Boris Kettelhoit

Architektur und Entwurf dynamisch rekonfigurierbarer FPGA-Systeme

UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

FAKULTÄT FÜR
ELEKTROTECHNIK,
INFORMATIK UND
MATHEMATIK

Architektur und Entwurf dynamisch rekonfigurierbarer FPGA-Systeme

Zur Erlangung des akademischen Grades

DOKTORINGENIEUR (Dr.-Ing.)

der Fakultät für Elektrotechnik, Informatik und Mathematik
der Universität Paderborn
vorgelegte Dissertation
von

Dipl.-Ing. Boris Kettelhoit

aus Gütersloh

Referent:	Prof. Dr.-Ing. Ulrich Rückert
Korreferent:	Prof. Dr. rer. nat. Sybille Hellebrand

Tag der mündlichen Prüfung: 22.09.2008

Paderborn, den 28.02.2009

Diss. EIM-E/242

Geleitwort

Verbindendes Forschungsziel des von mir geleiteten Fachgebietes Schaltungstechnik im Heinz Nixdorf Institut ist der systematische Entwurf und der bedarfsgerechte Einsatz von mikroelektronischen Systemen in konkreten Anwendungen der Informations- und Automatisierungstechnik. Unsere Aktivitäten umfassen Arbeiten auf System- und Schaltkreisebene sowohl in digitaler als auch analoger Schaltungstechnik. Besondere Berücksichtigung finden massiv-parallele Realisierungsvarianten sowie die Bewertung der Ressourceneffizienz entsprechender Implementierungen. Ressourceneffizienz bedeutet hier, mit den physikalischen Größen Raum, Zeit und Energie sorgfältig umzugehen.

Rekonfigurierbare Hardware-Architekturen bilden einen interessanten Mittelweg zwischen den bisher bekannten Extremen einer flexiblen Programmierbarkeit von Mikroprozessoren und einer anwendungsspezifischen Schaltungsoptimierung. Rekonfigurierbarkeit bezeichnet hier die Möglichkeit, Funktionsblöcke und deren Verschaltung vor oder während des Betriebes zu verändern und somit die zur Verfügung stehenden Ressourcen an sich ändernde Anwendungsanforderungen anzupassen. Diese Eigenschaft von Hardware-Architekturen klingt zunächst sehr verlockend, wird aber derzeit weder von verfügbaren Entwurfswerkzeugen noch von Laufzeitumgebungen unterstützt. Eine Entwurfs- und Anwendungsunterstützung ist aber wesentlich für den praktischen Einsatz dynamisch rekonfigurierbarer Hardware-Architekturen. Herr Kettelhoit stellt in seiner Dissertation eine Systemumgebung für FPGA-Systeme vor, die sowohl den Entwurf als auch den Einsatz automatischer Hardware-Konfigurierung unterstützt. Er leistet damit einen wichtigen Beitrag zur praktischen Umsetzung dieser neuen Systemeigenschaften.

Der Schwerpunkt der Dissertation liegt auf der Realisierung freier Platzierungsverfahren für dynamisch rekonfigurierbare FPGA-Systeme. Herr Kettelhoit zeigt auf, dass aufgrund der hohen Anzahl verfügbarer Positionen möglichst homogene Bereiche für die dynamische Rekonfigurierung ausgewiesen sein sollten. An die Kommunikationsinfrastruktur des Systems, die zum Teil in den rekonfigurierbaren Bereichen liegt, werden dadurch besondere Anforderungen gestellt. Es werden verschiedene Realisierungsvarianten für homogene Kommunikationsinfrastrukturen vorgestellt und hinsichtlich ihres Ressourcenbedarfs bewertet. Zusätzlich hat Herr Kettelhoit mehrere Entwurfswerkzeuge entwickelt, die die Realisierung freier Platzierungsverfahren ermöglichen und weitgehend automatisieren. Durch die Kombination aus Evaluierungssystem und Entwurfswerkzeugen steht somit eine Entwicklungsumgebung bereit, mit der eine Vielzahl verschiedener rekonfigurierbarer Systeme schnell und zuverlässig entwickelt und validiert werden können.

Herr Kettelhoit hat seine Dissertation in der International Graduate School „Dynamic Intelligent Systems“ durchgeführt. Dem Land Nordrhein-Westfalen und der Universität Paderborn sei an dieser Stelle noch einmal ausdrücklich für die finanzielle sowie strukturelle Unterstützung und das damit verbundene Vertrauen gedankt.

Prof. Dr.-Ing. Ulrich Rückert

Inhaltsverzeichnis

1 Einleitung

Seit mehr als 20 Jahren werden Feld-programmierbare Gatter Arrays (FPGAs) als digitale Schaltungen in elektronischen Systemen eingesetzt. Mit dem FPGA steht dem Schaltungsentwickler seitdem ein mikroelektronisches Bauteil zur Verfügung, auf das noch nach seiner Fertigung und selbst noch nach der Integration in ein elektronisches System eine Vielzahl von digitalen Schaltungen abgebildet werden kann. Dadurch entfällt die kostspielige Entwicklung und Fertigung eines anwendungsspezifischen Halbleiterchips. Ähnlich wie Prozessoren können FPGAs günstig in hohen Stückzahlen eingekauft und problemspezifisch eingesetzt werden. Während Prozessoren jedoch die gewünschten Aufgaben weitestgehend sequentiell abarbeiten, können bei der Verwendung von FPGAs verschiedene Teilprobleme als separate Schaltungsteile realisiert und somit parallel verarbeitet werden. Durch diese problemspezifische und parallele Realisierung können viele Berechnungsprobleme in weit kürzerer Zeit mit FPGAs als mit Prozessoren gelöst werden. Prozessoren haben jedoch den Vorteil, dass sie ihre Funktion flexibel im Betrieb ändern können, indem ein anderer Programmcode geladen wird. Dies ist mit herkömmlichen FPGAs nicht möglich, da sie nicht während des Betriebs neu programmiert (oder konfiguriert) werden können. Die Problemunabhängigkeit eines herkömmlichen FPGAs bezieht sich daher nur auf den Entwurf und geht im Betrieb verloren. Die Wahl des Systementwicklers zwischen einem FPGA und einem Prozessor lässt sich somit stark vereinfacht auf die Abwägung zwischen Leistungsfähigkeit und Flexibilität reduzieren.

Seit einigen Jahren existieren nun dynamisch rekonfigurierbare FPGAs, die sich dadurch auszeichnen, dass ihre Funktion auch zur Laufzeit geändert werden kann. Man könnte meinen, dass hierdurch die oben genannte Abwägung obsolet ist, da nun Leistungsfähigkeit und Flexibilität in einem Baustein vereint scheinen. Dass dies nicht ohne weiteres zutrifft, zeigt sich allein daran, dass Prozessoren und FPGAs weiterhin in ihren angestammten Einsatzgebieten verwendet werden. Eine Verschiebung der Marktanteile zugunsten der FPGAs aufgrund der dynamischen Rekonfigurierung konnte in den letzten Jahren nicht beobachtet werden. Die Gründe hierfür mögen vielfältig sein, zwei wesentliche können aber mit Bestimmtheit genannt werden. Zum einen sind die Entwicklungswerkzeuge noch unterentwickelt und verbesserungswürdig. Selbst der Entwurf kleiner, wenig komplexer Systeme ist mit vielen unerwarteten Fehlern und Entwurfsiterationen, hervorgerufen durch nicht ausgereifte Entwicklungsumgebungen und schlechte oder nicht vorhandene Dokumentation, verbunden. Zum anderen fehlt es immer noch an Anwendungen, die von der Nutzung dynamischer Rekonfigurierung entscheidend profitieren. Die akademische Welt zeigt großes Interesse an dynamisch rekonfigurierbaren FPGAs und sucht nach geeigneten Anwendungen für diese neue Technik. Die präsentierten Anwendungen, z.B. aus dem Bereich der Grafik- und Videoverarbeitung [VJS95, BHR04, Sed06], der Automobilindustrie [CZM07, UHG04], der Kommunikationstechnik [Rys05, EOT05] oder der Regelungstechnik [DBK03, SPH07] demonstrieren zwar die Möglichkeiten der dynamischen Rekonfigurierung, sind bislang aber alternativen Lösungen (Grafikchips, herkömmlichen Prozessorsystemen, etc.) aus technischen Überlegungen oder beim Entwurf unterlegen.

Die Probleme der mangelnden Entwicklungswerkzeuge und der fehlenden Musteranwendung mögen sich zum Teil gegenseitig bedingen. Ohne verlässliche Entwicklungswerkzeuge wird keine verantwortungsvolle Firma dynamisch rekonfigurierbare FPGA-Systeme entwickeln. Auf der anderen Seite ist auch die Bereitstellung einer zuverlässigen Entwicklungsumgebung mit hohen Kosten für die FPGA-Hersteller verbunden, die Investitionen in eine Technik mit ungewisser Zukunft scheuen. Neben den Entwurfswerkzeugen ist es aber auch der Entwurf selbst, der noch viele Fragen offen lässt. Ein durchgehender Entwurfsablauf existiert bislang nur für Systeme mit *festen Modulplätzen*, bei denen ein oder mehrere Bereiche des FPGAs ausgewiesen werden, deren Inhalt zur Laufzeit nur als Ganzes verändert werden kann. Systemkomponenten, die dynamisch geladen werden, belegen dabei unabhängig von ihrer Komplexität immer gleichviele Ressourcen. Eine freie und flexiblere Verwendung verfügbarer Ressourcen (*freie Platzierung*), bei der ein freier Bereich ebenso gut für viele kleine wie für wenige große Systemkomponenten genutzt werden kann, wird bislang nur unzureichend unterstützt.

Schließlich müssen, nachdem der Nutzen durch die neu gewonnene Flexibilität so offensichtlich scheint, die Kosten der dynamischen Rekonfigurierung noch genauer betrachtet werden. Hierbei ist vor allem der Ressourcenbedarf eines dynamisch rekonfigurierbaren Systems von Interesse. Gegenüber herkömmlichen, statischen FPGA-Systemen liegen die Vorteile dynamischer Rekonfigurierung auf der Hand: Systemkomponenten, die nicht gleichzeitig benötigt werden, können zeitversetzt in das FPGA geladen und dadurch Ressourcen gespart werden. Da jedoch für dynamische Rekonfigurierung zusätzliche Infrastrukturkomponenten gebraucht werden und – wie im Fall der bereits oben erwähnten festen Modulplätze – verfügbare Ressourcen nicht immer optimal genutzt werden können, ist hier eine genauere Betrachtung angebracht. Nicht zuletzt lohnt sich auch eine vergleichende Betrachtung der Ressourceneffizienz verschiedener Verfahren für dynamische Rekonfigurierung.

1.1 Zielsetzung

Ziel dieser Arbeit ist es, die Möglichkeiten der dynamischen Rekonfigurierung mit aktuell verfügbarer FPGA-Technologie zu evaluieren und zu erweitern. Es werden insbesondere drei Fragestellungen betrachtet:

- Welche Realisierungsoptionen der dynamischen Rekonfigurierung gibt es bei der Verwendung aktuell verfügbarer FPGAs?
- Wie können dynamisch rekonfigurierbare Systeme schnell und zuverlässig entworfen werden?
- Was sind Kosten und Nutzen eines dynamisch rekonfigurierbaren Systems im Vergleich zu statischen FPGA-Systemen?

Der Begriff „dynamisch rekonfigurierbares System“ ist auch im Zusammenhang mit FPGAs weit gefasst. Um ihn zu konkretisieren, wird in Abbildung 1-1 der stark abstrahierte Aufbau eines rekonfigurierbaren Systems gezeigt, das als Muster für die in dieser Arbeit betrachteten Systeme gilt. Es wird in statische und dynamische Bereiche unterteilt. Statische Bereiche enthalten statische Systemkomponenten, die zur gesamten Laufzeit dieselben FPGA-Ressourcen belegen. Dynamische Systemkomponenten werden zur Laufzeit bei Bedarf

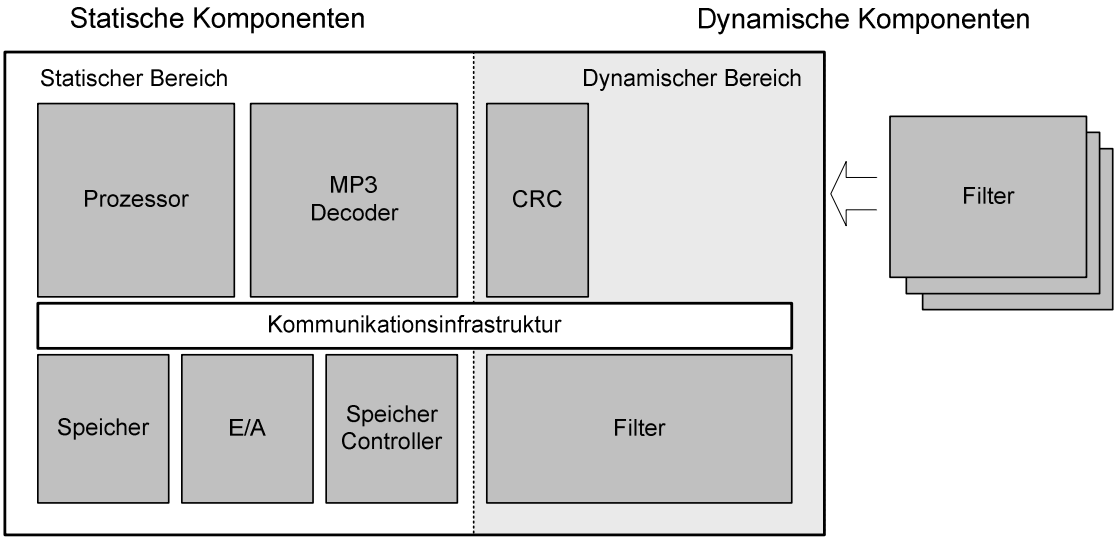

Abbildung 1-1: Beispiel eines dynamisch rekonfigurierbaren Systems

in freie Ressourcen der dynamischen Bereiche geladen. Alle Systemkomponenten kommunizieren über eine Kommunikationsinfrastruktur miteinander. Diese ist eine statische Komponente, die jedoch sowohl in statischen als auch in dynamischen Bereichen Ressourcen belegt. Die Kommunikationsinfrastruktur wird hier im Sinne eines Prozessorbusses betrachtet, so dass die dynamischen Systemkomponenten als Coprozessor dienen. Andere Architekturen, wie z.B. Paket-basierte On-Chip-Netzwerke, oder weitere mögliche Kopplungsvarianten zwischen Prozessor und dynamischen Komponenten, wie sie beispielsweise Compton et al. [CH02] vorschlagen, werden nur am Rande betrachtet.

Der statische Bereich unterscheidet sich prinzipiell nicht von klassischen, statischen FPGA-Systemen. Allerdings werden für den Betrieb eines rekonfigurierbaren Systems zusätzliche Hardware-Komponenten benötigt, denen besondere Beachtung geschenkt werden muss. Für die dynamischen Bereiche ist vor allem von Interesse, mit welchem Platzierungsverfahren dynamische Komponenten die verfügbaren Ressourcen nutzen. In dieser Arbeit liegt der Fokus auf den freien Platzierungsverfahren. Von ihnen verspricht man sich eine höhere Ressourceneffizienz als von festen Modulplätzen, für eine technische Realisierung müssen allerdings noch viele Herausforderungen bewältigt werden. Die Kommunikationsinfrastruktur ist dabei eine Schlüsselkomponente, die einen wesentlichen Einfluss auf die Realisierbarkeit und die Kosten der freien Platzierungsverfahren hat.

Als wichtigstes Bewertungskriterium wird in dieser Arbeit die Realisierbarkeit der vorgestellten Verfahren, technischen Neuerungen und Entwurfsmethodiken betrachtet. Hypothetische Betrachtungen zukünftiger FPGA-Architekturen werden nicht oder nur am Rande gemacht. Die tatsächliche Umsetzung als letzter Beweis der Realisierbarkeit wird daher als vorzügliches Mittel gewählt.

1.2 Aufbau der Arbeit

Im folgenden Kapitel wird zunächst der grundlegende Aufbau dynamisch rekonfigurierbarer FPGAs beschrieben. Im anschließenden dritten Kapitel werden die Grundzüge dynamischer Rekonfigurierung erläutert. Diese beiden Kapitel stellen somit den Stand der Technik

und die sich daraus ergebenden Möglichkeiten dar. Es werden aber auch schon die technischen Rahmenbedingungen und Herausforderungen aufgezeigt, die sich für eine Realisierung dieser Möglichkeiten ergeben. Als wesentlicher Bestandteil des dritten Kapitels wird zudem eine Modellierung dynamisch rekonfigurierbarer Systeme vorgenommen. Dabei geht es weniger um eine mathematisch korrekte Beschreibung rekonfigurierbarer Systeme als darum, eine geeignete Abstrahierung und Nomenklatur zu definieren. Dies ist deshalb notwendig, da sich hierfür noch keine allgemeingültigen Bezeichnungen eingebürgert haben.

Im dritten Kapitel wird außerdem gezeigt, dass die Architektur und der Entwurf der Kommunikationsinfrastruktur kritische Elemente bei der Realisierung freier Platzierungsverfahren sind. Da hier bislang kaum Lösungen existieren, wird der Kommunikation das vierte Kapitel gewidmet, in dem neue Verfahren für den Aufbau und den Entwurf einer geeigneten Kommunikationsinfrastruktur vorgestellt und miteinander verglichen werden.

Im fünften Kapitel wird eine Entwicklungsumgebung für dynamisch rekonfigurierbare Systeme vorgestellt. Sie wurde im Laufe dieser Arbeit entwickelt und ist gleichermaßen Hilfsmittel und Ergebnis dieser Arbeit. Mit ihr wurden die Untersuchungen zur Ressourceneffizienz gemacht und auch eine erste Anwendung, bei der Regelungen für mechatronische Systeme mit rekonfigurierbarer Hardware realisiert wurden, basiert auf ihr. Teil der Entwicklungsumgebung ist außerdem ein Entwurfsablauf für dynamisch rekonfigurierbare Systeme, für den eine Reihe eigens entwickelter Entwurfswerkzeuge existieren. Der Wert dieser Entwurfswerkzeuge kann nicht hoch genug geschätzt werden, da ohne sie die in dieser Arbeit durchgeführten Implementierungen nicht in diesem Maße möglich gewesen wären.

Mithilfe der Entwicklungsumgebung werden im sechsten Kapitel verschiedene Varianten dynamisch rekonfigurierbarer Systeme realisiert und bezüglich ihrer Ressourceneffizienz untersucht. Der Einfluss der Kommunikationsinfrastruktur als besonderes Element rekonfigurierbarer Systeme ist dabei ein Schwerpunkt der Betrachtungen. Durch einen Vergleich mit einer herkömmlichen, statischen Implementierung werden Kosten und Nutzen der rekonfigurierbaren Systeme dargestellt.

Im siebten Kapitel finden sich eine Zusammenfassung der Ergebnisse und ein Resümee.

2 Grundlagen rekonfigurierbarer FPGAs

1985 brachte die US-amerikanische Firma Xilinx das erste feldprogrammierbare Gatter-Array (FPGA[1]) auf den Markt und schuf somit das nun jüngste Mitglied der Familie der programmierbaren Logik (PLD[2]). Der Grundaufbau eines FPGAs bestand aus einer Matrix kleiner Logikelemente, deren Funktion und Verschaltung über einen Konfigurationsspeicher festgelegt werden konnte. Auf diese Matrix konnte nun fast jede beliebige digitale Schaltung abgebildet werden. Das wesentliche Merkmal aber – der bis heute wohl wichtigste Grund für die Verbreitung von FPGAs – war und ist die Möglichkeit, die Funktion des Bausteins erst *nach* der Fertigung (im Feld) festlegen zu müssen. Hierzu besitzen FPGAs einen Konfigurationsspeicher, der die Funktion des Bausteins bestimmt und dessen Inhalt noch nach der Fertigung verändert werden kann. Diese Konfigurierbarkeit ermöglicht es, den gleichen Chip in vielen verschiedenen Anwendungen mit jeweils unterschiedlicher Funktionalität zu verwenden und die hohen nicht wiederkehrenden Kosten der Halbleiterfertigung auf viele Anwender zu verteilen. Besonders für Schaltungen, die nur in geringer Stückzahl benötigt werden – sei es, weil für das Produkt nur ein kleiner Markt besteht, oder weil allgemeine Standards noch nicht endgültig definiert sind – sind FPGA-basierte Lösungen oft preiswerter als die Entwicklung einer anwendungsspezifischen Schaltung (ASIC[3]). Darüber hinaus kann mit programmierbarer Logik die Entwurfszeit einer digitalen Schaltung drastisch reduziert werden. Für die Verifikation von ASICs muss ein sehr hoher Aufwand für die Simulation getrieben werden, da die Produktion fehlerhafter Chips eine erneute Maskenfertigung und somit erhebliche Kosten und neuerlichen Entwicklungsaufwand bedeutet. Bei FPGAs kann aufgrund ihrer *Feld*-Programmierbarkeit das Testen einer Schaltung im Gesamtsystem viel stärker für die Verifikation genutzt werden. Änderungen an einer bestehenden Schaltung können noch in einem sehr späten Entwicklungsstadium und sogar nach der Markteinführung (z.B. durch ein *Firmware-Update*) gemacht werden. Bis heute sind die niedrigen Kosten bei Kleinserien und kurze Entwurfszeiten die beiden Hauptgründe für den Einsatz programmierbarer Logik. In 2005 wurden laut *In-Stat* [Ins06] weltweit insgesamt 1,9 Milliarden US-Dollar mit FPGAs umgesetzt. Xilinx und sein stärkster Konkurrent Altera hatten daran einen Anteil von etwa 45 % bzw. 35 %. Der Rest verteilt sich auf eine Reihe kleinerer Firmen.

Ein großer Vorteil von FPGAs gegenüber Prozessorsystemen ist die Möglichkeit, verschiedene Systemkomponenten auf einem einzigen Chip zu integrieren. Neben einer höheren Performanz bedeutet dies vor allem einen geringeren Platzbedarf und geringere Kosten beim Platinenentwurf. In dieser Arbeit wird untersucht, wie solche *Systeme-auf-einem-program-*

---

[1] Field-Programmable Gate Array

[2] Programmable Logic Device. Das Attribut „programmierbar“ ist irreführend, da mit der „Programmierung“ eines Bausteins kein Programm – also eine Sequenz von Befehlen oder Ereignissen – im eigentlichen Sinne übertragen wird. Vielmehr wird eine neue Konfiguration übertragen, die die Funktionalität des Bausteins bestimmt. Es wird daher im Folgenden ausschließlich die Terminologie „Konfiguration“ und „konfigurieren“ an Stelle von „Programm“ und „programmieren“ verwendet.

[3] Application Specific Integrated Circuit

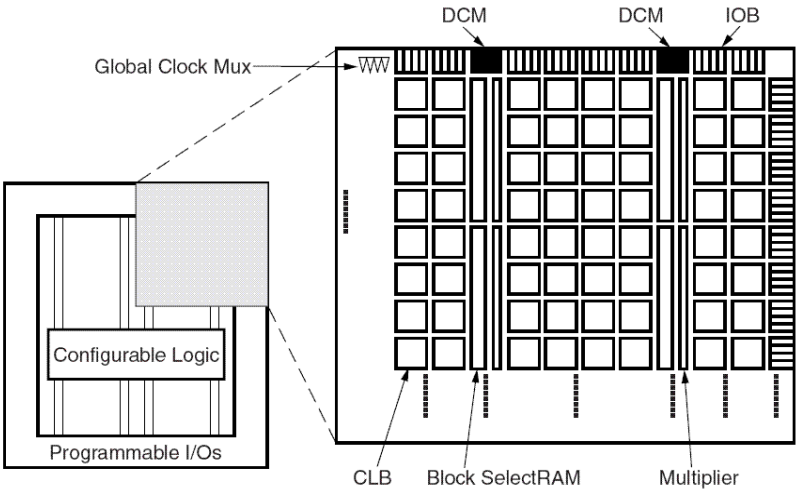

Abbildung 2-1: Aufbau eines Xilinx Virtex-II FPGAs [Xil05]

mierbaren-Chip (SoPC[4]) durch das dynamische Austauschen einzelner Systemkomponenten erweitert werden können. Die hierfür benötigte Basistechnologie, das dynamische Verändern des Konfigurationsspeichers, wird *dynamische Rekonfigurierung* genannt. Derzeit stellen nur die Firmen Xilinx und Atmel entsprechende FPGAs her. Das derzeit größte FPGA von Atmel kann nach Herstellerangaben 50 000 Gatteräquivalente abbilden [Atm06]. Ein Xilinx Virtex-5 FPGA (V5LX330T) integriert mehr als 7 Millionen Gatteräquivalente [Xil07a]. Für die Realisierung dynamisch rekonfigurierbarer SoPCs eignen sich daher zurzeit ausschließlich Xilinx FPGAs. Im Folgenden werden der Aufbau und die wesentlichen Merkmale dynamisch rekonfigurierbarer FPGAs am Beispiel der Xilinx Virtex-II [Xil05] und Virtex-4 [Xil07b] Architekturen erläutert. Sie lassen sich im Wesentlichen auf die FPGAs anderer Hersteller übertragen. Beispielhaft wird an geeigneten Stellen ein Vergleich mit Alteras *Stratix-II* Architektur [Alt07], dem derzeit leistungsstärksten Konkurrenzprodukt zu Xilinx FPGAs, gezogen. Der Aufbau des Konfigurationsspeichers und seiner Schnittstellen sowie die Angaben zum Entwurf rekonfigurierbarer Systeme sind wiederum Xilinx-spezifisch.

2.1 Konfigurierbare Logik

Abbildung 2-1 zeigt den prinzipiellen Aufbau von FPGAs am Beispiel eines Virtex-II FPGAs von Xilinx. Grundbestandteil ist eine Matrix rekonfigurierbarer Logik-Elemente, die zeilen- und spaltenweise auf dem FPGA angeordnet sind. Xilinx nennt diese Elemente *konfigurierbare Logikblöcke* (CLBs[5]). Mit jedem CLB können einfache logische Operationen sowie Speicherfunktionen auf Bit-Ebene realisiert werden. Deswegen gehören FPGAs der Familie der *fein-granularen* konfigurierbaren Architekturen an. Die CLBs können fast beliebig miteinander verschaltet werden. Hierzu ist jedoch eine sehr aufwändige Verdrahtungsstruktur nötig, so dass in heutigen FPGAs nur ein kleiner Teil der Fläche für Logikelemente verwendet wird. Um dem entgegenzuwirken, werden häufig benötigte, grob-granulare Funktionen als Hartmakros in die CLB-Matrix eingebettet. Im Falle des Virtex-II FPGAs sind dies Speicher-

[4] System on Programmable Chip
[5] Configurable Logic Blocks

blöcke (*Block SelectRAM*, kurz *BlockRAM*) und 18x18-Bit Multiplizierer, die als separate Spalten in die CLB-Matrix integriert sind. Einige FPGA-Familien (z.B. Virtex-2Pro und Virtex-4-FX) verfügen darüber hinaus über einen oder mehrere eingebettete Prozessoren. Neben diesen grobgranularen Funktionsblöcken gibt es eine Reihe weiterer Elemente, die für den Betrieb des FPGAs nötig oder förderlich sind. Hierzu gehören z.B. Module für die Taktversorgung und -erzeugung (DCM[6]), und die später noch näher beschriebene interne Konfigurierungsschnittstelle. Schließlich umgibt die CLB-Matrix ein Ring von ebenfalls konfigurierbaren Ein-/Ausgangsblöcken (IOBs[7]), über die die Kommunikation mit externen Schaltungen stattfindet. Mit Einführung der Virtex-4 Familie sind diese Blöcke jedoch nicht mehr als IO-Ring angeordnet, sondern als IO-Spalten in die CLB-Matrix eingebettet.

Alle konfigurierbaren Elemente sind über lokale und globale Routingressourcen miteinander verbunden. Zeilen- und spaltenweise durchziehen Signalleitungen unterschiedlicher Länge und Art das FPGA. In jedem CLB und neben den eingebetteten Funktionsblöcken befinden sich so genannte Switchboxen (*Switch Matrix*), die die funktionalen Einheiten an die globalen Routingressourcen anbinden. Jedes CLB ist dabei über eine eigene Switchbox angeschlossen, grob-granulare Elemente verfügen in der Regel über mehrere Switchboxen. Lokale Routingressourcen gibt es nur innerhalb der CLBs sowie zwischen benachbarten CLBs. Sie sind nicht an die globale Routing-Matrix angebunden. In Abbildung 2-1 nicht dargestellt ist der Konfigurationsspeicher, durch den die Funktion des FPGAs bestimmt wird. Er ist über das gesamte FPGA verteilt und kann durch die Konfigurations-Schnittstelle ausgelesen und beschrieben werden.

Aufgrund der in Abbildung 2-1 gegebenen Sicht auf das FPGA spricht man im Allgemeinen von einer horizontalen und vertikalen Ausdehnung der rekonfigurierbaren Ressourcen (CLBs). Entsprechend kann man für digitale Schaltungen, die auf einen Teil dieser Ressourcen abgebildet werden, eine Höhe und Breite angeben.

2.1.1 Konfigurierbare Logikblöcke

Abbildung 2-2 zeigt den Aufbau eines Virtex-II CLBs. Er besteht aus vier *Slices*, die die rekonfigurierbare Logik enthalten. Kernelemente eines Slice sind beim Virtex-II ein Funktionsgenerator und ein 1-Bit Register. Daneben gibt es eine Reihe einfacher Schaltelemente, die für die Realisierung häufig benutzter Funktionen, wie z.B. Addiererketten, gebraucht werden. Der Funktionsgenerator ist als 16-Bit Speicher realisiert, der wahlweise als Schieberegister (SRL16[8]), als Speicher mit vier Adresseingängen und einem Datenausgang (RAM16[9]), oder als Wahrheitstabelle für Boolesche Funktionen mit vier Variablen (LUT[10]) genutzt werden kann. Der 16-Bit Speicher des Funktionsgenerators ist Teil des Konfigurationsspeichers und kann über die Konfigurierungsschnittstellen verändert bzw. ausgelesen werden. Das bedeutet aber auch, dass bei einer Nutzung des Funktionsgenerators als Speicherelement (RAM16 oder SRL16) der aktuelle Speicherinhalt bei jeder dynamischen Rekonfigurierung verloren geht.

---

[6] Digital Clock Manager
[7] Input Output Blocks
[8] Shift Register, 16 Bit Depth
[9] Random Access Memory, 16 Bit
[10] Look-Up Table

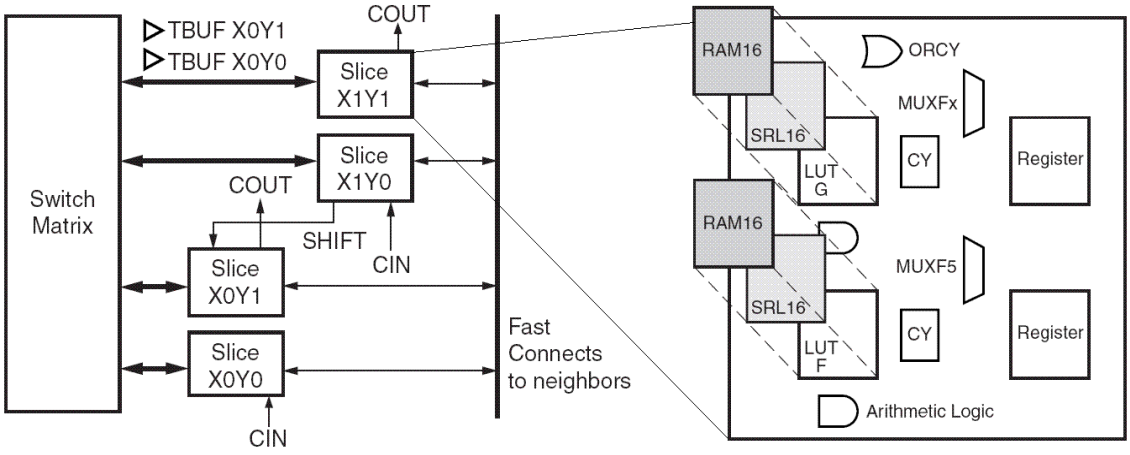

Abbildung 2-2: Aufbau eines Virtex-II CLBs [Xil05]

Das Flipflop einer Logikzelle ist ein synchrones D-Flipflop, das wahlweise mit asynchronem oder synchronem lokalem Reset genutzt werden kann. Der Speicherinhalt nach einem lokalen Reset und der Initialwert nach einem globalen Set/Reset des gesamten FPGAs können konfiguriert werden. Der Speicherinhalt selbst ist anders als beim Funktionsgenerator nicht Bestandteil des Konfigurationsspeichers. Das bedeutet zum einen, dass der Wert des Flipflops nach einer dynamischen Rekonfigurierung erhalten bleibt. Andererseits kann der Wert des Flipflops aber auch nicht ohne weiteres durch ein Auslesen des Konfigurationsspeichers ermittelt werden (siehe 2.2.2).

Für die Kommunikation mit anderen CLBs ist jedes Slice an die Switch-Box des CLBs angebunden. Darüber hinaus gibt es *Fast Connects*, über die die Slices untereinander direkt verbunden sind. Für einige Operationen mit Übertrag gibt es dedizierte Signale (CIN, COUT, SHIFT), die Slices benachbarter CLBs miteinander verbinden.

Der hier beschriebene, prinzipielle Aufbau eines CLB ist für alle Xilinx FPGAs gleich. Unterschiede gibt es allerdings in der Anzahl der Slices pro CLB, in der Anzahl der Funktionsgeneratoren und Flipflops pro Slice, und in der Komplexität der Funktionsgeneratoren. Ein Virtex-5 CLB besteht beispielsweise aus zwei Slices mit je vier Flipflops und Funktionsgeneratoren. Letztere basieren hier auf 64-Bit Speicher, so dass Wahrheitstabellen mit sechs Eingängen und entsprechend größere Speicherfunktionen realisiert werden können.

Die Stratix-II Architektur [Alt07] von Altera ähnelt dem hier beschriebenen Aufbau stark. Bei Altera werden die konfigurierbaren Logikblöcke *Logic Array Blocks* (LABs) genannt, die jeweils aus acht *Adaptive Logic Modules* (ALMs) bestehen. Ein ALM ist somit das Altera Äquivalent zu Xilinx Slices. Jedes ALM besteht aus je zwei *Adaptive LUTs* (ALUTs), zwei Registern sowie zusätzlichen Logikelementen für Übertrag-Ketten (*Carry-Chains*) und weiteren arithmetischen Funktionen.

2.1.2 Routingressourcen

Für die Verbindungen der CLBs nutzt Xilinx eine hierarchische Routing-Struktur, die aus verschiedenen globalen und lokalen Signalleitungen besteht. Globale Leitungen können dabei nur über Switchboxen erreicht werden, lokale Routingressourcen sind direkte Verbindungen zwischen benachbarten CLBs.

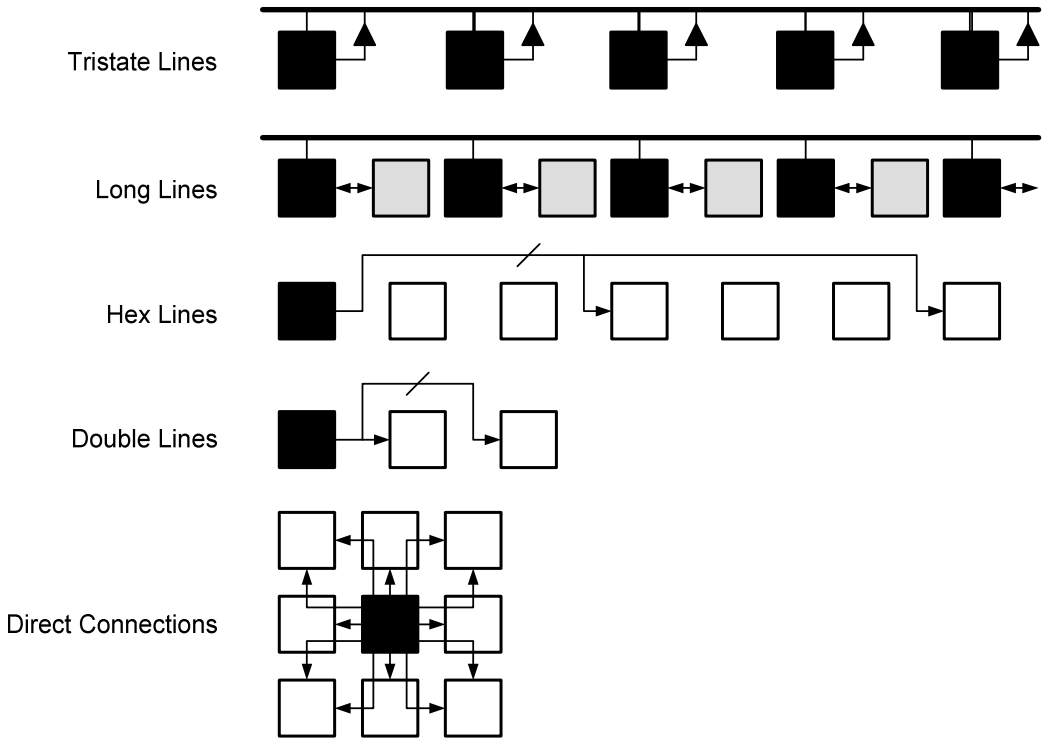

Abbildung 2-3: Verbindungen zwischen den Switchboxen eines Virtex-II FPGAs [Xil05]

Abbildung 2-3 zeigt die vorhandenen globalen Routingressourcen am Beispiel der Virtex-II Architektur. Sie unterscheiden sich im Wesentlichen durch die Länge der Verbindung gemessen in CLB-Spalten oder -Zeilen. Die kürzesten, und somit schnellsten Verbindungen sind die *Direct Connections*. Mit ihnen können Signale zu horizontal, vertikal oder diagonal benachbarten Switchboxen geroutet werden. Die nächst längeren Leitungen sind *Double Lines*. Sie verbinden eine Switchbox horizontal oder vertikal mit ihrem nächsten und übernächsten Nachbarn. Mit *Hex Lines* kann in gleicher Weise ein bis zu sechs Spalten oder Reihen entferntes CLB erreicht werden. Abgebildet ist hier eine *Unihex Line*, die nur zum dritten und sechsten CLB führt. Es gibt außerdem *Fullhex Lines*, die alle CLBs innerhalb der Reichweite kontaktieren. Direct Connections, Double Lines und Hex Lines können immer nur von einem CLB, das sich an einem Ende der Signalleitung befindet, getrieben werden. Für alle anderen kontaktierten CLBs ist das Signal ein Eingang, der nicht getrieben werden kann.

Long Lines umspannen den gesamten FPGA (ab der Virtex-4 Familie einen Abschnitt des FPGAs) und sind für die Kommunikation über große Entfernungen konzipiert. Eine Long Line kontaktiert dabei jeden sechsten CLB. Anders als bei den bisher beschriebenen Signalen kann eine Long Line prinzipiell von allen beteiligten CLBs getrieben werden. Bei statischen Systemen (ohne dynamische Rekonfigurierung) stellt das Entwurfs-Werkzeug sicher, dass im Betrieb tatsächlich nur ein CLB die Leitung treibt. Bei dynamisch veränderlichen Systemen kann dies unter Umständen nicht immer sichergestellt werden, weshalb Long Lines hier besondere Beachtung geschenkt werden muss. Für Long Lines wie auch für die zuvor beschriebenen Signale gilt, dass die Signallaufzeit nur unwesentlich von dem *Fan-Out*[11] des Signals

[11] Ausgangslast, hier Anzahl der angeschlossenen Elemente

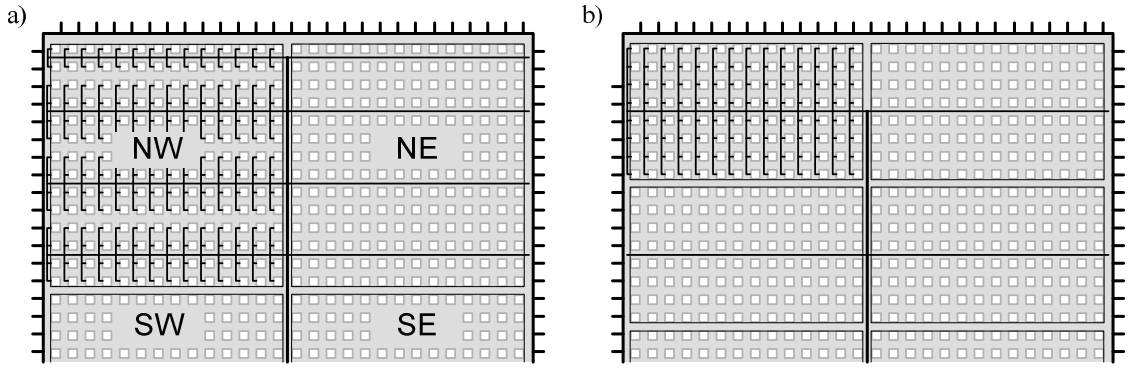

Abbildung 2-4: Schematische Darstellung der Taktbäume, a) Virtex-II, b) Virtex-4

abhängig ist, da für alle Signal-Ausgänge separate Treiber verwendet werden. Xilinx bezeichnet diese Technologie als *Active Interconnect Technology*. Sie wurde eingeführt, um zuverlässige und eindeutige Verzögerungszeiten für die Routingressourcen angeben zu können. Die tatsächlich anfallenden Verzögerungsunterschiede bei unterschiedlichem Fan-Out liegen in der Größenordnung einiger Pikosekunden.

Bis zur Virtex-II Familie integriert Xilinx zusätzlich Tristate-Signale (*Tristate Lines*) in die FPGAs, die horizontal den kompletten FPGA durchziehen. Eine Tristate-Leitung kann neben den logischen Zuständen ‚1' und ‚0' auch den Zustand ‚Z' (nicht getrieben) annehmen. Der Zugriff auf solche Leitungen erfolgt über Tristate-Treiberbausteine, deren Ausgang treibend oder nicht treibend (hochohmig) sein kann. Pro CLB-Reihe sind vier solcher Tristate Lines vorhanden, die in jeder vierten Spalte unterbrochen, also segmentiert werden können. Jedes CLB kann nur zwei dieser Leitungen treiben, jedoch alle vier Signale abgreifen. Die Tristate-Signale eignen sich besonders gut für die Realisierung gemeinsam genutzter Signale (*shared signals*), die später bei der Entwicklung der Kommunikationsinfrastruktur noch eine wichtige Rolle spielen.

Die letzte wichtige Gruppe von Leitungen dient der Übertragung von Taktsignalen. Taktsignale sind in digitalen Schaltungen üblicherweise die Signale mit den höchsten Anforderungen bezüglich des Timings, da sie alle getakteten Elemente möglichst synchron erreichen sollen. Um den *Clock-Skew*, also den Laufzeitunterschied eines Taktsignals zu den einzelnen Elementen, gering zu halten, besitzen FPGAs dedizierte Taktnetze, die in Form von Taktbäumen das FPGA überziehen. Abbildung 2-4 a) zeigt die Taktversorgung eines Virtex-II/Virtex2Pro FPGAs. In der Mitte des FPGA befindet sich ein vertikaler Takt-Strang, über den vier Quadranten (benannt nach den englischen Bezeichnungen der Himmelsrichtungen: North-East, North-West, usw.) versorgt werden können. Jeder Quadrant ist von mehreren horizontalen Taktsträngen durchzogen, die die zweite Ebene der Taktversorgung bilden. Als letzte Ebene sind in den FPGA-Spalten wiederum vertikale Taktleitungen angebracht, die die Switchboxen an das Taktnetz anbinden. Pro Quadrant existieren acht dedizierte Taktnetze, die zur Übertragung unabhängiger Taktsignale verwendet werden können. Alle Äste des Taktbaums sind auf jeder Ebenen separat aktivierbar. Nicht genutzte Elemente müssen daher nicht getaktet werden, so dass Clock-Skew und Energiebedarf nicht unnötig hoch sind.

Ab der Virtex-4 Serie hat Xilinx statt der Quadranten Taktregionen eingeführt. Jede Taktregion umfasst in der Breite den halben FPGA und in der Höhe genau 16 CLB-Reihen. Abbildung 2-4 b) zeigt den Taktbaum eines Virtex-4 FPGAs. Taktsignale werden auch hier zu-

nächst über einen vertikalen Taktstrang über das FPGA verteilt. Über je einen horizontalen Strang werden dann die Taktregionen angeschlossen. Als dritte Baumebene dient auch hier ein vertikaler Taktstrang pro FPGA-Spalte. Gegenüber der Virtex-II Architektur können die Taktregionen zusätzlich zu den globalen Takten mit lokalen Taktsignalen versorgt werden. Diese dienen der besseren Anbindung externer Komponenten (z.B. von Speicherbausteinen) an das FPGA.

2.2 Konfigurationsspeicher

FPGAs besitzen einen Konfigurationsspeicher, der die Funktion aller konfigurierbaren Elemente (Logik, Speicher, Verdrahtung) bestimmt, also die *Konfiguration* des FPGAs beinhaltet. In diesen Speicher werden durch *Konfigurierung*[12] die Konfigurationsdaten geschrieben. Ebenso ist ein Auslesen des Konfigurationsspeichers möglich, um die aktuelle Konfiguration des FPGAs zurück zu gewinnen. Sowohl Beschreiben wie auch Auslesen des Speichers finden über die Konfigurierungsschnittstellen des FPGA statt.

Bei einigen konfigurierbaren Bausteinen ist der Konfigurationsspeicher *fuse*- oder *anti-fuse*-basiert, wobei der Speicherinhalt wie bei einem PROM[13] eingebrannt wird und danach nicht mehr verändert werden kann. Diese Bausteine können somit genau einmal konfiguriert werden. Viele moderne FPGAs besitzen dagegen einen wiederbeschreibbaren, SRAM[14]-basierten Konfigurationsspeicher und sind somit *rekonfigurierbar*. Kann dieser Speicher auch zur Laufzeit des auf das FPGA geladenen Systems verändert werden, spricht man von *dynamisch rekonfigurierbaren* FPGAs. Alle FPGAs mit dieser Eigenschaft sind gleichzeitig auch *partiell*, d.h. teilweise, rekonfigurierbar, weshalb oft synonym der Begriff *partiell und dynamisch rekonfigurierbarer FPGA* verwendet wird. Nur FPGAs, die einen zur Laufzeit veränderbaren Konfigurationsspeicher haben, ermöglichen derzeit die Implementierung rekonfigurierbarer SoPCs und sind somit die technologische Grundlage dieser Arbeit.

2.2.1 Konfigurationsgranularität

Wie erwähnt besitzen rekonfigurierbare FPGAs einen SRAM-basierten Konfigurationsspeicher. Entgegen der eigentlichen Bedeutung erlaubt dieser in der Regel jedoch keinen wahlfreien Zugriff (*random access*), sondern kann nur segmentweise beschrieben oder ausgelesen werden. Dadurch kann insbesondere bei großen FPGAs, deren Konfigurationsspeicher heute bis zu mehreren Megabyte groß ist, der Konfigurations-Adressdekoder klein gehalten werden. Eine eindeutige Korrelation zwischen Speichersegmenten und den konfigurierbaren Elementen des FPGAs gibt es nicht. Das bedeutet, dass ein Speichersegment Konfigurationsdaten von mehreren Elementen (z.B. mehrerer CLBs) enthalten kann. Ebenso können die Konfigurationsdaten eines Elements auf mehrere Speichersegmente verteilt sein. Dieser Zusammenhang sowie die Größe der Speichersegmente wird im Folgenden als Konfigurations-

[12] Um Verwechslungen zu vermeiden, beschreibt in dieser Arbeit der Begriff *Konfiguration* eine Dateneinheit, die die Funktion des FPGA oder eines Teils von ihm bestimmt, während die *Konfigurierung* den Vorgang beschreibt, mit dem die Funktion des FPGA geändert wird. Durch *Konfigurierung* wird somit eine *Konfiguration* in das FPGA geladen.
[13] Programmable Read-Only Memory: einmal beschreibbarer Speicher.
[14] Static Random Access Memory: Speicher mit wahlfreiem Zugriff, i. d. R. beliebig oft beschreibbar.

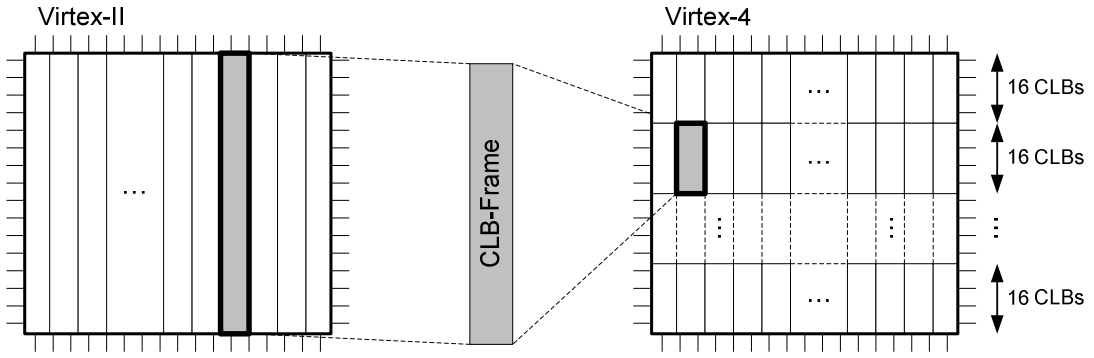

Abbildung 2-5: Konfigurationsgranularität von Virtex-II und Virtex-4 FPGAs

granularität bezeichnet. Xilinx bezeichnet ein Speichersegment als *Frame*, wobei hiermit sowohl das Speichersegment selbst als auch das entsprechende Datensegment im Konfigurationsdatenstrom gemeint ist, das die entsprechenden Konfigurationsdaten enthält.

Bezüglich der Konfigurationsgranularität verwendet Xilinx in seinen FPGA-Familien, die die dynamische Rekonfigurierung unterstützen, zwei verschiedene Konzepte. In den Virtex, Virtex-E, Virtex-2Pro, Virtex-II und Spartan-3 Familien wird eine spaltenweise Konfigurierung verwendet, d.h. ein Frame überspannt die komplette Höhe des FPGAs (siehe Abbildung 2-5). Ein Frame kann hier somit als Konfigurations-Spalte betrachtet werden, die jedoch nicht deckungsgleich mit einer Spalte auf der funktionalen Ebene (z.B. einer CLB-Spalte) ist. Eine CLB-Spalte in der Virtex-II Familie besteht beispielsweise aus 22 Frames. Die Größe der Frames ist wiederum von der Größe des FPGAs abhängig. Bei der Virtex-II Familie variiert sie zwischen 832 Bit beim XC2V40 und 9152 Bit beim XC2V8000. Für die verschiedenen Funktionseinheiten gibt es unterschiedliche Frames:

- *IOI-* und *IOB-Frames* am rechten und linken Rand für die Konfiguration der IO-Blöcke,
- *CLB-Frames* mit den Konfigurationsdaten der CLBs, der I/O-Blöcke des oberen und unteren Rands und der untersten Ebene der Taktbäume,
- *BRAM-Interconnect-Frames* für die Konfiguration der eingebetteten Speicher und Multiplizierer,
- *BRAM-Content-Frames*, die den Inhalt des BlockRAMs beschreiben, und
- *Center-Frames*, die den Rest der Taktnetze, die DCMs und weitere Einheiten konfigurieren.

Jede Spalte des FPGAs (CLB-Spalte, BlockRAM/Multiplizierer-Spalte, etc.) wird durch eine bestimmte Anzahl entsprechender Frames konfiguriert (z.B. 22 CLB-Frames pro CLB-Spalte). Die Größe der Frames ist innerhalb eines FPGAs immer gleich.

Mit der Virtex-4 Familie führte Xilinx ein neues Speicherkonzept mit fester Framegröße ein, das auch bei Virtex-5 FPGAs verwendet wird. Jede FPGA-Spalte setzt sich hier nicht mehr nur horizontal aus mehreren Frames zusammen, sondern ist auch vertikal auf mehrere Frames verteilt (siehe Abbildung 2-5). Die vertikale Segmentierung entspricht dabei genau der Einteilung der Taktregionen. Die Framegröße ist für alle FPGA-Typen einer Familie identisch. Jeder Virtex-4 Frame besteht aus 41 Datenwörtern zu je 32 Bit. Die Daten eines CLB-Frames verteilen sich dabei auf 16 übereinander angeordnete CLBs. Die Anzahl der CLB-Reihen eines Virtex-4 FPGAs ist somit ein Vielfaches von 16 und variiert von 64 bis 192. Um

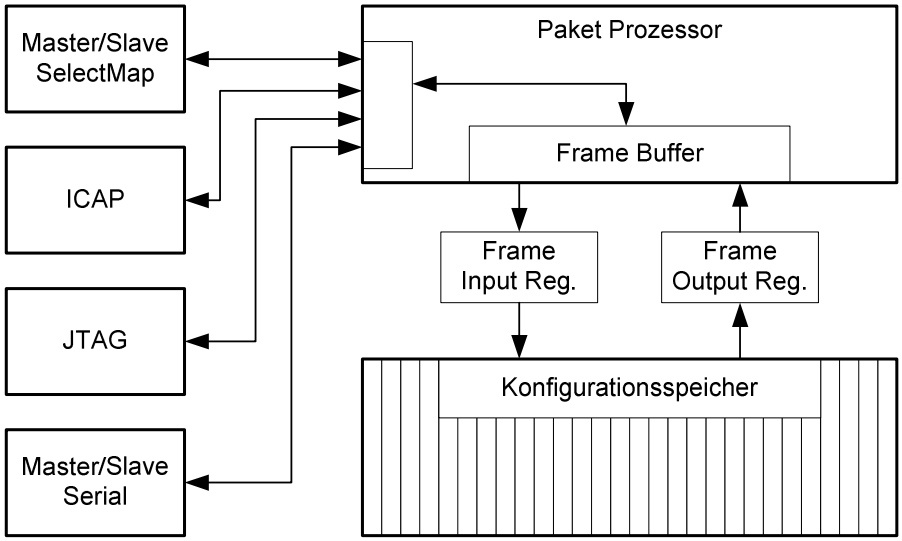

Abbildung 2-6: Zugriff auf den Konfigurationsspeicher durch die Konfigurierungsschnittstellen

einen Block von 16 CLBs komplett zu konfigurieren, müssen weiterhin 22 Frames geladen werden. Für die Konfiguration der IOs und eingebetteten Hartmakros ist die Segmentierung identisch.

In allen FPGA-Familien sind die Konfigurationsdaten für die Taktversorgung auf verschiedene Frame-Typen verteilt. Die Verschaltung von der obersten Taktbaumebene bis in die Quadranten bzw. Taktregionen ist in den Center-Frames definiert. Die unterste Ebene, also die Anbindung der getakteten Elemente an den Taktbaum, wird über andere Frames festgelegt. Dies ist insbesondere von Bedeutung, als dass die Center-Frames in der Regel nur bei der Initialkonfigurierung und nicht während dynamischer Rekonfigurierungen beschrieben werden. Ein Teil der Taktversorgung muss also schon vor der Laufzeit für alle möglichen Konfigurationen des FPGA ausgelegt werden.

2.2.2 Konfigurierungsschnittstellen

Für den Zugriff auf den Konfigurationsspeicher bietet Xilinx mehrere externe und in der Regel eine interne Konfigurierungsschnittstelle an. Von außen kann ein FPGA über *JTAG*- und *Master/Slave Serial*-Modi seriell, und über *SelectMap* bis zu 32 Bit parallel konfiguriert werden. Zusätzlich integriert Xilinx bei den meisten seiner FPGAs die interne Konfigurierungsschnittstelle ICAP (*Internal Configuration Access Port*), die wie die SelectMap-Schnittstelle funktioniert, allerdings nur partielle Rekonfigurierungen unterstützt. Keine der Schnittstellen hat einen direkten Zugriff auf den Konfigurationsspeicher, sondern lediglich eine Verbindung zu einem internen Konfigurations-Paket-Prozessor, über den alle Konfigurationsdaten an den Konfigurationsspeicher übertragen werden. Abbildung 2-6 veranschaulicht den prinzipiellen Aufbau der Konfigurationslogik in Xilinx FPGAs. Mit allen Konfigurierungsschnittstellen kann mit dem Konfigurationsprozessor eine Verbindung aufgebaut werden. Ist dies geschehen können in einem zweiten Schritt Daten zum Prozessor gesendet oder von ihm empfangen werden. Die Konfigurierungsschnittstellen selbst verfügen neben den Datenleitungen nur über wenige Steuer- und Kontrollsignale. Die gesamte Steuerung des Pa-

ketprozessors findet über den Datenstrom statt, der an ihn übertragen wird. Dieser Datenstrom wird als *Bitstrom* bezeichnet.

2.2.2.1 Initiale Konfigurierung

Nach dem Anlegen der Betriebsspannungen des FPGA, dem *Power-Up*, muss eine initiale Konfigurierung durchgeführt werden. Sie kann nur über eine der externen Schnittstellen durchgeführt werden und besteht aus vier Phasen:

- Löschen des Konfigurationsspeichers
- Initialisierung
- Übertragen der Konfigurationsdaten
- FGPA *Startup*

Das Löschen des Konfigurationsspeichers eines FPGAs nach dem Power-Up ist zwingend notwendig, da die SRAM-Speicherzellen zu Beginn keinen definierten Pegel haben. Diese im Prinzip wahllose Mischung aus logischen ‚1'en und ‚0'en hat eine ebenso wahllose Konfiguration der angeschlossenen Logik- und Speicherelemente zur Folge. Da das SRAM nicht durch ein *Reset* initialisiert werden kann, wird es in dieser ersten Phase durch vollständiges Beschreiben in einen definierten Zustand gebracht. Die hierfür benötigte Zeit hängt von der FPGA-Familie sowie von der Größe des FPGAs ab. In der Virtex-II Familie kann das Löschen bis zu 4 µs pro Frame dauern. Für den in dieser Arbeit für prototypische Realisierungen verwendeten XC2V4000 FPGA mit 2156 Frames bedeutet dies eine Verzögerung von bis zu 8,624 ms.

Erst in der anschließenden Initialisierungsphase wird eine der Konfigurierungsschnittstellen ausgewählt. Der Benutzer kann sie über drei *Mode*-Pins des FPGAs bestimmen, die in dieser Phase abgefragt werden. Die hierfür benötigte Zeit ist ebenfalls technologieabhängig und beträgt für den XC2V4000 bis zu 4 µs.

In der Konfigurierungsphase werden über die Datenleitungen Pakete an den Konfigurations-Paket-Prozessor gesendet. Diese Pakete beinhalten neben den eigentlichen Konfigurationsdaten, also den Frame-Inhalten, auch Kommandos, die den Paketprozessor steuern. Dies kann kontinuierlich oder auch unterbrochen geschehen, je nach dem, wie die Konfigurationsdaten bereitgestellt und vom Paketprozessor verarbeitet werden können. Die Verarbeitungsgeschwindigkeit des Paketprozessors ist unabhängig von der gewählten Konfigurierungsschnittstelle und in jeder FPGA-Familie unterschiedlich. Bei der Verwendung der SelectMap- oder ICAP-Schnittstelle, die aufgrund ihrer parallelen Dateneingänge die schnellste Konfigurierungsmöglichkeit darstellen, können Daten mit bis zu 50 MHz und 8 Bit parallel (für Virtex-II) kontinuierlich übertragen werden. Bei höheren Frequenzen muss eine durch *Handshaking*-Signale kontrollierte, unterbrochene Übertragung stattfinden. Seit der Virtex-4 Familie gibt es dieses Handshaking-Verfahren nicht mehr. Hier kann der Paketprozessor bis zu maximalen SelectMap- oder ICAP-Konfigurierungsfrequenz eine kontinuierliche Verarbeitung garantieren. Diese Frequenz beträgt bei Virtex-4 und Virtex-5 FPGAs 100 MHz, bei denen die Daten bis zu 32 Bit parallel übertragen werden können.

Die Konfigurationsdaten, die während der Konfigurationsphase übertragen werden, müssen nicht zwingend einem vollständigen Bitstrom entsprechen. Aufgrund der Segmentierung des Konfigurationsspeichers können beliebige, durchaus nicht zusammenhängende Bereiche des FPGAs konfiguriert werden. Statische Systeme werden normalerweise durch eine voll-

ständige, initiale Konfigurierung auf das FPGA geladen. Bei dynamischen Systemen wird in der Regel zu Beginn ebenfalls eine vollständige Konfigurierung und anschließend beliebig viele partielle und dynamische Konfigurierungen durchgeführt. Die hierfür verwendeten Bitströme werden daher *Initialbitstrom* und *partielle Bitströme* genannt. Unabhängig davon, ob vollständig oder partiell, am Ende einer initialen Konfigurierung muss durch einen Befehl an den Paketprozessor ein globales Set/Reset durchgeführt werden, durch das die Flipflops in den CLBs ihre in der Konfiguration vorgegebenen Initialwerte übernehmen.

Vor der abschließenden Startup-Phase ist das FPGA vollständig initialisiert und konfiguriert. Um einen sicheren Start des Benutzerdesigns sicher zu stellen, werden in dieser Phase zunächst die Ausgangstreiber des FPGAs freigeschaltet. Dann erst werden alle Speicherelemente (Flipflops, RAM16, SRL16 und BlockRAMs) aktiviert, so dass eine logische Zustandsveränderung stattfinden und das FPGA seine gewünschte Funktion wahrnehmen kann. Die Startup-Phase kann vom Benutzer beliebig lange hinausgezögert werden, z.B. um verschiedene FPGAs eines Systems zu synchronisieren. Im schnellsten Fall dauert sie acht Konfigurationstakte.

2.2.2.2 Dynamisches Rekonfigurieren und Auslesen

Für eine partielle und dynamische Rekonfigurierung kann neben den externen auch die interne ICAP Schnittstelle verwendet werden. Um sie durchführen zu können, sollte zuvor ein Initialbitstrom geladen worden sein. Von den vier Phasen der initialen Konfigurierung wird hier nur das Übertragen der Konfigurationsdaten durchgeführt. Dabei werden ebenfalls neben den eigentlichen Konfigurationsdaten Kommandos an den Paketprozessor übertragen. Gegenüber einer initialen Konfigurierung werden aber einige Befehle ausgelassen. Insbesondere ein globales Set/Reset sollte nicht durchgeführt werden, da dabei neben dem gerade geladenen Modul auch der gesamte Rest der Schaltung in seinen Initialzustand zurückgesetzt wird. Dies ist bei dynamischen Systemen natürlich nicht gewünscht.

Ein wichtiges Merkmal der Xilinx FPGAs ist, dass es beim Schreiben der Konfigurationsdaten nicht zu Störungen der Konfigurationsbits kommen kann. Das bedeutet, dass die Rekonfigurierung mit identischen Konfigurationsdaten keinen Einfluss auf die Funktion der betroffenen FPGA-Elemente hat, weder während noch nach der Rekonfigurierung. Trotz des segmentierten Konfigurationsspeichers kann daher selbst ein einzelnes Konfigurationsbit geändert werden, ohne die Funktion der Elemente zu beeinflussen, die demselben Frame angehören.

Wie später noch gezeigt wird, kann es für den Betrieb dynamisch rekonfigurierbarer Systeme nützlich sein, den aktuellen Inhalt des Konfigurationsspeichers auszulesen. Dies wird ähnlich wie eine dynamische Rekonfigurierung durchgeführt, nur dass hier abwechselnd Kommandos geschrieben und Daten gelesen werden. Analog zu dem globalen Set/Reset kann vor einem Auslesen ein so genanntes *Capture* durchgeführt werden. Dadurch werden die Inhalte aller Flipflops während des Betriebs in den Konfigurationsspeicher zurück geschrieben. Die Funktion des FPGAs wird dadurch nicht beeinträchtigt, so dass das Capture auch für ein partielles und dynamisches Auslesen verwendet werden kann.

2.2.2.3 Der Konfigurations-Paket-Prozessor

Der wahlfreie Zugriff auf den Konfigurationsspeicher, zumindest aber ein wahlfreier Zugriff auf Speichersegmente, ist zwingend erforderlich für die partielle und dynamische Konfi-

gurierung eines FPGA. Da die dynamische Rekonfigurierung aber bis heute fast ausschließlich von Forschungseinrichtungen genutzt wird, ist es unwahrscheinlich, dass Xilinx nur zu diesem Zweck die Segmentierung – und somit den wahlfreien Zugriff – des Konfigurationsspeichers eingeführt hat. Zumal dadurch sowohl die Bitströme durch die Verwendung einer Adressierung größer werden und auch ein zusätzlicher Adressdecoder in der Konfigurierungsschnittstelle benötigt wird. Von Xilinx gibt es hierzu kaum Angaben. Da der Paketprozessor in der von Xilinx verwendeten Konfigurierungsschnittstelle jedoch im Wesentlichen durch im Bitstrom enthaltene Befehle und Adressen gesteuert wird, kann er sehr leicht in verschiedenen FPGAs einer Familie eingesetzt werden. Diese Wiederverwendbarkeit ist von Bedeutung, da Xilinx immer mehrere unterschiedlich große FPGAs einer Familie entwickelt und somit Zeit gespart und die Fehleranfälligkeit reduziert werden kann. Die Anpassung an die Größe des FPGAs kann dann im Wesentlichen über den Bitstrom – und somit in Software – geschehen. In den jüngsten FPGA-Familien ist Xilinx sogar dazu übergegangen, feste Framegrößen einzuführen, obwohl dadurch die möglichen FPGA-Größen deutlich eingeschränkt werden. Der Aufbau des Paket-Prozessors wird allerdings weiter vereinheitlicht.

2.2.3 Bitstromgröße

Die Größe eines Bitstroms ist im Wesentlichen von der Menge der zu konfigurierenden Ressourcen abhängig. Sie bestimmt, wie viele Frames des Konfigurationsspeichers beschrieben werden müssen. Neben den Konfigurationsdaten (den Frames) müssen allerdings auch Kommandos und so genannte *Pad-Frames* und *Pad-Wörter* an den Paket-Prozessor gesendet werden. Pad-Frames dienen dem Leeren des Frame-Buffers des Paketprozessors und müssen am Ende einer Konfigurierung sowie am Ende eines Auslesevorgangs eingefügt werden. Pad-Wörter müssen am Ende jedes Frames eingefügt werden. Sie enthalten im Gegensatz zu den Pad-Frames Konfigurationsdaten. Kommandos an den Paket-Prozessor finden sich in jedem Fall am Anfang eines Bitstroms, können aber aus verschiedenen Gründen auch innerhalb des Bitstroms auftreten. Die genaue Größe eines Bitstroms lässt sich nur für eine vollständige Konfigurierung eindeutig angeben. Die Größe partieller Bitströme ist von der Menge und Art der zu konfigurierenden Ressourcen und der Konfiguration selbst abhängig. Aufeinander folgende, identische Frames brauchen nur einmal geschrieben zu werden und können dann in mehrere Stellen des Konfigurationsspeichers übernommen werden, so dass eine *Bitstromkompression* erreicht wird, die sich auch auf die Konfigurierungsdauer auswirkt. Für den Fall, dass ein partieller Bitstrom einen zusammenhängenden Bereich des Konfigurationsspeichers beschreibt – was bei der Verwendung von CLBs und BlockRAM nicht der Fall ist – und falls keine Bitstromkompression verwendet wird, kann die Bitstromgröße wie folgt bestimmt werden:

$$B = 48 + \left(S_{Frame} + 4\right) \cdot N_{Frame} + S_{Frame} \tag{1}$$

B ist hierbei die Bitstromgröße in Bytes. S_{Frame} bezeichnet die Größe eines Frames, N_{Frame} die Anzahl der Frames. Die Größe des Eingangsbuffers und somit auch des Pad-Worts sind 32 Bit (4 Bytes), ein Pad-Frame ist so groß wie alle anderen Frames auch. Die Kommandosequenz am Anfang eines partiellen Bitstroms ist in der Regel 48 Bytes groß.

Wie erwähnt fließt die Menge der Ressourcen, deren Funktion geändert werden soll, nur indirekt über die Anzahl und die Größe der Frames in die Bitstromgröße ein. Aufgrund der

Segmentierung des Konfigurationsspeichers kann es allerdings zu einem sehr ungünstigen Verhältnis aus *zu verändernden* Konfigurationsdaten und *zu beschreibenden* Konfigurationsdaten kommen. Für den (allerdings unwahrscheinlichen) Fall, dass nur ein CLB eines Virtex-II FPGAs umkonfiguriert werden soll, müssen aufgrund des Spalten-basierten Aufbaus des Konfigurationsspeichers die Konfigurationsdaten aller 80 CLBs der betreffenden CLB-Spalte im Bitstrom enthalten und an den Paket-Prozessor übermittelt werden. Auch wenn in diesem Fall 79 CLBs nicht in ihrer Funktion geändert werden und somit bereits vorhandene Daten erneut geschrieben werden.

2.2.4 Konfigurierungsgeschwindigkeit

Für die Dynamik rekonfigurierbarer Systeme ist die Konfigurationsgeschwindigkeit ein wichtiges Qualitätsmerkmal. Insbesondere in Systemen, wo schnell auf sich ändernde Umgebungszustände reagiert werden muss, sind hohe Konfigurierungslatenzen unerwünscht. Es muss also das Ziel sein, die unter Umständen recht großen Konfigurationsdatenmengen (derzeit bis zu einigen MB) schnell in den Konfigurationsspeicher zu schreiben. Die derzeit am Markt verfügbaren FPGAs wurden jedoch für statische Implementierungen entwickelt. In statischen Systemen spielt eine schnelle Konfigurierung keine Rolle, da in der Regel nur einmal nach dem Start des Systems komplett konfiguriert wird. Auf der anderen Seite sind die vielen Millionen Transistoren der SRAM-Zellen des Konfigurationsspeichers eine Quelle immenser statischer Verlustleistung. Mit den immer kleiner werdenden Strukturgrößen der Halbleiterfertigung sind die Leckströme durch das Gate-Oxid sowie zwischen den Source- und Drain-Anschlüssen der Transistoren drastisch angestiegen [Kle05]. Nachdem sie bis vor wenigen Jahren vernachlässigbar klein waren sind sie inzwischen zu gleichen Anteilen wie die dynamische Verlustleistung an der gesamten Stromaufnahme digitaler Schaltungen beteiligt. Xilinx wurde nach eigenen Angaben erstmals bei der in 90 nm Technologie gefertigten Virtex-4 Serie mit deutlich ansteigenden statischen Verlustleistungen konfrontiert [Tel06]. Um dem entgegen zu wirken werden alle nicht performanzrelevanten Transistoren auf Kosten der Schaltgeschwindigkeit mit einer höheren Gate-Oxid-Dicke gefertigt, um die Leckströme zu verringern. Im Wesentlichen hiervon betroffen sind die Transistoren des Konfigurationsspeichers, der dadurch längst nicht so schnell beschreibbar ist wie vergleichbare SRAM-Speicher. Ebenfalls betroffen sind Pass-Transistoren, die für das Routing von Signalen auf dem Chip verwendet werden [Cur06].

Auch wenn Xilinx hierzu keine Angaben macht, wird die Verwendung einer höheren Oxid-Dicke der Transistoren des Konfigurationsspeichers für die eher geringe Konfigurierungsgeschwindigkeit verantwortlich sein. Die höchste Konfigurationsgeschwindigkeit wiesen in Virtex-4 FPGAs die SelectMap- und ICAP-Schnittstellen auf. Beide sind für eine maximale Taktrate von 100 MHz ausgelegt. Bei Virtex-II- und Virtex2Pro-FPGAs können immerhin noch bei einer Frequenz von 50 MHz kontinuierlich Daten übertragen werden. Die Übertragungsdauer oder Konfigurierungszeit t_{config} kann unter Kenntnis der Bitstromgröße B direkt mit der Übertragungsfrequenz f_{config} und Datenbreite b ermittelt werden:

$$t_{config} = \frac{B}{b \cdot f_{config}} \tag{2}$$

Bei einem XC2V4000 FPGA beträgt die Konfigurierungszeit einer vollständigen Konfigurierung 35,54 ms. Dabei sind Initialisierung, Löschen des Konfigurationsspeichers o.ä. nicht mit berücksichtigt. Zum Vergleich: die Rekonfigurierung einer CLB-Spalte auf demselben FPGA dauert 380 µs.

2.3 FPGA-Entwurfsablauf

Wie bei allen technischen Systemen steht auch beim Systementwurf für FPGAs die Spezifikation des Systems gemäß gegebener Anforderungen am Anfang des Entwurfs. Aus dieser Spezifikation wird dann eine erste, abstrakte Systembeschreibung erstellt, die alle benötigten Systemkomponenten sowie ihre Verschaltung enthält. Die Spezifikation und die Beschreibung auf Systemebene werden bis heute in der Regel manuell durchgeführt. Sie sind für die Qualität des zu erstellenden Systems von entscheidender Bedeutung. Die folgenden Entwurfsschritte dienen der Erzeugung des Systems aus der Systembeschreibung, im Falle des FPGA muss also ein Bitstrom für die Konfigurierung erzeugt werden. Diese Entwurfsschritte können weitestgehend automatisiert durchgeführt werden. Hierfür muss die Systembeschreibung allerdings in einem für Entwurfswerkzeuge lesbarem Format vorliegen, das als Entwurfseingabe (engl. *Design Entry*) dient. In Abbildung 2-7 ist ein vereinfachter Entwurfsablauf für statische FPGA-Systeme von der Systembeschreibung bis zum fertigen Konfigurationsdatenstrom abgebildet. Der Entwurfsfluss ist dabei durch Pfeile gekennzeichnet. Nicht dargestellt sind Verifikationsschritte, z.B. Simulationen und Tests, mit denen die Funktionalität der entstehenden Schaltung überprüft wird. Iterationen im Entwurf, die nach negativen Verifikationsergebnissen durchgeführt werden müssen, sind ebenfalls nicht abgebildet.

Typische Eingabeformate für die automatisierte Erstellung von Bitströmen sind Hardwarebeschreibungssprachen (HDL[15]). Gegenüber anderen Eingangsformaten (Zustandsdiagramme, Blockschaltbilder, usw.) haben HDLs den Vorteil, dass sie gut von Simulations- und Synthesewerkzeugen interpretiert werden können und dass sie sowohl strukturelle Beschreibungen als auch Verhaltensbeschreibungen eines Systems ermöglichen. Aus einer HDL-Beschreibung wird durch eine *Synthese* eine strukturelle Beschreibung in Form einer Netzliste erstellt. Sie besteht aus einer Menge funktionaler Elemente (Knoten) sowie aus Verbindungen dieser Elemente (Kanten), die zusammen weiterhin die zu erstellende Schaltung beschreiben. Die Netzliste ist prinzipiell unabhängig von der Zieltechnologie und stellt das Eingangsformat für die nun folgenden FPGA-spezifischen Entwurfsschritte dar. Diese Schritte wurden hier zu einem Schritt (*Map & PAR*) zusammengefasst. Beim *Mappen* (engl. *to map* = abbilden) werden die Knoten der Netzliste auf die verfügbaren Elementtypen des Ziel-FPGAs abgebildet. Z.B. werden logische Funktionen wie XOR oder 1-Bit-Additionen durch die Funktionsgeneratoren (LUTs) der konfigurierbaren Logikblöcke realisiert, Speicherelemente werden auf die Flipflops oder eingebettete Speicherblöcke abgebildet. Beim anschließenden Platzieren und Verdrahten (*PAR*[16]) wird jedem Knoten eine eindeutige Ressource auf dem FPGA zugewiesen (die Knoten werden *platziert*). Ebenso werden die Verbindungen zwischen den Elementen gemäß den Kanten der Netzliste mit den verfügbaren Routingressourcen realisiert (*verdrah-*

[15] Hardware Description Language
[16] Place and Route

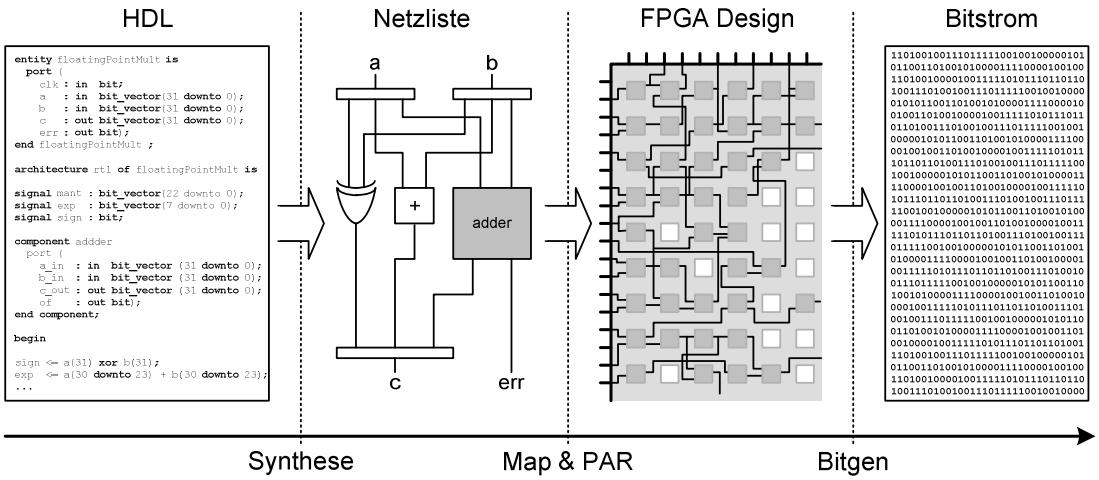

Abbildung 2-7: Vereinfachter Entwurfsablauf für statische FPGA-Systeme

tet). Somit entsteht eine vollständige Funktionsbeschreibung, die auch als *Design* bezeichnet wird. Hieraus kann nun in einem letzten Schritt ein Bitstrom erzeugt werden, mit dem durch Konfigurieren das ursprünglich beschriebene System auf das FPGA geladen wird. Gemäß dem Xilinx Werkzeug, das diesen Schritt übernimmt, ist er hier als *Bitgen* bezeichnet worden. Für den Abbildungsschritt von einer Netzliste zu einem FPGA-Design können Vorgaben bezüglich der Platzierung und des zu erzielenden Timings gemacht werden. Hierzu kann ein *User Constraints File (UCF)* in den Entwurf eingebunden werden, das die Vorgaben enthält.

