hier wird also nicht für ein bestimmtes Produkt, sondern für eine Menge von Produkten einer anvisierten Domäne entwickelt. Dazu wird zunächst analysiert, welche Anforderungen an die Produktlinie gestellt werden, nachdem dann in einem zweiten Schritt die benötigten wiederzuverwendenden Komponenten bestimmt werden. Innerhalb der Domänenimplementierung werden nun diese Komponenten für die Produktlinie implementiert. Die Anwendungsentwicklung hingegen beschäftigt sich mit der produktspezifischen Entwicklung der Software. Auch hier folgt nach der Analyse und dem Design die letztendliche Implementierung. In der Anwendungsentwicklung werden allerdings nur die bisher in der Plattform noch nicht verfügbaren und damit spezifischen Komponenten implementiert. Zusammen mit den wiederverwendbaren Komponenten ergibt sich so ein spezifisches Produkt der Produktlinie. Idealerweise stammen die meisten Komponenten aus der Domänenentwicklung und nur ein kleiner Teil muss produktspezifisch entwickelt werden. Komponenten, welche während der Anwendungsentwicklung entstanden sind, können nachträglich mit entsprechendem Aufwand zu der Plattform hinzugefügt werden, falls diese voraussichtlich auch in zukünftige Produkte einfließen sollen. Auch die nachträgliche Änderung von wiederverwendbaren Komponenten der Plattform ist möglich, birgt allerdings die Gefahr, dass Inkonsistenzen zu bereits existierenden Ausprägungen der Produktlinie auftreten können. Wegen des initialen Mehraufwandes und der komplexeren Zusammenhänge eignen sich Produktlinien deshalb nur dann, wenn vorauszusehen ist, dass zukünftig verschiedene Ausprägungen der Produktlinien benötigt werden.

Der hier vorgestellte Software-Produktlinien Referenzprozess wird in Kapitel 4 dazu verwendet, um dessen Übertragbarkeit auf die LavA *Manycore*-Hardware anhand der drei Phasen der Domänenentwicklung zu untersuchen.

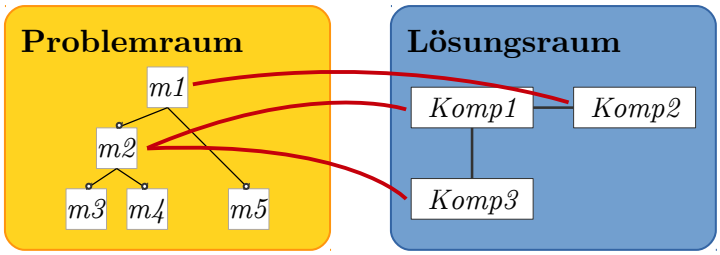


Abbildung 2.10: Unterteilung der Domänenentwicklung in Problem- und Lösungsraum

2.2.2 Softwaretechnische Konzepte zur Domänenentwicklung

In der Domänenentwicklung wird zwischen dem Problemraum und dem Lösungsraum unterschieden [CE00] (siehe Abbildung 2.10). Der Problemraum wird mittels der Domänenanalyse definiert und anhand von Merkmalen (engl. *Features*) [KCH+90] repräsentiert. Die Merkmale beschreiben dabei die Gemeinsamkeiten und die Unterschiede für verschiedene Ausprägungen der Produktlinie. Der Lösungsraum dagegen beinhaltet die wiederverwendbaren Komponenten der Produktlinie aus denen ein spezifisches Produkt zusammengesetzt werden kann. Damit anhand von selektierten Merkmalen aus dem Problemraum auch die entsprechenden Komponenten ihren Weg in ein spezifisches Produkt finden, sind Problem- und Lösungsraum über das Konfigurationswissen miteinander verknüpft. Das Konfigurationswissen gibt den Bauplan für die Produktlinie vor und beschreibt z. B. Abhängigkeiten zwischen Komponenten oder nicht erlaubte Kombinationen von Komponenten. In der Softwareentwicklung gibt es verschiedene Konzepte und Techniken, um Variabilität in Software auszudrücken oder den Problemraum für eine Domäne aufzuspannen. Einige zentrale Konzepte und Techniken werden nun im Folgenden vorgestellt.

2.2.2.1 Variabilität im Lösungsraum

Insbesondere bei implementierungsgetriebenen Softwaresystemen mit nur wenig Variabilität werden Lösungsraumtechniken oftmals auch ohne eine entsprechende Abbildung des Problemraums eingesetzt. Bei wachsender Funktionalität und damit unter Umständen auch steigender Variabilität der Software, stößt die ausschließliche Verwendung dieser Techniken allerdings schnell an ihre Grenzen. Dies betrifft natürlich komplexe Produktlinien umso mehr.

Bedingte Präprozessor-Direktiven Die einfachste Möglichkeit um in den Sprachen C oder C++ Programme mit der nötigen Variabilität zu versehen, sind Präprozessor-Direktiven zur bedingten Übersetzung des Quelltextes. Durch die Nutzung der bedingten Übersetzung können statisch zur Übersetzungszeit Komponenten in die Software eingebracht oder daraus entfernt werden. Diese Technik ist aber aufgrund der Unübersichtlichkeit und der schlechten Wartbarkeit

des Quelltextes weniger für Produktlinien geeignet. Lohmann *et al.* untersuchten diesbezüglich den Quelltext des konfigurierbaren Betriebssystems eCos [Mas02] und bezeichneten die Verwendung der bedingten Übersetzung, insbesondere bei hochkonfigurierbaren Systemen, als „`#ifdef`-Hölle“ [LST+06].

Frame-Technologie Die *Frame*-Technologie [Bas96] von Bassett ist ein wesentlich mächtigeres Konzept um Variabilität auszudrücken, welches zudem unabhängig von der jeweiligen Zielsprache eingesetzt werden kann. Ein Werkzeug, welches das Konzept der *Frame*-Technologie aufgreift, ist XVCL (*XML-based Variant Configuration Language*) [JBZZ03]. Das Grundgerüst in XVCL stellen die sogenannten x-*Frames*, welche sowohl den Quelltext der zu generierenden Zielsprache als auch XVCL-Befehle zur Beschreibung der Variabilität beinhalten. Über die Adpation von anderen x-*Frames* können Komponenten wiederverwendet werden. Neben der Adaption können noch Variablen, Bedingungen oder Schleifen eingesetzt werden, um den Quelltext der Zielsprache maßzuschneidern. Die Menge der in Relation stehenden x-*Frames* bilden den Lösungsraum der entwickelten Produktlinie.

Generische Programmierung Eine weitere Möglichkeit zur Umsetzung von Variabilität im Lösungsraum ist die generische Programmierung [CE00]. Die generische Programmierung erlaubt die datentyp-unabhängige Implementierung von Algorithmen und Datenstrukturen. So kann der unnötig entstehende Aufwand zur Implementierung von nahezu redundaten Lösungen für verschiedene Datentypen verhindert werden. Die Generierung einer Lösung für einen spezifischen Datentypen kann dann automatisiert vom Übersetzer durchgeführt werden. In der statisch getypten Programmiersprache C++ können generische Programme mittels Klassen- und Funktions-*Templates* realisiert werden. *Templates* können in C++ zur Meta-Programmierung verwendet werden und bilden zusammen mit anderen Spracheigenschaften eine Turing-vollständige Subsprache in C++ [CE00], um Quelltext zur Übersetzungszeit zu konfiguieren und zu generieren.

Merkmalorientierte Programmierung Eine direkt auf Merkmalen aufsetzende Variante zur Nutzung von Variabilität in Software ist die merkmalorientierte Programmierung (engl. *Feature-Oriented Programming*, FOP). Die bei der merkmalorientierten Programmierung implementierten Softwarekomponenten spiegeln die Merkmale der Produktlinie wieder und lassen sich mittels Komposition zu einem Produkt der Produktlinie zusammenfügen. Varianten der merkmalorientierten Programmierung sind GenVoca [BO92] und der Nachfolger AHEAD [BSR04], welche die Formulierung von Produktlinien in algebraischer Form ermöglichen.

Aspektorientierte Programmierung Die Aspektorientierte Programmierung (engl. *Aspect-Oriented Programming*, AOP) [KLM+97] ist ein Programmierparadigma zur Trennung von Belangen. Dazu werden querschneidende Belange,

welche konzeptionell verschiedene Belange betreffen, separat in Aspekten implementiert. Die so gekapselten querschneidenden Belange können dann über einen Aspektweber in den Quelltext eingewoben werden. Diese Technik kann – neben der Kapselung von Belangen – auch zur Umsetzung von Variabilität in Software-Produktlinien eingesetzt werden. Dazu werden die wiederverwendbaren Komponenten in Aspekten implementiert und bei Bedarf eingewoben. Diese Technik wird beispielsweise zur Maßschneiderung der Betriebssystem-Produktlinie CiAO [LHSP+09] oder dem hochkonfigurierbaren CiAO/IP-Stack eingesetzt [BLS12]. Die aspektorientierte Programmierung ist als Erweiterung für verschiedene Sprachen, wie C++ [SL07] oder Java [KHH+01] verfügbar.

2.2.2.2 Modellierung des Problemraums

Neben den bisher vorgestellten Techniken, mit denen Programmiersprachen um die benötigte Variabilität im Lösungsraum erweitert werden können, müssen für eine Produktlinie auch Techniken zur Darstellung des Problemraums zur Verfügung stehen. Die bisher vorgestellten Techniken eigenen sich dazu nicht. Man könnte zwar argumentieren, dass sich der Problemraum, zum Beispiel im Falle der bedingten Präprozessor-Direktiven, über eine Header-Datei abbilden lässt, für komplexe und hochkonfigurierbare Produktlinien ist so eine Umsetzung des Problemraums jedoch nicht geeignet. Im Folgenden werden deshalb Konzepte und in der Praxis verwendete Werkzeuge vorgestellt, welche sich für die Modellierung des Problemraums von Software-Produktlinien einsetzen lassen.

Merkmalmodelle Weit verbreitet ist das Konzept der Merkmalmodelle (engl. *Feature Models*) zur Repräsentation des Problemraums. In einem Merkmalmodell werden die Merkmale einer Produktlinie und deren Beziehung zueinander ausgedrückt. Des Weiteren kann angegeben werden, ob die Auswahl eines Merkmales optional oder bindend ist. Die graphische Notation eines Merkmalmodells wird als Merkmaldiagramm bezeichnet. Ein Beispiel für ein Merkmaldiagramm wird in Abbildung 2.11 präsentiert. In einem Merkmaldiagramm werden die Merkmale baumartig dargestellt und mit entsprechenden graphischen Notationen für die Abhängigkeiten zwischen den Merkmalen versehen. Unbedingt erforderliche Merkmale sind in dieser Notation mit einem ausgefüllten Kreis markiert und optionale Merkmale mit einem nicht ausgefüllten Kreis. Bei einem Auto ist zum Beispiel die Karosserie zwingend erforderlich, wohingegen eine Anhängerkupplung oftmals optional hinzugefügt werden kann. Merkmale können auch zu Gruppen zusammengefasst und so in Beziehung zueinander gesetzt werden. Die Beziehung zwischen Merkmalen kann durch einen Bogen ausgedrückt werden. Wenn die Fläche über dem Bogen nicht ausgefüllt ist, handelt es sich um alternative Merkmale. So kann bei einem Auto entweder ein manuelles Getriebe oder ein Automatikgetriebe verbaut werden, nicht jedoch beides. Bei einer ausgefüllten Fläche ist jede Kombination der Merkmale in dieser Gruppe erlaubt. Übertragen auf das Auto bedeutet dies zum Beispiel, dass zwischen einem Verbrennungsmotor, einem Elek-

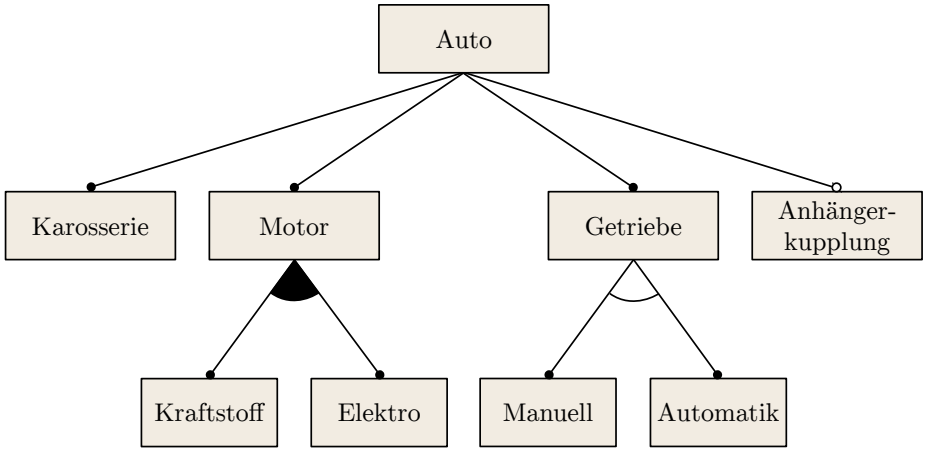


Abbildung 2.11: Beispiel für ein Merkmalmodell [CE99]

tromotor oder einer Kombination dieser beiden Motoren – einem Hybridantrieb – ausgewählt werden kann. Eine Erweiterung zu den klassischen Merkmalmodellen bilden die auf kardinalitäten-basierenden Merkmalmodelle [CHE05], mit denen eine Komponente unter Berücksichtigung von definierten Schranken, mehrfach in das Produkt eingebunden werden kann.

Das Konzept der Merkmalmodelle wird in verschiedenen Werkzeugen zur Modellierung des Problemraums eingesetzt. Eines dieser Werkzeuge ist pure::variants [pur06]. pure::variants beschränkt sich jedoch nicht alleine auf die Modellierung des Problemraums anhand von Merkmalmodellen, sondern bietet zudem ein Familienmodell zur Bereitstellung von Variabilität im Lösungsraum. In dem Familienmodell von pure::variants können Merkmale aus dem Merkmalmodell mit Textfragmenten aus der Zielsprache verknüpft werden und so zu maßgeschneiderten Quelltextdateien zusammengefügt werden.

Auch die Konfigurationssprache Kconfig kann zur Modellierung von Merkmalen verwendet werden. Kconfig wird typischerweise zur Beschreibung des Problemraums im Kontext des Linux-Kernels eingesetzt und erlaubt die textuelle Beschreibung der zu selektierenden Merkmale des Linux-Kernels sowie die Modellierung der Abhängigkeiten zwischen den Merkmalen. Denn ebenso wie viele Betriebssystem-Produktlinien aus dem Bereich der Eingebetteten Systeme, ist auch der Linux-Kernel mit einer Vielzahl von Optionen zur Konfigurierung ausgestattet und es werden ähnliche Ziele wie in der Software-Produktlinienentwicklung anvisiert [SSSPS07].

Modellgetriebene Entwicklung Ein weiteres Konzept mit dem der Problemraum einer Produktlinie dargestellt werden kann, wird durch die modellgetriebene Entwicklung (engl. *Model-Driven Development*, MDD) [SVEH07] bereitgestellt. Der modellgetriebene Ansatz hebt die Entwicklung auf eine höhere Abstraktionsebene, welche über ein Modell repräsentiert wird. Das Modell stellt, im Kontext

einer Produktlinie, ein konkretes Problem des Problemraums dar und kann mittels einer Transformation in den Quelltext der Zielsprache überführt werden. Werkzeuge, wie das EMF[2] (*Eclipse Modeling Framework*), stellen dazu Codegeneratoren zur Verfügung und bieten so auch Variabilität im Lösungsraum. Der Problemraum kann bei der modellgetriebenen Entwicklung über ein Metamodell spezifiziert werden, aus dem sich die konkreten Modelle mit den anvisierten Merkmalen ableiten lassen.

2.2.3 Ableitung von Produkten

Für die Ableitung von konkreten Produkten aus einer Produktlinie gibt es im Wesentlichen zwei verschiedene Strategien. Zum einen die vollständige automatisierte Ableitung eines Produktes und zum anderen kann ein Produkt auch manuell aus den wiederverwendbaren Komponenten der Produktlinie zusammengesetzt werden [ABKS13]. Bei der vollständigen Ableitung eines Produktes werden zuvor sämtliche Teile in der Domänenentwicklung implementiert. Abhängig von einem gegebenen Anwendungsfall können dementsprechend die maßgeschneiderten Produkte in vollem Umfang und automatisiert generiert werden. Umgesetzt wird diese Strategie zum Beispiel in den konfigurierbaren Betriebssystemen eCos und CiAO. So kann für eine Anwendung ein passendes Betriebssystem abgeleitet werden, ohne dass weitere Entwicklungszeit in Teile des Betriebssystems einfließen müssen.
Auf der anderern Seite können Produktlinien auch nur Teile eines Produktes enthalten, welche in der Anwendungsentwicklung zunächst manuell zu einem vollständigen Produkt zusammengesetzt werden müssen. Die Fragmente der Produktlinie können dabei beispielsweise wiederverwendbare Bibliotheken, Klassen oder *Templates* umfassen.

[2]EMF: `http://eclipse.org/modeling/emf/`

Kapitel 3

Verwandte Arbeiten

Inhalt

Werkzeuge zur (teil-)automatisierten Generierung sowie Optimierung von *Multi-* und *Manycore*-Systemen stehen seit einigen Jahren im Fokus der Forschung. Alle Werkzeuge haben zunächst das gemeinsame Ziel den Entwickler durch einen möglichst automatisierten Entwurfsprozess und mittels hoher Abstraktionsebenen bei der Umsetzung solcher Systeme zu unterstützen. Wie diese Unterstützung im Detail umgesetzt ist, variiert bei den einzelnen Werkzeugen vielfältig. Einige Werkzeuge optimieren allein die Zuordnung von Tasks zu Prozessoren für ein fest spezifiziertes Multiprozessorsystem [TBHH07]. Andere stellen Möglichkeiten zur Co-Simulation von Hard- und Software zur Verfügung, um eine spezifizierte Architektur bezüglich verschiedener nicht-funktionaler Eigenschaften, wie Ressourcen, Zeit, Energie oder Latenzen, zu bewerten [RBM+04, CdOCM09]. Wieder andere können aus einer Beschreibung der Hardware auf hoher Abstraktionsebene die entsprechende synthetisierbare Hardware aus einer Bibliothek von Hardwarekomponenten automatisiert generieren [BBA+11, LYBJ01, LF08, WLZ+12], wodurch die Entwicklungszeit drastisch verkürzt werden kann und so mit relativ geringem Zeitaufwand anwendungsspezifische Systeme entwickelt werden können. Denn insbesondere im Bereich der Eingebetteten Systeme werden spezialisierte Lösungen benötigt, um die eng gesteckten Restriktionen einhalten zu können. Neben diesen

auf einzelne Teilbereiche spezialisierten Lösungen existieren auch Werkzeuge, die nahezu alle aufgezählten Punkte abdecken und den Entwickler von der Anwendung bis zur Hardware mit einem automatisierten Entwurfsprozess unterstützen [TNS+07, KKO+06]. Bei dieser Art von Werkzeugen wird neben der Zuordnung von Tasks zu Prozessoren auch die Hardware selbst in einer Entwurfsraumexploration optimiert.
In den folgenden Unterkapiteln werden nun verwandte Arbeiten aus der Forschung mit dem in dieser Arbeit verfolgten Ansatz verglichen und diskutiert. Der Fokus liegt dabei auf den Themengebieten mit denen sich diese Arbeit auseinandersetzt und für die insbesondere der Stand der Kunst verbessert werden soll. Deshalb wird strukturiert nach den Themengebieten zunächst betrachtet, auf welcher Ebene und in welcher Form sowohl die Hardware als auch die Software in den verschiedenen Werkzeugen repräsentiert wird. Anschließend wird beleuchtet inwieweit die Systemsoftware in den Entwurfsprozess der Werkzeuge eingebunden ist und welche Techniken für die Untersuchung mittels Entwurfsraumexploration verwendet werden.

3.1 Repräsentation der Hard- und Software

Einer der wichtigsten Aspekte für eine Werkzeugkette zur Maßschneiderung von *Multi-* und *Manycore*-Systemen ist die gewählte Darstellung zur Repräsentation der Systeme und in welchem Umfang sich somit tiefgehende und komplexe Details des Systems vor dem Entwickler verbergen lassen, denn mit einer entsprechend gewählten Repräsentation kann der Entwicklungsprozess auf eine bestimmte Entwicklergruppe zugeschnitten werden. Diese Entwicklergruppe muss zum Beispiel bei einer geeigneten Repräsentation eines *Multi-* oder *Manycore*-Systems nicht einmal zwingend mit dem Hardwareentwurf vertraut sein. So könnten zum Beispiel auch Anwendungsentwickler mit einer entsprechenden Werkzeugunterstützung vollständige Systeme aus Software und Hardware entwickeln, denn wie in dem folgenden Zitat ausgeführt, sind es oftmals Anwendungsentwickler, die sich mit der Entwicklung von Eingebetteten Systemen beschäftigen:

> „...embedded systems are usually defined by the experts in application domain who understand application very well, but have only basic knowledge of design technology and practice. System-level design technology allows them to specify, explore and verify their embedded system products without expert knowledge of system engineering and manufacturing.“
>
> D. D. Gajski *et al.* [GAGS09]

Neben der Abstraktionsebene, auf der die Hardware repräsentiert wird, und welche sich bei vielen der Werkzeuge in der Forschung auf Systemebene befindet,

ist auch die Art und Weise in der die Eingabe für das jeweilige Werkzeug vorliegen muss entscheidend, ob es für die eine oder andere Entwicklergruppe zugänglich bzw. „programmiererkompatibel“ ist. Insbesondere für Anwendungsentwickler sollte deshalb die Hardwarestruktur weit in den Hintergrund rücken und die Maßschneiderung bzw. die Optimierung der Hardware automatisiert von einem Werkzeug durchgeführt werden. Der in dieser Arbeit verwirklichte softwarezentrierte Ansatz geht genau in diese Richtung. Dazu soll dem Anwendungsentwickler eine Abstraktion auf Ebene der Software in Form von APIs geboten werden, aus deren Nutzung sich die benötigten Parameter zur Maßschneiderung von Systemsoftware und Hardware ableiten lassen. Im Folgenden werden deshalb nun verschiedene Werkzeuge aus der Forschung bezüglich der Schnittstelle zum Entwickler, also wie Anwendung und Hardware für die Werkzeugkette spezifiziert werden müssen, näher untersucht.

3.1.1 Prozessnetzwerke

Ein weitverbreitetes und gängiges Mittel zur Modellierung von Anwendungen sind Prozessnetzwerke. Diese werden insbesondere für parallele Multimedia- und Datenstrom-Anwendungen eingesetzt. Das wohl bekannteste Beispiel für Prozessnetzwerke sind KPNs (Kahn-Prozessnetzwerke) [Kah74]. Sie erlauben die Modellierung von verteilten Systemen und Systemen zur Signalverarbeitung. Bei solch einem Netzwerk repräsentieren die Knoten die Prozesse der Anwendung und die Kommunikation zwischen den Prozessen wird über Kanten (unbegrenzte FIFOs) dargestellt. Eine durch Prozessnetzwerke modellierte Anwendung eignet sich insbesondere zur automatisierten Zuteilung von Tasks zu Prozessoren.

Ein Werkzeug, das sich mit dieser Zuordnung im Bereich der Eingebetteten Systeme beschäftigt ist DOL [TBHH07]. Bei DOL wird die Struktur der Anwendung über Prozessnetzwerke definiert und das Verhalten der Anwendung entsprechend über Quelltext. Diese Unterteilung vereinfacht die automatisierte Zuweisung von Tasks zu Prozessoren. Zusätzlich unterstützt DOL den Entwickler mit einem Programmiermodell, welches Kommunikationsprimitiven zur Kommunikation zwischen den Tasks bietet. Die Hardwarearchitektur wird bei DOL nicht maßgeschneidert, muss allerdings zur Bestimmung der Zuordnung in einer separaten Spezifikation festgelegt werden und wird in der folgenden Entwurfsraumexploration zur Optimierung der Zuordnung von Tasks zu Prozessoren nicht variiert.

Eine der bekanntesten Werkzeugketten zur automatisierten Maßschneiderung von Multiprozessorsystemen ist Daedalus [TNS+07]. Daedalus verwendet zur Modellierung der Anwendung ebenfalls KPNs, welche automatisiert mit KPNgen [VNS07] generiert werden, um die Anwendung zu parallelisieren. Die Möglichkeiten zur Parallelisierung beschränken sich bei KPNgen auf SANLPs (*Static Affine Nested Loop Programs*), welche mittels Übersetzertechniken zu KPNs überführt werden können. Im Gegensatz zu DOL wird bei Daedalus in der Entwurfsraumexploration nicht nur die Zuweisung von Tasks zu Prozessoren untersucht, sondern auch die Hardwarearchitektur bestimmt. Nachdem die Hardware und die Zuord-

nung von Tasks optimiert wurde, lässt sich mit Espam [NSD06] das Hardwaremodell auf Registertransferebene und der Quelltext für die einzelnen Prozessoren generieren.
Auch die Werkzeuge CASSE [RBM+04] und Open-Scale [BBA+11] nutzen zur Modellierung der Anwendung KPNs. CASSE erlaubt die Co-Simulation von Anwendung und Multiprozessorsystem zur Untersuchung des Entwurfsraums. Neben den KPNs muss der Entwickler die Architektur der Hardware und die Zuordnung der Tasks in einem einfachen Textdokument angeben. Bei CASSE nutzen die Anwendungen ein von KPNs abgeleitetes Protokoll zur Kommunikation zwischen den Tasks. Nachdem die Eingaben eingelesen und ausgewertet wurden, wird die Architektur simuliert und anschließend verschiedene Metriken und ein Ablaufprotokoll an den Entwickler zurückgegeben. Bei eventuell auftretenden Problemen muss der Entwickler manuell eingreifen und das Prozessnetzwerk, die Hardwarearchitektur oder die Zuordnung von Tasks zu Prozessoren abändern. Mit Open-Scale können Multiprozessorsysteme zwar nicht für eine Entwurfsraumexploration ausgewertet aber generiert werden. Die Kommunikation innerhalb des Multiprozessorsystems basiert auf einem konfigurierbaren NoC (*Network on Chip*). Zusätzlich stellt Open-Scale eine MPI-artige Schnittstelle zur Kommunikation für den Entwickler zur Verfügung.
Kim *et al.* präsentieren einen Ansatz zur Entwurfsraumexploration von Multiprozessorsystemen bezüglich der Energieaufnahme und der benötigten Ressourcen [KBDV06]. Der Entwickler muss bei diesem Ansatz die Anwendung mittels eines Taskgraphen modellieren und die in diesem Szenario zu verwendende Teilmenge an Typen von PEs (*Processing Elements*) angeben. Während der Entwurfszyklen wird dann die Anzahl der PEs erhöht, falls die Ressourcenbeschränkungen nicht verletzt werden. Ebenso wird für jedes PE basierend auf dem zu erwartenden Energieverbrauch die *Scheduling*-Strategie variiert. Wenn die Anzahl an PEs während der Entwurfszyklen erhöht wurde, wird auch die Zuordnung von Tasks zu den PEs neu bestimmt. Das Resultat dieses Ansatzes ist dann eine entsprechende Zuordnung von Tasks zu PEs und die jeweils zu verwendenden *Scheduling*-Strategien pro PE.

3.1.2 Unified Modeling Language

UML [HK03] (*Unified Modeling Language*) ist eine Modellierungssprache aus der Softwareentwicklung. Mit UML können Systeme graphisch visualisiert und entwickelt werden. Dabei beschränken sich die Fähigkeiten von UML nicht allein auf die Darstellung von Softwaresystemen. Eine Werkzeugkette, welche UML zum Entwurf von Multiprozessorsystem nutzt, ist Koski [KKO+06]. In Koski wird eine Anwendung über Klassendiagramme spezifiziert. Zusätzlich kann optional die Hardwarearchitektur über ein Kompositionsstrukturdiagramm definiert werden, welches aber im Laufe der Optimierung verändert werden kann. Falls keine initiale Hardwarearchitektur vom Entwickler festgelegt wird, bestimmt die Entwurfsraumexploration diese im Zuge der Optimierung. Auch die Zuteilung von

Task-Gruppen zu Prozessoren kann in UML zunächst mit einer initialen Belegung spezifiziert werden. Koski nutzt eine Softwarebibliothek, welche neben vorimplementierten Anwendungsalgorithmen auch den Plattform-Code enthält. Die Entwurfsraumexploration von Koski liefert zum Schluss ein anwendungsspezifisches Multiprozessorsystem mit einer optimierten Zuordnung der Tasks.

3.1.3 Domänenspezifische Sprachen

Einige Werkzeuge, wie auch das EDK[1] (*Embedded Development Kit*) von dem FPGA-Hersteller Xilinx, bieten zur Maßschneiderung von Multiprozessorsystemen eine graphische Benutzerschnittstelle, in der verschiedene Parameter des Systems auf der Ebene von Prozessoren, Speichern, Peripheriekomponenten oder Kommunikationsstrukturen per Hand gesetzt werden können. Auch in dem Werkzeug HeMPS [CdOCM09] wird eine graphische Oberfläche zur Konfigurierung der Hardware eingesetzt. Die Schnittstelle von HeMPS erlaubt es, ein zweidimensionales Netzwerk [MCM+04] zwischen einer konfigurierbaren Anzahl an Prozessoren aufzuspannen und die Speichergröße der Prozessoren zu konfigurieren. Neben einem Modell auf Registertransferebene zur Synthese kann HeMPS auch ein Modell auf Befehlssatzebene zur Simulation generieren, um die Performanz des Systems zu evaluieren.
Neben den graphischen Benutzerschnittstellen gibt es noch eine Reihe von Werkzeugen, welche eigens definierte Formate zur Spezifikation der Systeme einsetzen. Lyonnard *et al.* stellen zum Beispiel einen Ansatz zur automatisierten Generierung von anwendungsspezifischen und heterogenen Multiprozessorsystemen vor [LYBJ01]. Die Multiprozessorsysteme werden dabei aus einer Bibliothek von *Templates* zusammengesetzt und anhand einer speziellen Architekturspezifikation vom Entwickler bestimmt. Der Entwickler kann dabei Parameter, wie den Typen und die Anzahl der Prozessoren, die Größe der Speicher, sowie Kommunikationskanäle und -protokolle festlegen.
Ein Ansatz zur automatisierten Generierung von NoC-basierenden Multiprozessorsystemen wird in [LF08] vorgestellt. Der hier gezeigte Ansatz setzt auf dem EDK von Xilinx auf und versucht die Integration von benutzerdefinierten IP-Komponenten zu erleichtern. Die Systemarchitektur wird über eine textuelle Beschreibung spezifiziert und innerhalb der Werkzeugkette automatisiert zu Projektdateien des EDK transformiert. Die Beschreibung der Systemarchitektur beschränkt sich auf die Festlegung des Typen und der Anzahl der jeweils zu verwendenden IP-Komponenten. Die benötigten Verknüpfungen zwischen den IP-Komponenten zur Kommunikation werden wiederum automatisiert generiert.
Auch XML-artige Spezifikationen finden regen Gebrauch bei der Beschreibung von Hardware. OpTiMSoC (*Open Tiled Manycore System-on-Chip*) [WLZ+12] kann zum Beispiel über eine vom Entwickler in XML spezifizierte Plattformbeschreibung ein *Manycore*-System aus verschiedenen IP-Komponenten einer Bibliothek zusammensetzen. Auch mit der Werkzeugkette NoCMaker [JCRR+09]

[1] EDK: `http://www.xilinx.com/tools/platform.htm`

lässt sich ein spezifisches Multiprozessorsystem in einem XML-Format beschreiben. Neben der Generierung des NoC-basierenden Multiprozessorsystems, bietet die Werkzeugkette zusätzlich Softwareschichten zur Kommunikation mittels MPI, um dem Entwickler das Programmieren von parallelen Anwendungen zu erleichtern. Die unterstützte Teilmenge an MPI-Funktionen stellt, neben Primitiven zur Kommunikation zwischen zwei Teilnehmern, auch kollektive Kommunikationsoperationen bereit. Ähnlich wie NoCMaker, setzen auch Wu *et al.* bei ihrem Ansatz auf MPI, um die semantische Lücke zwischen der Anwendung und den inzwischen hochparallelen Hardwarearchitekturen zu verkleinern [WHH10]. Der vorgeschlagene Entwurfsprozess startet mit einer Spezifikation der Anwendung in SystemC, einer XML-Beschreibung für die gewünschte Topologie und der Zuordnung von Tasks zu Prozessoren. Die weiteren Schritte verlaufen automatisiert und generieren die Hardware sowie die benötigten Softwarebibliotheken.
Auch für die Beschreibung solcher Systeme eigens entwickelte domänenspezifische Sprachen sind in der Literatur zu finden. Takada *et al.* definieren beispielsweise eine Sprache zur Beschreibung der Systemkonfiguration [THNY03]. Aus dieser Beschreibung kann automatisiert die Konfiguration für die Hardwarearchitektur und das verwendete Echtzeitbetriebssystem abgeleitet werden. Durch diesen Schritt kann auf die teilweise redundanten Angaben zur Konfiguration von Hard- und Software verzichtet werden.

3.1.4 Fazit

Für die Repräsentation der Hard- und Software gibt es in der Forschung, wie in diesem Kapitel vorgestellt, viele verschiedene Ansätze und Herangehensweisen. Aus der Sicht eines Anwendungsentwicklers sind viele dieser Ansätze aufgrund der gewählten Repräsentation nicht optimal geeignet. Zum einen wird bei verschiedenen Werkzeugen die explizite Spezifikation der Hardware gefordert, was speziell für Hardwareplattformen mit vielen, feingranular, konfigurierbaren Parametern schwierig ist und letztendlich wieder Expertenwissen auf Ebene der Hardware voraussetzt. Dies gilt vor allem für Hardwareplattformen mit stark heterogenen Strukturen, wie zum Beispiel Plattformen mit verschiedenen Prozessortypen und unterschiedlichen Optionen zur Kommunikation, welche schon allein durch die gebotene Vielfalt nur schwer manuell an die jeweiligen Anforderungen anpassbar sind. Und zum anderen ist die mitunter verwendete Darstellung zur Spezifikation der Systeme in Form von eigens entwickelten Sprachen oder XML-Spezifikationen nicht ideal auf die Gruppe der Anwendungsentwickler zugeschnitten. Von den betrachteten Werkzeugen erfüllen einzig Daedalus und Koski die Anforderungen, dass die Hardware bei Bedarf vollständig in den Hintergrund rückt und allein die Anwendung bzw. zusätzliche UML-Modelle zur Spezifizierung der Anwendung als Eingabe für die Werkzeuge ausreichen. Solch einen vollständig automatisierten Entwurfsprozess strebt auch diese Arbeit an. Im Hinblick auf die voranschreitenden Möglichkeiten zur Parallelisierung durch *Manycore*-Systeme besteht allerdings das Problem, dass die Werkzeuge auch eine Schnittstelle für

den Entwickler bieten müssen, um die Parallelität der Hardware ausnutzen zu können. Die in dieser Arbeit anvisierte Lösung zu diesem Problem ist eine parallele Programmierschnittstelle, aus der anhand der getätigten Zugriffsmuster die benötigten Konfigurationsinformationen zur Maßschneiderung von Systemsoftware und Hardware automatisiert extrahiert werden können. Zudem muss der Entwickler weiterhin seine eigene Experten-Domäne – die Anwendungsentwicklung – nicht verlassen, wenn die Konfigurationsinformationen direkt aus der Anwendung gewonnen werden können. So kann der Entwickler indirekt, über die simple Verwendung von Kommunikationsprimitiven und Treibern der Peripheriekomponenten, die benötigten Konfigurationsinformationen für Systemsoftware und Hardware zur Verfügung stellen.

3.2 Einbeziehung der Systemsoftware

Bei der heutigen Komplexität von *Multi-* und *Manycore*-Systemen ist der Einsatz einer zumindest leichtgewichtigen Systemsoftwareschicht nahezu unausweichlich, wenn Anwendungen effizient für solche Systeme entwickelt werden sollen. Die Systemsoftware kann die unterschiedlichen Architekturen auf Hardwareebene verbergen und dem Entwickler so eine einheitliche Schnittstelle zur Entwicklung der Anwendungen bieten, damit von den verwendeten Prozessortypen und den genutzen Kommunikationsstrukturen abstrahiert werden kann. Insbesondere wenn innerhalb einer Entwurfsraumexploration die Zuordnung von Tasks zu Prozessoren variiert werden soll, ist eine entsprechende Systemsoftwareschicht unabdingbar, um Tasks flexibel auf beliebigen Prozessoren platzieren zu können. Zudem kann die Systemsoftware einen erheblichen Einfluss auf die letztendliche Qualität des Systems haben, weshalb es wichtig ist, auch die Systemsoftware in den Optimierungsprozess der Werkzeugkette miteinzubeziehen. Zum einen bringt die Systemsoftware natürlich einen zu beachtenden Overhead bezüglich Laufzeit und Code-Größe mit, aber zum anderen kann beispielsweise auch die gewählte *Scheduling*-Strategie beträchtliche Auswirkungen darauf haben, ob Tasks ihre gegebenen zeitlichen Schranken einhalten können oder nicht. Wenn die Systemsoftware nicht in den Konfigurations- bzw. Optimierungsprozess miteinbezogen wird, kann es deshalb zu Abstimmungsproblemen zwischen Hard- und Software kommen oder zu Differenzen zwischen dem durch eine Entwurfsraumexploration entworfenen System und der letztendlichen Implementierung.

Insgesamt beziehen recht wenige Arbeiten in diesem Kontext die Rolle der Systemsoftware in den Entwurfsprozess ein. Dies ist etwas überraschend, da in der Informatik typischerweise Systeme als Schichtenarchitektur aus Hardware und Software betrachtet werden, bei denen auch die Systemsoftware diverse Schichten zwischen Anwendung und Hardware stellt [Tan05]. Einige der Werkzeuge setzen zumindest eine dünne Systemsoftwareschicht zur Kommunikation mittels Kommunikationsprimitiven [TBHH07] oder MPI-artigen Bibliotheken [JCRR+09, WHH10] ein, um den Anwendungsentwickler bei der Umsetzung von

Kommunikation zwischen Tasks zu unterstützen und so zumindest bezüglich der Kommunikation eine Abstraktion zu schaffen.

In dem Werkzeug HeMPS [CdOCM09] wird ein Mikrokern-Betriebssystem zur Unterstützung von *Multitasking* auf dem konfigurierbaren Multiprozessorsystem verwendet. Zudem sorgen die zur Verfügung gestellten Kommunikationsprimitiven des Mikrokerns dafür, dass die Anwendung unabhängig von der gewählten Zuordnung von Tasks zu Prozessoren implementiert werden kann. Die Zuordnung von Tasks zu Prozessoren bestimmt bei diesem Ansatz der Entwickler über eine graphische Oberfläche. Anschließend kann das konfigurierte System über eine Simulation bewertet und gegebenenfalls durch Änderungen an der Zuordnung verfeinert werden. Das Werkzeug Hellfire [AJMH13] bezieht neben der Hardwarearchitektur auch ein Echtzeitbetriebssystem direkt in den Konfigurationsprozess mit ein. Das verwendete hochkonfigurierbare Betriebssystem HellfireOS ist durch den Austausch der Hardwareabstraktionsschicht auf verschiedenen Architekturen einsetzbar und bietet dem Entwickler simple Kommunikationsprimitiven zum Senden und Empfangen von Nachrichten. Das Betriebssystem und die Hardware können in dem Hellfire Werkzeug jeweils separat für eine gegebene Anwendung maßgeschneidert werden, die Konfiguration der Schichten liegt aber in der Verantwortung des Entwicklers. Nachdem das Betriebssystem und die Hardware konfiguriert wurden, kann das System generiert und simuliert werden. Das von der Simulation gelieferte Feedback zur aktuellen Konfiguration kann dann von dem Entwickler zur Abstimmung des Systems genutzt werden. Open-Scale [BBA+11] setzt ein Echtzeitbetriebssystem auf den Knoten des konfigurierbaren NoCs ein. Ähnlich wie das HellfireOS, muss das Betriebssystem auch bei diesem Ansatz vom Entwickler durch die Auswahl von verschiedenen Konfigurierungspunkten für die jeweilige Anwendung angepasst werden. Eine Auswertung des Systems zur Bestimmung der Performanz findet jedoch nicht statt.

Gauthier *et al.* verfolgen die automatisierte Generierung eines anwendungsspezifischen Betriebssystems [GYJ01] für ein heterogenes Multiprozessorsystem [LYBJ01]. Um das Betriebssystem zu generieren, werden vom Entwickler verschiedene Beschreibungen zum System benötigt: eine abstrakte Task-Beschreibung, eine strukturelle Beschreibung der Architektur und eine Tabelle zur Zuordnung der Kommunikationsmethoden. Die Task-Beschreibung benutzt zunächst abstrakte Schnittstellen zur Kommunikation, welche dann im Laufe der Betriebssystemgenerierung durch Systemaufrufe des Betriebssystems und damit konkreten Kommunikationsmethoden ersetzt werden. Am Ende der Generierung fällt ein maßgeschneidertes Betriebssystem, die transformierten Tasks und Makefiles zur Übersetzung der Softwareschichten heraus. Im Vergleich zu dem in dieser Arbeit verfolgten Ansatz, strebt der Ansatz von Gauthier *et al.* im Grunde den umgekehrten Weg an, denn hier ist die Spezifikation der Hardwarestruktur eine Voraussetzung, um ein passendes Betriebssystem zu generieren. Aus der Perspektive eines Anwendungsentwicklers ist dieser Weg nicht optimal, da die Software aus der Hardware abgeleitet wird und die Domäne der Hardwareentwicklung so nicht vor dem Entwickler verborgen werden kann.

Das bereits in Unterkapitel 3.1.2 diskutierte Werkzeug Koski [KKO+06] setzt das konfigurierbare Echtzeitbetriebssystem eCos zum *Scheduling* der Tasks auf den einzelnen Prozessoren ein. eCos ist Bestandteil von Koskis Softwarebibliothek und wird nach der Optimierung der Task-Zuordnung und der Architektur automatisiert generiert. Jede der für die einzelnen Prozessoren generierte Instanz von eCos beinhaltet die gesamten Tasks, wobei nur die jeweils zugeordneten Tasks von den eCos-Instanzen ausgeführt werden und alle anderen Tasks deaktiviert sind. Zudem kennt jede eCos-Instanz die vollständige Zuordnung von Tasks zu Prozessoren bzw. zu eCos-Instanzen und kann so die Zielorte der adressierten Tasks auflösen.

3.2.1 Fazit

Die Systemsoftware wird zwar vereinzelt in Werkzeugen zur Abstraktion, Kommunikation oder zum *Scheduling* von Tasks eingesetzt und lässt sich auch manuell oder automatisiert konfigurieren und generieren, aber der Fokus liegt bei den Werkzeugen zumeist auf der Hardwareplattform oder die Systemsoftware ist nicht direkt in eine Entwurfsraumexploration eingebunden. Zum Beispiel wird bei Werkzeugen mit Möglichkeiten zur Evaluation des Systems der Speicherplatzbedarf der Systemsoftware nicht berücksichtigt [CdOCM09, AJMH13, KKO+06]. Der in dieser Arbeit angestrebte Ansatz soll die Systemsoftware vollständig in die Entwurfsraumexploration integrieren und neue Konzepte vorstellen, wie Systemsoftware- und Hardware-Produktlinie gemeinsam und transparent in einem Konfigurationsprozess maßgeschneidert werden können.

3.3 Entwurfsraumexploration

Einige der Ansätze beschäftigen sich neben der Maßschneiderung von *Multi-* bzw. *Manycore*-Systemen auch mit der Suche nach einem optimalen Kandidaten für die gegebenen Anforderungen. Oftmals gibt es bei der Suche verschiedene Kriterien nach denen die Kandidaten bewertet werden müssen, wie zum Beispiel zeitliche Eigenschaften, die benötigte Energie oder die belegten Ressourcen. Diese Kriterien können unter Umständen untereinander konkurrieren und müssen dann gegeneinander abgewogen werden. Durch die Komplexität und die vielen Variationspunkte bei Multiprozessorsystemen ist der Entwurfsraum zu groß, um diesen vollständig zu untersuchen, weshalb sich verschiedene Techniken zur Annäherung an das Optimum etabliert haben. Eine Technik, welche häufig in dem Kontext von Multiprozessorsystemen eingesetzt wird [PEP06, TBHH07], sind evolutionäre Algorithmen [Wei15]. Evolutionäre Algorithmen sind Optimierungsverfahren, welche an die natürliche Evolution angelehnt sind. Sie versuchen, die Population von Kandidaten über mehrere Generationen sukzessive zu verbessern und trotzdem eine gewisse Diversität in den Generationen zu erhalten. Zwei der bekanntesten Vertreter von evolutionären Algorithmen sind SPEA2 [ZLT01] (*Strength Pareto Evolutionary Algorithm 2*) und NSGA-II [DPAM02] (*Nondominated Sorting Ge-*

netic Algorithm II). Eine weitere Heuristik zur Optimierung von multikriteriellen Problemen ist das *Simulated Annealing* [Kir84]. Bei diesem Verfahren wird ein Kandidat zunächst zufällig bestimmt und bewertet. Anschließend wird ein weiterer Kandidat aus dem Umfeld mit diesem verglichen. Falls der neue Kandidat bessere Eigenschaften besitzt, wird der erste Kandidat durch den neuen Kandidaten ersetzt. Aber auch wenn der neue Kandidat schlechter ist, besteht eine gewisse Wahrscheinlichkeit, dass diese Ersetzung stattfindet. Die Wahrscheinlichkeit für den Tausch mit einem schlechteren Kandidaten nimmt allerdings im Zuge des Verfahrens immer weiter ab. Dadurch ist die Wahrscheinlichkeit, ein lokales Optimum wieder verlassen zu können, zu Beginn relativ hoch und nimmt mit voranschreitender Zeit immer weiter ab. Das *Simulated Annealing* wird zum Beispiel in der Werkzeugkette Koski [KKO+06] eingesetzt und mit weiteren Verfahren kombiniert.
Um die Kandidaten an anwendungsspezifische Anforderungen anpassen zu können, bieten die Werkzeuge im Kontext der Multiprozessorsysteme viele verschiedene Variationspunkte. Typischerweise werden in den Werkzeugen Teilmengen der folgenden Parameter variiert:

- Zuordnung von Tasks zu Prozessoren
- Anzahl der Prozessoren
- Typ des Prozessors (bei heterogenen Systemen)
- Kommunikationsstrukturen
- *Scheduling*-Verfahren

Damit der anhand einer Entwurfsraumexploration ermittelte Kandidat sehr nah an der optimalen Lösung liegt, muss eine möglichst große Anzahl an verschiedenen Konfigurationen innerhalb der Entwurfsraumexploration untersucht werden können, weshalb eine schnelle Auswertung der anvisierten Kriterien von großer Bedeutung ist. Insbesondere die Auswertung der zeitlichen Kriterien kann, je nach verwendetem Verfahren, sehr viel Zeit in Anspruch nehmen. Die beiden von HeMPS generierten Modelle auf Registertransfer- und Instruktionssatzebene wurden beispielsweise bezüglich der Simulationszeiten für eine Anwendung zur Dekodierung von Motion-JPEG überprüft. Die Simulation auf Registertransferebene benötigte dabei fast die zwölffache Zeit im Vergleich zum Instruktionssatzsimulator [CdOCM09]. Neben Simulatoren finden sich in der Literatur auch einige analytische Ansätze zur Bestimmung der Performanz. Im Folgenden sollen die vertretenden Techniken zur Auswertung der Systeme näher vorgestellt und diskutiert werden.

3.3.1 Simulationen

Wie eingangs bereits erwähnt, können Systeme auf verschiedenen Abstraktionsebenen simuliert werden. Je höher die gewählte Ebene liegt, desto schneller lassen

sich die Systeme typischerweise simulieren, wodurch allerdings unter Umständen die Präzision der ermittelten Ergebnisse leiden kann. Zahlreiche Werkzeuge setzen zur Auswertung auf SystemC-basierende Instruktionssatzsimulatoren. Zum Beispiel kann in [XPS08] der Anwender eine Architektur spezifizieren und die entsprechenden SystemC-Modelle werden automatisiert zusammengesetzt. Der implementierte Instruktionssatzsimulator setzt bei diesem Ansatz die MicroBlaze-Architektur von Xilinx um. Die manuelle Auswertung der Ergebnisse kann über eine in den Simulator integrierte Visualisierung erfolgen. Ebenso verwendet auch HeMPS [CdOCM09] einen zyklenakkuraten und in SystemC geschriebenen Instruktionssatzsimulator. Ein Instruktionssatzsimulator für homogene Multiprozessorsysteme auf Basis der MIPS I-Architektur wurde von Filho *et al.* [FAMH08] entwickelt. Der Simulator zählt zum einen Zyklen und schätz zum anderen die zu erwartende Energieaufnahme des Systems ab. Neben den Prozessoren und der Kommunikation, können bei diesem Ansatz auch Peripheriekomponenten, wie ein UART simuliert werden. Ein weiterer auf SystemC-basierender Simulator wird in [RBM+04] präsentiert.
NoCMaker [CRJR+09] setzt auf einen in JHDL [BH98] implementierten zyklenakkuraten Simulator zur Bestimmung der Performanz und der Energieaufnahme für ein NoC. Zusätzlich werden bei diesem Ansatz über eine weitere Analyse die benötigten Ressourcen für das NoC abgeschätzt.
Koski [KKO+06] teilt die Entwurfsraumexploration in zwei Phasen ein, um den Overhead durch einen Simulator für die Untersuchung des Entwurfsraums in Grenzen zu halten. In der ersten Phase wird zunächst das Anwendungsmodell analysiert und über Heuristiken die Auswahl an Hardwarekomponenten und die Zuteilung von Tasks optimiert. Dabei wird versucht, die Kommunikation möglichst lokal zu halten und trotzdem die Parallelität der Architekturen auszunutzen. In der zweiten Phase werden dann die Architekturen anhand akkurater Simulationsmodelle optimiert. In dieser Phase werden nur noch kleine Änderungen an der Zuteilung von Tasks zu Prozessoren vorgenommen, um Ausführungszeiten und Kommunikationslatenzen zu verbessern.
Daedalus [TNS+07] nutzt zur Evaluierung der Architekturperformanz das Explorationswerkzeug Sesame [PEP06]. Sesame führt zur Bewertung der einzelnen Kandidaten in der Entwurfsraumexploration zunächst eine sehr kurze Simulation auf Systemebene durch. Erst für vielversprechende Kandidaten am Ende der Untersuchung wird eine detailliertere und damit auch zeitaufwendigere Simulation zur exakten Bestimmung der Performanz genutzt.

3.3.2 Analytische Methoden

Eine andere Herangehensweise zur Bewertung solcher Systeme bieten analytische Methoden aus Forschung und Industrie. Thiele *et al.* präsentieren beispielsweise eine modulare Performanzanalyse basierend auf dem Real-Time Calculus [TCN00] und nutzen diese zur Optimierung der Task-Zuordnung in dem Werkzeug DOL [TBHH07]. Diese Methode erlaubt die sehr schnelle Bewertung von

Rechen- und Kommunikationszeiten auf Systemebene und eignet sich dadurch speziell im Kontext von Entwurfsraumexplorationen, bei denen iterativ eine große Anzahl von Systemen bewertet werden muss. Auch andere Methoden aus dem Bereich der *Schedulability*-Analyse wie MAST [HGGPGDM01] bzw. MAST 2 [HGD+13], Cheddar [SLNM04] oder dem kommerziellen Werkzeug SymTA/S[2], können zur Bewertung der Task-Zuordnung auf unterschiedlichen Hardwarearchitekturen eingesetzt werden.
Einen Ansatz zur Modellierung und Verifikation von Echtzeitsystemen auf Basis von Echtzeitautomaten (engl. *Timed Automata*) bietet das Werkzeug Uppaal [LPY97]. Mit Uppaal lassen sich Systeme mittels Netzwerken von Echtzeitautomaten, also um Zeitvariablen erweiterte endliche Automaten, beschreiben. Madsen *et al.* benutzen Uppaal zum Beispiel zur Verifikation von Multiprozessorsystemen auf Systemebene [MHK+08], um Aussagen über zeitliche Eigenschaften, Speicherbedarf oder Energieaufnahme treffen zu können. Der Entwickler spezifiziert die Anwendung als Satz von Task-Graphen und gibt die Hardwarearchitektur sowie die Task-Zuordnung an. Das Werkzeug generiert dann automatisiert die passenden Echtzeitautomaten, welche mit Uppaal ausgewertet werden können.

3.3.3 Fazit

Eine schnelle Bewertung der Architekturen ist für eine automatisierte Entwurfsraumexploration unausweichlich. Werkzeuge wie Daedalus und Koski setzen deshalb auf ein zweistufiges Vorgehen und bewerten die zeitlichen Eigenschaften einer Architektur erst detailliert, nachdem der Entwurfsraum erheblich verkleinert wurde. Simulatoren auf Instruktionssatzebene, wie in vielen Werkzeugen zur manuellen Entwurfsraumexploration genutzt, sind für eine automatisierte Exploration bei tausenden zu bewertenden Kandidaten zu langsam. Der Instruktionssatzsimulator von HeMPS beispielsweise benötigt zur Auswertung der Motion-JPEG-Anwendung für nur einen Kandidaten über 6 Minuten [CdOCM09]. Eine Alternative zu Simulatoren sind die vorgestellten analytischen Methoden. Diese bieten neben einer recht schnellen Auswertung auch noch Vorteile bezüglich der Untersuchung von schwer identifizierbaren Randfällen, welche bei einem einzigen Simulatorlauf nicht zu entdecken sind. Somit kann die Einhaltung von gegebenen Schranken nicht nur vermutet, sondern garantiert werden. Auch die von einer Hardwarearchitektur belegten Ressourcen auf einem FPGA müssen aufgrund der langen Synthesezeiten für Multiprozessorsysteme über Modelle abgeschätzt werden und können nicht direkt aus den Synthesewerkzeugen der FPGA Hersteller ausgelesen werden.

3.4 Zusammenfassung

Wie in diesem Kapitel gezeigt, gibt es viele verschiedenartige Ansätze und Lösungen, um Hardwarearchitekturen automatisiert zu generieren oder, in einigen we-

[2] SymTA/S: `https://www.symtavision.com/symtas.html`

nigen Fällen, diese auch mittels der entwickelten Automatismen anwendungsspezifisch zu optimieren. Des Weiteren wurden auch die unterschiedlichen Abstraktionen und Schnittstellen vorgestellt, mit denen ein potenzieller Nutzer dieser Werkzeuge umgehen muss, um die entsprechenden Systeme erzeugen zu können. Viele dieser Ansätze überlassen dem Anwender zu viele Details und auch Entwurfsentscheidungen, welche von einem Anwendungsentwickler nur schwer abzuschätzen sind. Um dieses Problem zu lösen, werden in dieser Arbeit statische Analysen zur automatisierten Extraktion von Konfigurationsinformationen aus den Zugriffsmustern der Schnittstellen zwischen den Schichten eingesetzt.

Der Weg über die Schnittstellen bietet gleich mehrere Vorteile. Zum einen ist bei einer konsequenten Anwendung dieses Schemas und der schrittweisen Konfigurierung von der obersten bis zur untersten Schicht, jede einzelne Schicht des Systems, einschließlich der Systemsoftware, in diesen durchgängigen Konfigurationsprozess einbezogen und zum anderen kann durch eine entsprechende Schnittstelle auf der Anwendungsebene, zugleich die Programmierung von parallelen Hardwarearchitekturen unterstützt werden. Bei einigen der vorgestellten Werkzeuge zur Maßschneiderung von Hardwarearchitekturen ist dieser Trend bereits anhand des Wandels von einfachen Sende- und Empfangsprimitiven zu MPI-artigen, kollektiven Kommunikationsoperationen [JCRR+09, WHH10] zu beobachten, denn ohne eine entsprechende Unterstützung für solche komplexen Kommunikationsoperationen durch die Systemsoftware, sind *Manycore*-Systeme sonst nicht mehr handhabbar und die verfügbare Leistung durch immer mehr Chipfläche nicht abrufbar. Allerdings erfordern diese Werkzeuge, im Gegensatz zu dieser Arbeit, trotz der MPI-artigen Abstraktionsschicht noch die explizite Konfigurierung der Hardwarearchitektur durch den Anwender.

