Analyse und Entwurf von Methoden zur Ressourcenverwaltung partiell rekonfigurierbarer Architekturen

Zur Erlangung des akademischen Grades

DOKTOR-INGENIEUR

der Fakultät Elektrotechnik, Informatik und Mathematik

der Universität Paderborn

genehmigte Dissertation

von

M.Sc. Dipl.-Ing. Markus Köster

aus Detmold

Referent: Prof. Dr.-Ing. Ulrich Rückert

Korreferent: Prof. Dr. techn. Marco Platzner

Tag der mündlichen Prüfung: 6. Juni 2007

Paderborn, den 11. Juni 2007

D 14-231

„They say that time changes things, but you actually have to change them yourself.“
(Andy Warhol)

Danksagung

Die vorliegende Arbeit wurde während meiner Tätigkeit als Stipendiat des Graduiertenkollegs ”Automatische Konfiguration in offenen Systemen” der Deutschen Forschungsgemeinschaft (DFG) an der Universität Paderborn unter der Anleitung von Prof. Dr. Ulrich Rückert angefertigt. Für das von der DFG gewährte Stipendium möchte ich mich besonders bedanken.

Ich möchte mich herzlich bei all jenen Personen bedanken, die mich auf unterschiedliche Art und Weise unterstützt und zu dieser Arbeit beigetragen haben. Besonderem Dank gilt Prof. Ulrich Rückert dafür, dass er diese Arbeit ermöglicht hat und mich in seine Arbeitsgruppe aufgenommen hat. Seine konstruktive Kritik hat mir beim Abschluss dieser Arbeit sehr geholfen. Ein großer Dank gilt Dr. Mario Porrmann für die stete Hilfsbereitschaft und die zahlreichen fachlichen Diskussionen, die zu dieser Arbeit beigetragen haben. Ebenso möchte ich mich bei ihm für das sorgfältige Korrekturlesen der vorliegenden Arbeit bedanken.

Ein besonderer Dank geht an Dr. Heiko Kalte für die außergewöhnlich gute und intensive Zusammenarbeit und für die Unterstützung als Koautor in einigen der im Rahmen dieser Promotion entstanden Veröffentlichungen. Ich möchte mich außerdem bei Prof. Marco Platzner für die Übernahme des Koreferats bedanken. Mein Dank gilt ebenso Prof. Adolf Grauel, der mich dazu ermutigt hat, den Schritt der Promotion zu wagen.

Allen Mitgliedern des Fachgebietes Schaltungstechnik danke ich für das angenehme und gemeinschaftliche Arbeitsklima. Hier geht ein besonderer Dank an meinen Bürokollegen Ralf Eickhoff für viele interessante Diskussionen auch außerhalb des Themas der rekonfigurierbaren Hardware.

Besonders danke ich Alexandra Jacob für ihre Geduld, ihre liebevolle Unterstützung und das abschließende Korrekturlesen dieser Arbeit. Zuletzt möchte ich mich für den Rückhalt und die Unterstützung aus meiner Familie bedanken.

Inhaltsverzeichnis

1 Einleitung

Die Anforderungen an die Leistungsfähigkeit, den Flächenbedarf und die Energieeffizienz von mikroelektronischen Systemen sind in den letzten Jahren kontinuierlich gestiegen. Insbesondere mobile eingebettete Systeme, wie z. B. Mobiltelefone, müssen derart gestaltet sein, dass sie platzsparend, leistungsfähig, energieeffizient und flexibel sind. Zusätzlich verkürzen sich die Lebenszyklen mikroelektronischer Produkte, so dass die zur Verfügung stehenden Entwicklungszeiten (engl.: Time-to-Market) abnehmen müssen. Ein Ansatz, diesen Anforderungen gerecht zu werden, verspricht die Verwendung von *rekonfigurierbarer Hardware*.

In der Regel besteht rekonfigurierbare Hardware aus einem Feld von konfigurierbaren Verarbeitungseinheiten, die über eine programmierbare Kommunikationsinfrastruktur miteinander verbunden sind. Die Verarbeitungseinheiten beinhalten meist elementare Funktionen (z. B. logische oder arithmetische Operationen). Durch das Zusammenschalten der Verarbeitungseinheiten über die gegebene Kommunikationsinfrastruktur lassen sich einfache Schaltungen bis hin zu komplexen Systemen auf einem Chip (engl.: System-On-Chip) realisieren. Das Konzept der rekonfigurierbaren Hardware hat neue Perspektiven beim Entwurf mikroelektronischer Systeme eröffnet. Der Durchbruch der rekonfigurierbaren Hardware kam mit der Entwicklung des FPGAs (engl.: Field-Programmable Gate Array).

Im Jahr 1985 wurde von der Firma Xilinx das erste FPGA (XC2064) mit 64 konfigurierbaren Logikzellen und 1200 Gatteräquivalenten eingeführt [85]. Die Konfiguration der Logikzellen des XC2064 wurde ermöglicht durch das Programmieren eines SRAM-basierten Konfigurationsspeichers. Der Inhalt des Konfigurationsspeichers legt die Funktionen der Logikzellen und deren Verbindungen fest. Eine *partielle Rekonfiguration*, d. h., die Veränderungen der Funktionen einzelner Logikzellen zur Laufzeit war mit dem XC2064 nicht möglich. Erst etwa 10 Jahre später - im Jahr 1996 - wurden die ersten partiell rekonfigurierbaren FPGAs eingeführt (Xilinx XC6200 Serie [86]), welche aus bis zu 16384 Logikzellen und bis zu 100000 Gatteräquivalenten bestanden. Durch partielle Rekonfiguration können auf dem FPGA konfigurierte Systemkomponenten dynamisch angepasst oder komplett durch andere Systemkomponenten ausgetauscht werden. Auf diese Weise lassen sich Recheneinheiten realisieren, die in der Funktion variabel und zur Laufzeit anpassbar sind.

Seit der Einführung von partiell rekonfigurierbaren FPGAs sind weitere 10 Jahre vergangen. Die heutigen Xilinx Virtex-5 FPGAs [96] besitzen bis zu 330000 Logikzellen und darüber hinaus zusätzliche Zellen, wie z. B. verteilte Speicherblöcke. Mit der Integration von verschiedenen Zelltypen weichen die heutigen FPGAs von ihrer

einst homogenen Struktur ab und weisen eine heterogene Fläche von konfigurierbaren Zellen auf. Die Anzahl von Gatteräquivalenten ist von 1200 beim XC2064 auf über 7 Millionen beim Virtex-5 LX330T angestiegen.

FPGAs sind bis heute die am weitesten verbreiteten rekonfigurierbaren Architekturen. Die Marktzuwächse im Bereich der rekonfigurierbaren Hardware sind enorm. Die Firma Xilinx hat im Jahr 2006 einen Umsatz von $1,73$ Mrd.US$ erzielt und konnte den Umsatz im Vergleich zum Vorjahr um $10,2\%$ steigern. Das vielversprechende Konzept der partiellen Rekonfigurierbarkeit wurde bis jetzt jedoch nur vereinzelt genutzt, was unter anderem auf folgende Gründe zurückzuführen ist:

Fehlender automatisierter Entwurfsablauf: Wenn ein Universalprozessor mit einer rekonfigurierbaren Funktionseinheit gekoppelt wird, können Berechnungen einer Anwendung entweder auf dem Prozessor oder in Form von rekonfigurierbarer Hardware ausgeführt werden. Software-basierte Anwendungen sind meist in einer hohen Programmiersprache, wie z. B. ANSI-C oder Java, spezifiziert. Um eine solche Anwendung mit Hilfe von dynamisch rekonfigurierbarer Hardware zu beschleunigen, können Teile der Anwendung als dynamisch rekonfigurierbare Systemkomponenten realisiert werden. Die automatische Erzeugung von Systemkomponenten in einer synthetisierbaren Hardware-Beschreibungssprache aus einer Programmspezifikation in Form einer Hochsprache wie ANSI-C ist jedoch nur eingeschränkt möglich (vgl. [60]) und verlangt meistens zusätzliches Expertenwissen im Bereich des Hardware-Entwurfs. Ein einfacher Entwurfsablauf für Anwendungen zur Ausführung auf partiell rekonfigurierbarer Hardware ist damit nicht gegeben.

Fehlende Systemansätze zur Realisierung: Um den Datenaustausch zwischen Universalprozessor und einer rekonfigurierbaren Funktionseinheit zu ermöglichen, wird ein Systemansatz mit einer flexiblen Kommunikationsinfrastruktur benötigt. Bei der Platzierung von dynamisch rekonfigurierbaren Systemkomponenten muss eine Anbindung an die Kommunikationsinfrastruktur des vorhandenen Systems geschaffen werden. Dabei werden Mechanismen benötigt, die eine partielle Rekonfiguration erlauben, ohne die Funktionsweise anderer platzierter Systemkomponenten und der Kommunikationsinfrastruktur zur Laufzeit zu beeinträchtigen.

Fehlende Methoden zur Ressourcenverwaltung: Die dynamisch platzierbaren Systemkomponenten sind unterschiedlich komplex und variieren daher in der Anzahl der benötigten Ressourcen. Um die zur Verfügung stehenden Ressourcen der rekonfigurierbaren Architektur möglichst effizient zu nutzen, werden Methoden zur Ressourcenverwaltung benötigt. Die Methoden umfassen einerseits die Verwaltung der freien und belegten Ressourcen und andererseits die Platzierung der von der Anwendung geforderten Systemkomponenten. In Abhängigkeit des zugrunde liegenden Systemansatzes und der gegebe-

nen Kommunikationsinfrastruktur ergeben sich dabei Einschränkungen in der Menge der möglichen Positionen bei der Platzierung einer Systemkomponente.

Diese Arbeit widmet sich dem Thema der Ressourcenverwaltung von partiell rekonfigurierbaren Architekturen. Im Speziellen wird dabei auf die Platzierung von Systemkomponenten zur Laufzeit eingegangen. Die Ziele dieser Arbeit werden im Einzelnen im folgenden Abschnitt dargestellt.

1.1 Zielsetzung der Arbeit

Ein wesentliches Ziel dieser Arbeit ist die Analyse von Systemansätzen zur Umsetzung partiell rekonfigurierbarer Hardware. Der Schwerpunkt der Analyse liegt dabei auf dem Vergleich der Leistungsfähigkeit der verschiedenen Systemansätze im Hinblick auf die Platzierung von Hardware-Modulen zur Laufzeit. Im Einzelnen ergeben sich dabei folgende Fragen:

- Welcher Systemansatz führt zur größten Ressourcenauslastung der gegebenen rekonfigurierbaren Architektur?
- Die Bestimmung der Position eines Hardware-Moduls erfordert eine Rechenzeit, die von dem gewählten Platzierungsverfahren abhängt. Welchen Einfluss hat das Platzierungsverfahren und die benötigte Rechenzeit auf die Leistungsfähigkeit eines Systemansatzes?
- Nachdem die Position eines geforderten Hardware-Moduls bestimmt wurde, folgt die partielle Rekonfiguration der Architektur anhand der Konfigurationsdaten des Hardware-Moduls. Welchen Einfluss hat die für den Konfigurationsprozess erforderliche Konfigurationszeit auf die Leistungsfähigkeit eines Systemansatzes?
- Lässt sich die Leistungsfähigkeit eines Systemansatzes durch Defragmentierung verbessern?

Da heutige FPGAs aus Zellen verschiedenen Typs (Speicherblöcke, Signalverarbeitungszellen) bestehen, ergeben sich bei der Platzierung von dynamisch rekonfigurierbaren Systemkomponenten Einschränkungen bezüglich der möglichen Positionen der entsprechenden Hardware-Module. Auf diese Weise entsteht eine Heterogenität, die in den Platzierungsverfahren berücksichtigt werden muss. Ein wichtiges Ziel dieser Arbeit ist die Analyse, ob die im Zusammenhang mit homogenen rekonfigurierbaren Architekturen bekannten Platzierungsverfahren auch für heterogene Architekturen adaptiert werden können. Im Rahmen der Analyse heterogener rekonfigurierbarer Architekturen ergeben sich dabei folgende Fragen:

- Wie wirkt sich die Heterogenität heutiger partiell rekonfigurierbarer Architekturen auf die Platzierungsverfahren aus?
- Welche Eigenschaften muss ein Systemansatz für heterogene rekonfigurierbare Architekturen aufweisen, um eine hohe Ressourcenauslastung zu erzielen?
- Lässt sich die Leistungsfähigkeit eines Systemansatzes für heterogene rekonfigurierbare Architekturen durch Defragmentierung verbessern?

Neben der Beantwortung der oben aufgeführten Fragen werden neue Methoden zur Ressourcenverwaltung im Hinblick auf heterogene Architekturen vorgestellt, die dazu beitragen sollen, das Konzept der partiellen Rekonfigurierbarkeit auf einfache Weise nutzbar zu machen. Ein wichtiger Beitrag dieser Arbeit ist die Entwicklung eines neuen Platzierungsansatzes, welcher die im Zusammenhang mit heterogenen rekonfigurierbaren Architekturen vorhandene eingeschränkte Anzahl der möglichen Positionen von Hardware-Modulen berücksichtigt. Im folgenden Abschnitt wird der Aufbau der Arbeit skizziert.

1.2 Aufbau der Arbeit

Kapitel 2 behandelt die Grundlagen der rekonfigurierbaren Hardware. Dabei wird auf die verschiedenen Architekturkonzepte von rekonfigurierbarer Hardware eingegangen, die sich in unterschiedliche Granularitätsebenen klassifizieren lassen. Ferner werden die verschiedenen Varianten der Systemanbindung, ebenso wie die unterschiedlichen Konfigurationsansätze beschrieben. Die Merkmale von partiell rekonfigurierbaren Architekturen, sowie mögliche Anwendungsszenarien werden herausgestellt. Im Zusammenhang mit partiell rekonfigurierbarer Hardware ergeben sich verschiedene Systemansätze, die im Einzelnen kurz vorgestellt werden.

Kapitel 3 stellt die Modellierung vor, die die Grundlage der in dieser Arbeit beschriebenen Methoden bildet. Mithilfe des in dieser Arbeit entwickelten DMC-Modells lassen sich Methoden zur partiellen Rekonfigurierbarkeit wie die Platzierung oder die Ablaufplanung formal beschreiben. Um die im weiteren Verlauf der Arbeit betrachteten Systemansätze und die entsprechenden Platzierungsverfahren zu analysieren, wurde die Simulationsumgebung SARA entwickelt, welche auf dem DMC-Modell beruht. Die Bestandteile und die verschiedenen Parameter von SARA werden im Einzelnen erläutert.

Kapitel 4 beschäftigt sich mit der Platzierung von Hardware-Modulen in homogenen rekonfigurierbaren Architekturen. Die Platzierung von Hardware-Modulen zur Laufzeit weist eine Ähnlichkeit zum Online-Packungsproblem auf, so dass die Gemeinsamkeiten und Unterschiede charakterisiert werden. Die Platzierung eines Hardware-Moduls lässt sich dabei in die *Verwaltung der freien Zellen* und in die *Positionsbestimmung* unterteilen. Es werden verschiedene Verfahren zur Verwaltung der freien

Zellen vorgestellt. Ebenso wird auf die im Zusammenhang mit Online-Packungsproblemen bekannten Verfahren zur Positionsbestimmung eingegangen und die für rekonfigurierbare Hardware erforderlichen Anpassungen beschrieben. Um die verschiedenen Systemansätze und Platzierungsverfahren simulativ zu vergleichen, werden Metriken für dynamisch rekonfigurierbare Hardware eingeführt. Anhand von Simulationen und den beschriebenen Metriken werden die verschiedenen Systemansätze miteinander verglichen. Ferner wird der Einfluss der Platzierungszeit und der Konfigurationszeit der Hardware-Module analysiert. Des Weiteren werden verschiedene bekannte und neu entwickelte Defragmentierungsverfahren vorgestellt. Anhand von Simulationen wird gezeigt, inwieweit die Defragmentierung die Leistungsfähigkeit eines Systemansatzes im Hinblick auf die resultierende Ressourcenauslastung verbessern kann.

Kapitel 5 setzt sich mit der Platzierung von Hardware-Modulen in heterogenen rekonfigurierbaren Architekturen auseinander und ist ähnlich strukturiert wie das vorherige Kapitel. Dabei wird zunächst auf die Problematik der Platzierung von Hardware-Modulen in heterogenen Architekturen eingegangen. Die Unterschiede und die erforderlichen Anpassungen der von homogenen Architekturen bekannten Platzierungsverfahren werden dargestellt. Ein wichtiger Bestandteil dieses Kapitels ist die Einführung eines neuen Platzierungsansatzes für heterogene rekonfigurierbare Architekturen. Darauf aufbauend werden die Platzierungsverfahren SUP-Fit und RUP-Fit vorgestellt. Neben der Einführung neuer Platzierungsverfahren wird ein für heterogene rekonfigurierbare Architekturen geeignetes Defragmentierungsverfahren vorgestellt. Anhand von Simulationen wird anschließend der Einfluss der Defragmentierung auf die Leistungsfähigkeit eines Systemansatzes untersucht.

Im Kapitel 6 werden die wesentlichen Ergebnisse der vorliegenden Arbeit zusammengefasst und abschließend bewertet. Zusätzlich wird ein Ausblick auf mögliche Erweiterungen der in dieser Arbeit besprochenen Methoden zur Ressourcenverwaltung gegeben.

2 Rekonfigurierbare Hardware

Die Ausführung einer Anwendung oder eines Algorithmus kann im herkömmlichen Sinn auf grundsätzlich zwei verschiedene Arten umgesetzt werden. Eine Möglichkeit ist die Anwendung in Form von Hardware, z. B. als anwendungsspezifische Schaltung (ASIC), zu realisieren. Aufgrund des speziell für die Anwendung ausgelegten Hardware-Entwurfs erlauben ASICs eine schnelle und effiziente Verarbeitung der Anwendungsdaten. Jedoch kann die Schaltung nach der Fertigung nur begrenzt angepasst werden, so dass im Fall einer notwendigen Anpassung die Herstellung eines neuen ASICs unumgänglich ist. Ebenso sind die Kosten eines ASICs insbesondere bei geringer Stückzahl sehr hoch. Die zweite Art und Weise der Umsetzung einer Anwendung ist die Implementierung in Software. Hierbei wird die Anwendung als Programm realisiert, das aus einzelnen Instruktionen besteht, die sequenziell auf einem zugrunde liegenden Prozessor ausgeführt werden. Im Vergleich zur Hardware-Realisierung benötigt die Software-Realisierung durch die sequenzielle Ausführung der Instruktionen insgesamt eine längere Ausführungszeit der Anwendung. Die Verwendung von Software gestattet jedoch die Anpassung der Anwendung auch nach der Fertigung, so dass im direkten Vergleich zur statischen Hardware-Realisierung eine höhere Flexibilität geschaffen wird.

Rekonfigurierbare Hardware ist eine weitere Alternative zur hardware- und software-basierten Realisierung einer Anwendung. In der Literatur werden für den Begriff *Rekonfigurierbare Hardware* auch die Synonyme *Reconfigurable Computing* oder *Adaptive Computing* verwendet. Rekonfigurierbare Architekturen, wie z. B. Feldprogrammierbare Gatter Anordnungen (FPGAs), bestehen aus einer regelmäßigen Anordnung konfigurierbarer Zellen, die mit einer ebenfalls konfigurierbaren Verbindungsstruktur miteinander verbunden sind. Anwendungen werden ähnlich wie im klassischen Hardware-Entwurf z. B. in Form von Hardware-Beschreibungssprachen wie VHDL [43] spezifiziert. Mithilfe von computergestützten Entwurfswerkzeugen (EDA) werden entsprechende Konfigurationsdaten synthetisiert, die die einzelnen Funktionen der Zellen und deren Verbindungen beinhalten. Die rekonfigurierbare Architektur wird zu Beginn durch einen Konfigurationsprozess mit den Konfigurationsdaten initialisiert. Auf diese Weise lassen sich Anwendungen in Form von digitalen Schaltungen auf rekonfigurierbare Architekturen abbilden. Bezüglich rekonfigurierbarer Hardware werden Anwendungen somit in Form von Konfigurationsdaten repräsentiert. Rekonfigurierbare Hardware bietet damit eine ähnliche Flexibilität wie Software, denn die konfigurierten Schaltungen können auch nach der Fertigung angepasst werden. Bei einer Realisierung mit rekonfigurierbarer Hardware kann durch

Parallelisierung eine viel höhere Performanz erzielt werden als durch eine entsprechende Software-Realisierung. Beispiele von entsprechenden Anwendungen, die mit rekonfigurierbarer Hardware realisiert wurden, sind in [21, 32, 68, 74] aufgeführt.

In Abschnitt 2.1 werden die im Bereich der rekonfigurierbaren Hardware vorhandenen Architekturkonzepte beschrieben. Dabei ergeben sich unterschiedliche Granularitätsebenen, die sich grundsätzlich in grobgranulare und feingranulare Architekturen unterteilen lassen. Des Weiteren wird auf die unterschiedlichen Varianten der Systemanbindung von rekonfigurierbarer Hardware, ebenso wie auf die verschiedenen Konfigurationsansätze eingegangen. Abschnitt 2.2 behandelt die spezielle Klasse der partiell rekonfigurierbaren Architekturen. Im Hinblick auf das Konzept der partiellen dynamischen Rekonfiguration ergeben sich zweidimensionale und eindimensionale Systemansätze, sowie Systemansätze mit fester Aufteilung. Auf die Unterschiede der Ansätze wird im Einzelnen eingegangen.

2.1 Architekturkonzepte

Im Zusammenhang mit rekonfigurierbarer Hardware gibt es eine Vielzahl von verschiedenen Architekturkonzepten. Die wesentlichen Unterschiede ergeben sich in der Betrachtung der Granularitätsebene, der Systemanbindung und des Konfigurationsansatzes. In diesem Abschnitt wird auf die Unterschiede der einzelnen Architekturkonzepte eingegangen.

2.1.1 Granularitätsebenen

Rekonfigurierbare Architekturen lassen sich grundsätzlich in *feingranulare* und *grobgranulare* rekonfigurierbare Architekturen unterscheiden. Feingranulare rekonfigurierbare Architekturen zeichnen sich durch eine große Anzahl an Zellen aus, die aus einfachen Logikblöcken mit einer geringen Anzahl an Ein- und Ausgängen bestehen. Um eine beliebige Verschaltung der einzelnen Zellen zu ermöglichen, sind in feingranularen rekonfigurierbaren Architekturen eine Vielzahl von unterschiedlichen Verbindungsressourcen vorhanden, die meist in horizontaler und vertikaler Richtung verlaufen. Die Zellen sind üblicherweise über eine Switch-Matrix mit der Verbindungsstruktur angeschlossen. Ein Beispiel einer feingranularen Architektur ist das FPGA. Typische Beispiele sind die Xilinx Virtex FPGAs [88] und Altera Stratix FPGAs [5], deren Zellen aus Look-Up-Tables (LUT) mit einem 4-bit-Eingang und einem 1-bit-Ausgang bestehen. Weitere Beispiele feingranularer Architekturen sind GARP [40] ($4 \times$ 2-bit-Eingang, 2-bit-Ausgang) und LP-PGA [33] (3-bit-Eingang, 1-bit-Ausgang).

Die Zellen grobgranularer rekonfigurierbarer Architekturen sind typischerweise viel größer als die Zellen feingranularer Architekturen und bestehen meist aus arithmetischen Logikeinheiten (ALUs) mit Wortbreiten von 8-bit oder größer. Im Ver-

gleich zu feingranularen rekonfigurierbaren Architekturen ist die Anzahl der Zellen wesentlich geringer. Ebenso ist die Kommunikationsinfrastruktur einer grobgranularen rekonfigurierbaren Architektur meist nicht so flexibel und weniger umfangreich. Die geringe Anzahl an Zellen und die entsprechend geringere Anzahl an rekonfigurierbaren Kommunikationsressourcen führt zu einer geringen Menge an Konfigurationsdaten, so dass der Konfigurationsprozess einer grobgranularen rekonfigurierbaren Architektur oft kürzer ist als der einer feingranularen. Beispiele grobgranularer rekonfigurierbarer Architekturen sind RaPiD [27], MATRIX [57], Piperench [35], DReAM [9], Totem [20], PACT-XPP [7] und IMEC ADRES [56].

Manche Architekturen lassen sich jedoch nicht eindeutig den fein- oder grobgranularen rekonfigurierbaren Architekturen zuordnen. Daher ist zusätzlich die Klasse der *mittel-* oder *mediumgranularen* rekonfigurierbaren Architekturen entstanden. Ein Beispiel einer mediumgranularen Architektur stellt die CHESS-Architektur [55] dar, deren Zellen aus 4-bit-ALUs bestehen, die über eine 4-bit-Busarchitektur miteinander verbunden sind.

Um die Ausführung bestimmter Anwendungen zu optimieren, sind in den heutigen rekonfigurierbaren Architekturen verschiedenartige Zellen mit unterschiedlichen Größen integriert, so dass eine heterogene Struktur entsteht. So bestehen etwa die Xilinx Virtex-II Pro FPGAs [93] nicht nur aus Logikzellen, sondern ebenfalls aus eingebetteten Prozessoren und Speicherblöcken. Im Vergleich zu den Logikzellen sind in Virtex-II FPGAs nur eine geringe Anzahl von Speicherblöcken vorhanden, die nicht gleichmäßig sondern in gesonderten Spalten angeordnet sind. Diese Form der heterogenen rekonfigurierbaren Architekturen lässt sich nicht mehr eindeutig einer der genannten Granularitätsebenen zuordnen, so dass heterogene Architekturen daher auch als *multigranulare* Architekturen bezeichnet werden können.

2.1.2 Systemanbindung

Ein System mit rekonfigurierbarer Hardware besteht typischerweise aus mindestens einem Prozessor, einem oder mehreren Blöcken mit rekonfigurierbarer Hardware und Speicherelementen. Wie von Compton und Hauck in [21] beschrieben, lassen sich Systeme mit rekonfigurierbarer Hardware anhand der verschiedenen Arten der Prozessorkopplung klassifizieren. Eine Übersicht der verschiedenen Arten der Prozessorkopplung ist in Abbildung 2.1 dargestellt.

Innerhalb des Systems kann rekonfigurierbare Hardware als eigenständige externe Verarbeitungseinheit angebunden sein. Die Kommunikation zwischen dem Prozessor und der rekonfigurierbaren Hardware erfolgt dabei über die Ein-/Ausgabe-Schnittstelle. Bei dieser Art der Prozessorkopplung wird von einer unregelmäßigen Kommunikation ausgegangen, bei der geringe Datenmengen ausgetauscht werden oder die rekonfigurierbare Hardware zeitintensive Berechnungen autonom durchführt, ohne dabei mit dem Prozessor zu kommunizieren.

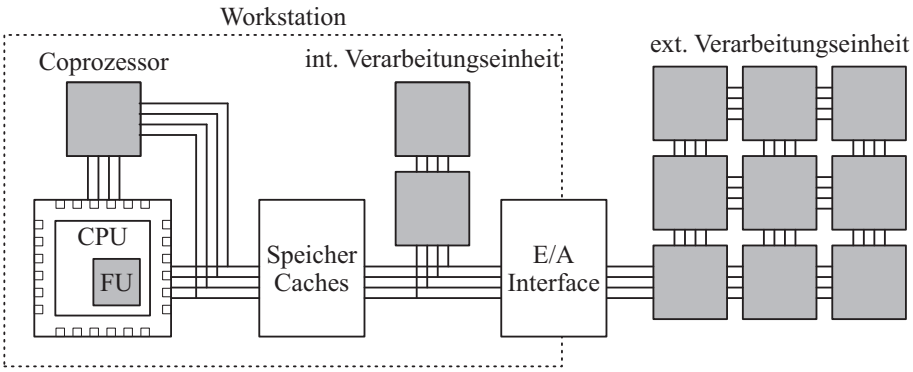


Abbildung 2.1: Verschiedene Arten der Prozessorkopplung in Systemen mit rekonfigurierbarer Hardware [21].

Rekonfigurierbare Hardware kann auch als interne Verarbeitungseinheit angebunden sein, so dass die rekonfigurierbare Hardware wie in einem Multi-Prozessorsystem als zusätzlicher Prozessor betrachtet werden kann. Die Kommunikation zwischen Prozessor und rekonfigurierbarer Hardware erfolgt dabei über den internen Prozessorbus oder den internen Speicher. Beispiele einer solchen Kopplung sind PCI-PipeRench [51] und PAM [80].

Eine noch engere Kopplung ist dann gegeben, wenn die rekonfigurierbare Hardware in Form eines Koprozessors mit dem Prozessor verbunden ist. Der Prozessor und die rekonfigurierbare Hardware können auf diese Art direkt oder indirekt über Speicher miteinander kommunizieren. Die rekonfigurierbare Hardware führt dabei die Berechnung unabhängig vom Prozessor durch. Beispiele von Koprozessorkopplungen sind GARP [40], REMARC [58] und NAPA [62].

Die engste Kopplung wird erreicht, wenn ein Block mit rekonfigurierbarer Hardware in dem Prozessor eingebettet ist (engl.: Reconfigurable Function Unit (RFU)). In diesem Fall wird die rekonfigurierbare Hardware Teil des Prozessors und kann genutzt werden, um den Befehlssatz des Prozessors mit anwendungsspezifischen Instruktionen zu erweitern. Die Kommunikation zwischen Prozessor und rekonfigurierbarer Hardware geschieht direkt über die internen Register des Prozessors. Beispiele dieser Art von Prozessorarchitekturen sind OneChip [16], Chimaera [39], FIP [66] und DISC [84].

Eine ähnliche enge Kopplungsvariante ergibt sich, wenn der Prozessor in einer rekonfigurierbaren Architektur eingebettet ist (vgl. [75]). Der Prozessor kann dabei als *Hardmacro*[1] realisiert sein, wie z. B. der eingebettete IBM PowerPC 405 in Xilinx Virtex-II/Pro FPGAs [93] oder der ARM 922T in Altera Excalibur Bausteinen [3]. Der Prozessor kann ebenso als so genannter *Softcore-Prozessor* in Form von Konfigurationsdaten vorliegen. Zu Beginn wird die rekonfigurierbare Hardware mit den

[1] In vielen Arbeiten wird anstelle des Begriffs *Hardmacro* der Begriff *Hardcore* verwendet, der in dieser Arbeit jedoch aus Gründen der Doppeldeutigkeit vermieden wird.

Konfigurationsdaten des Softcore-Prozessors initialisiert. Beispiele solcher Softcore-Prozessoren sind Xilinx Microblaze [94], Altera Nios II [4] und S-Core [50].

2.1.3 Konfigurationsansätze

Die meisten der derzeit verwendeten rekonfigurierbaren Architekturen besitzen einen SRAM-basierten Konfigurationsspeicher, der auf unterschiedliche Weise aufgebaut ist und adressiert werden kann. Daher ergeben sich verschiedene Konfigurationsansätze, die sich wie in [21] dargestellt in *Single-Kontext-Architekturen*, *Multi-Kontext-Architekturen* und *partiell rekonfigurierbare Architekturen* klassifizieren lassen.

Single-Kontext-Architekturen sind seriell rekonfigurierbare Architekturen, die sich nur komplett rekonfigurieren lassen. Beim Konfigurationsprozess müssen immer alle Speicherstellen des Konfigurationsspeichers beschrieben werden. D. h., die Konfigurationsdaten beinhalten die Konfigurationsbits aller rekonfigurierbaren Ressourcen (Zellfunktionen und Verbindungen) der Architektur und stellen somit einen *Kontext* der Architektur dar. Single-Kontext-Architekturen besitzen eine einfache Konfigurationsschnittstelle, da die Reihenfolge der Konfigurationsbits in den Konfigurationsdaten gegeben ist und keine Adressinformationen verarbeitet werden müssen. Während des Konfigurationsprozesses kann die Architektur nicht aktiv genutzt werden. Auch bei geringen Unterschieden zwischen der neuen und der aktuellen Konfiguration wird immer ein kompletter Konfigurationsprozess benötigt, so dass Single-Kontext-Architekturen nur eingeschränkt für Systeme mit dynamischer Rekonfiguration zur Laufzeit geeignet sind. Beispiele von Single-Kontext-Architekturen sind Altera Stratix FPGAs [5].

Ebenso wie Single-Kontext-Architekturen sind Multi-Kontext-Architekturen seriell rekonfigurierbar und erlauben ebenfalls nur eine komplette Rekonfiguration. Jedoch können im Gegensatz zu den Single-Kontext-Architekturen mehrere Kontexte parallel gespeichert werden, zwischen denen beliebig umgeschaltet werden kann. Der Umschaltvorgang ist vergleichbar mit dem Umschalten eines Multiplexers, so dass innerhalb von wenigen Nanosekunden ein anderer Kontext aktiviert werden kann. Ein wesentlicher Vorteil dieses Ansatzes ist das Konfigurieren von neuen Kontexten, während ein anderer Kontext aktiv ist. Dadurch wird die Ausführung des aktiven Kontextes nicht durch einen Konfigurationsprozess unterbrochen. Multi-Kontext-Architekturen sind damit für Systeme mit dynamischer Rekonfiguration zur Laufzeit geeignet. Die Verteilung der Hardware-Module in den einzelnen Kontexten stellt jedoch ein Partitionierungsproblem dar, bei dem sichergestellt werden muss, dass Hardware-Module, die zur gleichen Zeit benötigt werden, auch in den gleichen Kontexten vorhanden sind. Beispiele von Multi-Kontext-Architekturen sind DPGA [24], CSRC FPGA [63] und Time-Multiplexed FPGA [77].

Während bei Single- und Multi-Kontext-Architekturen immer die komplette Rekonfiguration eines Kontextes erforderlich ist, erlauben partiell rekonfigurierbare Architekturen die Rekonfiguration eines Teils der Architektur, ohne dabei die aktiven

Berechnungen des restlichen Teils zu beeinflussen. Die in den folgenden Kapiteln beschriebenen Methoden und Konzepte beziehen sich ausschließlich auf partiell rekonfigurierbare Architekturen, so dass im folgenden Abschnitt detailliert auf das Prinzip der partiellen Rekonfigurierbarkeit eingegangen wird.

2.2 Partiell rekonfigurierbare Architekturen

Partiell rekonfigurierbare Architekturen erlauben die gezielte Rekonfiguration eines Teils der Architektur, während der verbleibende Teil seine bisherige Funktion unverändert weiter ausführt. Auf diese Weise lassen sich bei Bedarf Systemkomponenten in Form von Hardware-Modulen zur Laufzeit auf der Architektur platzieren und wieder entfernen. Der Konfigurationsspeicher von partiell rekonfigurierbaren Architekturen ist mit einem konventionellen SRAM-Speicher vergleichbar. Die Konfigurationsdaten eines Hardware-Moduls beinhalten die Konfigurationsbits und die entsprechenden Adressen der benötigten rekonfigurierbaren Ressourcen (Zellfunktionen und Verbindungen). Die Adressen spiegeln somit die Positionen der einzelnen Ressourcen wider. Der Konfigurationsprozess eines Hardware-Moduls entspricht dem sequenziellen Schreiben der Konfigurationsdaten in dem Konfigurationsspeicher. Durch Adressverschiebungen kann das Hardware-Modul an unterschiedlichen Positionen platziert werden. Das gezielte Rekonfigurieren einzelner Bereiche der rekonfigurierbaren Architektur reduziert die Konfigurationsdatenmenge und ebenso die benötigte Konfigurationszeit erheblich.

Das Konzept der partiellen Rekonfigurierbarkeit wird - wie in [65] dargestellt - in Anwendungen in den Bereichen Software-Defined-Radio und Videodatenverarbeitung eingesetzt. Darüber hinaus gibt es bis jetzt jedoch nur wenige Anwendungen, die das Konzept der partiellen Rekonfigurierbarkeit nutzen, was im Wesentlichen auf den Mangel an Entwurfswerkzeugen zurückzuführen ist. Erste Methoden zur Unterstützung partieller Rekonfigurierbarkeit sind in den heutigen EDA-Werkzeugen integriert. Dennoch fehlt es an einem ganzheitlichen Systemansatz, der die einfache Nutzung partieller Rekonfigurierbarkeit ermöglicht.

Die uneingeschränkte Platzierung von Hardware-Modulen beliebiger Form bietet die größte Flexibilität hinsichtlich der Belegung der zur Verfügung stehenden Hardware-Ressourcen. Jedoch verursacht eine solch hohe Flexibilität ebenso eine hohe Komplexität in der Verwaltung der freien und belegten Ressourcen zur Laufzeit. Mit zunehmenden Einschränkungen in der Form und den Platzierungsmöglichkeiten der gegebenen Hardware-Module sinkt die Komplexität der Ressourcenverwaltung. Mit der Komplexität sinkt jedoch ebenso die Flexibilität des Systemansatzes. Bezüglich der Eigenschaft der partiellen Rekonfigurierbarkeit ergeben sich somit die folgenden Fragen:

- Wie flexibel muss ein Systemansatz im Hinblick auf partielle Rekonfigurierbarkeit sein?

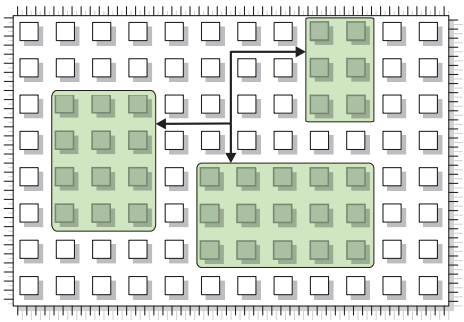


Abbildung 2.2: 2D-Systemansatz für partiell rekonfigurierbare Architekturen.

- Welchen Einfluss haben die Verfahren zur Verwaltung der freien und belegten Ressourcen auf die Leistungsfähigkeit eines Systemansatzes?

Der unterschiedliche Grad an Flexibilität wird in den folgenden Abschnitten anhand von drei Systemansätzen verdeutlicht. Die dargestellten Systemansätze spiegeln zu gleich die grundsätzlichen Möglichkeiten der Umsetzung eines Systems mit partiell rekonfigurierbarer Hardware wider.

2.2.1 Zweidimensionale Systemansätze

Im zweidimensionalen Systemansatz haben die zur Laufzeit platzierten Hardware-Module eine rechteckige Form und können beliebig auf der rekonfigurierbaren Architektur platziert werden. Abbildung 2.2 zeigt das Prinzip des 2D-Systemansatzes. Derzeit gibt es nur wenige Architekturen, die sich zweidimensional partiell rekonfigurieren lassen. Beispiel einer solchen Architektur ist das Xilinx XC6200 FPGA [86]. Der in Bazargan et al. [8] und Diessel et al. [25] dargestellte 2D-Systemansatz basiert auf dem Ansatz von Brebner [13], bei dem von einem System mit einem Prozessor und damit verbundener partiell rekonfigurierbarer Hardware ausgegangen wird. Das System verwendet ein Betriebssystem, dessen Tasks zur Laufzeit entweder als Hardware-Modul in rekonfigurierbarer Hardware oder als Software-Prozess des Prozessors implementiert werden kann. In dem Ansatz wird davon ausgegangen, dass keine Kommunikation zwischen den Tasks stattfindet. Wenn ein Hardware-Modul aus Mangel an freien Ressourcen nicht platziert werden kann, wird anstelle des Hardware-Moduls ein entsprechender Software-Prozess gestartet. In Steiger et al. [72] wird der gleiche Ansatz verwendet, wobei sich innerhalb der rekonfigurierbaren Hardware zusätzlich ein statischer Bereich befindet, der für Betriebssystemdienste reserviert ist.

Es ergeben sich grundsätzlich zwei Möglichkeiten, um eine Kommunikation zwischen den platzierten Hardware-Modulen zu ermöglichen. Die erste Möglichkeit ist die Realisierung einer statischen Kommunikationsinfrastruktur. Bestimmte Routingressourcen werden daher reserviert und von den Hardware-Modulen ausschließlich

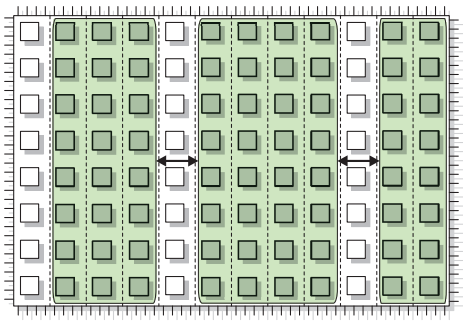


Abbildung 2.3: 1D-Systemansatz für partiell rekonfigurierbare Architekturen.

zur Kommunikation untereinander genutzt. Ein solcher Ansatz wird in Sedcole et al. [65] für Xilinx Virtex FPGAs vorgestellt. Jedoch lassen sich Hardware-Module bei diesem Ansatz nur an den Positionen platzieren, die eine Verbindung mit der Kommunikationsinfrastruktur ermöglichen. Die Flexibilität der Platzierung wird daher eingeschränkt. Die zweite Möglichkeit der Realisierung einer Kommunikation zwischen den Hardware-Modulen ist die Anpassung der Routingressourcen zur Laufzeit mit dem Ziel, die derzeit platzierten Hardware-Module miteinander zu verbinden. Hübner et al. haben in [41] einen Ansatz für Xilinx Virtex FPGAs vorgestellt, welcher die Hardware-Module über rekonfigurierbare vertikale Routingkanäle mit einer statischen horizontalen Kommunikationsinfrastruktur verbindet. Die Hardware-Module können vertikal beliebig entlang des Routingkanals verschoben werden. Um die Verbindung eines Hardware-Moduls mit dem Routingkanal zur Laufzeit herzustellen, muss der Routingkanal entsprechend rekonfiguriert werden. Im Gegensatz zu einer statischen Kommunikationsinfrastruktur wird daher ein zusätzlicher Konfigurationsaufwand benötigt. Jedoch bietet die Verwendung rekonfigurierbarer Routingkanäle eine größere Flexibilität in der Modulplatzierung.

Im 2D-Systemansatz gibt es im Allgemeinen keine Einschränkungen bezüglich der Seitenverhältnisse der Hardware-Module. Wenn jedoch die Höhe der Hardware-Module konstant gehalten wird, so dass die Module entsprechend ihrer Komplexität nur in der Breite variieren, dann ergibt sich der *1D-Systemansatz*, der als Sonderfall des 2D-Systemansatzes betrachtet werden kann und im folgenden Abschnitt näher erläutert wird.

2.2.2 Eindimensionale Systemansätze

Viele der heutigen verfügbaren rekonfigurierbaren Architekturen lassen sich eindimensional partiell rekonfigurieren. Beispiele solcher Architekturen sind GARP [40], Chimaera [39], PipeRench [35, 64] und Xilinx Virtex FPGAs [89, 92, 93]. Aufgrund der spaltenweisen partiellen Rekonfigurierbarkeit ergibt sich der 1D-Systemansatz, welcher in Abbildung 2.3 dargestellt ist.