Die dargestellten Entwurfsschritte müssen nicht für alle Systemkomponenten gemeinsam durchgeführt werden. Es ist möglich, einzelne Komponenten separat zu entwerfen und an geeigneter Stelle in das Gesamtsystem zu integrieren. Es ist üblich, dass Teile des Entwurfs vorsynthetisiert als Netzliste vorliegen. In Abbildung 2-7 ist dies für die Komponente *adder* angedeutet. Für sie muss hier eine weitere Netzliste vorliegen, die ihren internen Aufbau weiter beschreibt. Diese Netzliste wird bei der Synthese des Gesamtsystems in die hierarchisch übergeordnete Netzliste integriert. In ähnlicher Weise können bereits platzierte und verdrahtete Schaltungsteile in den Entwurfsablauf eingefügt werden. Solche Teil-Entwürfe werden *Makros* genannt. Sie werden beim PAR des Gesamtsystems als Ganzes platziert, wobei der innere Aufbau des Makros nicht verändert wird. Anschließend werden die Makroeingänge und -ausgänge mit dem Rest der Schaltung verbunden. Ein Makroeingang ist immer ein Eingang einer rekonfigurierbaren Ressource des FPGA, z.B. der Eingang eines Slice. Gleiches gilt für die Ausgänge.

Für die Entwurfseingabe gibt es inzwischen eine Reihe von Entwicklungsumgebungen, mit denen auf einer sehr abstrakten Ebene Systembeschreibungen erstellt werden können, aus denen automatisiert HDL-Code erzeugt wird. Ein Beispiel ist das *Xilinx Embedded Development Kit* (EDK [Xil08]), mit dem On-Chip-Prozessorsysteme auf FPGAs durch ein „Zusammenklicken“ von Prozessor, Speicher, Bussystemen und einer Reihe von Schnittstellen erstellt werden können. Für alle verwendbaren Systemkomponenten liegen parametrierbare HDL-Beschreibungen vor, die dann zu einem System zusammengefügt werden.

2.3.1 Prinzipieller Entwurfsablauf für dynamische Rekonfigurierung

Der statische Entwurfsablauf versucht, aus einer Verhaltensbeschreibung eine strukturelle Beschreibung zu erzeugen und schließlich eine Abbildung auf die Ressourcen eines FPGAs durchzuführen. Eine zeitliche Änderung der Strukturen ist nicht vorgesehen und bei klassischen FPGA-Systemen auch nicht möglich. Die dynamische Rekonfigurierung zielt allerdings genau auf eine zeitliche Veränderung der Systemstruktur ab, so dass für den Entwurf dynamisch rekonfigurierbarer Systeme der klassische Entwurfsablauf angepasst werden muss. Gegenüber einem statischen Entwurf kommen zwei Entwurfsschritte hinzu: Zum einen muss durch *Systempartitionierung* und *Floorplanning* die logische und geometrische Aufteilung des Systems in statische und dynamische Komponenten vollzogen werden. Zum anderen muss die Anbindung der dynamischen Komponenten an den Rest der Schaltung, wie z.B. Taktnetze oder Kommunikationsleitungen, sichergestellt werden. Abbildung 2-8 zeigt den Entwurfsablauf für dynamisch rekonfigurierbare Systeme nach einer bereits erfolgten Synthese des Systems. In dem gegebenen Beispiel soll ein Teil des Systems zwei unterschiedliche Funktionen annehmen, die durch zwei dynamische Komponenten (K1 und K2) realisiert werden. Für das Gesamtsystem muss hier eine Netzliste vorliegen, die neben den statischen Elementen einen Platzhalter (*Black Box*) für den Teil des Systems enthält, der zur Laufzeit rekonfiguriert werden soll. Für die beiden dynamischen Komponenten existiert je eine Netzliste. Die Schnittstellen der dynamischen Komponenten müssen mit den Schnittstellen der Black Box in der obersten Hierarchieebene (der sog. *Top-Level Entity*) übereinstimmen.

Mit mehreren Map- und PAR-Schritten werden aus den Netzlisten eine Initialsystem sowie partielle Schaltungen für die dynamischen Komponenten erstellt. Für jede dynamische Komponente können mehrere partielle Schaltungen unterschiedlicher Ausprägung erzeugt werden, die als Module bezeichnet werden (vgl. 3.1.2). In dem Beispiel in Abbildung 2-8 wurde für die Komponenten K1 und K2 je ein Modul (M1 und M2) generiert. Das Initialsystem wird durch eine vollständige Konfigurierung auf das FPGA geladen und konfiguriert somit alle FPGA-Ressourcen, unabhängig davon, ob sie für die Implementierung statischer oder dynamischer Komponenten verwendet werden. In Abbildung 2-8 werden vier CLB-Spalten am linken Rand des FPGA für statische Komponenten verwendet. Daneben sind zwei (fast) ungenutzte CLB-Spalten, in die die Module M1 und M2 geladen werden können. Entsprechend

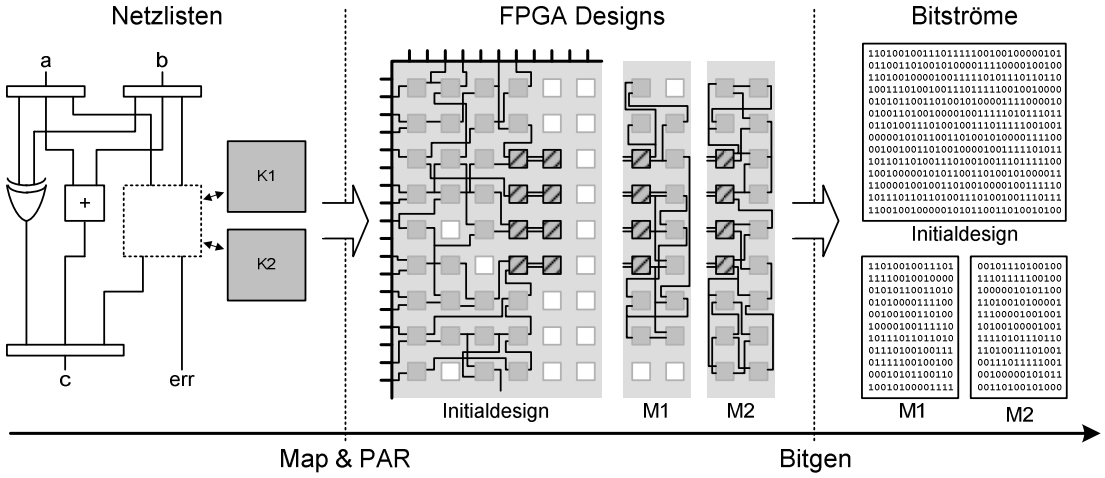

Abbildung 2-8: Vereinfachter Entwurfsablauf für dynamisch rekonfigurierbare FPGA-Systeme

der Komponenten, die durch einen FPGA-Bereich implementiert werden, unterscheidet man zwischen statischen und dynamischen Bereichen des FPGA. In Abbildung 2-8 umfasst der dynamische Bereich zwei CLB-Spalten. Die Größe von M1 und M2 ist durch den dynamischen Bereich begrenzt und kann hier somit nicht mehr als zwei CLB-Spalten betragen. Es ist möglich, dass das Initialdesign bereits dynamische Komponenten in den entsprechenden Bereichen enthält. In diesem Fall ist dies notwendig, damit nach der Initialkonfiguration ein funktionierendes System geladen ist. Die Schnittstellen zwischen statischen und dynamischen Bereichen sind nach dem PAR durch Routingressourcen des FPGAs realisiert. Um eine sichere Kommunikation garantieren zu können, müssen die Schnittstellen bei allen Modulen dieselben Ressourcen verwenden. Hierzu werden beim Entwurf *Kommunikationsmakros* verwendet (im Bild schraffiert dargestellt), die weiter unten beschrieben werden.

Abschließend wird aus dem Initialdesign ein Initialbitstrom generiert. Er enthält die Konfigurationsdaten für eine vollständige Konfigurierung, inklusive aller Speicherinhalte sowie die Information über die Taktnetze. Analog dazu werden für die Module partielle Bitströme erzeugt. Im einfachsten Fall wird ein Bitstrom pro Modul generiert. Die Größe der Bitströme ist dabei abhängig von der Größe und der Beschaffenheit der Module.

Xilinx unterstützt die dynamische und partielle Rekonfigurierung durch Werkzeuge und entsprechende Dokumentation. Inzwischen existieren mehrere Entwurfsabläufe für dynamische Rekonfigurierung (z.B. [Xil02] und [Xil06]), die zwar alle im Wesentlichen nach dem hier beschrieben Muster arbeiten, aber dennoch unterschiedliche Eigenschaften bezüglich der dynamischen Rekonfigurierung zur Laufzeit aufweisen. Wichtig ist, dass in dieser Arbeit mit „dynamischer Rekonfigurierung“ ausschließlich die *modulare* Rekonfigurierung gemeint ist. Wie oben beschrieben, werden dabei zusammenhängende Bereiche des verwendeten FPGAs ausgewiesen, in die unterschiedliche Module geladen werden können. Daneben existiert die *differentielle* Rekonfigurierung (*differential based partial reconfiguration*), die für Systeme vorgesehen ist, an denen zur Laufzeit nur minimale Änderungen vorgenommen werden sollen. Dazu wird zunächst eine Version des Systems komplett implementiert. Anschließend werden für alle möglichen Änderungen so genannte *guided PARs* durchgeführt, bei denen nur die Änderungen zum Originaldesign neu platziert und verdrahtet werden. Für die entstehenden Variationen werden dann Differenzbitströme erzeugt, die nur die vom Originalbitstrom unterschiedlichen Frames enthalten. Diese partiellen Bitströme werden dann für die Rekonfigurierung verwendet. Im Allgemeinen wird bei differentieller Rekonfigurierung allerdings keine zusammenhängende Fläche des FPGAs rekonfiguriert.

2.3.2 Kommunikationsmakros

Mit allen Modulen, die in ein System geladen werden, muss in irgendeiner Form kommuniziert werden. Hierzu müssen Module mit angrenzenden statischen Bereichen oder anderen, bereits geladenen Modulen über Signalleitungen miteinander verbunden werden. Signale, die die Modulgrenzen kreuzen, werden in dieser Arbeit globale Signale genannt. Signale, die sich ausschließlich innerhalb eines Moduls oder statischen Bereichs befinden, heißen lokale Signale. Gleiches gilt für die Signalleitungen, über die die Signale übertragen werden.

Bei globalen Signalleitungen liegt immer mindestens ein Ende in einem dynamischen Bereich, das daher in der Regel von mehreren verschiedenen Modulen benutzt wird. Dabei muss sichergestellt werden, dass für den Übergang vom dynamischen in den statischen Bereich bei

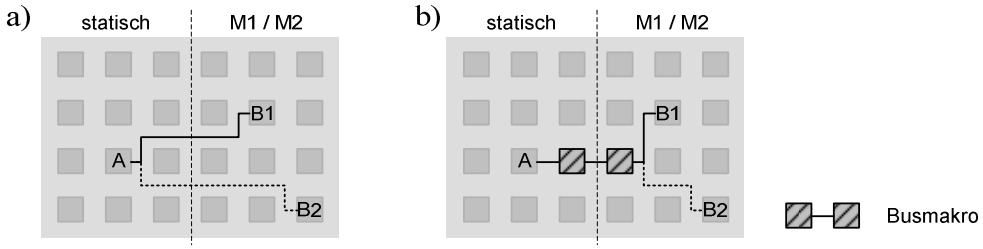

Abbildung 2-9: Routing globaler Signale, a) ohne Busmakro, b) mit Busmakro

allen Modulen dieselben Routingressourcen verwendet werden. Abbildung 2-9 a) zeigt den Übergang von einem statischen in einen dynamischen Bereich und das Routing eines globalen Signals S zwischen einem Element A im statischen Bereich und einem Element B, das Teil zweier Module M1 und M2 ist. Da M1 und M2 unabhängig voneinander platziert und verdrahtet werden, wird B in beiden Modulen sehr wahrscheinlich an unterschiedliche Positionen platziert, die hier mit B1 und B2 gekennzeichnet sind. Aus unterschiedlichen Positionen für B resultiert automatisch auch eine unterschiedliche Verdrahtung von S zwischen A und B bei der Implementierung der zwei Module. Zur Veranschaulichung dieser Situation sind die Implementierungen von M1 und M2 in Abbildung 2-9 überlagert dargestellt. Das dadurch entstehende Problem wird hier offensichtlich: Entweder muss das Routing des statischen Bereichs je nach geladenem Modul angepasst werden (wodurch der statische Bereich nicht mehr „statisch“ ist), oder alle vorkommenden Routingvarianten müssen im statischen Bereich gleichzeitig implementiert werden. Letzteres wird zum einen nicht von den Entwicklungswerkzeugen unterstützt und kann vor allem bei Signalen, die in den Modulen getrieben werden, zu einem erheblichen zusätzlichen Schaltungsaufwand führen. Um diese Probleme zu umgehen, werden *Kommunikationsmakros* für die Anbindung der Signale verwendet.

In Abbildung 2-9 b) wurde für das beschriebene Beispiel ein Kommunikationsmakro für das Routing des globalen Signals verwendet (schraffiert dargestellt). Es wurde (mithilfe einer Platzierungsvorgabe im UCF) so platziert, dass ein Teil des Makros im statischen Bereich, ein anderer Teil im rekonfigurierbaren Bereich liegt. Dieses Makro wurde nun bei der Implementierung der Module M1 und M2 für das Routing des Signals S verwendet. Da das Routing innerhalb des Makros nicht mehr verändert wird, kreuzt das Signal S die Grenze zwischen statischem und rekonfigurierbarem Bereich immer an derselben Stelle, bzw. immer mit derselben Routingressource. Für den statischen Bereich gibt es dadurch nur noch eine Verschaltung, die für beide Module M1 und M2 identisch ist. Erst innerhalb der Module ändert sich das Routing.

Es wurde oben bereits erwähnt, dass sich Ein- und Ausgänge eines Makros an Logik-Ressourcen des FPGA befinden müssen. Obwohl hier nur das Routing eines Signals fest vorgegeben werden soll, werden trotzdem auch zwei Logikelemente benötigt, an denen Makroeingang und -ausgang definiert werden können. Diese Logikelemente sind ein zusätzlicher Ressourcenbedarf des Gesamtsystems, der sich durch die Verwendung von Makros ergibt. In der Regel sind dies zusätzlich belegte Slices. Im einfachsten Fall werden die Signale nur durch die Ein-/Ausgangsslices hindurch geroutet, und die Flipflops und LUTs bleiben ungenutzt. Unter Umständen können aber auch weitere Schaltungsteile in diese Slices gepackt werden. Wie groß diese Kosten werden können, die letztlich durch dynamische Konfigurierung verursacht werden, wird in Kapitel 4 behandelt.

Eine weitere Möglichkeit, die benötigten Verbindungen zwischen statischen und dynamischen Komponenten herzustellen, ist das Routing globaler Signale nach der Konfigurierung durchzuführen. Diese Möglichkeit hat sich jedoch als bei weitem zu rechenaufwändig herausgestellt, als dass sie zur Laufzeit mit vertretbarem Aufwand gelöst werden könnte. Es gibt allerdings Ansätze, die versuchen den Rechenaufwand zu reduzieren, indem sie die Routingmöglichkeiten und somit die Komplexität der Berechnung reduzieren. Hübner et al. [HSB06] konnten zeigen, dass ein Online-Routing durchgeführt werden kann. Allerdings liegt die zusätzliche Verzögerung durch das Routing in nicht akzeptablen Bereichen, so dass Online-Routing derzeit noch keine Option für die Anbindung der Module darstellt. Stattdessen wird die sichere Anbindung der Module schon während des Entwurfs gewährleistet.

Es gibt überdies Ansätze, bei denen die Verbindungen zwischen dynamischen Komponenten und statischer Hardware über einen weiteren Baustein (z.B. ein CPLD[17] oder ein weiteres FPGA) außerhalb des FPGAs herzustellen [HLP02, BMA05]. Dieser Ansatz wird hier nicht weiter betrachtet, da er dem allgemeinen Trend, möglichst viele Funktionen auf einem Chip zu integrieren, genau entgegensteht. Da die Module hierbei zudem immer über IO-Blöcke kommunizieren müssen, können freie Platzierungsverfahren mit diesem Verfahren nur sehr eingeschränkt realisiert werden.

2.3.3 Offene Systeme

Mithilfe der dynamischen Rekonfigurierung ist es möglich, *offene* FPGA-Systeme zu erstellen. Als „offen“ werden Systeme bezeichnet, die zur Laufzeit ihnen bislang unbekannte Komponenten einbinden können. Offene Systeme zeichnen sich durch eine noch größere Flexibilität aus, da beim Entwurf des Systems nicht alle Systemkomponenten bekannt sein müssen, die jemals in das System geladen werden könnten. Der Entwurf des statischen Systems kann somit weitgehend von dem Entwurf der dynamischen Komponenten getrennt werden. Hierzu muss ein offenes System bekannte Schnittstellen zur Verfügung stellen, die dynamische Systemkomponenten implementieren können. Konkret müssen für die Implementierung der dynamischen Komponenten der FPGA-Typ, die dynamischen Bereiche auf dem FPGA und die verwendete Kommunikationsinfrastruktur (Busmakros) bekannt sein. Außerdem müssen Methoden implementiert werden, die die logische Einbindung dynamischer Systemkomponenten, beispielsweise die Zuweisung eines Adressraums im System, zur Laufzeit ermöglichen. Für die Implementierung der Module M1 und M2 in Abbildung 2-8 muss nur bekannt sein, dass die Spalten 5 und 6 des verwendeten FPGAs für dynamische Komponenten zur Verfügung stehen. Mit dem Busmakro und seiner Position können dann die Module erzeugt werden. Bei der Beschreibung der Module auf funktionaler Ebene muss dabei sichergestellt werden, dass die Kommunikationsschnittstelle korrekt verwendet und das Protokoll eingehalten wird.

[17] Complex Programmable Logic Device: Konfigurierbarer Logikbaustein.

3 Dynamisch rekonfigurierbare FPGA-Systeme

Dynamische und partielle Konfigurierung kann auf vielfältige Weise durchgeführt werden. Für die *partielle* Rekonfigurierung, also das Verändern von Teilen der FPGA-Konfiguration, kann eine fast beliebige Aufteilung der verfügbaren Ressourcen gewählt werden. Ebenso gibt es für die *dynamische* Rekonfigurierung, also das Verändern der Konfiguration zur Laufzeit, verschiedene Realisierungsmöglichkeiten. Da es hierbei mehr und weniger sinnvolle Lösungen gibt, die oft unterschiedliche technische Voraussetzungen mit sich bringen, werden in diesem Kapitel die grundsätzlichen Realisierungsoptionen aufgezeigt und diskutiert. Zunächst wird hierfür eine Modellierung rekonfigurierbarer Systeme vorgenommen, die alle wichtigen Elemente definiert und ihnen eindeutige Namen zuweist. Darauf aufbauend werden die Verfahren für die partielle und dynamische Konfigurierung behandelt.

3.1 Modellierung

Die für dynamische Rekonfigurierung verwendeten rekonfigurierbaren Ressourcen wie auch die dynamischen Komponenten selbst müssen sowohl zur Laufzeit, als auch beim Entwurf in irgendeiner Form verwaltet werden. In dieser Arbeit wird ein einheitliches Modell entwickelt, dass die Grundlage der in Kapitel 5 vorgestellten Laufzeit- und Entwicklungsumgebungen bildet. Darüber hinaus definiert das Modell eine einheitliche Nomenklatur für die in dieser Arbeit verwendeten Begriffe. Diese Nomenklatur orientiert sich an bestehenden Begriffen. Insbesondere werden die von Xilinx im *Early Access Partial Reconfiguration User Guide* [Xil06] verwendeten Namen verallgemeinert verwendet.

3.1.1 Modellierung dynamischer Ressourcen

Um dynamische Komponenten zur Laufzeit in das System laden zu können, müssen beim Entwurf rekonfigurierbare Ressourcen dafür ausgewiesen werden. Hierfür kommen alle rekonfigurierbaren Logik- und Speicherelemente eines FPGAs in Frage.

3.1.1.1 Rekonfigurierbare Ressourcen

Die für dynamische Rekonfigurierung zur Verfügung stehenden Ressourcen sind durch das verwendete FPGA vorgegeben. Jedes FPGA besitzt eine Grundmenge T_{FPGA} verschiedener Ressourcentypen τ, in der Regel Logik- oder Speicherelemente, mit

$$T_{FPGA} = \{\tau_1, \tau_2, ..., \tau_n\} \tag{3}$$

bei n verschiedenen Grundelementen des FPGAs. Diese Grundmenge kann nicht immer eindeutig aus der FPGA-Architektur abgeleitet werden. Es macht allerdings Sinn, sich bei den Grundelementen τ an die vom FPGA-Hersteller verwendeten Kernelemente zu halten, z.B. Slices, BlockRAMs und eingebettete Multiplizierer bei Virtex-II FPGAs. Jede Ressource ρ auf einem FPGA kann dann eindeutig beschrieben werden als 2-Tupel, bestehend aus ihrem Ressourcentyp τ sowie ihren Koordinaten k auf dem FPGA. Die Koordinaten k sind gegeben durch ein Koordinatensystem, dass für das FPGA festgelegt wird. Von Herstellerseite wird üblicherweise bereits ein Koordinatensystem verwendet, das hier sinnvoll übernommen wer-

den kann. Prinzipiell können beliebige Koordinatensysteme verwendet werden, solange sie allen Ressourcen eindeutige Koordinaten zuweisen. Die Ressourcen eines FPGAs werden dabei oft in der Draufsicht betrachtet. Für die relative Lage der Ressourcen können somit Begriffe wie „oben", „unten", „rechts" und „links" verwendet werden. Ebenfalls gebräuchlich ist die Verwendung von Himmelsrichtungen, wobei, wie bei geographischen Karten üblich, dem oberen Rand des FPGAs der Norden zugeordnet wird. Xilinx verwendet ein Koordinatensystem, das sich an den Zeilen und Spalten des FPGAs orientiert. Beispielsweise das Slice in der oberen linken Ecke eines XC2V4000 FPGAs die Koordinate (0, 0), das Slice in der unteren rechten Ecke die Koordinate (143, 159).

3.1.1.2 Rekonfigurierbare Flächen

Für die partielle Rekonfigurierung müssen zusammenhängende Flächen von Ressourcen ausgewiesen werden. Jede Fläche kann als Menge der in ihr enthaltenen Ressourcen ausgedrückt werden, wobei jede Ressource wie erwähnt als 2-Tupel aus Ressourcetyp und Koordinaten beschrieben werden kann:

$$R = \{\rho_1, \rho_2, ..., \rho_m\} = \{(\tau(\rho_1), k(\rho_1)), (\tau(\rho_2), k(\rho_2)), ..., (\tau(\rho_m), k(\rho_m))\} \quad (4)$$

Für die dynamische Rekonfigurierung ist vor allem interessant, wie viele Elemente jedes Ressourcetyps in R enthalten sind. Fasst man dafür in den Teilmengen R_1, …, R_n die Ressourcen gleichen Typs zusammen, also

$$R_i = \left\{ (\tau(\rho_j), k(\rho_j)) \mid (\tau(\rho_j), k(\rho_j)) \in R \wedge \tau(\rho_j) = \tau_i \right\},\ j \in \{1, ..., m\}, i \in \{1, ..., n\} \quad (5)$$

dann ist die Mächtigkeit dieser Teilmengen ist durch $\eta_i(R) = |R_i|$ gegeben. $\eta_1(R)$ könnte so beispielsweise die Anzahl der in R enthaltenen Slices angeben. Mit dem n-Tupel $\bar{r}(R) = (\eta_1(R), \eta_2(R), ..., \eta_n(R))$ ist daher eine quantitative Beschreibung aller in R enthaltenen Ressourcen gegeben.

Definition 1: Hat ein FPGA die Grundmenge T_{FPGA} verschiedener, rekonfigurierbarer Grundelemente mit $|T_{FPGA}| = n$ und ist R eine Menge rekonfigurierbarer Ressourcen, dann bezeichnet $\bar{r}(R) = (\eta_1(R), \eta_2(R), ..., \eta_n(R))$ die quantitative Anzahl der in R vorkommenden Grundelemente, wobei η_i die Häufigkeit des Ressourcentyps τ_i angibt. $\bar{r}$ wird *Anzahl belegter Ressourcen* genannt.

Im Zusammenhang mit dynamisch rekonfigurierbaren Systemen werden ausschließlich Ressourcenmengen R betrachtet, die eine zusammenhängende Fläche des FPGAs bilden. Macht man des Weiteren die Einschränkung, dass diese Flächen eine rechteckige Form haben müssen, kann eine Ressourcenmenge R in ein 3-Tupel $\overline{R}\ (\bar{r}, \Phi, k)$ überführt werden. Da die beinhalteten Ressourcen hier ohne ihre Koordinaten betrachtet werden, wird die Anordnung Φ eingeführt, die die geometrische Lage der verwendeten Ressourcen zueinander beschreibt. Der gesamten Ressourcenfläche $\overline{R}$ wird in dieser Darstellung durch die Koordinaten k eine eindeutige Position auf dem FPGA zugewiesen. Hierfür kann das Koordinatensystem der in R enthaltenen Ressourcen verwendet werden. Allerdings muss ein einheitlicher Standard definiert werden, welche der Ressourcen in R dafür verwendet wird. Bei rechteckigen Flächen bietet es sich an, immer die Koordinaten einer Eck-Ressource als Referenz anzugeben, um eindeutige Angaben zu garantieren.

Für das Arbeiten mit rekonfigurierbaren Flächen beim Entwurf und insbesondere zur Laufzeit ist die Darstellung der Flächen in Form des Tupels $(\bar{r}, \Phi, k)$ nicht gut geeignet, da sie immer noch mehr Informationen enthält als tatsächlich benötigt werden. Insbesondere wird die genaue Anordnung der enthaltenen Ressourcen nicht benötigt. Daher wird an dieser Stelle die *Fläche A* eingeführt, die eindeutig aus der Ressourcenfläche $\overline{R}$ abgeleitet werden kann. Mit den Darstellungsformen *R*, $\overline{R}$ und *A* kann dieselbe Menge rekonfigurierbarer Ressourcen auf unterschiedliche Weise beschrieben werden. *R* und $\overline{R}$ werden im Folgenden nur noch für die Begriffsdefinitionen verwendet, in der Praxis wird ausschließlich *A* verwendet. Die Ressourcen *R* einer Fläche *A* werden dabei mit *R*(*A*) bezeichnet.

Definition 2: Eine rekonfigurierbare Fläche *A* ist ein 5-Tupel $(\bar{r}, \varphi, w, h, k)$. Es beschreibt einen rechteckigen Bereich auf einem FPGA, dessen Breite durch *w* und dessen Höhe durch *h* gegeben sind. Als Referenzkoordinaten *k* dieser Fläche werden die Koordinaten derjenigen Ressource verwendet, die sich in der oberen, linken Ecke der Fläche *A* befindet. Mit $\bar{r}$ sind die quantitativen Mengenangaben der in *A* enthaltenen rekonfigurierbaren Grundelemente gegeben. φ bezeichnet den Flächentyp der Fläche *A*.

Der hier eingeführte Flächentyp φ wird verwendet, um Flächen mit gleichen Ausmaßen und gleicher Anzahl verwendeter Grundelemente, aber unterschiedlichen Anordnungen der Ressourcen unterscheiden zu können. Für zwei Flächen $A_1 = (\bar{r}_1, \varphi_1, w_1, h_1, k_1)$ und $A_2 = (\bar{r}_2, \varphi_2, w_2, h_2, k_2)$, die den Ressourcenflächen $\overline{R}_1 = (\bar{r}_1, \Phi_1, k_1)$ und $\overline{R}_2 = (\bar{r}_2, \Phi_2, k_2)$ entsprechen, gilt:

$$\varphi_1 = \varphi_2 \Leftrightarrow (\bar{r}_1 = \bar{r}_2) \wedge (\Phi_1 = \Phi_2) \tag{6}$$

A_1 und A_2 heißen *disjunkt* oder *überlappungsfrei*, wenn gilt:

$$R(A_1) \cap R(A_2) = \{\ \}. \tag{7}$$

Darüber hinaus ist die *Kongruenz* zweier Flächen ein wichtiges Merkmal. Sie ist gegeben, wenn zwei nicht-identische Flächen $A_1(\bar{r}_1, \varphi_1, w_1, h_1, k_1)$ und $A_2(\bar{r}_2, \varphi_2, w_2, h_2, k_2)$ die gleichen geometrischen Eigenschaften haben, wenn also gilt:

$$(\bar{r}_1 = \bar{r}_2) \wedge (\varphi_1 = \varphi_2) \wedge (w_1 = w_2) \wedge (h_1 = h_2) \wedge (k_1 \neq k_2) \tag{8}$$

Die hier verwendete Beschreibung der äußeren Eigenschaften einer Fläche über Höhe *h*, Breite *w* und Koordinate *k* eignet sich besonders beim Auffinden geeigneter Modulpositionen zur Laufzeit oder bei Simulationen. Alternativ können Flächen als 4-Tupel $(\bar{r}, \varphi, k_{ol}, k_{ur})$ beschrieben werden, wobei k_{ol} und k_{ur} die Koordinaten der oberen linken bzw. unteren rechten Ressource bezeichnen. Diese Beschreibungsform kommt wiederum den bestehenden Entwurfswerkzeugen von Xilinx entgegen. Es ist leicht ersichtlich, dass beide Formen bei rechteckigen Flächen einfach ineinander überführt werden können.

3.1.1.3 Ausweisung dynamischer Ressourcen

Bevor dynamische Systemkomponenten auf rekonfigurierbare Hardware abgebildet werden können, müssen von den verfügbaren Ressourcen eines FPGAs einige ausgewählt werden, in die zur Laufzeit ausschließlich dynamische Komponenten geladen werden können.

Definition 3: Eine Menge rekonfigurierbarer Ressourcen *R* heißt *rekonfigurierbarer Bereich* R_D, wenn alle in ihr enthaltenen Ressourcen für dynamische Rekonfigurierung genutzt werden können. Existieren mehrere rekonfigurierbare Bereiche in einem System, so sind ihre Flächen in jedem Fall disjunkt. Dynamische Komponenten können immer nur in genau einen rekonfigurierbaren Bereich geladen werden.

Ein rekonfigurierbarer Bereich kann wie eine Fläche als 5-Tupel $(\bar{r}, \varphi, W, H, k)$ beschrieben werden. Für die Breite und Höhe des Bereichs werden jedoch Großbuchstaben (*W* bzw. *H*) verwendet. Während des Entwurfs müssen den dynamischen Komponenten Ressourcen zugewiesen werden, die innerhalb *eines* dynamischen Bereichs liegen. Die oben erwähnten Grundelemente eines FPGAs (Slices, BlockRAMs, etc.) gelten dabei für die heutigen Entwurfswerkzeuge als atomare Einheiten, die einer Komponente entweder ganz oder gar nicht zugewiesen werden können. Das heißt: Benötigt eine Komponente fünf Slices können ihr auch genau fünf zusammenhängende Slices zugeteilt werden. Dies entspricht einer feingranularen Ressourcenzuweisung, von der in der Praxis aus Gründen, die weiter unten erläutert werden, kein Gebrauch gemacht wird. Stattdessen werden vom Entwickler Flächen ausgewiesen, die beim Entwurf als atomare Ressourcen-Einheiten dienen, wodurch eine gröbere Granularität entsteht. Solche Flächen werden *Kacheln* (oder engl. *tiles*) genannt.

Definition 4: Eine Kachel ist eine rechteckige Fläche *A*, auf deren Ressourcen $R = R(A)$ dynamische Komponenten abgebildet werden können. Dabei kann nie nur eine Teilmenge von *R* der dynamischen Komponente zugewiesen werden. Eine Kachel stellt daher eine unteilbare Einheit bei der dynamischen Rekonfigurierung dar.

Für den Entwurf der dynamischen Komponenten muss ein rekonfigurierbarer Bereich immer aus einer Menge disjunkter Kacheln bestehen, also

$$R_D = \{R_1, R_2, ..., R_n\} \tag{9}$$

Je nach Gestaltung der Kacheln entsteht für den rekonfigurierbaren Bereich eine mehr oder weniger regelmäßige Rasterung. Sie ist im Allgemeinen umso regelmäßiger, je weniger unterschiedliche Kacheln es gibt. Bei der Abbildung einer dynamischen Komponente auf einen Teil eines rekonfigurierbaren Bereichs muss die durch die Kachelung gegebene Rasterung eingehalten werden. Gültige Flächen für die dynamische Rekonfigurierung bestehen daher immer aus einer ganzen Zahl aneinander grenzender Kacheln.

Definition 5: Eine für dynamische Rekonfigurierung benutzte Fläche muss immer komplett innerhalb eines rekonfigurierbaren Bereichs liegen und darf nur aus den in ihm definierten Kacheln zusammengesetzt sein.

Wie in Kapitel 4 noch ausführlich dargelegt wird, stellt die Kommunikationsinfrastruktur einen statischen Bestandteil des Systems dar, der allerdings mithilfe gewöhnlicher Logik- und Routingressourcen innerhalb der dynamischen Bereiche realisiert werden muss. Da diese Ressourcen den dynamischen Komponenten nicht mehr zur Verfügung stehen, werden sie modelliert, indem die verfügbaren Ressourcen der Kacheln entsprechend reduziert werden.

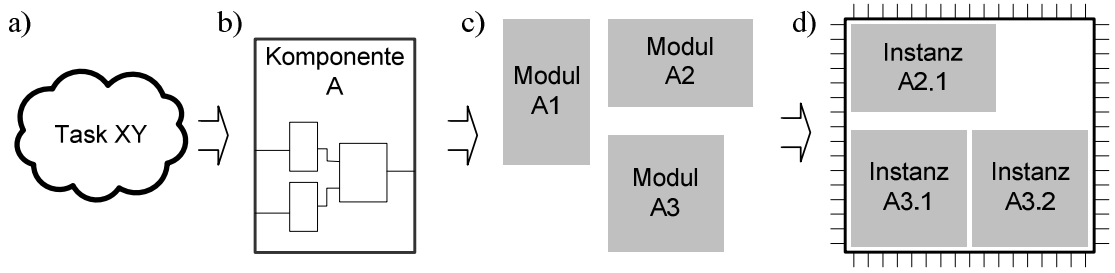

Abbildung 3-1: a) abstrakte Aufgabenbeschreibung, b) strukturelle Beschreibung einer Komponente c) konkrete Realisierungen (Module), d) geladene Modulinstanzen zur Laufzeit

3.1.2 Modellierung dynamischer Komponenten

Für die verschiedenen Erscheinungsformen einer dynamischen Komponente zur Laufzeit und während des Entwurfs wird im Folgenden die Nomenklatur verwendet, die in Abbildung 3-1 dargestellt ist. Die abstrakteste Erscheinungsform ist dabei die Beschreibung der Aufgabe (*Task*).

Definition 6: Eine Task beschreibt eine Aufgabe, die in einem dynamisch rekonfigurierbaren System erledigt oder wahrgenommen werden soll.

Die Beschreibung einer Task kann sehr abstrakt oder auch bereits in einer Hardwarebeschreibungssprache erfolgen. Beispielsweise könnte hier die Verschlüsselung von Daten gemäß einem definierten Standard beschrieben sein. Als *Komponente* wird dann eine konkrete Beschreibung dieser Verschlüsselungseinheit bezeichnet, die bereits ein fest definiertes Verhalten und eine fest definierte Struktur besitzt.

Definition 7: Eine Komponente ist ein 2-Tupel $(f, \tilde{r})$ und stellt eine strukturelle Beschreibung einer Task dar, mit nicht mehr veränderbarer Funktion *f* und den Ressourcenanforderungen $\tilde{r}$.

Eindeutig beschrieben und repräsentiert wird eine Komponente entweder durch eine Netzliste oder durch eine HDL-Beschreibung und eine Synthesevorschrift (z.B. Syntheseskript), aus denen durch Synthese genau eine Netzliste erzeugt werden kann. $\tilde{r}$ gibt die *Anzahl benötigter Ressourcen* an. In Analogie zu der Anzahl belegter Ressourcen einer Fläche werden die benutzten Ressourcen einer Komponente nach Grundelementen aufgeschlüsselt, so dass $\tilde{r}$ ein n-Tupel $(\eta_1, \eta_2, ..., \eta_n)$ ist.

Es ist möglich, zu einer Aufgabe mehrere Komponenten $K_1, ..., K_n$ zu erzeugen, die unterschiedliche Ressourcenanforderungen und Leistungsmerkmale aufweisen, z.B. aufgrund unterschiedlicher Pipelinetiefen oder Datenbreiten. Es kann durchaus sinnvoll sein, in einem System unterschiedliche Komponenten für dieselbe Aufgabe bereit zu halten. Zur Laufzeit kann z.B. zwischen einer großen, leistungsstarken und einer kleinen, weniger performanten Komponente gewählt werden, je nach Anforderung oder verfügbaren Ressourcen.

Jede konkrete Implementierung einer Komponente auf FPGA-Ressourcen wird als *Modul* bezeichnet. Bei der Implementierung wird der Komponente *K* eine rekonfigurierbare Fläche *A* zugewiesen. Durch Platzieren und Verdrahten werden die Netzlistenelemente von *K* auf die Ressourcen *R*(*A*) verteilt und miteinander verbunden. Ist dieser Schritt erfolgreich, so ist ein

gültiges Modul *M* zu *K* entstanden. Schlägt er fehl, muss das Platzieren und Verdrahten mit einer anderen Fläche wiederholt werden.

Definition 8: Wird eine Komponente $K = (f, \tilde{r}_K)$ erfolgreich auf einer Fläche $A = (\bar{r}_A, \varphi_A, w_A, h_A, k_A)$ platziert und verdrahtet, so wird die dadurch entstandene Schaltung als Modul *M* der Komponente *K* bezeichnet. Das Modul *M* ist ein 6-Tupel $(f_M, h_M, w_M, \tilde{r}_M, \bar{r}_M, \varphi_M)$, mit $f_M = f_K$, $h_M = h_A$, $w_M = w_A$, $\tilde{r}_M = \tilde{r}_K$, $\bar{r}_M = \bar{r}_A$, und $\varphi_M = \varphi_A$.

Ein Modul erbt von der Komponente, aus der es erzeugt wurde, somit die abstrakte Funktion *f* und die Anzahl benötigter Ressourcen $\tilde{r}$, die nun, nach der Implementierung, *Anzahl benutzter Ressourcen* genannt wird. Da Komponenten meist nicht optimal auf die verfügbaren Ressourcen abgebildet werden können, ist im Allgemeinen die Anzahl benutzter Ressourcen ungleich der Anzahl belegter Ressourcen, also $\tilde{r} \neq \bar{r}$. Mit $\tilde{r}$ und $\bar{r}$ können somit Maße für die Ressourcenauslastung definiert werden (vgl. 6.1). Von der Fläche *A*, in der ein Modul implementiert ist, erbt das Modul alle geometrischen Eigenschaften. Allerdings ist das Modul unabhängig von den Koordinaten von *A*. Das heißt, dass mit jeder zu *A* kongruenten Fläche A' dasselbe Modul *M* erzeugt werden kann. Außerdem kann *M* zur Laufzeit in jede zu *A* kongruente Fläche geladen werden. Einem Modul sind daher keine Koordinaten zugewiesen. Diese Eigenschaft ergibt sich aus dem Aufbau des Konfigurationsspeichers, auf den später genauer eingegangen wird.

Die Parameter *f* und φ bezeichnen zwei abstrakte Eigenschaften eines Moduls. Die Funktion *f* wird in dieser Arbeit nur dafür verwendet, Module ihren entsprechenden Komponenten zuordnen zu können, und somit Module mit identischen geometrischen Eigenschaften unterscheiden zu können. Der Typ φ beschreibt den Aufbau der Ressourcen, auf die ein Modul abgebildet wurde. Aufgrund der Heterogenität der FPGAs kann es vorkommen, dass zu einer Komponente zwei oder mehr Module mit der gleichen Breite *w* und Höhe *h* auf nichtidentischen Flächen erzeugt werden. φ wird dann gebraucht, um zwischen diesen Modulen unterscheiden zu können.

Abbildung 3-1 c) zeigt beispielhaft die Generierung dreier verschiedener Module für die Komponente *A*. Die Erzeugung mehrerer Module für eine Komponente kann aus verschiedenen Gründen erfolgen. Zum einen kann das Vorhandensein verschiedener Modulformen das Platzieren zur Laufzeit erleichtern, zum anderen kann ein Modul allein nicht alle Ressourcen eines rekonfigurierbaren Bereichs erschließen, wenn heterogene FPGAs verwendet werden. Beide Fälle werden weiter unten genauer betrachtet.

Die letzte hier beschriebene Erscheinungsform einer dynamischen Komponente ist die *Modulinstanz*. Sie entsteht in einem rekonfigurierbaren Bereich, wenn zur Laufzeit ein Modul an eine seiner möglichen Modulpositionen geladen wird. Eine Modulinstanz ist ein 4-Tupel (f, k, h, w). *f* wird auch hier für die Zuordnung zu einer Systemkomponente benötigt. *K* bezeichnet die eindeutigen Modulkoordinaten innerhalb aller dynamischen Bereiche, *h* und *w* bezeichnen Höhe und Breite der Modulinstanz. Unter Umständen können gleichzeitig mehrere Instanzen eines Moduls geladen werden. Man spricht dann von *Mehrfachinstanziierung* eines Moduls. Analog dazu kann auch von Mehrfachinstanziierung einer Komponente gesprochen werden, wenn mehrere Instanzen von unterschiedlichen Modulen derselben Komponente geladen sind.

Mehrfachinstanziierung ist nur dadurch möglich, dass Module an verschiedene Positionen geladen werden können. Die Module selbst beinhalten noch keine Position, sie wird erst zur Laufzeit gewählt. Unabhängig davon, mithilfe welcher Fläche *A* sie beim Entwurf erzeugt wurden, können sie zur Laufzeit auf alle zu *A* kongruenten Flächen geladen werden. Jede für ein Modul *M* gültige Fläche wird *gültige Modulposition p für M* genannt. Im Umkehrschluss zu obiger Definition gilt somit folgende Definition:

Definition 9: Eine Fläche $A = (\bar{r}_A, \varphi_A, w_A, h_A, k_A)$ heißt *gültige Modulposition p* eines Moduls $M = (f_M, h_M, w_M, \tilde{r}_M, \bar{r}_M, \varphi_M)$, falls $\bar{r}_A = \bar{r}_M$, $\varphi_A = \varphi_M$, $w_A = w_M$ und $h_A = h_M$ gilt.

Die Menge $P_M = \{p_1, p_2, ..., p_n\}$ ist eine Menge gültiger Modulpositionen, wenn alle in ihr enthaltenen Positionen gültige Modulpositionen des Moduls *M* sind. Sie heißt *vollständig*, wenn sie alle existierenden gültigen Positionen enthält. Soll zur Laufzeit eine dynamische Komponente in das System geladen werden, wird von der Laufzeitumgebung für ein Modul *M* dieser Komponente eine geeignete Modulposition aus *p* ausgewählt, die dann entsprechend konfiguriert wird. Die Repräsentation eines Moduls zur Laufzeit ist daher ein partieller Bitstrom. Nach diesem Modell ist sowohl das Modul als auch sein Bitstrom positionsunabhängig, erst zur Laufzeit werden die nötigen Informationen hinzugefügt. In der Praxis kann jedoch keine Implementierung einer Komponente zu einem Modul vorgenommen werden, ohne ihm dabei eine Position in einem rekonfigurierbaren Bereich zuzuweisen, da die verfügbaren Entwurfswerkzeuge keinen anderen Entwurf unterstützen. Um diesem Umstand Rechnung zu tragen, wird in dieser Arbeit das *positionierte Modul* $\overline{M}$ verwendet. Ein positioniertes Modul enthält gegenüber normalen Modulen als zusätzlichen Parameter die Koordinaten k_m. $\overline{M}$ wird somit für genau eine mögliche Modulposition *p* implementiert. Ein für $\overline{M}$ generierter, partieller Bitstrom kann ohne Weiteres nur für die Konfiguration der Modulposition *p* verwendet werden. Mit den bestehenden Entwurfswerkzeugen muss für jede mögliche Modulposition ein positioniertes Modul samt zugehörigem Bitstrom erzeugt werden, um ein Modul an alle möglichen Modulpositionen laden zu können. Mithilfe der weiter unten beschriebenen Bitstrommanipulation kann dem entgegengewirkt werden, so dass im besten Fall nur ein partieller Bitstrom pro Modul erzeugt werden muss, aus dem zur Laufzeit alle möglichen positionierten Module erzeugt werden können.

3.2 Partielle und dynamische Rekonfigurierung

Das Schlüsselmerkmal partiell und dynamisch rekonfigurierbarer FPGAs ist die Möglichkeit, den Konfigurationsspeicher teilweise (partiell) und zur Laufzeit eines auf dem FPGA befindlichen Systems (dynamisch) verändern zu können. In den folgenden Abschnitten wird erläutert, welche Wahlmöglichkeiten es bei der Nutzung dieser Eigenschaften gibt. Bezüglich der *partiellen* Rekonfigurierung muss beim Systementwurf durch eine *Systempartitionierung* entschieden werden, welche Bereiche eines FPGAs für dynamische Systemkomponenten bereitgestellt werden und welche Kachelungen in diesen Bereichen verwendet werden. Dabei muss der Aufbau des Konfigurationsspeichers berücksichtigt werden. Je nach Kachelung und Granularität des Konfigurationsspeichers kann es unterschiedliche Ergebnisse hinsichtlich der Bitstromgrößen sowie der zu erwartenden Konfigurierungsdauer geben.

Je nach Partitionierung der Systemressourcen sind verschiedene Techniken möglich, mit denen die *dynamische* Rekonfigurierung durchgeführt wird. Sie bestimmen, wie ein partieller Bitstrom in den Konfigurierungsspeicher geschrieben wird und wie gegebenenfalls Daten zurück gelesen werden. Ein weiteres Verfahren, das nicht zwingend für die dynamische Rekonfigurierung erforderlich ist, mit dem aber der Speicherbedarf für partielle Bitströme deutlich gesenkt werden kann, ist die *Modulrelozierung*. Mit ihr ist es möglich, aus den beim Entwurf erzeugten positionierten Modulen positionsunabhängige Module zu generieren.

Im Folgenden werden nun zunächst die Entwurfsmöglichkeiten bei der Systempartitionierung diskutiert. Daran anschließend werden die Modulrelozierung und verschiedene Konfigurierungsverfahren vorgestellt. Schließlich wird der Einfluss dieser Entwurfsentscheidungen auf die Bitstromgröße und Konfigurierungsdauer erörtert.

3.2.1 Systempartitionierung

Bei der Zuteilung verfügbarer Ressourcen eines FPGAs für dynamische Komponenten bestehen aus technischer Sicht kaum Einschränkungen. Mit den erhältlichen FPGAs ist es möglich, jedes Konfigurationsbit ohne Beeinflussung der restlichen Konfiguration zu verändern. Einschränkungen gibt es allerdings bei der Verwendung der aktuellen Entwurfswerkzeuge von Xilinx, die derzeit ein Slice als kleinste zuweisbare Einheit vorgeben. Bei der Wahl einer Kachelung ist somit ein Slice die kleinste mögliche Kachelgröße, wobei die Lage der Kachel auf dem FPGA beliebig sein kann. Nach obiger Definition 4 gibt die Kachelung die möglichen Größen und Formen der zur Laufzeit geladenen Module vor. Gleichzeitig kann der segmentierte Konfigurationsspeicher des FPGAs nur frameweise beschrieben und ausgelesen werden, wodurch die Bitstromgröße und somit auch die Konfigurierungsdauer der Module bestimmt werden. Somit hat die Wahl der Kachelung einen Einfluss auf Bitstromgröße und Konfigurierungsdauer.

In Abbildung 3-2 sind drei verschiedene Fälle dargestellt, wie die Partitionierung der rekonfigurierbaren Ressourcen die Segmentierung des Konfigurationsspeichers überlagern kann. Sie werden im Folgenden als Typ A, Typ B und Typ C Partitionierungen bezeichnet. Dargestellt ist jeweils ein Ausschnitt eines FPGAs. Die Rechtecke stellen dabei FPGA-Ressourcen dar, die über ein gemeinsames Speichersegment (Frame) konfiguriert werden. Zusammenhängende graue Bereiche symbolisieren Ressourcen, die von einem zu ladenden Modul beansprucht werden. Bei einer Rekonfigurierung können die Speichersegmente immer nur komplett beschrieben, bzw. die entsprechenden Ressourcen immer nur komplett neu konfiguriert werden. Die hier gezeigte Partitionierung könnte die eines Virtex-4 FPGAs sein (sie-

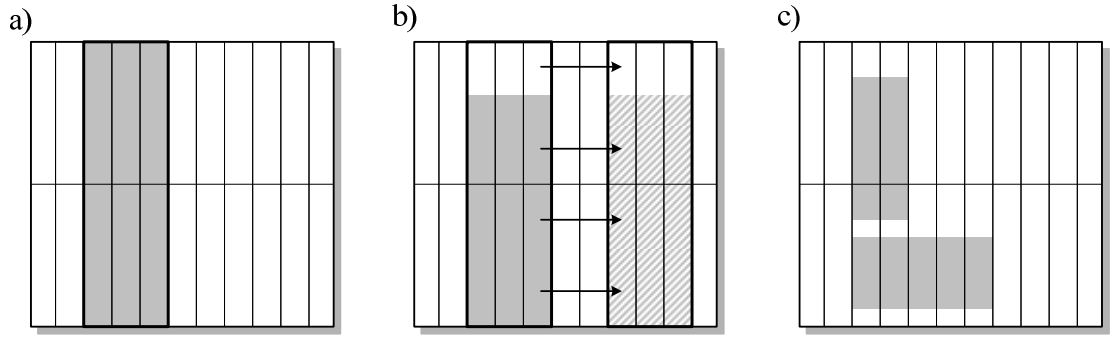

Abbildung 3-2: Verschiedene Situationen beim Konfigurieren eines Moduls, a) Typ A, b) Typ B, c) Typ C

he auch Abbildung 2-5 auf Seite 12), kann aber prinzipiell auf alle gängigen, dynamisch rekonfigurierbaren FPGAs übertragen werden.

Abbildung 3-2 a) zeigt eine Typ A Partitionierung, die den einfachsten Fall für eine partielle Rekonfigurierung darstellt. Das zu ladende Modul belegt hier genau sechs komplette Speichersegmente (Frames). Beim Beschreiben dieser Segmente werden nur genau die Konfigurationsdaten der benötigten Modulressourcen geändert. Der Bitstrom dieses Moduls, der hier aus sechs Frames besteht, enthält somit nur Informationen der gewünschten dynamischen Komponente. Anders verhält es sich bei einer Typ B Partitionierung, die in Abbildung 3-2 b) dargestellt ist. Das hier abgebildete Modul belegt einige Speichersegmente nur zum Teil. Der Rest dieser Segmente gehört zu einer statischen Systemkomponente. Um das Modul zu laden müssen die beteiligten Speichersegmente trotzdem komplett beschrieben werden. Der dazugehörige Bitstrom besteht hier ebenfalls aus sechs Frames, beschreibt aber nun auch Bereiche, die nicht zu der gewünschten dynamischen Komponente gehören. Dabei muss unbedingt darauf geachtet werden, dass die Konfiguration der nicht zum Modul gehörenden Ressourcen (also der statischen Komponente) nicht verändert wird. Die entsprechenden Bereiche der Speichersegmente müssen dazu mit der bereits vorhandenen Konfiguration überschrieben werden, so dass Funktion und Verschaltung der FPGA-Ressourcen identisch bleiben. Da das Konfigurieren bei Xilinx FPGAs Glitch-frei ist (vgl. 2.2.2), behalten die entsprechenden Schaltungsteile auch während der Konfigurierung ihre Funktion. Somit werden nur die gewünschten (grau dargestellten) Ressourcen umkonfiguriert, während die durch die Speichersegmentierung zusätzlich betroffenen Ressourcen quasi statisch bleiben. Vorsicht ist jedoch bei der Verwendung von Speicherelementen geboten. Wie in 2.1 erläutert, gibt es hierfür drei Realisierungsoptionen: BlockRAM, Flipflops und LUTs. Der Speicherinhalt von Flipflops ist nicht Bestandteil des Konfigurationsspeichers und bleibt bei einer Rekonfigurierung somit bestehen. Der Inhalt von BlockRAM oder von als Speicher verwendeten LUTs ist dagegen Teil des Konfigurationsspeichers. Werden solche Speicherelemente zur Laufzeit von einer Schaltung auf dem FPGA verändert, wird somit auch der Konfigurationsspeicher verändert. Andersherum wird bei einer Rekonfigurierung auch der Speicherinhalt dieser Speicherelemente verändert. Bei der Erstellung der Bitströme ist nur der Initialzustand der Speicher bekannt. Ein Überschreiben dieser Speicherelemente mit identischem Inhalt ist daher aufgrund der Dynamik des Systems (der Speicherinhalt ändert sich in der Regel ständig während der Laufzeit) nicht ohne weiteres möglich. Um hier Konflikte zu vermeiden, dürfen statische Komponenten in den mit dynamischen Komponenten geteilten Speichersegmenten keine Speicherelemente mit LUTs oder BlockRAM realisieren. Alternativ kann der aktuelle Speicherinhalt vor der Konfigurierung ausgelesen und zur Laufzeit in den Bitstrom der dynamischen Komponente eingefügt werden (*Read-Modify-Writeback*-Verfahren, s.u.). Auch hier muss jedoch bedacht werden, dass die Konfigurierung mehrere Millisekunden dauern kann. In dieser Zeit kann der Speicherinhalt durch die statischen Komponenten mehrmals geändert werden, so dass weiterhin Speicherinkonsistenzen auftreten können. Ein Beschreiben des Konfigurationsspeichers durch statische Komponenten muss während der Konfigurierung vermieden werden, was in der Praxis recht aufwändige Kontrollmechanismen erfordert.

Ein weiteres Problem bei einer Partitionierung nach Typ B ergibt sich, wenn ein Modul an verschiedene Positionen geladen werden soll, wie in Abbildung 3-2 b) angedeutet. Dies ist mit der weiter unten vorgestellten Modulrelozierung möglich, mit der ein Bitstrom an eine

andere als der ursprünglich vorgesehenen Fläche geladen werden kann. Da ein Bitstrom nur komplett reloziert werden kann, werden bei einer Typ B Konfigurierung die statischen Bereiche der Zielposition mit einer falschen Konfiguration überschrieben. Auch hier kann nur durch ein Read-Modify-Writeback-Verfahren Abhilfe geschaffen werden. Falls dies nicht möglich ist, müssen entweder separate Bitströme für alle möglichen Modulpositionen erzeugt werden, oder es muss eine *Bitstromkomposition* erfolgen. Hierbei werden die Konfigurationsdaten des Bitstroms zur Laufzeit aus den (positionsunabhängigen) Konfigurationsdaten des Moduls und den Konfigurationsdaten der statischen Komponenten an der Zielposition zusammengesetzt. Hierfür sind effiziente Realisierungen denkbar, nach Kenntnisstand des Autors jedoch bislang nicht umgesetzt. In jedem Fall bleiben die Einschränkungen bezüglich der Verwendung von Speicherelementen bestehen.

Die Problematik mit dem Überschreiben der Konfiguration unbeteiligter Ressourcen verschärft sich noch, wenn mehrere Module in zum Teil identische Speichersegmente geladen werden können. Diese Situation wird als Typ C Partitionierung bezeichnet und ist in Abbildung 3-2 c) dargestellt. Um eines der beiden Module laden zu können, müssen Konfigurationsdaten überschrieben werden, die möglicherweise zu einem anderen Modul gehören. Ob in diese Bereiche tatsächlich ein Modul geladen ist, und um welches Modul es sich dabei handelt, ist erst zur Laufzeit bekannt. Für die Konfigurierung gelten hier dieselben Bedingungen wie bei einer Typ B Partitionierung. In dem Bitstrom des zu ladenden Moduls müssen die Konfigurationsdaten des anderen, bereits geladenen Moduls enthalten sein. Prinzipiell ist denkbar, dass für jede mögliche Belegung des FPGAs entsprechende Bitströme vorgehalten werden. In der Praxis führt dies aber schnell zu einer nicht mehr handhabbaren Anzahl von Bitströmen, die mit der Anzahl der Module und Modulpositionen steigt. Bei einer Typ C Partitionierung, bei der es mindestens ein Speichersegment gibt, das Konfigurationsdaten mehrerer Module beinhaltet, muss daher auf eine Bitstromkomposition zur Laufzeit zurückgegriffen werden. Dadurch bleibt die Zahl benötigter Bitströme gering, wenn auch weiterhin die Einschränkungen bezüglich der Nutzung von Speicherelementen bestehen bleibt, insbesondere während einer dynamischen Konfigurierung.

3.2.2 Modulrelozierung

Um zur Laufzeit tatsächlich möglichst viele Positionen nutzen zu können, müssen die entsprechenden Module erzeugt werden, mit denen die gewünschten Flächen innerhalb der dynamischen Bereiche konfiguriert werden können. Bei der Definition des Moduls in Abschnitt 3.1.2 wurde vorausgesetzt, dass mit dem zum Modul gehörenden Bitstrom alle möglichen Modulpositionen konfiguriert werden können. Das setzt voraus, dass partielle Bitströme positionsunabhängig sind. Dies ist jedoch nicht der Fall, da jedes Konfigurationsbit über seine Position innerhalb des Frames sowie der Frame-Adresse genau einer rekonfigurierbaren Ressource zugeordnet ist (siehe 2.2.1). Im allgemeinen Fall muss daher für jedes Modul und jede mögliche Position ein separater Bitstrom vorgehalten werden, was schnell zu einem außerordentlich hohen Speicherbedarf für Bitströme führen kann (siehe 3.3.3). Diesem Effekt kann durch *Modulrelozierung* entgegengewirkt werden. Hierbei wird ein Bitstrom zur Laufzeit durch *Bitstrommanipulation* an eine gewünschte Position angepasst. Es muss also nur noch ein Bitstrom gespeichert werden, der an verschiedene Positionen geladen werden kann.

Die einfachste Möglichkeit der Modulrelozierung durch Bitstrommanipulation ist die Anpassung der Frame-Adressen eines Bitstroms, so dass das entsprechende Modul an eine andere als die ursprünglich festgelegte Position geladen wird. Für Xilinx Virtex(-E) und Virtex-II/Virtex2Pro FPGAs können Bitstrommanipulationen sowohl in Hardware (z.B. REPLICA [KLP05]) wie auch in Software (z.B. PARBIT [HL01] oder XPART [GLS99, BJK03]) durchgeführt werden. Besonders die Hardware-Lösungen können mit wenigen Ressourcen und ohne merkliche Einbußen bei der Konfigurierungsgeschwindigkeit realisiert werden. Ein Nachteil der Adressmanipulation ist, dass nur komplette Frames verschoben werden können. Die Möglichkeiten ein Modul zu relozieren sind somit durch die Konfigurationsgranularität des verwendeten FPGA vorgegeben. Für die spaltenweise rekonfigurierbaren Virtex-II Bausteine heißt dies z.B., dass Module nur entlang der Horizontalen verschoben werden können. Mit den mehrzeilig aufgebauten Konfigurationsspeichern der Virtex-4 und Virtex-5 FPGAs kann horizontal beliebig und vertikal um komplette Frame-Zeilen verschoben werden (vgl. Abbildung 2-5).

Um eine beliebige Verschiebung der Module auch vertikal zu erreichen, müssen die Konfigurationsinformationen *innerhalb* der Frames verändert werden. Das ist durch Verschieben der Frame-Inhalte möglich, da bei allen Xilinx FPGAs die Daten innerhalb der Frames ebenfalls regelmäßig angeordnet sind. Sedcole et al. [SBA05] haben dieses Verfahren beispielhaft für eine Virtex-II Implementierung realisiert. Sie verfügten dabei jedoch über Informationen über den inneren Aufbau der Bitströme, der von Xilinx nicht offen gelegt wurde. Generell kann festgehalten werden, dass eine Modulrelozierung, die sich nicht an die Granularität des Konfigurationsspeichers hält, prinzipiell machbar ist. Praktisch hat sie aber so lange keine Relevanz, bis Xilinx seine Bitstromarchitektur völlig offen legt.

Die Modulrelozierung durch Adressmanipulation ist besonders für die freie Platzierung ein wichtiges Qualitätsmerkmal, da ohne sie entweder die Anzahl der möglichen Positionen klein gehalten werden muss, oder aber der Speicherbedarf für partielle Bitströme sehr hoch ist. Außerdem können mithilfe der Modulrelozierung und mithilfe von Verfahren zum Auslesen des Konfigurationsspeichers Module zur Laufzeit umpositioniert werden [KKP06].

Bei einigen dynamischen Komponenten kann es wünschenswert oder sogar nötig sein, dass sie zur gleichen Zeit mehrfach in das System geladen werden. Dieser Umstand wurde oben als Mehrfachinstanziierung eingeführt. Für die Mehrfachinstanziierung eines Moduls ist die Modulrelozierung zwingend erforderlich. Ohne sie müssen mehrere Instanzen einer Komponente immer verschiedenen Modulen abstammen.

Falls eine Modulrelozierung zur Laufzeit nicht verwendet werden kann oder soll, kann sie auch schon beim Entwurf durchgeführt werden. Hier müsste normalerweise jeder Bitstrom eines Moduls durch viele Entwurfsschritte separat generiert werden. Dies bedeutet nicht nur einen enormen zeitlichen Aufwand, sondern führt in der Regel zu nicht einheitlichen Verschaltungen der Bitströme desselben Moduls. Stattdessen kann ein Bitstrom erzeugt werden, der beliebig oft kopiert und mit Bitstrommanipulation an die gewünschte Position angepasst wird. Nur so kann sichergestellt werden, dass alle Bitströme zu einer identischen Verschaltung der betroffenen FPGA-Flächen führen.

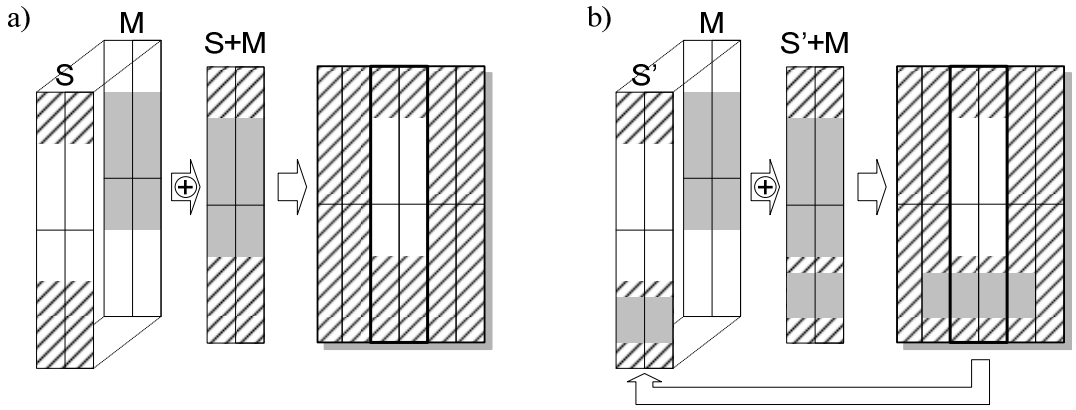

Abbildung 3-3: Bitstromkomposition, a) direkt, b) mit vorherigem Auslesen von Konfigurationsdaten

3.2.3 Konfigurierungsverfahren

Wie bereits im vorangegangenen Abschnitt angedeutet, können für die dynamische Konfigurierung verschiedene Verfahren angewendet werden. Der einfache, klassische Weg ist die *direkte partielle Konfigurierung* (*direct partial configuration* [Xil06]). Hierzu wird zum Laden eines Moduls der entsprechende Bitstrom direkt in den Konfigurationsspeicher übertragen und die existierende Konfiguration überschrieben. Die direkte Konfigurierung führt immer dann zu Problemen, wenn die durch einen Bitstrom konfigurierte FPGA-Fläche nicht ausschließlich einem Modul zusteht, sondern Bereiche anderer Module oder statischer Komponenten enthält (Typ B und Typ C Segmentierung). In diesem Fall muss der Bitstrom derart geschaffen sein, dass die zusätzlich betroffenen FPGA-Ressourcen nicht durch die Konfigurierung beeinträchtigt werden. Schließt man die Möglichkeit, separate Bitströme für alle möglichen Positionen und Modulkombinationen zu erzeugen, aus, muss eine Bitstromkomposition zur Laufzeit durchgeführt werden, für die zwei Fälle unterschieden werden: Bei der *direkten Komposition* werden die benötigten Konfigurationsdaten aus vorhandenen, statisch erzeugten Bitströmen zusammengesetzt. In Abbildung 3-3 a) ist eine direkte Komposition schematisch dargestellt. Rechts im Bild dargestellt ist ein Ausschnitt einer FPGA-Fläche, die über zwölf Speichersegmente konfiguriert werden kann. Schraffiert dargestellt ist der statische Bereich des FPGAs, die weiße Fläche ist ein leerer Modulplatz. Die vier Speichersegmente, die zum Laden eines Moduls beschrieben werden müssen, beinhalten auch Teile der statischen Komponenten. Der partielle Bitstrom des Moduls M wird nun bei der Bitstromkomposition durch eine geeignete Funktion mit einem Ausschnitt S des Initialbitstroms, der die Konfiguration der statischen Hardware enthält, überlagert. Der so entstandene Bitstrom kann nun in den Konfigurationsspeicher des FPGAs übertragen werden, ohne die Funktion der statischen Komponenten zu beeinflussen. Wie bereits mehrfach erwähnt, darf der betroffene Teil der statischen Schaltungen keine durch LUTs oder BlockRAM realisierten Speicher enthalten, die zur Laufzeit geändert werden, da sonst durch die Konfigurierung der Initialzustand wiederhergestellt wird. Nur-Lese-Speicher (*ROM*[18]) kann dagegen verwendet werden.

Im Gegensatz zur direkten Komposition werden bei der *indirekten Komposition* zunächst die Konfigurationsdaten der bereits aktiven Komponenten aus dem FPGA zurück gelesen.

[18] Read-Only Memory

Falls es sich dabei nur um statische Hardware handeln sollte, entsprechen die zurück gelesenen Daten dem zuvor beschriebenen Ausschnitt aus dem Initialbitstrom. Allerdings werden hier automatisch die aktuellen Inhalte der Speicherelemente gelesen, so dass die Komposition eines Modulbitstroms M mit den gelesenen Konfigurationsdaten S' zu einem Bitstrom mit aktuellem Speicherinhalt führt. Vorausgesetzt, dass sich die Speicherinhalte zwischen dem Auslesen des Speichers und dem erneuten Beschreiben nicht ändert, können LUTs und BlockRAMs verwendet werden. In Abbildung 3-3 b) ist beispielhaft eine indirekte Komposition dargestellt. Sie wurde erstmals von Sedcole et al. vorgestellt [SBA05] und wird aufgrund der Vorgehensweise auch als *Read-Modify-Writeback*-Verfahren (RMW) bezeichnet. In dem gegebenen Beispiel ist neben den statischen Komponenten ein weiteres, bereits geladenes Modul grau dargestellt. Für das Read-Modify-Writeback-Verfahren spielt dies keine Rolle, da beim Auslesen des Konfigurationsspeichers nicht zwischen statischen und dynamischen Komponenten unterschieden werden kann. Als Überlagerungsfunktion schlagen Sedcole et al. eine XOR-Verknüpfung der Bitströme vor. Voraussetzung hierfür ist jedoch, dass die verknüpften Konfigurationsdaten nie an identischen Stellen des Bitstroms Daten enthalten. Im Beispiel in Abbildung 3-3 b) bedeutet dies, dass die weiß gekennzeichneten Stellen der Bitströme S' und M nur Nullen enthalten dürfen, um durch eine XOR-Verknüpfung zum Bitstrom (S'+M) zusammengefügt werden zu können. Das wiederum bedeutet, dass bestehende Module nicht einfach durch ein neues Modul überschrieben werden können, sondern erst gelöscht werden müssen, wodurch eine weiter Verzögerung beim Laden eines Moduls auftritt.

Kennzeichnend für die von Blodget vorgeschlagene Methode ist, dass die benötigten Bitströme nicht mit dem Standard-Entwurfsablauf für partielle Konfigurierung erstellt werden können. Es sind stattdessen angepasste Entwurfswerkzeuge vonnöten, die sicherstellen, dass die partiellen Bitströme der Module keine Konfigurationsdaten statischer oder anderer dynamischer Komponenten enthalten. Dies ist mit den herkömmlichen Entwurfswerkzeugen nicht möglich, die nur die direkte Konfigurierung unterstützen.

3.2.4 Bitstromgröße und Konfigurierungsdauer

Die vorgestellten Partitionierungen unterscheiden sich nicht nur darin, dass sie unterschiedliche Konfigurierungsverfahren voraussetzen. Auch die Größe der partiellen Bitströme und die Konfigurierungsgeschwindigkeit variieren je nach gewählter Partitionierung deutlich.

Die Größe der Bitströme hängt fast ausschließlich davon ab, wie viele Frames des Konfigurationsspeichers geändert werden müssen (vgl. 2.2.3). Den optimalen Fall stellt die Typ A Partitionierung dar, bei der das Verhältnis zwischen zu verändernden und tatsächlich überschriebenen Konfigurationsdaten gleich Eins ist (siehe Abbildung 3-2). Bei einer Typ B oder Typ C Partitionierung ist dieses Verhältnis aufgrund der ungünstigeren Überlagerung der Segmentierung der rekonfigurierbaren Ressourcen und der Segmentierung des Konfigurationsspeichers kleiner Eins. Es müssen also mehr Konfigurationsdaten geschrieben werden als zum Laden eines Moduls eigentlich geändert werden müssen. Dieses Verhältnis kann fast beliebig schlecht ausfallen. Wird beispielsweise für einen Virtex-II FPGA, der eine spaltenbasierte Segmentierung des Konfigurationsspeichers aufweist, eine zeilenbasierte Partitionierung der Ressourcen gewählt, gleicht jede partielle Rekonfigurierung einer vollständigen Konfigurierung, da für die Änderung einer FPGA-Zeile Informationen in allen Spalten des Konfigurationsspeichers geändert werden müssen.

Tabelle 3-1: Mögliche Konfigurierungsmodi

	keine Einschränkungen	Modulrelozierung	BlockRAM/ LUT-Speicher
Typ A	direkte Konfigurierung	direkte Konfigurierung	–
Typ B	direkte Konfigurierung	statische Komposition	RMW
Typ C	statische Komposition	statische Komposition	RMW

Mit der Konfigurierungsgeschwindigkeit verhält es sich ähnlich wie mit der Bitstromgröße. Je mehr Frames geändert werden müssen, desto höher ist der Konfigurierungsaufwand. Solange beim Konfigurieren ausschließlich Schreibzugriffe stattfinden (direkte Konfigurierung), ist die Konfigurierungsdauer in guter Näherung proportional zur Bitstromgröße. Eine ungünstige Segmentierung der rekonfigurierbaren Ressourcen wirkt sich somit ebenfalls ungünstig auf die Konfigurierungsdauer aus. Sobald wie beim RMW-Verfahren der Speicher auch ausgelesen wird, erhöht sich Konfigurierungsaufwand zusätzlich. Im besten Fall verdoppelt sich die Dauer einer Konfigurierung. Dieser tritt dann ein, wenn die benötigten Konfigurationsdaten zunächst komplett ausgelesen und nach der Manipulation ebenfalls am Stück wieder in den Konfigurationsspeicher übertragen werden. Für die Manipulation selbst kann angenommen werden, dass sie keine Verzögerung mit sich bringt. Die von Blodget verwendete XOR-Verknüpfung kann beispielsweise ohne zusätzliche Latenz in den Datenpfad eingefügt werden. Um wie oben beschrieben die Daten am Stück auszulesen und zurückzuschreiben, ist ein lokaler Speicher erforderlich, der den gesamten partiellen Bitstrom aufnehmen kann. Ist dies nicht möglich, müssen die Konfigurationsdaten abschnittsweise ausgelesen, manipuliert und wieder zurückgeschrieben werden. Da zum Abschluss jedes Schreib- oder Lesevorgangs bei Xilinx FPGAs ein Pad-Frame geschrieben/gelesen werden muss, erhöht sich die Konfigurierungslatenz mit der Anzahl der Schreib- und Lesezugriffe entsprechend. Im schlimmsten Fall wird das RMW frameweise durchgeführt, so dass der Zeitaufwand mindestens vier Mal höher als bei einer direkten Konfigurierung ist.

In der Praxis ist die Konfigurierungsgeschwindigkeit bzw. Konfigurierungsdauer neben dem verwendeten Konfigurierungsverfahren auch von der Leistungsfähigkeit der Konfigurierungshardware abhängig. Ein quantitativer Vergleich von Realisierungen verschiedener Konfigurierungsverfahren ist in Abschnitt 5.3.1.1 gegeben.

3.2.5 Zusammenfassung

In Tabelle 3-1 ist zusammengefasst, welches der hier vorgestellten Konfigurierungsverfahren (direkte Konfigurierung, indirekte Konfigurierung mit statischer Bitstromkomposition, indirekte Konfigurierung mit RMW) für die drei oben vorgestellten Segmentierungsfälle geeignet ist. Es wird unterschieden, ob Modulrelozierung verwendet werden soll, ob BlockRAM- oder LUT-Speicher fremder Komponenten in den Konfigurationsspeichersegmenten enthalten ist, oder ob keine dieser Beschränkungen vorliegt.

Mit den Möglichkeiten der statischen Komposition und RMW können alle vorgestellten Partitionierungstypen auf heutigen FPGAs realisiert werden. Bei der Partitionierung der re-

konfigurierbaren Ressourcen muss daher prinzipiell keine Rücksicht auf die Segmentierung des Konfigurationsspeichers genommen werden. Allerdings bedeuten statische Komposition und RMW immer einen zusätzlichen technischen Aufwand für die Manipulation der Konfigurationsdaten. Bei Typ B und Typ C Partitionierungen kommt es neben der Notwendigkeit der Bitstrommanipulation außerdem zu größeren partiellen Bitströmen. Hierdurch steigt zum einen der Speicherbedarf für die partiellen Bitströme, zum anderen steigt die Konfigurierungsdauer. Letztere wird bei Verwendung eines Read-Modify-Writeback-Verfahrens zusätzlich erhöht. Wenn möglich sollte daher eine Typ A Partitionierung verwendet werden. Die Kacheln innerhalb eines rekonfigurierbaren Bereichs sollten also immer nur aus vollständigen Frames bestehen.

3.3 Modulplatzierung

Dynamische Komponenten werden zur Laufzeit durch das Laden entsprechender Module auf das FPGA in das System eingebunden. An welche Stellen die Module dabei geladen werden können, kann erst zur Laufzeit entschieden werden. Welche Positionen aber in Frage kommen, wird beim Entwurf festgelegt. In einem ersten Schritt müssen die verfügbaren Ressourcen in dynamische und statische Bereiche unterteilt werden. Dieser Prozess wird als *Floorplanning* bezeichnet. Anschließend müssen innerhalb der dynamischen Bereiche Kacheln definiert und dadurch eine *Segmentierung* der Bereiche vorgenommen werden. Mit den geometrischen Vorgaben aus Floorplanning und Segmentierung können die Module für die vorhandenen Komponenten erzeugt werden. Der Entwickler kann dabei noch weitere Einschränkungen bezüglich Größe und Form der Module vorgeben. Üblicherweise werden z.B. rechteckige Formen vorgegeben. Die Summe aller Vorgaben und Einschränkungen bezüglich der geometrischen Eigenschaften (Größe, Form) der Module wird in dieser Arbeit als *Platzierungsverfahren* bezeichnet.

Es ist somit das Platzierungsverfahren, das die Menge möglicher Module zu einer Komponente bestimmt. Welche Module aus dieser Menge letztlich wirklich erstellt werden, bleibt dem Entwickler vorbehalten. Hier gilt es, den Speicherbedarf der erzeugten Module ebenso im Blick zu behalten wie die Vielfalt der Komponentenpositionen, die in der Regel mit der Anzahl der Module steigt. Zur Laufzeit muss für eine zu ladende Komponente eine der in Frage kommenden Positionen in Abhängigkeit der Belegung der rekonfigurierbaren Bereiche gewählt werden. Wie zur Laufzeit freie Komponentenpositionen gefunden werden und nach welchen Maßstäben die Auswahl getroffen wird, wird durch eine *Platzierungsstrategie* bestimmt.

3.3.1 Platzierungsverfahren

Bei der Festlegung eines Platzierungsverfahrens muss für jeden rekonfigurierbaren Bereich eine Kachelung definiert werden, die vorgibt, wie Module die verfügbaren Ressourcen nutzen dürfen. Hierfür gibt es prinzipiell keine Einschränkungen, allerdings müssen die geometrischen Vorgaben für die Module auch mit den verfügbaren Entwurfswerkzeugen umgesetzt werden können. Je nach Art der Kachelung kann man verschiedene Platzierungsverfahren zueinander abgrenzen. Im Folgenden wird zwischen *festen Modulplätzen* und *freier Platzierung* unterschieden.