Zudem müssen sich auch die Werkzeuge zur automatisierten Entwurfsraumexploration diesem Wandel stellen und Konzepte bereithalten, um für die immer komplexer werdenden Anwendungen passend abgestimmte Systeme zu erzeugen. Die in dieser Arbeit angestrebte Lösung sieht dazu vor – neben der Anzahl der Prozessoren, der Zuordnung zu den Prozessoren und der Maßschneiderung der Kommunikationsstrukturen – auch den Grad der Parallelität einer Anwendung direkt in der Entwurfsraumexploration zu bestimmen. So kann die Entwurfsraumexploration direkten Einfluss auf die Parallelität der Anwendung nehmen, um so beispielsweise die vorgegebenen zeitlichen Bedingungen einhalten zu können.

Bisher nicht thematisiert wurde in diesen Arbeiten, wie der eigentliche Entwicklungsprozess der dort eingesetzten konfigurierbaren Hardwareplattformen aussehen könnte, also mit welchen Methodiken sich diese Familien von Hardwarearchitekturen gezielt entwickeln lassen, denn im Vergleich zu einfachen Hardwarearchitekturen, sind für die Entwicklung von konfigurierbaren Hardwareplattformen weitere Faktoren zu berücksichtigen. So ist beispielsweise für eine konfigurierbare Hardwareplattform zu entscheiden, welche Komponenten für die zukünftig abzuleitenden Varianten überhaupt in die konfigurierbare Hardwareplattform integriert werden sollten, welche Techniken sich überhaupt dazu eignen, um die benötigte Variabilität der konfigurierbaren Hardwareplattform zu implementie-

ren oder auch, wie die Variabilität der konfigurierbaren Hardwareplattform überhaupt repräsentiert werden könnte. In dem folgenden Kapitel wird deshalb der Entwicklungsprozess von konfigurierbaren Hardwareplattformen detailliert betrachtet und zudem diskutiert, mit welchen Methodiken sich diese Familien von Hardwarearchitekturen konzipieren und umsetzen lassen. Da es in der Softwareentwicklung bereits zahlreiche Forschungsbeiträge gibt, welche sich im Kontext von Produktlinien mit diesen Fragestellungen beschäftigen, soll zudem die Anwendbarkeit der Produktlinientechniken auf die Entwicklung von konfigurierbaren Hardwareplattformen untersucht werden. Somit könnten die Produktlinientechniken als Basis dienen, um sowohl die Hardware als auch die Systemsoftware in einem durchgängigen Konfigurationsprozess anwendungsspezifisch abzuleiten und den Entwickler somit bei der Realisierung von komplexen Systemen zu unterstützen.

Kapitel 4

Hardware-Produktlinien

Inhalt

In diesem Kapitel werden die wichtigsten Techniken zur Software-Produktlinienentwicklung aus Unterkapitel 2.2 bezüglich ihrer Anwendbarkeit auf Hardware-Produktlinien untersucht. Als Beispiel dient die LavA Hardware-Produktlinie, deren Entwicklung anhand des Software-Produktlinien Referenzprozesses vorgestellt und diskutiert wird.

4.1 Motivation

Der Einsatz von Produktlinientechniken für Hardware stellt eine vielversprechende Lösung dar, um die Entwicklungszeiten und -kosten für die Entwicklung von Hardware erheblich zu senken und spezialisierte Systeme innerhalb von sehr kurzen Zeitspannen zur Verfügung stellen zu können. Dass der Bedarf an verschiedenen Ausprägungen von Hardware vorhanden ist, lässt sich zum Beispiel an

der AVR-Mikrocontrollerfamilie von Atmel erkennen. Atmel bietet für spezifische Anforderungen verschiedene Derivate der AVR-Mikrocontrollerfamilie an, welche sich im Hinblick auf die zur Verfügung stehende Speichergröße und dem Funktionsumfang unterscheiden. So kann abhängig von den Projektanforderungen der entsprechende Mikrocontroller aus der Familie gewählt werden. Jedoch bieten die angebotenen Derivate eher grobgranulare Abstufungen, wodurch möglicherweise Kompromisse eingegangen werden müssen. Produktlinien auf Basis von Hardwarebeschreibungssprachen bieten die Möglichkeit, die gewünschten Hardwarestrukturen sehr feingranular an die gesetzten Anforderungen anzupassen und diese dementsprechend gezielt zu optimieren.

Das mögliche Potenzial von Hardware-Produktlinien soll nun anhand der im Zuge des LavA-Projektes entstandenen und auf Hardware basierenden transaktionalen Speicherfamilie TMPL (*Transactional Memory Product Line*) demonstriert werden [MASS11]. Transaktionale Speicher werden als vielversprechende Technologie gesehen, um der Nebenläufigkeit in zukünftigen *Multi-* und *Manycore*-Systemen zu begegnen, da sie ohne präventives Sperren von Speicherbereichen verwendet werden können [HM93], denn typischerweise werden in Multiprozessorsystemen Spinlocks eingesetzt, um den Zugriff auf gemeinsame Datenstrukturen zwischen nebenläufigen Tasks zu synchronisieren [SGG08]. Die Verwendung von Spinlocks ist jedoch komplex sowie fehleranfällig und kann bei falscher Verwendung zu Problemen wie Verklemmungen oder korrupten Datenstrukturen führen. Zudem kann das Sperren von Speicherbereichen, abhängig von der verwendeten Granularität der Spinlocks, die Performanz des Systems stark beeinträchtigen. So kann es bei der Nutzung von sehr grobgranularen Spinlocks beispielsweise dazu kommen, dass zwei eigentlich auf disjunkten Speicherbereichen arbeitende Tasks nicht parallel auf ihre jeweiligen Datenstrukturen zugreifen können. Transaktionale Speicher vermeiden diese Problematik, denn ähnlich wie bei Zugriffen auf Datenbanken, werden bei transaktionalen Speichern Folgen von Lese- und Schreiboperationen in Transaktionen gekapselt. Die Transaktionen von einzelnen Tasks können dabei parallel durchgeführt werden und müssen nur im Falle eines Konfliktes zwischen verschiedenen Transaktionen zurückgerollt und im Anschluss wiederholt werden.

Bei einem genaueren Blick auf die Domäne der transaktionalen Speicher zeigt sich, dass es diverse grundlegende Strategien gibt, nach denen ein transaktionaler Speicher umgesetzt werden kann. Zentrale Strategien bei einem transaktionalen Speicher sind zum Beispiel die Konflikterkennung, die Versionierung oder die Konfliktauflösung.

Konflikterkennung Die Konflikterkennung gibt an, zu welchem Zeitpunkt die Transaktionen hinsichtlich aufgetretener Konflikte überprüft werden. Bei einer optimistischen Konflikterkennung geschieht die Überprüfung erst nach dem Abschluss einer jeden Transaktion, wohingegen bei der pessimistischen Konflikterkennung nach jedem Zugriff erneut auf Konflikte getestet wird.

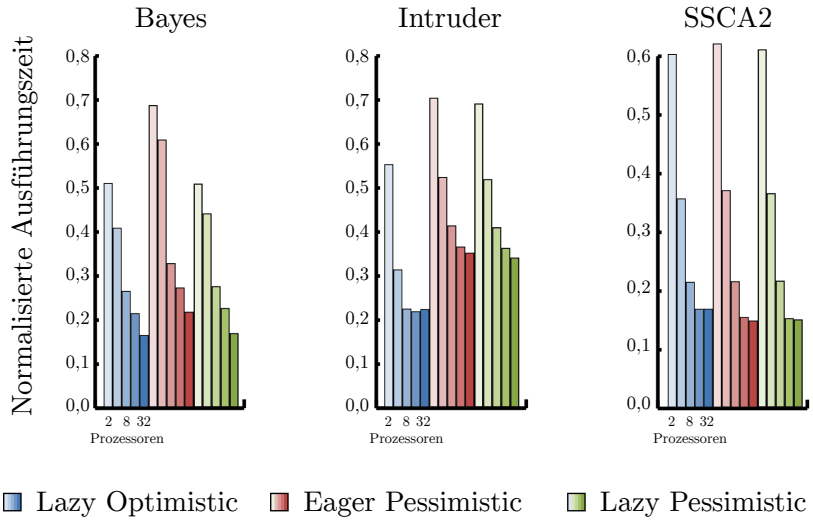


Abbildung 4.1: Ergebnisse für verschiedene transaktionale Speicher aus [McD09]

Versionierung Auch die Versionierung bietet zwei mögliche Optionen, denn diese gibt an, ob die modifizierten Speicherinhalte direkt in den Speicher geschrieben werden (*Eager*) oder die Änderungen erst zum Ende der Transaktion in den Speicher übernommen werden (*Lazy*). Für den ersten Fall werden die alten Speicherinhalte zunächst in einem separaten Puffer gehalten, wohingegen für den zweiten Fall die neuen Werte vorläufig separat gesichert werden.

Konfliktauflösung Die Konfliktauflösung zur Auswahl der abzubrechenden Transaktion bietet für transaktionale Speicher ebenfalls Möglichkeiten zur Variation. Zum Beispiel kann eine Konfliktauflösung verwendet werden, bei der die jüngste der in konfliktstehenden Transaktionen abgebrochen wird oder eine Konfliktauflösung, bei der die konfliktverursachende Transaktion fortgesetzt werden darf.

Für Produktlinien sind transaktionale Speicher interessant, da die Performanz dieser unterschiedlichen Strategien merklich von dem Anwendungsszenario abhängt. Eine ausführliche Auswertung von verschiedenen Strategien mittels des STAMP-Benchmarks [MCKO08] wurde von McDonald [McD09] durchgeführt. Ein Auszug zu diesen Ergebnissen wird in Abbildung 4.1 präsentiert. Diese Auswertung basiert auf drei verschiedenen Ausprägungen von transaktionalen Speichern, den Varianten *Lazy Optimistic*, *Eager Pessimistic* und *Lazy Pessimistic*. Für die Bayes-Anwendung weisen die beiden *Lazy* Varianten sehr ähnliche Ergebnisse auf, wohingegen die *Eager Pessimistic* Variante recht deutlich abfällt (größere normalisierte Ausführungszeit). Bei der Intruder-Anwendung übertrifft die *Lazy Optimistic* Variante die anderen beiden Varianten recht deutlich. Wobei

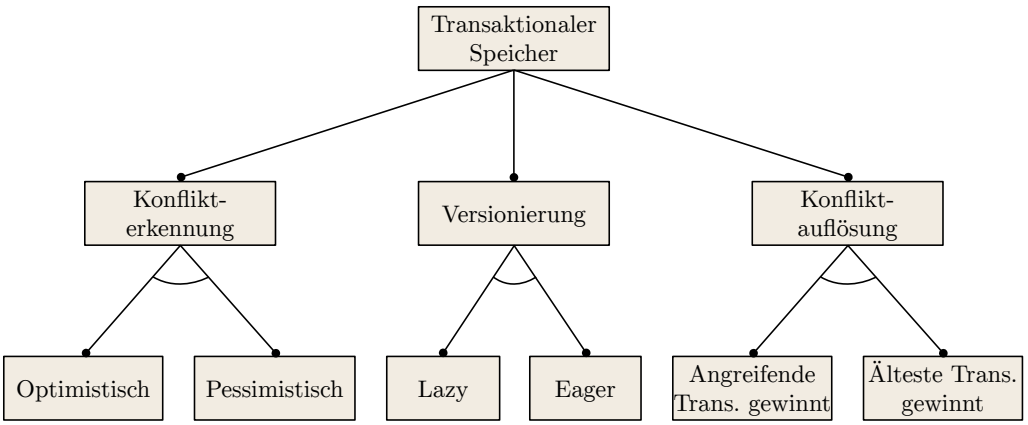


Abbildung 4.2: Merkmaldiagramm für TMPL

diese wiederum bei der SSCA2-Anwendung die *Lazy Optimistic* Variante ab einer Prozessoranzahl von 16 leicht übertreffen können.

Die Ergebnisse zeigen, dass es nicht die eine, ideale Variante für alle möglichen Szenarien gibt und eine Produktlinie von transaktionalen Speichern zur Optimierung der Performanz durchaus sinnvoll ist. Die weiter oben im Kontext der transaktionalen Speicher analysierten variablen Anforderungen, wie zum Beispiel die unterschiedlichen Möglichkeiten zur Konflikterkennung, werden in der Software-Produktlinienentwicklung über die bereits angesprochenen Merkmaldiagramme (siehe Unterkapitel 2.2.2.2) visualisiert. Ein entsprechendes Merkmaldiagramm für die transaktionale Speicherfamilie ist in Abbildung 4.2 dargestellt. Die einzelnen Eigenschaften der Speicherfamilie, wie Konflikterkennung, Versionierung und Konfliktauflösung, werden für jede Instanz der Produktlinie zwingend benötigt und die jeweils untergeordneten Eigenschaften dieser schließen sich gegenseitig aus. Für die Konflikterkennung bedeutet dies beispielsweise, dass nur eine der Eigenschaften Teil eines abgeleiteten Produktes sein kann, entweder das Merkmal zur optimistischen oder zur pessimistischen Konflikterkennung. Ohne eine Konflikterkennung wäre der transaktionale Speicher nicht funktionsfähig und auch beide Strategien lassen sich nicht in einem System zusammen umsetzen.

Schon dieser kurze Abstecher zu der transaktionalen Speicherfamilie verdeutlicht, welche Vorteile eine Hardware-Produktlinie mit sich bringen kann. Ob sich die einzelnen Techniken aus den drei Phasen des Software-Produktlinien Referenzprozesses (Domänenanalyse, -entwurf und -implementierung) auch auf die Entwicklung einer Multiprozessor-Plattform übertragen lassen, wird nun in den nachfolgenden Kapiteln untersucht. Dazu wird exemplarisch die LavA Hardware-Produktlinie verwendet und diese anhand der drei Phasen im Detail vorgestellt.

4.2 Domänenanalyse

Die Entwicklung einer Produktlinie beginnt zunächst mit der Domänenanalyse, in der die Ausrichtung und der Umfang der zu entwerfenden Produktlinie entsprechend der gestellten Anforderungen festgelegt sowie modelliert wird. Hier liegt auch ein entscheidender Unterschied im Vergleich zu der traditionellen Softwareentwicklung. In der traditionellen Softwareentwicklung wird typischerweise eine spezifische Lösung für ein konkretes Problem umgesetzt. Anders sieht es bei der Entwicklung einer Produktlinie für eine bestimmte Domäne aus, in der eine Menge von Produkten für diese Domäne entstehen sollen. Hier muss sehr genau abgewogen werden, welche Bereiche der Domäne die Produktlinie abdecken soll, damit gegenwärtige und möglicherweise auch zukünftige Kunden mit der Produktlinie bedient werden können. Andernfalls wäre die Produktlinie nicht auf die Bedürfnisse des Marktes abgestimmt und würde sich dementsprechend nicht rentieren als auch dem erhöhten Entwicklungsaufwand nicht gerecht werden. Auch eine zu breit angelegte Produktlinie, bei der ein erheblicher Teil der Entwicklungszeit in später nicht verwendbare Komponenten fließt, muss Aufgrund der zusätzlich entstehenden Kosten vermieden werden.
Ziel der Domänenanalyse ist es, diese Probleme in dieser frühen Phase der Entwicklung zu minimieren und die angestrebte Variabilität der Produktlinie über Modelle zu dokumentieren. Die Domänenanalyse wird dazu typischerweise in die Domänenabgrenzung (engl. *Domain Scoping*) und die Domänenmodellierung unterteilt [CE00].

4.2.1 Domänenabgrenzung

Multi- und *Manycore*-Systeme bieten vielzählige Facetten und dementsprechend zahlreiche Richtungen, in die eine Produktlinie in diesem Kontext entwickelt werden kann. Da die vollständige Abdeckung dieses Bereiches durch eine Produktlinie nicht möglich ist, muss die Domäne, in der sich die LavA Hardware-Produktlinie bewegen soll, sorgfältig anhand der gestellten Anforderungen abgesteckt werden. Dabei kann auf das Wissen von Experten in dieser Domäne zurückgegriffen oder auch existierende *Multi-* und *Manycore*-Systeme analysiert werden, um die Grenzen der Domäne zu schärfen. Im Folgenden werden die Anforderungen an die LavA Hardware-Produktlinie im Detail vorgestellt, um die Domäne der Produktlinie einzugrenzen.

Basissatz an Merkmalen für Demonstrator In erster Linie soll die LavA Hardware-Produktlinie in dieser Arbeit als Demonstrator dienen und zunächst einen Basissatz an konfigurierbaren Merkmalen aufweisen. Ziel ist es, sowohl die Anwendbarkeit der Produktlinientechniken auf die Hardwareentwicklung zu untersuchen, als auch für den LavA-Ansatz (Unterkapitel 1.2) eine Plattform zur automatisierten Generierung von *Multi-* und *Manycore*-Systemen zu realisieren.

Nicht-funktionale Eigenschaften Die Produktlinie soll sich bezüglich verschiedener elementarer, nicht-funktionaler Eigenschaften maßschneidern lassen. Im Bereich der Eingebetteten Systeme liegt der Fokus in der Regel auf Eigenschaften, wie Zeit, Ressourcen oder Energie. Da jedoch die Energieaufnahme für Prototypen eine untergeordnete Rolle spielt und die Erstellung von Energiemodellen für Hardware auf FPGAs ein eigener Forschungszweig ist, wird für diese Produktlinie die Energieaufnahme nicht betrachtet. Im Gegenzug sollen sich allerdings Eigenschaften, wie Zeit und Ressourcen, über die Anpassung von Parametern der Produktlinie indirekt steuern lassen.

Prozessorunterstützung Die LavA Hardware-Produktlinie soll über stark heterogene Eigenschaften verfügen. Dazu sollen mehrere Prozessortypen mit unterschiedlichen Leistungskriterien integriert werden, um für unterschiedlich rechenintensive Anwendungen den passenden Prozessortypen bezüglich Rechenzeit und Ressourcen auswählen zu können.

Kommunikationsstrukturen Auch für die Netzwerkstrukturen zwischen den Prozessoren muss, je nach Anwendung, ein Kompromiss zwischen der Leistungsfähigkeit und den benötigten Ressourcen gefunden werden. Für Anwendungen mit geringen Anforderungen an die Kommunikation könnten einfache Busse zwischen den Teilnehmern genutzt werden. Im Gegensatz dazu, könnte für Szenarien mit intensiver und möglicherweise paralleler Kommunikation, ein leistungsfähiges, aber zugleich auch ressourcen-hungriges NoC (*Network on Chip*) integriert werden.

Skalierbarkeit der Systeme Die Anzahl der Prozessoren soll nahezu uneingeschränkt skaliert werden können und einzig durch die FPGA-Ressourcen limitiert werden, damit *Manycore*-Systeme, ebenso wie einfache Systeme, generiert werden können. Des Weiteren sollen auch die unterstützten Kommunikationsstrukturen zur beliebigen Vernetzung der Prozessoren in einem System genutzt werden können und nicht auf eine Kommunikationsstruktur pro System beschränkt werden, wodurch auch für größere Systeme die Kommunikation keinen Engpass bezüglich der Systemperformanz darstellen muss.

Gemeinsame Schnittstelle für Peripheriekomponenten Neben Prozessoren und Kommunikation soll die Produktlinie auch diverse Peripheriekomponenten umfassen. Die Peripheriekomponenten sollen dem System die Interaktion mit der Umgebung erlauben und den Funktionsumfang der Systeme erweitern. Damit die Peripheriekomponenten universell an verschiedenen Prozessortypen eingesetzt werden können, sollen diese über eine einheitliche Schnittstelle zu den Prozessoren verfügen.

Einbindung in automatisierten Designprozess Da die LavA Hardware-Produktlinie in einen automatisierten Entwurfsprozess eingebunden werden soll,

um die Hardwarekonfiguration entsprechend zu optimieren, muss die Hardware vollständig generiert werden können. Deshalb können nach der Generierung der wiederverwendbaren Hardwarekomponenten nicht – wie typischerweise in dem Software-Produktlinien Referenzprozess in Abbildung 2.9 vorgeschlagen – weitere produktspezifische Teile der Hardware in der Anwendungsentwicklung hinzugefügt werden. Neue wiederverwendbare Komponenten sollen allerdings weiterhin über die Rückführung in die Produktlinie integriert werden, um diese zu erweitern.

Unterstützung von verschiedenen FPGA-Typen Die Produktlinie soll zudem verschiedene FPGA-Typen unterstützen, um die Hardwaresysteme gezielt für verschiedene FPGAs optimieren zu können und die zum Teil individuellen Ressourcen der diversen FPGA-Typen ausnutzen zu können. Unterschiede kann es zum Beispiel zwischen den FPGA-Familien bezüglich der BRAMs geben, selbst dann, wenn die FPGA-Familien vom selben Hersteller stammen. Für die LavA Hardware-Produktlinie soll die Unterstützung zunächst auf FPGA-Familien von Xilinx beschränkt werden.

Hardwarebeschreibungssprachen Damit die im weiteren Verlauf dieses Kapitels zu untersuchenden Techniken zur Unterstützung von Variabilität im Lösungsraum (siehe Unterkapitel 4.4) auf eine einheitliche Zielsprache angewendet werden können, soll die eingesetzte Sprache zur Beschreibung der LavA-Hardwarekomponenten auf VHDL beschränkt werden.

4.2.2 Domänenmodellierung

Nachdem die Grenzen der Domäne definiert und die Anforderungen an die Produktlinie festgelegt wurden, kann in der Domänenmodellierung die Produktlinie weiter präzisiert und die Variabilität der Produktlinie mittels eines Modells beschrieben und visualisiert werden. In einem ersten Schritt müssen dazu die zuvor festgelegten Anforderungen ausgewertet werden. Ziel dieser Analyse ist sowohl die Identifizierung von Merkmalen als auch die Bestimmung der Beziehungen zwischen den Merkmalen. Die Merkmale beschreiben dabei die Gemeinsamkeiten und Unterschiede – also die Variabilität – der angestrebten Produktlinie.

4.2.2.1 Identifizierung der Merkmale

Ein heterogenes *Multi-* bzw. *Manycore*-System setzt sich aus einer variierenden Anzahl von Prozessoren unterschiedlichen Typs und mit verschiedenen Charakteristika zusammen. Je nach Anforderungen an ein Produkt, sollten dementsprechend die adäquaten Prozessortypen für das System konfiguriert werden können. Die einzelnen Prozessortypen sind also wiederverwendbare Komponenten, welche ihren Weg in die Produktlinie finden sollten und durch entsprechende Merkmale in der Produktlinie repräsentiert werden müssen. Gleiches gilt für

die potenziellen Kommunikationsstrukturen, welche zur Verknüpfung der Prozessoren konfiguriert werden sollen. Auch die diversen Peripheriekomponenten sollen wiederverwendet werden können und werden dementsprechend über eigene Merkmale repräsentiert. Die bisher aufgeführten Merkmale spiegeln explizit die wesentlichen Hardwarebausteine eines *Multi-* bzw. *Manycore*-Systems wieder und entsprechen daher deutlich den ohnehin Komponenten-orientierten, modularisierten Implementierungen in Hardwarebeschreibungssprachen. Weitere, aus den Anforderungen ableitbare Eigenschaften für die Produktlinie, sind zum Beispiel die zu unterstützenden FPGA-Familien. Damit ein abzuleitendes Produkt auf eine bestimmte FPGA-Familie abgestimmt werden kann, müssen auch die FPGA-Familien über konfigurierbare Merkmale repräsentiert werden. Im Gegensatz zu den bisher aufgeführten Merkmalen, resultieren diese Merkmale nicht in einer dedizierten Hardwarekomponente, sondern haben einen eher querscheidenden Charakter mit einem Einfluss auf das gesamte System.

4.2.2.2 Merkmalmodell

Nachdem die elementaren Merkmale der LavA Hardware-Produktlinie identifiziert wurden, können diese in einem Merkmaldiagramm visualisiert und die Beziehungen zwischen den Merkmalen spezifiziert werden. Das zur besseren Übersicht vereinfachte Merkmaldiagramm der LavA Hardware-Produktlinie ist in Abbildung 4.3 dargestellt. Im Wesentlichen sind die wiederverwendbaren Komponenten der Produktlinie in zwei Unterbäume aufgeteilt. Zum einen die Kommunikation und zum anderen die Knoten.
Ein Knoten setzt sich aus genau einem Prozessor und beliebig vielen Peripheriekomponenten zusammen. In einem System können sich dabei ein oder mehr Knoten befinden. Zur Annotierung der Kardinalitäten wurde die von Czarnecki *et al.* vorgeschlagene Notation [CHE05] verwendet. Für jeden Knoten kann alternativ zwischen drei Prozessortypen ausgewählt werden. Die Prozessortypen MB-Lite[1] bzw. MB-Lite+, Plasma MIPS[2] und ZPU[3] sind quelloffen, in VHDL beschrieben und haben unterschiedliche Eigenschaften, bezüglich der Ausführungsgeschwindigkeit als auch dem erforderlichen Ressourcenbedarf. Durch die verschiedenen Eigenschaften der Prozessoren kann ein abzuleitendes Produkt auf das jeweils vorliegende Szenario abgestimmt werden, wodurch den gegebenen Anforderungen an die Produktlinien nachgekommen wird. Zusätzlich können für einzelne Prozessoren optionale Erweiterungen konfiguriert werden, welche in diesem Beispiel durch das Merkmal zur Konfigurierung der JTAG-Unterstützung angedeutet werden. Des Weiteren können für jeden Knoten Peripheriekomponenten konfiguriert werden. Die Peripheriekomponenten können beliebig miteinander kombiniert und auch mehrfach für einen Knoten instanziiert werden. Für die Klasse der Peripheriekomponenten sind in dem Merkmaldiagramm beispielhaft eine Zeitgeber-

[1] MB-Lite (Vorgänger des aktuellen MB-Lite+): `http://opencores.org/project,mblite`
[2] Plasma MIPS: `http://opencores.com/project,plasma`
[3] ZPU: `http://opensource.zylin.com/zpu.htm`

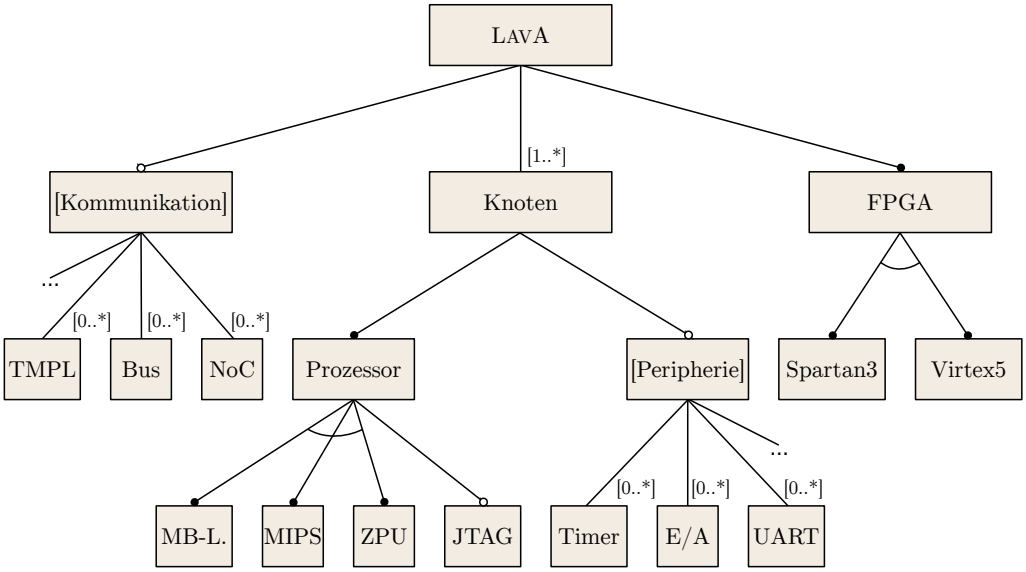


Abbildung 4.3: Merkmaldiagramm der LavA Hardware-Produktlinie

komponente (engl. *Timer*), eine Komponente zur Ein- und Ausgabe und eine UART-Komponente zur seriellen Kommunikation dargestellt. Das Merkmal zur Repräsentation der Peripheriekomponenten ist mit eckigen Klammern markiert, da diese Markierung, in Kombination mit dem weiteren, angedeuteten Merkmal, die Erweiterbarkeit um weitere Peripheriekomponenten darstellt.

Der Unterbaum zur Beschreibung der Variabilität für die Kommunikationstrukturen stellt in diesem Diagramm drei Optionen zur Verfügung: die bereits präsentierte transaktionale Speicherfamilie (TMPL), einen Bus und ein NoC. Die Kommunikation in einem System kann mittels verschiedener Kommunikationsstrukturen realisiert werden. Auf eine weitere Verfeinerung der Merkmale für die einzelnen Kommunikationsstrukturen wird in dem Merkmaldiagramm der Übersicht halber verzichtet. Die Merkmale der transaktionalen Speicherfamilie wurden zudem bereits detailliert in Abbildung 4.2 vorgestellt.

In einem weiteren Unterbaum kann außerdem die FPGA-Familie bestimmt werden, auf der das abzuleitende System verwendet werden soll. In dem abgebildeten Merkmaldiagramm kann alternativ zwischen den beiden Xilinx FPGA-Familien Spartan-3 und Virtex-5 gewählt werden.

Um zwischen beliebigen Merkmalen des Merkmaldiagramms zusätzliche Abhängigkeiten spezifizieren zu können, werden typischerweise Nebenbedinungen (engl. *Constraints*) verwendet. Die Form, in der die Nebenbedingungen definiert werden können, hängt für gewöhnlich von dem eingesetzten Werkzeug ab und kann demzufolge variieren. Für die LavA Hardware-Produktlinie soll beispielsweise mittels einer Nebenbedingung ausgeschlossen werden, dass vollständig isolierte Knoten in einem System konfiguriert werden können. Vollständig isoliert meint in diesem Fall, dass der Knoten weder über eine Peripheriekomponete mit einer Anbindung

an die Umgebung verfügt, noch über einen anderen Knoten an die Umgebung angebunden ist. Solche Systeme können zwar von den Synthesewerkzeugen verarbeitet werden, aber die Optimierung würde für den isolierten Knoten keine Hardwarestrukturen erzeugen. Dementsprechend kann der Entwickler durch die Nebenbedingung bereits auf der Modellebene auf den Konfigurationsfehler hingewiesen werden.

4.2.2.3 Abgleich mit den gestellten Anforderungen

Die anhand des Merkmaldiagramms repräsentierte Variabilität der Produktlinie zeigt bereits an diesem Punkt der Entwicklung deutlich, dass die meisten Anforderungen an die Produktlinie in das Modell eingeflossen und abgedeckt sind. Der Problemraum ist so gestaltet, dass heterogene Systeme dargestellt werden können und auch der Skalierbarkeit des Systems sind auf der Modellebene zunächst keine Grenzen gesetzt. Durch diese Variabilität lassen sich zudem auch die nicht-funktionalen Eigenschaften des abzuleitenden Systems indirekt steuern. Wie eine gemeinsame Schnittstelle zwischen Prozessortypen und Peripheriekomponenten implementiert werden soll, ist zu diesem Zeitpunkt zwar noch nicht definiert, allerdings lässt sich bereits an dem Merkmaldiagramm erkennen, dass die Peripheriekomponenten an beliebigen Prozessoren verwendet werden sollen. Auch die Unterscheidung zwischen den FPGA-Familien ist in dem Merkmaldiagramm modelliert. Der Umfang der repräsentierten Variabilität zeigt zudem, dass ein Basissatz an Merkmalen für den in dieser Arbeit angestrebten Demonstrator vorhanden ist, um maßgeschneiderte *Multi-* bzw. *Manycore*-System zu generieren.

4.2.2.4 Architekturdominanzproblem

Die Domäne der LavA Hardware-Produktlinie wird sehr stark durch die Architektur der abzuleitenden Hardwaresysteme geprägt. Schon bei der Domänenabgrenzung und den dort definierten Anforderungen an die Hardware-Produktlinie ist diese Tendenz zu erkennen, denn die dort formulierten Anforderungen beziehen Hardwarekomponenten und damit Bestandteile der Architektur, wie Prozessortypen oder Kommunikationsstrukturen, mit ein. Bei der Software-Produktlinienentwicklung sind Belange, welche die Architektur betreffen, jedoch üblicherweise nicht Teil der Domänenanalyse, sondern werden erst später im Laufe des Domänenentwurfs betrachtet.
Ungeachtet dessen lassen sich Hardwarekomponeten, wie Prozessoren oder Kommunikationsstrukturen mit den typischen Prozessen aus der Domänenanalyse vereinbaren und beispielsweise über Merkmale im Merkmaldiagramm repräsentieren. Dies trifft allerdings nicht auf alle im Merkmaldiagramm abzubildenden Konfigurationsinformationen zu, denn um die Variabilität der LavA Hardware-Produktlinie vollständig im Merkmaldiagramm abbilden zu können, müssten auch die Verbindungen zwischen den Knoten bzw. Prozessoren über die diversen Kommunikationsstrukturen modellierbar sein. Für diese eher strukturellen Konfigu-

rationsinformationen sind Merkmaldiagramme jedoch nicht die adäquate Wahl und lassen eine entsprechende Modellierung nicht zu.
Diese Problematik gilt zwar gleichermaßen für Software- und Hardware-Produktlinien, allerdings ist diese Architekturdominanz und die damit verbundenen strukturellen Abhängigkeiten häufig im Umfeld der Hardwareentwicklung anzutreffen. Die daraus resultierende Konsequenz ist, dass die vollständige Ableitung eines Produktes anhand einer Merkmalauswahl, wie in Unterkapitel 2.2.3 beschrieben, für die LavA Hardware-Produktlinie nicht möglich ist. Alternativ können die zuvor automatisiert generierten Hardwarekomponenten in der dann erforderlichen Anwendungsentwicklung manuell durch weiteren Quelltext verknüpft werden. Dies ist allerdings mit erheblichem Aufwand verbunden und aufgrund der Komplexität eine mögliche Fehlerquelle. Zudem ist dieses Vorgehen nicht in einem automatisierten Designprozess nutzbar.

4.2.3 Fazit

Wie in diesem Unterkapitel gezeigt, lässt sich das typische Vorgehen aus der Domänenanalyse größtenteils auf die Entwicklung von Hardware-Produktlinien übertragen. Wichtige Bestandteile der Domänenanalyse, wie die Abgrenzung der Domäne oder die systematische Bestimmung der Merkmale, sind ebenso für die Entwicklung einer Hardware-Produktlinie wichtig, damit die Ausrichtung einer Produktlinie zu einem frühen Zeitpunkt festgelegt und dokumentiert werden kann. Auch die Visualisierung der Variabilität mittels Merkmaldiagrammen kann Abhängig von der angestrebten Hardware-Produktlinie potenziell verwendet werden, um den Problemraum abzubilden. Dies wurde zum Beispiel anhand der transaktionalen Speicherfamilie (TMPL) demonstriert. Dennoch bereitet das in dieser Arbeit identifizierte Architekturdominanzproblem bei Produktlinien mit strukturellen Konfigurationsinformationen Schwierigkeiten, wenn die Modellierung mittels Merkmaldiagrammen erfolgen soll und aus einer Merkmalauswahl die konkreten Produkte automatisiert abgeleitet werden sollen. Da diese Problematik die LavA Hardware-Produktlinie betrifft, ist dies ein Problem, welches noch zu lösen sein wird (Unterkapitel 4.3.2).

4.3 Domänenentwurf

Der nächste Schritt in der Entwicklung einer Produktlinie ist der Domänenentwurf. Das Ziel des Domänenentwurfs ist die Entwicklung der Referenzarchitektur und die Ausarbeitung eines Bauplans, nachdem die konkreten Produktinstanzen zusammengesetzt und generiert werden können [CE00].

4.3.1 Referenzarchitektur der LavA Hardware-Produktlinie

Die Referenzarchitektur beschreibt die „Entwurfskomponenten und Variationspunkte der Produktlinie sowie deren Eigenschaften und Beziehungen untereinander“ [BKPS04]. In der Software-Produktlinienentwicklung werden typischerweise

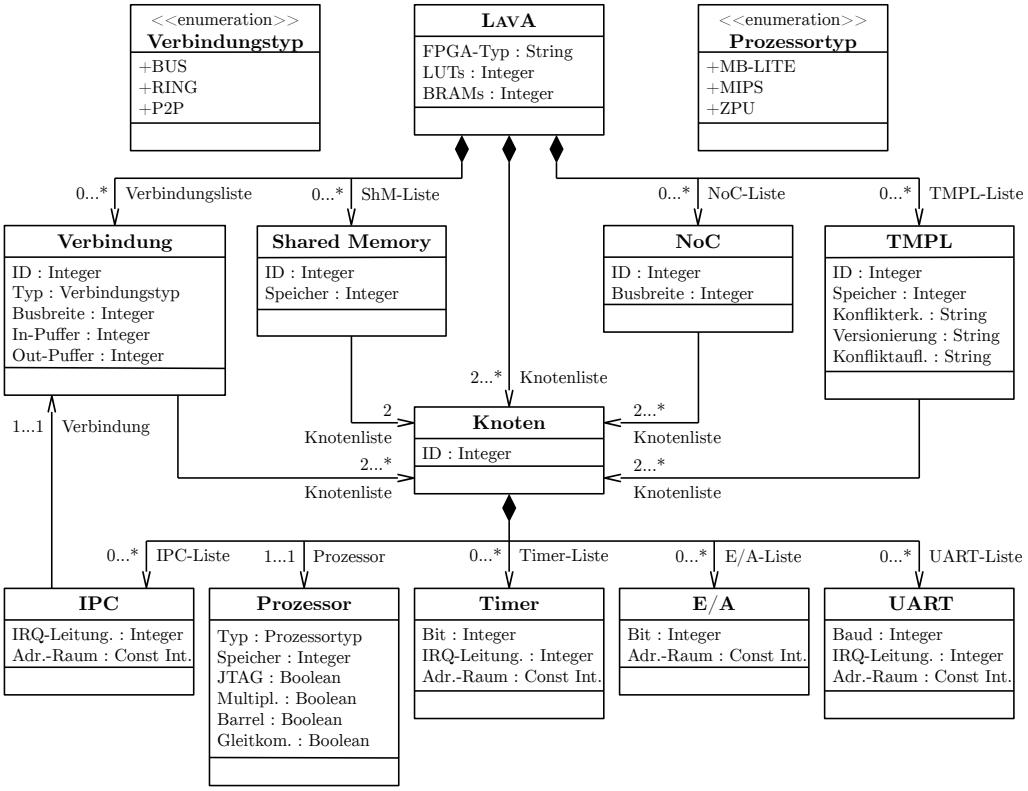


Abbildung 4.4: Referenzarchitektur der LavA Hardware-Produktlinie

Modellierungssprachen, wie zum Beispiel UML, verwendet, um die verschiedenen Aspekte der Architektur zu beschreiben. Auch zur Beschreibung der Referenzarchitektur der LavA Hardware-Produktlinie wird die Notation von Klassendiagrammen verwendet, um die Beziehungen zwischen den Hardwarekomponenten und deren Eigenschaften zu dokumentieren. Die Referenzarchitektur selbst wird dabei über das Metamodell in Abbildung 4.4 repräsentiert. Der Aufbau des Metamodells ähnelt in großen Teilen der Struktur des Merkmaldiagramms aus der Domänenanalyse. Dies liegt im Wesentlichen an der starken Korrelation zwischen den diversen Hardwarekomponenten und den Merkmalen der LavA Hardware-Produktlinie, denn viele der Merkmale spiegeln unmittelbar konfigurierbare Hardwarekomponenten wider.

Die verschiedenen Komponenten der Hardware-Produktlinie werden in dem Metamodell mittels Komposition in einer strikt hierarchischen Struktur angeordnet, weshalb sich die Architektur unmittelbar auf die ebenfalls hierarchisch aufgebauten Implementierungen in Hardwarebeschreibungssprachen übertragen lässt. Alle Attribute und Relationen bilden dabei Variationspunkte. An oberster Stelle der Hierarchie ist die LavA-Komponente und deren Eigenschaften dargestellt. Die Eigenschaften dieser Komponente wirken sich global auf das System aus, wie zum Beispiel der FPGA-Typ oder die zur Verfügung stehenden Ressourcen

des FPGAs. Zudem werden in dieser Komponente die weiteren Hardwarekomponenten zur Kommunikation und die Knoten instanziiert. Alle Kommunikationsstrukturen können bei Bedarf konfiguriert und auch mehrfach im selben System instanziiert werden, um heterogene Kommunikationsstrukturen generieren zu können. Sowohl die NoC-Komponente als auch TMPL können jeweils mit zwei oder mehr Knoten bzw. Prozessoren assoziiert werden. Die gemeinsamen Speicher (engl. *Shared Memory*) können hingegen nur zwischen genau zwei Prozessoren aufgespannt werden. Die Einschränkung auf zwei Prozessoren hat den technischen Hintergrund, dass die gemeinsamen Speicher über BRAMs realisiert werden sollen, welche genau zwei Schnittstellen für den Zugriff besitzen und so keine zusätzliche Komponente zur Arbitrierung der Zugriffe benötigt wird. Zudem kann der gemeinsame Speicher beispielsweise in der Größe parametrisiert und das NoC bezüglich der Breite des Datenbusses konfiguriert werden.

Die Interprozessorkommunikation (engl. *Inter-processor Communication*, IPC) stellt weitere Strukturen zur Kommunikation in Form von Bus-, Ring- oder Punkt-zu-Punkt-Verbindungen bereit. Diese nachrichtenbasierten Methoden nehmen eine Sonderrolle ein, da diese über zwei Komponenten modelliert werden: die Verbindung selbst und die Peripheriekomponente `IPC`. Die Verbindungen bestimmen die grundlegenden Parameter für alle angeschlossenen Knoten, wie die bereits angesprochenen Typen, die Datenbreite oder die Größe der Puffer. Die IPC-Komponenten sind gewöhnliche Peripheriekomponenten, mit deren Hilfe der Prozessor auf die jeweilige Struktur zugreifen kann. Da ein Prozessor durchaus mit verschiedenen IPC-Strukturen verbunden sein kann und so auch mehrere IPC-Peripheriekomponenten an den Prozessor angeschlossen sein können, wird über die präsentierte Modellierung die korrekte Zuordnung sichergestellt.

In jedem Knoten kann genau ein Prozessor instanziiert werden, wobei die zuvor genannten Prozessortypen angeboten werden. Für einen Prozessor kann darüber hinaus die Größe des lokalen Speichers konfiguriert werden und optionale Eigenschaften, wie die Integration eines JTAGs, eines Hardware-Multiplizierers, eines Hardware-*Barrel-Shifters* oder einer Hardware-Gleitkommaeinheit parametrisiert werden.

Die reduzierte Auswahl der in diesem Metamodell präsentierten Peripheriekomponenten umfasst neben der IPC-Komponente auch die bereits im Zusammenhang mit dem Merkmaldiagramm vorgestellten Peripheriekomponenten. Damit die Peripheriekomponenten möglichst universell an verschiedenen Prozessortypen eingesetzt werden können, werden die Register der Peripheriekomponenten über *Memory Mapped I/O* in den Adressraum der Prozessoren eingebunden. Der Datenaustausch erfolgt dabei über das Protokoll und die Schnittstellenspezifikation des offenen Wishbone[4] Bus-Standards. Die Größe des jeweils benötigten Adressraumes wird für die einzelnen Peripheriekomponenten über den konstanten Parameter `Adr.-Raum` in dem Metamodell hinterlegt. Des Weiteren können die Peripheriekomponenten über zusätzliche Parameter maßgeschneidert werden, im

[4]Wishbone: `http://opencores.org/opencores,wishbone`

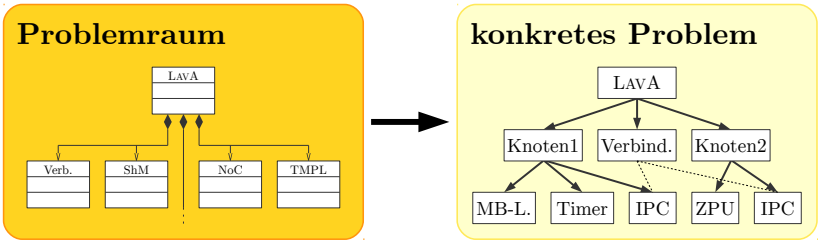


Abbildung 4.5: Modellgetriebene Entwicklung der LavA Hardware-Produktlinie

Falle des UARTs kann zum Beispiel die gewünschte Baudrate statisch konfiguriert werden.

4.3.2 Bauplan der LavA Hardware-Produktlinie

Die Beschreibung der Referenzarchitektur mittels eines Metamodells hat zudem den Vorteil, dass das Metamodell auch den Bauplan zur Ableitung von konkreten Systemen vorgeben kann. Typischerweise werden Metamodelle in der modellgetriebenen Softwareentwicklung verwendet, um auf abstraktem Wege die Struktur für eine Menge von Modellen zu definieren. Übertragen auf die LavA Hardware-Produktlinie, beschreibt das Metamodell also die Menge aller ableitbaren Produkte und die entsprechenden Vorschriften zur Konstruktion dieser auf Modellebene (Abbildung 4.5). Das Metamodell kann also, ähnlich wie ein Merkmalmodell, den Problemraum für eine Domäne aufspannen. Die konkreten Produkte werden dabei allerdings nicht mehr über eine Merkmalauswahl repräsentiert, sondern über ein abgeleitetes Modell, welches individuelle Produktinstanzen von Hardwaresystemen auf einer abstrakten Ebene beschreibt. Das Modell umfasst dabei die benötigten Konfigurationsinformationen zur späteren Generierung des Quelltextes für die Hardwaresysteme, was in der modellgetriebenen Softwareentwicklung typischerweise als Modell-zu-Code-Transformation bezeichnet wird.

4.3.3 Fazit

Ähnlich wie in der Software-Produktlinienentwicklung lassen sich auch in der „Hardwarewelt“ die Architekturen mittels UML-Notationen beschreiben. Dies gilt sowohl für die Hardwarekomponenten selbst als auch für deren Beziehungen und Eigenschaften. Das Metamodell visualisiert zudem, ähnlich wie das Merkmaldiagramm aus Unterkapitel 4.2.2.2, die Variabilität und den Problemraum der Produktlinie. Das Metamodell auf der Ebene der Hardwarekomponenten und das Merkmaldiagramm auf der Ebene der Merkmale. Durch die enge Bindung zwischen den Merkmalen und den Hardwarekomponenten der LavA Hardware-Produktlinie zeigen beide allerdings ein sehr ähnliches Bild. Jedoch lassen sich mit Hilfe des Metamodells auch die Beziehungen zwischen den Prozessoren und der Kommunikation, also die strukturellen Konfigurationsinformationen, modellieren, was mit dem Merkmaldiagramm im Rahmen der LavA Hardware-Produktlinie

nicht möglich ist. Dadurch kann mit Hilfe der Techniken aus der modellgetriebenen Entwicklung das Architekturdominanzproblem gelöst werden, denn die konkreten Produkte der LavA Hardware-Produktlinie können über die Modelle vollständig beschreiben und somit auch in einen automatisierten Designprozess eingebunden werden.

4.4 Domänenimplementierung

Die Implementierung der Produktlinienplattform ist der letzte Schritt der Domänenentwicklung. Die Produktlinienplattform muss die zuvor im Entwurfsprozess definierten Gemeinsamkeiten und Unterschiede der verschiedenen Produktlinienvarianten widerspiegeln können, wozu die in Unterkapitel 2.2.2.1 angesprochenen Techniken zur Umsetzung der Variabilität im Lösungsraum verwendet werden können. Vier dieser Techniken wurden auf ihre Eignung bezüglich der Implementierung von generischen und wiederverwendbaren Hardwarekomponenten im Kontext der LavA Hardware-Produktlinie untersucht: die VHDL-Sprachmittel, die *Frame*-Technologie, die Codegeneratoren aus der modellgetriebenen Entwicklung und die Aspektorientierte Programmierung. Damit die unterschiedlichen Techniken bewertet werden können, wurde die LavA Hardware-Produktlinie oder zumindest Teile dieser mit den aufgeführten Techniken umgesetzt. Dabei entstand für die Hardwarebeschreibungssprache VHDL auch eine eigens entwickelte Erweiterung zur Umsetzung der Aspektorientierten Programmierung in VHDL. Im Folgenden werden diese Techniken im Detail vorgestellt und die entsprechenden Ergebnisse beleuchtet.

4.4.1 VHDL-Sprachmittel

Wie bereits in Unterkapitel 2.1.1.3 vorgestellt, bietet VHDL standardmäßig das Sprachmittel der *Generics*, um Hardwarekomponenten generisch beschreiben und somit flexibel einsetzen zu können. Ähnlich wie bei bedingten Präprozessor-Direktiven aus C oder C++, sind auch *Generics* mit dem Quelltext verwoben und können an verschiedenen Stellen des Quelltextes wirken, um diesen entsprechend zu parametrisieren oder zu modifizieren.
Die in der Motivation zu Hardware-Produktlinien (Unterkapitel 4.1) vorgestellte transaktionale Speicherfamilie (TMPL) ist ein typisches Beispiel für ein komplexes und hochkonfigurierbares Hardwaresystem, dessen Variabilität allein mit den in VHDL zur Verfügung stehenden Sprachmitteln realisiert wurde. Durch den Verzicht auf externe Konfigurationswerkzeuge kann TMPL sehr flexibel und universell in beliebigen VHDL-Projekten als Blackbox eingebunden werden.

4.4.1.1 Schnittstelle der transaktionalen Speicherfamilie

Die Schnittstellenbeschreibung von TMPL ist angesichts der internen Komplexität sehr übersichtlich aufgebaut (Abbildung 4.6). Dies liegt vorrangig daran,

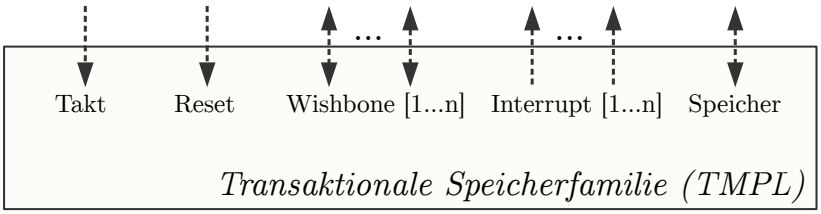


Abbildung 4.6: Schnittstelle der transaktionalen Speicherfamilie

dass die Kommunikation zwischen den angebundenen Knoten bzw. Prozessoren und TMPL ausschließlich über Wishbone-Busse erfolgt. Für jeden Teilnehmer hält TMPL eine eigene Wishbone-Schnittstelle bereit über die sowohl Daten als auch Steuerungsinformationen ausgetauscht werden können. Nur die Signalisierung von *Interrupts* läuft nicht über Wishbone-Busse, sondern wird über dedizierte Leitungen an die Prozessoren übermittelt. Zusätzlich wird sonst, neben Takt- und *Reset*-Leitung, nur eine Schnittstelle für den gemeinsamen Speicher benötigt, welcher nicht direkt in TMPL integriert ist, sondern ebenfalls extern angebunden werden muss. Im Folgenden werden die verwendeten Techniken zur Maßschneiderung im Kontext von TMPL vorgestellt sowie deren Vor- und Nachteile diskutiert.

4.4.1.2 Parametrisierung der transaktionalen Speicherfamilie

Die grundlegende Parametrisierung von TMPL erfolgt über sechs *Generic*-Variablen. Die hier definierten Konstanten werden durch die diversen Komponenten von TMPL propagiert, um an den entsprechenden Stellen beispielsweise die Busbreite von Signalen zu konfigurieren oder zusätzliche Funktionalitäten in das System zu integrieren. Die in TMPL verwendeten *Generics* werden im Folgenden erläutert:

Anzahl der Teilnehmer Über einen der *Generics* kann die Anzahl der angeschlossenen Teilnehmer festgelegt werden. Je nach Konfiguration stellt TMPL so die entsprechende Anzahl an Schnittstellen für die angeschlossenen Knoten bzw. Prozessoren zur Verfügung. Dieser Parameter beeinflusst also insbesondere auch die in Abbildung 4.6 dargestellte Anzahl an Wishbone-Schnittstellen und *Interrupt*-Leitungen.

Größe des Speichers Auch die Größe des verwendeten Speichers muss für TMPL spezifiziert werden, damit zum Beispiel die Breite des Adressbusses entsprechend dimensioniert werden kann.

Breite des Datenbusses Ebenso lässt sich über einen weiteren Parameter die Datenbreite für den Zugriff auf den gemeinsamen Speicher bestimmen.

Größe des separaten Puffers (pro Teilnehmer) Für jeden angeschlossenen Teilnehmer kann die Größe des separaten Puffers angegeben werden, indem,

abhängig von der Variante, die neuen oder alten Werte einer Transaktion abgelegt werden können. Die Puffergröße lässt sich für jeden Teilnehmer unabhängig spezifizieren, da die Transaktionen der verschiedenen Teilnehmer durchaus unterschiedliche Längen aufweisen können und dementsprechend abweichende Anforderungen an den Platzbedarf in einem Puffer gestellt werden.

Byte-Zugriff (pro Teilnehmer) Falls benötigt, kann für einzelne Teilnehmer auch die Möglichkeit zum Byte-Zugriff auf den gemeinsamen Speicher statisch aktiviert werden.

nichttransaktionale Zugriffe auf Speicher (pro Teilnehmer) Eine weitere Eigenschaft von TMPL ist, dass sich zusätzlich für beliebige Teilnehmer ein direkter Zugriff auf den Speicher außerhalb einer Transaktion konfigurieren lässt. Diese Funktion kann zum Beispiel von einem Knoten zur schnellen Initialisierung des Speichers genutzt werden. Die Integrität des Speichers muss in diesem Fall über die Software sichergestellt werden.