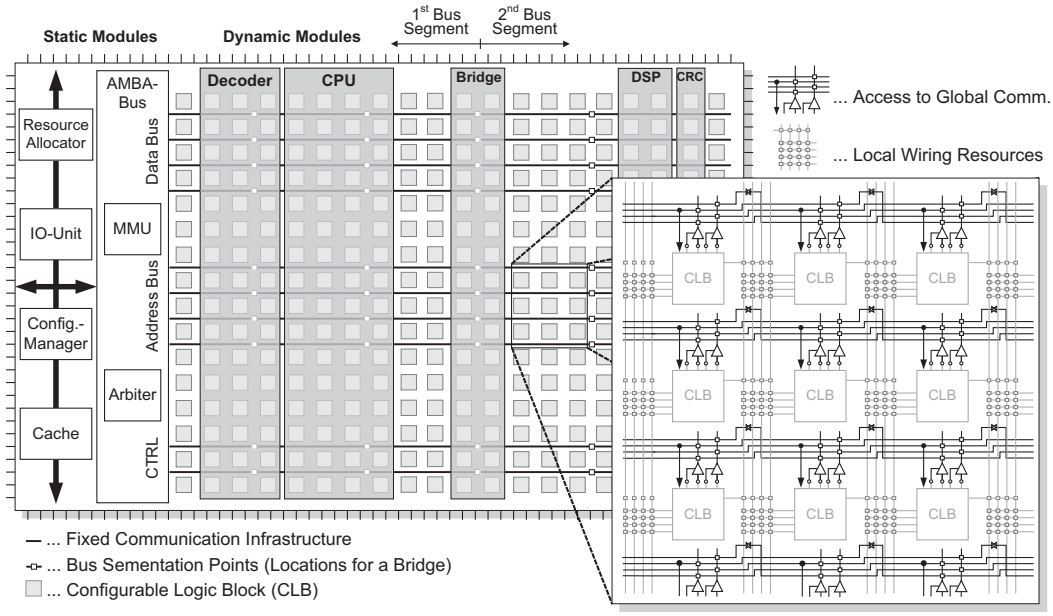


Abbildung 2.4: 1D-Systemansatz für Xilinx Virtex FPGAs [E1].

Eine der ersten Arbeiten, die sich mit dem 1D-Systemansatz für partiell rekonfigurierbare Hardware beschäftigt, ist Brebner et al. [15]. In dem beschriebenen 1D-Systemansatz haben alle Hardware-Module eine konstante Höhe und können lediglich in horizontaler Richtung platziert werden. Die Hardware-Module variieren in der Breite in Abhängigkeit ihrer Komplexität. Die Fläche innerhalb eines Hardware-Moduls wird vertikal in die Bereiche *Betriebssystemschaltung*, *Modulkommunikationsschaltung* und *Anwendungsschaltung* unterteilt. Die Betriebssystemschaltung beinhaltet Systemfunktionen wie die Verwaltung der Hardware-Module. Der Austausch von Daten zwischen den Hardware-Modulen geschieht durch die Modulkommunikationsschaltung. Die Anwendungsschaltung stellt den größten Teil des Hardware-Moduls dar und repräsentiert die eigentliche Anwendung des Hardware-Moduls.

Kalte et al. [E1] haben einen ähnlichen 1D-Systemansatz vorgestellt, der für Xilinx Virtex-FPGAs ausgelegt ist. Abbildung 2.4 zeigt einen Überblick des Ansatzes. Die rekonfigurierbare Architektur wird in einen statischen Teil (Static Modules) und einen dynamischen Teil (Dynamic Modules) partitioniert. Der statische Teil umfasst die Verwaltung der Konfigurationsdaten und der freien Ressourcen. Ebenso wird im statischen Teil die Kommunikation zwischen den Hardware-Modulen und den externen Komponenten organisiert. Hardware-Module können zur Laufzeit entlang der statischen horizontalen Kommunikationsinfrastruktur platziert werden und über einen Systembus miteinander kommunizieren. Bei der Konfiguration eines Hardware-Moduls kann die Position durch Manipulation der Konfigurationsdaten anhand des in [44] beschriebenen REPLICA-Filters beeinflusst werden. Die Hardware-Module sind der-

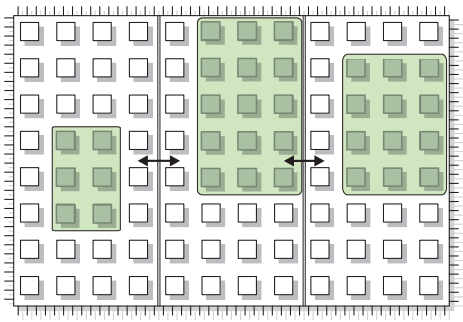


Abbildung 2.5: Systemansatz mit fester Aufteilung für partiell rekonfigurierbare Architekturen.

art synthetisiert, dass die zur Realisierung des Systembusses verwendeten Routingressourcen nicht anderweitig genutzt werden.

Im 1D-Systemansatz gibt es keine Einschränkungen bezüglich der Breite eines Hardware-Moduls. Wenn neben der Höhe auch die Breite der Hardware-Module konstant gehalten wird, so dass jedes Hardware-Modul die gleiche Fläche besitzt, dann ergibt sich der *Systemansatz mit fester Aufteilung*, der somit als Sonderfall des 1D-Systemansatzes betrachtet werden kann und im folgenden Abschnitt detailliert beschrieben wird.

2.2.3 Systemansatz mit fester Aufteilung

Im Systemansatz mit fester Aufteilung wird die rekonfigurierbare Architektur in eine feste Anzahl von Blöcken gleicher Größe unterteilt. Die Blöcke sind mit einer statischen Kommunikationsinfrastruktur miteinander verbunden. Zur Laufzeit werden Hardware-Module in entsprechende freie Blöcke platziert. Innerhalb eines Blocks sind die Schnittstellen zur Kommunikationsinfrastruktur an einer festen Position. Die Hardware-Module sind derart synthetisiert, dass sie sich mit der Kommunikationsinfrastruktur verbinden. Daher kann immer nur ein Hardware-Modul in einem Block konfiguriert werden. Einer der ersten Systemansätze mit fester Aufteilung wird von Brebner in [13, 14] beschrieben. Brebner stellt dabei die *Swappable Logic Unit* (SLU) als eine elementare rekonfigurierbare Einheit vor, die eine feste Größe und Position auf der rekonfigurierbaren Architektur besitzt. Eine Anwendung kann dabei nicht nur aus einer, sondern auch aus mehreren SLUs aufgebaut sein. Die SLU kann somit als rekonfigurierbarer Block betrachtet werden.

Ein Systemansatz mit fester Aufteilung für Xilinx Virtex FPGAs wird in [91] vorgestellt. Abbildung 2.6 zeigt eine Übersicht des Ansatzes mit zwei partiell rekonfigurierbaren Blöcken. Das FPGA wird dabei in statische Flächen (Fixed Logic) und dynamische Flächen (PR Logic), die die partiell rekonfigurierbaren Blöcke darstel-

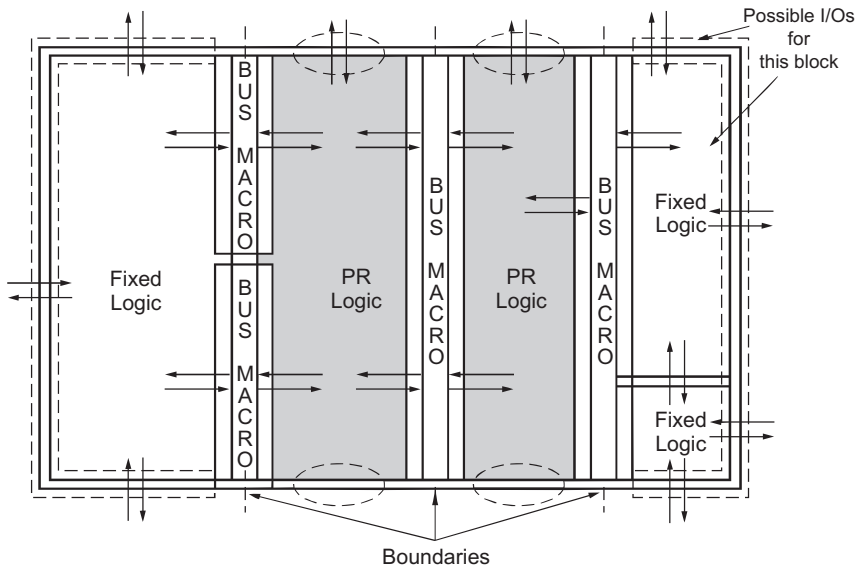


Abbildung 2.6: Systemansatz mit fester Aufteilung für Xilinx Virtex FPGAs [91].

len, unterteilt. Zur Kommunikation werden so genannte *Bus Macros* verwendet, die die Kommunikation zwischen benachbarten Blöcken ermöglicht.

Ein erweitertes System mit 4 partiell rekonfigurierbaren Blöcken wird in Ullman et al. [78] vorgestellt. Das System ist für Xilinx Virtex-II FPGAs ausgelegt und beinhaltet neben den 4 Blöcken den Softcore-Prozessor Microblaze. Die partiell rekonfigurierbaren Blöcke sind über einen Systembus mit dem Prozessor verbunden. Das System dient der Umsetzung elektronischer Funktionen im Automobilbereich und markiert damit zugleich ein konkretes Anwendungsbeispiel für partiell rekonfigurierbare Hardware.

Weitere Beispiele für Systemansätze mit fester Aufteilung sind SCORE [17], Caronte [31] und Erlangen Slot Machine [10].

3 Modellierung

Mit der steigenden Komplexität und Heterogenität heutiger partiell rekonfigurierbarer Architekturen bedarf es einem allgemein verwendbaren Modell, welches den Vergleich der verschiedenen Architekturen und deren Methoden zur Umsetzung partieller Rekonfiguration ermöglicht. Bestehende Modellierungsansätze (wie z. B. [8, 11, 49]) sind meist architekturspezifisch oder widmen sich speziellen Problemen der Rekonfigurierbarkeit. Die in diesem Kapitel vorgestellte Modellierung und deren grundlegende Mechanismen zur Platzierung und Ablaufplanung eignen sich für Architekturen beliebiger Granularitätsebenen und ermöglichen die Analyse der Architektur und deren Mechanismen zur Realisierung partieller Rekonfiguration auf hoher Abstraktionsebene. Die in diesem Kapitel beschriebenen Modelle und Verfahren gelten für die im Kapitel 4 erläuterten homogenen rekonfigurierbaren Architekturen ebenso wie für die im Kapitel 5 beschriebenen heterogenen rekonfigurierbaren Architekturen.

Grundlage der in dieser Arbeit definierten Methoden bildet das im Abschnitt 3.1 vorgestellte *DMC-Modell*[1]. Das Modell unterteilt die Platzierung eines Hardware-Moduls in die drei Abstraktionsebenen *Systemkomponente*, *Hardware-Modul* und *Modulinstanz*. Das DMC-Modell stellt den Zusammenhang der verschiedenen Abstraktionsebenen her und beschränkt sich auf die wesentlichen Größen, die für die Umsetzung partieller Rekonfigurierbarkeit zur Laufzeit erforderlich sind.

Mithilfe des DMC-Modells lassen sich Methoden zur partiellen Rekonfiguration, wie die Platzierung oder die Ablaufplanung, formal beschreiben. Im Abschnitt 3.2 werden verschiedene Verfahren zur Ablaufplanung diskutiert. Die Ablaufplanung wird dabei auf der Ebene der Platzierung und der Ebene des Konfigurationsprozesses betrachtet. Im Gegensatz zur Platzierung, bei der unterschiedliche Verfahren für homogene und heterogene Architekturen existieren, ist die Ablaufplanung - wie im Abschnitt 3.2 beschrieben - architekturunabhängig.

Eine analytische Bewertung von verschiedenen Architekturen und deren Methoden zur Umsetzung partieller Rekonfiguration ist wie im Folgenden dargelegt nur begrenzt möglich. Aus diesem Grund ist die im Abschnitt 3.3.1 vorgestellte Simulationsumgebung SARA (Simulationsumgebung zur Analyse Rekonfigurierbarer Architekturen) entwickelt worden, welche auf dem DMC-Modell basiert. Mithilfe von

[1] Die Abkürzung *DMC* leitet sich aus den Bezeichnungen der Mengen der Abstraktionsebenen her (*Systemkomponenten D*, *Hardware-Module M* und *Modulinstanzen C*).

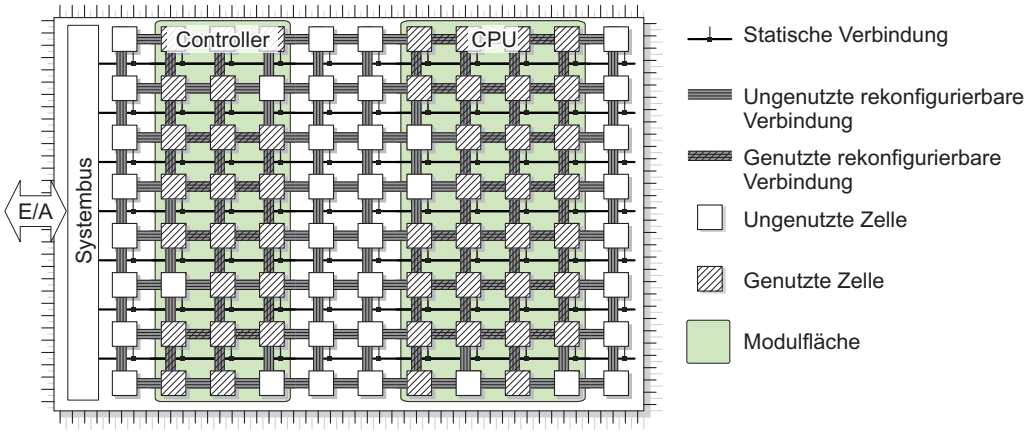


Abbildung 3.1: Schematische Darstellung einer rekonfigurierbaren Architektur mit statischen und rekonfigurierbaren Verbindungsressourcen.

SARA lassen sich verschiedene Systemansätze und entsprechende Platzierungsverfahren simulativ bewerten.

3.1 DMC-Modell

Das hier beschriebene Modell legt eine rekonfigurierbare Architektur (wie in Abbildung 3.1 dargestellt) zugrunde. Die Architektur besteht aus partiell rekonfigurierbaren Zellen, die in einem rechteckigen Feld angeordnet sind und über eine Verbindungsstruktur miteinander verbunden sind. Die Modellierung ermöglicht die Betrachtung feingranularer rekonfigurierbarer Architekturen, wie z. B. FPGAs, und grobgranularer rekonfigurierbarer Architekturen, wie z. B. RaPiD [27], PipeRench [35] oder PACT-XPP [7]. Im Folgenden werden jedoch lediglich feingranulare Architekturen betrachtet.

Die Architektur kann, wie in homogenen Architekturen üblich, aus Zellen eines Typs bestehen. Ein Beispiel für eine homogene Architektur ist ein FPGA, das lediglich aus konfigurierbaren Logikzellen besteht, wie das Xilinx XC6200 FPGA [86]. Ebenso lassen sich heterogene Architekturen mit verschiedenen Zelltypen betrachten. Ein Beispiel einer solchen heterogenen Architektur ist ein FPGA mit Logikzellen, Speicherblöcke (BlockRAM) und Zellen zur Signalverarbeitung, wie das Xilinx Virtex-4 FPGA [95]. Bezüglich der Modellierung gilt die Annahme, dass alle Zellen gleich groß sind, so dass die Architektur in Form einer Matrix repräsentiert werden kann.

Die Verbindungsressourcen der rekonfigurierbaren Architektur werden unterteilt in statische Verbindungsressourcen für die *systemweite Kommunikation* und rekonfigurierbare Verbindungsressourcen für die *modulinterne Kommunikation*. Die systemweite Verbindungsstruktur ermöglicht die Anbindung der zur Laufzeit partiell kon-

figurierten Hardware-Entwürfe an das System. Eine mögliche Form der systemweiten Kommunikation ist ein Systembus, wie in [47] beschrieben. Ein solcher Systembus ist ebenfalls in Abbildung 3.1 angedeutet. Eine Alternative ist die paketbasierte Kommunikation über ein Network-On-Chip, wie in [2, 54, 76] erläutert. Zur Laufzeit wird die für die modulinterne Kommunikation reservierte rekonfigurierbare Verbindungsstruktur anhand der Konfigurationsdaten des partiell zu rekonfigurierenden Hardware-Entwurfs angepasst. Abbildung 3.1 zeigt als Beispiel die konfigurierten Hardware-Entwürfe *Controller* und *CPU*.

Durch partielle Rekonfiguration ungenutzter Zellen und ungenutzter rekonfigurierbarer Verbindungsressourcen können Hardware-Entwürfe zur Laufzeit dem System hinzugefügt werden, indem die Konfigurationsdaten des Hardware-Entwurfs an die entsprechende Konfigurationsschnittstelle der Architektur gesendet werden. Bezüglich der Modellierung wird angenommen, dass der Konfigurationsvorgang sequenziell geschieht, d. h., es können nicht mehrere Hardware-Entwürfe zur gleichen Zeit konfiguriert werden.

Das im Folgenden vorgestellte *DMC-Modell* basiert auf der obigen spezifizierten rekonfigurierbaren Architektur und beschreibt die für die Platzierung relevanten Parameter auf den Ebenen *Systemkomponente*, *Hardware-Modul* und *Modulinstanz*. Abbildung 3.2 stellt den Zusammenhang der drei Abstraktionsebenen dar. Eine *Systemkomponente* $d \in D$ ist als abstrakte Spezifikation eines Hardware-Entwurfs zu verstehen. Eine solche Spezifikation kann in Form einer Hardware-Beschreibungssprache wie z. B. VHDL oder in Form einer schematischen Darstellung vorliegen. D sei die Menge aller dem System bekannten Systemkomponenten. Beim Hardware-Entwurf auf statischen Architekturen wird aus einer solchen Systemkomponente mithilfe eines Synthesewerkzeugs eine Implementierung mit fest zugewiesenen Ressourcen und entsprechend fest zugewiesenen Verbindungen erzeugt. Im Zusammenhang mit partiell rekonfigurierbaren Architekturen ergibt sich ein weiterer Schritt zwischen der Beschreibung und der endgültigen Implementierung, denn im Gegensatz zum Entwurf auf statischen Architekturen ist die endgültige Position der Implementierung zum Zeitpunkt der Synthese unbekannt. Bei der Layoutsynthese auf rekonfigurierbaren Architekturen erzeugt das Platzieren und Verdrahten nicht eine absolute, sondern eine relative Zuordnung der gegebenen Zellen und deren Verbindungen, denn die endgültige Position der Implementierung wird erst zur Laufzeit bestimmt. Das Ergebnis der Layoutsynthese ist somit eine vorsynthetisierte Implementierung der Systemkomponente, welche im Folgenden als *Hardware-Modul* (kurz Modul) bezeichnet wird.

Definition 3.1 (Hardware-Modul)**.** *Ein* Hardware-Modul $m = (d, r, a, q) \in M$ *ist eine vorsynthetisierte Implementierung einer Systemkomponente* $d \in D$*, wobei* $r \in \mathbb{N}^{N_{cell}}$ *die Anzahl der benötigten Ressourcen ist.* N_{cell} *gibt die Anzahl der verschiedenen Zelltypen an. Der Parameter* $a \in \mathbb{N}^{N_{sys}}$ *beschreibt die benötigte Fläche des Moduls auf der Architektur. Die Dimension* N_{sys} *des Parameters a entspricht dabei der Dimension des zugrunde liegenden Systemansatzes. Der Vektor* $q \in \mathbb{R}^{N_{cost}}$ *beinhaltet*

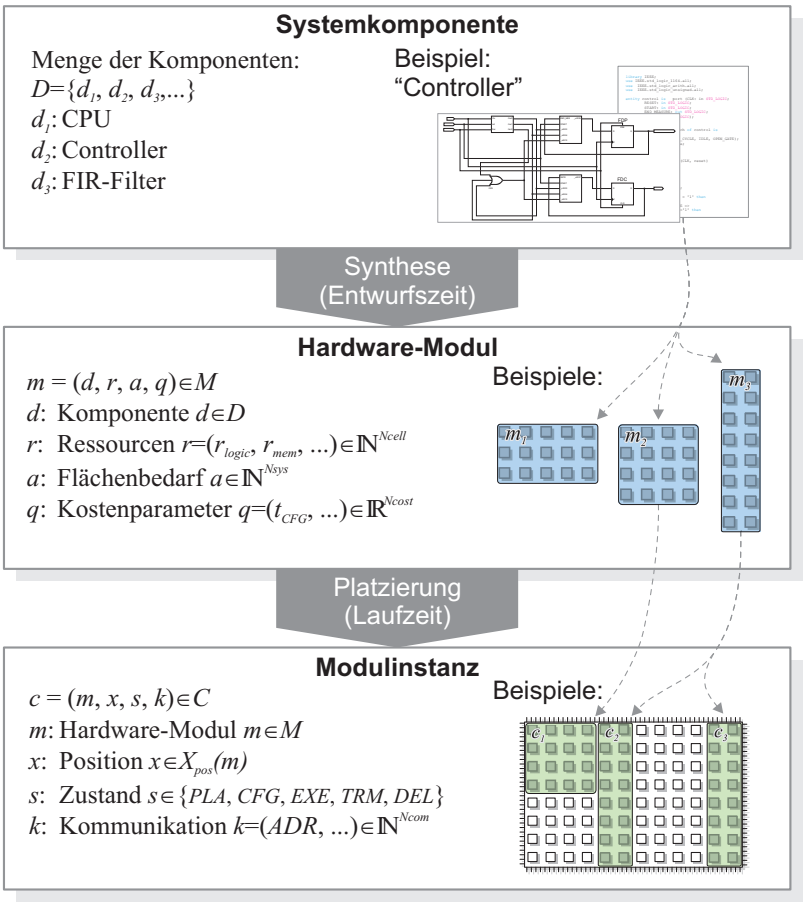


Abbildung 3.2: Übersicht des DMC-Modells.

Implementierungskosten. N_{cost} gibt die Anzahl der verschiedenen Kostenparameter wieder. M beschreibt die Menge aller dem System bekannten Module.

Die Ressourcen $r = (r_{CLB}, r_{MEM}, r_{MUL}, \ldots) \in \mathbb{N}^{N_{cell}}$ eines Hardware-Moduls $m = (d, r, a, q)$ beinhalten lediglich die Anzahl der benötigten Zellen. Verbindungsstrukturen und I/O-Blöcke werden hier nicht als Ressourcen betrachtet. Die Implementierungskosten $q \in \mathbb{R}^{N_{cost}}$ eines Hardware-Moduls $m = (d, r, a, q)$ können z. B. die Konfigurationszeit oder die Priorität des Moduls sein.

Wenn die übergeordnete Anwendung eine Systemkomponente, wie z. B. einen Controller, anfordert, dann ist die geometrische Form der resultierenden Implementierung und deren Position auf der rekonfigurierbaren Architektur für die Anwendung unbedeutend, solange jede Implementierung auf der gleichen Beschreibung aufbaut und eine einheitliche ortsunabhängige Kommunikationsinfrastruktur zugrunde liegt.

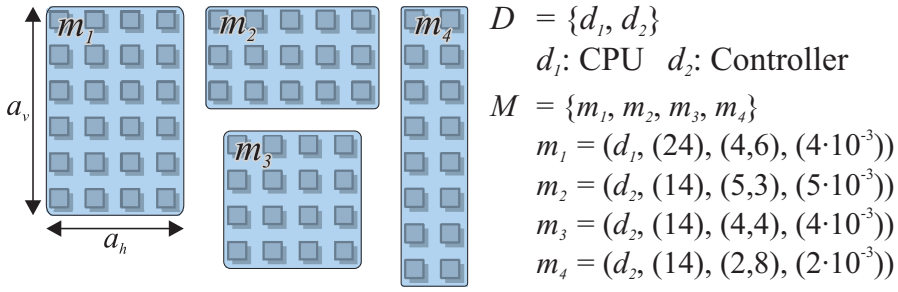


Abbildung 3.3: Beispiele verschiedener Hardware-Module.

Beispiel 3.1. *Abbildung 3.3 zeigt Beispiele verschiedener Hardware-Module in einem 2D-Systemansatz. Jedes Modul $m = (d, r, a, q) \in M$ basiert auf einer Systemkomponente $d \in D$, wobei die Menge der Systemkomponenten D aus einer CPU und einem Controller besteht. Der Ressourcenvektor $r = (r_{CLB})$ beschreibt die Anzahl der benötigten Logikzellen. Im 2D-Systemansatz ist der Flächenbedarf $a = (a_h, a_v) \in \mathbb{N}^2$. Der Kostenparameter $q = (t_{CFG})$ stellt die erforderliche Konfigurationszeit dar.*

Die Erzeugung eines Hardware-Moduls geschieht durch einmalige statische Implementierung der Systemkomponente zur Entwurfszeit. Aus dieser Implementierung werden die für das Hardware-Modul benötigten Konfigurationsdaten extrahiert und gespeichert. In Bezug auf FPGAs existiert somit zu jedem Hardware-Modul ein entsprechender partieller Konfigurationsdatenstrom. Mögliche Positionen für die Platzierung eines Hardware-Moduls ergeben sich überall dort, wo die gleichen Anordnungen der verwendeten Ressourcen und der modulinternen Verbindungsstruktur auftreten. Erst zur Laufzeit wird die Position eines Hardware-Moduls bezüglich der rekonfigurierbaren Architektur festgelegt und die entsprechenden Ressourcen zugewiesen. Der Prozess des Bestimmens einer Position wird im Folgenden als *Platzierung* bezeichnet.

Wird das Hardware-Modul an eine geeignete Position platziert, muss das Modul an die systemweite Verbindungsstruktur angebunden werden. Die Anbindung an die systemweite Verbindungsstruktur erfolgt durch Zuweisung einer individuellen Adresse. Das platzierte und adressierbare Hardware-Modul steht dem System individuell zur Verfügung und wird als *Modulinstanz* bezeichnet. Die Bildung einer Modulinstanz aus einem Hardware-Modul ist vergleichbar mit der Bildung einer Komponenteninstanz aus einer VHDL-Komponente im Zusammenhang mit der Hardware-Beschreibungssprache VHDL [43].

Definition 3.2 (Modulinstanz)**.** *Eine* Modulinstanz $c = (m, x, s, k) \in C$ *beschreibt ein auf der Architektur abgebildetes Modul $m \in M$, wobei $x \in X_{pos}(m)$ die Position der Instanz ist und $X_{pos}(m)$ die Menge der möglichen Positionen für das Modul m. Die Position beschreibt den Zellabstand der unteren linken Ecke der Modulinstanz zur unteren linken Ecke der Architektur. Die Dimension der Menge*

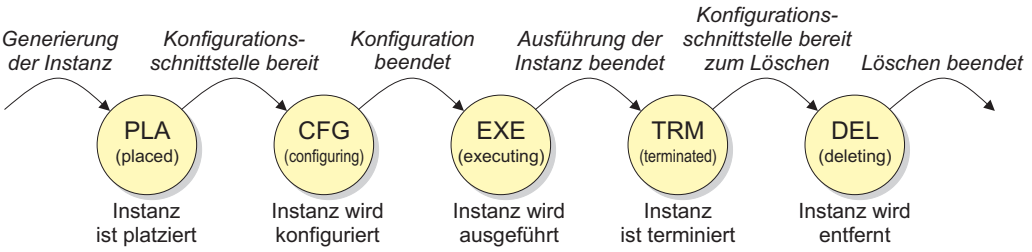


Abbildung 3.4: Zustände einer Modulinstanz und deren Transitionen.

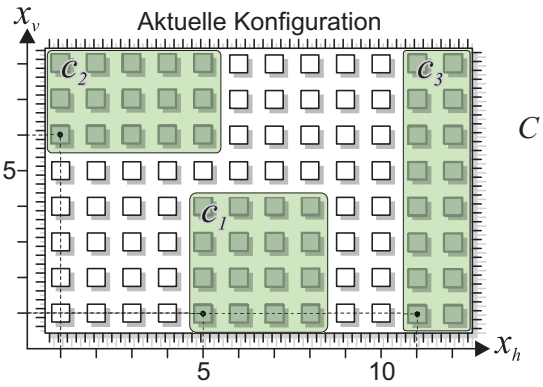
X_{pos} entspricht dabei der Dimension des Systemansatzes. Der Parameter $s \in S$ beschreibt den aktuellen Zustand der Modulinstanz, wobei die Menge der Zustände $S = \{PLA, CFG, EXE, TRM, DEL\}$ ist. Die Bedeutungen der Zustände und deren Transitionen werden im Abschnitt 3.1.1 näher erläutert. Der Vektor $k \in \mathbb{N}^{N_{com}}$ beinhaltet Kommunikationsparameter, wie z. B. die Adresse, die der Instanz zugewiesen ist. C stellt die Menge der momentan platzierten Modulinstanzen dar.

Wenn die systemweite Kommunikationsinfrastruktur aus einem Systembus besteht, so beinhaltet der Kommunikationsparameter eine der Modulinstanz dynamisch zugewiesene individuelle Adresse. Die dynamische Adresszuweisung ermöglicht die Platzierung mehrerer Instanzen eines gleichen Hardware-Moduls. In [36] werden verschiedene Verfahren zur Realisierung einer dynamischen Adresszuweisung bezüglich dynamisch rekonfigurierbarer Systeme vorgestellt. Ein einfaches Verfahren ist z. B. die positionsbasierte Adresszuweisung, d. h., die Adresse der Modulinstanz hängt von deren Position ab.

3.1.1 Zustände einer Modulinstanz

Von dem Zeitpunkt der Generierung bis zum Ende des Löschvorgangs durchläuft eine Modulinstanz mehrere Zustände. Gemäß Definition 3.2 sei die Zustandsmenge $S = \{PLA, CFG, EXE, TRM, DEL\}$. Abbildung 3.4 zeigt die einzelnen Zustände einer Modulinstanz und deren Transitionen.

Im Folgenden werden die Zustände im Einzelnen beschrieben, ausgehend von der Situation, dass die übergeordnete Anwendung eine bestimmte Systemkomponente $d \in D$ anfordert. Infolgedessen ermittelt der Platzierungsalgorithmus ein passendes Modul $m \in M(d)$ mit dazugehöriger Position $x \in X_{pos}(m)$ und generiert eine entsprechende Modulinstanz c, die der Menge C hinzugefügt wird. Die einzelnen Schritte der Platzierung werden im späteren Abschnitt 3.1.2 näher erläutert. Nach der Generierung befindet sich die Modulinstanz c in dem Zustand *PLA* (engl.: Placed). Die für die Ausführung erforderlichen Zellen sind nun der Modulinstanz zugewiesen. Die Zellen sind jedoch noch nicht konfiguriert und die Instanz verbleibt solange im Zu-



$C = \{c_1, c_2, c_3\}$
$c_1 = (m_3, (5,1), EXE, (1))$
$c_2 = (m_2, (1,6), CFG, (2))$
$c_3 = (m_4, (11,1), PLA, (3))$

Abbildung 3.5: Beispiele verschiedener Modulinstanzen.

stand *PLA*, bis die Konfigurationsschnittstelle des Systems für die Konfiguration der Zellen verfügbar ist.

Mit Beginn des Konfigurationsprozesses wechselt die Instanz in den Zustand CFG (engl.: Configuring) und verbleibt dort solange, bis alle Zellen konfiguriert sind. Nach Beendigung des Konfigurationsvorgangs wechselt die Modulinstanz in den Zustand EXE (engl.: Executing). Die Instanz ist nun aktiv und führt die geforderte Funktion aus. Nachdem die Ausführung der Modulinstanz beendet ist, wechselt die Instanz in den Zustand TRM (engl.: Terminated). Die der Instanz zugewiesenen Zellen werden nun nicht mehr benötigt und müssen durch Rekonfiguration in einen initialen Zustand zurückgesetzt werden. Das Zurücksetzen der Zellen ist erforderlich, um ein unvorhersehbares Fehlverhalten der Zellen und deren Verbindungsstruktur bei nachfolgenden Konfigurationsprozessen zu vermeiden. Ähnlich wie im Zustand *PLA* verbleibt die Instanz solange im Zustand TRM, bis die Konfigurationsschnittstelle des Systems für das Zurücksetzen der Zellen verfügbar ist. Mit Beginn des Zurücksetzens der Zellen, was im Folgenden auch als *Löschvorgang* bezeichnet wird, wechselt die Instanz in den Zustand DEL (engl.: Deleting). Nach Beendigung des Löschvorgangs wird die Instanz aus der Menge der platzierten Modulinstanzen C entfernt.

Beispiel 3.2. *Abbildung 3.5 zeigt Beispiele verschiedener Modulinstanzen in einem 2D-Systemansatz. Jede Instanz $c = (m,x,s,k) \in C$ basiert auf einem Hardware-Modul $m \in M$, wobei M der Menge der Module aus Beispiel 3.1 entspricht. Im 2D-Systemansatz ist die Position $x = (x_h,x_v) \in X_{pos}(m)$. Der Kommunikationsparameter $k = (ADR)$ stellt die Adresse der Instanz dar. Während die Instanzen c_1 im Zustand der Ausführung ($s=EXE$) ist, befindet sich die Instanz c_2 im Zustand der Konfiguration ($s=CFG$). Die Instanz c_3 ist bereits platziert ($s=PLA$) und wartet auf die Freigabe der Konfigurationsschnittstelle, die derzeit noch mit der Konfiguration von c_2 beschäftigt ist.*

Auf die hier beschriebene Weise durchläuft jede Modulinstanz die verschiedenen Zustände der Zustandsmenge S. Ausgangspunkt ist dabei immer die Generierung der

Modulinstanz, welche das Ergebnis der erfolgreichen Platzierung eines Moduls ist. Die einzelnen Schritte der Platzierung eines Moduls werden im nachfolgenden Abschnitt näher erläutert.

3.1.2 Platzierung eines Moduls

Im Folgenden wird die Platzierung eines Moduls anhand des DMC-Modells erläutert. Wie im Abschnitt 2.2.1 beschrieben, stellen der Systemansatz mit fester Aufteilung und der 1D-Systemansatz eine Untermenge des 2D-Systemansatzes dar. Daher wird im Folgenden die Platzierung im allgemein gültigen 2D-Systemansatz angegeben. Im 2D-Systemansatz variieren die Flächenausdehnung eines Moduls und die Position einer Instanz in horizontaler und vertikaler Richtung. Die Fläche eines Moduls ist daher durch $(a_h, a_v) \in \mathbb{N}^2$ definiert, wobei a_h die Flächenausdehnung in horizontaler Richtung und a_v die Flächenausdehnung in vertikaler Richtung angibt. In Analogie zur Fläche wird die Position einer Instanz durch $(x_h, x_v) \in X_{pos}(m)$ beschrieben, wobei x_h die horizontale Position und x_v die vertikale Position darstellt.

Welche Zellen der rekonfigurierbaren Architektur belegt oder frei sind, lässt sich anhand der *Zellbelegung* $b(i, j)$ erkennen.

Definition 3.3 (Zellbelegung)**.** *Sei N_{col} die Anzahl der Spalten und N_{row} die Anzahl der Reihen der rekonfigurierbaren Architektur. Die* Zellbelegung $b(i, j)$ *mit $1 \leq i \leq N_{col}$ und $1 \leq j \leq N_{row}$ stellt die aktuelle Belegung der rekonfigurierbaren Architektur dar. Ist $b(i, j) = 0$, so wird die Zelle an der Position (i, j) von einer vorhandenen Modulinstanz $c \in C$ genutzt. Ist $b(i, j) = 1$, so ist die Zelle an der Position (i, j) frei. Gegeben sei die horizontale Position $x_h(c)$ und die vertikale Position $x_v(c)$ einer vorhandenen Instanz $c \in C$. Ferner sei $m(c)$ das zugrunde liegende Modul $m \in M$, $a_h(m)$ die Breite des Moduls und $a_v(m)$ die Höhe des Moduls. Dann ergibt sich die Zellbelegung $b(i, j)$ wie folgt:*

$$b(i,j) = \begin{cases} 0 & \text{wenn } \exists c \in C \mid x_h(c) \leq i < x_h(c) + a_h(m(c)) \wedge \\ & \qquad x_v(c) \leq j < x_v(c) + a_v(m(c)) \\ 1 & \text{sonst .} \end{cases} \tag{3.1}$$

Anhand der Zellbelegung $b(i, j)$ lässt sich die *Menge der freien Positionen* $X_{free}(m)$ eines Moduls m bestimmen an den das Modul platziert werden kann.

Definition 3.4 (Menge der freien Positionen)**.** *Gegeben sei die Zellbelegung $b(i, j)$ der rekonfigurierbaren Architektur. Dann ist die* Menge der freien Positionen $X_{free}(m)$ *eines Moduls m wie folgt definiert:*

$$X_{free}(m) = \left\{ (x_h, x_v) \mid (x_h, x_v) \in X_{pos}(m) \wedge \sum_{i=x_h}^{n_h} \sum_{j=x_v}^{n_v} b(i,j) = a_h(m) \cdot a_v(m) \right\} \tag{3.2}$$

$$n_h = x_h + a_h(m) - 1 \,, \qquad n_v = x_v + a_v(m) - 1$$

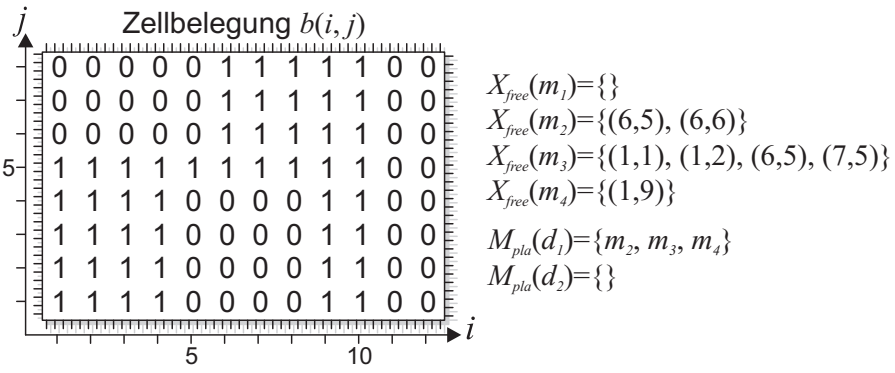


Abbildung 3.6: Beispiel einer Zellbelegung mit der Menge der freien Positionen und der resultierenden Menge der platzierbaren Module.

Mithilfe der Menge der freien Positionen lässt sich die *Menge der platzierbaren Module* $M_{pla}(d)$, die auf der geforderten Systemkomponente d basieren, bestimmen.

Definition 3.5 (Menge der platzierbaren Module)**.** *Es sei $M(d) \subseteq M$ die Menge der Module, die auf der Systemkomponente d basieren. Dann ist die* Menge der platzierbaren Module $M_{pla}(d)$ *einer Systemkomponente d wie folgt definiert:*

$$M_{pla}(d) = \{m \mid m \in M(d) \wedge X_{free}(m) \neq \{\}\} \tag{3.3}$$

Abbildung 3.6 zeigt ein Beispiel der Zellbelegung $b(i, j)$ mit der Menge der freien Positionen $X_{free}(m)$ der einzelnen Module und der resultierenden Menge der platzierbaren Module $M_{pla}(d)$ der Systemkomponenten. Mithilfe der Mengen $X_{free}(m)$ und $M_{pla}(d)$ kann die Platzierung eines Moduls in den folgenden vier Schritten beschrieben werden:

1. Die Grundvoraussetzung für die Platzierung eines Moduls der geforderten Systemkomponente d ist das Vorhandensein hinreichend freier Ressourcen auf der Architektur. Dabei muss folgende Bedingung erfüllt sein:

   $$M_{pla}(d) \neq \{\} \tag{3.4}$$

   D. h., es muss mindestens ein Modul der geforderten Systemkomponente d auf der Architektur platzierbar sein. Ist diese Bedingung nicht erfüllt, so kann die Platzierung abgewiesen oder so lange hinausgezögert werden, bis durch das Entfernen von ausgeführten Modulinstanzen hinreichend viele Ressourcen verfügbar sind. Eine weitere Möglichkeit, die Platzierung dennoch zu ermöglichen, ist die Neuanordnung der bereits platzierten Modulinstanzen mit dem Ziel, ausreichend freie zusammenhängende Ressourcen zu erlangen, die die Platzierung eines Moduls ermöglichen. Diese Art der Umplatzierung wird auch als *Defragmentierung* bezeichnet und wird im Abschnitt 4.3 näher erläutert.

2. Wenn die obige Platzierungsbedingung erfüllt ist, so wird anhand eines Platzierungsalgorithmus ein geeignetes Modul $m \in M(d)$ mit einer entsprechenden Position $x = (x_h, x_v) \in X_{free}(m)$ bestimmt. Die hierbei verwendeten Platzierungsalgorithmen lassen sich grundsätzlich in Algorithmen für homogene Architekturen (siehe Abschnitt 4.1) und Algorithmen für heterogene Architekturen (siehe Abschnitt 5.1) unterteilen.

3. Mithilfe des aus dem vorherigen Schritt hervorgegangenen Moduls m und der dazugehörigen Position $x = (x_h, x_v)$ wird eine Modulinstanz $c = (m, x, s, k)$ erzeugt. Wie im Abschnitt 3.1.1 erläutert, ist der initiale Zustand einer Modulinstanz $s = PLA$. Der Kommunikationsparameter k beinhaltet z. B. die Adresse, die der Instanz zugewiesen ist. Ein mögliches Verfahren zur Bestimmung der Adresse ist die in [36] beschriebene positionsbasierte Adresszuweisung, bei der jeder möglichen Position eine feste Adresse zugeordnet ist.

4. Die zuvor generierte Modulinstanz c wird der Menge der momentan platzierten Modulinstanzen hinzugefügt $C \leftarrow C \cup c$. Die entsprechenden Ressourcen gelten nun als belegt, so dass sich folgende Änderung der Zellbelegung ergibt:

$$b(i,j) \leftarrow 0 \quad \forall i,j : x_h \leq i < x_h + a_h(m) \ \wedge \ x_v \leq j < x_v + a_v(m) \tag{3.5}$$

Die nun generierte Instanz c ist solange Element der Menge C, bis die verschiedenen Zustände durchlaufen sind und die Instanz durch das Zurücksetzen der Zellen von der Architektur entfernt ist. Nach dem Entfernen ist die resultierende Menge der momentan platzierten Modulinstanzen daher $C \leftarrow C \setminus c$.

3.1.3 Vergleichbare Modelle

Vergleichbare Arbeiten zu dem zuvor beschriebenen DMC-Modell sind das HySAM-Modell von Bondalapati und Prasanna [11]. Dieses Modell kann verwendet werden, um Anwendungen daraufhin zu analysieren, ob sie zu Ausführung auf rekonfigurierbarer Logik geeignet sind. Das Modell basiert auf einer Systemarchitektur, bestehend aus einem Mikroprozessor, der über einen Bus mit einer rekonfigurierbarer Logikeinheit (engl.: Configurable Logic Unit (CLU)) verbunden ist. Die auszuführende Anwendung wird dabei in eine Sequenz von Funktionen $F = \{F_1, \ldots, F_n\}$ unterteilt, wobei jede einzelne Funktion entweder auf dem Mikroprozessor oder auf der CLU ausgeführt werden kann. Auf diese Weise kann zur Laufzeit entschieden werden, ob eine Funktion in Software oder in rekonfigurierbarer Hardware ausgeführt wird. Jede Funktion ist mindestens einer CLU-Konfiguration $C = \{C_1, \ldots, C_m\}$ zugeordnet, wobei eine CLU-Konfiguration aus mehreren Funktionen bestehen kann. Eine CLU-Konfiguration stellt damit den Kontext der rekonfigurierbaren Fläche dar. Das Modell orientiert sich demnach an dem Prinzip der Multi-Kontext-Architekturen, wie

in [63, 77] beschrieben. Das HySAM Modell findet Einsatz in der Simulationsumgebung DRIVE [12], welche eine interaktive Analyse der zugrunde liegenden Systemarchitektur ermöglicht. Mithilfe von DRIVE lassen sich Ausführungszeiten bestimmen und die Auslastung der Ressourcen nachbilden. Bezüglich des DMC-Modells kann eine Funktion mit einem Hardware-Modul gleichgesetzt werden. Ebenso kann eine Konfiguration mit der Menge der momentan platzierten Modulinstanzen C verglichen werden.