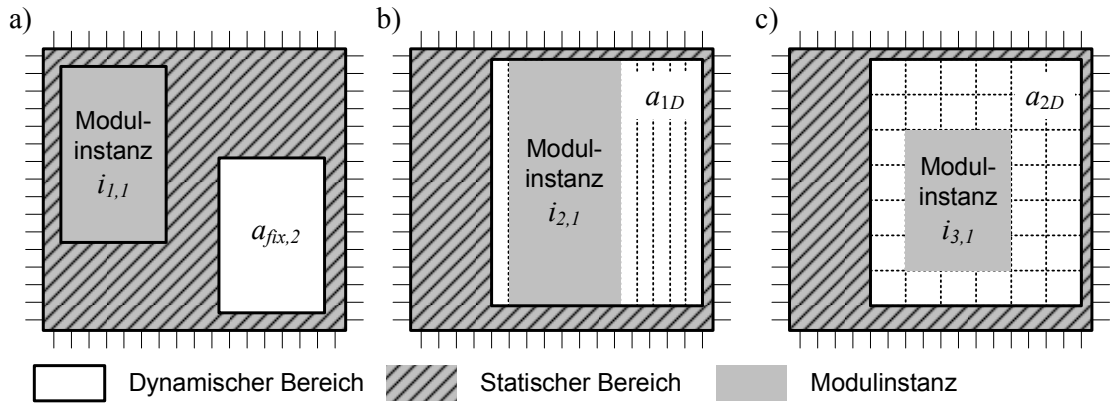

Abbildung 3-4: Mögliche Platzierungsverfahren: a) feste Modulplätze, b) freie 1D-Platzierung, c) freie 2D-Platzierung

Der triviale Fall für die Kachelung ist der, bei dem jeder rekonfigurierbare Bereich aus genau einer Kachel besteht. Die rekonfigurierbaren Bereiche werden dann *feste Modulplätze* genannt. In Abbildung 3-4 a) ist ein System mit zwei festen Modulplätzen dargestellt. Aufgrund der einfachen Kachelung ist die Größe der Module dabei durch die Größe der rekonfigurierbaren Bereiche vorgegeben. Bei unterschiedlich gearteten rekonfigurierbaren Bereichen bedeutet das, dass es für jedes Modul nur genau eine gültige Modulposition gibt. Nur falls mehrere, zu einander kongruente Bereiche existieren, erhöht sich entsprechend die Anzahl möglicher Modulpositionen. Bei festen Modulplätzen gibt es keine Möglichkeit, die Größe eines Moduls an die Menge der benötigten Ressourcen anzupassen. Da die Nutzung dynamischer Rekonfigurierung nur dann Sinn macht, wenn zur Laufzeit mindestens zwei verschiedene Module einen rekonfigurierbaren Bereich nutzen, muss die Größe der Bereiche an die Komponente mit dem größten Ressourcenbedarf angepasst werden. Für andere, kleinere Komponenten ergibt sich dann eine niedrige Ressourceneffizienz (siehe 6.1). Ein Vorteil fester Modulplätze ist jedoch, dass entsprechende Platzierungsstrategien sehr einfach und mit wenig Rechenleistung zur Laufzeit realisiert werden können. Entscheidend dafür, ob eine Komponente geladen werden kann, ist ausschließlich das Vorhandensein eines unbelegten dynamischen Bereichs. Darüber hinaus ist die Anzahl der möglichen Modulpositionen durch die Anzahl der dynamischen Bereiche begrenzt, so dass nur zwischen wenigen Modulpositionen abgewägt und entschieden werden muss.

Feste Modulplätze sind momentan am einfachsten zu realisieren, sowohl was den Entwurf mit bestehenden Werkzeugen betrifft, als auch was die Komplexität der Laufzeitumgebung für dynamische Rekonfigurierung angeht. Vor allem kann für Module in festen Modulplätzen relativ einfach eine Kommunikationsinfrastruktur mit Hilfe von Busmakros bereitgestellt werden. Xilinx stellt derzeit allein für die dynamische Rekonfigurierung mit festen Modulplätzen Entwicklungswerkzeuge, Busmakros und Entwurfsanleitungen bereit. Kennzeichnend für diesen Ansatz ist, dass der Ressourcenbedarf aller Module zur Entwurfszeit bekannt sein muss, damit die Größe der Modulplätze entsprechend angepasst werden kann.

Existiert in den dynamischen Bereichen eine nicht-triviale Kachelung, also zwei oder mehr Kacheln, dann gibt es im Allgemeinen mehrere Möglichkeiten der Abbildung von Komponenten auf die verfügbaren Kacheln. Die Existenz mehrerer Modulpositionen führt zu einer

prinzipiell freien Auswahl bei der Platzierung von Modulen zur Laufzeit, so dass hier von *freier Platzierung* gesprochen wird. Je nachdem, ob die Partitionierung (Kachelung) nur über eine Dimension (z.B. nur die Horizontale) oder zwei Dimensionen vorgenommen wurde, spricht man von 1D- bzw. 2D-Platzierung. Abbildung 3-4 b) zeigt einen rekonfigurierbaren Bereich a_{1D} mit einer Kachelung für freie horizontale 1D-Platzierung. Alle Module müssen hierbei die volle Höhe des zur Verfügung stehenden rekonfigurierbaren Bereichs einnehmen. Die Modulbreite wird dann den Ressourcenanforderungen des Moduls angepasst. Mögliche Modulpositionen befinden sich entlang einer horizontalen Linie, die im Folgenden auch als Platzierungsachse bezeichnet wird. Bei der in Abbildung 3-4 c) gezeigten 2D-Platzierung können sowohl Höhe wie Breite eines Moduls variiert werden, so dass Module mit fast beliebigen Formen für eine dynamische Komponente generiert werden können.

Dadurch, dass bei freier Platzierung die Größe eines Moduls an den tatsächlichen Ressourcenbedarf angepasst werden kann, ergibt sich eine andere, womöglich effizientere Nutzung der rekonfigurierbaren Ressourcen. Ein Modul kann nun immer dann geladen werden, wenn genügend zusammenhängende Ressourcen verfügbar sind, unabhängig davon, ob schon andere Module geladen sind. Betrachtet man die Ressourcenauslastung auf einer so abstrakten Ebene, so ist leicht ersichtlich, dass Module bei freier Platzierung in fast allen Fällen weniger Ressourcen beanspruchen als bei unflexiblen, festen Modulplätzen. Tatsächlich steigen mit der Anzahl möglicher Modulpositionen aber auch die Anforderungen an die Kommunikationsinfrastruktur, über die mit den Modulen kommuniziert wird. Darüber hinaus können die frei bleibenden Ressourcen nicht immer durch weitere Module genutzt werden. Eine Bewertung verschiedener Platzierungsverfahren unter Berücksichtigung dieser Einflüsse wird in den Kapiteln 4 und 6 vorgenommen.

3.3.2 Platzierungsstrategien

Beim Laden einer Komponente wird zur Laufzeit ein Verfahren benötigt, das aus den möglichen Komponentenpositionen eine geeignete auswählt. Gegebenenfalls muss hierzu zuvor eine Metrik definiert werden, mit der verschiedene Positionen bewertet werden können. Eine Metrik und ein dazu gehörendes Auswahlverfahren geeigneter Positionen wird *Platzierungsstrategie* genannt. Bei festen Modulplätzen beschränkt sich die Platzierungsstrategie auf das Auffinden eines freien dynamischen Bereichs, während bei freier Platzierung freie Positionen innerhalb eines dynamischen Bereichs gefunden werden müssen. Die meisten hierfür existierenden Platzierungsstrategien versuchen Module so zu platzieren, dass möglichst viele Module in die verfügbaren Flächen geladen werden können. In Abbildung 3-4 c) ist offensichtlich, dass die Modulinstanz $i_{3,1}$ nicht optimal platziert wurde. Hätte man es anstelle der gewählten Position in eine der vier Ecken geladen, wären die Platzierungsmöglichkeiten für weitere Module wesentlich besser. Man spricht dabei von einer *Fragmentierung* eines rekonfigurierbaren Bereichs, wenn freie Flächen nicht zusammenhängend genutzt werden können. Gängige Platzierungsstrategien sind beispielsweise *FirstFit* und *BestFit* [KPR05], die jeweils nur nach geometrischen Gesichtspunkten Positionen auswählen. Das Problem der Fragmentierung kann durch eine geeignete Platzierungsstrategie zwar vermindert, im Allgemeinen aber nicht vollständig behoben werden. Aus diesem Grund werden Platzierungsstrategien nicht nur für das Laden eines Moduls verwendet, sondern auch zum Defragmentieren, bei dem die geladenen Module neu auf den rekonfigurierbaren Flächen angeordnet werden [DE97, CCK02,

KKP05c]. Weitere mögliche Ziele für Platzierungsstrategien können die Optimierung der Kommunikationslast oder der Leistungsaufnahme (durch optimale Nutzung der Taktbäume) sein. Hierfür finden sich aber bislang keine Ansätze in der Literatur, vermutlich weil hierfür konkrete Implementierungen komplexer, dynamisch rekonfigurierbarer Systeme vonnöten sind, die kaum existieren.

Die Platzierungsstrategien haben neben den Platzierungsverfahren einen merklichen Einfluss auf die Ressourceneffizienz des Gesamtsystems. Für eine vergleichende Bewertung muss jedoch ein konkretes Anwendungsbeispiel oder ein Benchmark bekannt sein, so dass eine Menge dynamischer Komponenten und die dazugehörigen Module bekannt sind. Zusätzlich muss die Reihenfolge, mit der die Komponenten ins System geladen werden, und deren Verweildauer im System vorgegeben werden. Für verschiedene 1D- und 2D-Platzierungsverfahren und mehrere verschiedene Platzierungsstrategien wurden in dieser Arbeit Vergleiche mit einem Simulator durchgeführt [KKK04]. Ein wesentliches Ergebnis ist, dass mit 2D-Ansätzen trotz der wesentlich höheren Freiheit bezüglich der Modulplatzierung keine durchweg höhere Ressourcenauslastung für identische Benchmarks erzielt werden kann. Es kann im Gegenteil gezeigt werden, dass aufgrund der höheren Fragmentierung der Ressourcen bei 2D-Platzierungsverfahren die verfügbaren Ressourcen oft weniger effizient genutzt werden können als bei 1D-Verfahren mit gleicher Ressourcenmenge.

Eine weitere Erkenntnis aus den Simulationen ist jedoch, dass die Modelle der Simulatoren deutlich von den existierenden oder zumindest realisierbaren rekonfigurierbaren Systemen abweichen. In dieser Arbeit werden daher die statischen Eigenschaften verschiedener Platzierungsverfahren, insbesondere der Einfluss der benötigten Kommunikationsinfrastruktur, hinsichtlich des Ressourcenbedarfs untersucht. Dynamische Einflüsse, also Ablaufpläne für die Nutzung dynamischer Komponenten sowie Platzierungsstrategien, sind nicht Bestandteil dieser Arbeit. Sie wurden in einer parallel angefertigten Doktorarbeit von Markus Köster untersucht [Koe07], in der auch die angesprochenen Simulationsmodelle weiterentwickelt wurden.

3.3.3 Speicherbedarf für partielle Bitströme

Der Speicherbedarf für partielle Bitströme wird naturgemäß durch die Anzahl der benötigten Bitströme sowie ihrer jeweiligen Größe bestimmt. Die Größe eines Bitstroms ist dabei von der Menge der belegten Ressourcen des entsprechenden Moduls sowie der Deckung der Ressourcensegmentierung mit der Segmentierung des Konfigurationsspeichers abhängig (siehe 3.2.4). Die Anzahl der insgesamt benötigten Bitströme pro Komponente hängt wiederum von der Kachelung der rekonfigurierbaren Bereiche ab. Sie bestimmt, wie viele unterschiedliche Module zu einer Komponente erzeugt werden können. Je mehr Module existieren und je mehr gültige Modulpositionen es jeweils gibt, desto höher sind die Platzierungsmöglichkeiten zur Laufzeit. Gleichzeitig steigt jedoch auch der Speicherbedarf für partielle Bitströme.

3.3.3.1 Gültige Modulpositionen in homogenen FPGAs

Für eine Betrachtung der möglichen Modulpositionen in homogenen FPGA sei im Folgenden eine Menge $\mathfrak{R} = \{R_{D,1}, R_{D,2}, \ldots, R_{D,m}\}$ rekonfigurierbarer Bereiche gegeben, deren Breite und Höhe jeweils W_i bzw. H_i ($i = 1, \ldots, m$) betragen. Für ein beliebiges, rechteckiges Modul M seien mit w und h ebenfalls Breite und Höhe bekannt. Die Einheit der Breite und Höhe wird aus der Kachelung abgeleitet und sei hier beispielhaft als ein CLB angenommen, d.h. es

gibt eine gleichmäßige Kachelung mit einer Rasterbreite und -höhe von einem CLB. In einem homogenen FPGA ist dann die Anzahl der möglichen Positionen $\left|P_M^{fix}\right|$ bei Platzierung mit festen Modulplätzen gegeben durch die Anzahl der dynamischen Bereiche, es gilt:

$$\left|P_M^{fix}\right| = \left|\Re\right| \tag{10}$$

Hierbei wird vorausgesetzt, dass jedes Modul in alle Modulplätze geladen werden kann. Bei freier Platzierung gibt es für jedes Modul wesentlich mehr mögliche Positionen. Hier spielen die Ausmaße der dynamischen Bereiche und der Module eine Rolle. Es gilt

$$\left|P_M^{2D}\right| = \sum_{i=1}^{m}\left(\left(W_i - w + 1\right)\cdot\left(H_i - h + 1\right)\right) \tag{11}$$

bei zweidimensionaler Platzierung, und

$$\left|P_M^{1D}\right| = \sum_{i=1}^{m}\left(W_i - w + 1\right) \tag{12}$$

für den eindimensionalen Fall mit horizontaler Platzierung ($H_i = h$). Beim Vergleich der freien Platzierung mit festen Modulplätzen muss beachtet werden, dass die deklarierten festen Modulplätze in der Praxis deutlich kleiner gewählt werden als die dynamischen Bereiche bei freier Platzierung. Das liegt daran, dass feste Modulplätze nur für ein Modul ausgelegt werden, während in die dynamischen Bereiche bei freier Platzierung mehrere Module gleichzeitig geladen werden können. Ein Vergleich der freien Platzierungen ist allerdings zulässig. Als Beispiel sei ein homogener dynamischer Bereich R_D angenommen, der aus 72 CLB-Spalten und 80 CLB-Zeilen besteht ($W = 72$, $H = 80$). Dies entspricht der Größe eines XC2V4000 bei Nichtberücksichtigung der eingebetteten BlockRAMs und Multiplizierer. Für eine Komponente K, die mit 960 CLBs knapp 17 % der verfügbaren Ressourcen benötigt, ist M_1 mit $w_1 = 12$ und $h_1 = 80$ ein gültiges Modul bei horizontaler eindimensionaler Platzierung. Nach obiger Gleichung gibt es für M_1 insgesamt $\left|P_1^{1D}\right| = 60$ mögliche Modulpositionen. Ein Modul M_2 für dieselbe Komponente mit $w_2 = 40$ und $h_2 = 24$ kann bei 2D-Platzierung an $\left|P_2^{2D}\right| = 1881$ Positionen geladen werden. Dabei wird immer vorausgesetzt, dass eine völlige Homogenität der FPGA-Ressourcen vorliegt und ein partieller Bitstrom mithilfe der Modulrelozierung an alle Modulpositionen geladen werden kann. Partielle Bitströme für die obigen Beispielmodule wären etwa 213,24 KB (M_1) und 708,93 KB (M_2) groß. Ohne Modulrelozierung muss für jede Modulposition ein separater Bitstrom erzeugt werden. Um alle möglichen Modulpositionen zur Laufzeit nutzen zu können, würden für den 1D Fall 60 Bitströme mit einem Speicherbedarf von 12,79 MB, und für den 2D Fall 1881 Bitströme mit einem Speicherbedarf von über 1,3 GB benötigt. Berücksichtigt man zudem, dass bei zweidimensionaler Platzierung mehrere Module mit unterschiedlichen Ausmaßen zu einer Komponente erstellt werden können, werden nicht erfüllbare Speicheranforderungen durch nur eine Komponente erreicht. Um dem entgegenzuwirken, kann die Anzahl unterstützter Positionen reduziert werden, z.B. durch die Verwendung einer gröberen Kachelung. Wird bei obigem Beispiel die Granularität von einem CLB auf vier CLBs erhöht, reduziert sich die Anzahl möglicher Positionen drastisch. Es gilt nun $W_A = 30$ und $H_A = 20$. Für dieselbe Komponente K können nun die Module M_1 ($w_1 = 3$, $h_1 = 20$) und M_2 ($w_2 = 10$, $h_2 = 6$) erzeugt werden, für die $P_1^{1D} = 28$ bzw. $P_2^{2D} = 315$ mögliche Positionen existieren. Der Speicherbedarf verringert sich dadurch auf 5,97 MB bzw. 223,3 MB.

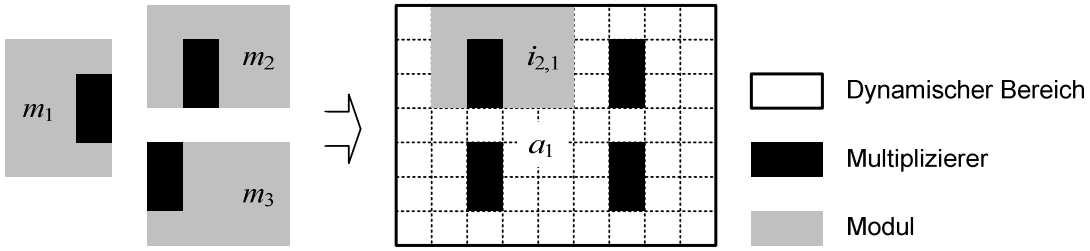

Abbildung 3-5: Modulplatzierung in heterogenen FPGAs

3.3.3.2 Gültige Modulpositionen in heterogenen FPGAs

Bei fast allen verfügbaren dynamisch rekonfigurierbaren FPGAs wird die homogene FPGA-Architektur mit den regelmäßig angeordneten und regelmäßig verdrahteten konfigurierbaren Logikzellen an mehreren Stellen von eingebetteten Blöcken unterbrochen. Es handelt sich also um einen heterogenen Aufbau, der sich auch im Konfigurationsspeicher widerspiegelt. Dynamische Komponenten können diese eingebetteten Elemente nutzen, um Teile einer Berechnung zu beschleunigen (z.B. mithilfe eingebetteter Multiplizierer) oder um Speicherelemente mit einzubeziehen. Für den Fall, dass die heterogenen Elemente zwingend in einer Komponente benötigt werden, schränkt das die möglichen Modulpositionen auf die Stellen des FPGAs ein, die entsprechende Elemente bereitstellen. Abbildung 3-5 zeigt einen dynamischen Bereich a_1 mit eingebetteten Multiplizierern und drei Module einer Komponente, die zehn CLBs und einen Multiplizierer verwendet. Da die interne Verschaltung der Module zur Laufzeit nicht geändert werden kann, gibt es für die Module m_1 und m_2 nur je vier mögliche Positionen, zu denen sie zur Laufzeit geladen werden können. Für das Modul m_3 gibt es sogar nur zwei gültige Positionen.

Um zur Laufzeit die größtmögliche Entscheidungsfreiheit zu haben, muss auch mit Modulrelozierung und gleichbleibender Form (z.B. 3 Zeilen mal 4 Spalten) für jeden möglichen Positionstyp ein Modul generiert werden. Wenn dies einen zu großen Speicherbedarf für partielle Bitströme hervorruft, kann nur die Anzahl der möglichen Positionen eingeschränkt werden. In jedem Fall können Module in heterogenen FPGAs mithilfe der Modulrelozierung nur an Positionen verschoben werden, die exakt ihre geometrischen Bedingungen erfüllen. Bei dem hier angenommenen, imaginären FPGA und den oben gegebenen Ressourcenanforderungen der Komponente (zehn CLBs, ein Multiplizierer) können theoretisch acht verschiedene Module mit demselben Formfaktor erzeugt werden. Sie können an insgesamt 24 verschiedene Positionen geladen werden. Da selbst bei diesem stark inhomogenen Aufbau des FPGAs noch viele kongruente Flächen existieren, müssen dank der Modulrelozierung nur acht statt 24 Bitströme gespeichert werden.

3.4 Zusammenfassung

In diesem Kapitel wurden die Grundlagen dynamisch rekonfigurierbarer FPGA-Systeme vorgestellt und modelliert. Nach dem Modell wird in dieser Arbeit zwischen abstrakten Aufgaben, entsprechenden Systemkomponenten, Modulen und Modulinstanzen unterschieden. Für die rekonfigurierbaren Ressourcen wurden die Begriffe „dynamischer Bereich“ und „Kachel“ definiert. Demnach sind die für dynamische Systemkomponenten vorgesehenen dyna-

mischen Bereiche eines FPGAs immer in Kacheln unterteilt. Wird eine Systemkomponente beim Entwurf auf einen dynamischen Bereich abgebildet, belegt das dadurch entstehende Modul dieser Komponente immer eine ganze Anzahl zusammenhängender Kacheln. Im trivialen Fall besteht ein dynamischer Bereich aus genau einer Kachel, wodurch ein fester Modulplatz entsteht. Je größer die Zahl der Kacheln eines dynamischen Bereichs ist, desto vielfältiger sind die Möglichkeiten, eine Komponente auf diesen Bereich abzubilden und es entsteht ein System mit freier Platzierung.

Für die Realisierung dynamisch rekonfigurierbarer Systeme wurden die derzeit bekannten Verfahren vorgestellt. Neben den unterschiedlichen Konfigurierungsverfahren (direkte Konfigurierung, Read-Modify-Writeback) wurde insbesondere auf die Möglichkeit der Modulrelozierung eingegangen. Sie ermöglicht ein Modul an all jene Positionen innerhalb eines dynamischen Bereichs zu verschieben, die kongruent zu der für die Implementierung des Moduls verwendeten Fläche sind. Es wurde gezeigt, dass Systeme mit freier Platzierung, die große Freiheiten bezüglich der Abbildung von Systemkomponenten und der Platzierung entsprechender Module aufweisen, ohne dieses Hilfsmittel praktisch nicht realisierbar sind, da ein zu großer Speicherbedarf für partielle Bitströme entsteht.

Voraussetzung für die Nutzung der Modulrelozierung ist jedoch ein möglichst homogener Aufbau der rekonfigurierbaren Ressourcen. Die Anzahl benötigter Bitströme steigt mit der Anzahl unterschiedlicher Kacheln. Je weniger kongruente Teilflächen existieren, desto geringer ist die Chance, dass ein Modul durch Modulrelozierung an mehrere Positionen geladen werden kann. Für den heterogenen Aufbau heutiger FPGAs sollte daher eine möglichst gleichmäßige, homogene Kachelung mit möglichst wenigen unterschiedlichen Kacheltypen gewählt werden. Dies betrifft auch die Kommunikationsinfrastruktur, die in die Kacheln integriert ist. Die dabei verwendeten Slices und Leitungen haben denselben Einfluss wie etwa eingebettete Multiplizierer oder BlockRAM. Die Forderung nach einer homogenen Kachelung impliziert somit auch die Forderung nach einer homogenen Kommunikationsinfrastruktur. Im idealen Fall wird die durch den internen Aufbau des FPGAs gegebene Heterogenität durch die Kommunikationsinfrastruktur nicht weiter erhöht. Da homogene Kommunikationsinfrastrukturen für freie Platzierungsverfahren bislang nicht existieren, widmet sich das folgende Kapitel ausgiebig diesem Thema.

4 Homogene Kommunikationsinfrastrukturen

Anders als bei statischen Systemen wird bei dynamisch rekonfigurierbaren Systemen eine Kommunikationsinfrastruktur benötigt, die das Einbinden von Systemkomponenten zur Laufzeit ermöglicht. Dies heißt zum einen, dass die Signalleitungen zwischen neu geladenen und bestehenden Komponenten korrekt geschaltet werden müssen. Hierbei müssen insbesondere während der Konfigurierung korrekte Verschaltungen garantiert sein. Zum anderen muss die logische Einbindung dynamischer Komponenten, z.B. in den Adressraum des Systems, gewährleistet sein.

Kommunikationsinfrastrukturen können zudem nicht wie Module zur Laufzeit in das System geladen oder dynamisch angepasst werden, da ein Online-Routing einen viel zu hohen Rechenbedarf mit sich bringt. Stattdessen muss eine statische Kommunikationsinfrastruktur in den rekonfigurierbaren Bereichen mithilfe von Kommunikationsmakros realisiert werden. Wie im vorherigen Kapitel erläutert wurde, muss darauf geachtet werden, dass durch die Kommunikationsleitungen keine zusätzliche Heterogenität in das System gebracht wird. Für die Realisierung komplexer rekonfigurierbarer Systeme existieren keine Methoden oder Lösungen für die Erstellung homogener Kommunikationsinfrastrukturen. Mit „komplex“ sind insbesondere Systeme mit freier Platzierung gemeint, die weitaus mehr Modulpositionen ermöglichen als Systeme mit festen Modulplätzen. In diesem Kapitel wird der Entwurf von Kommunikationsinfrastrukturen für dynamisch rekonfigurierbare Systeme beschrieben. Es werden technische Lösungsmuster vorgestellt, die die Implementierung homogener, eindimensionaler Kommunikationsinfrastrukturen ermöglichen. Sie werden im Bezug auf die Kosten entsprechender Realisierungen sowie auf ihre Skalierbarkeit bewertet. Die Entwurfsschritte und die durch die Verwendung dynamischer Konfigurierung bedingten technischen Herausforderungen werden im Folgenden anhand verschiedener Abstraktionsebenen beim Entwurf erläutert.

4.1 Abstraktionsebenen beim Entwurf

Abbildung 4-1 zeigt verschiedene Abstraktionsebenen, auf denen beim Entwurf einer Kommunikationsinfrastruktur für dynamische Systemkomponenten Entscheidungen getroffen werden müssen. Die Ebenen beziehen sich ausdrücklich auf die wesentlichen Entwurfsentscheidungen und unterscheiden nicht zwischen unterschiedlichen Sichten auf ein Problem. Als unterste Entwurfsebene sind hier die verfügbaren Routingressourcen eines FPGAs dargestellt. Der Entwickler eines dynamisch rekonfigurierbaren Systems hat keinen Einfluss auf den Aufbau des FPGAs, da diese Entscheidungen bei den FPGA-Herstellern getroffen werden. Da die verfügbaren FPGA-Familien aber unterschiedlich aufgebaut sind, kann durch die Wahl des FPGAs Einfluss auf die verfügbaren Routingressourcen genommen werden. Auf der anderen Seite dieser Entwurfsdarstellung steht die Auswahl eines Kommunikationsprotokolls, mit dem in dem rekonfigurierbaren System kommuniziert werden soll. Hier werden die Schnittstellen der dynamischen Komponenten festgelegt. In der später präsentierten Beispielarchitektur wird beispielsweise das Wishbone-Protokoll [Her02] ausgewählt. In einem stati-

schen System reicht eine korrekte Beschreibung der Schnittstellen aller Kommunikationsteilnehmer (Module), der Arbitrierung und Adressdekodierung, sowie eine korrekte Beschreibung der Verschaltung. Diese Beschreibungen können dabei als HDL vorliegen und mit den vorhandenen Werkzeugen automatisiert auf das FPGA abgebildet werden. Die Abbildung des Protokolls findet in statischen Systemen somit üblicherweise als weitgehend automatisierter Top-Down-Entwurf statt.

In dynamisch rekonfigurierbaren Systemen ist der Entwurf der Kommunikationsinfrastruktur aus den zuvor diskutierten Gründen aufwendiger und erfolgt insbesondere bislang nicht automatisiert. Bei der Abbildung eines Busses auf das FPGA müssen Homogenität und Konfigurierungsaspekte berücksichtigt werden. Zunächst muss hierfür die Topologie des Busses festgelegt werden. Sie ist von der Art und Anzahl der rekonfigurierbaren Bereiche und der gewählten Kachelung abhängig. Da hier oft proprietäre Strukturen verwendet werden müssen, um keine zu hohen Leistungseinbußen zu erleiden, wird in 4.2 das Prinzip des virtuellen Busses eingeführt. Mit ihm werden standardisierte Bus-Schnittstellen an proprietäre Busse angepasst.

Nachdem eine Busarchitektur gewählt wurde, müssen die benötigten Signale des Busses auf die vorhandenen Routingressourcen des FPGAs abgebildet werden. Diese Abbildung findet auf zwei Ebenen statt. Zunächst muss eine *strukturelle Verschaltung* von Signalleitungen gefunden werden, über die Signale übertragen werden. In der FPGA-Terminologie entspricht dies der Synthese und dem Mapping. Hier wird festgelegt, auf welche Typen von Signalleitungen und mithilfe welcher logischen Verschaltungen ein Signal abgebildet wird. Anschließend müssen die gewünschten Strukturen auf die Routingressourcen des FPGAs abgebildet werden. Es handelt es sich dann um die *physikalische Verschaltung* der Signalleitungen. Die Repräsentation eines oder mehrerer Signale auf dieser Ebene ist ein Kommunikationsmakro. Da Makros neben den Signalleitungen auch andere FPGA-Ressourcen (z.B. Slices) beinhalten können, handelt es sich bei der physikalischen Verschaltung streng genommen auch um ein

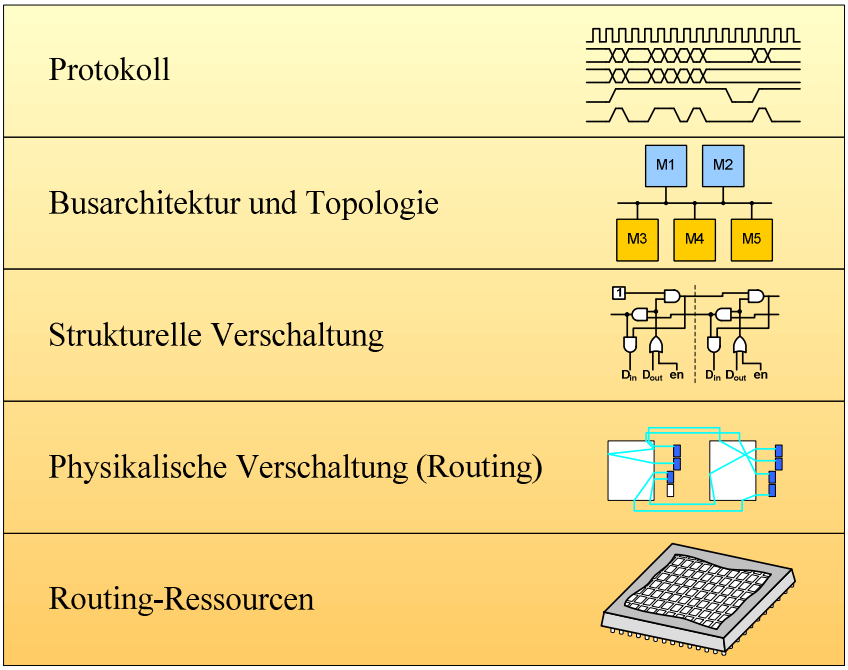

Abbildung 4-1: Abstraktionsebenen für Kommunikationsinfrastrukturen in rekonfigurierbaren Systemen

Platzieren. Sie wird im Folgenden trotzdem nur kurz mit *Routing* bezeichnet.

4.2 Topologien und Protokolle

Der erste Schritt bei der Erstellung einer Kommunikationsinfrastruktur ist die Festlegung einer grundlegenden Topologie, also die Art der Verschaltung der einzelnen Kommunikationsteilnehmer. Abbildung 4-2 zeigt drei grundsätzlich verschiedene Topologien, die in On-Chip-Systemen verwendet werden. Abbildung 4-2 a) zeigt eine Kommunikation über einen Tristate-Bus. Von den fünf Bus-Teilnehmern haben die Module M1 und M2 *Master*-Funktionalität, d.h. sie können aktiv auf den Bus zugreifen. M3, M4 und M5 sind passive Busteilnehmer (*Slaves*), die selbst keinen Buszugriff initiieren können. Kennzeichnend für diese Art der Kommunikation ist, dass das Transferieren der Daten über dasselbe Medium, sprich dieselben Signalleitungen stattfindet. Die angeschlossenen Busteilnehmer müssen sich somit die Bandbreite des Busses teilen, weshalb bei Systemen mit vielen Komponenten schnell der Bus zum Flaschenhals der Systemleistung werden kann. Darüber hinaus müssen hier spezielle Treiberbausteine verwendet werden, die die Mehrfachnutzung einer Leitung ermöglichen. Diese so genannten Tristate-Treiber (von engl. *three states*) können eine angeschlossene Leitung auf logisch „1“, logisch „0“ und hochohmig schalten.

Der in Abbildung 4-2 b) dargestellte Multiplex-Bus verwendet dagegen nur unidirektionale Leitungen, an die die Teilnehmer über Multiplexer angeschlossen sind. Dabei sind für die Datenübertragung von den Master-Teilnehmern zu den Slaves (Schreibrichtung) und umgekehrt (Leserichtung) separate Leitungen vorgesehen. Dies ermöglicht die Verwendung unidirektionaler Leitungen und somit die Verwendung binärer Schaltungselemente. Außerdem kann dadurch gleichzeitig gelesen und geschrieben werden, wodurch die Bandbreite gegenüber Tristate-Bussen gesteigert wird. Der Preis hierfür ist jedoch ein erheblich höherer Verdrahtungsaufwand, da wesentlich mehr Leitungen als bei einem Tristate-basierten Bussystem benötigt werden. In Abbildung 4-2 b) ist ebenfalls ersichtlich, dass Master nicht gleichzeitig auch passive Teilnehmer sind. Falls eine Komponente passiver und aktiver Teilnehmer sein soll, müssen beide Schnittstellen implementiert werden.

Busse, die mit Tristates oder wie hier beschrieben mit Multiplexern realisiert werden, werden *Shared-Bus* Architekturen genannt, da sich alle Busteilnehmer das Übertragungsmedium teilen. Es ist möglich, mithilfe von Multiplexern eine vollständige Vernetzung aller Teilnehmer zu erreichen, d. h. jeder Master ist mit jedem Slave über separate Leitungen verbunden.

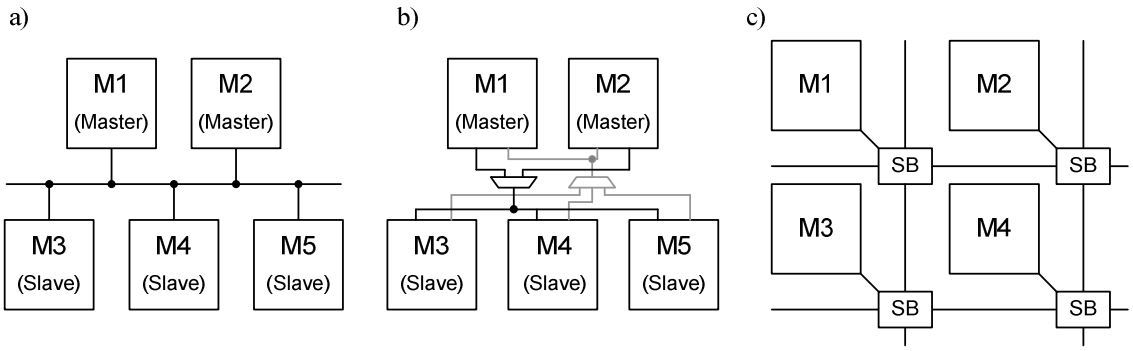

Abbildung 4-2: On-Chip-Kommunikationsinfrastrukturen: a) Tristate-Bus, b) Multiplex-Bus, c) On-Chip-Netzwerk

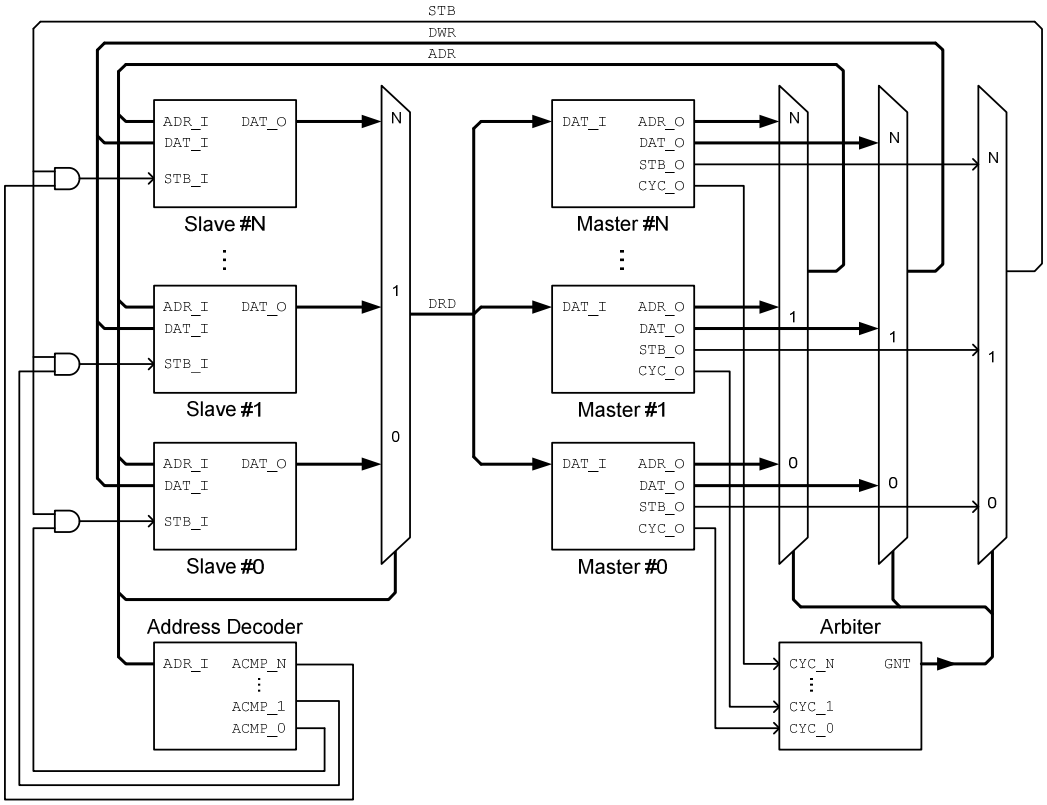

Abbildung 4-3: Mögliche Arbitrierung und Adress-Dekodierung eines Wishbone-Busses nach [Her02]

Die Bandbreite des Systems wird dadurch deutlich gesteigert, da beliebig viele disjunkte Master-Slave-Paare gleichzeitig miteinander kommunizieren können. Aufgrund des ebenfalls deutlich steigenden Verdrahtungsaufwands wird diese auch als Bus-Matrix bezeichnete Verschaltung im Rahmen dieser Arbeit nicht weiter behandelt.

Sowohl Tristate-Busse als auch Multiplex-Busse benötigen eine zentrale Kontrolleinheit, die die Buszugriffe steuert und Kollisionen auf dem Bus verhindert. In Abbildung 4-3 ist eine schematische Darstellung eines Multiplex-Busses nach der Wishbone-Spezifikation [Her02] abgebildet. Neben den Master- und Slave-Teilnehmern und der Verschaltung über Multiplexer sind hier ein Arbiter und ein Adressdekoder abgebildet. Der Arbiter steuert die Zugriffe der Master auf die Slaves, der Adressdekoder wertet zentral die durch die Master angelegten Adressen aus und selektiert den entsprechenden Slave. Zusammen bilden sie die Kontrolllogik für den Wishbone-Bus. Die Verwendung einer solchen Kontrolleinheit bedingt das Zusammenführen der Kontrollsignale des Busses an einen Punkt. Wie später noch gezeigt wird hat dies bei der Nutzung solcher Busse in rekonfigurierbaren Systemen besondere Relevanz während des Entwurfs. Der Großteil der insgesamt benötigten Signale sind Daten- und Adressleitungen (typischerweise je 32/64 Bit). Im Vergleich dazu werden nur sehr wenige Kontrollsignale benötigt.

Mit der steigenden Integrationsdichte digitaler Schaltungen werden immer häufiger paketbasierte Kommunikationsstrukturen als Möglichkeit der On-Chip-Kommunikation betrachtet [PC06]. Bei diesen On-Chip-Netzwerken (*Network-on-Chip* – NoC) ist jeder Teilnehmer über eine Schaltbox an ein globales Netzwerk angeschlossen (siehe Abbildung 4-2 c). Zu versendende Daten werden paketbasiert von einem Modul zunächst an eine angeschlossene Schaltbox übertragen und durch diese über das Netzwerk an das Zielmodul gesendet. Die Kommu-

nikation im Netzwerk findet immer nur zwischen benachbarten Schaltboxen statt. Da die Daten in jeder Schaltbox zwischengespeichert werden, können hier hohe Datenraten erzielt werden. Allerdings steigt bei einer Datenübertragung mit jeder involvierten Schaltbox die Latenz des Transfers. Der Vorteil dieser Architektur ist die gute Skalierbarkeit und somit gute Nutzbarkeit in sehr großen Systemen. Der Vergleich mit herkömmlichen Bussystemen gestaltet sich dennoch schwierig, da die Leistungsfähigkeit eines Busses stark von den Anforderungen der Anwendung abhängt. Paket-basierte Übertragungsverfahren werden hauptsächlich in Systemen eingesetzt, in denen viele gleichberechtigte Teilnehmer unabhängige Aufgaben bearbeiten. In eingebetteten Systemen mit einem oder wenigen Prozessoren werden dagegen meist Shared- oder Multiplex-Busse verwendet. Letztere stellen die Zielarchitektur dieser Arbeit darstellen und werden näher betrachtet. NoCs werden weiterhin zum Vergleich wichtiger Merkmale herangezogen.

4.2.1 Abbildung einer Bustopologie auf einen rekonfigurierbaren Bereich

Bei freien Platzierungsverfahren muss die Topologie eines Busses innerhalb eines rekonfigurierbaren Bereichs an die Kachelung angepasst werden. Im Folgenden wird davon ausgegangen, dass jede Kachel einen Anschlusspunkt (Port) an die Kommunikationsinfrastruktur bereitstellt, um die möglichen Modulpositionen zur Laufzeit nicht einzuschränken. Es wird weiterhin angenommen, dass jeder Kommunikationsport Master- und Slave-Funktionalität besitzt. Es lässt sich dann im Allgemeinen nicht vermeiden, dass Bussignale durch mehrere Kacheln hindurch geroutet werden müssen. Einzige Ausnahme sind Partitionierungen, bei denen alle Kacheln direkt an den statischen Bereich grenzen. In Abbildung 4-4 sind die oben vorgestellten Bustopologien auf einen rekonfigurierbaren Bereich abgebildet worden. Grau dargestellt sind jeweils sechs Kacheln, die weiße Fläche stellt den statischen Bereich dar. Abbildung 4-4 a) zeigt einen Multiplex-Bus. Aus Gründen der Übersichtlichkeit ist hier nur eine der zwei unidirektionalen Verbindungen, z.B. die Leserichtung, dargestellt. Hierfür müssen Datenleitungen von jedem Slave, also jeder Kachel, zu einem Multiplexer im statischen Bereich geführt werden. Der Ausgang des Multiplexers muss mit allen Mastern, also ebenfalls allen Kacheln, verbunden werden. Die gleiche Anordnung ergibt sich für die Schreibrichtung. Hinzu kommen noch Kontrollleitungen von den Kacheln zur zentralen Kontrolllogik (Arbiter und Dekoder) sowie von der Kontrolllogik zu den Kacheln (hier ebenfalls nicht dargestellt). Wie leicht zu erkennen ist, wurde die Verschaltung der zweidimensional angeordneten Kacheln zeilenweise, also eindimensional, durchgeführt. Die Verschaltung der Zeilen erfolgt im statischen Bereich. Prinzipiell sind auch beliebige andere Verschaltungen denkbar, die in der Praxis jedoch nur sehr schwer oder gar nicht realisierbar sind.

Bei der hier gezeigten Anordnung werden durch die an den statischen Bereich grenzenden Kacheln die Signale aller anderen Kacheln derselben Zeile geroutet. Je nach Komplexität des Busses können dadurch schnell die verfügbaren Routingressourcen erschöpft sein. Tristatebasierte Kommunikationsmakros für Datenleitungen bei freier Platzierung (siehe [KPR04a] oder 4.4), mit denen das Routing hier durchgeführt werden kann, ermöglichen das Routing von vier Signalen pro CLB-Zeile. Bei der Nutzung der gesamten Höhe eines Xilinx Virtex-II XC2V4000 (80 CLB-Zeilen) für einen rekonfigurierbaren Bereich können somit maximal 320 Signale horizontal geroutet werden. Die auf Xilinx FPGAs häufig verwendeten CoreConnect-Busse, der *Processor Local Bus* (PLB) [Xil04a] und der *On-Chip Peripheral Bus* (OPB)

[Xil04b], die beide Multiplex-Busse sind, können selbst bei Ausnutzung der vollen Höhe nicht oder nur sehr eingeschränkt für die Anbindung dynamischer Komponenten genutzt werden. In [GKP06] wurde gezeigt, dass bei Nutzung des PLB maximal ein Kommunikationsport als Master angeschlossen werden kann. Für die Datenleitungen in Lese- und Schreibrichtung (je 64 Bit) und die Adressleitungen (32 Bit) werden dann bereits 160 Leitungen benötigt. Hinzu kommen noch etwa 40 Kontrollsignale. Soll ein Port zusätzlich als Slave angebunden werden, kommen erneut 160 Daten- und Adressleitungen sowie einige Kontrollsignale hinzu, so dass ein Port mit Master- und Slave-Funktionalität über 400 Ein- und Ausgangssignale bereitstellen muss. Bei weiteren Ports reduziert sich zwar die Zahl der zusätzlich benötigten Kontrollsignale etwas, insgesamt werden aber etwa 300 Leitungen pro Port benötigt. Mit dem OPB (32 Bit Daten und 32 Bit Adressen) könnten zwei vollwertige Ports verbunden werden. Führt man die Beschränkung ein, dass dynamische Komponenten nur als Slaves angebunden werden sollen, können mit dem OPB immerhin fünf Ports verbunden werden. Man kann festhalten, dass Multiplex-Busse nur in rekonfigurierbaren Bereichen mit sehr wenigen Kacheln eingesetzt werden können. Komplexe freie Platzierungsverfahren mit einer feingranularen Kachelung sind mit ihnen daher nicht realisierbar.

Die oben bereits vorgestellte Wishbone-Architektur kann sowohl als Multiplex-Bus wie auch als Tristate-Bus implementiert werden. Wählt man Letztere für die Verschaltung zweidimensional angeordneter Kacheln, entsteht eine Kommunikationsinfrastruktur wie in Abbildung 4-4 b) dargestellt. Auch hier gilt, dass ein anderes Routing der Leitungen durch die Kacheln möglich, aber meist nicht sinnvoll ist. Durch die gemeinsame und bidirektionale Nutzung der Daten-Leitungen vereinfacht sich der Busaufbau deutlich, da keine separaten Leitungen von den Kommunikationsports in den statischen Bereich geführt werden müssen. Darüber hinaus können dieselben Leitungen für die Lese- und Schreibrichtung und für die Master- und Slave-Anbindung verwendet werden. Eine Shared-Bus Implementierung nach dem Wishbone-Standard (je 32 Bit Daten und Adressen) benötigt 70 Signale für einen vollständigen Port und vier Signale für jeden weiteren Port. Bei dem oben als Beispiel genommenen Virtex-II FPGA kann für einen solchen Bus in jeder der insgesamt 120 CLB-Spalten ein Port bereitgestellt werden, ohne die verfügbaren Routingressourcen voll auszuschöpfen.

Bei einer Network-on-Chip Architektur sind verschiedene Topologien möglich. In Abbildung 4-4 c) ist eine Gitterstruktur gezeigt, in der benachbarte Knoten vertikal und horizontal miteinander verbunden sind. Die in der Literatur vorgeschlagenen Network-on-Chip Architekturen [BMK04, KPA06] für dynamisch rekonfigurierbare Systeme schlagen eine Daten-

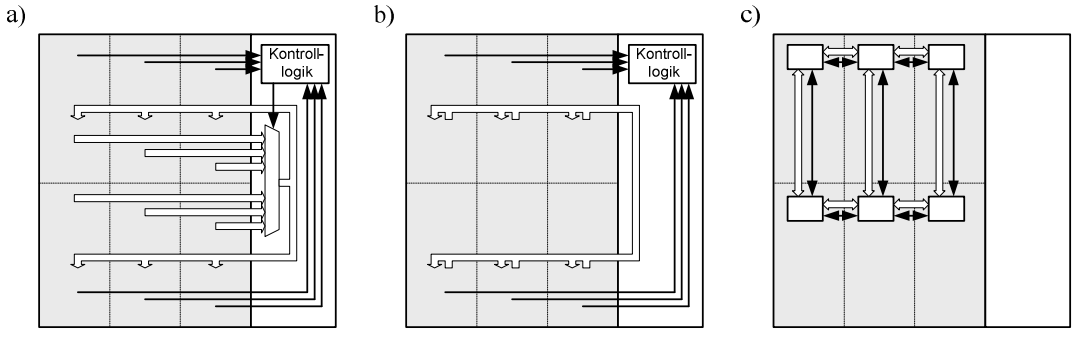

Abbildung 4-4: Abbildung verschiedener Bustopologien auf zweidimensionale Kachelanordnungen: a) Multiplex-Bus, b) Shared-Bus, c) Network-on-Chip

breite von 32 Bit und eine Vollduplex-Übertragung der Daten zwischen zwei Knoten vor. Je nach Implementierung werden zusätzlich einige Kontrollsignale benötigt, so dass auch hier etwa 70 Signale pro Verbindung geroutet werden müssen. Da es bei diesen NoCs keine zentrale Kontrollinstanz gibt, sind nur Verbindungen zwischen benachbarten Kacheln nötig. Dies vereinfacht einen homogenen Aufbau des Netzwerks. Allerdings werden insgesamt mehr Routingressourcen (durch zweidimensionales Routing) und auch zusätzliche Logik- und Speicher-Ressourcen für das Paket-Routing sowie zum Zwischenspeichern der Pakete gebraucht. Daher eignen sich NoC-Ansätze eher für grobgranulare Kachelungen eines FPGA. Genauere Betrachtungen stehen hier allerdings noch aus.

Die hier gezeigten Topologien können miteinander kombiniert und somit fast beliebig erweitert werden. In der Literatur finden sich entsprechende Vorschläge für Kommunikationsinfrastrukturen mit verschiedenen Hierarchieebenen [MBV02]. Kalte et al. [KPR04a] schlagen sogar eine dynamische Anpassung der Hierarchieebenen durch Segmentierung der Busstruktur vor. Der Bedarf an Routingressourcen kann dadurch jedoch nicht gesenkt werden sondern steigt gegebenenfalls. Das dynamische Anpassen der Kommunikationsinfrastruktur wird für verschiedene Topologien auch als Mittel zum Einsparen von Ressourcen verwendet [ABF05, BPS07]. Die veröffentlichten Ansätze nehmen dabei aber immer in Kauf, dass während einer Rekonfigurierung nicht kommuniziert werden kann. Im Folgenden werden ausschließlich Verschaltungen für homogene eindimensionale Kommunikationsinfrastrukturen betrachtet. Die in Abschnitt 4.5 präsentierten Ergebnisse bezüglich der Performanz und des Ressourcenbedarfs können auf die (ebenfalls jeweils eindimensionalen) Verbindungen in NoCs oder hierarchischen Busarchitekturen übertragen werden.

4.2.2 Virtuelle Topologien und Protokolle

Zu jeder Busarchitektur gehört neben der oben diskutierten Topologie ein Protokoll, das vorschreibt, wie die Busteilnehmer den Bus benutzen dürfen. In FPGAs werden häufig standardisierte Busarchitekturen verwendet, z.B. die bereits erwähnten CoreConnect- und Wishbone-Busse oder auch AMBA [ARM08]. Gerade in dynamisch rekonfigurierbaren Systemen macht es Sinn, solche standardisierten Schnittstellen zu den dynamischen Komponenten zu verwenden, um die Nutzung und den Austausch bestehender Komponenten (IP-Cores) zu erleichtern. Proprietäre Lösungen würden insbesondere der Idee offener Systeme entgegenstehen, die den freien Austausch von Komponenten zur Laufzeit vorsieht. Wie später noch gezeigt wird ist die Umsetzung der existierenden Busarchitekturen in dynamisch rekonfigurierbaren Systemen nicht immer effizient möglich. Daher wird hier das Konzept *virtueller Topologien* und Protokolle eingeführt. Dabei wird eine an das gewählte Platzierungsverfahren, insbesondere die Partitionierung, angepasste Busarchitektur implementiert und mittels eines proprietären Protokolls genutzt. Für die Anbindung der Komponenten muss dann eine Schnittstelle bereitgestellt werden, die das proprietäre Protokoll auf ein Standardisiertes umsetzt. Diese Schnittstelle kann dann beim Entwurf eingebunden werden. Das standardisierte Protokoll, das aus Sicht der dynamischen Komponenten weiterhin verwendet wird, wird in dieser Arbeit *virtuelles Protokoll* genannt, da für die Übertragung der Daten über den Bus ein anderes Protokoll verwendet wird.

Wird neben dem Protokoll auch die Topologie des Busses angepasst, z.B. wenn eine Multiplex-Bus Architektur auf einen Shared-Bus abgebildet wird, kann in der gleichen Weise von

einer *virtuellen Topologie* gesprochen werden. Die Leistung eines Busses wird aber natürlich ausschließlich durch den realen Bus mit seiner realen Topologie und dem realen Protokoll bestimmt.

4.2.3 Überwachung der Anschlusspunkte

Für Kommunikationsinfrastrukturen in dynamisch rekonfigurierbaren Systemen ist kennzeichnend, dass sie nicht für eine feste Zahl von Kommunikationsteilnehmern, sondern für eine feste Zahl von Kommunikationsanschlusspunkten (so genannten Ports) dimensioniert werden. Während des Betriebs können beliebig viele dieser Anschlusspunkte tatsächlich genutzt werden. Für die Extremfälle heißt das, dass zu einem Zeitpunkt alle Anschlusspunkte oder auch kein Anschlusspunkt genutzt werden, je nachdem, wie die dynamischen Bereiche genutzt werden. Für alle Anschlusspunkte muss dabei zu jeder Zeit sichergestellt werden, dass die Signalleitungen der Kommunikationsinfrastruktur korrekt verschaltet sind und das Protokoll eingehalten wird. Es gibt dabei zwei grundsätzliche Verfahren: Entweder muss verteilt in den Kacheln sichergestellt werden, dass kein Signal falsch getrieben wird, oder alle kritischen Signale müssen zentral aus dem statischen Bereich heraus kontrolliert werden. Folgende Fälle gilt es für das Verteilte Verfahren zu unterscheiden:

1) Normaler Betrieb.

Im Normalfall wird ein Anschlusspunkt von einem Modul für die Buskommunikation genutzt. Dabei kann vorausgesetzt werden, dass alle Makroeingänge korrekt beschaltet sind und definierte Pegel anliegen. Der Betrieb unterscheidet sich nicht von einem statischen System, so dass keine besonderen Maßnahmen ergriffen werden müssen.

2) Ein Modul hat aufgrund seiner Größe Zugang zu mehreren Anschlusspunkten, von denen es nur einen zur Kommunikation verwendet.

In diesem Fall kann recht einfach sichergestellt werden, dass die ungenutzten Anschlusspunkte nicht fehlerhaft Signale treiben. Hierzu werden alle Signalausgänge dieser Anschlusspunkte fest auf einen definierten Pegel (logisch ‚1' oder logisch ‚0') gelegt. Mögliche Probleme zur Laufzeit sind dadurch ausgeschlossen, der Anschlusspunkt verhält sich wie ein Busteilnehmer, der nie kommuniziert. Beim Entwurf ist jedoch ein nicht unerheblicher zusätzlicher Aufwand erforderlich, da bekannt sein muss, wie viele ungenutzte Anschlusspunkte innerhalb eines Moduls liegen. Das ist wiederum erst nach einer Implementierung bekannt, da es stark von der Größe und der Form eines Moduls abhängt. Es ist somit eine zusätzliche Entwurfsiteration notwendig. Eine einheitliche Schnittstelle für alle Komponenten (mit nur einem Anschlusspunkt) kann es nicht mehr geben.

3) Die Kachel um einen Anschlusspunkt wird nicht mehr verwendet.

Falls ein Modul nicht mehr benötigt wird, kann es vorkommen, dass es zum Teil durch ein neu geladenes Modul überschrieben wird. Auf diese Weise können Modulfragmente mit nicht verwendeten Anschlusspunkten entstehen. In diesem Fall kann im Allgemeinen keine Aussage darüber gemacht werden, wie die Makroeingänge mit den Logikfragmenten des zuletzt geladenen Moduls verschaltet sind. Es kann dann zu fehlerhaften Signalen auf dem Bus kommen. Kritisch sind gemeinsam genutzte Signale (z.B. Tristate-Leitungen) sowie dedizierte Signale vom Anschlusspunkt in den statischen Bereich (z.B. MASTERREQUEST-Signale).

Falls solche Signale fehlerhaft getrieben werden, kann das zu einer Fehlfunktion der gesamten Kommunikationsinfrastruktur führen. Um eine Fehlfunktion auszuschließen, können „leere“ Module geladen werden, die nur das Kommunikationsmakro und eine definierte Beschaltung der Makroeingänge enthalten. Die entsprechenden Bitströme können einfach erzeugt werden. Zur Laufzeit muss aber das Laden dieser Bitströme an nicht mehr genutzte Stellen des FPGAs sichergestellt werden. Dies erfordert einen zusätzlichen Verwaltungsaufwand und Konfigurierungsaufwand.

4) Die Kachel um einen Anschlusspunkt wird rekonfiguriert.

Am kritischsten ist die Situation während einer Konfigurierung, da der Konfigurationsspeicher sequentiell beschrieben wird. Die bestehende Konfiguration wird Frame für Frame durch die neue Konfiguration ersetzt. Daher kann nur am Anfang und am Ende des Konfigurierungsvorgangs eine sichere Aussage über die Verschaltung der kritischen Makroeingänge gemacht werden. Sedcole et al. [SBA05] schlagen hierzu eine Konfigurierung in drei Schritten vor. Durch die erste Konfigurierung werden die kritischen Makroeingänge vom bestehenden Modul getrennt und auf einen definierten Pegel gelegt. Danach folgt die eigentliche Konfigurierung. Durch die dritte Konfigurierung werden die Makroeingänge schließlich mit dem neuen Modul verbunden. Ein fehlerhaftes Verhalten des Busses kann dadurch ausgeschlossen werden. Allerdings sind neben den zusätzlichen Konfigurierungen auch Entwurfsschritte notwendig, die durch keine bestehenden Entwurfswerkzeuge unterstützt werden.

Insbesondere die Problematik während einer Rekonfigurierung macht eine verteilte Kontrolle der Makroeingänge wenig attraktiv. Als Alternative kann eine externe Kontrolle der Makroeingänge durch eine separate Systemkomponente im statischen Bereich erfolgen. Dabei müssen alle Makroeingänge durch ein Enable-Signal blockiert werden können. Das Enable-Signal und die benötigte Enable-Logik muss Teil des Makros sein, um eine permanente Verbindung zu gewährleisten. Da alle Anschlusspunkte separat kontrolliert werden müssen, ist mindestens ein dediziertes Signal pro Anschlusspunkt für die Steuerung notwendig. Im Abschnitt über dedizierte Signale (4.3.4) wird diese Problematik an einem konkreten Beispiel diskutiert. Die Kosten der externen Kontrolle äußern sich in zusätzlich benötigten Ressourcen für die Übertragung der dedizierten Signale (siehe 4.5.2) und für die entsprechende Systemkomponente. Eine entsprechende Beispielimplementierung ist in Abschnitt 5.3.2 beschrieben.

4.2.4 Logische Einbindung der Modulinstanzen

Neben der physikalischen Anbindung an die Kommunikationsinfrastruktur müssen die geladenen Modulinstanzen auch logisch in das System eingebunden werden. Konkret bedeutet dies die Zuweisung eines Adressbereichs im systemweiten Adressraum. Die hier betrachteten Bussysteme verwenden ausschließlich *Master-Slave*-Verfahren. Systemkomponenten können dabei gleichzeitig als Master und Slave an den Bus angeschlossen sein. Da zu jedem Zeitpunkt immer nur ein Master den Buszugriff erhalten kann, müssen die Zugriffe über eine Arbitrierung geregelt werden. Hierfür wird ein zentraler Arbiter verwendet, der die Zugriffswünsche (*Request*) der Master entgegennimmt und gegebenenfalls eine Zugriffserlaubnis erteilt (*Grant*). Um festzustellen, auf welchen Slave zugegriffen wird, müssen die Adressleitungen ausgewertet und mit den Adressbereichen der Slaves verglichen werden. Diese Auswertung kann durch einen zentralen Adressdekoder durchgeführt werden, der den gewünschten Teil-

nehmer durch ein *ChipSelect*-Signal auswählt. Die Adressauswertung kann auch in jedem Slave mit separaten Adressdekodern durchgeführt werden. In diesem Fall entscheidet jeder Slave selbstständig, ob die Busanfrage an ihn gerichtet ist. Die Einteilung des Adressraums kann sowohl statisch als auch dynamisch geschehen. Bei einer statischen Zuweisung wird jeder dynamischen Komponente zur Entwurfszeit ein Adressraum zugewiesen, während dies bei einer dynamischen Zuweisung erst zur Laufzeit geschieht.

Bei einer *statischen Einbindung* dynamischer Komponenten mit einer zentralen Adressdekodierung analysiert der Dekoder die auf dem Bus angelegte Adresse und signalisiert dies dem gesuchten Modul durch ein dediziertes Signal. Das angesprochene Modul wertet seinerseits die angelegte Adresse aus und bearbeitet die Anfrage. Der Vorteil dieser zentralen Arbitrierung ist, dass vorhandene Adressdekoder aus statischen Systemen ohne Modifikationen übernommen werden können. Voraussetzung hierfür ist jedoch, dass die verwendeten dedizierten Signale Modul-dediziert sind, dass also für jedes Modul eine eigene Signalleitung existiert. Mehrfachinstantiierungen von Modulen sind dann nicht mehr möglich, da zwischen mehreren Modulinstanzen eines Moduls nicht unterschieden werden kann. Bei einer dezentralen oder verteilten Dekodierung kann auf zusätzliche dedizierte Signale verzichtet werden. Es muss jedoch dafür Sorge getragen werden, dass nie zwei Module gleichzeitig den Bus für sich beanspruchen. Dies ist aber der Fall, wenn eine Komponente mehrfach in das System geladen wurde, wenn also zwei Instanzen desselben Moduls gleichzeitig an die Kommunikationsinfrastruktur angeschlossen sind. Eine Mehrfachinstantiierung eines Moduls ist bei einer statischen Adressbereichszuweisung mit dezentraler Arbitrierung somit ebenfalls nicht möglich. Falls eine Komponente doch mehrfach geladen werden soll, müssen beim Entwurf mehrere Module mit verschiedenen Adressbereichen erzeugt werden. Auch bei der zentralen Arbitrierung werden Mehrfachinstantiierungen nur mit einer Erweiterung des Arbiters ermöglicht. Darüber hinaus ist die Nutzung des Adressraums bei einer statischen Einbindung ineffizient. Der Adressraum muss unter allen statischen und dynamischen Komponenten aufgeteilt werden, obwohl sich zu keinem Zeitpunkt alle Komponenten im System befinden. Diese Ineffizienz kann mit einer dynamischen Einbindung behoben werden.

Bei einer *dynamischen Einbindung* dynamischer Komponenten in den Adressraum wird jedem Modul zur Laufzeit ein freier Adressbereich zugeteilt. Nachdem das Modul aus dem dynamischen Bereich des FPGAs gelöscht wird, wird auch der von ihm belegte Adressbereich wieder freigegeben. So ist zu jedem Zeitpunkt nur der benötigte Adressraum auch tatsächlich belegt. Eine dynamische Anpassung der Arbitrierung erzwingt eine Anpassung der existierenden Arbiter, egal ob zentral oder verteilt. Zentrale Arbiter müssen mit einer Tabelle mit der aktuellen Adressraumbelegung ausgestattet sein. Diese Tabelle muss bei jeder Rekonfigurierung angepasst werden. Die Verwaltung des Adressraums und die Auswahl geeigneter Adressbereiche für die Module muss von einer übergeordneten Systemkomponente übernommen werden (z.B. von einem Betriebssystem, vgl. 5.3.3). Für die Art der dedizierten Signale ergeben sich keine Einschränkungen, da aus Sicht des Arbiters ein Adressbereich ebenso gut einem Modul wie einer Modulposition zugewiesen werden kann und somit Modul-dedizierte und Positions-dedizierte Signale gleichermaßen verwendet werden können. Die Module selbst sind von der dynamischen Einbindung unberührt. Dies ändert sich, wenn ein Bus mit einer dezentralen Arbitrierung verwendet wird. In diesem Fall müssen geeignete Verfahren imple-

mentiert werden, z.B. die Initialisierung neu geladener Module über einen reservierten Adressraum.

4.3 Homogene Verschaltungsstrukturen

Die Signale einer Kommunikationsinfrastruktur können, insbesondere was die Implementierung dieser Signale angeht, in zwei Klassen unterteilt werden. Die erste Klasse bilden dabei Signale, die mehr als zwei Teilnehmer miteinander verbinden. Diese Signale werden im Folgenden Multipunkt-Signale oder Multipunkt-Verbindungen genannt. Ein Beispiel hierfür sind die Datensignale in klassischen Bussystemen. Auf sie kann von allen Teilnehmern lesend und schreibend zugegriffen werden. Einige Multipunkt-Signale haben die Besonderheit, dass sie nur von einem Teilnehmer getrieben werden können, so z.B. die Adressleitungen in einem Single-Master-System. In diesem Fall müssen die Signalleitungen nur von einem Punkt getrieben werden können. Da Informationen nur in eine Richtung übertragen werden, spricht man von einer unidirektionalen Übertragung bzw. Verbindung. In jedem anderen Fall müssen bidirektionale Verbindungen verwendet werden, die von mehreren Teilnehmern getrieben werden können. Die zweite Signalklasse bilden Signale zwischen genau zwei Teilnehmern, also Punkt-zu-Punkt Verbindungen. In Bussystemen sind die Kontrollsignale oft Punkt-zu-Punkt Signale, z.B. *Chip-Select* Signale von einem Adressdekoder zu einem Modul oder *Master-Request* Signale von einem Modul zu einem Bus-Arbiter. Punkt-zu-Punkt Signale werden in diesem Zusammenhang auch als dedizierte Signale bezeichnet, wenn für jeden Teilnehmer separate Signale existieren. Die in dieser Arbeit betrachteten Punkt-zu-Punkt Signale werden immer nur von einer Seite getrieben, sind also unidirektionale Verbindungen.

Sowohl für Multipunkt-Verbindungen als auch für Punkt-zu-Punkt-Verbindungen können bei der Abbildung auf ein FPGA viele verschiedene Strukturen gefunden werden. Gleiches gilt für das Routing der Signale. Möchte man jedoch der Forderung nach homogenen Kommunikationsstrukturen Rechnung tragen, werden die Entwurfsmöglichkeiten deutlich eingeschränkt. Im Folgenden wird zunächst der Einfluss des globalen Routings auf die Homogenität des Gesamtsystems betrachtet.

4.3.1 Globales Routing

Abbildung 4-5 a) zeigt im linken Teil einen rekonfigurierbarer Bereich. Er ist beispielhaft in sechs Kacheln eingeteilt, die sich auf zwei Zeilen und drei Spalten verteilen. Alle Kacheln sind hier kongruent, da der rekonfigurierbare Bereich keine eingebetteten Speicherelemente oder Ähnliches enthält. Es besteht folglich eine völlig homogene Kachelung. Für diesen rekonfigurierbaren Bereich sei hier eine Kommunikationsinfrastruktur erforderlich, die aus einem Multipunkt-Signal sowie aus einem dedizierten Signal pro Teilnehmer besteht. Zusätzlich sollen alle Kacheln einen Anschluss an die Kommunikationsinfrastruktur enthalten. In Abbildung 4-5 a) ist eine mögliche Realisierung dieser Kommunikationsinfrastruktur abgebildet, bei der die Platzierung der Anschlüsse sowie das Routing der Signale willkürlich durchgeführt wurden. Die Anschlüsse des gemeinsam genutzten Signals sind schwarz unterlegt, die der dedizierten Signale grau. In Abbildung 4-5 a) rechts dargestellt sind zwei Module, die auf die Kacheln links oben bzw. links unten geladen werden können. Rein technisch gesehen können beide Module auch auf jede andere Kachel geladen werden, da der Konfigu-

rationsspeicher für alle Kacheln identisch aufgebaut ist (keine heterogenen FPGA-Elemente). Würde aber z.B. das obere Modul auf die Kachel oben rechts geladen, würden die dort existierenden Anschlüsse und Signalleitungen dieser Kachel gelöscht und stattdessen die Strukturen der Kachel oben links geladen. Als Folge wäre keine Kommunikation mehr möglich, da die Signalleitungen außerhalb der Kachel nicht mehr korrekt verbunden wären (vgl. 2.3.2). Die Kommunikationsinfrastruktur wäre zum Teil zerstört. Hier wird deutlich, dass die Kommunikationsinfrastruktur zwar unveränderlich, also statisch sein soll, dies aufgrund ihrer Realisierung mit rekonfigurierbaren Elementen aber nicht automatisch gegeben ist. Dies muss beim Entwurf durch den Entwickler sicher gestellt werden. In diesem konkreten Fall ist durch die Kommunikationsinfrastruktur auf einem homogenen FPGA ein völlig heterogener dynamischer Bereich eingerichtet worden. Für eine beliebige Komponente, die zur Laufzeit in das System geladen werden soll, müssen daher in diesem Beispiel bis zu sechs verschiedene Module erzeugt werden, um alle Positionen zur Laufzeit nutzen zu können.

Um eben dies zu vermeiden und somit die Anzahl der benötigten Module klein zu halten, müssen homogene Kommunikationsinfrastrukturen verwendet werden. In Abbildung 4-5 b) ist für das gleiche System eine alternative Kommunikationsinfrastruktur implementiert worden. Für das gemeinsam genutzte Signal wurde in beiden Kachel-Zeilen ein einheitliches horizontales Routing verwendet, so dass die Position der Anschlüsse und das Routing dieses Signals in allen sechs Kacheln identisch sind und die Kongruenz der Kacheln vorerst erhalten bleibt. Da die Anschlusspunkte hier unmittelbar innerhalb des dynamischen Bereichs miteinander verbunden sind, wird hierfür in dieser Arbeit der Begriff *eingebettetes Routing* verwendet. Die Kommunikationsmakros, mit denen diese Strukturen beim Entwurf realisiert werden, werden entsprechend *eingebettete Multipunkt-Makros* genannt.

Für die dedizierten Signale wurden hier zwei verschiedene Methoden angewandt. In der unteren Hälfte wurden sie direkt aus dem statischen Bereich in die Kachel mit dem Anschlusspunkt geroutet. Da die Kommunikationsmakros nur zur Überbrückung des Übergangs vom statischen in den dynamischen Bereich benötigt werden, werden sie entsprechend *Kantenmakros* genannt. Das Ergebnis ist eine homogene Struktur, wie das in Abbildung 4-5 b) rechts unten dargestellte Modul deutlich macht. Es kann an jede der drei unteren Positionen geladen werden. Kantenmakros sind immer auch *Punkt-zu-Punkt-Makros*. Ihr Einsatz bedingt jedoch, dass die entsprechenden Kacheln direkt an den statischen Bereich grenzen. Ist dies nicht der Fall, muss eine andere Art der Verschaltung verwendet werden, die hier für die drei

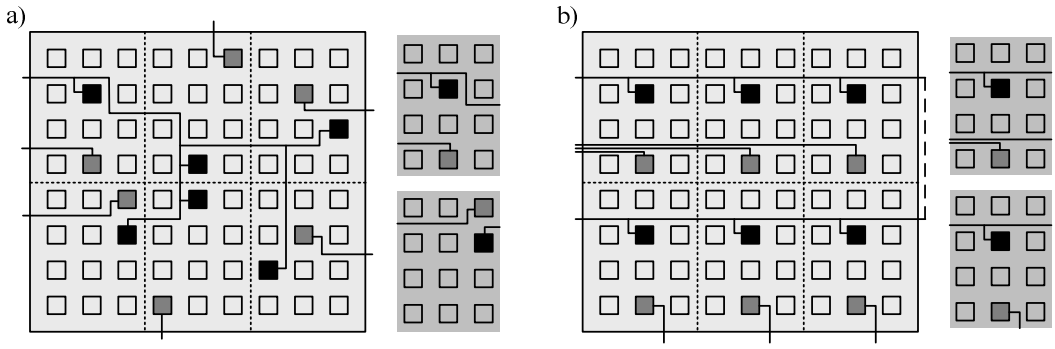

Abbildung 4-5: Kommunikationsinfrastrukturen für eine horizontale 1D-Platzierung: a) willkürliche Verschaltung, b) systematische Verschaltung

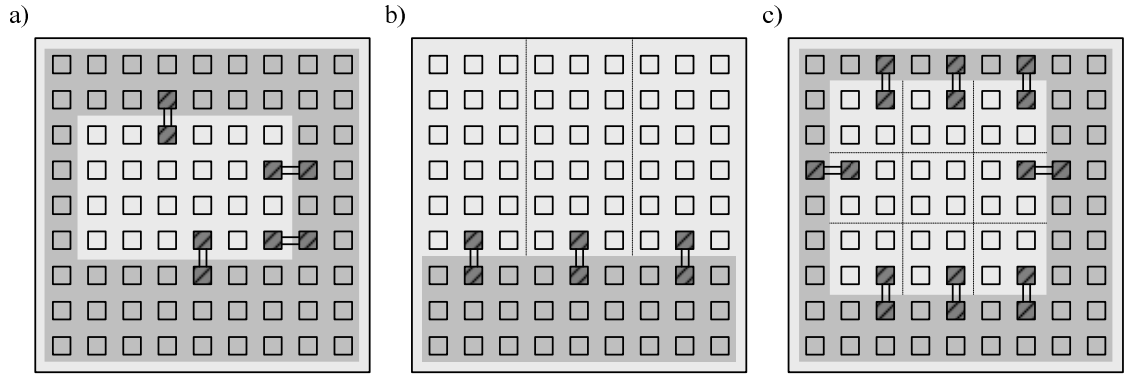

Abbildung 4-6: Routing mit Kantenmakros, a) ein fester Modulplatz, b) freie 1D-Platzierung mit drei Kacheln, c) freie 2D-Platzierung mit neun Kacheln

oberen Kacheln angedeutet ist. Hier werden die dedizierten Signale durch andere Kacheln hindurch in den statischen Bereich geroutet. Es handelt sich dabei ebenfalls um Punkt-zu-Punkt Makros. Da sie sich in der Regel über mehrere Kacheln erstrecken, werden sie im Folgenden *eingebettete Punkt-zu-Punkt-Makros* genannt. Wie man in Abbildung 4-5 b) sehen kann, wird durch den Einsatz eingebetteter Punkt-zu-Punkt-Makros leicht die Homogenität zerstört. Das obere der dargestellten Module wurde für die obere mittlere Kachel erstellt. Es ist leicht ersichtlich, dass dieses Modul in keine der anderen Kacheln geladen werden kann, ohne dabei die Kommunikationsinfrastruktur zu zerstören. Folglich müssen für die Kacheln der oberen Zeile jeweils separate Module für jede dynamische Komponente erzeugt und gespeichert werden. Dass dies nicht zwangsläufig so sein muss, wird weiter unten gezeigt.

Die hier vorgenommene Klassifizierung der Signalabbildung gilt für die willkürliche Verschaltung in Abbildung 4-5 a) ohne Einschränkung auch. Essentiell für die Nutzung dynamischer Rekonfigurierung ist jedoch die Erhaltung der vorhandenen Homogenität der FPGA-Ressourcen. Auch wenn die Homogenität letztlich nur durch den Aufbau der Makros – also der geometrischen Abbildung der Signale auf die Routingressourcen des FPGAs – bestimmt wird, die Möglichkeiten homogenen Routings sind schon stark durch die strukturelle Verschaltung der Signale beeinflusst. In den nachfolgenden Abschnitten wird daher auf die Einsatzmöglichkeiten und die Eigenschaften bezüglich der Homogenität von eingebetteten Multipunkt-Makros, eingebetteten Punkt-zu-Punkt-Makros und Kantenmakros gesondert eingegangen und es werden Lösungsvorschläge zur homogenen Verschaltung gemacht.

4.3.2 Kantenmakros

Das oben beschriebene, direkte Herausführen eines Signals von einer Kachel in den statischen Bereich stellt für den Entwurf in der Regel die einfachste Lösung des Routings dar. Für die Realisierung des Übergangs zwischen rekonfigurierbarem und statischem Bereich können beliebige Punkt-zu-Punkt-Makros verwendet werden. Im statischen Bereich, in dem keine Homogenität erforderlich ist, können die Signale dann ohne Restriktionen weiter verdrahtet werden. Das bedeutet insbesondere, dass mit Kantenmakro sowohl dedizierte Signale als auch gemeinsam genutzte Signale abgebildet werden können.