4.4.1.3 Variabilität durch verschiedene Architekturen

Die wesentlichen Strategien von TMPL, wie die Versionierung, die Konflikterkennung oder die Konfliktauflösung, werden nicht über *Generics* konfiguriert, sondern über die Einbindung von verschiedenen Implementierungen realisiert. In VHDL setzen sich Komponenten aus Schnittstellenbeschreibung und Architekturbeschreibung zusammen, wobei für eine Schnittstellenbeschreibung verschiedene Architekturimplementierungen genutzt werden können. Spätestens bei der Instanziierung einer Komponente muss jedoch angegeben werden, welche dieser Architekturbeschreibungen verwendet werden soll. Dieser Mechanismus kann zum Beispiel dazu genutzt werden, um verschiedene Implementierungen für die Simulation oder die Hardwaresynthese bereit zu halten. Speziell wenn VHDL-Sprachelemente verwendet werden müssen, welche nicht zur Synthese geeignet sind, ist dieser Vorgehen ein möglicher Ausweg.
Für die transaktionale Speicherfamilie wird dieser Mechanismus genutzt, um die diversen Strategien der Familie entsprechend der jeweiligen Anforderungen auszuwählen. In Abbildung 4.7 werden die Struktur und das Konfigurierungsprinzip von TMPL verdeutlicht. Die vier Basiskomponenten von TMPL setzen sich jeweils aus einer Schnittstellenbeschreibung (oben) und unterschiedlich vielen Architekturbeschreibungen (unten) zusammen. Die oberste Komponente in der Hierarchie mit der Schnittstellenbeschreibung `tm_interface`, bietet die in Abbildung 4.6 dargestellte Schnittstelle zur Umgebung und muss vom Anwender instanziiert sowie über die bereits vorgestellten *Generics* parametrisiert werden. Zusätzlich muss bei der Instantiierung die gewünschte Architekturbeschreibung ausgewählt werden. Die vier Architekturbeschreibungen dieser Komponente decken alle implementierten Varianten von TMPL ab. Wenn der Anwender diese Komponente mit der Architekturbeschreibung `lp_oldest` instanziiert, wird ein entsprechen-

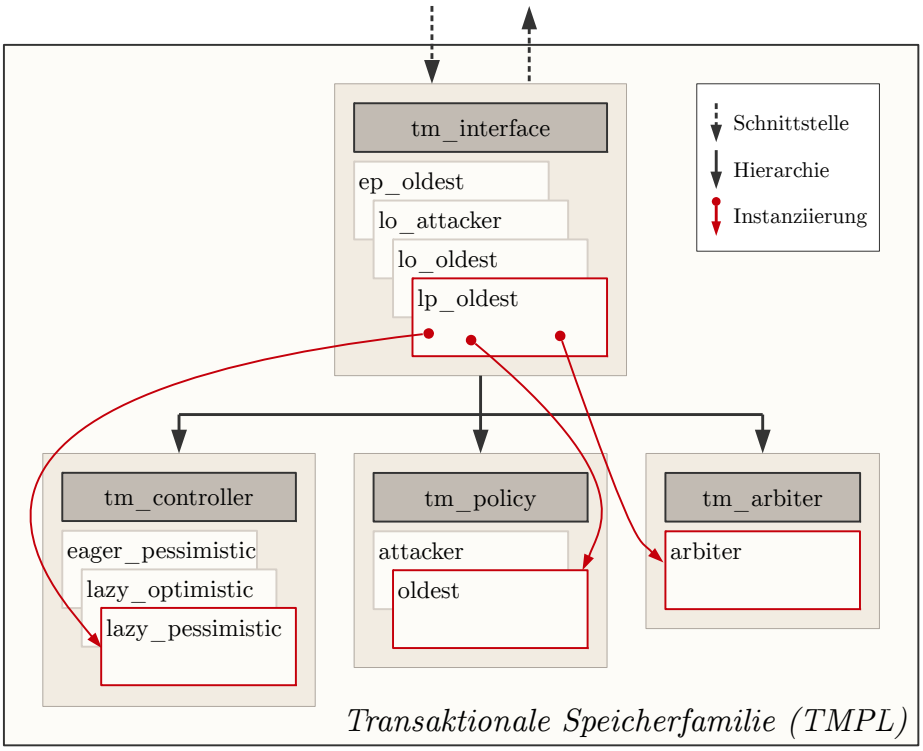


Abbildung 4.7: Struktur der transaktionalen Speicherfamilie (vereinfacht)

des System mit der Versionierung *Lazy*, der Konflikterkennung *Pessimistic* und der Konfliktauflösung *Oldest* konfiguriert. In der gewählten Architektur werden dann automatisch für alle Subkomponenten die entsprechenden Architekturen ausgewählt und instanziiert (roten Pfeile).

4.4.1.4 Iterative Generierung

Die im vorherigen Kapitel genutzte Konfigurierung von TMPL über verschiedene Architekturbeschreibungen eignet sich nur für eine sehr grobgranulare Maßschneiderung des Systems, da ansonsten für alle möglichen Kombinationen von Merkmalen dedizierte Architekturen mit oft redundantem Quelltext implementiert werden müssten. Insbesondere für die Teilnehmeranzahl sind viele feingranulare Anpassungen an den Komponenten nötig, damit für jeden Teilnehmer die benötigten Signale und Strukturen zur Verfügung stehen. Eine separate Architekturbeschreibung für alle Strategien in Kombination mit allen möglichen Teilnehmerzahlen ist in der Praxis also kaum sinnvoll einsetzbar. Zur Vermeidung dieser Problematik wird in TMPL die `generate`-Anweisung verwendet, um diese feingranularen Signale und Strukturen beispielsweise iterativ für alle Teilnehmer zu generieren. Zudem können die `generate`-Anweisungen über Bedingungen kontrolliert werden. So ist es zum Beispiel möglich, nur für Teilnehmer mit Byte-Zugriff ein bestimmtes Quelltextfragment zu generieren. Auch die benötigten Block-RAMs

zum zwischenzeitlichen Puffern der Transaktionen, werden über diesen Mechanismus zu einem Speicher der spezifizierten Größe zusammengesetzt.

4.4.1.5 Fazit

Wie die transaktionale Speicherfamilie zeigt, lassen sich mit den in VHDL gebotenen Sprachmitteln komplexe Hardware-Produktlinien implementieren. Die Konfigurierung erfolgt dabei direkt bei der Instanziierung der Hardware über die Parametrisierung der *Generics* und die Auswahl der gewünschten Architekturbeschreibung. Dadurch kann der Quelltext, aus Anwendersicht, als Blackbox verwendet werden und muss nicht manuell in den Tiefen des generischen Quelltextes modifiziert werden.
Etwas anders stellt sich die Situation für den Entwickler der Produktlinie dar, denn bei einer exzessiven Nutzung dieser Mittel leidet die Übersicht und damit auch die Wartbarkeit des Quelltextes unter Umständen erheblich. Eine ähnliche Problematik lässt sich auch bei der übermäßigen Verwendung der bedingten Präprozessor-Direktiven in C bzw. C++ finden. Auch der transaktionalen Speicherfamilie ist diese Problematik stellenweise anzumerken. So muss beispielweise die Busbreite von vielen Signalen über extra Funktionen berechnet werden, welche zusätzlich von *Generics* beeinflusst werden, was die schnelle Erfassung der Signalbreiten für den Entwickler sehr schwierig macht. Des Weiteren werden viele Quelltextfragmente über `generate`-Anweisung generiert. Ebenfalls problematisch ist die Duplizierung von Quelltextfragmenten für Komponenten mit mehreren Architekturbeschreibungen, denn auch wenn diese Beschreibungen oftmals andere Strategien implementieren, ähneln sich einige Teile recht deutlich oder sind sogar identisch. So finden sich in den beiden Architekturen der Konfliktauflösung zum Beispiel identische VHDL-Prozesse wieder. Für weit komplexere, hochkonfigurierbare Systeme, bei denen zudem noch Prozessoren, Peripheriekomponenten und weitere Kommunikationsstrukturen einbezogen werden, wird die Situation durch den Einsatz dieser Mittel noch weiter verschärft.

4.4.2 Frame-Technologie

Neben den in VHDL enthaltenen Sprachmitteln zur Beschreibung von generischen Hardwarekomponenten, wird in dieser Arbeit auch die *Frame*-Technologie auf ihre Eignung im Kontext von Hardware-Produktlinien untersucht [MESS10]. Zur Beschreibung der Variabilität in der LavA Hardware-Produktlinie wird dazu der *Frame*-Prozessor XVCL verwendet, mit dem sich auch der VHDL-Quelltext der konfigurierten Instanzen generieren lässt.

4.4.2.1 Spezifizierung der Hardwarestruktur

Wie auch bei einigen anderen Arbeiten [WLZ+12, WHH10], die in Unterkapitel 3.1.3 vorgestellt wurden, wird in XVCL die Konfiguration einer Variante direkt in einem XML-Format spezifiziert. Diese Konfiguration wird für XVCL in

```
1 <set-multi var="Node"           value="Node1,Node2"/>
2 <set-multi var="CoreID"         value="1,2"/>
3 <set-multi var="CoreType"       value="MB-Lite,ZPU"/>
4 <set-multi var="Interrupt"      value="1,0"/>
5 <set-multi var="HW-FPU"         value="1,0"/>
6 <set-multi var="RoundingMode"   value="0,0"/>
7 <set-multi var="HW-Mult"        value="0,0"/>
8 <set-multi var="HW-Barrel"      value="0,0"/>
9 <set-multi var="Memory"         value="8,4"/>
```

Abbildung 4.8: Konfigurationsoptionen für die Prozessoren in XVCL

einem gesonderten x-*Frame* angegeben, dem sogenannten SPC (*Specification x-Frame*). Der für die LavA Hardware-Produktlinie entworfene SPC enthält neben allgemeinen Parametern, wie dem FPGA-Typen oder der Frequenz mit dem das Hardwaredesign betrieben werden soll, verschiedene Konfigurationspunkte zu den Prozessoren, zu den Peripheriekomponenten der Knoten oder zu den Kommunikationsstrukturen.

Die zur Verfügung stehenden Konfigurationsparameter für die Prozessoren der LavA Hardware-Produktlinien sind in Abbildung 4.8 dargestellt. Zur Beschreibung der Prozessoren wurde das `<set-multi>`-Kommando von XVCL verwendet. Ähnlich zu Feldern in anderen Programmiersprachen, können diese Variablen mehrere Werte aufnehmen. In dem Beispiel sind die Parameter für zwei Prozessoren beschrieben. Der erste Prozessor mit der ID `1` ist vom Typ `MB-Lite`, soll Unterbrechungen unterstützen, hat eine integrierte Hardware-Gleitkommaeinheit und 8 KiB Speicher. Der Prozessor mit der ID `2` ist vom Typ `ZPU` und hat 4 KiB Speicher. Der Vorteil bei der hier verwendeten Art der Variablen ist, dass die Konfigurationsinformationen relativ übersichtlich dargestellt und modifiziert werden können. Zudem kann über diese Variablen mittels Schleifen iteriert werden, wodurch für weniger komplexe Strukturen relativ einfach der gewünschte Quelltext generiert werden kann. Auch die Iteration über mehrere Variablen mit der gleichen Anzahl an Elementen ist möglich, so dass zum Beispiel auf sämtliche Konfigurationsinformationen des jeweiligen Prozessors pro Iteration zugegriffen werden kann.

Die Variable `Node` enthält keine Konfigurationsinformationen für den Prozessor im eigentlichen Sinne, sondern einen Bezeichner zur internen Verwendung, damit die Peripheriekomponenten den Prozessoren bzw. den Knoten zugeordnet werden können. Die Konfigurationsinformationen für die Peripheriekomponenten des Prozessors mit der ID `1` sind in Abbildung 4.9 zu sehen. Die Variablennamen zu diesen Konfigurationsinformationen sind nach einem festen Schema aufgebaut und müssen zu Beginn den zuvor genannten Bezeichner tragen. Die Variablennamen können aufgrund des festen Schemas und dem für jeden Prozessor definierten Bezeichner dynamisch zusammengesetzt werden, wodurch der Zugriff auf die zugehörigen Konfigurationsinformationen für jeden Prozessor sichergestellt wird. In dem Beispiel werden für den Knoten ein Zeitgeber (engl. *Timer*) und eine

```
1 <set-multi var="Node1"          value="Timer_Node1,Outport_Node1"/>
2 <set-multi var="Node1_IOType"   value="Timer,Outport"/>
3 <set-multi var="Node1_IRQ"      value="Timer_Node1"/>
```

Abbildung 4.9: Konfigurationsoptionen für die Peripheriekomponenten eines Knotens in XVCL

```
1 <set-multi var="Com"            value="Bus1"/>
2 <set-multi var="ComID"          value="1"/>
3 <set-multi var="ComType"        value="Bus"/>
4 <set-multi var="ComWidth"       value="32"/>
5
6 <set-multi var="Bus1"           value="1,2"/>
```

Abbildung 4.10: Konfigurationsoptionen für die Kommunikationsstrukturen in XVCL

Schnittstelle zur Ausgabe (engl. *Outport*) spezifiziert. Zudem wird der Zeitgeber über eine Leitung zur Unterbrechungsanforderung mit dem Knoten verbunden. Weitere zur Verfügung stehende Parameter, wie die Wahl zwischen einem 32 oder 64 Bit Zeitgeber oder die Spezifizierung der Baudrate eines UARTs, wurden der Übersicht halber nicht aufgeführt.
Nach einem ähnlichen Muster, wie bei den Prozessoren, können auch die Kommunikationsstrukturen in dem SPC spezifiziert werden. Es muss zunächst ein Bezeichner angegeben werden, gefolgt von der ID, dem Typen der Kommunikationsstruktur und der Konfiguration der Busbreite. In dem Beispiel in Abbildung 4.10 wird eine Kommunikationsstruktur mit der ID `1` vom Typ `Bus` beschrieben, welche 32 Bit breit ist. Auch hier können aufgrund der *Multi*-Wert Variablen, mehrere Kommunikationsstrukturen spezifiziert werden. Der etwas abgesetzte XVCL-Befehl in Zeile sechs beschreibt die Zuordnung von Prozessoren zu der jeweiligen Kommunikationsstruktur. Der Variablenname entspricht dabei dem spezifizierten Bezeichner aus Zeile eins. An dem Bus mit dem Bezeichner `Bus1` hängen in diesem Fall also die Prozessoren mit den IDs `1` und `2`. So lassen sich beliebige Strukturen aufbauen, um zwischen den spezifizierten Prozessoren zu kommunizieren.

4.4.2.2 Statische Konfigurierung der Hardwarestruktur

Im Gegensatz zu den zuvor genannten Arbeiten [WLZ+12, WHH10], in denen die Hardwarestruktur ausschließlich in einem XML-Format spezifiziert wird, kann mittels XVCL auch die Zielsprache um die benötigtet Variabilität erweitert und der Quelltext generiert werden. Das zugrunde liegende Schema nach dem der VHDL-Quelltext der LavA Hardware-Produktlinien zusammengesetzt wird, ist in Abbildung 4.11 dargestellt. Startpunkt ist der bereits angesprochene SPC. Die bereits vorgestellten Konfigurationsoptionen des SPCs werden direkt zu Beginn

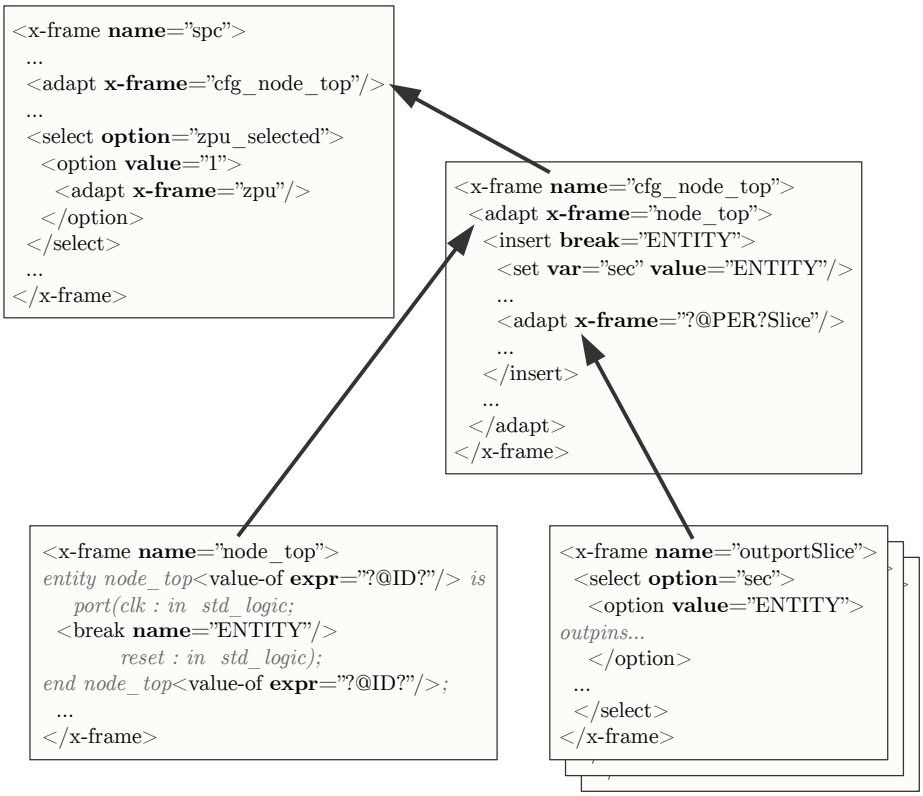


Abbildung 4.11: Variabilität in der LavA Hardware-Produktlinie mittels XVCL

dieses speziellen x-*Frames* spezifiziert und sind in dieser Abbildung nicht erneut aufgeführt. Neben der Spezifikation der Hardwarestruktur, adaptiert (`<adapt>`-Kommando) der SPC weitere, mittels x-*Frames* beschriebene Komponenten der Hardwarestruktur (siehe Pfeile), um so das gesamte System aus den x-*Frames* zusammenzufügen.

Damit die entsprechenden Komponenten adaptiert werden, müssen zum Teil gewisse Bedingungen erfüllt werden. Zum Beispiel wird der Quelltext der ZPU nur dann eingebunden, wenn zumindest einer der gewählten Prozessoren im System vom Typ ZPU ist. Dazu wird die *Multi*-Wert Variable `CoreType` aus Abbildung 4.8 in einer Schleife durchlaufen und die Variable `zpu_selected` bei dem Vorkommen von zumindest einer ZPU auf Eins gesetzt. Da der Kern der ZPU keine weiteren Möglichkeiten zur Konfigurierung bietet, besteht der x-*Frame* der ZPU (nicht abgebildet) im wesentlichen aus dem zu kopierenden VHDL-Quelltext, welcher von dem `<x-frame>`-Kommando umschlossen wird.

Wesentlich komplizierter ist die Maßschneiderung derjenigen Komponenten, in denen zum Beispiel die Kommunikationsstrukturen verknüpft werden müssen oder in denen die Signale von Prozessor, zugehörigem lokalen Speicher und den Peripheriekomponenten eines einzelnen Knotens zusammenlaufen. Auf diese Komponenten nehmen sehr viele Konfigurationsoptionen Einfluss und müssen

dementsprechend stark an die jeweils gegebene Spezifikation angepasst werden. Durch die Vielzahl an justierbaren Optionen besteht allerdings die Gefahr, dass die Übersicht in diesen Komponenten leidet und demzufolge eventuell fehlerhafte Systeme generiert werden. Auch die Integration von weiteren Komponenten kann im Laufe der Zeit dazu führen, dass der durch XVCL-Befehle angereicherte VHDL-Quelltext immer komplexer wird und sich die Wartbarkeit verschlechtert. Um diese Probleme zu reduzieren, wurde zur Integration der Peripheriekomponenten die in Abbildung 4.11 präsentierte Methodik verwendet. Die VHDL-Komponente in der die Signale eines Knotens zusammenlaufen, setzt sich aus zwei x-*Frames* zusammen: dem `cfg_node_top` x-*Frame* und dem `node_top` x-*Frame*. Der x-*Frame* `cfg_node_top` wird von dem SPC adaptiert und beinhaltet nur Konfigurationsinformationen zur Maßschneiderung der Komponente und keine VHDL-Fragmente. Im Gegensatz dazu stellt der x-*Frame* `node_top` das Grundgerüst der VHDL-Komponente zur Verfügung. Dieses Grundgerüst wird durch die VHDL-Fragmente der einzelnen x-*Frames* der Peripheriekomponenten entsprechend erweitert. Der VHDL-Quelltext in den x-*Frames* ist in Kursivschrift dargestellt. Der x-*Frame* `cfg_node_top` adaptiert direkt zu Beginn den x-*Frame* `node_top`. Über die `<break>`-Kommandos können dabei an beliebigen Stellen Marker gesetzt werden, bei denen von dem x-*Frame* `node_top` zurück in den Konfigurations-*Frame* `cfg_node_top` gesprungen werden kann, um an diesen Stellen maßgeschneiderten VHDL-Quelltext einzufügen. So kann der VHDL-Quelltext soweit wie möglich von dem Konfigurationswissen getrennt werden. In dem hier skizzierten Fall können weitere ein- und ausgehenden Signale zur Schnittstellenbeschreibung der VHDL-Komponente `node_top` hinzugefügt werden. Dazu wird zum gleichnamigen `<insert>`-Kommando in dem Konfigurations-*Frame* gesprungen und dort die entsprechenden XVCL-Befehle abgearbeitet. Am Ende des *Insert*-Blocks wird der Konfigurations-*Frame* wieder verlassen und die Abarbeitung in dem x-*Frame* `node_top` fortgesetzt.

Die an diversen Markern einzufügenden Fragmente der Peripheriekomponenten werden für jede Komponente in einem eigenen x-*Frame* beschrieben und an den passenden Stellen von dem Konfigurations-*Frame* eingefügt. In dem Beispiel werden die ausgehenden Signale der Ausgabeschnittstelle, welche in dem x-*Frame* `outportSlice` beschrieben sind, an der durch das `<break>`-Kommando spezifizierten Position eingefügt. Die Zuordnung von den Fragmenten zu den Positionen geschieht über den Namen des `<break>`-Kommandos, in diesem Fall also `ENTITY`. Auch an weiteren Positionen im VHDL-Quelltext werden für die LavA Hardware-Produktlinie verschiedene `<break>`-Kommandos eingesetzt, um andere Fragmente der Peripheriekomponenten dort einzufügen. Durch diese Methodik sind alle Fragmente der Peripheriekomponenten jeweils in einem eigenen x-*Frame* gekapselt und neue Komponenten müssen nicht an vielen verstreuten Stellen in das Konfigurationssystem eingearbeitet werden. Auch die nachträgliche Bearbeitung von Peripheriekomponenten ist durch die zentrale Kapselung des Quelltextes erheblich vereinfacht, da die Fragmente nicht über viele x-*Frames* verteilt liegen.

4.4.2.3 Fazit

Die mit XVCL zur Verfügung stehende *Frame*-Technologie ist recht mächtig und kann, wie gezeigt, auch zur Maßschneiderung der LavA Hardware-Produktlinie eingesetzt werden und ebenso die Abbildung der Kommunikationsstrukturen bewältigen. Des Weiteren kann unter Verwendung der passenden Methoden, auch die nachträgliche Erweiterung der Hardware-Produktlinien durch neue Komponenten durchaus erleichtert werden. Auch die mögliche Trennung zwischen dem Konfigurationswissen und dem Quelltext der Zielsprache verbessert die Übersicht und reduziert so mögliche Fehler.
Die Beschreibung des Konfigurationswissens ist mit XVCL jedoch teilweise sehr umständlich, wenn komplexere Strukturen generiert werden sollen. Insbesondere für die Generierung der vorgestellten `node_top`-Komponente mussten tief verschachtelte Schleifen und Bedingungen verwendet werden, um an das gewünschte Ziel zu gelangen. Dies liegt unter anderem an der Einschränkung, dass Schleifen nur über *Multi*-Wert Variablen iterieren können und keine Zählschleifen mit einfachen Variablen eingesetzt werden können. Auch die dynamische Verwendung von *Multi*-Wert Variablen ist in XVCL nicht vorgesehen, wodurch Berechnungen stets erneut durchgeführt werden müssen und nicht für eine spätere Verwendung abgelegt werden können. Durch diese Einschränkungen sind Teile des Konfigurationswissens oftmals umfangreich spezifiziert und im Nachhinein schwerer nachzuvollziehen.

4.4.3 Codegeneratoren aus der modellgetriebenen Entwicklung

Aufgrund der geschilderten Defizite bei der Verwendung von XVCL und dem ohnehin modellbasierten Entwurf der Referenzarchitektur in dieser Arbeit, werden auch Codegeneratoren aus der modellgetriebenen Entwicklung auf ihre Eignung im Kontext von Hardware-Produktlinien untersucht. Zur Modellierung des Metamodells wird das EMF (*Eclipse Modeling Framework*) verwendet und demzufolge werden die vom EMF gebotenen Techniken zur Generierung und Maßschneiderung von Quelltext betrachtet.

4.4.3.1 Spezifizierung der Hardwarestruktur

Die Spezifizierung der Hardware erfolgt bei der modellgetriebenen Entwicklung allein auf Basis von Modellen. Die Modelle beschreiben dabei jeweils konkrete Systeme, die nach den Konstruktionsvorschriften des Metamodells zusammengesetzt werden können. Im Gegensatz zu XVCL müssen die Konfigurationsinformationen nicht allein über Listen verwaltet werden, sondern können direkt über das abgeleitete Modell abgerufen werden. Die Modelle können sowohl über die grafische Benutzeroberfläche des EMF erstellt werden als auch direkt per XML-Format in einer Datei abgelegt werden.
Das in Abbildung 4.12 konfigurierte Modell zeigt, wie ein einfaches LavA-System über die grafische Benutzeroberfläche des EMF beschrieben werden kann und ver-

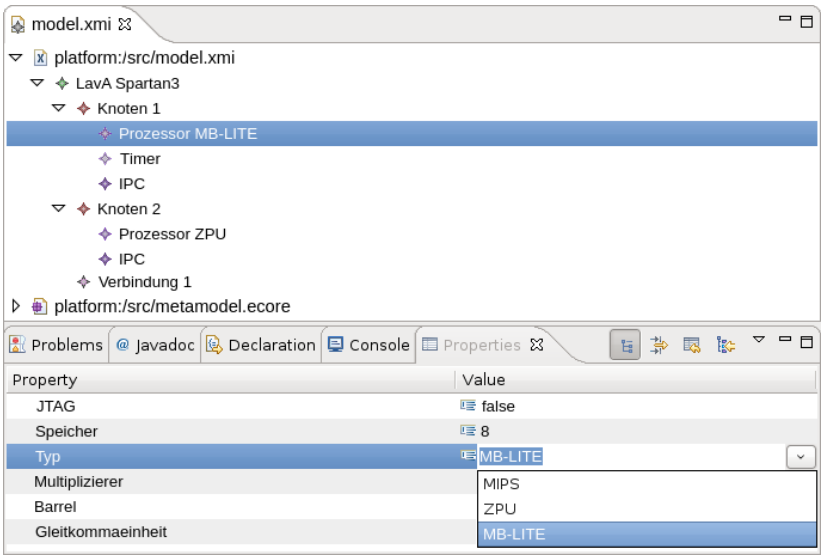


Abbildung 4.12: Modellierung von Systemen über die Benutzeroberfläche des EMF

deutlicht zudem, auf welcher Abstraktionsebene ein Anwender mit diesem modellgetriebenen Ansatz arbeitet, damit für ein spezifiziertes System über die Modell-zu-Code-Transformation des EMF der zugehörige VHDL-Quelltext automatisiert generiert werden kann. Die einzelnen Punkte in dem oberen Fenster spiegeln die verschiedenen Komponenten aus dem Metamodell wider und können dort nach den Vorschriften des Metamodells zu einem System zusammengesetzt werden. So könnte das System beispielsweise um weitere Knoten oder auch bereits existierende Knoten um weitere Peripheriekomponenten ergänzt werden. In dem unteren Bereich des Fensters, können die im Metamodell hinterlegten Eigenschaften der einzelnen Komponenten konfiguriert werden und so, wie in der Abbildung gezeigt, zum Beispiel der Prozessortyp festgelegt werden.

4.4.3.2 Statische Konfigurierung der Hardwarestruktur

Die Sprache zur Generierung von Quelltext einer beliebigen Zielsprache aus EMF-Modellen heißt Xpand. Auf den ersten Blick verwendet Xpand sehr ähnliche Mechanismen wie auch XVCL. Die Pendants zu den x-*Frames* aus XVCL werden beispielsweise als *Templates* bezeichnet und können ebenso andere *Templates* einbinden, um den Quelltext der Zielsprache aus verschiedenen Teilen zusammenzusetzten. Des Weiteren ist der Konfigurationscode von Xpand zur Beschreibung der Variabilität, wie auch bei XVCL, mit den Fragmenten der Zielsprache vermischt. Aufgrund der sehr ähnlichen Mechanismen ist die Herangehensweise zur Maßschneiderung der LavA-Hardware nahezu Analog zu dem Vorgehen in XVCL. So werden zum Beispiel auch in Xpand die zur Implementierung der Peripheriekomponenten benötigten VHDL-Fragmente in einem jeweils eigenen *Template*

```
1 context Knoten ERROR
2       "Knoten (ID " + id + ") hat keine Peripheriekomponente" :
3       Devices.quantity > 0;
```

Abbildung 4.13: Beispielregel zur Validierung eines Modells

gebündelt und erst im Zuge der Transformation an den passenden Stellen der diversen VHDL-Komponenten integriert, um die einfache Erweiterung der Produktlinie um neue Peripheriekomponenten zu ermöglichen.
Falls die gebotenen Mittel der Xpand-Sprache nicht ausreichen, können Java-Methoden über die Xtend-Erweiterung eingebunden werden. So kann zum Beispiel relativ einfach die Anzahl der benötigten Adressbits für einen Adressbus in Abhängigkeit von der Speichergröße berechnet werden oder auch eine Wandlung von Dezimalzahlen in entsprechende Bitfolgen vorgenommen werden.

4.4.3.3 Validierung der Modelle

Um die Korrektheit von abgeleiteten Modellinstanzen zu gewährleisten, können über die Spracherweiterung *Check* weitere Regeln definiert werden. Dies kann insbesondere dann von Vorteil sein, wenn sich diese Vorschriften nicht über das Metamodell formulieren lassen. Ein mögliches Beispiel für solch eine Regel ist in Abbildung 4.13 abgebildet. In der ersten Zeile wird zunächst der Kontext, in dem diese Regel wirken soll, festgelegt und welche Maßnahme ergriffen werden soll, falls diese Regel nicht erfüllt werden kann. Die hier gezeigte Regel bezieht sich auf die Knoten und im Fehlerfall soll die Generierung abgebrochen werden. Für weniger gravierende Probleme kann die Regel zu einer einfachen Warnung reduziert werden. Die zweite Zeile zeigt die im Fehlerfall auszugebende Nachricht an. Die zu erfüllende Bedingung ist in Zeile drei beschrieben. Die Bedingung ist in dem Beispiel erfüllt, wenn für diesen Knoten mindestens eine Peripheriekomponente konfiguriert wurde. Falls also für einen Knoten keine Peripheriekomponente konfiguriert werden sollte, würde diese Regel mit der angegebenen Fehlermeldung die Codegenerierung abbrechen.

4.4.3.4 Fazit

Der Codegenerator Xpand bietet ähnliche Mechanismen wie XVCL, der erforderliche Konfigurationscode ist gegenüber XVCL allerdings deutlich reduziert. Dies liegt zum Beispiel an den unterschiedlichen Formen in denen die Konfigurationsinformationen vorliegen. Für XVCL liegen sämtliche Informationen in Listen (*Multi*-Wert Variablen) vor, weshalb der Konfigurationscode für die hochkonfigurierbaren Hardwarekomponenten teilweise bis zu sechsfach geschachtelte XML-Befehle enthält, damit durch die Listen iteriert und Bedingungen ausgewertet werden können. Die baumartige Struktur der Modelle aus der modellgetriebenen Entwicklung vereinfacht den Zugriff auf die benötigten Konfigurationsinforma-

tionen erheblich und reduziert damit zugleich den Konfigurationscode. Auch die Möglichkeit einfache Berechnungen in Java-Methoden auszulagern, reduziert den Konfigurationscode in den Xpand-*Templates* weiter. Eine quantitative Untersuchung zu den Größenordnungen des jeweils verwendeten Konfigurationscode, wird in Unterkapitel 4.4.5 präsentiert.

Zudem bietet das EMF die Validierung der Modelle unter Verwendung der *Check*-Regeln. Diese Regeln können losgelöst vom Konfigurationscode implementiert werden und beeinflussen so nicht die Implementierung der variablen Hardwarekomponenten.

4.4.4 Aspektorientierte Programmierung

Die letzte untersuchte Technik zur Implementierung der LavA Hardware-Produktlinie ist die aspektorientierte Programmierung. Die aspektorientierte Programmierung kann neben der Kapselung von querschneidenden Belangen auch zur Umsetzung von Variabilität genutzt werden, weshalb diese Technik durchaus auch für den Einsatz in Produktlinien von Interesse ist. Im Gegensatz zur Softwareentwicklung finden sich im Kontext der Hardwareentwicklung bisher nur wenige Arbeiten, welche sich mit der Übertragung dieser Konzepte auf Hardwarebeschreibungssprachen beschäftigen. Erste Ansätze [DM06, MF14] in diese Richtung setzen sich mit der Anwendung von Aspekten in der C++-Erweiterung SystemC auseinander. Der Ansatz von Déharbe und Medeiros verwendet zur Erweiterung von SystemC, um die aspektorientierte Programmierung, AspectC++ [DM06]. Zudem werden verschiedene Beispiele von Hardwaredesigns vorgestellt, in denen die aspektorientierte Programmierung eingesetzt werden kann. In diesen Szenarien werden jedoch allein die Auswirkungen der Aspekte auf die Simulationszeiten der Hardwaremodelle untersucht und nicht, inwiefern die Synthese bzw. die letztlich synthetisierte Hardware beeinflusst wird. Einen Schritt weiter in Richtung synthetisierbare Hardware gehen Mück und Fröhlich, denn in deren Ansatz wird ausschließlich eine synthetisierbare Teilmenge von SystemC verwendet [MF14]. Da der vom AspectC++-Weber erzeugte Quelltext nicht synthetisierbare Konstrukte enthalten kann, verwenden die Autoren ein verwandtes Konzept zur Realisierung der Aspekte in SystemC, welches jedoch im Vergleich zu AspectC++ weniger mächtig und bezüglich der *Pointcut*-Ausdrücke weniger Ausdrucksstark ist. Zudem erlaubt SystemC zwar die Beschreibung von Hardware auf einer hohen Abstraktionsebene, wird aber in der Praxis hauptsächlich zu Simulationszwecken eingesetzt und ist eher weniger für die Synthese von Hardwaredesigns in Gebrauch.

Auch zur Verifikation von Hardware wurde ein aspektorientierter Ansatz im Kontext der Hardwareverifikationssprache „e“ untersucht [Vax07]. Allerdings sind die Möglichkeiten, welche die Sprache „e“ bietet, gegenüber der aspektorientierten Programmierung stark eingeschränkt. So können beispielsweise *Advices* jeweils nur an einen *Join Point* wirken.

ADH (*Aspect Described Hardware Programming Language*) [BSP05] ist eine domänenspezifische Sprache zur Unterstützung der aspektorientierten Programmierung im Kontext des Hardwaredesigns. ADH kann mithilfe eines Übersetzers zu regulärem VHDL-Quelltext übersetzt werden und dann im Anschluss synthetisiert werden. Eine neue Sprache wie ADH birgt jedoch immer die Gefahr, dass diese von den Entwicklern nicht angenommen wird. Die Erweiterung von existierenden Sprachen, wie VHDL oder Verilog um die Aspektorientierung, hat vermutlich mehr Potenzial, um von einer breiten Menge von Entwicklern akzeptiert zu werden.
Für die Verwendung von Aspekten in den weitverbreiteten Hardwarebeschreibungssprachen fehlt bisher jedoch die entsprechende Unterstützung. Einen ähnlichen Ansatz, jedoch auf Basis der weniger mächtigen merkmalorientierten Programmierung, verfolgen die Autoren in [JQTG11] mit FeatureVerilog. Zur aspektorientierten Programmierung zeigen Engel und Spinczyk erste Ideen auf, wie zum Beispiel *Join Points* und *Pointcuts* in VHDL aussehen könnten [ES08]. Das hier vorgestellte *Join Point*-Modell umfasst beispielsweise *Join Points* für die „Ausführung“ von Prozessen und das Setzen von Signalen. Zudem wurde beleuchtet, für welche Anwendungsfälle in der Hardwareentwicklung bei dem Einsatz von Hardwarebeschreibungssprachen querschneidende Belange zu finden sind. Identifiziert wurden in dieser Arbeit beispielsweise Bussysteme oder die Behandlung von *Resets*. Eine vollständige Sprache zur Nutzung von Aspekten in VHDL wurde in dieser Arbeit jedoch nicht präsentiert. Diese Lücke soll nun mit dieser Arbeit geschlossen werden und die Spracherweiterung AspectVHDL [MHS12] vorgestellt werden. Wodurch Aspekte Einzug in die Hardwarebeschreibungssprache VHDL erhalten.

4.4.4.1 Fallstudie: MB-Lite Erweiterungen

Begleitet von einer Fallstudie soll im Folgenden die Spracherweiterung AspectVHDL vorgestellt, bewertet und diskutiert werden. Die Fallstudie verdeutlicht dabei noch einmal, in welcher Form querschneidende Belange auch im Kontext der Hardwareentwicklung zu finden sind und warum Aspekte auch in dieser Domäne dem Entwickler dabei helfen können, den Quelltext zu modularisieren, um die Struktur des VHDL-Quelltextes zu verbessern. Als Beispiel für diese Fallstudie dient der MB-Lite Prozessor. Der MB-Lite Prozessor ist synthetisierbar und implementiert eine klassische 5-stufige RISC-Architektur. Der Prozessor besteht aus fünf VHDL-Komponenten, welche im Wesentlichen die einzelnen *Pipeline*-Stufen des Prozessors repräsentieren. Zusätzlich gehören zu dem Prozessor noch drei VHDL-Pakete, in denen beispielsweise Typen deklariert oder Hilfsfunktionen implementiert sind.
Einer der typischen Anwendungsfälle von aspektorientierter Programmierung in der Softwareentwicklung ist die Implementierung von optionalen Erweiterungen mittels Aspekten. Beispielsweise nutzen Lohmann *et al.* die AspectC++ Sprache zur Erweiterung der Funktionalität von Betriebssystemen mittels Aspekten,

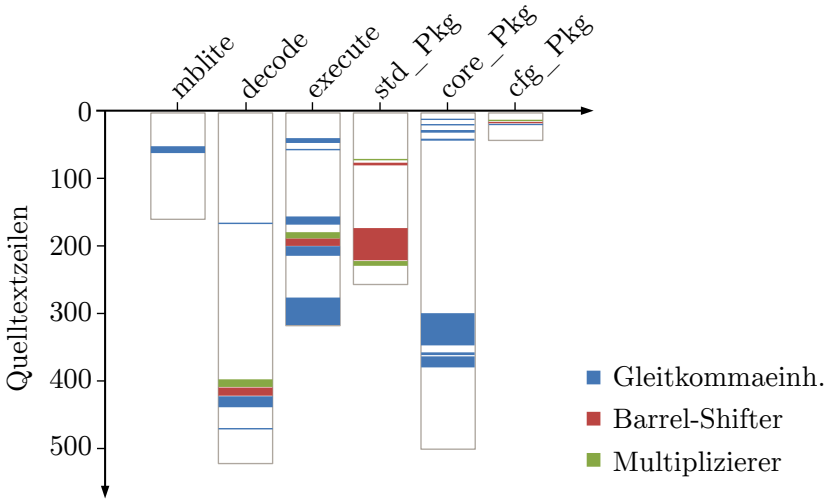


Abbildung 4.14: Verstreuung des Quelltextes für optionale MB-Lite Erweiterungen

da diese Erweiterungen oftmals einen querschneidenden Charakter haben und verschiedene Kernsysteme betreffen [LHSPS11]. Ein ähnliches Muster ist auch bei dem MB-Lite Prozessor zu erkennen, denn bei diesem schneiden drei optional konfigurierbare Erweiterungen diverse Teile des Quelltextes. Abbildung 4.14 zeigt die Verstreuung des Quelltextes für die optionalen Erweiterungen: Gleitkommaeinheit[5], *Barrel-Shifter* und Multiplizierer, über die Quelltextdateien des Prozessors. Die Quelltextfragmente der optionalen Beschleuniger finden sich in drei der fünf Komponenten und in allen VHDL-Paketen des MB-Lite Prozessors. Ein genauerer Blick auf den VHDL-Quelltext (insgesamt 5.294 Quelltextzeilen) zeigt, dass die drei Erweiterungen über den bereits in Unterkapitel 4.4.1 diskutierten Mechanismus der *Generics* in den Prozessor integriert werden, womit die bereits aufgezeigten Nachteile einhergehen können. Mit AspectVHDL soll deshalb im Folgenden ein modularer Ansatz vorgestellt werden, mit dem sich diese Erweiterungen in separaten Aspekten implementieren und bei Bedarf durch einen Aspektweber integrieren lassen.

4.4.4.2 Aspekte in VHDL

In VHDL beschriebene Hardwarestrukturen haben einen strikt hierarchischen Aufbau: die Komponente auf oberster Ebene kann Komponenten enthalten und verbinden, welche wiederum weitere Unterkomponenten enthalten und verbinden können. Die Komponenten setzen sich, wie bereits angemerkt, aus der Schnittstellenbeschreibung und aus der Architekturbeschreibung zusammen, wobei es zu einer Schnittstelle verschiedene Architekturen geben kann. Der MB-Lite Prozes-

[5]Die Gleitkommaeinheit gehört normalerweise nicht zum MB-Lite Prozessor und wurde für ein früheres Projekt nach dem gleichen Schema wie *Barrel-Shifter* und Multiplizierer integriert.

sor hat allerdings genau eine Architektur für jede Komponente. Die VHDL-Pakete enthalten in der Regel wiederverwendbare Definitionen von Typen, Funktionen oder Prozeduren, welche innerhalb der Komponenten genutzt werden können. Funktionen und Prozeduren enthalten Sequenzen von VHDL-Befehlen und kapseln wiederverwendbare Logik, Berechnungen oder Signalzuweisungen. Im Gegensatz zur Softwareentwicklung führt der „Aufruf“ einer Funktion oder einer Prozedur innerhalb einer Architektur jedoch zur Instanziierung der Hardwarestruktur, welche durch die Funktion bzw. Prozedur beschrieben wurde, denn Dinge wie das Konzept des Stapels für die Funktionsaufrufe oder die Wiederverwendung von Binärcode, gibt es auf Hardwareebene nicht. Dennoch bietet VHDL auf der Quelltextebene Funktionen und Prozeduren, welche Softwareentwicklern sehr vertraut erscheinen. Eine weitere interessante Eigenschaft von VHDL ist, dass Befehle nicht nacheinander abgearbeitet werden, wenn diese im nebenläufigen Teil der Architektur beschrieben werden, denn gegenüber Software gibt es hier keinen Prozessor der die Befehle ausführen könnte, sondern es wird die Hardware selbst beschrieben. Wenn Teile der Hardware sequentiell beschrieben werden sollen, muss der Entwickler dieses Verhalten explizit in Prozessen innerhalb einer Architektur implementieren.

Join Points Der Startpunkt für die im weiteren Verlauf vorgestellte Aspekt-Erweiterung von VHDL ist das *Join Point*-Modell. Das *Join Point*-Modell bestimmt die Ereignisse oder Strukturen, an denen die Aspekte später wirken können. In dieser ersten Variante von AspectVHDL werden die Folgenden *Join Points* unterstützt:

Prozeduren/Funktionen Alle Befehle in Funktionen oder Prozeduren werden sequentiell abgearbeitet – ungeachtet dessen, ob diese innerhalb eines Prozesses oder in dem nebenläufigen Teil einer Architektur genutzt werden. Dies macht Funktionen und Prozeduren zu idealen Kandidaten für die klassischen *Before/After/Around Advices*. Anders als Funktionen, haben Prozeduren keinen Resultatstyp und Rückgabewert. Im Falle von Prozeduren können jedoch `in`, `out` und `inout` Parameter verwendet werden, um den Signalfluss zu gewährleisten.

Typen Durch die Modifizierung von Typen ist es möglich, strukturelle Erweiterungen in verschiedenen Komponenten einzuführen, da Typen üblicherweise an verschiedenen Punkten im Quelltext verwendet werden. Die erste Variante von AspectVHDL unterstützt die Modifikation von *Record*- und *Enumeration*-Typen.

Architekturen Da Architekturen die Struktur und das Verhalten der Komponenten beschreiben, muss es möglich sein, auch hier neue Quelltextkonstrukte, wie zum Beispiel nebenläufige Befehle, Prozesse oder Signale, einzufügen. In AspectVHDL können diese Erweiterungen in Fragmenten gruppiert werden, welche – angelehnt an AspectC++ – *Slices* genannt werden.

```
1 within(core_Pkg) and type(ctrl_execution)
2 within(arch of execute) and process(execute_comb)
3 within(* of core) and call(enable(*))
4 architecture(* of *)
```

Abbildung 4.15: Beispiele für *Pointcuts* in AspectVHDL

Sensitivitätslisten Im Gegensatz zu Funktionen oder Prozeduren, werden Prozesse nicht direkt „aufgerufen“, sondern infolge von Signaländerung ausgelöst. Welche Signale einen Prozess beeinflussen, wird durch die Sensitivitätsliste festgelegt. Aspekte sollen die Möglichkeit besitzen, neue Signale einer Sensitivitätsliste hinzufügen zu können.

Pointcut-Ausdrücke Aspektorientierte Sprachen bieten typischerweise spezielle syntaktische Ausdrücke an, um einen Satz von *Join Points*, an denen ein Aspekt wirken soll, beschreiben zu können. Diese Ausdrücke werden in der aspektorientierten Programmierung als *Pointcuts* bezeichnet. Die *Pointcut*-Ausdrücke, welche für AspectVHDL entworfen wurden, spiegeln eins-zu-eins das *Join Point*-Modell wider. Die *Pointcuts* in Abbildung 4.15 zeigen beispielhaft einige der Möglichkeiten, die mit dieser Spracherweiterung von VHDL geboten werden und für die Fallstudie zur Erweiterung des MB-Lite Prozessors verwendet wurden. Die *Pointcut*-Funktionen `type`, `process`, `call` und `architecture` werden benutzt, um *Join Points* von bestimmter Art und Namen zu identifizieren. Die `process`-Funktion in Zeile zwei wird beispielsweise verwendet, um den Prozess `execute_comb` zu erkennen. Aktuell kann dieser *Join Point* allerdings nur dazu verwendet werden, um die Sensitivitätsliste eines Prozesses zu erweitern und nicht zur Erweiterung eines Prozesses um zusätzliche VHDL-Befehle. Die `call`-Funktion kann zur Identifizierung einer Funktion bzw. Prozedur verwendet werden. Für AspectVHDL macht es – im Gegensatz zu den bekannten Aspekt-Erweiterungen für Softwaresprachen – keinen Sinn, explizit durch verschiedene *Pointcut*-Funktionen zwischen dem „Aufruf“ oder der „Ausführung“ einer Funktion bzw. Prozedur zu unterscheiden, da Funktionen und Prozeduren ohnehin für ihre jeweilige Verwendung gesondert in Hardware instanziiert werden müssen. Weshalb für AspectVHDL eine alleinige `call`-Funktion ausreicht, um die *Join Points* von Funktionen und Prozeduren zu beschreiben. Wie auch in anderen aspektorientierten Sprachen können Platzhaltersymbole für Argumente oder Rückgabetypen (nur Funktionen) verwendet werden. Um einen *Join Point* eindeutig zu bestimmen, kann die *Pointcut*-Funktion `within` eingesetzt werden. Diese kann in Kombination mit Typen, Prozessen oder Funktionen bzw. Prozeduren verwendet werden, wie in den Zeilen eins bis drei zu sehen ist. Die Argumente der `within`-Funktion beschreiben entweder ein Paket (Zeile 1) oder eine Architektur (Zeilen 2 und 3). Da der Name einer Architektur nur innerhalb einer Komponente eindeutig ist, wird für AspectVHDL die Syntax `<Architekturname> of`

```
1 advice around(...) : within(arch of core)
2     and call(enable(*))
3     and args(ena_i, mem_i.ena_i, stall) is
4         ena_i <= mem_i.ena_i and (not stall);
5 end advice;
```

Abbildung 4.16: Beispiel für einen *Around Advice* in AspectVHDL

`<Komponentenname>` benutzt. Um mit AspectVHDL auch komplexere *Pointcut*-Ausdrücke zusammensetzen zu können, werden die algebraischen Operatoren `and`, `or` und `not` unterstützt. Die *Pointcut*-Funktion `args` wird in dem nächsten Abschnitt genauer erläutert.

Advice-Quelltext In AspectVHDL werden zwei Typen von *Advices* unterstützt: *Before/After/Around Advices* und *Advices* zur Einfügung (engl. *Introduction*). Einfügungen können verwendet werden, um entsprechende Quelltextfragmente (engl. *Slices*) in Typen, Architekturen oder Sensitivitätslisten einzufügen. *Before/After/Around Advices* können hingegen zur „Umleitung des Kontrollflusses“ im Falle von Funktionen oder Prozeduren verwendet werden.
Der *Around Advice* in Abbildung 4.16 fängt zum Beispiel jeden „Aufruf“ der Prozedur `enable()` in der Architektur `arch` der Komponente `core` ab. Innerhalb des *Around Advice* kann die von AspectVHDL zur Verfügung gestellte Prozedur `proceed` genutzt werden, um die ursprüngliche Funktion oder Prozedur einzubinden. In dem hier gezeigten Beispiel wird `proceed` nicht verwendet und anstelle der ursprünglichen Prozedur wird letztendlich die im Rumpf des *Advices* implementierte Signalzuweisung durchgeführt. Die im *Advice*-Rumpf verwendeten Bezeichner werden mittels der `args` *Pointcut*-Funktion an die Funktions- bzw. Prozedurargumente gebunden. Dies entspricht dem gleichen Mechanismus, wie er auch in AspectJ oder AspectC++ verwendet wird. Die Typen der Bezeichner müssen in der Argumentliste des *Around Advices* deklariert werden, welche in diesem Beispiel durch Punkte angedeutet sind.
Mit Einfügungen ist es möglich, beispielsweise VHDL-Befehle im deklarativen Teil einer Architektur einzubinden, neue Prozesse in eine Architektur einzufügen oder Typen wie *Records* um neue Elementdeklarationen zu erweitern. In Abbildung 4.17 ist die Erweiterung eines *Records* dargestellt. Ein *Record* ist, ähnlich wie ein *Struct* in C, ein zusammengesetzter Typ. Die *Slice* beschreibt in diesem Beispiel das in den *Record* einzufügende Quelltextfragment. Die Einfügung wirkt in diesem Fall bei dem *Record* `ctrl_execution` in dem VHDL-Paket `core_Pkg`.

Aspekte Wie auch in anderen aspektorientierten Sprachen werden Aspekte zur Gruppierung von *Advices* und *Slices* genutzt. In der Fallstudie zum MB-Lite Prozessor ist der Quelltext jedes optionalen Beschleunigers in einem eigenen Aspekt gekapselt. Innerhalb eines Aspekts ist es möglich, die üblichen VHDL-Konstrukte, wie Funktionen, Prozeduren, Typen, Konstanten oder Signale zu nutzen als auch

```
1  slice ctrl_slice is record
2      fpu_op : fpu_operation;
3  end record ctrl_slice;
4  ...
5  advice slice : within(core_Pkg)
6    and type(ctrl_execution) is ctrl_slice;
```

Abbildung 4.17: Beispiel für eine Einfügung in AspectVHDL

```
1  aspect FPU is
2      type fpu_states is (idle, running, ready);
3
4      advice around() : within(arch of execute)
5        and call (execution(*)) is
6          ...
7      end advice;
8  end aspect FPU;
```

Abbildung 4.18: Grundgerüst des Aspekts für die Gleitkommaeinheit

Unterkomponenten zu instanziieren. Das Beispiel in Abbildung 4.18 zeigt das Grundgerüst für den Aspekt zur Erweiterung des Prozessors um die Gleitkommaeinheit. In diesem Beispiel wird in dem Aspekt der zusätzliche Typ `fpu_state` definiert und ein *Around Advice* implementiert, welcher beim „Aufruf“ der Prozedur `execution()` innerhalb der Komponente `execute` wirkt. Da Aspekte potenziell den gesamten VHDL-Quelltext betreffen können, muss ein Mechanismus zur Verfügung stehen, um die implementierten Aspekte automatisiert und effizient zu identifizieren. Der Aspekt-Quelltext wird deshalb für AspectVHDL in Dateien mit der Endung „.avhd“ abgelegt.

Syntax von AspectVHDL Die vollständige Syntax von AspectVHDL ist in Abbildung 4.19 in Backus-Naur-Form dargestellt, um eine präzise Definition der Spracherweiterung und ihrer derzeit vorhanden Möglichkeiten zur Verfügung zu stellen. Die Syntax ist ähnlich wortreich wie VHDL und stark an diese Sprache angelehnt, damit Hardwareentwickler, auch mit der Aspekterweiterung, in einem vertrauten Umfeld arbeiten können. Die hier präsentierte Syntax von AspectVHDL muss als Erweiterung der Grammatik aus dem IEEE Standard 1076-1993 [The93] verstanden werden und reiht sich dort nahtlos ein.

4.4.4.3 Evaluation

Um die Anwendungsmöglichkeiten von AspectVHDL an einem realen VHDL-Projekt zu zeigen, soll die Integration der optionalen Beschleunigerkomponenten für den eingangs präsentierten MB-Lite Prozessor nun mit den hier vorgestellten Sprachmitteln realisiert und untersucht werden. Dazu müssen die bisherigen Implementierungen der optionalen Komponenten mittels *Generics* zunächst aus

```
aspect ::=
    aspect identifier is
      aspect_declaration
    end [ aspect ] [ aspect_name ];

aspect_declaration ::=
    { aspect_declarative_item }

aspect_declarative_item ::=
      advice
    | slice
    | subprogram_declaration
    | subprogram_body
    | aspect_type_declaration
    | constant_declaration
    | signal_declaration
    | component_declaration

advice ::=
      introduction_advice
    | subprogram_advice

introduction_advice ::=
    advice slice : pc_expr is slice_name;

subprogram_advice ::=
    advice advice_type : pc_expr is
      subprogram_statement_part
    end [ advice ];

advice_type ::=
      before(formal_parameter_list)
    | after(formal_parameter_list)
    | around(formal_parameter_list)

slice ::=
    slice identifier is aspect_slice_def;

pc_expr ::=
      pc_expr and pc_expr
    | pc_expr or pc_expr
    | not pc_expr
    | (pc_expr)
    | builtin_pointcut_function

builtin_pointcut_function ::=
      call(subprogram_pat)
    | architecture(name_pat of name_pat)
    | type(name_pat)
    | process(name_pat)
    | within(name_pat [ of name_pat ])
    | args(parameter_pat)

aspect_slice_def ::=
      aspect_architecture_def
    | aspect_type_def
    | aspect_process_def

aspect_architecture_def ::=
    architecture is
      architecture_declarative_part
    begin
      architecture_statement_part
    end [ architecture ]

aspect_type_def ::=
      enumeration_type_definition
    | record_type_definition

aspect_process_def ::=
    process(sensitivity_list)
```

Abbildung 4.19: Erweiterung der VHDL-Grammatik um AspectVHDL

	VHDL	VHDL (refakt.)	AspectVHDL
max. Frequenz (MHz)	17,366	17,366	17,396
Slices	8.712	8.530	8.479
LUTs	13.671	13.302	13.277
Flipflops	2.883	2.884	2.849

Tabelle 4.1: Ergebnisse der Synthese

dem Prozessor entfernt werden und über Aspekte erneut eingefügt werden, um anschließend Vergleiche zwischen den beiden Varianten bezüglich der benötigten Ressourcen und der zu erreichenden Frequenzen auf einem FPGA anstellen zu können.
Bei einem genaueren Blick auf den Quelltext fällt jedoch auf, dass die Implementierung die benötigten *Join Points* zur Nutzung von Aspekten vermissen lässt. So sind zum Beispiel die *Pipeline*-Stufen zur Dekodierung oder zur Ausführung der Befehle in sehr langen Prozessen implementiert, in denen nahezu keine Funktionen oder Prozeduren verwendet werden. Dieser „Spaghetticode“ macht es zunächst unmöglich Aspekte für die Erweiterung der *Pipeline*-Stufen zu nutzen. Um die Aspekte trotzdem für den MB-Lite Prozessor verwenden zu können, wurde der Quelltext refaktorisiert, indem Quelltextfragmente aus den langen Prozessen in Prozeduren ausgelagert wurden.
Basierend auf der refaktorisierten, modularen Variante des MB-Lite Prozessors, wurden die drei Erweiterungen entfernt und durch drei separierte Aspekte für Multiplizierer, *Barrel-Shifter* und Gleitkommaeinheit ersetzt. Die auf Aspekten basierende Variante vermeidet dabei vollständig die in Abbildung 4.14 gezeigte Problematik der Verstreuung des Quelltextes. Das Einweben des Quelltextes wurde durch einen im Rahmen einer Bachelorarbeit implementierten Aspektweber [Ste13] durchgeführt, welcher den gesamten Umfang von AspectVHDL 1.0 unterstützt. Der gewobene VHDL-Quelltext enthält dann ausschließlich Befehle nach dem VHDL-Standard und kann dementsprechend in beliebigen Synthesewerkzeugen oder Simulatoren verwendet werden.
Die Ergebnisse in Tabelle 4.1 zeigen einen Vergleich zwischen der ursprünglichen Variante des MB-Lite Prozessors, der refaktorisierten Implementierung und der Variante mit eingewobenen Aspekten. Die Syntheseergebnisse stammen von dem Synthesewerkzeug des Xilinx ISE 10.1.03 mit den Standardeinstellungen für ein Xilinx Spartan-3E XC3S1200E (*Speed Grade* -4). Die maximal mögliche Frequenz liegt für die drei Hardwaredesigns recht nah beinander, mit einem leichten Vorteil für die Variante mit AspectVHDL. Die leichten Unterschiede zwischen den Varianten sind aufgrund der proprietären Synthesewerkzeuge der Hersteller nur schwer zu erklären. Positiv sind allerdings die Hinweise, dass die hier präsentierten Mechanismen zur Unterstützung von Aspekten in VHDL nicht unbedingt einen negativen Einfluss auf die mit einem Hardwaredesign zu erreichenden Frequenzen haben. Dieselbe Beobachtung lässt sich auch für die Ressourcen des FPGAs, wie *Slices*, LUTs oder Flipflops feststellen, denn auch hier lassen sich nur Vermutun-

gen anstellen, warum die Variante mit AspectVHDL im Vergleich zu den anderen beiden Varianten einen etwas geringeren Ressourcenbedarf auf einem FPGA hat.