Lange und Middendorf beschreiben in [48, 49] das Modell der *hyperrekonfigurierbaren Architektur*. Hyperrekonfigurierbarkeit meint dabei eine dynamische Anpassung der Rekonfigurationsmöglichkeiten der Architektur, so dass die Rekonfigurationsschritte und die damit verbundene Rekonfigurationszeit für eine gegebene Anwendung optimiert werden können. Die Rekonfiguration wird dabei auf der Ebene der *Hyperkontexte* und der Ebene der *Kontexte* betrachtet. Hyperkontexte definieren die Rekonfigurationsmöglichkeiten der Architektur. Ähnlich wie im HySAM Modell spezifizieren Kontexte das Verhalten von aktiven Ressourcen. Das Modell betrachtet im Wesentlichen Rekonfigurationseigenschaften einer Architektur und geht dabei nicht im Speziellen auf partielle Rekonfigurierbarkeit ein. Bezüglich des DMC-Modells kann ein Kontext mit der Menge der momentan platzierten Modulinstanzen C verglichen werden.

Bazargan et al. verwenden in [8] ein Modell, welches auf so genannte *RFUOPs* (engl.: Reconfigurable Functional Unit Operations) aufbaut. Das Modell betrachtet den Flächenbedarf, den Start- und Endzeitpunkt einer *RFUOP*. Die Menge *ACC* beschreibt alle erfolgreich platzierten *RFUOPs* und deren zugewiesenen Positionen. Da der Start- und Endzeitpunkt der *RFUOPs* als gegeben betrachtet wird, dient das Modell zur Beschreibung von Offline-Platzierungsverfahren, d. h., die Start- und Endzeitpunkte und die Positionen der einzelnen *RFUOPs* werden zur Entwurfszeit bestimmt. Das Modell lässt dabei den Aspekt der Rekonfigurationszeit von *RFUOPs* außer Acht.

Das in Steiger et al. [72] beschriebene Modell betrachtet Tasks als rechteckförmige Flächen rekonfigurierbarer Ressourcen. Das Modell wird verwendet, um Platzierungs- und Ablaufplanungsverfahren für echtzeitfähige rekonfigurierbare Systeme zu beschreiben. Ein Task T_i ist gegeben durch die Breite w_i und Höhe h_i und durch den Zeitpunkt der Anfrage a_i. Darüber hinaus beschreibt e_i die Ausführungszeit und d_i die Deadline, wobei gilt $d_i \geq a_i + e_i$. Durch die Betrachtung gegebener Ausführungszeiten und Deadlines ist das Modell für Echtzeitsysteme ausgelegt. Da die Ausführungszeiten der Tasks gegeben sind, gestaltet sich die Platzierung der Tasks zur Laufzeit als dreidimensionales Packungsproblem. Neben der Breite w_i und der Höhe h_i beschreibt die Ausführungszeit e_i die dritte Dimension eines Tasks. Ein geplanter Task (engl.: Scheduled Task) beschreibt einen Task, dessen Anfangszeitpunkt und Position vom Platzierungsalgorithmus zugewiesen wurde. Insofern lässt sich ein geplanter Task mit einer Modulinstanz im DMC-Modell vergleichen.

Das DMC-Modell hebt sich im Wesentlichen durch Betrachtung der Ebenen Systemkomponente, Hardware-Modul und Modulinstanz von den oben genannten Modellierungen ab. Insbesondere die Betrachtung der Zustände der derzeit platzierten Modulinstanzen ermöglicht unter anderem die Berücksichtigung von Konfigurations- und Löschvorgängen, welche in den meisten Modellierungen nicht betrachtet werden. Darüber hinaus erlaubt das DMC-Modell neben der Modellierung homogener rekonfigurierbarer Architekturen ebenso die Modellierung heterogener rekonfigurierbarer Architekturen.

Die in dieser Arbeit betrachteten Platzierungsverfahren beziehen sich auf den Fall, dass Anfragezeitpunkte und Ausführungszeiten der Systemkomponenten zur Entwurfszeit unbekannt sind. Die Ausführungszeit eines entsprechenden Hardware-Moduls ist z. B. dann unbekannt, wenn das Ende der Ausführung an Bedingungen geknüpft ist oder von äußeren Parametern des Systems abhängt. Bei der Platzierung werden daher lediglich die Modulflächen der entsprechenden Hardware-Module betrachtet. Trotz unbekannter Ausführungszeiten ist bei der Platzierung eines Hardware-Moduls eine Ablaufplanung erforderlich, wie sie im folgenden Abschnitt näher beschrieben wird.

3.2 Ablaufplanung

Die Ablaufplanung ist insbesondere dann mit der Modulplatzierung verknüpft, wenn die Ausführungszeit der Modulinstanz bekannt ist. In diesem Fall lässt sich die Ablaufplanung und Platzierung als gemeinsames dreidimensionales Packungsproblem formulieren und zur Entwurfszeit (siehe [23, 30]) oder zur Laufzeit (siehe [71]) mit entsprechenden Verfahren lösen. Sobald die Ausführungszeiten der Hardware-Module unbekannt sind, können Ablaufplanung und Platzierung nicht mehr als gemeinsames Optimierungsproblem betrachtet werden. Trotz der unbekannten Ausführungszeiten ist dennoch eine Ablaufplanung erforderlich, denn sowohl die Platzierung eines Moduls als auch der Konfigurationsprozess einer Modulinstanz kann nur sequenziell durchgeführt werden. Wenn beispielsweise zur Laufzeit mehrere Komponentenanfragen zur gleichen Zeit gestellt werden, so muss anhand einer Ablaufplanung bestimmt werden, welche der Komponentenanfragen als Erstes der Platzierung übergeben wird. Die Ablaufplanung unterteilt sich demnach in Platzierungsablaufplanung und Konfigurationsablaufplanung. Bei der Platzierungsablaufplanung wird entschieden, welches Modul der derzeit angeforderten Systemkomponenten als Nächstes platziert wird. Nach der Platzierung befindet sich die generierte Modulinstanz im Zustand *PLA* (vgl. Abbildung 3.4). Bei der Konfigurationsablaufplanung wird entschieden, welche der im Zustand *PLA* oder im Zustand TRM befindlichen Modulinstanzen als Nächstes konfiguriert bzw. gelöscht wird. Es besteht daher ein zeitlicher Zusammenhang zwischen der Platzierungsablaufplanung und der Konfigurationsablaufplanung.

3.2.1 Platzierungsablauf

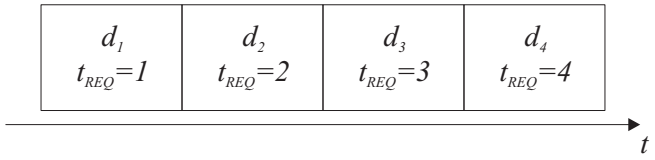
Die Ablaufplanung der Platzierung beinhaltet zum einen die Festlegung der Platzierungsreihenfolge der derzeit angeforderten Systemkomponenten und zum anderen die Handhabung fehlgeschlagener Modulplatzierungen. Modulplatzierungen können grundsätzlich nicht parallel durchgeführt werden, da der Platzierungsalgorithmus zur Bestimmung der Position eines Moduls einen eindeutigen Suchraum benötigt. Eine Platzierungsablaufplanung wird genau dann benötigt, wenn gleichzeitig oder mit geringer Verzögerung mehrere Systemkomponenten angefragt werden, so dass mindestens eine Platzierung nicht unmittelbar durchgeführt werden kann. Die Platzierungsablaufplanung trifft die Entscheidung, welche Systemkomponente als Nächstes platziert wird.

Es besteht daher eine Parallele zu nicht-präemptiven Prozess-Scheduling-Verfahren im Bereich der Betriebssysteme. In [73] werden verschiedene Scheduling-Verfahren diskutiert, wobei sich z. B. die folgenden Verfahren auf die Platzierungsablaufplanung übertragen lassen:

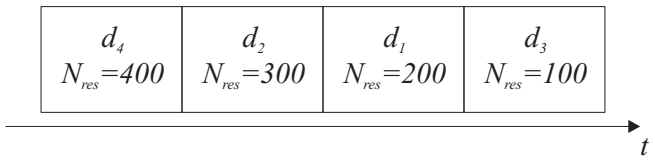
First-Come First-Served (FCFS): Eines der einfachsten Verfahren zur Festlegung der Platzierungsreihenfolge ist das *First-Come-First-Served*-Verfahren, welches die Platzierungen in der Reihenfolge der Komponentenanfragen durchführt. Auf diese Weise wird sichergestellt, dass die Modulplatzierung einer neuen Komponentenanfrage nicht beliebig verzögert wird. Das Verfahren ist besonders gut geeignet, wenn Komponentenanfragen gleichberechtigt sind und keine Priorisierung einzelner Komponentenanfragen vorhanden ist.

Größenabhängige Platzierungsreihenfolge: Das Verfahren mit größenabhängiger Platzierungsreihenfolge wählt die größte unplatzierte Systemkomponente. Die Platzierung einer größeren Systemkomponente erfordert das Vorhandensein entsprechend großer zusammenhängender freier Flächen, was zur Folge hat, dass die Anzahl der freien Positionen gering ist (vgl. (3.2)). Um zu vermeiden, dass die Platzierung einer großen unplatzierten Systemkomponente durch eine zuvor platzierte kleine Systemkomponente verhindert wird, kann die Reihenfolge der Platzierung entsprechend der Größe sortiert werden, so dass die Platzierung großer Systemkomponenten kleineren Systemkomponenten vorgezogen wird.

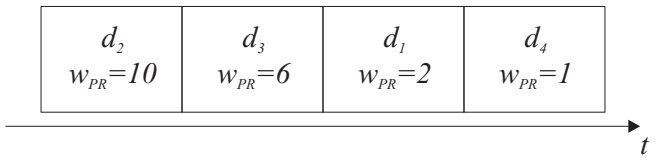
Prioritätsbasierte Platzierungsreihenfolge: Die derzeit angeforderten unplatzierten Systemkomponenten können für die Anwendung von unterschiedlicher Wichtigkeit sein. Bei dem Verfahren mit prioritätsbasierter Platzierungsreihenfolge weist die Anwendung jeder Komponentenanfrage einen Prioritätswert $w_{PR}(d_i) \in [0, N_{PR}]$ zu, wobei $N_{PR} > 0$ die höchste Priorität darstellt. Komponentenanfragen mit dem höchsten Prioritätswert werden als Nächstes platziert. Auf diese Weise wird die benötigte Zeit für die Platzierung wichtiger Systemkomponenten gering gehalten.



(a) First-Come First-Served.



(b) Größenabhängige Reihenfolge.



(c) Prioritätsbasierte Reihenfolge.

Abbildung 3.7: Beispiel für verschiedene Verfahren zur Platzierungsablaufplanung.

In dem folgenden Beispiel werden die zuvor beschriebenen Verfahren zur Platzierungsablaufplanung gegenübergestellt.

Beispiel 3.3. *Sei $D_{REQ} = \{d_1, d_2, d_3, d_4\}$ die Menge der unplatzierten Komponentenanfragen, wobei $t_{REQ}(d_i)$ den Zeitpunkt der Platzierungsanfrage der Komponentenanfragen d_i markiert. Die Reihenfolge der Anfragen sei gegeben entsprechend der Indexwerte, so dass $t_{REQ}(d_1) \leq t_{REQ}(d_2) \leq t_{REQ}(d_3) \leq t_{REQ}(d_4)$. Sei $N_{res}(d_i)$ die Anzahl der benötigten Zellen der Systemkomponente $d_i \in D$, wobei $N_{res}(d_1) = 200$, $N_{res}(d_2) = 300$, $N_{res}(d_3) = 100$ und $N_{res}(d_4) = 400$. Des Weiteren sei $w_{PR}(d_i) \in [1,10]$ die der Komponentenanfrage d_i zugeteilte Priorität, wobei $w_{PR}(d_1) = 2$, $w_{PR}(d_2) = 10$, $w_{PR}(d_3) = 6$ und $w_{PR}(d_4) = 1$. Die entsprechenden Platzierungsreihenfolgen der zuvor beschriebenen Platzierungsablaufverfahren sind in Abbildung 3.7 dargestellt.*

Wenn die Platzierungsbedingung (3.4) nicht erfüllt ist, kann die Modulplatzierung einer Komponentenanfrage nicht durchgeführt werden. In solch einer Situation gibt es mehrere Methoden zur Handhabung erfolgloser Modulplatzierungen. Die Anwendung kann z. B. derart ausgelegt sein, dass erfolglose Modulplatzierungen akzeptiert werden und anstelle eines Hardware-Moduls eine entsprechende Software-Implementierung ausgeführt wird. In diesem Fall werden Komponentenanfragen mit erfolgloser Modulplatzierung einfach abgewiesen.

Wenn sich Modulplatzierungen nicht abweisen lassen, so kann die Platzierung so lange verzögert werden, bis durch das Zurücksetzen abgelaufener Modulinstan-

zen hinreichend viele Ressourcen frei werden und die Platzierungsbedingung (3.4) wieder erfüllt ist. Im Zusammenhang mit dem FCFS-Verfahren führt eine Platzierungsverzögerung ebenso zu einer Platzierungsverzögerung aller anderen vorhandenen unplatzierten Komponentenanfragen in D_{REQ}. Bei der Verwendung des FCFS-Verfahrens ergibt sich jedoch nicht das folgende Problem: Bei Verwendung des Verfahrens der größenabhängigen Platzierungsreihenfolge kann die Platzierungsverzögerung der größten Systemkomponente dazu führen, dass Anfragen kleiner Systemkomponenten durch fortlaufend neue Anfragen größerer Systemkomponenten sehr lange verzögert werden. Das gleiche Problem taucht bei der prioritätsbasierten Ablaufplanung auf, bei der Komponentenanfragen mit niedriger Priorität durch fortlaufend neue Komponentenanfragen mit höherer Priorität verzögert werden.

3.2.2 Konfigurationsablauf

Eine Modulinstanz beansprucht die Konfigurationsschnittstelle der rekonfigurierbaren Architektur bei ihrem Konfigurations- und Löschvorgang. Beim Konfigurationsvorgang werden die der Modulinstanz zugewiesenen Zellen der Architektur dem Konfigurationsdatenstrom entsprechend rekonfiguriert. Beim Löschvorgang werden die der Modulinstanz zugewiesenen Zellen in einen initialen Zustand zurückgesetzt, um ein unvorhersehbares Fehlverhalten der Zellen und deren Verbindungsstruktur bei nachfolgenden Konfigurationsprozessen zu vermeiden.

Konfigurations- und Löschvorgänge sind nach heutigem Stand der Technik nur sequenziell möglich (vgl. [87]), so dass eine Konfigurationsablaufplanung benötigt wird. Die Konfigurationsablaufplanung entscheidet, welche der im Zustand $s = PLA$ oder $s = TRM$ befindlichen Modulinstanzen als Nächstes konfiguriert oder gelöscht wird.

Ähnlich der Platzierungsablaufplanung lassen sich bereits bekannte Verfahren aus dem Bereich des nicht-präemptiven Prozess-Schedulings adaptieren. Der entscheidende Unterschied zur Platzierungsablaufplanung besteht darin, dass zwei unterschiedliche Vorgänge (Konfigurations- und Löschvorgang) in einem Konfigurationsablauf verarbeitet werden. Darüber hinaus hängt die benötigte Zeit der Konfigurations- und Löschvorgänge $t_{CFG}(c)$ von der Größe der Modulinstanzen ab (vgl. (3.21), (3.22)).

Grundsätzlich besteht die Möglichkeit, Konfigurations- und Löschvorgänge in der Ablaufplanung als gleichwertig zu betrachten und eine gemeinsame Reihenfolge für beide Vorgänge zu bestimmen. Eine andere Vorgehensweise wäre eine separate Reihenfolge für Konfigurations- und Löschvorgänge zu bestimmen und entweder die Löschvorgänge den Konfigurationsvorgängen vorzuziehen oder umgekehrt. Werden die Löschvorgänge den Konfigurationsvorgängen vorgezogen (engl.: Delete Before Configure (DBC)), dann ist die Reihenfolge der Löschvorgänge für die anstehenden Konfigurationsvorgänge irrelevant, denn die Konfigurationsvorgänge beginnen erst dann, wenn alle geforderten Löschvorgänge beendet sind. Die Zeit zwischen dem

Platzierungsende (engl.: End Of Placement (EOP)) und dem Konfigurationsbeginn (engl.: Begin Of Configuration (BOC)) einer Modulinstanz $c \in C$ entspricht der Zeitspanne $t_{BOC-EOP}$ in der sich die Modulinstanz im Zustand $s(c) = PLA$ befindet. Die untere Schranke der Verzögerung $t_{BOC-EOP}(c_{PLA})$ des Konfigurationsvorgangs einer Modulinstanz $c_{PLA} \in C : s(c_{PLA}) = PLA$ ist somit

$$t_{BOC-EOP}(c_{PLA}) \geq \sum_{c_{TRM}} t_{CFG}(c_{TRM}) \,, \quad c_{TRM} \in C : s(c_{TRM}) = TRM \,. \tag{3.6}$$

Jedoch beeinflusst die Reihenfolge der Löschvorgänge die Zellbelegung $b(i, j)$ und dementsprechend die Menge der platzierbaren Module $M_{pla}(d)$ einer geforderten Systemkomponente d. Somit wirkt sich die Reihenfolge der Löschvorgänge auf die Modulplatzierung aus. Da beim DBC-Verfahren eine abgelaufene Modulinstanz umgehend gelöscht wird, wird die Verzögerung bis zur erneuten Freigabe der von der Modulinstanz belegten Ressourcen gering gehalten. Die schnelle Freigabe zuvor belegter Zellen erhöht damit die Anzahl der platzierbaren Module. Daher ist das DBC-Verfahren besonders gut geeignet, wenn die Modulplatzierung nur einen Platzierungsversuch zulässt und bei Erfolglosigkeit die entsprechende Komponentenanfrage abweist.

Wenn die Konfigurationsvorgänge den Löschvorgängen vorgezogen werden (engl.: Configure Before Delete (CBD)), entspricht die obere Schranke für die Verzögerung des Konfigurationsvorgangs einer Modulinstanz $c_{PLA} \in C : s(c_{PLA}) = PLA$ der Zeit, die für die Konfiguration aller ungenutzten Zellen der Architektur benötigt wird. Ebenso wie beim DBC-Verfahren wirkt sich die Reihenfolge der Löschvorgänge grundsätzlich nur auf die Modulplatzierung aus. Lediglich Löschvorgänge, die zum Zeitpunkt der Konfigurationsanfrage einer Modulinstanz bereits gestartet wurden, können zu einer Verzögerung des entsprechenden Konfigurationsvorgangs führen. Da beim CBD-Verfahren eine platzierte Modulinstanz umgehend konfiguriert wird, wird die Verzögerung bis zum Ausführungsbeginn der Modulinstanz gering gehalten. Somit wird die Anzahl der Modulinstanzen, die sich im Zustand $s = PLA$ befinden, ebenfalls gering gehalten. Das CBD-Verfahren ist daher besonders gut für eine Platzierungsablaufplanung geeignet, die fehlgeschlagene Modulplatzierungen verzögert.

Unabhängig davon, ob eine gemeinsame Reihenfolge oder zwei getrennte Reihenfolgen unter Verwendung des CBD- oder DBC-Verfahren vorliegen, können die entsprechenden Vorgänge innerhalb einer Reihenfolge anhand der folgenden Verfahren angeordnet werden:

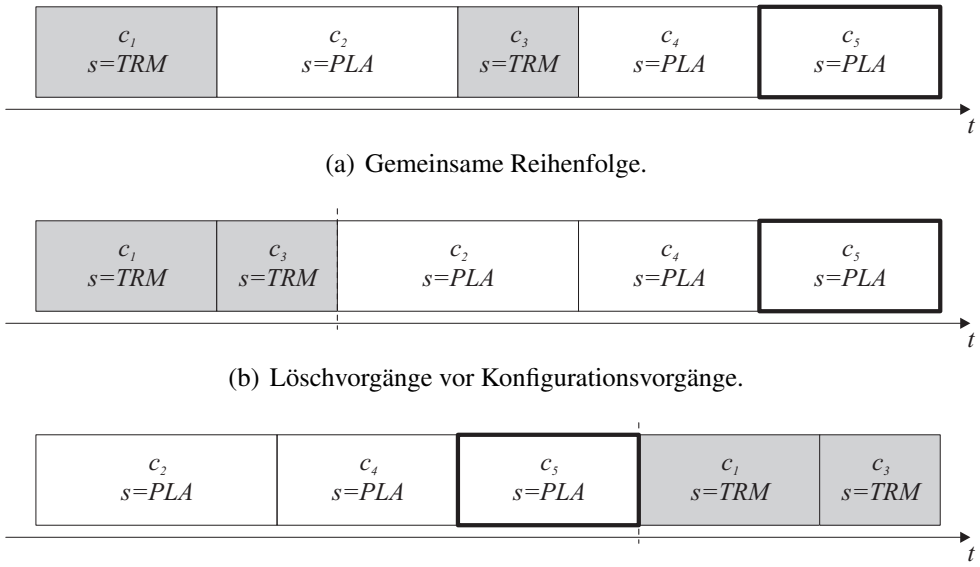
First-Come First-Served (FCFS): Beim First-Come-First-Serve-Verfahren werden die Konfigurations- oder Löschvorgänge in der Reihenfolge bearbeitet, in der sie angefragt werden. Wird eine gemeinsame Reihenfolge für Konfigurations- oder Löschvorgänge verwendet, so kann zum Zeitpunkt der

Konfigurationsanfrage die Verzögerung des Konfigurationsvorgangs einer Modulinstanz bestimmt werden, da zukünftige Vorgänge immer zu einem späteren Zeitpunkt durchgeführt werden. Auf diese Weise ist das FCFS-Verfahren mit gemeinsamer Reihenfolge für Echtzeitsysteme geeignet, denn zum Zeitpunkt der Konfigurationsanfrage einer Modulinstanz lässt sich die Verzögerung bis zum Beginn der Ausführung bestimmen. Die Verwendung des FCFS-Verfahren mit gemeinsamer Reihenfolge wirkt sich ebenso vorteilhaft auf die Modulplatzierung aus, denn die Zellen bereits ausgeführter oder im Vorgang des Zurücksetzens befindlicher Modulinstanzen ($s(c) \in \{TRM, DEL\}$) können als verfügbar betrachtet werden, da der der Modulplatzierung folgende Konfigurationsvorgang immer zu einem späteren Zeitpunkt durchgeführt wird, zu dem die besagten Zellen wieder verfügbar sind. Bei der Berechnung der Zellbelegung brauchen daher nur Modulinstanzen in den Zuständen $s(c) \in \{PLA, CFG, EXE\}$ betrachtet werden, so dass

$$b_{FCFS}(i,j) = \begin{cases} 0 & \text{wenn } \exists c \in C \mid x_h(c) \leq i < x_h(c) + a_h(m(c)) \wedge \\ & \qquad x_v(c) \leq j < x_v(c) + a_v(m(c)) \wedge \\ & \qquad s(c) \in \{PLA, CFG, EXE\} \quad , \\ 1 & \text{sonst .} \end{cases} \tag{3.7}$$

Wenn getrennte Reihenfolgen und das DBC-Verfahren verwendet werden, kann bei der Berechnung der Zellbelegung ebenfalls b_{FCFS} verwendet werden, da zum Zeitpunkt des Konfigurationsvorgangs der aus der Modulplatzierung hervorgegangenen Modulinstanz alle vorangegangenen Löschvorgänge abgeschlossen sind. Die Verzögerung des Konfigurationsvorgangs kann jedoch nicht mehr exakt bestimmt werden, da unter Umständen zukünftige Löschvorgänge dem Konfigurationsvorgang vorgezogen werden. Das CBD-Verfahren erlaubt nicht die Verwendung der Zellbelegung b_{FCFS}, da die Löschvorgänge abgelaufener Module erst nach der Konfiguration der aus der Modulplatzierung hervorgehenden Modulinstanz durchgeführt werden.

Beispiel 3.4. *Gegeben seien die Modulinstanzen c_1, c_2, c_3, c_4, wobei c_1 und c_3 gelöscht werden sollen (Zustand $s(c_1) = s(c_3) = TRM$) und c_2 und c_4 konfiguriert werden sollen (Zustand $s(c_2) = s(c_4) = PLA$). Die zeitliche Reihenfolge der Konfigurations- bzw. Löschanfragen entspricht der Indexwerte. Die Modulplatzierung erzeugt eine neue Modulinstanz c_5, dessen Konfigurationsvorgang in die Konfigurationsablaufplanung entsprechend dem First-Come-First-Served-Verfahren eingefügt werden soll. Abbildung 3.8 zeigt die resultierende Ablaufplanung für eine gemeinsame Reihenfolge und getrennte Reihenfolgen gemäß dem DBC- und CBD-Verfahren.*



(a) Gemeinsame Reihenfolge.

(b) Löschvorgänge vor Konfigurationsvorgänge.

(c) Konfigurationsvorgänge vor Löschvorgänge.

Abbildung 3.8: Beispiel für gemeinsame und getrennte Reihenfolgen für die Konfigurationsablaufplanung anhand des FCFS-Verfahrens.

Größenabhängige Reihenfolge: Ähnlich dem gleichnamigen Platzierungsablaufverfahren wird hier die Reihenfolge der Konfigurations- und Löschvorgänge anhand der Größe oder der Konfigurationszeit der entsprechenden Modulinstanz bestimmt. Ob der kürzeste oder längste Konfigurationsvorgang ausgewählt wird, hängt von der zugrunde liegenden Anwendung ab. Wenn z. B. die zu erwartenden Ausführungszeiten der Modulinstanzen mit der Größe ansteigen, dann führt eine Reihenfolge beginnend mit dem kürzesten Konfigurationsvorgang zu einem ausgeglichenen Verhältnis zwischen der Verzögerung vom Zeitpunkt der Platzierung bis zum Ausführungsbeginn und der entsprechenden Ausführungszeit.

Prioritätsbasierte Reihenfolge: Wie schon im Zusammenhang mit dem Platzierungsablauf erwähnt, kann die Konfigurationsablaufplanung ebenfalls anhand der Prioritätswerte der Komponentenanfragen durchgeführt werden. Auf diese Weise kann den Komponentenanfragen mit hoher Wichtigkeit eine kurze Verzögerung vom Zeitpunkt der Platzierung bis zum Ausführungsbeginn ermöglicht werden. Die prioritätsbasierte Reihenfolge ist nur für Konfigurationsvorgänge geeignet, da die Prioritätswerte für die Löschvorgänge nicht definiert sind. Demnach kommt das Verfahren mit prioritätsbasierter Reihenfolge nur für getrennte Reihenfolgen in Frage.

Die oben beschriebenen Verfahren lassen sich auch kombinieren, so dass unterschiedliche Verfahren für die Festlegung der Reihenfolge der Konfigurationsvorgänge und

Verfahren zur Ablaufplanung	FCFS			größenabhängig			prioritätsbasiert		
	gem.	DBC	CBD	gem.	DBC	CBD	gem.	DBC	CBD
Platzierung anhand b_{FCFS}	√	√	×	×	√	×	×	√	×
Berechenbarkeit der Verzögerung	a)	b)	b)	c)	b)	b)	×	b)	a)
Platzierung mit Abweisung	+	+	-	○	+	-	×	+	-
Platzierung mit Verzögerung	○	-	+	○	-	+	×	-	+

gem.: gemeinsame Reihenfolge √: möglich ×: unmöglich

DBC: Lösch- vor Konfigurationsvorgänge a): berechenbar b): obere Schranke berechenbar c): nicht berechenbar

CBD: Konfigurations- vor Löschvorgängen +: gut geeignet ○: durchschnittlich geeignet -: schlecht geeignet

Tabelle 3.1: Qualitative Bewertung der verschiedenen Verfahren zur Konfigurationsablaufplanung.

der Reihenfolge der Löschvorgänge verwendet werden. Eine qualitative Bewertung der einzelnen Verfahren zur Konfigurationsablaufplanung ist in Tabelle 3.1 dargestellt. Das First-Come-First-Served-Verfahren mit gemeinsamer Reihenfolge ist das einzige Verfahren, bei dem die Verzögerung der Konfigurationsvorgänge berechenbar ist und die Zellbelegung b_{FCFS} verwendet werden kann. Die Verwendung von b_{FCFS} ist bei der Platzierungsablaufplanung mit Modulabweisung vorteilhaft, da die Ressourcen terminierter und im Löschvorgang befindlicher Modulinstanzen für die Modulplatzierung als verfügbar betrachtet werden können. Für die Platzierungsablaufplanung, die fehlgeschlagene Modulplatzierungen verzögert, ist es vorteilhaft, Konfigurationsvorgänge den Löschvorgängen vorzuziehen, um die Anzahl der verzögerten Konfigurationsvorgänge gering zu halten.

3.3 Analyseverfahren

Die in dynamisch rekonfigurierbaren Architekturen zur Laufzeit verwendeten Methoden, wie die Modulplatzierung oder das Defragmentieren, gehören der Klasse der Online-Algorithmen an. Die Platzierung wird z. B. zur Laufzeit anhand der gegebenen Modulinstanzen durchgeführt, wobei dem Platzierungsverfahren zukünftige Komponentenanfragen nicht bekannt sind. Um die Qualität eines Online-Algorithmus analytisch zu bewerten, wurde von Sleater und Tarjan [70] die *kompetitive Analyse* vorgestellt. Bei der kompetitiven Analyse wird die Qualität eines Online-Verfahrens mit einem Offline-Verfahren, welches vollständiges Wissen über die zukünftige Eingabesequenz besitzt, verglichen. Maßstab zur Bewertung des Online-Verfahrens ist dabei das Maximum des Verhältnisses der Zielgröße des Online-Verfahrens zu der Zielgröße des optimalen Offline-Verfahrens. Die kompetitive Analyse dient daher als Worst-Case-Analyse und ist grundsätzlich pessimistisch. In Bezug auf die Platzie-

rung von Hardware-Modulen lässt sich eine kompetitive Analyse nur dann durchführen, wenn ein Offline-Platzierungsverfahren existiert, welches die optimale Platzierung einer gegebenen Liste von Hardware-Modulen bestimmt. Zielgrößen der Optimierung können dabei die verwendete Fläche der Architektur oder die Gesamtausführungszeit sein.

Einen ersten Ansatz eines solchen Verfahrens wurde von Fekete et al. in [30] beschrieben. Bei dem Verfahren wird eine gegebene Anwendung in Form eines Datenflussgraphen dargestellt. Jeder Knoten des gegebenen Graphen wird durch ein entsprechendes Hardware-Modul repräsentiert, das auf eine homogene rekonfigurierbare Architektur platziert werden kann. Bezüglich des DMC-Modells wird daher zu jeder Systemkomponente $d \in D$ nur ein Hardware-Modul berücksichtigt ($\forall d \in D : |M(d)| = 1$). Da die Ausführungszeiten der Hardware-Module gegeben sind, kann die Offline-Platzierung als dreidimensionales Packungsproblem interpretiert werden. Hardware-Module werden dabei als dreidimensionale Objekte dargestellt. Neben dem Flächenbedarf beschreibt die Ausführungszeit die dritte Dimension eines Hardware-Moduls. Das beschriebene Offline-Platzierungsverfahren bestimmt die entsprechenden Start-/ End-Zeitpunkte und Positionen der einzelnen Hardware-Module unter Berücksichtigung der Datenabhängigkeiten.

Danne et al. [23] verwenden einen erweiterten Algorithmus, der verschiedene Modulvarianten mit unterschiedlichen Seitenverhältnissen zulässt. D. h., zu jeder Systemkomponente $d \in D$ können mehrere Hardware-Module existieren ($\forall d \in D : |M(d)| \geq 1$). Für eine gegebene Systemkomponente $d \in D$ wird die Bestimmung eines Hardware-Moduls $m \in M(d)$ anhand heuristischer Verfahren durchgeführt. Der Platzierungsalgorithmus erzeugt eine Pareto-Menge von Lösungen mit den Zielen, die Ausführungszeit und die benötigte Fläche zu optimieren.

Die hier beschriebenen Verfahren vernachlässigen jedoch die Konfigurationszeit der einzelnen Modulinstanzen und die Tatsache, dass Konfigurationsvorgänge in heutigen Architekturen nur sequenziell durchgeführt werden können und daher einer Ablaufplanung, wie in Abschnitt 3.2 beschrieben, unterliegen. Ebenso werden Heterogenitäten der rekonfigurierbaren Architektur nicht betrachtet. Die Bestimmung der optimalen Lösung des Offline-Platzierungsproblems ist mit heutigen Verfahren somit nur in Teilen gelöst, so dass die kompetitive Analyse von Platzierungsverfahren nicht ohne weiteres möglich ist.

Aus diesen Gründen werden in vielen Arbeiten die Methoden für dynamisch rekonfigurierbare Architekturen anhand von Simulationen bewertet (z. B. [8, 12, 29, 72]). Die simulative Analyse ermöglicht die Berücksichtigung real existierender rekonfigurierbarer Architekturen und deren Anwendungen. Simulationen spiegeln das Verhalten realer Implementierungen wieder, ohne dabei die zu testenden Methoden auf der Zielarchitektur implementieren zu müssen. Auf diese Weise lassen sich neue Methoden schnell und unter realistischen Bedingungen testen. Im folgenden Abschnitt wird die Simulationsumgebung zur Analyse rekonfigurierbarer Architekturen

(SARA) vorgestellt, welche zur Bewertung der in den folgenden Kapiteln beschriebenen Platzierungs- und Defragmentierungsverfahren genutzt wurde.

3.3.1 Simulationsumgebung SARA

Einer der ersten Ansätze einer architekturunabhängigen Simulationsumgebung für dynamische rekonfigurierbare Architekturen wurde von Lysaght et al. [53] beschrieben. Das Simulationsverfahren abstrahiert den Vorgang der partiellen Rekonfiguration durch dynamisches Ein- und Ausschalten einzelner rekonfigurierbarer Schaltungselemente (engl.: Dynamic Circuit Switching (DCS)). Bondalapati und Prasanna haben die Simulationsumgebung DRIVE [12] entwickelt, welche das HySAM-Modell [11] verwendet. Die Simulationsumgebung DRIVE kann zur interaktiven Analyse der Architektur und des Entwurfsraums genutzt werden. Dabei können Leistungsmerkmale, wie z. B. die Gesamtausführungszeit einer Anwendung oder die Ressourcenauslastung der Architektur, untersucht werden.

Die in [E2, E3] vorgestellte Simulationsumgebung zur Analyse rekonfigurierbarer Architekturen (SARA) basiert auf dem im Abschnitt 3.1 spezifizierten DMC-Modell. Eine Übersicht der Simulationsumgebung ist in Abbildung 3.9 dargestellt. Die Simulationsumgebung unterteilt sich in die Blöcke *Vorgaben*, *Simulation* und *Analyse*. Die für eine Simulation benötigten Eingabeparameter sind in einer Tabelle von Systemkomponenten (Designs) spezifiziert. Die Tabelle enthält die Namen der einzelnen Systemkomponenten und den erforderlichen Ressourcenbedarf für die entsprechenden Modulimplementierungen. Zusätzlich lassen sich Angaben zur *Auswahlwahrscheinlichkeit*, zur *Priorität* und zur *Ausführungszeit* machen. Die Auswahlwahrscheinlichkeit $p_{sel}(d)$ beschreibt die Wahrscheinlichkeit, dass bei einer zufälligen Anfrage eine bestimmte Systemkomponente ausgewählt wird. Geeignete Platzierungs- und Ablaufplanungsverfahren ermöglichen eine bevorzugte Platzierung bestimmter Systemkomponenten durch Verwendungen von entsprechenden Prioritätswerten. Neben der Priorität kann auch die benötigte Ausführungszeit einer Systemkomponente angegeben werden.

Der Block *Vorgaben* besteht aus den Funktionen *Ablaufgenerator* und *virtuelle Synthese*. Beide Funktionen dienen zur Vorbereitung der Simulation und werden daher zur Entwurfszeit ausgeführt. In dem Bereich der rekonfigurierbaren Hardware haben sich zum Testen von Platzierungsverfahren bisher noch keine einheitlichen Benchmarks etabliert. Ein Grund dafür ist, dass die bisherigen Umsetzungen von Systemen mit partiell rekonfigurierbarer Hardware meist auf Systemansätze mit fester Aufteilung und Anwendungen mit einer sehr geringen Anzahl von Hardware-Modulen basieren (z. B. [65, 78]). Um bei einer unbekannten Anwendung dennoch das Testen von Platzierungsverfahren zu ermöglichen, werden in den Arbeiten zu Platzierungsverfahren (z. B. [8, 72]) zufällig erzeugte Abläufe von Komponentenanfragen verwendet. In der Simulationsumgebung SARA besteht einerseits die Möglichkeit einen bestehenden Ablauf von Komponentenanfragen einer gege-

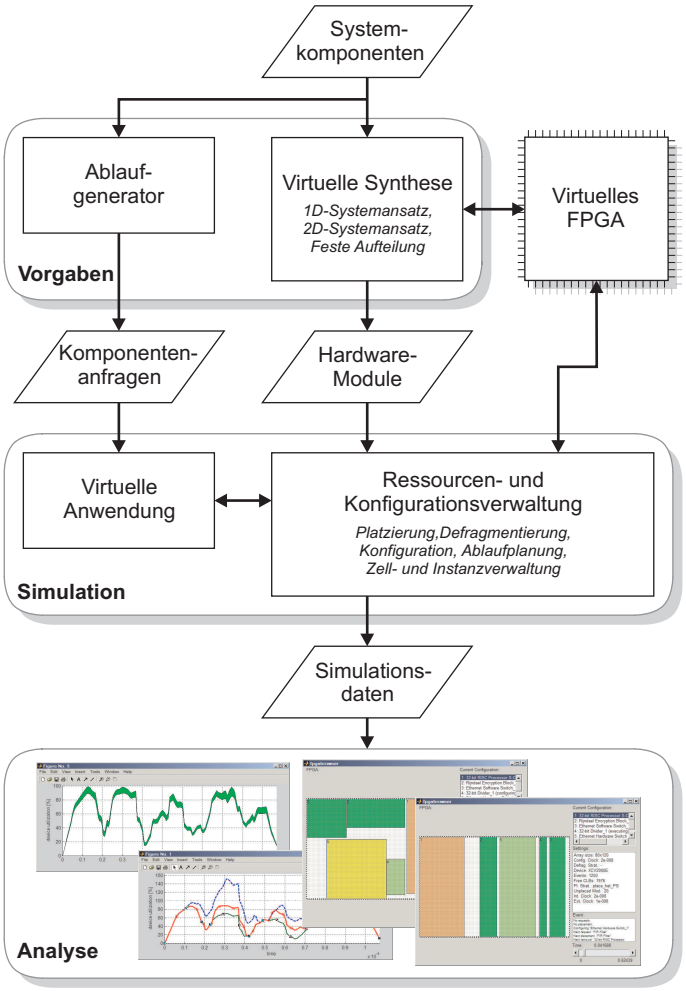


Abbildung 3.9: Übersicht der Simulationsumgebung SARA.

benen Anwendung zu importieren. Andererseits kann anhand des Ablaufgenerators ein zufälliger Ablauf von Komponentenanfragen mit entsprechenden Ausführungszeiten erzeugt werden, der auf der zuvor spezifizierten Tabelle von Systemkomponenten basiert. Der erzeugte Ablauf der Komponentenanfragen ist in der Liste $\sigma = (d_1, d_2, \ldots, d_{N_\sigma})$ angegeben, wobei N_σ die Anzahl der Komponentenanfragen ist. Jeder Komponentenanfrage d_i ist ein Anfragezeitpunkt $t_{REQ}(d_i)$ zugeordnet, wobei gilt

$$0 \leq t_{REQ}(d_i) \leq t_{REQ}(d_{i+1}) \quad \forall i \in [1, N_\sigma - 1]. \tag{3.8}$$

Zusätzlich ist jeder Komponentenanfrage d_i eine Ausführungszeit $t_{EXE}(d_i)$ zugeordnet. Die Erzeugung des Ablaufs lässt sich anhand der folgenden Optionen beeinflussen:

Simulationszeit: Die *Simulationslänge* N_{sim} beschreibt die Anzahl der Zeiteinheiten in der Simulation. Die *Simulationszeit* T_{sim} ist definiert durch $T_{sim} = N_{sim} \cdot t_{sim}$, wobei t_{sim} der *Simulationstakt* der *virtuellen Anwendung* ist. Alle Komponentenanfragen werden innerhalb der Simulationszeit gestellt, daher gilt $t_{REQ}(d_{N_\sigma}) \leq T_{sim}$.

Anfragewahrscheinlichkeit: Die *Anfragewahrscheinlichkeit* p_{req} beschreibt die Wahrscheinlichkeit einer Komponentenanfrage pro Zeiteinheit. Demnach entspricht die *Anzahl der Komponentenanfragen* $N_\sigma = \lceil N_{sim} \cdot p_{req} \rceil$.

Verteilung der Auswahlwahrscheinlichkeiten: Die *Auswahlwahrscheinlichkeiten* $p_{sel}(d)$ der einzelnen Systemkomponenten sind durch entsprechende Vorgabewerte in der Tabelle der Systemkomponenten angegeben. Alternativ besteht die Möglichkeit, die Auswahlwahrscheinlichkeiten an die Größe der Systemkomponenten zu koppeln, oder gleichverteilte Auswahlwahrscheinlichkeiten vorauszusetzen. Im Folgenden sei $N_{res}(d)$ die Anzahl der benötigten Zellen der Systemkomponente $d \in D$. Dann ergeben sich folgende Alternativen zur Berechnung der Auswahlwahrscheinlichkeit $p_{sel}(d)$:

- $p_{sel}(d) \propto N_{res}(d)$: Mit zunehmender Anzahl an Zellen wächst die Auswahlwahrscheinlichkeit, so dass

$$p_{sel}(d) = \frac{N_{res}(d)}{\sum_{\acute{d} \in D} N_{res}(\acute{d})} \,. \tag{3.9}$$

- $p_{sel}(d) \propto 1/N_{res}(d)$: Mit zunehmender Anzahl an Zellen sinkt die Auswahlwahrscheinlichkeit, so dass

$$p_{sel}(d) = \frac{1}{\sum_{\acute{d} \in D} \frac{N_{res}(d)}{N_{res}(\acute{d})}} \,. \tag{3.10}$$

- $p_{sel}(d) = 1/|D|$: Die Auswahlwahrscheinlichkeit ist konstant für alle Systemkomponenten.

Verteilung der Ausführungszeiten: Ähnlich der Auswahlwahrscheinlichkeit ergeben sich unterschiedliche Methoden zur Bestimmung der Ausführungszeit $t_{EXE}(d_i)$ einer Komponentenanfrage der Liste $\sigma = (d_1, d_2, \ldots, d_{N_\sigma})$. Sofern die Ausführungszeiten $t_{EXE}(d_i)$ bekannt sind, können sie durch entsprechende Vorgabewerte in der Tabelle der Systemkomponenten spezifiziert werden. Zusätzlich besteht die Möglichkeit konstante oder zufällige Ausführungszeiten anzunehmen oder die Ausführungszeit mit der Größe der Systemkomponente zu verändern. Demnach ergeben sich folgende Alternativen zur Berechnung der Ausführungszeit $t_{EXE}(d_i)$:

- $t_{EXE}(d_i) \propto N_{res}(d_i)$: Mit zunehmender Anzahl an Zellen wächst die Ausführungszeit. Sei T_{EXE} eine Zeitkonstante, dann berechnet sich die Ausführungszeit für eine Systemkomponente d_i:

$$t_{EXE}(d_i) = N_{res}(d_i) \cdot T_{EXE} \tag{3.11}$$

- $t_{EXE}(d_i) \propto 1/N_{res}(d_i)$: Mit zunehmender Anzahl an Zellen sinkt die Ausführungszeit. Sei T_{EXE} eine Zeitkonstante, dann berechnet sich die Ausführungszeit für eine Systemkomponente d_i:

$$t_{EXE}(d_i) = \frac{T_{EXE}}{N_{res}(d_i)} \tag{3.12}$$

- $t_{EXE}(d_i) = T_{EXE}$: Die Ausführungszeit ist für jede Systemkomponente konstant und entspricht der Zeit T_{EXE}.
- $t_{EXE}(d_i) = T_{EXE} \cdot n_{rnd}$: Die Ausführungszeit ist zufällig, wobei T_{EXE} die maximale Ausführungszeit beschreibt und $n_{rnd} \in [0,1]$ eine gleichverteilte Zufallszahl ist.