Ein Nachteil der Kantenmakros ist jedoch, dass mit zunehmender Komplexität des Platzierungsverfahrens die Schwierigkeit steigt, direkt vom statischen Bereich in die Kacheln mit Kommunikationsports zu gelangen. In Abbildung 4-6 ist die Verwendung von Kantenmakros

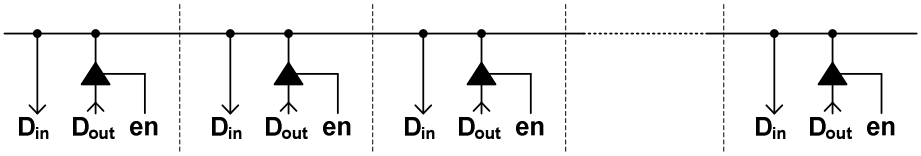

Abbildung 4-7: Multi-Punkt-Makro mit Tristate Lines

für verschiedene Platzierungsverfahren dargestellt. Bei festen Modulplätzen (Abbildung 4-6 a) können die Makros an allen vier Kanten eines rekonfigurierbaren Bereichs platziert werden, falls dieser nicht an den Rand des FPGAs grenzt. Falls es mehrere kongruente, rekonfigurierbare Bereiche gibt, sollten die verwendeten Makros möglichst immer an der gleichen Stelle platziert werden, um auch hier die Homogenität zu erhalten. Bei freier eindimensionaler Platzierung wie in Abbildung 4-6 b) können die Makros nur noch an zwei Seiten der Kacheln angebracht werden, da die Kacheln in einer Dimension aneinander grenzen. Lediglich die Kacheln an Anfang und Ende der Platzierungsachse könnten ggf. von drei Seiten kontaktiert werden. Eine homogene Kommunikationsinfrastruktur ist jedoch mit Kantenmakros nur bei einer Platzierung der Makros senkrecht zur Platzierungsachse möglich, wie hier dargestellt. Die von Koh und Diessel [KD06] vorgestellte COMMA-Architektur ist ein Beispiel für eine homogene Nutzung von Kantenmakros.

Bei freier zweidimensionaler Platzierung kann nur noch in Ausnahmefällen eine homogene Kommunikationsinfrastruktur mit Punkt-zu-Punkt Makros konstruiert werden. Mit Makros wie dem oben beschriebenen Xilinx Makro, die nur der Überbrückung der Grenze zwischen rekonfigurierbarem und statischem Bereich dienen, können nur die Kacheln am Rand des rekonfigurierbaren Bereichs erreicht werden (siehe Abbildung 4-6 c). Alle anderen Kacheln (hier die Kachel in der Mitte) können nicht direkt erreicht werden. Die Kommunikationsmöglichkeiten innerhalb des rekonfigurierbaren Bereichs werden dadurch deutlich eingeschränkt. Alle Module müssen somit mit mindestens einer Kante am Rand des rekonfigurierbaren Bereichs liegen. Auch die Erhaltung der Homogenität ist nicht möglich. In dem abgebildeten Beispiel wurde bereits eine optimale Platzierung für die Makros gewählt. Trotzdem ist nun eine sehr viel ungleichmäßigere Kachelung mit insgesamt fünf verschiedenen Kacheln entstanden.

4.3.3 Eingebettete Makros für gemeinsam genutzte Signale

Gemeinsam genutzte Signale müssen an allen Ports abgegriffen (unidirektionale Signale) oder zusätzlich getrieben werden können (bidirektionale Signale). Ist die Nutzung von Kantenmakros ausgeschlossen und somit eine Verschaltung im statischen Bereich nicht möglich, müssen hierfür eingebettete Multipunkt-Makros verwendet werden. Bei eindimensionaler Platzierung der Module können horizontale Verbindungsstrukturen verwendet werden, die sich, wie später noch gezeigt wird, recht einfach homogen auf das FPGA abbilden lassen. Eine Möglichkeit, gemeinsam genutzte Signale zu implementieren, ist die Verwendung von Tristate-Leitungen, die in einigen FPGA-Familien verfügbar sind. Jeder Teilnehmer verfügt über einen Tristate-Treiber, mit dem er das gewünschte Signal auf die Leitung legen kann. Der Ruhepegel und Normalzustand für alle Teilnehmer ist dabei der hochohmige Zustand. Abbildung 4-7 zeigt eine Tristate-Leitung mit mehreren Teilnehmern. Jeder Teilnehmer kann den Pegel der Leitung über einen Dateneingang (D_{in}) lesen. Sein Datenausgang (D_{out}) wird

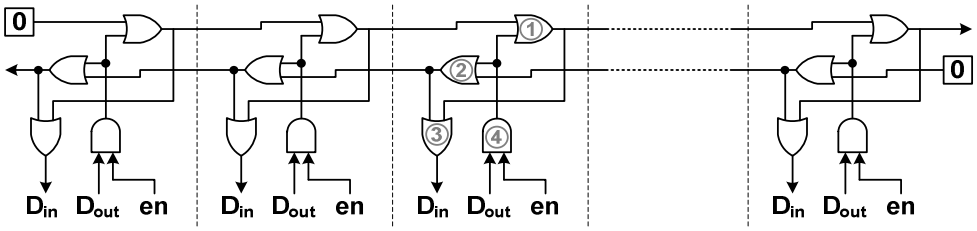

Abbildung 4-8: ODER-basiertes, gemeinsam genutztes Signal

immer dann über einen Treiberbaustein auf die Leitung getrieben, wenn dessen Enable-Signal *en* aktiv (z.B. logisch „1“) ist. Um ein ungewolltes Treiben der Leitung (z.B. während einer Rekonfigurierung) zu vermeiden, sollte dieses Enable-Signal aus dem statischen Bereich heraus kontrolliert werden.

Die Realisierung eingebetteter Multipunkt-Makros mithilfe von Tristate-Leitungen ist prinzipiell eine einfache und effiziente Lösung. Es muss jedoch sichergestellt werden, dass solche Leitungen während des Betriebs nicht doppelt getrieben werden, da beim gegensätzlichen Treiben einer Leitung durch zwei Treiberbausteine relativ hohe Ströme fließen, die das FPGA beschädigen können. Die derzeit aktuellen FPGA-Familien (Virtex-4 und Virtex-5) besitzen keine Tristate-Leitungen in der internen Routing-Matrix. Auf diesen FPGAs müssen also andere Techniken für gemeinsam genutzte Signale verwendet werden.

Eine mögliche Alternative ist die Emulation von Tristate-Signalen durch Signale, die nur zwei logische Pegel annehmen können. Abbildung 4-8 zeigt eine mögliche Variante. Die grundlegende Idee ist, das Signal über zwei separate Leitungsstränge in zwei Richtungen zu propagieren. An jedem Anschlusspunkt werden diese Stränge jeweils über ein ODER-Gatter (Gatter ① und ② in Abbildung 4-8) mit dem lokalen Datenausgang verknüpft. Das propagierte Signal ist somit logisch „1“, wenn der Datenausgang „1“ ist, ansonsten bleibt der ursprüngliche Pegel erhalten. Der Ruhepegel der beiden Signalstränge ist also logisch „0“, da durch die ODER-Verknüpfung nur logisch „1“, nicht aber logisch „0“ aufgeprägt werden kann. Da der Datenausgang eines Teilnehmers hier im oberen Strang nur nach rechts, im unteren Strang nur nach links propagiert wird, müssen die beiden Stränge in jedem Teilnehmer erneut verodert werden (Gatter ③), um gültige Dateneingänge zu erhalten. Prinzipiell können die Datenausgänge direkt mit den ODER-Gattern ① und ② verbunden werden. Insbesondere während eines Konfigurierungsvorgangs kann jedoch kein definierter Pegel für D_{out} garantiert werden, so dass auch hier ein extern kontrolliertes Enable-Signal *en* verwendet werden muss. Es ist hier über ein UND-Gatter (Gatter ④)mit dem Datenausgang verknüpft, so dass nur dann eine logische „1“ auf die Datenleitungen gegeben wird, wenn auch das Enable logisch „1“ ist. Ansonsten wird der Datenausgang auf logisch „0“ und somit auf den Ruhepegel der Datenleitungen gezwungen. Alternativ kann die hier vorgestellte Verschaltung auch

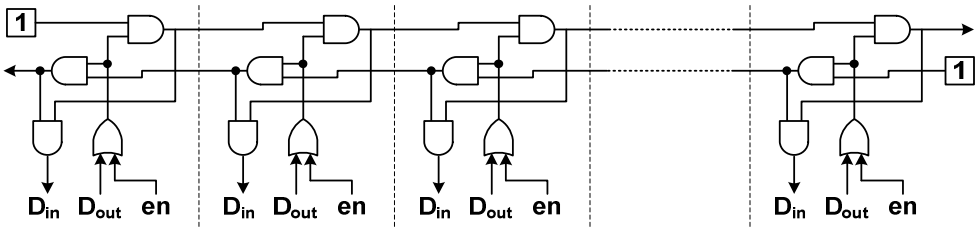

Abbildung 4-9: UND-basiertes, gemeinsam genutztes Signal

komplementär mit UND-Gattern für die Gatter ①, ② und ③ sowie einem ODER-Gatter anstelle des Gatters ④ realisiert werden (siehe Abbildung 4-9). Als wesentlicher Unterschied ist hier der Ruhepegel logisch „1“. Beide gezeigten Varianten erlauben die Realisierung gemeinsam genutzter Signale ohne Verwendung von Tristate-Treibern. Der Verschaltungs- und Routingaufwand ist jedoch prinzipiell höher, und auch die Übertragungslatenz erhöht sich. Beides wird in Abschnitt 4.5 analysiert. Für die Realisierung der logischen Verknüpfungen müssen LUTs in den Slices des FPGAs verwendet werden. Daher werden die hier zuletzt vorgestellten Makros auch *Slice-basierte* Makros genannt.

Schon in den hier gezeigten schematischen Darstellungen wird deutlich, dass sowohl Tristate-basierte als auch Slice-basierte Makros eine inhärente Homogenität aufweisen. Lediglich bei den Slice-basierten Makros muss für die äußeren Anschlüsse eine konstante „0“ bzw. „1“ erzeugt werden. Falls möglich sollten diese Signale im statischen Bereich erzeugt werden, da sonst in den äußeren Kacheln Inhomogenitäten vorliegen. Bei der Implementierung spielen die unterschiedlichen Eigenschaften bezüglich Performanz und Ressourcenbedarf von Slice-basierten und Tristate-basierten Busmakros in der Praxis schon jetzt nur eine untergeordnete Rolle. Entscheidend ist, dass die Virtex-4 und Virtex-5 FPGAs keine internen Tristate-Treiber und -leitungen mehr besitzen, so dass hier Slice-basierte Makros die einzige Realisierungsmöglichkeit darstellen.

4.3.4 Eingebettete Makros für dedizierte Signale

Bei komplexen Platzierungsverfahren mit vielen Kacheln ist man oft auf eingebettete Makros angewiesen, weil Kantenmakros nicht verwendet werden können. Gerade in diesen komplexen Systemen ist die Homogenität des dynamischen Bereichs besonders wichtig, weil sonst Entwurfs- und Speicheraufwand für die Module schnell inakzeptabel hoch werden. Wie schon zu Beginn dieses Kapitels angedeutet, sind homogene Routingstrukturen für dedizierte Signale bei Verwendung eingebetteter Punkt-zu-Punkt-Makros nicht trivialerweise gegeben. Im Folgenden werden daher mehrere Techniken vorgestellt, die für das Routing dedizierter Signale verwendet werden können. Die wichtigsten Kriterien für eingebettete Makros seien hier erneut in Erinnerung gerufen:

1. Homogenität, um die Anzahl der benötigten Bitströme minimal zu halten,
2. Möglichkeit der Mehrfachinstantiierung,
3. Permanente Verbindung, um Fehlverhalten während einer Rekonfigurierung auszuschließen, sowie
4. Ressourcenbedarf und Latenz der Signalübertragung.

4.3.4.1 Direkte, inhomogene, dedizierte Signale

Abbildung 4-10 zeigt eine horizontale Kommunikationsinfrastruktur, in der ein gemeinsam genutztes Signal auf eine Tristate-Leitung abgebildet wurde. Die Datenausgangstreiber der einzelnen Ports werden über Enable-Signale gesteuert. Während des normalen Betriebs kann über ein Protokoll gesteuert werden, welcher Port zu welchem Zeitpunkt die Tristate-Leitung treiben darf. Jedes Modul kann somit über eine eigene Enable-Logik und weitere Steuersignale (die hier nicht abgebildet sind) ein internes Enable-Signal erzeugen. Da dieses interne Enable während des Lade- oder Löschvorgangs eines Moduls möglicherweise nicht korrekt funktioniert, muss zusätzlich ein externes Enable-Signal existieren, dass zu dieser Zeit ein

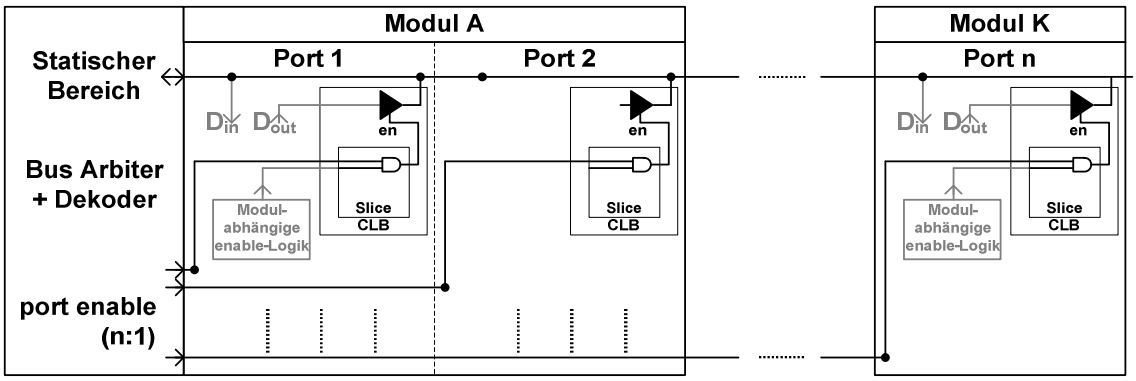

Abbildung 4-10: Direkte, inhomogene, dedizierte Enable-Signale

fehlerhaftes Treiben des Kommunikationsports verhindert. Dieses externe Signal wird von einer Einheit im statischen Bereich, z.B. vom *Bus Arbiter + Dekoder* (siehe 4.2), generiert. In Abbildung 4-10 sind die internen und externen Enable-Signale jeweils über ein UND-Gatter miteinander verknüpft, so dass die Tristate-Leitung nur dann getrieben wird, wenn beide Enable-Signale auf logisch „1“ liegen. In der Abbildung sind alle Signale, die Teil der Kommunikationsinfrastruktur sind, schwarz dargestellt. Nicht-permanente Signale, die zwar in jedem möglichen Modul existieren können aber nicht mit Makros und daher mit jeweils unterschiedlichen Routingressourcen realisiert werden, sind grau dargestellt.

Das Routing-Verfahren in Abbildung 4-10 verwendet *Port-dedizierte Enable-Signale*, da die Bus-Kontroll-Einheit im statischen Bereich jeden Port separat aktivieren oder deaktivieren kann. Hier im Beispiel ist das Modul A so groß, dass es zwei Kacheln belegt und somit auf zwei Ports zugreifen kann. Ein Port bleibt dabei ungenutzt, die Makros und das dedizierte Enable-Signal bleiben jedoch trotzdem vorhanden. Für das Routing der externen Enable-Signale werden Punkt-zu-Punkt Makros verwendet, so dass hier eine Situation entsteht, wie sie schon in Abbildung 4-5 abgebildet ist. Je näher ein Port am statischen Bereich liegt, desto mehr dedizierte Leitungen der anderen Ports werden durch ihn hindurch geroutet. Somit bieten Port-dedizierte Signale zwar eine sichere Verbindung auch während einer Rekonfigurierung, allerdings sind sie in hohem Maße inhomogen. Es müssen somit für alle möglichen Positionen einer Komponente separate Module erzeugt werden. Da folglich keine Modulrelozierung möglich ist, spielt die Mehrfachinstantiierung hier auch keine Rolle.

Diese Art des Routings kann überall dort verwendet werden, wo Homogenität keine Rolle spielt, weil z.B. nur wenige mögliche Modulpositionen existieren. Entsprechende Realisierungen finden sich in der Literatur, z.B. bei Hübner et al. [HBB04]. Die folgenden Verfahren

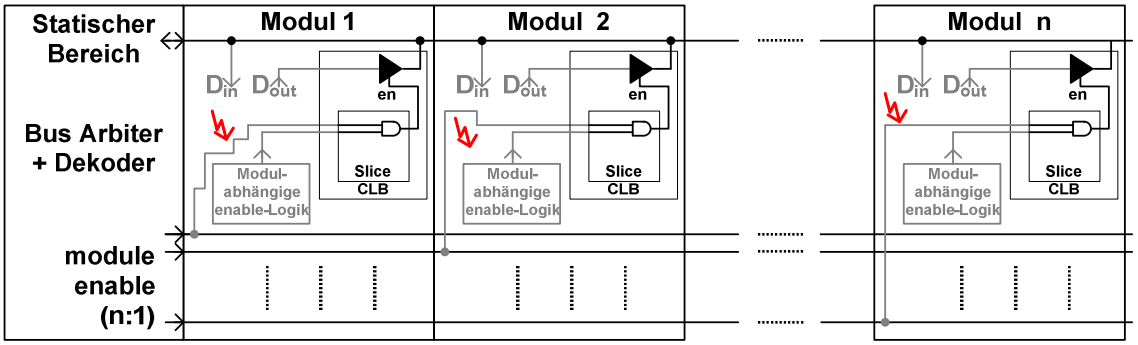

Abbildung 4-11: Modul-dedizierte Enable-Signale

wurden in dieser Arbeit mit dem Ziel homogener Strukturen entwickelt und in [HKP06] vorgestellt.

4.3.4.2 Modul-dedizierte Signale

Abbildung 4-11 zeigt ein Verfahren mit Modul-dedizierten Signalen. Hierbei wird für jedes existierende Modul eine separate Enable-Leitung bereitgestellt. Da zur Entwurfszeit die Platzierung eines Moduls noch nicht bekannt ist, werden alle Enable-Signale durch alle Kacheln geroutet. Bei horizontaler Platzierung liegt jedes Signal vertikal an einer anderen Position. Als Routingressourcen können hierfür z.B. horizontale Long Lines (siehe 2.1.2) verwendet werden. Innerhalb der Module wird dann die Verbindung von der Long Line zum UND-Gatter der Enable-Logik verschaltet. Diese Verschaltung muss jedoch aufgrund der unterschiedlichen horizontalen Routingressourcen der Enable-Signale für jedes Modul unterschiedlich sein, und kann daher nicht mit einem Makro realisiert werden. Stellt man sich vor, dass im Beispiel in Abbildung 4-11 Modul 2 an die Stelle von Modul 1 geladen wird (Modul 1 wird überschrieben), dann wird deutlich, dass während der Konfigurierung keine permanente Verbindung zwischen dem statischen Bereich und dem Enable-Port des Tristate-Treibers bestehen kann. Allerdings ist durch die variable vertikale Verschaltung nun eine beliebige Platzierung jedes Moduls möglich, so dass Modulrelozierung genutzt werden kann. Eine Mehrfachinstantiierung eines Moduls ist jedoch auch hier nicht möglich, da alle Instanzen eines Moduls über dieselbe externe Enable-Leitung gesteuert würden. In diesem konkreten Fall würden folglich immer alle Instanzen eines Moduls gleichzeitig die gemeinsame Tristate-Leitung treiben. Schließlich sei hier erwähnt, dass die Verwendung Modul-dedizierter Signalleitungen die Kenntnis über alle jemals verfügbaren Module schon zur Entwurfszeit bedingt, da jedem Modul schon bei der Implementierung eine Enable-Leitung zugewiesen werden muss. Offene Systeme können mit dieser Methode also nicht entworfen werden.

Anhand der Modul-dedizierten Signale wird das Kernproblem beim Routen dedizierter Signale mit Punkt-zu-Punkt-Makros deutlich: Um separate Signale an alle Module oder Ports zu routen, müssen separate Routingressourcen verwendet werden. Innerhalb der Module sollten jedoch immer dieselben, positionsunabhängigen Ressourcen für das Routing verwendet werden, da sonst entweder die Homogenität verloren geht, oder aber während einer Konfigurierung keine dauerhafte Kontrolle über alle Signale besteht. Prinzipiell unkritisch ist das Routen entlang der Platzierungsachse. Erst das Routing senkrecht zur Platzierungsachse verursacht Inhomogenitäten. Die folgenden Verfahren zeigen, wie dieses Problem technisch gelöst werden kann.

4.3.4.3 Modulplatz-dedizierte Signale

Eine Möglichkeit wird durch den in Abbildung 4-12 dargestellten Ansatz mit *Modulplatz-dedizierten* Enable-Signalen aufgezeigt. Hier wird vor jeder Kachel, die einen Kommunikationsport enthält, eine CLB-Spalte für das vertikale Routen der dedizierten Signale reserviert. Innerhalb der Kacheln ist die Kommunikationsinfrastruktur völlig homogen, da hier nur horizontal geroutet wird. Die zusätzlich eingefügten Routing-Spalten können ausschließlich für das Routing der dedizierten Signale verwendet werden, Module können diese Ressourcen nicht verwenden. Die Routing-Spalten stellen somit Barrieren für die Platzierung der Module dar. Da zwischen zwei Spalten immer nur genau ein Port liegt, kann auch immer nur genau ein Modul in die Zwischenräume geladen werden. Somit handelt es sich automatisch um eine

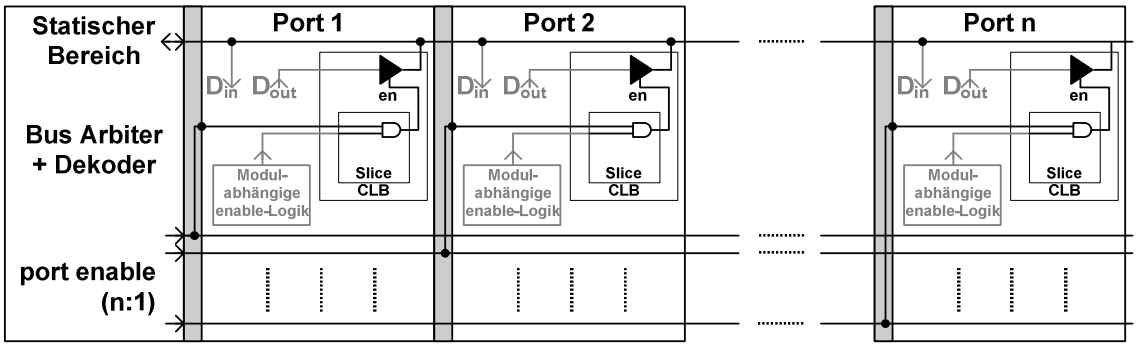

Abbildung 4-12: Modulplatz-dedizierte Enable-Signale

Platzierung mit festen Modulplätzen, wovon der Name *Modulplatz-dedizierte* Signale abgeleitet wurde. Diese Form der eindimensionalen Platzierung stellt einen Spezialfall der Platzierungsverfahren mit festen Modulplätzen dar. Insbesondere auf Virtex-II FPGAs stellt sie jedoch eine leicht zu implementierende Realisierungsoption dar.

Trotz der Einschränkungen bezüglich der Modulplatzierung haben Modulplatz-dedizierte Signale mit eingefügten Routing-Spalten Vorteile gegenüber den zuvor gezeigten Verfahren. Die rekonfigurierbaren Bereiche sind mit Bezug auf die Kommunikation völlig kongruent zueinander, so dass Modulrelozierung ohne weiteres eingesetzt werden kann. Da hier die Modulplätze aktiviert bzw. deaktiviert werden können, sind auch Mehrfachinstantiierungen möglich. Der Preis hierfür ist jedoch der Verlust einer ganzen CLB-Spalte pro Modulplatz, die ausschließlich für das Routing der dedizierten Signale verwendet wird. Zusätzlich muss die meist schlechtere Ressourcenauslastung von Platzierungsverfahren mit festen Modulplätzen in Kauf genommen werden.

4.3.4.4 Schieberegister-basierte dedizierte Signale

Das kritische Routen senkrecht zur Platzierungsachse wird bei Modulplatz-dedizierten Signalen schlicht dadurch entschärft, dass es außerhalb der rekonfigurierbaren Bereiche stattfindet. Um jedoch eine freie Platzierung zu ermöglichen, müssen Wege gefunden werden, ohne inhomogenes vertikales Routen auszukommen. Einer dieser Wege ist in Abbildung 4-13 dargestellt. Die Werte des externen Enable-Signals werden hier lokal in Flipflops (FFs) gespeichert. Diese Enable-FFs (im Bild jeweils der obere der zwei dargestellten FFs) können aus einer weiteren Flipflop-Stufe geladen werden. Die FFs dieser zweiten Stufe sind zu einem großen Schieberegister zusammengeschaltet, das vom statischen Bereich aus geladen werden kann. Die Buskontrolllogik kann nun durch das Setzen eines *shift enable* Signals die Schiebe-

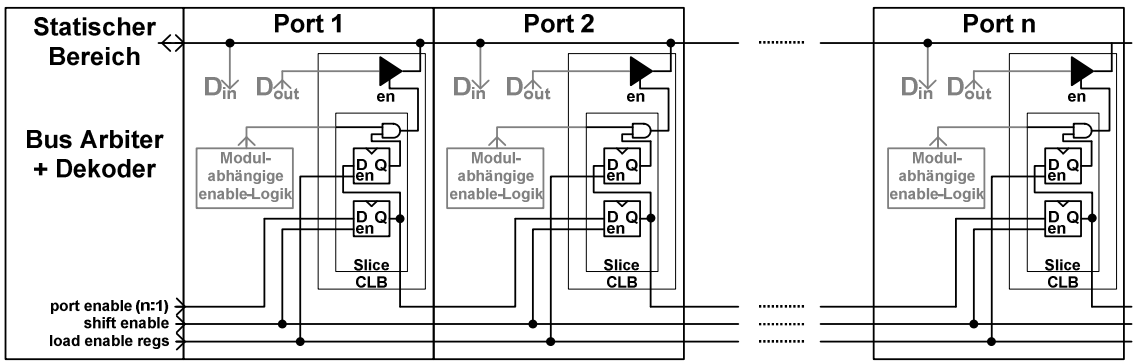

Abbildung 4-13: Schieberegister-basierte dedizierte Signale

operation des Schieberegisters aktivieren. Über den Dateneingang des Schieberegisters (hier: *port enable*) können dann die gewünschten Werte seriell geladen werden. Nachdem das Schieberegister komplett geladen wurde, wird das shift enable wieder deaktiviert. Nun werden durch einmaliges Setzen des *load enable regs* Signals die Tristate-Enable Werte von den Flipflops des Schieberegisters in die Enable-Flipflops übernommen. Ein neuer Aktivierungszustand der Tristate-Treiber aller Ports wurde somit übernommen. Die Verwendung von zwei Flipflop-Stufen ist notwendig, da die Werte der Schieberegister-Flipflops während eines Ladevorgangs mehrmals wechseln können (sog. *Toggeln*, von engl. *to toggle* = kippen). Durch das Einfügen einer weiteren FF-Stufe wird dieses Toggeln von den Tristate-Treibern entkoppelt. Es können somit keine Glitches auf den Enable-Leitungen auftreten.

Die Propagierung der dedizierten Signale über ein Schieberegister führt zu einer homogenen Routingstruktur. Der Grund hierfür ist, dass die dedizierten Signale hier nicht wie bei allen voran beschriebenen Verfahren „aneinander vorbei" geroutet werden müssen, sondern in jeder Kachel die exakt gleichen Routingressourcen (bezüglich der vertikalen Position) verwenden. Die Menge der benötigten Routingressourcen sinkt bei diesem Verfahren sogar. Es werden unabhängig von der Anzahl der Ports drei Signale durch alle Kacheln geroutet, wogegen in allen bisher beschriebenen Verfahren die Zahl benötigter Signale linear mit der Anzahl der Ports steigt. Allerdings werden bei der Verwendung von Schieberegister-basierten dedizierten Signalen in jedem Port zwei Flipflops benötigt, während die anderen Verfahren ohne zusätzliche Speicherelemente auskommen. Ein entscheidender Unterschied ist darüber hinaus die Latenz bei der Übertragung der Enable-Werte. Während mit den vorangehend beschriebenen Verfahren immer unmittelbar auf die Ports zugegriffen werden kann, muss bei der Verwendung eines Schieberegisters eine Verzögerung von einem Takt pro Port hingenommen werden. Diese Eigenschaft ist für eine Reihe typischer dedizierter Signale nicht akzeptabel. Insbesondere Signale, die zur Steuerung der Kommunikation eines Busses dienen (Kontrollsignale), müssen direkt übertragen werden. Für das obige Beispiel der Steuerung der Tristate-Treiberbausteine kann der Schieberegister-Ansatz durchaus verwendet werden, da die Treiber nur während der Rekonfigurierung einer Port-Kachel deaktiviert werden müssen. Die Latenz der Aktivierung oder Deaktivierung über das Schieberegister (typischerweise < 1 µs) ist hier immer sehr klein gegenüber der Rekonfigurierungszeit (typischerweise > 1 ms).

4.3.4.5 Verdrillte dedizierte Signale

Abbildung 4-14 zeigt erstmals eine Möglichkeit, direkte und homogene Verbindungen für freie Platzierung zu implementieren. Die dedizierten Signale werden hier parallel entlang der

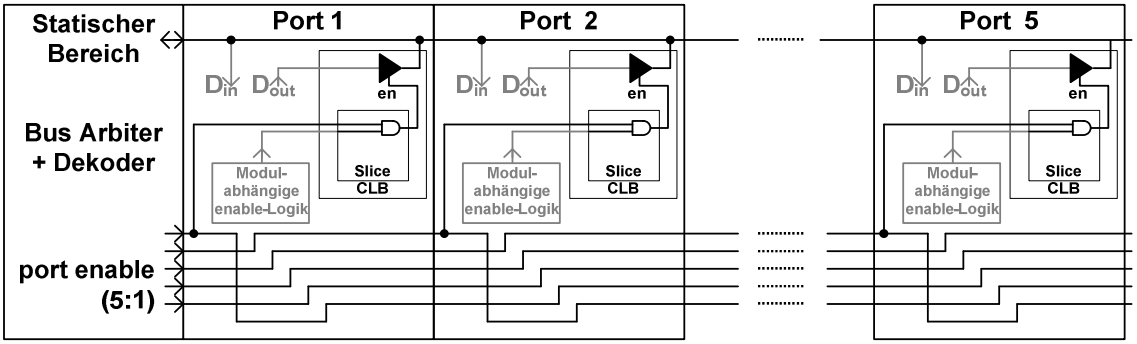

Abbildung 4-14: Verdrillte dedizierte Signale

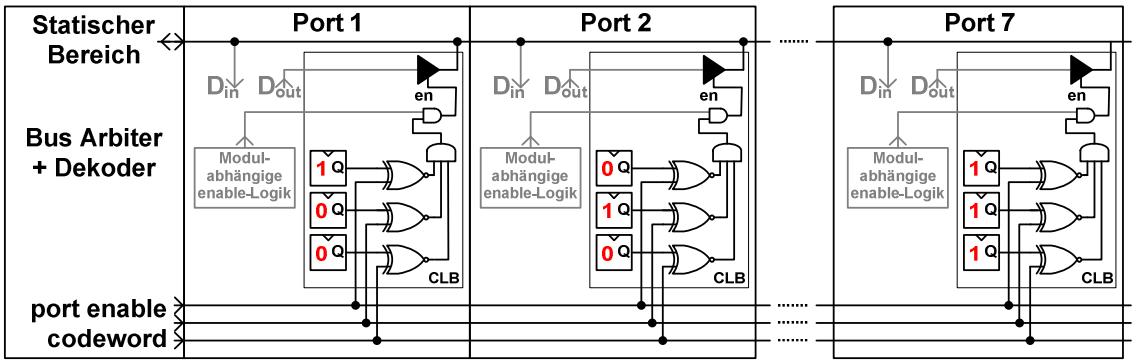

Abbildung 4-15: Binär kodierte, dedizierte Signale

Platzierungsachse geroutet und in jedem Port einmal verdrillt. Somit kann an jedem Port ein anderes Signal an der gleichen Stelle abgegriffen werden. Hierdurch entsteht eine völlig homogene Struktur, die prinzipiell beliebig skalierbar ist. Diese Skalierbarkeit kann bislang noch nicht auf das Routing entsprechender Makros übertragen werden. In dieser Arbeit wurden Makros mit bis zu vier verdrillten Signalen erzeugt. Für komplexere Makros wurde bislang kein homogenes Routing gefunden.

4.3.4.6 Signalkodierung mit lokalen Dekodern

Als letztes wird nun die Routingvariante vorgestellt, die auch in der weiter unten vorgestellten Beispielimplementierung eines dynamisch rekonfigurierbaren Systems verwendet wurde. Bei dieser Methode gibt es keine direkte Zuordnung zwischen Signalleitungen und Ports, Modulen oder Modulplätzen. Stattdessen wird auf die Enable-Leitungen ein Codewort gelegt, das von allen Ports abgegriffen wird. In jedem Port wird das angelegte Codewort mit einem in internen Flipflops gespeicherten Codewort verglichen. Für den Fall, dass beide Codewörter identisch sind, gilt das externe Enable als gesetzt. Abbildung 4-15 zeigt eine Realisierung dieses Verfahrens. In Port 1 ist in drei Flipflops das Codewort „100“ gespeichert. Die Ausgänge der Flipflops werden bitweise über XNOR-Gatter mit den entsprechenden globalen Enable-Leitungen verknüpft. Die Ausgänge dieser Gatter sind logisch „1“, wenn ihre Eingänge jeweils identisch sind. Durch eine UND-Verknüpfung der XNOR-Ausgänge kann somit das externe Enable-Signal bestimmt werden. Entscheidend für dieses Verfahren ist, dass die Register-Inhalte, also die lokalen Codewörter, nicht Bestandteil partieller Bitströme sind. Wird also ein Modul für die Kachel mit Port 1 erzeugt, kann es durch Modulrelozierung an die Stelle von Port 7 geladen werden, ohne die lokalen Registerzustände dabei zu überschreiben. Zur Laufzeit ist eine Kommunikationsinfrastruktur mit kodierten dedizierten Signalen

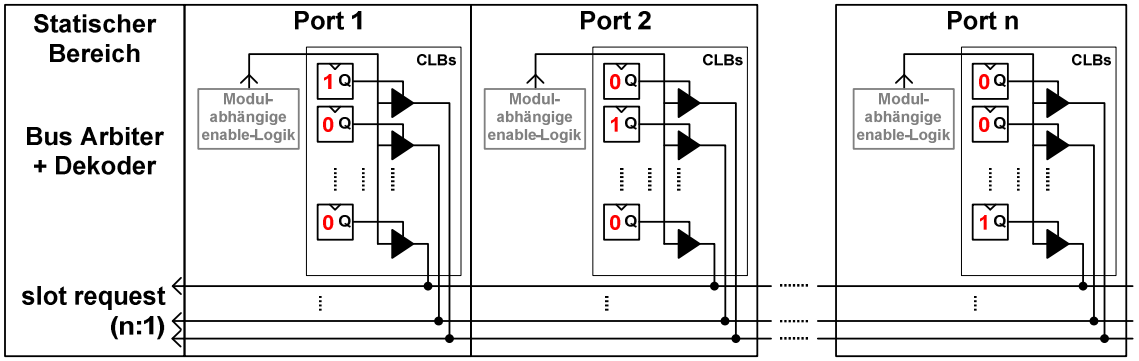

Abbildung 4-16: One-hot kodierte, dedizierte Signale von den Ports in den statischen Bereich

daher völlig homogen. Bei der Erstellung des Initialbitstroms müssen allerdings die unterschiedlichen Codewörter in die Ports eingeprägt werden. Der Entwurf der Module ist hiervon unberührt.

Das Prinzip der Signalkodierung erfordert zusätzliche Logik in jedem Port. Bei der hier verwendeten binären Kodierung kann zu jeder Zeit immer nur ein Port aktiviert werden. Es ist jedoch möglich, hinter dem UND-Gatter ein Flipflop und eine Rückkopplung einzufügen, und damit den Aktivierungszustand zu speichern. Durch Anlegen des korrekten Codeworts kann der Flipflop-Inhalt invertiert werden. Wählt man statt der binären Kodierung eine *one-hot* Kodierung, können alle Ports gleichzeitig kontrolliert werden. In Abbildung 4-16 ist eine one-hot Kodierung für dedizierte Signale von den Ports in den statischen Bereich abgebildet. Als Beispiel wurden hier *Master Request*-Signale gewählt, mit denen Module einen Zugriffswunsch auf den Bus signalisieren können. Hier muss also die Möglichkeit geschaffen werden, von jedem Port ein Request durchführen zu können. Alle Request-Leitungen können von jedem Port über Tristate-Treiber getrieben werden. Die Treiber-Eingänge sind dabei in jedem Port jeweils identisch. Die Enable-Eingänge der Treiber werden jedoch über Flipflops gesteuert, die auch hier wieder einen kodierten Inhalt haben. In diesem Fall ist eine one-hot Kodierung zwingend notwendig, da sonst zwei Ports identische Leitungen treiben können. Auch hier besteht das Problem, dass für das im Modul erzeugte Request Signal während eines Rekonfigurierungsvorgangs kein Pegel garantiert werden kann. Um sicher zu stellen, dass keine ungewollten Requests auftreten, muss die Buslogik die Anfragen nicht benutzter Ports ignorieren. Alternativ können die Request-Signale auch mit einem der oben beschriebenen Verfahren von außen deaktiviert werden. Es ist bemerkenswert, dass bei diesem Verfahren durch die verteilte Dekodierung zwar weiterhin dedizierte Signale übertragen werden, im Gegensatz

Tabelle 4-1: Vergleich der Verfahren zur Verschaltung dedizierter Signale

Art der Verschaltung	Homogenität	Mehrfach-Instanzen	Permanente Verbindung	Latenz	Ressourcen pro Port
Direkt	×	×	✓	0	-
Modul-dediziert	✓	×	×	0	-
Modulplatz-dediziert [1)]	✓	✓	✓	0	1 CLB-Spalte
Schieberegister-basiert	✓	✓	✓	n Takte	2 FFs
Verdrillte Leitungen [2)]	✓	✓	✓	0	-
Kodiert (binär)	✓	✓	✓	0	ld(n) FFs ld(n) LUTs
Kodiert (one-hot)	✓	✓	✓	0	n FFs n+1 LUTs

n: Anzahl der Ports der Kommunikationsinfrastruktur

[1)] nur einsetzbar für Systeme mit festen Modulplätzen

[2)] zurzeit existieren keine homogenen Routing-Techniken für mehr als vier Signale

zu den oben beschriebenen Verfahren werden hierfür aber keine dedizierten Signalleitungen mehr verwendet. Streng genommen handelt es sich daher nicht mehr um eingebettete Punkt-zu-Punkt-Makros, sondern um eingebettete Multipunkt-Makros.

4.3.4.7 Vergleich der vorgestellten Ansätze

In Tabelle 4-1 findet sich eine Zusammenfassung aller hier beschriebenen Ansätze für die Verschaltung dedizierter Signale. Bewertet werden hier die wichtigsten Qualitätsmaße: Homogenität, die Möglichkeit der Mehrfachinstantiierung, eine permanente Verbindung während einer Konfigurierung, sowie die Latenz eines Signals und die für die Implementierung benötigten Ressourcen. Bei den Ressourcen sind hier nur die zusätzlich zu einer direkten Verschaltung benötigten Logik- und Speicherelemente angegeben.

Das hier zuerst betrachtete Verfahren der direkten Verschaltung kann als triviale Herangehensweise für die Implementierung dedizierter Signale betrachtet werden, die mit bestehenden Verfahren realisiert werden kann. Diese herkömmliche Verschaltung hat den geringsten Ressourcenbedarf und ermöglicht auch während einer Konfigurierung eine direkte Verbindung zu den Ports. Allerdings wird dadurch die Homogenität der rekonfigurierbaren Kacheln zerstört, wodurch die Suche nach alternativen Verfahren für freie Platzierung erst begründet ist. Modul-dediziere Signale sind zwar homogen, erlauben aber keine Mehrfachinstantiierung und können nicht für alle Typen von Signalen verwendet werden, da sie keine permanente Verbindung darstellen. Die Übertragung über Schieberegister nimmt in der Auflistung eine Sonderstellung ein. Es ist das einzige vorgestellte Verfahren, das keine direkte Verbindung realisiert, und dadurch unter Umständen eine recht hohe Latenz mit sich bringt. Daher kann es für die meisten Signale nicht verwendet werden. Für einige Aufgaben jedoch, wie zum Beispiel das Deaktivierten von Tristate-Treibern während einer Konfigurierung, kann es eine effiziente Lösung darstellen, da hier unabhängig von der Anzahl der Module und Ports nur drei Leitungen horizontal geroutet werden müssen. Die zwei benötigten Flipflops pro Port können effizient in den ohnehin für kombinatorische Logik (LUTs) verwendeten Slices untergebracht werden.

Für alle Signale jedoch, die eine direkte Verbindung zwingend benötigen, müssen entweder verdrillte Leitungen oder aber eine kodierte Übertragung gewählt werden. Wie bereits erwähnt, können mit verdrillten Signalleitungen bislang nur bis zu vier Signalen übertragen werden. Sie können daher nur für Platzierungsverfahren verwendet werden, die in mindestens einer Dimension nicht mehr als vier Ports aufweisen. Somit stellt sich eine kodierte Übertragung bislang als einzige wirkliche Option für die Übertragung dedizierter Signale in komplexen Kommunikationsinfrastrukturen dar. Die hier vorgestellte binäre Kodierung hat dabei einen recht geringen Ressourcenbedarf, kann jedoch nur für Signale verwendet werden, die im statischen Bereich getrieben werden. Für Signale, die von Modulen getrieben werden, muss eine one-hot Kodierung gewählt werden.

4.4 Homogenes Routing

Im vorangehenden Abschnitt wurde betrachtet, wie Signale *strukturell* verschaltet werden können, um eine insgesamt homogene Kommunikationsinfrastruktur erzeugen zu können. Entscheidend für die Homogenität ist jedoch die Abbildung dieser Strukturen auf die Res-

sourcen eines FPGA. Daher sollte das nun gezeigte Routing der Signale, die *physikalische Verschaltung*, in jedem Port identisch durchgeführt werden. Dies ist zwar aufgrund des heterogenen Aufbaus der FPGAs nicht immer möglich, es kann jedoch immer eine Verschaltung gefunden werden, die keine zusätzliche Heterogenität in das System bringen. Befinden sich zum Beispiel innerhalb eines rekonfigurierbaren Bereichs eingebettete Multiplizierer, kann an dieser Stelle im Allgemeinen kein identisches Routing eines Signals zu einer reinen CLB-Fläche gefunden werden. Allerdings kann auch ohne Kommunikationsinfrastruktur keine einheitliche Kachelung dieser Fläche gefunden werden, so dass im besten Fall zwei verschiedene Kacheltypen existieren (Kacheln mit Multiplizierern und solche ohne Multiplizierer). Per Definition besitzen alle Kacheln eines Typs die gleichen Routing- und Logikressourcen, so dass im Allgemeinen immer ein einheitliches Routing für jeden Kacheltyp gefunden werden kann. Jedoch können am Ende der Kommunikationsinfrastruktur Terminierungen notwendig sein, die nicht immer außerhalb des rekonfigurierbaren Bereichs implementiert werden können (vgl. 4.3.3). In diesem Fall können die betroffenen Kacheln einzigartige Strukturen aufweisen und somit einen eigenen Kacheltyp darstellen.

Im Folgenden wird zunächst ein Kantenmakro von Xilinx vorgestellt. Anschließend wird ein mögliches eingebettetes Makro für gemeinsam genutzte und ein Makro für dedizierte Signale auf Basis von Tristate-Signalen betrachtet. Beide wurden im Rahmen dieser Arbeit erstellt. Weitere Realisierungen finden sich in [Hag06D].

4.4.1 Kantenmakro

Abbildung 4-17 zeigt eine schematische Darstellung eines von Xilinx erstellten Busmakros. Dargestellt sind sechs CLBs, die jeweils aus vier Slices bestehen. Je ein Slice kann als Anschlusspunkt (Eingang oder Ausgang) für zwei Signale verwendet werden. Hier grau hinterlegt ist der statische Bereich, der rekonfigurierbare Bereich ist weiß gehalten. Das hier abgebildete Makro wurde folglich für einen horizontalen Übergang erstellt. Es können allerdings analog hierzu auch Makros für vertikale Übergänge erstellt werden.

Da dieses Makro für beliebige Signale verwendet werden kann, können mit ihm beliebig große Kommunikationsinfrastrukturen durch eine einfache Mehrfachinstantiierung des Makros erstellt werden. Der Entwurfsaufwand für die Erstellung des Makros ist somit sehr gering,

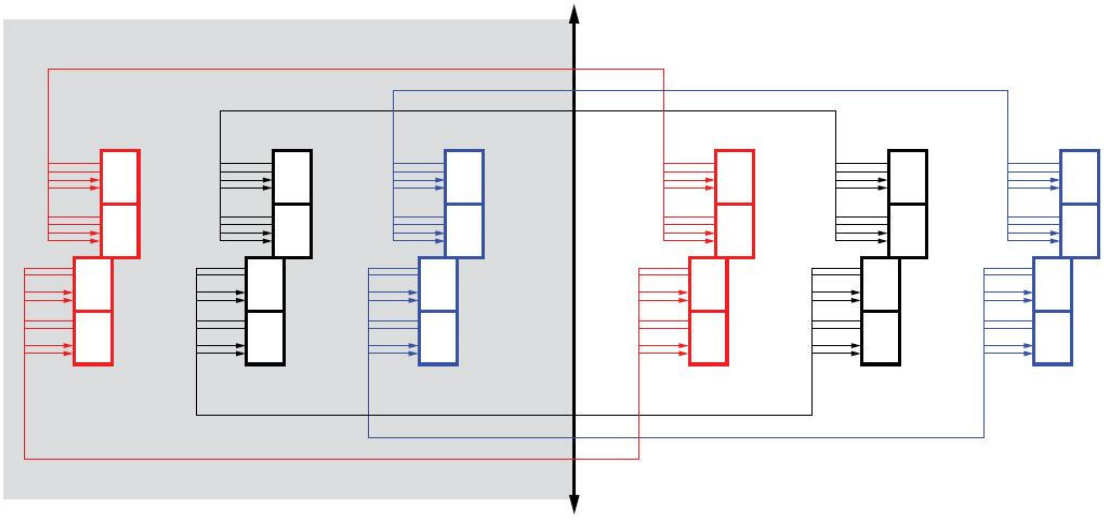

Abbildung 4-17: Xilinx Punkt-zu-Punkt Busmakro für 24 Signale [Xil06]

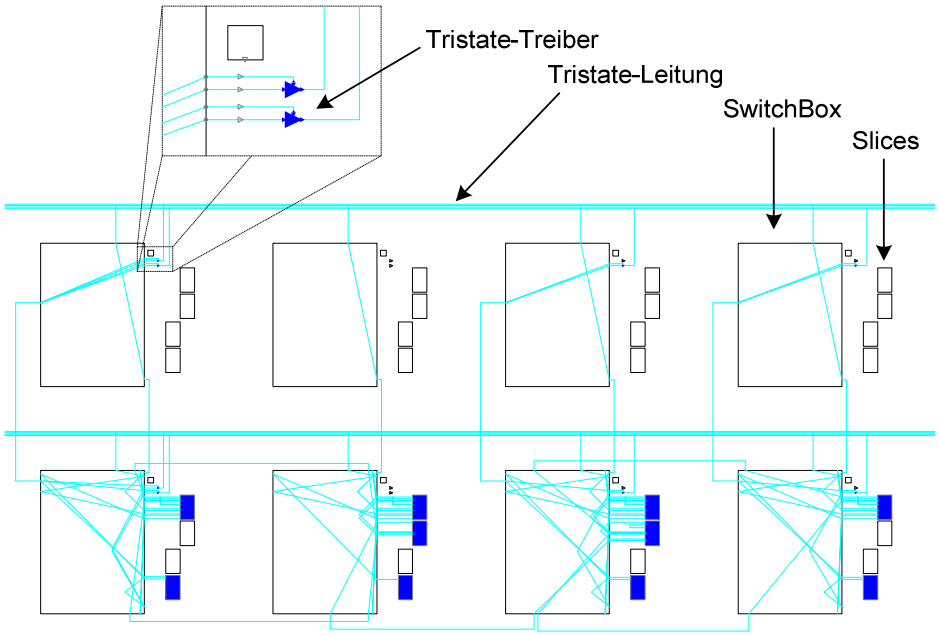

Abbildung 4-18: Anschlusspunkt eines horizontalen, dedizierten Signals

lediglich die Platzierung vieler Makroinstanzen muss aufeinander abgestimmt sein.

4.4.2 Eingebettete Makros für dedizierte Signale

Abbildung 4-18 zeigt die Implementierung eines Anschlusspunkts für ein dediziertes Signal, das über insgesamt acht Tristate-Leitungen binär kodiert übertragen wird. Abgebildet ist ein Ausschnitt eines Virtex-II FPGAs, der sich über zwei CLB-Zeilen und vier CLB-Spalten erstreckt. Wie eingangs erwähnt, besteht jedes CLB dieser FPGA-Familie aus einer Switchbox, die die Verschaltung der Signalleitungen bestimmt, und vier Slices, die Logik und Speicherelemente enthalten. Virtex-II FPGAs enthalten vier Tristate-Leitungen pro CLB-Zeile, von denen jede angrenzende Switchbox genau eine abgreifen und zwei treiben kann. Im CLB oben links werden diese Möglichkeiten voll ausgeschöpft. Die zwei Tristate-Treiber sind vergrößert dargestellt. Erkennbar sind Ein- und Ausgangssignale der Treiber sowie die Enable-Signale, mit denen zwischen treibendem und hochohmigem Zustand umgeschaltet werden kann. Rechts und links an diesen Ausschnitt schließen sich identische Anschlusspunkte an, die zusammen ein horizontales Punkt-zu-Punkt-Makro bilden.

In diesem Beispiel werden acht Signale vom statischen Bereich in den rekonfigurierbaren Bereich übertragen und in einigen Slices der unteren CLB-Reihe mit einem in Flipflops gespeicherten Code verglichen. Mit der 8-Bit-Kodierung können insgesamt 255 verschiedene Anschlusspunkte selektiert werden. Da pro Switchbox nur ein Tristate-Signal abgegriffen werden kann, sind hier insgesamt acht CLBs involviert. Für die Speicherung des Codes und die notwendigen logischen Verknüpfungen werden zehn Slices verwendet, die somit durch das Makro belegt sind und nicht anderweitig verwendet werden können. Die von diesem Makro verwendeten Tristate-Treiber haben keine Funktion während des Betriebs. Im Gegenteil, da die Signalleitungen zu jeder Zeit aus dem statischen Bereich getrieben werden müssen, müssen diese Treiber immer deaktiviert sein. Die Treiber wurden hier nur verwendet, um mit den Xilinx Werkzeugen gültige Makros erstellen zu können. Diese zunächst nutzlose Belegung von Ressourcen stellt im Betrieb keinen wesentlichen Nachteil dar. Da die an die

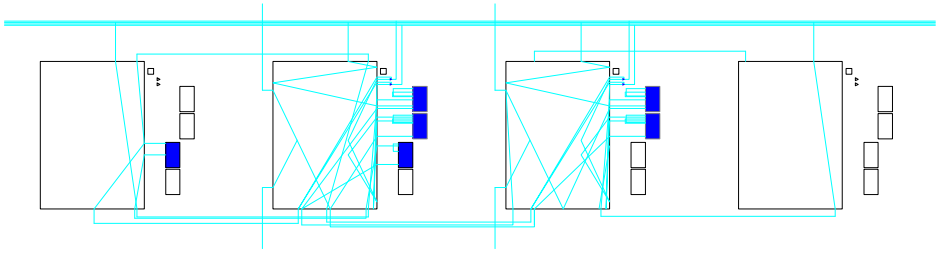

Abbildung 4-19: Zugangspunkt für vier horizontale, gemeinsam genutzte Signale

Treiber angeschlossenen Tristate-Leitungen durch das Makro verwendet werden, kann auch kein Modul diese Treiber sinnvoll nutzen.

Festzuhalten bleibt, dass die in Abbildung 4-18 dargestellte Schaltung die Übertragung eines dedizierten Signals ermöglicht. Mit der hier gezeigten Realisierung werden also pro Kommunikationsport und dediziertem Signal zehn Slices belegt. Dies sind Kosten, die durch den Wunsch nach Homogenität der Kommunikationsinfrastruktur entstehen. Ein weiterer Nachteil gegenüber Kantenmakros ist, dass ein Makro wie das hier dargestellte nicht einfach durch Mehrfachinstantiierung für beliebig viele Anschlusspunkte genutzt werden kann. Um ein immer gleiches, homogenes Routing zwischen benachbarten Anschlusspunkten zu garantieren, müssen alle Anschlusspunkte Bestandteil eines einzigen Makros sein. Je nach der Anzahl von Anschlusspunkten können so Makros entstehen, die aus vielen Tausend Routingsegmenten des FPGAs zusammengesetzt sind. Mit den bestehenden Entwurfswerkzeugen müssen diese Makros manuell durch eine grafische Benutzerschnittstelle erstellt werden. Werden dabei die Namen der Makrosegmente nicht einheitlich gewählt, oder wird die Homogenität an irgendeiner Stelle verletzt, ist ein Makro im späteren Entwurfsablauf nicht mehr nutzbar oder führt zur Laufzeit zu Fehlern. Da ein solcher Makroentwurf nicht nur fehleranfällig sondern auch äußerst zeitaufwendig ist, wurde im Rahmen dieser Arbeit ein Werkzeug erstellt, dass große monolithische Makros automatisch aus einer Makrospezifikation und so genannten Makroprimitiven erstellt [HKK07b]. Abbildung 4-18 entspricht einer solchen Makroprimitive. Das entwickelte Entwurfswerkzeug wird X-BBG genannt und in Abschnitt 5.5.4 weiter behandelt.

4.4.3 Eingebettete Makros für gemeinsam genutzte Signale

Wesentlich effizienter können gemeinsam genutzte Signale übertragen werden. In Abbildung 4-19 ist ein Anschlusspunkt für vier bidirektionale Signale dargestellt. Die vier beteiligten Switchboxen greifen wieder je ein Signal der vier Tristate-Leitungen ab (Signaleingänge des Ports). Getrieben werden die Leitungen von den Treibern der zwei mittleren CLBs (Signalausgänge). Die verwendeten Slices beinhalten Eingangs- und Ausgangsregister der vier übertragenen Signale und des Enable-Signals. Insgesamt werden für die Übertragung gemeinsam genutzter Signale wesentlich weniger Makro-Slices benötigt als bei der Übertragung dedizierter Signale. In diesem Fall werden sechs Slices für vier Signale benötigt.

4.5 Leistungsfähigkeit homogener Busmakros

Datenrate und Latenz der Übertragungen sind wichtige Kriterien für die Güte einer Kommunikationsinfrastruktur [PAK07]. Beide Kriterien werden zum einen durch die Busarchitektur und das verwendete Protokoll beeinflusst. Diesbezügliche Implementierungsvarianten sind

weitestgehend unabhängig davon, ob sie in einem statischen oder dynamisch rekonfigurierbaren System eingesetzt werden und werden daher hier nicht weiter betrachtet. Latenz und Datenrate sind allerdings auch von der Taktfrequenz abhängig, mit der der Bus betrieben werden kann. Die Taktfrequenz wird wiederum durch die konkrete Abbildung des Busses auf das FPGA, also auf den Verschaltungsebenen (logisch und physikalisch) bestimmt. Hier gibt es für die in dieser Arbeit betrachteten, horizontalen Kommunikationsinfrastrukturen nur zwei alternative Implementierungsvarianten: Tristate-basiert und Slice-basiert. Um einen Eindruck der Leistungsfähigkeit komplexer Busmakros zu bekommen wurden zwei entsprechende Busmakros je einmal Tristate-basiert und Slice-basiert für einen Virtex-II FPGA implementiert. Anschließend wurden sie hinsichtlich der maximal möglichen Taktrate und des Ressourcenbedarfs untersucht.

4.5.1 Untersuchte Busstruktur

Um für den Ressourcenbedarf eines eingebetteten Busmakros sowie die erzielbaren Taktraten realistische Werte zu bekommen, wurde beispielhaft ein Wishbone-Bus mit je 32 Daten- und Adressleitungen sowie den entsprechenden Steuer- und Kontrollsignalen implementiert. Er kann in seiner Komplexität stellvertretend für eine Reihe von Bussen gesehen werden, die typischerweise in On-Chip-Architekturen eingesetzt werden. Die Spezifikation und Implementierung des Busses gemäß dem hier eingeführten Abstraktionsmodell (Abbildung 4-1 auf Seite 48) ist wie folgt:

4.5.1.1 Protokoll

Auf oberster Ebene wird ein Wishbone-Protokoll [Her02] verwendet. Alle Module sind Masterfähig, d.h. sie können eigenständig Buszugriffe initiieren. Daten und Adressen sind jeweils 32 Bit breit angeschlossen.

4.5.1.2 Busarchitektur und Topologie

Für die horizontale Kommunikation wird aus den in 4.2.1 genannten Gründen eine Shared-Bus-Architektur verwendet. Um die maximale Taktfrequenz zu erhöhen werden in alle Ein- und Ausgangspfade der Module Register eingefügt. Ohne diese Register würde der kritische Pfad des Busses zudem zum Teil in den Modulen verlaufen. Gültige Angaben über die maximal mögliche Taktfrequenz ließen sich dann nur unter Kenntnis aller Module und unter Betrachtung aller möglichen Anschlussvariationen an das Busmakro machen. In der Praxis ist das bei freier Platzierung aufgrund der großen Anzahl verschiedener Anschlussvariationen jedoch nicht möglich, und das Einfügen der Registerstufen ist somit unumgänglich.

Zusätzlich wird hier in die Kommunikationsinfrastruktur ein Status-Bus integriert, mit dem je vier Statusbits pro Modul unabhängig von der restlichen Kommunikation abgefragt werden können. Dieses Prinzip wird auch in anderen Busarchitekturen verwendet, zum Beispiel beim CoreConnect *Device Control Register* Bus [IBM06]. Speziell bei dynamisch rekonfigurierbaren Systemen können so Informationen über die Module abgefragt werden, die die Entwicklung und Fehlersuche deutlich erleichtern. Während des normalen Betriebs wird über den Statusbus der Zustand der dynamischen Module überwacht.

Für die Erstellung des Busmakros ist entscheidend, ob eine Signalleitung von allen Modulen gemeinsam genutzt wird, oder ob sie zu jedem Modul separat (dediziert) geführt werden

muss. Darüber hinaus ist wichtig, welche Instanz eine Signalleitung treiben kann. Signalleitungen, die nur aus dem statischen Bereich getrieben werden können, sind einfacher aufgebaut, da sie nur in eine Richtung (unidirektional) übertragen müssen (siehe 4.3). Signalleitungen, die von mehreren Teilnehmern (z.B. den Modulen) getrieben werden können, müssen bidirektional übertragen werden können, was zusätzlichen Schaltungsaufwand bedeutet. Die Signale der hier betrachteten Kommunikationsinfrastruktur können entsprechend ihrer Übertragung über Signalleitungen demnach wie folgt klassifiziert werden:

- 84 bidirektionale, gemeinsam genutzte Signale (32 ADDRESS, 32 DATA, 4 BYTEENABLE, 4 STATUS, 8 BLOCKLENGTH, RESET, CYCLE, ACKNOWLEDGE, WRITEENABLE)
- 5 dedizierte Signale vom statischen Bereich (Arbiter, Adressdekoder) zu jedem Modul (MODULESELECT, STROBE, MODULERESET, STATUSSELECT, MASTERGRANT)
- 1 dediziertes Signal von jedem Modul zum statischen Bereich (MASTERREQUEST)

Unidirektionale, gemeinsam genutzte Signale gibt es demnach bei dieser Implementierung nicht. Sie wären notwendig, wenn zum Beispiel keine Masterfähigkeit der Module verlangt wird. In diesem Fall wären alle Module an die Adressleitungen angeschlossen, die jedoch nur aus dem statischen Bereich getrieben würden.

4.5.1.3 Strukturelle Verschaltung

Auf der Ebene der strukturellen Verschaltung tritt erstmals ein Unterschied zwischen den zwei Implementierungen (Tristate-basiert und Slice-basiert) auf. Die gemeinsam benutzten Signale werden wie in 4.3.3 umgesetzt. Dabei wird für die Slice-basierte Implementierung die vorgestellte Variante mit den ODER-Verknüpfungen eingesetzt.

Bei den dedizierten Signalen besteht jeweils ein größerer Gestaltungsfreiraum bezüglich der homogenen Abbildung der Signale. Hier werden bei beiden Implementierungen dieselben Verfahren eingesetzt. Für die dedizierten Signale zu den Modulen wird jeweils eine binär kodierte Übertragung gewählt (siehe 4.3.4). Die MODULESELECT- und STATUSSELECT-Signale werden im Busmakro in einem Flipflop gespeichert, der bei Anlegen des korrekten Codes seinen Wert invertiert (Toggle-FF). Dadurch ist es möglich, dass mehrere Module trotz binärer Codierung gleichzeitig aktiviert sind, wenn auch die Aktivierung bzw. Deaktivierung mehrerer Module nacheinander geschehen muss.

Die MASTERREQUEST-Signale von den Modulen an den Arbiter werden ebenfalls kodiert übertragen. Hier wird eine one-hot Kodierung verwendet, die zwar mehr Ressourcen als eine binäre Kodierung benötigt, dafür aber ohne weitere Kontrollstrukturen gleichzeitige Requests der Module erlaubt.

4.5.1.4 Physikalische Verschaltung

Es soll eine Kommunikationsinfrastruktur für ein eindimensionales, horizontales Platzierungsverfahren erstellt werden. Das bedeutet, dass die Kacheln, die die Anschlusspunkte der Busstruktur beinhalten, eine hohe vertikale Ausdehnung (z.B. die volle Höhe des FPGAs), aber eine geringe horizontale Ausdehnung haben. Auf Virtex-II FPGAs verlaufen vier Tristate Lines pro CLB-Zeile, die pro Spalte einmal verdrillt werden. Ihre Anordnung wiederholt sich demnach alle vier CLB-Spalten. Für die Tristate-basierten Makros wird deshalb eine maximale Breite von vier CLBs vorgegeben. Eine größere Breite ist nicht notwendig, eine kleinere

Tabelle 4-2: Auflistung der Makro-Primitiven

		Bezeichnung	η_{Slice}	η_{FF}	η_{LUT}	S	Besonderheiten
Tristate-basiert	ohne BlockRAM	T_s_4	6	9	10	4	
		T_s_en_4	10	12	16	4	separate Enable-Eingänge
		T_d_bin_8	10	9	18	8	
		T_d_bin_t_8	10	9	18	8	Signal invertiert FF
		T_d_hot_16	48	32	65	16	
	mit BlockRAM	TB_s_4	7	9	11	4	
		TB_s_en_4	11	12	17	4	separate Enable-Eingänge
		TB_d_bin_8	10	9	18	8	
		TB_d_bin_t_8	10	9	18	8	Signal invertiert FF
		TB_d_hot_16	48	32	65	16	
Slice-basiert	ohne BlockRAM	S_s_en_4	6	12	12	4	separate Enable-Eingänge
		S_d_bin_8	16	25	32	8	
		S_d_bin_t_8	18	27	36	8	Signal invertiert FF
		S_d_bin_tv_8	19	27	38	8	s.o., zus. Vcc und Gnd
		S_d_hot_16	32	64	48	16	
	mit BlockRAM	SB_s_en_4	8	12	16	4	separate Enable-Eingänge
		SB_d_bin_8	16	24	32	8	
		SB_d_bin_t_8	18	26	36	8	Signal invertiert FF
		SB_d_bin_tv_8	19	26	38	8	s.o., zus. Vcc und Gnd
		SB_d_hot_16	32	64	48	16	

bringt keine Vorteile, da dadurch inhomogene Strukturen entstehen. Für die Slice-basierten Makros existieren solche Beschränkungen nicht, da Double Lines und Hex Lines an jedem CLB identisch angeschlossen sind. Aus Gründen der Vergleichbarkeit wird jedoch auch für die Slice-basierten Makros eine maximale Breite von vier CLB-Spalten vorgegeben. Für die Kacheln bedeutet dies im Umkehrschluss, dass sie in jedem Fall mindestens vier CLB-Spalten breit sein müssen. Die vertikale Ausdehnung der Makros ist nun im Wesentlichen davon abhängig, wie viele Signale pro CLB-Zeile übertragen werden können. Auch hier unterliegt die Tristate-basierte Implementierung stärkeren Beschränkungen: Da pro CLB-Zeile nur vier Tristate Lines existieren, können nur vier Signale (bei binärer Kodierung 16 Signale) pro Zeile übertragen werden. Diese Einschränkung gilt ebenfalls nicht für das Slice-basierte Busmakro. Trotzdem werden auch hier nur in Ausnahmen mehr als vier Leitungen pro Zeile verwendet.

Um den Entwurf der Busmakros zu erleichtern, wird für jeweils eine Gruppe von Signalen und deren Verschaltung ein Sub-Makro erstellt. Ein solches Sub-Makro wird im Rahmen dieser Arbeit als *Makro-Primitive* bezeichnet. Makro-Primitiven spielen insbesondere in der spä-

ter vorgestellten automatisierten Busmakro-Generierung eine besondere Rolle. Sie erleichtern den Makro-Entwurf erheblich, da sie nur einmal erstellt werden müssen und dann beliebig oft instanziiert werden können. Darüber hinaus ermöglicht die Aufteilung des Busmakros in Sub-Makros eine detaillierte Betrachtung des Ressourcenbedarfs.

Um alle der oben aufgeführten Signale abbilden zu können, wurden je fünf Tristate-basierte und Slice-basierte Makro-Primitiven erstellt. Um auch rekonfigurierbare Bereiche zu unterstützen, die eingebettete Elemente (also BlockRAM- und Multiplizierer-Spalten) enthalten, wurden für jede Primitive zwei Versionen erstellt: Eine für vier aufeinander folgende CLB-Spalten, eine weitere für vier CLB-Spalten, die in der Mitte eine BlockRAM/Multiplizierer-Spalte umschließen. Insgesamt wurden somit 20 Makro-Primitiven erstellt, die in vier Gruppen unterteilt sind: Tristate-basiert und Slice-basiert jeweils einmal mit und einmal ohne BlockRAM-Spalten. Sie sind einschließlich der Angabe der belegten Ressourcen und der Anzahl der Signalleitungen in Tabelle 4-2 aufgelistet. Um sie unterscheiden zu können, wurde eine Bezeichnung eingeführt. Mit dem ersten Buchstaben wird zwischen Tristate-basierten (‚T') und Slice-basierten (‚S') Makros unterschieden. Bei Makros für Kacheln mit BlockRAM-Spalte folgt darauf ein ‚B'. Als nächstes folgt ein Kleinbuchstabe, mit dem zwischen gemeinsam genutzten (‚s' für *shared*) und dedizierten Signalen (‚d') unterschieden wird. Weitere Suffixe deuten z.B. auf die Art der dedizierten Übertragung sowie auf weitere Eigenschaften hin, die in der Rubrik ‚Besonderheiten' aufgeführt sind. Schlussendlich markiert eine Zahl die Anzahl der für die Übertragung eines oder mehrerer Signale verwendeten Leitungen. Als betrachtete Ressourcen sind die Anzahl der Slices η_{Slice} sowie die insgesamt verwendeten Flipflops η_{FF} und Look-Up-Tables η_{LUT} in den Slices. Zusätzlich ist die Anzahl der durch die Makroprimitive übertragenen Signale *S* noch einmal separat aufgeführt.

Der genaue Aufbau aller Makro-Primitiven kann [Hag06D] entnommen werden. Als Beispiel für die Verdrahtung und Verschaltung einer Busmakro-Primitive sei hier auf Abbildung 4-18 und Abbildung 4-19 verwiesen, die die Primitiven *T_d_bin_8* und *T_s_4* darstellen. In jeder der vier Gruppen finden sich Makros für die auf der Architekturebene vorgegebenen Signale mit der auf der strukturellen Verschaltungsebene bestimmten Verschaltung wieder:

- Gemeinsam genutzte Signale (z.B. *T_s_4*),
- binär kodierte, dedizierte Signale zu den Modulen (z.B. *T_d_bin_8*),
- binär kodierte, dedizierte Signale zu den Modulen, mit denen der Wert eines FFs invertiert werden kann (z.B. *T_d_bin_t_8*),
- und one-hot kodierte, dedizierte Signale von den Modulen in den statischen Bereich (z.B. *T_d_hot_16*).

Bei den Tristate-basierten Makros wird eine zusätzliche Primitive für die Übertragung der gemeinsam genutzten Signale benötigt. Bei *T_s_4* gibt es für alle vier Ausgangssignale ein gemeinsames *Enable*-Signal. D.h., alle Ausgänge können nur gemeinsam getrieben werden. Für die gemeinsam genutzten Kontrollsignale kann diese Primitive im Gegensatz zu Daten- oder Adressleitungen nicht genutzt werden. Es wurde daher die Variante *T_s_en_4* erstellt, die separate Enable-Eingänge für jedes Signal bereitstellt. Da hier auch separate FFs benötigt werden, ist der Ressourcenbedarf entsprechend höher.

Bei den Slice-basierten Makros bietet es sich aufgrund des internen Aufbaus an, in jedem Fall separate Enable-Eingänge bereit zu stellen, so dass hier nur jeweils eine entsprechende Primitive existiert (*S_s_en_4* bzw. *SB_s_en_4*). Stattdessen tritt hier eine andere Besonderheit

Tabelle 4-3: Zusammensetzung der Anschlusspunkte

			Ohne BlockRAM				Mit BlockRAM			
			η_{Slice}	η_{FF}	η_{LUT}	P [%]	η_{Slice}	η_{FF}	η_{LUT}	P [%]
Tristate-basiert	20	T(B)_s_4	120	180	200	83,3	140	180	220	78,6
	1	T(B)_s_en_4	10	12	16	80,0	11	12	17	81,8
	3	T(B)_d_bin_8	30	27	54	90,0	30	27	54	90,0
	2	T(B)_d_bin_t_8	20	18	36	90,0	20	18	36	90,0
	1	T(B)_d_hot_16	48	32	65	68,8	48	32	65	68,8
		Total:	228	269	371	81,6	249	269	392	78,7
Slice-basiert	21	S(B)_s_en_4	126	252	252	100,0	168	252	336	100,0
	3	S(B)_d_bin_8	48	75	96	100,0	48	72	96	100,0
	1	S(B)_d_bin_t_8	18	27	36	100,0	18	26	36	100,0
	1	S(B)_d_bin_tv_8	19	27	38	100,0	19	26	38	100,0
	1	S(B)_d_hot_16	32	64	48	100,0	32	64	48	100,0
		Total:	243	445	470	96,7	285	440	554	97,2

auf: Aufgrund des Xilinx Entwurfsablaufs müssen in jedem Anschlusspunkt eines Busmakros zumindest einmal eine logische Eins (Vcc) und eine logische Null (Gnd) bereitgestellt werden. In den Tristate-basierten Makros konnte dies ohne weiteres in einem nicht vollständig genutzten Slice erfolgen. Die hohe Packdichte der Slice-basierten Makros macht es hier erforderlich, zusätzliche Slices für die Erzeugung von Vcc und Gnd zu verwenden. Dies geschieht in den Primitiven *S_d_bin_tv_8* und *SB_d_bin_tv_8*, die ansonsten die gleiche Funktion wie *S_d_bin_t_8* bzw. *SB_d_bin_t_8* aufweisen.

4.5.1.5 Routing Ressourcen

Die Busmakros werden für Virtex-II XC2V4000 FPGAs erstellt. Alle Angaben über den Ressourcenbedarf sind somit nur für Virtex-II FPGAs gültig. Die Slice-basierten Makros können ebenfalls für Virtex-4 und Virtex-5 FPGAs entworfen werden. Für Virtex-4 FPGAs ist aufgrund des sehr ähnlichen Aufbaus ein gleich bleibender Ressourcenbedarf wahrscheinlich. Aufgrund der moderneren Halbleitertechnologie mit kleineren Strukturgrößen (90 nm bei Virtex-4 bzw. 65 nm bei Virtex-5) sind hier kleinere Abstände der Kommunikationsports und somit geringere Verzögerungszeiten zu erwarten.

4.5.2 Ressourcenbedarf

Um ein vollständiges Busmakro für den oben beschriebenen Bus zu erhalten, werden aus mehreren Makro-Primitiven komplette Anschlusspunkte des Makros zusammengesetzt. Mehrere verschaltete Anschlusspunkte bilden schließlich das vollständige Makro. Die Auflistung in Tabelle 4-3 zeigt die Zusammensetzung der vier verschiedenen Anschlusspunkte aus den Makroprimitiven. Sie ist für die Varianten mit bzw. ohne BlockRAM jeweils identisch, führt aufgrund der unterschiedlichen Primitiven aber zu unterschiedlichem Ressourcenbedarf. Für

die Übertragung der MASTERREQUEST-Signale wird jeweils eine Primitive mit one-hot Kodierung verwendet. Da mit ihr jeweils 16 dedizierte Signale übertragen werden können, ist die maximale Anzahl von Anschlusspunkten auf 16 begrenzt. Neben der Anzahl benötigter FFs und LUTs ist in Tabelle 4-3 angegeben, auf wie viele Slices sich diese FFs und LUTs verteilen. Pro Slice stehen je zwei FFs und LUTs zur Verfügung. Die ebenfalls angegebene Packdichte *P* gibt an, wie effizient die Slices genutzt werden. Sie wird berechnet aus

$$P = \frac{\max(\eta_{FF}, \eta_{LUT})}{2 \cdot Slices} \quad . \tag{13}$$

Generell bedeutet eine Packdichte von 100 %, dass das Busmakro nicht mit weniger Slices hätte realisiert werden können. Es bedeutet jedoch nicht, dass alle LUTs und FFs der belegten Slices auch benutzt werden. Es fällt auf, dass die Slice-basierten Makros wesentlich mehr Flipflops und Look-Up-Tables benötigen. Aufgrund der höheren Packdichte fällt der zusätzliche Bedarf an Slices nicht ganz so hoch aus. Da die Größe einiger Makroprimitiven durch FFs, die anderer durch LUTs bestimmt wird, liegt die Gesamt-Packdichte der Slice-basierten Makros unter 100 %, obwohl alle Makro-Primitiven optimal gepackt sind.

In einem Virtex-II XC2V4000 FPGA enthält jede CLB-Spalte 320 Slices. Da hier von einer Mindest-Kachelbreite von vier CLB-Spalten ausgegangen wurde, stehen pro Kachel – bei Ausnutzung der vollen FPGA-Höhe – 1280 Slices zur Verfügung. Bezogen auf diesen Wert benötigen die Tristate-basierten Anschlusspunkte 17,8 % bzw. 19,4 % der vorhandenen Slices, die Slice-basierten 19,0 % bzw. 22,3 %. Diese Werte können nicht vollständig als Kosten der dynamischen Rekonfigurierung gesehen werden. Auch in statischen Systemen müssen Ein- und Ausgangsregister von Komponenten durch FFs in Slices realisiert werden. Der zusätzliche Ressourcenbedarf eines dynamischen Systems besteht darin, dass die hier aufgeführten Ressourcen allein schon für die *Möglichkeit*, eine dynamische Komponente anzuschließen, belegt werden müssen. Eine fein-granulare Kachelung, die viele Platzierungsmöglichkeiten bietet, muss daher viele Ressourcen für Anschlusspunkte an die Kommunikationsinfrastruktur bereitstellen, unabhängig davon, wie viele Module tatsächlich jemals geladen werden. Dieser Einfluss auf die Ressourceneffizienz des Gesamtsystems wird in Kapitel 5 weiter untersucht.

Die einzige Alternative zu den hier vorgestellten, eingebetteten Makros stellen Kantenmakros dar (siehe 4.3.2), wie zum Beispiel das in Abbildung 4-17 dargestellte Makro von Xilinx, wenngleich sie auch nicht in beliebigen Platzierungsverfahren eingesetzt werden können. Bei der hier betrachteten, eindimensionalen Platzierung auf einem Virtex-II FPGA können sie allerdings verwendet werden, wenn sie senkrecht zur Platzierungsachse die Grenze zwischen statischem und dynamischen Bereich überbrücken, wie in Abbildung 4-6 b) auf Seite 59 dargestellt. Die Kacheln erstrecken sich dann jedoch nicht mehr über die komplette Höhe des FPGAs. Mit dem Xilinx Makro können zwischen einem Slice im dynamischen Bereich und einem Slice im statischen Bereich zwei Signale unidirektional übertragen werden. Bidirektionale Signale müssen auf jeweils zwei unidirektionale Signale abgebildet werden. Zwischen dedizierten und gemeinsam genutzten Signalen muss bei Kantenmakros, die immer nur Punkt-zu-Punkt-Verbindungen schaffen, nicht unterschieden werden. Für einen Anschlusspunkt des hier verwendeten Referenzbusses mit 84 bidirektionalen und 6 unidirektionalen Signalen werden somit 174 Slices benötigt. Dabei ist wichtig anzumerken, dass diese 174 Slices wie bei allen Kantenmakros jeweils zur Hälfte im statischen und dynamischen Bereich liegen.