4.4.4.4 Fazit

Die Fallstudie um den MB-Lite Prozessor zeigt, dass bereits diese relativ einfache Aspekterweiterung für VHDL produktiv eingesetzt werden kann, um den Quelltext von praxisnahen Hardwaredesigns zu modularisieren und maßzuschneidern, ohne das Hardwaredesign bezüglich der benötigten Ressourcen oder der maximalen Frequenz negativ zu beeinflussen. Aktuell unterstützt AspectVHDL im Wesentlichen Konzepte, welche in ähnlicher Form auch in der Domäne der aspektorientierten Softwareentwicklung zu finden sind. Zukünftig könnte AspectVHDL noch um neue Konzepte erweitert werden, um den speziellen Bedürfnissen der Hardwareentwicklung gerechter zu werden. Dies könnte zum Beispiel über sehr feingranulare sowie hardwarespezifische *Join Points* realisiert werden. Beispielsweise könnten *Join Points* zur Modifizierung von Übergängen in Zustandsautomaten nützlich sein, denn Zustandsautomaten sind ein häufig eingesetztes Mittel im Kontext des Hardwaredesigns.
Eine weitere denkbare Erweiterung für AspectVHDL wäre zum Beispiel eine Unterstützung für die Weiterleitung von Signalen, denn durch den typischerweise, hierarchischen Aufbau der Hardwaredesigns, ist es unter Umständen sehr aufwendig zwei Komponenten über Signale miteinander zu verknüpfen. Dies gilt insbesondere dann, wenn Signale einer Unterkomponente temporär zu *Debug*-Zwecken mit der „Außenwelt“ verbunden werden sollen. Ein neuer *Join Point*-Typ zu diesem Zwecke, würde die Implementierung dieser Art der Signalweiterleitung erheblich vereinfachen und zugleich modularisieren.
Auch die bisher vorgestellte Sprache zur Beschreibung der *Pointcut*-Ausdrücke bietet noch Potenzial für Erweiterungen, denn diese unterstützt zwar die Auswahl einer Architektur, unterscheidet aber nicht zwischen den einzelnen Instanzen. Daraus ergibt sich, dass ein Aspekt alle Instanzen einer Architektur in derselben Art und Weise beeinflussen würde. Bezogen auf die Fallstudie zum MB-Lite Prozessor bedeutet dies, dass wenn der VHDL-Quelltext des Prozessors in einem *Multicore*-System verwendet wird, alle Prozessoren die gleichen Beschleuniger integriert hätten. Abhilfe könnte eine *Pointcut*-Funktion `instance` schaffen, mit der sich dann durch die Angabe des jeweiligen Labels für eine Instanz, diese explizit auswählen lässt.

4.4.5 Vergleich der vorgestellten Techniken

Ein adäquater Vergleich zwischen den vier präsentierten Techniken zur Implementierung der Variabilität im Lösungsraum ist schwierig, da sich nicht mit allen Techniken der volle Umfang der LavA Hardware-Produktlinie umsetzen lässt. Im Falle von AspectVHDL 1.0 liegt dies beispielsweise am zurzeit noch eingeschränkten *Join Point*-Modell, mit dem aktuell noch nicht alle notwendi-

```
1  entity lava is
2      generic(cpu : cpu_array := ((0, true), (2, false)));
3      port(...);
4  end lava;
5
6  architecture behavioral of lava is
7  begin
8      for i in 0 to nodes - 1 generate
9          node generic map(cpu => cpu(i)) port map(...);
10     end generate;
11 end behavioral;
```

Abbildung 4.20: *Generic* zur Konfigurierung der Prozessoren

```
1  constant nodes  : integer := 2;
2
3  type cpu_config is record
4      cpu_type    : integer range 0 to 2;
5      jtag        : boolean;
6  end record;
7
8  type cpu_array is array (0 to nodes - 1) of cpu_config;
```

Abbildung 4.21: Deklarierung der komplexen Typen

gen Punkte beschrieben werden können, an denen die Aspekte für die LavA Hardware-Produktlinie wirken müssten.

Aber auch die alleinige Verwendung von VHDL-Sprachmitteln zur Maßschneiderung der gesamten LavA Hardware-Produktlinie bereitet Schwierigkeiten aufgrund der gegebenen Limitierungen bei Verwendung dieser Sprachmittel. Das gesamte Konfigurationswissen für das System müsste bei diesem Ansatz in der Wurzelkomponente spezifiziert werden und von dort bis zu den Komponenten an den Blättern des Systems durchgereicht werden. Ein Beispiel zu diesem Vorgehen ist in Abbildung 4.20 dargestellt. In diesem Beispiel beschränkt sich das Konfigurationswissen allein auf die Parameter zur Konfigurierung von zwei Prozessoren in dem System. Der verwendete Datentyp für den *Generic* `cpu` ist in einem separaten VHDL-Paket deklariert (siehe Abbildung 4.21) und beschreibt ein Feld, dessen Größe mit der Anzahl der konfigurierten Knoten bzw. Prozessoren korrespondiert. Jedes Element dieses Feldes beschreibt dabei die Konfigurationsinformationen für einen Prozessor. Damit ein Element dieses Feldes mehrere Konfigurationsinformationen für einen Prozessor aufnehmen kann, setzen sich die Elemente des Feldes aus dem komplexen Typ `cpu_config` zusammen, der in dem VHDL-Paket über einen *Record* deklariert wird. In diesem Beispiel können sowohl der Prozessortyp als auch die JTAG-Unterstützung über den *Generic* konfiguriert werden. Die Instanziierung der zwei Knoten erfolgt generisch über die *Generate*-Anweisung in Abbildung 4.20. Den hier instanziierten Knoten wird nur noch das

		Konfigurationscode	VHDL-Code
XVCL	node_top.xvcl	131	56
	cfg_node_top.xvcl	214	—
	Summe	**345**	**56**
EMF	node_top.xpt	84	54
	node_top.ext	36	—
	Summe	**120**	**54**

Tabelle 4.2: Vergleich zwischen XVCL und EMF (Angaben in Zeilen)

jeweils relevante Element des Feldes über einen *Generic* übergeben, um in diesen Unterkomponenten den entsprechenden Prozessor zu instanziieren.

Wenngleich der hier gezeigte Ansatz für die Konfigurierung der Prozessoren des Systems durchaus noch sinnvoll erscheint, lässt sich dieses Vorgehen nicht auf die gesamten Konfigurationsinformationen der LavA Hardware-Produktlinie übertragen. Schwierig wird es immer dann, wenn die Anzahl der Konfigurationsinformationen nicht für alle Knoten identisch ist. So kann zum Beispiel die Anzahl der Peripheriekomponenten oder die Anzahl der angeschlossenen Kommunikationsstrukturen für jeden Knoten unterschiedlich sein, wodurch dieser Ansatz für diese Konfigurationsinformationen nicht mehr anwendbar ist. Aufgrund dieser Schwierigkeiten und den schon in Unterkapitel 4.4.1 geschilderten Einschränkungen, wurde die LavA Hardware-Produktlinie nicht allein auf Basis von VHDL-Sprachmitteln umgesetzt.

Vollständig implementiert wurde die LavA Hardware-Produktlinie hingegen mit den wesentlich mächtigeren Mitteln aus der *Frame*-Technologie (XVCL) und Codegeneratoren aus der modellgetriebenen Entwicklung (*Eclipse Modelling Framework*), welche im Folgenden gegenübergestellt werden sollen. Wie bereits in Unterkapitel 4.4.3 ausgeführt, ähneln sich diese beiden Techniken bezüglich der Art und Weise wie der VHDL-Quelltext konfiguriert wird deutlich, wobei die Implementierung mittels der modellgetriebenen Entwicklung mit wesentlich kompakterem Konfigurationscode zu realisieren ist. Um die Unterschiede zwischen den Techniken bezüglich des erforderlichen Konfigurationscodes quantitativ bewerten zu können, wird nachfolgend die hochkonfigurierbare Knotenkomponente untersucht, in der Prozessor und Peripheriekomponenten instanziiert sowie deren Vernetzung konfiguriert werden muss. Dazu wurden sowohl die Zeilen mit Konfigurationscode als auch die Zeilen mit VHDL-Quelltext für all diese Dateien ausgewertet, welche zur Generierung des Grundgerüsts der Knotenkomponente von Belang sind. Die Ergebnisse dieses Vergleiches sind in Tabelle 4.2 dargestellt. Dabei ist deutlich zu erkennen, dass im Vergleich zu der Umsetzung mit dem *Eclipse Modelling Framework* (EMF), die XVCL-Variante knapp die dreifache Menge an Konfigurationscode-Zeilen benötigt, um die Knotenkomponente der LavA Hardware-Produktlinie zu generieren.

4.5 Zusammenfassung

Die Abläufe und Techniken aus der Software-Produktlinienentwicklung lassen sich größtenteils auch auf die Entwicklung der LavA Hardware-Produktlinie übertragen. Einschränkungen gibt es zum Beispiel bei den Merkmalmodellen, deren Stärken nicht in der Darstellung der aufgezeigten strukturellen Konfigurationsinformationen liegen. Diese Einschränkungen bedeuten allerdings nicht, dass die Merkmalmodelle gänzlich ungeeignet zur Entwicklung einer Hardware-Produktlinie sind, wie zum Beispiel anhand der transaktionalen Speicherfamilie (TMPL) verdeutlicht, sondern vielmehr ist bei der Wahl der adäquaten Technik zu berücksichtigen, ob die zu entwerfende Hardware-Produktlinie ähnliche strukturelle Abhängigkeiten zwischen den Komponenten aufweist, wie dies bei der LavA Hardware-Produktlinie der Fall ist. Sollte sich also in der frühen Phase der Domänenanalyse abzeichnen, dass diese strukturellen Konfigurationsinformationen abgebildet werden müssen, bieten die Ansätze aus der modellgetriebenen Entwicklung in Form der Metamodelle, eine passende Alternative, um den Problemraum der Produktlinie zu modellieren. Ungeachtet dessen, verschaffen die Merkmalmodelle dem Entwickler zumindest einen ersten Eindruck von dem Umfang des Problemraums und damit der Variabilität der zu entwerfenden Produktlinie, um den Entwickler in einer frühen Phase der Entwicklung zu unterstützen.

Die Techniken zur Implementierung der Variabilität im Lösungsraum haben auch im Kontext der Hardware-Produktlinien mit ähnlichen Schwierigkeiten zu kämpfen, denn abhängig von der verwendeten Technik und dem Umfang der Produktlinie kann die Wartbarkeit und die Erweiterung der Produktlinie womöglich negativ beeinflusst werden, weshalb die Auswahl von adäquaten Techniken eine wichtige Rolle spielen kann. Nach welchen Kriterien eine passende Technik bestimmt werden kann, wird im Folgenden anhand der LavA Hardware-Produktlinie diskutiert.

Eine Übersicht über die LavA Hardware-Produktlinie ist in Abbildung 4.22 dargestellt. Für die LavA Hardware-Produktlinie wird zur Modellierung des Problemraums ein Metamodell aus der modellgetriebenen Entwicklung eingesetzt, das sowohl die Variabilität der Produktlinie als auch den Bauplan zur Ableitung von konkreten Produkten vorgibt. Die aus dem Metamodell abgeleiteten Modelle bilden in abstrakter Form ein konkretes Problem aus dem Problemraum ab. Zudem umfassen diese Modelle das gesamte Konfigurationswissen für die zu generierenden Produkte.

Zur Implementierung der Variabilität im Lösungsraum werden in der LavA Hardware-Produktlinie drei der präsentierten Techniken eingesetzt und zwar die Sprachen des EMF, die Spracherweiterung AspectVHDL und die VHDL-*Generics*. Jede dieser drei Techniken besitzt spezifische Stärken und Schwächen, wodurch sich die Techniken jeweils für unterschiedliche Anwendungsfälle anbieten. Ein Großteil der Variabilität der LavA Hardware-Produktlinie wird über die Sprache Xpand des EMF ausgedrückt, denn diese Sprache bietet insbesondere das Potenzial, auch die hochkonfigurierbaren Komponenten der Produktlinie,

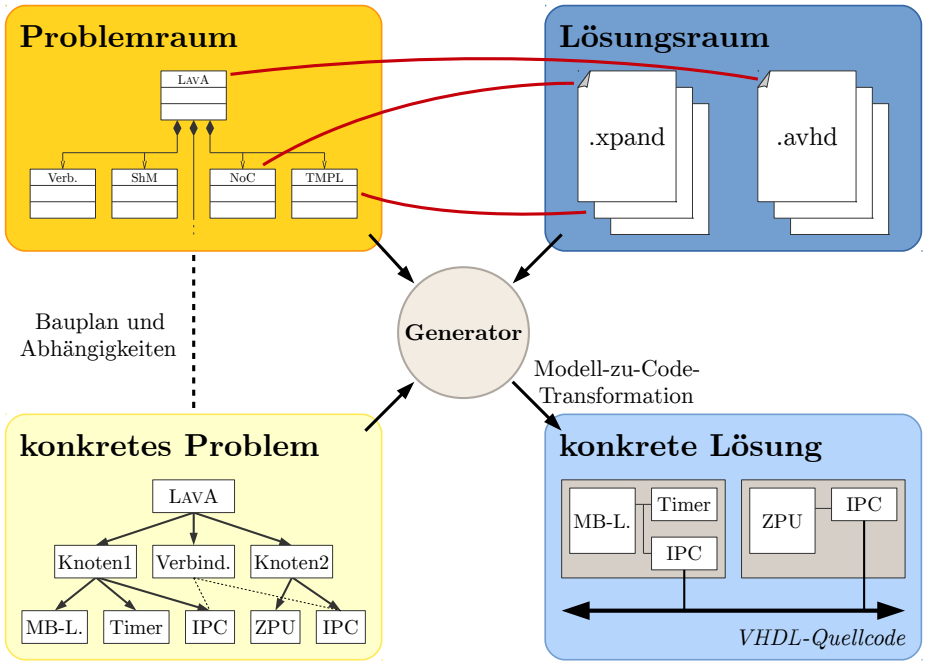


Abbildung 4.22: Ableitung eines konkreten Produktes aus der LavA Hardware-Produktlinie

wie die LavA- oder die Knoten-Komponente, in denen die komplexen Verbindungsstrukturen implementiert sind, maßzuschneidern. Aufgrund der ähnlichen Ausdrucksstärke, ermöglicht auch der *Frame*-Prozessor XVCL die Generierung dieser Strukturen, jedoch auf Kosten des wesentlich ausführlicheren Konfigurationscodes (siehe Unterkapitel 4.4.5), der damit einhergehenden schwierigeren Wartbarkeit und dem fehlenden Zugriff auf das Metamodell sowie auf das abgeleitete Modell, weshalb für die LavA Hardware-Produktlinie auf diese Technik verzichtet wird. Die Stärke der VHDL-*Generics* liegt im Wesentlichen in der einfachen Parametrisierung von Subkomponenten, wie zum Beispiel der Parametrisierung von Busbreiten oder der Baudrate für einen UART. Der Vorteil bei dieser Art der Konfigurierung ist, dass der Wert nun über den *Generic* durch die Subkomponenten propagiert werden kann und so der VHDL-Quelltext der Subkomponenten nicht zwingend mit weiterem Konfigurationscode einer anderen Sprache vermischt werden muss. Die Subkomponente kann demnach ohne den Einsatz eines Codegenerators, beispielsweise in VHDL-Simulatoren getestet oder in beliebigen anderen Projekten wiederverwendet werden. AspectVHDL wird in der LavA Hardware-Produktlinie bisher für die optionalen Beschleunigerkomponenten des MB-Lite Prozessors verwendet. Diese Maßschneiderung wäre zwar auch über die anderen Techniken realisierbar, allerdings bietet AspectVHDL den Vorteil, der sehr losen Kopplung von Prozessor und optionalen Beschleunigerkomponenten. Mittels AspectVHDL können die Belange der Beschleunigerkomponenten in separaten Aspekten implementiert werden, welche nur bei Bedarf in

den Prozessor eingewoben werden, wodurch der eigentliche VHDL-Quelltext des Prozessors für einen Entwickler unberührt und frei von Konfigurationscode bleibt. Falls bei der LavA Hardware-Produktlinie die Konfiguration für ein abzuleitendes Produkt eine Beschleunigerkomponente vorsieht, wird der entsprechende Aspekt bei der Modell-zu-Code-Transformation des EMF kopiert und anschließend über den Aspektweber in die generierten VHDL-Dateien eingewoben.
Durch die Verflechtung der LavA Hardware-Produktlinie mit den Techniken aus der modellgetriebenen Entwicklung, können die gewünschten *Multi-* bzw. *Many-core*-Systeme so vollständig aus den vorgegebenen Modellen abgeleitet und der entsprechende VHDL-Quelltext generiert werden. Dies ermöglicht die Verwendung der Produktlinie in einem automatisierten Entwurfsprozess, in dem die Systeme ohne weitere Eingriffe des Entwicklers optimiert werden können.

Kapitel 5

Software-Produktlinien

Inhalt

Der Fokus dieser Arbeit liegt auf der Co-Konfiguration und der damit verbundenen Fragestellung, wie sowohl die Hardwareschicht als auch die Systemsoftwareschicht in einem durchgängigen Prozess konfiguriert und aufeinander abgestimmt werden können. Der in Abbildung 1.1 vorgestellte Ansatz dieser Arbeit verfolgt ein softwarezentriertes Konzept, bei dem das Konfigurationswissen anhand der Zugriffsmuster auf die Programmierschnittstellen zwischen den Schichten gewonnen und für die Maßschneiderung der darunterliegenden Schicht eingesetzt wird. Das Ziel dieses Kapitels ist es, sowohl die Programmierschnittstellen zwischen den Schichten als auch die verwendete Betriebssystem-Produktlinie im Detail vorzustellen. Die Softwareschichten werden in diesem Kapitel von unten nach oben vorgestellt. Zunächst wird deshalb erläutert, wie die Informationen zur Konfigurierung der Hardware aus der untersten Softwareschicht gewonnen werden können. Darauf folgend wird die eingesetzte Betriebssystem-Produktlinie CiAO vorgestellt und abschließend das parallele Programmiermodell, welches dem

Anwendungsentwickler zur Implementierung von parallelen Anwendungen dient. Die entwickelte Anwendung bildet zusammen mit den Zugriffsmustern auf die parallele Programmierschnittstelle den Ausgangspunkt für den in Abbildung 1.1 dargestellten Ansatz zur automatisierten Generierung von Hardware und Systemsoftware aus den jeweiligen Produktlinien.

5.1 Softwarezentrierte Ableitung der Hardware

Mit der in Kapitel 4 vorgestellten LavA Hardware-Produktlinie kann der Entwickler zwar auf einer hohen Abstraktionsebene die benötigten Hardwarestrukturen beschreiben und generieren, die Abstimmung zwischen der Softwareschicht und der Hardwarestruktur, muss der Entwickler jedoch selbst übernehmen. Insbesondere bei komplexer Systemsoftware, wie Betriebssystemen oder Kommunikationsbibliotheken, welche zudem im Bereich der Eingebetteten Systeme oftmals hochkonfigurierbar sind, kann es bei der manuellen Abstimmung zu Konfigurationsfehlern und damit zu Inkonsistenzen zwischen Hardware und Software kommen. Der in diesem Kapitel angestrebte softwarezentrierte Ansatz soll hingegen die nahtlose Co-Konfiguration von Hardware und Software erlauben, indem die Hardware automatisiert aus den Softwareschichten abgeleitet wird. Damit die Hardwarestrukturen aus der Software abgeleitet werden können, müssen die Hardwarekomponenten der *Multi-* bzw. *Manycore*-Systeme auf der Softwareebene in einer Form repräsentiert werden, in der die Hardwarekonfiguration zur Übersetzungszeit statisch aus der Software extrahiert werden kann.

In der Forschung gibt es bereits einige Ansätze, welche sich mit der Repräsentation von Hardware auf der Ebene von Software beschäftigen. Kumar *et al.* präsentieren in ihrer Arbeit [KAJW94] eine objektorientierte Technik um Hardware mittels C++ zu modellieren. Dabei richtet sich der Fokus dieser Arbeit auf die Zustände der Hardware und die entsprechenden Operationen, um diese Zustände zu modifizieren. Um Hardwarekomponenten zu erstellen, werden bei diesem Ansatz C++ Klassen zur Laufzeit instanziiert. Da bei diesem Ansatz die Konfigurationsinformationen zu den Hardwarestrukturen nicht über eine statische Analyse zur Übersetzungszeit gewonnen werden können, kann dieser Ansatz nicht zur Repräsentation der Hardwarekomponenten aus der LavA Hardware-Produktlinie eingesetzt werden. In [RR99] wird eine Entwurfsmethodik vorgestellt, mit der Hardware in C++ auf hoher Abstraktionsebene beschrieben werden kann. Zu diesem Zweck wurde C++ um neue Klassenbibliotheken erweitert, welche Nebenläufigkeit und Reaktivität zur Verfügung stellen, damit Hardware in Software beschrieben werden kann. Dieser Ansatz hat eine gewisse Ähnlichkeit mit der weit verbreiteten C++ Erweiterung SystemC. Dementsprechend ist dieser Ansatz eher dazu geeignet, das Verhalten von Hardware zu beschreiben, als die anwendungsspezifischen Anforderungen zur Konfigurierung von Hardwarekomponenten zu repräsentieren.

```
1 template < int Baud = 57600, int IRQ = 1 >
2 struct UART : public AbstractDevice<IRQ> {
3     enum { Size = 8 };
4 };
```

Abbildung 5.1: Das UART-*Template* – Teil der Hardware-API

5.1.1 Repräsentation von Hardware in Software

Da die zuvor angesprochenen Repräsentationen nicht statisch zur Übersetzungszeit analysiert werden können, muss ein anderer Mechanismus verwendet werden, um die Hardwarekomponenten der LavA Hardware-Produktlinie in Software zu repräsentieren. Da Software für Eingebettete Systeme häufig in C oder C++ programmiert wird, sind passende Sprachelemente dieser Sprachen die erste Wahl zur Repräsentation der LavA-Hardwarekomponenten. Das erforderliche Sprachelement auf Softwareebene muss dabei flexibel genug sein, um mit der umfangreichen Anzahl von möglichen Hardwarekonfigurationen umgehen zu können. Insbesondere, wenn Hardwarestrukturen, wie *Multi-* oder *Manycore*-Systeme, betrachtet werden, welche sich typischerweise aus Prozessoren, Kommunikationsinfrastruktur, Peripheriekomponenten und Speichern zusammensetzen. Ebenso muss das gewählte Sprachelement die Analyse des hardwarerepräsentierenden Quelltextes statisch zur Übersetzungszeit erlauben, um die Hardwarekomponenten zu ermitteln und anschließend zu konfigurieren. Zudem muss es möglich sein, Hardwarekomponenten von derselben Art mit verschiedenen Konfigurationen repräsentieren zu können. Andernfalls würde jede Instanz einer Hardwarekomponente die gleichen Eigenschaften besitzen, was die Konfigurierung der Hardwarestruktur unnötigerweise einschränken würde.

Ein Sprachelement das die aufgeführten Anforderungen zur Repräsentation der Hardware erfüllt, ist der C++ *Template*-Mechanismus. *Templates* sind ein sehr mächtiger und generischer Mechanismus. Sie erlauben die Konfigurierung der Hardwarekomponenten zur Übersetzungszeit, da die Sprache die Spezifizierung der Parameter als Konstanten erzwingt, was für die Nutzung von statischen Analysen von elementarer Bedeutung ist. Die diversen Hardwarekomponenten werden dazu mittels C++ Klassen-*Templates* für jede Art von Hardwarekomponente beschrieben. Abbildung 5.1 zeigt exemplarisch das Klassen-*Template* zur Repräsentation der UART-Komponente. Die UART-Komponente kann anhand von zwei Parametern konfiguriert werden: Zunächst kann die gewünschte Baudrate definiert werden und im Anschluss folgt die Konfigurierung der Unterbrechung für diese Komponente. Der benötigte Adressbereich für diese Komponente wird in Byte über die symbolische Konstante `Size` angegeben. Die Menge aller Klassen-*Templates* zur Repräsentation der diversen Hardwarekomponenten ergeben zusammen die sogenannte Hardware-API [MBS14], die unterste Schicht der Software.

```
1  // einfache Instanziierung
2  UART<9600, 0> singleUART;
3
4  // vielfache Instanziierung anhand von Feldern
5  UART<57600, 1> multipleUARTs[3];
6
7  // Instanziierung mittels Komposition
8  class CompUART {
9      public:
10         UART<19200, 1> memberUART;
11 } compUART;
12
13 // Instanziierung mittels Vererbung
14 class DerivedUART : public UART<> {} derivedUART;
```

Abbildung 5.2: Mögliche Varianten zur Instanziierung der Hardwarekomponenten

5.1.2 Instanziierung der Hardware in Software

Um eine Hardwareinstanz einer ausgewählten Komponente in Software zu erstellen, muss das korrespondierende Klassen-*Template* instanziiert werden. Verschiedene Beispiele zur Instanziierung der Hardware mit unterschiedlichen Parametern sind in Abbildung 5.2 dargestellt. In der zweiten Zeile wird beispielsweise gezeigt, wie ein UART-Objekt mit dem Bezeichner `singleUART` und den spezifizierten Parametern angelegt werden kann. Auch die Instanziierung von mehreren UARTs durch den Einsatz von Feldern lässt sich, wie in Zeile 5 dargestellt, mit der gezeigten Methodik realisieren. Durch die Verwendung von Kompositions- oder Vererbungstechniken hat der Entwickler zudem vielfältige weitere Möglichkeiten zur Instanziierung der Hardwarekomponenten. Da die Klassen-*Templates* das Bindeglied zwischen Hardware und Software darstellen, könnten diese beispielsweise direkt in den Gerätetreibern verwendet werden. Dadurch wäre es möglich, dass ein Gerätetreiber ein Hardwareobjekt als Attribut nutzt (Zeilen 8 bis 11) oder die Klasse des Gerätetreibers von dem Klassen-*Template* der Hardwarekomponente erbt (Zeile 14). So könnte die Hardwarekomponente indirekt instanziiert werden, sobald der entsprechende Gerätetreiber in der Software eingebunden wird.

5.1.3 Modellgetriebene Entwicklung der Hardware-API

Die Hardware-API erfüllt zwei verschiedene Zwecke im Zusammenhang mit dem LavA-Ansatz. Zum einen können die Instanzen der Hardware-API statisch analysiert und zur Konfiguration der LavA Hardware-Produktlinie verwendet werden und zum anderen wird der entstehende Quelltext zur Laufzeit für den Zugriff auf die synthetisierten Hardwarekomponenten verwendet. Die Hardware-API interagiert also direkt mit der Hardware und ist daher eng mit dieser verknüpft. Entsprechende Erweiterungen der Hardware haben deshalb unter Umständen auch Auswirkungen auf die Hardware-API. Damit sowohl die Hardware als auch die Hardware-API konsistent gehalten werden können, wird auch die Hardware-API

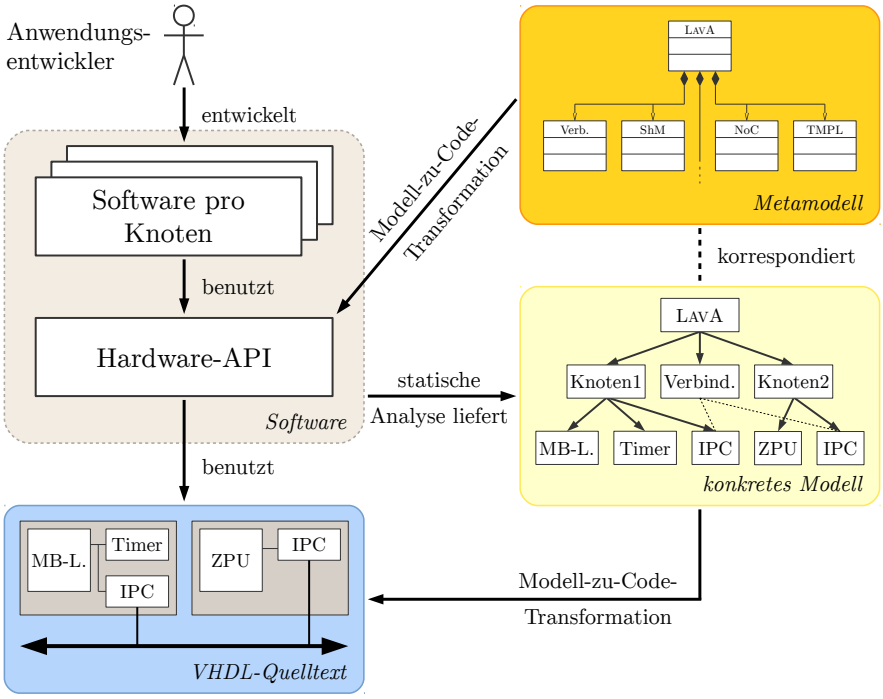


Abbildung 5.3: Modellgetriebene Entwicklung der Hardware-API

mittels modellgetriebener Ansätze automatisiert generiert. Das Gesamtbild dieses Prozesses wird in Abbildung 5.3 präsentiert. Die zentrale Rolle spielt dabei das Metamodell der LAVA Hardware-Produktlinie (Abbildung 4.4), denn neben der Repräsentation des Problemraums und der Abbildung des Bauplans zur Ableitung von konkreten Hardwaremodellen, werden über das Metamodell mittels Modell-zu-Code-Transformation auch die einzelnen Klassen-*Templates* der Hardware-API generiert.
Die Generierung der Klassen-*Templates* wird im Wesentlichen über die im Metamodell hinterlegten Attribute der einzelnen Hardwarekomponenten gesteuert. Attribute der Hardwarekomponenten, welche in dem Metamodell als Konstanten deklariert sind, beschreiben für alle Instanzen dieser Komponente einheitliche Eigenschaften und werden in dem korrespondierenden Klassen-*Template* als symbolische Konstante ausgedrückt. Im Gegensatz dazu bilden die anderen Attribute die Variabilität der Komponente ab und werden demzufolge als Parameter für das jeweilige Klassen-*Template* abgebildet. Auf diese Weise kann die Hardware-API bei Änderungen an der Hardware, stets automatisiert auf den neusten Stand gebracht werden und Inkonsistenzen so vermieden werden.
Auf der linken Seite von Abbildung 5.3 ist die Beziehung zwischen den Hardware- und Softwareschichten dargestellt. Der Anwendungsentwickler implementiert für die jeweiligen Knoten unter Verwendung der Hardware-API die entsprechende Software. Die Software kann dabei sowohl aus einfachen Anwendungen bestehen als auch eine mehrschichtige und komplexe Struktur samt Systemsoftware umfas-

sen. Wichtig ist nur, dass zum Zugriff auf die Hardware die zur Verfügung gestellte Hardware-API benutzt wird. Wenn der Entwickler nun für jeden Knoten, Prozessor und Peripheriekomponenten über die Hardware-API instanziiert hat, können dann zur Übersetzungszeit die Konfigurationsinformationen für die Hardwarestruktur anhand der *Template*-Argumente der Hardware-API extrahiert und ein korrespondierendes Modell der Hardware abgeleitet werden. Über das abgeleitete Modell kann dann in einem letzten Schritt der zugehörige VHDL-Quelltext zur Synthese der Hardware generiert werden.

5.1.4 Übertragbarkeit der Methodik

Die vorgestellte Methodik zur Repräsentation der Hardware mittels der Hardware-API ist nicht auf die LavA Hardware-Produktlinie beschränkt. So ist es zum Beispiel denkbar, die LavA Hardware-Produktlinie durch Xilinx' EDK oder eine softwarebasierende Emulation der Hardwarekomponenten zu ersetzen. Dazu ist es lediglich nötig, ein Metamodell zur Verfügung zu stellen, welches sowohl die Konfigurationsoptionen als auch die Transformation des Hardwaremodells in das Modell des jeweiligen Werkzeuges, wie zum Beispiel Xilinx' EDK, abbildet. Falls verschiedene Hardwareplattformen durch identische Metamodelle repräsentiert werden könnten, wären auch die daraus resultierenden Hardware-APIs kompatibel, so dass die Software zwischen den beiden Plattformen austauschbar wäre und der Wechsel zwischen den Plattformen nur eine Sache der passenden Konfigurierung wäre.
Zudem kann die Hardware-API sowohl in einer komplexen Systemsoftware als auch in einer einfachen Anwendung verwendet werden. In jedem Fall versteckt die Hardware-API die Hardwaredetails vor dem Anwendungsentwickler.

5.1.5 Fazit

Die in diesem Unterkapitel vorgestellte Methodik erlaubt die Maßschneiderung von *Multi-* bzw. *Manycore*-Systemen allein auf Basis von statischen Quelltextanalysen auf Softwareebene. Durch den automatisierten Konfigurationsprozess wird die Entwicklung von Anwendungen vereinfacht, da kein explizites Wissen über die Hardwarearchitektur notwendig ist und auch kein konkretes Modell der Hardware vom Entwickler „per Hand“ erstellt werden muss. Stattdessen kann der Anwendungsentwickler die gewünschten Hardwareobjekte direkt in der Software instanziieren, wobei die Hardware-API durchaus mit Hilfe verschiedener Abstraktionsschichten verborgen werden kann. Welche Vorteile die Hardware-API beispielsweise im Zusammenspiel mit einem Betriebssystem bringen kann, wird in dem Unterkapitel 5.2.2 gezeigt. Zudem können durch die automatisierte Ableitung der Hardware aus der Softwareschicht Konfigurationsfehler und Inkonsistenzen, wie sie sonst bei der manuellen Konfigurierung der Hardware auftreten können, schon allein durch die angewendete Methodik ausgeschlossen werden.
Durch die enge Bindung zwischen Hardware und Software über die Hardware-API kann eine Entwurfsraumexploration auf dieser Ebene nicht eingesetzt werden,

denn die Instanzen der Hardware-API repräsentieren exakt die später generierten Hardwarestrukturen. Um den benötigten Freiheitsgrad für eine Optimierung der Hardware-/Softwaresysteme zu erhalten und den Entwickler von schwierigen Entwurfsentscheidungen, wie der Zuordnung von Tasks zu Prozessoren oder der Wahl der passenden Kommunikationsinfrastruktur zu befreien, wird im späteren Verlauf dieses Kapitels die ebenfalls statisch analysierbare parallele Programmierschnittstelle vorgestellt (siehe Unterkapitel 5.3).

5.2 CiAO Betriebssystem-Produktlinie

Wie die LavA Hardware-Produktlinie, muss auch ein potenzielles Betriebssystem flexibel und anpassbar sein, um mit der konfigurierbaren Hardwareschicht und ebenso den Ressourcenbeschränkungen in der Domäne der Eingebetteten Systeme umgehen zu können. Für das LavA-Projekt wird das bereits in anderen Forschungsprojekten entstandene, hochkonfigurierbare Betriebssystem CiAO [LHSP+09] eingesetzt. Damit der Platzbedarf des Betriebssystems minimiert werden kann, wurde CiAO von seinen Entwicklern so konzipiert, dass sich dies über Merkmale konfigurieren lässt und nur die jeweils erforderlichen Funktionen ihren Weg in die generierbare Betriebssysteminstanz finden. Das rekursive Akronym CiAO steht für *CiAO is Aspect Oriented* und deutet damit bereits auf die verwendete Technik zur Implementierung der Variabilität in der Betriebssystem-Produktlinie CiAO hin, denn die Konfigurierbarkeit von CiAO wird mittels Konzepten aus der Aspektorientierten Programmierung umgesetzt. CiAO setzt die Betriebssystemstandards von OSEK/AUTOSAR um und unterstützt diverse Prozessorarchitekturen, angefangen bei der im Automobilbereich genutzten Infineon TriCore-Architektur bis zur ausgewachsenen x86-Architektur. Zudem wurde CiAO im Zuge einer Diplomarbeit [Vog10] im Kontext des LavA-Projektes für die MicroBlaze-Architektur portiert, auf welcher die MB-Lite bzw. MB-Lite+ Prozessoren basieren.

5.2.1 Schichtenarchitektur der Betriebssystem-Produktlinie

Die im vorigen Unterkapitel vorgestellte Hardware-API hat einen wesentlichen Einfluss auf die Sicht des CiAO-Betriebssystems auf die zugrunde liegende Hardware [MBS15]. Abbildung 5.4 stellt die Schichten der LavA Hard- und Software dar, welche aus der Systemschicht, der Hardwareschicht und der Hardware bestehen. Plattformunabhängige Betriebssystemfunktionen, wie der *Scheduler* oder der *Alarm Manager* sind Teil der Systemschicht und im CiAO-Kernel implementiert. Die Hardwareschicht ist abhängig von der zugrunde liegenden Hardwarearchitektur und ist in drei weitere Schichten unterteilt: die Hardwareabstraktionsschicht, die Hardwaretreiber und die bereits vorgestellte Hardware-API. Der Zweck der Hardwareabstraktionsschicht ist es, eine standardisierte, abstrakte Schnittstelle zu bieten, um die Portabilität des Kernels zwischen verschiedenen Hardwarearchitekturen zu gewährleisten. In der nächsten Schicht werden die ar-

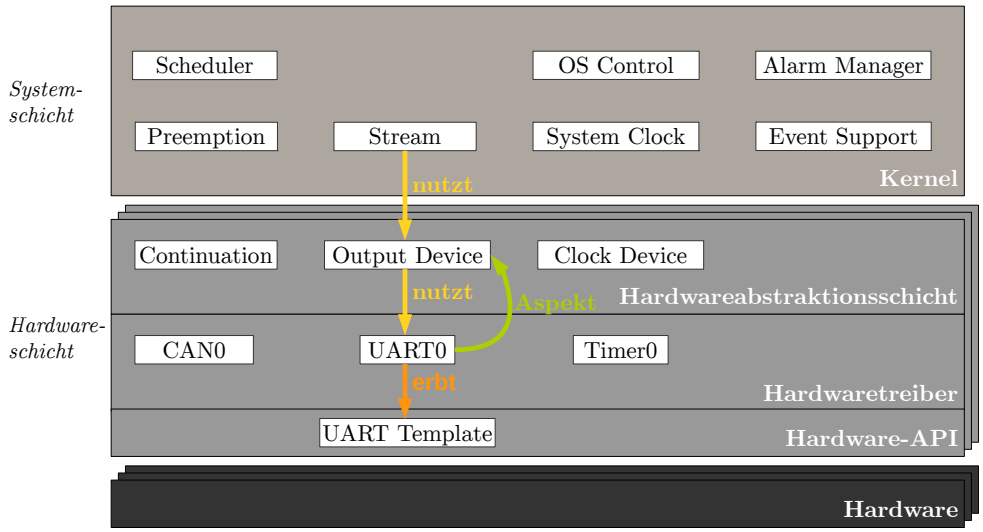


Abbildung 5.4: LavAs Hardware-/Softwareschichten

chitekturspezifischen Treiber implementiert, gefolgt von der Hardware-API. Diese beiden Schichten sind eng miteinander verknüpft, da der Entwickler in dem CiAO-Konfigurierungsprozess ausschließlich die erforderlichen Gerätetreiber konfiguriert und jeder ausgewählte Gerätetreiber von dem entsprechenden Klassen-*Template* der Hardware-API erbt. Dieses Vorgehen wird in Abbildung 5.4 anhand des Gerätetreibers `UART0` verdeutlicht. Durch diese Vorgehensweise kann das Betriebssystem einfach diejenigen Hardwarekomponenten instanziieren, welche – aus der Perspektive der Anwendung und des Betriebssystems – für die gegebenen Anforderungen benötigt werden.

5.2.2 Abstrahierung von der Hardware-API

Den vollen Nutzen für einen Anwendungsentwickler kann die Hardware-API entfalten, wenn Betriebssystemabstraktionen, wie beispielsweise eine Systemuhr oder ein Standardausgabe-*Stream* verwendet werden, da diese Betriebssystemabstraktionen die Hardwaredetails vor dem Anwendungsentwickler vollständig verbergen können.
Im Falle der Systemuhr, welche die Betriebszeit eines Systems zählt, kann der zugehörige Gerätetreiber für die Zeitgeberkomponente zum Beispiel einfach automatisiert für die Betriebssysteminstanz konfiguriert werden, falls die Systemuhr vom Anwendungsentwickler verwendet wird. Dadurch kann im Zuge des LavA-Prozesses auch die passende Hardwarekomponente automatisiert generiert werden. Der Anwendungsentwickler muss also nicht explizit die Hardwarekomponente instanziieren oder konfigurieren, sondern es kann für die Betriebssysteminstanz das abstrakte Merkmal „Systemuhr“ selektiert werden und zum Beispiel über eine abstrakte `getTickCount`-Funktion mittels der diversen Softwareschichten indirekt auf die Zeitgeberkomponente zugegriffen werden.

```
1  aspect hw_dev_UART0_OutputDevice {
2      advice execution("% hw::hal::OutputDevice::clearScreen()") :
3      after() {
4          hw::dev::UART0::Inst().clearScreen();
5      }
6
7      advice execution("% hw::hal::OutputDevice::putChar(char)") :
8      after() {
9          hw::dev::UART0::Inst().putChar(*((char*) tjp->arg(0)));
10     }
11
12     ...
13 };
```

Abbildung 5.5: Aspekt zur Bindung eines UARTs an das `Output Device`

Wenn der Anwendungsentwickler andererseits die *Stream*-Abstraktion verwendet, ist es lediglich erforderlich, das gewünschte Ausgabegerät während der CiAO-Konfigurierung zu spezifizieren. Die Standardimplementierungen der Methoden des `Output Devices`, wie beispielsweise `putChar` oder `clearScreen`, sind leer. Damit nun der Gerätetreiber des gewünschten Ausgabegerätes an das `Output Device` gebunden werden kann (Pfeil von `UART0` zum `Output Device` in Abbildung 5.4), wird die Aspektorientierte Programmierung verwendet. Abbildung 5.5 zeigt wie ein Aspekt aussehen muss, damit der `UART0`-Gerätetreiber an das `Output Device` gebunden werden kann. Dazu werden innerhalb des automatisiert generierten Aspekts `hw_dev_UART0_OutputDevice` die `UART0`-Treiberfunktionen `clearScreen` und `putChar` an die korrespondierenden, leeren Funktionen des `Output Devices` gebunden. Dadurch werden nach einem Aufruf der Funktionen des `Output Devices` die gerätespezifischen Funktionen ausgeführt. Aufgrund der Charakteristik der Aspektorientierten Programmierung ist es darüber hinaus auch möglich, mehrere Ausgabegeräte an einen Standardausgabe-*Stream* zu binden, falls dies gewünscht ist.
Infolgedessen können Betriebssystemfunktionen der Systemschicht auf architekturunabhängige Abstraktionen, wie dem `Clock Device` oder dem `Output Device`, aufsetzen. Selbst wenn die Systemschicht komplexe Betriebssystemabstraktionen bietet, welche in dem Quelltext der Anwendung verwendet werden, besteht eine Verbindung durch alle involvierten Schichten, die letztendlich zu einer Instanz der Hardware-API führen.

5.2.3 Konfigurierung der CiAO Betriebssystem-Produktlinie

Die Konfigurierung der CiAO Betriebssystem-Produktlinie unterscheidet, wie typischerweise in der Produktlinienentwicklung üblich, zwischen dem Problemraum und dem Lösungsraum. Zur Beschreibung des Problemraums wird das Konzept der Merkmalmodelle verwendet. Über die verschiedenen Merkmale, welche die Variabilität der CiAO Betriebssystem-Produktlinie beschreiben, lassen sich die ab-

zuleitenden Betriebssysteme feingranular an die spezifischen Anforderungen anpassen. Zur Modellierung der Merkmalmodelle wird für die CiAO-Konfigurierung die bereits vorgestellte Konfigurationssprache Kconfig genutzt. Neben den konfigurierbaren Merkmalen werden auch die Abhängigkeiten zwischen einzelnen Merkmalen über diese Konfigurationssprache modelliert.
Das Merkmalmodell kann mit einem beliebigen Werkzeug zur Konfigurierung des Linux-Kernels, wie beispielsweise Menuconfig, geöffnet und eine Auswahl an Merkmalen selektiert werden. Die Konfiguration wird in einer Konfigurationsdatei abgelegt und beschreibt damit eine konkrete Instanz eines CiAO-Betriebssystems auf der Ebene der Merkmale.
Die Konfigurationsdatei mit den selektierten Merkmalen nutzen die Entwickler von CiAO als Basis, um mittels verschiedener Lösungsraumtechniken den Quelltext einer Betriebssysteminstanz zu generieren. Die Verknüpfung der diversen Merkmale mit den zugehörigen Implementierungen bzw. den entsprechenden Quelltextdateien ist über Perl-Skripte realisiert. Aufgabe dieser Skripte ist es, alle erforderlichen Quelltextdateien zu kopieren oder auch weitere Quelltextdateien anhand der Konfigurationsinformationen zu generieren. Da zahlreiche Betriebssystemfunktionen mit Hilfe der Aspektorientierten Programmierung umgesetzt sind, befinden sich unter den Dateien auch Aspekte, welche vor der Übersetzung der Betriebssysteminstanz mit einem Aspektweber eingewoben werden müssen.
Für jeden Knoten der LavA-Hardware wird eine dedizierte als auch maßgeschneiderte Instanz des CiAO-Betriebssystems generiert (siehe Abbildung 5.6). Die einzelnen Instanzen können dementsprechend für verschiedene Prozessorarchitekturen ausgelegt sein, im Umfang der Betriebssystemfunktionalitäten variieren oder auch, abhängig von den Peripheriekomponenten eines Knotens, verschiedene Gerätetreiber mitbringen. So wäre in dem Beispiel in Abbildung 5.6 für die erste Instanz des CiAO-Betriebssystems der entsprechende *Timer*-Gerätetreiber eingebunden, für die letzte CiAO-Instanz jedoch nicht. Die verschiedenen Tasks einer Anwendung werden in dem CiAO-Konfigurationsprozess mit statischen Prioritäten versehen sowie den einzelnen CiAO-Instanzen und damit den Hardwareknoten fest zugeordnet.

5.2.4 Fazit

Durch die Verknüpfung der Hardware-API und der CiAO Betriebssystem-Produktlinie können die Hardwarestrukturen automatisiert aus den Schichten der Systemsoftware abgeleitet werden. Gleichzeitig lassen die Betriebssystemabstraktionen die Hardwaredetails für einen Entwickler weiter in den Hintergrund rücken oder verbergen diese sogar vollständig, wenn zum Beispiel Abstraktionen wie die Systemuhr verwendet werden können.
Allerdings ist die Maßschneiderung der Betriebssysteminstanzen, insbesondere für Systeme mit vielen Knoten, an diesem Punkt noch mit viel Aufwand verbunden, da jede Instanz des CiAO-Betriebssystems separat vom Entwickler konfiguriert werden muss. Eine Lösung zu diesem Problem wird durch den automatisierten

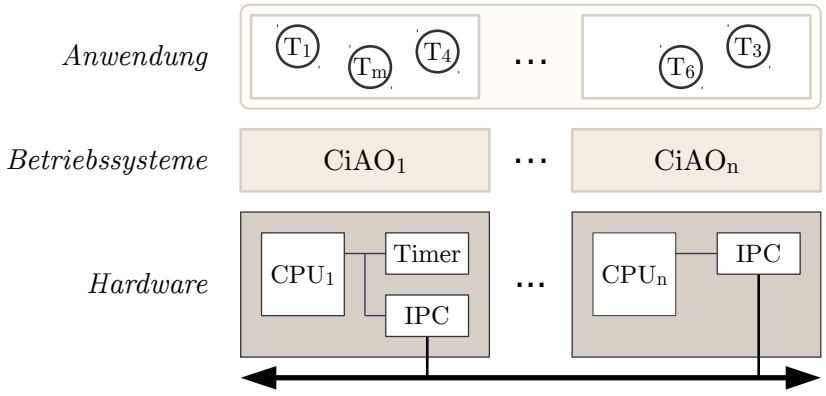


Abbildung 5.6: Instanziierung der Betriebssysteme für die LavA-Hardware

Optimierungsprozess in Kapitel 6 vorgestellt. Zudem bietet CiAO bisher keine passenden Kommunikationabstraktionen, um innerhalb eines *Multi-* bzw. *Many-core*-Systems bequem kommunizieren zu können. Eine Lösung zur Erweiterung der CiAO Betriebssystem-Produktlinie zu dieser Einschränkung wird nun in dem folgenden Unterkapitel diskutiert.

5.3 Paralleles Programmiermodell

Einer der wichtigsten Faktoren in einem *Multi-* bzw. *Manycore*-System ist die Kommunikation, denn ansonsten kann die gebotene Rechenleistung durch die zahlreichen Kerne nicht effektiv ausgenutzt werden. Neben den Kommunikationsstrukturen auf Hardwareebene, welche bei der Vernetzung von solchen Systemen keinen Flaschenhals darstellen dürfen, müssen einem Anwendungsentwickler auch entsprechende Abstraktionen zur Verfügung gestellt werden, um die Parallelität der Hardware auf Anwendungsebene ausnutzen zu können. Die Tasks sind in solchen Systemen über viele Knoten verteilt und müssen untereinander Daten austauschen können, um gemeinsam eine Lösung für ein gegebenes Problem zu berechnen. Die bisher vorgestellten Softwareschichten von LavA bieten keine hinreichenden Mechanismen, um einen Entwickler bei der Kommunikation in einem solchen System zu unterstützen. Die Treiber der diversen Kommunikationsstrukturen bieten zwar Sende- und Empfangsprimitiven über die kommuniziert werden kann, aber als Schnittstelle für den Anwendungsentwickler sind diese Primitiven für die in dieser Arbeit betrachteten Systeme nicht geeignet. Zum einen müsste der Entwickler die Kommunikation zwischen jedem Task-Paar aus der Vielzahl an Tasks explizit implementieren und zum anderen müsste die Kommunikation manuell angepasst werden, falls sich die Zuteilung einzelner Tasks zu Prozessoren, beispielsweise im Zuge einer Systemoptimierung, ändert.
Ähnliche Probleme gibt es im Bereich der Hochleistungsrechner und werden dort seit vielen Jahren untersucht und entsprechende Lösungen in Form von parallelen Programmiermodellen entwickelt. Diese Programmiermodelle bieten dem

Entwickler eine mächtige Schnittstelle mit zum Teil wesentlich komplexeren Mechanismen zur Kommunikation und abstrahieren zudem von der konkreten Zuordnung der Tasks auf die verteilten Rechner. Einer der prominentesten Vertreter dieser Programmiermodelle ist MPI [Mes12] (*Message Passing Interface*). MPI sieht die nachrichtenbasierte Kommunikation zwischen Prozessen über Rechnergrenzen hinweg vor. Dabei werden sowohl Mechanismen zur Punkt-zu-Punkt-Kommunikation als auch zur kollektiven Kommunikation innerhalb von Prozess-Gruppen berücksichtigt. Allerdings unterliegen Eingebettete Systeme anderen Beschränkungen als Hochleistungsrechner, welche in erster Linie durch Restriktionen bezüglich der vorhandenen Ressourcen geprägt werden. Aufgrund des Overheads und des großen Speicherbedarfs ist deshalb eine vollumfängliche Umsetzung des MPI-Standards für Eingebettete Systeme eher ungeeignet [PA08].

Die Verantwortung für die Konfigurierung der Betriebssysteminstanzen liegt zum jetzigen Zeitpunkt allein beim Entwickler und muss manuell von diesem durch die entsprechende Auswahl von konfigurierbaren Merkmalen der CiAO Betriebssystem-Produktlinie vorgenommen werden. Damit liegen auch schwer entscheidbare Entwurfsentscheidungen, wie die optimale Zuteilung der Tasks zu den Prozessoren oder die Wahl einer adäquaten Kommunikationsinfrastruktur, bisher in der Verantwortung des Entwicklers. Aufgrund der Komplexität der in dieser Arbeit anvisierten Systeme, ist die Bestimmung dieser Parameter für eine nahezu optimale Konfiguration aufwendig und setzt wiederum Expertenwissen voraus. Aber auch bei diesen schwierigen Entwurfsentscheidungen kann ein Entwickler mit einem entsprechenden Programmiermodell unterstützt werden, falls möglichst viele der Konfigurationsinformationen aus den Zugriffsmustern auf die Programmierschnittstelle gewonnen werden können und so ohne weitere manuelle Eingriffe für die Konfigurierung der Betriebssystemschicht verwendet werden können. In dieser Hinsicht ist das Programmiermodell von MPI nicht ideal, da viele Informationen erst zur Laufzeit zur Verfügung stehen und nicht über statische Analysen zur Übersetzungszeit ermittelt werden können. So erlaubt MPI zum Beispiel die dynamische Erzeugung von Prozessen oder die Erstellung von neuen Gruppen zur Laufzeit. Auch die Parameter der MPI-Kommunikationsoperationen beruhen nicht auf Konstanten und können dementsprechend nicht zur Übersetzungszeit ausgewertet werden.

Aufgrund der geschilderten Probleme wird in dieser Arbeit ein paralleles Programmiermodell vorgestellt, welches an MPI angelehnt ist und damit auch die Vorteile des SPMD (engl. *Single-Program Multiple-Data*) Programmierparadigmas für die Programmierung der hier angestrebten parallelen Systeme bietet, jedoch auf die speziellen Bedürfnisse von *Manycore*-Systemen im Kontext von Eingebetteten Systemen abgestimmt ist und – ähnlich wie die Hardware-API – die Möglichkeit zur statischen Analyse bietet, um auch auf dieser Ebene Konfigurationsinformationen anhand der Zugriffsmuster automatisiert extrahieren zu können.

5.3.1 Anforderungen an das LavA-Programmiermodell

Das parallele Programmiermodell soll die kommunikationsbezogenen Elemente der Anwendungsschnittstelle bereitstellen und somit einen wesentlichen Bestandteil der obersten Schicht des CiAO-Betriebssystems bilden. Im Folgenden werden die einzelnen Anforderungen an das parallele Programmiermodell im Detail ausgeführt.

Anlehnung an MPI Das angestrebte Programmiermodell soll an MPI angelehnt sein. Zum einen ist MPI der De-facto-Standard zur Programmierung von parallelen Systemen und damit vielen Entwicklern vertraut und zum anderen bietet MPI viele nützliche Mechanismen, wie die Einteilung von Prozessen in Gruppen und die damit einhergehenden kollektiven Kommunikationsmechanismen, die dem Entwickler die Implementierung von parallelen Anwendungen erheblich erleichtern.

Statische Analysierbarkeit Damit ein Anwendungsentwickler von schweren Entwurfsentscheidungen befreit werden kann, sollen möglichst viele Konfigurationsinformationen anhand der Zugriffsmuster auf die Programmierschnittstelle abgeleitet werden, um die Betriebssysteminstanzen analog zu der Hardwareschicht automatisiert generieren zu können. Gewinnbringende Informationen, welche aus den Zugriffsmustern abgeleitet werden sollen, sind zum Beispiel die in einer Anwendung auftretenden Kommunikationsbeziehungen zwischen Tasks, die Gruppierung der Tasks, die auszutauschenden Datenmengen oder auch die verwendeten Kommunikationsabstraktionen. Aus diesen Informationen können dann Rückschlüsse bezüglich der zu konfigurierenden Hardware- und Betriebssystem-Produktlinie gezogen werden.