Die virtuelle Synthese simuliert den Syntheseprozess und erzeugt aus den Systemkomponenten entsprechende Hardware-Module. Die Erzeugung der Hardware-Module hängt dabei im Wesentlichen von der zugrunde liegenden rekonfigurierbaren Architektur und von dem gewählten Systemansatz ab. Die virtuelle Synthese unterstützt derzeit die Xilinx Virtex-FPGAs [89, 92, 93]. Die folgenden Systemansätze werden berücksichtigt:

2D-Systemansatz: Im 2D-Systemansatz werden die Module mit einem vorgegebenen Seitenverhältnis γ synthetisiert. Für eine Systemkomponente können daher mehrere Modulvarianten mit unterschiedlichen Seitenverhältnissen synthetisiert werden. Auf diese Weise wird dem Platzierungsverfahren eine größere Anzahl an Modulalternativen zur Verfügung gestellt. Zur Laufzeit kann das Verfahren in Abhängigkeit der freien Ressourcen entscheiden, welche Modulvariante platziert wird. Für eine homogene rekonfigurierbare Architektur mit nur einem Zelltyp ($N_{cell} = 1$) sei der Ressourcenvektor $r = (N_{res}(d))$. Dann gelten bei der Bestimmung der Breite $a_h(m)$ und der Höhe $a_v(m)$ eines Hardware-Moduls m die Bedingung $a_h(m) \approx a_v(m) \cdot \gamma$, wobei $a_h(m) \cdot a_v(m) \geq N_{res}(d)$. Die Bestimmung der Modulfläche $a = (a_h, a_v)$ erfolgt in Abhängigkeit des gewählten Seitenverhältnisses γ.

$$\text{wenn } \gamma < 1: \quad a_h = \left\lceil \sqrt{N_{res}(d) \cdot \gamma} \right\rceil \quad a_v = \left\lceil \frac{N_{res}(d)}{a_h} \right\rceil \tag{3.13}$$

$$\text{wenn } \gamma \geq 1: \quad a_h = \left\lceil \frac{N_{res}(d)}{a_v} \right\rceil \quad a_v = \left\lceil \sqrt{\frac{N_{res}(d)}{\gamma}} \right\rceil \tag{3.14}$$

Obige Berechnungsvorschrift führt in der Regel zu einer minimalen Anzahl ungenutzter Zellen $a_h(m) \cdot a_v(m) - N_{res}(d)$ innerhalb des Moduls.

1D-Systemansatz: Im 1D-Systemansatz haben die Module eine konstante Höhe und variieren lediglich in der Breite. Die Höhe der Module entspricht der Höhe N_{row} des verwendeten FPGAs. Für eine homogene rekonfigurierbare Architektur mit einem Zelltyp ($N_{cell} = 1$) sei der Ressourcenvektor $r = (N_{res}(d))$. Bei der Bestimmung der Breite $a(m)$ eines Hardware-Moduls m gilt die Bedingung $a_h(m) \cdot N_{row} \geq N_{res}(d)$, so dass die Breite eines Hardware-Moduls wie folgt berechnet werden kann:

$$a_h(m) = \left\lceil \frac{N_{res}(d)}{N_{row}} \right\rceil \tag{3.15}$$

Die resultierenden Module haben daher eine minimale Breite.

Systemansatz mit fester Aufteilung: In dem Systemansatz mit fester Aufteilung wird die Fläche des FPGAs in N_{seg} gleichgroße Blöcke unterteilt. Die Modulfläche entspricht der Fläche eines Blocks und ist somit für alle Hardware-Module gleich. Die Höhe der Module entspricht der Höhe N_{row} des verwendeten FPGAs. Für eine homogene rekonfigurierbare Architektur mit einem Zelltyp ($N_{cell} = 1$) ergibt sich folgende Breite der Module:

$$a_h(m) = \left\lfloor \frac{N_{col}}{N_{seg}} \right\rfloor \tag{3.16}$$

Die aus der virtuellen Synthese hervorgehenden Hardware-Module haben idealisierte Modulflächen, so dass die Anzahl der ungenutzten Zellen innerhalb der Module gering ist. Ob sich die Hardware-Module bei vorgegebener Modulfläche in einem realen System synthetisieren lassen, wurde in [46] untersucht. Grundlage der Untersuchung waren verschiedene Systemkomponenten mit unterschiedlichem Ressourcenbedarf. Es konnte gezeigt werden, dass selbst bei strengen Flächenvorgaben Syntheseergebnisse erzeugt wurden, die nur geringfügig von den optimalen Syntheseergebnissen hinsichtlich der maximalen Taktfrequenz und der Verlustleistung abwichen. Daher sind die aus der virtuellen Synthese idealisiert berechneten Modulflächen in vielen Fällen mit den aus der realen Synthese hervorgehenden Modulflächen vergleichbar.

Die Funktionen im Block *Vorgaben* werden einmalig ausgeführt. Anhand des erzeugten Ablaufs von Komponentenanfragen und der aus der Synthese hervorgehenden Hardware-Module können verschiedene Simulationen mit unterschiedlichen Platzierungsverfahren und Systemansätzen durchgeführt werden. In dem Block *Simulation* führt die *virtuelle Anwendung* die Liste der Komponentenanfragen aus und richtet dementsprechende Komponentenanfragen und Komponentenfreigaben an die Ressourcen- und Konfigurationsverwaltung. Die Ressourcen- und Konfigurationsverwaltung bearbeitet die Anfragen und führt die Platzierung und Konfiguration von

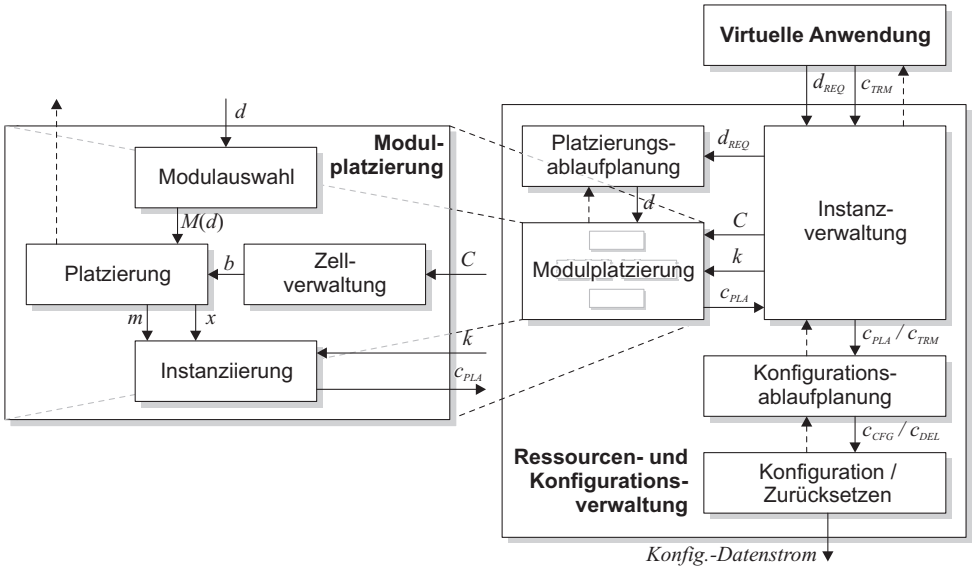


Abbildung 3.10: Schritte innerhalb der Ressourcen- und Konfigurationsverwaltung für die Platzierung eines Hardware-Moduls.

Hardware-Modulen aus. Da eine feste Reihenfolge an Komponentenanfragen vorliegt, können verschiedenen Simulationen mit unterschiedlichen Einstellungen durchgeführt werden, um auf diese Weise die einzelnen Platzierungsverfahren miteinander zu vergleichen. Abbildung 3.10 zeigt die Schritte für das Platzieren und Entfernen eines Hardware-Moduls innerhalb der Ressourcen- und Konfigurationsverwaltung.

Die *virtuelle Anwendung* fordert gemäß des gegebenen Ablaufs die Platzierung einer Systemkomponente $d_{REQ} \in D$. In der Regel wird die geforderte Systemkomponente über die Instanzverwaltung in der *Platzierungsablaufplanung* eingereiht. Eine alternative Herangehensweise ist die Überprüfung, ob eine bereits terminierte Modulinstanz eines Hardware-Moduls der geforderten Systemkomponente vorhanden ist, und diese zu reinitialisieren. Auf diese Weise kann die Platzierung eines neuen Hardware-Moduls vermieden werden. Das hier beschriebene Konzept der Reinitialisierung terminierter Modulinstanzen wird im Folgenden jedoch nicht betrachtet.

Sofern nicht anders vorgegeben, wird in der Simulationsumgebung das FCFS-Verfahren (vgl. Abschnitt 3.2.1) zur Bestimmung des Platzierungsablaufs verwendet. Die Platzierungsablaufplanung leitet demnach die am längsten wartende Komponentenanfrage d an die *Modulplatzierung* weiter. Innerhalb der *Modulplatzierung* bestimmt die *Modulauswahl* die Menge der Hardware-Module $M(d) \subseteq M$, die auf der Systemkomponente d basieren. Die *Zellverwaltung* berechnet die Zellbelegung b in Abhängigkeit der Menge der momentan platzierten Modulinstanzen C. Basierend auf $M(d)$ und b ermittelt die Platzierung ein passendes Modul $m \in M(d)$ mit dazugehöriger Position $x \in X_{pos}(m)$. Nachdem die Platzierung durchgeführt wurde, wird dem Platzierungsablauf signalisiert, ob die Platzierung erfolgreich war, und dass die

Modulplatzierung für die nächste Platzierung zur Verfügung steht. Bei erfolgreicher Platzierung erzeugt die *Instanziierung* eine neue Modulinstanz c_{PLA}, welche aus dem Modul m, der dazugehörigen Position x, dem initialen Zustand $s = PLA$ und dem Kommunikationsparameter k besteht.

Die Instanzverwaltung fügt die neue Modulinstanz c_{PLA} der Menge der momentan platzierten Modulinstanzen C hinzu und signalisiert der *virtuellen Anwendung*, dass die Instanz c_{PLA} platziert ist. Ebenso wird die Instanz c_{PLA} der *Konfigurationsablaufplanung* übermittelt. Die *Konfigurationsablaufplanung* legt die Reihenfolge der noch ausstehenden Konfigurations- und Löschvorgänge der entsprechenden Modulinstanzen fest. Ähnlich der Platzierungsablaufplanung wird bei der Konfigurationsablaufplanung standardmäßig das FCFS-Verfahren verwendet (vgl. Abschnitt 3.2.2). Im Fall einer Platzierung wird das *virtuelle FPGA* mit dem partiellen Konfigurationsdatenstrom der entsprechenden Instanz c_{CFG} rekonfiguriert. Beim Entfernen einer Instanz c_{DEL} werden die entsprechenden Zellen und deren Verbindungen in eine initiale Konfiguration zurückgesetzt. Das Zurücksetzen ist erforderlich, weil bei einer erneuten Konfiguration eines anderen Hardware-Moduls die zuvor gesetzten Verbindungen eine fehlerhafte Konfiguration verursachen können. Das Zurücksetzen der Zellen und deren Verbindungen geschieht durch Rekonfiguration des *virtuellen FPGAs* mit einem initialen partiellen Konfigurationsdatenstrom. Nach Beendigung des Konfigurations- oder des Löschvorgangs wird der Konfigurationsablaufplanung signalisiert, dass die Konfigurationsschnittstelle für den nächsten Vorgang bereit ist. Die Konfigurationsablaufplanung teilt der Instanzverwaltung mit, dass die Instanz c_{CFG} konfiguriert bzw. die Instanz c_{DEL} zurückgesetzt wurde. Die Instanzverwaltung meldet der *virtuellen Anwendung* die Zustandswechsel der einzelnen Instanzen.

Wenn eine Modulinstanz ihre Ausführung beendet hat und nicht mehr von der *virtuellen Anwendung* benötigt wird, dann wird der Instanzverwaltung mitgeteilt, dass die entsprechende Modulinstanz (c_{TRM}) entfernt werden kann. Die Instanzverwaltung signalisiert der Konfigurationsablaufplanung, dass die Zellen der Instanz c_{TRM} zurückgesetzt werden sollen.

Die Ressourcen- und Konfigurationsverwaltung protokolliert alle für die spätere Analyse relevanten Daten, wie z. B. die Positionen und Zustandswechsel der einzelnen Instanzen. Zu jeder generierten Instanz sind verschiedene Zeiten gespeichert worden, die im folgenden Abschnitt im Einzelnen definiert werden. Nach Beendigung einer Simulation können die protokollierten Simulationsdaten im Block *Analyse* bezüglich entsprechender Metriken (vgl. Abschnitt 4.2.1) ausgewertet und z. B. als graphische Funktionen dargestellt werden.

3.3.2 Zeitenmodell

Das in diesem Abschnitt definierte Zeitenmodell bezieht sich auf die von der *virtuellen Anwendung* kommenden Platzierungsanfragen, d. h., für jede Komponentenanfrage in der Liste $\sigma = (d_1, d_2, \ldots, d_{N_\sigma})$ ergeben sich die im Folgenden näher beschriebe-

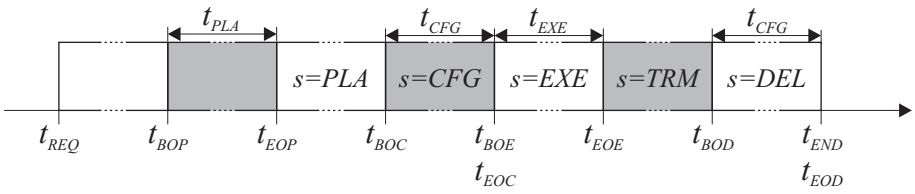


Abbildung 3.11: Zeitpunkte und Zeitspannen des Zeitenmodells.

nen Zeitpunkte. In Abbildung 3.11 ist der zeitliche Zusammenhang der verschiedenen Zeitpunkte dargestellt.

Platzierungsanfrage t_{REQ}: Die Platzierungsanfrage $t_{REQ}(d_i)$ ist der Zeitpunkt, bei dem die *virtuelle Anwendung* die Komponentenanfrage d_i an die Ressourcen- und Konfigurationsverwaltung richtet, die später die Erzeugung der Modulinstanz $c(d_i)$ zur Folge hat.

Platzierungsbeginn t_{BOP}: Der Platzierungsbeginn $t_{BOP}(d_i)$ stellt den Moment dar, in dem die Platzierungsablaufplanung die Komponentenanfrage d_i der Modulplatzierung übergibt.

Platzierungsende t_{EOP}: Das Platzierungsende $t_{EOP}(d_i)$ ist erreicht, wenn die Modulplatzierung beendet ist und die Instanziierung durchgeführt wurde. Von diesem Zeitpunkt an existiert die der geforderten Systemkomponente d_i entsprechende Modulinstanz $c(d_i)$, die sich zunächst in dem Zustand $s = PLA$ befindet.

Konfigurationsbeginn t_{BOC}: Der Konfigurationsbeginn $t_{BOC}(d_i)$ markiert den Zeitpunkt, in dem die Konfigurationsablaufplanung die Modulinstanz $c(d_i)$ der Konfiguration übergibt und der Konfigurationsvorgang beginnt. Zu diesem Zeitpunkt wechselt die Instanz $c(d_i)$ in den Zustand $s = CFG$.

Konfigurationsende t_{EOC}: Das Konfigurationsende $t_{EOC}(d_i)$ stellt das Ende des Konfigurationsvorgangs dar. Zu diesem Zeitpunkt sind die der Modulinstanz $c(d_i)$ zugewiesenen Zellen des FPGAs dem Konfigurationsdatenstrom entsprechend rekonfiguriert. Der *virtuellen Anwendung* wurden die für die Kommunikation relevanten Parameter (k) übermittelt.

Ausführungsbeginn t_{BOE}: Der Ausführungsbeginn $t_{BOE}(d_i)$ beschreibt den Moment in dem die aktive Ausführung der Modulinstanz $c(d_i)$ begonnen hat. Dementsprechend wechselt die Modulinstanz $c(d_i)$ in den Zustand $s = EXE$.

Ausführungsende t_{EOE}: Das Ausführungsende $t_{EOE}(d_i)$ ist der Zeitpunkt, an dem die *virtuelle Anwendung* signalisiert, dass die Modulinstanz $c(d_i)$ terminiert ist und nicht mehr benötigt wird. Demnach wechselt die Modulinstanz $c(d_i)$ in den Zustand $s = TRM$ und wird der Konfigurationsablaufplanung übergeben.

Löschbeginn t_{BOD}: Der Löschbeginn $t_{BOD}(d_i)$ beschreibt den Zeitpunkt, in dem die Konfigurationsablaufplanung den Löschvorgang der Modulinstanz $c(d_i)$ einleitet. Die der Modulinstanz zugewiesenen Zellen werden durch einen initialen Konfigurationsdatenstrom zurückgesetzt. Die Modulinstanz $c(d_i)$ wechselt in den Zustand $s = DEL$.

Löschende t_{EOD}: Das Löschende $t_{EOD}(d_i)$ markiert den Moment, in dem alle Zellen der Modulinstanz $c(d_i)$ zurückgesetzt sind und der Löschvorgang beendet ist.

Freigabezeitpunkt t_{END}: Der Freigabezeitpunkt $t_{END}(d_i)$ ist der Moment, in dem die Modulinstanz $c(d_i)$ aus der Menge der momentan platzierten Modulinstanzen C entfernt wird. Von diesem Zeitpunkt an existiert die Instanz nicht mehr.

Die Zeitpunkte, in der Reihenfolge, wie sie oben aufgeführt sind, beschreiben erneut den chronologischen Ablauf der Zustände einer Modulinstanz c. Dementsprechend ergibt sich folgende Abhängigkeit der Zeitpunkte:

$$\begin{aligned} t_{REQ}(d_i) \leq t_{BOP}(d_i) \leq t_{EOP}(d_i) \leq t_{BOC}(d_i) \leq t_{EOC}(d_i) \\ \leq t_{BOE}(d_i) \leq t_{EOE}(d_i) \leq t_{BOD}(d_i) \leq t_{EOD}(d_i) \leq t_{END}(d_i) \end{aligned} \quad (3.17)$$

Neben den Zeitpunkten lassen sich zu jeder Instanz die folgenden Zeiten zuordnen:

Platzierungszeit t_{PLA}: Die Platzierungszeit $t_{PLA}(d_i)$ entspricht der von der Modulplatzierung benötigten Zeit für die Platzierung des Moduls m der Instanz $c(d_i)$. Bezüglich der Zeitpunkte gilt

$$t_{EOP}(d_i) = t_{BOP}(d_i) + t_{PLA}(d_i) \; . \quad (3.18)$$

Die Platzierungszeit hängt im Wesentlichen von der Ausführungszeit des gewählten Platzierungsalgorithmus ab.

Konfigurationszeit t_{CFG}: Die Konfigurationszeit $t_{CFG}(d_i)$ beschreibt die Zeit, die für die Rekonfiguration des *virtuellen FPGAs* anhand des partiellen Konfigurationsdatenstroms benötigt wird. Für die in den folgenden Kapiteln betrachteten Simulationen sei angenommen, dass die Zeit für den Konfigurationsvorgang mit der Zeit für den entsprechenden Löschvorgang identisch ist. Daher sei

$$t_{EOC}(d_i) = t_{BOC}(d_i) + t_{CFG}(d_i) \; , \quad (3.19)$$

$$t_{EOD}(d_i) = t_{BOD}(d_i) + t_{CFG}(d_i) \; . \quad (3.20)$$

Wie in [E7] beschrieben, kann man die Konfigurationszeit eines partiellen Konfigurationsdatenstroms für Xilinx Virtex-II FPGAs wie folgt abschätzen: Sei $N_{CLB-COL}$ die Anzahl der zu konfigurierenden CLB-Spalten und $N_{RAM-COL}$

die Anzahl der zu konfigurierenden Block-RAM-Spalten. $N_{Byte/Frame}$ sei eine typenspezifische Konstante, die von der Anzahl der Zellen pro Spalte abhängt (vgl. [92]). $f_{SelectMap}$ sei die Taktfrequenz der Konfigurationsschnittstelle. Dann ist

$$t_{CFG} \approx \frac{(22 \cdot N_{CLB-COL} + 86 \cdot N_{RAM-COL}) \cdot N_{Bytes/Frame}}{f_{SelectMap}} . \tag{3.21}$$

Beispiel 3.5. *Gegeben sei ein Modul $m = (d, r, a, q) \in M$, welches auf der Systemkomponente $d = d_i$ basiert. Das Modul sei für ein Xilinx Virtex-II $XC2V4000$ synthetisiert. Der Ressourcenvektor sei $r = (r_{CLB})$, so dass das Modul keine RAM-Zellen enthält und $N_{RAM-COL} = 0$ ist. Der Flächenbedarf sei gegeben durch $a = (a_h, a_v)$ und der Kostenparameter sei gegeben durch $q = (t_{CFG})$. Bei einem $XC2V4000$ ist $N_{Byte/Frame} = 824$. Die Taktfrequenz der Konfigurationsschnittstelle sei $f_{SelectMap} = 50 \cdot 10^6 Hz$. Die Anzahl der zu konfigurierenden CLB-Spalten entspricht der Breite des Moduls, so dass $N_{CLB-COL} = a_h(m)$. Für eine Modulinstanz $c = (m, x, s, k)$, die auf dem Modul m basiert, ist die Konfigurationszeit etwa*

$$t_{CFG}(d_i) \approx a_h(m) \cdot 362,65 \cdot 10^{-6} s . \tag{3.22}$$

Bei Xilinx Virtex-II (Pro) FPGAs hängt demnach die Konfigurationszeit eines Moduls im Wesentlichen von der Anzahl der verwendeten CLB-Spalten und der Anzahl der verwendeten Block-RAM-Spalten ab. Zusätzlich können die Konfigurationsdaten von Xilinx Virtex-II (Pro) FPGAs komprimiert werden (Multiple-Frame-Write), was in einigen Fällen zur Verringerung der Konfigurationszeit führt (vgl. [92]). Insbesondere beim Löschvorgang, bei dem die Konfigurationen der Zellen identisch sind, lässt sich die benötigte Zeit durch Verwendung von Kompressionsverfahren verringern. Wie in [E7] beschrieben, kann daher der Löschvorgang in einigen Fällen auf bis zu 5% der ursprünglichen Dauer verkürzt werden. Die oben dargestellte Abschätzung zur Berechnung der Konfigurationszeit kann daher als Worst-Case Abschätzung betrachtet werden.

Ausführungszeit t_{EXE}: Die Ausführungszeit $t_{EXE}(d_i)$ beschreibt die aktive Rechenzeit der der Modulinstanz zugewiesenen Zellen. Aus Sicht der Ressourcen- und Konfigurationsverwaltung ist die Ausführungszeit der einzelnen Modulinstanzen unbekannt. Die *virtuelle Anwendung* indes kennt die einzelnen Ausführungszeiten der Systemkomponenten aus der Liste der Komponentenanfragen σ. Sobald die Ressourcen- und Konfigurationsverwaltung der *virtuellen Anwendung* den Beginn der Ausführung einer Instanz signalisiert, kann die *virtuelle Anwendung* anhand der aus den Vorgaben bekannten Ausführungszeit das

Ausführungsende der Instanz bestimmen. Bezüglich der Zeitpunkte gilt demnach

$$t_{EOE}(d_i) = t_{BOE}(d_i) + t_{EXE}(d_i) \; . \tag{3.23}$$

Für die in den folgenden Kapiteln betrachteten Simulationen sei angenommen, dass die Ausführung einer Modulinstanz unmittelbar nach dem Ende des Konfigurationsvorgangs beginnt. Daher gilt

$$t_{BOE}(d_i) = t_{EOC}(d_i) \; . \tag{3.24}$$

Ebenso sei angenommen, dass der Freigabezeitpunkt $t_{END}(d_i)$ mit dem Löschende übereinstimmt, so dass gilt

$$t_{END}(d_i) = t_{EOD}(d_i) \; . \tag{3.25}$$

Neben den Zeiten t_{PLA}, t_{CFG}, und t_{EXE} können sich die folgenden Verzögerungen ergeben. Wenn $t_{BOP}(d_i) - t_{REQ}(d_i) > 0$, dann konnte die geforderte Systemkomponente d_i nicht sofort platziert werden, da die Modulplatzierung noch mit der Platzierung einer vorherigen Komponentenanfrage beschäftigt war. Die Platzierung der Komponentenanfrage d_i wurde daher durch die Platzierungsablaufplanung um die Zeit $t_{BOP}(d_i) - t_{REQ}(d_i)$ verzögert.

Wenn $t_{BOC}(d_i) - t_{EOP}(d_i) > 0$, dann konnte die Konfiguration der Modulinstanz $c(d_i)$ nicht sofort nach der Instanziierung durchgeführt werden, da zu dem Zeitpunkt $t_{EOP}(d_i)$ die Konfigurationsschnittstelle mit der Konfiguration oder dem Zurücksetzen einer anderen Modulinstanz beschäftigt war. Die Konfiguration der Modulinstanz $c(d_i)$ wurde daher durch die Konfigurationsablaufplanung um die Zeit $t_{BOC}(d_i) - t_{EOP}(d_i)$ verzögert. Gleiches gilt, wenn $t_{BOD}(d_i) - t_{EOE}(d_i) > 0$, d. h., das Zurücksetzen der Modulinstanz $c(d_i)$ konnte nicht sofort nach der Instanziierung durchgeführt werden, da zu dem Zeitpunkt $t_{EOP}(d_i)$ die Konfigurationsschnittstelle belegt war.

Für den Fall, dass eine Systemkomponente d_i nicht platziert werden konnte und die Platzierungsablaufplanung eine Platzierung zu einem späteren Zeitpunkt nicht vorsieht, werden folgende Annahmen getroffen:

$$t_{BOC}(d_i) = t_{EOC}(d_i) = t_{BOE}(d_i) = t_{EOE}(d_i) = t_{BOD}(d_i) = t_{EOD}(d_i) = \infty \tag{3.26}$$

Auf diese Weise bleiben die in (3.17) dargestellten Abhängigkeiten der Zeitpunkte bestehen.

3.4 Zusammenfassung

Das in diesem Kapitel vorgestellte DMC-Modell bildet die Grundlage der in den folgenden Kapiteln aufgeführten Methoden zur partiellen Rekonfigurierbarkeit. Wesentlicher Bestandteil des Modells ist die Unterteilung in die drei Abstraktionsebenen

Systemkomponente, *Hardware-Modul* und *Modulinstanz*. Eine Systemkomponente stellt die Spezifikation eines Hardware-Entwurfs dar, wobei ein Hardware-Modul eine konkrete Implementierung einer Systemkomponente für eine bestimmte rekonfigurierbare Architektur ist. Eine Modulinstanz besteht aus einem Hardware-Modul, welches an einer bestimmten Position platziert ist und über eine individuelle Adresse an die systemweite Verbindungsstruktur angebunden ist. Eine Modulinstanz kann sich im Zustand *platziert* (PLA), *Konfiguration* (CFG), *Ausführung* (EXE), *terminiert* (TRM) und *Löschen* (DEL) befinden.

Da sowohl die Platzierung, als auch die Konfigurations- und Löschvorgänge einer Modulinstanz nur sequenziell erfolgen können, ist bei mehreren gleichzeitig vorhandenen Platzierungs- oder Konfigurationsanfragen eine Ablaufplanung erforderlich. Die Platzierungsablaufplanung dient zur Festlegung der Platzierungsreihenfolge der momentan angeforderten Systemkomponenten und zur Handhabung fehlgeschlagener Modulplatzierungen. Die Konfigurationsablaufplanung dient zur Bestimmung der Reihenfolge der Konfigurations- und Löschvorgänge der Modulinstanzen.

Die Qualität eines Online-Algorithmus kann analytisch mit der kompetitiven Analyse bewertet werden. Voraussetzung einer kompetitiven Analyse ist jedoch die Existenz eines optimalen Offline-Algorithmus. Bezüglich partiell rekonfigurierbarer Architeturen ist die Bestimmung der optimalen Lösung des Offline-Platzierungsproblems jedoch mit heutigen Verfahren nur in Teilen gelöst, so dass die kompetitive Analyse von Online-Platzierungsverfahren nicht ohne weiteres möglich ist. Aus diesem Grund wird in vielen Arbeiten (z. B. [8, 12, 29, 72]) die Analyse von Methoden zur partiellen Rekonfigurierbarkeit anhand von Simulationen vorgenommen. Die Analyse der in den folgenden Kapiteln beschriebenen Methoden wird anhand der Simulationsumgebung SARA durchgeführt, welche aus den Blöcken *Vorgaben*, *Simulation* und *Analyse* besteht. In dem Block *Vorgaben* werden die für die Simulation benötigten Hardware-Module erzeugt und eine dazu passende Liste von zufällig erzeugten Komponentenanfragen. In dem Block *Simulation* verwendet die *virtuelle Anwendung* die Liste der zufällig erzeugten Komponentenanfragen und gibt entsprechende Platzierungs- und Löschanfragen an die *Ressourcen- und Konfigurationsverwaltung* weiter. Die *Ressourcen- und Konfigurationsverwaltung* führt unter anderem die Modulplatzierung der entsprechenden Komponentenanfragen durch und verwaltet die freien Ressourcen, ebenso wie die vorhandenen Modulinstanzen. Nach Beendigung der Simulation können die Simulationsdaten entsprechend verschiedener Metriken analysiert werden. Als Grundlage der Analyse dient das in diesem Kapitel beschriebene Zeitenmodell, welches zu jeder Komponentenanfrage verschiedene Zeitpunkte und Zeitspannen zuordnet.

4 Homogene Architekturen

Die einfachste Form partiell rekonfigurierbarer Architekturen stellen die homogenen Architekturen dar. Sie zeichnen sich dadurch aus, dass in der Regel nur ein Typ von Zellen vorhanden ist und die Kommunikationsinfrastruktur derart ausgelegt ist, dass eine uneingeschränkte feingranulare Platzierung der Hardware-Module ermöglicht werden kann.

Obwohl die meisten modernen partiell rekonfigurierbaren Architekturen (z. B. Xilinx Virtex-II FPGAs [92]) Heterogenitäten durch das Vorhandensein mehrerer Zelltypen aufweisen, so gibt es dennoch homogene Architekturen, wie z. B. das in Compton et al. [22] spezifizierte R/D FPGA. Frühe rekonfigurierbare Architekturen, wie z. B. das Xilinx XC6200 FPGA [86] oder die Xilinx Virtex FPGA Serie [88], können ebenfalls als homogen betrachtet werden, da eine feingranulare Platzierung der Hardware-Module möglich ist. Lediglich die hierarchisch aufgebaute Kommunikationsinfrastruktur schränkt die Platzierung ein. Im Zusammenhang mit partieller Rekonfigurierbarkeit wird der Einfachheit halber in vielen Arbeiten (z. B. [8, 72]) ebenfalls eine homogene rekonfigurierbare Architektur zugrunde gelegt.

In diesem Kapitel wird zunächst in Abschnitt 4.1 auf die Platzierungsverfahren im Zusammenhang mit homogenen rekonfigurierbaren Architekturen eingegangen. Die Platzierung eines Hardware-Moduls teilt sich dabei grundsätzlich in die Verwaltung der freien Ressourcen und die Bestimmung der Position eines geforderten Hardware-Moduls zur Laufzeit auf. Im Abschnitt 4.2 werden die verschiedenen Systemansätze zur Realisierung partieller Rekonfigurierbarkeit anhand von Simulationen miteinander verglichen. Dazu werden unterschiedliche Metriken zur Bewertung der Systemansätze definiert. Neben dem Vergleich der Systemansätze werden ebenso die Einflüsse der Platzierungszeiten und der Konfigurationszeiten der Modulinstanzen im Hinblick auf den 1D-Systemansatz analysiert.

Eine Methode zur Verbesserung der Leistungsfähigkeit eines Platzierungsansatzes stellt die Defragmentierung dar. Die Defragmentierung ermöglicht das Bündeln der freien Ressourcen durch Neuanordnung bereits vorhandener Modulinstanzen. In Abschnitt 4.3 werden Methoden zur Defragmentierung für den 1D-Systemansatz vorgestellt. Anhand von Simulationen wird der Einfluss der Defragmentierung auf die Leistungsfähigkeit eines Platzierungsverfahrens analysiert.

4.1 Platzierungsverfahren

Im Folgenden sei angenommen, dass die Anfragezeitpunkte und Ausführungszeiten der von der Anwendung geforderten Systemkomponenten unbekannt sind. Wie in homogenen rekonfigurierbaren Architekturen üblich, sei ferner angenommen, dass die gegebenen Hardware-Module uneingeschränkt feingranular platziert werden können. Sei N_{col} die Anzahl der Spalten der Architektur und N_{row} die Anzahl der Reihen. Die Menge der möglichen Positionen $X_{pos}(m)$ eines Moduls $m \in M$ ergibt sich demnach wie folgt:

$$X_{pos}(m) = \big\{(x_h, x_v) \,|1 \leq x_h \leq N_{col} - a_h(m) + 1 \wedge 1 \leq x_v \leq N_{row} - a_v(m) + 1\big\} \tag{4.1}$$

Unter den gegebenen Annahmen kann die Platzierung eines Hardware-Moduls nur zur Laufzeit in Abhängigkeit der Menge der platzierten Modulinstanzen C durchgeführt werden. Bazargan et al. haben in [8] gezeigt, dass die Platzierung eines Hardware-Moduls zur Laufzeit mit einem zweidimensionale Online-Packungsproblem vergleichbar ist. Das zweidimensionale Online-Packungsproblem kann als Erweiterung eines klassischen eindimensionalen Packungsproblems aufgefasst werden, welches von Coffman et al. [19] wie folgt definiert wurde:

Definition 4.1 (Eindimensionales Packungsproblem)**.** *Gegeben sei eine Liste von Objekten $\sigma_{PP} = (o_1, o_2, \ldots, o_{N_{PP}})$, wobei jedes Objekt eine Größe $a(o_i) \in (0,1]$ besitzt. Die Objekte der Liste σ_{PP} sollen nacheinander in eine Menge von Boxen B mit konstanter Größe (=1) gepackt werden mit dem Ziel, die Anzahl der mit Objekten bepackten Boxen zu minimieren. Daher gilt*

$$\forall j \in [1, N_B] : \sum_{o \in B_j} a(o) \leq 1,$$

wobei jedes Objekt der Liste $\sigma_{PP} = (o_1, o_2, \ldots, o_{N_{PP}})$ in einer Box vorhanden ist

$$\bigcup_{j=1}^{N_B} B_j = \{o_1, o_2, \ldots, o_{N_{PP}}\}.$$

D. h., die Anzahl N_B der Teilmengen $B_1, B_2, \ldots, B_{N_B}$ soll minimiert werden unter der Bedingung, dass die Boxen nicht überfüllt werden. Bezüglich der Modulplatzierung können freie zusammenhängende Flächen der rekonfigurierbaren Architektur als Boxen aufgefasst werden und das Hardware-Modul der geforderten Systemkomponente als das zu platzierende Objekt. Im Moment der Platzierung ergeben sich daher zu einem gegebenen Objekt (Hardware-Modul) verschieden große Boxen (freie zusammenhängende Flächen). Es existieren dennoch Unterschiede zum klassischen Packungsproblem, wenn man die Modulplatzierungen mehrerer in zeitlichen

Abständen aufeinander folgender Komponentenanfragen betrachtet. Im Gegensatz zum klassischen Packungsproblem, bei dem die Objekte nach Zuordnung in der Box verbleiben, werden im Zusammenhang mit rekonfigurierbarer Hardware abgelaufene Modulinstanzen wieder von der Architektur entfernt. Aussagen, die sich über das zeitliche Verhalten von Verfahren zum Lösen des klassischen Packungsproblems machen lassen, sind daher nicht ohne Weiteres auf das Problem der Modulplatzierung übertragbar. Äquivalenz besteht daher nur für den Moment der Platzierung.

Wie schon in Abschnitt 3.1.2 beschrieben, ergeben sich bei der Platzierung die folgenden zwei Schritte. Im ersten Schritt werden die freien zusammenhängenden Flächen der rekonfigurierbaren Architektur bestimmt und daraus die Menge der freien Positionen $X_{free}(m)$ für jedes Hardware-Modul $m \in M(d)$ hergeleitet. Im zweiten Schritt wird ein passendes Hardware-Modul $m \in M(d)$ mit dazugehöriger Position $x \in X_{free}(m)$ ermittelt. Entsprechend dieser Schritte wird in Abschnitt 4.1.1 auf die verschiedenen Verfahren zur Verwaltung der freien Zellen eingegangen. Abschnitt 4.1.2 beschäftigt sich mit Algorithmen zur Bestimmung der Position $x \in X_{free}(m)$ eines gegebenen Hardware-Moduls $m \in M(d)$.

4.1.1 Verwaltung freier Zellen

Bezüglich der Äquivalenz zum Packungsproblem werden die freien Zellen in Form von rechteckigen Flächen, die als Boxen dienen, verwaltet. Die Menge aller freien Rechtecke ergibt sich anhand des DMC-Modells wie folgt:

Definition 4.2 (Menge aller freien Rechtecke)**.** *Sei N_{col} die Anzahl der Spalten und N_{row} die Anzahl der Reihen der rekonfigurierbaren Architektur. Ferner sei b die aktuelle Zellbelegung der Architektur. Ein freies Rechteck $e = (x_{eh}, x_{ev}, a_{eh}, a_{ev}) \in E$ wird beschrieben durch die horizontale Position $x_{eh} \in [1, N_{col}]$ und die vertikale Position $x_{ev} \in [1, N_{row}]$ der unteren linken freien Zelle. Die Breite des freien Rechtecks wird durch $a_{eh} \in [1, N_{col} - x_{eh} + 1]$ und die Höhe durch $a_{ev} \in [1, N_{row} - x_{ev} + 1]$ dargestellt. Dann ist die* Menge aller freien Rechtecke *gegeben durch:*

$$E_{all} = \big\{(x_{eh}, x_{ev}, a_{eh}, a_{ev}) \mid i \in [x_{eh}, x_{eh} + a_{eh}) \ \wedge \ j \in [x_{ev}, x_{ev} + a_{ev}) \ \wedge \ \forall i, j : b(i, j) = 1\big\} \tag{4.2}$$

Um alle ungenutzten Zellen der rekonfigurierbaren Architektur zu erfassen, haben Bazargan et al. in [8] die Verwendung *maximaler freier Rechtecke* (engl.: Maximum Empty Rectangle) vorgeschlagen.

Definition 4.3 (Menge der maximalen freien Rechtecke)**.** *Ein freies Rechteck $e \in E_{all}$ ist dann maximal, wenn es sich in keine Richtung ausweiten lässt. Daher ergibt sich die* Menge der maximalen freien Rechtecke *wie folgt:*

$$\begin{aligned} E_{max} = \big\{ (x_{eh}, x_{ev}, a_{eh}, a_{ev}) \mid (x_{eh}, x_{ev}, a_{eh}, a_{ev}) \in E_{all} \wedge \\ \forall (\acute{x}_{eh}, \acute{x}_{ev}, \acute{a}_{eh}, \acute{a}_{ev}) \in E_{all} \setminus (x_{eh}, x_{ev}, a_{eh}, a_{ev}) : \\ \acute{x}_{eh} > x_{eh} \vee \acute{x}_{ev} > x_{ev} \vee \\ \acute{x}_{eh} + \acute{a}_{eh} < x_{eh} + a_{eh} \vee \acute{x}_{ev} + \acute{a}_{ev} < x_{ev} + a_{ev} \big\} \end{aligned} \tag{4.3}$$

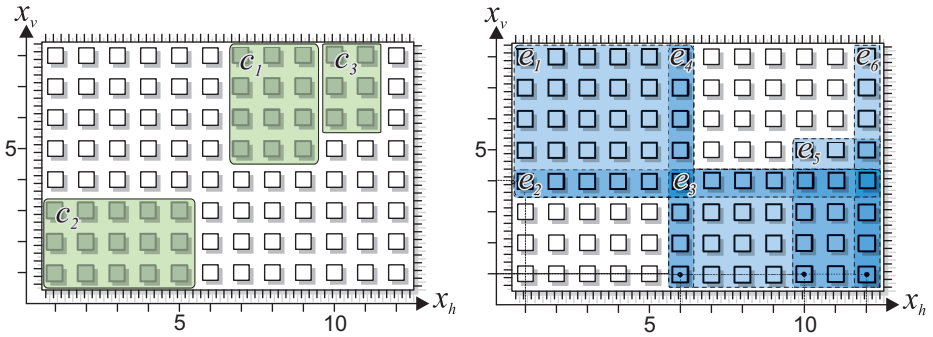
D. h., ein maximales freies Rechteck ist in keinem anderen freien Rechteck der Menge E_{all} enthalten.

Beispiel 4.1. *Gegeben sei eine homogene rekonfigurierbare Architektur mit $N_{col} = 12$ Spalten und $N_{row} = 8$ Reihen. Die Menge der aktuell platzierten Modulinstanzen sei $C = \{c_1, c_2, c_3\}$, wobei $c_1 = (m_1, (7,5), EXE, (1))$, $c_2 = (m_2, (1,1), EXE, (2))$ und $c_3 = (m_3, (10,6), EXE, (3))$. Die Flächenausdehnung der entsprechenden Module sei $a(m_1) = (3,4)$, $a(m_2) = (5,3)$ und $a(m_3) = (2,3)$. Die resultierende Menge der maximalen freien Rechtecke ist:*

$$E_{max} = \{e_1, e_2, e_3, e_4, e_5, e_6\}$$

$$\begin{aligned} e_1 &= (1,4,6,5) & e_2 &= (1,4,12,1) \\ e_3 &= (6,1,7,4) & e_4 &= (6,1,1,8) \\ e_5 &= (10,1,3,5) & e_6 &= (12,1,1,12) \end{aligned}$$

Die entsprechenden maximalen freien Rechtecke sind in Abbildung 4.1 dargestellt.



(a) Derzeit konfigurierte Modulinstanzen. (b) Maximale freie Rechtecke.

Abbildung 4.1: Beispiel der Menge der maximalen freien Rechtecke bei gegebener Menge der momentan platzierten Modulinstanzen.

Die Menge der maximalen freien Rechtecke E_{max} hängt von der aktuellen Zellbelegung b der Architektur ab. Daher ändert sich E_{max} nicht nur nach jeder Modulplatzierung, sondern ebenso nach jedem Löschen einer ausgeführten Modulinstanz. Um die Menge der maximalen freien Rechtecke E_{max} zu bestimmen, haben Edmonds et al. [28] ein Verfahren vorgestellt, welches eine Laufzeit von $\mathcal{O}(N_{col} \cdot N_{row})$ und einen Speicherbedarf von $\mathcal{O}(\min(N_{col}, N_{row}))$ benötigt. Das Verfahren ermittelt zunächst so genannte *Treppen*, aus denen in einem zweiten Schritt die maximalen freien Rechtecke extrahiert werden. Eine Treppe $s_t(x_h, x_v)$ wird beschrieben durch die Position (x_h, x_v) der unteren rechten Ecke und der Menge der oberen linken Ecken aller der für den Punkt (x_h, x_v) nach oben links verlaufenden maximalen freien rechteckigen Flächen. Die Treppe $s_t(x_h, x_v)$ mit $x_h > 1$ kann wie in [28] beschrieben auf einfache Weise aus der vorherigen Treppe $s_t(x_h - 1, x_v)$ konstruiert werden.