Tabelle 4-4: Slicebedarf verschiedener Busmakros

	pro Kachel			statischer Bereich
	Bidirektional	Dediziert	Gesamt	
Tristate-basiert	130	98	228	214
Tristate-basiert (BRAM)	151	98	249	
Slice-basiert	126	117	243	210
Slice-basiert (BRAM)	168	117	285	
Xilinx Kantenmakro	84	3	87	N · 87

N: Gesamtanzahl der Kacheln

Tabelle 4-4 zeigt die Anzahl benötigter Slices der oben vorgestellten, eingebetteten Makros und der Xilinx Kantenmakros für den hier diskutierten Bus. Aufgelistet sind die benötigten Slices pro Kachel mit einem Kommunikations-Anschlusspunkt sowie die benötigten Slices im statischen Bereich. Die Slices pro Kachel sind wiederum aufgeschlüsselt danach, ob sie für die Übertragung bidirektionaler, gemeinsam genutzter Signale oder für dedizierte Signale verwendet werden. Es fällt auf, dass mit den Kantenmakros in den Kacheln nur etwa ein Drittel der Slices benötigt werden wie in den eingebetteten Makros. Dramatisch ist der Unterschied beim Ressourcenbedarf für die dedizierten Signale. Hier wirkt sich bei den eingebetteten Makros negativ aus, dass aufgrund der Homogenitätsbestrebungen alle dedizierten Signale durch alle Kacheln geleitet werden. Zusätzlich erfordern die hier verwendeten, kodierten Übertragungen zusätzliche Ressourcen. Bei den Kantenmakros werden im Gegensatz dazu in jede Kachel tatsächlich nur die lokal erforderlichen Signale geroutet.

Bei den bidirektionalen Signalen ist der Vorteil des Kantenmakros nicht ganz so stark, aber immer noch deutlich vorhanden. Hier muss jedoch angemerkt werden, dass die eingebetteten Makros bereits Verschaltungen enthalten, die bei Verwendung der Kantenmakros im statischen Bereich außerhalb der Makros durchgeführt werden müssen. Insbesondere müssen die Kantenmakros zu einem Bus verschaltet werden, was bei eingebetteten Makros implizit gegeben ist. Ein weiterer wichtiger Punkt ist, dass bei eingebetteten Makros nur ein einziger Anschlusspunkt im statischen Bereich liegen muss, während bei Kantenmakros bei jedem Anschlusspunkt einer Kachel auch Slices im statischen Bereich belegt werden. Somit fallen bei Verwendung der eingebetteten Makros im statischen Bereich nur 214 bzw. 210 Slices an, wobei für das Kantenmakro 87 Slices pro angeschlossener Kachel belegt werden.

Wird mithilfe dieser Makros eine Kommunikationsinfrastruktur für ein Virtex-II XC2V4000 FPGA geschaffen (vgl. auch Abbildung 5-13 auf Seite 101: 16 Kacheln, fünf davon mit BlockRAM), werden Tristate-basiert 3967 Slices, Slice-basiert 4308 Slices und bei Nutzung von Kantenmakros 2784 Slices durch das Kommunikationsmakro belegt. Allein die Differenz zwischen dem Tristate-basierten, eingebetteten Makro und dem Xilinx Kantenmakro macht bereits 5,1 % der insgesamt auf dem XC2V4000 zur Verfügung stehenden Slices aus. Diese Differenz wird auch nicht durch den zusätzlichen Verschaltungsaufwand der Kantenmakros im statischen Bereich kompensiert. Die Verwendung eingebetteter Makros ist daher insbesondere aufgrund der hohen Kosten für dedizierte Signale wesentlich ressourcenintensiver als die Nutzung herkömmlicher Kantenmakros.

Tabelle 4-5: Zeitparameter der Busmakros

	Tristate-basiertes Makro	Slice-basiertes Makro
t_{unc}	0,063 ns	0,063 ns
t_{in}	1,645 ns	0,371 ns
t_{prop}	(0,511 ns · (i_A + i_B)) + 1,488 ns	(1,247 ns · i_A) + (2,223 ns · i_B)
t_{out}	2,645 ns	1,331 ns
t_{skew}	0 … 0,33 ns	0 … 0,33 ns

4.5.3 Maximale Taktrate

Es ist offensichtlich, dass die maximale Taktrate, mit der die durch Makros realisierten Busse betrieben werden können, von der Anzahl der Anschlusspunkte sowie deren Verteilung auf dem FPGA abhängt. Wie bereits eingangs erwähnt, wurden zur Performanzsteigerung Register in allen Moduleingängen und -ausgängen eingefügt, die die Angabe einer maximalen Taktrate ohne Kenntnis der Module erst möglich macht. Die minimale Periodendauer t_{clock} des Taktsignals kann dann bestimmt werden durch

$$t_{clock} = t_{delay} + t_{unc} + t_{skew}. \tag{14}$$

Mit t_{unc} wird der *Jitter* des Taktsignals berücksichtigt, also die maximale Abweichung einer Periode von der mittleren Taktperiode. Er ist abhängig von dem verwendeten Quarz-Oszillator, der das Taktsignal erzeugt. Für das RAPTOR2000-System (siehe 5.2), dass in dieser Arbeit für die Implementierung dynamisch rekonfigurierbarer Systeme verwendet wurde, wurde ein Jitter von 63 ps gemessen. Dieser Wert wird im Folgenden für die Modellierung verwendet. Mit t_{skew} fließt der Takt-*Skew*, also der Laufzeitunterschieds des Taktsignals vom Takteingang des FPGAs zu den Registern, mit in die Berechnung ein. Sein Wert ist davon abhängig, in welchem Bereich des Taktbaumes die Anschlusspunkte platziert sind (vgl. 2.1.2). Der größte Skew zwischen zwei horizontal platzierten Anschlusspunkten wurde mithilfe des Xilinx Timing Analyzers bestimmt und beträgt 330 ps. Dieser *Worst Case* wird für die Abschätzung der maximalen Taktrate verwendet, so dass es sich um eine konservative Abschätzung handelt.

t_{delay} bezeichnet die Laufzeit des kritischen Pfads von einem Ausgangsregister eines Anschlusspunkts zu dem Eingangsregister eines anderen Anschlusspunkts. Er setzt sich aus der Verzögerung t_{out} vom Ausgangsregister zum Bus, der Signallaufzeit t_{prop} (*propagation delay*) über den Bus und der Eingangsverzögerung t_{in} vom Bus zum Eingangsregister zusammen. Für t_{delay} gilt somit:

$$t_{delay} = t_{out} + t_{prop} + t_{in} \tag{15}$$

t_{out}, t_{prop} und t_{in} können für einen Anschlusspunkt mithilfe des *Xilinx FPGA Editors* sehr genau angegeben werden. Sie basieren damit auf dem Timing-Modell von Xilinx, dass die Grundlage aller Angaben zu zeitlichem Verhalten in den Werkzeugen von Xilinx darstellt. Tabelle 4-5 listet die Werte der Modellparameter für die Tristate- und Slice-basierten Makros auf. t_{prop} ist jeweils abhängig von der Anzahl der zu überbrückenden Kacheln (à vier CLB-

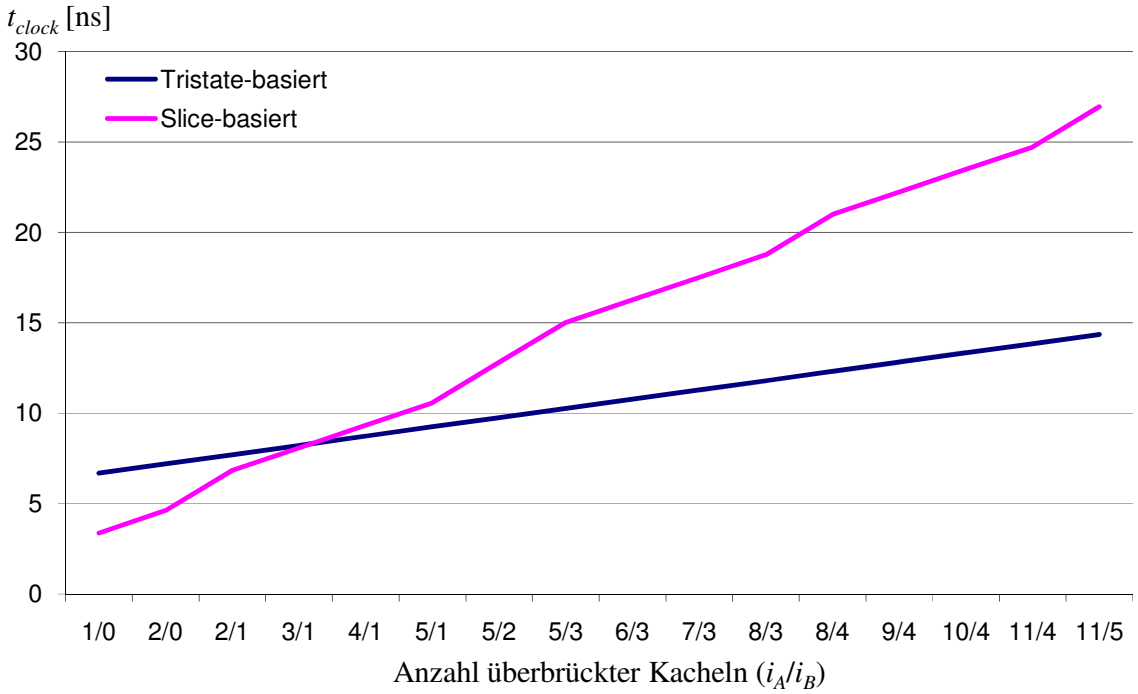

Abbildung 4-20: Minimale Taktperiode der Busmakros

Spalten). Beim Slice-basierten Makro bewirken Kacheln mit BlockRAM eine höhere Signallaufzeit als Kacheln ohne BlockRAM. i_A bezeichnet hier die Anzahl der Kacheln im kritischen Pfad ohne BlockRAM, i_B bezeichnet analog die Anzahl der Kacheln mit BlockRAM. Bei Tristate-basierten Makros besitzen alle Kacheln die gleiche Verzögerung.

Der Vergleich der Parameter zeigt, dass Eingangs- und Ausgangsverzögerungen bei Slice-basierten Makros relativ klein sind. Der Grund hierfür ist, dass sich sowohl Makroeingänge und -ausgänge wie auch die logischen Verknüpfungen für die Erzeugung des Pseudo-bidirektionalen Bussignals unmittelbar in oder an Slices befinden, und somit effizient geroutet werden kann. Der Bus selbst weist durch die nötigen logischen Verknüpfungen im kritischen Pfad eine relative hohe Verzögerung auf. Das Tristate-basierte Makro besitzt aufgrund der Nutzung der dedizierten Tristate Lines auf dem FPGA eine wesentlich geringere Verzögerung auf dem Bus. Hier ist jedoch das Routing von den Tristate-Treibern zu den Makroeingängen und -ausgängen etwas aufwändiger, so dass höhere Eingangs- und Ausgangsverzögerungen auftreten.

Das Wissen um diese Parameter kann nun genutzt werden, um die erzielbare Taktfrequenz eines Busmakros zu modellieren. Als Beispiel soll nun wiederum ein XC2V4000 FPGA mit maximal 16 Kacheln betrachtet werden. In Abbildung 4-20 ist die minimale Taktperiode der betrachteten Makro-Varianten dargestellt. Es ist dabei die nach (16) und (17) berechnete Zeit t_{clock} über der durch die Anzahl enthaltener Kacheln gegebenen Breite des Makros aufgetragen. Ausgehend von der Kachel am linken Rand der rekonfigurierbaren Fläche wird das Makro dabei entlang der X-Achse sukzessive um jeweils eine Kachel erweitert. Die Werte bei 1/0 gelten folglich für eine Kommunikation zwischen zwei benachbarten Kacheln am linken FPGA-Rand, die Werte bei 11/5 sind für Busmakros in der maximalen Ausbaustufe (16 Anschlusspunkte im dynamischen Bereich, einer im statischen Bereich, vgl. Abbildung 5-13) gültig. Der Vergleich der Makros zeigt, dass Slice-basierte Implementierungen bis zu einer

Größe von fünf Slots, einer Breite der rekonfigurierbaren Fläche von 20 CLBs entsprechend, eine höhere oder zumindest gleichwertige Performanz aufweisen. Erst bei größeren Makros wirkt sich ihre hohe Verzögerung auf dem Bus deutlich negativ aus. Für das Größte hier betrachtete Makro sind Tristate-basiert etwa 70 MHz Bustakt möglich, Slice-basiert nur etwa 37 MHz.

Zur Verifizierung des Modells wurden die in Abbildung 4-20 verglichenen Makrogrößen konkret implementiert und mit dem Xilinx *Static Timing Analyzer* untersucht. Die ermittelten Referenzwerte lagen maximal 330 ps unter den durch das Modell prognostizierten Werten. Dies entspricht genau der maximalen Abweichung, die aufgrund der Worst Case Annahme des Clock-Skews zu erwarten ist. In keinem Fall lag der Referenzwert über dem Modellwert, so dass die Konservativität des Modells bestätigt wurde.

4.6 Zusammenfassung

Die Kommunikationsinfrastruktur eines dynamisch rekonfigurierbaren Systems sollte ähnlich leistungsfähig sein wie die eines statischen Systems. Daher werden in dieser Arbeit typische On-Chip-Bussysteme als Maß für die Bewertung der Kommunikationsinfrastruktur verwendet. Es wurde gezeigt, dass die Verwendung von Multiplex-Bussen für freie Platzierungsverfahren aufgrund der hohen Anzahl von Signalleitungen schnell zu einem nicht mehr erfüllbaren Ressourcenbedarf führt. Dies gilt immer dann, wenn nicht alle Kacheln direkt an den statischen Bereich grenzen und einige Signale durch andere Kacheln hindurch geroutet werden müssen, um den statischen Bereich mit der Zielkachel zu verbinden. Das stellt insbesondere bei Systemen mit freier Platzierung die Regel dar. Verwendet man dagegen Shared-Bus Architekturen, bei denen die Anzahl der Daten- und Adressleitungen unabhängig von der Anzahl der Busteilnehmer ist, sinkt der Ressourcenbedarf erheblich. Es müssen nur noch die Kontrollsignale separat zu jedem Anschlusspunkt geführt werden, so dass der Gesamtbedarf an Ressourcen nur noch gering mit der Anzahl der Anschlusspunkte steigt.

Neben dem Ressourcenbedarf stellt die in Kapitel 1 geforderte Homogenität der Kacheln die größte Herausforderung beim Entwurf einer Kommunikationsinfrastruktur dar. Daher wurden für gemeinsam genutzte Signale wie auch für dedizierte Signale eines Busses homogene Realisierungsoptionen vorgestellt und hinsichtlich des Ressourcenbedarfs und der entstehenden Latenzzeit bewertet. Für die gemeinsam genutzten Signale wurden Lösungen für FPGA-Systeme mit und ohne Tristate-Leitungen präsentiert. Die Realisierung gemeinsam genutzter Signale über logische Verknüpfungen (Slice-basierter Ansatz) braucht nur geringfügig mehr Ressourcen als eine Tristate-basierte Lösung, skaliert bezüglich der Signallaufzeit aber schlechter mit der Anzahl der Bus-Anschlusspunkte.

Zusammenfassend kann jedoch festgehalten werden, dass die Kommunikationsinfrastruktur eines dynamisch rekonfigurierbaren Systems mit freier Platzierung deutlich komplexer ist als die vergleichbarer statischer Systeme. Einige beispielhaft implementierte Systeme mit aus 16 Kacheln bestehenden dynamischen Bereichen benötigen bis zu 22,3 % der Ressourcen eines Virtex-II FPGAs allein für Busmakros. Zudem ist der Entwurf wesentlich aufwändiger, da abstrakte Beschreibungssprachen wie VHDL nicht für eine automatische Synthese genutzt werden können. Stattdessen müssen alle Signale manuell geroutet werden, um die geforderte Homogenität zu erreichen.

5 Entwicklungsumgebung für dynamisch rekonfigurierbare Systeme

Die Umsetzung der entwickelten Konzepte in funktionierende technische Systeme und die Bewertung der dynamischen Rekonfigurierung anhand realer Systeme ist ein Kernaspekt dieser Arbeit. Im Folgenden wird die Entwicklungsumgebung vorgestellt, mit der die vorgestellten Kommunikationsinfrastrukturen, Platzierungs- und Konfigurierungsverfahren entwickelt, ihre Funktion verifiziert und Beispielanwendungen realisiert wurden. Die Entwicklungsumgebung basiert auf dem in der Fachgruppe Schaltungstechnik entwickelten Rapid-Prototyping-System RAPTOR2000. Das RAPTOR2000-System wurde ursprünglich für die Entwicklung statischer FPGA-Systeme geschaffen und im Rahmen dieser Arbeit für dynamische Rekonfigurierung erweitert. Auf Basis dieser erweiterten RAPTOR2000-Umgebung wurde ein Evaluierungssystem erstellt, mit dem die in Kapitel 5 vorgestellten Untersuchungen vorgenommen wurden. Es stellt exemplarisch ein dynamisch rekonfigurierbares System dar, an dem die Komplexität und die Zusammenhänge der Konfigurierungs- und Kommunikationsaspekte deutlich werden. Die Realisierung einer konkreten Anwendung auf Basis des Evaluierungssystems wird im Anschluss vorgestellt. Schließlich wird in diesem Kapitel ein Entwurfsablauf für dynamisch rekonfigurierbare Systeme nebst den dazugehörigen Entwurfswerkzeugen präsentiert, ohne den die Realisierung dynamisch rekonfigurierbarer Systeme mit freier Platzierung sehr aufwändig und eine anwendungsbezogene Nutzung kaum möglich wäre.

Zunächst wird nun jedoch ein Schichtenmodell vorgestellt, das die in Kapitel 1 vorgestellten technischen Verfahren für dynamische Rekonfigurierung strukturiert. Es soll als Leitfaden beim Entwurf dienen und wird als solcher auch bei der Entwicklung des anschließend vorgestellten Systems verwendet.

5.1 Das Schichtenmodell PALMERA

In den vorherigen Abschnitten wurden verschiedene Möglichkeiten der Nutzung rekonfigurierbarer Ressourcen vorgestellt und diskutiert. Um die Nutzung in einem technischen System wirklich zu ermöglichen, muss eine Systemkomponente geschaffen werden, die alle mit der dynamischen Rekonfigurierung verbundenen Aufgaben übernimmt. Um für diese Komponente eine hohe Wiederverwendbarkeit zu erreichen und die Fehleranfälligkeit im Entwurf zu verringern, wurde im Rahmen dieser Arbeit das Schichtenmodell PALMERA (Paderborn Layer Model for Embedded Reconfigurable Architectures) entwickelt, das eine schrittweise Abstraktion von der Konfigurierungsschnittstelle des FPGAs zur Anwendung vorschreibt [KP06]. Abbildung 5-1 zeigt das Modell mit seinen sechs Schichten, die zwischen einer Anwendung und der rekonfigurierbaren Hardware liegen. Daten und Informationen werden in diesem Modell immer nur zwischen benachbarten Schichten übergeben. Wie in Schichtenmodellen üblich werden auch hier der jeweils nächst höheren Schicht Dienste angeboten. Durch die Spezifizierung dieser Dienste und der Schnittstellen zwischen den Schichten, können die

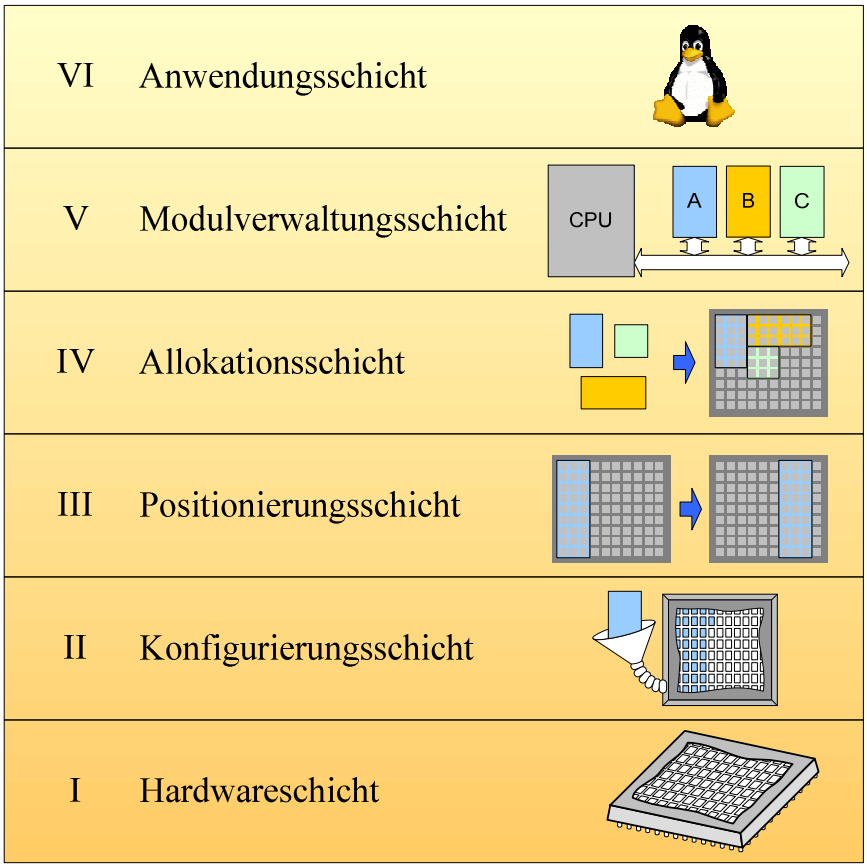

Abbildung 5-1: Das PALMERA-Schichtenmodell für dynamische Rekonfigurierung

einzelnen Schichten unabhängig voneinander entworfen und realisiert werden. Ebenfalls möglich ist es, verschiedene Implementierungen einer Schicht auszutauschen.

5.1.1 Hardwareschicht

Die Hardwareschicht repräsentiert die zugrunde liegende, rekonfigurierbare Hardware. Für die Realisierung einer Laufzeitumgebung sind hier ausschließlich die angebotenen Konfigurierungsschnittstellen relevant. Die Umsetzung der Hardwareschicht findet implizit durch die Auswahl eines FPGAs als Plattform für die Systemintegration statt.

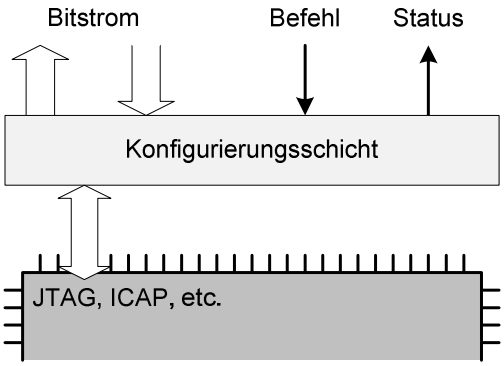

Abbildung 5-2: Hardware- und Konfigurierungsschicht

5.1.2 Konfigurierungsschicht

Als unterste vom Entwickler zu implementierende Schicht stellt die Konfigurierungsschicht die direkte Anbindung an eine Konfigurierungsschnittstelle der verwendeten Hardware sicher, z.B. SelectMap, ICAP, JTAG oder Master/Slave Serial bei den hier verwendeten Xilinx FPGAs (siehe Abbildung 5-2). Die Konfigurierungsschicht kann partielle und komplette Bitströme auf das FPGA laden und beliebige Teile der Konfiguration wieder zurück lesen. Der über ihr liegenden Schicht stellt die Konfigurierungsschicht somit die Dienste *Konfigurieren* und *Auslesen* zur Verfügung. Der Fortschritt eines Befehls und eventuelle Fehlermeldungen werden als Statusinformationen weitergegeben. Die Konfigurierungsschicht abstrahiert von der zugrunde liegenden Konfigurierungsschnittstelle.

5.1.3 Positionierungsschicht

Über der Konfigurierungsschicht ist die Positionierungsschicht angeordnet (siehe Abbildung 5-3). Sie sorgt dafür, dass beim Laden eines Moduls ein der gewünschten Position entsprechender Bitstrom an die Konfigurierungsschicht geleitet wird. Bei einem Ansatz mit festen Modulplätzen kann das lediglich die Auswahl des für den gewünschten Modulplatz implementierten Bitstroms bedeuten. Existiert eine große Zahl von Modultypen und gültigen Modulpositionen, was in der Regel bei freien Platzierungsverfahren der Fall ist, findet in der Positionierungsschicht die Bitstrommanipulation statt. In diesem Fall wird für jeden Modultyp nur ein Konfigurationsdatenstrom erstellt, der zur Laufzeit an beliebige Positionen angepasst werden kann. Die Positionierungsschicht bietet der über ihr liegenden Schicht den Dienst *Positionieren*, der einen Konfigurationsdatenstrom an eine gegebene Position verschiebt. Der manipulierte Datenstrom wird anschließend der Konfigurationsschicht zur Konfigurierung der Hardware übergeben. Für den Fall, dass ein Modul neu geladen wird, wird der Positionierungsschicht der Modultyp übergeben, anhand dessen der entsprechende Bitstrom im Speicher eindeutig bestimmt werden kann. Gleiches gilt für das Löschen eines Moduls, bei dem ein entsprechend großer, leerer Bitstrom an die angegebene Position des FPGA geladen wird. Bei einem Umplatzieren eines Moduls, wie es zum Beispiel beim Defragmentieren notwendig ist, werden der Modultyp sowie alte und neue Position des betrachteten Moduls übergeben. Sollen auch die internen Statusinformationen beibehalten werden, führt die Positionierungsschicht die dazu notwendigen Zwischenschritte ohne Mitwirken der über ihr liegenden Schichten aus. Die Positionierungsschicht ist die oberste Schicht, die Zugriff auf die Bitströme hat. Die Schnittstelle zur Allokationsschicht abstrahiert somit vollständig von den physikalischen Repräsentationen der dynamischen Komponenten. Wie schon bei der Konfigurie-

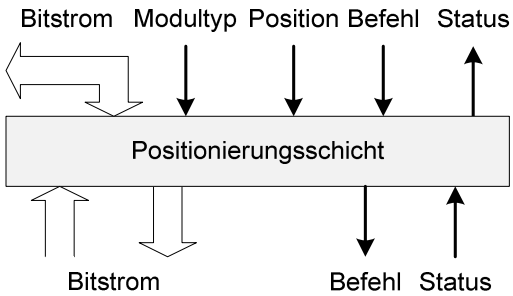

Abbildung 5-3: Die Positionierungsschicht

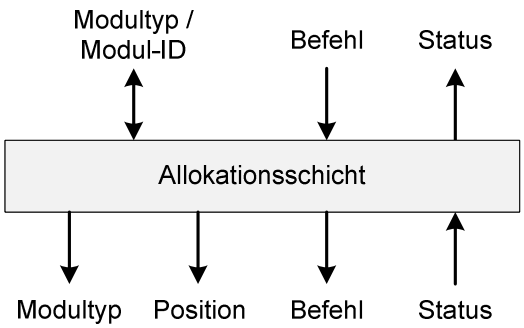

Abbildung 5-4: Die Allokationsschicht

rungsschicht gibt auch die Positionierungsschicht Statusinformationen über den Fortgang bearbeiteter Befehle an die über ihr liegende Schicht weiter.

5.1.4 Allokationsschicht

Auf die Positionierungsschicht folgt die Allokationsschicht. Zentrale Aufgabe dieser Schicht ist die Verwaltung der zur Verfügung stehenden Ressourcen, d.h. hier wird gespeichert, welche Teile der rekonfigurierbaren Hardware belegt sind und welche für das Laden weiterer Module genutzt werden können. Zusätzlich existiert hier eine Liste der geladenen Module, ihrer Position auf der Hardware und gegebenenfalls erforderlicher Statusinformationen. Allen Modulen wird beim Laden durch die Allokationsschicht eine eindeutige Identifikationsnummer (ID) zugewiesen, anhand derer auch Module gleichen Typs unterschieden werden können. Soll eine neue Komponente geladen werden, sucht die Allokationsschicht mit Hilfe eines Platzierungsalgorithmus (Platzierungsstrategie) nach geeigneten, freien Flächen für eines der in Frage kommenden Module. Ist diese Suche erfolgreich, werden der Modultyp des zu ladenden Moduls und die gewünschte Position an die Positionierungsschicht weitergeleitet. Nach einer erfolgreichen Konfigurierung wird die ID des neuen Moduls an die Modulverwaltungsschicht zurückgegeben, die dadurch bei zukünftigen Ereignissen jedes Modul eindeutig bezeichnen kann. Für den Fall, dass keine geeignete Fläche gefunden werden kann, müssen weitergehende Strategien implementiert werden. Dies kann im einfachsten Fall die schlichte Ablehnung der Platzierungsanfrage sein. Aber auch eine Defragmentierung der rekonfigurierbaren Fläche, also ein erneutes Arrangieren der vorhandenen Module, kann zur Problembehandlung implementiert werden. Alle Mechanismen zum Platzieren eines Moduls werden durch die Allokationsschicht gekapselt, d.h. der über ihr liegenden Modulverwaltungsschicht wird der Dienst angeboten, ein Modul auf der rekonfigurierbaren Hardware unterzubringen. Welche Platzierungsstrategie dabei verfolgt wird und ob defragmentiert wird bleibt verborgen. Die Allokationsschicht bietet somit nach oben eine Schnittstelle, die völlig von den zugrunde liegenden Ressourcen und dem Platzierungsverfahren abstrahiert.

Nicht mehr benötigte Module können durch die Allokationsschicht entweder gelöscht oder nur deaktiviert werden. Das Löschen kann wiederum aktiv und passiv durchgeführt werden. Beim passiven Löschen eines Moduls wird sein Eintrag von der Modulliste entfernt, d.h. das Modul ist bei allen zukünftigen Entscheidungen aus Sicht der Allokationsschicht nicht mehr existent. Die zuvor belegten Ressourcen werden wieder freigegeben und gelten als nicht belegt. Beim aktiven Löschen werden zusätzlich auch die Konfigurationsdaten vom FPGA gelöscht, indem ein leerer Bitstrom an die entsprechende Position geladen wird. Bei einer Deak-

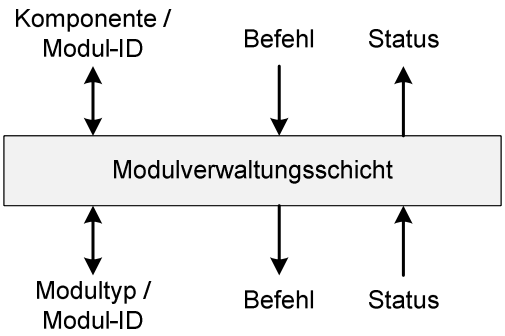

Abbildung 5-5: Die Modulverwaltungsschicht

tivierung eines Moduls bleibt das Modul in der Modulliste. Die entsprechenden FPGA-Ressourcen werden erst dann als nicht belegt betrachtet, wenn unter Berücksichtigung der inaktiven Module nicht genügend freie Ressourcen für ein neu zu platzierendes Modul gefunden wurden. Findet sich dann eine geeignete Position, werden zunächst die inaktiven Module gelöscht, die die Neuplatzierung behindern, und dann das neue Modul geladen. Falls zu einer Modulanfrage ein inaktives Modul gleichen Typs existiert, kann durch einfache Reaktivierung des Moduls die Anfrage bearbeitet werden. Das Aufsuchen geeigneter Flächen und vor allem das zeitaufwändige Konfigurieren des FPGA können dann entfallen. In Anlehnung an entsprechende Techniken bei der Speicherverwaltung wird hierfür auch der Begriff *Caching* verwendet.

5.1.5 Modulverwaltungsschicht

Die Modulverwaltungsschicht verwaltet alle auf der rekonfigurierbaren Hardware befindlichen Module (siehe Abbildung 5-5). Die wesentliche Aufgabe besteht darin, jedem Modul einen Adressraum zuzuordnen und der über ihr liegenden Anwendungsschicht den Zugriff auf die gewünschte Komponente zu ermöglichen. Hierfür führt die Modulverwaltungsschicht eine Modulliste mit allen derzeit geladenen Modulen und den dazugehörigen Adressräumen. Um eine eindeutige Bezeichnung der Module zu ermöglichen, werden hier die von der Allokationsschicht gewählten Modul-IDs ebenfalls vermerkt. Falls eine Anwendung eine Hardware-Komponente benötigt, wird zunächst geprüft, ob für die gewünschte Komponente ausreichend Adressraum verfügbar ist. Ist dies der Fall, wird eine Modulanfrage an die Platzierungsschicht gegeben. Sobald ein entsprechendes Modul geladen oder reaktiviert wurde, wird es von der Modulverwaltungsschicht für die Kommunikation mit anderen Systemkomponenten initialisiert. Wenn ein Modul nicht mehr benötigt wird, kann es entweder von der rekonfigurierbaren Hardware gelöscht oder als inaktiv gekennzeichnet werden. Inaktive Module können bei einer erneuten Anforderung reaktiviert oder bei Ressourcenmangel gelöscht werden. Auch hierbei handelt es sich um eine Caching-Methode, im Gegensatz zur Allokationsschicht allerdings auf einer rein logischen Ebene.

Die Modulverwaltungsschicht bietet der über ihr liegenden Anwendungsschicht die Dienste *Laden* sowie *Löschen* dynamischer Systemkomponenten. Der Anwendungsschicht wird beim erfolgreichen Laden eines Moduls lediglich der entsprechende Adressraum mitgeteilt. Die Modulverwaltungsschicht abstrahiert somit vollständig von der Rekonfigurierung der Hardware.

Es ist vorstellbar, dass die Modulverwaltungsschicht nicht Systemkomponenten, sondern Aufgaben (Tasks) lädt. Zu jeder Task können dann mehrere Systemkomponenten mit unterschiedlichen Eigenschaften (zum Beispiel bezüglich der Ausführungszeit oder den benötigten Ressourcen) erstellt werden. Dies ermöglicht die Verwendung von *Quality-of-Service*-Verfahren. Wird dann von einer Anwendung beispielsweise die abstrakte Task *Fließkomma-Einheit* mit bestimmten Leistungseigenschaften angefordert, kann die Modulverwaltungsschicht diejenige Komponente auswählen, die die entsprechende Performanz bietet und dabei den geringsten Ressourcenbedarf aufweist.

5.1.6 Anwendungsschicht

Die Anwendungsschicht repräsentiert alle Anwendungen, die die Möglichkeit des dynamischen Hinzuladens von Systemkomponenten in Anspruch nehmen. Für den Zugriff auf die Module – also die Schnittstelle zwischen Anwendungsschicht und Modulverwaltungsschicht – können verschiedene Verfahren implementiert werden (Treiber, Einbindung von Bibliotheken, usw.). Insbesondere ist denkbar, dass mehrere Anwendungen gleichzeitig dynamische Komponenten benutzen. Es ist nicht Inhalt dieser Arbeit, entsprechende Verfahren zu entwickeln. Eine Beispielimplementierung ist in Kapitel 5 beschrieben.

5.1.7 Erweiterung bei Echtzeitanforderungen

Das bislang beschriebene Schichtenmodell dient der Kapselung verschiedener Aufgaben in unabhängigen Komponenten. Je nach Implementierung der jeweiligen Schichten kann jedoch keine Aussage darüber gemacht werden, wie viel Zeit eine Schicht von einer Anfrage bis zur Erledigung der Aufgabe benötigt. In Systemen mit Echtzeitanforderungen kann eine solche Implementierung daher nicht verwendet werden. Um dem zu begegnen, können zwei Maßnahmen getroffen werden. Zum einen können alle Schichten so implementiert werden, dass feste obere Grenzen für ihre Ausführungszeiten angegeben werden können. Dann kann schon beim Entwurf unter Kenntnis eines Ablaufplans die Echtzeitfähigkeit eines Systems geprüft werden. Darüber hinaus können die Schnittstellen der Schichten dahingehend erweitert werden, dass zwischen einer Anfrage zum Laden einer Systemkomponente und dem tatsächlichen Auftrag unterschieden wird. Bei einer Anfrage gibt dann jede Schicht die von ihr benötigte Zeit zur Durchführung des Auftrags zurück an die anfragende Schicht. Dadurch kann eine Anwendung zunächst in Erfahrung bringen, wie lange das Laden einer Komponente dauern würde. Dies macht insbesondere dann Sinn, wenn dynamische Systemkomponenten als Beschleuniger für Berechnungen in einem Prozessorsystem verwendet werden. Unter Umständen ist dann eine Berechnung in Software schneller als durch eine dynamische Systemkomponente. In Verbindung mit den oben vorgestellten Quality-of-Service Verfahren kann festgestellt werden, welche von mehreren alternativen Systemkomponenten am schnellsten geladen werden kann oder bereits geladen wurde.

5.1.8 Ablauf einer Rekonfigurierung

Wird das PALMERA-Schichtenmodell für die Implementierung eines Konfigurierungsmanagers verwendet, ergeben sich während einer Konfigurierung typische Kommunikationsmuster innerhalb des Konfigurierungsmanagers. In Abbildung 5-6 sind einige Konfigurierungsabläufe beispielhaft als Diagramm dargestellt. Die Horizontale wird hier als Zeitachse

verstanden, in der Vertikalen sind die sechs Schichten von PALMERA, von der Anwendungsschicht (APL) bis zur Hardwareschicht (HL), aufgetragen. Durch Pfeile ist hier die Kommunikation zwischen benachbarten Schichten, also das Anfragen und Erweisen von Diensten, symbolisiert.

Jeder Konfigurationsprozess startet mit der Anfrage einer Anwendung an die Modulverwaltungsschicht (MML), eine benötigte Komponente zur Verfügung zu stellen. Im einfachsten Fall ist die gewünschte Komponente bereits geladen und derzeit inaktiv. Dann kann sie schlicht reaktiviert und der anfragenden Anwendung direkt zur Verfügung gestellt werden (1). Ist das nicht der Fall, macht die Modulverwaltungsschicht eine entsprechende Platzierungsanfrage an die Allokationsschicht (AL). Diese prüft nun, welche Module für die gewünschte Komponente vorhanden sind und wählt für eines dieser Module eine freie, gültige Modulposition. Falls alle in Frage kommenden Ressourcen belegt sind und auch nicht durch eine Defragmentierung Abhilfe geschaffen werden kann, wird der Anwendungsschicht über die Modulverwaltungsschicht eine negative Antwort auf ihre Anfrage gesendet (2), und der Konfigurierungsvorgang wäre beendet. Im Folgenden wird jedoch angenommen, dass ein gewünschtes Modul nach einer bereits geladenen Modulinstanz platziert werden kann.

Zunächst stellt die Allokationsschicht eine Anfrage zur Modulrelozierung an die Positionierungsschicht (PL). Diese beginnt dann (3) eigenständig mithilfe der Konfigurierungsschicht (CL) das Auslesen des Konfigurationsspeichers des FPGAs (Hardwareschicht, HL). Nach Beendigung des Lesevorgangs (4) wird die ausgelesene Modulinstanz durch Bitstrommanipulation an die Zielposition verschoben und durch Konfigurierung erneut auf das FPGA geladen. Hiernach wird die erfolgreiche Verschiebung der Allokationsschicht mitgeteilt (5). Aufgrund der durch das Verschieben erfolgten Defragmentierung kann die Allokationsschicht nun das anfangs ausgesuchte Modul der angefragten Komponente platzieren. Hierfür werden der Positionierungsschicht der Bitstrom (oder die Position des Bitstroms im Bitstrom-Speicher) und die gewünschte Position mitgeteilt. Die Positionierungsschicht führt die erforderlichen Bitstrommanipulationen durch und lädt dann über die Konfigurierungsschicht das Modul auf das FPGA. Falls dabei keine Fehler auftreten, wird die erfolgreiche Konfigurierung schrittweise an die höheren Schichten propagiert, so dass schließlich die Anwendung die gewünschte dynamische Komponente verwenden kann.

Wie bereits erwähnt, handelt es sich bei den hier beschriebenen Abläufen um ausgewählte Beispiele. Je nach Implementierung der einzelnen Schichten können die Abläufe, z.B. für eine

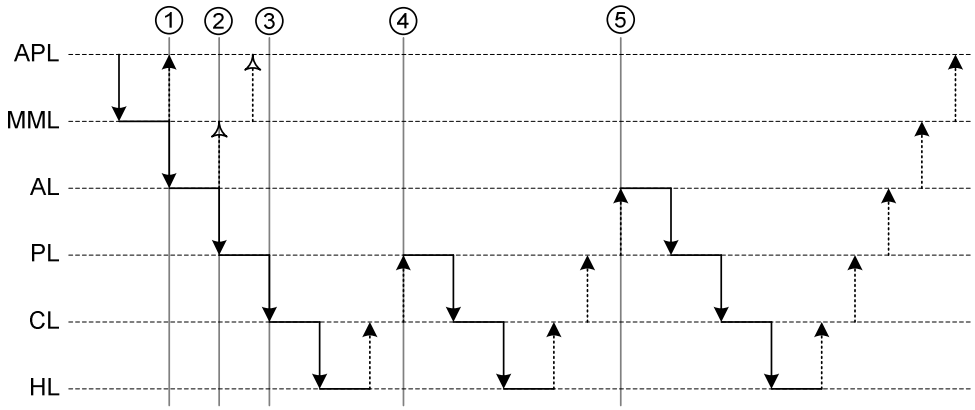

Abbildung 5-6: Mögliche Konfigurierungsabläufe bei Nutzung des PALMERA-Modells

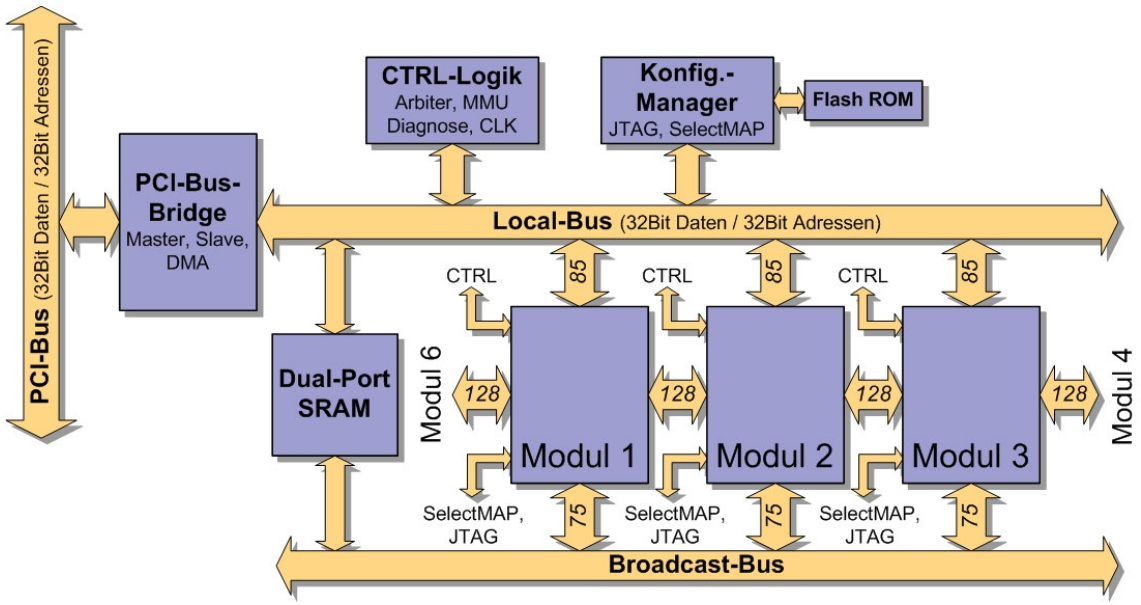

Abbildung 5-7: Das RAPTOR2000-System [Kal04]

Modulplatzierung oder eine Defragmentierung, im Detail anders aussehen. Kernpunkt des PALMERA-Modells ist jedoch, dass die auf einer Schicht durchgeführten Aktionen immer transparent, also nicht sichtbar, für die höher gelegenen Schichten sind. Die strikte Trennung der Aufgaben, die dies bewirkt, ermöglicht somit die weitgehend unabhängige Entwicklung der einzelnen Schichten und eine hohe Flexibilität bei der Erstellung einer Konfigurierungsumgebung.

5.2 Erweiterte RAPTOR2000-Umgebung

Das RAPTOR2000-System [KPR00, Kal04] wurde am Fachgebiet Schaltungstechnik für die prototypische Implementierung mikroelektronischer Schaltungen entwickelt. Es besteht aus einer Basisplatine und verschiedenen Erweiterungsplatinen. Die Basisplatine ist mit einer PCI-Bus-Schnittstelle ausgestattet und kann somit in jedem handelsüblichen PC betrieben werden. Über sechs Steckplätze können die Erweiterungsplatinen als Module in das System eingebunden werden. Abbildung 5-7 zeigt den schematischen Aufbau der Basisplatine. Neben den Erweiterungsmodulen, von denen hier drei dargestellt sind, enthält sie noch Speicher (*Dual-Port SRAM*), eine *PCI-Bus-Bridge*, Kontrolllogik (*CTRL-Logik*) und einen Konfigurierungs-Manager (*Konfig.-Manager*). Als wichtigste Erweiterungsplatinen stehen FPGA-Module zur Verfügung, die mit verschiedenen Xilinx FPGAs bestückt sind (z.B. Virtex-E, Virtex-II, Virtex-2Pro). Daneben existieren verschiedene Schnittstellenmodule (z.B. Ethernet, VGA).

5.2.1 Kommunikation auf RAPTOR2000

Für die Kommunikation innerhalb des RAPTOR2000-Systems stehen drei verschiedene Bus-Infrastrukturen zur Verfügung: LocalBus, Broadcast-Bus und Rechts-Links-Verbindungen. Der LocalBus ist ein Multi-Master-Bus, in dem alle Module als eigenständige Teilnehmer eingebunden sind. Ebenfalls an den LocalBus angeschlossen sind das SRAM, der Konfigurierungsmanager und die PCI-Bus-Bridge. Letztere ermöglicht eine Kommunikation

zwischen den prototypischen Schaltungen auf den Modulen und beliebiger Software auf dem Host-PC. Die komplette Steuerung des LocalBus, inklusive Arbitrierung, Adressraumverwaltung und Taktmanagement, wird von einem CPLD übernommen.

An den Broadcast-Bus sind alle Modulsteckplätze und das SRAM angeschlossen, so dass von einem Modul Nachrichten an mehrere Teilnehmer gleichzeitig geschickt werden können. Eine Multi-Master-Funktionalität ist hier nicht vorgesehen, kann aber durch entsprechende Logik auf FPGA-Modulen implementiert werden.

Die Rechts-Links-Verbindungen befinden sich zwischen benachbarten Modulsteckplätzen und sind somit Punkt-zu-Punkt-Verbindungen. Mit jeweils 128 Leitungen kann eine hohe Bandbreite erzielt werden, was insbesondere bei der Aufteilung großer Schaltungen auf mehrere FPGA-Module von Nutzen ist.

5.2.2 Konfigurierungsinfrastruktur

Da das RAPTOR2000-System FPGAs als Kernkomponenten verwendet, beinhaltet es eine leistungsfähige Konfigurierungsinfrastruktur. Vom Konfigurierungsmanager führen dedizierte Konfigurierungsleitungen an jeden Modulsteckplatz. Auf den FPGA-Erweiterungsmodulen sind diese Leitungen mit den SelectMap-Schnittstellen der FPGAs verbunden. Der Konfigurierungsmanager kann somit alle im System befindlichen FPGAs mit der maximalen Konfigurierungsgeschwindigkeit von 50 MB/s konfigurieren. Für eine Konfigurierung wird dem Konfigurierungs-Manager der zu konfigurierende Steckplatz sowie die LocalBus-Speicheradresse des Bitstroms mitgeteilt. Der Konfigurierungsmanager lädt dann als *aktiver* LocalBus-Teilnehmer den Bitstrom von der gegebenen Adresse und konfiguriert das gewünschte FPGA über die SelectMap Schnittstelle. Der Bitstrom kann sich an einem beliebigen Ort innerhalb des Systems befinden (z.B. im SRAM oder auf einem Erweiterungsmodul). Da der PCI-Bus über die Bridge in den LocalBus-Adressraum eingeblendet wird, können auch Bitströme aus dem Speicher des Host-PCs geladen werden. Diese Möglichkeit ist das übliche Verfahren beim Prototyping digitaler Schaltungen: Ein System wird am PC entwickelt und ein entsprechender Bitstrom wird generiert. Mithilfe einer Software-Bibliothek, der RAPTOR2000-DLL (*Dynamic Link Library*), wird dann der Konfigurierungs-Manager auf dem RAPTOR2000-System angesprochen und die Konfigurierung eingeleitet. Anschließend kann mithilfe der diversen Schnittstellen der RAPTOR2000-Erweiterungsmodule oder mit Diagnose-Software das entwickelte Design getestet werden. Für eine genauere Beschreibung der Nutzungsmöglichkeiten des RAPTOR2000-Systems sei hier auf [Kal04, HNI07] verwiesen.

Bei der Evaluierung statischer Systeme ist der Konfigurierungs-Manager lediglich ein notwendiger Bestandteil der Prototyping-Plattform und gehört somit zur Testumgebung. Dies ändert sich bei der Evaluierung dynamisch rekonfigurierbarer Systeme, bei denen der Konfigurierungsmanager Bestandteil des zu testenden Systems ist. Insbesondere fällt sein Aufgabenbereich in die Spezifikation des PALMERA-Modells. Um hier eine Anpassung der gewünschten Systemarchitektur an die Prototyping-Umgebung zu verhindern, wurde stattdessen das RAPTOR2000-System an die Bedürfnisse der dynamischen Rekonfigurierung angepasst. Hierzu wurde die Funktionalität des Konfigurierungs-CPLDs auf der RAPTOR2000-Basisplatine geändert. Ziel der neuen Architektur war, die Übertragung von Konfigurationsdaten zwischen beliebigen Modulsteckplätzen des RAPTOR2000-Systems zu ermöglichen.

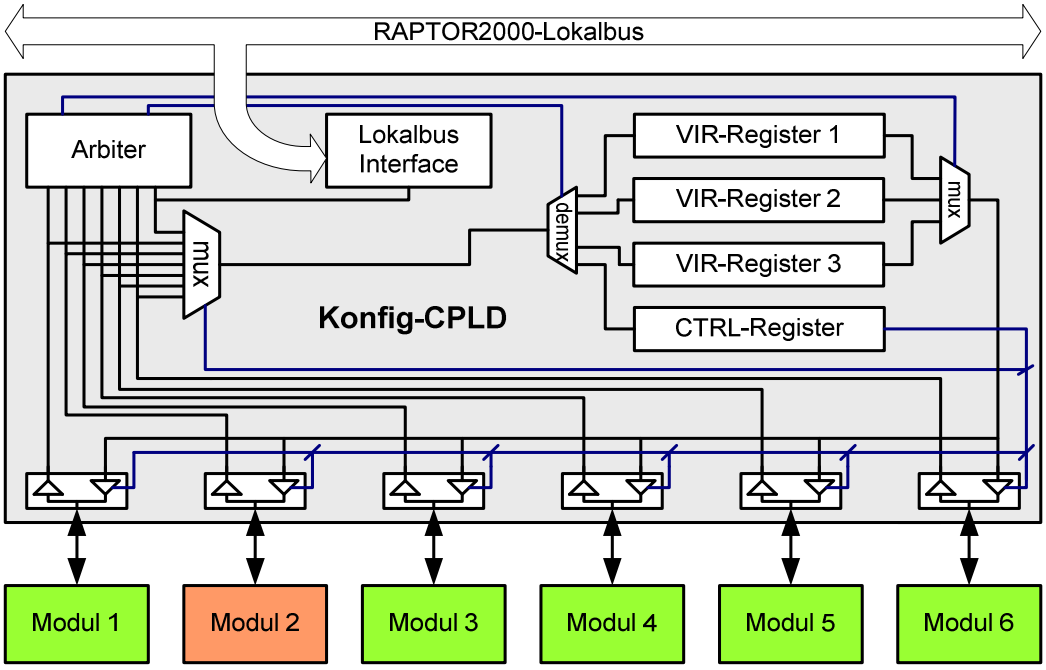

Abbildung 5-8: Angepasstes Konfigurierungs-CPLD mit angeschlossenen RAPTOR2000-Modulen

Die ursprüngliche Konfigurierungsart, nämlich das Herunterladen von Konfigurationsdaten vom Host-PC über PCI- und LocalBus sollte dabei erhalten bleiben.

Abbildung 5-8 zeigt den neuen Aufbau des Konfigurierungs-CPLDs [Hag05S]. Kern der neuen Struktur sind drei Konfigurationsregister, über die alle RAPTOR2000-Module konfiguriert werden können. Diese Register können wiederum von allen Modulen sowie über den LocalBus beschrieben werden. Der Konfigurationsfluss wird über einen zentralen Arbiter und ein weiteres Kontrollregister (CTRL-Register) gesteuert. Als wesentliche Änderung gegenüber der ursprünglichen Funktionalität werden die Konfigurationsleitungen zwischen CPLD und Modulsteckplätzen nun bidirektional genutzt. Für die Konfigurierung über den LocalBus kann das CPLD nicht mehr die Daten vom Host-PC laden. Stattdessen muss die Host-Software die Bitströme nun aktiv schreiben. Um die Kompatibilität älterer Test-Programme mit dem neuen Konfigurierungs-Manager des RAPTOR2000-Systems zu gewährleisten, wurde die RAPTOR2000-DLL entsprechend angepasst, so dass die Änderungen aus Nutzersicht transparent sind. Falls Konfigurationsdaten zwischen zwei Modulen übertragen werden sollen, muss das konfigurierende Modul die Konfigurierungsschnittstellen des Konfigurierungs-CPLDs unterstützen. Die korrekte Übertragung der Daten vom CPLD zum Ziel-Modul stellt das CPLD-Design sicher.

5.3 Das Evaluierungssystem

Abbildung 5-9 zeigt ein Prinzipschaltbild des Evaluierungssystems. Die dynamisch rekonfigurierbaren Systemteile sind hier auf mehrere FPGAs verteilt. Links im Bild ist ein Virtex-2Pro zu sehen, auf den die statischen Komponenten des Systems abgebildet sind. Der eingebettete PowerPC (PPC) bildet den Kern eines On-Chip-Prozessorsystems mit Ein- und Ausgabeschnittstellen, Speicheranbindung und Konfigurationslogik. Auf dem PPC läuft ein Linux-Betriebssystem. Über die RAPTOR2000-Basisplatine sind ein oder mehrere Virtex-II-

FPGAs mit dem Virtex-2Pro verbunden. Sie werden als dynamische Bereiche zum Laden dynamischer Komponenten genutzt. Durch die Aufteilung des Systems in separate FPGAs für dynamische und statische Komponenten können große zusammenhängende dynamische Bereiche ausgewiesen werden, so dass auch freie Platzierungsverfahren mit ausreichender Komplexität realisiert werden können. Darüber hinaus kann durch diesen modularen Aufbau die Menge der für dynamische Komponenten zur Verfügung stehenden Ressourcen durch Hinzufügen oder Entfernen von FPGA-Tochterplatinen verändert werden. Bei einer vollen Ausnutzung aller RAPTOR2000-Modulplätze können somit sehr große rekonfigurierbare Systeme emuliert werden, die in dieser Arbeit allerdings noch nicht betrachtet werden. Die in Abbildung 5-9 dargestellte Ethernet-Schnittstelle ermöglicht das Zusammenschließen mehrerer RAPTOR2000-Systeme zu einem Cluster. Diese Betriebsart wird von dem hier vorgestellten System zwar unterstützt, jedoch ebenfalls in dieser Arbeit nicht behandelt.

5.3.1 Statische Systemkomponenten

Der Großteil der statischen Systemkomponenten wurde auf einer separaten RAPTOR2000-Tochterplatine (DB-V2Pro, [Gri03]) mit einem Virtex-2Pro FPGA realisiert. Abbildung 5-10 zeigt ein Blockschaltbild der statischen Systemkomponenten und ihrer Verschaltung über die Systembusse. Es handelt sich dabei um ein On-Chip-Prozessorsystem mit einem eingebetteten PowerPC Prozessor (PPC 405) als Kernelement. Als Prozessorbus wird ein 64 Bit breiter CoreConnect PLB verwendet, der mit 100 MHz betrieben wird. Hieran sind der Konfigurationsmanager (*Virtex Configuration Manager – VCM*), zwei Speichercontroller für externen und internen Speicher (*SDRAM Controller* und *BRAM Controller*), und je eine Bus-Bridge zum Peripherie-Bus (*PLB2OPB Bridge*) sowie dem LocalBus der RAPTOR2000 Basisplatine (*Raptor Local Master/Slave*) angeschlossen. Zu Letzterem bestehen sowohl eine Master-Anbindung wie auch eine Slave-Verbindung. Dadurch kann vom PLB aus auf das gesamte RAPTOR2000-System und sogar auf den Host-PC zugegriffen werden. Ebenso kann über den LocalBus auf das lokale PPC-System zugegriffen werden.

Der interne Speicher wird mittels der eingebetteten Speicherblöcke (BlockRAM) des Virtex-2Pro FPGAs realisiert. Es werden 64 Speicherblöcke mit insgesamt 128 KB Kapazität verwendet. Kleine Programme des PPC, insbesondere die Initialisierung zu Beginn der Laufzeit (*Boot-Loader*), können direkt aus dem BlockRAM ausgeführt werden. Für weiteren Speicherbedarf stehen auf dem DB-V2Pro-Modul 128 MB SDRAM zur Verfügung, auf den über den SDRAM-Controller zugegriffen werden kann. Interner und externer Speicher werden wie der PLB mit 100 MHz betrieben. Da es sich bei den BlockRAMs um statischen Speicher

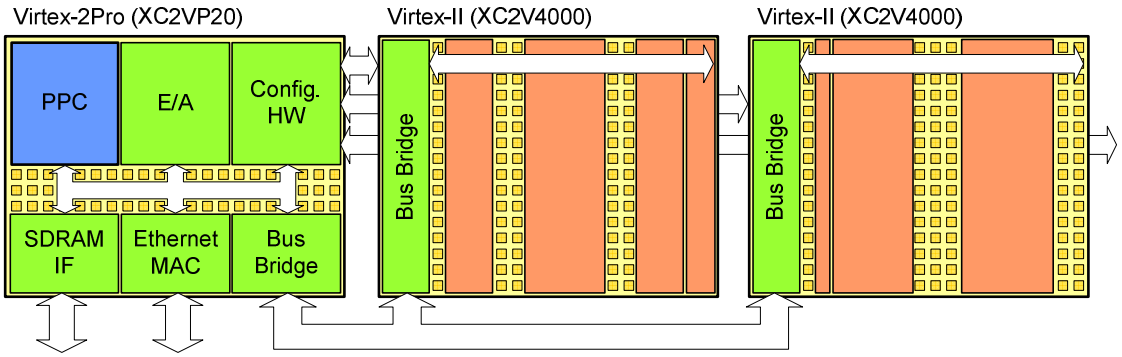

Abbildung 5-9: Mögliche Ausprägung des Evaluierungssystems [RSS07]

(SRAM) handelt, kann auf ihn etwas schneller zugegriffen werden. Da das BlockRAM mit 64 Bit doppelt so breit wie das SDRAM angeschlossen ist, ist der Durchsatz hier entsprechend höher. Für die Inbetriebnahme des Systems ist außerdem wichtig, dass der Inhalt des BlockRAMs durch die Initialkonfigurierung beschrieben werden kann. Das SDRAM muss nach dem StartUp erst noch separat mit den benötigten Daten beschrieben werden.

Der Peripherie-Bus (OPB – *On-Chip Peripheral Bus*) schließt System-Komponenten mit geringeren Bandbreiten-Anforderungen an. Sie werden vor allem für Test- und Debug-Zwecke verwendet. Über einen GPIO-Block (*General Purpose Input Output*) sind LEDs der DB-V2Pro Platine angeschlossen. Über sie können einfache Statussignale an den Entwickler ausgegeben werden. Für die Übertragung komplexerer Informationen ist eine UART-Schnittstelle integriert. Über sie können Ausgaben versandt und beispielsweise über ein Terminal-Programm auf dem Host-PC ausgegeben werden. Ebenso können Informationen von außen an den eingebetteten Prozessor geschickt werden. Als letzte Peripherie-Komponente ist ein *Interrupt Controller* an den OPB angeschlossen, der die Verwendung von Interrupts im System ermöglicht. Er wird hier im Wesentlichen von der UART-Schnittstelle verwendet, die bei eingehenden Daten oder nach einem erfolgreichen Versenden von Daten ein Interrupt auslöst.

Das gesamte System wurde mit dem Xilinx *Embedded Development Kit* (EDK) erstellt. Bis auf das VCM und die Brücken zum RAPTOR2000 LocalBus sind alle Systemkomponenten den von Xilinx mitgelieferten IP-Core Bibliotheken entnommen worden. Für eine genauere Beschreibung dieser Komponenten sei an die Dokumentation des EDK [Xil08] verwiesen. Um die Wiederverwertbarkeit des VCM und der Brücken zwischen LocalBus und PLB zu erhöhen, und um den Entwurf neuer Systeme zu erleichtern, wurden auch diese Komponenten als EDK-kompatible IP-Cores erstellt und in die EDK-Bibliotheken integriert. Auf den Aufbau dieser Komponenten wird nun detailliert eingegangen.

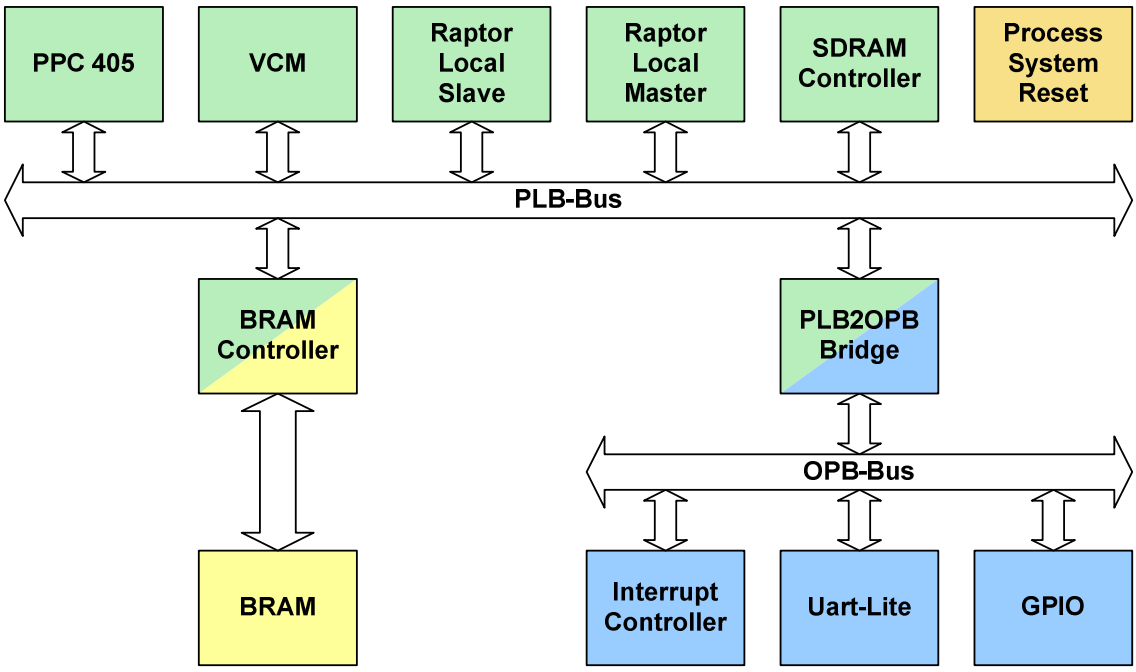

Abbildung 5-10: Statische Systemkomponenten auf dem Virtex-2Pro

5.3.1.1 Virtex Configuration Manager

Der Virtex Configuration Manager stellt nach dem PALMERA-Schichtenmodell die Konfigurierungsschicht dar. Alle höheren Schichten sind in Software auf dem PowerPC realisiert (siehe 5.3.3). Um die Evaluierung möglichst vieler Systemarchitekturen auf unterschiedlichen RAPTOR2000-Konfigurationen zu ermöglichen, wurde das VCM mit einer umfangreichen Funktionalität ausgestattet, u.a. unterstützt es das Auslesen des Konfigurationsspeichers.

Die Kommunikation mit der Positionierungsschicht findet über den PLB statt. Hierfür verfügt das VCM über Kontroll- und Statusregister, die vom PPC gelesen und beschrieben werden können. Zusätzlich ist das VCM als Master an den PLB angeschlossen. Im Falle einer Konfigurierung kann es DMA[19]-Zugriffe auf den SDRAM-Controller initiieren und den erforderlichen Bitstrom eigenständig aus dem Speicher laden. Das VCM kann sowohl zur Selbstrekonfigurierung als auch zur Konfigurierung eines anderen FPGA innerhalb der RAPTOR2000-Umgebung genutzt werden. Hierzu ist es mit der ICAP-Schnittstelle und der RAPTOR2000-Konfigurierungsinfrastruktur verbunden. Die Konfigurierungsschnittstellen sind mit 50 MHz getaktet und 8 Bit breit angeschlossen. Für Virtex-II und Virtex-2Pro FPGAs ist dies die höchstmögliche Konfigurierungsbandbreite, die ohne zusätzliches Handshaking erzielt werden kann. Über alle angeschlossenen Schnittstellen können Bitströme geschrieben und gelesen werden. Mit dem VCM können somit sowohl direkte Konfigurierung als auch Read-Modify-Writeback Verfahren realisiert werden. Ebenso ist die Möglichkeit gegeben, Defragmentierungsstrategien zu implementieren und aktive Module zur Laufzeit zu relozieren.

Die Verarbeitungsgeschwindigkeit des VCM liegt nahe an dem durch die Konfigurierungsschnittstellen vorgegebenen Maximum. Allgemein formuliert ergibt sich die Konfigurierungszeit t_{config} zu:

$$t_{config} = \frac{\left(T_{begin} + B + N_{cyc} \cdot T_{switch} + T_{end}\right)}{f_{config}} \tag{16}$$

Hierbei ist B die Anzahl der Konfigurationsbytes, die übertragen werden. Es spielt keine Rolle, ob gelesen oder geschrieben wird. N_{cyc} ist die Häufigkeit der Wechsel zwischen Lese- und Schreibzugriffen, T_{switch} ist die für einen Wechsel benötigte Zeit. T_{begin} und T_{end} stellen dabei die Zeit in Takten dar, die vom VCM vor und nach einer Konfigurierung für die Initialisierung bzw. den Abschluss einer Operation gebraucht werden.

Zu Begin eines Konfigurierungsvorgangs werden die Kontrollregister des VCM durch den PPC beschrieben. Danach benötigt ein VCM-interner Zustandsautomat die Zeit $T_{start,}$ um die Register auszuwerten und die gewünschte Aktion zu beginnen. Im Falle einer Konfigurierung eines anderen FPGAs (Fremdrekonfigurierung) benötigt das VCM T_{arbit} Takte, um den Zugriff auf die Konfigurierungsinfrastruktur des RAPTOR2000-Systems zu erhalten. Falls es sich um eine Initialkonfigurierung handelt, muss danach zunächst das Ziel-FPGA vollständig gelöscht werden, wofür die Zeit T_{clear} benötigt wird. Sie kann durch $T_{clear} = T_{prog} + T_{reinit}$ abgeschätzt werden, wobei T_{reinit} die Dauer der eigentlichen Initialisierung und T_{prog} die Übertragungsdauer des entsprechenden Kommandos an die Konfigurierungsschnittstelle bezeichnet.

[19] Direct Memory Access: direkter Speicherzugriff.

Hiernach kann die eigentliche Konfigurierung beginnen. Lediglich für das Laden der ersten Daten aus dem SDRAM und das Füllen interner Pipeline-Stufen werden noch T_{init} Takte gebraucht. T_{begin} berechnet sich folglich aus

$$T_{begin} = T_{start} + T_{arbit} + T_{clear} + T_{init}. \tag{17}$$

Nachdem alle Daten übertragen wurden, müssen die Datenpuffer des Paketprozessors (siehe 2.2.2) der angesprochenen Konfigurierungsschnittstelle geleert werden, indem ein sog. *Padword* geschrieben wird (T_{pad}). Hiermit ist die Konfigurierung auf Seiten des konfigurierten FPGAs abgeschlossen. Das VCM setzt anschließend seine Statusregister und löst ggf. ein Interrupt aus. Hierfür werden abschließend noch einmal T_{finish} Takte benötigt, so dass sich T_{end} wie folgt zusammensetzt:

$$T_{end} = T_{pad} + T_{finish} \tag{18}$$

In Tabelle 5-1 sind die Parameter des VCM aufgelistet. Je nachdem, ob es zur partiellen Selbstrekonfigurierung, zur initialen Konfigurierung eines fremden FPGAs oder zur partiellen Konfigurierung eines fremden FPGAs verwendet wird, ändern sich die Parameter, da jeweils unterschiedliche Konfigurierungsabläufe vorliegen. In der untersten Zeile der Tabelle sind beispielhaft die Konfigurierungszeiten eines Virtex-II XC2V4000 FPGAs angegeben. Für die initiale Fremdrekonfigurierung wird eine vollständige Konfigurierung, für die anderen beiden Fälle die Konfigurierung von vier CLB-Spalten angenommen. Ebenfalls vorausgesetzt wird eine direkte Konfigurierung, so dass keine Lesezyklen stattfinden und N_{cyc} somit gleich 0 ist. Die Zeit t_{config} wurde dabei durch die oben gegebene Formel (16) berechnet. t_{oh} gibt die Konfigurationszeit ohne Beachtung der Übertragungszeit für die Konfigurationsdaten an, also die zusätzlich durch das VCM benötigte Bearbeitungszeit.

Im Falle der initialen Fremdrekonfigurierung wird deutlich, dass die durch das VCM benö-

Tabelle 5-1: Performanz-Parameter des VCM

	Initiale Fremdrekonf.	Fremdrekonfigurierung	Selbstrekonfigurierung
T_{start}	11 Takte	11 Takte	11 Takte
T_{arbit}	29 Takte	29 Takte	-
T_{clear}	18 Takte + T_{reinit}	-	-
T_{init}	12 Takte	12 Takte	12 Takte
T_{pad}	16 Takte	16 Takte	16 Takte
T_{finish}	26 Takte	26 Takte	8 Takte
T_{begin}	70 Takte + T_{reinit}	52 Takte	23 Takte
T_{switch}	72 Takte	72 Takte	67 Takte
T_{end}	42 Takte	42 Takte	24 Takte
t_{oh} *)	8626 µs	1,88 µs	0,94 µs
t_{config} *)	47 776 µs	1 469,56 µs	1 468,62 µs

*) Komplettes Design / Vier-CLB-Spalten-Modul auf einem Virtex-II FPGA (XC2V4000)

tigte Bearbeitungszeit unbedeutend ist. Bei einer Gesamtdauer von fast 48 ms können die vom VCM benötigten 112 Takte (2,24 µs) vernachlässigt werden. Auch bei der partiellen Konfigurierung eines relativ kleinen Moduls (vier von 72 CLB-Spalten des Virtex-II) liegen die Bearbeitungszeiten des VCM immer deutlich unter einem Prozent der Gesamt-Konfigurierungsdauer. Dabei wird immer angenommen, dass die Kommunikation des VCM mit dem SDRAM nicht den Engpass bei der Datenübertragung darstellt. Bei den verfügbaren Bandbreiten (SDRAM: max. 400 MB/s, SelectMap: 50 MB/s) ist dies im Normalbetrieb immer gewährleistet. Um sicherzustellen, dass der PLB nicht gleichzeitig von anderen Busteilnehmern verwendet wird, kann das VCM den PLB blockierend benutzen, d.h. während einer Konfigurierung exklusiv verwenden. Für die Initiierung des Buszugriffs wird allerdings noch eine gewisse Zeit benötigt, die üblicherweise deutlich unter einer Mikrosekunde liegt.

Um sowohl direkte Konfigurierungen als auch Read-Modify-Writeback-Konfigurierungen zu ermöglichen, sollte das VCM abwechselnde Lese- und Schreibzugriffe mit der maximalen Übertragungsgeschwindigkeit durchführen können. Eine existierende Lösung von Blodget et al. [BJK03], die inzwischen als HWICAP zum Umfang der EDK-Bibliotheken gehört und die bei Nutzung der ICAP-Schnittstelle einen vergleichbaren Funktionsumfang bietet, wartet mit wesentlich höheren Übertragungszeiten auf. In [SBA05] werden Übertragungszeiten für die Selbstrekonfigurierung eines Virtex-2Pro FPGAs (XC2VP7) mit der HWICAP-Komponente angegeben. Das Laden eines Vier-Spalten-Moduls würde mit dieser Lösung 12083,84 µs dauern. Das VCM wäre in diesem Fall mit 755,66 µs um den Faktor 16 schneller. Der Geschwindigkeitsnachteil des HWICAP hat seine Ursache zum einen in einer wesentlich langsameren Busanbindung, da hier der OPB anstelle des PLB verwendet wird. Darüber hinaus werden grundsätzlich alle Frames zunächst in einem lokalen BlockRAM zwischengespeichert, wodurch weitere Zeit verloren geht. Der Geschwindigkeitsnachteil gegenüber dem VCM verringert sich allerdings bei der Nutzung der Read-Modify-Writeback Konfigurierung, da hier auch das VCM die Konfigurationsdaten zwischenspeichern muss.

5.3.1.2 PLB-LocalBus-Bridge

Der Zugriff auf die rekonfigurierbaren Ressourcen vom Prozessorsystem (PLB) erfolgt ausschließlich über den LocalBus des RAPTOR2000-Systems. Hierfür wurde eine PLB2LocalBus-Bridge entwickelt, deren Eigenschaften hier nur knapp beschrieben werden sollen. Als wesentliche Eigenschaft ermöglicht sie eine Master- und Slave-Anbindung des PLB an den LocalBus, d.h. es kann von beiden Domänen aktiv auf den jeweils anderen Bus zugegriffen werden. Die Bridge besteht daher de facto aus zwei Bridges: PLB2LocalBus und LocalBus2PLB. Die Komponente PLB2LocalBus ist als Slave an den PLB angebunden und agiert am LocalBus als Master. Über sie finden daher die Zugriffe vom PPC-System auf den LocalBus statt. Für die LocalBus2PLB-Komponente gilt entsprechendes für die umgekehrte Zugriffsrichtung. Der innere Aufbau der Bridges ist in Abbildung 5-11 beispielhaft für die PLB2LocalBus-Bridge dargestellt.

Für beide Bridges ist die Anbindung aufgrund der unterschiedlichen unterstützten Taktfrequenzen der Busse asynchron. Der PLB wird im Normalfall mit 100 MHz, der LocalBus mit bis zu 50 MHz betrieben. Die Daten werden zu diesem Zweck über je zwei FIFOs (Leserichtung und Schreibrichtung) übermittelt. Da auch die Datenbreite der Busse unterschiedlich sein kann – der PLB ermöglicht bis zu 64 Datenbits, während der LocalBus immer 32 Bit breit ist

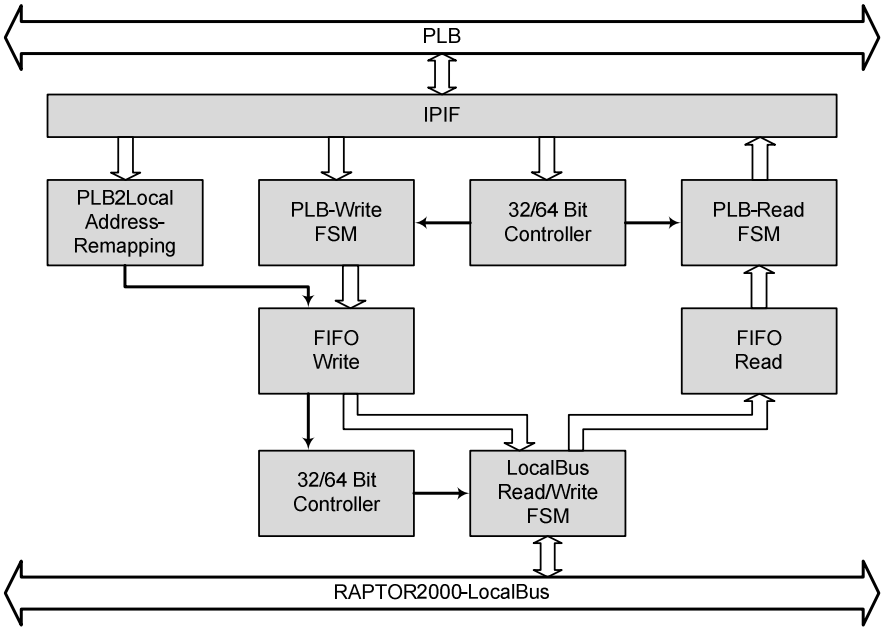

Abbildung 5-11: PLB2LocalBus Bridge mit PLB-Slave- und LocalBus-Master-Anbindung

– sorgen zwei 32/64 Bit Controller für die korrekte Übertragung der Daten. Ein 64 Bit Wort wird dabei in zwei 32 Bit Wörter aufgeteilt. Falls weniger als 64 Bit übertragen werden, wird die Datenmaskierung (byteweise) entsprechend gesetzt. Die Bus-Protokolle werden über endliche Automaten (*Finite State Machines – FSM*) realisiert.

Für die Nutzung des Gesamtsystems ist die Adressverwaltung wichtig. Um vom LocalBus auf den gesamten Adressraum des PLB zugreifen zu können wird *Address-Remapping* verwendet, mit dem beliebige Ausschnitte des PLB-Adressraums in den LocalBus eingeblendet werden können. Gleiches gilt für Zugriffe vom PLB auf den LocalBus.