Abstraktion von der Architektur Parameter, welche sich nicht direkt aus den Zugriffsmustern ablesen lassen, wie die konkrete Zuteilung von Tasks zu Prozessoren oder die optimale Kommunikationsinfrastruktur für eine gegebene Anwendung, sollen mittels Entwurfsraumexploration (siehe Kapitel 6) variiert, bewertet und so optimiert werden. Damit diese Parameter in einer automatisierten Optimierung variiert werden können, muss die von der Anwendung verwendete Programmierschnittstelle von der zugrunde liegenden Architektur abstrahieren und die beliebige Zuordnung von einzelnen Tasks zu Prozessoren erlauben, ohne dass manuelle Eingriffe durch den Entwickler an der Anwendung erforderlich sind.

Skalierung der Anwendung Auch der notwendige Grad an Parallelität einer Anwendung, um ein Problem innerhalb von gegebenen zeitlichen Schranken zu lösen, ist für einen Anwendungsentwickler nur schwer exakt zu bestimmen, weshalb die Skalierung der Anwendung ebenfalls von der Entwurfsraumexploration übernommen werden soll. Jedoch soll dem Entwickler über eine entsprechende Abstraktion des Programmiermodells die Möglichkeit

geboten werden, die Parallelität der Anwendung zumindest grob beeinflussen zu können.

Einschränkungen zur Laufzeit Eingebettete Systeme haben oftmals einen eher statischen Charakter, denn die zu berechnende Aufgabe ist in den meisten Fällen zum Zeitpunkt des Entwurfs bereits genau spezifiziert. Diese Charakteristik von vielen Eingebetteten Systemen soll auch für das Programmiermodell ausgenutzt werden, damit mehr Wissen über die implementierte Anwendung aus den Zugriffsmustern gewonnen werden kann. So sollen zum Beispiel keine Tasks dynamisch erzeugt und auch keine neuen Task-Gruppen zur Laufzeit angelegt werden können. Zudem sollen die Task-Prioritäten, ähnlich wie bei AUTOSAR, statisch zur Übersetzungszeit festgelegt werden.

Annotation der Ausführungszeiten Damit die Systeme in der Entwurfsraumexploration bewertet werden können, ist es erforderlich, dass die Ausführungszeiten der einzelnen Quelltextabschnitte bekannt sind. Die Bestimmung der WCET (*Worst-Case Execution Time*) ist jedoch ein eigener Forschungszweig und wird in dieser Arbeit nicht behandelt. Das Programmiermodell soll allerdings die Annotation der Zeiten ermöglichen, damit diese zur Bewertung der Systeme genutzt werden können.

5.3.2 Entwurf der Programmierschnittstelle

Wie schon für die Hardware-API, wird auch für das Programmiermodell der C++ *Template*-Mechanismus genutzt, um die Konfigurationsinformationen statisch analysieren zu können. Im Folgenden wird die Programmierschnittstelle sowie deren unterstützte Operationen vorgestellt und die entwickelten Konzepte in Bezug zu den Anforderungen gesetzt.

5.3.2.1 Nachrichtenobjekte

Zu versendende Nutzdaten werden in Nachrichtenobjekten gekapselt. Die Nachrichtenobjekte werden mittels des Klassen-*Templates* `Message` beschrieben. Bei der Instanziierung eines Nachrichtenobjektes muss der Datentyp der Nutzdaten und die Anzahl der Nutzdatenelemente als Konstante vom Anwendungsentwickler spezifiziert werden:

```
lava::Message<int, 16> msg;
```

In dem Beispiel wird ein Nachrichtenobjekt mit dem Datentyp `int` und mit `16` Nutzdatenelementen instanziiert. In dem Nachrichtenobjekt wird dann der benötigte Speicherplatz für den Nachrichten-*Header* und die Nutzdaten selbst reserviert. Über die Methode `getMessagePtr()` kann die Speicheradresse für die Nutzdaten abgefragt werden und zum Beispiel mit eingehenden Sensordaten befüllt werden.

Der Umweg über ein Nachrichtenobjekt ist nötig, um die Vorteile des NoCs der LavA Hardware-Produktlinien ausnutzen zu können, denn das NoC kann, unabhängig von dem jeweiligen Prozessor, über eine weitere Schnittstelle direkt auf den Datenspeicher der angeschlossenen Prozessoren zugreifen. Ein Sendevorgang kann deshalb durch die Angabe des Empfängers, den Speicherort der Nachricht und die Datenmenge gestartet werden. Dazu ist es jedoch notwendig, dass sowohl der Nachrichten-*Header* als auch die Nutzdaten nahtlos aufeinander folgend im Speicher liegen, wozu die Nachrichtenobjekte dienen. Der Aufwand für das lokale umkopieren der Nutzdaten in einen Puffer des NoCs entfällt somit. Falls die Hardware-Produktlinie in Zukunft um weitere Geräte mit einem direkten Zugriff auf den Datenspeicher erweitert wird, ist dieser Vorteil auch für diese Geräte nutzbar.

5.3.2.2 Punkt-zu-Punkt-Kommunikation

Die einfachste Methode, um Nachrichten zwischen zwei Tasks auszutauschen, sind die Sende- und Empfangsoperationen. Die Kommunikation über diese Operationen wird asynchron ausgeführt. Dementsprechend muss der Sender nicht auf die Empfangsbereitschaft des Empfängers warten und kann die Abarbeitung des weiteren Programms direkt nach dem Versand der Nachricht fortsetzen. Falls der Empfänger bereits empfangsbereit ist und der Sender noch nicht die Sendeoperation durchgeführt hat, blockiert der Empfänger bis zum Versand der Daten. Der Task, der die Sendeoperation aufruft, muss den Datentyp der Nachricht spezifizieren, die Anzahl der Nutzdatenelemente, die Task-ID des Empfängers angeben und dem Funktions-*Template* das zu übertragende Nachrichtenobjekt übergeben:

```
template <typename datatype, int count, tid_t dest>
    int send(Message<datatype, count> *msg);
```

Die Anzahl der Nutzdatenelemente und die Empfänger-ID werden als Nichttypparameter an das Funktions-*Template* übergeben. Durch die Angabe dieser Parameter mittels Konstanten ist beispielsweise der Empfänger der Nachricht zur Übersetzungszeit über statische Quelltextanalysen ermittelbar. Die Rückgabewerte der Funktions-*Templates* signalisieren eventuell auftretende Fehler für diese und die im Weiteren folgenden Kommunikationsoperationen.

Der vorgesehene Empfänger des Nachrichtenobjektes muss die korrespondierende Empfangsoperation `recv` aufrufen:

```
template <typename datatype, int count, tid_t source>
    int recv(Message<datatype, count> *msg);
```

Im Gegensatz zur Sendeoperation, wird bei der Empfangsoperation, die Task-ID des Senders spezifiziert und in dem übergebenden Nachrichtenobjekt werden die zu empfangenden Daten abgelegt.

5.3.2.3 Gruppen

Die LavA-Programmierschnittstelle unterstützt, ähnlich wie MPI, die Einteilung von Tasks[1] in Gruppen. Bei MPI dient eine Gruppe in erster Linie zur Strukturierung von Tasks und der Zuordnung von eindeutigen Task-Nummern (Rängen) innerhalb der jeweiligen Gruppen. Ein Task kann bei MPI zu mehreren Gruppen gehören und abhängig von der Gruppe kann auch der Rang eines Tasks in der jeweiligen Gruppe variieren.
Das LavA-Programmiermodell sieht für den Einsatz von Gruppen weitreichendere Aufgaben vor. Neben der Strukturierung von Tasks und der Zuordnung von Task-Nummern, wird über die Gruppen auch die Anzahl der Tasks für die zu implementierende Anwendung spezifiziert. Die Gruppen werden mittels des Klassen-*Templates* `TaskGroup` beschrieben und können von dem Anwendungsentwickler durch eine Typdefinition instanziiert werden:

```
typedef lava::TaskGroup<PRIORITY, TID_START, SIZE> grp;
```

Der Gruppe `grp` muss dabei eine Priorität (`PRIORITY`), die Identifikationsnummer von der aus die Durchnummerierung der Tasks in dieser Gruppe beginnen soll (`TID_START`) und die Gruppengröße (`SIZE`), also die Anzahl der gewünschten Tasks in dieser Gruppe, zugewiesen werden. Innerhalb einer Anwendung können auf diesem Wege beliebig viele Gruppen definiert werden. Die aufaddierte Anzahl der Tasks in allen Gruppen ergibt dann die Gesamtanzahl der Tasks für eine Anwendung. Im Gegensatz zu MPI wird bei diesem Ansatz sowohl die Einteilung der Tasks in Gruppen als auch die Anzahl der Tasks statisch festgelegt und kann zur Laufzeit nicht mehr geändert werden. Der wesentliche Faktor für diese Entwurfsentscheidung ist die Grundanforderung an das Programmiermodell und zwar über statische Analysen möglichst viele Konfigurationsinformationen ableiten zu können. Insbesondere die Anzahl der verwendeten Tasks ist ein elementarer Baustein, um Betriebssystem und Hardware anwendungsspezifisch konfigurieren zu können.

***Master*-Task** Die Tasks mit den niedrigsten Identifikationsnummern sind für eine Sonderrolle in ihren jeweiligen Gruppen für die später vorgestellten kollektiven Kommunikationsoperationen vorgesehen. Auf die Identifikationsnummern dieser Tasks kann der Anwendungsentwickler über den Aliasnamen der entsprechenden Gruppe und einen Aufzählungstypen des `TaskGroup` Klassen-*Templates* zugreifen. Für die Gruppe mit dem Alias `grp` wäre dies zum Beispiel: `grp::MASTER`.

Zugehörigkeit Analog zu dem *Master* einer Gruppe, kann auch die Zugehörigkeit eines Tasks zu einer Gruppe abgefragt werden. Falls sich ein Task in der angegebenen Gruppe befindet, wird das Konstrukt `grp::IN` zum Zeitpunkt der

[1] Im Folgenden wird nicht explizit zwischen dem Begriff Task aus dem Kontext der CiAO Betriebssystem-Produktlinie und dem entsprechenden Begriff Prozess aus dem MPI-Kontext unterschieden.

```
1 enum grp_scale {A = 2, B = 3};
2 enum grp_priority {PRIO1, PRIO2, PRIO3};
3
4 typedef lava::TaskGroup<PRIO1, 1, A> grp1;
5 typedef lava::TaskGroup<PRIO2, grp1::NEXT, A*2> grp2;
6 typedef lava::TaskGroup<PRIO3, grp2::NEXT, B> grp3;
```

Abbildung 5.7: Beispiel zur Implementierung von Gruppen

Übersetzung zu einer Eins aufgelöst und andernfalls zu einer Null. So kann der Entwickler dafür sorgen, dass ein gewünschter Quelltextblock zum Beispiel nur von Tasks einer bestimmten Gruppe ausgeführt wird.

Skalierung der Gruppen Ein komplexeres Beispiel für die Implementierung von Gruppen ist in Abbildung 5.7 dargestellt. In diesem Beispiel werden drei Gruppen mit den Aliasnamen `grp1`, `grp2` und `grp3` angelegt. Der Aufzählungstyp `grp_scale` wird zur Bestimmung der Gruppengrößen verwendet. Die symbolische Konstante `A` für die Gruppen in den Zeilen 4 bzw. 5 und die symbolische Konstante `B` für die in Zeile 6 spezifizierte Gruppe. Auf diese Weise können zwischen den Gruppen Abhängigkeiten ausgedrückt werden. In diesem Beispiel hat die Gruppe mit dem Aliasnamen `grp2` im Vergleich zu der Gruppe `grp1` beispielsweise die doppelte Anzahl an Tasks. Aufgrund der einfachen Modifizierung der Gruppengrößen über den Aufzählungstypen `grp_scale` kann dieser Mechanismus auch für die Entwurfsraumexploration genutzt werden, die in Kapitel 6 eingeführt wird, um die Gruppengröße automatisiert zu skalieren, womit auch der zugehörigen Anforderung nachgekommen wird. Dazu interpretiert die Entwurfsraumexploration die vom Entwickler spezifizierten Konstanten, als die maximal zulässigen Werte. Dementsprechend kann die Entwurfsraumexploration die Werte von Eins bis zum Maximum skalieren und die letztendlich ausgewählten Werte durch die Manipulation des Quelltextes einsetzen. Trotz dieser automatisierten Skalierung innerhalb der Entwurfsraumexploration bleiben die vom Entwickler gewünschten Abhängigkeiten zwischen den Gruppengrößen bestehen.

Damit die Identifikationsnummern der Tasks in der gesamten Anwendung eindeutig sind, kann für nachfolgend spezifizierte Gruppen die symbolische Konstante `NEXT` des `TaskGroup` Klassen-*Templates* verwendet werden. Diese Konstante gibt die nächste freie Identifikationsnummer an und kann dementsprechend als Startpunkt zur Zuweisung der Identifikationsnummern für die nächste Gruppe verwendet werden.

Die korrespondierende, visuelle Repräsentation zur Gruppierung der Tasks in die drei Gruppen aus Abbildung 5.7 ist in Abbildung 5.8 dargestellt. Die *Master*-Tasks mit den jeweils niedrigsten Identifikationsnummern in ihren Gruppen sind jeweils hervorgehoben.

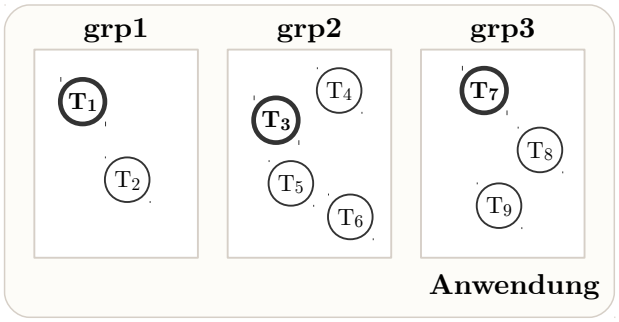


Abbildung 5.8: Resultierende Gruppierung der Tasks

5.3.2.4 Kollektive Kommunikation

Die kollektiven Kommunikationsoperationen vereinfachen die Implementierung der Kommunikation für den Anwendungsentwickler erheblich, denn diese Operationen implementieren komplexe Kommunikationsmuster, welche häufig innerhalb von Task-Gruppen auftreten können. Somit ist der Entwickler nicht gezwungen, diese Kommunikationsmuster aufwendig über einzelne Sende- und Empfangsoperationen umzusetzen. Für die kollektive Kommunikation zwischen Task-Gruppen müssen, im Gegensatz zu MPI, keine Kommunikatoren verwendet werden, sondern es kann die bereits vorgestellte Abstraktion der Gruppen genutzt werden.

Synchronisation Zur Synchronisation einer Task-Gruppe kann die `barrier`-Operation der Programmierschnittstelle verwendet werden:

```
template <typename group>
    int barrier();
```

Diese Operation erwartet ausschließlich die Spezifizierung der zu synchronisierenden Gruppe über den vorgesehenen *Template*-Parameter. Bei dieser Operation blockieren alle Tasks der angegebenen Gruppe, bis auch der letzte Task die Barriere erreicht hat. Im Anschluss können die Tasks die `barrier`-Operation synchronisiert verlassen und die Berechnung fortsetzen.

Broadcast Eine weitere Möglichkeit zur Kommunikation innerhalb einer Gruppe bietet die `broadcast`-Operation. Die ersten beiden Parameter bestimmen, wie bei der Punkt-zu-Punkt-Kommunikation, den Nachrichtentypen und die Menge der Nutzdatenelemente. Zusätzlich wird über den weiteren *Template*-Parameter `root` die Quelle für die zu übertragende *Broadcast*-Nachricht angegeben. Dazu kann typischerweise der *Master*-Task einer Gruppe verwendet werden. Bei der *Broadcast*-Operation muss zudem der Geltungsbereich über die Angabe der beteiligten Gruppe (`group1`) definiert werden. Die Verwendung des letzten *Template*-Parameters (`group2`) wird für diese und die folgenden kollektiven Kommunikationsoperationen in Unterkapitel 5.3.2.5 vorgestellt. Zuletzt muss, wie be-

reits im Kontext der Punkt-zu-Punkt-Kommunikation beschrieben, auch für die `broadcast`-Operation ein Nachrichtenobjekt übergeben werden:

```
template <typename datatype, int count, tid_t root,
    typename group1, typename group2 = NoneGrp>
    int bcast(Message<datatype, count> *msg);
```

Das Nachrichtenobjekt wird von dem spezifizierten `root`-Task an alle übrigen Gruppenmitglieder verteilt. Gegenüber der Punkt-zu-Punkt-Kommunikation rufen bei den kollektiven Kommunikationsoperationen alle Tasks der Gruppe das gleiche Funktions-*Template* auf. Die Rolle der Tasks wird dabei einzig über den `root`-Parameter bestimmt.

Scatter Die `scatter`-Operation verhält sich ähnlich zu der `broadcast`-Operation, jedoch erhalten bei dieser Operation nicht alle Mitglieder der angegebenen Gruppe die vollständige Nachricht, sondern jeweils gleich große Fragmente von der ursprünglichen Nachricht. Dazu muss der `scatter`-Operation ein weiteres Nachrichtenobjekt für die zu empfangenden Fragmente übergeben werden:

```
template <typename datatype, int count, tid_t root,
    typename group1, typename group2 = NoneGrp>
    int scatter(Message<datatype, count> *sendmsg,
    Message<datatype, count> *recvmsg);
```

Die Reihenfolge in der die Fragmente an die Tasks der Gruppe versendet werden, wird durch die Identifikationsnummern der Tasks bestimmt. Die Fragmente werden dabei vom `root`-Task in aufsteigender Reihenfolge weitergeleitet, wobei auch der `root`-Task eines der Fragmente lokal in sein eigenes `recvmsg` Nachrichtenobjekt kopiert.

Gather Das entsprechende Gegenstück zur `scatter`-Operation ist die Operation `gather`. Bei dieser Operation wird der `root`-Task zum Empfänger und speichert die eingehenden Nachrichtenfragmente der Gruppenmitglieder wiederum in aufsteigender Reihenfolge in dem spezifizierten `recvmsg` Nachrichtenobjekt ab:

```
template <typename datatype, int count, tid_t root,
    typename group1, typename group2 = NoneGrp>
    int gather(Message<datatype, count> *sendmsg,
    Message<datatype, count> *recvmsg);
```

5.3.2.5 Kommunikation zwischen Gruppen

Die Programmierschnittstelle ermöglicht über die bereits vorgestellten kollektiven Kommunikationsoperationen zudem die Kommunikation zwischen zwei Gruppen. So können beispielsweise Zwischenergebnisse, welche zuvor von einer anderen Gruppe berechnet wurden, an eine zweite Gruppe weitergeleitet werden. Dadurch ist es zum Beispiel möglich, eine Anwendung zur Datenstromverarbeitung

zu implementieren, bei der typischerweise die Berechnung in diverse Stufen gegliedert ist und die Ergebnisse von Stufe zu Stufe weitergereicht werden. Die kollektiven Kommunikationsoperationen *Broadcast*, *Scatter* und *Gather* bieten dazu den *Template*-Parameter `group2` für die Spezifizierung der Empfängergruppe. Dieser Parameter besitzt einen *Default*-Wert und muss dementsprechend bei der Kommunikation innerhalb einer einzigen Gruppe nicht angegeben werden (siehe Unterkapitel 5.3.2.4). Mittels *Template*-Spezialisierung wird dann die passende Variante für die Kommunikation innerhalb einer Gruppe oder zwischen zwei Gruppen aufgerufen. Der *Default*-Wert `NoneGrp` ist eine leere Hilfsklasse und wird nur zur Durchführung der *Template*-Spezialisierung verwendet.
Bei der Kommunikation zwischen zwei Gruppen ist zu beachten, dass für die hier vorgestellten Kommunikationsmuster der `root`-Task aus der ersten Gruppe stammen muss und nur dieser an der Kommunikation mit der zweiten Gruppe teilnimmt.

5.3.2.6 Annotation der Ausführungszeiten

Um die Anforderung zur Annotation der Ausführungszeiten zu erfüllen, bietet die Programmierschnittstelle das Klassen-*Template* `wcet`. Dieses *Template* bietet zwei *Template*-Parameter zur Angabe der WCET und der BCET (*Best-Case Execution Time*):

```
lava::wcet<89, 81> wcet_init;
```

Die BCET ist dabei optional und muss vom Entwickler nicht spezifiziert werden. Die zu verwendende Zeiteinheit ist nicht festgelegt. Allerdings muss die gewählte Zeiteinheit in der gesamten Anwendung konsistent genutzt werden und der Entwurfsraumexploration bekanntgemacht werden.
Die manuelle Spezifizierung der Ausführungszeiten über die Annotationen ist derzeit notwendig, da für die hier vorgestellten, konfigurierbaren Prozessoren kein Werkzeug zur automatisierten Analyse der Ausführungszeiten zur Verfügung steht und ein solches Werkzeug zudem nicht im Fokus dieser Arbeit steht. Falls solch ein Werkzeug zukünftig verfügbar sein sollte, wäre dieser Teil des Programmiermodells obsolet und der Entwickler könnte weiter entlastet werden.

5.3.3 Fazit

Das vorgestellte Programmiermodell abstrahiert nicht nur von der Zuteilung der Tasks zu den Prozessoren, sondern auch von der konkreten Kommunikationsinfrastruktur. Der Entwickler kann so unabhängig von der zugrunde liegenden Architektur die Anwendung implementieren. Somit verschafft das Programmiermodell der Entwurfsraumexploration den nötigen Freiheitsgrad zur Skalierung der Anwendung, zur Bestimmung der Architektur und zur Zuordnung der Tasks. Zudem trägt die Schnittstelle aufgrund der *Template*-Parameter eine Vielzahl an Konfigurationsinformationen in sich, welche für die Maßschneiderung der tieferen Schichten genutzt werden können. Diese Konfigurationsinformationen sollen

automatisiert aus der Schnittstelle extrahiert (Unterkapitel 6.2) und in ein Anwendungsmodell überführt (Unterkapitel 6.2.2) werden. Neben den Kommunikationsbeziehungen und der Kommunikationsdichte, welche zur Bestimmung der idealen Kommunikationsinfrastruktur verwendet werden können, lassen sich mit den zusätzlich annotierten Ausführungszeiten die Systeme bewerten und so in der Entwurfsraumexploration optimieren. Auch die Verwendung von bestimmten Kommunikationsoperationen in der Anwendung, wie beispielsweise der *Broadcast*-Operation, kann Hinweise für die zu realisierende Architektur geben, denn der Bus der LavA Hardware-Produktlinie unterstützt zum Beispiel die Umsetzung eines *Broadcasts* direkt in Hardware. Wenn also ein entsprechender *Broadcast* in der Anwendung eingesetzt wird, kann dies anhand einer statischen Analyse erkannt werden und die Hardwareunterstützung für einen *Broadcast* kann im Laufe des Konfigurationsprozesses automatisiert integriert werden.

5.4 Zusammenfassung

Die in diesem Kapitel vorgestellten Schichten der Betriebssystem-Produktlinie bilden die Grundlage, um die Hardware, deren Implementierungsdetails und selbst abstrakte Modelle der Hardware, auf Systemebene vor einem Anwendungsentwickler verbergen zu können. Die unterste Abstraktionsschicht wird von der Hardware-API gestellt, welche die Abbildung von Hardwarekomponenten auf der Softwareebene erlaubt und einem Anwendungsentwickler so eine vertraute Schnittstelle zur Maßschneiderung der Hardware in Software bietet. Dennoch muss sich der Entwickler auf dieser Ebene noch mit Hardwaredetails, wie der direkten Auswahl der Hardwarekomponenten oder mit deren Parametrisierung, auseinandersetzen.
Abhilfe schaffen hier die höheren Schichten der CiAO Betriebssystem-Produktlinie, welche die Hardwarekomponenten hinter Betriebssystem-Abstraktionen verbergen können. Die Instanziierung der Hardwarekomponenten kann dann indirekt über die entsprechende Konfigurierung einer Betriebssystem-Abstraktion erfolgen, wodurch die eigentliche Instanziierung der Hardwarekomponente in den tieferen Schichten des Betriebssystems vor dem Entwickler verborgen wird.
Zudem erweitert die parallele Programmierschnittstelle die oberste Schicht der CiAO Betriebssystem-Produktlinie und schafft so eine Abstraktion, welche nicht nur die Programmierung für einen Entwickler von solch parallelen Systemen erleichtert, sondern gleichzeitig einer Entwurfsraumexploration die erforderliche Freiheit bietet, die Betriebssysteminstanzen und die Hardware anwendungsspezifisch maßschneidern und optimieren zu können. Dadurch rückt die oberste Betriebssystemschnittstelle in das Zentrum der Anwendungsentwicklung.

Kapitel 6

Automatisierte Optimierung

Inhalt

Produktlinien erlauben zwar in der Regel die feingranulare Anpassung von Produkten, aber die Vielfalt an zur Verfügung stehenden Konfigurationsmerkmalen, anhand derer sich die Produkte an spezifische Anforderungen anpassen lassen, bieten oftmals eine nahezu grenzenlose Vielfalt an möglichen Lösungen. Insbesondere, wenn die Produkte bezüglich diverser nicht-funktionaler Eigenschaften optimiert werden sollen, sind die Auswirkungen bei der Wahl eines bestimmten Merkmals auf die nicht-funktionalen Eigenschaften nur schwer abzuschätzen. Dies gilt erst recht für komplexe, mehrschichtige Hardware-/Softwaresysteme. In dieser Arbeit wird deshalb, wie üblich in dieser Domäne [PEP06, KKO+06, TBHH07], der Weg über eine automatisierte Optimierung der Systeme eingeschlagen. Im Gegensatz zu anderen Arbeiten, soll allerdings die Softwareschicht bzw. das Betriebssystem stärker in diesen Prozess einbezogen werden. Zudem wird die Schnittstelle, über die der Anwendungsentwickler mit diesem Prozess interagieren soll, durch

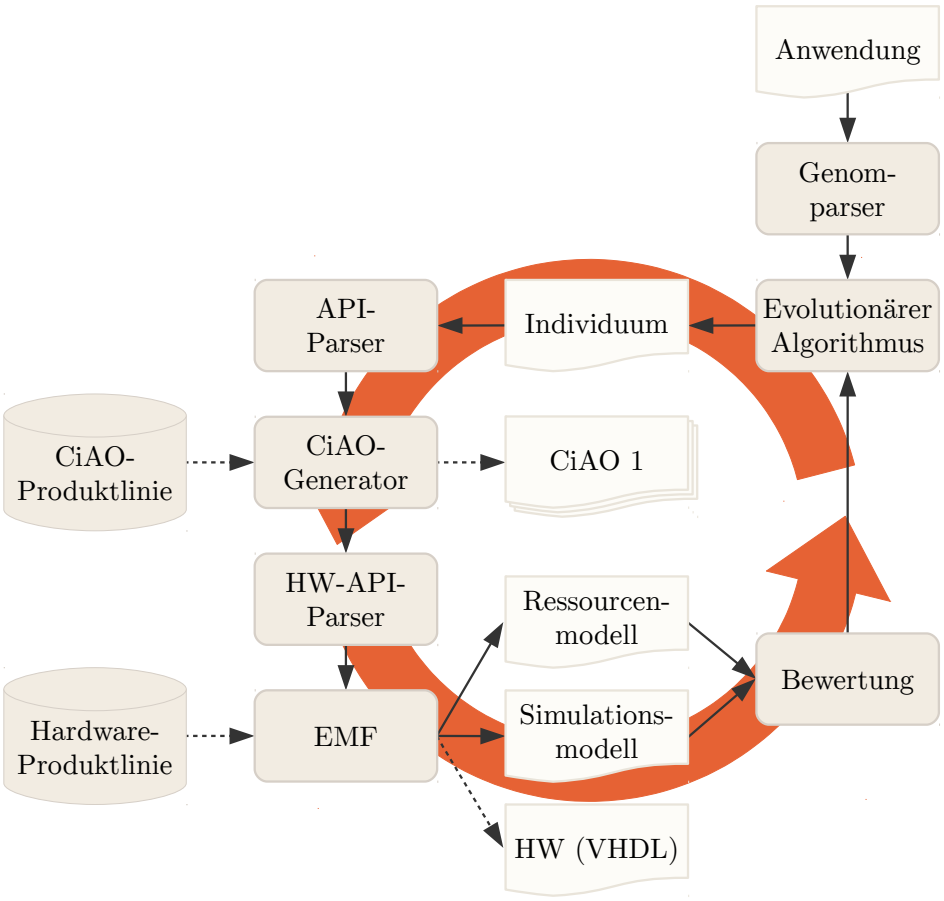


Abbildung 6.1: Ablaufdiagramm zum LavA-Optimierungsprozess

das vorgestellte parallele Programmiermodell bestimmt und setzt dementsprechend keine unzugänglichen Repräsentationen der Hardware oder des Betriebssystems für einen Anwendungsentwickler voraus. Die in den Kapiteln 4 und 5 vorgestellten Hardware- und Betriebssystem-Produktlinien bilden die Grundlage für diesen Prozess und bieten der Optimierung die nötige Flexibilität, um mit den ableitbaren Systemen einen möglichst vielfältigen Entwurfsraum zu untersuchen.

Der grobe Ablauf des LavA-Optimierungsprozesses ist in Abbildung 6.1 dargestellt. Ziel dieses Kapitels ist es, diesen Prozess im Detail vorzustellen und insbesondere die Vorteile, welche das Konzept der automatisierten Analyse der Programmierschnittstellen in Verbindung mit den maßschneidbaren Produktlinien bietet, zu vermitteln. Den Startpunkt für diesen Prozess stellt die Anwendung des Entwicklers dar. Diese Anwendung muss, wie in dem vorangegangenen Kapitel beschrieben, gegen die parallele Programmierschnittstelle implementiert werden. Nach der Fertigstellung der Anwendung sind alle weiteren Schritte des Prozesses vollständig automatisiert und setzen keine weiteren Eingriffe durch den Entwickler voraus. Die Schaltzentrale dieses Prozesses ist ein evolutionärer Algorithmus

(EA), welcher die anwendungsspezifische Optimierung eines LavA-Systems zum Ziel hat. Da die Problemrepräsentation für den EA von der Anwendung abhängt, wird in einem ersten Schritt (Genomparser) die Anwendung bezüglich der benötigten Problemrepräsentation untersucht und diese an den EA weitergegeben.
Die eigentliche Optimierung der Systeme findet in einem Kreislauf statt. Dieser Kreislauf wird erst mit dem Erreichen einer gegebenen Abbruchbedingung verlassen, zum Beispiel, wenn eine zuvor festgelegte Anzahl von Generationen bzw. von Individuen erreicht wurde. Die Eigenschaften der Individuen werden von dem EA bestimmt und jeweils in einem Genom kodiert. Die einzelnen Individuen beinhalten also Konfigurationsmerkmale, welche zum Beispiel die Zuordnung von Tasks zu Prozessoren oder die zu nutzende Kommunikationsinfrastruktur festlegen, und so wichtige Aspekte zur Umsetzung einer konkreten Lösung für ein Hardware-/Softwaresystem spezifizieren.
Jedes erzeugte Individuum durchläuft anschließend die in den Kapiteln 4 und 5 vorgestellten Prozesse zur Maßschneiderung der Betriebssysteminstanzen und der Hardwarestrukturen. In einem ersten Schritt werden dazu die von der Anwendung verwendeten Zugriffe auf die parallele Programmierschnittstelle untersucht. Die so gewonnenen Konfigurationsinformationen werden dann zusammen mit den Informationen aus dem Genom des Individuums zur Generierung der erforderlichen CiAO-Instanzen eingesetzt. Im Weiteren werden dann alle Vorkommen von HW-API Instanzen in den Softwareschichten analysiert und anhand dieser Informationen ein Modell der Hardwarestrukturen generiert. Dieses Modell wird entsprechend der Konstruktionsregeln des Metamodells (siehe Unterkapitel 4.3.1) konstruiert und stellt somit eine konkrete Instanz einer Hardwarestruktur auf abstrakter Ebene dar. Aus den erzeugten Modellen kann dann mittels des EMF (*Eclipse Modeling Framework*) sowohl der VHDL-Quelltext als auch die entsprechenden Ressourcen- und Simulationsmodelle zur Bewertung der Individuen generiert werden.
Nach der Bewertung der Individuen bezüglich diverser nicht-funktionaler Eigenschaften, werden die Ergebnisse an den EA weitergeleitet und der nächste Zyklus mit einem weiteren Individuum wird angestoßen.
In den folgenden Unterkapiteln werden die einzelnen Schritte dieses Prozesses im Detail vorgestellt und erläutert. In Unterkapitel 6.1 wird zunächst die Optimierung von LavA-Systemen über einen EA diskutiert. Im Anschluss folgt in den Unterkapiteln 6.2 und 6.3 die automatisierte Analyse der Anwendung sowie der Hardware-API. Zum Ende des Kapitels werden in Unterkapitel 6.4 die untersuchten Methoden zur Bewertung von LavA-Systemen bezüglich Ressourcen und zeitlichen Kriterien vorgestellt.

6.1 Optimierung der Hardware-/Softwaresysteme

Zur Optimierung der Hardware-/Softwaresysteme wird in dieser Arbeit der multikriterielle, evolutionäre Algorithmus SPEA2 (*The Strength Pareto Evolutionary Algorithm 2*) [ZLT01] eingesetzt. Evolutionäre Algorithmen bilden im Allgemei-

nen die natürlichen Abläufe der Evolution nach. Die grundlegende Funktionsweise von evolutionären Algorithmen lässt sich wie folgt beschreiben: Ausgehend von einer Startpopulation, werden zunächst die Individuen dieser Population bewertet. Im Anschluss werden von dem Algorithmus einige Individuen, zum Beispiel Individuen mit vielversprechenden Eigenschaften, zur Rekombination selektiert. Zusätzlich wird die Vielfalt im Genpool durch Mutationen erhöht. Eine Mutation kann dabei sowohl einen positiven als auch einen negativen Einfluss auf die Güte (Fitness) eines Individuums haben. Die neu entstandenen Individuen werden anschließend bewertet und der Algorithmus bestimmt, unter Berücksichtigung der Diversität und der Güte, die nächste Generation von Individuen, womit der nächste Durchlauf bis zum Erreichen der Abbruchbedingung beginnen kann.
Der SPEA2-Algorithmus übernimmt dabei primär die Rolle der Selektion der Individuen. Die Selektion ist typischerweise unabhängig von der gegebenen Problemstellung und lässt sich somit für unterschiedliche Probleme einsetzen. Die Repräsentation und die Variation der Individuen, zum Beispiel über Rekombinations- oder Mutationsoperatoren, hängen hingegen stark von der jeweiligen Problemstellung ab. Zur Anbindung der problemspezifischen Implementierung an SPEA2 wird in dieser Arbeit die PISA-Schnittstelle [BLTZ03] eingesetzt, welche eine Bibliothek zur Kommunikation mit verschiedenen Selektoren, wie SPEA2 oder NSGA-II, über Textdateien bietet. In den nachfolgenden Unterkapiteln werden die problemspezifischen Details des evolutionären Algorithmus für LavA-Systeme diskutiert.

6.1.1 Kodierung der Individuen

Damit die Individuen von dem EA rekombiniert und mutiert werden können, müssen deren unterschiedliche Ausprägungen entsprechend kodiert werden. Ursprünglich werden im Kontext von EAs binäre Kodierungen zur Repräsentation der Ausprägungen eingesetzt. Diese bringen allerdings den Nachteil mit, dass bereits minimale Änderungen an der binären Repräsentation einen erheblichen Einfluss auf das Individuum haben können, etwa wenn ein höherwertiges Bit eines bestimmten Merkmals durch den Mutationsoperator invertiert wird. Zur Vermeidung dieser Problematik kann die Repräsentation der Individuen beispielsweise in Gray-Kodierung erfolgen oder, wie im Falle dieser Arbeit, mittels Dezimalzahlen umgesetzt werden.
Eine allgemeine Darstellung der Genomstruktur für LavA-Systeme ist in Abbildung 6.2 visualisiert. Das dort gezeigte Genom ist in vier Bereiche unterteilt, welche unterschiedliche, variable Facetten der Systeme, wie die Skalierung, die Zuordnung von Task-Gruppen zu Knoten sowie die gruppeninternen und -externen Kommunikationsstrukturen bestimmen. Die in einem Genom spezifizierten Eigenschaften eines LavA-Systems sollen dem Entwickler schwierige Entwurfsentscheidungen abnehmen, welche sich nicht direkt aus der Programmierschnittstelle ableiten lassen und so über den EA optimiert werden können. Die Anzahl der Gene bzw. Parameter ist für jeden Bereich variabel und hängt von der gegebenen

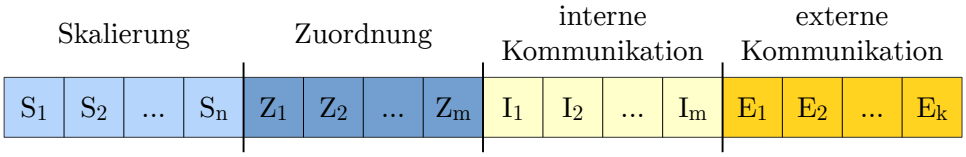


Abbildung 6.2: Struktur des Genoms

Anwendung ab. Die Bestimmung der Bereichsgrößen wird durch den Genomparser vorgenommen und in Unterkapitel 6.1.2 erläutert.
Die Auswirkungen der Gene aus den vier verschiedenen Bereichen des Genoms auf ein LavA-System sind in Abbildung 6.3 dargestellt. Die Gene des ersten Bereichs legen indirekt die Skalierung der Gruppengröße fest. Die Bestimmung der Gruppengröße erfolgt hier indirekt, da der in Unterkapitel 5.3.2.3 vorgestellte Mechanismus zur Skalierung der Gruppen angewendet wird und dementsprechend die Werte der symbolischen Konstanten des Aufzählungstyps `grp_scale` in den Genen kodiert werden und nicht unmittelbar die Anzahl der Tasks pro Gruppe. Ungeachtet dessen können, abhängig von der Implementierung der Gruppen, diese Werte auch direkt, wie in dem Beispiel aufgezeigt, auf die Gruppengrößen projiziert werden.
Der nächste Abschnitt spezifiziert die Zuordnung der Task-Gruppen zu Knoten. Die Anzahl der Gene in diesem Abschnitt werden direkt durch die vom Entwickler angelegten Gruppen bestimmt. Da die Tasks einer Gruppe dazu genutzt werden, um parallel eine Lösung für ein Problem zu berechnen, werden die Tasks einer Gruppe nicht zusammen auf einem Knoten platziert. In dem Genom wird deshalb bestimmt, zu welchem Knoten der erste Task der Gruppe zugeordnet wird. Alle weiteren Tasks werden nacheinander auf die nachfolgenden Knoten des Systems verteilt, wodurch deren Zuordnung nicht explizit im Genom kodiert werden muss. So wird die gemeinsame Belegung eines Knotens durch Tasks derselben Gruppe direkt durch die Kodierung verhindert. Die beiden Tasks der ersten Gruppe werden ab dem dritten Knoten platziert und die vier Tasks der zweiten Gruppe auf den Knoten 1 bis 4. Die Anzahl der Knoten in einem System wird direkt aus der gegebenen Zuordnung und dem damit einhergehenden Bedarf an Knoten abgeleitet. Bei diesem Ansatz werden dementsprechend die Hardwarestrukturen an die Anforderungen der Softwareschichten angepasst.
Auch der nächste Bereich wird durch die Anzahl der Gruppen geprägt. Dieser legt nämlich die zu verwendende Kommunikationsstruktur zur Kommunikation innerhalb einer Gruppe fest. Die erste Gruppe soll in diesem Beispiel über die Struktur `2` kommunizieren und die zweite Gruppe über die Struktur `0`. Jede Gruppe erhält in diesem Fall also jeweils ihre eigene Infrastruktur zur internen Kommunikation. Falls mehrere Gruppen den gleichen Wert zugeordnet bekommen, wird diese Kommunikationsstruktur nur einmal instanziiert und zwischen den Gruppen geteilt. Neben der konkreten Instanz einer Kommunikationsstruktur, spezifizieren die Werte der Gene auch den Typ der Kommunikationsstruktur. Innerhalb der automatisierten Optimierung der Systeme stehen dem EA drei verschiedene Ty-

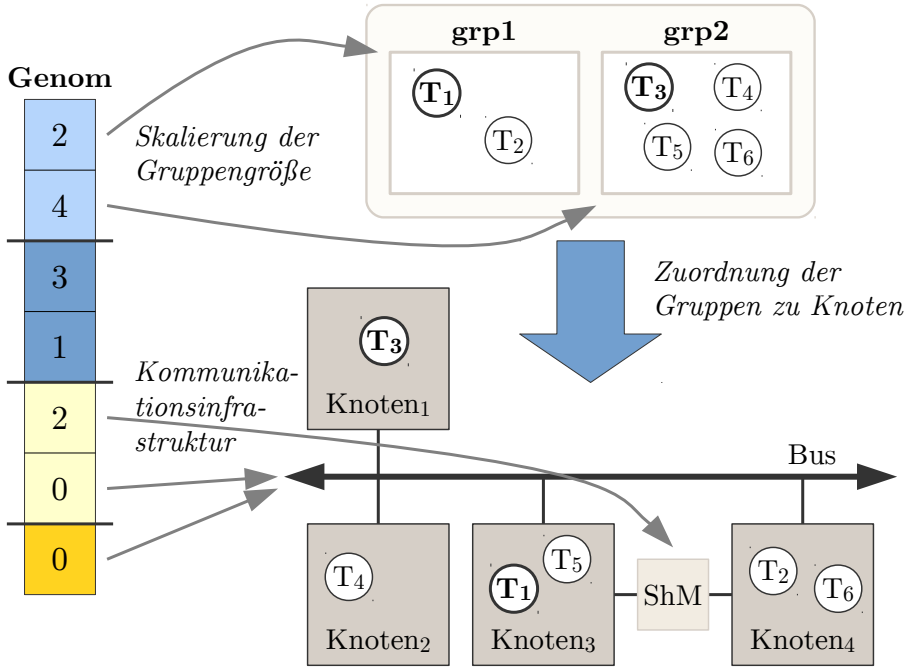


Abbildung 6.3: Übertragung der Genominformationen auf ein LavA-System

pen von Kommunikationsstrukturen zur Verfügung: Busse, NoCs (*Networks on Chip*) und gemeinsame Speicher (ShM). Der in einem Gen kodierte Typ lässt sich anhand einer Restwertdivision durch 3 (Modulo) ermitteln. Bei einem Restwert von 0 handelt es sich um einen Bus, bei einem Restwert von 1 um ein NoC und bei einem Restwert von 2 wird ein Netz von gemeinsamen Speichern zwischen den kommunizierenden Tasks bzw. deren Knoten aufgespannt.
Der letzte Abschnitt des Genoms spezifiziert die zu instanziierende Kommunikationsstruktur zur Kommunikation zwischen zwei Gruppen. Die Kodierung der Kommunikation entspricht dabei der zuvor erläuterten Repräsentation zur gruppeninternen Kommunikation. In Abbildung 6.3 wird zur gruppenübergreifenden Kommunikation wiederum der bereits instanziierte Bus verwendet.

6.1.2 Genomparser

Die Anzahl der Gene in den vier Bereichen des Genoms wird über eine Analyse der Anwendung bestimmt. Dazu untersucht der Genomparser in erster Linie die von dem Entwickler festgelegten Gruppen. Die so gewonnenen Informationen werden dem EA vor dem Start der Optimierung zur Verfügung gestellt.
Eine mögliche Spezifizierung der Gruppenlandschaft durch den Entwickler, wie sie der Genomparser vorfinden könnte, ist in Abbildung 6.4 abgebildet. Die Anzahl der Gene zur Skalierung der Gruppen korrespondiert dabei mit der Anzahl der symbolischen Konstanten in dem Aufzählungstyp `grp_scale`. Neben der Anzahl der symbolischen Konstanten, werden dem EA auch die vom Entwickler definier-

```
1 enum grp_scale {A = 2, B = 4};
2 enum grp_priority {PRIO1, PRIO2};
3
4 typedef lava::TaskGroup<PRIO1, 1, A> grp1;
5 typedef lava::TaskGroup<PRIO2, grp1::NEXT, B> grp2;
```

Abbildung 6.4: Analyse der Gruppenspezifizierung

ten Werte der jeweiligen symbolischen Konstanten mitgeteilt. Diese Werte werden im Laufe der späteren Optimierung als Maximalwert verstanden und von dem EA nicht überschritten. Die symbolische Konstante `B` könnte dementsprechend während des Optimierungsprozesses ganzzahlige Werte in dem Intervall von 1 bis 4 annehmen. So kann der Entwickler – bis zu einem gewissen Grad – Einfluss auf die Skalierung nehmen und den Entwurfsraum so erheblich verkleinern.

Die Größen der nächsten beiden Bereiche werden allein durch die Anzahl der definierten Gruppen (siehe Zeilen 4 und 5) bestimmt. Für jede Gruppe wird in dem jeweiligen Bereich demzufolge ein eigenes Gen zur Bestimmung der Platzierung der Gruppe bzw. zur Zuweisung einer Kommunikationsstruktur zu der Gruppe genutzt.

Für den letzten Bereich wird die Anwendung nach dem Vorkommen von Kommunikationsoperationen durchsucht, welche die Kommunikation zwischen zwei Gruppen durchführen. Für jedes Gruppenpaar wird in diesem Bereich dann ein eigenes Gen zur Variation der Kommunikationsstruktur eingesetzt.

Die Bestimmung des Genoms wird einmalig zu Beginn des Optimierungsprozesses angestoßen. Während der Optimierung verändert sich die Form des Genoms nicht und auch der Einfluss eines bestimmten Gens bleibt unverändert einer Eigenschaft eines LavA-Systems zugeordnet.

6.1.3 Rekombinationsoperator

Die Rekombination von Individuen ist eine der Möglichkeiten, um die Population im Laufe des Optimierungsprozesses zu verbessern und neue Individuen zu generieren. Bei der Rekombination selektiert der EA typischerweise zwei oder mehr Elternindividuen und rekombiniert deren Erbinformationen zu einem neuen Genom für einen Nachkommen.

Die in diesem Projekt verwendete Rekombination von zwei Elternindividuen basiert auf der Ein-Punkt-Kreuzung derer Genome. Dazu wählt der EA zufällig einen Schnittpunkt zwischen zwei Genen und kombiniert das neue Individuum aus dem ersten Abschnitt des ersten Elternteils und dem zweiten Abschnitt des zweiten Elternteils. In dem Beispiel in Abbildung 6.5 wird dieser Schnittpunkt zwischen dem fünften und sechsten Gen der jeweiligen Genome angesetzt und die beiden Teile der Eltern zu einem neuen Individuum zusammengesetzt.

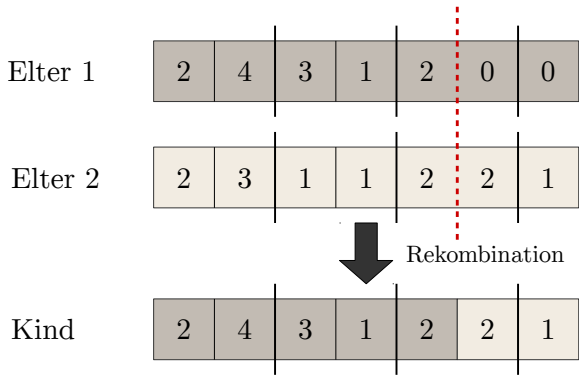


Abbildung 6.5: Ein-Punkt-Kreuzung von LavA-Individuen zur Rekombination

Durch diese Herangehensweise können mögliche positive Eigenschaften der beiden Elternindividuen in einem neuen Individuum vereint werden und so möglicherweise zu einem verbesserten System führen.

6.1.4 Mutationsoperator

Eine weitere Möglichkeit um die Eigenschaften von Individuen potenziell zu verbessern, ist die Mutation. Die Mutation verändert ein oder mehrere Gene eines Individuums. Ob ein Individuum mutiert werden soll und welche Gene betroffen sind, wird über den Zufall bestimmt. Die Wahrscheinlichkeit für eine Mutation kann dem EA zu Beginn des Optimierungsprozesses übergeben werden. Die Mutationen haben nicht zwingend einen positiven Einfluss auf die Güte eines Individuums und können diese auch negativ beeinflussen oder sich neutral zu der Güte des Individuums verhalten.

In Abbildung 6.6 wird ein Beispiel für eine Mutation eines LavA-Individuums aufgezeigt. Bisher wird in der hier entwickelten problemspezifischen Implementierung des EAs für LavA-Systeme jeweils genau ein Gen ausgewählt und mutiert. In dem Beispiel ist das dritte Gen des Genoms zur Mutation ausgewählt. Abhängig von dem Bereich des Genoms, in dem sich das selektierte Gen befindet, ob dieses also beispielsweise die Skalierung des Systems oder die Zuordnung von Tasks zu Knoten bestimmt, wird ein unterschiedlicher Mutationsoperator eingesetzt, um den neuen Wert dieses Gens zufällig auszuwählen. Dieser Schritt ist notwendig, damit die Auswirkungen der Mutation auf das Individuum in einem kontrollierten Rahmen ablaufen. Ansonsten könnte zum Beispiel die Mutation eines für die Skalierung verantwortlichen Gens dazu führen, dass ein gutes Individuum mit eher wenigen Tasks, zu einem Individuum mit sehr vielen Tasks und möglicherweise einer sehr schlechten Güte wird. In dem Beispiel in Abbildung 6.6 wird durch die angedeutete Mutation, nun eine der Gruppen ab dem fünften Knoten platziert und nicht mehr ab dem dritten Knoten.

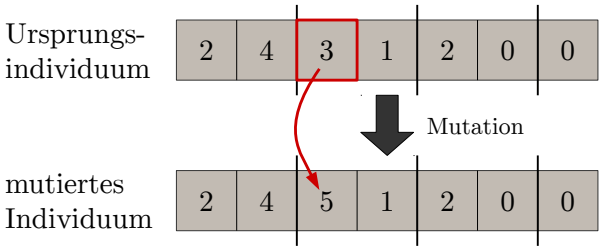


Abbildung 6.6: Mutation eines LavA-Individuums

6.1.5 Generierung der Startpopulation

Die Genome von Individuen der Startpopulation werden von dem EA überwiegend zufällig bestimmt. Eine Ausnahme bilden dabei speziell ausgewählte Gene von zehn Individuen der Startpopulation, welche nach speziellen Vorschriften bestimmt werden, um wichtige Konstellationen und Eigenschaften in der Startpopulation garantieren zu können. Dabei werden besonders solche Eigenschaften in den Genpool eingestreut, welche über die zufällige Bestimmung der Gene mit einer relativ geringen Wahrscheinlichkeit auftreten würden, jedoch trotzdem eine erhebliche Relevanz für die anvisierte Kategorie von Systemen haben können. Insbesondere betrifft dies die Platzierung der Tasks und die Kommunikationsinfrastruktur.
Bei der Platzierung der Tasks wird auf der einen Seite darauf geachtet, dass ein Teil dieser Individuen einen eigenen Knoten für jeden einzelnen Task bereitstellen und auf der anderen Seite werden auch Individuen generiert, welche bei der Verteilung der Tasks mit der minimalen Anzahl an Knoten auskommen. Für den zweiten Fall korrespondiert die Anzahl der Knoten dementsprechend mit der Anzahl der Tasks in der größten Gruppe.
Zudem wird hinsichtlich der Kommunikationsinfrastruktur darauf geachtet, dass die drei zur Verfügung stehenden Typen von Kommunikationsstrukturen jeweils bei zwei Individuen für die vollumfängliche Kommunikation zwischen allen Gruppen eingesetzt werden, in den Systemen folglich genau eine Kommunikationsstruktur angeboten wird.

6.1.6 Reparaturmechanismen

Die zufällig generierten Individuen können Geninformationen beinhalten, welche dazu führen, dass die Individuen sich entweder konstruktionsbedingt nicht generieren lassen oder brachliegende Ressourcen umfassen. Diese Defekte können sowohl bei Individuen aus der Startpopulation als auch im späteren Verlauf der Optimierung durch die Rekombination bzw. Mutation von Individuen auftreten. Um diesen Problemen entgegenzuwirken, werden verschiedene Reparaturmechanismen eingesetzt, damit diese Defekte im Genom der Individuen korrigiert werden können.

Nicht generierbare Individuen können beispielsweise entstehen, wenn an einem Knoten mehr als ein NoC angeschlossen werden soll. Da das NoC eine direkte Schnittstelle zum Zugriff auf den Datenspeicher des Knotens benötigt und die als Datenspeicher genutzten BRAMs nur zwei Schnittstellen zur Verfügung stellen, sind die Schnittstellen durch den Prozessor selbst und ein angeschlossenes NoC bereits belegt. Die Korrektur dieses Defekts wird durch die zufällige Auswahl eines der NoCs und der entsprechenden Übertragung des ausgewählten NoCs in die konfliktauslösenden Gene behoben.
Außerdem kann es dazu kommen, dass einem Knoten kein Task zugeordnet wird, da beispielsweise bei der Platzierung von zwei Gruppen eine Lücke zwischen diesen entsteht. In diesem Fall werden die ungenutzten Knoten aus den Genen zur Bestimmung der Zuordnung herausgerechnet und die betroffenen Gene korrigiert.

6.2 Analyse der Anwendung

Die Analyse der Zugriffsmuster von der Anwendung auf die Programmierschnittstelle des CiAO-Betriebssystems, ist der erste Schritt zur Ableitung der folgenden Schichten, nachdem die im Genom kodierten Konfigurationsinformationen für ein Individuum von dem EA festgelegt wurden. Zu diesem Zeitpunkt ist bereits anhand der kodierten Genominformationen bekannt, wie viele Tasks das System verarbeiten muss und welche Ressourcen zur Berechnung sowie zur Kommunikation verwendet werden sollen. Zur vollständigen Konfigurierung der Betriebssysteminstanzen und zur Bewertung der Individuen, müssen jedoch weitere Informationen aus der Anwendung gewonnen werden. In Unterkapitel 6.2.1 wird zunächst beschrieben, wie die benötigten Informationen zur Maßschneiderung der Betriebssysteminstanzen bestimmt werden. Im Anschluss wird in Unterkapitel 6.2.2 ausgeführt, wie aus den Zugriffsmustern ein abstraktes Anwendungsmodell zur Bewertung der Individuen extrahiert wird.

6.2.1 Ableitung der CiAO-Instanzen

Die im Genom hinterlegten Konfigurationsinformationen beschreiben zwar einige grundlegende Eigenschaften der späteren Systeme, aber zur Maßschneiderung dieser Systeme, müssen diese Informationen zunächst analysiert und durch weitere Informationen aus der Anwendung ergänzt werden. Diese Informationen werden dann genutzt, um für jeden Knoten eines *Multi-* bzw. *Manycore*-Systems eine individuell konfigurierte Instanz des CiAO-Betriebssystems zu generieren. Der Fokus dieser Arbeit liegt dabei primär auf der Konfigurierung der erforderlichen Gerätetreiber und den Betriebssystemfunktionalitäten, welche zur Umsetzung der Kommunikation relevant sind.

6.2.1.1 Zuordnung von Tasks zu Betriebssysteminstanzen

Damit die Betriebssysteminstanzen konfiguriert werden können, wird zunächst anhand der Zuordnung der Gruppen die konkrete Platzierung der Tasks zu Kno-

ten ermittelt und somit den Betriebssysteminstanzen die jeweils auszuführenden Tasks zugeordnet. Da das vorgestellte parallele Programmiermodell dem SPMD Programmierparadigma folgt, führen alle Tasks das gleiche Programm aus. Die von den Tasks zu verarbeitenden Daten hängen allerdings von der jeweiligen Task-ID ab. Dementsprechend wird der Quelltext der Anwendung für jeden Task kopiert, mit der zugehörigen Task-ID versehen und der entsprechenden Betriebssysteminstanz bereitgestellt. Zudem werden die zugeordneten Tasks in den Konfigurationsdateien der Betriebssysteminstanzen registriert und die Prioritäten der Tasks entsprechend der Spezifizierung aus den zugehörigen `TaskGroup` Klassen-*Templates* konfiguriert.