Beispiel 4.2. *Gegeben sei die Menge der momentan platzierten Modulinstanzen aus Beispiel 4.1. Die in Abbildung 4.2 dargestellte Treppe wird beschrieben durch* $s_t(11,2) = \{(10,5),(6,4)\}$.

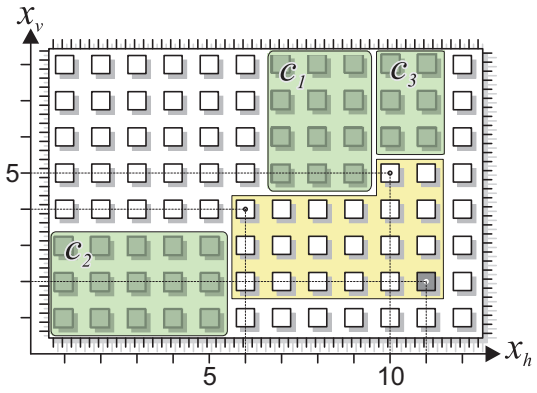


Abbildung 4.2: Beispiel einer Treppe bei gegebener Menge der momentan platzierten Modulinstanzen.

Handa et al. [38] haben die in [28] definierte Treppen-Datenstruktur verwendet, um ein effizientes Verfahren im Hinblick auf partiell rekonfigurierbare Architekturen zu beschreiben. Dabei haben Handa et al. folgende Sätze bewiesen:

1. Jedes maximale freie Rechteck ist in genau einer Treppe enthalten. Eine maximale Treppe ist eine Treppe, welche mindestens ein maximales freies Rechteck enthält.

2. Jede maximale Treppe befindet sich in der Zeile oberhalb einer Modulinstanz oder in der untersten Zeile der rekonfigurierbaren Architektur.

3. Die untere rechte Ecke jeder maximalen Treppe liegt in der Spalte links neben einer Modulinstanz oder in der äußersten rechten Spalte der rekonfigurierbaren Architektur.

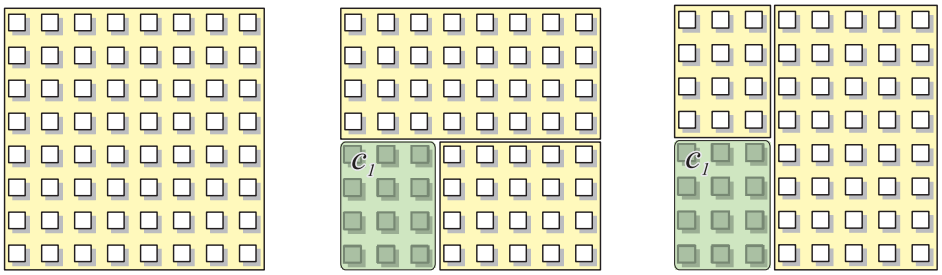
Auf diese Weise wird die Suche nach den Positionen der maximaler Treppen auf wenige Punkte eingeschränkt. Sei N_{toprow} die Anzahl der Zeilen, die unmittelbar über mindestens einer Modulinstanz liegen. Dann entspricht laut [38] die Anzahl der benötigten Schritte zur Erzeugung aller maximalen freien Rechtecke höchstens $(N_{toprow}+1)\cdot N_{col}$, was ebenfalls eine Laufzeit von $\mathcal{O}(N_{col}\cdot N_{row})$ bedeutet.

Die obere Schranke maximaler freier Rechtecke ist gemäß Naamad et al. [59] auf $\mathcal{O}(|C|^2)$ begrenzt, wobei $|C|$ der Anzahl der momentan platzierten Modulinstanzen ist. Edmonds et al. [28] haben zusätzlich gezeigt, dass die obere Schranke maximaler freier Rechtecke ebenso durch die Anzahl der freien Zellen auf $\mathcal{O}(N_{free})$ begrenzt ist, wobei $N_{free}=\sum_{i,j} b(i,j)$ mit $i\in[1,N_{col}]$ und $j\in[1,N_{row}]$ ist.

Gemäß [8] ist die Verwendung eines Verfahrens zur Bestimmung aller maximalen freien Rechtecke insofern optimal, als dass zu jedem platzierbaren Modul $m\in M$ mit $X_{free}(m)\notin\{\}$ immer ein maximales freies Rechteck $e\in E_{max}$ existiert, in dem das Modul nach der Platzierung vollständig enthalten wäre. D. h., wenn die Platzierung eines Moduls $m\in M$ möglich ist, findet das entsprechende Platzierungsverfahren auch immer eine passende freie Position $x\in X_{free}(m)$.

In Abhängigkeit der Größe der Architektur kann die Anzahl aller maximalen freien Rechtecke so groß sein, dass die Bestimmung aller maximalen freien Rechtecke zur Laufzeit zu aufwendig ist, so dass Bazargan et al. [8] ein heuristisches Verfahren entwickelt haben, welches die freien Flächen der Architektur in nicht überlappende freie Rechtecke partitioniert. Ein zu platzierendes Hardware-Modul wird dabei immer in der unteren linken Ecke eines gewählten freien Rechtecks platziert. Nach der Platzierung wird die verbleibende freie Fläche in der Regel in zwei neue nicht überlappende Rechtecke partitioniert. Wie in Abbildung 4.3 dargestellt, ergeben sich zwei Möglichkeiten der Partitionierung. In [8] werden verschiedene Verfahren vorgestellt, die in Abhängigkeit von unterschiedlichen Kriterien über eine horizontale oder vertikale Partitionierung entscheiden. Eine ungünstige Partitionierung der Rechtecke kann dazu führen, dass ein gefordertes Hardware-Modul $m\in M$ nicht platziert werden kann, obwohl hinreichend viele freie zusammenhängende Zellen verfügbar sind $\left(X_{free}(m)\notin\{\}\right)$. In diesem Sinn ist die Verwendung freier nicht überlappender Rechtecke nicht optimal.

Nach dem Löschen einer Modulinstanz entsteht ein neues freies Rechteck, welches der Fläche der gelöschten Modulinstanz entspricht. Um die Anzahl der freien Rechtecke gering zu halten, wird in [8] ein Verfahren beschrieben, welches gegebenenfalls benachbarte freie Rechtecke zu einem gemeinsamen großen freien Rechteck zusammenführt. Die präsentierten heuristischen Verfahren freier nicht überlappender Rechtecke wurden durch Simulationen mit dem Verfahren der Erhaltung aller maximalen freien Rechtecke verglichen. Entsprechende Simulationsergebnisse belegen,



(a) Freies Rechteck vor der Platzierung.

(b) Resultierende freie Rechtecke nach horizontaler Partitionierung.

(c) Resultierende freie Rechtecke nach vertikaler Partitionierung.

Abbildung 4.3: Beispiel für die Möglichkeiten der Partitionierung freier Rechtecke nach einer Modulplatzierung.

dass die Verwendung aller maximalen freien Rechtecke zu einer besseren Platzierung führt.

In [82] haben Walder et al. das zuvor beschriebene Verfahren von Bazargan et al. [8] unter anderem wie folgt erweitert. Bei dem beschriebenen Verfahren wird die Entscheidung der horizontalen oder vertikalen Partitionierung der verbleibenden freien Fläche nicht unmittelbar nach der Modulplatzierung getroffen, sondern zum Zeitpunkt der nächsten Modulplatzierung in Abhängigkeit des Flächenbedarfs des geforderten Hardware-Moduls. Darüber hinaus werden die freien Rechtecke nur dann verkleinert, wenn sie mit dem zu platzierenden Modul überlappen. Anhand von Simulationen konnte gezeigt werden, dass im Vergleich zu dem heuristischen Verfahren von [8] die Qualität der Platzierung um bis zu 70% verbessert werden konnte.

Die obigen Verfahren zur Verwaltung freier Zellen beziehen sich auf das Lösen des Packungsproblems und verfolgen daher die Bestimmung einer Menge von freien rechteckigen Flächen. Eine alternative Herangehensweise zur Verwaltung der freien Zellen ist von Ahmadinia et al. [1] vorgestellt worden. Das Verfahren bestimmt die Menge der freien Positionen eines geforderten Hardware-Moduls, indem zunächst die Regionen identifiziert werden, in denen das Modul anhand der momentan platzierten Modulinstanzen C nicht platziert werden kann. In Analogie zu (3.1) und (3.2) bildet die Differenz aller möglichen Positionen und der unmöglichen Regionen die Menge der freien Positionen $X_{free}(m)$ für das geforderte Hardware-Modul $m \in M$. Der Vorteil des Verfahrens gegenüber der Verwendung maximaler freier Rechtecke ist, dass bei der Berechnung der freien Positionen von den belegten Ressourcen ausgegangen wird, so dass der Speicherbedarf nur $\mathcal{O}(|C|)$ ist.

Die in diesem Abschnitt genannten Verfahren zur Verwaltung der freien Ressourcen bilden die Grundlage der im folgenden Abschnitt beschriebenen Verfahren zur Bestimmung der Position eines gegebenen Hardware-Moduls.

4.1.2 Positionsbestimmung

Im vorherigen Abschnitt wurde auf die Berechnung der Position und der Größe der freien rechteckigen Flächen eingegangen. In Bezug auf das gegebene Packungsproblem dienen die freien rechteckigen Flächen als Boxen und das zu platzierende Hardware-Modul als Objekt. Dieser Abschnitt beschäftigt sich mit Methoden zur Auswahl einer passenden Box zu dem gegebenen Objekt. Die Zuordnung einer Box zu dem Objekt ist gleichzusetzen mit der Zuordnung einer Position zu dem geforderten Hardware-Modul. Im Zusammenhang mit dem in Definition 4.1 charakterisierten eindimensionalen Packungsproblemen existieren unter anderem die folgenden in [19] aufgeführten Approximationsverfahren:

First-Fit: Bei dem *First-Fit*-Verfahren wird das zu platzierende Objekt o_{req} der ersten gefundenen Box B_j zugeordnet, die hinreichend Platz bietet und daher die folgende Bedingung erfüllt:

$$1 - \sum_{o \in B_j} a(o) \geq a(o_{req}) \tag{4.4}$$

Best-Fit: Das *Best-Fit*-Verfahren wählt die Box B_j, welche bei der Platzierung des Objektes o_{req} den geringsten verbleibenden Platz verursacht. Es wird das $j \in [1, N_B]$ ermittelt, welches $\sum_{o \in B_j} a(o)$ maximiert unter Berücksichtigung der aus dem First-Fit-Verfahren bekannten Bedingung (4.4).

Worst-Fit: Der gegenteilige Ansatz zum Best-Fit- ist das *Worst-Fit*-Verfahren, bei dem die Teilmenge B_j gewählt wird, welche bei Platzierung des Objekts o_{req} den größten verbleibenden Platz verursacht. Daher wird das $j \in [1, N_B]$ ermittelt, welches $\sum_{o \in B_j} a(o)$ minimiert unter Berücksichtigung der Bedingung (4.4).

Bei der Zuordnung des Objekts zu einer Box betrachten die hier aufgeführten Verfahren im schlechtesten Fall alle derzeit vorhandenen Boxen, so dass für die Zuordnung eine Laufzeit von $\mathcal{O}(N_B)$ benötigt wird. In [8] werden die Verfahren für die zweidimensionale Modulplatzierung in rekonfigurierbaren Architekturen angepasst. Als Boxen dienen hier die im vorherigen Abschnitt beschriebenen freien Rechtecke. Module werden immer in die untere linke Ecke innerhalb eines freien Rechtecks platziert. Im zweidimensionalen Fall muss nicht nur die horizontale Flächenausdehnung eines Hardware-Moduls bei der Zuordnung zu einem freien Rechteck berücksichtigt werden, sondern auch die vertikale Flächenausdehnung. D. h., die im eindimensionalen Fall beschriebene Bedingung (4.4) muss im zweidimensionalen Fall wie folgt erweitert werden.

Definition 4.4 (Menge der geeigneten freien Rechtecke)**.** *Sei E die Menge der gegebenen freien Rechtecke. Dann ist die Menge der für die Platzierung von einem geforderten Hardware-Modul $m_{req} \in M$ geeigneten freien Rechtecke:*

$$E_{free}(m_{req}) = \{(x_{eh}, x_{ev}, a_{eh}, a_{ev}) \mid (x_{eh}, x_{ev}, a_{eh}, a_{ev}) \in E \wedge a_{eh} \geq a_h(m_{req}) \wedge a_{ev} \geq a_v(m_{req})\} \tag{4.5}$$

Die zur Bestimmung von E_{free} benötigte Menge der freien Rechtecke E basiert entweder auf der Menge der maximalen freien Rechtecke, oder der Menge der nicht überlappende freie Rechtecke. Für ein gefordertes Hardware-Modul $m_{req} \in M$ wählt das First-Fit-Verfahren das erste Rechteck $(x_{eh}, x_{ev}, a_{eh}, a_{ev}) \in E_{free}(m_{req})$. Beim Best-Fit-Verfahren wird das Rechteck mit der geringsten Fläche $(a_{eh} \cdot a_{ev})$ gewählt, und entsprechend wird beim Worst-Fit-Verfahren das Rechteck mit der größten Fläche gewählt. Wenn das Hardware-Modul innerhalb des Rechtecks in der unteren linken Ecke platziert wird, entspricht die resultierende Position des Hardware-Moduls m_{req} der Position (x_{eh}, x_{ev}) des gewählten Rechtecks. In [8] wird anhand von Simulationen das First-Fit- mit dem Best-Fit-Verfahren bezüglich der Anzahl der nicht platzierbaren Module einer gegebenen Liste von Modulen verglichen. Die dargestellten Simulationsergebnisse zeigen, dass das Best-Fit-Verfahren in allen betrachteten Simulationen eine geringfügig bessere Platzierung hervorbringt als das First-Fit-Verfahren.

Walder et al. haben in [82] das Best-Fit- und Worst-Fit-Verfahren um einen weiteren Schritt erweitert. In dem *Best-Fit-Exact-Fit*-Verfahren wird zunächst nach dem kleinsten passenden freien Rechteck gesucht, welches genau der Breite oder Höhe des zu platzierenden Hardware-Moduls entspricht. Existiert kein solches Rechteck, wird nach dem normalen Best-Fit-Verfahren vorgegangen. In dem *Worst-Fit-Exact-Fit*-Verfahren wird das gleiche Prinzip angewendet. D. h., es wird zunächst nach dem größten passenden freien Rechteck gesucht, welches genau der Breite oder Höhe des zu platzierenden Hardware-Moduls entspricht. Existiert kein solches Rechteck, wird das normale Worst-Fit-Verfahren verwendet.

Die bisher erwähnten Verfahren zur Positionsbestimmung beziehen sich auf das klassische Packungsproblem. Ein ähnliches Problem ist das zweidimensionale Strip-Packing-Problem (vgl. [52]), bei dem von nur einer Box mit gegebener Breite a_{bin} und unendlicher Höhe ausgegangen wird. Die Objekte der Liste $\sigma_{PP} = (o_1, o_2, \ldots, o_{N_{PP}})$ sollen derart in die gegebene Box gepackt werden, dass die benötigte Höhe nach Platzierung aller Objekte minimal ist. Eine mögliche Heuristik zur Lösung des Strip-Packing Problems ist das von Chazelle [18] präsentierte *Bottom-Left*-Verfahren. Das Bottom-Left-Verfahren platziert Objekte zur Laufzeit an der untersten möglichen linken Position. In [8] wird neben dem Best-Fit- und First-Fit-Verfahren ebenso das Bottom-Left-Verfahren zur Lösung des Platzierungsproblems in rekonfigurierbaren Architekturen beschrieben. Anhand von Simulationen wurde gezeigt, dass die Anzahl der nicht platzierbaren Module bei Verwendung des Bottom-Left-Verfahrens etwa zwischen der Anzahl derer des Best-Fit-Verfahrens und

derer des First-Fit-Verfahrens liegt. Das Bottom-Left-Verfahren ist in den dargestellten Simulationen daher geringfügig besser als das First-Fit-Verfahren.

Handa et al. haben in [37] ein Maß definiert, welches darstellt, wie sehr eine freie Zelle zur Fragmentierung der gesamten rekonfigurierbaren Architektur beiträgt. Auf der Grundlage des in [38] angegebenen Verfahrens zur Bestimmung aller maximalen freien Rechtecke, wird bei dem in [37] beschriebenen Platzierungsverfahren zu einem geforderten Hardware-Modul das passende maximale freie Rechteck ermittelt, welches die geringste Fragmentierung verursachen würde. Innerhalb des gewählten maximalen freien Rechtecks wird das Hardware-Modul in der Ecke platziert, welche ebenso die geringste Fragmentierung hervorrufen würde. Das Verfahren wurde anhand von Simulationen mit dem Best-Fit- und First-Fit-Verfahren verglichen. Simulationsergebnisse zeigen, dass das vorgestellte Verfahren in den betrachteten Simulationen eine geringere Fragmentierung als First-Fit oder Best-Fit hervorruft. Ebenso konnte das in [8] dargestellte Simulationsergebnis, dass das Best-Fit-Verfahren im Vergleich zum First-Fit-Verfahren eine geringere Anzahl nicht platzierbarer Module hervorbringt, bestätigt werden.

Ein alternativer Ansatz zur Positionsbestimmung ohne Bezug zum Packungsproblem ist in Ahmadinia et al. [1] dargestellt. Der Ansatz verfolgt die Platzierung eines Hardware-Moduls unter Berücksichtigung der Routing-Kosten zu bereits platzierten Modulinstanzen. Als Maß der Routing-Kosten dient hierbei die Summe der gewichteten euklidischen Distanzen vom Mittelpunkt des Hardware-Moduls zu den Mittelpunkten der Modulinstanzen. Die Gewichte stellen dabei den Kommunikationsbedarf zwischen dem zu platzierenden Hardware-Modul und der einzelnen platzierten Modulinstanzen dar. Das Verfahren bestimmt zunächst die optimale Position mit den geringsten Routing-Kosten. Wenn die Platzierung des Moduls an der optimale Position nicht möglich ist, wird in einem weiteren Schritt die nächstgelegene freie Position bestimmt. Mittels Simulationen wurde das Verfahren mit dem First-Fit- und dem Best-Fit-Verfahren verglichen. Da weder das First-Fit- noch das Best-Fit-Verfahren die Routing-Kosten berücksichtigt, konnte anhand der Simulationsergebnisse gezeigt werden, dass das vorgestellte Verfahren die geringsten Routing-Kosten verursacht.

Die Mehrzahl der in diesem Abschnitt beschriebenen Verfahren zur Positionsbestimmung eines gegebenen Hardware-Moduls verfolgen das Ziel die Platzierung derart durchzuführen, dass freie zusammenhängende Flächen erhalten werden. Dieses gilt für das First-Fit-, das Best-Fit-, das Worst-Fit- und das Bottom-Left-Verfahren. Das von Handa et al. [37] beschriebenen Verfahren verfolgt das Ziel, die Fragmentierung zu minimieren. Eine hohe Fragmentierung entsteht jedoch genau dann, wenn viele kleine freie zusammenhängende Flächen existieren (vgl. Abschnitt 4.2.1), so dass indirekt ebenso das Ziel der Erhaltung großer freien Flächen verfolgt wird. Einzig der Ansatz von Ahmadinia et al. [1] verfolgt das Ziel die Routing-Kosten zu minimieren.

Die im Abschnitt 4.1.1 beschriebenen Verfahren zur Verwaltung freier Zellen und die in diesem Abschnitt dargestellten Verfahren zur Positionsbestimmung beziehen

sich auf den 2D-Systemansatz. Im Hinblick auf den Spezialfall des 1D-Systemansatzes ergeben sich jedoch hinsichtlich der Verwaltung freier Zellen und der Positionsbestimmung einige Vereinfachungen, die im folgenden Abschnitt im Einzelnen beschrieben werden.

4.1.3 Vereinfachungen im 1D-Systemansatz

Beim 1D-Systemansatz haben alle Hardware-Module eine konstante Höhe, die in der Regel der Höhe der rekonfigurierbaren Architektur entspricht, so dass gilt $a(m) = (a_h, a_v)$ mit $\forall m \in M : a_v(m) = N_{row}$. Durch diese Einschränkung vereinfacht sich die in (4.1) angegebene Menge der möglichen Positionen $X_{pos}(m)$ eines Hardware-Moduls $m \in M$ im 1D-Systemansatz wie folgt. Da $N_{row} - a_v(m) + 1 = 1$, ist die Menge der mögliche Positionen

$$X_{pos}(m) = \big\{(x_h, x_v) \mid 1 \leq x_h \leq N_{col} - a_h(m) + 1 \ \wedge \ x_v = 1\big\}. \tag{4.6}$$

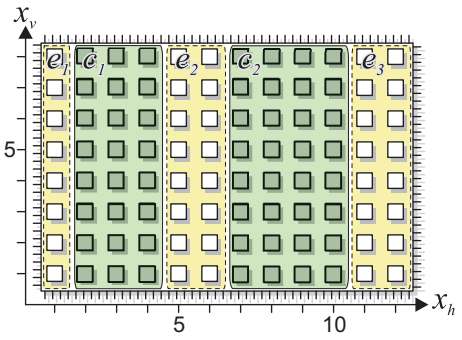
Sowohl die Modulfläche als auch die möglichen Positionen variieren daher nur in horizontaler Richtung. Das Packungsproblem vereinfacht sich daher zu einem eindimensionalen Problem. Wie schon im 2D-Systemansatz dienen rechteckige freie Flächen als Boxen. Jedoch haben alle maximalen freien Rechtecke die gleiche Höhe wie die Hardware-Module, so dass für die Menge der maximalen freien Rechtecke folgende Vereinfachung gilt. Die Menge der freien Rechtecke mit der Höhe $a_{ev} = N_{row}$ ist gemäß (4.2):

$$\begin{aligned} E_{all} = \big\{(x_{eh}, x_{ev}, a_{eh}, a_{ev}) \mid i \in [x_{eh}, x_{eh} + a_{eh}) \ \wedge \ \forall i : b(i,1) = 1 \ \wedge \\ x_{ev} = 1 \ \wedge \ a_{ev} = N_{row}\big\} \end{aligned} \tag{4.7}$$

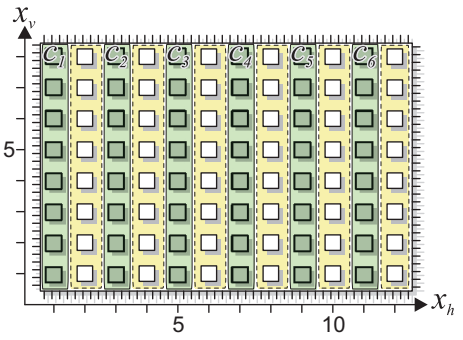
Aus der obigen Menge der freien Rechtecke ergibt sich folgende Vereinfachung der Menge der maximalen freien Rechtecke.

$$\begin{aligned} E_{max} = \big\{(x_{eh}, x_{ev}, a_{eh}, a_{ev}) \mid (x_{eh}, x_{ev}, a_{eh}, a_{ev}) \in E_{all} \ \wedge \\ \forall (\acute{x}_{eh}, \acute{x}_{ev}, \acute{a}_{eh}, \acute{a}_{ev}) \in E_{all} \setminus (x_{eh}, x_{ev}, a_{eh}, a_{ev}) : \\ \acute{x}_{eh} > x_{eh} \ \vee \ \acute{x}_{eh} + \acute{a}_{eh} < x_{eh} + a_{eh}\big\} \end{aligned} \tag{4.8}$$

Im Vergleich zum 2D-Systemansatz gibt es beim 1D-Systemansatz keine Überlappung einzelner maximaler freier Rechtecke. Ein maximales freies Rechteck wird im 1D-Systemansatz immer vom Rand der rekonfigurierbaren Architektur oder dem Rand einer platzierten Modulinstanz komplett umschlossen. Wie in Abbildung 4.4(a) angedeutet, ist die Anzahl der maximalen freien Rechtecke im schlechtesten Fall $\mathcal{O}(|C|)$. Wenn jede Modulinstanz isoliert ist und sowohl linksseitig als auch rechtsseitig von einem maximalen freien Rechteck umschlossen ist, dann ergeben sich genau $|C| + 1$ maximale freie Rechtecke. Die obere Grenze der Anzahl der maximalen freien Rechtecke ist in Abbildung 4.4(b) dargestellt und ergibt sich genau dann,



(a) Beispiel von zwei isolierten Modulinstanzen und den entsprechenden maximalen freien Rechtecke.



(b) Obere Schranke der maximalen freien Rechtecke.

Abbildung 4.4: Beispiele von Mengen maximaler freier Rechtecke bei gegebenen Mengen momentan platzierter Modulinstanzen im 1D-Systemansatz.

wenn jede Modulinstanzen genau eine Spalte belegt, so dass eine freie Spalte von einer belegten und wiederum von einer freien Spalte gefolgt wird. Das Hinzufügen einer weiteren Modulinstanz würde die Anzahl der maximalen freien Rechtecke erneut verringern, so dass die obere Schranke an maximalen freien Rechtecken im 1D-Systemansatz $\lceil N_{col}/2 \rceil$ ist. Da im 1D-Systemansatz die maximalen freien Rechtecke nicht miteinander überlappen, kann die Anzahl der maximalen freien Rechtecke die Anzahl der freien Spalten nicht überschreiten. Zusätzlich zu der zuvor dargestellten oberen Schranke gilt daher, dass die Anzahl der maximalen freien Rechtecke höchstens $\sum_{i=1}^{N_{col}} b(i,1)$ ist.

Die Anzahl der maximalen freien Rechtecke im 1D-Systemansatz ist damit wesentlich geringer als im 2D-Systemansatz. Ein einfaches Verfahren zur Bestimmung aller maximalen freien Rechtecke ist in Algorithmus 4.1 angegeben. Der Algorithmus bestimmt die Menge der maximalen freien Rechtecke E_{max} für den 1D-Systemansatz anhand der aktuellen Zellbelegung gemäß Definition 3.3. Der Algorithmus benötigt dabei insgesamt N_{col} Iterationen der Hauptschleife (2)-(15) und hat damit eine Laufzeit von $\mathcal{O}(N_{col})$. Die Schritte (4)-(8) werden durchlaufen, wenn die derzeit gewählte Spalte i frei ist. Wenn die Spalte i frei ist, wird zunächst überprüft, ob die vorherige Spalte belegt ist. In diesem Fall stellt die Spalte i den linken Rand eines neuen maximalen freien Rechtecks dar, der ebenso die horizontale Position x_{eh} markiert. Die Breite des maximalen freien Rechtecks wird initialisiert und auf $a_{eh} \leftarrow 1$ gesetzt (Schritt (5)). Mit jeder weiteren freien Spalte wird die Breite des aktuellen maximalen freien Rechtecks um 1 erhöht (Schritt (7)).

Wenn die Spalte i belegt ist oder die letzte Spalte $i = N_{col}$ erreicht ist, werden die Schritte (11)-(13) durchlaufen. Wenn die vorherige Spalte frei ist, dann beschreibt die vorherige Spalte den rechten Rand des aktuellen maximalen freien Rechtecks.

Eingabe: Zellbelegung $b(i,j)$ gemäß Definition 3.3.

Ausgabe: Menge der maximalen freien Rechtecke E_{max} für den 1D-Systemansatz.

(1) $E_{max} \leftarrow \{\}$; $b_{prev} \leftarrow 0$
(2) **for** $i \leftarrow 1,\ldots,N_{col}$
(3) **if** $b(i,1) = 1$
(4) **if** $b_{prev} = 0$
(5) $x_{eh} \leftarrow i$; $a_{eh} \leftarrow 1$; $b_{prev} \leftarrow 1$
(6) **else**
(7) $a_{eh} \leftarrow a_{eh} + 1$
(8) **end if**
(9) **end if**
(10) **if** $b(i,1) = 0$ **or** $i = N_{col}$
(11) **if** $b_{prev} = 1$
(12) $E_{max} \leftarrow E_{max} \cup (x_{eh}, 1, a_{eh}, N_{row})$; $b_{prev} \leftarrow 0$
(13) **end if**
(14) **end if**
(15) **end for**

Algorithmus 4.1: Bestimmung der maximalen freien Flächen im 1D-Systemansatz.

Damit sind die Position und die Fläche eindeutig bestimmt und das maximale freie Rechteck wird der Menge E_{max} hinzugefügt (Schritt (12)).

Wenn man die Algorithmen zur Bestimmung der maximalen freien Rechtecke vom 1D-Systemansatz und vom 2D-Systemansatz miteinander vergleicht, so besteht ein wesentlicher Unterschied in der Komplexität des Suchraums. Der Suchraum für die Platzierung eines Hardware-Moduls ist somit im 1D-Systemansatz wesentlich geringer als im 2D-Systemansatz. Welche Auswirkungen diese Tatsache auf die Platzierung von Modulen zur Laufzeit hat, wird im folgenden Abschnitt anhand von Simulationen untersucht.

4.2 Simulative Analyse

Die in diesem Abschnitt durchgeführten simulativen Analysen homogener rekonfigurierbarer Architekturen basieren auf der in Abschnitt 3.3.1 vorgestellten Simulationsumgebung SARA und verfolgen unter anderem das Ziel der Beantwortung folgender Fragen:

- Erweist sich die höhere Flexibilität der Platzierung von Hardware-Modulen im 2D-Systemansatz als vorteilhaft gegenüber der geringeren Flexibilität der Platzierung von Modulen im 1D-Systemansatz?

- Da Modulplatzierungen nur sequenziell durchgeführt werden können, stellt sich die Frage nach dem Einfluss der Ausführungszeit des Platzierungsalgorithmus auf das Gesamtverhalten des Systems.

- Ebenso wie die Modulplatzierung können Konfigurationsprozesse auch nur sequenziell durchgeführt werden, so dass sich die Frage nach dem Einfluss der Konfigurationszeit der Modulinstanzen auf das Gesamtverhalten des Systems ergibt.

Wie bereits erwähnt, haben sich in dem Bereich der partiell rekonfigurierbaren Hardware bisher noch keine einheitlichen Benchmarks zum Testen von Systemansätzen und der entsprechenden Methoden zur Ressourcenverwaltung, wie z. B. Platzierungsverfahren, etabliert. Um dennoch das Testen von Platzierungsverfahren zu ermöglichen, werden in den entsprechenden Arbeiten (z. B. [8, 72]) zufällig erzeugte Abläufe von Komponentenanfragen verwendet. Im Folgenden wird daher ähnlich vorgegangen, indem eine hypothetische Anwendung erzeugt wird, welche aus einem zufällig erzeugten Ablauf von der in Tabelle 4.1 dargestellten Systemkomponenten besteht. Die Systemkomponenten entsprechen im Wesentlichen den in Kalte et al. [46] verwendeten Hardware-Entwürfen. Die Syntheseergebnisse der dargestellten Hardware-Entwürfe bilden die Grundlage zur Erzeugung der Hardware-Module innerhalb der Simulationsumgebung. D. h., die in den Simulationen verwendeten Hardware-Module basieren auf real existierende Implementierungen.

In den Simulationen werden exemplarisch drei verschieden große FPGAs der Xilinx Virtex-II Serie betrachtet. Das kleinste verwendete FPGA ist das XC2V2000 mit einer Fläche von 48×56 Zellen, gefolgt von dem XC2V4000 mit einer Fläche von 72×80 Zellen und dem XC2V6000 mit einer Fläche von 88×96 Zellen. Im Folgenden gelte die Annahme, dass die verwendeten FPGAs homogen seien. Die in den FPGAs integrierten BlockRAM-Spalten werden dabei zunächst außer Acht gelassen. Dementsprechend lassen sich die Menge der möglichen Positionen der Hardware-Module für den 2D-Systemansatz gemäß (4.1) und die Menge der möglichen Positionen der Hardware-Module für den 1D-Systemansatz anhand (4.6) bestimmen.

In Tabelle 4.1 sind die in den Simulationen verwendeten Systemkomponenten und der Flächenbedarf der aus der Synthese hervorgegangenen Hardware-Module dargestellt. Die Anzahl der verwendeten Ressourcen ist in Form von Slices[1] und der

[1] Bei den in der Simulation verwendeten Xilinx Virtex-II FPGAs besteht ein Slice aus zwei 4-bit/1-bit Lookup-Tables, Carry-Logik, zusätzliche Logikgatter, Multiplexer und zwei Speicherelementen. Eine Logikzelle besteht aus vier Slices. Weitere Details zur Verschaltung der Slices sind in [92] zu finden.

Systemkomponente	Slices	Zellen	(a_h, a_v) im 2D-Systemansatz			(a_h, a_v) im 1D-Systemansatz		
			1×2	1×1	2×1	XC2V2000	XC2V4000	XC2V6000
FIR-Filter	306	77	(7,11)	(9,9)	(11,7)	(2,56)	(1,80)	(1,96)
32-bit Divider	844	211	(11,20)	(15,15)	(20,11)	(4,56)	(3,80)	(3,96)
Digital Controller	1055	264	(12,22)	(16,17)	(22,12)	(5,56)	(4,80)	(3,96)
Rijndael Encr.	2120	530	(17,32)	(25,25)	(32,17)	(11,56)	(8,80)	(7,96)
3D-Graphic Accel.	3778	945	(23,45)	(32,32)	(45,23)	(17,56)	(12,80)	(10,96)
Ethernet Switch	4573	1144	(25,50)	(35,35)	(50,25)	(22,56)	(16,80)	(13,96)
32-bit RISC CPU	5730	1433	(28,56)	(39,39)	(56,28)	(28,56)	(19,80)	(16,96)

Tabelle 4.1: In den Simulationen verwendeten Systemkomponenten und Flächenbedarf der resultierenden Hardware-Module für den 1D- und 2D-Systemansatz.

sich daraus ergebenden Zellen aufgeführt. Die verwendeten Systemkomponenten haben unterschiedliche Größen und reichen von einem FIR-Filter mit 77 Zellen (306 Slices) bis zu einer 32-bit RISC CPU mit 1433 Zellen (5730 Slices). Für den 2D-Systemansatz wurden zu jeder Systemkomponente drei Hardware-Module mit unterschiedlichen Seitenverhältnissen (1×2, 1×1 und 2×1) synthetisiert. Auf diese Weise wird die Anzahl der Platzierungsmöglichkeiten und damit die Flexibilität des 2D-Systemansatzes erhöht. Im Falle einer realen Implementierung hat die Verwendung von drei Modulvarianten jedoch zur Konsequenz, dass sich der für die entsprechenden Konfigurationsdatenströme erforderliche Speicherbedarf ebenfalls um den Faktor 3 erhöht. Die Synthese von Hardware-Modulen im 2D-Systemansatz ist grundsätzlich unabhängig von der Größe des zugrunde liegenden FPGAs. Lediglich der Flächenbedarf des Hardware-Moduls darf die Fläche des FPGAs nicht überschreiten. Daher können für das XC2V2000 FPGA nur zwei Modulvarianten (1×2 und 1×1) der Systemkomponente 32-bit RISC CPU berücksichtigt werden, da das Modul mit dem 2×1 Seitenverhältnis die Fläche des FPGAs (48×56) überschreitet. Alle anderen Systemkomponenten sind in allen drei Varianten auf dem Xilinx XC2V2000 FPGA platzierbar.

Im Gegensatz zum 2D-Systemansatz hängt die Flächenausdehnung der Hardware-Module im 1D-Systemansatz von dem gewählten FPGA ab. Die Höhe der synthetisierten Hardware-Module entspricht der des FPGAs, so dass lediglich die Breite der Module variiert. Somit ergeben sich für die verschiedenen FPGAs für jede Systemkomponente unterschiedliche Hardware-Module, die in Tabelle 4.1 dargestellt sind. Durch die gegebene Höhe sind die Seitenverhältnisse kleiner Module mitunter relativ extrem, wie z. B. das Modul der Systemkomponente FIR-Filter, welches beim XC2V6000 ein Seitenverhältnis von 1×96 hat. Ob ein solch schmales Modul wirklich synthetisierbar ist, hängt im Wesentlichen von der Art der Systemkomponente ab. Kalte et al. konnten in [46] zeigen, dass eine Synthese von solch kleinen Modulen im 1D-Systemansatz in vielen Fällen möglich ist. Die maximalen Signallaufzeiten dieser Module waren nur geringfügig länger als die der besten Implementierung. Die Systemkomponenten, bei denen eine Synthese in einer Spalte nicht möglich war, konnten

FPGA	Fläche (a_h,a_v)			Zellen (größte Komponente)		
	3 Blöcke	4 Blöcke	5 Blöcke	3 Blöcke	4 Blöcke	5 Blöcke
XC2V2000	(16,56)	(12,56)	(9,56)	896	672	504
XC2V4000	(24,80)	(18,80)	(14,80)	1920	1440	1120
XC2V6000	(29,96)	(22,96)	(17,96)	2784	2112	1632

Tabelle 4.2: Blockgrößen der verwendeten FPGAs bei unterschiedlicher Anzahl von Blöcken (3, 4, oder 5).

durch Hinzunahme einer weiteren Spalte synthetisiert werden. Für die im Folgenden durchgeführten Simulationen sei daher angenommen, dass alle Systemkomponenten unter den Vorgaben des 1D-Systemansatzes synthetisierbar seien.

Im Systemansatz mit fester Aufteilung ergeben sich *n* gleich große Blöcke, in denen die Hardware-Module zur Laufzeit platziert werden. Die Flächenausdehnung der Module hängt nicht mehr von den benötigten Zellen ab, sondern ergibt sich in Abhängigkeit der Größe des FPGAs und der Anzahl der Blöcke. Die Anzahl der Zellen eines Blocks beschreibt daher gleichzeitig auch die obere Schranke der Größe einer Systemkomponente. In Tabelle 4.2 ist für jedes der gewählten FPGAs eine Übersicht der Blockgrößen und der größtmöglichen platzierbaren Systemkomponente dargestellt. Vergleicht man die Anzahl der Zellen der größtmöglichen Systemkomponente mit der in Tabelle 4.1 dargestellten Anzahl der Zellen der gegebenen Systemkomponenten, zeigt sich, dass nicht alle Systemkomponenten im Systemansatz mit fester Aufteilung platzierbar sind. Bei Verwendung des XC2V2000 FPGAs sind z. B. die Systemkomponenten *3D-Graphic Accellerator*, *Ethernet Switch* und *32-bit RISC CPU* weder bei 3, bei 4 oder bei 5 Blöcken platzierbar. Das XC2V4000 FPGA gestattet maximal 4 Blöcke, um Hardware-Module aller Systemkomponenten platzieren zu können. Lediglich das XC2V6000 FPGA ermöglicht die Verwendung aller Systemkomponenten, selbst bei 5 Blöcken.

Im Systemansatz mit fester Aufteilung ist die Anzahl der möglichen Positionen der Hardware-Module durch die Anzahl der Blöcke gegeben. Im 1D-Systemansatz und im 2D-Systemansatz hängt die Anzahl der möglichen Positionen eines Hardware-Moduls von dessen Größe und der Größe des FPGAs ab. Tabelle 4.3 zeigt als Beispiel die Anzahl der möglichen Positionen der einzelnen Hardware-Module für das XC2V4000 FPGA. Die Spalte *2D (ges.)* beschreibt die Summe der Anzahl der möglichen Positionen für den 2D-Systemansatz. Wenn man die Anzahl der möglichen Positionen beider Systemansätze miteinander vergleicht, lässt sich feststellen, dass der Suchraum für die Platzierung eines Hardware-Moduls im 2D-Systemansatzes um ein Vielfaches größer ist als der Suchraum für die Platzierung eines Hardware-Moduls im 1D-Systemansatz.

Im folgenden Abschnitt werden Metriken beschrieben, mit denen die verschiedenen Systemansätze und deren Methoden zur Platzierung eines Moduls anhand der durchgeführten Simulationen bewertet werden können. Zum einen werden Metriken genannt, die aus dem Bereich des Speichermanagements adaptiert wurden. Zum an-

Systemkomponente	Anzahl der möglichen Positionen für XC2V4000				
	2D (1×2)	2D (1×1)	2D (2×1)	2D (ges.)	1D
FIR-Filter	4620	4608	4588	13816	72
32-bit Divider	3782	3828	3710	11320	70
Digital Controller	3599	3648	3519	10766	69
Rijndael Encr.	2744	2688	2624	8056	65
3D-Graphic Accel.	1800	2009	1624	5433	61
Ethernet Switch	1488	1748	1288	4524	57
32-bit RISC CPU	1125	1428	901	3454	54

Tabelle 4.3: Beispiel für die Anzahl der möglichen Positionen der Hardware-Module für den 1D- und 2D-Systemansatz bei Verwendung eines Xilinx XC2V4000 FPGAs.

deren werden neue Metriken definiert, welche speziell für partiell rekonfigurierbare Architekturen geeignet sind.

4.2.1 Metriken

Wie schon in Abschnitt 3.2 erwähnt, bestehen einige Gemeinsamkeiten zwischen dem Speichermanagement in Zusammenhang mit Betriebssystemen und der Platzierung von Hardware-Modulen in dynamisch rekonfigurierbaren Systemen. Unter anderem lassen sich die im Speichermanagement verwendeten Metriken, wie z. B. die Fragmentierung, auf partiell dynamisch rekonfigurierbare Hardware übertragen.

Wie von Randell in [61] beschrieben, lässt sich die Fragmentierung in *externe Fragmentierung* und *interne Fragmentierung* aufteilen. Randell beschreibt die externe Fragmentierung als den Verlust an Speichermenge, hervorgerufen durch das Zerteilen des Speichers in eine Vielzahl von getrennten Speicherblöcken. Die interne Fragmentierung ist der Verlust an Speichermenge, hervorgerufen durch das Aufrunden der geforderten Speichermenge bei der Zuweisung zu einem Speicherblock.

Im Zusammenhang mit partiell rekonfigurierbaren Architekturen gibt es verschiedene Ansätze zur Definition der externen Fragmentierung. In [83] bestimmen Wigley et al. die externe Fragmentierung anhand der Menge der maximalen freien Rechtecke. Zu jedem Rechteck wird das größtmögliche freie zusammenhängende Quadrat innerhalb des Rechtecks ermittelt. Der Mittelwert der Fläche der resultierenden freien Quadrate dient als charakteristische Größe für die externe Fragmentierung. Eine ähnliche Herangehensweise wird in Walder et al. [81] dargestellt, in dem die externe Fragmentierung anhand des Mittelwerts der Flächen der maximalen freien Rechtecke bestimmt wird. Ejnioui und DeMara berechnen in [29] die externe Fragmentierung anhand des Produkts der relativen Flächen der maximalen freien Rechtecke.

Ein alternativer Ansatz zur Berechnung der externen Fragmentierung ist in Handa et al. [37] dargestellt. Der Ansatz ermittelt anhand der freien und belegten Zellen, wie sehr jede einzelne freie Zelle zur gesamten Fragmentierung beiträgt. Auf diese Weise

lassen sich Bereiche im FPGA bestimmen, die im hohen Maße zur Fragmentierung beitragen. Diese Bereiche werden bei der Platzierung bevorzugt berücksichtigt, um die Fragmentierung zu reduzieren. Die Fragmentierung wird in diesem Ansatz daher nicht bezüglich der gesamten Architektur, sondern bezüglich jeder einzelnen Zelle bestimmt.

Ein für die Platzierung entscheidender Parameter ist das größte maximale freie Rechteck, da Hardware-Module mit einer größeren Fläche nicht ohne Weiteres platziert werden können. Aus diesem Grund wurde in [E8] ein weiteres Maß zur Quantifizierung der Fragmentierung anhand des größten maximalen freien Rechtecks eingeführt, welches als *relative Verfügbarkeit* bezeichnet wird und wie folgt definiert ist.