5.3.2 Rekonfigurierbare Bereiche

Für die dynamischen Bereiche werden in der Prototyping-Umgebung separate FPGAs verwendet. Die Betrachtung verschiedener FPGA-Architekturen wird dadurch erleichtert, da die Konfigurierungsinfrastruktur des Systems bestehen bleibt und somit ohne Änderungen weiter genutzt werden kann. Zudem kann so ein Großteil Fläche eines FPGAs als dynamischer Bereich ausgewiesen werden. Dies ermöglicht die Untersuchung komplexer rekonfigurierbarer Systeme, die für die Betrachtung freier Platzierungsverfahren besonders interessant sind. Die rekonfigurierbaren Ressourcen können beliebig genutzt werden, einzig die Schnittstelle zur statischen Hardware über den LocalBus des RAPTOR2000-Systems ist fest vorgegeben. Dies ermöglicht die Realisierung und den Vergleich verschiedener Platzierungsverfahren und Kommunikationsinfrastrukturen. Um ein Beispiel für ein vollständiges rekonfigurierbares System zu geben und um die Nutzung der Prototyping-Umgebung zu demonstrieren, wird im Folgenden eine konkrete Ausprägung eines dynamischen Bereichs beschrieben [HKK07c]. Sie orientiert sich an den Implementierungen, die in vorangegangenen Kapiteln zum Teil schon ausführlich beschrieben wurden. Grundlage der Implementierung ist ein RAPTOR2000-Modul mit einem Virtex-II XC2V4000 FPGA [Kla04].

Eine Übersicht des Virtex-II Designs ist in Abbildung 5-12 gegeben. Es zeigt die Aufteilung der FPGA-Ressourcen in je einen Bereich für statische und dynamische Komponenten. Für die Module, die in den dynamischen Bereich geladen werden können, ist eine Wishbone-Schnittstelle (WB) [Her02] vorgesehen. Das Wishbone-Protokoll unterstützt zwar Shared-Bus Architekturen, eine direkte Realisierung dieses Protokolls durch ein Busmakro führt jedoch zu hohen Latenzen auf den Leitungen und somit zu niedrigen Übertragungsfrequenzen und Datenraten (vgl. [Hag06D]). Aus diesem Grund wurde hier eine proprietäre, Wishbone-ähnliche, aber wesentlich leistungsstärkere Kommunikationsinfrastruktur implementiert, die als Encapsulated Wishbone (EWB) bezeichnet wird. Der Wishbone-Bus stellt hier somit den virtuellen Bus dar, der die Einbindung bestehender, frei verfügbarer IP-Cores in das System erleichtert. Der EWB ist der reale Bus.

Innerhalb des dynamischen Bereichs wird für den EWB ein Busmakro verwendet, im statischen Bereich wird er mithilfe der PAR-Werkzeuge automatisch verdrahtet. Die Buszugriffe werden durch den Bus Manager gesteuert. Die Kommunikation mit dem Rest des Systems findet über den LocalBus statt, der über zwei Brücken (Master- und Slave-Anbindung) an den EWB angeschlossen ist. Die Anpassung der Kommunikationsinfrastruktur bei oder nach einer Rekonfigurierung (Anpassung des Adressraums, Deaktivierung nicht genutzter Anschlusspunkte) wird ebenfalls vom Bus Manager durchgeführt. Die notwendigen Informationen werden vom statischen Bereich über den LocalBus und EWB in entsprechende Register des Bus Managers geschrieben.

5.3.2.1 Partitionierung der FPGA-Ressourcen

Abbildung 5-13 zeigt die gewählte Segmentierung des FPGAs. Grau dargestellt sind die Ressourcen des verwendeten XC2V4000-FPGAs (vgl. Abschnitt 2.1). Zur Veranschaulichung der Größenordnung wurde ein Ausschnitt des FPGAs vergrößert. Er zeigt vier CLBs (mit je-

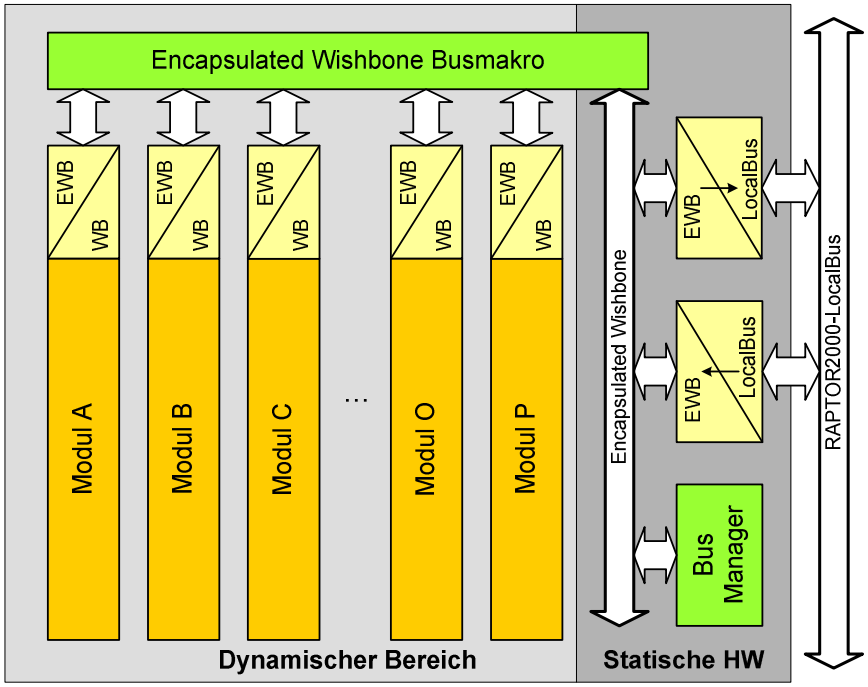

Abbildung 5-12: Schematische Übersicht des Virtex-II Designs

weils vier Slices) und Teile einer Spalte mit BlockRAM und Multiplizierern. Farblich gekennzeichnet ist das platzierte und verdrahtete Initialdesign, also die Realisierung der in Abbildung 5-12 gezeigten Architektur ohne jegliche Module. Am rechten Rand befinden sich die statischen Komponenten: der Bus-Manager und die Brücken zwischen EWB und LocalBus. Ihre Lage ist durch die für den Anschluss des LocalBus verwendeten I/O-Blöcke vorgegeben, die sich ebenfalls am rechten Rand des FPGAs befinden. Die statischen Komponenten und der in sie hineinragende Teil des Busmakros benötigen zusammen mit 1693 Slices und 9 BlockRAMs etwa 7,3 % bzw. 7,5 % der Ressourcen des XC2V4000. Tatsächlich belegen sie jedoch sechs CLB-Spalten (1920 Slices) und eine BlockRAM-Spalte (20 BlockRAMs), so dass 8,3 % bzw. 16,7 % der Ressourcen nicht anderweitig verwendet werden können.

Für die Einteilung des dynamischen Bereichs wurde hier ein eindimensionales Platzierungsverfahren mit einer Kachelbreite von vier CLBs gewählt. Aufgrund des spaltenbasierten Aufbaus der Virtex-II FPGAs werden dadurch sowohl der Entwurf wie auch der Konfigurierungsvorgang selbst nicht unnötig erschwert. Die Kachelung ist hier überdies so gewählt, dass ein möglichst hoher Grad an Homogenität erreicht wird. Sie ist in Abbildung 5-13 am unteren Rand des FPGAs skizziert. Es existieren insgesamt 16 Kacheln mit zwei verschiedenen Kacheltypen (A und B). Beide Typen beinhalten je vier CLB-Spalten, der Typ A enthält zusätzlich eine Spalte mit BlockRAM und Multiplizierern. Durch die kleine Anzahl von Kacheltypen wird die Anzahl von Modulen pro Komponente minimiert und – für den Fall, dass Modulrelozierung verwendet wird – der Speicherbedarf für die partiellen Bitströme ebenfalls gering gehalten.

Eine durch den noch fehlerhaften Xilinx-Entwurfsablauf für dynamische Rekonfigurierung hervorgerufene Besonderheit der hier verwendeten Segmentierung ist eine Pufferzone zwischen statischem und dynamischem Bereich. Insbesondere bei der Implementierung der statischen Komponenten routet das Verdrahtungswerkzeug auch außerhalb der ihm vorgegebenen Grenzen. Es kann daher passieren, dass einige statische Signale durch die äußeren Kacheln des dynamischen Bereichs geroutet werden. Wird zur Laufzeit in diese Kacheln ein Modul geladen, werden die statischen Signale gelöscht und die betroffenen statischen Komponenten können nicht mehr korrekt funktionieren. Die hier zwei Spalten breite Pufferzone ist somit ein Tribut an das Verdrahtungswerkzeug, durch den 640 Slices (2,8 % aller Slices) vergeudet werden. Es hat sich herausgestellt, dass nur äußerst selten mehr als zwei Spalten außerhalb des vorgeschriebenen Bereichs geroutet wird. Falls es, wie bei dem in Abbildung 5-13 gezeigten Beispiel, doch passiert, müssen nach dem PAR von Hand Anpassungen gemacht werden, oder die Pufferzone muss vergrößert werden.

5.3.2.2 Aufbau der Kommunikationsinfrastruktur

Nach den in 4.1 gegebenen Abstraktionsebenen einer Kommunikationsinfrastruktur stellt der virtuelle Wishbone-Bus in dieser Implementierung die oberste Ebene dar. Der reale Bus ist der bereits erwähnte Encapsulated Wishbone Bus [Hag06D], der je 32 Daten- und Adressbits aufweist. Eines seiner wesentlichen Leistungsmerkmale ist eine Registerstufe in allen Eingangs- und Ausgangssignalen der Module, die die maximale Taktrate des Busses erhöht und feste Aussagen zum Timing erst möglich macht (vgl. 4.5.3). Zusätzlich ermöglicht er neben Einzelzugriffen auch Block-basierte Datentransfers, was die Datenrate gegenüber einem herkömmlichen Wishbone-Bus weiter erhöht. In der vorliegenden Implementierung kann

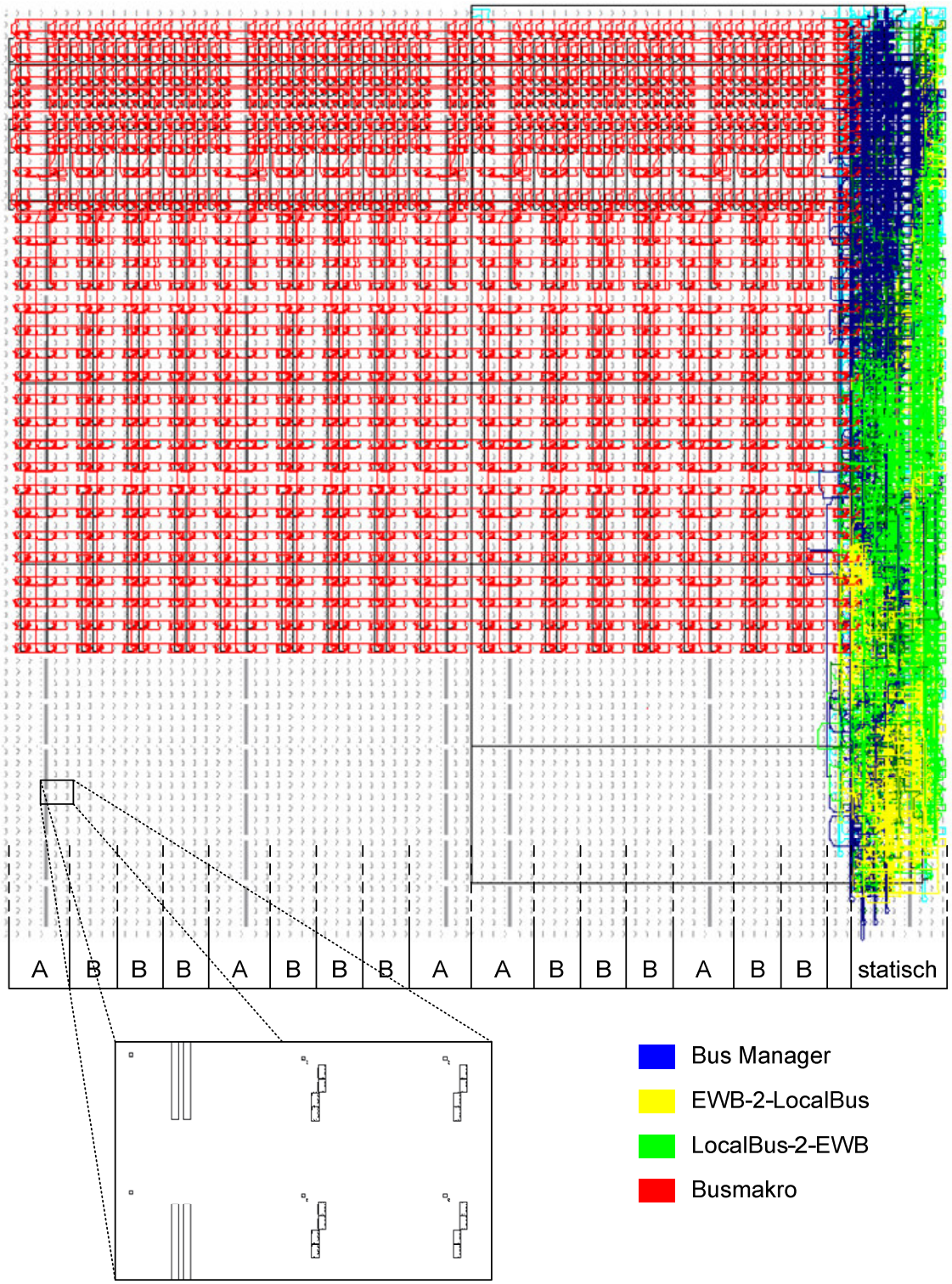

Abbildung 5-13: physikalische Ansicht des Virtex-II Designs im Xilinx FPGA-Editor

der EWB mit 70 MHz betrieben werden und erreicht dank der Block-Zugriffe einen maximalen Durchsatz von bis zu 260 MB/s. Das theoretische Maximum bei 32 Datenbits und 70 MHz liegt bei 280 MB/s. Mit Einzelzugriffen (wie sie beim Wishbone ausschließlich verwendet werden), können maximal 30 MB/s erreicht werden. Das Busmakro ist, wie in Kapitel 4 beschrieben, aus Tristate-basierten Makroprimitiven zusammengesetzt.

In Abbildung 5-13 wird die Komplexität eingebetteter Makros sehr deutlich. Die in die Kacheln eingebetteten Leitungen stellen ein einziges, monolithisches Makro dar. Makros mit einer solchen Komplexität sind ohne zusätzliche Entwurfswerkzeuge – wie das später vorgestellte X-BBG – praktisch nicht realisierbar. Neben den eigentlichen Kommunikationssignalen enthält das Busmakro Kontrollsignale für die Aktivierung/Deaktivierung der Makro-Anschlusspunkte sowie Leitungen für die Übertragung von Statusinformationen der Module.

Für die Übersetzung des Wishbone-Protokolls auf das EWB-Protokoll existiert eine Schnittstellenbeschreibung, die beim Entwurf der Module als Wrapper die Modulbeschreibung umschließt. Bei einer Änderung des virtuellen oder des realen Protokolls muss dann nur diese Schnittstelle angepasst werden, alle anderen Systemkomponenten können weiter genutzt werden.

5.3.2.3 Bus Manager

Der Bus Manager vereint die klassischen Kontrollkomponenten eines Busses – den Arbiter und den Adressdekoder – mit den zusätzlich benötigten Kontrollmechanismen einer Kommunikationsinfrastruktur für dynamisch rekonfigurierbare Systeme. Abbildung 5-14 zeigt den inneren Aufbau des Bus Managers. Er ist verantwortlich für die Steuerung des EWB. Das bedeutet, dass er selbst Teilnehmer des EWB ist und gleichzeitig einen Teil der Steuersignale aktiv setzt. Um auch hier die Anpassung an ein anderes Busprotokoll zu erleichtern, wird intern ein Wishbone-Protokoll verwendet, das durch die oben beschriebene Schnittstelle auf das EWB-Protokoll umgesetzt wird.

Status Register / Port Status

Über die Status Register kann der aktuelle Zustand aller Anschlusspunkte des Busmakros abgefragt werden. Pro Anschlusspunkt stehen je vier Bits zur Verfügung. Aufgrund der hohen

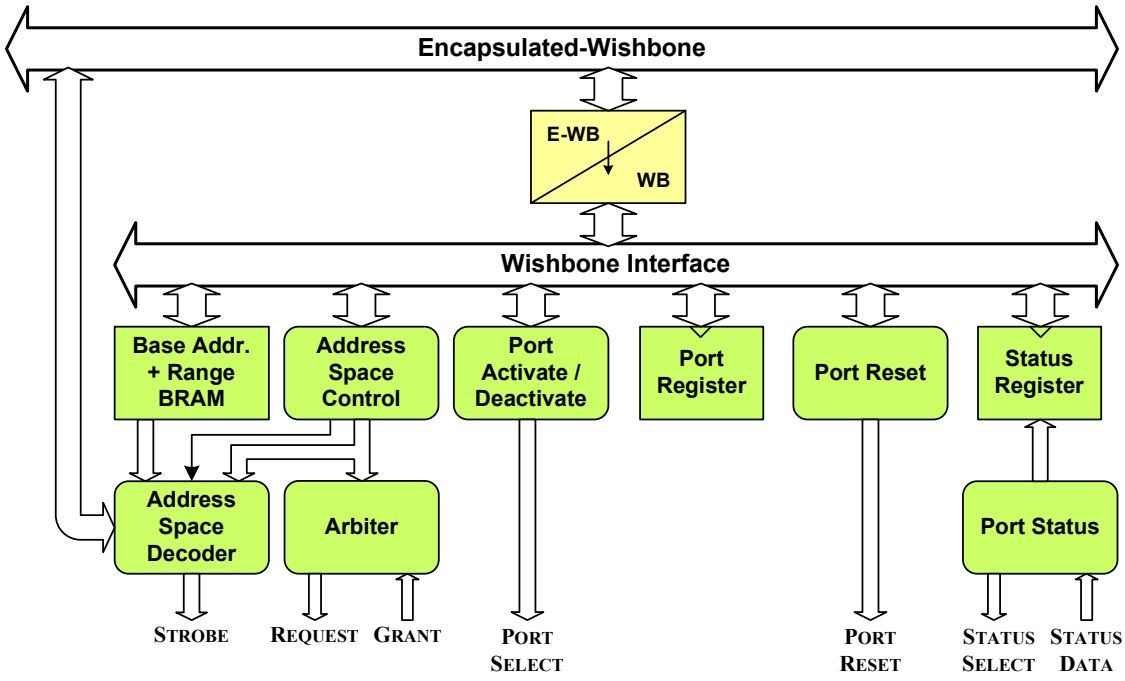

Abbildung 5-14: Der EWB Bus Manager

Kosten für dedizierte Signale in Busmakros werden die Statuswörter über gemeinsam genutzte Signale (STATUS DATA) übertragen. Über Adressleitungen (STATUS SELECT) können die Anschlusspunkte selektiert werden. Ein Zustandsautomat fragt reihum die Statuswörter ab und speichert sie lokal in Registern (*Status Regs*). Diese Register können wiederum über den Wishbone-Bus (und alle daran angeschlossenen Busse: EWB, LocalBus) abgefragt werden. Im hier beschriebenen System werden die niederwertigsten Bits aller Statuswörter zudem mit einer LED-Matrix des RAPTOR2000 DB-VII Moduls ausgegeben und die aktuelle Belegung der rekonfigurierbaren Ressourcen visuell dargestellt.

Port Reset

Alle Anschlusspunkte besitzen ein Reset-Signal, mit dem angeschlossene Module zurückgesetzt werden können. Diese Reset-Signale können durch eine Zustandsmaschine gesetzt werden. Um die Anschlusspunkte dabei unterscheiden zu können, sind sie aufsteigend durchnummeriert. Die Kontrolleinheiten für das Reset (*Port Reset*) und für das Aktivieren/Deaktivieren der Anschlusspunkte (*Port Activate/Deactivate*) teilen sich hierfür ein Register (*Port Register*), in das von außen eine Anschlussnummer geschrieben werden kann, wodurch der entsprechende Anschlusspunkt als ausgewählt gilt. Durch einen Schreibzugriff auf die Wishbone-Adresse der Slot Reset SM wird der Anschlusspunkt dann zurückgesetzt.

Port Activate/Deactivate

Diese bereits angesprochene Komponente kontrolliert die Eingangssignale der Anschlusspunkte und kann ein ungewolltes Treiben von Signalleitungen verhindern (vgl. 4.2.3). Die Bedienung erfolgt analog zu *Port Reset*. Durch einen Schreibzugriff auf eine Activate-Adresse wird der im *Port Register* spezifizierte Anschlusspunkt aktiviert, d.h. die kritischen Signaleingänge des Makros werden freigeschaltet. Durch Beschreiben einer Deactivate-Adresse können Anschlusspunkte gesperrt werden. Aufgrund der Verwendung des *Port Registers* kann mit jedem Zugriff auf *Port Activate/Deactivate* nur ein Anschlusspunkt aktiviert oder deaktiviert werden. Da auch große Module im Normalfall nur einen Anschlusspunkt verwenden, stellt dies keine Einschränkung dar. In jedem Fall müssen pro Aktivierung oder Deaktivierung zwei Buszugriffe getätigt werden.

Innerhalb des Bus Managers gibt es keine Informationen über die tatsächlich genutzten und ungenutzten Anschlusspunkte des Busmakros. Es muss durch die Allokationsschicht sichergestellt werden, dass ungenutzte Anschlusspunkte rechtzeitig deaktiviert werden.

Base Address + Range BRAM

In diesem Speicher wird die Aufteilung des Adressraums festgelegt. Für jeden Anschlusspunkt können die Basisadresse und die Größe des zugewiesenen Adressraums bestimmt werden. Eine Fehlerüberwachung, z.B. sich überschneidende Adressräume zweier Anschlusspunkte, findet nicht statt. Sie muss durch die den Speicher beschreibende, höhere Instanz durch geführt werden. Neue, in das *Base Address + Range BRAM* geschriebene Adressinformationen sind solange vorläufig, bis sie durch die *Address Space Control* bestätigt werden.

Address Space Control

Durch einen Zugriff auf diesen Zustandsautomaten werden die Werte des *Base Address + Range BRAM* in Register (Flipflops) des *Address Decoders* übernommen. Dies ist notwendig, da nur durch die Speicherung in Registern die Adressdekodierung für alle Anschlusspunkte

gleichzeitig durchgeführt werden kann, da bei BlockRAM immer nur auf ein Datum gleichzeitig zugegriffen werden kann. Die zweistufige Speicherstruktur (BlockRAM → Register) erlaubt zudem simultane Adressraumänderungen, da während des Übertragens der Werte aus dem BlockRAM in die Register alle Buszugriffe geblockt werden.

Neben der Übertragung der Adressraum-Daten registriert *Address Space Control* genutzte Anschlusspunkte beim Adressdekoder und beim Arbiter. Während durch *Port Activate/Deactivate* die Eingangssignale physikalisch von der Busstruktur entkoppelt werden, findet hier durch die Registrierung eine Aktivierung auf einer logischen Ebene statt.

Address Space Decoder

Der Adressdekoder selektiert je nach angelegter EWB-Adresse den gewünschten Slave über ein STROBE Signal. Ausgewertet werden nur die Adressräume registrierter Anschlusspunkte.

Arbiter

Der Arbiter steuert die Buszugriffe der Master-Busteilnehmer über REQUEST und GRANT Signale. Nicht registrierte Anschlusspunkte werden auch hier nicht beachtet.

5.3.3 Softwarearchitektur

Für einen großen Teil des PALMERA-Modells bietet sich eine Realisierung in Software an. Insbesondere Verwaltungsaufgaben (Modulverwaltungs- und Allokationsschicht) und die dafür erforderlichen Algorithmen können effizient für einen Prozessor entwickelt und auf ihm ausgeführt werden. Eine proprietäre Softwarebibliothek, die auf die Konfigurierungshardware zugreifen (VCM) und in beliebige Anwendungssoftware eingebunden werden kann, stellt eine Realisierungsoption dar. Für eine vielfältige Nutzung des Systems, die in einem Prototyping System unbedingt gegeben ist, erweist sich eine solche Lösung allerdings schnell als nicht ausreichend. Sobald mehrere Anwendungen gleichzeitig Komponenten laden und nutzen wollen, ist eine Fülle von weiteren Funktionen erforderlich, die traditionell durch Betriebssysteme (Operating Systems – OS) bereitgestellt werden. Es bietet sich also an, ein existierendes Betriebssystem für die Nutzung dynamischer Rekonfigurierung zu erweitern. Die Idee, rekonfigurierbare Ressourcen durch ein OS zu verwalten ist nicht neu [Bre96]. Für das hier entwickelte Evaluierungssystem hat die Nutzung eines Betriebssystems zusätzlich den Vorteil, dass es Schnittstellen zur Verfügung stellt, die zur Fehlerausgabe (Debugging), zum Überwachen (Monitoring) oder für interaktive Eingriffe in das System genutzt werden können.

Die Entwicklung oder Erweiterung eines OS hätte den Rahmen dieser Arbeit sowohl fachlich wie auch zeitlich gesprengt. Durch eine Kooperation mit dem Departement für Elektronik und Informatik der Universität Mailand (Dipartimento di Elettronica e Informazione, Politecnico di Milano) konnte jedoch ein MontaVista-Linux Betriebssystem [Mon08] für die Nutzung dynamisch rekonfigurierbarer Hardware erweitert werden [Ran06M]. Es wird im Folgenden vorgestellt. Entscheidend im Sinne dieser Arbeit ist sind nicht die Implementierungsdetails des Betriebssystems, sondern vielmehr die Umsetzung des PALMERA-Schichtenmodells.

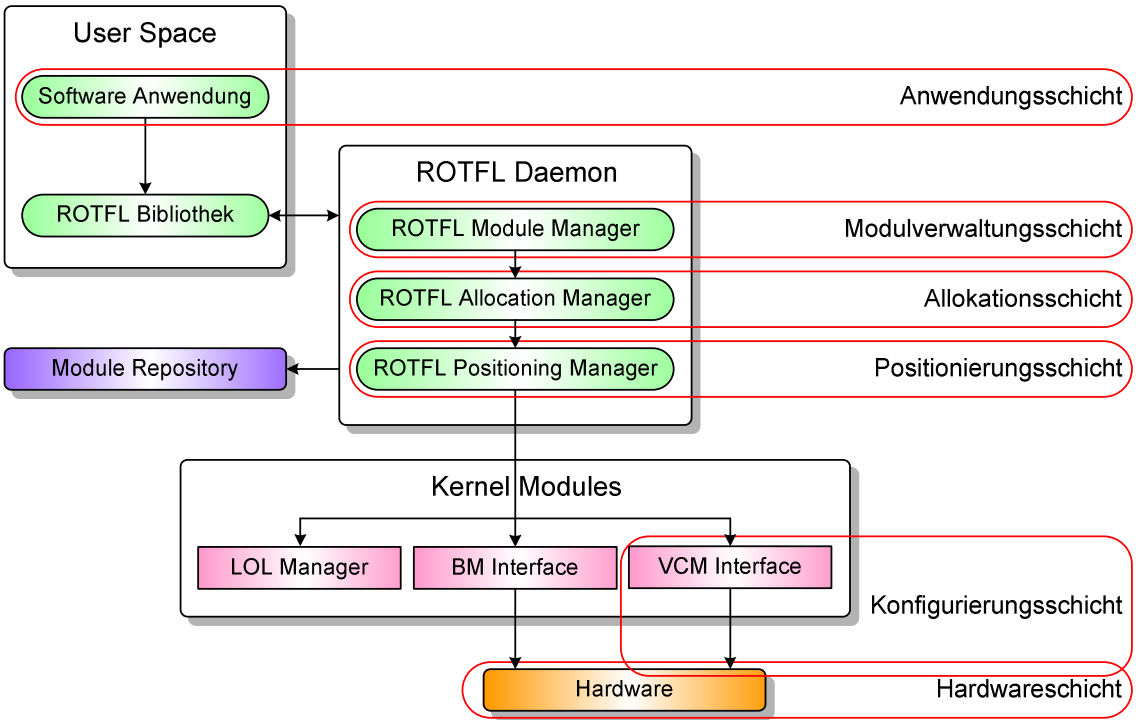

Abbildung 5-15: Erweiterung eines Linux-Derivats nach dem PALMERA-Modell

5.3.3.1 Softwarekomponenten

Abbildung 5-15 zeigt den Aufbau der entwickelten Softwarekomponenten des Betriebssystems und deren Zuordnung zu den Schichten des PALMERA-Modells. Wie dort spezifiziert, findet die Kommunikation zwischen der Anwendung und der Konfigurierungshardware (VCM) auf dem FPGA gerichtet und schrittweise abstrahiert statt. Konkret werden hier Anfragen der Anwendung über einen *Daemon*[20] und *Kernel*[21]*-Module* an die Hardware weitergeleitet.

Die Software-Anwendungen eines Benutzers stellen die Anwendungsschicht dar. Um die Nutzung der dynamischen Rekonfigurierung aus Anwendersicht möglichst einfach zu gestalten, wurde eine Bibliothek erstellt, die alle notwendigen Zugriffe auf den Daemon, und somit auf die unteren PALMERA-Schichten, in Funktionen kapselt. Durch das Einbinden dieser Bibliothek in die Anwendungs-Software können diese Funktionen dann bequem aufgerufen werden. Entsprechend ihrer Aufgabe wird die Bibliothek (und ebenso der Daemon) mit *Reconfiguration Of The FPGA using Linux* – ROTFL bezeichnet. Die Software-Anwendung samt Bibliothek wird im so genannten *User Space* ausgeführt, einem vom *Kernel-Space* separierten Bereich im virtuellen Speicher des Systems.

Der ROTFL Daemon bietet einen Konfigurierungsdienst für andere Programme im System an. In ihm werden alle für die Rekonfigurierung benötigten Verwaltungsaufgaben bearbeitet. Dies beinhaltet, neben der Modul- und Ressourcenverwaltung, zumindest auch einen Teil der

[20] Daemons sind im Allgemeinen kleine Hintergrundprogramme, die anderen Programmen bestimmte Dienste anbieten. Diese können mit ihnen unter anderem über so genannte *Sockets* kommunizieren.
[21] Der Kernel ist der Betriebssystem-Kern.

Positionierungsschicht. In der vorliegenden Implementierung wird keine Modulrelozierung verwendet. Nach der Wahl einer geeigneten Position eines Moduls durch die Allokationsschicht muss hier daher nur die Speicheradresse des entsprechenden Bitstroms an die Konfigurierungsschicht übertragen werden. Die Positionierung ist hier folglich ausschließlich in Software realisiert. Die Allokationsschicht verwendet als Platzierungsstrategie und für die Adressraumverwaltung einen FirstFit-Algorithmus. Die Modulverwaltungsschicht wendet Modul-Caching an, d.h. Module werden nach Beendigung einer Aufgabe nicht sofort gelöscht, sondern nur deaktiviert. Module werden nur geladen, falls kein anderes Modul derselben Komponente bereits geladen und inaktiv ist.

Modulverwaltungs-, Allokations- und Positionierungsschicht werden hier zwar von einem Programm, dem *ROTFL Daemon*, übernommen, im Programm*code* wird jedoch strikt zwischen den Schichten unterschieden. Hiermit wird der bei PALMERA vorgesehene modulare Aufbau erreicht und ein Austauschen oder Anpassen einzelner Schichten leicht möglich gemacht. Die Realisierung dieser Schichten als Daemon hat mehrere Vorteile. Die Kommunikation über Sockets erlaubt eine relativ einfache Nutzung der Daemon-Dienste von mehreren Anwendungen. Für die Socket-Kommunikation spielt es außerdem keine Rolle, von wo der Dienst in Anspruch genommen wird. Bei geeigneten Schnittstellen des Systems (z.B. Ethernet, für das entsprechende RAPTOR2000-Module verfügbar sind) kann daher von außen eine partielle, dynamische Fremdkonfiguration über ein Netzwerk erfolgen. Eine Liste mit verfügbaren Modulen wird außerhalb des Daemons in einer Datenbank gespeichert (*Module Repository*), ebenso sind die Bitströme selbst nicht Bestandteil des Daemons. Dadurch ist die Funktionalität des Daemons unabhängig von den Modulen und die Implementierung eines offenen Systems mit einer dynamisch veränderbaren Moduldatenbank deutlich erleichtert.

Die Kommunikation zwischen Software- und Hardware-Komponenten findet über Kernel-Module statt. Für eine Rekonfigurierung müssen dem Rekonfigurierungsmanager (VCM) die Basisadresse und Größe des Bitstroms sowie ein Befehl durch das *VCM Interface* übermittelt werden. Der Bitstrom selbst wird vom VCM per DMA-Zugriff aus dem Speicher geladen (siehe 5.3.1.1).

Die dynamische Einteilung des Adressraums und die Modul-(De)Aktivierung werden durch den *ROTFL Allocation Manager* vorgenommen. Die gewünschten Anpassungen der Kommunikationsinfrastruktur müssen dem Bus Manager (BM) mitgeteilt werden (siehe 5.3.2.3). Die entsprechenden Zugriffe erfolgen über das Kernel Modul *BM Interface*.

Schlussendlich muss es den Anwendungen ermöglicht werden, auf die geladenen Komponenten zuzugreifen. Hierzu wird durch das Kernel Modul *LOL Manager (LOL – Load On Linux)* ein Gerätetreiber geladen und die neu geladene Modulinstanz im System registriert. Jeder Gerätetreiber wird nur einmal geladen. Die Registrierung muss für jede Modulinstanz vorgenommen werden.

5.3.3.2 Ausführungszeiten

Die Steigerung der Systemleistung durch das Hinzuladen dynamischer Komponenten wurde zu Beginn als ein mögliches Argument für dynamische Rekonfigurierung genannt. Nicht zuletzt deswegen sind die Ausführungszeiten der gerade vorgestellten Software-Komponenten ein wichtiges Qualitätsmerkmal des gesamten Systems. Tabelle 5-2 zeigt die wichtigsten Zeiten. Sie beruhen auf Messungen, die mit einem Hardware-Zähler im System ermittelt wurden.

Tabelle 5-2: Einige Ausführungszeiten

Aufgabe	Zeit (µs)	Bemerkungen
Daemon laden	~ 500	einmalig
Gerätetreiber laden	~ 650	einmal pro Treiber
Modul laden (reaktivieren)	~ 2500	bei jeder Komponentenanfrage
Modul laden (konfigurieren)	~ 3450	bei jeder Komponentenanfrage
Daten lesen	3,6	ein Datenwort (vier Bytes)
Daten schreiben	2,7	ein Datenwort (vier Bytes)

Zu Beginn der Laufzeit, also nach dem Starten des Betriebssystems, muss der ROTFL Daemon geladen werden, um den Konfigurierungs-Dienst anbieten zu können. Dies dauert etwa 500 µs. Bei einer Komponenten-Anfrage seitens einer Anwendung setzt sich die Bearbeitungsdauer zusammen aus der Socket-Kommunikation zwischen Anwendung und Daemon, der Ausführungszeit des Daemons sowie der Ausführungszeit der Hardware-Komponenten inklusive der Verzögerung durch die Kernel-Module. Im besten Fall ist die gewünschte Komponente bereits geladen und kann einfach reaktiviert werden. Hierfür werden etwa 2,5 ms benötigt. Falls eine Komponente durch Rekonfigurierung geladen werden muss, beträgt die gesamte Verzögerung 3,45 ms für ein vier CLB-Spalten Modul. Bei größeren Modulen steigt die Verzögerung linear mit der Bitstromgröße. Aufgrund der Verwaltung der Module und FPGA-Ressourcen in Listen und durch die Nutzung einer FirstFit-Strategie schwankt die Ausführungszeit je nach Belegung des FPGAs. Beim erstmaligen Laden einer Komponente muss zusätzlich der entsprechende Treiber geladen werden. Hierfür werden, je nach Treiber, etwa 650 µs benötigt. Der Zugriff auf eine Komponente mithilfe des bereit gestellten Treibers dauert schließlich 2,7 µs für das Schreiben eines Datenworts und 3,6 µs für das Lesen.

5.3.4 Nutzung des Evaluierungssystems

Der modulare und hierarchische Aufbau sowohl der RAPTOR2000-Umgebung als auch des dynamisch rekonfigurierbaren Systems ermöglicht nicht nur flexible Änderungen, sondern auch eine effiziente Beobachtung aller wichtigen Systemkomponenten während des Betriebs.

Abbildung 5-16 zeigt eine strukturelle Darstellung der Kommunikations- und Konfigurierungsinfrastruktur des Gesamtsystems. Dargestellt sind die RAPTOR2000-Basisplatine, je ein Virtex2Pro und Virtex-II-Modul für die statischen bzw. dynamischen System-Komponenten und der Host-PC, in dem sich das RAPTOR2000-System befindet. Diese Anordnung stellt derzeit die Minimal-Konfiguration der Prototyping-Umgebung dar. Wie das Virtex-II Modul können jedoch noch weitere RAPTOR2000-Module mit dynamisch rekonfigurierbaren Ressourcen hinzugefügt werden. Die farbigen Hinterlegungen in Abbildung 5-16 deuten unterschiedliche Taktdomänen des Systems an. Dank der Eigenschaften der oben vorgestellten Bridges sind die unterschiedlichen Systembereiche voneinander entkoppelt. Bei einem Umstieg auf eine neuere FPGA-Technologie bei einem der FPGAs können somit höhere *Speed-Grades* vollständig zur Leistungssteigerung genutzt werden.

5.3.4.1 Aufteilung des Adressraums

Ein typischer Zugriff einer Software-Komponente auf dem PPC auf ein dynamisches Hardware-Modul erfolgt über den PLB auf den LocalBus und von dort auf den EWB. Abbil-

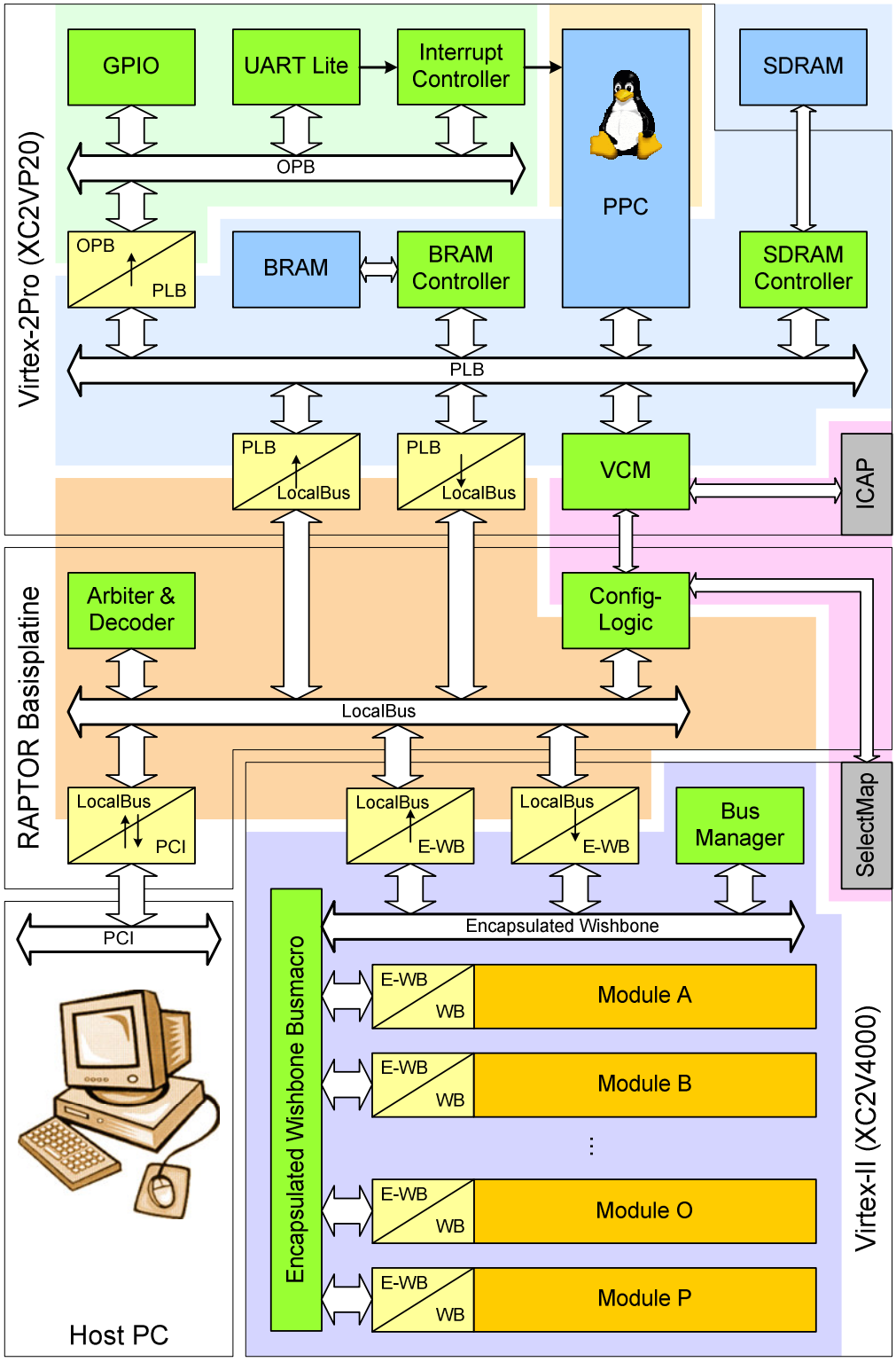

Abbildung 5-16: Kommunikations- und Konfigurierungsinfrastruktur der Entwicklungsumgebung

dung 5-16 zeigt die Adressräume dieser Busse. Links neben den Adressräumen sind jeweils die Basis-Adressen der Systemkomponenten dargestellt. Rechts daneben die *Ranges* in Maskendarstellung. Eine Range von 0xFFFFF000 bedeutet demnach, dass bei einem Zugriff die oberen 20 Bit der Adresse maskiert und somit nicht beachtet werden. Der adressierbare Bereich ist entsprechend noch 12 Bit groß, es kann also auf 2^{12} = 4096 Adressen zugegriffen werden. Die Verbindung zwischen den Bussen stellen *Bridges* her, die zum Teil in vorangehenden Abschnitten beschrieben wurden. Mit Ausnahme der PLB-OPB-Bridge erlauben alle Bridges bidirektionale Zugriffe. Hierdurch kann ein beliebiger Master auf den gesamten Adressraum aller Busse zugreifen, den PCI-Bus des Host-PCs eingeschlossen. Ebenso kann von beliebigen Stellen des Systems auf die Eingabe/Ausgabe-Schnittstellen zugegriffen werden. Die Kommunikationsmöglichkeiten innerhalb des Systems sind also fast unbegrenzt.

Die verschiedenen Adressräume und ihre Aufteilung sind in Abbildung 5-17 dargestellt. PLB und OPB teilen sich einen gemeinsamen Adressraum, der PCI-Bus ist nicht mit abgebildet. Durch die gestrichelten Linien wird beispielhaft gezeigt, wie ein Adressraum in einen Adressbereich eines anderen Adressraums durch das Remapping eingeblendet wird. Hier dargestellt ist die Sicht des PPCs auf dem Virtex-2Pro FPGA. Innerhalb des PLB/OPB Adressraums, in dem er sich befindet, kann er auf alle anderen Slave-Teilnehmer direkt zugreifen. Ein solcher Zugriff wird zum Beispiel durch das *VCM Interface* Kernel Modul durchgeführt, um eine Konfigurierung durch das *VCM* zu initiieren. Zugriffe auf die Hardware-Module in den dynamischen Bereichen des Systems müssen über den LocalBus und EWB stattfinden. Hierzu wird zunächst die *Remap*-Adresse der PLB2LocalBus-Bridge auf die (Basis-)Adresse des Virtex-II RAPTOR2000-Moduls gesetzt. Ein 2 MB großer Adressraum dieses Moduls wird dadurch in den PLB/OPB-Adressraum eingeblendet. Um ein weiteres Remapping zu vermeiden, werden derzeit nur 2 MB des EWB Adressraums genutzt. Somit kann von Seiten des LocalBus jederzeit direkt auf alle dynamischen Module zugegriffen werden. Der Adressbereich des Bus Managers ist ebenfalls statisch auf dem LocalBus eingeblendet. Wie in 5.3.2.3 erläutert, kann der Adressbereich der Makro-Anschlusspunkte zur Laufzeit dynamisch angepasst werden. Insgesamt steht für die 16 Anschlusspunkte ein 20 Bit Adressraum zur Verfügung. Bei einer gleichmäßigen Verteilung, wie sie in Abbildung 5-17 gewählt wurde, stehen für jedes Modul 65536 Adressen zur Verfügung.

Dadurch, dass auf dem EWB nur ein Adressraum verwendet wird, der nicht größer ist als

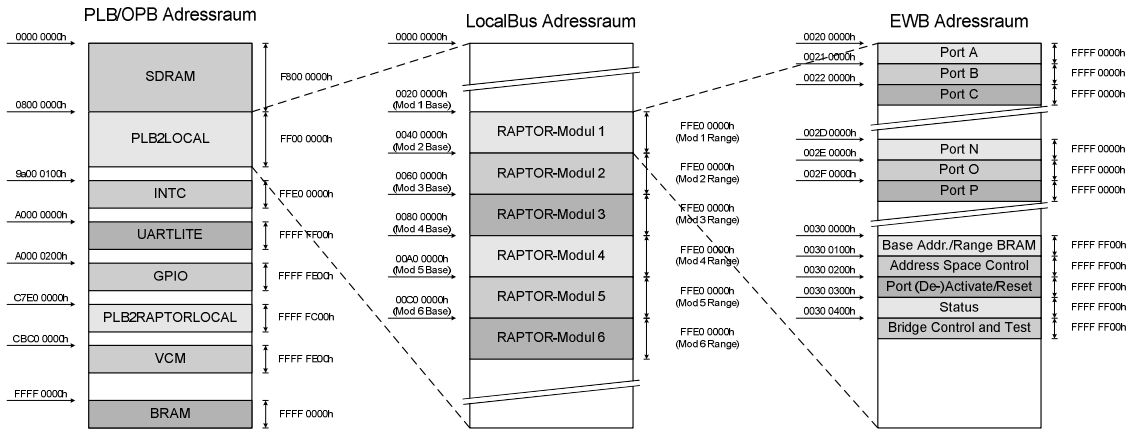

Abbildung 5-17: Adressräume der Prototyping-Umgebung

der Adressbereich eines RAPTOR2000-Moduls auf dem LocalBus, sind alle dynamischen Hardware-Module immer direkt auf dem LocalBus eingeblendet. Jeder Modulinstanz kann für die Dauer ihrer Existenz eine global gültige Adresse zugewiesen werden. Zwei Modulinstanzen können unabhängig von ihrer Position im System über diese Adressen miteinander kommunizieren. Falls sie sich auf demselben FPGA befinden wird direkt über den EWB kommuniziert. Liegt die Zieladresse außerhalb des eigenen LocalBus-Adressraums, wird der Buszugriff vom Bus Manager über die EWB2LocalBus-Bridge auf den LocalBus gegeben. Der LocalBus Arbiter (siehe Abbildung 5-16) selektiert das Ziel-RAPTOR2000-Modul, auf dem dann der Buszugriff über die LocalBus2EWB-Bridge an das Ziel-Hardware-Modul geleitet wird.

Die statische Hardware auf dem Virtex-2Pro wird nicht auf dieselbe Art auf den LocalBus eingeblendet. Stattdessen muss hier die *Remap*-Adresse in der LocalBus2PLB-Bridge entsprechend der Zielkomponente gesetzt werden. Dies kann zu Konflikten führen, wenn zwei Teilnehmer aus dem LocalBus-Adressraum im gleichen Zeitraum auf zwei unterschiedliche Komponenten des PLB/OPB-Adressraums zugreifen wollen. Dies wäre z.B. der Fall, wenn eine Modulinstanz einer dynamischen Systemkomponente mit dem PPC kommuniziert, während vom PCI-Bus auf das SDRAM zugegriffen wird. Da sich diese Zugriffe nicht blockierend durchgeführt, sondern abwechselnd durchgeführt werden, kann nicht garantiert werden, dass die Remap-Adresse immer korrekt ist. Um solche Konflikte zu vermeiden, wird während des Testbetriebs nur der PPC auf den LocalBus eingeblendet. Das Verändern der PLB-Remap-Adresse muss dann unterlassen werden.

5.3.4.2 Monitoring

Die Überwachung und Beobachtung stellt bei mikroelektronischen Systemen generell eine besondere Herausforderung dar, da die Vorgänge oft nur indirekt beobachtet werden können. In dem vorgestellten System sind derzeit vier verschiedene Schnittstellen für die Überwachung implementiert. Die einfachste, in frühen Entwicklungsstadien oft aber auch wichtigste Methode wurde oben bereits erwähnt: Die visuelle Anzeige des Konfigurationszustands des Virtex-II-FPGAs über LEDs (siehe 5.3.2.3). Sie ist insofern effektiv, als dass sie auch dann funktioniert, wenn große Teile der Kommunikationsinfrastruktur fehlerbehaftet sind.

Für die Überwachung der statischen Hardware auf dem Virtex2Pro-FPGA wird die UART-Schnittstelle verwendet, die an die serielle Schnittstelle des Host-PCs angeschlossen werden kann. Ihre Nutzung setzt voraus, dass das PowerPC System mit OPB und PLB funktioniert und dass auf dem PowerPC eine Software läuft, die Statusmeldungen an die UART-Schnittstelle schickt. Hierzu wird nicht zwingend das Betriebssystem benötigt, sondern es kann ebenso gut eine wesentlich weniger komplexe Software verwendet werden. Die UART-Schnittstelle ermöglicht überdies eine bidirektionale Datenübertragung, kann also auch zur Interaktion mit dem System genutzt werden. Derzeit werden alle Text-Ausgaben des Betriebssystems über die serielle Schnittstelle übertragen. Über Befehle können außerdem Statusangaben der Modulverwaltungs- und Allokationsschichten ausgegeben werden.

Auf dem Virtex-2Pro FPGA werden außerdem LEDs zur Statusausgabe genutzt. Sie sind als *General Purpose IOs* (*GPIO*) an den OPB angebunden. Die Möglichkeiten der Datenausgabe sind hier naturgemäß deutlich geringer als bei UART. Trotzdem können die LEDs sinn-

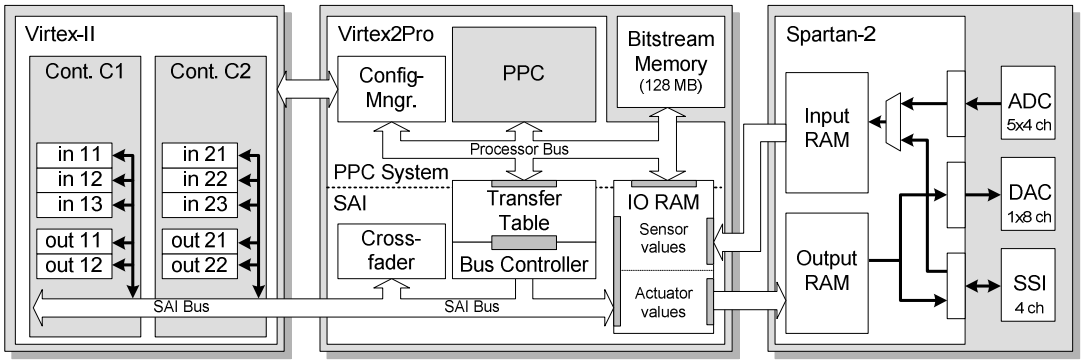

Abbildung 5-18: Dynamisches Austauschen Hardware-basierter Regelungen [PKP07]

voll genutzt werden, falls die serielle Schnittstelle selbst fehlerhaft ist, oder falls das gesamte RAPTOR2000-System im *Stand-Alone*-Betrieb verwendet wird, es also keinen Host-PC gibt.

Die Schnittstelle mit der größten Bandbreite stellt zweifelsohne die PCI-Anbindung zum Host-PC dar. Während es für serielle Kommunikation aber viele frei verfügbare Terminal-Programme gibt, die Text-Nachrichten darstellen können, muss für eine PCI-Überwachung eine eigene Software geschrieben werden. Für die Übertragung nicht Text-basierter Nachrichten bietet sich der PCI-Bus aufgrund seiner Bandbreite und Übertragungslatenz jedoch an. Eine mögliche Nutzung ist die Visualisierung der Modulverwaltungs- und Allokationsschichten. Hiermit könnte sehr komfortabel beobachtet werden, welche dynamischen Komponenten durch Modulinstanzen geladen sind, welche Systemressourcen sie belegen und welche Ressourcen entsprechend noch frei sind. Im Gegensatz zu der derzeit implementierten aktiven Abfrage dieser Werte könnte hier durch *Polling* eine permanente, automatische Aktualisierung vorgenommen werden. Dabei muss allerdings sichergestellt werden, dass dadurch das Betriebssystem nicht zu sehr belastet wird.

5.4 Anwendungsbeispiel: rekonfigurierbare Regelungen

Die Möglichkeiten der Prototyping- und Entwicklungsumgebung auf Basis des RAPTOR2000-Systems wurden bereits parallel zu dieser Arbeit ihm Rahmen des Sonderforschungsbereichs 614 – Selbstoptimierende Systeme des Maschinenbaus – der Deutschen Forschungsgemeinschaft genutzt. In diesen Arbeiten wird zum einen eine leistungsfähige Hardware für Regelungssysteme benötigt, zum anderen sollen diese Systeme flexibel auf sich ändernde Umweltbedingungen reagieren können. Dynamisch rekonfigurierbare Hardware in Form von FPGAs stellt hier eine mögliche Zielarchitektur dar [PKK05].

5.4.1 Systemübersicht

Die in dieser Arbeit entstandene Entwicklungsumgebung wurde ebenso wie der oben präsentierte Entwurfsablauf für dynamisch rekonfigurierbare Hardware-Regelungen erweitert. Abbildung 5-18 zeigt ein Blockdiagramm des erweiterten Systems, das in [PKP07] vorgestellt wurde. Wie gehabt werden auch hier die statischen Systemkomponenten auf einem Virtex-2Pro, die dynamischen auf einem Virtex-II FPGA abgebildet. Als Schnittstelle zu den analogen Regel- und Stellgrößen wurde eine neue RAPTOR2000-Tochterplatine entwickelt, die mit einem eher kleinen Spartan-2 FPGA sowie Analog-Digital-Wandlern (*Analogue to Digital Converter – ADC*) und Digital-Analog-Wandlern (*Digital to Analogue Converter – DAC*)

bestückt ist [Ise04S]. Darüber hinaus besitzt sie synchrone serielle Datenschnittstellen (*Serial Synchronous Interface – SSI*).

Mithilfe der Spartan-2 Platine kann ein klassischer Regelkreis realisiert werden. Analoge Ist-Werte werden auf der Spartan-2 Platine in digitale Werte gewandelt. Die als digitale Schaltung ausgelegten Regler (engl. *controller*) auf dem Virtex-II berechnen hieraus entsprechende Stellgrößen, die wiederum auf der Spartan-2 Platine in analoge Größen gewandelt und an das System zurückgegeben werden. Es ist möglich, mehrere, von einander unabhängige, Regler parallel zu betreiben. Diese Regler können dabei dieselben Ein- und Ausgangsgrößen verwenden, wobei die Ausgangsgrößen im Normalbetrieb immer nur durch einen Regler gestellt werden sollten. Durch die Auslegung der Regler als digitale Schaltungen (im Gegensatz zu sequentiellen Regelprogrammen auf Mikroprozessoren oder Signalprozessoren) erhofft man sich sehr leistungsfähige Regelungen mit kurzen Berechnungszeiten und somit kleinen Zykluszeiten. Durch die Verwendung dynamischer Rekonfigurierung können diese Regelungen zudem fast beliebig zur Laufzeit verändert werden, was bislang ein Privileg sequentiell arbeitender Hardware war. Neben der benötigten Infrastruktur für dynamische Rekonfigurierung, die in den vorangehenden Kapiteln ausgiebig diskutiert wurde, ergeben sich in Regelsystemen einige weitere Randbedingungen durch die geforderte Echtzeitfähigkeit des Regelsystems. Das bedeutet, das im normalen Betrieb nicht nur die Ausführungszeit der Regler auf dem Virtex-II, sondern auch die Übertragungszeit der Informationen im RAPTOR2000-System zwischen ADC/DAC und Regler immer unterhalb einer festgelegten oberen Schranke bleiben müssen. Außerdem darf eine dynamische Rekonfigurierung, die auf einem Virtex-II bis zu 40 ms dauern kann, nicht zu einer Unterbrechung der Regelung führen.

5.4.2 Funktionsweise des Systems

Die Einhaltung der Echtzeitanforderungen wird durch einige zusätzliche statische Systemkomponenten auf dem Virtex-2Pro FPGA sichergestellt (siehe Abbildung 5-18). Alle Ein- und Ausgangswerte (auch Sensor- bzw. Aktorwerte genannt) werden in einem lokalen Speicher (*IO RAM*) zwischengespeichert. Hierdurch bleibt die Modularität des RAPTOR2000-Systems erhalten und es ist möglich, die Anzahl der Ein- und Ausgänge durch die Verwendung mehrerer Spartan-2 Platinen zu erhöhen. Die Inhalte der Ein- und Ausgangsspeicher der Spartan-2-Platinen und des IO-RAMs auf dem Virtex-2Pro werden permanent abgeglichen. Der Datentransfer innerhalb des Systems findet über einen *Sensor-Actuator-Interface-Bus* (SAI Bus) statt. Alle Datentransfers werden von einem Bus-Controller initiiert, so dass keine Konflikte auftreten können und ein deterministisches Übertragungsverhalten vorliegt. Zu Beginn eines Regelzyklus werden die Eingangswerte durch den Bus-Controller vom IO-RAM in Eingangsregister der Regler-Module geschrieben. Nach der Berechnung der neuen Ausgangsgrößen werden diese von den Ausgangsregistern der Regler in das IO-RAM kopiert. Die Zuordnung zwischen den existierenden Ein- und Ausgängen des Systems und den Ein- und Ausgängen der aktuell geladenen Regler ist in einer Transfer-Tabelle (*Transfer Table*) gespeichert, die vom Bus-Controller zyklisch abgearbeitet wird.

Dieser Ablauf von Einlesen, Berechnen und Auslesen stellt den Normalbetrieb dar. Während des Normalbetriebs überwacht ein Supervisor, der hier als Programm auf einem der eingebetteten Prozessoren des Virtex-2Pro realisiert ist, die aktuellen Sensor- und Aktorwerte. Hierzu ist das IO-RAM an den Prozessorbus angebunden. Der Supervisor entscheidet, ob der

aktuell verwendete Regler weiterhin geeignet ist. Ist dies nicht der Fall, wird ein Regler-Austausch eingeleitet. Hierzu wird zunächst mithilfe des Konfigurations-Managers der gewünschte Regler als zusätzliche dynamische Systemkomponente in das Virtex-II FPGA geladen. Anschließend wird die Regelaufgabe von dem alten auf den neuen Regler übertragen. Im ersten Schritt passt der Supervisor die Transfertabelle so an, dass der neue Regler bereits die benötigten Sensoreingangswerte erhält. Anschließend kann ggfs. Gewartet werden bis evtl. vorhandene Einschwingverhalten abgeklungen sind. Danach kann im einfachsten Fall schlicht zwischen den beiden Reglern umgeschaltet werden. Hierzu wird erneut die Transfertabelle angepasst, so dass nun die Ausgangswerte des neuen Reglers an die Aktorausgänge übertragen werden. Der alte Regler wird danach nicht mehr benötigt, die von ihm belegten FPGA-Ressourcen können wieder freigegeben werden.

Im Allgemeinen Fall bewirkt ein hartes Umschalten zwischen zwei Reglern einen Sprung in den Ausgangswerten. Um solche Sprünge zu vermeiden, können Überblendfunktionen verwendet werden, die für fließende Übergänge zwischen den jeweiligen Ausgangswerten des alten und neuen Reglers sorgen. In dem hier vorgestellten System ist daher eine *Cross-Fader*-Komponente enthalten, die eine oder mehrere Überblendfunktionen realisieren kann. Der Cross-Fader ist ebenfalls am SAI-Bus angeschlossen und kann somit vom Bus-Controller adressiert werden. Während eines Überblendvorgangs werden die Ausgangswerte von altem und neuem Regler an den Cross-Fader übertragen. Dieser erzeugt hieraus einen gemeinsamen Ausgangswert, der dann in das IO-RAM geschrieben wird.

Aufgrund der Echtzeitbedingungen müssen für einen neu zu ladenden Regler unbedingt ausreichend Ressourcen im dynamischen Bereich verfügbar sein. Dies kann sichergestellt werden, indem feste Modulplätze verwendet und für jede Regelung zwei Modulplätze reserviert werden. Bei freien Platzierungsverfahren ist dies nur noch unter besonderen Bedingungen möglich, da die Gefahr der Fragmentierung der freien Flächen besteht.

5.4.3 Inverses Pendel

Die Funktionsweise des Systems wurde anhand einer Regelung für ein inverses Pendel überprüft. Bei diesem Demonstrator ist ein Pendel an einem Schlitten befestigt, der in der Waagerechten entlang einer Achse bewegt werden kann. Die Position des Schlittens auf der Führungsschiene und die Abweichung des Pendelausschlags zur Ruheposition werden gemessen. Ziel der Regelung ist es, das Pendel durch eine geeignete Schlittenbewegung zunächst aufzuschwingen und dann senkrecht über dem Schlitten zu balancieren (siehe Abbildung

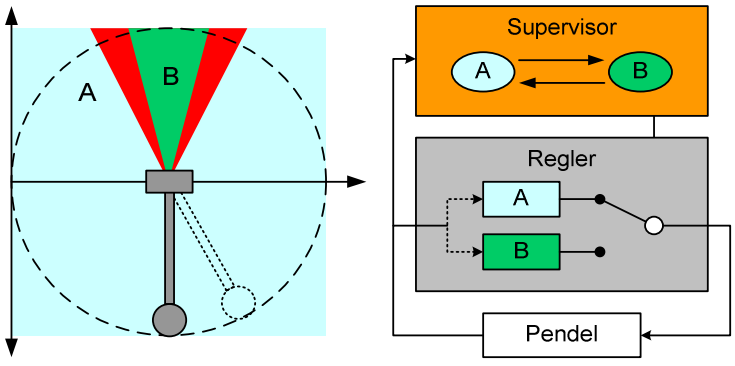

Abbildung 5-19: Prinzipschaltbild der Regelung des inversen Pendels

5-19). Das Beispiel des inversen Pendels wurde gewählt, weil es zum einen ein klassisches Lehrbeispiel der Regelungstechnik ist, zum anderen veranschaulicht es sehr schön zwei verschiedene Betriebszustände: Aufschwingen und Balancieren. Für die Veranschaulichung der dynamischen Rekonfigurierung wurde für beide Betriebszustände je ein Regler implementiert.

Abbildung 5-19 zeigt den prinzipiellen Aufbau des Systems. Der Betriebszustand ist hier ausschließlich von der Position des Pendels – dem aktuellen Einschlagwinkel – und der Winkelgeschwindigkeit abhängig. Der Betriebszustand *Aufschwingen* ist mit A gekennzeichnet, der Zustand *Balancieren* mit B. im rechten Teil der Abbildung ist das Regelschema dargestellt. Der Supervisor kennt zu jedem Zeitpunkt durch Auslesen des IO RAMs Winkel und Winkelgeschwindigkeit des Pendels. Verlässt das Pendel den für den aktuellen Betriebszustand gültigen Bereich, wird der jeweils andere Regler geladen und umgeschaltet. Die Reglermodule für das Aufschwingen und Balancieren benötigen mit 1088 Slices bzw. 1723 Slices zusammen etwa so viele Ressourcen wie ein Referenzregler (2710 Slices), der Aufschwingen und Balancieren beherrscht [KKP05b]. Berücksichtigt man die für dynamische Rekonfigurierung benötigten, zusätzlichen Hardware-Komponenten sowie den Verschnitt fester Modulplätze, fällt der Ressourcenbedarf des Regelsystems mit dynamischer Rekonfigurierung deutlich höher aus als die statische Implementierung mit einem etwas komplexeren Regler. Es kann jedoch argumentiert werden, dass in einem System mit vielen verschiedenen Betriebszuständen und vielen verschiedenen Teil-Regelungen ein Einsparpotential bezüglich der benötigten Ressourcen besteht. Insbesondere, da hier in keinem Fall mehr als zwei Regler gleichzeitig geladen werden müssen.

5.5 Integrierter Entwurfsablauf für dynamisch rekonfigurierbare Systeme

Neben den technischen Voraussetzungen, die für die Realisierung feingranularer, eindimensionaler Platzierungsverfahren geschaffen werden müssen, ist vor allem das „nutzbar machen“, also das Bereitstellen eines weitgehend automatisierten Entwurfsablaufs ein wichtiges Qualitätsmerkmal eines Platzierungsverfahrens. Wie schon mehrfach erwähnt, ist die Realisierung komplexer rekonfigurierbarer FPGA-basierter Systeme zurzeit nur mit den FPGAs von Xilinx möglich. Für die Nutzung dieser FPGAs, insbesondere für die Abbildung digitaler Schaltungen, ist man automatisch auf die von Xilinx bereitgestellten Werkzeuge für das Mapping, Place & Route, sowie das Erzeugen der Bitströme angewiesen. In den letzten Jahren hat Xilinx nach und nach die Unterstützung partieller und dynamischer Rekonfigurierung ausgebaut. Ein wichtiger Meilenstein war die *Xilinx Application Note 290* [Xil02], die eine schrittweise Anleitung für den Entwurf dynamisch rekonfigurierbarer Systeme auf Basis der Virtex Technologie gab und für einige Jahre die Grundlage fast aller Implementierungen im akademischen Umfeld war. Damals wurden die bestehenden Entwicklungswerkzeuge erweitert, um dynamische Rekonfigurierung erstmals zu ermöglichen. Inzwischen wird ein halbwegs fehlerfreier und dokumentierter Entwurfsablauf angeboten, der von einem kleinen Entwicklungsteam gepflegt und weiterentwickelt wird. Die derzeitige Version wird als *Early Access Partial Reconfiguration* [Xil06] vertrieben und stellt noch kein vollwertiges Produkt dar. Der Fokus liegt hier offensichtlich darin, überhaupt einen Entwurfsablauf bereitzustellen, der ein akzeptables Maß an Zuverlässigkeit erreicht und neben der Implementierung auch die Verifikation

(funktionale Simulation, statische Timing Analyse) unterstützt [LBM06]. Hierin kann auch die Ursache dafür vermutet werden, dass Xilinx derzeit ausschließlich die Rekonfigurierung mit festen Modulplätzen ermöglicht. Für die in dieser Arbeit betrachteten Systeme mit freier Platzierung gibt es keine Entwurfswerkzeuge. Sie können daher nur durch eine Modifikation des bestehenden Entwurfsablaufs realisiert werden. Die Komplexität des Entwurfs nimmt dabei im Vergleich zu Systemen mit festen Modulplätzen stark zu. Die Definition einer geeigneten Kachelung der rekonfigurierbaren Bereiche, die wesentlich höhere Anzahl möglicher Modulpositionen und der damit gestiegene Implementierungsaufwand, und nicht zuletzt der Bedarf einer wesentlich komplexeren Kommunikationsinfrastruktur machen den manuellen Entwurf sehr fehleranfällig und langwierig. Aus diesem Grund wurde in dieser Arbeit eine Entwurfsumgebung entwickelt, die den Entwurf komplexer dynamisch rekonfigurierbarer Systeme zu großen Teilen automatisiert. Mit ihr ist es möglich, die in dieser Arbeit entwickelten Konzepte und Methoden für dynamisch rekonfigurierbare Systeme nicht nur prototypisch umzusetzen, sondern je nach Anforderung flexibel, zuverlässig und schnell Systeme mit freier Platzierung zu erstellen. Der zugehörige Entwurfsablauf wird *INDRA – Integrated Design Flow for Reconfigurable Architectures* – genannt [HKK07a].

5.5.1 Der INDRA-Entwurfsablauf

Abbildung 5-20 zeigt den INDRA-Entwurfsablauf mit den beinhalteten Entwurfsschritten und den wesentlichen erzeugten Zwischenformaten und Dateien. Ausgangspunkt ist die Partitionierung des Systems in statische und dynamische Systemkomponenten. Dies kann entweder manuell durch den Entwickler geschehen, oder mithilfe von Partitionierungsverfahren zum Teil automatisiert durchgeführt werden. Die Partitionierung selbst ist nicht Bestandteil von INDRA, wohl aber die durch ihr erzeugten Dateien, die die Eingabe des INDRA-Entwurfsablaufs darstellen. Im optimalen Fall liegen hier Beschreibungen für die statischen und dynamischen Komponenten sowie deren Verschaltung auf der obersten Hierarchieebene (*Top Level*) in Form von HDL-Code oder Netzlisten vor. Darüber hinaus kann hier ein Ablaufplan vorliegen, der angibt, zu welchen Zeitpunkten die dynamischen Komponenten zur Laufzeit in das System geladen werden können oder müssen.

Mit diesen Informationen muss im nächsten Schritt, dem Layout oder Floorplanning, ein Plan der benötigten Ressourcen erstellt werden. Für die statischen Komponenten kann dies relativ einfach erfolgen, da sie zu allen Zeiten die gleiche Menge Ressourcen belegen. Sie kann bei Netzlisten direkt und bei HDL nach erfolgter Synthese abgeschätzt werden. Für die dynamischen Komponenten ist die Größe der ausgewiesenen rekonfigurierbaren Bereiche relevant für die Ressourcenabschätzung. Diese hängt wiederum von der Größe der dynamischen Komponenten, dem Ablaufplan und dem gewählten Platzierungsverfahren ab, da sich die dynamischen Komponenten diese Bereiche sowohl in der Zeit wie auch in der Fläche teilen.

Um verschiedene Entwurfsvarianten bewerten zu können, wurde die Simulationsumgebung SARA entwickelt, mit der die Ressourcenauslastung rekonfigurierbarer Bereiche für einen gegebenen Satz dynamischer Komponenten und einen Ablaufplan bestimmt werden kann (siehe 5.5.2). SARA erleichtert also die Wahl eines geeigneten Platzierungsverfahrens, das mit möglichst kleinen rekonfigurierbaren Bereichen die Anforderungen erfüllt.

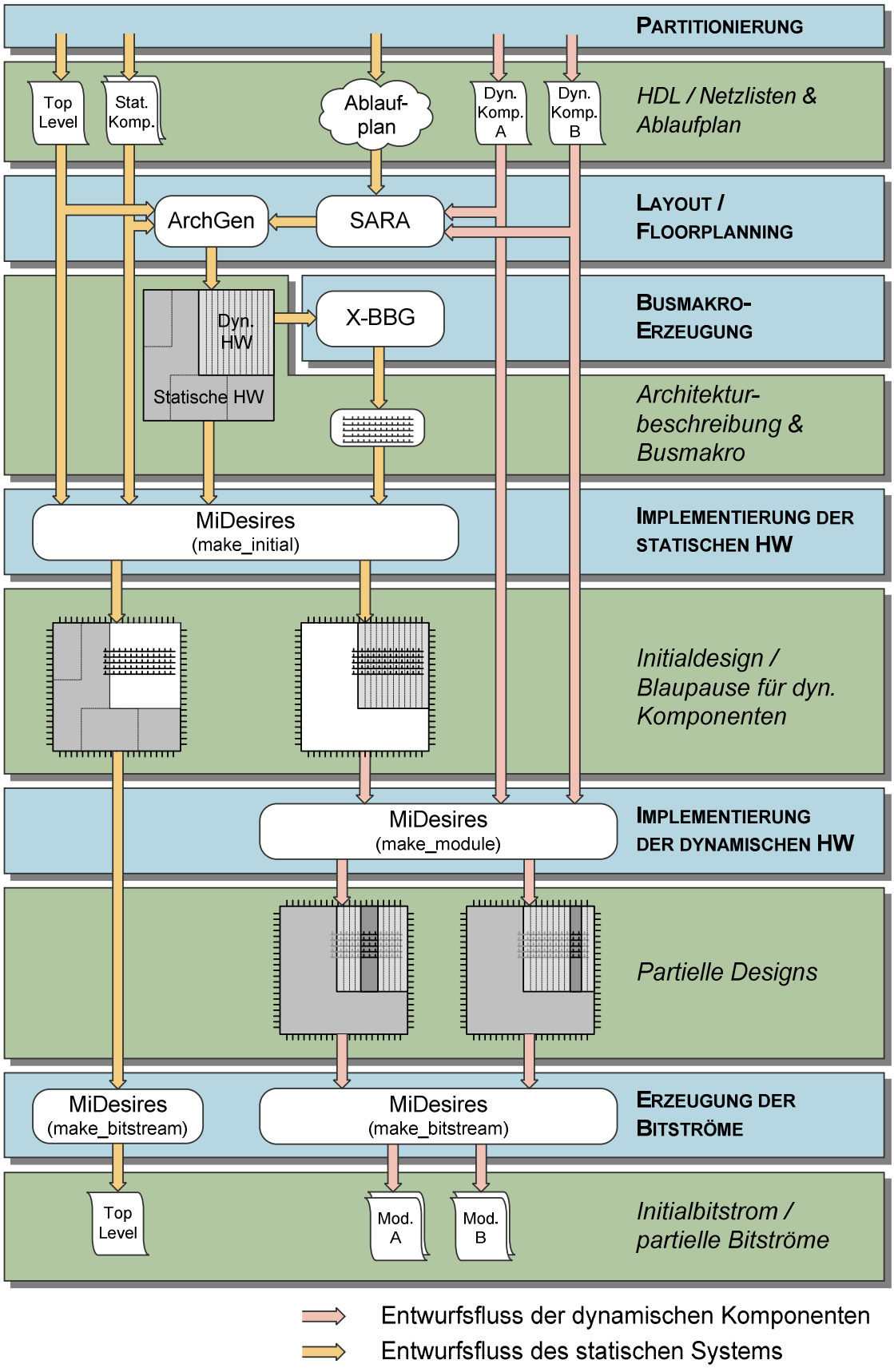

Abbildung 5-20: Der INDRA-Entwurfsablauf für dynamisch rekonfigurierbare FPGA-Systeme

Mithilfe der mit SARA gewonnenen Erkenntnisse und des Ressourcenbedarfs der statischen Komponenten kann nun ein geeigneter FPGA ausgewählt und eine Architekturbeschreibung erstellt werden. Dieser Schritt wird in INDRA *Architecture Generation* (ArchGen) genannt. Eine Architekturbeschreibung enthält Platzierungsvorgaben für die statische Hardware und beschreibt die Art und Form der Kacheln, die für dynamische Rekonfigurierung genutzt werden können. Die Erzeugung einer Architekturbeschreibung könnte prinzipiell sehr elegant über eine graphische Benutzeroberfläche geschehen, wie sie *Floorplanner* in der Regel besitzen. Zusätzlich zu den existierenden Lösungen [Jac07] muss hier aber eine Beschreibung der rekonfigurierbaren Bereiche und der Platzierungsverfahren ermöglicht werden. Aufgrund des relativ hohen Entwicklungsaufwands für einen eigenen Floorplanner und dem überschaubaren Aufwand für einen Handentwurf wird die Architekturbeschreibung derzeit manuell erzeugt.

Nachdem mit der Architekturbeschreibung die Einteilung der Ressourcen und das Platzierungsverfahren festgelegt wurde, muss eine entsprechende Kommunikationsinfrastruktur erstellt werden. Hierzu wird das Werkzeug X-BBG verwendet, das aus der Architekturbeschreibung die Positionen extrahiert, an denen Zugangspunkte (Ports) für die Kommunikation bereitgestellt werden müssen, und automatisch ein passendes Busmakro und entsprechende VHDL-Wrapper erzeugt. Hiermit sind nun alle notwendigen Systembestandteile und Beschreibungen vorhanden, so dass das statische Initialdesign und die dynamischen Komponenten implementiert werden können. In den bis hierher erfolgten Entwurfsschritten hat ein Entwickler Einflussmöglichkeiten auf die Ausgestaltung des Systems. Sie bilden daher das *Frontend* von INDRA. Alle nun folgenden Schritte haben nur zum Ziel, für die erfolgte Systembeschreibung entsprechende Bitströme für das statische Design und die dynamischen Komponenten zu erzeugen. Sie werden als *Backend* bezeichnet.

Das Backend wird in INDRA mithilfe des Werkzeugs MiDesires durchgeführt. Mit ihm wird zunächst die statische Hardware implementiert. Als Ergebnis liegen danach ein fertig platziertes und verdrahtetes Initialdesign sowie eine Vorlage (Blaupause) für die Erstellung der dynamischen Komponenten vor. Die Blaupausen enthalten das platzierte Busmakro und Informationen über die vorhandenen Kacheln. Mit diesen Informationen können anschließend für alle dynamischen Komponenten mögliche Modulpositionen gefunden und entsprechende Module implementiert werden. Hiernach liegen somit komplett platzierte und verdrahtete, partielle Designs vor, die die Module repräsentieren. In einem letzten Schritt werden dann aus dem Initialdesign und den partiellen Designs Bitströme erzeugt womit der Entwurfsablauf abgeschlossen ist.