6.2.1.2 Maßschneiderung der Kommunikation

Einer der zentralen Punkte in dieser Arbeit stellt die Kommunikation dar, denn diese lässt viel Spielraum für die Optimierung und kann einen erheblichen Einfluss auf die Qualität der Systeme haben. Die in einem System zu verwendenden Kommunikationsstrukturen werden zwar über das Genom bestimmt, aber eine genaue Zuordnung dieser Strukturen zu den einzelnen Knoten wurde bisher nur implizit über die Gruppen festgelegt. Durch die Zuordnung der Tasks zu Knoten, können nun auch die verschiedenen Kommunikationsstrukturen bestimmt werden, welche später an die betroffenen Knoten angebunden werden.

Für die Kommunikationsstrukturen Bus oder NoC ist das Vorgehen sehr geradlinig. So wird beispielsweise für jede CiAO-Instanz, auf der sich ein Task einer Gruppe befindet, die zugehörige Kommunikationsstruktur indirekt in Form der benötigten Gerätetreiber konfiguriert, falls innerhalb dieser Gruppe kommuniziert wird. Zudem wird bei dem Vorfinden einer `broadcast`-Operation und bei der gleichzeitigen Verwendung eines Busses, für die CiAO-Instanz die zur Verfügung stehende Hardwareunterstützung für diese `broadcast`-Operation konfiguriert.

Etwas aufwendiger ist die Verknüpfung der Tasks, wenn diese über ein Netz aus gemeinsamen Speichern kommunizieren sollen. Dazu wird ermittelt, welche Tasks einer Gruppe eine direkte Kommunikationsverbindung über die gemeinsamen Speicher benötigten und nur für diese Tasks wird ein Netz aus Speichern aufgespannt und die entsprechenden Gerätetreiber in den CiAO-Instanzen eingebunden. Falls zum Beispiel die Kommunikation innerhalb einer Gruppe allein aus einer abgesetzten `broadcast`-Operation besteht, werden gemeinsame Speicher nur zwischen dem Sender und den empfangenden Gruppenmitgliedern installiert und kein vollständiges Netz aus gemeinsamen Speichern zwischen allen Gruppenmitgliedern aufgespannt. So werden die verfügbaren Ressourcen, zum Beispiel die eines FPGAs, gezielt für die erforderlichen anwendungsspezifischen Anforderungen eingesetzt.

Damit die Tasks für jeden möglichen Empfänger die jeweils vorgesehene Kommunikationsstruktur verwenden, wird den Betriebssysteminstanzen zusätzlich eine generierte *Header*-Datei zur Verfügung gestellt, welche anhand von Klassen-*Templates* die Zuordnung von Task-Paaren zu Kommunikationsstrukturen spezi-

fiziert. Falls sich zwei kommunizierende Tasks unterschiedlicher Gruppen gemeinsam auf einem Knoten befinden, wird für diese Tasks die Kommunikation über den lokalen Speicher des Knotens realisiert und in der *Header*-Datei entsprechend vermerkt.

6.2.1.3 Gerätetreiber der Peripheriekomponenten

Neben der Maßschneiderung der Kommunikation, werden auch die von den Tasks genutzten Peripheriekomponenten in Form von Gerätetreibern automatisiert in die zugehörigen CiAO-Instanzen eingebunden. Welche Gerätetreiber konfiguriert werden, hängt von den getätigten Zugriffsmustern der Tasks auf die Funktionen der verschiedenen Gerätetreiber ab. Die Zugriffsmuster werden dabei mittels statischer Analyse identifiziert.

Da sich nicht alle Eigenschaften einer Peripheriekomponente aus den Zugriffsmustern ableiten lassen, müssen einige Eigenschaften von dem Entwickler vor der Entwurfsraumexploration manuell spezifiziert werden. Darunter fällt zum Beispiel die gewünschte Zähltiefe der Zeitgeberkomponenten, welche sich auf 32 oder 64 Bit einstellen lässt. Da die Zähltiefe auch Auswirkungen auf den Gerätetreiber hat, muss die entsprechende Variante in den zugehörigen CiAO-Instanz mittels dieser Spezifikation konfiguriert werden. Im Sinne des LavA-Konzeptes könnten solche Merkmale von Komponenten zukünftig über simple Merkmaldiagramme konfiguriert werden.

Aber auch die Nutzung von abstrakten Betriebssystemfunktionen innerhalb der Anwendung, wie dem Standardausgabe-*Stream* (siehe Unterkapitel 5.2.2) oder der Systemuhr, wird über eine statische Quelltextanalyse erkannt und sowohl das Merkmal für die Betriebssystemabstraktion als auch der zugehörige Gerätetreiber für die CiAO-Instanz konfiguriert. Zur Konfigurierung der Gerätetreiber werden zur Zeit plausible Voreinstellungen verwendet. Für die Systemuhr wird dementsprechend ein Gerätetreiber für eine Zeitgeberkomponente mit 64 Bit Zähltiefe konfiguriert und der Standardausgabe-*Stream* wird standardmäßig mit einem UART verknüpft.

6.2.1.4 Konfigurierung von Unterbrechungen

Ein weiterer Teil der automatisierten Konfigurierung der CiAO-Instanzen, umfasst die Behandlung von Unterbrechungen. Falls der Anwendungsentwickler eine Unterbrechungsbehandlung für eine Peripheriekomponente implementiert, wird diese automatisch erkannt, über den Parameter der Behandlungsfunktion dem entsprechenden Gerätetreiber zugeordnet und für dieses Gerät die Eigenschaft zur Unterstützung von Unterbrechungen aktiviert. Zudem wird für alle Knoten auf denen sich Geräte mit aktivierter Unterbrechungsunterstützung befinden, automatisch der Treiber für einen *Interrupt*-Controller konfiguriert.

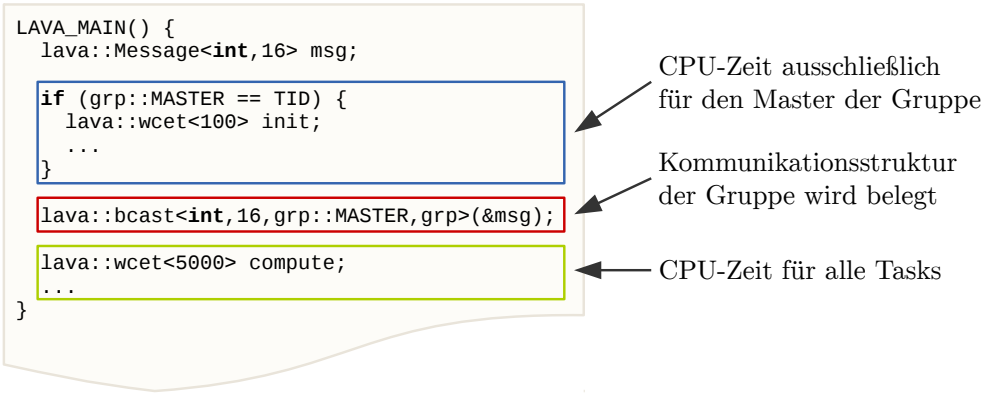


Abbildung 6.7: Analyse der Anwendung zur Bestimmung eines Anwendungsmodells

6.2.2 Anwendungsmodell

Um die zeitlichen Eigenschaften eines Individuums bewerten zu können, wird ein abstraktes Modell auf Basis der Anwendung bestimmt. Wie auch bei der Konfigurierung der CiAO-Instanzen, soll das Anwendungsmodell automatisiert mittels statischer Analysen aus der Anwendung gewonnen werden. Aus dem resultierenden, generischen Anwendungsmodell soll für unterschiedliche Werkzeuge zur Analyse von nebenläufigen Systemen, die benötigte Eingabe abgeleitet werden (siehe Unterkapitel 6.4). So kann der Anwender abhängig von den jeweiligen Anforderungen, verschiedene Werkzeuge zur Verifikation oder zur Abschätzung der zeitlichen Systemeigenschaften nutzen.

6.2.2.1 Zerlegung der Tasks

Die einzelnen Tasks der Anwendung belegen während ihrer Ausführung verschiedene Ressourcen eines Systems, zum Beispiel einen Prozessor oder auch eine oder mehrere Kommunikationsstrukturen. Um die Systeme auswerten zu können, müssen die Tasks bezüglich der jeweils verwendeten Ressourcen in einzelne Blöcke zerlegt werden.
Eine einfache Anwendung mit einem typischen Verhalten ist in Abbildung 6.7 dargestellt. In dieser Beispielanwendung berechnen die Tasks der Gruppe `grp` gemeinsam eine Lösung (grüner Block) für eine Datenmenge, welche von dem *Master*-Task zuvor initialisiert (blauer Block) und mittels `broadcast`-Operation (roter Block) an die übrigen Gruppenmitglieder verteilt wurde. Der blaue Block, zur Initialisierung der Daten, wird ausschließlich von dem *Master*-Task betreten und verursacht dementsprechend nur auf dem Knoten dieses Tasks CPU-Zeit. Die Berechnung der Lösung wird hingegen von allen Tasks durchgeführt, wodurch die CPU-Zeit auf alle Knoten dieser Gruppe entfällt. Die Ausführungszeiten für diese Blöcke muss der Entwickler über das Klassen-*Template* `wcet` der Programmierschnittstelle annotieren. Auch die Informationen zur durchzuführenden Kommu-

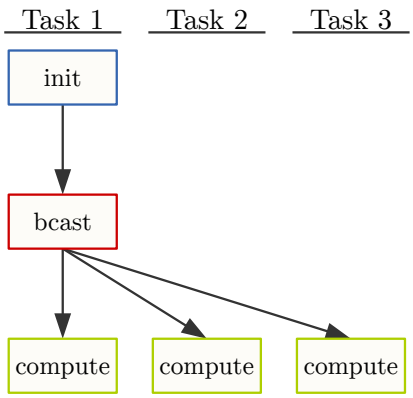


Abbildung 6.8: Abhängigkeiten zwischen den Task-Blöcken einer Anwendung

nikation können direkt aus den Aufrufen der Programmierschnittstelle ausgelesen werden. So ist zum Beispiel ersichtlich welche Operation durchgeführt werden soll, welche Tasks bei dieser Operation involviert sind und welche Datenmenge über die zugehörige Kommunikationsstruktur übermittelt werden soll. Aufgrund der statischen Charakteristik der parallelen Programmierschnittstelle sind diese Informationen auch für eine Analyse zur Übersetzungszeit sichtbar.
Damit aus den einzelnen Task-Blöcken der Anwendung ein Abbild der Anwendung bzw. des Systems entstehen kann, werden die Blöcke den zugehörigen Ressourcen zugeordnet. Blöcke, welche CPU-Zeit erfordern, können unmittelbar anhand der Task-ID einem Knoten und damit einem bestimmten Prozessor zugeordnet werden. Demgegenüber werden Blöcke zur Kommunikation mittels der involvierten Gruppe bzw. Gruppen einer konkreten Kommunikationsstruktur zugewiesen.

6.2.2.2 Bestimmung der Abhängigkeiten

Die bisher gesammelten Informationen bilden noch keine Abhängigkeiten zwischen den Task-Blöcken ab. Die Abhängigkeiten sind jedoch erforderlich, um die zeitlichen Abläufe innerhalb eines Systems mittels der verschiedenen Werkzeuge präzise darstellen und analysieren zu können. Um die Abhängigkeiten zwischen den Task-Blöcken zu ermitteln, werden insbesondere die Operationen zur Kommunikation automatisiert untersucht, da diese Kommunikationsmuster eine Interaktion mit anderen Tasks erfordern und die weiteren Berechnungsschritte von den zu übertragenden Daten abhängen.
In Abbildung 6.8 sind die Abhängigkeiten zwischen den Blöcken für die Beispielanwendung aus dem vorherigen Unterkapitel task-übergreifend eingezeichnet. In diesem Beispiel besteht die Gruppe `grp` aus drei Tasks. Der Task mit der ID `1` übernimmt in diesem Beispiel die Rolle des *Master*-Tasks und führt die Initialisierung der Daten sowie das Absenden dieser mittels der `broadcast`-Operation durch. Die Berechnung kann für alle Tasks aufgrund der Abhängig-

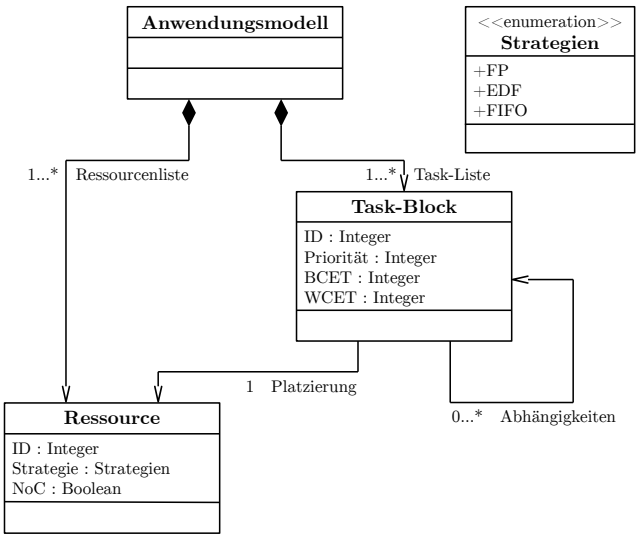


Abbildung 6.9: Modellierung der Anwendung anhand eines Metamodells

keiten erst dann erfolgen, wenn diese beiden Schritte von dem Task mit der ID `1` abgearbeitet wurden. In diesem Beispiel wird davon ausgegangen, dass die Ressource zur Kommunikation eine hardwareseitige Unterstützung zur Umsetzung der `broadcast`-Operation bietet, weshalb für das Anwendungsmodell nur ein Block zum Absetzen der `broadcast`-Operation für beide Empfänger-Tasks generiert wird. Falls die Kommunikationsstruktur diese Unterstützung nicht bieten würde und jeder Empfänger nur einzeln adressiert werden könnte, müssten zwei dieser Blöcke eingesetzt werden. Beide Empfänger würden dann von einem dieser Blöcke abhängen und könnten mit der Berechnung beginnen, sobald der zugehörige Block abgearbeitet wurde.

6.2.2.3 Modellierung der Anwendung

Die gesammelten Informationen werden, ähnlich wie bei der Hardware-Produktlinie, in einem Modell festgehalten, um für das gewünschte Analysewerkzeug mittels Modell-zu-Text-Transformation die jeweils notwendige Beschreibung generieren zu können. Durch die neutrale Beschreibung der Anwendung über ein abstraktes Modell können zukünftig auch weitere Analysewerkzeuge an LavA angebunden werden, ohne dass umfangreiche Eingriffe an dem komplexen Analyseprozess erforderlich sind.
Analog zur Hardware-Produktlinie wird der Bauplan eines Anwendungsmodells von einem Metamodell bestimmt. Das mit Hilfe des *Eclipse Modelling Frameworks* (EMF) modellierte Metamodell zur Repräsentation der Anwendung wird in Abbildung 6.9 präsentiert. Im Wesentlichen lassen sich die Anwendungsmodelle aus zwei Bausteinen zusammensetzen: Ressourcen und Task-Blöcke. Ressourcen repräsentieren Hardwarekomponenten auf denen Task-Blöcke abgearbeitet

werden können, also zum Beispiel Prozessoren oder Kommunikationsstrukturen. Diese sind mit einer ID versehen und können bezüglich verschiedener Zuteilungsstrategien, wie FP (*Fixed Priority*), EDF (*Earliest Deadline First*) oder FIFO (*First In – First Out*), konfiguriert werden. Bisher werden die Zuteilungsstrategien in dem LavA-Prozess noch nicht variiert und die Ressourcen arbeiten grundsätzlich mit festen Prioritäten. Da die NoC-Kommunikationsstrukturen potenziell mehrere Task-Blöcke gleichzeitig verarbeiten können, fällt diesen Ressourcen eine Sonderrolle zu, welche bei der zeitlichen Bewertung der Individuen beachtet werden muss und durch den letzten Parameter signalisiert werden kann. Auch die Task-Blöcke bekommen zur Identifizierung IDs zugewiesen und zudem die Priorität ihrer Tasks zugeordnet. Die zeitlichen Eigenschaften für Task-Blöcke mit CPU-Zeit werden direkt anhand der annotierten Zeiten aus der Programmierschnittstelle gefüllt. Für Task-Blöcke, welche die Kommunikation betreffen, werden die Zeiten automatisiert anhand der genutzten Kommunikationsstruktur sowie der zu versendenden Datenmenge berechnet und müssen nicht händisch durch den Entwickler angegeben werden. Zusätzlich können in dem Modell die zuvor ermittelten Abhängigkeiten zwischen den Task-Blöcken spezifiziert werden.

6.2.3 Fazit

Zur Maßschneiderung der hochkonfigurierbaren CiAO-Betriebssysteme lassen sich mittels des hier präsentierten Ansatzes viele Konfigurationsinformationen direkt aus der Anwendung ableiten oder über die Kodierung des Genoms zur Verfügung stellen. Insbesondere die Kommunikation und die damit einhergehenden Gerätetreiber werden vollständig von der Programmierschnittstelle verborgen und können von dem EA optimiert werden, ohne dass der Quelltext des Entwicklers an die veränderten Konditionen angepasst werden muss.

Zusätzlich kann die gebotene Schnittstelle auch zur Bestimmung eines abstrakten Anwendungsmodells verwendet werden und die Systeme so hinsichtlich zeitlicher Kriterien bewertet oder auch verifiziert werden.

Zukünftig könnten noch weitere Punkte zur Maßschneiderung der Systeme in diesen Ansatz einfließen, wie die Bestimmung der optimalen Zuteilungsstrategien für die gesamte Anwendung oder dediziert für einzelne Ressourcen des Systems. Zudem wird bisher nur eine Prozessorarchitektur der Hardware-Produktlinie von CiAO unterstützt, weshalb die anderen beiden Architekturen zum aktuellen Zeitpunkt von der automatisierten Optimierung nicht berücksichtigt werden. Allerdings müsste zur Integration von weiteren Architekturen in den Optimierungsprozess auch die bisher eingesetzte Methode zur Annotierung der WCET bzw. BCET erweitert werden, denn durch die unterschiedlichen Architekturen ändern sich auch die CPU-Zeiten der einzelnen Knoten. Dazu könnten zum Beispiel automatisierte Verfahren zur Bestimmung der zeitlichen Eigenschaften von Task-Blöcken eingesetzt werden und den Entwickler somit von der manuellen Annotation der CPU-Zeiten befreien.

6.3 Analyse der Hardware-API

Nachdem die Softwareschichten für die einzelnen Knoten konfiguriert und zusammengesetzt wurden, kann in dem nächsten Schritt die zugehörige Hardwarestruktur maßgeschneidert werden. Durch die Verwendung der Hardware-API in den CiAO-Betriebssysteminstanzen, wird die Konfigurierung der Hardwarestruktur stark vereinfacht, denn die benötigten Konfigurationsinformationen stecken bereits in den Softwareschichten, beispielsweise in den Gerätetreibern, und müssen dementsprechend nur noch identifiziert und herausgefiltert werden. Dazu werden die Softwareschichten analysiert, die Hardwarekomponenten anhand der Hardware-API-Instanzen identifiziert und diese Informationen, einschließlich der zugehörigen Parameter der Klassen-*Templates*, in das Hardwaremodell übernommen. So wird sichergestellt, dass die zugrunde liegende Hardwarestruktur passend auf die Anforderung der Software abgestimmt und generiert werden kann.
Nachdem die Konfigurationsinformationen für alle Knoten in das Hardwaremodell übertragen wurden, können den Peripheriekomponenten der Knoten automatisiert Adressen zugewiesen werden. Über diese Adressen können die Gerätetreiber der Betriebssysteminstanzen auf die Register der Peripheriekomponenten zugreifen. Um sicherzustellen, dass Hard- und Software eine konsistente Sicht auf diese Adressen haben, werden diese sowohl im Hardwaremodell vermerkt als auch in entsprechend generierten *Header*-Dateien für die einzelnen Betriebssysteminstanzen hinterlegt.
Da nach diesen Schritten die Betriebssysteminstanzen nun vollständig generiert wurden, kann in einem letzten Schritt die Größe der lokalen Instruktions- und Datenspeicher der Prozessoren ermittelt werden. Um die Größen dieser Speicher zu ermitteln, wird die Software für jeden Knoten übersetzt und die notwendige Speichergröße über das GNU-Werkzeug `size` bestimmt. Aufgerundet auf die nächste über die Hardware-Produktlinie generierbare Speichergröße, werden die einzelnen Größen in das Hardwaremodell übernommen. Anhand der Modell-zu-Code-Transformation des EMF kann dann im Anschluss aus dem Hardwaremodell der VHDL-Quelltext zu der konfigurierten Hardwarestruktur generiert werden.
Für jeden Zyklus innerhalb der Entwurfsraumexploration liegen zu diesem Zeitpunkt des LavA-Prozesses also die vollständig maßgeschneiderten Schichten der Soft- und Hardware für einen Punkt im Entwurfsraum bzw. dessen Repräsentation durch ein Individuum vor.

6.4 Bewertung der Individuen

Damit der EA die Individuen über Generationen hinweg stetig optimieren kann, muss dem EA die Güte der einzelnen Individuen bezüglich verschiedener Kriterien bereitgestellt werden. Die wichtigsten Kriterien für einen Anwendungsentwickler sind die zeitlichen Eigenschaften des Systems und die für das System benötigten Ressourcen. So kann der Entwickler erkennen, ob das generierte System die vorgegebenen Aufgaben in dem gewünschten zeitlichen Rahmen abarbeiten kann

und, inwiefern beispielsweise ein FPGA, welches das System aufnehmen soll, auch die nötigen Kapazitäten dazu bereitstellt. Wie bereits zuvor erläutert, wird die Energieaufnahme der Systeme in dieser Arbeit nicht betrachtet.
In den folgenden Unterkapiteln wird ausgeführt, wie sowohl die Ressourcen für die Systeme bestimmt (Unterkapitel 6.4.1) als auch die Performanz der Systeme untersucht (Unterkapitel 6.4.2) wird.

6.4.1 Ressourcenmodelle für FPGA-Familien

Die auf dem Markt angebotenen FPGAs decken hinsichtlich der Leistungsfähigkeit und der bereitgestellten konfigurierbaren Ressourcen ein breites Spektrum ab. Die Ressourcen der FPGAs bieten, je nach Typ, Platz für wenige bis zu vielen Dutzenden Prozessoren. Die benötigten FPGA-Ressourcen sind dementsprechend für ein System eine entscheidende Eigenschaft zur Abbildung der Qualität, denn die notwendigen Ressourcen können beispielsweise darüber entscheiden, ob für ein System das nächst größere und damit auch teurere FPGA verwendet werden muss. Kritische FPGA-Ressourcen für die generierten LavA-Systeme stellen in erster Linie die LUTs (*Look-Up Tables*) und die BRAMs (Block-RAMs) dar. Um Aussagen über die erforderlichen Ressourcen eines Systems bzw. des repräsentierenden Individuums treffen zu können, müssen diese Ressourcen bereits während der Entwurfsraumexploration ausgewertet und an den EA weitergereicht werden.
Eine exakte Bestimmung der notwendigen FPGA-Ressourcen für ein System ist zum Beispiel über die Synthesewerkzeuge der FPGA-Hersteller möglich. Problematisch ist dabei jedoch, dass die Synthesezeiten für größere Systeme viel Rechenzeit in Anspruch nehmen und die Synthese oftmals auch auf Rechnern mit Mehrkernarchitekturen nur schlecht skaliert. Insbesondere innerhalb einer Entwurfsraumexploration in welcher es erforderlich ist, dass mehrere Hundert oder sogar Tausende Systeme synthetisiert werden, um die benötigten FPGA-Ressourcen ermitteln zu können, verlängert sich die Optimierung durch die Einbindung der Synthesewerkzeuge erheblich.
Die Synthesezeiten für verschiedene Ausprägungen von LavA-Systemen auf einem Rechner mit vier Intel Xeon E5-4640 Prozessoren (jeweils acht Kerne) und 256 GB Hauptspeicher sind in Abbildung 6.10 dargestellt. Zu beachten ist bei diesen Ergebnissen jedoch, dass die volle Rechenleistung dieses Rechners mit dem Synthesewerkzeug von Xilinx (ISE Design Suite 14.7), aufgrund der erwähnten schlechten Skalierbarkeit, nicht ausgenutzt wird. Auf der x-Achse des Diagramms sind die verschiedenen LavA-Systeme aufgetragen, wobei die Zahlenwerte für die Anzahl der MB-Lite+ Prozessoren in dem System stehen und darauf folgend die eingesetzte Kommunikationsstruktur zur Verbindung der Prozessoren aufgeführt ist. Auf der y-Achse kann die für die Systeme benötigte Synthesezeit in Minuten abgelesen werden. An diesem Diagramm ist zu erkennen, dass die Synthese bereits für Systeme mit 16 Prozessoren, abhängig von der Kommunikationsstruktur, zwischen 11 und 15 Minuten liegen kann. Diese Problematik wird noch weiter verschärft, wenn noch größere Systeme von dem EA erzeugt werden und diese die

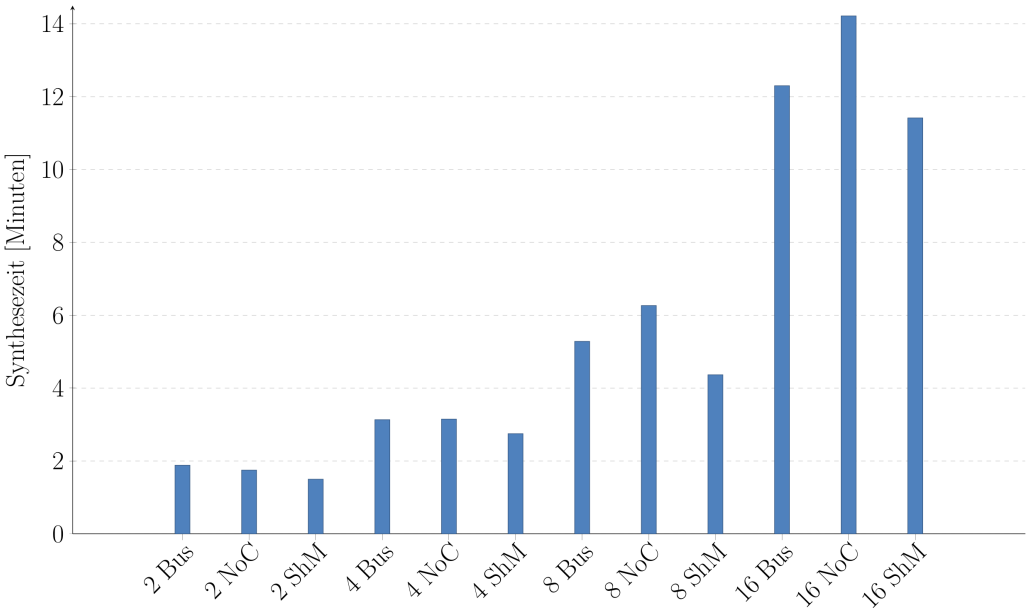


Abbildung 6.10: Synthesezeiten für verschiedene Varianten von LavA-Systemen (für ein Xilinx Virtex-5 XC5VLX110T FPGA)

Dimensionen von *Manycore*-Systemen mit beispielsweise 64 oder mehr Prozessoren erreichen.

Damit die Optimierung durch die Synthesewerkzeuge nicht unnötig ausgebremst wird, werden die Ressourcen in dieser Arbeit mittels Modellen abgeschätzt. Dieser Ansatz wurde bereits in dem Kontext des LavA-Projektes in dem folgenden Artikel [MS11] veröffentlicht. Da die entwickelten Modelle hochgradig von dem FPGA-Hersteller und dem FPGA selbst abhängen, sind die Modelle jeweils nur für eine kleine Familie von ähnlichen FPGAs nutzbar. Die Ressourcenmodelle des LavA-Projektes unterstützen die Xilinx FPGA-Familien Spartan-3E und Virtex-5. Für die beiden FPGA-Familien wurden dazu jeweils eigene Modelle angelegt, denn obwohl beide Familien vom selben Hersteller stammen, unterscheiden sich diese maßgeblich in ihrer Architektur. Die Virtex-5 Familie bietet zum Beispiel LUTs mit sechs Eingängen, wohingegen die LUTs der – bereits in dem Grundlagenkapitel vorgestellten – Spartan-3E Familie nur vier Eingänge bereitstellen und demzufolge eine einzelne LUT der Spartan-3E Familie weniger Logik realisieren kann. Neben den LUTs unterschieden sich die beiden Familien zudem bezüglich des zur Verfügung stehenden Speicherplatzes innerhalb eines BRAMs. Zudem muss beachtet werden, dass die letztendlich benötigten Ressourcen für ein LavA-System auf einem FPGA auch von dem Synthesewerkzeug und dessen Einstellungen abhängen können. Zur Generierung der Ressourcenmodelle wurde die Xilinx ISE Design Suite 14.7 mit den Standardeinstellungen genutzt.

Aufgrund der feingranularen Konfigurierungsoptionen der Hardware-Produktlinie und dem daraus resultierenden großen Konfigurationsraum, ist es nicht mög-

lich, sämtliche Systeme aus dem Konfigurationsraum zu untersuchen. Stattdessen müssen anhand von ausgewählten Systemen Trends in dem Ressourcenbedarf der verschiedenen Komponenten erkannt werden. Die Informationen dazu können über die ISE Design Suite gewonnen werden, welche eine hierarchische Auflistung der Komponenten und deren jeweiligen Ressourcenbedarf zur Verfügung stellen kann. Am schwierigsten ist die Abschätzung für Komponenten, in denen andere Subkomponenten verknüpft und somit deren Signale zusammenlaufen, wie zum Beispiel die LavA- oder die Knotenkomponente. Problematisch ist dabei, dass diese Komponenten praktisch keine eigene Logik implementieren, sondern aufgrund der Bindegliedfunktion für die Subkomponenten, im Vergleich zu anderen Komponenten bezüglich des Ressourcenbedarfs einer stärkeren Streuung unterliegen. Damit diese Komponenten trotzdem in dem Ressourcenmodell berücksichtigt werden können, werden diese approximiert. Eine Bewertung der Ressourcenmodelle bezüglich der Abweichung zu den Ergebnissen von Synthesewerkzeugen wird in Unterkapitel 7.2 vorgestellt.
Die folgende Gleichung 6.1 zeigt exemplarisch, wie die Ressourcen für die einzelnen Komponenten der LavA Hardware-Produktlinie in dem Metamodell annotiert und berechnet werden können:

$$\begin{aligned} MBLite{+}_{LUT4} = {} & 1724 + \\ & 212 * (BarrelShifter\ ?\ 1 : 0) + \\ & 40 * (Multiplier\ ?\ 1 : 0)) + \\ & 331 * (JTAG\ ?\ 1 : 0) \end{aligned} \tag{6.1}$$

Diese Gleichung beschreibt, abhängig von den gewählten Konfigurationsmerkmalen, die benötigten LUTs auf einem FPGA der Spartan-3E Familie für den MB-Lite+ Prozessor. Dadurch, dass die Gleichungen ein weiteres Attribut der jeweiligen Hardwarekomponenten im Metamodell darstellen, kann innerhalb dieser Gleichungen auch auf die entsprechende Komponentenkonfiguration zugegriffen werden. In der dargestellten Gleichung ist die Berechnung der LUTs beispielsweise davon abhängig, ob ein Hardware-*Barrel-Shifter*, ein Hardware-Multiplizierer oder eine JTAG-Schnittstelle ausgewählt wird.
In der ersten Zeile der Gleichung sind die Grundkosten des Prozessors ohne die optionalen Erweiterungen angegeben. Die optionalen Erweiterungen des Prozessors werden in den folgenden drei Zeilen in die Berechnung der Gesamtkosten einbezogen. Zur Auswertung der hier gegebenen bedingten Ausdrücke, wird auf das Modell und die dort hinterlegte Konfiguration zurückgegriffen. Die überraschend geringen Kosten für den Hardware-Multiplizierer, sind durch die dedizierten Ressourcen zur Multiplikation auf dieser FPGA-Familie zu erklären. Dadurch muss der optionale Multiplizierer nicht mittels der generischen LUTs realisiert werden, sondern wird von dem Synthesewerkzeug auf die spezialisierten Multiplikationseinheiten des FPGAs gelegt. Die wenigen, dennoch aufzuwendenden LUTs, werden zum Beispiel zur nun erforderlichen Dekodierung der hinzugefügten Multiplikationsbefehle benötigt.

Die Berechnung der Gesamtkosten für ein vollständiges *Multi-* bzw. *Manycore*-System wird während des Optimierungsprozesses automatisiert über die *Xtend*-Erweiterung des *Eclipse Modelling Frameworks* vorgenommen und das Ergebnis an den EA weitergeleitet. So steht die Abschätzung der erforderlichen Ressourcen umgehend zur Verfügung und die zum Teil langwierige Synthese kann zur Ermittlung der Ressourcen umgangen werden. Zusätzlich können die Ressourcenmodelle auch für manuell erstellte LavA-Systeme eingesetzt werden und dem Entwickler eine schnelle Rückmeldung darüber geben, ob das anvisierte FPGA geeignet ist oder Änderungen an dem Design erforderlich sein werden.

6.4.2 Untersuchung der Performanz

Die Bestimmung von zeitlichen Eigenschaften für nebenläufige Systeme ist ein intensiv untersuchter Forschungszweig, aus dem bereits einige Techniken und Werkzeuge hervorgegangen sind. Wie in Unterkapitel 3.3 bereits aufgezeigt, lassen sich die Techniken zur Untersuchung der Performanz grob in zwei Kategorien gliedern. Zum einen die auf Simulatoren basierenden Techniken und zum anderen die analytischen Methoden. Die Vor- und Nachteile der Ansätze aus diesen beiden Kategorien wurden bereits in dem referenzierten Unterkapitel dargelegt.

In diesem Unterkapitel soll insbesondere die Anwendbarkeit dieser Techniken bezüglich der generierbaren LavA-Systeme untersucht werden. So können auf der einen Seite Erkenntnisse darüber gewonnen werden, in welchen Bereichen diese Ansätze noch Defizite aufweisen und möglicherweise nicht eingesetzt werden können und auf der anderen Seite sollen Ansätze, welche sich für eine zeitliche Analyse von LavA-Systemen eignen, in den LavA-Prozess eingebunden werden und den EA hinsichtlich der verschiedenen Individuen mit Performanzdaten versorgen. Da ein wichtiger Faktor für die automatisierte Entwurfsraumexploration die Analysedauer darstellt, werden besonders zeitaufwendige Ansätze, wie beispielsweise Instruktionssatzsimulatoren, bei dieser Untersuchung nicht berücksichtigt.

6.4.2.1 Real-Time Calculus

Ein Werkzeug, welches der Kategorie der analytischen Methoden zuzuordnen ist, ist der Real-Time Calculus (RTC) [TCN00, Wan06]. Der RTC ist ein Kalkül, welches als *Toolbox* für Matlab umgesetzt ist und ermöglicht die Berechnung von harten oberen und unteren Schranken für die Ende-zu-Ende Latenz und Puffergrößen von eingebetteten Echtzeitsystemen. Die mathematische Grundlage für den RTC bildet die Max-Plus Algebra [BCOQ92]. Im Gegensatz zu Simulatoren können mit dem RTC auch Randfälle abgedeckt und Garantien bezüglich der ermittelten Latenzen und Puffergrößen ausgesprochen werden. Diese Garantien werden allerdings oftmals auf Kosten von eher pessimistisch ermittelten Schranken erreicht. Das Verhalten der zu untersuchenden Systeme wird für den RTC im Wesentlichen über die folgenden Modelle repräsentiert:

Umgebungsmodelle Die Umgebungsmodelle beschreiben die Interaktion des Systems mit ihrer Umgebung und repräsentieren zum Beispiel die Anzahl der eingehenden, externen Ereignisse, welche in dem System verschiedene Komponenten anstoßen können. Die eingehenden Ereignisse werden für den RTC über ein Tupel von Ereigniskurven (engl. *Arrival Curves*) definiert. Die Ereigniskurven legen dabei die untere und obere Schranke für die innerhalb eines bestimmten Zeitintervalls minimal bzw. maximal eingehenden Ereignisse fest.

Ressourcenmodelle Die Ressourcenmodelle bilden die zur Verfügung stehende Leistung der Systemressourcen ab. Dabei wird nicht zwischen den Ressourcen zur Ausführung von Tasks und den Ressourcen zur Kommunikation, wie beispielsweise Bussen, unterschieden. Analog zu den Umgebungsmodellen, wird auch die Leistung einer Ressource innerhalb eines bestimmten Zeitintervalls über ein Tupel von Kurven ausgedrückt, welche die minimale und maximale Leistung einer Ressource abbilden. Diese Kurven werden als Servicekurven bezeichnet.

Komponentenmodelle Die Tasks einer Anwendung werden mittels Komponenten dargestellt und verarbeiten die eingehenden Ereignisströme mithilfe der zugeordneten Ressourcen. Innerhalb der Komponenten werden dazu die eingehenden Ereignis- und Servicekurven transformiert und die neu berechneten Ereignis- und Servicekurven über die Ausgänge der Komponente an weitere Komponenten weitergereicht. Die ausgehende Servicekurve bildet dabei beispielsweise die verbleibende Leistung der genutzten Ressource ab und wird den nachfolgenden Tasks dieser Ressource bereitgestellt. Die über die Ereignis- und Service-Kurven verbundenen Komponenten bilden so ein ganzes Netzwerk von Komponenten und beschreiben somit das Gesamtsystem.

In Abbildung 6.11 ist ein LavA-System beispielhaft für den RTC visualisiert. In diesem Beispiel gibt es vier Ressourcen: drei MB-Lite+ Prozessoren und einen Bus. Unterhalb der Ressourcen befinden sich die jeweils zugeordneten Komponenten, die von den Ressourcen verarbeitet werden sollen. Die Komponenten spiegeln die in Unterkapitel 6.2.2 ermittelten Task-Blöcke einer LavA-Anwendung wieder, denn wie die Komponenten, sind auch die Task-Blöcke genau einer Ressource zugeordnet. Jede Komponente besitzt zwei Ein- und zwei Ausgänge für die ein- und ausgehenden Ereigniskurven (grün) bzw. Servicekurven (rot). In diesem Beispielsystem gibt es zwei Ereignisströme. Der eine Strom wird von einer ganzen Reihe von aufeinanderfolgenden Komponenten sowie auf verschiedenen Ressourcen verarbeitet, wohingegen der andere Strom von einer einzelnen Komponente abgearbeitet wird.
Wie anhand dieses Beispiels gezeigt, lassen sich einfache LavA-Systeme anhand des RTC modellieren und im Folgenden auch analysieren, um die Systemperformanz zu bestimmen. Probleme treten jedoch auf, wenn komplexe Kommunika-

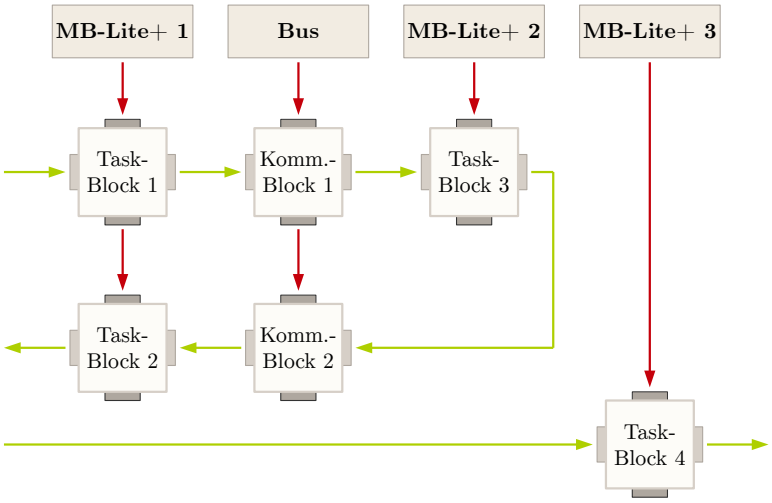
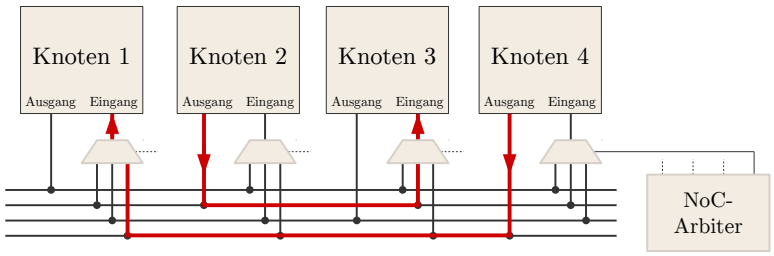


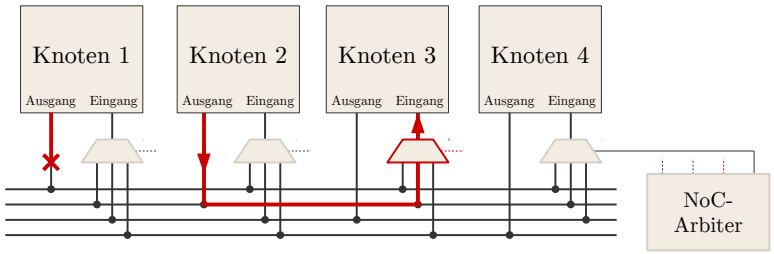
Abbildung 6.11: Visualisierung eines RTC-Modells für eine Beispielarchitektur

tionsstrukturen wie das NoC der LavA Hardware-Produktlinie in die Modellierung und Analyse einbezogen werden sollen. Im Gegensatz zu einem klassischen Bus, ermöglicht das LavA-NoC die parallele Kommunikation zwischen zwei unterschiedlichen Paaren von Kommunikationsteilnehmern. Ein Szenario, das die Möglichkeiten der parallelen Kommunikation mittels des LavA-NoCs aufzeigt, wird in Abbildung 6.12 (a) dargestellt. Da in diesem Fall unterschiedliche Teilnehmer miteinander kommunizieren, ist diese Art der Kommunikation gestattet und kann parallel erfolgen. Falls jedoch, wie in Abbildung 6.12 (b) gezeigt, Knoten 1 mit Knoten 3 kommunizieren möchte und Knoten 3 zu diesem Zeitpunkt besetzt ist, wird die Kommunikation zwischen den Knoten 1 und 3 vom Arbiter des NoCs zunächst blockiert und Knoten 1 muss mit dem Versand der Daten warten, bis die aktuell stattfindende Kommunikation beendet ist.

An diesen Szenarien lässt sich leicht erkennen, dass eine einfache Entscheidung, wie beispielsweise, ob die Ressource augenblicklich belegt oder frei ist, für die Kommunikationsstruktur NoC nicht direkt ersichtlich ist, denn dies hängt immer von dem Faktor ab, mit welchem Teilnehmer kommuniziert werden soll. Da das NoC aufgrund der potenziellen Leistungsfähigkeit ein wichtiger Bestandteil der LavA Hardware-Produktlinie ist und keine Möglichkeit gefunden wurde, um das NoC mittels des RTC zu modellieren, ist der RTC zumindest nicht für alle Ausprägungen von LavA-Systemen anwendbar. Auch für das aus der *Schedulability*-Analyse stammende Werkzeug MAST [HGGPGDM01] konnte zum Zeitpunkt der Untersuchung kein Mittel gefunden werden, um die Funktionsweise des LavA-NoCs entsprechend nachzustellen und auszuwerten.



(a) Parallele Kommunikation im NoC



(b) Nachricht von Knoten 1 zu Knoten 3 wird blockiert

Abbildung 6.12: Potenziell nutzbare parallele Kommunikation innerhalb des LavA-NoCs

6.4.2.2 Echtzeitautomaten

Als Alternative zu den Analysewerkzeugen RTC und MAST werden in dieser Arbeit im Folgenden Echtzeitautomaten (engl. *Timed Automata*), also um Zeitvariablen erweiterte endliche Automaten, auf ihre Eignung zur Performanzanalyse im Kontext des LavA-Projektes untersucht, wozu das Werkzeug Uppaal [LPY97] eingesetzt werden soll. Uppaal ermöglicht die Modellierung und Verifikation von Echtzeitsystemen auf Basis von Echtzeitautomaten. Aufgrund der gebotenen Möglichkeiten zur Verifikation kann Uppaal, wie der RTC, ebenfalls Garantien zu den zu untersuchenden Systemen aussprechen.

Mit Uppaal können Systeme über ein Netzwerk von Echtzeitautomaten modelliert werden. Zur Synchronisation bzw. Kommunikation zwischen den Automaten können Synchronisationskanäle genutzt werden. Zudem erlaubt Uppaal die Implementierung von C-artigen Funktionen, die zum Beispiel während beliebiger Transitionen zwischen Zuständen aufgerufen werden können. Dadurch werden die Echtzeitautomaten zu einem mächtigen Instrument mit dem das Systemverhalten detailliert nachgebildet werden kann. So lässt sich beispielsweise ein einfacher Algorithmus implementieren, um die ausführbereiten Tasks in einer Warteschlange zu verwalten. Zur Verarbeitung der Daten können Variablen eingesetzt und dazu sowohl auf einfache Datentypen als auch auf komplexe Datentypen, wie Felder oder Struktur-Datentypen, zurückgegriffen werden.

Zur Modellierung der LavA-Systeme werden nachfolgend drei Echtzeitautomaten vorgestellt. Jeder der Automaten beschreibt auf abstrakter Ebene einen Teilbe-

reich eines LavA-Systems. Die Automaten modellieren das Verhalten der Tasks, der Ressourcen sowie der Zuteilungsstrategien. Die Automaten selbst stellen zunächst nur Schablonen dar. Abhängig von dem jeweils zu untersuchenden LavA-System, kann dann die erforderliche Anzahl von Automaten beliebig instanziiert werden. Falls also das Anwendungsmodell eines LavA-Systems vier Task-Blöcke beinhaltet, wird für jeden Task-Block ein eigener Automat der zugehörigen Kategorie instanziiert, um das System zu analysieren. Die Automaten basieren ursprünglich auf einem Uppaal-Beispielprojekt und wurden bezüglich der Anforderungen der LavA-Komponenten sukzessive umgebaut. Insbesondere die erforderliche zeitliche Analyse der Systeme konnte mit dem Beispielprojekt zunächst nicht durchgeführt werden, weshalb die Bedingungen zur Aktivierung diverser Transitionen angepasst werden mussten. Ebenso war in dem ursprünglichen Modell eine Ressource, wie sie LavA mit dem NoC besitzt, nicht vorgesehen und dementsprechend konnten zunächst nur einfache Kommunikationsressourcen, wie Busse, einbezogen werden.

Modellierung der Task-Blöcke Der Echtzeitautomat in Abbildung 6.13 repräsentiert das Verhalten der Task-Blöcke. Der Automat kann sowohl für periodisch auftretende Tasks als auch für einmalig auszuführende Tasks verwendet werden. Die wesentlichen Zustände des Automaten sind mit einem Bezeichner in schwarzer Schrift versehen. Der Startzustand des Automaten ist mit einem doppelten Kreis markiert. Für jeden Automaten muss genau ein Startzustand existieren.
Der Automat bietet die Option die Tasks nach dem Starten des Systems mit einer individuell zu spezifizierenden Verzögerung auszuführen. Dies wird über die Bedingung am Startzustand und die Bedingung an der ausgehenden Kante des Startzustandes realisiert. Die an dem Startzustand annotierte zeitliche Bedingung (lila Schrift) gibt vor, zu welchem Zeitpunkt der Zustand spätestens verlassen werden muss und die zeitliche Bedingung an der Kante (grüne Schrift) gibt vor, zu welchem Zeitpunkt diese Transition frühestens durchgeführt werden darf. Eine Transition kann also erst dann stattfinden, wenn die gegebene Bedingung erfüllt ist. Die Bedingungen selbst setzen sich folgendermaßen zusammen. Das Feld `time` hält für jeden Task einen eigenen Zeitgeber bereit. Die jeweilige Identifikationsnummer der Tasks kann als Index verwendet werden, um auf das zugehörige Element des Feldes zugreifen zu können. Die Funktion `initOffset()`, auf der anderen Seite der Bedingung, gibt die für den Task spezifizierte Verzögerung zurück. So wird sichergestellt, dass der betreffende Task den Startzustand erst dann verlässt, wenn der angegebene Zeitpunkt exakt eingetroffen ist.
Der folgende Zustand ist ein Hilfszustand, um für periodische Tasks den nächsten Durchlauf zu initiieren und wird umgehend wieder verlassen. Beim Verlassen dieses Zustandes wird der lokale Zeitgeber des Tasks zurückgesetzt und die Hilfsfunktion `new_period()` aufgerufen. Zustände, die mit dem Buchstaben „C“ (*Committed*) markiert sind, nehmen eine Sonderrolle ein, denn wenn solch ein Zustand aktiv ist, wird die Zeit eingefroren, bis dieser wieder verlassen wird.

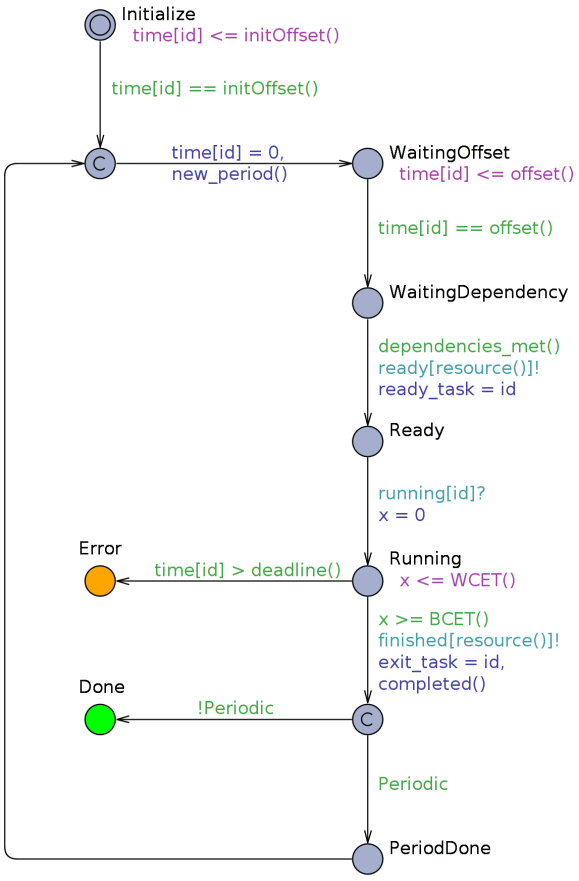


Abbildung 6.13: Zustandsautomat zur Modellierung der Task-Blöcke

Wie schon der Startzustand, wird auch der Zustand `WaitingOffset` verwendet, um die Ausführung ausgewählter Tasks zu verzögern. Im Gegensatz zur initialen Verzögerung, wirkt diese Verzögerung für periodische Tasks bei jeder neuen Periode erneut.

Der Zustand `WaitingDependency` wird genutzt, um die Abhängigkeiten, welche im Anwendungsmodell spezifiziert wurden, aufzulösen. Dementsprechend kann der Task diesen Zustand erst dann verlassen, wenn Tasks von denen er abhängt, abgearbeitet wurden. Die Überprüfung der Abhängigkeiten wird von der Funktion `dependencies_met()` kontrolliert und das Ergebnis über einen Wahrheitswert zurückgegeben.

Falls alle Abhängigkeiten aufgelöst sind, kann der Task in den Zustand `Ready` wechseln. Während der Transition wird einer der Synchronisationskanäle des Feldes `ready` verwendet, um dem Automaten der zugehörigen Ressource die Ausführbereitschaft des Tasks mitzuteilen. So wird sichergestellt, dass die beiden Automaten zeitgleich in den nächsten Zustand wechseln und der Task von dem Automaten der Ressource verarbeitet werden kann. Die Anzahl der Elemente in

dem Feld `ready` korrespondieren mit der Anzahl der Ressourcen im dem zu untersuchenden System. Das Gegenstück zu diesem Synchronisationskanal wird in dem Automaten der Ressource eingesetzt und ist mit einem Fragezeichen anstelle des Ausrufezeichens versehen. Zudem wird die lokal gespeicherte Task-Nummer während des Übergangs in den Zustand `Ready` in der globalen Variable `ready_task` abgelegt. Dadurch kann auf diese Information auch aus dem Automaten der Ressource zugegriffen werden und der ausführbereite Task anhand dieser Task-Nummer identifiziert und in die Warteschlange der Ressource eingeordnet werden.
Damit der Task in den Zustand `Running` wechseln kann, muss dieser das entsprechende Signal von dem Automaten der zugehörigen Ressource über den Synchronisationskanal `running` erhalten. Dieses Signal wird abgesetzt, sobald die Ressource nicht belegt und der Task die höchste Priorität in der Warteschlange besitzt. Die Variable `x` repräsentiert einen weiteren Zeitgeber und soll die Zeit des Tasks im Zustand `Running` festhalten. Diese Zeit wird als Referenz verwendet, damit der Task sich mindestens solange wie die spezifizierte BCET und höchstens solange wie die angegebene WCET in diesem Zustand befindet.
Falls der Task seine spezifizierte *Deadline* nicht einhalten kann, wechselt der Task in den Zustand `Error`. Ansonsten gibt der Task die Ressource über den Synchronisationskanal `finished` wieder frei und wechselt, für einmalig auszuführende Tasks, in den Zustand `Done` oder in den Zustand `PeriodDone`, falls es sich um periodische Tasks handelt.

Modellierung der Ressourcen In Abbildung 6.14 ist der Automat zur Modellierung der Ressourcen dargestellt. Wie beim RTC, wird auch hier nicht zwischen Ressourcen zur Ausführung von Tasks und den Ressourcen zur Kommunikation unterschieden. Der Automat der Ressourcen besitzt im Wesentlichen zwei zentrale Zustände in denen die Zeit voranschreitet: der Startzustand `Idle` und der Zustand `Busy`. Der Startzustand `Idle` wird verlassen, sobald das Synchronisationssignal eines auf dieser Ressource ausführbereiten Tasks eintrifft. Bei dieser Synchronisation wird das bereits angesprochene Pendant des Synchronisationskanals `ready` verwendet.
In dem nächsten Zustand wird überprüft, ob die Warteschlange der Ressource zurzeit leer ist oder sich darin bereits ausführbereite Tasks befinden. Falls die Warteschlange leer ist, wechselt der Automat direkt in den Zustand `Busy` und signalisiert dem Automaten des Tasks über den Synchronisationskanal `running`, dass dieser nun ausgeführt wird. Um den Task im weiteren Verlauf zu verwalten, wird der Task direkt an die erste Stelle (Position 0) der Warteschlange gesetzt. Für den Fall, dass sich bereits Tasks in der Warteschlange befinden, wechselt der Automat in den Zustand `QueueTask`. Während des Übergangs in diesen Zustand, werden die Informationen zur Verarbeitung des Tasks über die Funktion `setParams` in einer Datenstruktur zwischengespeichert und der Synchronisationskanal `insert_task` zur Synchronisation mit dem Automaten der Zuteilungsstrategie verwendet. Der Automat der Zuteilungsstrategie (siehe nächster Abschnitt),

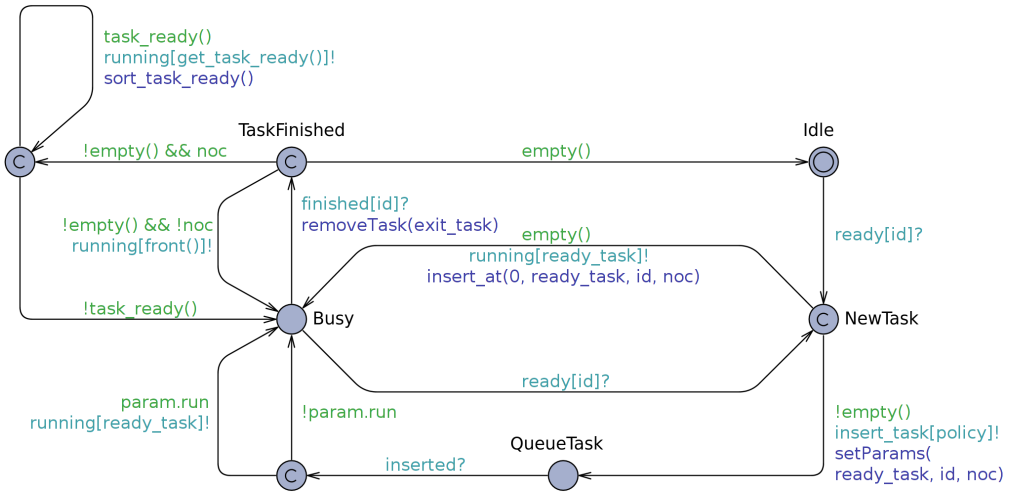


Abbildung 6.14: Zustandsautomat zur Modellierung der Ressourcen

ordnet den Task entsprechend der gewählten Strategie in die Warteschlange ein und teilt der Ressource im Anschluss über den Synchronisationskanal `inserted` mit, wenn dieser Vorgang durchgeführt wurde. Die strikte Trennung zwischen der Ressource und der Zuteilungsstrategie hat den Vorteil, dass verschiedene Strategien in einem System verwendet werden können und diese jeweils modular in einem eigenen Automaten modelliert werden können.