Definition 4.5 (Relative Verfügbarkeit)**.** *Sei* $e_{best} = (x_{eh}, x_{ev}, a_{eh}, a_{ev}) \in E_{max}$ *das größte maximale freie Rechteck der aktuellen Zellbelegung, so dass gilt*

$$\forall (\acute{x}_{eh}, \acute{x}_{ev}, \acute{a}_{eh}, \acute{a}_{ev}) \in E_{all} \setminus e_{best} : a_{eh} \cdot a_{ev} \geq \acute{a}_{eh} \cdot \acute{a}_{ev} \; .$$

Die relative Verfügbarkeit α *ist definiert als das Verhältnis der Anzahl der Zellen des größten maximalen freien Rechtecks zur Anzahl aller freien Zellen. Wenn die Menge der momentan platzierten Modulinstanzen* $C \neq \{\}$ *ist, ergibt sich*

$$\alpha(C) = \frac{a_{eh} \cdot a_{ev}}{\sum\limits_{i,j} b(i,j)} \; , \quad i \in [1, N_{col}] \; , \quad j \in [1, N_{row}] \; . \tag{4.9}$$

Bei leerer Menge der momentan platzierten Modulinstanzen $C = \{\}$ *gilt* $\alpha(C) = 1$.

Wenn die relative Verfügbarkeit den Wert $\alpha(C) = 1$ annimmt, bedeutet das, dass alle freien Zellen in einem zusammenhängenden Rechteck vorhanden sind. Dieser Fall ist optimal für die Platzierung, denn es gibt nur ein maximales freies Rechteck und alle freien Zellen können für die kommende Platzierung genutzt werden. Der geringste Wert der relativen Verfügbarkeit $\alpha(C) = 1/\lceil 0{,}5 \cdot N_{col} \cdot N_{row} \rceil$ ist genau dann erreicht, wenn sich belegte und freie Zellen wie in einem Schachbrettmuster abwechseln. In diesem Fall ist das größte platzierbare Hardware-Modul genau eine Zelle.

Im Zusammenhang mit partiell rekonfigurierbarer Hardware beschreibt die interne Fragmentierung den Verlust an freien Zellen, hervorgerufen durch die ungenutzten Zellen innerhalb der derzeit platzierten Modulinstanzen. Bezüglich eines Hardware-Moduls ist die interne Fragmentierung daher der Anteil der ungenutzten Zellen im Verhältnis zum Flächenbedarf des Moduls. In homogenen rekonfigurierbaren Architekturen ist die Anzahl der verschiedenen Zelltypen auf $N_{cell} = 1$ begrenzt, so dass der Ressourcenvektor r nur ein Element enthält, d. h., $r = (r_{CLB}) \in \mathbb{N}$. Im Hinblick auf die gesamte rekonfigurierbare Architektur ist die interne Fragmentierung wie folgt definiert.

Definition 4.6 (Interne Fragmentierung)**.** *Sei $r_{CLB}(m)$ die Anzahl der Zellen, die das Hardware-Modul $m \in M$ verwendet. Die interne Fragmentierung ϕ_{int} ist wie folgt definiert:*

$$\phi_{int}(C) = \begin{cases} 1 - \dfrac{\sum\limits_{c \in C} r_{CLB}(m(c))}{\sum\limits_{c \in C} a_h(m(c)) \cdot a_v(m(c))} & \text{wenn } C \neq \{\} \text{ ,} \\ 0 & \text{wenn } C = \{\} \text{ .} \end{cases} \tag{4.10}$$

Wenn die interne Fragmentierung den Wert $\phi_{int}(C) = 0$ annimmt, existieren keine ungenutzten Zellen innerhalb der Flächen der momentan platzierten Modulinstanzen. Der theoretische Maximalwert der internen Fragmentierung ergibt sich genau dann, wenn eine Modulinstanz platziert ist, die die komplette Fläche der Architektur belegt, wobei nur eine Zelle tatsächlich genutzt wird, d. h., $\phi_{int}(C) = 1 - 1/(N_{col} \cdot N_{row}) \approx 1$.

Neben dem Maß der Fragmentierung lässt sich das im Zusammenhang mit Speichermanagement verwendete Maß der *Ressourcenauslastung* ebenso auf den Bereich der dynamisch rekonfigurierbaren Architekturen übertragen. Die im Folgenden definierte Ressourcenauslastung bezieht sich dabei nur auf die tatsächlich genutzten aktiven Zellen aller im Zustand EXE befindlicher Modulinstanzen.

Definition 4.7 (Ressourcenauslastung)**.** *Die Ressourcenauslastung beschreibt das Verhältnis der Anzahl aktiver Zellen zur Anzahl aller auf der Architektur vorhandenen Zellen. Sei $C_{EXE} = \{c \mid c \in C \wedge s(c) = EXE\}$,$C_{EXE} \subseteq C$ die Menge der im Zustand EXE befindlichen Modulinstanzen. Dann ergibt sich für die Ressourcenauslastung:*

$$\upsilon(C) = \frac{\sum\limits_{c \in C_{EXE}} r_{CLB}(m(c))}{N_{col} \cdot N_{row}} \tag{4.11}$$

Wenn die Ressourcenauslastung $\upsilon(C) = 1$ ist, dann sind alle Zellen der Architektur aktiv, was ebenfalls bedeutet, dass die interne Fragmentierung $\phi_{int}(C) = 0$ ist. Wenn die Ressourcenauslastung $\upsilon(C) = 0$ ist, dann sind entweder keine Modulinstanzen vorhanden oder keine der momentan platzierten Modulinstanzen ist aktiv (in dem Zustand EXE).

Wenn Komponentenanfragen bei fehlgeschlagenen Modulplatzierungen abgewiesen werden, so kann die Anzahl der Abweisungen als weitere Metrik betrachtet werden. Nach Beendigung der Simulation kann rückwirkend die Anzahl der erfolglosen Komponentenanfragen bestimmt werden, jedoch sagt die Anzahl der Abweisung nichts über die Größe der abgewiesenen Komponentenanfragen aus. Größere Systemkomponenten sind wegen des größeren Flächenbedarfs und der geringeren Anzahl der möglichen Positionen schwieriger zu platzieren. Eine Metrik, die die Größe der abgewiesenen Systemkomponenten berücksichtigt, ist die *Zellabweisung* ν.

Definition 4.8 (Zellabweisung)**.** *Gegeben sei die in der Simulation verwendete Liste der Komponentenanfragen* $\sigma = (d_1, d_2, \ldots, d_{N_\sigma})$*. Sei* D_{rej} *die Menge der abgewiesenen Komponentenanfragen, die aus der Simulation hervorgegangen ist.* $N_{res}(d)$ *sei die Anzahl der benötigten Zellen der Systemkomponente* $d \in D$*.*

$$\nu(D_{rej}) = \frac{\sum_{d \in D_{rej}} N_{res}(d)}{\sum_{i=1}^{N_\sigma} N_{res}(d_i)} \tag{4.12}$$

Wenn jede Komponentenanfrage zu einer entsprechenden Modulplatzierung führt und die Menge der abgewiesenen Komponentenanfragen $D_{rej} = \{\}$ ist, ist die resultierende Zellabweisung $\nu(D_{rej}) = 0$. Die Metriken Zellabweisung und Ressourcenauslastung sind miteinander verknüpft. Wenn zwei Simulationen, die mit der gleichen Liste der Komponentenanfragen durchgeführt wurden, eine unterschiedliche Ressourcenauslastung hervorgerufen haben, so hat die Simulation mit der geringeren Zellabweisung die größere Ressourcenauslastung. Dieses kann an den im folgenden Abschnitt präsentierten Simulationsergebnissen beobachtet werden.

4.2.2 Vergleich der Systemansätze

Im Folgenden wird der 2D-Systemansatz mit dem 1D-Systemansatz und dem Systemansatz mit fester Aufteilung anhand von Simulationen miteinander verglichen. Erste Ergebnisse des Vergleichs wurden in [E1, E2] präsentiert. Die Vorteile des 2D-Systemansatzes liegen in der Flexibilität im Hinblick auf die Menge der möglichen Positionen der einzelnen Module. Ebenso ist zu erwarten, dass die interne Fragmentierung im 2D-Systemansatz am geringsten ist, da die Hardware-Module günstige Seitenverhältnisse haben. Die Vorteile des 1D-Systemansatzes liegen in der Realisierbarkeit des Ansatzes mit heutigen Technologien. Es ist zu erwarten, das die interne Fragmentierung der Hardware-Module höher ist, im direkten Vergleich zum 2D-Systemansatz, da die Seitenverhältnisse der Module weniger günstig sind. Die Vorteile des Systemansatzes mit fester Aufteilung liegen in der einfachen Platzierung, da nur ein freier Block gefunden werden muss und Einflüsse durch externe Fragmentierung nicht vorhanden sind. Im Hinblick auf die interne Fragmentierung ist zu erwarten, dass der Systemansatz mit fester Aufteilung den höchsten Wert liefern wird.

In den folgenden Simulationen werden die Einflüsse der Platzierungszeiten und der Zeiten der Konfigurations- bzw. Löschvorgänge zunächst außer Acht gelassen. D. h., nach erfolgreicher Modulplatzierung wechselt die entstandene Modulinstanz direkt in den Zustand EXE. Sobald die Ausführung beendet ist, wird die Modulinstanz von der Menge der momentan platzierten Modulinstanzen entfernt, und die zuvor belegten Zellen werden wieder freigegeben. Die Simulationen beschränken sich damit auf

Anw.-Klasse	N_{sim}			p_{req}			$p_{sel}(d)$	$t_{EXE}(d_i)$
	XC2V2000	XC2V4000	XC2V6000	XC2V2000	XC2V4000	XC2V6000		
A	1,0E+06	0,5E+06	0,25E+06	0,5E-03	1,0E-03	2,0E-03	$\propto 1/N_{res}(d)$	0,25
B	1,0E+06	0,5E+06	0,25E+06	0,5E-03	1,0E-03	2,0E-03	$\propto 1/N_{res}(d)$	$1E\text{-}04 \cdot N_{res}(d_i)$
C	1,0E+06	0,5E+06	0,25E+06	0,5E-03	1,0E-03	2,0E-03	$\propto 1/N_{res}(d)$	$0{,}50 \cdot n_{rnd}$
D	2,0E+06	1,0E+06	0,5E+06	0,25E-03	0,5E-03	1,0E-03	$= 1/\|D\|$	0,125
E	2,0E+06	1,0E+06	0,5E+06	0,25E-03	0,5E-03	1,0E-03	$= 1/\|D\|$	$5E\text{-}05 \cdot N_{res}(d_i)$
F	2,0E+06	1,0E+06	0,5E+06	0,25E-03	0,5E-03	1,0E-03	$= 1/\|D\|$	$0{,}25 \cdot n_{rnd}$

Tabelle 4.4: Parameter der Anwendungsklassen.

die Analyse der Leistungsfähigkeit der Modulplatzierungen und die damit verbundene Ressourcenauslastung.

Das für die Platzierungsablaufplanung verwendete Verfahren ist das First-Come-First-Served-Verfahren (vgl. Abschnitt 3.2). Komponentenanfragen werden nur einmal bearbeitet und bei erfolgloser Modulplatzierung abgewiesen. Auf diese Weise sind die Zeitpunkte der einzelnen Modulplatzierungen für Simulationen, die auf der gleichen Liste der Komponentenanfragen basieren, identisch. Dieses ermöglicht einen Vergleich der Ressourcenauslastung oder der internen Fragmentierung der einzelnen Systemansätze über die Zeit.

In den Simulationen werden verschiedene Anwendungsklassen betrachtet. Eine Anwendungsklasse beschreibt die unterschiedlichen Parameter einer virtuellen Anwendung, die zur Erzeugung der zufälligen Listen von Komponentenanfragen herangezogen werden. Eine Anwendungsklasse ergibt sich durch Festlegung der Parameter *Simulationslänge* N_{sim}, *Anfragewahrscheinlichkeit* p_{req} einer Komponentenanfrage, *Auswahlwahrscheinlichkeit* $p_{sel}(d)$ einer Systemkomponente $d \in D$ und *Ausführungszeit* $t_{EXE}(d_i)$ der Systemkomponente d_i in der erzeugten Liste von Komponentenanfragen. Die einzelnen Parameter lassen sich auf vielfältige Weise miteinander kombinieren, so dass die simulative Analyse unter Berücksichtigung aller möglichen Parametervariationen zu umfangreich wäre. Aus diesem Grund beschränkt sich die in diesem Abschnitt betrachtete simulative Analyse auf eine exemplarische Auswahl von sechs Anwendungsklassen, die in Tabelle 4.4 dargestellt sind[2].

In den Anwendungsklassen A, B und C werden bei der Erzeugung der Liste der Komponentenanfragen die Auswahlwahrscheinlichkeiten der Systemkomponenten entsprechend (3.10) verwendet, so dass die Auswahlwahrscheinlichkeit einer Systemkomponente mit zunehmender Anzahl an benötigten Zellen sinkt. In den Anwendungsklassen D, E und F wird eine Gleichverteilung der Auswahlwahrscheinlichkeiten angenommen ($p_{sel} = 1/|D|$).

[2]Zusätzliche simulative Analysen mit weiteren Anwendungsklassen (z. B. $p_{sel}(d) \propto N_{res}(d)$ und $t_{EXE}(d_i) \propto 1/N_{res}(d_i)$) wurden außerhalb dieser Arbeit durchgeführt. Die Ergebnisse unterscheiden sich jedoch nicht grundlegend von denen der hier betrachteten Anwendungsklassen, so dass auf eine Diskussion der Ergebnisse im Rahmen dieser Arbeit nicht eingegangen wird.

Wegen der kleineren Fläche können beim XC2V2000 nicht so viele Modulinstanzen parallel ausgeführt werden wie beim XC2V6000. Dementsprechend variieren die Anfragewahrscheinlichkeiten p_{req} in Abhängigkeit zu der Größe des verwendeten FPGAs. Damit die Anzahl der Komponentenanfragen in jeder Simulation gleich bleibt, ist N_{sim} derart gewählt, dass in einer Simulation immer genau 500 Komponentenanfragen gestellt werden.

Die Ausführungszeiten $t_{EXE}(d_i)$ der Systemkomponenten in den Anwendungsklassen A und D sind konstant. In den Anwendungsklassen B und E hängen die Ausführungszeiten $t_{EXE}(d_i)$ der Systemkomponenten von der Anzahl der benötigten Zellen ab und werden entsprechend (3.11) spezifiziert. In den Anwendungsklassen C und F werden die Ausführungszeiten $t_{EXE}(d_i)$ der Systemkomponenten zufällig erzeugt.

Die hier betrachteten Anwendungsklassen decken einen Großteil möglicher Anwendungsszenarien ab. Dennoch gibt es neben den hier betrachteten Anwendungsklassen noch weitere Kombinationen der einzelnen Parameter, wie z. B. die Annahme, dass die Ausführungszeiten mit der Größe der Systemkomponente abnehmen. Die im Folgenden dargestellten Simulationsergebnisse der Anwendungsklassen A-F ergaben jedoch keine signifikant unterschiedlichen Ergebnisse im direkten Vergleich der einzelnen Systemansätze untereinander, so dass die Betrachtung weiterer Anwendungsklassen außer Acht gelassen wurde.

Für jedes FPGA wurden 20 Simulationen in jeder Anwendungsklasse durchgeführt. Sowohl im 1D-Systemansatz als auch im 2D-Systemansatz diente der Best-Fit-Algorithmus als Platzierungsverfahren. Die Systemansätze mit fester Aufteilung benötigen kein gesondertes Platzierungsverfahren, sondern die Module werden einfach in einen der freien Blöcke platziert.

In Tabelle 4.5 ist die mittlere Ressourcenauslastung aller Simulationen der einzelnen Anwendungsklassen dargestellt. In allen durchgeführten Simulationen war die Ressourcenauslastung vom 1D-Systemansatz am größten. Der Mittelwert bewegt sich zwischen $27,93 \pm 0,63\%$ für die Anwendungsklasse D beim XC2V2000 und $58,51 \pm 2,23\%$ für die Anwendungsklasse A beim XC2V6000. Mitunter deutlich geringer fällt die mittlere Ressourcenauslastung des 2D-Systemansatzes aus, die sich zwischen $24,76 \pm 0,46\%$ für die Anwendungsklasse D beim XC2V2000 und $51,42 \pm 1,58\%$ für die Anwendungsklasse A beim XC2V6000 bewegt. Im Systemansatz mit fester Aufteilung führen die geringe Flexibilität und die geringe Anzahl gleichzeitig platzierbarer Hardware-Module zu der entsprechend geringen mittleren Ressourcenauslastung. Als Folge dessen ist die mittlere Zellabweisung entsprechend groß.

Vergleicht man die in Tabelle 4.6 dargestellte mittlere Zellabweisung des 1D-Systemansatzes mit der des 2D-Systemansatzes, so lässt sich auch hier feststellen, dass der 1D-Systemansatz die Hardware-Module der geforderten Komponentenanfragen besser platzieren konnte. Der Mittelwert der Zellabweisung im 1D-Systemansatz bewegt sich zwischen $5,33 \pm 1,46\%$ für die Anwendungsklasse D beim XC2V6000 und $30,06 \pm 1,63\%$ für die Anwendungsklasse E beim XC2V2000.

FPGA	System-ansatz	Mittlere Ressourcenauslastung [%]					
		A	B	C	D	E	F
XC2V2000	2D	37,38	32,71	36,27	24,76	32,58	24,84
	1D	**42,76**	**38,37**	**41,10**	**27,93**	**37,15**	**28,08**
	5 Blöcke	17,25	7,40	17,28	4,13	1,48	4,12
	4 Blöcke	18,23	13,01	18,01	8,26	4,75	8,13
	3 Blöcke	14,13	11,85	13,84	7,95	4,72	7,84
XC2V4000	2D	42,18	39,07	41,19	27,63	35,87	27,40
	1D	**48,38**	**46,71**	**45,74**	**31,15**	**42,99**	**30,49**
	5 Blöcke	14,31	20,53	14,50	13,34	15,08	13,35
	4 Blöcke	13,23	25,04	13,52	18,01	26,34	18,28
	3 Blöcke	11,40	24,86	11,00	20,61	32,97	20,83
XC2V6000	2D	51,42	49,82	50,29	37,85	47,46	38,03
	1D	**58,51**	**57,68**	**54,83**	**41,94**	**55,74**	**41,99**
	5 Blöcke	13,87	32,03	13,98	27,56	43,76	28,23
	4 Blöcke	10,80	26,20	11,17	22,87	36,11	23,19
	3 Blöcke	8,24	19,76	7,81	17,78	27,68	17,97

Tabelle 4.5: Mittlere Ressourcenauslastung aller Simulationen der einzelnen Anwendungsklassen.

FPGA	System-ansatz	Mittlere Zellabweisung [%]					
		A	B	C	D	E	F
XC2V2000	2D	34,32	29,46	35,99	29,97	37,37	30,41
	1D	**24,83**	**22,98**	**26,74**	**20,97**	**30,06**	**21,43**
	5 Blöcke	69,66	57,62	70,01	88,31	88,04	88,29
	4 Blöcke	67,94	47,44	67,97	76,64	77,18	77,01
	3 Blöcke	75,14	52,07	75,08	77,51	77,32	77,87
XC2V4000	2D	22,82	17,75	22,92	16,05	26,22	15,00
	1D	**11,54**	**10,36**	**13,39**	**5,37**	**15,32**	**5,58**
	5 Blöcke	73,85	41,72	72,77	59,46	55,68	58,09
	4 Blöcke	75,84	45,57	74,50	45,28	39,09	42,86
	3 Blöcke	79,19	55,61	78,10	37,44	37,61	35,39
XC2V6000	2D	28,11	20,66	27,21	14,55	27,08	16,51
	1D	**18,25**	**15,06**	**19,92**	**5,33**	**17,75**	**7,57**
	5 Blöcke	80,80	58,86	80,72	37,84	38,45	38,16
	4 Blöcke	85,06	66,63	84,76	48,47	49,06	48,87
	3 Blöcke	88,61	74,69	89,29	59,98	60,99	60,64

Tabelle 4.6: Mittlere Zellabweisung aller Simulationen der einzelnen Anwendungsklassen.

Systemkomponente	Komp.-Anfragen	Abgewiesene Komp.-Anfragen	
		2D	1D
FIR-Filter	71	0	0
32-bit Divider	71	0	0
Digital Controller	76	0	0
Rijndael Encr.	63	0	0
3D-Graphic Accel.	74	7	0
Ethernet Switch	67	12	5
32-bit RISC CPU	78	27	9

Tabelle 4.7: Beispiel für die Verteilung der abgewiesenen Komponentenanfragen für den 1D- und 2D-Systemansatz (Anwendungsklasse D, XC2V4000 FPGA).

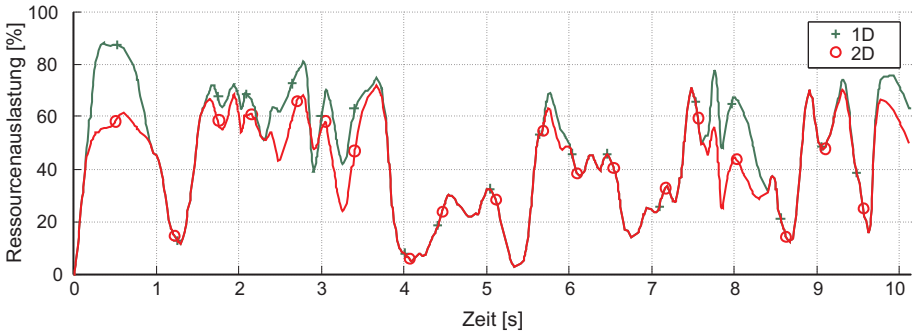


Abbildung 4.5: Beispiel des geglätteten zeitlichen Verlaufs der Ressourcenauslastung für den 1D- und 2D-Systemansatz (Anwendungsklasse B, XC2V4000 FPGA).

Der 2D-Systemansatz hat durchweg eine höhere mittlere Zellabweisung, die sich zwischen $14,55 \pm 1,69\%$ für die Anwendungsklasse D beim XC2V6000 und $37,37 \pm 1,50\%$ für die Anwendungsklasse E beim XC2V2000 bewegt. Ein Beispiel für die Verteilung der abgewiesenen Komponentenanfragen einer Simulation für die Anwendungsklasse B bei Verwendung des XC2V4000 ist in Tabelle 4.7 abgebildet.

Beim 2D-Systemansatz wurden insgesamt 46 Komponentenanfragen abgelehnt, wobei beim 1D-Systemansatz nur 14 Komponentenanfragen abgelehnt wurden. In beiden Simulationen konnten alle Anfragen der kleineren Systemkomponenten bearbeitet werden. Die Tatsache, dass nur Anfragen großer Systemkomponenten abgewiesen wurden, hängt mit den hohen Anforderungen an die Platzierung zusammen, denn bei der Platzierung von großen Hardware-Modulen wird eine dementsprechend große Fläche freier zusammenhängender Zellen benötigt. Trotz der geringeren Flexibilität des 1D-Systemansatzes haben die Simulationen gezeigt, dass der 1D-Systemansatz eine höhere Leistungsfähigkeit im Hinblick auf die Platzierung aufweist. Ein Beispiel für den zeitlichen Verlauf der Ressourcenauslastung beider Ansätze ist in Abbildung 4.5 dargestellt.

FPGA	System-ansatz	Mittlere relative Verfügbarkeit [%]					
		A	B	C	D	E	F
XC2V2000	2D	37,14 ±1,07	57,82 ±1,04	36,63 ±0,82	69,01 ±1,20	77,51 ±1,16	68,30 ±0,74
	1D	**82,64 ±1,15**	**87,21 ±0,99**	**76,33 ±1,72**	**93,00 ±0,42**	**93,71 ±0,74**	**91,74 ±0,65**
XC2V4000	2D	24,84 ±0,75	36,48 ±1,27	24,37 ±1,08	45,32 ±0,61	47,14 ±1,11	44,32 ±0,95
	1D	**78,84 ±1,53**	**77,69 ±1,24**	**67,78 ±2,38**	**86,62 ±0,66**	**82,76 ±1,37**	**82,45 ±1,19**
XC2V6000	2D	19,21 ±0,93	27,24 ±1,06	17,83 ±1,10	30,56 ±0,82	33,67 ±0,94	30,94 ±0,63
	1D	**77,24 ±2,88**	**68,21 ±2,90**	**56,18 ±3,46**	**80,48 ±1,16**	**72,79 ±1,74**	**72,14 ±1,01**

Tabelle 4.8: Mittlere relative Verfügbarkeit aller Simulationen des 1D- und 2D-Systemansatzes.

Bei geringem Ressourcenbedarf verhalten sich beide Systemansätze bezüglich der Ressourcenauslastung gleich, da ausreichend viele freie zusammenhängende Zellen zur Verfügung stehen, um Hardware-Module kommender Komponentenanfragen zu platzieren. Mit steigendem Ressourcenbedarf zeigt sich, dass der 1D-Systemansatz auch bei höherer Ressourcenauslastung noch in der Lage ist, Hardware-Module zu platzieren. Ein wesentlicher Grund dafür wird bei Betrachtung der externen Fragmentierung sichtbar. In Tabelle 4.8 ist die mittlere relative Verfügbarkeit des 1D- und des 2D-Systemansatzes abgebildet. In den Anwendungsklassen A-C, in denen überwiegend kleine Module platziert werden, macht sich der Einfluss der externen Fragmentierung besonders stark bemerkbar. Als Beispiel konnten in der Anwendungsklasse A unter Verwendung des XC2V4000 im 2D-Systemansatz im Mittel aller Simulationen maximal $24,84 \pm 0,75\%$ der freien Zellen für die Platzierung eines Hardware-Moduls genutzt werden. Im Vergleich dazu konnten im 1D-Systemansatz im Mittel aller Simulationen maximal $78,84 \pm 1,53\%$ der freien Zellen für die Platzierung eines Hardware-Moduls genutzt werden. In Abbildung 4.6 ist ein Beispiel des zeitlichen Verlaufs der relativen Verfügbarkeit beider Systemansätze in der Anwendungsklasse A unter Verwendung des XC2V4000 dargestellt.

In der Betrachtung des zeitlichen Verlaufs der relativen Verfügbarkeit des 1D-Systemansatzes zeigt sich, dass in den ersten $250 \cdot 10^{-3}$ *s* der Wert unverändert bei 100% liegt. Der Wert ergibt sich dadurch, dass die ersten Modulinstanzen direkt nebeneinander platziert wurden, so dass die verbleibenden freien Zellen in einer zusammenhängenden Fläche verfügbar sind. Erst mit dem Entfernen einer der zu Beginn platzierten Modulinstanzen ergeben sich mehrere freie zusammenhängende Flächen,

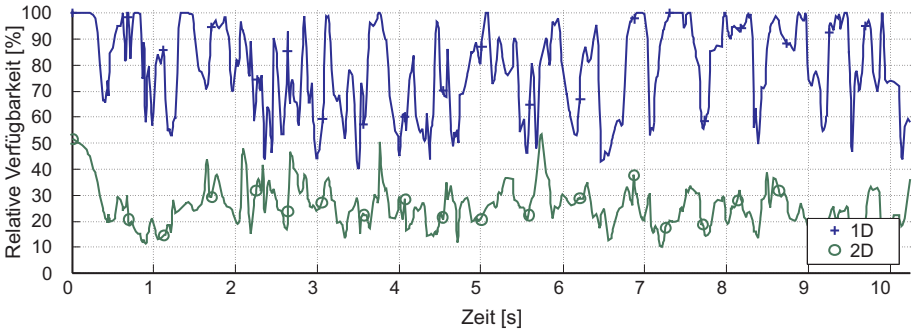
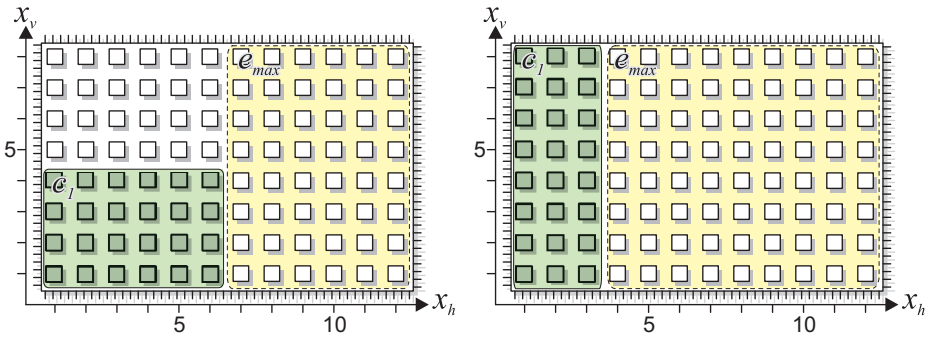


Abbildung 4.6: Beispiel des geglätteten zeitlichen Verlaufs der relativen Verfügbarkeit für den 1D- und 2D-Systemansatz (Anwendungsklasse A, XC2V4000 FPGA).

so dass der Wert der relativen Verfügbarkeit sinkt. Im Verlauf der Simulation wird im Gegensatz zum 2D-Systemansatz dennoch häufiger der bestmögliche Wert von 100% erreicht. In der Betrachtung des 2D-Systemansatzes lässt sich erkennen, dass gleich zu Beginn der Wert der relativen Verfügbarkeit auf etwa 50% sinkt und im Verlauf nur zu wenigen Zeiten erneut den Wert von 50% erreicht. Ein Grund dafür ist im folgenden Beispiel illustriert.

Beispiel 4.3. *Gegeben sei eine homogene rekonfigurierbare Architektur mit* $N_{col} = 12$ *Spalten und* $N_{row} = 8$ *Zeilen. Es sei derzeit keine Modulinstanz platziert, so dass die Menge der momentan platzierten Modulinstanzen* $C = \{\}$ *ist. Gegeben sei die Komponentenanfrage einer Systemkomponente* d *mit einem Ressourcenbedarf von* $N_{res}(d) = \frac{1}{4} \cdot N_{col} \cdot N_{row}$. *Das resultierende Hardware-Modul* m_{2D} *für den 2D-Systemansatz habe eine Flächenausdehnung* $a(m_{2D}) = (6,4)$, *und das entsprechende Hardware-Modul* m_{1D} *für den 1D-Systemansatz habe eine Flächenausdehnung von* $a = (3,8)$. *Abbildung 4.7 zeigt die Belegung der Architektur nach Platzierung der Hardware-Module mit den entsprechenden größten zusammenhängenden rechteckigen freien Flächen. In Abbildung 4.7(a) ist die Belegung des 2D-Systemansatzes dargestellt. Nach Platzierung des Hardware-Moduls steht maximal eine Fläche von* 48 *Zellen für die Platzierung des nächsten Moduls zur Verfügung, und der entsprechende Wert der relativen Verfügbarkeit ist demnach* $\alpha(C) = \frac{48}{72} = \frac{2}{3}$. *Im Vergleich dazu ist in Abbildung 4.7(b) die Belegung des 1D-Systemansatzes dargestellt. Hier stehen für die Platzierung des nächsten Moduls alle freien Zellen (72) zur Verfügung, so dass der Wert der relativen Verfügbarkeit* $\alpha(C) = \frac{72}{72} = 1$ *ist.*

Das obige Beispiel zeigt, dass im 2D-Systemansatz bereits nach einer Modulplatzierung eine externe Fragmentierung vorhanden ist, die die maximale Größe der kommenden Komponentenanfrage stärker einschränkt als im 1D-Systemansatz bei gleichen Voraussetzungen.



(a) 2D-Systemansatz. (b) 1D-Systemansatz.

Abbildung 4.7: Beispiel für die größten zusammenhängenden rechteckigen freien Flächen nach Platzierung eines Hardware-Moduls.

Im Vergleich zur externen Fragmentierung hat die interne Fragmentierung im 1D- und 2D-Systemansatz einen weitaus geringeren Einfluss auf die Leistungsfähigkeit der Modulplatzierung. In Tabelle 4.9 ist die mittlere interne Fragmentierung aller Simulationen des 1D- und 2D-Systemansatzes dargestellt. Die mittlere interne Fragmentierung im 1D-Systemansatz bewegt sich zwischen $2,33 \pm 0,05\%$ für die Anwendungsklasse D beim XC2V2000 und $10,42 \pm 0,39\%$ für die Anwendungsklasse A beim XC2V6000. Wie zu erwarten, ist die interne Fragmentierung beim 1D-Systemansatz bei Verwendung des größten FPGAs (XC2V6000) am höchsten, weil das Seitenverhältnis der Module ebenfalls am größten ist und die Hardware-Module nur in der Breite variieren können. Die mittlere interne Fragmentierung im 2D-Systemansatz verändert sich in allen Simulationen nur geringfügig und bewegt sich zwischen $1,60 \pm 0,29\%$ für die Anwendungsklasse C beim XC2V2000 und $3,55 \pm 0,16\%$ für die Anwendungsklasse E beim XC2V6000. Wie sehr die interne Fragmentierung die Ressourcenauslastung beider Ansätze beeinflusst, ist in einem Beispiel in Abbildung 4.8 dargestellt.

In der Abbildung sind für den 1D-, den 2D-Systemansatz und den Systemansatz mit fester Aufteilung jeweils zwei Kurven und die dazwischen liegende Fläche dargestellt. In der unteren Kurve ist der Anteil der aktiven Zellen dargestellt, was der Ressourcenauslastung entspricht. Die obere Kurve zeigt den Anteil der Zellen, die den momentan platzierten Modulinstanzen zugewiesen sind. Die dazwischenliegende hervorgehobene Fläche stellt den Verlust an Zellen durch interne Fragmentierung dar.

Beim 2D-Systemansatz ist die interne Fragmentierung so gering, dass sie kaum einen Einfluss auf die Leistungsfähigkeit des Systemansatzes hat. Im Gegensatz dazu lässt sich beim 1D-Systemansatz erkennen, dass zeitweise alle Zellen der Architektur denn derzeit platzierten Modulinstanzen zugewiesen sind (z. B. bei $t = 0,5\ s$), so dass der Verlust an freien Zellen und die entsprechend geringere Ressourcenaus-

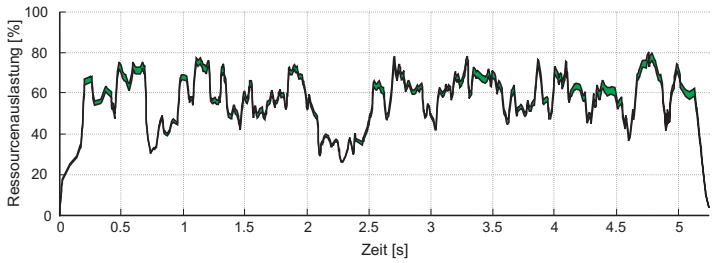
FPGA	Systemansatz	Mittlere interne Fragmentierung [%]					
		A	B	C	D	E	F
XC2V2000	2D	**1,75 ±0,19**	**2,44 ±0,26**	**1,60 ±0,29**	**1,86 ±0,06**	**2,74 ±0,10**	**1,88 ±0,11**
	1D	6,34 ±0,20	3,14 ±0,21	6,20 ±0,23	2,33 ±0,05	2,74 ±0,10	2,40 ±0,07
XC2V4000	2D	**2,63 ±0,18**	**3,17 ±0,24**	**2,51 ±0,14**	**2,27 ±0,06**	**3,16 ±0,10**	**2,25 ±0,11**
	1D	5,52 ±0,26	4,62 ±0,37	5,44 ±0,38	2,98 ±0,11	3,52 ±0,19	2,93 ±0,15
XC2V6000	2D	**2,18 ±0,16**	**3,32 ±0,20**	**2,13 ±0,14**	**2,57 ±0,09**	**3,55 ±0,16**	**2,60 ±0,10**
	1D	10,42 ±0,39	6,53 ±0,27	10,23 ±0,36	4,62 ±0,19	4,64 ±0,10	4,47 ±0,20

Tabelle 4.9: Mittlere interne Fragmentierung aller Simulationen des 1D- und 2D-Systemansatzes.

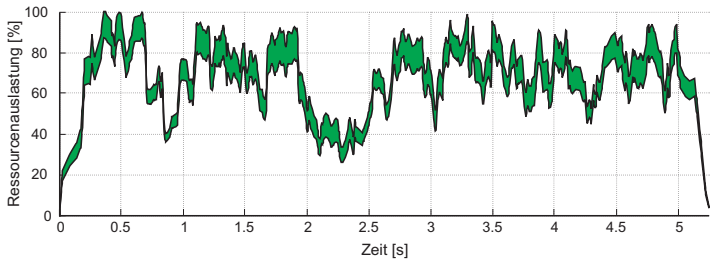
lastung einzig durch interne Fragmentierung verursacht wird. Dennoch zeigt sich im Vergleich der Ressourcenauslastungen des 1D- und des 2D-Systemansatzes in Abbildung 4.8, dass der 1D-Systemansatz bei hohem Ressourcenbedarf leistungsfähiger ist. In Abbildung 4.8 liegt der Mittelwert der Ressourcenauslastung beim 1D-Systemansatz bei $59,75\%$ und der entsprechende Mittelwert beim 2D-Systemansatz bei $53,58\%$.

Bei Betrachtung des Verlaufs der internen Fragmentierung des Systemansatzes mit fester Aufteilung in Abbildung 4.8(c) wird sichtbar, dass die geringere Ressourcenauslastung einzig der internen Fragmentierung zuzuschreiben ist, denn während der gesamten Simulation sind fast durchgängig alle Zellen der Architektur den platzierten Modulinstanzen zugewiesen. Die gelegentlichen kurzzeitigen Abweichungen der oberen Kurve von 100% entstehen, wenn eine längere Zeitspanne vom Freigabezeitpunkt einer Modulinstanz bis zum Platzierungsbeginn der nächsten Modulinstanz gegeben war. Eine externe Fragmentierung ist im Systemansatz mit fester Aufteilung in dem Sinne nicht vorhanden, da die gegebene Fläche der Architektur in gleich große feste Blöcke unterteilt ist.

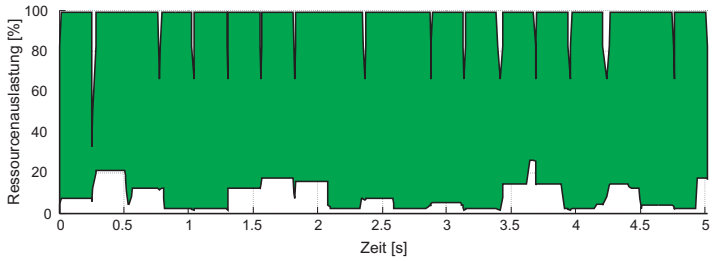
Als Fazit der Simulationen lässt sich Folgendes festhalten: Im 2D-Systemansatz führte die hohe Flexibilität bei der Platzierung von Hardware-Modulen im Verlauf der Simulationen zu einer hohen externen Fragmentierung, die die Ressourcenauslastung hemmte. Die weitaus geringere Flexibilität der Platzierung von Hardware-Modulen im 1D-Systemansatz hat sich nicht negativ auf die Ressourcenauslastung ausgewirkt. In den Simulationen war die interne Fragmentierung zwar größer im Vergleich zum 2D-Systemansatz, jedoch bedingt durch die geringere externe Fragmentierung hat im



(a) 2D-Systemansatz.



(b) 1D-Systemansatz.



(c) Systemansatz mit fester Aufteilung (3 Blöcke).

Abbildung 4.8: Beispiel für den Verlauf der internen Fragmentierung bei Verwendung des XC2V6000 FPGAs.

Vergleich der 1D-Systemansatz die höchste Ressourcenauslastung bewirkt. In allen Simulationen war der Systemansatz mit fester Aufteilung dem 1D- und dem 2D-Systemansatz bezüglich der Ressourcenauslastung weit unterlegen.

In diesem Abschnitt wurden die verschiedenen Systemansätze ohne Berücksichtigung der Platzierungszeit und der Konfigurationszeit untersucht. Im Hinblick auf eine reale Implementierung stellt sich jedoch die Frage, inwieweit die Platzierungszeit des gewählten Algorithmus und die benötigten Konfigurationszeiten der Modulinstanzen die Ressourcenauslastung beeinflussen. Da der 2D-Systemansatz mit heutigen verfügbaren Architekturen nicht ohne Weiteres realisierbar ist und der Systemansatz mit fester Aufteilung zu einer geringen Ressourcenauslastung führt, wird in den kommenden Abschnitten die Analyse des Einflusses der Platzierungszeit und des Einflusses der Konfigurationszeit ausschließlich bezüglich des 1D-Systemansatzes durchgeführt.

4.2.3 Einfluss der Platzierungszeit

Die benötigte Zeit für die Platzierung eines Hardware-Moduls hängt im Wesentlichen von der Implementierung der Ressourcenverwaltung ab. Da die Funktionalität der Ressourcenverwaltung sehr umfangreich ist und dadurch ein hoher Ressourcenbedarf bei einer entsprechenden Hardware-Implementierung entstehen würde, wird im Folgenden angenommen, dass die Ressourcenverwaltung in Software realisiert wird. Als Grundlage dient das in [E9] beschriebene System, welches den eingebetteten Prozessor (IBM PowerPC 405) eines FPGAs der Xilinx Virtex-II Pro Serie [93] verwendet. Der Prozessor verwendet ein Linux-basiertes Betriebssystem mit zusätzlichen Mechanismen zur einfachen Handhabung partieller Rekonfigurierbarkeit von FPGAs. Die in diesem Abschnitt betrachteten Platzierungsalgorithmen sind daher in Software für den IBM PowerPC 405 realisiert worden.

Analysiert wurden die Platzierungszeiten der in Abschnitt 4.1.2 beschriebenen Platzierungsalgorithmen First-Fit und Best-Fit. Bevor auf die Analyse der Laufzeit beider Verfahren eingegangen wird, wird zunächst die für die Laufzeitanalyse entscheidende Umsetzung beider Verfahren im Einzelnen beschrieben.

In der Umsetzung beider Verfahren wird dabei eine erweiterte Darstellung der Zellbelegung $b(i,j)$ verwendet, die als *rechtsseitiger Zellzusammenhang* $\delta_r(i,j)$ bezeichnet wird und wie folgt definiert ist.

Definition 4.9 (Rechtsseitiger Zellzusammenhang)**.** *Gegeben sei die Zellbelegung $b(i,j)$ der momentan platzierten Modulinstanzen. Wenn die Zellbelegung $b(i,j) = 1$ ist, ist der Wert von $\delta_r(i,j)$ die Anzahl der nach rechts verlaufenden aufeinander folgenden freien Zellen ausgehend von der Zelle an Position (i,j). Wenn die Zellbelegung $b(i,j) = 0$ ist, ist der entsprechende rechtsseitige Zellzusammenhang $\delta_r(i,j) < 0$ und der Betrag $|\delta_r(i,j)|$ gibt die Anzahl der nach rechts verlaufenden aufeinander folgenden belegten Zellen ausgehend von der Zelle an Position (i,j) wieder.*

Beispiel 4.4. *Gegeben sei eine homogene rekonfigurierbare Architektur mit $N_{col} = 12$ Spalten im 1D-Systemansatz. Die Zellbelegung $b(i,j)$ der momentan platzierten Modulinstanzen in der ersten Zeile sei*

$$b(i,1) = (\ 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1\ ).$$

Der entsprechende rechtsseitige Zellzusammenhang $\delta_r(i,j)$ für die erste Zeile ist

$$\delta_r(i,1) = (-4\ -3\ -2\ -1 \quad 3 \quad 2 \quad 1\ -1 \quad 2 \quad 1\ -2\ -1 \quad 1\ ).$$

Ein Verfahren zur Bestimmung des rechtsseitigen Zellzusammenhangs im 1D-Systemansatz ist in Algorithmus 4.2 abgebildet. Der Algorithmus ermittelt den rechtsseitigen Zellzusammenhang $\delta_r(i,1)$ für die erste Zeile der Architektur anhand der Zellbelegung $b(i,j)$ der momentan platzierten Modulinstanzen. Dabei benötigt der Algorithmus insgesamt N_{col} Iterationen der Hauptschleife (2)-(15) und hat damit eine

Eingabe: Zellbelegung $b(i,j)$ gemäß Definition 3.3, Anzahl der Spalten N_{col}.