Mit dem INDRA-Entwurfsablauf und den dazugehörigen Entwurfswerkzeugen ist es möglich, statische und dynamische Systemkomponenten getrennt voneinander zu entwerfen. Dies ist zum Beispiel bei der Entwicklung eines offenen Systems notwendig, bei dem zur Entwurfszeit des statischen Systems noch nicht bekannt ist, welche Komponenten zur Laufzeit in das System geladen werden. Für den Entwurf der dynamischen Komponenten muss der Aufbau des statischen Systems jedoch bekannt sein, so wie beispielsweise beim Entwurf von Prozessorsoftware auch die Zielarchitektur bekannt sein muss.

5.5.2 Simulation mit SARA

SARA (*Simulation Framework for the Analysis of Reconfigurable Architectures*) wurde entwickelt, um auf einer abstrakten Ebene verschiedene Systemarchitekturen und Platzierungsverfahren miteinander vergleichen zu können [KKK05]. Abbildung 5-21 zeigt den dreigeteilten Aufbau des Simulators: Die Vorbereitung der Simulation durch Vorgaben, die Simulation selbst und eine abschließende Analyse. Die betrachteten rekonfigurierbaren Ressourcen werden im Simulator durch ein virtuelles FPGA repräsentiert.

Eingabeformat des Simulators sind Komponentenbeschreibungen in der in Abschnitt 3.1 beschriebenen Form. In der Simulationsvorbereitung werden für diese Komponenten durch eine virtuelle Synthese Module erstellt. Hierfür wird zunächst ein Platzierungsverfahren festgelegt, das bestimmt, wie die Komponenten auf das virtuelle FPGA abgebildet werden. Mit den gegebenen geometrischen Rahmenbedingungen (Kachelung) können zu den Komponenten virtuelle Module erstellt werden. Daneben kann mit einem Ablaufgenerator ein Ablaufplan erstellt werden, der vorgibt, zu welchem Zeitpunkt welche Komponente angefordert wird und wie lange sie benötigt wird. Hierfür können verschiedene Modi für Anforderungshäufigkeit und Verweildauer gewählt werden. Die automatische Erzeugung eines Ablaufplans ist natürlich willkürlich und wird in der Regel nicht die tatsächliche Situation während der Laufzeit widerspiegeln. Sie ist jedoch hilfreich, um verschiedene mögliche Szenarien durchzuspielen, wenn kein realer Ablaufplan existiert.

Mit den virtuellen Modulen und einem Ablaufplan kann eine Simulation durchgeführt werden. Während der Simulation übernimmt eine Ressourcen- und Konfigurationsverwaltung das Laden von Modulen auf das virtuelle FPGA. Die Modulanfragen gemäß dem vorhandenen Ablaufplan erzeugt eine virtuelle Anwendung. Wie in einem realen System werden Module nach einer Platzierungsstrategie auf das FPGA geladen. Die belegten Ressourcen werden nach Ablauf der Verweildauer eines Moduls wieder freigegeben, oder das Modul wird lediglich als inaktiv gekennzeichnet. Um das Verhalten eines rekonfigurierbaren Systems möglich detailgetreu wiederzugeben, berücksichtigt der Simulator auch die Zeiten, die für das Auffinden geeigneter Positionen und für das Konfigurieren des FPGAs benötigt werden.

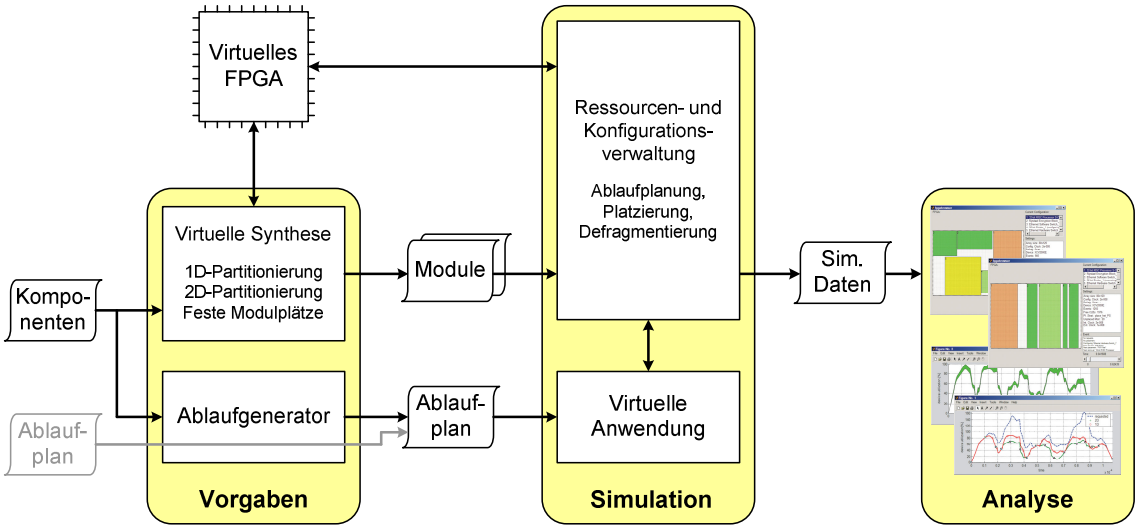

Abbildung 5-21: Simulationsumgebung SARA

Während der Simulation werden alle auftretenden Ereignisse gespeichert. Die Simulationsdaten können somit anschließend analysiert und graphisch aufbereitet dargestellt werden. Es ist möglich, die Konfigurationszustände des virtuellen FPGAs zu jedem Zeitpunkt der Simulation einzeln zu betrachten oder den gesamten Ablauf erneut als Animation abspielen zu lassen. Darüber hinaus können Qualitätsmaße wie Ressourcenauslastung und Fragmentierung in Abhängigkeit von der Zeit dargestellt werden. Insbesondere können die Ergebnisse verschiedener Simulationen übereinander aufgetragen werden. Dies ermöglicht den Vergleich verschiedener Platzierungsverfahren für identische Komponenten und Ablaufpläne. Ebenso können verschieden große rekonfigurierbare Bereiche für denselben Benchmark verglichen werden. Eben diese Möglichkeit ist im INDRA-Entwurfsablauf sehr hilfreich bei der Partitionierung der rekonfigurierbaren Ressourcen und der Auswahl eines geeigneten Platzierungsverfahrens. Darüber hinaus kann SARA bei der Entwicklung und Bewertung neuartiger Platzierungsverfahren oder FPGA-Architekturen verwendet werden, wie zum Beispiel in [Koe07] gezeigt.

5.5.3 Architekturbeschreibung

Für die Architekturbeschreibungen wurde ein eigenes Format definiert, das mit *reconfigurable architecture description* (rad) bezeichnet wird. In Abbildung 5-22 ist ein Ausschnitt aus einer rad-Datei abgebildet. Zu sehen sind die Beschreibungen zweier Kacheln (*tile a* und *tile b*) und einer statischen Komponente (*static staticComp*). Die Kacheln werden gemäß der in Abschnitt 3.1.1 gegebenen Modellierung durch die Anzahl der enthaltenen Ressourcen $\bar{r}$, einen Kachel-Typ φ, und den äußeren geometrischen Eigenschaften, gegeben durch die Koordinaten k_{ol} und k_{ur}, beschrieben. Da Xilinx für jeden Ressourcentyp separate Koordinatensysteme verwendet, sind auch hier Koordinaten angegeben, die überlappt immer eine zusammenhängende, rechteckige Fläche des FPGAs bilden. Die ebenfalls angegebenen Masken (*partMask0*, etc.) werden für die Erzeugung der partiellen Bitströme benötigt und bezeichnen den Bereich des Konfigurationsspeichers, der die Konfigurationsdaten der spezifizierten Kachel enthält. Auch hier wird zwischen den Ressourcen-Typen unterschieden. Die Angaben der benutzten Ressourcen, die auch aus den Koordinaten ersichtlich sind, zeigen, dass es sich bei Kachel *a* um eine Kachel mit 1280 Slices und je 20 eingebetteten BlockRAMs und Multiplizierer handelt. Kachel *b* enthält ausschließlich Slices und hat deshalb eine andere Typenbezeichnung φ (*Type*) als Kachel *a*.

Für das Floorplanning der statischen Komponenten sind keine Angaben zu den Ressourcen erforderlich. Es müssen lediglich Flächenrestriktionen angegeben werden, um zu verhindern, dass statische Komponenten in die dynamischen Bereiche platziert und verdrahtet werden.

5.5.4 Busmakro-Erzeugung mit X-BBG

Bei der Verwendung von Kantenmakros können mit einem wenig komplexen Makro durch Mehrfachinstantiierung beliebig viele Signale zwischen statischem und dynamischem Bereich übertragen werden. Die Mehrfachinstantiierung wird in der HDL-Beschreibung vorgenommen, die Verdrahtung der Makroinstanzen findet ausschließlich im statischen Bereich statt und kann daher automatisch vom PAR-Werkzeug durchgeführt werden.

Eingebettete Makros weisen aufgrund der gewünschten Homogenität auch sehr regelmäßige Strukturen auf. Eine Mehrfachinstantiierung eines wenig komplexen Teil-Makros (das bei-

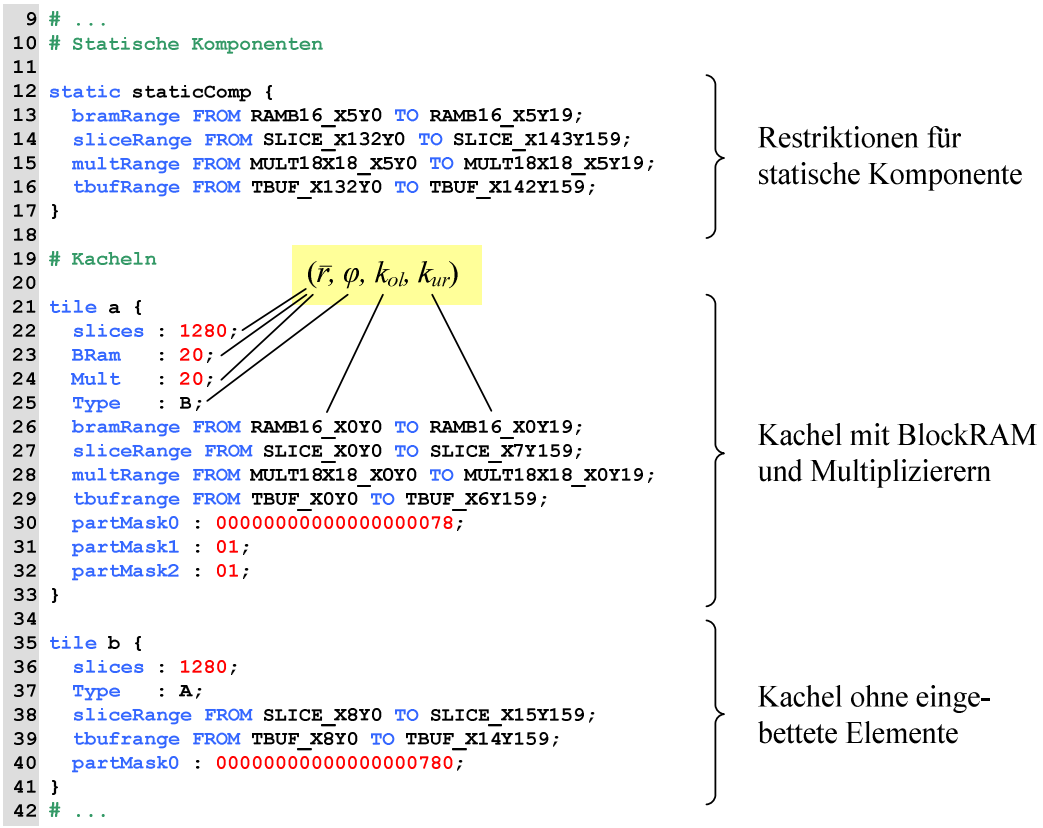

Abbildung 5-22: Ausschnitt einer Architekturbeschreibung

spielsweise einen Anschlusspunkt bildet) bietet sich auch hier an. Allerdings liegen die Verbindungen dieser Makroinstanzen fast ausschließlich innerhalb des dynamischen Bereichs. Die Verwendung des PAR-Werkzeugs für die Verdrahtung würde hier zu inhomogenen Verbindungen führen und ist somit ausgeschlossen. Ein manuelles Routen dieser Signale (z.B. mit dem Xilinx FPGA-Editor) ist zwar prinzipiell möglich, hat aber deutliche Nachteile. Für das in Kapitel 4 betrachtete Beispielmakro müssen mehrere tausend Signalleitungen korrekt und homogen geroutet werden und zusätzlich zu ebenfalls mehr als tausend Ein- und Ausgangssignalen mit einheitlichen Namen versehen werden. Es ist leicht ersichtlich, das ein manuelles Vorgehen hier eine immense Fehleranfälligkeit und einen nicht zu unterschätzenden Zeitaufwand mit sich bringt. Die Entwicklung eines eigenen Entwurfswerkzeugs für die Busmakroerzeugung ist somit aufgrund der Komplexität der für freie Platzierung benötigten Busmakros erforderlich.

Der im Rahmen dieser Arbeit entstandene Busmakro-Generator X-BBG (*XDL-based Busmacro Generator*, [Hag06D]) basiert auf der Xilinx-eigenen Entwurfssprache XDL (*Xilinx Design Language*). Mit XDL können Makros wie auch gewöhnliche FPGA-Schaltungen textuell beschrieben werden. X-BBG nutzt diese Möglichkeiten, um kleine Teil-Makros (Makro-Primitiven) zu duplizieren und homogen miteinander zu einem einzigen Makro zusammenzufügen. In einem ersten Schritt werden aus Elementar-Primitiven komplette Anschlusspunkte erzeugt. Die Anzahl der Adress- und Datenleitungen sind dabei parametrierbar. Im zweiten Schritt wird aus diesen Anschlusspunkt-Primitiven das Gesamt-Makro erzeugt, wobei die Anzahl, Art und Position der Anschlusspunkten vorgegeben werden muss. Auch die Verdrah-

tung findet dabei voll-automatisch durchgeführt. Beispiele für die Zusammensetzung von Anschluss-Primitiven aus Elementar-Primitiven sind in Tabelle 4-3 auf Seite 77 dargestellt.

Um generierte Busmakros besser handhabbar im weiteren Entwurf des Gesamtsystems zu machen, erzeugt X-BBG eine VHDL-Datei, die viele Signale zu Bussen zusammenfasst. Dieser so genannte *Wrapper* kann ohne weiteres in die Beschreibung des statischen Systems eingefügt werden.

In der Literatur finden sich Veröffentlichungen über ähnliche Busmakro-Generatoren. Sie beschränken sich aber entweder auf die Erzeugung von Kantenmakros (Claus et al. [CZH07]) oder erzeugen keine homogenen Kommunikationsmakros.

5.5.5 Backend mit MiDesires

Aus einer Architekturbeschreibung, dem vollständigen Busmakro sowie den HDL-Beschreibungen der statischen Hardware und der dynamischen Komponenten können die gewünschten partiellen Bitströme und der Initialbitstrom weitgehend automatisiert erzeugt werden. Diese Entwurfsschritte werden daher als *Backend* bezeichnet. Bekannte Entwurfsabläufe von Xilinx [Xil02, Xil06] können nicht verwendet werden, da sie nicht für Systeme mit freier Platzierung geeignet sind. Auf die verfügbaren Werkzeuge von Xilinx kann jedoch nicht verzichtet werden. Es wurde daher ein Software-Paket entwickelt, das die Xilinx Werkzeuge geeignet ansteuert und das als MiDesires (*Module implementation Design flow for reconfigurable systems*) bezeichnet wird. Im Gegensatz zu anderen Backend-Werkzeugen (z.B. [NKD05, RI04]) liegt der Fokus von MiDesires auf der vollständigen Unterstützung freier Platzierungsverfahren und der Einbindung der vorher generierten, homogenen Kommunikationsinfrastrukturen.

MiDesires arbeitet in drei Entwurfsschritten. Im ersten Schritt (*make_initial*) werden alle statischen Komponenten (einschließlich des Busmakros) zu einem Initialdesign zusammengefügt. Gleichzeitig werden die benötigten Dateien für die Implementierung der dynamischen Komponenten erzeugt. Sie werden hier zusammenfassend als Blaupause bezeichnet. Die Blaupause enthält alle statischen Informationen der dynamischen Bereiche, wie z.B. die Größe der Bereiche und Kacheln oder die Schaltungsbeschreibung des Busmakros. Im zweiten Schritt werden mithilfe der Blaupause die möglichen Module zu allen dynamischen Komponenten erzeugt (*make_module*). Da dynamische Bereiche in der Regel nicht vollständig homogen aufgebaut sind, prüft MiDesires, wie viele unterschiedliche Module je Komponente erzeugt werden müssen, um eine maximale Entscheidungsfreiheit bei der Platzierung zur Laufzeit zu erhalten, und erzeugt diese Module automatisch. Schließlich werden in einem dritten Schritt (*make_bitstream*) die entsprechenden Bitströme zu den partiellen Designs sowie dem Initialdesign erzeugt. Hierbei kann angegeben werden, ob zur Laufzeit Modulrelozierung angewendet wird. Falls dies nicht der Fall ist, wird die Modulrelozierung bereits von MiDesires durchgeführt, so dass am Ende für jedes Modul Bitströme zu allen möglichen Modulpositionen existieren.

Mit einem Softwarepaket wie MiDesires ist es möglich, die große Anzahl von Modulimplementierungen, die für dynamisch rekonfigurierbare Systeme mit freier Platzierung typisch ist, automatisiert durchführen zu lassen. Ein System mit nur wenigen Komponenten benötigt dutzende (erfolgreiche) Modulimplementierungen, von denen jede einzelne mit modernen, leistungsfähigen Desktop-PCs Stunden dauern kann. Mit MiDesires können diese Prozesse

unbeaufsichtigt laufen. Alle benötigten Dateien (UCFs, VHDL-Wrapper, etc.) werden automatisch erzeugt, die korrekte Parametrierung der beteiligten Xilinx-Werkzeuge wird automatisiert durchgeführt. Die Quelldateien der statischen und dynamischen Systemkomponenten (VHDL-Dateien, Verilog-Dateien und Netzlisten) müssen vom Benutzer lediglich in ein ebenfalls von MiDesires bereitgestelltes Dateiverzeichnis kopiert werden. Alle von Xilinx-Werkzeugen erzeugte Protokolldateien (*Log-Files*) werden für jede Implementierung gesondert gesammelt. Um dem Benutzer am Ende eine Übersicht über den Erfolg der Implementierungen zu geben, wird eine zusammenfassende Protokolldatei durch MiDesires erstellt.

5.6 Zusammenfassung

In diesem Kapitel wurde eine Entwicklungsumgebung vorgestellt, die sowohl die prototypische Realisierung dynamisch rekonfigurierbarer Systeme als auch Untersuchungen der dynamischen Rekonfigurierung selbst ermöglicht. Als Basissystem wurde das RAPTOR2000-System gewählt, das für prototypische Entwicklungen statischer FPGA-Systeme entwickelt wurde und somit einen Großteil der benötigten Infrastruktur für die Kommunikation und Konfigurierung bereits enthielt. Durch die Erweiterung der Konfigurierungsinfrastruktur kann der modulare Aufbau des RAPTOR2000-Systems mit einer Hauptplatine und bis zu sechs FPGA-Tochterplatinen vollständig für dynamisch rekonfigurierbare Anwendungen genutzt werden. Für den Aufbau dynamisch rekonfigurierbarer Systeme wurde das PALMERA-Schichtenmodell vorgestellt. PALMERA weist die für eine dynamische Rekonfigurierung benötigten Aufgaben unterschiedlichen Schichten zu und definiert die Schnittstellen zwischen diesen Schichten. Auf diese Weise wird eine schrittweise Abstrahierung von der rekonfigurierbaren Hardware bis zur Anwendung erreicht.

Mithilfe des PALMERA-Schichtenmodells wurde aufbauend auf der erweiterten RAPTOR2000-Umgebung ein Evaluierungssystem für die Bewertung unterschiedlicher Platzierungsverfahren entwickelt, das unter anderem für die Untersuchung der Ressourceneffizienz im folgenden Kapitel verwendet wird. Die statischen und dynamischen Systemkomponenten wurden auf unterschiedlichen FPGAs untergebracht. Dadurch können die Ressourcen eines FPGAs nahezu vollständig für dynamische Komponenten verwendet werden. Noch größere Systeme können durch die Verwendung mehrerer FPGA-Tochterplatinen realisiert werden. Wichtigste statische Systemkomponenten sind der neu entwickelte Konfigurierungsmanager (VCM) sowie das Prozessorsystem, auf dem ein PALMERA-konformes Linux-Betriebssystem läuft. Es ermöglicht Software-Anwendungen, dynamische Hardware-Komponenten anzufordern, lädt diese automatisch und bindet entsprechende Treiber ebenfalls automatisch ein. Ausgehend von diesen statischen Hardware- und Software-Komponenten können fast beliebige Platzierungsverfahren auf den für dynamische Komponenten vorgesehenen FPGAs realisiert werden. Durch die konsequente Einhaltung des PALMERA-Schichtenmodells können alle relevanten Parameter eines dynamisch rekonfigurierbaren Systems (Platzierungsverfahren, Platzierungsstrategie, Größe der dynamischen Bereiche, Ziel-FPGA, etc.) durch nur geringfügige Änderungen realisiert und miteinander verglichen werden.

Insbesondere für die Implementierung der dynamischen Komponenten wird jedoch ein Entwurfsablauf benötigt, der von bestehenden Entwurfswerkzeugen nicht unterstützt wird. In

dieser Arbeit wurde daher der INDRA-Entwurfsablauf erstellt und eine entsprechende Werkzeugkette realisiert. Mit INDRA wird insbesondere die Realisierung freier Platzierungsverfahren, die durch komplexe Busmakros und eine Vielzahl unterschiedlicher Modulpositionen gekennzeichnet sind, erst ermöglicht.

Die Nutzung dynamischer Rekonfigurierung wurde beispielhaft anhand digitaler Regelungen gezeigt. Es wurde ein System vorgestellt, dass es ermöglicht, mehrere digitale Regler als dynamische Systemkomponenten zu implementieren und sie unter Einhaltung von Echtzeitbedingungen zur Laufzeit auszutauschen. Dadurch wird die Leistungsfähigkeit FPGA-basierter Regler mit einer Flexibilität erweitert, die bislang Prozessorsystemen vorbehalten war. Zusätzlich konnte anhand eines einfachen Beispiels das Potential, Ressourcen zu sparen, aufgezeigt werden.

6 Untersuchung der Ressourceneffizienz

Neben der gewonnenen Flexibilität ist das Einsparpotential bezüglich der benötigten rekonfigurierbaren Ressourcen ein wichtiges Argument für die Verwendung dynamisch rekonfigurierbarer FPGA-Systeme. Da einige Systemkomponenten zeitlich versetzt auf dieselben Systemressourcen geladen werden können, sollten insgesamt weniger FPGA-Ressourcen benötigt werden als bei einem entsprechenden statischen FPGA-System. Anders formuliert: Der zusätzliche Entwurfsaufwand eines dynamisch rekonfigurierbaren Systems ist nur dann gerechtfertigt, wenn signifikant Ressourcen gegenüber einer statischen Realisierung eingespart werden können. Aber auch wenn der Entschluss für eine dynamisch rekonfigurierbare Lösung bereits gefallen ist, stellt sich die Frage, welche der vielfältigen Realisierungsoptionen möglichst effizient mit den verfügbaren Ressourcen umgeht und möglichst wenige Ressourcen beansprucht. Um diese Frage beantworten zu können, wird im Folgenden eine Fallstudie vorgestellt, die verschiedene Platzierungsverfahren hinsichtlich ihrer Ressourceneffizienz untersucht. Hierzu wird in einem ersten Schritt betrachtet, welchen Einfluss die Kachelung auf den Ressourcenbedarf verschiedener Komponenten hat. In einem zweiten Schritt wird der Einfluss der Kommunikationsinfrastruktur auf die Ressourceneffizienz eines rekonfigurierbaren Systems untersucht. Als Effizienzmaß wird die interne Ressourcenauslastung eingeführt, in die ausschließlich statische Effekte einfließen. Dynamische Effekte, die von eventuell vorhandenen Ablaufplänen und Platzierungsstrategien abhängig sind, bleiben unberücksichtigt. Sie werden detailliert in der Dissertation von Markus Köster [Koe07] behandelt.

Die Ergebnisse der Fallstudie basieren auf realen Implementierungen beispielhaft gewählter Komponenten auf einem Virtex-II FPGA (XC2V4000). Aufgrund des sehr ähnlichen Aufbaus der heute verfügbaren FPGAs ist eine Übertragung der hier gewonnenen Erkenntnisse auf andere FPGA-Familien und weitere Systemkomponenten zulässig. In ähnlichen Untersuchungen von Kalte et al. [KPR04b] wurden die Möglichkeiten der Modulkompaktierung (PAR auf möglichst wenig FPGA-Fläche) auf Virtex-E FPGAs erörtert. Darüber hinaus wird in den folgenden Abschnitten der Einfluss von Modulform, Modulgröße und der Kommunikationsinfrastruktur auf die Ressourceneffizienz dynamischer Komponenten analysiert.

Zunächst sind jedoch einige theoretische Betrachtungen sinnvoll. Hierzu wird im folgenden Abschnitt ein Maß für die Ressourceneffizienz definiert. Darüber hinaus werden die verschiedenen Einflussfaktoren auf die Ressourceneffizienz modelliert.

6.1 Ein Effizienzmaß für die Ressourcennutzung

Bei der Bewertung der dynamischen Rekonfigurierung spielt die Frage, wie effizient die verfügbaren Ressourcen genutzt werden, eine wichtige Rolle. Als Maß hierfür wird die *interne Ressourcenauslastung* U_I eingeführt. Sie wird gebildet als Quotient der benötigten Ressourcen $\tilde{r}$ einer Komponente K zu den tatsächlich belegten Ressourcen $\bar{r}$ eines Moduls M der Komponente K:

$$U_I := \frac{\tilde{r}}{\bar{r}} \tag{19}$$

Die interne Ressourcenauslastung kann für jeden Ressourcentyp des Ziel-FPGAs (Slices, CLBs, eingebettete Multiplizierer, BlockRAM, usw.) separat angegeben werden. Da die CLBs (bzw. die Slices der CLBs) die mit Abstand häufigste und am meisten Fläche beanspruchende Ressource sind, werden sie in dieser Arbeit bevorzugt betrachtet.

6.1.1 Benötigte Ressourcen einer Komponente

Bei der Spezifizierung belegter und benötigter Logikressourcen können CLBs gut zur Veranschaulichung verwendet werden. Bei realen Implementierungen von Komponenten zu Modulen liegt tatsächlich aber eine feinere Einteilung der FPGA-Ressourcen zugrunde. Von den bis zu vier Slices eines CLBs können einige ungenutzt bleiben. Auch Slices enthalten wiederum LUTs und FFs. Diese können aufgrund der Slice-internen Routingressourcen jedoch nicht mehr völlig unabhängig voneinander genutzt werden. In dieser Arbeit werden daher Slices anstelle von CLBs für die Angabe der benötigten Ressourcen einer Komponente verwendet. Dies verhindert auch, dass Komponenten, die beispielsweise nur aus kombinatorischer Logik bestehen (und somit keine FFs benötigen) nicht per se eine schlechte interne Ressourcenauslastung aufweisen.

Um für eine Komponente, die per Definition noch nicht auf das FPGA abgebildet wurde, sondern nur durch eine Netzliste beschrieben wird, Angaben zum Slicebedarf machen zu können, müssen die Aussagen des Synthesewerkzeugs zum Ressourcenbedarf herangezogen werden. Nach der Synthese werden Zahlenwerte für die benötigten FFs ($\tilde{r}_{FF}$), LUTs ($\tilde{r}_{LUT}$), Slices ($\tilde{r}_{Slice}$) und zu weiteren Ressourcen vom Synthesewerkzeug ausgegeben. Da die FFs und LUTs der Slices beim Mapping, Platzieren und Verdrahten praktisch nie optimal ausgenutzt werden, ist bei der Angabe der benötigten Slices bereits ein zu erwartender Verschnitt mit eingerechnet. Dieser basiert jedoch auf Erfahrungswerte von statischen Entwürfen. Die Besonderheiten des partiellen Entwurfsablaufs fließen nicht mit ein. Die Angaben zu den benötigten Slices können daher nur schlecht als Ressourcenbedarf verwendet werden. Im Gegensatz dazu können die Angaben zu den benötigten FFs und LUTs als tatsächlicher Bedarf betrachtet werden[22]. Aufgrund dieser Angaben wird daher in dieser Arbeit eine eigene Maßzahl für die benötigten Slices $\tilde{r}_{Slice}$ einer Komponente definiert:

$$\tilde{r}_{Slice} = \left\lceil \frac{\max(\eta_{LUT}, \eta_{FF})}{2} \right\rceil = S_{\min} \tag{20}$$

Da bei den in dieser Arbeit betrachteten Architekturen immer je zwei FFs und LUTs pro Slice zur Verfügung stehen, wird von den benötigten LUTs und FFs die größere Anzahl durch zwei geteilt und aufgerundet. Es wird also die Anzahl von Slices berechnet, die benötigt würden, wenn alle FFs und LUTs während des Entwurfs optimal in Slices gepackt und verdrahtet werden könnten. $\tilde{r}_{Slice}$ wird daher *minimaler Slicebedarf* genannt und mit S_{min} bezeichnet.

[22] Hierbei muss ein entwurfsspezifischer Zusammenhang unbedingt beachtet werden: Die Werte benötigter FFs und LUTs ändern sich üblicherweise während des Mappings, also der Abbildung der Netzlisten auf die FPGA-Elemente, da hier noch Optimierungen, wie z.B. das Entfernen redundanter Logik, vorgenommen werden. Die benötigten FFs und LUTs einer Komponente können sich demnach von denen eines aus ihr erzeugten Moduls unterscheiden. Diese Optimierungen müssen derzeit für den partiellen Entwurfsablauf ohnehin deaktiviert werden, so dass die Anzahl der FFs und LUTs konstant bleibt.

Da die Kommunikationsinfrastruktur ein durch die dynamische Konfigurierung bedingter Mehraufwand ist, der zudem von der Größe der zu erstellenden Module abhängig ist, wird sie für die *benötigten* Ressourcen einer Komponente nicht berücksichtigt. Für die Menge der durch ein Modul *belegten* Ressourcen spielt sie insbesondere beim Vergleich verschiedener Platzierungsverfahren jedoch eine wesentliche Rolle.

6.1.2 Benutzte Ressourcen und belegte Ressourcen eines Moduls

Die Schaltungselemente einer Komponente werden erst während des Mappings und PAR auf die FPGA-Ressourcen abgebildet. Wie auch bei statischen Implementierungen ist diese Abbildung nicht optimal. Ein Teil der LUTs und FFs und weiterer Ressourcen bleiben ungenutzt und es entsteht ein Verschnitt V_{Abb}, der hier *Abbildungsverschnitt* genannt wird. Wird nur der Abbildungsverschnitt berücksichtigt, ergibt sich eine Ressourcenbenutzung $\bar{r}'$ nach der Abbildung, wie sie in einem statischen System auftritt:

$$\bar{r}' = \tilde{r} + V_{Abb} \tag{21}$$

$\bar{r}'$ gibt somit die Menge *benutzter Ressourcen* an. Es ist ebenso möglich, den Abbildungsverschnitt als eine Eigenschaft des PAR-Werkzeugs zu modellieren, die angibt, wie dicht die Schaltungselemente einer Komponente in die FPGA-Ressourcen gepackt werden können. Hierfür wurde in Kapitel 4 bereits die Packdichte *P* eingeführt, mit der gilt:

$$\bar{r}' = \tilde{r}/P \tag{22}$$

Bezogen auf die durch ein Modul benutzten Slices bedeutet eine Packdichte von 100 %, dass das Modul nicht auf weniger Slices abgebildet werden kann. Die Menge benutzter Ressourcen ist in der Praxis nur schwer greifbar. Um sie zu ermitteln, muss mit Werkzeugen wie dem FPGA-Editor der fertige Entwurf betrachtet und die belegten Slices gezählt werden. In einigen Fällen können auch die Ausgaben der Entwurfswerkzeuge hilfreich sein. In dynamisch rekonfigurierbaren Systemen können qualitative Veränderungen der Packdichte für verschiedene Platzierungsverfahren beobachtet werden, wie später noch gezeigt wird. Für Busmakros gilt der Zusammenhang (22) ebenfalls. Da die Schaltungselemente der Busmakros allerdings zum großen Teil manuell platziert und verdrahtet werden, wird oft eine Packdichte von 100 % erreicht, so dass $\bar{r}'_{Komm} = \tilde{r}_{Komm}$ angenommen werden kann[23] (vgl. Tabelle 4-3 auf Seite 77).

Dass in rekonfigurierbaren Systemen die Menge benutzter Ressourcen nicht der Menge belegter Ressourcen entspricht, liegt an zwei weiteren Faktoren: Der Kachelung und der Kommunikationsinfrastruktur. Durch die Kachelung sind starre Grenzen für die Module vorgegeben. Da immer nur ganze Kacheln durch Module belegt werden können, tritt ein weiterer Verschnitt V_{Kachel} auf, der hier *Kachelverschnitt* genannt wird. Wird beispielsweise ein Modul mit 280 benutzten Slices in eine Kachel mit 300 Slices abgebildet, sind zur Laufzeit immer 300 Slices durch das Modul belegt, da kein anderes Modul diese Kachel gleichzeitig nutzen kann. Die 20 zusätzlich benötigten Slices sind hier dementsprechend der Kachelverschnitt. Da an-

[23] Nicht verwendete LUTs und FFs in den Slices eines Busmakros können von einem Modul verwendet werden. In der Praxis konnte dies, wahrscheinlich aufgrund der hohen Packdichte der Busmakros, nur sehr selten beobachtet werden.

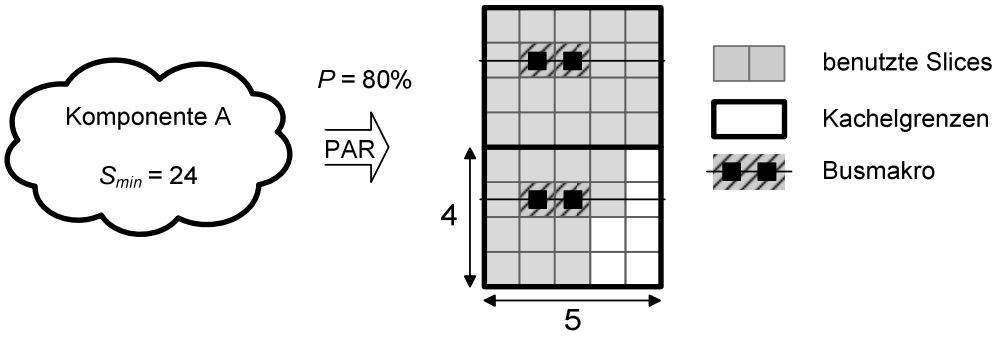

Abbildung 6-1: Abbildung einer Komponente auf zwei Kacheln mit je 20 Slices

genommen werden kann, dass zusätzliche Kacheln bei der Modulerzeugung nur dann verwendet werden, wenn sie auch tatsächlich benötigt werden, ist der Kachelverschnitt immer kleiner als eine Kachel. Es gilt also $V_{Kachel} < \bar{r}_{Kachel}$. In statischen Systemen tritt ein ähnlicher Effekt dadurch auf, dass ein die Größe eines FPGAs in der Regel nicht exakt dem Ressourcenbedarf des Systems entspricht. Der Gesamt-Verschnitt eines statischen Systems ist jedoch geringer als die Summe der Kachelverschnitte dynamischer Komponenten, wie weiter unten gezeigt wird.

Die Kommunikationsinfrastruktur reduziert die pro Kachel verfügbaren Ressourcen. Wird in obigem Beispiel ein Busmakro verwendet, das 50 Slices pro Kachel belegt, dann muss das Beispielmodul auf zwei Kacheln abgebildet werden. Zur Laufzeit werden dann durch das Modul immer 600 Slices belegt. Für die belegten Ressourcen $\bar{r}$ eines Moduls gilt somit:

$$\bar{r} = \frac{\tilde{r}}{P} + k \cdot \tilde{r}_{Komm} + V_{Kachel} \,, \tag{23}$$

wobei k die Anzahl der belegten Kacheln und $\tilde{r}_{Komm}$ die Menge der Kommunikationsressourcen pro Kachel angeben. Abbildung 6-1 veranschaulicht diesen Zusammenhang. Dargestellt ist eine Komponente A mit einem minimalen Slicebedarf von 24 Slices. Zu dieser Komponente wird ein Modul erstellt, dass auf Kacheln mit einer Ausdehnung von fünf mal vier Slices abgebildet wird. In jeder Kachel sind zwei Slices durch das verwendete Busmakro belegt. Für das PAR wird eine Packdichte von 80 % angenommen, so dass das Modul 30 Slices effektiv benutzt. Das Modul erstreckt sich somit mindestens über zwei Kacheln. In diesem Fall ergibt sich ein Kachelverschnitt von sechs Slices.

Eine äquivalente Darstellung zu Formel (23) ist

$$\bar{r} = \left\lceil \frac{\tilde{r} \,/\, P}{r_{Kachel} - \tilde{r}_{Komm}} \right\rceil \cdot \bar{r}_{Kachel} \,, \tag{24}$$

wobei eine homogene Kachelung und eine homogene Kommunikationsinfrastruktur angenommen werden. Der in Gaußschen Klammern stehende Term gibt die Anzahl benötigter Kacheln an. Multipliziert mit den Ressourcen einer Kachel kann so auf die belegten Ressourcen geschlossen werden. Der Kachelverschnitt wird in dieser Darstellung durch die Aufrundung des Klammerterms berücksichtigt. Einziger nicht bezifferbarer Parameter ist die Packdichte P bzw. der Abbildungsverschnitt. Um mithilfe der Formel (24) von den benötigten Ressourcen ohne PAR auf die Modulgröße schließen zu können, muss hier ein Schätzwert angenommen werden.

In Analogie zum minimalen Slicebedarf S_{min} werden die belegten Ressourcen $\bar{r}$ eines Moduls mit S_{bel} bezeichnet, wenn ausschließlich Slices als maßgebliche Ressource betrachtet werden.

6.1.3 Eigenschaften der internen Ressourcenauslastung

Stellt man Formel (23) nach $\tilde{r}$ um, ergibt sich:

$$\tilde{r} = P \cdot \left(\bar{r} - k \cdot \tilde{r}_{komm} - V_{Kachel} \right) \tag{25}$$

Hiermit kann die interne Ressourcenauslastung dargestellt werden als

$$U_I = \frac{\tilde{r}}{\bar{r}} = P \cdot \left(1 - \frac{k \cdot \tilde{r}_{Komm} + V_{Kachel}}{\bar{r}} \right). \tag{26}$$

Anhand dieser Darstellung können einige wesentliche Eigenschaften der internen Ressourcenauslastung deutlich gemacht werden:

1) Die interne Ressourcenauslastung kann nie besser sein als die Packdichte. Im idealen Fall (keine Kommunikationskosten, kein Kachelverschnitt) ist sie gleich der Packdichte.
2) Je geringer der Kachelverschnitt V_{Kachel}, desto besser die interne Ressourcenauslastung. Der Kachelverschnitt hängt zwar von den benutzten Ressourcen eines Moduls ab. Wegen $V_{Kachel} < \bar{r}_{Kachel}$ gilt aber trotzdem: Je kleiner die Kacheln, desto höher ist die interne Ressourcenauslastung.
3) Je geringer die Kosten für die Kommunikation $k \cdot \tilde{r}_{Komm}$, desto höher ist die interne Ressourcenauslastung. Da diese Kosten von der Anzahl der Kacheln, nicht aber von der Größe der Kacheln abhängen, können sie durch Erhöhen der Kachelgröße reduziert werden. Dies ist insofern bemerkenswert, als dass es in einem Gegensatz zu Punkt 2 steht.

6.2 Fallstudie

Um mithilfe einer Fallstudie aussagekräftige Ergebnisse erzielen zu können, wurden Komponenten ausgewählt, die im Sinne eines Co-Prozessors in einem eingebetteten System sinnvoll erscheinen. Die verwendeten IP-Cores, die als dynamische Komponenten dienen, stammen dabei aus verschiedenen Anwendungsgebieten, die durchweg rechenintensiv sind. Die Wahl der Komponenten ist dabei nicht zufällig, wohl aber willkürlich geschehen. Es gibt keine Standardanwendungen für dynamische Rekonfigurierung, auf die hier hätte zurückgegriffen werden können. Neben der Funktion wurde darauf geachtet, dass die betrachteten Komponenten ein möglichst breites Spektrum bezüglich der benötigten Ressourcen abdecken. Die Beschleunigung, die mit ihnen gegenüber einer Software-Realisierung erzielt werden kann, wurde im Einzelnen nicht ermittelt und ist auch nicht Gegenstand der Fallstudie. Lediglich am Beispiel der vollständigen Fließkomma-Einheit (*fpu_full*, s. u.) wurden die Möglichkeiten dynamisch rekonfigurierbarer Co-Prozessoren evaluiert. Die Ergebnisse, unter anderem der Beschleunigungsfaktor gegenüber einer Software-Lösung, sind in [GKP06] beschrieben.

Für die Fallstudie wurden ausschließlich frei verfügbare IP-Cores verwendet. Hauptquelle hierfür war die Internetseite OpenCores.org, die eine beträchtliche Anzahl synthetisierbarer

Hardwarebeschreibungen frei zur Verfügung stellt. Darüber hinaus wurde der Xilinx Core Generator verwendet (Version 8.1.01i_PR_8 mit 8.1i IP Update 1), mit dem einige IP-Cores automatisch als Netzliste erzeugt werden können. Die Verwendung frei zugänglicher IP-Cores ist bei Vergleichen über Benchmarks im FPGA Bereich nicht unüblich. Auch Xilinx und Altera greifen hierauf regelmäßig zurück, um allgemein anerkannte Bedingungen zu schaffen, da die verfügbaren Module in der Regel nicht für eine bestimmte Architektur optimiert sind. Dies trifft auf die Cores des CoreGenerators natürlich nicht zu. Für die Kommunikation mit den Komponenten wurde der bereits beschriebene Wishbone-Bus verwendet.

6.2.1 Der IP-Core Datensatz

Im Folgenden werden die verwendeten Cores knapp und nach Anwendungsgebiet zusammengefasst dargestellt. Für eine ausführliche Beschreibung sei auf die entsprechende Dokumentation von OpenCores.org [Lam07] und die des Xilinx Core Generators [Xil07c] verwiesen. Die Beschreibung der entsprechenden Entwurfsprojekte samt der Anbindung an den Wishbone-Bus findet sich in [Sch06D].

AES Ver-/Entschlüsselung

aes128_decrypt Entschlüsselung eines Datenstroms nach dem AES-Standard [NIS01]. Verwendet wird ein 128 Bit Schlüssel. Bei der Benutzung werden zunächst 128 Bit verschlüsselter Daten an die Komponente gesendet, die dann die Entschlüsselung durchführt. Anschließend können die entschlüsselten Daten zurückgelesen werden.

aes128_encrypt Verschlüsselung analog zur *aes128_decrypt*-Komponente.

Bildverarbeitung

bmp2jpeg Komprimierung von Bitmap-Daten nach dem JPEG-Standard. Da das komprimierte Bild lokal gespeichert wird, werden sehr viele eingebettete Speicherelemente (BlockRAMs) benötigt. Unterstützt werden Bilder mit einer Auflösung bis zu 352*288 Pixeln.

Trigonometrie

cordic_arctan Berechnung des Arkustangens mit einer Genauigkeit von 32 Bit aus gegebenen Sinus- und Kosinuswerten. Die Berechnung basiert auf dem CORDIC Verfahren, mit dem u.a. trigonometrische Berechnungen effizient in digitalen Schaltungen (insbesondere auch FPGAs) realisiert werden können [And98].

cordic_rec2polar Umformung zweier kartesischer Koordinaten (je 16 Bit) in polare Koordinaten (Winkel und Radius, je 20 Bit).

cordic_polar2rec Umformung polarer Koordinaten in kartesische Koordinaten, analog zu *cordic_rec2polar*.

cordic_sinh_cosh Berechnung des Sinus Hyperbolicus und Kosinus Hyperbolicus aus einem gegebenen Bogenmaß mit 32 Bit Genauigkeit.

Festkomma-/Fließkomma-Berechnungen

cordic_squareroot Berechnung der Quadratwurzel eines 32 Bit Werts.

divider Festpunktdivision zweier 32 Bit Werte.

fpu_full Fließkommaberechnung nach IEEE754 [IEE85] mit einfacher Genauigkeit (32 Bit: 1 Bit Vorzeichen, 8 Bit Exponent und 23 Bit Mantisse). Unterstützt Addition, Subtraktion, Multiplikation, Division, sowie Umformung von Integer- zu Fließkommazahlen und umgekehrt.

fpu_add_sub Fließkommaaddition und -subtraktion analog zu *fpu_full*.

fpu_div Fließkommadivision analog zu *fpu_full*.

fpu_mult Fließkommamultiplikation analog zu *fpu_full*.

Signalverarbeitung

fft Schnelle Fourier-Transformation (*Fast Fourier Transformation – FFT)*. Verarbeitet werden 1024 Eingangswerte mit je 12 Bit Genauigkeit in Ausgangswerte gleicher Dimension. Die Daten des Zeit- und Bildbereichs werden komplett in BlockRAM gespeichert.

In Tabelle 6-1 ist der Ressourcenbedarf $\tilde{r}$ aller Komponenten dargestellt. Angegeben ist jeweils die Anzahl benötigter Slices, Flipflops (FFs), Look-Up-Tables (LUTs), Multiplizierer (MULTs) und BlockRAMs (BRAMs). Bei den Slices ist der minimale Slicebedarf angegeben, der nach Formel (17) aus der Anzahl der FFs und LUTs bestimmt wurde. FFs und LUTs wer-

Tabelle 6-1: Ressourcenbedarf der Beispielkomponenten

Komponente	S_{min}	FFs	LUTs	MULTs	BRAMs
aes128_decrypt	2382	942	4764	-	4
aes128_encrypt	2118	931	4236	-	-
bmp2jpeg	6655	4182	13310	2	45
cordic_arctan	2065	4077	4129	-	-
cordic_rec2polar	658	1206	1315	-	-
cordic_polar2rec	459	850	917	-	-
cordic_sinh_cosh	2200	4286	4399	-	-
cordic_squareroot	811	1595	1622	-	-
divider	1729	3457	1439	-	-
fft	2676	5351	4421	18	18
fpu_full	1760	1065	3520	4	-
fpu_add_sub	449	694	897	-	-
fpu_div	780	1560	1100	-	-
fpu_mult	474	872	948	-	-

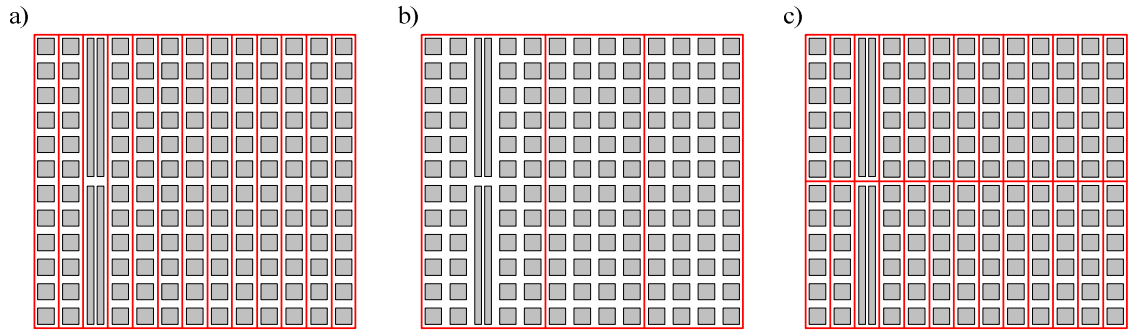

Abbildung 6-2: Die drei betrachteten freien Platzierungsverfahren: a) 1D_1x80, b) 1D_4x80, c) 1D_1x40

den daher nachfolgend nicht mehr separat betrachtet. Die Größe der Module variiert bezogen auf das verwendete FPGA von 2,1 % (449) bis 28,9 % (6655) der verfügbaren Slices. Für die Abbildung der MULTs und BRAMs stehen sechs FPGA-Spalten mit je 20 Multiplizierern und BlockRAMs zur Verfügung.

6.2.2 Verglichene Platzierungsverfahren

Bei der Festlegung der zu vergleichenden Platzierungsverfahren wurden ausschließlich solche betrachtet, die mit den derzeit verfügbaren Konfigurierungsverfahren und Entwurfsabläufen tatsächlich realisierbar sind. Insbesondere bedeutet dies auch, dass eine funktionierende Kommunikationsinfrastruktur bereitgestellt werden kann. Um festzustellen, welchen Einfluss der Formfaktor auf andere Qualitätsmerkmale der Module bzw. des Gesamtsystems hat, wurden vier verschiedene Ansätze miteinander verglichen. Das einfachste und meist verwendete Platzierungsverfahren stellen dabei *feste Modulplätze* dar. Auf Virtex-II FPGAs überspannen diese üblicherweise die gesamte Höhe des FPGAs und variieren je nach Implementierung in der Breite. Sie wird so gewählt, dass das größte Modul in den Steckplatz geladen werden kann. Um zu überprüfen, ob und wie mit freier Platzierung die FPGA-Ressourcen effizienter genutzt werden können, werden eindimensionale Platzierungsverfahren untersucht. Bei diesen Verfahren belegen Module stets die volle Höhe des FPGAs und variieren in der Breite. Sie unterscheiden sich jedoch in ihrer Granularität, also in der Breite der Kacheln. Sie beträgt beim ersten Verfahren eine CLB-Spalte, beim Zweiten vier CLB-Spalten. Das bedeutet, dass die Größe der Module beim ersten Verfahren sehr genau angepasst werden kann, während beim zweiten Verfahren immer ganze Blöcke mit vier CLB-Spalten die kleinste Einheit darstellen. Zur einfacheren Unterscheidung werden die zwei Verfahren im Folgenden mit 1D_1x80 und 1D_4x80 bezeichnet mit Bezug auf die eindimensionale Platzierung sowie die gewählte Granularität in CLB-Spalten mal CLB-Zeilen. Das Prinzip der entsprechenden Kachelungen ist in Abbildung 6-2 dargestellt, wobei die Höhe der Kacheln aus Darstellungsgründen nicht die genannten 40 bzw. 80 CLBs beträgt. Herauszuheben ist die Einbindung der BlockRAM- und Multiplizierer-Spalten. Sie stellen bei 1D_1x80 einen eigenen Kacheltyp dar, der wie alle anderen Kacheln frei genutzt werden kann, so dass es bei 1D_1x80 genau zwei Kacheltypen gibt: CLB-Spalten und BlockRAM/Multiplizierer-Spalten. Bei 1D_4x80 sind BlockRAMs und Multiplizierer immer in einen Viererblock von CLB-Spalten eingebettet. Aufgrund der Anordnung der BlockRAMs/Multiplizierer auf Virtex-II FGPAs konnte hier eine Kachelung gefunden werden, die ebenfalls nur zwei verschiedene Kacheltypen aufweist:

reine CLB-Blöcke und solche mit einer BlockRAM/Multiplizierer-Spalte in der Mitte. Um den Vergleich der verschiedenen Ansätze zu erleichtern, wird im Folgenden immer die Breite der Module bzw. Kacheln in CLB-Spalten angegeben. Unter Umständen vorhandene BlockRAM/Multiplizierer-Spalten werden nicht mitgezählt.

Ein Kritikpunkt an eindimensionaler Platzierung ist, dass insbesondere kleine Module ein sehr ungünstiges Seitenverhältnis aufweisen. Bei den gerade genannten Verfahren auf einem XC2V4000 FPGA sind sie immer 80 CLBs hoch, aber nur wenige CLBs breit. Um zu untersuchen, ob dies zu einer ungünstigen oder gar unmöglichen Platzierung und Verdrahtung führt, wurde ein weiteres Verfahren implementiert, das mit 1D_1x40 bezeichnet wird. Hierbei strecken sich die Kacheln jeweils nur über die Hälfte des FPGA, so dass gegenüber 1D_1x80 Module entstehen können, die nur die halbe Höhe aber in etwa die doppelte Breite aufweisen. Das Seitenverhältnis verringert sich dadurch etwas, auch wenn es gegenüber etwa einer quadratischen Form immer noch ungünstig erscheint. Allerdings kann für 1D_1x40 prinzipiell eine funktionierende Kommunikationsinfrastruktur bereitgestellt werden, was schließlich den Ausschlag für die Untersuchung dieser Kachelung gab. Streng genommen handelt es sich bei 1D_1x40 um ein zweidimensionales Platzierungsverfahren. Da die Platzierung der Module in der Vertikalen im Vergleich zur Horizontalen sehr stark eingeschränkt ist, wird es als quasi-1D betrachtet und auch entsprechend bezeichnet.

In den folgenden Abschnitten wird zunächst untersucht, wie effizient die genannten Komponenten für die betrachteten Platzierungsverfahren implementiert werden können. Der Fokus liegt dabei auf dem Einfluss der Modulgröße und -form auf die erreichbare interne Ressourcenauslastung. Anschließend wird die Kommunikationsinfrastruktur mit in die Betrachtungen einbezogen. Schlussendlich wird für einen ausgewählten Fall ein Vergleich mit einer statischen Implementierung der Komponenten durchgeführt, bei dem keine Beschränkungen bezüglich der Platzierung gemacht werden, der allerdings auch keine Flexibilität zur Laufzeit erlaubt. Hierdurch können die Kosten bzw. der Nutzen der partiellen Konfigurierung dargestellt werden.

6.3 Einfluss der Kachelung auf die Ressourcenauslastung

Um eine Abschätzung der internen Ressourcenauslastung machen zu können, wurden alle Testkomponenten für jedes der oben beschriebenen Platzierungsverfahren zumindest einmal implementiert, d.h. es wurde mindestens ein funktionierendes Modul erstellt. Bei den Komponenten, bei denen aufgrund der Heterogenität zwei oder mehr Module erzeugt werden können, wird jeweils nur das Modul mit dem geringsten Ressourcenbedarf betrachtet. Bei allen Implementierungen wird zunächst keine vollständige Kommunikationsinfrastruktur verwendet. Somit kann der Einfluss der Platzierungsverfahren, insbesondere die damit Verbundenen Moduleigenschaften Größe und Form, auf den Ressourcenbedarf betrachtet werden. Für die Entwurfswerkzeuge, die keine unverbundenen Moduleingänge erlauben, wird dafür ein Pseudo-Busmakro verwendet. Die Slices, aus denen dieses Makro besteht, stellen wie gewohnt Slice-Ausgänge als Anschlusspunkte für die Moduleingänge zur Verfügung. Alle anderen Elemente eines Busmakros, wie Register, Logikverknüpfungen und vor allem die Signalverbindungen zu benachbarten Kacheln, sind nicht enthalten. Der Ressourcenbedarf dieses Mak-

Tabelle 6-2: Anzahl der belegten Slices der erzeugten Module (Formfaktor in CLBs)

Komponente	1D_1x80	1D_4x80	1D_1x40	FM_17x80
aes128_decrypt	3840 (12x80)	3840 (12x80)	3680 (23x40)	5440 (17x80)
aes128_encrypt	5440 (17x80)	6400 (20x80)	3200 (20x40)	5440 (17x80)
bmp2jpeg	6720 (21x80)	7680 (24x80)	8480 (53x40)	- -
cordic_arctan	3200 (10x80)	3840 (12x80)	2720 (17x40)	5440 (17x80)
cordic_rec2polar	1280 (4x80)	1280 (4x80)	960 (6x40)	5440 (17x80)
cordic_polar2rec	960 (3x80)	1280 (4x80)	800 (5x40)	5440 (17x80)
cordic_sinh_cosh	2880 (9x80)	3840 (12x80)	3040 (19x40)	5440 (17x80)
cordic_squareroot	1600 (5x80)	2560 (8x80)	1440 (9x40)	5440 (17x80)
divider	3520 (11x80)	3840 (12x80)	2400 (15x40)	5440 (17x80)
fft	4480 (14x80)	5120 (16x80)	- -	- -
fpu_full	2560 (8x80)	2560 (8x80)	2720 (17x40)	5440 (17x80)
fpu_add_sub	1280 (4x80)	1280 (4x80)	960 (6x40)	5440 (17x80)
fpu_div	1600 (5x80)	2560 (8x80)	2240 (14x40)	5440 (17x80)
fpu_mult	960 (3x80)	1280 (4x80)	1120 (7x40)	5440 (17x80)

ros ist dadurch minimal, es werden lediglich 76 LUTs benötigt. Es wird darüber hinaus nur einmal pro Modul eingebunden, nicht einmal pro Kachel, wie es sonst üblich ist.

In Tabelle 6-2 ist die Menge der belegten Slices für die vier betrachteten Platzierungsverfahren aufgezeigt. Für die beiden 1D-Ansätze, die die volle Höhe des FPGAs beanspruchen, ergibt sich eine Höhe von 80 CLB-Reihen und eine Breite, die in Schritten von einer bzw. vier CLB-Spalten angepasst werden kann (1D_1x80 bzw. 1D_4x80). Dementsprechend ergeben sich für den dritten Ansatz (halbe FPGA-Höhe) eine Kachelhöhe von 40 CLBs und eine Kachelbreite von einer CLB-Spalte (1D_1x40). Zur Veranschaulichung der Modulausmaße ist in Tabelle 6-2 der Formfaktor der Module (Breite mal Höhe) in CLB-Spalten bzw. -Zeilen in Klammern angegeben.

Für den Ansatz mit festen Modulplätzen muss die Modulplatzgröße an die Anforderungen der verwendeten Komponenten angepasst werden, d.h., sie müssen so groß gewählt sein, dass auch für die Komponente mit dem größten Ressourcenbedarf ausreichend Ressourcen bereit stehen. Dies gilt konkret sowohl für die benutzten Slices (FFs, LUTs), wie auch für BlockRAM und eingebettete Multiplizierer. Daher wurden bei dem hier umgesetzten Verfahren mit festen Modulplätzen die beiden größten Module (*bmp2jpeg* und *fft*) nicht berücksichtigt und stattdessen die nächst kleinere Komponente (*aes128_encrypt*) als Maß verwendet. Der hohe Bedarf an BlockRAM und Multiplizierern der beiden größten Komponenten würde zwei komplette BlockRAM/Multiplizierer-Spalten pro Modulplatz erfordern, so dass auf dem XC2V4000-FPGA nur zwei Modulplätze eingerichtet werden könnten. Für alle anderen Komponenten reicht hier eine Spalte, so dass vier feste Modulplätze verwendet werden können. Aufgrund der minimalen Breite der *aes128_encrypt*-Komponente von 17 Spalten, die

sich aus dem 1D_1x80-Ansatz ergibt, sind die festen Modulplätze für alle Komponenten 17 Spalten breit.

Bei der *fft*-Komponente war es nicht möglich, ein funktionierendes Modul für den 1D_1x40 Ansatz zu erzeugen. Selbst bei größtmöglicher Breite konnte das PAR nicht alle Signale verdrahten. Die Gründe des Scheiterns sind nicht bekannt. Es kann nur vermutet werden, dass auch hier die eingebetteten BlockRAMs und Multiplizierer die Ursache darstellen, da die Anzahl der verfügbaren Slices die der benötigten um ein Vielfaches überstieg.

6.3.1 Erzielte interne Ressourcenauslastungen der Fallstudie

Die interne Ressourcenauslastung (siehe 6.1) ist ein Maß dafür, wie effizient eine Komponente auf die vorhandenen Ressourcen abgebildet werden kann. Hierbei wird der minimale Slicebedarf, also die Größe eines Moduls bei optimaler Packdichte, mit den tatsächlich belegten Ressourcen in Relation gesetzt. In Abbildung 6-3 ist die interne Ressourcenauslastung, also der Quotient aus benötigten Slices (Tabelle 6-1) zu den belegten Slices (Tabelle 6-2), für alle Komponenten und die vier Platzierungsverfahren dargestellt. Aufgeschlüsselt nach Segmentierungsansätzen liegt sie im Mittel bei 53,0 % (1D_1x80), 44,9 % (1D_4x80), 57,6 % (1D_1x40) und 23,8 % (FM_17x80). Die niedrigen Werte weisen darauf hin, dass beim Mappen, Platzieren und Verdrahten die FFs und LUTs nicht optimal in die verfügbaren Slices gepackt werden können. Erklärt werden kann dies dadurch, dass der Verdrahtungsaufwand nicht gemessen wird und nicht in die Berechnung des minimalen Slicebedarfs mit einfließt. Die Synthesewerkzeuge von Xilinx schätzen den Verdrahtungsaufwand nach der Synthese mit ab und berechnen daher einen höheren Wert für die benötigten Slices. Wird dieser Schätzwert

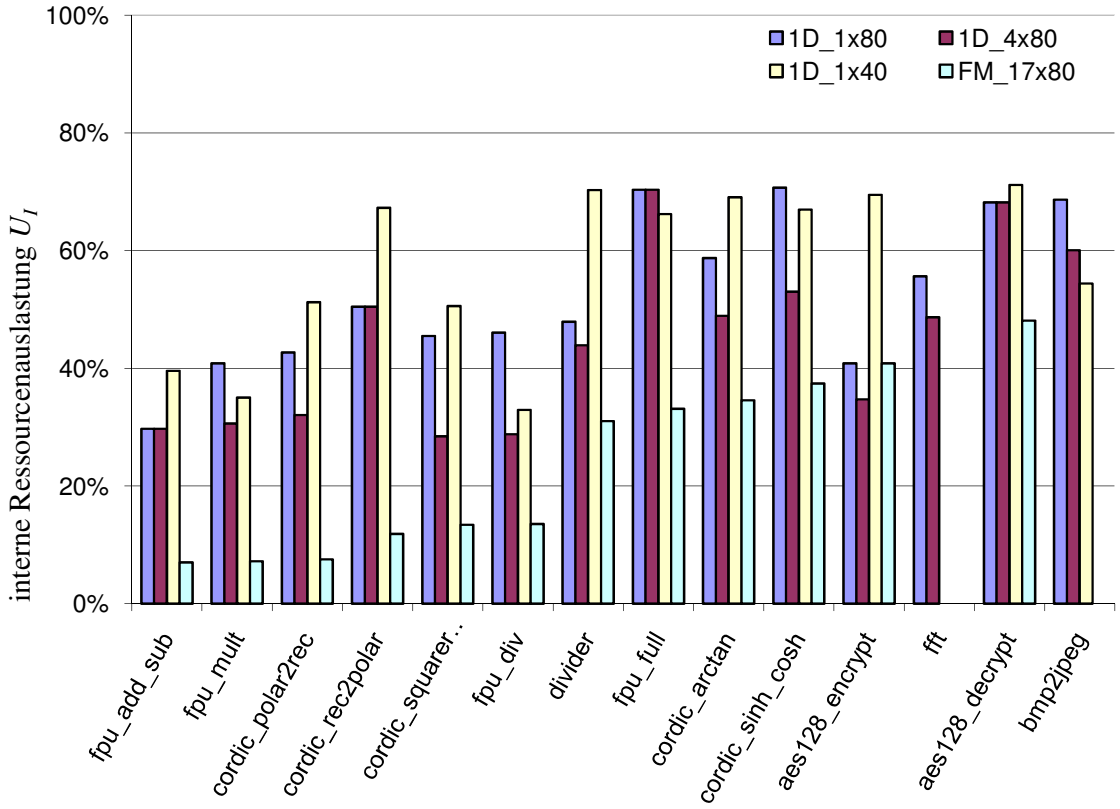

Abbildung 6-3: Interne Ressourcenauslastung der Module

anstelle des minimalen Slicebedarfs für die Berechnung der internen Ressourcenauslastung verwendet, ergeben sich hierfür wesentlich höhere Zahlen. Diese können im Einzelfall sogar 100 % übersteigen, was dann eine zu konservative Schätzung anzeigt. Da die Heuristik, mit der die Anzahl benötigter Slices abgeschätzt wird, nicht bekannt ist und somit die resultierenden Werte schlecht oder gar nicht vergleichbar sind, wird hier ausschließlich der minimale Slicebedarf verwendet.

Auffällig bei der Betrachtung aller Werte ist, dass es eine Schwelle bei etwa 70 % interner Ressourcenauslastung gibt, die in keinem Fall übertreten wird. Eine optimale Packung, bei der alle FFs oder LUTs der belegten Slices wirklich verwendet werden, scheint mit den verwendeten Mapping- und PAR-Tools nicht erreichbar zu sein. Betrachtet man dagegen jedes Slice, von dem mindestens ein FF oder LUT verwendet wird, als benutzt und berechnet so die Ressourcenauslastung als Quotient der benutzten zu belegten Slices, so sind Werte bis annähernd 100 % Ressourcenauslastung nicht selten.

Da hier die Module ohne Kommunikationsinfrastruktur betrachtet werden, wird die Ressourcenauslastung nach dem oben gegebenen Modell nur noch durch den Kachelverschnitt und die Packdichte beeinflusst. Im Folgenden wird zunächst untersucht, wie sich die Kachelgranularität und somit der Kachelverschnitt auf die Ressourcenauslastung auswirkt. Danach wird untersucht, ob die Packdichte durch die Größe und Form der Module beeinflusst wird.

6.3.2 Einfluss der Kachelgranularität

Betrachtet man die Durchschnittswerte der internen Ressourcenauslastung, bestätigt sich die naheliegende Vermutung, dass die Ressourcenauslastung steigt, wenn die Kachelung feiner wird. Mit kleinen Kacheln kann die Modulfläche genauer an die Mindestgröße angepasst werden und der Kachelverschnitt sinkt. Allerdings sind die Einzelwerte starken Schwankungen unterworfen. Vergleicht man 1D_1x80[24] mit 1D_1x40, so ist die interne Ressourcenauslastung der feiner unterteilten 1D_1x40 Segmentierung im Mittel um 4,8 %-Punkte besser. Die Standardabweichung der vorliegenden Messungen von diesem Wert beträgt 12,4 %-Punkte, so dass es auf den Einzelfall ankommt, welche Segmentierung bessere Ergebnisse liefert. Von den 13 betrachteten Komponenten (ohne die *fft*), hat in fünf Fällen der gröbere 1D_1x80 Ansatz eine höhere Ressourcenauslastung.

Vergleicht man dagegen 1D_1x80 mit 1D_4x80, beträgt die Standardabweichung nur etwa 6,7 %-Punkte von im Mittel 8,2 %-Punkten Unterschied in der Ressourcenauslastung. Insbesondere gibt es hier keinen Fall, in dem die gröbere Einteilung die Feinere übertrifft. Da alle Kacheln des 1D_4x80 Ansatzes aus den 1D_1x80-Kacheln zusammengesetzt werden können, ist ein anderes Ergebnis hier allerdings auch kaum vorstellbar.

Vergleicht man schließlich die 1D_4x80 Segmentierung, die unter den freien Platzierungsverfahren die geringste Ressourcenauslastung hat, mit den festen Modulplätzen, werden die Stärken der freien Platzierung deutlich. Die interne Ressourcenauslastung ist bei 1D_4x80 im Schnitt fast doppelt so hoch wie die der festen Modulplätze. Wären die beiden größten Komponenten bei der Festlegung der Modulplatzgröße mit berücksichtigt worden, fiele der Vergleich noch deutlicher zugunsten der freien Platzierung aus. Wie zu erwarten, ist die Ressour-

[24] Unter Vernachlässigung der *fft*, die für 1D_1x40 nicht implementiert werden konnte, ergibt sich eine mittlere interne Ressourcenauslastung von 52,8%.

cenauslastung der festen Modulplätze bei kleinen Komponenten (wie hier *cordic_polar2rec*, *fpu_mult*, *fpu_add_sub*) besonders gering. Aber selbst im besten Fall beträgt sie nur 48,1 %. Das liegt daran, dass die Komponente *aes128_encrypt*, nach der die Größe der festen Modulplätze bestimmt wurde, für kein Platzierungsverfahren effizient platziert und verdrahtet werden konnte.

6.3.3 Einfluss der Komponentengröße

In Abbildung 6-3 sind die Komponenten entsprechend ihres minimalen Slicebedarfs in aufsteigender Reihenfolge abgebildet, also von der kleinsten Komponente *fpu_add_sub* bis zur Größten *bmp2jpeg*. Es ist ein Trend erkennbar, dass große Komponenten effizienter abgebildet werden können, da sie eine höhere interne Ressourcenauslastung aufweisen. Die mittlere interne Ressourcenauslastung der sechs kleinsten Komponenten liegt für die 1D_1x80 und 1D_1x40 Segmentierungen mit 42,5 % und 46,1 % jeweils mehr als 10 %-Punkte unter dem entsprechenden Gesamtmittel. Die sechs größten Komponenten (ohne *fft*) liegen mit 62,9 % und 66,2 % mittlerer interner Ressourcenauslastung 10 %-Punkte über dem jeweiligen Gesamtmittel.

Ein Grund hierfür ist sicherlich durch die Granularität der Kacheln gegeben. Da die Kacheln nicht beliebig klein sein können, kann die Größe der Module nicht beliebig genau angepasst werden. Der hierdurch verursachte Kachelverschnitt wirkt sich bei kleinen Modulen prozentual stärker auf die interne Ressourcenauslastung aus als bei großen Modulen. Im Folgenden wird gezeigt, dass dieser Effekt nicht der einzige Grund für die geringere interne Ressourcenauslastung kleiner Module ist. Hierfür wird zunächst angenommen, dass die Packdichte aller Module – ohne Berücksichtigung der Kachelung – identisch ist. Tabelle 6-3 zeigt die sechs kleinsten Komponenten und den aus ihren Ressourcenanforderungen berechneten minimalen Slicebedarf S_{min}. Für jede Komponente sind unter der Rubrik *Reale Implementierung* die implementierten Module für den 1D_1x80- und den 1D_1x40 Ansatz mit Höhe H und Breite B sowie mit der daraus resultierenden Anzahl belegter Slices S_{bel} erneut abgebildet (vgl. Tabelle 6-2). Wie bereits erwähnt ergeben sich für diese Implementierungen mittlere interne Auslastungen der sechs betrachteten Module von 42,5 % und 46,1 %. Geht man davon aus, dass diese Module mit der gleichen internen Ressourcenauslastung wie die großen Komponenten abgebildet werden können (62,9 % bzw. 66,2 %), ergeben sich die Zahlen, die unter der Rubrik *Ideale Prognose* abgebildet sind. Die Angaben zu den belegten Slices S_{bel} wurden hier aus dem minimalen Slicebedarf unter Annahme der soeben genannten höheren Ressourcenauslastung berechnet und aufgerundet. Aus den gleich bleibenden Kachelhöhen von 80 CLBs bzw. 40 CLBs wurde dann die erforderliche Modulbreite berechnet. Sie wurde ebenfalls aufgerundet, da nur ganze Kacheln verwendet werden können. Vergleicht man nun die Modulbreiten der realen Implementierungen mit denen der Prognose, stellt man fest, dass für neun der zwölf betrachteten Module theoretisch eine kleinere Fläche ausreichend sein sollte, wenn die Packdichte aller Module in etwa gleich ist. Praktisch ließen sich die Module jedoch nicht mit einer geringeren Breite implementieren.

Dieser Effekt kann mit den unterschiedlichen Möglichkeiten beim Platzieren und Verdrahten der Module erklärt werden. Aufgrund der hohen Anzahl von verfügbaren Ressourcen auch schon bei kleinen Modulen spricht nichts dafür, dass das Platzieren Probleme bereiten könnte. Mit anderen Worten: Wenige Schaltungselemente lassen sich auf wenige Ressourcen ebenso

Tabelle 6-3: Prognose der Modulgrößen bei gleichbleibender Packdichte

		Reale Implementierung						Ideale Prognose					
		1D_1x80 $\overline{U}_I = 42{,}5\%$			1D_1x40 $\overline{U}_I = 46{,}1\%$			1D_1x80 $\overline{U}_I = 62{,}9\%$			1D_1x40 $\overline{U}_I = 66{,}2\%$		
	S_{min}	S_{bel}	h	b	S_{bel}	h	b	S_{bel}	h	b	S_{bel}	h	b
cordic_polar2rec	410	960	80	3	800	40	5	652	80	3	620	40	4
cordic_rec2polar	646	1280	80	4	960	40	6	1027	80	4	976	40	7
cordic_squareroot	728	1600	80	9	1440	40	9	1158	80	4	1100	40	7
fpu_add_sub	380	1280	80	4	960	40	6	605	80	2	575	40	4
fpu_div	737	1600	80	5	2240	40	14	1172	80	4	1114	40	7
fpu_mult	392	960	80	3	1120	40	7	624	80	2	593	40	4

gut abbilden, wie viele Elemente auf viele Ressourcen. Die Anzahl der verfügbaren Routingressourcen (siehe 2.1.2) nimmt jedoch bei kleiner werdender Modulfläche überproportional ab. Geht man davon aus, dass die vorhandenen Long Lines und Tristate Lines beim Verdrahten nicht verwendet werden, dann machen Double Lines, Hex Lines und Direct Connections jeweils etwa ein Drittel der verfügbaren Routingressourcen aus. Sobald eine solche Routingressource nicht vollständig in einem Modul liegt, sie also die Modulgrenzen kreuzt, können zumindest die außerhalb liegenden Enden dieser Leitung nicht mehr verwendet werden. Da diese Leitungen zusätzlich nur von einem Punkt aus getrieben werden können, nehmen die Routingmöglichkeiten am Rand der Module ab, wovon bei kleinen Modulen schnell die gesamte Fläche betroffen sein kann. Die schmalsten hier implementierten Module (*fpu_mult* sowie *cordic_polar2rec*) haben im 1D_1x80 Ansatz eine Breite von drei CLB-Spalten. Dadurch können keine horizontalen Hex Lines verwendet werden. Umgekehrt werden bei schmalen Modulen überproportional viele vertikale Ressourcen beansprucht. Es liegt also die Vermutung nahe, dass bei kleinen Modulen die verfügbaren Routingressourcen maßgebend für die Modulgröße sind.

Eine weitere Bestätigung dieser These findet sich bei der Betrachtung der Fehlermeldungen der Entwurfswerkzeuge. Die minimale Modulgröße muss beim Entwurf durch ein Versuch-und-Fehler-Verfahren mit mehreren, iterativen PAR-Durchläufen ermittelt werden. Hierzu wird zunächst eine angestrebte Modulgröße angegeben und ein PAR durchgeführt. Kann das Modul erfolgreich platziert und verdrahtet werden, wird versucht, das Modul mit der nächst-kleineren Modulgröße zu implementieren. Kann es nicht platziert und verdrahtet werden, wird entsprechend ein PAR mit der nächst-größeren Modulgröße durchgeführt. Auf diese Weise wird für jedes Modul die minimale Modulgröße ermittelt. Auch bei der Erstellung der hier betrachteten Module wurde auf diese Weise für jedes Modul (und jeden Platzierungsansatz) mindestens eine erfolgreiche, aber auch mindestens eine fehlgeschlagene Implementierung durchgeführt[25]. Bei der Betrachtung der durch die Entwurfswerkzeuge gene-

[25] Ausnahme hiervon sind die Implementierungen kleiner Module für den 1D_4x80 Ansatz sowie die meisten Implementierungen zu den festen Modulplätzen, die beim ersten Versuch erfolgreich waren und nicht weiter eingeschränkt werden konnten.

rierten Fehlermeldungen konnte beobachtet werden, dass die Implementierung großer Module an der Platzierung, die der kleinen Module an der Verdrahtung scheiterten. Es kann daher gefolgert werden, dass die Packdichte bei kleinen Modulen aufgrund der schlechteren Routingmöglichkeiten abnimmt.

6.3.4 Einfluss des Formfaktors

Geht man davon aus, dass in den Randzonen der Module schlechter geroutet werden kann, dann liegt die Vermutung nahe, dass neben der Größe der Module auch ihre Form Einfluss auf die Ressourcenauslastung hat. Definiert man den Formfaktor F der Module als

$$F = \begin{cases} h/b, \text{falls } h \geq b \\ b/h, \text{falls } b > h \end{cases} \tag{27}$$

wobei h und b Höhe bzw. Breite der Module bezeichnen, dann führt ein großer Formfaktor zu relativ vielen Randzonen. In Abbildung 6-4 ist die Ressourcenauslastung aller Module der 1D_1x80 und 1D_1x40 Segmentierungen über ihrem Formfaktor aufgetragen. Zusätzlich sind die Datenpunkte nach dem minimalen Slicebedarf der Module in vier Gruppen unterteilt.

Betrachtet man zunächst die kleinen Module mit einem minimalen Slicebedarf von bis zu 1000 Slices, ist kaum eine Verschlechterung der internen Ressourcenauslastung bei sehr großem Formfaktor zu erkennen. Mit einer Ausnahme liegen die Module dieser Größenkategorie jedoch alle im unteren Bereich der Ressourcenauslastung zwischen 29,7 % und 51,3 %. Insbesondere weist das Modul mit dem geringsten Formfaktor dieser Gruppe (2,86) die insgesamt zweitschlechteste Ressourcenauslastung (32,9 %) auf.