Der mit einem „C“ markierte Hilfszustand, zwischen den Zuständen `QueueTask` und `Busy`, ist der Sonderrolle des NoCs geschuldet. Hiermit wird für Ressourcen des Typs NoC unter anderem sichergestellt, dass die Kommunikation zwischen zwei Teilnehmern, welche von der bisher stattfindenden Kommunikation unabhängig sind, parallel durchgeführt werden kann. Falls diese Konstellation für einen Task-Block, der diese Kommunikation repräsentiert, gegeben ist, wird dies von der Zuteilungsstrategie erkannt und über den Wahrheitswert `param.run` an den Automaten der Ressource weiter gegeben. Falls der Wahrheitswert erfüllt ist, wird dem betreffenden Task-Block über den Synchronisationskanal `running` signalisiert, dass dieser nun ausgeführt wird. Ansonsten wechselt der Automat ohne diese Synchronisationsnachricht in den Zustand `Busy` und der Task-Block wird zu einem späteren Zeitpunkt ausgeführt.

Sobald über den Synchronisationskanal `finished` das Signal eingeht, dass einer der Task-Blöcke abgearbeitet wurde, wechselt der Automat in den Zustand `TaskFinished` und entfernt den Task mit der in `exit_task` gespeicherten Identifikationsnummer aus der Warteschlange. Falls sich nun keine Tasks mehr in der Warteschlange befinden, wechselt der Automat zurück in den Zustand `Idle`. Andernfalls wird unterschieden, ob es sich bei der Ressource um ein NoC oder eine andere Ressource handelt. Im Falle von anderen Ressourcen wird einfach der erste Task-Block aus der Warteschlange entnommen, diesem die Ausführung mitgeteilt und zurück in den Zustand `Busy` gewechselt. Für das NoC gibt es wiederum eine

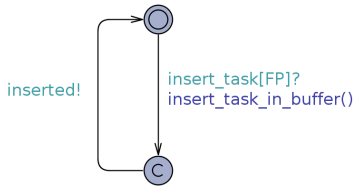


Abbildung 6.15: Zustandsautomat zur Modellierung der Zuteilungsstrategie für feste Prioritäten

Sonderbehandlung, da durch die nun beendete Kommunikation eventuell mehrere Task-Blöcke gleichzeitig als ausführbereit markiert werden können. Um alle parallel auszuführenden Task-Blöcke über die Ausführung zu informieren, wird der Zustand auf der linken Seite der Abbildung 6.14 verwendet. Dabei überprüft die Funktion `task_ready`, ob es noch Task-Blöcke gibt, welche die Voraussetzungen zur Ausführung erfüllen. Falls es diese gibt, werden die Tasks über die Ausführung informiert. Sobald es keine entsprechenden Task-Blöcke mehr gibt, wechselt der Automat in den Zustand `Busy`.

Modellierung der Zuteilungsstrategie In Abbildung 6.15 ist der Zustandsautomat zur Modellierung der Zuteilungsstrategie für feste Prioritäten abgebildet. Letztendlich besteht dieser Automat aus nur zwei Zuständen. Falls von einer Ressource das entsprechende Signal zur Einordnung eines Task-Blocks in die zugehörige Warteschlange eintrifft, wird die Funktion `insert_task_in_buffer` aufgerufen. Diese Funktion sortiert den neuen Task-Block abhängig von der jeweiligen Priorität in die Warteschlange. Nachdem der Task-Block einsortiert und die Ressource darüber informiert wurde, wechselt der Automat umgehend zurück in seinen Startzustand.

Auswertung der LavA-Systeme In jedem Zyklus des LavA-Optimierungsprozesses wird automatisiert für das zu untersuchende System das entsprechende Uppaal-Modell mittels des EMF zusammengesetzt. Grundlage bilden die vorgestellten Echtzeitautomaten und das Anwendungsmodell aus Unterkapitel 6.2.2, welches die benötigten Informationen zu dem LavA-System, wie die Eigenschaften der Task-Blöcke, die Abhängigkeiten oder die Zuteilung zu Ressourcen, beisteuert.
Über eine eigene Anfragesprache des Uppaal-Werkzeugs lassen sich dann die zu verifizierenden Systemeigenschaften spezifizieren. Eine wichtige Systemeigenschaft für die generierten LavA-Systeme ist zum Beispiel, ob das System die gegebene *Deadline* zur Berechnung der gestellten Aufgabe einhalten kann. Bezogen auf die Uppaal-Echtzeitautomaten könnte diese Systemeigenschaft für Systeme mit einmalig auszuführenden Tasks folgendermaßen ausgedrückt werden: haben alle Echtzeitautomaten der Tasks den Zustand `Done` erreicht und ist die vergangene Zeit, welche zum Erreichen dieser Zustände erforderlich war, kleiner als die *Deadline*? Falls eine solche Anfrage mit Uppaal verifiziert werden kann, würde

das untersuchte System also für die gegebene Anwendung die Anforderungen bezüglich der zeitlichen Eigenschaften erfüllen. In der Uppaal-Anfragesprache kann diese Systemeigenschaft wie folgt spezifiziert werden:

$$A[]\ timer > x\ imply\ forall\ (i\ :\ tid)\ Task(i).Done \tag{6.2}$$

Im Detail drückt diese Anfrage folgendes aus: auf allen Pfaden und in allen Zuständen gilt, wenn die vergangene Zeit `timer` größer als die angegebene *Deadline* `x` ist, impliziert dies, dass sich alle Tasks im Zustand `Done` befinden. Diese Anfrage ist also genau dann erfüllt, falls für dieses Modell garantiert werden kann, dass alle Tasks unter allen auftretenden Umständen abgearbeitet wurden, bevor die *Deadline* überschritten wurde.
Neben der Verifikation ist es mit Uppaal zusätzlich möglich die modellierten Systeme auch zu simulieren. Damit können dann zwar offensichtlich keine Garantien bezüglich der einzuhaltenden Schranken ausgesprochen werden, da die Randfälle mit einer Simulation nur schwer bzw. nicht abzudecken sind, aber die Simulation liefert genaue Zeiten für die einzelnen Durchläufe zurück. Die ermittelten Zeiten können so ein guter Indikator für die zeitliche Distanz zur *Deadline* sein und liefern nicht nur zurück, ob die Anfrage erfüllt oder nicht erfüllt ist. Die Anfrage zur Simulation der LavA-Systeme lässt sich folgendermaßen spezifizieren:

$$simulate\ 1000\ [<= x]\ \{timer\}\ :\ 1000:\ forall\ (i\ :\ tid)\ Task(i).Done \tag{6.3}$$

Das Schlüsselwort `simulate` in Verbindung mit der folgenden Zahl veranlasst Uppaal das Modell 1.000-mal zu simulieren. Die maximale Simulationszeit wird durch das `x` angegeben. Für LavA kann an dieser Stelle beispielsweise die *Deadline* verwendet werden. Die Zeitgebervariable `timer` des Modells wird in dieser Anfrage zur Ausgabe der vergangenen Systemlaufzeit verwendet. Die folgende Zahl spezifiziert, wie viele Simulationsläufe, die am Ende angegebene Bedingung erfüllen müssen, bis die Simulation vorzeitig abgebrochen wird. In dem Beispiel würden also alle 1.000 Läufe in jedem Fall durchgeführt und ein vorzeitiger Abbruch der gesamten Simulation würde nicht stattfinden. Die zu erfüllende Bedingung am Ende der Anfrage beschreibt, analog zur Verifikation, dass sich alle Tasks im Zustand `Done` befinden müssen, bevor die *Deadline* überschritten wird, damit der jeweilige Durchlauf als erfolgreich gewertet wird. Nachdem diese Bedingung erfüllt ist, wird der aktuelle Durchlauf beendet und die im Modell verstrichene Zeit mittels der Zeitgebervariable `timer` ausgegeben. Anhand der ausgegebenen Zeiten aller Simulationsläufe kann dann zum Beispiel die minimale oder maximale Laufzeit des Gesamtsystems abgeschätzt werden.

Laufzeiten zur Verifikation oder Simulation der Modelle Wie bereits zuvor angesprochen, ist auch die benötigte Zeit zur Bestimmung der Systemperformanz entscheidend, wenn ein entsprechendes Werkzeug in dem LavA-Optimierungsprozess verwendet werden soll. In der Tabelle 6.1 sind deshalb die

		Verifikation		Simulation (10.000 Läufe)	
Task-Blöcke	Ressourcen	Laufzeit	Hauptspeicher	Laufzeit	Hauptspeicher
4	2	0,03 s	6,66 MB	1,96 s	7,70 MB
8	4	1,29 s	12,54 MB	4,59 s	8,46 MB
12	6	3,57 min	0,67 GB	9,48 s	9,70 MB
13	6	13,48 min	2,25 GB	10,27 s	9,88 MB
14	7	51,35 min	7,58 GB	11,62 s	10,56 MB
15	7	3,13 h	22,57 GB	13,55 s	10,79 MB
16	8	11,18 h	73,53 GB	14,96 s	10,87 MB

Tabelle 6.1: Auswertung der Uppaal-Verifikation und -Simulation

benötigten Laufzeiten und die genutzte Menge Hauptspeicher für verschiedene Systeme, sowohl für die Verifikation als auch für die Simulation, angegeben. Zur Ermittlung der Zeiten wurde ein Rechner mit vier Intel Xeon E5-4640 Prozessoren (jeweils acht Kerne) und 256 GB Hauptspeicher eingesetzt. Die Basis zur Ermittlung der Laufzeiten bildet die 64-Bit-Version von Uppaal 4.1.19[1].

Der entscheidende Faktor für die Analysezeiten der LavA-Systeme mittels Uppaal, ist die Anzahl der vorhandenen Zustände in dem zu untersuchenden System. Um die Anzahl der Zustände für die zu untersuchenden Systeme zu erhöhen, wurde für jedes weitere System in der Tabelle die Anzahl der Task-Block- und Ressourcenautomaten schrittweise angehoben.

In der Tabelle ist zu erkennen, dass bereits ab dem vierten System mit 13 Task-Blöcken und 6 Ressourcen die Verifikation fast 14 Minuten benötigt. Selbst für diese vergleichsweise kleinen Systeme ist die Zeit zur Verifikation bereits erheblich zu lang, um mehrere Hundert oder Tausende Systeme in dem Optimierungsprozess zu untersuchen. Aufgrund der exponentiell wachsenden Verifikationsdauer, steigt die benötigte Zeit für ein System mit 16 Task-Blöcken und 8 Ressourcen sogar auf über elf Stunden und ist damit innerhalb der Entwurfsraumexploration nicht mehr handhabbar. Grund für diesen exponentiellen Anstieg der Verifikationszeiten ist der wachsende Zustandsraum. Dieses als Zustandsexplosion bezeichnete Phänomen ist ein grundlegendes Problem bei dieser Art von Werkzeugen und bereits bekannt. Ungeachtet dessen zeigen diese Ergebnisse für das in dieser Arbeit genutzte Modell, dass diese kritische Grenze sehr schnell überschritten wird und diese Werkzeuge bezüglich der Verifikation von Systemen mit einer realistischen Komplexität, noch relativ große Einschränkungen mitbringen.

Einen Ausweg bietet die Simulation der Modelle. Diese gibt zwar keine belastbaren Schranken zurück, liefert bei einer großen Anzahl von Durchläufen aber eine gute Näherung an diese. Die in der Tabelle gesammelten Zeiten für die Simulation beschreiben die benötigte Zeit für jeweils 10.000 Durchläufe. Die dargestellten Ergebnisse zeigen, dass selbst das größte hier ausgewertete System mit unge-

[1]Der Uppaal-Verifizierer wurde mit den folgenden Parametern aufgerufen: `verifyta -WsAC`

fähr 15 Sekunden Laufzeit in akzeptabler Zeit die gewünschten Performanzdaten liefert.

Uppaal eignet sich also prinzipiell für die Bewertung der LavA-Systeme, denn diese können modelliert, simuliert und mit Einschränkungen auch verifiziert werden. Die zeitliche Performanz der Systeme kann jedenfalls für die Entwurfsraumexploration verwendet und die Systeme dementsprechend klassifiziert werden. Wenn die Verifikation für den Anwender von elementarer Bedeutung sein sollte, kann das finale System gegebenenfalls mit einem anderen Werkzeug verifiziert oder, falls die Systemgröße dies noch gestattet, auch mit Uppaal verifiziert werden. Da zu diesem Zeitpunkt nur noch das finale System verifiziert werden muss, können möglicherweise auch längere Laufzeiten akzeptiert werden.

6.4.2.3 Simulatoren

Neben Uppaal, wurde mit OMNeT++ [Var01] noch ein Werkzeug zur Auswertung der LavA-Systemperformanz untersucht, welches ausschließlich auf die Simulation ausgerichtet ist. OMNeT++ ist ein diskreter Ereignissimulator, der eine auf C++ basierende Bibliothek zur Implementierung von Netzwerk-spezifischen Simulatoren bereitstellt. OMNeT++ erlaubt die modulare Implementierung von Komponenten. Anhand einer abstrakten Beschreibung können die einzelnen Komponenten dann zu komplexen Netzen zusammengesetzt werden. Dieser Vorteil wird auch zur automatisierten Generierung der LavA-Modelle für OMNeT++ genutzt. Die Task-Blöcke und Ressourcen sind jeweils in eigenen Komponenten implementiert und werden anhand des Anwendungsmodells automatisiert über das EMF – also analog zu den Uppaal-Modellen – zu einem Gesamtsystem zusammengesetzt.

In Abbildung 6.16 ist ein Beispiel für ein in OMNeT++ modelliertes LavA-System abgebildet. Die Task-Blöcke werden hier mit Zahnradsymbolen und die Ressourcen mit einem Quadratsymbol dargestellt. Die Linien bilden die Zuteilung von Task-Blöcken zu Ressourcen ab und die Pfeile stellen die Abhängigkeiten zwischen den Task-Blöcken dar. Die Task-Blöcke 2 und 3 können beispielsweise erst dann ausgeführt werden, wenn Task-Block 1 bereits abgearbeitet wurde.

Das angewendete Schema zur Modellierung der Systeme ist sehr stark an das bereits vorgestellte Konzept des Uppaal-Modells angelehnt. Die entscheidenden Unterschiede zwischen diesen beiden Modellen bestehen darin, dass die Logik der verschiedenen Uppaal-Echtzeitautomaten, nun in Komponenten mittels C++ beschrieben wird, und zwischen den implementierten Komponenten keine Synchronisationskanäle zur Auslösung von Zustandsübergängen verwendet werden, sondern über Nachrichtenobjekte kommuniziert wird. Diese Nachrichtenobjekte lösen bei der empfangenden Komponente Ereignisse aus und ermöglichen so den schrittweisen zeitlichen Fortschritt in dem Modell. Ungeachtet dessen liefern beide Simulatoren aufgrund der analogen Modellierung der Systeme, auch vergleichbare Performanzdaten zurück.

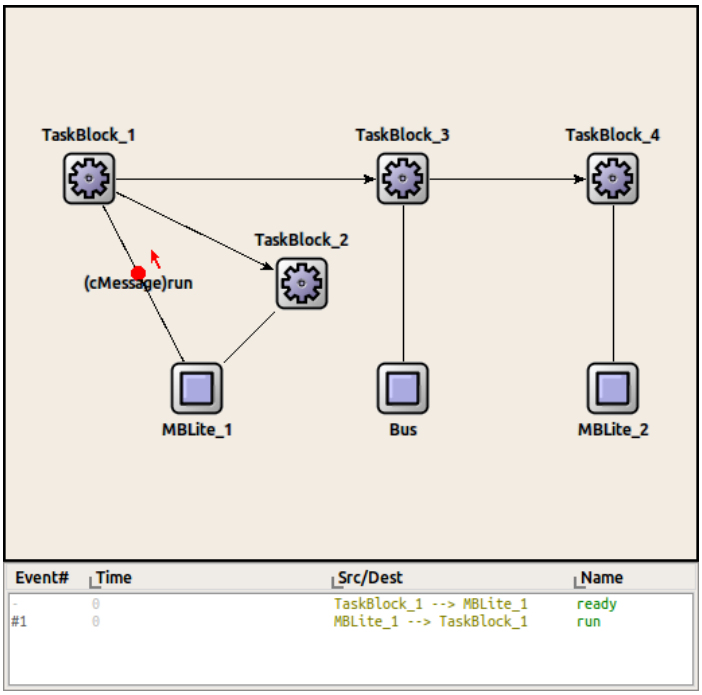


Abbildung 6.16: Simulation eines LavA-Systems in OMNeT++

Das Ereignisprotokoll unter dem Modell in Abbildung 6.16 zeigt die während der Simulation aufgetretenen Ereignisse. Neben der Ereignisnummer sind dort die verstrichene simulierte Zeit, die involvierten Komponenten und der Name des Nachrichtenobjektes aufgeführt. Das Ereignisprotokoll zeigt in diesem Fall den Systemzustand unmittelbar nach dem Start. Da der `TaskBlock_1` keine Abhängigkeiten zu anderen Task-Blöcken aufweist, setzt dieser noch während der Initialisierung das Nachrichtenobjekt `ready` ab, welches seine Ausführbereitschaft ankündigt. Da die Ressource `MBLite_1` nicht belegt ist, sendet diese unverzüglich das Nachrichtenobjekt `run` ab, um dem Task-Block die Ausführung anzuzeigen. Bisher ist in dem System keine Zeit vergangen und beide Ereignisse sind zum Zeitpunkt 0 eingegangen. Wie schon bei Uppaal, liegt die Ausführungsdauer eines Task-Blocks zwischen der BCET und WCET. Die genaue Dauer der Ausführung wird aus dieser Zeitspanne über den Zufall bestimmt und schaltet entsprechend des ermittelten Wertes die Simulationszeit weiter.
Sobald in dem System keine weiteren Ereignisse mehr auftreten, sind alle Task-Blöcke abgearbeitet und die in dem Modell simulierte Zeit kann für die Bewertung des Systems verwendet werden. Um die Performanzdaten zu verfeinern, können die Modelle, wie schon bei Uppaal, wiederholt simuliert werden.
Zur Einordnung der Laufzeiten der OMNeT++ Simulationen, sind in Tabelle 6.2 für verschiedene Systeme die Zeiten von OMNeT++ (Version 4.6) den entsprechenden Zeiten der Uppaal-Simulationen gegenübergestellt. Zur Ermittlung dieser Zeiten wurde der selbe Rechner verwendet, wie schon bei dem zeitlichen Vergleich

Task-Blöcke	Ressourcen	Uppaal (Laufzeit)	OMNeT++ (Laufzeit)
4	2	0,30 s	14,80 s
8	4	0,36 s	14,84 s
16	8	0,72 s	14,99 s
32	16	2,55 s	15,35 s
64	32	14,00 s	15,59 s
128	64	124,59 s	16,83 s

Tabelle 6.2: Laufzeitvergleich zwischen dem Uppaal-Simulator und OMNeT++

zwischen der Uppaal-Verifikation und -Simulation. Auch bei dieser Gegenüberstellung spiegeln die ermittelten Werte in der Tabelle die Laufzeiten von jeweils 10.000 Simulationsläufen wieder. Um die Performanz zu steigern, werden für beide Simulatoren die 10.000 Durchläufe auf 64 (Anzahl der auf dem Rechner parallel ausführbaren *Threads*) parallel ausgeführte Instanzen der Simulatoren aufgeteilt. Für die verschiedenen Instanzen werden jeweils eigene *Seeds* (Startwerte für die Zufallsgeneratoren) zur Verfügung gestellt, damit die parallel ausgeführten Durchläufe nicht identische Szenarien simulieren. Auch für diesen Vergleich werden die Systeme anhand der Task-Blöcke sowie Ressourcen in ihrer Größe variiert und reichen von vier Task-Blöcken auf zwei Ressourcen bis zu 128 Task-Blöcken auf 64 Ressourcen. Für OMNeT++ weichen die Zeiten für die verschiedenen Systeme nur in geringem Maße voneinander ab und selbst das größte System ist nach unter 17 Sekunden 10.000 Mal simuliert. Bei OMNeT++ hat die Größe der Systeme also einen eher untergeordneten Einfluss auf die Laufzeit, denn diese wird für die hier untersuchten Systeme im Wesentlichen durch den anfallenden Verwaltungsaufwand zur Durchführung der Simulationen bestimmt. Genau gegenteilig verhalten sich die Laufzeiten der Uppaal-Simulationen. Für Uppaal reichen die Zeiten von wenigen Zehntelsekunden bis über 2 Minuten. Insbesondere für größere Systeme mit über 64 Task-Blöcken und 32 Ressourcen lohnt sich dementsprechend der Einsatz des OMNeT++ Simulators, um die Performanzanalyse von Systemen zu beschleunigen.

6.4.3 Fazit

Nachdem in diesem Unterkapitel eine Reihe von Werkzeugen zur Performanzanalyse untersucht wurden ist festzustellen, dass es zwar einige Werkzeuge gibt, die die Verifikation solcher Systeme grundsätzlich ermöglichen, aber abhängig von dem Anwendungsgebiet, sind diese nicht immer anwendbar. Sei es auf der einen Seite aufgrund einer spezialisierten Hardware, wie dem NoC, die sich infolge der mangelnden Ausdrucksstärke nicht modellieren lässt oder auf der anderen Seite die Komplexität der Systeme, der diese Techniken noch nicht gewachsen sind und bereits für relativ kleine Systeme an ihre Grenzen stoßen. Für LavA wird deshalb innerhalb des Optimierungsprozesses auf Simulatoren gesetzt, welche auf einer hohen Abstraktionsebene arbeiten, um so innerhalb kurzer Zeitspannen zu-

mindest eine Näherung an die Systemperformanz bestimmen zu können. Wie bereits angesprochen, schließt dieses Vorgehen jedoch nicht aus, dass das letztendlich ausgewählte System einer zeitaufwendigeren Verifikation unterzogen wird, um sicherzustellen, dass die Zeiten eingehalten werden können.
Die vorgestellten Techniken zur Bestimmung der Ressourcen (Ressourcenmodelle) und Performanz (Uppaal und OMNeT++) sind vollständig automatisiert in den Optimierungsprozess eingebunden und können die ebenfalls automatisiert zusammengestellten und generierten Systeme selbstständig bewerten. Dem Anwender bleibt beim Start des LavA-Optimierungsprozesses überlassen, welches Werkzeug zur Performanzanalyse verwendet werden soll und auch, ob Uppaal simulieren oder, für kleine Systeme, verifizieren soll.
Zudem bietet der Einsatz der Simulatoren in Kombination mit den wiederholten Durchläufen die Möglichkeit, eine Abschätzung zur BCET, ACET (*Average-Case Execution Time*) und WCET zu berechnen. Diese Zeiten werden für jedes System an den EA weitergeleitet und dem Entwickler bleibt so die Wahl, welches Kriterium bzw. welche Kombination von Kriterien für die Optimierung verwendet werden sollen.

6.5 Zusammenfassung

Dieses Kapitel hat gezeigt, dass die Idee der durchgehenden, automatisierten Co-Konfiguration von Hardware- und Systemsoftware-Produktlinien auf Basis eines softwarezentrierten Analyseprozesses umgesetzt werden kann. Konfigurationsinformationen, die sich nicht direkt aus den Programmierschnittstellen ableiten lassen, werden über eine Entwurfsraumexploration eingebracht und optimiert. Die mittels des EAs bestimmten Konfigurationsinformationen werden allerdings nicht willkürlich ausgewürfelt, sondern anhand der Informationen aus der Programmierschnittstelle geleitet. So kann letztendlich ein maßgeschneidertes System für eine vom Softwareentwickler vorgegebene Anwendung generiert werden, bei dem, aufgrund des durchgehenden und über Programmierschnittstellen gesteuerten Prozesses selbst, Systemsoftware und Hardware passend aufeinander abgestimmt werden.
Am Ende des LavA-Optimierungsprozesses wird eine finale Individuumspopulation zurückgegeben. Aus dieser Menge kann der Entwickler dann, abhängig von seiner eigenen Gewichtung der jeweiligen Kriterien, ein System auswählen. Diese Entscheidung erfordert jedoch immer einen Kompromiss zwischen dem erforderlichen Ressourcenbedarf und der zur Verfügung gestellten Leistung und kann zum Beispiel anhand der gegebenen *Deadline* oder dem verfügbaren FPGA fest gemacht werden. Eine Evaluation der Ergebnisse des Optimierungsprozesses für verschiedene Anwendungen, die diesen Sachverhalt verdeutlicht, folgt in Kapitel 7.
Aufgrund des Aufbaus der vorgestellten Methodik kann ein Entwickler an verschiedenen Stellen in den LavA-Prozess einsteigen, um die gewünschten Systeme zu entwickeln. Dies liegt an dem vorgeschlagenen Konfigurationsprozess, wel-

cher die Generierung der Systeme, angefangen beim Betriebssystem bis hin zur Hardwarearchitektur, von oben nach unten vorsieht. Dabei kann sowohl der volle Umfang des LavA-Prozesses mitsamt der automatisierten Optimierung und der zur Verfügung gestellten hohen Abstraktionsebene ausgenutzt als auch auf einer tieferen Ebene eingestiegen werden. Falls zum Beispiel die Architektur des Systems und die Zuordnung der Tasks bereits spezifiziert ist oder eventuell vorgegeben wird, könnten trotzdem die Hardware- und Betriebssystem-Produktlinie verwendet werden. Dazu müsste der Entwickler dann zunächst manuell die CiAO-Instanzen über ein Werkzeug, wie Menuconfig, konfigurieren und könnte sich dann über LavA die Betriebssysteme und Hardwarestruktur generieren lassen. Auch die alleinige Generierung der Hardware mittels LavA ist möglich, wenn auf Ebene des EMF ein entsprechendes Modell der Hardware angefertigt wird. So kann abhängig vom Anwendungsfall und der Expertise des jeweiligen Entwicklers die geeignete Abstraktion und Unterstützung durch den LavA-Prozess gewählt werden.

Kapitel 7

Evaluation

Inhalt

Nachdem in dem letzten Kapitel beschrieben wurde, wie sich die einzelnen Bausteine dieser Arbeit, wie beispielsweise die Hardware- und Software-Produktlinien sowie die Extraktion von Konfigurationsinformation aus den Schnittstellen der Softwareschichten zu einem automatisierten Optimierungsprozess zusammenfügen lassen, sollen in diesem Kapitel nun die durch diesen Optimierungsprozess hervorgebrachten Ergebnisse betrachtet und ausgewertet werden.

In Unterkapitel 7.1 wird zunächst anhand einer einfachen Anwendung die Funktionsweise des Optimierungsprozesses gezeigt. Zudem wird verdeutlicht, wie der Entscheidungsprozess eines Anwendungsentwicklers auf Basis der vom Optimierungsprozess gelieferten Informationen vollzogen und dementsprechend ein System aus der Menge aller untersuchten Systeme selektiert werden kann. In dem folgenden Unterkapitel 7.2 werden dann die Ergebnisse der Ressourcenmodelle bezüglich ihrer Abweichungen von den Ergebnissen der Synthesewerkzeuge untersucht und somit nicht nur deren Eignung bezüglich der benötigten Laufzeit, sondern auch hinsichtlich der gelieferten Ergebnisqualität überprüft. In dem letzten Unterkapitel wird eine verteilte Anwendung zur Dekodierung von Audiodaten vorgestellt und die dort gewonnen Erkenntnisse diskutiert.

Sämtliche im Folgenden präsentierten Ergebnisse und die damit verbundenen Durchläufe des LavA-Optimierungsprozesses wurden auf einem Rechner mit vier Intel Xeon E5-4650 Prozessoren (jeweils acht Kerne) und 192 GB Hauptspeicher durchgeführt. Einzelne Phasen dieses Prozesses, wie das Übersetzen der Betriebssysteminstanzen oder die Simulation mittels Uppaal bzw. OMNeT++ nutzen dabei zeitweise die volle Parallelität des Rechners aus. Bei den im weiteren Verlauf dieses Kapitels präsentierten Resultaten zur Laufzeit des Optimierungsprozesses muss jedoch berücksichtigt werden, dass sich die Laufzeiten auch für ein und dieselbe Anwendung bei verschiedenen Durchläufen relativ deutlich unterscheiden

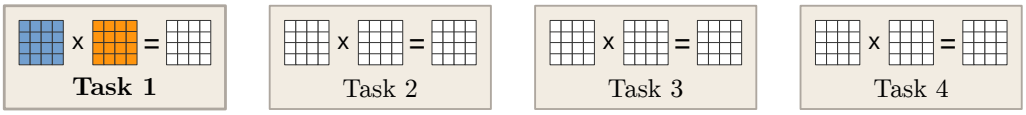
```
1 enum grp_scale {A = 4};
2 enum grp_priority {PRIO1};
3
4 typedef lava::TaskGroup<PRIO1, 1, pow<2, A>::result> Matrix;
```

Abbildung 7.1: Implementierung der `Matrix`-Gruppe

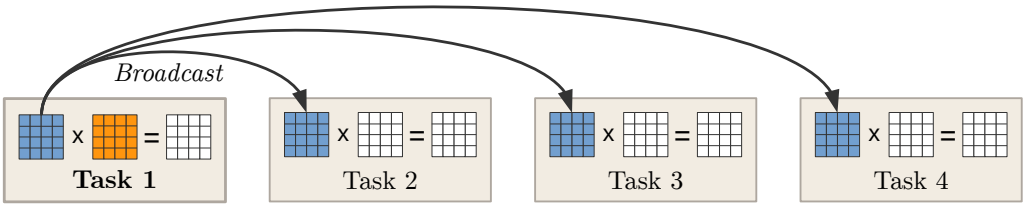
können und somit nur eine grobe Orientierung zur Einordnung der Optimierungsdauer darstellen.
Das wesentliche Kriterium für die Auswertungsdauer einer Systemkonfiguration ist die Anzahl der enthaltenen Prozessoren und damit einhergehend die Anzahl der zu untersuchenden und zu generierenden Betriebssysteminstanzen. Da diese Anzahl von dem EA über den Zufall gesteuert wird, kann es bei einem Optimierungslauf mit vergleichsweise vielen großen Systemen eine entsprechende Differenz bezüglich der Laufzeit im Vergleich zu anderen Optimierungsläufen geben.
Die Anzahl der untersuchten Systeme bzw. Individuen für die im Folgenden präsentierten Optimierungen ist konstant. Die Startpopulation besteht dabei immer aus exakt 50 Individuen. In den folgenden neun Generationen werden jeweils 100 neue Individuen generiert, so dass insgesamt 950 Systeme in einem Optimierungslauf untersucht werden. Diese Anzahl ist für einen Evolutionären Algorithmus noch relativ gering, aber für die hier untersuchten Anwendungen ausreichend, da bei den späteren Generationen kaum noch nennenswerte Verbesserungen bezüglich der Eigenschaften der Systeme festgestellt werden konnten.

7.1 Bewertung des Optimierungsprozesses

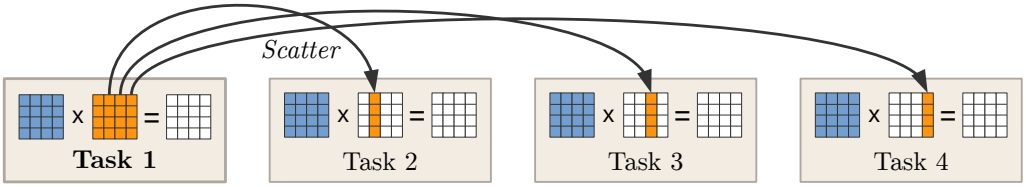
Die Arbeitsweise des LavA-Optimierungsprozesses wird im Folgenden anhand einer Anwendung zur Matrixmultiplikation vorgestellt. Diese Anwendung soll unter Verwendung des parallelen Programmiermodells kommunizieren und die Ergebnismatrix parallel auf mehreren Knoten berechnen. Die zu multiplizierenden Matrizen haben eine Spalten- und Zeilenzahl von jeweils 16. Der implementierte Algorithmus kann zur Berechnung der Ergebnismatrix mit Gruppengrößen von Zweierpotenzen mit bis zu 16 Tasks umgehen. Der Optimierungsprozess soll dementsprechend die Gruppengröße zur Berechnung der Ergebnismatrix aus der Menge $M_1 = \{2, 4, 8, 16\}$ auswählen können. Damit von dem Optimierungsprozess nur Gruppengrößen aus dieser Menge ausgewählt werden können, wird zur Spezifizierung der `Matrix`-Gruppe das Hilfs-*Template* `pow` eingesetzt (siehe Abbildung 7.1). Dieses *Template* kann die Zweierpotenzen statisch anhand der symbolischen Konstante `A` berechnen, wodurch sichergestellt ist, dass die gewählte Gruppengröße zur Übersetzungszeit ausgewertet werden kann. Der C++ *Template*-Mechanismus kann so für beliebige Anwendungen verwendet werden, um die jeweiligen Algorithmen und die zufällig bestimmten Gruppengrößen aufeinander abzustimmen.



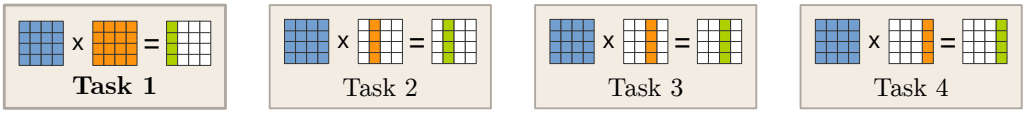
(a) die zu multiplizierenden Matrizen liegen beim *Master*-Task



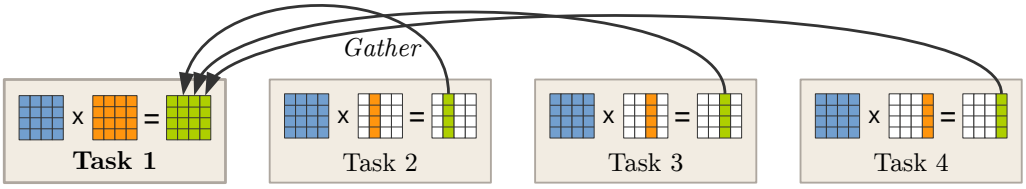
(b) die erste Matrix wird per *Broadcast* verteilt



(c) die zweite Matrix wird in Fragmenten per *Scatter* bereitgestellt



(d) parallele Berechnung der einzenen Teile der Ergebnismatrix



(e) Zusammenführung der Teilergebnisse mittels *Gather*

Abbildung 7.2: Funktionsweise der parallelen Anwendung zur Matrixmultiplikation

Die Funktionsweise der Anwendung zur Matrixmultiplikation zeigt die Abbildung 7.2 exemplarisch. Die Ausgangssituation für diese Anwendung wird in Abbildung 7.2 (a) gezeigt. Die beiden zu multiplizierenden Matrizen liegen bei dem *Master*-Task (Task 1). Die weiteren drei Tasks müssen demnach zunächst mit den entsprechenden Daten versorgt werden. In Abbildung 7.2 (b) ist der nächste Schritt im Ablauf dieser Anwendung dargestellt. Hier wird die erste Matrix (blau)

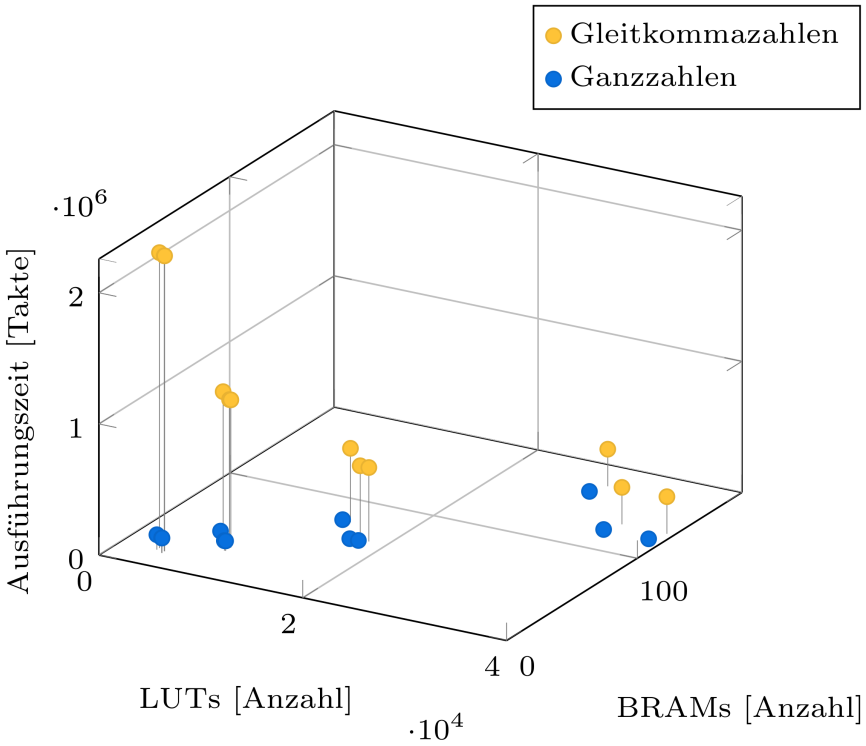


Abbildung 7.3: Bewertung der Individuen zur Matrixmultiplikation

von dem *Master*-Task vollständig mittels der `broadcast`-Operation übertragen. In dem nächsten Schritt wird dann die zweite Matrix (orange) verteilt. Aufgrund der Eigenschaften des umgesetzten Algorithmus, werden von dieser Matrix nur bestimmte Spalten für die jeweiligen Tasks benötigt. Diese Eigenschaften werden hier ausgenutzt und nur die jeweils notwendigen Fragmente der Matrix mittels der `Scatter`-Operation kommuniziert. So kann die Belegung der verwendeten Kommunikationsstruktur auf ein Minimum reduziert werden. Nachdem die notwendigen Daten übertragen wurden, können die Tasks die verschiedenen Teile der Ergebnismatrix berechnen (Abbildung 7.2 (d)). In dem letzten Schritt in Abbildung 7.2 (e) werden die berechneten Teilergebnisse über die `Gather`-Operation durch den *Master*-Task wieder zusammengeführt.

Die Ergebnisse in Abbildung 7.3 zeigen die Bewertung für zwei verschiedene Varianten der vorgestellten Matrixmultiplikation. In der ersten Variante (gelbe Punkte) enthalten die Matrizen Gleitkommazahlen und in der zweiten Variante (blaue Punkte) Ganzzahlen. Diese Varianten wurden in zwei separaten LavA-Optimierungsprozessen untersucht und in der Abbildung gegenüber gestellt. Neben den verschiedenen Datentypen unterscheiden sich die Anwendungsvarianten insbesondere bezüglich der benötigten Rechenzeit zur Berechnung der Matrixfragmente, weshalb jeweils andere Werte für die WCET bzw. BCET annotiert sind. Um diese Varianten bezüglich der erzeugten Last deutlicher von-

einander abzugrenzen, wird die Berechnung der Ergebnismatrix für die Variante mit Gleitkommazahlen in Software durchgeführt und keine zusätzliche Hardware-Gleitkommaeinheit in die Prozessoren integriert.

Die Achsen des Diagramms zeigen jeweils eines der drei Optimierungskriterien. Auf der x-Achse ist die Anzahl der LUTs dargestellt, auf der y-Achse ist die zeitliche Charakteristik der Systeme in Form der Gesamtausführungszeit der Anwendung auf Basis der ermittelten WCET abgebildet und auf der z-Achse ist die Anzahl der BRAMs abzulesen. Jeder der gezeigten Punkte repräsentiert die Eigenschaften eines Systems, welches innerhalb einer der beiden Entwurfsraumexplorationen untersucht wurde. Bei diesen sehr einfachen Anwendungsvarianten sind die einzelnen Systeme noch sehr einfach zu identifizieren, wodurch sich die vorzufindenden Effekte sehr gut an diesem Beispiel erläutern lassen. Für jede Anwendungsvariante gruppieren sich jeweils drei Systeme in vier verschiedenen Clustern. Innerhalb dieser Gruppierungen besitzen alle Systeme die gleiche Anzahl an Prozessoren. Die Systeme der Gruppierung mit der längsten Ausführungszeit, oben links in der Abbildung, besitzen alle genau zwei Prozessoren. Bei den nach rechts hin folgenden Gruppierungen der Gleitkommavariante (gelbe Gruppierungen) ist zu erkennen, dass sowohl die Ausführungszeit abnimmt als auch die benötigten Ressourcen für diese Gruppen ansteigen. Bei jeder dieser Gruppierungen steigt also die Prozessorenanzahl, wodurch sich eine bessere Parallelität der Anwendung und damit auch eine niedrigere Ausführungszeit ergibt. Das gleiche Verhalten ist auch für die Variante mit Ganzzahlen zu beobachten, aber aufgrund der wesentlich geringeren Ausführungszeit nur schwer in dem Diagramm abzulesen.

Die unterschiedlichen Ausprägungen der Systeme innerhalb einer Gruppierung werden durch die verschiedenen Kommunikationsstrukturen in diesen Systemen ausgelöst. Besonders deutlich sind diese Unterschiede für die Gruppen mit einem höheren Ressourcenbedarf sichtbar. Hier liegen die Systeme von einer Gruppierung am weitesten auseinander, denn je mehr Prozessoren in einem System integriert sind, desto offensichtlicher wirken sich die unterschiedlichen Implementierungsansätze der drei eingesetzten Kommunikationsstrukturen auf die Ressourcen aus. Das System, welches über gemeinsame Speicher verknüpft ist, hat, im Vergleich zu den anderen beiden Systemen einer Gruppierung, einen erhöhten Bedarf an BRAMs, benötigt dafür allerdings kaum zusätzliche Logik in Form von LUTs. Die beiden anderen Systeme, mit Bus bzw. NoC, werden hingegen vollständig in LUTs umgesetzt. Den höchsten Bedarf an LUTs haben die Systeme mit einem integrierten NoC.

Die tatsächliche Laufzeit zur Optimierung der beiden Anwendungsvarianten beträgt für die Variante zur Multiplikation von Gleitkommazahlen 15 Stunden sowie für die Ganzzahlen-Variante 15 Stunden und 42 Minuten. Die aufgetretene Differenz ist, wie schon zu Beginn dieses Kapitels erläutert, auf die Größe der untersuchten Systeme zurückzuführen und damit vom Zufall bei der Bestimmung der Individuumseigenschaften abhängig.

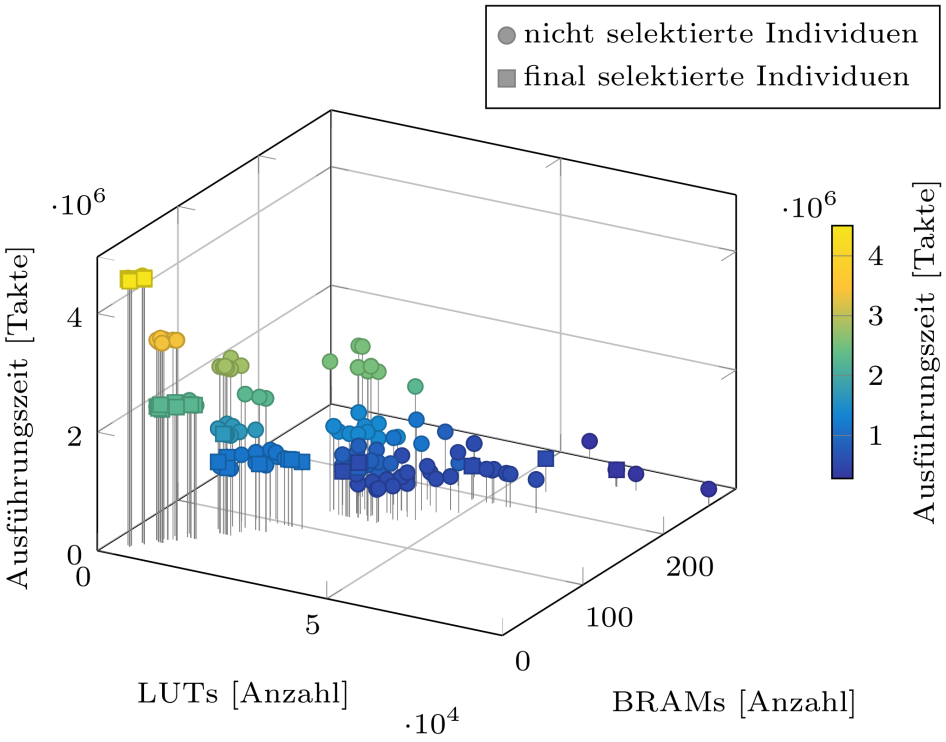


Abbildung 7.4: Bewertung der Individuen zur Matrixmultiplikation mit zwei Gruppen

Die in Abbildung 7.3 gezeigten Ergebnisse hätten noch nicht zwingend eine aufwendige Entwurfsraumexploration erfordert und könnten, bei letztendlich nur zwölf verschiedenen Systemen für jede Anwendungsvariante, sicherlich auch „per Hand“ ermittelt und ausgewählt werden. Jedoch demonstriert dieses Beispiel sehr schön die generelle Funktionsweise des LavA-Optimierungsprozesses und zeigt, dass dieser Ansatz plausible Ergebnisse liefert. Der entscheidende Grund für die wenigen ausgewerteten Systeme ist, dass dem EA bei dieser Anwendung kein Spielraum für die Zuordnung von Tasks zu Prozessoren eingeräumt wird, denn wie in Unterkapitel 6.1.1 beschrieben, werden alle Tasks einer Gruppe auf dedizierten Prozessoren untergebracht, wodurch der Optimierung diese Variabilität genommen wird. Ein viel größerer Spielraum wird dem Optimierungsprozess geboten, wenn zwei oder mehr Gruppen in der Anwendung vorhanden sind und demzufolge mehrere Tasks auf einem Prozessor platziert werden können. Die Ergebnisse des Optimierungsprozesses für solch eine Anwendung sind in Abbildung 7.4 abgebildet.

In dieser Anwendung befinden sich zwei voneinander unabhängige Gruppen. Jede dieser Gruppen multipliziert jeweils für sich zwei Gleitkommamatrizen parallel auf einem System. Die Systemvielfalt nimmt für diese Anwendung bereits deutlich zu und zeigt die sich abzeichnende Komplexität auf, obwohl diese Anwendung

noch immer eine relativ einfache Struktur aufweist und zum Beispiel nicht gruppenübergreifend kommuniziert.

Die final von dem EA selektierten Individuen bzw. Systeme, sind in der Abbildung anhand von Quadraten dargestellt. Demgegenüber werden die nicht selektierten Systeme mittels kreisförmiger Markierungen repräsentiert. Zur Einordnung der Position im dreidimensionalen Raum kann entweder die Hilfslinie zu den verschiedenen Markierungen oder die Farbskala am rechten Rand der Abbildung verwendet werden. Die Systeme mit einem Blauton haben eine eher niedrige Ausführungszeit, wohingegen die Punkte mit Gelbtönen, Systeme mit einer hohen Ausführungszeit repräsentieren.

Die final von dem EA selektierten Systeme bilden prinzipiell die Pareto-Menge. Jedes dieser Systeme ist also bezüglich der untersuchten Lösungen pareto-optimal und wird dementsprechend nicht von einem anderen System hinsichtlich der verschiedenen Kriterien dominiert. Eine Lösung wird dabei genau dann dominiert, wenn es eine andere Lösung gibt, die im Hinblick auf ein Kriterium echt besser ist und bezüglich der anderen Kriterien nicht schlechter ist. Da die Anzahl der final selektierten Systeme allerdings parametrisierbar ist und SPEA2 genau diese Anzahl an Systemen zurückliefert, ergänzt SPEA2 diese Menge zusätzlich um die besten dominierten Systeme, falls zu wenige nichtdominierte Systeme vorhanden sind oder übernimmt nicht alle nichtdominierten Systeme in diese Menge, falls zu viele dieser Systeme gefunden wurden.

Bei der letztendlichen Auswahl eines Systems aus dieser Menge muss, bei den hier vorzufindenden, untereinander konkurrieren Kriterien, ein Kompromiss zwischen der zu erwartenden Performanz und dem Ressourcenbedarf eingegangen werden. Die Entscheidung, welches System aus der Pareto-Menge am Ende eingesetzt werden soll, muss letztlich der Entwickler treffen. Dabei kann sich der Entwickler von seiner eigenen Gewichtung der Kriterien leiten lassen. Die grundlegende Entscheidung, welche Systeme überhaupt relevant sind, ist an diesem Punkt allerdings anhand verschiedener Indikatoren relativ einfach zu fällen. So können zum Beispiel alle Systeme, welche die *Deadline* nicht einhalten oder nicht auf das ausgewählte FPGA passen, ignoriert werden. In diesem Fall würden nur noch die Systeme übrig bleiben, welche zentral auf der Pareto-Front liegen und die Systeme an den Rändern wären an den entsprechenden Stellen abgeschnitten.

Der Optimierungsprozess für diese Anwendung schlägt mit einer Laufzeit von fast 34 Stunden zu Buche und hat damit – im Vergleich zu den beiden zuvor präsentierten Optimierungsprozessen – etwas mehr als die doppelte Laufzeit eingenommen. Wie schon vorab erläutert, liegt dies an den nun potenziell wesentlich größeren Systemen, denn es können nun Systeme mit bis zu 32 Prozessoren entstehen, falls sich die beiden Gruppen auf den Prozessoren des Systems nicht überlappen und diese jeweils die maximale Anzahl an Tasks zugewiesen bekommen.

7.2 Bewertung der Ressourcenmodelle

In diesem Unterkapitel soll nun untersucht werden, ob die im Zuge des LavA-Optimierungsprozesses verwendeten Ressourcenmodelle für die Xilinx FPGA-Familien Spartan-3E und Virtex-5 ihrer Aufgabe gerecht werden und akkurate Ergebnisse zur unmittelbaren Abschätzung des Ressourcenbedarfs liefern können. Zur Bewertung der Ressourcenmodelle wurden die ermittelten Hardwarestrukturen aus dem LavA-Optimierungsprozess zur Ganzzahl-Matrixmultiplikation aus Unterkapitel 7.1 synthetisiert und der so ermittelte Ressourcenbedarf mit den gelieferten Werten aus den Ressourcenmodellen verglichen. Dieser Vergleich ist in dem Balkendiagramm von Abbildung 7.5 dargestellt.

Auf der x-Achse sind die zwölf verschiedenen Systeme zur Ganzzahl-Matrixmultiplikation (Systeme aus Abbildung 7.3) aufgetragen. Die Zahlenwerte stehen abermals für die Anzahl der MB-Lite+ Prozessoren in den Systemen und darauf folgend ist die eingesetzte Kommunikationsstruktur angegeben. Die zwölf verschiedenen Systeme wurden für ein Xilinx Virtex-5 XC5VLX110T FPGA synthetisiert. Die über Synthesen ermittelten LUTs sind mittels der blauen Balken eingezeichnet und die benötigten BRAMs anhand von roten Balken. Die zugehörigen abgeschätzten Werte auf Basis der Ressourcenmodelle sind über die schwarzen Querlinien eingezeichnet, so dass der Grad der Abweichung zwischen Modell und Synthese direkt identifiziert werden kann. Auf der y-Achse ist sowohl die Anzahl der LUTs als auch der BRAMs aufgetragen. Zur Markierung der maximal verfügbaren BRAMs auf diesem FPGA, ist eine gestrichelte Linie in das Balkendiagramm eingezeichnet. Die Gesamtzahl der LUTs auf diesem FPGA liegt mit 69.120 LUTs jenseits des dargestellten Bereiches.

Die auftretende Abweichung über alle Systeme bezüglich der LUTs beträgt im arithmetischen Mittel ca. 2,22 % (Median: ca. 1,41 %). Der über das Ressourcenmodell abgeschätzte Bedarf an LUTs trifft die Syntheseergebnisse für die untersuchten Systeme grundsätzlich also recht genau und weicht insgesamt nur mit geringen Differenzen in beide Richtungen ab. Die einzige Ausnahme bildet das System mit 16 MB-Lite+ Prozessoren und den gemeinsamen Speichern zur Kommunikation, denn hier liegt das Ressourcenmodell um ca. 8 % über den Werten der Synthese. Warum hier eine recht deutliche Abweichung vorliegt ist aufgrund des proprietären Synthesewerkzeugs kaum zu ermitteln und zudem auch verwunderlich, da die hier verwendete Kommunikationsstruktur nahezu ausschließlich in BRAMs implementiert wird. Außerdem ist es auffällig, dass insbesondere für die drei anderen Systeme mit gemeinsamen Speichern die Abschätzung der LUTs sehr nahe an den Syntheseergebnissen liegt. So wird der erforderliche Bedarf an LUTs für das System mit zwei Prozessoren exakt vorhergesagt. Bei dem System mit vier Prozessoren liegt das Ressourcenmodell um ca. 0,5 % über den Werten der Synthese und bei dem System mit acht Prozessoren beträgt die Abweichung nur 0,1 %. Deshalb ist nicht davon auszugehen, dass ein generelles Problem bei dem Ressourcenmodell für Systeme mit gemeinsamem Speicher vorliegt, sondern es sich in diesem Fall eher um eine Anomalie seitens der Synthese handelt.