Ausgabe: Rechtsseitiger Zellzusammenhang $\delta_r(i,1)$ gemäß Definition 4.9 für den 1D-Systemansatz.

```
(1)  b_prev ← 0
(2)  v ← −1
(3)  for i ← N_col, ..., 1
(4)     if b(i,1) = 1
(5)        if b_prev = 0
(6)           v ← 1
(7)        end if
(8)        δ_r(i,1) ← v;   v ← v+1;   b_prev ← 1
(9)     else
(10)       if b_prev = 1
(11)          v ← −1
(12)       end if
(13)       δ_r(i,1) ← v;   v ← v−1;   b_prev ← 0
(14)    end if
(15) end for
```

Algorithmus 4.2: Bestimmung des rechtsseitigen Zellzusammenhangs im 1D-Systemansatz.

Laufzeit von $\mathcal{O}(N_{col})$. Die Spalten werden von rechts nach links durchlaufen. Wenn die gewählte Spalte i frei ist, werden die Schritte (5)-(10) durchlaufen. Wenn zusätzlich die vorherige Spalte $i+1$ von einem Hardware-Modul belegt ist, liegt ein Wechsel von einer belegten zu einer freien Spalte vor und der rechtsseitige Zellzusammenhang wird auf $\delta_r(i,1) \leftarrow 1$ gesetzt (Schritt (6)). Mit jeder weiteren freien Spalte wird auch der rechtsseitige Zellzusammenhang der entsprechenden Spalte um 1 erhöht (Schritt (8)).

Wenn die gewählte Spalte i belegt ist, werden die Schritte (10)-(14) durchlaufen. Sofern zusätzlich die vorherige Spalte $i+1$ frei ist, liegt ein Wechsel von einer freien zu einer von einem Hardware-Modul belegten Spalte vor und der rechtsseitige Zellzusammenhang wird auf $\delta_r(i,1) \leftarrow -1$ gesetzt (Schritt (11)). Mit jeder weiteren freien Spalte wird auch der rechtsseitige Zellzusammenhang der entsprechenden Spalte um 1 verringert (Schritte (13)).

Eingabe: Rechtsseitiger Zellzusammenhang $\delta_r(i,1)$ gemäß Definition 4.9 für den 1D-Systemansatz, gefordertes Hardware-Modul m_{req} und die Menge der möglichen Positionen $X_{pos}(m_{req}) = \{(x_{ph}(m_{req},n),x_{pv}(m_{req},n))\}$ mit $n \in [1,|X_{pos}(m_{req})|]$.

Ausgabe: Horizontale Position x_h des geforderten Hardware-Moduls $m_{req} \in M$ für den 1D-Systemansatz.

(1) $i \leftarrow 1$; $x_h \leftarrow 0$;
(2) **while** $i \leq |X_{pos}(m_{req})|$ **and** $x_h = 0$
(3) **if** $\delta_r(x_{ph}(m_{req},i),1) \geq a_h(m_{req})$
(4) $x_h \leftarrow x_{ph}(m_{req},i)$
(5) **end if**
(6) $i \leftarrow i+1$
(7) **end while**

Algorithmus 4.3: Positionsbestimmung im 1D-Systemansatz anhand des First-Fit-Verfahrens.

Das erste hier betrachtete Platzierungsverfahren ist das First-Fit-Verfahren. Eine entsprechende Realisierung des Verfahrens, die den zuvor definierten rechtsseitigen Zellzusammenhang $\delta_r(i,j)$ verwendet, ist in Algorithmus 4.3 dargestellt. In dem Algorithmus sei $(x_{ph}(m_{req},n),x_{pv}(m_{req},n))$, $n \in [1,|X_{pos}(m_{req})|]$ die n-te mögliche Position des geforderten Hardware-Moduls $m_{req} \in M$. Der Algorithmus wählt eine freie Position $(x_h,1)$ aus der Menge der möglichen Positionen $X_{pos}(m_{req})$ des geforderten Hardware-Moduls $m_{req} \in M$ entsprechend der First-Fit-Methode. Wenn keine freie Position gefunden werden kann, wird der Wert $x_h \leftarrow 0$ gesetzt und damit die Platzierung abgewiesen.

Die Hauptschleife (2)-(7) wird dabei so oft durchlaufen, bis eine freie Position gefunden wird. Wenn nach Durchsuchen aller möglichen Positionen keine gültige freie Positionen gefunden wurde, wird die Schleife ebenfalls verlassen. In diesem Fall entspricht die Anzahl der Iterationen der Anzahl der möglichen Positionen des geforderten Hardware-Moduls $|X_{pos}(m_{req})|$. Die obere Grenze der Anzahl möglicher Positionen im 1D-Systemansatz entspricht der Anzahl der Spalten, so dass der Algorithmus daher maximal N_{col} Iterationen der Hauptschleife (2)-(7) benötigt und damit eine Laufzeit von $\mathcal{O}(N_{col})$ hat. Bezüglich des in den vorherigen Simulationen verwendeten größten FPGAs (Xilinx XC2V6000, 88×96 Zellen) werden bei gegebenem rechtsseitigen Zellzusammenhang δ_r maximal 88 Iterationen für die Platzierung eines Moduls anhand des First-Fit-Verfahrens benötigt. Wie im Folgenden gezeigt, wird daher für das First-Fit-Verfahren nur eine geringe Rechenzeit benötigt.

Falls der rechtsseitige Zellzusammenhang $\delta_r(i,j) > 0$ ist, gibt $\delta_r(i,j)$ die Anzahl der nach rechts verlaufenden aufeinander folgenden freien Zellen ausgehend von der Zelle an Position (i,j) wieder. Da im 1D-Systemansatz die Modulinstanzen spaltenweise angeordnet sind, ergibt $\delta_r(i,1)$ die Anzahl der nach rechts verlaufenden aufeinander folgenden freien Spalten. Wenn daher der Betrag des rechtsseitigen Zellzusammenhangs mindestens der horizontalen Flächenausdehnung des geforderten Hardware-Moduls entspricht, also $\delta_r(x,1) \geq a_h(m_{req})$, und $(x,1)$ ein Element der möglichen Positionen des geforderten Hardware-Moduls ist, dann ist $(x,1)$ eine freie Position, die zur Platzierung des Moduls verwendet werden kann. Mithilfe des rechtsseitigen Zellzusammenhangs lässt sich also im 1D-Systemansatz in einem Schritt überprüfen, ob die Zellen einer möglichen Position frei sind. Daher wird im Inneren der Schleife in Schritt (3) überprüft, ob der rechtsseitige Zellzusammenhang der Spalte $x_{ph}(m_{req},i)$ der i-ten möglichen Position $(x_{ph}(m_{req},i),x_{pv}(m_{req},i)) \in X_{pos}(m_{req})$ des geforderten Hardware-Moduls m_{req} größer oder gleich der Breite des Moduls $a_h(m_{req})$ ist. Wenn die Bedingung in Schritt (3) erfüllt ist, wird die horizontale Position $x_{ph}(m_{req},i)$ der gewählten möglichen Position in die Ausgangsvariable x_h geschrieben. Die Änderung von x_h verursacht einen Abbruch der Schleife (2)-(7) und beendet damit den Algorithmus.

Das zweite hier betrachtete Platzierungsverfahren ist das Best-Fit-Verfahren, welches neben dem rechtsseitigen Zellzusammenhang auch den im Folgenden definierten *beidseitigen Zellzusammenhang* verwendet.

Definition 4.10 (Beidseitiger Zellzusammenhang)**.** *Gegeben sei die aktuelle Zellbelegung $b(i,j)$ der momentan platzierten Modulinstanzen. Ebenso wie bei dem rechtsseitigen Zellzusammenhang ist der beidseitige Zellzusammenhang $\delta_b(i,j) > 0$, wenn die Zelle an Position (i,j) frei ist. In diesem Fall gibt $\delta_b(i,j)$ die Anzahl der in beiden Richtungen verlaufenden aufeinander folgenden freien Zellen ausgehend von der Zelle an Position (i,j) wieder. Wenn die Zelle an Position (i,j) belegt ist, ist der beidseitige Zellzusammenhang $\delta_b(i,j) < 0$. In dem Fall ist der Betrag $|\delta_b(i,j)|$ die Anzahl der in beiden Richtungen verlaufenden aufeinander folgenden belegten Zellen ausgehend von der Zelle an Position (i,j).*

Beispiel 4.5. *Gegeben sei eine homogene rekonfigurierbare Architektur mit $N_{col} = 12$ Spalten im 1D-Systemansatz. Die Zellbelegung $b(i,j)$ der momentan platzierten Modulinstanzen in der ersten Zeile sei*

$$b(i,1) = (\ 0\quad 0\quad 0\quad 0\quad 1\quad 1\quad 1\quad 0\quad 1\quad 1\quad 0\quad 0\quad 1\ ).$$

Der entsprechende beidseitige Zellzusammenhang $\delta_b(i,j)$ für die erste Zeile ist

$$\delta_b(i,1) = (-4\ -4\ -4\ -4\quad 3\quad 3\quad 3\ -1\quad 2\quad 2\ -2\ -2\quad 1\ ).$$

Eingabe: Rechtsseitiger Zellzusammenhang $\delta_r(i,1)$ gemäß Definition 4.9 für den 1D-Systemansatz, Anzahl der Spalten N_{col}.

Ausgabe: Beidseitiger Zellzusammenhang $\delta_b(i,1)$ gemäß Definition 4.10 für den 1D-Systemansatz.

(1) $v \leftarrow 0$
(2) **for** $i \leftarrow 1,\ldots,N_{col}$
(3) **if** $(\delta_r(i,1) < 0$ **and** $v > 0)$ **or** $(\delta_r(i,1) > 0$ **and** $v < 0)$
(4) $v \leftarrow \delta_r(i,1)$
(5) **end if**
(6) $\delta_b(i,1) \leftarrow v$
(7) **end for**

Algorithmus 4.4: Bestimmung des beidseitigen Zellzusammenhangs im 1D-Systemansatz.

Ein Verfahren zur Berechnung des beidseitigen Zellzusammenhangs für den 1D-Systemansatz ist in Algorithmus 4.4 dargestellt.

Der Algorithmus ermittelt den beidseitigen Zellzusammenhang $\delta_b(i,1)$ anhand des rechtsseitigen Zellzusammenhangs $\delta_r(i,j)$. Dafür benötigt der Algorithmus insgesamt N_{col} Iterationen der Hauptschleife (2)-(7) und hat damit ebenfalls eine Laufzeit von $\mathcal{O}(N_{col})$. Im Gegensatz zum Algorithmus 4.2 werden die Spalten von links nach rechts durchlaufen. Die Variable v wird genau dann aktualisiert, wenn die Spalte i frei ist und die vorherige Spalte belegt ist, oder die Spalte i belegt ist und die vorherige Spalte frei ist (Schritt (3)). In diesem Fall entspricht der Wert des rechtsseitigen Zellzusammenhangs $\delta_r(i,1)$ dem des beidseitigen Zellzusammenhangs. Die Variable v behält somit ihren Wert, bis erneut die Bedingung in Schritt (3) erfüllt ist. In Schritt (6) wird in jeder Iteration der Wert von v in $\delta_b(i,1)$ übernommen.

Da der Wert des beidseitigen Zellzusammenhangs $\delta_b(i,1)$ bei einer freien Spalte i im 1D-Systemansatz die Anzahl der in beiden Richtungen verlaufenden aufeinander folgenden freien Spalten beschreibt, kann auf diese Weise die Größe des maximalen freien Rechtecks, in der sich die Spalte i befindet, ermittelt werden. Diese Eigenschaft wird in der in Algorithmus 4.5 dargestellten Realisierung des Best-Fit-Verfahrens genutzt. Der Algorithmus bestimmt die Position $(x_h,1) \in X_{pos}(m_{req})$ eines geforderten Hardware-Moduls $m_{req} \in M$ anhand der Menge der möglichen Positionen $X_{pos}(m_{req})$ mithilfe des rechtsseitigen und des zuvor definierten beidseitigen Zellzusammenhangs im 1D-Systemansatz. Wenn das Modul nicht platzierbar ist, wird die horizontale Position auf $x_h = 0$ gesetzt. Der Algorithmus benötigt $|X_{pos}(m_{req})|$ Iterationen der Hauptschleife (2)-(7). Wie zuvor erwähnt, entspricht die obere Grenze der

Eingabe: Rechtsseitiger Zellzusammenhang $\delta_r(i,1)$ und beidseitiger Zellzusammenhang $\delta_b(i,1)$ für den 1D-Systemansatz, gefordertes Hardware-Modul $m_{req} \in M$ und die dazu gehörige Menge der möglichen Positionen $X_{pos}(m_{req}) = \{(x_{ph}(m_{req},n), x_{pv}(m_{req},n))\}$ mit $n \in [1, |X_{pos}(m_{req})|]$.

Ausgabe: Horizontale Position x_h des geforderten Hardware-Moduls $m_{req} \in M$ für den 1D-Systemansatz.

(1) $a_{best} \leftarrow \infty$; $x_h \leftarrow 0$
(2) **for** $i \leftarrow 1, \ldots, |X_{pos}(m_{req})|$
(3) $v \leftarrow x_{ph}(m_{req}, i)$
(4) **if** $\delta_r(v,1) \geq a_h(m_{req})$ **and** $\delta_b(v,1) < a_{best}$
(5) $x_h \leftarrow v$; $a_{best} \leftarrow \delta_b(v,1)$
(6) **end if**
(7) **end for**

Algorithmus 4.5: Positionsbestimmung im 1D-Systemansatz anhand des Best-Fit-Verfahrens.

Anzahl der möglichen Positionen im 1D-Systemansatz der Anzahl der Spalten der Architektur, so dass der Algorithmus damit eine Laufzeit von $\mathcal{O}(N_{col})$ hat.

Der Wert a_{best} beschreibt die Breite des maximalen freien Rechtecks, in dem das geforderte Modul platziert wird. Zu Beginn wird in Schritt (1) $a_{best} \leftarrow \infty$ und $x_h \leftarrow 0$ gesetzt. In jeder Iteration wird in Schritt (3) die horizontale Position der i-ten möglichen Position des Hardware-Moduls m_{req} in die Variable v geschrieben. Wenn hinreichend viele freie Spalten zur Platzierung des Moduls an der gewählten Position zur Verfügung stehen ($\delta_r(v,1) \geq a_h(m_{req})$) und das entsprechende maximale freie Rechteck kleiner ist, als das maximale freie Rechteck, in dem sich die derzeitige beste Position befindet ($\delta_b(v,1) < a_{best}$), dann ist eine bessere Lösung als die bisherige gefunden (Schritt (4)). D. h., die derzeit beste Position wird durch die gewählte Position ersetzt ($x_h \leftarrow v$) und die Breite des dazugehörigen maximalen freien Rechtecks angepasst ($a_{best} \leftarrow \delta_b(v,1)$).

Im Folgenden wird das Verfahren erläutert, welches zur Approximation der benötigten Ausführungszeit der in diesem Abschnitt aufgeführten Algorithmen verwendet wurde. Jeder Algorithmus ist auf dem IBM PowerPC 405 implementiert worden und mithilfe eines zyklenakkuraten Befehlssatzsimulators (ISS) [42] getestet worden. Der Befehlssatzsimulator ermöglicht die Erfassung der Zyklen für jede Anweisung eines Algorithmus.

Um die tatsächliche Ausführungszeit eines Algorithmus in der Simulation in SARA mitberücksichtigen zu können, wurden für jeden Algorithmus die Anzahl der Zyklen

Algorithmus	Zyklen ausserhalb der Hauptschleife		Zyklen in einer Iteration der Hauptschleife			
			feste Anweisungen		bedingte Anweisungen	
	Schritte	Zyklen	Schritte	Zyklen	Schritte	Zyklen
Rechtsseit. Zellzusammenhang (Alg. 4.2)	(1),(2)	25	(3)-(15)	24	(5)-(8)	12
					(6)	3
					(10)-(14)	14
					(11)	5
First-Fit (Alg. 4.3)	(1)	37	(2)-(7)	37	(4)	2
Beidseit. Zellzusammenhang (Alg. 4.4)	(1)	31	(2)-(7)	43	(4)	1
Best-Fit (Alg. 4.5)	(1)	34	(2)-(7)	38	(5)	6

Tabelle 4.10: Anzahl der Zyklen der einzelnen Schritte der Algorithmen.

außerhalb und innerhalb der Hauptschleifen ermittelt. Ebenso wurden zusätzlich benötigte Zyklen, die z. B. bei Erfüllung einer Auswahlbedingung entstehen, ermittelt. Die Ergebnisse der Untersuchung der Anzahl der Zyklen sind in Tabelle 4.10 dargestellt. Jeder Algorithmus ist derart modifiziert worden, dass die entsprechende Anzahl der benötigten Zyklen während der Ausführung des Algorithmus in der Simulationsumgebung SARA mit berechnet wird. Dieses geschieht durch entsprechendes Aufsummieren der in Tabelle 4.10 dargestellten Werte für die Anzahl der benötigten Zyklen. Die Summe aller Zyklen ergibt dann in Abhängigkeit des verwendeten Prozessortakts eine Approximation der Ausführungszeit des Algorithmus.

Die Gesamtausführungszeit einer Modulplatzierung setzt sich dabei aus den Ausführungszeiten der einzelnen Algorithmen zusammen. Für die komplette Platzierung eines Hardware-Moduls anhand des First-Fit-Verfahrens muss zunächst der rechtsseitige Zellzusammenhang bestimmt werden (Algorithmus 4.2). Anschließend kann die Platzierung anhand des First-Fit-Verfahrens (Algorithmus 4.3) durchgeführt werden. Für die Platzierung eines Hardware-Moduls anhand des Best-Fit-Verfahrens muss ebenfalls erst der rechtsseitige Zellzusammenhang bestimmt werden (Algorithmus 4.2). Anschließend wird der beidseitige Zellzusammenhang (Algorithmus 4.4) bestimmt, bevor die Platzierung anhand des Best-Fit-Verfahrens (Algorithmus 4.5) durchgeführt werden kann. Sowohl bei der First-Fit- als auch bei der Best-Fit-Platzierung wird noch zusätzliche Zeit zum Übertragen der für die Platzierung relevanten Daten (z. B. die Zellbelegung) aus dem angeschlossenen Speicher benötigt. Die Aktualisierung der Zellbelegung wird unmittelbar nach der Platzierung durchgeführt[3].

[3]Zur Vorbereitung der nächsten Platzierung, kann der rechtsseitige und beidseitige Zellzusammenhang unmittelbar nach der derzeitigen Platzierung aktualisiert werden. Auf diese Weise lässt sich die benötigte Zeit für die nächste Platzierung verringern.

Um die Güte der Approximation der Ausführungszeit zu überprüfen, wurden die in diesem Abschnitt beschriebenen Algorithmen ebenfalls auf einem eingebetteten IBM PowerPC 405 des Xilinx Virtex-II Pro XC2VP20 implementiert und mit einer Taktfrequenz von 300 MHz getestet. Die dabei verwendete Hardware-Plattform war das Rapid-Prototyping System RAPTOR2000 [45].

Der IBM PowerPC 405 beinhaltet ein 64-bit Register (engl.: Time Base Register (TBR)), welches mit jedem Taktzyklus inkrementiert wird (vgl. [90]). Die Ausführungszeiten der Algorithmen wurden mithilfe des TBR-Registers gemessen, indem der Registerwert zu Beginn und nach Ende einer Platzierung ausgelesen wurde. Die Differenz der beiden Registerwerte ergibt die Anzahl der benötigten Taktzyklen für die Platzierung und der Zeitmessung. Die für das Auslesen des TBR-Registers benötigten Taktzyklen wurden von der Anzahl der gemessenen Taktzyklen abgezogen, um den Einfluss des Zeitmessverfahrens auf die Anzahl der gemessenen Taktzyklen zu eliminieren. Zur Überprüfung der Ausführungszeiten wurden Platzierungen von unterschiedlich großen Hardware-Modulen und verschiedenen Zellbelegungen für einen XC2V4000 FPGA durchgeführt. Als Grundlagen dienten die in Tabelle 4.3 aufgeführten Hardware-Module für den 1D-Systemansatz.

Die durchschnittliche gemessene Ausführungszeit der PowerPC-Implementierung des First-Fit-Verfahrens betrug $8,03 \pm 0,31 \cdot 10^{-6}$ s. Die entsprechende durchschnittliche approximierte Ausführungszeit des First-Fit-Verfahrens ergab $7,92 \pm 0,31 \cdot 10^{-6}$ s, so dass die Abweichung des Mittelwerts der Approximation von dem gemessenen Mittelwert daher bei $107,22 \pm 1,92 \cdot 10^{-9}$ s lag. Für das Best-Fit-Verfahren betrug die gemessene durchschnittliche Ausführungszeit der PowerPC-Implementierung $14,03 \pm 0,31 \cdot 10^{-6}$ s. Die entsprechende durchschnittliche approximierte Ausführungszeit des Best-Fit-Verfahrens ergab $13,95 \pm 0,31 \cdot 10^{-6}$ s. Die Abweichung des Mittelwerts der Approximation von dem gemessenen Mittelwert lag daher bei $76,39 \pm 3,00 \cdot 10^{-9}$ s. Sowohl beim First-Fit- als auch beim Best-Fit-Verfahren entspricht die Approximation der Ausführungszeit bis auf eine geringe Abweichung in der Größenordnung von Nanosekunden der realen Ausführungszeit.

In Tabelle 4.11 sind die Gesamtausführungszeiten des Best-Fit- und des First-Fit-Verfahrens in Abhängigkeit des zugrunde liegenden FPGAs und der einzelnen Anwendungsklassen (vgl. Tabelle 4.4) dargestellt. Vergleicht man die mittleren Ausführungszeiten der beiden Platzierungsverfahren, zeigt sich, dass die mittlere Ausführungszeit des First-Fit-Verfahrens etwa die Hälfte der mittleren Ausführungszeit des Best-Fit-Verfahrens beträgt. Der Unterschied ist im Wesentlichen dadurch zu begründen, dass im Gegensatz zum First-Fit-Verfahren das Best-Fit-Verfahren zusätzlich die Berechnung des beidseitigen Zellzusammenhangs erfordert. Wenn man die mittleren Ausführungszeiten beider Verfahren in Abhängigkeit der Größe des FPGAs betrachtet, so lässt sich eine lineare Abhängigkeit zur Anzahl der Spalten (N_{col}) erkennen.

Um den Einfluss der Platzierungszeit auf die Ressourcenauslastung zu analysieren, wurden die in Abschnitt 4.2.2 beschriebenen Simulationen erneut mit Berücksichti-

Platz.-Verfahren	FPGA	Approximation der Ausführungszeit [s]					
		A	B	C	D	E	F
Best-Fit	XC2V2000	11,6E-06 ± 34E-09	11,8E-06 ± 33E-09	11,6E-06 ± 37E-09	11,5E-06 ± 45E-09	11,3E-06 ± 29E-09	11,5E-06 ± 27E-09
	XC2V4000	17,4E-06 ± 53E-09	17,5E-06 ± 99E-09	17,5E-06 ± 61E-09	17,6E-06 ± 41E-09	17,3E-06 ± 49E-09	17,6E-06 ± 57E-09
	XC2V6000	20,6E-06 ± 71E-09	20,7E-06 ± 100E-09	20,6E-06 ± 65E-09	21,0E-06 ± 62E-09	20,6E-06 ± 65E-09	20,9E-06 ± 41E-09
First-Fit	XC2V2000	6,6E-06 ± 55E-09	6,3E-06 ± 70E-09	6,6E-06 ± 32E-09	6,4E-06 ± 46E-09	6,6E-06 ± 42E-09	6,4E-06 ± 39E-09
	XC2V4000	9,5E-06 ± 73E-09	9,1E-06 ± 120E-09	9,4E-06 ± 70E-09	9,3E-06 ± 42E-09	9,7E-06 ± 75E-09	9,3E-06 ± 74E-09
	XC2V6000	11,8E-06 ± 97E-09	11,0E-06 ± 169E-09	11,7E-06 ± 84E-09	11,4E-06 ± 88E-09	11,8E-06 ± 122E-09	11,5E-06 ± 75E-09

Tabelle 4.11: Mittlere approximierte Gesamtausführungszeit des First-Fit- und Best-Fit-Verfahrens der Simulationen der einzelnen Anwendungsklassen.

gung der Platzierungszeit durchgeführt. Unabhängig von der Größe des FPGAs oder der gewählten Anwendungsklasse wichen die resultierenden Ressourcenauslastungen dabei nicht von den entsprechenden Ressourcenauslastungen der Simulationen ohne Berücksichtigung der Platzierungszeit ab. In den durchgeführten Simulationen ließ sich daher keinerlei Einfluss der Platzierungszeit auf die Ressourcenauslastung feststellen. Ob sich diese Beobachtung auch auf den Einfluss der Konfigurationszeit übertragen lässt, wird im folgenden Abschnitt erläutert.

4.2.4 Einfluss der Konfigurationszeit

Im Gegensatz zur Platzierungszeit ist die Konfigurationszeit einer Modulinstanz nicht von der Laufzeit eines Algorithmus abhängig, sondern von dem Flächenbedarf des entsprechenden Hardware-Moduls und der Geschwindigkeit der Konfigurationsschnittstelle. Wie bereits in Abschnitt 3.3.2 beschrieben, lässt sich die Konfigurationszeit für Xilinx Virtex-II FPGAs anhand (3.21) abschätzen. Im Folgenden sei angenommen, dass die Taktfrequenz der Konfigurationsschnittstelle $f_{SelectMap} = 50\,MHz$ ist. Die Anzahl der zu konfigurierenden Spalten entspricht der horizontalen Flächenausdehnung $a_h(m)$ des Hardware-Moduls $m \in M$. Im Hinblick auf homogene rekonfigurierbare Architekturen wird daher angenommen, dass in den Simulationen keine Block-RAM-Spalten verwendet werden, so dass die Berechnung der Konfigurationszeit eines Hardware-Moduls anhand der in Tabelle 4.12 dargestellten Formeln[4] durchgeführt werden kann.

[4]Die in Tabelle 4.12 dargestellten Abschätzungen sind pessimistisch, da in realen Konfigurationsvorgängen die Konfigurationszeiten durch Verwendung von Kompressionsverfahren (Multiple-Frame-Write) meist geringer sind.

FPGA	Approximation der Konfigurationszeit
XC2V2000	$t_{CFG}(m) = a_h(m) \cdot 12848 / f_{SelectMap}$
XC2V4000	$t_{CFG}(m) = a_h(m) \cdot 18128 / f_{SelectMap}$
XC2V6000	$t_{CFG}(m) = a_h(m) \cdot 21648 / f_{SelectMap}$

Tabelle 4.12: Approximation der Konfigurationszeit von Hardware-Modulen im 1D-Systemansatz.

Um den Einfluss der Konfigurationszeit auf die Ressourcenauslastung zu analysieren, wurden die bereits in Abschnitt 4.2.2 betrachteten Simulationen für den 1D-Systemansatz erneut mit Berücksichtigung der Platzierungszeit und der Konfigurationszeit der Modulinstanzen durchgeführt. Es wurden die gleichen Listen der Komponentenanfragen der einzelnen Anwendungsklassen verwendet, um ein Vergleich zu den Simulationen ohne Berücksichtigung der Platzierungszeit und Konfigurationszeit zu ermöglichen. Als Platzierungsverfahren wurde der im vorherigen Abschnitt beschriebene Best-Fit-Algorithmus verwendet. In Tabelle 4.13 sind die resultierenden mittleren Ressourcenauslastungen aller Simulationen des 1D-Systemansatzes in Abhängigkeit der verschiedenen FPGAs und der unterschiedlichen Taktfrequenzen der Konfigurationsschnittstelle dargestellt. Die längsten Konfigurationszeiten werden bei der Taktfrequenz $f_{SelectMap} = 5\ MHz$ verursacht. Zum Vergleich ist ebenfalls die Ressourcenauslastung bei nicht vorhandener Konfigurationszeit ($f_{SelectMap} = \infty$) dargestellt. Im Gegensatz zur Platzierungszeit beeinflusst die Konfigurationszeit die Ressourcenauslastung. Allgemein ist zu beobachten, dass mit zunehmender Taktfrequenz der Konfigurationsschnittstelle die resultierenden Konfigurationszeiten der Modulinstanzen abnehmen. Bei gleichbleibenden Ausführungszeiten der Modulinstanzen wird die Zeit, in der sich die Modulinstanzen im Zustand CFG und DEL befinden, verkürzt, so dass die resultierende Ressourcenauslastung steigt.

Der Einfluss der Konfigurationszeit macht sich besonders bei großen FPGAs, wie dem XC2V6000, bemerkbar. Eine große FPGA-Fläche ermöglicht das parallele Ausführen vieler Modulinstanzen, die jedoch sequenziell konfiguriert werden müssen. Die Beanspruchung der Konfigurationsschnittstelle ist daher größer im Vergleich zu einem kleinen FPGA, wie dem XC2V2000, der das parallele Ausführen einer geringen Anzahl von Modulinstanzen gestattet. Vergleicht man die mittlere Ressourcenauslastung bei langen Konfigurationszeiten ($f_{SelectMap} = 5\ MHz$) mit der mittleren Ressourcenauslastung bei kurzen Konfigurationszeiten ($f_{SelectMap} = 50\ MHz$), so ist die Verringerung der Ressourcenauslastung des XC2V2000 FPGAs wesentlich geringer, als die Verringerung der Ressourcenauslastung des XC2V6000 FPGAs.

Bezüglich der Größe des zugrunde liegenden FPGAs variieren die Konfigurationszeiten der einzelnen Hardware-Module nur geringfügig. So beträgt beispielsweise die Konfigurationszeit für ein Hardware-Modul m der Systemkomponente Ethernet Switch bei Verwendung des XC2V2000 FPGAs und einem Konfigurationstakt von

FPGA	$f_{SelectMap}$	Mittlere Ressourcenauslastung [%]					
		A	B	C	D	E	F
XC2V2000	5 MHz	35,22	26,52	33,58	19,09	25,19	19,28
	10 MHz	40,72	36,94	39,16	25,04	32,97	24,54
	20 MHz	42,19	39,13	40,39	26,94	36,41	27,38
	50 MHz	43,07	39,24	41,35	28,13	38,29	28,50
	∞	43,40	39,52	41,62	28,95	39,26	29,06
XC2V4000	5 MHz	18,31	8,62	18,23	9,39	10,25	9,14
	10 MHz	36,66	28,85	36,47	19,12	27,11	19,15
	20 MHz	47,65	45,28	45,33	29,75	40,69	28,60
	50 MHz	48,91	47,17	46,50	31,08	43,07	30,32
	∞	49,29	48,07	46,70	31,57	44,17	30,83
XC2V6000	5 MHz	11,68	6,06	12,07	6,18	5,57	6,24
	10 MHz	23,41	12,42	23,81	12,79	14,15	12,80
	20 MHz	47,15	36,87	45,76	26,33	37,46	26,71
	50 MHz	58,99	57,74	55,71	41,72	55,56	41,62
	∞	60,15	59,92	56,49	42,51	57,67	42,77

Tabelle 4.13: Mittlere Ressourcenauslastung aller Simulationen im 1D-Systemansatz bei unterschiedlichen Taktfrequenzen ($f_{SelectMap}$) der Konfigurationsschnittstelle.

$f_{SelectMap} = 50\ MHz$ etwa $t_{CFG}(m) = 5,65 \cdot 10^{-3}\ s$. Das gleiche Modul benötigt bei gleichem Konfigurationstakt und Verwendung des XC2V6000 FPGAs eine Konfigurationszeit von etwa $t_{CFG}(m) = 5,63 \cdot 10^{-3}\ s$.

Die durchschnittlich benötigte Zeit von Platzierungsanfrage bis zum Beginn der aktiven Ausführung der einzelnen Modulinstanzen ist in Tabelle 4.14 dargestellt. Die Zeit setzt sich damit aus der Platzierungszeit, der Zeit im Zustand *PLA* und der Konfigurationszeit der entsprechenden Modulinstanz zusammen. Beim XC2V2000 wird die Verzögerung zwischen Anfrage und Ausführung im Wesentlichen durch die Konfigurationszeit verursacht. Bei niedriger Taktfrequenz der Konfigurationsschnittstelle wächst beim XC2V6000 die Anzahl der zu konfigurierenden bzw. zu löschenden Modulinstanzen so schnell an, dass die Konfigurationsschnittstelle ununterbrochen Modulinstanzen konfiguriert oder löscht. Es entstehen mehr und mehr Konfigurationsanfragen, die nur stark verzögert bearbeitet werden können, so dass ein Verzögerung im Bereich von Sekunden entsteht. Bei hoher Taktfrequenz der Konfigurationsschnittstelle ($f_{SelectMap} = 50\ MHz$) entsteht keine solche Anhäufung der Konfigurations- und Löschanfragen, so dass die Verzögerung im Bereich der benötigten Konfigurationszeit liegt.

In Tabelle 4.15 sind die durchschnittlichen Konfigurationszeiten der Simulationen der einzelnen Anwendungsklassen bei einer Taktfrequenz der Konfigurationsschnittstelle von $f_{SelectMap} = 50\ MHz$ dargestellt. Vergleicht man die Konfigurations-

FPGA	$f_{SelectMap}$	$t_{BOE}(d_i)$-$t_{REQ}(d_i)$ [s]					
		A	B	C	D	E	F
XC2V2000	5 MHz	0,1193	0,1653	0,1118	0,1762	0,1609	0,1799
	10 MHz	0,0241	0,0263	0,0226	0,0514	0,0488	0,0523
	20 MHz	0,0084	0,0089	0,0082	0,0199	0,0193	0,0205
	50 MHz	0,0028	0,0029	0,0027	0,0070	0,0066	0,0070
XC2V4000	5 MHz	0,8177	0,9115	0,7743	0,8752	0,8361	0,8428
	10 MHz	0,2042	0,2222	0,1683	0,2983	0,2416	0,2715
	20 MHz	0,0189	0,0181	0,0170	0,0425	0,0345	0,0441
	50 MHz	0,0039	0,0037	0,0036	0,0087	0,0080	0,0088
XC2V6000	5 MHz	1,2307	1,3056	1,2205	1,4209	1,3984	1,3953
	10 MHz	0,5233	0,6109	0,4975	0,6350	0,5936	0,5984
	20 MHz	0,1107	0,1520	0,0887	0,2110	0,1532	0,1877
	50 MHz	0,0054	0,0056	0,0049	0,0126	0,0111	0,0134

Tabelle 4.14: Durchschnittlich benötigte Zeit von Platzierungsanfrage ($t_{REQ}(d_i)$) bis Ausführungsbeginn ($t_{BOE}(d_i)$) für jede erfolgreich platzierte Modulinstanz.

zeiten mit den in Tabelle 4.11 dargestellten Platzierungszeiten, zeigt sich, dass die Konfigurationszeiten wesentlich größer sind als die Platzierungszeiten. Das kleinste Verhältnis der durchschnittlichen Konfigurationszeit zur entsprechenden mittleren Platzierungszeit für das Best-Fit-Verfahren ist $(2,57 \cdot 10^{-3})/(20,6 \cdot 10^{-6}) = 124,8$ (XC2V6000, Anwendungsklasse C). Das größte Verhältnis der durchschnittlichen Konfigurationszeit zur entsprechenden mittleren Platzierungszeit für das Best-Fit-Verfahren ist $(6,25 \cdot 10^{-3})/(11,5 \cdot 10^{-6}) = 543,5$ (XC2V2000, Anwendungsklasse F). Bezüglich der Xilinx Virtex-FPGAs zeigt sich daher, dass im 1D-Systemansatz die benötigte Zeit für die Platzierung eines Hardware-Moduls im Vergleich zur benötigten Zeit zur Konfiguration des Moduls vernachlässigt werden kann. Im Gegensatz zur Platzierungszeit wirkt sich die Konfigurationszeit entscheidend auf die Ressourcenauslastung aus. Um den Einfluss der Konfigurationszeit auf die Ressourcenauslastung so gering wie möglich zu halten, sollte die Konfigurationszeit der Modulinstanzen so kurz wie möglich sein. Die Xilinx Virtex-II FPGAs erlauben laut [92] eine maximale Taktfrequenz der Konfigurationsschnittstelle (no-handshake SelectMAP) von $f_{SelectMap} = 50\ MHz$. Bei Verwendung dieser Taktfrequenz ist der Einfluss der Konfigurationszeit auf die Ressourcenauslastung für die gegebenen Anwendungsklassen relativ gering.

In den bisherigen Simulationen wurden Komponentenanfragen von nicht platzierbaren Hardware-Modulen einfach abgewiesen. Wenn das Hardware-Modul einer Komponentenanfrage nicht platziert werden kann, obwohl ausreichend freie Spalten zur Verfügung stehen - diese jedoch keine zusammenhängende freie Fläche bilden, dann kann die Platzierung des Hardware-Moduls durch Umplatzieren der bereits konfi-

FPGA	Mittlere Konfigurationszeit [s]					
	A	B	C	D	E	F
XC2V2000	2,49E-03 ±9,93E-05	2,56E-03 ±6,08E-05	2,44E-03 ±8,9E-05	6,23E-03 ±1,9E-04	6,01E-03 ±1,72E-04	6,25E-03 ±1,51E-04
XC2V4000	2,74E-03 ±9,96E-05	2,72E-03 ±9,E-05	2,64E-03 ±8,99E-05	6,64E-03 ±1,01E-04	6,34E-03 ±1,97E-04	6,65E-03 ±1,88E-04
XC2V6000	2,66E-03 ±9,53E-05	2,74E-03 ±5,23E-05	2,57E-03 ±1,08E-04	6,48E-03 ±2,54E-04	6,12E-03 ±1,43E-04	6,47E-03 ±1,59E-04

Tabelle 4.15: Mittlere Konfigurationszeit der einzelnen Anwendungsklassen bei einer Taktfrequenz der Konfigurationsschnittstelle von $f_{SelectMap} = 50\,MHz$.

gurierten Modulinstanzen dennoch ermöglicht werden. Aus diesem Grund wird im folgenden Abschnitt das Konzept der Defragmentierung näher erläutert.

4.3 Defragmentierung

Das Platzieren und Entfernen von unterschiedlich großen Modulinstanzen verursacht mit der Zeit eine externe Fragmentierung, die die Platzierung eines Hardware-Moduls verhindern kann, obwohl hinreichend viele freie Ressourcen zur Verfügung stehen. Um die externe Fragmentierung zu verringern, können die bereits platzierten Modulinstanzen zur Laufzeit so umplatziert werden, dass große Flächen freier zusammenhängender Ressourcen entstehen. Dieses Konzept wird auch als *Defragmentierung* bezeichnet. Die Realisierung einer Defragmentierung erfordert dabei zum einen Mechanismen, die eine Umplatzierung einer im Zustand der Ausführung befindlichen Modulinstanz ermöglichen, und dabei z. B. die internen Registerzustände erhalten. Zum anderen werden Verfahren benötigt, die neue Positionen für die bereits platzierten Modulinstanzen zur Laufzeit bestimmen.

Es gibt verschiedene Ansätze zur Realisierung der Umplatzierung einer Modulinstanz zur Laufzeit. Compton et al. beschreiben in [22] Umplatzierungsmethoden für das speziell für Defragmentierung ausgelegte R/D FPGA. Grundlage hierfür ist eine homogene Zellanordnung, ebenso wie eine homogene Kommunikationsinfrastruktur, so dass eine zellenweise Verschiebung der Modulinstanzen ermöglicht wird. Das R/D FPGA ist spaltenweise konfigurierbar und speziell für den 1D-Systemansatz ausgelegt. Gericota et al. beschreiben in [34] das Konzept der aktiven Vervielfältigung (engl.: Active Replication), bei der die umzuplatzierende Modulinstanz während des Betriebs dupliziert wird unter Berücksichtigung der internen Registerzustände. Nach Beendigung des Kopiervorgangs übernimmt das Duplikat nahtlos die Funktionalität der originalen Modulinstanz, so dass das Original von der Architektur entfernt werden kann. Die Ausführung der Modulinstanz wird daher zu keinem Zeitpunkt unterbrochen. Das Umplatzieren anhand der aktiven Vervielfältigung erfordert jedoch das Vorhandensein einer freien zusammenhängenden Fläche, die mindestens so groß

ist wie die Fläche der zu vervielfältigenden Modulinstanz. Da eine Defragmentierung jedoch meistens bei hoher externer Fragmentierung und mangelnden zusammenhängenden freien Flächen eingeleitet wird, ist das Verfahren nur bedingt einsetzbar.

In Simmler et al. [69] wird eine Methode dargestellt, welche die internen Registerzustände einer Modulinstanz durch das Zurücklesen der Konfigurationsdaten ermittelt. Die Modulinstanz kann nun entfernt und an einer neuen Position platziert werden. Bei der erneuten Konfiguration werden die zuvor extrahierten Registerzustände wieder gesetzt, so dass die Ausführung der Modulinstanz nach Beendigung des Konfigurationsvorgangs in dem vorherigen Zustand fortgesetzt wird. Im Gegensatz zur aktiven Vervielfältigung wird die Ausführung der Modulinstanz während der Umplatzierung unterbrochen. In [E6] wird ein erweitertes Verfahren vorgestellt, welches auf dem gleichen Prinzip beruht, jedoch eine wesentlich kürzere Zeit zur Umplatzierung einer Modulinstanz benötigt. Bei dem Verfahren werden nicht alle Konfigurationsdaten zurückgelesen werden, sondern nur solche, die relevante Zustandsinformationen enthalten.

Die Durchführung einer Defragmentierung kann - wie in Septién et al. [67] dargestellt - unterschiedliche Gründe haben. Wenn die Ressourcenverwaltung zur Laufzeit die externe Fragmentierung ermitteln kann, so kann eine Defragmentierung grundsätzlich bei einem bestimmten Schwellwert der externen Fragmentierung durchgeführt werden. Dieser Ansatz kann als *präventive Defragmentierung* bezeichnet werden. Eine ähnliche Herangehensweise ist das Defragmentieren in fest vorgegebenen Zyklen oder zu Zeitpunkten, in denen eine geringe Anzahl an Modulinstanzen vorhanden ist. Ein Beispiel für einen solchen Ansatz ist in van der Veen et al. [79] für den 2D-Systemansatz beschrieben. Die Defragmentierung wird hier zu periodisch wiederkehrenden Zeitpunkten durchgeführt, in denen das System in einem Ruhezustand ist und die Menge der momentan platzierten Modulinstanzen für eine längere Zeit konstant bleibt. Der beschriebene Defragmentierungsalgorithmus basiert auf der Bestimmung der exakten Lösung eines Optimierungsproblems mit dem Ziel, die bestehenden Modulinstanzen derart umzuplatzieren, dass die resultierende externe Fragmentierung minimal ist.