Etwas aussagekräftiger sind die Ergebnisse der Module der zwei nächst größeren Katego-

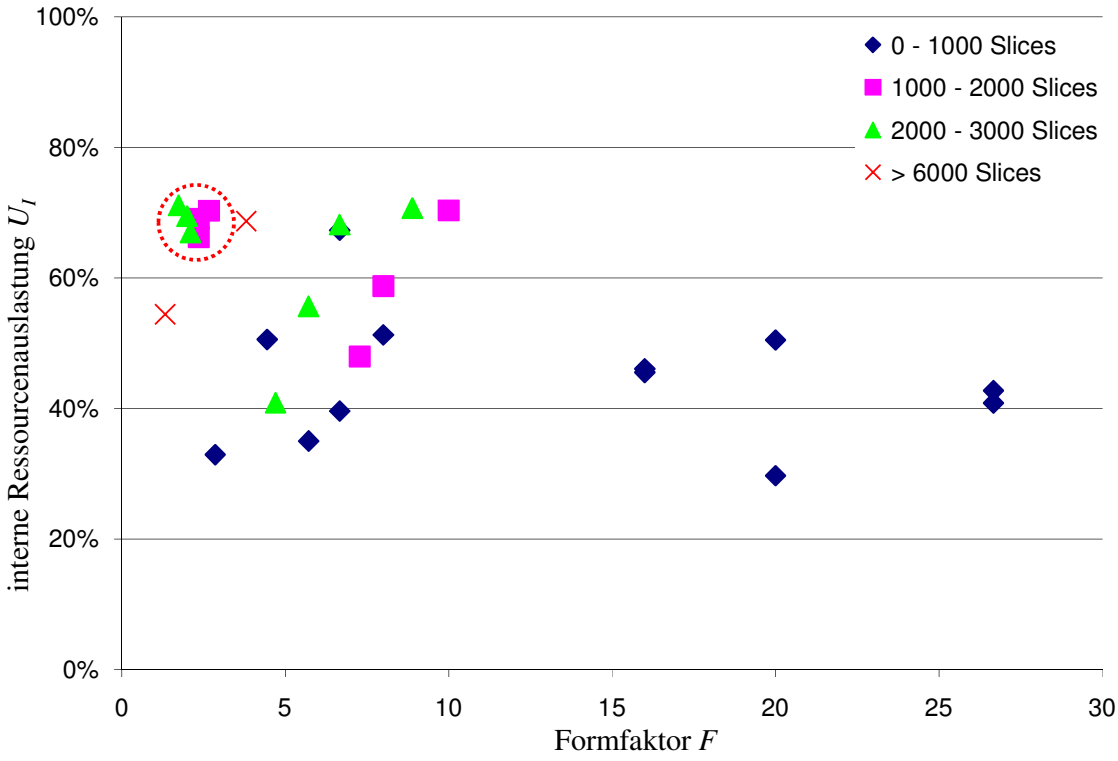

Abbildung 6-4: Ressourcenauslastung der Module in Abhängigkeit ihres Formfaktors

Tabelle 6-4: Abhängigkeit der Ressourcenauslastung von der Position des Kommunikationsmakros

		Makro in der Mitte			Makro am unteren Rand			
	S_{min}	*b*	S_{bel}	U_I	*b*	S_{bel}	U_I	*Diff.*
aes128_decrypt	2618	12	3840	68,2 %	16	5120	51,1 %	17,0 %
aes128_encrypt	2223	17	5440	40,9 %	16	5120	43,4 %	-2,6 %
bmp2jpeg	4615	21	6720	68,7 %	22	7040	65,6 %	3,1 %
cordic_arctan	1879	10	3200	58,7 %	8	2560	73,4 %	-14,7 %
cordic_squareroot	728	5	1600	45,5 %	8	2560	28,4 %	17,1 %
divider	1687	11	3520	47,9 %	10	3200	52,7 %	-4,8 %
fpu_mult	392	3	960	40,8 %	5	4480	24,5 %	16,3 %

rien (1000-2000 Slices und 2000-3000 Slices). Alle Module für den 1D_1x40-Ansatz liegen sowohl was ihren Formfaktor betrifft, als auch bei der internen Ressourcenauslastung sehr nah beieinander. Diese Gruppe ist in Abbildung 6-4 durch einen Kreis markiert. Die entsprechenden Module für den 1D_1x80-Ansatz haben erwartungsgemäß einen höheren Formfaktor, weisen außerdem aber deutlich unterschiedliche Ressourcenauslastungen auf. Lediglich drei Module erreichen Werte um 70 %, die anderen vier (hier inklusive der *fft*) liegen zwischen 40 % und 60 %.

Die Komponente *bmp2jpeg*, die deutlich größer als alle anderen Komponenten und hier der einzige Vertreter der Gruppe mit mehr als 6000 Slices ist, hat bei der 1D_1x80 Implementierung eine gute Ressourcenauslastung. Ihr 1D_1x40 Modul, das als einziges eine größere Breite als Höhe aufweist, schneidet eher mittelmäßig ab. Hier ist der außergewöhnlich hohe Bedarf an BlockRAM (45 BlockRAMs) maßgebend für die Modulgröße und bestimmt somit die interne Ressourcenauslastung. Um hier sichere Aussagen machen zu können, müssen jedoch weitere Module mit ähnlichen Anforderungen untersucht werden.

Aus den hier vorliegenden Daten kann somit kein direkter Zusammenhang zwischen dem Formfaktor eines Moduls und der internen Ressourcenauslastung abgeleitet werden. Allerdings führt ein großer Formfaktor zu einem größeren Anteil von Randzonen in einem Modul. Die Verminderten Routingmöglichkeiten in diesen Zonen können zu einer insgesamt schlechteren internen Ressourcenauslastung führen. Dieser Effekt konnte hier allerdings nur bei wenigen Komponenten mit Größen zwischen 1000 und 3000 Slices beobachtet werden. Kleinere Komponenten haben ohnehin einen sehr hohen Anteil von Randzonen, so dass der Formfaktor hier keinen wesentlichen Einfluss hat. Große Komponenten können aufgrund der begrenzten Größe des verwendeten FPGAs keine großen Formfaktoren aufweisen.

6.3.5 Weitere Einflüsse

Die Ergebnisse des PAR der Module sind nicht nur von der Struktur der Module und ihrer Form abhängig. Ein weiterer Faktor sind die beim Platzieren und Verdrahten verwendeten Algorithmen. Diese sind von den Herstellern der Entwurfswerkzeuge verständlicherweise nicht offengelegt, es kann aber vermutet werden, dass das PAR mit einer pseudozufälligen Anfangsverteilung beginnt. Bei den hier verwendeten Beispielkomponenten wurden bei unterschiedlichen Durchläufen der Werkzeuge verschiedene Ergebnisse erzielt. Es kann somit

nicht garantiert werden, dass eine erfolgreiche Platzierung eines Moduls in eine vorgegebene Fläche reproduziert werden kann. Dieses Phänomen wurde zwar selten beobachtet, muss bei der Bewertung der Ergebnisse aber doch berücksichtigt werden.

Ebenfalls Einfluss auf die Ergebnisse des PAR hat die Lage inhomogener Elemente in den Modulpositionen. Hierzu gehören die eingebetteten Elemente des FPGAs wie BlockRAM- und Multiplizierer-Spalten ebenso wie die Kommunikationsinfrastruktur. Eine BlockRAM-Spalte am rechten Rand eines Moduls führt zu anderen Platzierungs- und Verdrahtungsbedingungen wie eine BlockRAM-Spalte am linken Rand. Bei den in dieser Arbeit betrachteten, horizontalen Kommunikationsinfrastrukturen hat deren vertikale Position innerhalb der Kacheln einen Einfluss auf das PAR. Um dies zu zeigen, wurde der 1D_1x80-Ansatz zweimal implementiert. Bei der ersten Implementierung wurde das Pseudo-Busmakro, das sich über 47 CLB-Zeilen erstreckt, vertikal zentriert platziert. Beim zweiten Durchlauf wurde es an den unteren Rand gelegt. Die dabei erzielte minimale Breite der Module und die entsprechenden Werte für die interne Ressourcenauslastung sind in Tabelle 6-4 dargestellt. Aufgelistet sind nur die sieben Komponenten, für die sich ein Unterschied ergab. Er ist als Differenz der jeweils erreichten internen Ressourcenauslastung in Prozentpunkten dargestellt. Mit bis zu 17,1 %-Punkten variieren die Ergebnisse nicht unerheblich. Eine Korrelation mit der Größe der Module ist nicht erkennbar. Ebenso wenig kann von den zwei Platzierungsvarianten eine eindeutig bessere ausgemacht werden.

6.4 Einfluss der Kommunikationsinfrastruktur

Wie im vorangegangenen Abschnitt gezeigt wurde, nutzen Module die FPGA-Ressourcen umso effizienter, je kleiner die zugrunde liegenden Kacheln sind. Betrachtet wurden dabei die erzielbare Packdichte bei der Abbildung eines Moduls auf die rekonfigurierbaren Ressourcen und der Einfluss des Kachelverschnitts. Durch die Einbeziehung der Kommunikationsinfrastruktur stehen einige Ressourcen der Kacheln nicht mehr für die Abbildung der Module zur Verfügung. Wie viele Ressourcen davon betroffen sind, hängt von zwei Faktoren ab: Von der Komplexität des verwendeten Busses (Anzahl der Daten-/Adressleitungen, dedizierte Signale, etc.) und von der Anzahl der Anschlusspunkte (Ports) des erstellten Busmakros. Beide Faktoren wurden in den vorangehenden Kapiteln eingehend diskutiert.

Im Folgenden wird auf Grundlage des in Abschnitt 6.1 vorgestellten Modells eine Abschätzung gemacht, welchen Einfluss die Kommunikationsinfrastruktur auf den Ressourcenbedarf eines Systems hat. Betrachtet werden wiederum die in Abschnitt 6.2 beschriebenen Beispielmodule für zwei freie Platzierungsverfahren (1D_1x80 und 1D_4x80) und für feste Modulplätze. Für die Kommunikation soll das in Kapitel 4 vorgestellte, Tristate-basierte Busmakro verwendet werden. Es bildet einen Wishbone-Bus ab, und stellt in seiner Komplexität (228 Slices pro Anschlusspunkt[26]) somit ein realistisches Beispiel dar. Um eine größtmögliche Homogenität der Kacheln und somit eine größtmögliche Freiheit bei der Modulplatzierung zu erreichen, wird festgelegt, dass in jeder Kachel ein Anschlusspunkt des Busmakros existiert. Es wird außerdem angenommen, dass alle Module mit einer Packdichte von 70 % abgebildet werden können. Unter diesen Voraussetzungen kann nach Formel 24 aus dem mi-

[26] Der zusätzliche Aufwand bei Kacheln mit BlockRAM und Multiplizierern wird hier nicht betrachtet.

nimalen Slicebedarf S_{min}, der die benötigten Ressourcen darstellt, und aus der erwarteten Packdichte P auf die Menge der durch ein Modul belegten Slices S_{bel} geschlossen werden:

$$S_{bel} = \left\lceil \frac{S_{\min} / P}{\bar{r}_{Kachel} - \bar{r}_{Komm}} \right\rceil \cdot \bar{r}_{Kachel} \tag{28}$$

Die pro Kachel verfügbaren Slices $\bar{r}_{Kachel}$ können unter Kenntnis der Kachelhöhe H_{Kachel} und der Kachelbreite B_{Kachel} bestimmt werden:

$$\bar{r}_{Kachel} = H_{Kachel} \cdot B_{Kachel} \cdot 4 \tag{29}$$

Als Einheit für die Kachelausdehnung werden üblicherweise CLBs verwendet, so dass das Produkt aus Kachelhöhe und -breite noch mit 4 (bei vier Slices pro CLB für Virtex-II FPGAs) multipliziert wird. In gleicher Weise können die belegten Slices durch die Höhe h und Breite b eines Moduls mit

$$S_{bel} = h \cdot b \cdot 4 \tag{30}$$

dargestellt werden. Berücksichtigt man nun noch, dass bei eindimensionalen Platzierungsverfahren immer $h = H_{Kachel}$ gilt, und setzt man (29) und (30) in (28) ein, ergibt sich:

$$B = \left\lceil \frac{\frac{S_{\min}}{P}}{H_{Kachel} \cdot B_{Kachel} \cdot 4 - \bar{r}_{Komm}} \right\rceil \cdot B_{Kachel} \tag{31}$$

Abbildung 6-5 zeigt die nach (31) prognostizierten Modulbreiten für alle Komponenten. Als Breite der festen Modulplätze wurden 13 CLB-Spalten festgelegt. Die Komponente *bmp2jpeg* wurde dabei nicht berücksichtigt, da hier weiterhin die Anzahl benötigter BlockRAMs die Modulbreite bestimmt. In der Abbildung ist die Breite des *bmp2jpeg*-Moduls mit 26 CLB-Spalten angegeben, was zwei Modulplätzen entspricht. Nach der Definition fester Modulplätze ist dies zwar nicht zulässig, erlaubt aber die Betrachtung des FM_16x80-Ansatzes als freies Platzierungsverfahren mit grobgranularer Kachelung.

Für die Überprüfung des Modells wurden die Komponenten für den 1D_4x80 Ansatz implementiert. Die tatsächlich erzielten Modulbreiten sind ebenfalls in Abbildung 6-5 dargestellt. Hierzu sei erwähnt, dass die Komponenten *fft* und *bmp2jpeg* mit den zur Verfügung stehenden Xilinx Werkzeugen nicht erfolgreich implementiert werden konnten. Die Abweichung der tatsächlichen Kachelgrößen bei 1D_4x80 von den geschätzten Werten beträgt maximal eine Kachel, also vier CLB-Spalten bzw. 1280 Slices. Interessanterweise weichen zwei dieser Fälle nach unten ab, d.h. die Module wurden kleiner als erwartet. Hier wurde eine interne Ressourcenauslastung höher als die erwarteten 70 % erreicht. Tatsächlich liegt sie bei der Komponente *divider* bei 85 %. Die Größe der Modulplätze bei FM_13x80 wurde der prognostizierten Breite der *aes128_decrypt* Komponente angepasst, die den größten minimalen Slicebedarf hat (*fft* und *bmp2jpeg* wurden wie in Abschnitt 6.3 aufgrund der besonderen Anforderungen an BlockRAM und Multiplizierern für den Feste-Modulplätze-Ansatz nicht betrachtet). Für eine tatsächliche Implementierung erweist sie sich nicht als ausreichend, da *aes128_decrypt* einen 16 Spalten großen Modulplatz benötigt.

Nimmt man für alle Prognosen eine ähnliche Güte an, zeigt sich der Einfluss der Kommunikationsinfrastruktur deutlich. War bei den Implementierungen in Abschnitt 6.3 noch der 1D_1x80 Ansatz in allen Fällen optimal, so ist er bei Berücksichtigung der Kommunikationsinfrastruktur ganz offensichtlich die schlechteste Wahl. Der 1D_4x80 Ansatz ist in allen Fällen besser und auch feste Modulplätze stellen für fast jedes Modul die Bessere oder eine gleichwertige Wahl dar. Darüber hinaus sind für kleine Module geringere Packdichten als die hier angenommenen 70 % zu erwarten, was zu noch schlechteren internen Ressourcenauslastungen der kleinen Module führt.

Eine Modulplatzierung mit festen Modulplätzen ist erwartungsgemäß da am besten, wo aufgrund der Modulgröße ein geringer Kachelverschnitt auftritt. Hier können feste Modulplätze in Einzelfällen sogar die freien Platzierungsverfahren übertreffen, da sie weniger Ressourcen für die Kommunikationsinfrastruktur benötigen. Bei den kleinen Komponenten tritt jedoch naturgemäß ein hoher Kachelverschnitt auf, wie er auch schon in Abschnitt 6.3.1 festgestellt wurde. Unter Umständen können bei festen Modulplätzen aber weitere Ressourcen eingespart werden. Wie in Kapitel 4 gezeigt wurde, werden die komplexen eingebetteten Busmakros vor allem bei freier Platzierung benötigt. Für feste Modulplätze können in der Regel Kantenmakros verwendet werden, die wesentlich weniger Ressourcen benötigen. Verwendet man hierfür das in 4.5.2 untersuchte Xilinx Kantenmakro, werden pro Kachel nun 87 anstatt 214 Slices für die Kommunikationsinfrastruktur benötigt. Aufgrund der Größe der

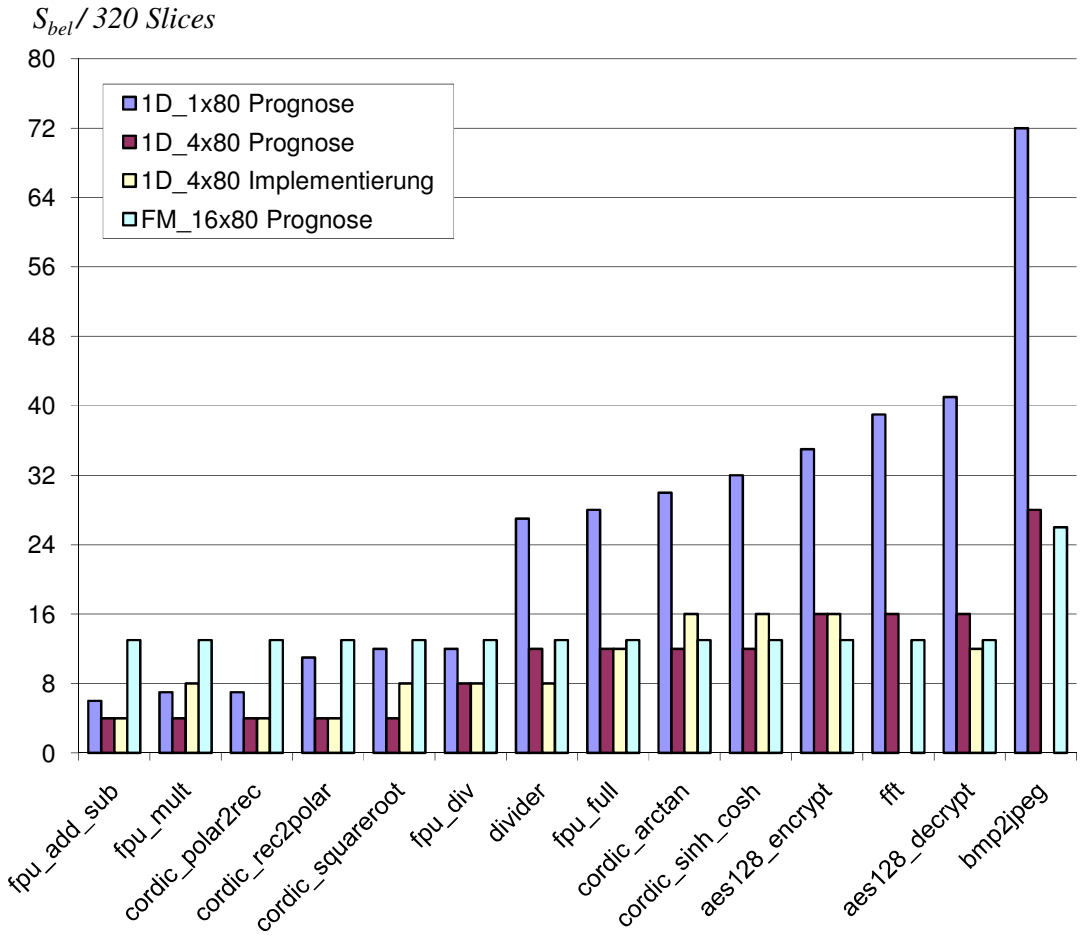

Abbildung 6-5: Erwartete Modulgrößen für verschiedene Platzierungsverfahren

festen Modulplätze (4160 Slices) hätte die Einsparung von 127 Slices jedoch kaum Einfluss.

Die Wahl eines geeigneten Platzierungsverfahren und somit die Wahl der Kachelgrößen ist stark abhängig von den dynamischen Komponenten. Es muss ein Kompromiss gefunden werden, der den Kachelverschnitt bei kleinen Komponenten und den Anteil nicht genutzter Ressourcen für die Kommunikationsinfrastruktur bei großen Komponenten gering hält. Wenn möglich, müssen auch dynamische Effekte betrachtet werden. Komponenten, die häufig verwendet werden, haben einen größeren Einfluss auf die gesamte Ressourcenauslastung eines rekonfigurierbaren Systems als selten genutzte Komponenten.

6.5 Bewertung verschiedener Implementierungen

In der Einleitung dieser Arbeit wurden zwei wesentliche Gründe für die Verwendung dynamischer Rekonfigurierung genannt: Zum einen die Möglichkeit, zur Laufzeit ein System an veränderte Rahmenbedingungen anzupassen, zum anderen einen verminderten Ressourcenbedarf durch eine zeitversetzte Nutzung der Ressourcen. In den letzten beiden Abschnitten wurde gezeigt, dass die Kachelung und die Kommunikationsinfrastruktur in dynamisch rekonfigurierbaren Systemen einen nicht unerheblichen Einfluss auf den Ressourcenbedarf haben, den es in statischen Systemen nicht gibt. Im Folgenden wird anhand eines Beispiels gezeigt, in welchen Situationen statische bzw. dynamische Systeme bezüglich des Ressourcenbedarfs die bessere Wahl darstellen.

Um den Ressourcenbedarf eines statischen Systems bestimmen zu können, müssen beim Entwurf alle Systemkomponenten bekannt sein, die zur Laufzeit des Systems gebraucht werden. Eine statische Implementierung muss alle Komponenten beinhalten, da sie während des Betriebs nicht angepasst werden kann. Die Größe des FPGAs muss dann entsprechend den Anforderungen dieser rein statischen Hardware ausgewählt werden. Um einen fairen Vergleich zu erzielen, muss auch für das statische System eine möglichst hohe Packdichte bzw. interne Ressourcenauslastung angestrebt werden. Als Ziel-FPGA wurde auch hier das bislang stets verwendete XC2V4000 genutzt. Da es nicht alle Beispielkomponenten gleichzeitig fassen kann, wurden neun Komponenten[27] ausgewählt, mit denen die höchste Ressourcenauslastung des FPGAs erzielt wurde. Sie decken das vorhandene Größenspektrum gut ab, es gibt etwa gleich viele kleine, mittlere und große Komponenten. Die größten Komponenten *bmp2jpeg* und *fft* sind allerdings nicht dabei, da sie für den 1D_4x80 Ansatz nicht implementiert werden konnten (s.o.). Der statische Entwurf belegt damit 20291 der 23040 verfügbaren Slices. An ihm werden nun ein Verfahren mit festen Modulplätzen (FM_16x80) und ein freies, eindimensionales Platzierungsverfahren mit einer Kachelbreite von vier CLB-Spalten gemessen. Wie im vorangehenden Abschnitt gezeigt wurde stellen diese zwei Verfahren für die Verwendung eines Wishbone-Busmakros sinnvolle Realisierungsoptionen dar. Die Modulgrößen wurden durch Implementierungen ermittelt. Verfahren mit noch feineren Kachelungen als der 1D_4x80 (z.B. 1D_1x80 oder 1D_1x40) wurden nicht betrachtet, weil sie entweder nicht realisierbar sind oder weil die Modulgrößen von der benötigten Kommunikationsinfrastruktur dominiert würden.

---

[27] a*es128_decrypt, aes128_encrypt, cordic_arctan, cordic_sinh_cosh, cordic_squareroot, divider, fpu_add_sub, fpu_div, fpu_mult*

6.5.1 Einsparpotential durch dynamische Rekonfigurierung

Systeme mit dynamisch rekonfigurierbaren Ressourcen können nach verschiedenen Maßstäben ausgelegt werden. Ist neben den Modulen und dem Platzierungsverfahren ein Ablaufplan bekannt, nach dem die Komponenten in das System geladen werden sollen, kann der tatsächliche Ressourcenbedarf bestimmt werden. Er kann im einfachsten Fall (feste Modulplätze) anhand der maximalen Anzahl gleichzeitig im System befindlicher Komponenten bestimmt werden. Bei freien Platzierungsverfahren spielt die Platzierungsstrategie und eine mögliche Fragmentierung der Ressourcen zur Laufzeit ebenfalls eine Rolle. Hier kann dann mithilfe eines Simulators eine gültige Ausprägung der rekonfigurierbaren Bereiche gefunden werden. Geht man allerdings davon aus, dass die rekonfigurierbaren Bereiche defragmentiert werden können und dass jede Komponente durch entsprechende Module an jede beliebige Stelle geladen werden kann, dann ist der Ressourcenbedarf unter Kenntnis der Modulbreiten und des Ablaufplans auch ohne Simulation ermittelbar. Im Gegensatz zu festen Modulplätzen ist hier aber nicht nur wichtig, welche Komponenten zu einem Zeitpunkt gleichzeitig benötigt werden, sondern auch, wie groß die entsprechenden Module sind.

Das Einsparpotential der dynamischen Rekonfigurierung gegenüber einem statischen System hängt folglich im Wesentlichen davon ab, wie viele Komponenten zur Laufzeit gleichzeitig in das System geladen werden sollen oder müssen. Bei freier Platzierung kommt es zudem darauf an, welche Komponenten jeweils gleichzeitig geladen werden. Sofern kein Ablaufplan für die verwendeten Komponenten bekannt ist, kann sowohl der *Best Case* (BC), also immer nur die kleinsten Komponenten werden gleichzeitig geladen, als auch der *Worst Case* (WC) betrachtet werden.

Abbildung 6-6 zeigt den Vergleich der statischen Implementierung mit den dynamisch rekonfigurierbaren Systemen mit 1D_4x80 bzw. FM_16x80 Platzierung. Aufgetragen ist die Menge benötigter Slices in Abhängigkeit von der Anzahl gleichzeitig benötigter Komponenten. Für alle Implementierungen sind neben den Komponenten auch die benötigten Ressourcen für die Kommunikationsbridge, den Arbiter und den Dekoder berücksichtigt. Die Werte der statischen Implementierung sind konstant, da in jedem Fall alle Komponenten Ressourcen beanspruchen, unabhängig davon, wie viele tatsächlich benötigt werden. Die sich ergebende waagerechte Gerade markiert gleichzeitig die Größe des verwendeten FPGAs, die auch die anderen Verfahren in der Praxis nicht überschreiten können.

Wie zu erwarten war, sind die dynamisch rekonfigurierbaren Systeme bei nur wenigen gleichzeitig benötigten Komponenten besser als die statische Implementierung. Ein System mit einem festen Modulplatz ist nur etwa ein Drittel so groß wie die das statische System. Schon bei vier festen Modulplätzen können jedoch kaum noch Ressourcen gegenüber einem statischen System eingespart werden. Im schlimmsten Fall, in dem alle neun Komponenten gleichzeitig gebraucht werden, würden auch ein FPGA der doppelten Größe des hier verwendeten nicht alle Modulplätze fassen. Bei freier Platzierung können im schlimmsten Fall (WC) auch nur die vier größten Module auf das vorhandene FPGA geladen werden. Für bis zu drei geladene Komponenten ist der Ressourcenbedarf identisch mit dem der freien Modulplätze. Dies ist darin begründet, dass es hier drei Komponenten gibt, deren Module für die freie Platzierung mit 16 CLB-Spalten genauso breit wie ein Modulplatz sind. Falls also eine Mindestanzahl geladener Module gegeben ist, wäre in diesem Fall durch die freie Platzierung kein wesentlicher Vorteil gegenüber festen Modulplätzen gegeben. Ist stattdessen eine bestimmte

FPGA-Fläche für dynamische Komponenten bereitgestellt, z.B. der gesamte FPGA, können im besten Fall (BC) bis zu sieben Module (statt vier) geladen werden.

Der durch die partielle Rekonfigurierung bedingte, zusätzliche Ressourcenbedarf kann sehr gut an den Datenpunkten für neun gleichzeitig benötigte Komponenten abgelesen werden. Das freie Platzierungsverfahren benötigt mit 32640 Slices 41,7 % mehr als die statische Implementierung. Bei festen Modulplätzen werden sogar 108,3 % mehr Ressourcen benötigt. Dieser zusätzliche Bedarf ist der *strukturelle Mehraufwand* partieller Rekonfigurierung, der durch den Kachelverschnitt und den Mehraufwand für die Kommunikationsinfrastruktur hervorgerufen wird. In der Praxis existiert zusätzlich der *dynamische Mehraufwand*, der durch die Fragmentierung der rekonfigurierbaren Ressourcen (nicht bei festen Modulplätzen) bedingt wird.

6.5.2 Dimensionierung der rekonfigurierbaren Bereiche

Abbildung 6-6 zeigt die Werte tatsächlicher Implementierungen. Sie ist für die Wahl einer geeigneten Implementierung insofern ungeeignet, als dass alle betrachteten Implementierungsvarianten zunächst realisiert werden müssen. Verwendet man stattdessen modellbasierte Werte, wie in Abschnitt 6.4 gezeigt, kann dieselbe Darstellungsform verwendet werden, um eine geeignete Implementierung zu finden und gegebenenfalls die rekonfigurierbaren Bereiche zu Dimensionieren. Auch hier muss je nach existierenden Randbedingungen unterschiedlich vorgegangen werden. Ist eine Mindestanzahl gleichzeitig geladener Komponenten bekannt, können die wahrscheinlich auftretenden Kosten für die modellierten Varianten verglichen und die günstigste ausgewählt werden. Ist umgekehrt die Anzahl verfügbarer Ressourcen bekannt, kann durch Eintragen einer horizontalen Geraden an der entsprechenden Stelle die

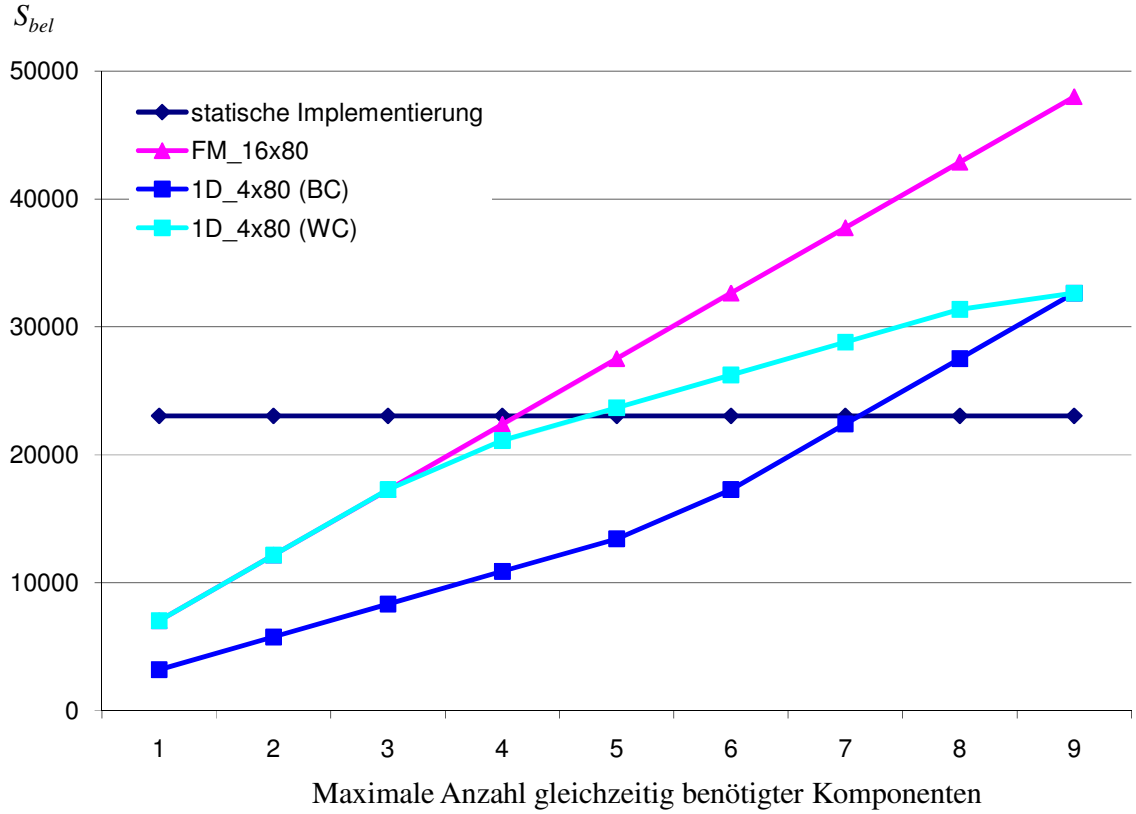

Abbildung 6-6: Dynamische Rekonfigurierung vs. Statische Implementierung

Zahl der gleichzeitig ladbaren Komponenten miteinander verglichen werden. Existieren für die verfügbaren Ressourcen und die Anzahl gleichzeitig geladener Komponenten Vorgaben, können mithilfe des Graphen die möglichen Implementierungsvarianten ermittelt werden. In jedem Fall sollten hier nicht berücksichtigte Kosten (Speicherbedarf für Module, Ressourcenbedarf für Konfigurationsmanager, Modul-/Komponentenverwaltung, Entwurfsaufwand, usw.) mit in den Entscheidungsprozess einfließen.

6.5.3 Dimensionierung der Kommunikationsinfrastruktur

Um den strukturellen Mehraufwand dynamisch rekonfigurierbarer Systeme zu verringern, muss vor allem der hohe Ressourcenbedarf der Kommunikationsinfrastruktur reduziert werden. Eine Möglichkeit ist die Reduzierung der Komplexität des Busmakros, z.B. durch Einsparen von Daten- und Adressleitungen. Da dann auch die Leistungsfähigkeit des Busses vermindert wird, wird diese Möglichkeit in der Regel keine Realisierungsoption darstellen. In Kapitel 4 wurde gezeigt, dass die hohen Kosten der Kommunikationsinfrastruktur im Wesentlichen durch dedizierte Signale verursacht werden, die mithilfe eingebetteter Punkt-zu-Punkt-Makros realisiert werden. Für eindimensionale Platzierungsverfahren können diese Signale mit Ressourcen-schonenden Kantenmakros realisiert werden, wenn alle Kacheln direkt vom statischen Bereich aus erreichbar sind.

Die bei weitem günstigste Option wäre eine von den rekonfigurierbaren Ressourcen unabhängig realisierte Kommunikationsinfrastruktur auf dem FPGA. Diese würde nicht nur wesentlich weniger Chipfläche benötigen, sondern auch keinen Einfluss mehr auf die Homogenität der rekonfigurierbaren Flächen haben. Da dies aber mit einem erheblichen Entwicklungsaufwand seitens der FPGA-Hersteller bedeuten würde, kann diese Option vorläufig nur hypothetisch betrachtet werden.

6.6 Zusammenfassung

Die in diesem Kapitel durchgeführten Untersuchungen zeigen, dass der Ressourcenbedarf einer Komponente in allen untersuchten dynamisch rekonfigurierbaren Systemen deutlich höher ist als in statischen Realisierungen. Es konnte gezeigt werden, dass die Kachelung der rekonfigurierbaren Bereiche einen wesentlichen Einfluss auf den strukturellen Mehraufwand gegenüber statischen Systemen hat. Es können zwei entgegengesetzt wirkende Effekte beobachtet werden. Geht man davon aus, dass aus Gründen der Homogenität jede Kachel einen Anschlusspunkt an die Kommunikationsinfrastruktur enthält, sinkt die Anzahl der frei verfügbaren Ressourcen, wenn die Anzahl der Kacheln in einem rekonfigurierbaren Bereich erhöht wird. Wird also eine fein-granulare Kachelung gewählt, um etwa zur Laufzeit eine möglichst große Entscheidungsfreiheit bezüglich der Platzierung der Module zu erhalten, sinkt dadurch die Ressourceneffizienz des entstehenden Systems. Erhöht man dagegen die Kachelgröße – was im Extremfall zu festen Modulplätzen führt – steigt der Kachelverschnitt, da die Modulgröße mit steigender Kachelgröße schlechter an den tatsächlichen Ressourcenbedarf einer Komponente angepasst werden kann. Kleine Module weisen bei den hier durchgeführten Untersuchungen eine verminderte Packdichte auf. Dies kann mit den verminderten Routingmöglichkeiten am Rand der Module erklärt werden. Durch den relativ hohen Anteil an Randflächen sind kleine Module besonders von diesem Effekt betroffen. Hinweise darauf,

dass die Form der Module zusätzlich einen Einfluss auf die Packdichte und somit die Ressourceneffizienz hat, konnten nicht gefunden werden.

Im Vergleich mit einem statischen System konnte anhand einer Auswahl von neun benötigten Systemkomponenten gezeigt werden, wie hoch die strukturellen Kosten der dynamischen Rekonfigurierung sind. Ein freies 1D-Platzierungsverfahren mit einer Kachelbreite von vier CLBs wies im Vergleich mehrerer dynamisch rekonfigurierbarer Implementierungen für die gegebenen Komponenten die größte Ressourceneffizienz auf. Im Vergleich mit der statischen Implementierung bringt es dennoch nur dann Vorteile, wenn zur Laufzeit nie mehr als vier der neun Komponenten gleichzeitig benötigt werden. In diesem Fall erweist sich ein System mit festen Modulplätzen als gleichwertig, das darüber hinaus wesentlich leichter zu realisieren ist.

Verallgemeinert man die Ergebnisse dieser Fallstudie, so zeigt sich, dass die Wahl einer geeigneten Systemarchitektur sehr stark anwendungsabhängig ist. Falls die zur Laufzeit benötigten Systemkomponenten beim Systementwurf bekannt sind, stellen statische Architekturen in einigen Fällen die bessere Lösung dar. Erst wenn hinreichend viele Systemkomponenten existieren, die nicht permanent benötigt werden und von denen zudem zu jeder Zeit nur ein kleiner Teil benötigt wird, können mit Rekonfigurierungsverfahren Ressourcen eingespart werden. Auch in diesem Fall müssen besondere Bedingungen erfüllt sein, um mit freier Platzierung Vorteile gegenüber festen Modulplätzen zu haben. Durch die hohen Kosten der Kommunikationsinfrastruktur lohnt der Einsatz freier Platzierung immer dann, wenn Module mit stark unterschiedlichen Größen existieren. Nur wenn die Kosten der Kommunikationsinfrastruktur in Zukunft gesenkt oder sogar eliminiert werden können, werden die freien Platzierungsverfahren den Vorteil der flexiblen Ressourcenauslastung voll ausspielen können.

7 Zusammenfassung und Fazit

In dieser Arbeit wurden die Möglichkeiten der dynamischen Rekonfigurierung auf aktuellen FPGAs untersucht. Bei der Betrachtung der Realisierungsoptionen wurde aufgezeigt, wie dynamische Systemkomponenten beim Entwurf auf die FPGA-Ressourcen abgebildet werden können und wie sie zur Laufzeit auf den dann verfügbaren Ressourcen platziert werden können. Der Schwerpunkt lag dabei auf der Untersuchung und Realisierung freier Platzierungsverfahren. Im Gegensatz zu Verfahren mit festen Modulplätzen werden die rekonfigurierbaren Bereiche eines FPGAs bei freier Platzierung in Kacheln unterteilt. Dabei stellt eine Kachel die kleinste Ressourceneinheit dar, die für die Abbildung einer dynamischen Systemkomponente verwendet werden kann. Je kleiner die Kacheln gewählt werden, und je feiner damit die Kachelung eines rekonfigurierbaren Bereichs ist, desto vielfältiger werden die Möglichkeiten, Systemkomponenten zur Laufzeit zu platzieren. Das Resultat ist eine Vielzahl möglicher Modulformen und -positionen bei der Nutzung freier Platzierungsverfahren, durch die man sich eine flexiblere und effizientere Nutzung der rekonfigurierbaren Ressourcen verspricht.

Bei der Betrachtung der heutigen Möglichkeiten, freie Platzierungsverfahren umzusetzen, wurde die Forderung nach möglichst homogenen rekonfigurierbaren Bereichen aufgestellt, um die Anzahl der benötigten Bitströme und den damit verbundenen Entwurfsaufwand in einem realisierbaren Rahmen zu halten. Diese Forderung betrifft vor allem den Aufbau der Kommunikationsinfrastruktur, die sich dem Aufbau des verwendeten FPGA anpassen muss, um keine zusätzliche Inhomogenität in das System zu bringen. Es wurden daher homogene Kommunikationsinfrastrukturen auf Basis eingebetteter Busmakros entwickelt, die die Realisierung von freien ein- und zweidimensionalen Platzierungsverfahren mit homogenen Kachelungen ermöglichen. Es konnte gezeigt werden, dass gegenüber statischen Systemen keine wesentlichen Performanzeinbußen hingenommen werden müssen. Allerdings führen besonders die dedizierten Signale, die zu jeder möglichen Modulposition separat geführt werden müssen, zu einem erheblichen zusätzlichen Ressourcenbedarf.

Um belastbare Aussagen über die Ressourceneffizienz verschiedener Platzierungsverfahren machen zu können, wurde eine Fallstudie angefertigt, in der vier Platzierungsverfahren miteinander verglichen wurden. In der Fallstudie wurden 14 Systemkomponenten unterschiedlicher Komplexität ausgewählt und für jedes Platzierungsverfahren je einmal mit und ohne Kommunikationsinfrastruktur implementiert. Durch die Analyse des resultierenden Ressourcenbedarfs konnten Aussagen über den Einfluss von Modulgröße, Modulform, Granularität der Kachelung sowie der Kommunikationsinfrastruktur gemacht werden. Es wurde festgestellt, dass nicht etwa die Modulform, wohl aber die Modulgröße Einfluss auf die Ressourceneffizienz eines Moduls hat. In den Randzonen eines Moduls kann ein Teil der Routingressourcen nicht genutzt werden. Da kleine Module einen besonders hohen Anteil an Randflächen aufweisen, brauchen sie in der Regel unverhältnismäßig viele FPGA-Ressourcen. Als entscheidend erwies sich jedoch der Einfluss der Kommunikationsinfrastruktur. Ihr Ressourcenbedarf steigt mit einer kleiner werdenden Kachelgröße. Daher dominiert bei feingranularen Platzierungsverfahren die Kommunikationsinfrastruktur den Ressourcenbedarf des gesamten Systems. Der Vorteil eines geringeren Verschnitts gegenüber grobgranularen Platzie-

rungsverfahren oder festen Modulplätzen wird dadurch zum Teil aufgehoben. Im Vergleich zu statischen Implementierungen lohnt sich der Einsatz dynamischer Rekonfigurierung immer dann, wenn zu jeder Zeit nur wenige der dynamischen Komponenten gleichzeitig benötigt werden. Freie Platzierungsverfahren haben immer dann einen Vorteil gegenüber festen Modulplätzen, wenn viele dynamische Systemkomponenten mit stark unterschiedlichen Ressourcenanforderungen zur Laufzeit in das System geladen werden sollen.

Für die Umsetzung und Analyse aller betrachteten Rekonfigurierungsverfahren wurde in dieser Arbeit eine Entwicklungsumgebung für dynamisch rekonfigurierbare FPGA-Systeme geschaffen. Sie besteht aus einer Evaluierungsplattform und einem Entwurfsablauf mit entsprechenden Entwurfswerkzeugen. Die Evaluierungsplattform basiert auf dem RAPTOR2000-System, auf dem bis zu sechs FPGAs für dynamische Rekonfigurierung verwendet werden können. Für die Untersuchungen im Rahmen dieser Arbeit wurden statische und dynamische Systemkomponenten auf getrennten FPGAs untergebracht. Hierdurch konnten große Flächen eines FPGAs für dynamische Rekonfigurierung ausgewiesen werden, was insbesondere für die Untersuchung freier Platzierungsverfahren notwendig war. Die statischen Systemkomponenten verfügen über eine Vielzahl von Schnittstellen für die Dateneingabe und -ausgabe sowie für Debugging- und Monitoring-Zwecke. Sie können direkt oder über den Host-PC des RAPTOR2000-Systems angesprochen werden. Für den Aufbau der für dynamische Rekonfigurierung zuständigen Systemkomponenten wurde das Schichtenmodell PALMERA vorgestellt. Es unterteilt die für eine Konfigurierung benötigten Aufgaben in verschiedene Schichten und erreicht dadurch eine schrittweise Abstrahierung von der rekonfigurierbaren Hardware bis zur Anwendung. Durch diese Strukturierung wird die Umsetzung der Aufgaben vereinfacht und die Wiederverwendung bestehender Komponenten deutlich erleichtert.

Für den Entwurf eines dynamisch rekonfigurierbaren Systems wurde der INDRA-Entwurfsablauf vorgestellt. Mit INDRA werden bestehende Entwurfsabläufe erweitert, so dass insbesondere der Entwurf von Systemen mit freier Platzierung weitgehend automatisiert durchgeführt werden kann. Die wichtigsten der hierzu entwickelten Entwurfswerkzeuge sind der Busmakro-Generator X-BBG und das Backend MiDesires. Ohne sie hätten die in dieser Arbeit betrachteten freien Platzierungsverfahren aufgrund ihrer Komplexität kaum realisiert werden können. Neben ihnen ist der Simulator SARA ein wichtiges Hilfsmittel beim Entwurf. Er ermöglicht die Simulation verschiedener Anwendungsszenarien und kann schon in einem frühen Entwurfsstadium wertvolle Hinweise auf die Größe der benötigten dynamischen Bereiche geben und verschiedene Platzierungsverfahren miteinander vergleichen.

Als Beispielanwendung wurde in dieser Arbeit ein System realisiert, in dem dynamisch rekonfigurierbare Ressourcen in ein eingebettetes Linux-Betriebssystem integriert wurden. Durch die konsequente Anwendung des PALMERA-Schichtenmodells wurden die vielen mit dynamischer Rekonfigurierung verbundenen Aufgaben von der Anwendungssoftware getrennt. Für die Anwendungen, die über Treiber oder Bibliotheksaufrufe auf die rekonfigurierbare Hardware zugreifen, existiert somit kein Unterschied zwischen dynamischen und statischen Systemkomponenten. In einem weiteren Beispiel wurde gezeigt, wie digitale Regelungen auf dynamisch rekonfigurierbarer Hardware realisiert werden können. Für zwei Betriebszustände eines inversen Pendels wurde je ein Regler als dynamische Komponente implemen-

tiert. Je nach Betriebszustand wurde der erforderliche Regler für das Aufschwingen oder Balancieren des Pendels dynamisch geladen.

7.1 Fazit

Dass die heute verfügbaren FPGAs für dynamische Rekonfigurierung genutzt werden können, wurde in dieser Arbeit ausgiebig demonstriert. Die Realisierung von Systemen mit einer weitgehend freien und somit flexiblen Nutzung der Ressourcen ist technisch und seitens des Entwurfs machbar. Allerdings muss hierzu festgestellt werden, dass der hohe Realisierungsaufwand nicht mit dem statischer Systeme vergleichbar ist. Auch wenn mit den in dieser Arbeit erstellten Entwicklungswerkzeugen die prototypische Realisierung rekonfigurierbarer Systeme dramatisch vereinfacht wurde, müssen noch weitere Anstrengungen unternommen werden, um einen reibungslosen Entwurfsablauf zur Verfügung zu stellen. Hier besteht vor allem seitens der FPGA-Hersteller noch ein großer Entwicklungsbedarf, damit dynamisch rekonfigurierbare Systeme einmal so einfach wie statische Systeme entwickelt werden können. Ebenso wichtig wie die fehlerfreie Funktion der Entwurfswerkzeuge ist dabei eine ergonomische Benutzerschnittstelle.

Aufgrund der Untersuchungen zur Ressourceneffizienz muss die flexible Nutzung der Ressourcen mit freien Platzierungsverfahren auf aktuellen FPGAs kritisch hinterfragt werden. Die Integration der Kommunikationsinfrastruktur in die rekonfigurierbaren Bereiche eines FPGAs kostet mit den derzeit bekannten Verfahren bei weitem zu viele Ressourcen. Besonders der Erhalt der Homogenität führt hier zu einem hohen Ressourcenbedarf. Dadurch geht der theoretische Vorteil gegenüber festen Modulplätzen in vielen Fällen verloren, und auch statische Systeme weisen nicht automatisch einen höheren Ressourcenbedarf auf. Dieses Problem könnte ebenfalls durch die FPGA-Hersteller gelöst werden, wenn für einen kleinen Teil der Routingressourcen separate Frames des Konfigurationsspeichers verwendet würden. Mit diesen Ressourcen könnten dann die Kommunikationsmakros realisiert werden, die dadurch nicht mehr Teil der Module wären und somit auch nicht der Forderung nach Homogenität unterliegen würden. Auch die Einbettung einer statischen Kommunikationsinfrastruktur als zusätzliche Routingressource ist denkbar. Die Kachelung eines freien Platzierungsverfahrens wäre damit allerdings schon weitgehend vorgegeben.

Der Einsatz freier Platzierungsverfahren ist demnach aufgrund seines hohen Entwurfsaufwands und nur mäßigem Nutzens auf aktuellen FPGAs nur unter den genannten Voraussetzungen zu empfehlen. Feste Modulplätze stellen zurzeit die leichter zu implementierende und oft gleichwertige Option dar. Sobald von Seiten der Anwendungen die Nachfrage nach dynamisch rekonfigurierbaren Systemen steigt, könnten sich die oben angesprochenen Erweiterungen des Entwurfsablaufs sowie der FPGA-Architektur für die FPGA-Hersteller lohnen. Dann kann das Potential der freien Platzierungsverfahren voll ausgeschöpft und zur Steigerung der Ressourceneffizienz genutzt werden.

Symbolverzeichnis

A	Rekonfigurierbare Fläche
B	Größe eines Bitstroms
b	Datenbreite der Konfigurierungsschnittstelle
f	Funktion einer Task, einer Komponente, eines Moduls oder einer Modulinstanz
F	Formfaktor eines Moduls
f_{config}	Taktfrequenz der Konfigurierungsschnittstelle
h	Höhe eines rekonfigurierbaren Bereichs oder eines Moduls
H	Höhe eines rekonfigurierbaren Bereichs
k	Koordinaten einer Ressource, Ressourcenfläche oder eines Moduls
k_{ol}	Koordinaten der oberen linken Ecke einer Fläche
k_{ur}	Koordinaten der unteren linken Ecke einer Fläche
N_{cyc}	Anzahl der Wechsel zwischen Schreib- und Lesezugriffen während einer Konfigurierung
N_{Frame}	Anzahl der Frames eines Bitstroms
p	Gültige Modulposition
P	Packdichte
P_M	Menge gültiger Modulpositionen eines Moduls M
$\tilde{r}$	Anzahl benötigter Ressourcen
$\bar{r}'$	Benutzte Ressourcen eines Moduls
$\bar{r}$	Anzahl belegter Ressourcen einer Ressourcenmenge, Fläche oder Moduls
$\bar{r}_{Kachel}$	Ressourcen einer Kachel
$\bar{r}_{Komm}$	Belegte Ressourcen eines Kommunikations-Anschlusspunkts
$\overline{R}$	Ressourcenfläche
R	Menge rekonfigurierbarer Ressourcen
R_D	Rekonfigurierbarer Bereich
$\mathfrak{R}$	Menge rekonfigurierbarer Bereiche
S	Anzahl der Übertragenen Signale eines Busmakros
S_{bel}	Anzahl der durch ein Modul belegten Slices

S_{Frame}	Größe eines Frames
S_{min}	Minimaler Slicebedarf einer dynamischen Komponente
T_{arbit}	Dauer der Arbitrierung der Konfigurierungsschnittstelle
T_{begin}	Zeit, die der Konfigurierungsmanager zum Starten einer Konfigurierung benötigt
T_{clear}	Dauer eines vollständigen Löschvorgangs eines FPGAs
t_{clock}	Minimale Periodendauer eines Taktsignals
t_{config}	Konfigurierungszeit
t_{delay}	Signallaufzeit
T_{end}	Zeit, die der Konfigurierungsmanager zum Beenden einer Konfigurierung benötigt
T_{FPGA}	Menge der Ressourcentypen eines FPGAs
t_{in}	Signallaufzeit von der Busleitung zum Eingangsregister eines Anschlusspunkts
T_{init}	Dauer der Initialisierungsphase einer Konfigurierung
t_{oh}	Durch den Konfigurierungsmanager verursachte, zusätzliche Konfigurierungszeit
t_{out}	Signallaufzeit vom Ausgangsregister eines Anschlusspunkts zur Busleitung
T_{pad}	Zeit, die zum Schreiben eines Pad-Worts benötigt wird
T_{prog}	Übertragungszeit eines Initialisierungs-Kommandos an das FPGA
T_{reinit}	Dauer einer FPGA-Reinitialisierung
t_{skew}	Taktskew
T_{start}	Reaktionszeit des Konfigurierungsmanagers
T_{switch}	Umschaltzeit zwischen Schreib- und Lesezugriffen einer Konfigurierung
t_{unc}	Maximaler Jitter eines Taktsignals
U_I	Interne Ressourcenauslastung eines Moduls
V_{Abb}	Abbildungsverschnitt
V_{Kachel}	Kachelverschnitt
w	Breite eines rekonfigurierbaren Bereichs oder eines Moduls
W	Breite eines rekonfigurierbaren Bereichs
η_{FF}	Anzahl der FFs in einer Ressourcenmenge
η_i	Häufigkeit der Ressourcen des Typs τ_i in einer Ressourcenmenge
η_{LUT}	Anzahl der LUTs in einer Ressourcenmenge
η_{Slice}	Anzahl der Slices in einer Ressourcenmenge
ρ	FPGA-Ressource

τ	Ressourcentyp
Φ	Geometrische Anordnung der Ressourcen einer Ressourcenmenge R
φ	Flächentyp einer Fläche oder eines Moduls

Abkürzungen

ADC	Analogue to Digital Converter
AES	Advanced Encryption Standard
AL	Allocation Layer
ALM	Adaptive Logic Module
ALUT	Adaptive Look Up Table
APL	Application Layer
ASIC	Application Specific Integrated Circuit
BM	Bus Manager
CL	Configuration Layer
CLB	Configurable Logic Block
CORDIC	Coordinate Rotation Digital Computer
CPLD	Complex Programmable Logic Device
CRC	Cyclic Redundancy Check
DAC	Digital to Analogue Converter
DCM	Digital Clock Manager
DMA	Direct Memory Access
E/A	Eingabe/Ausgabe
EDK	Embedded Development Kit
EWB	Encapsulated Wishbone
FF	Flipflop
FFT	Fast Fourier Transformation
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
FSM	Finite State Machine
GPIO	General Purpose Input Output
HDL	Hardware Description Language
HL	Hardware Layer
HW	Hardware

ICAP	Internal Configuration Access Port
ID	Identifier
INDRA	Integrated Design Flow for Reconfigurable Architectures
IO	Input Output
IOB	Input Output Block
IOI	Input Output Interconnect
IP	Intellectual Property
JPEG	Joint Photographic Experts Group
JTAG	Joint Test Action Group
LAB	Logic Array Block
LED	Light Emitting Diode
LOL	Load on Linux
LUT	Look Up Table
MiDesires	Module Implementation Design Flow for Reconfigurable Systems
MML	Module Management Layer
NoC	Network on Chip
OPB	On-chip Peripheral Bus
OS	Operating System
PALMERA	Paderborn Layer Model for Reconfigurable Architectures
PAR	Place and Route
PC	Personal Computer
PCI	Peripheral Component Interconnect
PL	Positioning Layer
PLB	Processor Local Bus
PLD	Programmable Logic Device
PowerPC	Performance Optimization with enhanced RISC Performance Chip
PPC	PowerPC
PROM	Programmable Read Only Memory
RAM	Random Access Memory
RISC	Reduced Instruction Set Computing
RMW	Read Modify Writeback

ROTFL	Reconfiguration of the FPGA using Linux
SAI	Sensor Actuator Interface
SARA	Simulation Framework for the Analysis of Reconfigurable Architectures
SDRAM	Synchronous Dynamic Random Access Memory
SoPC	System on Programmable Chip
SRAM	Static Random Access Memory
SSI	Serial Synchronous Interface
SW	Software
UCF	User Constraints File
VCM	Virtex Configuration Manager
VHDL	Very High Speed Integrated Circuit Hardware Description Language
WB	Wishbone
X-BBG	XDL-based Busmacro Generator

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[ABF05] A. AHMADINIA, C. BOBDA, S. FEKETE, M. MAJER, J. TEICH, J. VAN DER VEEN: *DyNoC: A Dynamic Infrastructure for Communication in Dynamically Reconfigurable Devices*. In: *Proceedings of the 2005 International Conference on Field Programmable Logic and Applications (FPL)*, Tampere, Finnland, 2005, Seiten 153-158.

[Alt07] ALTERA CORPORATION: *Stratix-II Device Handbook: Complete Two Volume Set.* Produktdatenblatt, v4.3, 2007.

[And98] R. ANDRAKA: *A survey of CORDIC algorithms for FPGA based computers.* In: *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays (FPGA),* Monterey, USA, Seiten 191-200, 1998.

[ARM08] ARM LTD: *AMBA 2.0 Specification*, http://www.arm.com/products/solutions/AMBAHomePage.html, Zugriff am 23.01.2008.

[Atm06] ATMEL CORPORATION: *AT40K05AL: 5K - 50K Gates Coprocessor FPGA with FreeRAM.* Produktdatenblatt, Revision F, 2006.

[BHR04] P. BUTEL, G. HABAY, A. RACHET: *Managing Partial Dynamic Reconfiguration in Virtex-II Pro FPGAs.* In: *X-Cell Journal,* 2004, Issue 50.

[BJK03] B. BLODGET, P. JAMES-ROXBY, E. KELLER, S. MCMILLAN, P. SUNDARARAJAN: *A Self-reconfiguring Platform.* In: *Proceedings of the 2003 International Conference on Field Programmable Logic and Applications (FPL)*, Lissabon, Portugal, 2003, Seiten 565-574.

[BMA05] C. BOBDA, M. MAJER, A. AHMADINIA, T. HALLER, A. LINARTH, J. TEICH: *Increasing the Flexibility in FPGA-Based Reconfigurable Platforms: The Erlangen Slot Machine.* In: *Proceedings of the IEEE 2005 Conference on Field-Programmable Technology (FPT)*, Singapur, 2005, Seiten 37-42.

[BMK04] C. BOBDA, M. MAJER, D. KOCH, A. AHMADINIA, J. TEICH: *A Dynamic NoC Approach for Communication in Reconfigurable Devices*. In: *Proceedings of the 2004 International Conference on Field Programmable Logic and Applications (FPL)*, Antwerpen, Belgien, 2004, Seiten 1032-1036.

[BPS07] L. BRAUN, T. PERSCHKE, V. SCHATZ, S. BACH, M. HÜBNER, J. BECKER: *Circuit Switched Run-Time Adaptive Network-on-Chip for Image Processing Applications.* In: *Proceedings of the 2007 International Conference on Field Programmable Logic and Applications (FPL)*, Amsterdam, Niederlande, 2007.

[Bre96] G. J. BREBNER: *A Virtual Hardware Operating System for the Xilinx XC6200*. In: *Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers (FPL)*, London, Großbritannien, 1996, Seiten 327-336.

[CCK02] K. COMPTON, Z. LI, J. COOLEY, S. KNOL, S. HAUCK: *Configuration Relocation and Defragmentation for Run-time Reconfigurable Computing.* In*: IEEE Transactions on Very Large Scale Integration (VLSI) Systems 10,* 2002, Nr. 3, Seiten 209-220.

[CH02] K. COMPTON, S. HAUCK: *Reconfigurable Computing: A Survey of Systems and Software.* In: *ACM Computing Surveys 34*, 2002, Nr. 2, Seiten 171-210.

[Cur06] D. CURD, *Power Consumption in 65 nm FPGAs*. Xilinx Whitepaper, 2006.

[CZH07] C. CLAUS, B. ZHANG, M. HÜBNER, C. SCHMUTZLER, J. BECKER, W. STECHELE: *An XDL-based busmacro generator for customizable communication interfaces for dynamically and partially reconfigurable systems.* In: *Proceedings of the the 2nd International Workshop on Reconfigurable Computing Education,* Porto Allegre, Brasilien, 2007.

[CZM07] C. CLAUS, J. ZEPPENFELD, F. MÜLLER, W. STECHELE: *Using Partial-Run-Time Reconfigurable Hardware to accelerate Video Processing in Driver Assistance System.* In: *Proceedings of the International Conference on Design, Automation, and Test in Europe (DATE)*, Nizza, Frankreich, 2007.

[DBK03] K. DANNE, C. BOBDA, H. KALTE: *Run-time exchange of mechatronic controllers using partial hardware reconfiguration*. In: *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, Lissabon, Portugal, 2003.

[DE97] O. DIESSEL, H. A. ELGINDY: *Run-Time Compaction of FPGA Designs.* In: *Field-Programmable Logic and Applications, 7th International Workshop (FPL)*, Bd. 1304, Springer, 1997, Seiten 131-140.

[EOT05] J. ESQUIAGOLA, G. OZARI, M. TERUYA, M. STRUM, W. CHAU: *A dynamically reconfigurable Bluetooth baseband unit.* In: *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, Tampere, Finnland, 2005, Seiten 148-152.

[GLS99] S.A. GUCCIONE, D. LEVI, P. SUNDARARAJAN: *JBits: A Java-based interface for reconfigurable computing.* In: *Second Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, Laurel, USA, 1999.

[Gri03] B. GRIESE: *RAPTOR2000 Erweiterungsmodul DB-V2Pro,* User Guide, Fachgebiet Schaltungstechnik, Universität Paderborn, 2003.

[HBB04] M. HUEBNER, T. BECKER, J. BECKER: *Real-time LUT-based network topologies for dynamic and partial FPGA self-reconfiguration.* In: *Proceedings of the 17th Symposium on Integrated Circuits and System Design (SBCCI)*, Pernambuco, Brasilien, 2004, Seiten 28-32.

[Her02] R. HERVEILLE: *WISHBONE SoC Architecture Specification, Revision B.3*. www.opencores.org, 2002.

[HL01] E. HORTA, J. W. LOCKWOOD: *PARBIT: a tool to transform bitfiles to implement partial reconfiguration of field programmable gate arrays (FPGAs)*, Tech. Rep. WUCS-01-13, Washington University in Saint Louis, Department of Computer Science, 2001.

[HLP02] E.L. HORTA, J.W. LOCKWOOD, D. PARLOUR: *Dynamic Hardware Plugins in an FPGA with partial run-time reconfiguration.* In: *Proceedings of the 39th Design Automation Conference*, New Orleans, USA, 2002, Seiten 343-348.

[HNI07] HEINZ NIXDORF INSTITUT: *RAPTOR2000*. http://www.raptor2000.de, 2007. Zugriff am 20. Januar 2008.

[HSB06] M. HUEBNER, M. SCHUCK, J. BECKER: *Elementary block based 2-dimensional dynamic and partial reconfiguration for Virtex-II FPGAs.* In: *20th International Parallel and Distributed Processing Symposium (IPDPS)*, Rhodos, Griechenland, 2006.

[IBM06] IBM: *Device Control Register Bus 3.5 Architecture Specifications,* 2006.

[IEE85] INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS (IEEE): *IEEE 754: Standard for Binary Floating-Point Arithmetic*, 1985.

[Jac07] B. JACKSON: *Partial Reconfiguration Design with PlanAhead 9.2.* Xilinx Technical Report, 2007.

[Kal04] H. KALTE: *Einbettung dynamisch rekonfigurierbarer Hardwarearchitekturen in eine Universalprozessorumgebung.* Dissertation, Universität Paderborn, Fakultät für Elektrotechnik, Informatik und Mathematik, Fachbereich Schaltungstechnik, 2004.

[KD06] S. KOH, O. DIESSEL: *COMMA: A Communications Methodology for Dynamic Module-based Reconfiguration of FPGAs.* In: *Proceedings of the International Conference on Architectures of Computing Systems (ARCS)*, Frankfurt, Deutschland, 2006, Seiten 173-182.

[KKP05c] M. KÖSTER, H. KALTE, M. PORRMANN: *Run-Time Defragmentation for Partially Reconfigurable Systems.* In: *Proceedings of the International Conference on Very Large Scale Integration of System-on-Chip (IFIP VLSISoC)*, Perth, Australien, 2005, Seiten 109-115.

[KKP06] M. KÖSTER, H. KALTE, M. PORRMANN: *Relocation and Defragmentation for Heterogeneous Reconfigurable Systems.* In: *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, Las Vegas, USA, 2006, Seiten 70-76.

[Kla04] A. KLASSEN: *RAPTOR2000 Erweiterungsmodul DB-V2,* User Guide, Fachgebiet Schaltungstechnik, Universität Paderborn, 2004.

[Kle05] M. KLEIN: *Static Power and the Importance of Realistic Junction Temperature Analysis*, Xilinx Whitepaper, 2005.

[KLP05] H. KALTE, G. LEE, M. PORRMANN, U. RÜCKERT: *REPLICA: A Bitstream Manipulation Filter for Module Relocation in Partial Reconfigurable Systems.* In: *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS) – Reconfigurable Architectures Workshop (RAW)*, Denver, USA, 2005.

[Koe07] M. KÖSTER: *Analyse und Entwurf von Methoden zur Ressourcenverwaltung-partiell rekonfigurierbarer Architekturen.* Dissertation, Universität Paderborn, Fakultät für Elektrotechnik, Informatik und Mathematik, Fachbereich Schaltungstechnik, 2007.

[KPA06] R. KOCH, T. PIONTECK, C. ALBRECHT, E. MAEHLE: *An Adaptive System-on-Chip for Network Applications.* In: *Proceedings of the 20th IEEE International Parallel & Distributed Processing Symposium (IPDPS), Reconfigurable Architectures Workshop (RAW)*, Rhodos, Griechenland, 2006.

[KPR00] H. KALTE, M. PORRMANN, U. RÜCKERT: *Rapid Prototyping System für dynamisch rekonfigurierbare Hardwarestrukturen*. In: *Architekturentwurf und Entwicklung eingebetteter Systeme (AES2000)*, Karlsruhe, Deutschland, 2000, Seiten 149-157.

[KPR04a] H. KALTE, M. PORRMANN, U. RÜCKERT: *System-on-Programmable-Chip Approach Enabling Online Fine-Grained 1D-Placement*. In: *Proceedings of the 18th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, Santa Fé, USA, 2004.

[KPR04b] H. KALTE, M. PORRMANN, U. RÜCKERT: *Study on Column Wise Design Compaction for Reconfigurable Systems.* In: *Proceedings of the IEEE International Conference on Field Programmable Technology (FPT*), Brisbane, Australien, 2004.

[KPR05] M. KÖSTER, M. PORRMANN, U. RÜCKERT: *Placement-Oriented Modeling of Partially Reconfigurable Architectures.* In: *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS) – Reconfigurable Architectures Workshop (RAW)*, Denver, USA, 2005.

[Lam07] DAMJAN LAMPRET: *Arithmetic Cores*. http://www.opencores.org/browse.cgi/filter/category_arithmetic. Zugriff am 22. Februar 2007.

[LBM06] P. LYSAGHT, B. BLODGET, J. MASON, J. YOUNG, B. BRIDGEFORD: *Enhanced Architectures, design methodologies and CAD tools for dynamic reconfiguration on XILINX FPGAs.* In: *Proceedings of the 2006 International Conference on Field Programmable Logic and Applications (FPL),* Madrid, Spanien, 2006.

[MBV02] T. MARESCAUX, A. BARTIC, D. VERKEST, S. VERNALDE, R. LAUWEREINS: *Interconnection networks enable fine-grain dynamic multi-tasking on FPGAs.* In: *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications (FPL)*, Montpellier, Frankreich, 2002, Seiten 795-805.

[Mon08] MONTAVISTA SOFTWARE INC.: *Real-Time Linux Development from MontaVista*. http://www.mvista.com/real_time_linux.php. Zugriff am 19. Dezember 2007.

[NIS01] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY (NIST): *Advanced Encryption Standard (AES).* Federal Information Processing Standards Publication 197, USA, 2001.

[NKD05] K. NASI, T. KAROUBALIS, M. DANĚK, Z. POHL: *FIGARO – An Automatic Tool Flow for Designs with Dynamic Reconfiguration.* In: *Proceedings of the 2005 International Conference on Field Programmable Logic and Applications (FPL)*, Tampere, Finnland, 2005.

[PAK07] T. PIONTECK, C. ALBRECHT, R. KOCH, E. MAEHLE, M. HÜBNER, J. BECKER*: Communication Architectures for Dynamically Reconfigurable FPGA Designs.* In: *Proceedings of the 2007 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Reconfigurable Architectures Workshop (RAW),* Long Beach, USA, 2007, Seiten 174-181.

[PC06] D. PUSCHINI, F. CLERMIDY: *A Comparison Between NoC and Bus Architectures Based on a Real-Application.* In: *Proceedings of the 2nd International Workshop on Reconfigurable Communication-centric System-on-Chips (ReCoSoC)*, Montepellier, Frankreich, 2006, Seiten 161-168.

[RI04] I. ROBERTSON, J. IRVINE: *A Design Flow for Partially Reconfigurable Hardware*. In: *ACM Transactions on Embedded Computing Systems*, Vol. 3, No. 2, 2004, Seiten 257-283.

[Rys05] P. RYSER: *Software Defined Radio with Reconfigurable Hardware and Software: A Framework for a TV Broadcast Receiver.* In: *Proceeding of Embedded Systems Conference*, San Francisco, USA, 2005.

[SBA05] P. SEDCOLE, B. BLODGET, J. ANDERSON, P. LYSAGHT, T. BECKER: *Modular Partial Reconfiguration in Virtex FPGAs.* In: *Proceedings of the 2005 International Conference on Field Programmable Logic and Applications (FPL)*, Tampere, Finnland, 2005, Seiten 211-216.

[Sed06] P. SEDCOLE: *Reconfigurable Platform-Based Design in FPGAs for Video Image Processing*, Dissertation, Department of Electrical and Electronic Engineering, Imperial College of Science, Technology and Medicine, University of London, 2006.

[SPH07] B. SCHULZ, C. PAIZ, J. HAGEMEYER, S. MATHAPATI, M. PORRMANN, J. BÖCKER: *Run-Time Reconfiguration of FPGA-based Drive Controllers.* In: *Proceedings of the European Conference on Power Electronics and Applications (EPE 2007)*, Aalborg, Dänemark, 2007.

[Tel06] ANIL TELIKEPALLI: *Power vs. Performance: The 90 nm Inflection Point*, Xilinx Whitepaper, 2006.

[UHG04] M. ULLMANN, M. HÜBNER, B. GRIMM, J. BECKER: *On-Demand FPGA Run-Time System for Dynamical Reconfiguration with Adaptive Priorities.* In: *Proceedings of the 14th International Workshop on Field-Programmable Logic and Applications (FPL)*, Antwerpen, Belgien, 2004, Seiten 454-463.

[VJS95] J. VILLASENOR, C. JONES, B. SCHONER: *Video communications using rapidly reconfigurable hardware.* In: *IEEE Transactions on Circuits and Systems for Video Technology*, Vol. 5, No. 6, 1995, Seiten 565-567.

[Xil02] XILINX INC.: *XAPP290: Two Flows for Partial Reconfiguration: Module Based or Difference Based*, Application Note, v1.2, 2004.

[Xil04a] XILINX INC.: *Processor Local Bus (PLB) v3.4 (v1.01a)*, Produktspezifikation, 2006.

[Xil04b] XILINX INC.: *On-Chip Peripheral Bus (OPB) v2.0 with OPB Arbiter (v1.10b)*, Produktspezifikation, 2006.

[Xil05] XILINX INC.: *Virtex-II Platform FPGAs: Complete Data Sheet.* Produktdatenblatt, v3.4, 2005.

[Xil06] XILINX INC.: *Early Access Partial Reconfiguration User Guide UG208 (v1.1)*, 2006.

[Xil07a] XILINX INC.: *Virtex-5 User Guide.* Produktdatenblatt, v3.2, 2007.

[Xil07b] XILINX INC.: *Virtex-4 User Guide.* Produktdatenblatt, v2.3, 2007.

[Xil07c] XILINX INC.: *CORE Generator User Guide.* http://toolbox.xilinx.com/docsan/xilinx9/help/iseguide/mergedProjects/coregen/coregen.htm, Zugriff am 27.01.2008.

[Xil08] XILINX INC.: *Platform Studio Documentation*, http://www.xilinx.com/ise/embedded/edk_docs.htm, Zugriff am 16.12.2007.

Eigene Veröffentlichungen

[GKP06] B. GRIESE, B. KETTELHOIT, M. PORRMANN: *Evaluation of on-chip interfaces for dynamically reconfigurable coprocessors.* In: *Proceedings of the 5th International symposium on Parallel Computing in Electrical Engineering,* Bialystok, Poland, 2006, Seiten 214-219.

[HKK07a] J. HAGEMEYER, B. KETTELHOIT, M. KÖSTER, M. PORRMANN: *INDRA: Integrated Design Flow for Reconfigurable Architectures*. In: *Proceedings of the International Conference on Design, Automation, and Test in Europe (DATE)*, Nizza, Frankreich, 2007.

[HKK07b] J. HAGEMEYER, B. KETTELHOIT, M. KÖSTER, M. PORRMANN: *Design of Homogeneous Communication Infrastructures for Partially Reconfigurable FPGAs*. In: *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, Las Vegas, USA, 2007.

[HKK07c] J. HAGEMEYER, B. KETTELHOIT, M. KÖSTER, M. PORRMANN: *A Design Methodology for Communication Infrastructures for Partially Reconfigurable FPGAs*. In: *Proceedings of the 2007 International Conference on Field Programmable Logic and Applications (FPL)*, Amsterdam, Niederlande, 2007.

[HKP06] J. HAGEMEYER, B. KETTELHOIT, M. PORRMANN: *Dedicated Module Access in Dynamically Reconfigurable Systems.* In: *Proceedings of the 20th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, Rhodos, Griechenland, 2006.

[KKK04] H. KALTE, M. KÖSTER, B. KETTELHOIT, M. PORRMANN, U. RÜCKERT: *A Comparative Study on System Approaches for Partially Reconfigurable Architetures*. In: *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, Las Vegas, USA, 2004, Seiten 70-76.

[KKK05] H. KALTE, B. KETTELHOIT, M. KÖSTER, M. PORRMANN, U. RÜCKERT: *A System Approach for Partially Reconfigurable Architectures*. In: *International Journal of Embedded Systems (IJES)*, 2005, Seiten 274-290.

[KKP05a] B. KETTELHOIT, H. KALTE, M. PORRMANN, U. RÜCKERT: *Dynamically Reconfigurable Hardware for Self-Optimizing Mechatronic Systems*. In: *GMM/ITG/GI-Workshop Multi-Nature Systems*, Dresden, Deutschland, 2005, Seiten 97-101.

[KKP05b] B. KETTELHOIT, A. KLASSEN, C. PAIZ, M. PORRMANN, U. RÜCKERT: *Rekonfigurierbare Hardware zur Regelung machatronischer Systeme*. In: *3. Paderborner Workshop: Intelligente Mechatronische Systeme*, Paderborn, Deutschland, 2005, Seiten 195-205.

[KP06] B. KETTELHOIT, M. PORRMANN: *A Layer Model for Systematically Designing Dynamically Reconfigurable Systems.* In: *Proceedings of the 2006 International Conference on Field Programmable Logic and Applications (FPL),* Madrid, Spanien, 2006, Seiten 547-552.

[PKK05] C. PAIZ, B. KETTELHOIT, A. KLASSEN, M. PORRMANN, U. RÜCKERT: *Dynamically Reconfigurable Hardware for Digital Controllers in Mechatronic Systems*. In: *Proceedings of the IEEE International Conference on Mechatronics (ICM)*, Taipeh, Taiwan, 2005.

[PKP07] C. PAIZ, B. KETTELHOIT, M. PORRMANN: *A Design Framework for FPGA-based dynamically reconfigurable digital controllers.* In: *IEEE International Symposium on Circuits and Systems,* New Orleans, USA, 2007, Seiten 3709-3711.

[RSS07] V. RANA, M. SANTAMBROGIO, D. SCIUTO, B. KETTELHOIT, M. KÖSTER, M. PORRMANN, U. RÜCKERT: *Partial Dynamic Reconfiguration in a Multi-FPGA clustered Architecture based on Linux.* In: *Proceedings of the 21st IEEE International Parallel & Distributed Processing Symposium (IPDPS),* Long Beach, USA, 2007.

Betreute Studien- und Diplomarbeiten

[Sti04D] ANDREAS STICKART: *Anbindung rekonfigurierbarer Hardware an einen eingebetteten Prozessor*. Diplomarbeit, Fachgebiet Schaltungstechnik, Universität Paderborn, Juli 2004.

[Ise04S] CHRISTIAN ISEKE: *Ein RAPTOR2000-Modul für regelungstechnische Anwendungen*. Studienarbeit, Fachgebiet Schaltungstechnik, Universität Paderborn, Dezember 2004.

[Mog05D] RAOUL MOGE: *FPGA-basierte Regelung eines stufenlosen MOSFET-Widerstands*. Diplomarbeit, Fachgebiet Schaltungstechnik, Universität Paderborn, Oktober 2005.

[Kla05D] ALEXANDER KLASSEN: *Eine Systemarchitektur für dynamisch rekonfigurierbare Regelungen*. Diplomarbeit, Fachgebiet Schaltungstechnik, Universität Paderborn, Februar 2005.

[Sch05S] MARKUS SCHUMACHER: *Implementierung eines Schichtenmodells zur dynamischen Hardware-Rekonfigurierung*. Studienarbeit, Fachgebiet Schaltungstechnik, Universität Paderborn, Dezember 2005.

[Hag05S] JENS HAGEMEYER: *Partielle dynamische Selbst-Rekonfiguration mit VirtexII-FPGAs*. Studienarbeit, Fachgebiet Schaltungstechnik, Universität Paderborn, Dezember 2005.

[Ran06M] VINCENZO RANA: *A novel methodology for dynamically reconfigurable embedded systems design*. Masterarbeit, Dipartimento di Elettronica e Informazione, Politecnico di Milano, August 2006.

[Sch06D] MARKUS SCHUMACHER: *Leistungsbewertung von Platzierungsverfahren für dynamisch rekonfigurierbare Hardware*. Diplomarbeit, Fachgebiet Schaltungstechnik, Universität Paderborn, September 2006.

[Hag06D] JENS HAGEMEYER: *Homogene On-Chip-Kommunikationsstrukturen für dynamisch rekonfigurierbare Systeme*. Diplomarbeit, Fachgebiet Schaltungstechnik, Universität Paderborn, Dezember 2006.