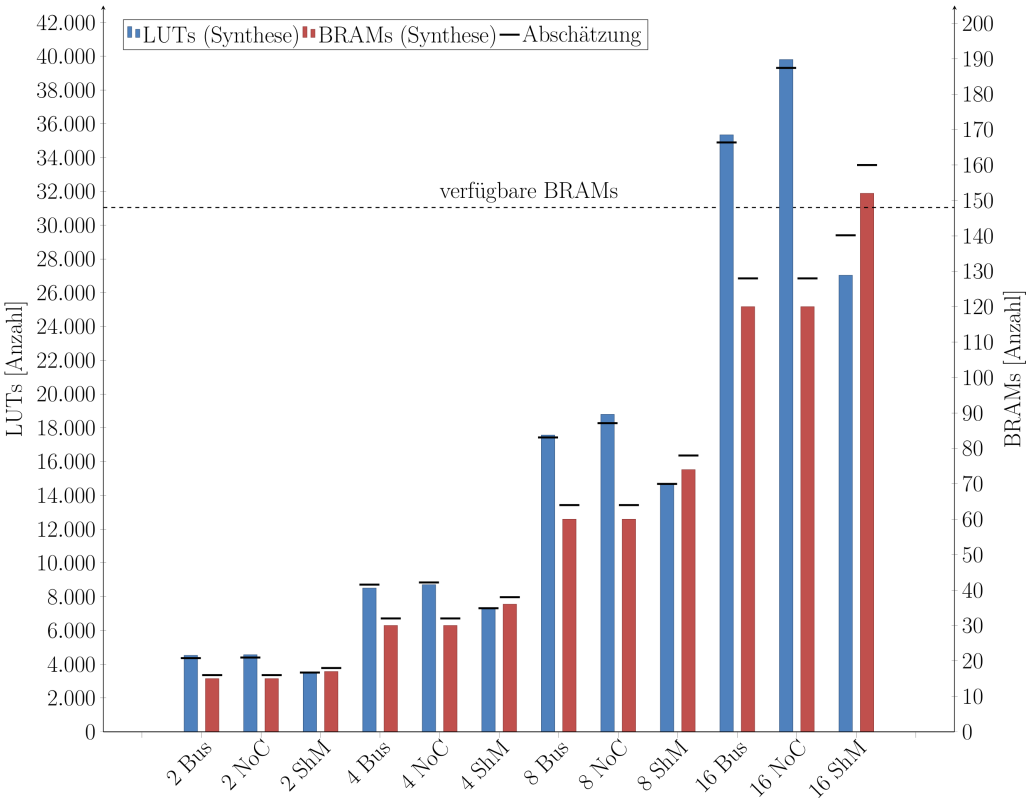


Abbildung 7.5: Abschätzung der FPGA-Ressourcen anhand von Modellen im Vergleich zu Ergebnissen der Hardwaresynthese (Xilinx Virtex-5 XC5VLX110T)

Für die BRAMs liegt die Abweichung zwischen den Ergebnissen der Ressourcenmodelle und der Synthese über alle Systeme im arithmetischen Mittel bei ca. 6,29 % (Median: ca. 6,67 %). Der Bedarf an BRAMs wird dabei über die Ressourcenmodelle durchgängig eher pessimistisch abgeschätzt. So wird zwar verhindert, dass ein letztendlich synthetisiertes System im Nachhinein mehr BRAMs erfordert als zunächst erwartet, aber speziell bei größeren Systemen macht sich die Abweichung zwischen dem erwarteten und dem tatsächlichen Bedarf durchaus bemerkbar und könnte einen Entwickler unter Umständen dazu veranlassen ein System nicht zu berücksichtigen, obwohl es möglicherweise sehr gute Eigenschaften für die anvisierten Ziele mitbringt. Ein Grund für diese Abweichung ist die Umsetzung der Registerbank für die beiden Varianten des MB-Lite Prozessors, welche BRAMs zum Speichern der Registerinhalte verwenden. Damit ein Prozessor in einem einzigen Taktzyklus auf alle für eine Operation benötigten Daten zeitgleich zugreifen kann, werden die Registerinhalte dreifach in jeweils eigenen kleinen Speichern abgelegt. Da ein BRAM der Virtex-5 Familie jedoch auch in zwei unabhängige Speicherblöcke der jeweils halben Größe geteilt werden kann und eine Instanz der Registerbank nur wenig Speicher benötigt, können zwei der redundanten Instanzen einer Registerbank auch in einem BRAM aufgenommen

```
1 enum grp_scale {A = 5};
2 enum grp_priority {PRIO1, PRIO2};
3
4 typedef lava::TaskGroup<PRIO1, 1, 1> Stream;
5 typedef lava::TaskGroup<PRIO2, Stream::NEXT,
6     pow<2, A>::result> Decode;
```

Abbildung 7.6: Implementierung der Gruppen des Audiodekoders

werden. Falls dieses Vorgehen langfristig auch für weitere Systeme erkannt wird, könnte das Ressourcenmodell in dieser Hinsicht noch optimiert werden, um akkuratere Ergebnisse zu generieren.
Trotz der einen oder anderen Abweichung, liefern die Ressourcenmodelle für Systeme der LavA Hardware-Produktlinie einen sehr guten Indikator für den Ressourcenbedarf. Insbesondere bei Systemen nahe der Kapazitätsgrenze eines FPGAs sollte sich der Entwickler jedoch bewusst sein, dass Abweichungen bis zu einem gewissen Grad auftreten können. Da der LavA-Optimierungsprozess allerdings eine ganze Menge von pareto-optimalen Systemen vorschlägt, kann der Entwickler ohne weiteres ein System mit einem etwas geringeren Ressourcenbedarf und ähnlichen zeitlichen Eigenschaften auswählen.

7.3 Anwendungsbeispiel: Audiodekoder

Nachdem in einem der vorangegangenen Unterkapitel bereits die grundlegende Funktionsweise des LavA-Optimierungsprozesses anhand einer einfachen Anwendung zur Matrixmultiplikation vorgestellt wurde, sollen in diesem Unterkapitel nun die Ergebnisse für eine komplexere Anwendung zur Dekodierung von Dolby Digital Audiodaten folgen. Dolby Digital (ATSC/A52) ist ein Standard zur Kompression von bis zu sechs Audiokanälen. Der Audiodatenstrom von Dolby Digital unterteilt sich in einzelne Synchronisations-*Frames*, welche die Audioinformationen für die sechs Audiokanäle beinhalten. Ein Synchronisations-*Frame* ist die kleinste unabhängig dekodierbare Einheit des Audiodatenstroms, so dass sich mehrere Synchronisations-*Frames* in einem System parallel dekodieren lassen. Die hier zu optimierende Anwendung soll 32 der Synchronisations-*Frames* parallel in ein unkomprimiertes Stereosignal wandeln. Die Anwendung gliedert sich dazu in zwei Gruppen, welche in Abbildung 7.6 spezifiziert werden. Die Gruppe `Stream` besteht dabei grundsätzlich aus genau einem Task und übernimmt die Verteilung des Audiodatenstroms in Form der verschiedenen Synchronisations-*Frames*. Die Tasks der zweiten Gruppe dekodieren die übertragenen *Frames* parallel und weisen einen hohen Rechenaufwand auf. Dementsprechend wird diese Gruppe von der Optimierung skaliert und kann eine Gruppengröße aus der Menge $M_2 = \{2, 4, 8, 16, 32\}$ annehmen. Zur Dekodierung der Audiodaten konnte die ursprünglich für die x86-Architektur entwickelte Standardbibliothek liba52[1] mit

[1] liba52: `http://liba52.sourceforge.net/`

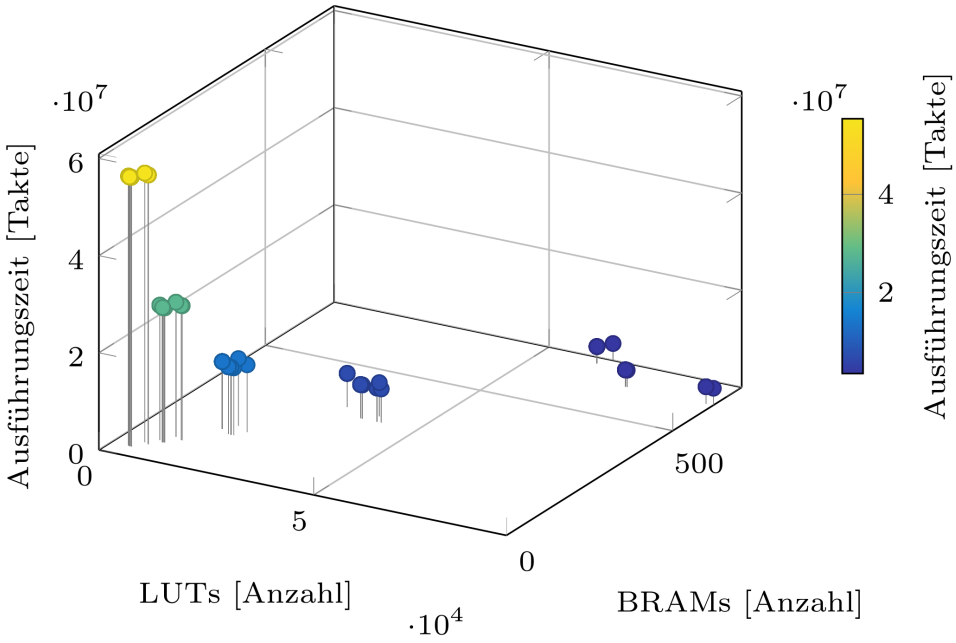


Abbildung 7.7: Bewertung der Individuen zum Audiodekoder

wenigen Änderungen wiederverwendet und so eine aufwendige Neuimplementierung vermieden werden. Dies spiegelt zudem eine der Stärken des LavA-Ansatzes wieder, denn obwohl die Hardwarestrukturen gezielt für ein gegebenes Problem maßgeschneidert werden und damit spezialisierte Systeme entstehen, werden Vielzweckprozessoren eingesetzt, welche die prinzipielle Wiederverwendung von *Legacy Code* ermöglichen.

In Abbildung 7.7 sind die Ergebnisse des LavA-Optimierungsprozesses für den Audiodekoder dargestellt. Wie schon bei der Anwendung zur Matrixmultiplikation, lassen sich die ausgewerteten Systeme auch hier in Gruppen mit ähnlichen Ressourcen- und Performanzeigenschaften einteilen. Diese Anhäufungen von Systemen werden, wie schon bei der Anwendung zur Matrixmultiplikation, maßgeblich von der Skalierung der Gruppengröße beeinflusst. Demnach hat die Variierung der Gruppengröße auch für diese Anwendung den größten Einfluss auf die zu erreichende Performanz sowie die benötigten Ressourcen, wodurch sich die einzelnen Gruppierungen eindeutig den Zweierpotenzen aus der Menge M_2 zuordnen lassen. Im Gegensatz zu der Anwendung zur Matrixmultiplikation befinden sich in diesen Ansammlungen jedoch sechs verschiedene Systeme und nicht nur drei. Die Verdopplung hängt in diesem Fall mit den nun entstehenden Möglichkeiten zur Platzierung des einen Tasks aus der Gruppe `Stream` zusammen, denn dieser kann zusammen mit einem `Decode`-Task auf einem Prozessor untergebracht werden oder einen eigenen Prozessor zugewiesen bekommen. In Kombination mit den verschiedenen Kommunikationsstrukturen entstehen so die sechs verschiedenen Systeme für jede Gruppengröße. Durch die eindeutige Zuordbarkeit der einzelnen

Konfigurationen zu den Gruppierungen ist gleichzeitig sehr leicht zu erkennen, dass sich die Platzierung des `Stream`-Tasks auf einem eigenen Prozessor, aufgrund des hohen aufzubringenden Rechenaufwands für die Dekodierung, kaum in der Performanz eines Systems niederschlägt und dementsprechend hauptsächlich einen höheren Ressourcenbedarf verursacht.

Obwohl die Komplexität des Anwendungsquelltextes für dieses Beispiel mit der Integration der liba52 stark angestiegen ist, bleibt der Entwurfsraumexploration auch hier nur wenig Spielraum zur Bestimmung von zahlreichen verschiedenartigen Konfigurationen. Es gibt zwar in dieser Anwendung nun zwei verschiedene Gruppen, da die Gruppengröße allerdings nur für eine dieser Gruppen durch die Entwurfsraumexploration aktiv variiert und weiterhin nur eine Kommunikationsstruktur für diese Anwendung bestimmt werden kann, bleiben die Möglichkeiten limitiert. Der Ansatz der hinter LavA steckt, kann seine volle Leistungsfähigkeit demzufolge erst dann ausspielen, wenn verschiedene Funktionalitäten in einem System integriert werden sollen und damit auch der händische Entwurf der Systeme immer schwieriger werden würde. Da die Entwicklung solcher Anwendungen – mit einem zugleich praxisnahen Hintergrund – extrem viel Aufwand erfordert und in diesem Rahmen nicht geleistet werden kann, werden in Abbildung 7.8 die Ergebnisse zu einer Anwendung präsentiert, welche die Funktionalität des Audiodekoders und der Matrixmultiplikation auf einem System vereint. So kann das Hauptaugenmerk auf den eigentlichen Fokus dieser Arbeit gerichtet werden und zwar den LavA-Prozess selbst und die Beleuchtung der daraus hervorgebrachten Ergebnisse für Anwendungen mit wachsender Komplexität.

Dieses Anwendungsszenario ermöglicht der Entwurfsraumexploration nun drei verschiedene Gruppen auf einem System zu platzieren, die über bis zu zwei Kommunikationsstrukturen miteinander vernetzt sein können. Schon bei einem flüchtigen Blick auf die Abbildung ist direkt ersichtlich, dass sich der zur Verfügung stehende Entwurfsraum für diese Anwendung deutlich vergrößert hat und in Folge dessen ein größerer Spielraum für die Systemkonfigurierung vorhanden ist. Damit in dieser Menge von Systemen die final von dem EA selektierten Systeme zu identifizieren sind, werden diese auch in dieser Darstellung mittels Quadraten repräsentiert. Anhand der so markierten Pareto-Front, ist auch für dieses Anwendungsbeispiel der letztendlich einzugehende Kompromiss zwischen der besseren Performanz (Systeme in Blautönen) oder einem geringeren Ressourcenbedarf (Systeme in Gelbtönen) zu erkennen.

Aufgrund der jeweils unterschiedlich ausgeprägten Anforderungen der beiden integrierten Funktionen an die erforderliche Rechenzeit, ist in diesem Diagramm ein interessanter Effekt zu identifizieren. Besonders deutlich wird dieser Effekt für Systeme mit gelb gefärbten Markierungen, denn scheinbar verbessert sich die Performanz für diese Systeme trotz ansteigender Ressourcen nur marginal. Die gelb markierten Systeme haben allesamt gemeinsam, dass für die wesentlich rechenzeitintensivere Audiodekodierung genau zwei Tasks generiert werden. Die abhängig von der weiteren Systemkonfiguration eventuell entstehenden Ressourcen, werden bei diesen Systemen allein für die Skalierung der Task-Gruppe

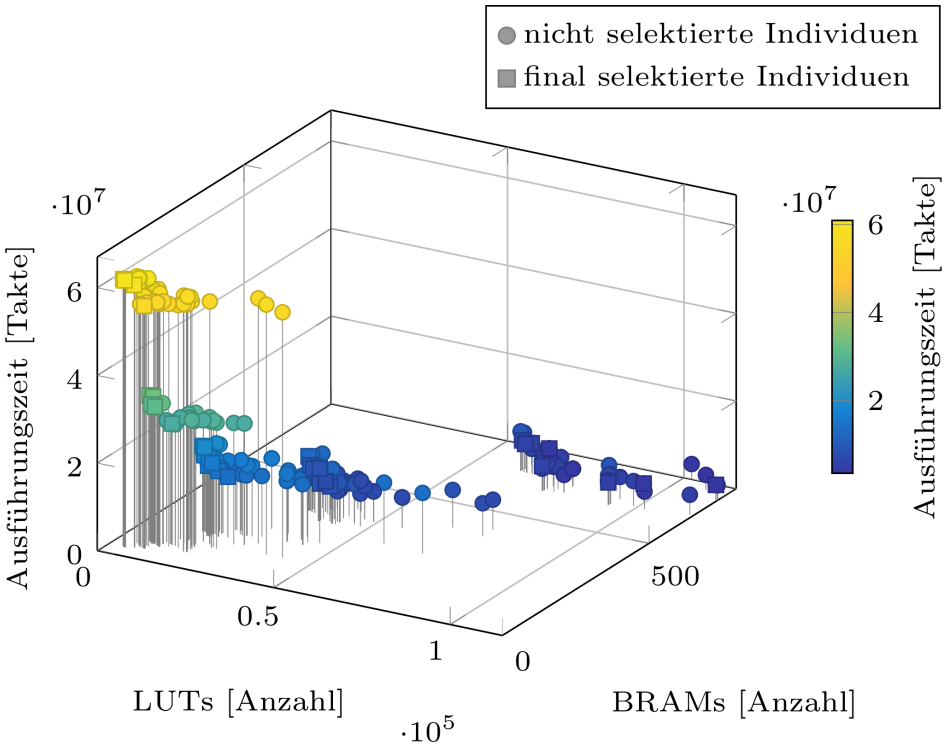


Abbildung 7.8: Bewertung der Individuen zum Audiodekoder mit Matrixmultiplikation

zur Matrixmultiplikation verwendet. Da die Matrixmultiplikation jedoch deutlich weniger Rechenzeit benötigt, wirkt sich die Skalierung dieser Gruppe kaum auf eine verringerte Gesamtausführungszeit der Systeme aus. Wegen des schlechten Verhältnisses zwischen der Performanz und den Ressourcen, gehören demzufolge die weiter rechts liegenden Systeme auch nicht zur final selektierten Population.

Einzig das System mit einem leicht dunkler gefärbten gelben Quadrat sticht im Vergleich zu den anderen gelbgefärbten Systemen bezüglich der Performanz etwas hervor. Bei diesem System wird für die Gruppe zur Matrixmultiplikation die minimale Task-Anzahl konfiguriert und dementsprechend werden nur wenige Ressourcen für diese weniger rechenintensive Funktion aufgewendet. Zudem sind die beiden Funktionen auf separaten Prozessoren untergebracht, was zwar zwei zusätzliche Prozessoren erfordert, aber zugleich auch die beiden Prozessoren zur Dekodierung der Audiodaten entlastet und dadurch eine etwas bessere Performanz zeigt. Damit erreicht dieses System, wie zu erwarten, eine vergleichbare Performanz, wie die Systeme mit zwei `Decode`-Tasks aus Abbildung 7.7.

Bei diesem komplexen Anwendungsszenario können Systeme mit 2 bis 49 Prozessoren und mit bis zu zwei Kommunikationsstrukturen generiert werden. Aufgrund der Vielzahl an potenziellen Ressourcen steigt auch die Dauer für die Untersu-

chung der 950 betrachteten Systeme durch den LavA-Optimierungsprozess auf gemessene 58 Stunden und 28 Minuten.

7.4 Zusammenfassung

Dieses Kapitel hat gezeigt, dass der LavA-Optimierungsprozess plausible Ergebnisse liefert und die Optimierungslaufzeit auch für größere Systeme in einem annehmbaren Rahmen liegt. Insbesondere wenn berücksichtigt wird, dass nicht nur nach einer optimierten Systemkonfiguration gesucht wird, sondern am Ende dieses Prozesses ein vollständig abgestimmtes System zur Verfügung steht, bei dem in einem letzten Schritt lediglich ein Synthese-Werkzeug angestoßen werden muss, bevor das Gesamtsystem aus Anwendung, Betriebssystem und Hardware eingesetzt werden kann.
Des Weiteren wurde mit den hier vorgestellten Ergebnissen gezeigt, dass das Zusammenspiel aus der Analyse der Programmierschnittstellen, den Produktlinien und dem Optimierungsprozess konstruktiv eingesetzt werden kann, um – ohne fundiertes Wissen in den Bereichen der Hardwarearchitektur oder der Betriebssysteme – maßgeschneiderte Systeme generieren zu können.
Allerdings wurde ebenso deutlich, dass die Restriktionen bezüglich der Zuteilung von Gruppen zu Prozessoren, dem Optimierungsprozess etwas an Variabilität nimmt. Dieses Problem tritt allerdings in erster Linie für kleinere Anwendungen auf, welche nicht im eigentlichen Fokus von LavA liegen, denn LavA sucht insbesondere nach Lösungen im Kontext von *Manycore*-Systemen und unterscheidet sich damit unter anderem von den bisherigen Arbeiten in diesem Forschungszweig. Zudem ist nicht gesagt, dass eine Aufhebung dieser Restriktionen zugleich zu besseren Systemen führt, denn der LavA-Prozess sieht vor, dass die Performanz der Anwendungen über die Skalierung der Gruppen angepasst werden kann. Falls letztendlich also doch alle Tasks einer Gruppe auf demselben Prozessor platziert werden könnten, ist über diesen Mechanismus keine Justierung der Performanz mehr möglich und lediglich der *Overhead* für die nun anfallende Kommunikation wird sichtbar. So würde zwar der zu untersuchende Entwurfsraum vergrößert, aber die Qualität der Systeme nicht weiter gesteigert.
Zusätzlich wurde in diesem Kapitel gezeigt, dass die entwickelten Ressourcenmodelle für die Mehrheit der untersuchten Systeme eine sehr genaue Abschätzung der FPGA-Ressourcen ermöglichen und so auch die Ressourcen in die zeitkritische Optimierung der Entwurfsraumexploration einbezogen werden können.

Kapitel 8

Zusammenfassung und Ausblick

8.1 Zusammenfassung

In dieser Arbeit wurde ein Ansatz zur anwendungsspezifischen Co-Konfiguration von Systemsoftware und Hardwarearchitektur vorgestellt. Das Konfigurationswissen wird bei diesem Ansatz automatisiert anhand der Zugriffe auf die Schnittstellen zwischen den Schichten gewonnen und zur Maßschneiderung der jeweils tieferen Schicht eingesetzt. So entsteht ein durchgängiger Konfigurierungsprozess, bei dem das Konfigurationswissen durch die Schichten gereicht wird und die tieferen Schichten genau die Dienste zur Verfügung stellen, die von den oberen Schichten zur Abarbeitung der gestellten Aufgaben benötigt werden. Inkonsistenzen zwischen den Schichten können so direkt durch den vorgeschlagenen Konfigurierungsprozess ausgeschlossen werden.

Neben dem vorgeschlagenen Konzept zur Abstimmung der Hardware- und Softwareschichten, wurde in dieser Arbeit zudem die Fragestellung diskutiert, wie sich generische Hardwarekomponenten möglichst effizient wiederverwenden und zu komplexen, maßgeschneiderten Systemen zusammensetzen lassen, um trotz der immer kürzeren Entwicklungszyklen, möglichst robuste und hochspezialisierte Hardwarearchitekturen entwickeln zu können. Dazu wurde in dieser Arbeit untersucht, inwieweit sich die diversen Konzepte aus der Software-Produktlinienentwicklung zur organisierten Wiederverwendung von Komponenten auch auf die Entwicklung von Hardware-Produktlinien übertragen lassen, denn durch die Etablierung von Hardwarebeschreibungssprachen, wie VHDL oder Verilog, kann auch Hardware mittels Quelltext entworfen werden. Bei der Untersuchung der Übertragbarkeit fiel insbesondere auf, dass das häufig eingesetzte Mittel der Merkmaldiagramme, welches zur Spezifizierung der Variabilität in der Domäne der Software-Produktlinien eingesetzt wird, sich nicht in jeder Hinsicht auf eine Hardware-Produktlinie übertragen lässt. Dies liegt insbesondere darin begründet, dass die Variabilität einer Hardware-Produktlinie viel stärker durch die Architektur selbst geprägt wird und sich diese strukturellen Konfigurationsinformationen nicht in dem erforderlichen Rahmen über Merkmale abbilden lassen. Zudem wurden in dieser Arbeit verschiedene Techniken zur Implementierung von generischen und wiederverwendbaren Hardwarekomponenten untersucht, wobei unter anderem die Spracherweiterung AspectVHDL entstanden ist, welche erste

Möglichkeiten zur Beschreibung von querschneidenden sowie optionalen Belangen mittels Aspekten in VHDL bietet.

Um die in dieser Arbeit erlangten Erkenntnisse über Produktlinien im Kontext von Hardwarearchitekturen zu überprüfen, wurde parallel eine Hardware-Produktlinie zur Maßschneiderung von *Multi-* bzw. *Manycore*-Systemen prototypisch umgesetzt. Diese Hardware-Produktlinie erlaubt die modulare Verknüpfung von Prozessoren, Kommunikationsstrukturen, Speichern und Peripheriekomponenten. Die Beschreibung einer Hardwarekonfiguration erfolgt dabei über ein abstraktes Modell, welches sämtliche Details der generischen Komponenten sowie die eigentliche Implementierung mittels VHDL ausblendet. Die entstandene Hardware-Produktlinie bildet zusammen mit der portierten Betriebssystem-Produktlinie CiAO eine gemeinsame Plattform, über welche der vorgestellte Ansatz zur anwendungsspezifischen Co-Konfiguration anhand der Konfigurationsinformationen aus den Schnittstellen umgesetzt wurde. Die Schnittstelle zwischen dem Betriebssystem und der zugrunde liegenden Hardwarearchitektur, bildet die – ebenfalls in dieser Arbeit vorgestellte – statisch analysierbare Hardware-API. Anhand der Zugriffsmuster auf die Hardware-API, können die Konfigurationsinformationen für die passende Hardwarearchitektur automatisiert ermittelt und im Anschluss der VHDL-Quelltext mit Hilfe der Hardware-Produktlinie generiert werden.

Da der Fokus dieser Arbeit zudem auf dem Bereich der *Multi-* bzw. *Manycore*-Systeme liegt, wurde außerdem untersucht, wie solche parallelen Systeme zum einen effizient programmiert und zum anderen auch automatisiert optimiert werden können, denn ohne solche Mechanismen liegen schwierige Entwurfsentscheidungen, wie die Zuteilung der Tasks zu Prozessoren oder der Grad der Parallelität, trotz der entstandenen Produktlinien, weiterhin beim Entwickler. Zur Realisierung dieser Ziele wurde der in dieser Arbeit vorgeschlagene Ansatz – zur Gewinnung der Konfigurationsinformationen aus den Schnittstellen – konsequent weitergeführt. Die dazu entworfene parallele Programmierschnittstelle erlaubt einem Entwickler, die Implementierung von parallelen Anwendungen, ohne dabei auf die spätere Architektur oder die Zuteilung der Tasks Rücksicht nehmen zu müssen. Die parallele Programmierschnittstelle wurde als oberste Schicht des CiAO-Betriebssystems umgesetzt und ist, wie schon die Hardware-API, ebenfalls statisch analysierbar und erlaubt so die Extraktion der Konfigurationsinformationen für die Betriebssystem- und Hardwareschicht. Dadurch dass die parallele Programmierschnittstelle von der konkreten Systemarchitektur abstrahiert, wird zudem einer Entwurfsraumexploration der erforderliche Freiheitsgrad eingeräumt, um verschiedene Konfigurationen für eine Anwendung zu untersuchen. Die Konfigurationen werden automatisiert anhand von generierten Modellen bezüglich verschiedener nicht-funktionaler Eigenschaften bewertet und von einem Evolutionären Algorithmus optimiert.

Letztendlich steht einem Entwickler auf Basis dieser Konzepte ein Werkzeug zur Verfügung, mit dem sich für eine Anwendung, welche die parallele Programmierschnittstelle nutzt, die weiteren Schichten anwendungsspezifisch generieren und

optimieren lassen. Dementsprechend hängt auch der Nutzen, welcher dieses Werkzeug einbringen kann, maßgeblich davon ab, ob das vorgeschlagene parallele Programmiermodell zu der umzusetzenden Anwendung passt. Insbesondere für kleinere Anwendungen mit wenigen Tasks, die kaum Möglichkeiten zur Parallelisierung bieten, macht die Optimierung mittels LavA womöglich keinen Sinn, da der Entwurfsraumexploration für solche Anwendungen der benötigte Freiraum fehlt. Durch den modularen Aufbau des LavA-Prozesses ist es allerdings möglich, an verschiedenen Stellen des Prozesses mit der Entwicklung einzusteigen und die Optimierung so auszulassen. Der Entwickler kann beispielsweise die CiAO-Instanzen auch manuell konfigurieren. In diesem Fall übernimmt der LavA-Prozess allein die Generierung der konfigurierten CiAO-Instanzen sowie die Abstimmung und Generierung der passenden Hardwarestruktur. Falls der Entwickler auch auf das Betriebssystem verzichten möchte, kann die Hardware-API auch direkt aus der Anwendung angesprochen werden oder der Entwickler konfiguriert die Hardware über ein abstraktes Modell auf der Ebene der Hardwarekomponenten, wodurch diesem zumindest die gewünschte Hardwarestruktur automatisiert generiert werden kann.

8.2 Ausblick

Der in dieser Arbeit vorgestellte LavA-Prozess, bietet zahlreiche Anknüpfungspunkte zur Untersuchung von weiteren interessanten Forschungsfragen und Ideen im Kontext der anwendungsspezifischen Co-Konfiguration von Systemsoftware und Hardwarearchitektur. Einige dieser potenziellen Anknüpfungspunkte werden zum Abschluss dieser Arbeit in den folgenden Unterkapiteln als Ausblick skizziert.

8.2.1 Steigerung der Heterogenität

Eine wichtige Facette von *Multi-* bzw. *Manycore*-Systemen, welche in dieser Arbeit – zumindest für den Optimierungsprozess noch nicht betrachtet wurde – ist die Berücksichtigung von verschiedenen Prozessorarchitekturen. Diese Variabilität könnte der Entwurfsraumexploration weitere Optionen eröffnen, um die Systeme noch gezielter und feingranularer an die jeweiligen Anforderungen anzupassen. Bei dem bisher umgesetzten Optimierungsprozess beschränkt sich die Heterogenität innerhalb eines untersuchten Systems allein auf die eingesetzten Kommunikationsstrukturen sowie die spezifisch integrierten Peripheriegeräte für die einzelnen Knoten. Obwohl die LavA Hardware-Produktlinie bereits verschiedene Prozessorarchitekturen unterstützt, werden zur Optimierung der Systeme derzeit allein die MB-Lite(+) Prozessoren eingesetzt. Zum einen liegt dies an dem noch aufzubringenden Aufwand zur Portierung der CiAO Betriebssystem-Produktlinie für die weiteren Architekturen aus der LavA Hardware-Produktlinie, aber zum anderen auch an der zurzeit noch erforderlichen Methode, um die WCET bzw. BCET für die einzelnen Task-Blöcke in den Optimierungsprozess einzubringen

und zwar durch Annotationen im Quelltext mittels der parallelen Programmierschnittstelle. Diese Methode würde schnell an ihre Grenzen stoßen, falls verschiedene Prozessorarchitekturen verwendet würden und der Entwickler diese Zeiten für jede zur Verfügung stehende Prozessorarchitektur manuell vermessen und annotieren müsste.
Einen Ausweg könnten an dieser Stelle möglicherweise Werkzeuge zur automatisierten Ermittlung der Zeiten liefern. In diesem Fall würde es ausreichen, wenn der Entwickler allein die Zerlegung der Anwendung in Task-Blöcke vornimmt und die Bestimmung der Zeiten für die spezifizierten Blöcke den Analysewerkzeugen überlässt. So könnten, abhängig von den verfügbaren Analysewerkzeugen, verschiedene Prozessorarchitekturen in dem Optimierungsprozess verwendet und dem Entwickler auch die manuelle Ermittlung und Annotation der Zeiten abgenommen werden.
Des Weiteren könnte auch die Performanz der Systeme gesteigert werden, wenn nicht nur verschiedene Prozessorarchitekturen, sondern auch Beschleunigerkomponenten auf Hardwareebene von dem Optimierungsprozess hinzugeschaltet werden könnten. So könnten die Möglichkeiten der FPGAs voll ausgeschöpft und spezifische Hardwarebeschleuniger in Form von feingranularen parallelen Hardwarestrukturen für besonders zeitkritische Funktionalitäten integriert werden. Dazu müsste die LavA Hardware-Produktlinie um die entsprechenden Hardwarebeschleuniger, wie beispielsweise einer Komponente zur Berechnung der Fourier-Transformation, erweitert werden und zudem müsste dem Entwickler eine entsprechende Schnittstelle zum Zugriff auf diese Hardwarebeschleuniger bereitgestellt werden. Wenn diese Schnittstelle von der konkreten Umsetzung der Berechnung abstrahiert, könnte der Optimierungsprozess zum Beispiel entscheiden, ob eine bestimmte Funktionalität auf Basis einer Softwarebibliothek umgesetzt oder ein Hardwarebeschleuniger integriert werden soll.

8.2.2 Erweiterung des parallelen Programmiermodells

Das in Unterkapitel 5.3 vorgestellte parallele Programmiermodell könnte zudem um weitere hilfreiche Kommunikationsmuster ergänzt werden. Diese Kommunikationsmuster könnten zum einen dem Entwickler die Implementierung von hochgradig parallelen Anwendungen weiter erleichtern, aber zum anderen könnten diese Muster auch auf ihre Eignung zur Extraktion von zusätzlichen Konfigurationsinformationen untersucht werden. Denkbar wären zum Beispiel neue Abstraktionen, um die Implementierung von mehrstufigen, *pipeline*-ähnlichen Berechnungsfolgen zu unterstützen. Eine mögliche Abstraktion könnte dabei zum Beispiel so aussehen, dass die Tasks einer ersten Gruppe die berechneten Zwischenergebnisse an jeweils einen zugeordneten Task aus einer zweiten Gruppe weiterleiten, welche auf Basis dieser Ergebnisse dann die Berechnungen fortsetzen könnten. Auch die Unterstützung von unterschiedlichen Gruppengrößen wäre denkbar, falls die zweite Gruppe ein Vielfaches an Tasks besitzt, um beispielsweise einen höheren Rechenaufwand auszugleichen. Bei asymmetrischen Gruppengrö-

ßen würden die Zwischenergebnisse eines Tasks dann an entsprechend mehrere Tasks aus der zweiten Gruppe weitergereicht.

8.2.3 Unterstützung von periodischen Tasks

Für viele Anwendungen im Kontext von Eingebetteten Systemen werden periodische Task-Modelle verwendet, bei denen die einzelnen Tasks zyklisch zu einem definierten Zeitpunkt erneut ausgeführt werden. Die Perioden der implementierten Tasks unterscheiden sich dabei recht häufig, wodurch die Anzahl der Aufrufe für die Tasks innerhalb einer bestimmten Zeitspanne variieren kann. Entsprechend muss das Verhalten der Tasks auch zur Bestimmung der Performanz für den Optimierungsprozess berücksichtigt werden. Allerdings stellt der LavA-Prozess dem Entwickler bisher keine passende Abstraktion zur Verfügung, um die Perioden für die implementierten Tasks anzugeben, weshalb zurzeit angenommen wird, dass die Tasks einmalig ausgeführt werden und alle Tasks zum Systemstart ausführbereit sind. Da das vorgestellte Uppaal-Modell, zur Ermittlung der Performanz, bereits die Simulation von periodischen Tasks gestattet, müsste demnach primär die Programmierschnittstelle um eine neue Abstraktion zur Spezifizierung der Perioden erweitert werden, damit der LavA-Prozess durchgängig mit periodischen Tasks umgehen kann. Bei der Verwendung eines periodischen Task-Modells ist jedoch zu beachten, dass aufgrund der repetitiven Ausführung der Tasks keine konkreten Ausführungszeiten mittels der Simulationen ermittelt werden können. Stattdessen kann allerdings bestimmt werden, ob für eine gewisse Anzahl von Zyklen die jeweils gesetzten *Deadlines* der Tasks eingehalten und so die zeitliche Eignung des Systems validiert werden.

8.2.4 Feingranulare Maßschneiderung des Betriebssystems

Einer der Schwerpunkte dieser Arbeit lag auf der Fragestellung, wie Systemsoftware und Hardware anwendungsspezifisch in einem gemeinsamen Prozess konfiguriert und abgestimmt werden können. Insbesondere auf der Seite des Betriebssystems gibt es noch Konfigurationspunkte, die bislang nicht in der erforderlichen Tiefe betrachtet werden konnten, da der Fokus zunächst auf die zentralen Elemente für den LavA-Prozess gerichtet wurde, wie die Kommunikationsschicht oder die Gerätetreiber. Außer Acht gelassen wurde bisher beispielsweise die Flexibilität, welche durch die Nutzung von verschiedenartigen *Scheduling*-Strategien erzielt werden könnte. Die Wahl der passenden Strategie könnte dem Optimierungsprozess überlassen werden, indem man diesem einen weiteren globalen Schalter bereitstellt oder sogar für jeden Knoten gesondert die Konfigurierung erlaubt. Durch den modularen Aufbau der Simulations-Modelle, können diese bereits mit verschiedenen Strategien umgehen, weshalb lediglich die hinzugefügte Strategie selbst modelliert werden muss. Gerade im Hinblick auf die vorgeschlagene Unterstützung von periodischen Tasks könnte die Auswahl einer adäquaten Strategie von Vorteil sein, da diese einen maßgeblichen Einfluss darauf haben kann, ob ein Task seine *Deadline* einhalten kann oder nicht.

8.2.5 Erweiterung von AspectVHDL

Auch die Spracherweiterung AspectVHDL bietet, wie schon in Unterkapitel 4.4.4 angedeutet, viel Potenzial für die Untersuchung von neuen Konzepten, um die Aspektorientierte Programmierung weiter auf die spezifischen Anforderungen im Bereich der Hardwareentwicklung abzustimmen. Für den aktuellen Stand von AspectVHDL wurde zunächst vorrangig untersucht, wie sich die bekannten Konzepte aus der Aspektorientierten Programmierung auf die Hardwarebeschreibungssprachen respektive VHDL übertragen lassen. Der Grundstein für die Untersuchung von hardwarespezifischen Konzepten ist mit der entwickelten Spracherweiterung AspectVHDL jedenfalls gelegt.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[ABKS13] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*, volume 1. Springer-Verlag Berlin Heidelberg, 2013.

[AJMH13] Alexandra Aguiar, Sergio Johann, Felipe Magalhaes, and Fabiano Hessel. Customizable RTOS to support communication infrastructures and to improve design space exploration in MPSoCs. In *Proceedings of the 24th IEEE/IFIP International Symposium on Rapid System Prototyping (RSP '13)*, pages 130–135. IEEE, 2013.

[Aus10] David Austin. Eine Familie von gemeinsamen Speichern für MPSoCs. Diplomarbeit, Technische Universität Dortmund, 2010.

[AUT14] AUTOSAR. Specification of operating system (version 4.2.1), October 2014.

[Bas96] Paul G. Bassett. *Framing Software Reuse - Lessons From The Real World*. Prentice-Hall, Inc., 1996.

[BBA+11] Remi Busseuil, Lyonel Barthe, Gabriel Marchesan Almeida, Luciano Ost, Florent Bruguier, Gilles Sassatelli, Pascal Benoit, Michel Robert, and Lionel Torres. Open-Scale: A scalable, open-source NoC-based MP-SoC for design space exploration. In *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '11)*, pages 357–362. IEEE, 2011.

[BCOQ92] François Baccelli, Guy Cohen, Geert Jan Olsder, and Jean-Pierre Quadrat. *Synchronization and linearity: an algebra for discrete event systems*. John Wiley & Sons Ltd, 1992.

[BH98] Peter Bellows and Brad Hutchings. JHDL-an HDL for reconfigurable systems. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 175–184. IEEE, 1998.

[BKPS04] Günter Böckle, Peter Knauber, Klaus Pohl, and Klaus Schmid. *Software-Produktlinien: Methoden, Einführung und Praxis*. dpunkt.verlag, 2004.

[BLS12] Christoph Borchert, Daniel Lohmann, and Olaf Spinczyk. CiAO/IP: A highly configurable aspect-oriented IP stack. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys '12)*, pages 435–448. ACM, June 2012.

[BLTZ03] Stefan Bleuler, Marco Laumanns, Lothar Thiele, and Eckart Zitzler. PISA - a platform and programming language independent interface for search algorithms. In *Evolutionary multi-criterion optimization*, pages 494–508. Springer, 2003.

[BO92] Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(4):355–398, October 1992.

[BSP05] Andrew Bainbridge-Smith and Su-Hyun Park. ADH: an aspect described hardware programming language. In *Proceedings of the 2005 IEEE International Conference on Field-Programmable Technology (FPT '05)*, pages 283–284. IEEE, 2005.

[BSR04] Don Batory, Jacob N. Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, June 2004.

[CDK+01] Rohit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, 2001.

[CdOCM09] Everton A. Carara, Roberto P. de Oliveira, Ney L. V. Calazans, and Fernando G. Moraes. HeMPS - a framework for NoC-based MPSoC generation. In *IEEE International Symposium on Circuits and Systems (ISCAS '09)*, pages 1345–1348, May 2009.

[CE99] Krzysztof Czarnecki and Ulrich W. Eisenecker. Synthesizing objects. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP '99)*, pages 18–42. Springer, 1999.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Publishing Co., 2000.

[CHE05] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[CMK+11] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security Symposium*. San Francisco, 2011.

[CN07] Paul Clements and Linda Northrop. *Software product lines: practices and patterns*. Addison-Wesley Publishing Co., 2007.

[Cra14] Timo Cramer. Entwurf und Implementierung einer Kommunikationsbibliothek für parallele LavA-Anwendungen. Bachelorarbeit, Technische Universität Dortmund, 2014.

[CRJR+09] David Castells-Rufas, Jaume Joven, Sergi Risueño, Eduard Fernandez, and Jordi Carrabina. NocMaker: A cross-platform open-source design space exploration tool for networks on chip. In *Proceedings of the 3rd International Workshop on Interconnection Network Architecture: On-Chip, Multi-Chip (INA-OCMC '09)*, 2009.

[DM06] David Déharbe and Sergio Medeiros. Aspect-oriented design in SystemC: implementation and applications. In *Proceedings of the 19th annual symposium on Integrated circuits and systems design (SBCCI '06)*, pages 119–124. ACM, 2006.

[DPAM02] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[ES08] Michael Engel and Olaf Spinczyk. Aspects in hardware: what do they look like? In *Proceedings of the 2008 AOSD workshop on Aspects, components, and patterns for infrastructure software (ACP4IS '08)*, page 5. ACM, 2008.

[FAMH08] Sérgio J. Filho, Alexandra Aguiar, César A. Marcon, and Fabiano P. Hessel. High-level estimation of execution time and energy consumption for fast homogeneous MPSoCs prototyping. In *Proceedings of the 19th IEEE/IFIP International Symposium on Rapid System Prototyping (RSP '08)*, pages 27–33. IEEE, 2008.

[GAGS09] Daniel D. Gajski, Samar Abdi, Andreas Gerstlauer, and Gunar Schirner. *Embedded system design: modeling, synthesis and verification*. Springer Science & Business Media, 2009.

[GBD+94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel virtual machine: a users' guide and tutorial for networked parallel computing*. MIT press, 1994.

[GYJ01] Lovic Gauthier, Sungjoo Yoo, and Ahmed A. Jerraya. Automatic generation and targeting of application-specific operating systems and embedded systems software. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(11):1293–1301, 2001.

[HGD+13] Michael González Harbour, J. Javier Gutiérrez, José M. Drake, Patricia López Martínez, and J. Carlos Palencia. Modeling distributed real-time systems with MAST 2. *Journal of Systems Architecture*, 59(6):331–340, 2013.

[HGGPGDM01] Michael González Harbour, J. Javier Gutiérrez Garcia, J. Carlos Palencia Gutiérrez, and José M. Drake Moyano. MAST: Modeling and analysis suite for real time applications. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS '01)*, pages 125–134. IEEE, 2001.

[HK03] Martin Hitz and Gerti Kappel. *UML@Work - Von der Analyse bis zur Realisierung.* dpunkt.verlag, 2003.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA '93)*, volume 21, pages 289–300. ACM, 1993.

[JBZZ03] Stan Jarzabek, Paul Bassett, Hongyu Zhang, and Weishan Zhang. XVCL: XML-based variant configuration language. In *25th International Conference on Software Engineering (ICSE '03)*, pages 810–811, 2003.

[JCRR+09] Jaume Joven, David Castells-Rufas, Sergi Risueño, Eduard Fernandez, and Jordi Carrabina. NoCMaker & j2eMPI - A complete HW-SW rapid prototyping EDA tool for design space exploration of NoC-based MPSoCs. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '09)*. European Design and Automation Association, 2009.

[JQTG11] Ye Jun, Tan Qingping, Li Tun, and Cao Guorong. FeatureVerilog: Extending verilog to support feature-oriented programming. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW '11)*, pages 302–305. IEEE, 2011.

[Kah74] Gilles Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, August 1974. North Holland, Amsterdam.

[KAJW94] Sanjaya Kumar, James H. Aylor, Barry W Johnson, and Wm. A. Wulf. Object-oriented techniques in hardware design. *Computer*, 27(6):64–70, 1994.

[KB13] Frank Kesel and Ruben Bartholomä. *Entwurf von digitalen Schaltungen und Systemen mit HDLs und FPGAs: Einführung mit VHDL und SystemC*, volume 1. Oldenbourg Verlag, 2013.

[KBDV06] Minyoung Kim, Sudarshan Banerjee, Nikil Dutt, and Nalini Venkatasubramanian. Design space exploration of real-time multi-media MPSoCs with heterogeneous scheduling policies. In *Proceedings of the 4th International Conference Hardware/Software Codesign and System Synthesis (CODES+ISSS '06)*, pages 16–21. IEEE, 2006.

[KCH+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA, November 1990.

[Kes12] Frank Kesel. *Modellierung von digitalen Systemen mit SystemC: von der RTL-zur Transaction-Level-Modellierung.* Walter de Gruyter, 2012.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer Berlin Heidelberg, June 2001.

[Kir84] Scott Kirkpatrick. Optimization by simulated annealing: Quantitative studies. *Journal of statistical physics*, 34(5-6):975–986, 1984.

[KKO+06] Tero Kangas, Petri Kukkala, Heikki Orsila, Erno Salminen, Marko Hännikäinen, Timo D. Hämäläinen, Jouni Riihimäki, and Kimmo Kuusilinna. UML-based multiprocessor SoC design framework. *ACM Transactions on Embedded Computing Systems (TECS)*, 5(2):281–320, 2006.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin Heidelberg, 1997.

[KM07] Martin V. Künzli and Marcel Meli. *Vom Gatter zu VHDL: eine Einführung in die Digitaltechnik*. vdf Hochschulverlag AG, 2007.

[LF08] Slobodan Lukovic and Leandro Fiorin. An automated design flow for NoC-based MPSoCs on FPGA. In *Proceedings of the 19th IEEE/IFIP International Symposium on Rapid System Prototyping (RSP '08)*, pages 58–64. IEEE, 2008.

[LHSP+09] Daniel Lohmann, Wanja Hofer, Wolfgang Schröder-Preikschat, Jochen Streicher, and Olaf Spinczyk. CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems. In *2009 USENIX Annual Technical Conference*, pages 215–228, Berkeley, CA, USA, June 2009. USENIX Association.

[LHSPS11] Daniel Lohmann, Wanja Hofer, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. Aspect-aware operating system development. In *Proceedings of the 10th International Conference on Aspect-oriented software development (AOSD '11)*, pages 69–80. ACM, 2011.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152, 1997.

[LST+06] Daniel Lohmann, Fabian Scheler, Reinhard Tartler, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. A quantitative analysis of aspects in the ecos kernel. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys '06)*, pages 191–204, New York, NY, USA, 2006. ACM.

[LYBJ01] Damien Lyonnard, Sungjoo Yoo, Amer Baghdadi, and Ahmed A. Jerraya. Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip. In *Proceedings of the 38th Annual Design Automation Conference (DAC '01)*, pages 518–523, New York, NY, USA, 2001. ACM.

[Mar11] Peter Marwedel. *Embedded system design: Embedded systems foundations of cyber-physical systems*. Springer Science & Business Media, 2 edition, 2011.

[Mas02] Anthony Massa. *Embedded Software Development with eCos*. Prentice Hall Professional Technical Reference, 2002.

[MASS11] Matthias Meier, David Austin, Horst Schirmeier, and Olaf Spinczyk. TMPL: A hardware transactional memory product line. In *Proceedings of the Workshop on Multiprocessor Systems on (Programmable) Chips (MPSoC '11)*, Istanbul, Turkey, July 2011. IEEE Computer Society Press.

[MBS14] Matthias Meier, Mark Breddemann, and Olaf Spinczyk. Hardware APIs: A software-centric approach for automated derivation of MPSoC hardware structures based on static code analysis. In *Proceedings of the 27th GI/ITG International Conference on Architecture of Computing Systems (ARCS '14)*, volume 8350 of *Lecture Notes in Computer Science*, pages 111–122, Lübeck, Germany, February 2014. Springer-Verlag.

[MBS15] Matthias Meier, Mark Breddemann, and Olaf Spinczyk. Interfacing the hardware API with a feature-based operating system family. *Journal of Systems Architecture*, 61(10):531–538, November 2015.

[McD09] Austen McDonald. *Architectures for transactional memory*. PhD thesis, Stanford University, 2009.

[MCKO08] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC '08)*, pages 35–46. IEEE, 2008.

[MCM+04] Fernando Moraes, Ney Calazans, Aline Mello, Leandro Möller, and Luciano Ost. Hermes: an infrastructure for low area overhead packet-switching networks on chip. *Integration, the VLSI Journal*, 38(1):69–93, 2004.

[Mes12] Message Passing Interface Forum. MPI: A message-passing interface standard (version 3.0). Technical report, Knoxville, TN, USA, September 2012.

[MESS10] Matthias Meier, Michael Engel, Matthias Steinkamp, and Olaf Spinczyk. LavA: An open platform for rapid prototyping of MPSoCs. In *Proceedings of the 20th International Conference on Field Programmable Logic and Applications (FPL '10)*, pages 452–457, Milano, Italy, August 2010. IEEE Computer Society Press.

[MF14] Tiago R. Mück and Antônio A. Fröhlich. Aspect-oriented RTL HW design using SystemC. *Microprocessors and Microsystems*, 38(2):113–123, 2014.

[MHK+08] Jan Madsen, Michael R. Hansen, Kristian S. Knudsen, Jens E. Nielsen, and Aske W. Brekling. System-level verification of multi-core embedded systems using timed-automata. In *Proceedings of the 17th World Congress International Federation of Automatic Control*, pages 9302–9307, 2008.

[MHS12] Matthias Meier, Stefan Hanenberg, and Olaf Spinczyk. AspectVHDL Stage 1: The prototype of an aspect-oriented hardware description language. In *Proceedings of the 2nd Workshop on Modularity in Systems Software (MISS '12)*, pages 3–8, Potsdam, Germany, March 2012. ACM.

[MS11] Matthias Meier and Olaf Spinczyk. LavA: Model-driven development of configurable MPSoC hardware structures for robots. In *Proceedings of the Workshop on Software Language Engineering for Cyber-physical Systems (WS4C '11)*, Lecture Notes in Informatics, Berlin, Germany, October 2011. German Society of Informatics.

[NSD06] Hristo Nikolov, Todor Stefanov, and Ed Deprettere. Efficient automated synthesis, programing, and implementation of multi-processor platforms on FPGA chips. In *Proceedings of the 16th International Conference on Field Programmable Logic and Applications (FPL '06)*, pages 1–6. IEEE, 2006.

[PA08] James Psota and Anant Agarwal. rMPI: Message passing on multicore processors with on-chip interconnect. In *Proceedings of the 3rd International Conference on High Performance Embedded Architectures and Compilers (HiPEAC '08)*, volume 4917 of *Lecture Notes in Computer Science*, pages 22–37. Springer Berlin Heidelberg, 2008.

[Pal04] Samir Palnitkar. *Design Verification with e*. Prentice Hall Professional, 2004.

[PEP06] Andy D. Pimentel, Cagkan Erbas, and Simon Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Transactions on Computers*, 55(2):99–112, 2006.

[pur06] pure-systems GmbH. Variant management with pure::variants, technical white paper. 2006.

[RBM+04] Victor Reyes, Tomas Bautista, Gustavo Marrero, Pedro P. Carballo, and Wido Kruijtzer. CASSE: A system-level modeling and design-space exploration tool for multiprocessor systems-on-chip. In *Euromicro Symposium on Digital System Design (DSD '04)*, pages 476–483. IEEE, 2004.

[RR99] Ray Roth and Dinesh Ramanathan. A high-level hardware design methodology using C++. In *Proceedings of the 4th High Level Design Validation and Test Workshop*, pages 73–80, 1999.

[RS13] Jürgen Reichardt and Bernd Schwarz. *VHDL-Synthese: Entwurf digitaler Schaltungen und Systeme*. Oldenburg Verlag, 2013.

[SDF06] Stuart Sutherland, Simon Davidmann, and Peter Flake. *SystemVerilog for Design Second Edition: A Guide to Using SystemVerilog for Hardware Design and Modeling*. Springer Science & Business Media, 2006.

[SGG08] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*, volume 8. Wiley Publishing, 2008.

[SL07] Olaf Spinczyk and Daniel Lohmann. The design and implementation of AspectC++. *Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software*, 20(7):636–651, 2007.

[SLNM04] Frank Singhoff, Jérôme Legrand, Laurent Nana, and Lionel Marcé. Cheddar: a flexible real time scheduling framework. In *ACM SIGAda Ada Letters*, volume 24, pages 1–8. ACM, 2004.

[SS07] Horst Schirmeier and Olaf Spinczyk. Tailoring infrastructure software product lines by static application analysis. In *Proceedings of the 11th International Software Product Line Conference (SPLC '07)*, pages 255–260. IEEE Computer Society Press, 2007.

[SSSPS07] Julio Sincero, Horst Schirmeier, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. Is the linux kernel a software product line. In *Proceedings of the 2nd International Workshop on Open Source Software and Product Lines*, 2007.

[Ste10] Matthias Steinkamp. Maßschneiderung von MPSoCs durch Code-Analyse. Diplomarbeit, Technische Universität Dortmund, 2010.

[Ste13] Karl Stelzner. AspectVHDL: Entwurf und Implementierung eines Aspektwebers für VHDL. Bachelorarbeit, Technische Universität Dortmund, 2013.

[Str13] Sebastian Struwe. Entwurf einer API für konfigurierbare eingebettete MPSoCs in Anlehnung an HPC-Konzepte. Bachelorarbeit, Technische Universität Dortmund, 2013.

[SVEH07] Thomas Stahl, Markus Völter, Sven Efftinge, and Arno Haase. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. 2007.

[Tan05] Andrew S. Tanenbaum. *Structured Computer Organization*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005.

[TBHH07] Lothar Thiele, Iuliana Bacivarov, Wolfgang Haid, and Kai Huang. Mapping applications to tiled multiprocessor embedded systems. In *Proceedings of the 7th International Conference on Application of Concurrency to System Design (ACSD '07)*, pages 29–40. IEEE, 2007.

[TCN00] Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time calculus for scheduling hard real-time systems. In *Proceedings of the 2000 IEEE International Symposium on Circuits and Systems (ISCAS '00)*, volume 4, pages 101–104. IEEE, 2000.

[TH13] Klaus Ten Hagen. *Abstrakte Modellierung digitaler Schaltungen: VHDL vom funktionalen Modell bis zur Gatterebene.* Springer-Verlag, 2013.

[The93] The Design Automation Standards Committee of the IEEE. *IEEE Standard 1076-1993: VHDL.* 1993.

[THNY03] Hiroaki Takada, Shinya Honda, Reiji Nishiyama, and Hiroshi Yuyama. Hardware/software co-configuration for multiprocessor SoPC. In *Proceedings of the IEEE Workshop on Software Technologies for Future Embedded Systems (WSTFES '03)*, page 7. IEEE, 2003.

[TM98] Donald E. Thomas and Philip R. Moorby. *The Verilog® Hardware Description Language*, volume 2. Springer Science & Business Media, 1998.

[TNS+07] Mark Thompson, Hristo Nikolov, Todor Stefanov, Andy D. Pimentel, Cagkan Erbas, Simon Polstra, and Ed F. Deprettere. A framework for rapid system-level exploration, synthesis, and programming of multimedia MP-SoCs. In *5th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '07)*, pages 9–14. ACM, 2007.

[Var01] András Varga. The OMNeT++ discrete event simulation system. In *Proceedings of the 15th European Simulation Multiconference (ESM '01)*, volume 9, page 65, 2001.

[Vax07] Matan Vax. Conservative aspect-orientated programming with the e language. In *Proceedings of the 6th International Conference on Aspect-oriented software development (AOSD '07)*, pages 149–160. ACM, 2007.

[vdL02] Frank van der Linden. Software product families in europe: the esaps & café projects. *Software, IEEE*, 19(4):41–49, July 2002.

[VNS07] Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov. PN: A tool for improved derivation of process networks. *EURASIP Journal on Embedded Systems*, 2007(1):19–19, January 2007.

[Vog10] Stephan Vogt. LavA OS: Ein Betriebssystem für konfigurierbare MP-SoCs. Diplomarbeit, Technische Universität Dortmund, 2010.

[Wan06] Ernesto Wandeler. *Modular performance analysis and interface-based design for embedded real-time systems.* PhD thesis, ETH Zurich, 2006.

[Wei15] Karsten Weicker. *Evolutionäre algorithmen.* Springer Science & Business Media, 2015.

[WHH10] Yun Jie Wu, Dominique Houzet, and Sylvain Huet. A programming model and a NoC-based architecture for streaming applications. In *Proceedings of the 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD '13)*, pages 393–397. IEEE, 2010.

[WLZ+12] Stefan Wallentowitz, Andreas Lankes, Aurang Zaib, Thomas Wild, and Andreas Herkersdorf. A framework for open tiled manycore system-on-chip. In *Proceedings of the 22nd International Conference on Field Programmable Logic and Applications (FPL '12)*, pages 535–538. IEEE, 2012.

[Xil13] Xilinx, Inc. Spartan-3E FPGA family: Data sheet (ds312). 2013.

[Xil15] Xilinx, Inc. Multiplying the value of 16nm - staying a generation ahead. 2015.

[XPS08] Susan Xu and Hugh Pollitt-Smith. A multi-microblaze based SOC system: from SystemC modeling to FPGA prototyping. In *Proceedings of the 19th IEEE/IFIP International Symposium on Rapid System Prototyping (RSP '08)*, pages 121–127. IEEE, 2008.

[ZLT01] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. SPEA2: Improving the strength pareto evolutionary algorithm. Technical report, Eidgenössische Technische Hochschule Zürich (ETH), Institut für Technische Informatik und Kommunikationsnetze (TIK), 2001.