Ein anderer Ansatz ist die *ereignisgesteuerte Defragmentierung*, bei der die Defragmentierung aufgrund eines bestimmten Ereignisses, wie z. B. eine fehlgeschlagene Modulplatzierung bei hinreichender Anzahl an freien Ressourcen, durchgeführt wird. Den ersten Ansatz eines solchen Verfahrens haben Diessel und ElGindy in [26] beschrieben. Die Arbeit markiert zudem den ersten Ansatz eines Verfahrens zur Defragmentierung von partiell rekonfigurierbaren Architekturen. Das Verfahren ist eine Heuristik zur Umplatzierung von Modulinstanzen zur Laufzeit im 2D-Systemansatz. Die Defragmentierung wird erreicht durch Verschiebung der platzierten Modulinstanzen in einer fest vorgegebenen Richtung. Auf diese Weise wird die externe Fragmentierung verringert und die maximalen freien Rechtecke vergrößert. Um das Verfahren zu testen, wurden Simulationen mit verschiedenen Parametern durchgeführt. Anhand der Simulationen wurde gezeigt, dass das Verfahren insbesondere bei hohem Res-

sourcenbedarf und kleinen Hardware-Modulen die Ressourcenauslastung verbessert. Diessel et al. haben in [25] ein passendes Verfahren zur Ablaufplanung der Umplatzierungen dargestellt. Zusätzlich wird ein Defragmentierungsverfahren beschrieben, welches die neuen Positionen der Modulinstanzen anhand eines genetischen Algorithmus bestimmt.

In [E4, E6] ist die im folgenden Abschnitt beschriebene Methode der *partiellen Kompaktierung* beschrieben, welche eine Defragmentierungsmethode für homogene rekonfigurierbare Architekturen im 1D-Systemansatz darstellt, die auf dem Prinzip der Verschiebung von Modulinstanzen beruht.

4.3.1 Partielle Kompaktierung

Die partielle Kompaktierung ist eine Methode zur Defragmentierung homogener rekonfigurierbarer Architekturen im 1D-Systemansatz. Bei der partiellen Kompaktierung wird die Defragmentierung nicht bezüglich der gesamten Fläche der rekonfigurierbaren Architektur ausgeführt, sondern nur innerhalb eines so genannten *Defragmentierungssegments*. Das Defragmentierungssegment ist die Fläche innerhalb zweier Spalten und wird zur Laufzeit bestimmt. Die Spalte an der horizontalen Position $x_{dl} \in [1, N_{col}-1]$ beschreibt die linke Grenze und die Spalte an der horizontalen Position $x_{dr} \in [x_{dl}+1, N_{col}]$ definiert die rechte Grenze des Defragmentierungssegments.

Sei Menge $C_{defrag}(x_{dl}, x_{dr}) \subseteq C$ die Menge der Modulinstanzen, die sich komplett innerhalb des Defragmentierungssegments befinden.

$$C_{defrag}(x_{dl}, x_{dr}) = \{c \mid c \in C \wedge x_h(c) \geq x_{dl} \wedge x_h(c) + a_h(m(c)) \leq x_{dr} + 1\} \quad (4.13)$$

Bei der partiellen Kompaktierung werden die Modulinstanzen der Menge C_{defrag} in eine vorgegebene Richtung verschoben mit dem Ziel, dass alle Modulinstanzen direkt nebeneinander platziert sind. Ohne Beschränkung der Allgemeinheit sei angenommen, dass die Modulinstanzen nach rechts verschoben werden, so dass nach der Verschiebung alle freien Spalten des Defragmentierungssegments links in einem zusammenhängenden Bereich vorhanden sind. Algorithmus 4.6 beschreibt ein entsprechendes Verfahren zur Berechnung der neuen Positionen der Modulinstanzen innerhalb des Defragmentierungssegments. Die Variable i markiert die aktuell gewählte Spalte innerhalb des Defragmentierungssegments. Die Spalten des Segments werden von rechts nach links durchlaufen, beginnend mit der äußerst rechten Spalte $i \leftarrow x_{dr}$. Die Menge C_{temp} wird mit der Menge der Modulinstanzen im Defragmentierungssegment $C_{defrag}(x_{dl}, x_{dr})$ initialisiert. Die Anzahl der Iterationen der Hauptschleife (2)-(7) entspricht der Anzahl der Modulinstanzen in $C_{defrag}(x_{dl}, x_{dr})$. Innerhalb der Schleife wird zunächst in Schritt (3) die am weitesten rechts gelegene Modulinstanz $c \in C_{temp}$ ausgewählt. Die aktuelle Spalte i wird in Schritt (4) um die Breite der Modulinstanz c verringert. In Schritt (5) wird die neue horizontale Position $\tilde{x}_h(c)$ der

Eingabe: Linke Grenze des Defragmentierungssegments x_{dl}, rechte Grenze des Defragmentierungssegments x_{dr}.

Ausgabe: Kompaktierte horizontale Position $\tilde{x}_h(c)$ der Modulinstanzen innerhalb des Defragmentierungssegments für den 1D-Systemansatz.

(1) $i \leftarrow x_{dr}$; $C_{temp} \leftarrow C_{defrag}(x_{dl}, x_{dr})$
(2) **while** $C_{temp} \neq \{\}$
(3) **select an** $c \in C_{temp}$ **with maximum** $x_h(c)$
(4) $i \leftarrow i - a_h(m(c))$
(5) $\tilde{x}_h(c) \leftarrow i + 1$
(6) $C_{temp} \leftarrow C_{temp} \setminus c$
(7) **end while**

Algorithmus 4.6: Bestimmung von neuen Positionen der Modulinstanzen innerhalb des Defragmentierungssegments anhand partieller Kompaktierung.

Modulinstanz c anhand der aktuellen Spalte bestimmt. Anschließend wird in Schritt (6) die gewählte Modulinstanz aus der Menge der im Defragmentierungssegment befindlichen Modulinstanzen C_{temp} entfernt. Nach Durchlaufen einer Iteration hat sich somit das Defragmentierungssegment um die Breite der gewählten Modulinstanz verkleinert. Die Schritte (3)-(6) werden daher so oft wiederholt, bis keine Modulinstanz mehr in der Menge C_{temp} vorhanden ist.

Die Defragmentierung der kompletten rekonfigurierbaren Architektur (d. h., $x_{dl} = 1$, $x_{dr} = N_{col}$) führt zwar zu einer optimalen Anordnung der freien Spalten mit keiner externen Fragmentierung. Dennoch müssen unter Umständen alle Modulinstanzen umplatziert werden, was einen hohen Rekonfigurationsaufwand und eine dementsprechend hohe Beanspruchung der Konfigurationsschnittstelle fordert. Die Gesamtzeit der Defragmentierung ergibt sich demnach aus der Summe der Umplatzierungszeiten der im Defragmentierungssegment enthaltenen Modulinstanzen.

In [E7] wird eine Methode zur Umplatzierung von Modulinstanzen für den Xilinx Virtex-II FPGA beschrieben, die die internen Zustände der Register der Modulinstanz bei der Umplatzierung speichert. Auf diese Weise lassen sich Modulinstanzen im Betrieb unterbrechen und nach der Umplatzierung an einer anderen Position im vorherigen Zustand wieder fortsetzen. Sei $N_{CLB-COL}$ die Anzahl der CLB-Spalten, die umplatziert werden sollen, und $N_{RAM-COL}$ die Anzahl der BlockRAM-Spalten, die umplatziert werden sollen. Die Taktrate der Konfigurationsschnittstelle sei $f_{SelectMap}$ und $N_{Byte/Frame}$ sei eine FPGA-typenspezifische Konstante. Dann kann die maximal

FPGA	Approximation der Umplatzierungszeit
XC2V2000	$t_{RELOC}(m) = a_h(m) \cdot 28032 / f_{SelectMap}$
XC2V4000	$t_{RELOC}(m) = a_h(m) \cdot 39552 / f_{SelectMap}$
XC2V6000	$t_{RELOC}(m) = a_h(m) \cdot 47232 / f_{SelectMap}$

Tabelle 4.16: Abschätzung der erforderlichen Zeit zur Umplatzierung einer Modulinstanz im 1D-Systemansatz gemäß der in [E7] beschriebenen Methode.

benötigte Zeit zur Umplatzierung einer Modulinstanz im 1D-Systemansatz anhand der in [E7] beschriebenen Methode wie folgt abgeschätzt werden.

$$t_{RELOC} \approx \frac{(48 \cdot N_{CLB-COL} + 237 \cdot N_{RAM-COL}) \cdot N_{Byte/Frame}}{f_{SelectMap}} \tag{4.14}$$

Im Hinblick auf homogene rekonfigurierbare Architekturen sei im Folgenden angenommen, dass keine BlockRAM-Spalten verwendet werden und die Modulinstanzen beliebig umplatziert werden können, so dass die Umplatzierungszeiten für eine Modulinstanz für die bereits betrachteten FPGAs anhand der in Tabelle 4.16 dargestellten Formeln abgeschätzt werden kann.

Die in Tabelle 4.16 dargestellten Approximationen zeigen, dass die Umplatzierungszeit einer Modulinstanz von der Taktfrequenz der Konfigurationsschnittstelle $f_{SelectMap}$ und der Fläche der Modulinstanz $a_h(m(c))$ abhängt. Da die Taktrate der Konfigurationsschnittstelle konstant ist und im Falle des Xilinx Virtex-II maximal $f_{SelectMap} = 50\ MHz$ beträgt, verändert sich die Umplatzierungszeit einzig durch die Fläche der Modulinstanz.

Wenn die Platzierung eines Hardware-Moduls $m_{req} \in M$ im 1D-Systemansatz fehlschlägt, obwohl genügend freie Spalten auf der Architektur vorhanden sind, diese jedoch in kleinen Fragmenten verteilt sind, so kann durch partielle Kompaktierung eine hinreichend große freie zusammenhängende Fläche geschaffen werden, um die Modulplatzierung zu ermöglichen. Grundvoraussetzung ist jedoch das Vorhandensein von genügend freien Spalten, so dass folgende Bedingung erfüllt sein muss:

$$\sum_{i=1}^{N_{col}} b(i,1) \geq a_h(m_{req}) \tag{4.15}$$

In Abhängigkeit des Verfahrens zur Konfigurationsablaufplanung kann anstelle der in (3.1) angegebenen Zellbelegung $b(i,j)$ ebenfalls die in (3.7) definierte Zellbelegung $b_{FCFS}(i,j)$ verwendet werden. Um die Verzögerung der Platzierung des geforderten Moduls zu minimieren, ist das Defragmentierungssegment derart zu wählen, dass mindestens die erforderliche freie zusammenhängende Fläche entsteht

$\big(a_h(m_{req})\big)$. Ähnlich zu (4.15) ergibt sich bei der Wahl des Defragmentierungssegments folgende Bedingung:

$$\sum_{i=x_{dl}}^{x_{dr}} b(i,1) \geq a_h(m_{req}) \tag{4.16}$$

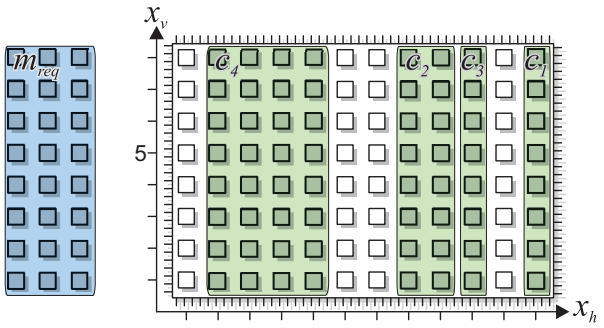
Unter Einhaltung der obigen Bedingung kann das Defragmentierungssegment auf verschiedene Weise festgelegt werden. Die Auswahl der Grenzen des Segments kann dabei anhand der folgenden Kriterien getroffen werden:

Modulverschiebungen: Das Umplatzieren einer Modulinstanz erfordert das Unterbrechen der Ausführung. Um die Anzahl der zu unterbrechenden Modulinstanzen so gering wie möglich zu halten, kann die Auswahl des Defragmentierungssegments als Optimierungsproblem formuliert werden. Ziel der Optimierung ist, die Anzahl der Modulinstanzen in $C_{defrag}(x_{dl}, x_{dr})$ zu minimieren unter Berücksichtigung der Bedingung (4.16).

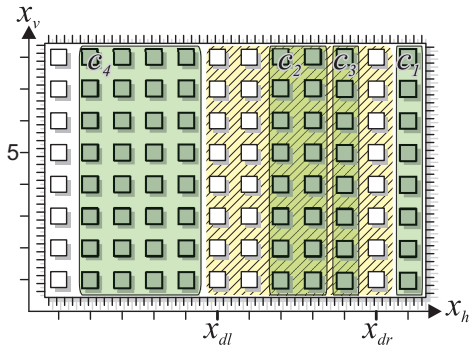
Spaltenverschiebungen: Die Minimierung der Anzahl der zu verschiebenden Modulinstanzen führt nicht unbedingt zu der geringsten Umplatzierungszeit, da die zu verschiebenden Modulinstanzen sehr groß sein können, und daher eine lange Umplatzierungszeit benötigen (vgl. Tabelle 4.16). Um die erforderliche Umplatzierungszeit zu minimieren, muss daher die Anzahl der zu verschiebenden Spalten minimiert werden. Die Auswahl des Defragmentierungssegments kann daher als Optimierungsproblem formuliert werden, mit dem Ziel die Fläche $x_{dr} - x_{dl}$ des Segments zu minimieren unter Einhaltung der Bedingung (4.16).

Kosten: Unabhängig von Aspekten wie der Umplatzierungszeit kann das Defragmentierungssegment ebenfalls im Hinblick auf andere Kosten, wie z. B. Prioritäten oder die zu erwartende verbleibende Ausführungszeit der einzelnen Modulinstanzen, bestimmt werden. Sei $w_{PR}(c) \in [0,1]$ eine der Modulinstanz c zugewiesene Priorität, wobei $w_{PR}(c) = 0$ die niedrigste Priorität und $w_{PR}(c) = 1$ die höchste Priorität markiert. Dann kann die Auswahl des Defragmentierungssegments als Optimierungsproblem formuliert werden, mit dem Ziel, die Summe $\sum_{c \in C_{defrag}(x_{dl}, x_{dr})} w_{PR}(c)$ zu minimieren unter Einhaltung der Bedingung (4.16).

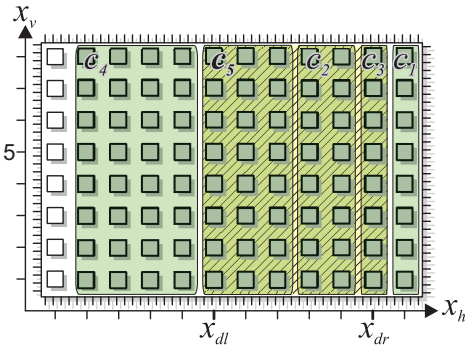
Beispiel 4.6. *Gegeben sei die in Abbildung 4.9(a) dargestellte Belegung einer homogenen rekonfigurierbaren Architektur mit der Menge der momentan platzierten Modulinstanzen $C = \{c_1, c_2, c_3, c_4\}$ im 1D-Systemansatz. Gefordert sei die Platzierung eines Hardware-Moduls $m_{req} \in M$, welches eine horizontale Flächenausdehnung von $a_h(m_{req}) = 3$ besitzt. Die Platzierung von m_{req} ist nicht unmittelbar möglich, da die freien Spalten in kleinen Fragmenten verteilt sind. Die Modulplatzierung ist jedoch nach Defragmentierung möglich, da Bedingung* (4.15) *erfüllt ist.*



(a) Gefordertes Hardware-Modul m_{req} und die derzeit konfigurierten Modulinstanzen C.



(b) Defragmentierungssegment bei minimaler Spaltenverschiebung.



(c) Resultierende Belegung der Architektur nach Defragmentierung.

Abbildung 4.9: Beispiel für die Defragmentierung anhand partieller Kompaktierung.

Die Defragmentierung soll anhand der partiellen Kompaktierung mit dem Ziel der minimalen Spaltenverschiebungen durchgeführt werden. Die zu minimierende Zielfunktion ist daher $x_{dr} - x_{dl}$ *unter Einhaltung der Bedingung* (4.16)*. Das resultierende Defragmentierungssegment ist in Abbildung 4.9(b) dargestellt und befindet sich zwischen den Spalten* $x_{dl} = 6$ *und* $x_{dr} = 11$*. Die zu verschiebenden Modulinstanzen sind* $C_{defrag}(6,11) = \{c_2, c_3\}$*. Die Bestimmung der neuen Positionen anhand Algorithmus 4.6 ergibt, dass Modulinstanz* c_3 *nach Position* $x_h(c_3) = 11$ *und* c_2 *nach Position* $x_h(c_2) = 9$ *verschoben wird. Anschließend kann das geforderte Hardware-Modul* m_{req} *an dem linken Rand des Defragmentierungssegments* x_{dl} *platziert und die entsprechende Modulinstanz* $c_5 = (m_{req}, (6,1), PLA, (5))$ *erzeugt werden (vgl. Abbildung 4.9(c)).*

Welchen Einfluss die partielle Kompaktierung auf die Ressourcenauslastung hat, wird im folgenden Abschnitt anhand von Simulationen analysiert.

4.3.2 Simulative Analyse

Jede Defragmentierung ist mit einem zusätzlichen Konfigurationsaufwand verbunden, der entsteht, wenn bereits platzierte Modulinstanzen umplatziert werden. Anhand der Simulationsergebnisse aus Abschnitt 4.2.4 konnte gezeigt werden, dass die benötigten Konfigurationszeiten der Modulinstanzen die resultierende Ressourcenauslastung erheblich verringern können. Daher stellt sich die Frage, ob der bei der Defragmentierung entstehende zusätzliche Konfigurationsaufwand die Ressourcenauslastung derart verringert, so dass der erwünschte Effekt der Defragmentierung nicht zum Tragen kommt.

Aus diesem Grund werden in diesem Abschnitt verschiedene Simulationen im 1D-Systemansatz mit und ohne Defragmentierung miteinander verglichen. Als Defragmentierungsmethode dient die im vorherigen Abschnitt vorgestellte partielle Kompaktierung. Dabei wird die partielle Kompaktierung sowohl mit der Zielsetzung einer minimalen Anzahl von Spaltenverschiebungen (PK_{col}), als auch mit der Zielsetzung einer minimalen Anzahl von Verschiebungen von Modulinstanzen (PK_{mod}) betrachtet. Zusätzlich wird die Defragmentierung der ganzen Fläche anhand partieller Kompaktierung (PK_{all}) untersucht. Pro Simulation werden 500 Komponentenanfragen gestellt, und als Platzierungsverfahren wird der Best-Fit-Algorithmus (vgl. Algorithmus 4.5) verwendet. Bei fehlgeschlagener Modulplatzierung und Einhaltung der Bedingung (4.15) wird die Defragmentierung entsprechend dem gewählten Verfahren ausgeführt. Fehlgeschlagene Modulplatzierungen durch mangelnde Anzahl freier Spalten werden abgewiesen. Die erzeugten Listen der Komponentenanfragen basieren auf der Anwendungsklasse B (vgl. Tabelle 4.4), d. h., die Ausführungszeiten der Modulinstanzen hängen von der Größe des Hardware-Moduls ab. Die Ausführungszeiten und Anfragewahrscheinlichkeiten sind mit einem Faktor versehen, so dass sich die einzelnen Simulationen durch das Verhältnis der Ausführungszeit zur Konfigurationszeit t_{EXE}/t_{CFG} einer Modulinstanz unterscheiden.

In Tabelle 4.17 ist die mittlere Ressourcenauslastung der unterschiedlichen Simulationen dargestellt. Unabhängig von dem zugrunde liegenden FPGA lässt sich feststellen, dass die Ressourcenauslastung im Allgemeinen mit steigendem Verhältnis von t_{EXE}/t_{CFG} ebenfalls zunimmt, wobei die Unterschiede der Ressourcenauslastung bei großen Verhältnissen, wie $t_{EXE}/t_{CFG} > 30$, nur geringfügig sind. In Abhängigkeit der Größe des FPGAs sinkt die mittlere Ressourcenauslastung deutlich bei Unterschreiten eines bestimmten Verhältnisses t_{EXE}/t_{CFG}. Beim XC2V6000 fällt die mittlere Ressourcenauslastung unterhalb des Verhältnisses $t_{EXE}/t_{CFG} \approx 20$ deutlich ab, wobei bei dem XC2V2000 die mittlere Ressourcenauslastung erst unterhalb des Verhältnisses von $t_{EXE}/t_{CFG} \approx 5$ deutlich abfällt. Grund für diesen Unterschied ist die Tatsache, dass die Anzahl der parallel platzierbaren Modulinstanzen beim XC2V6000 größer ist als beim kleineren XC2V2000. Eine höhere Anzahl parallel platzierter Modulinstanzen erfordert ebenso eine höhere Beanspruchung der Konfigurationsschnittstelle.

FPGA	Defrag.-Alg.	Mittlere Ressourcenauslastung [%]						
		t_{EXE}/t_{CFG} ≈5	t_{EXE}/t_{CFG} ≈10	t_{EXE}/t_{CFG} ≈20	t_{EXE}/t_{CFG} ≈30	t_{EXE}/t_{CFG} ≈40	t_{EXE}/t_{CFG} ≈50	t_{EXE}/t_{CFG} ≈100
XC2V2000	(keine)	34,09	38,25	39,20	39,23	39,33	39,38	39,38
	PK_{col}	39,17	41,87	42,66	42,94	42,92	42,96	43,17
	PK_{mod}	36,51	41,45	42,41	43,02	42,98	**43,08**	43,14
	PK_{all}	**39,64**	**41,89**	**42,74**	**43,08**	**42,99**	43,02	**43,20**
XC2V4000	(keine)	14,54	41,76	46,66	46,98	47,03	47,43	48,13
	PK_{col}	28,22	45,38	49,35	49,71	**50,34**	**50,46**	50,18
	PK_{mod}	24,45	39,02	47,99	48,96	49,49	49,74	49,93
	PK_{all}	**31,92**	**46,92**	**49,76**	**49,92**	49,94	50,42	**50,54**
XC2V6000	(keine)	8,78	18,23	54,32	57,70	58,71	58,62	59,72
	PK_{col}	22,92	32,93	57,14	61,52	62,47	62,69	63,65
	PK_{mod}	18,27	27,02	45,17	57,44	59,76	61,22	62,87
	PK_{all}	**25,11**	**38,79**	**59,61**	**63,04**	**63,14**	**63,61**	**64,01**

Tabelle 4.17: Mittlere Ressourcenauslastung der Simulationen in Abhängigkeit des Verhältnisses der Ausführungszeit zur Konfigurationszeit t_{EXE}/t_{CFG}.

Vergleicht man die Ressourcenauslastung der Simulationen ohne Defragmentierung mit den Ressourcenauslastungen der Simulationen mit Defragmentierung, so lässt sich erkennen, dass die Defragmentierung sich positiv auf die Ressourcenauslastung auswirkt. Bis auf wenige Ausnahmen zeigt sich, dass bei einer kompletten Defragmentierung (PK_{all}) die resultierende Ressourcenauslastung am größten ist. Ein Grund für die Zunahme der mittleren Ressourcenauslastung ist, dass sich bei einer Defragmentierung die Ausführungszeit der umplatzierten Modulinstanzen entsprechend verlängert. Ob sich jedoch die Leistungsfähigkeit der Platzierung durch Defragmentierung bei den durchgeführten Simulationen verbessert hat, wird bei Betrachtung der Zellabweisung deutlich.

Die mittlere Zellabweisung der Simulationen ist in Tabelle 4.18 abgebildet. Ähnlich wie bei der Ressourcenauslastung ist zu erkennen, dass die mittlere Zellabweisung bei Unterschreiten eines bestimmten Verhältnisses von t_{EXE}/t_{CFG} deutlich ansteigt. Beim XC2V6000 wird die mittlere Zellabweisung unterhalb des Verhältnisses $t_{EXE}/t_{CFG} \approx 20$ deutlich größer, wobei bei dem XC2V2000 die mittlere Zellabweisung erst unterhalb des Verhältnisses von $t_{EXE}/t_{CFG} \approx 5$ deutlich zunimmt.

Im Gegensatz zur Ressourcenauslastung macht sich der Effekt der Defragmentierung erst ab einem bestimmten Verhältnis von t_{EXE}/t_{CFG} bei der mittleren Zellabweisung bemerkbar. Beim XC2V2000 ist ab einem Verhältnis von $t_{EXE}/t_{CFG} \approx 20$ die mittlere Zellabweisung bei Verwendung der partiellen Kompaktierung (PK_{col} und PK_{mod}) geringer als bei der gleichen Simulation ohne Defragmentierung. Beim XC2V4000 zeigt sich eine Verbesserung der Zellabweisung erst ab einem Verhältnis von etwa $t_{EXE}/t_{CFG} \approx 30$ und beim XC2V6000 erst ab einem Verhältnis von etwa $t_{EXE}/t_{CFG} \approx 100$. Beim Vergleich der mittleren Zellabweisung der einzelnen Defragmentierungsverfahren lässt sich erkennen, dass die komplette Defragmentie-

FPGA	Defrag.-Alg.	Mittlere Zellabweisung [%]						
		t_{EXE}/t_{CFG} ≈5	t_{EXE}/t_{CFG} ≈10	t_{EXE}/t_{CFG} ≈20	t_{EXE}/t_{CFG} ≈30	t_{EXE}/t_{CFG} ≈40	t_{EXE}/t_{CFG} ≈50	t_{EXE}/t_{CFG} ≈100
XC2V2000	(ohne)	**28,56**	23,25	22,32	22,11	21,98	21,98	21,89
	PK_{col}	30,94	23,56	20,97	20,35	20,06	19,89	19,37
	PK_{mod}	30,25	**22,97**	**20,95**	**20,12**	**19,82**	**19,63**	**19,32**
	PK_{all}	31,28	23,79	21,29	20,51	20,10	19,88	19,39
XC2V4000	(ohne)	**49,47**	**15,25**	**9,88**	9,57	9,48	9,18	8,81
	PK_{col}	52,84	18,71	10,31	**9,09**	**8,18**	**7,97**	7,63
	PK_{mod}	56,22	24,08	10,96	9,20	8,29	8,13	**7,50**
	PK_{all}	55,51	23,83	12,12	10,82	9,53	8,71	7,62
XC2V6000	(ohne)	**71,08**	**50,71**	**17,17**	**15,00**	**13,88**	**14,12**	13,29
	PK_{col}	73,46	55,40	24,53	17,28	14,93	14,40	12,64
	PK_{mod}	73,65	57,90	30,83	18,72	16,12	14,33	**12,25**
	PK_{all}	74,28	58,87	30,18	22,16	17,63	17,39	14,34

Tabelle 4.18: Mittlere Zellabweisung der Simulationen in Abhängigkeit des Verhältnisses der Ausführungszeit zur Konfigurationszeit t_{EXE}/t_{CFG}.

rung bei fast allen Simulationen einen schlechteren Wert der mittleren Zellabweisung aufweist.

Die hier dargestellten Simulationsergebnisse zeigen, dass sich die Leistungsfähigkeit eines Platzierungsverfahrens durch Defragmentierung geringfügig steigern lässt, wenn das Verhältnis von der Ausführungszeit zur Konfigurationszeit t_{EXE}/t_{CFG} hinreichend groß ist. Grundsätzlich gilt: Je größer das FPGA ist, umso größer muss das Verhältnis von t_{EXE}/t_{CFG} sein, damit sich der Effekt der Defragmentierung bemerkbar macht. In den Simulationen hat sich gezeigt, dass die Minimierung des Konfigurationsaufwands durch Verwendung der partiellen Kompaktierung mit dem Ziel, die Spalten- oder Modulverschiebungen zu minimieren, eine geringere Zellabweisung hervorruft als bei der Verwendung der kompletten Defragmentierung.

4.4 Zusammenfassung

Die Platzierung von Hardware-Modulen in homogenen rekonfigurierbaren Architekturen ist mit einem Packungsproblem vergleichbar und vollzieht sich in zwei Schritten. Der erste Schritt ist die Verwaltung der freien Ressourcen der Architektur. Ein verbreiteter Ansatz zur Einteilung der freien Flächen einer homogenen rekonfigurierbaren Architektur ist die Bestimmung der maximalen freien Rechtecke. Ein freies Rechteck ist genau dann maximal, wenn es sich in keine Richtung ausweiten lässt. Die Bestimmung aller maximalen freien Rechtecke ist insofern optimal, als dass zu jedem platzierbaren Hardware-Modul immer ein maximales freies Rechteck existiert, in dem das Hardware-Modul nach einer entsprechenden Platzierung vollständig enthalten wäre. In Abhängigkeit der Größe der Architektur kann die Anzahl der ma-

ximalen freien Rechtecke jedoch so groß sein, dass die Bestimmung aller maximalen freien Rechtecke sehr aufwendig ist. Aus diesem Grund haben sich heuristische Verfahren etabliert, welche die freien Flächen der rekonfigurierbaren Architektur in nicht überlappende freie Rechtecke partitionieren. Bei Verwendung nicht überlappender freier Rechtecke kann jedoch die Platzierung eines Hardware-Moduls auch bei hinreichend vorhandenen freien Ressourcen nicht garantiert werden.

Nach Berechnung der freien Flächen folgt der zweite Schritt der Platzierung, welcher die Positionsbestimmung ist. Die Positionsbestimmung erfolgt durch Zuweisung des geforderten Hardware-Moduls zu einer passenden freien rechteckigen Fläche. Diese Art der Zuweisung ist mit einem Online-Packungsproblem vergleichbar, so dass die entsprechenden Approximationsverfahren First-Fit, Best-Fit und Worst-Fit verwendet werden können. Durch die Position der freien Fläche ergibt sich dementsprechend auch die Position des Hardware-Moduls.

Im Vergleich zum 2D-Systemansatz ergeben sich im 1D-Systemansatz grundsätzliche Vereinfachungen. Die maximalen freien Rechtecke überschneiden sich nicht mehr wie im 2D-Systemansatz, sondern sind vom Rand der rekonfigurierbaren Architektur oder dem Rand einer Modulinstanz umschlossen. Daher wächst die Anzahl der maximalen freien Rechtecke höchstens linear mit der Anzahl der Modulinstanzen. Ebenso haben die Platzierungsverfahren First-Fit, Best-Fit und Worst-Fit eine Laufzeit, die lediglich von der Anzahl der Spalten der rekonfigurierbaren Architektur abhängt.

Mithilfe der Simulationsumgebung SARA sind die verschiedenen Systemansätze miteinander verglichen worden. Grundlage der Simulationen sind hypothetische Anwendungen, die aus zufällig erzeugten Komponentenanfragen von real existierenden Systemkomponenten der Studie [46] bestehen. Als Metriken zur Analyse der verschiedenen Systemansätze dienen die Ressourcenauslastung, die Zellabweisung, die interne und die externe Fragmentierung. Die externe Fragmentierung wird anhand der relativen Verfügbarkeit quantifiziert.

Im 2D-Systemansatz können Hardware-Module beliebig in horizontaler und vertikaler Richtung auf der rekonfigurierbaren Architektur platziert werden, so dass mit der Zeit eine größere externe Fragmentierung entsteht als im 1D-Systemansatz, bei dem die Hardware-Module nur in horizontaler Richtung platziert werden können. Dadurch, dass die vertikale Flächenausdehnung der Hardware-Module im 1D-Systemansatz immer die komplette Höhe der rekonfigurierbaren Architektur umfasst, entsteht im 1D-Systemansatz eine größere interne Fragmentierung als im 2D-Systemansatz. Im direkten Vergleich erwies sich jedoch der 1D-Systemansatz bei den durchgeführten Simulationen als der Ansatz mit der besten mittleren Ressourcenauslastung und der geringsten mittleren Zellabweisung. Der Einfluss der externen Fragmentierung wirkt sich damit stärker auf die Ressourcenauslastung aus als der Einfluss der internen Fragmentierung. Der Systemansatz mit fester Aufteilung hat in allen Simulationen die schlechtesten Ergebnisse erzeugt.

Der mit heutigen verfügbare Xilinx Virtex FPGAs am meisten genutzte Systemansatz ist der Systemansatz mit fester Aufteilung (z. B. [91]). Für den 2D-Systemansatz existieren erste konzeptionelle Arbeiten, die das Problem der Anpassung der Kommunikationsinfrastruktur zur Laufzeit aufgreifen (vgl. [41]). Mögliche Architekturen für den 2D-Systemansatz sind die FPGAs der Xilinx Virtex-4 Serie, die sich stark eingeschränkt zweidimensional partiell rekonfigurieren lassen, wobei die atomare rekonfigurierbare Einheit ein so genannter *Frame* ist, welcher 16 Zellen innerhalb einer Spalte umfasst. Im Gegensatz zum 2D-Systemansatz lässt sich der 1D-Systemansatz wie in [E1] beschrieben mit heutigen FPGAs umsetzen. Aus diesem Grund wurden Analysen zur Platzierungs- und Konfigurationszeit im Hinblick auf den 1D-Systemansatz durchgeführt.

In den betrachteten Simulationen diente der in Xilinx Virtex-II/Pro FPGAs eingebettete Prozessor IBM PowerPC 405 als Plattform für die Ausführung der Platzierungsalgorithmen. Die gemessenen und approximierten Platzierungszeiten aller untersuchten Platzierungsalgorithmen war dabei so gering, dass in den Simulationen keine Auswirkung auf die mittlere Ressourcenauslastung zu beobachten war. Die Konfigurationszeit hingegen ist beim Xilinx Virtex FPGA im Bereich von Millisekunden und wirkte sich in den betrachteten Simulationen in Abhängigkeit vom Verhältnis zur Ausführungszeit der Modulinstanzen mitunter entscheidend auf die Ressourcenauslastung aus.

Eine mögliche Methode zur Reduzierung der externen Fragmentierung ist die Defragmentierung. Die partielle Kompaktierung ist ein ereignisgesteuertes Defragmentierungsverfahren, welches für den 1D-Systemansatz geeignet ist. Das Verfahren bewirkt eine Verschiebung vorhandener Modulinstanzen mit dem Ziel, eine zuvor unmögliche Modulplatzierung zu ermöglichen. Die Auswahl der zu verschiebenden Modulinstanz kann unter unterschiedlichen Zielsetzungen geschehen, wie z. B. der Minimierung der Umplatzierungszeit. Anhand von Simulationen wurde gezeigt, dass trotz des erhöhten Konfigurationsaufwands, welcher durch die Umplatzierung der Modulinstanzen entsteht, eine Defragmentierung zur Laufzeit anhand der partiellen Kompaktierung eine geringe Verbesserung der mittleren Ressourcenauslastung bewirkt.

5 Heterogene Architekturen

Die im vorherigen Kapitel betrachteten Untersuchungen beziehen sich ausschließlich auf homogene rekonfigurierbare Architekturen. Die Eigenschaft der Homogenität ist nur dann gegeben, wenn alle Zellen der Architektur identisch sind und die Kommunikationsinfrastruktur eine uneingeschränkt feingranulare Platzierung von Hardware-Modulen zulässt. Diese Voraussetzungen sind in den meisten heutigen partiell rekonfigurierbaren Architekturen jedoch nicht gegeben. In den Xilinx Virtex-II FPGAs befinden sich z. B. neben den Logikzellen auch Speicherblöcke (BlockRAM) oder DSP-Zellen, die in unregelmäßigen Abständen auf der Architektur verteilt sind. Neben der Anordnung der verschiedenen Zellen ergibt sich darüber hinaus eine Heterogenität, die durch die Kommunikationsinfrastruktur hervorgerufen wird, denn die Kommunikationsinfrastruktur beinhaltet Verbindungen unterschiedlicher Länge, die eine uneingeschränkte feingranulare Platzierung ebenfalls verhindern.

Die in den heutigen partiell rekonfigurierbaren Architekturen vorliegende Heterogenität wirft die Frage auf, ob die Methoden und Konzepte im Bereich der homogenen rekonfigurierbaren Architekturen ohne Weiteres auf heterogene rekonfigurierbare Architekturen übertragbar sind. In diesem Kapitel werden daher die notwendigen Schritte zur Anpassung bestehender Verfahren verdeutlicht und neue Verfahren beschrieben, die den Anforderungen heterogener rekonfigurierbarer Architekturen gewachsen sind.

Dieses Kapitel ist wie folgt strukturiert. In Abschnitt 5.1 wird auf die Platzierungsverfahren eingegangen, die speziell für heterogene rekonfigurierbare Architekturen geeignet sind. Dabei werden zunächst die nötigen Anpassungen der klassischen Platzierungsverfahren (First-Fit, Best-Fit, etc.) für den 2D-Systemansatz beschrieben. Des Weiteren werden zwei neue Platzierungsverfahren vorgestellt, die die Einschränkungen durch die unregelmäßige Verteilung der möglichen Positionen der Hardware-Module in der Platzierung berücksichtigen. Ebenso wird auf Vereinfachungen der zuvor beschriebenen Platzierungsalgorithmen bei Verwendung im 1D-Systemansatz eingegangen. Der Abschnitt 5.2 behandelt die detaillierte simulative Analyse der Platzierungsverfahren sowohl im 2D-Systemansatz als auch im 1D-Systemansatz. In Abschnitt 5.3 wird erneut auf das Konzept der Defragmentierung zur Laufzeit eingegangen. Dabei wird ein für heterogene rekonfigurierbare Architekturen geeignetes Defragmentierungsverfahren vorgestellt. Anhand von Simulationen wird der Einfluss des Defragmentierungsverfahrens auf die Ressourcenauslastung analysiert.

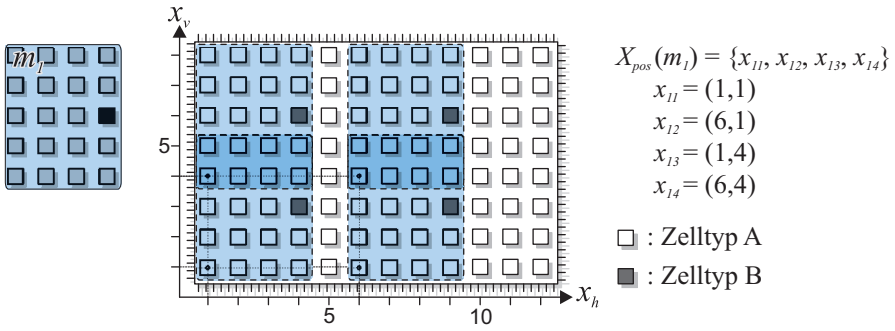
5.1 Platzierungsverfahren

Für die folgenden Abschnitte sei wie bereits im vorherigen Kapitel angenommen, dass die Anfragezeitpunkte und die Ausführungszeiten der von der Anwendung geforderten Systemkomponenten unbekannt sind. Durch die Heterogenität der Architektur können die verschiedenen Hardware-Module nicht mehr uneingeschränkt platziert werden. Wie bereits in Abschnitt 3.1 beschrieben, wird bei der Synthese der Systemkomponenten ein Hardware-Modul erzeugt, welches keine absolute, sondern eine relative Zuordnung der gegebenen Zellen und deren Verbindungen darstellt. Die endgültige Zuordnung der Zellen wird erst zur Laufzeit bei der Modulplatzierung vorgenommen. Da homogene rekonfigurierbare Architekturen aus Zellen eines Typs bestehen, ergeben sich keine Einschränkungen bei der Platzierung eines Hardware-Moduls. Wenn jedoch eine heterogene rekonfigurierbare Architektur vorliegt, und ein Hardware-Modul aus verschiedenen Zelltypen besteht, so muss bei der Platzierung des Moduls gewährleistet sein, dass bei der endgültigen Zuordnung der Zellen die verschiedenen Zelltypen berücksichtigt werden. Das resultierende Hardware-Modul kann also nur dort platziert werden, wo die gleiche Anordnung der verschiedenen Zellen - also das gleiche Muster - auf der Architektur vorhanden ist. Die Menge der möglichen Positionen eines Hardware-Moduls $X_{pos}(m)$ unterliegt der Anordnung der verschiedenen Zellen und kann daher nicht mehr mithilfe von (4.1) hergeleitet werden. Die Bestimmung der Menge der möglichen Positionen eines Hardware-Moduls wird anhand des folgenden Beispiels verdeutlicht.

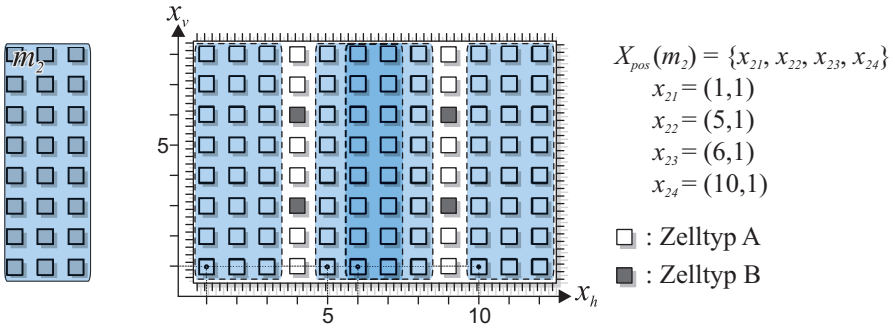
Beispiel 5.1. *Gegeben sei eine heterogene rekonfigurierbare Architektur mit* $N_{col} = 12$ *Spalten und* $N_{row} = 8$ *Reihen. Die Architektur bestehe aus Zellen von zwei unterschiedlichen Zelltypen (A,B). Der Zelltyp A könnte z. B. Logikzellen darstellen, wobei Zelltyp B z. B. Speicherblöcke darstellt. Die Anordnung der Zellen ist in Abbildung 5.1 dargestellt. Gegeben sei die Menge der Hardware-Module* $M = \{m_1, m_2\}$. *Das in Abbildung 5.1(a) dargestellte Hardware-Modul* m_1 *besteht aus 19 Zellen des Typs A und einer Zelle des Typs B. Entsprechend der Anordnung der Zellen innerhalb des Moduls* m_1 *ergeben sich die angedeuteten möglichen Positionen* $X_{pos}(m_1) = \{(1,1),(6,1),(1,4),(6,4)\}$. *Das zweite Modul* m_2 *ist in Abbildung 5.1(b) dargestellt und besteht aus 24 Zellen des Typs A. Die für die gegebene Architektur resultierende Menge der möglichen Positionen ist* $X_{pos}(m_2) = \{(1,1),(5,1),(6,1),(10,1)\}$.

Die Anzahl und Verteilung der möglichen Positionen der einzelnen Hardware-Module hängt also von der Anordnung der verschiedenen Zellen der Architektur und der Anordnung der benötigten Zellen innerhalb der einzelnen Hardware-Module ab.

Die Platzierung eines Hardware-Moduls lässt sich, wie im Zusammenhang mit homogenen Architekturen, erneut in die Verwaltung der freien Ressourcen und die Positionsbestimmung eines geforderten Hardware-Moduls unterteilen. Die Verwaltung



(a) Menge der möglichen Positionen für m_1.



(b) Menge der möglichen Positionen für m_2.

Abbildung 5.1: Beispiel einer heterogenen rekonfigurierbaren Architektur mit den entsprechenden Mengen der möglichen Positionen für die Hardware-Module m_1, m_2.

der freien Ressourcen besteht erneut im Wesentlichen aus der Bestimmung der freien zusammenhängenden Flächen der heterogenen Architektur. Hierbei können die in Verbindung mit homogenen Architekturen im Abschnitt 4.1.1 angegebenen Methoden verwendet werden.

Bei der Bestimmung einer Position $x \in X_{free}(m)$ eines geforderten Hardware-Moduls ergeben sich grundlegende Unterschiede, die auf die unregelmäßige Verteilung der möglichen Positionen zurückzuführen sind. Auf die Unterschiede und die erforderlichen Anpassungen der bestehenden Platzierungsverfahren wird im folgenden Abschnitt eingegangen.

5.1.1 Anpassung bestehender Platzierungsverfahren

Die Platzierung eines Hardware-Moduls zur Laufzeit ist erneut mit einem zweidimensionalen Packungsproblem vergleichbar. Freie rechteckige Flächen dienen als Boxen und das zu platzierende Hardware-Modul stellt das Objekt dar. Jedoch ergeben sich zusätzliche Einschränkungen bei der Zuordnung von Hardware-Modulen zu den freien rechteckigen Flächen, denn neben dem Flächenbedarf des Hardware-Moduls

muss ebenso die Anordnung der benötigten Zellen der verschiedenen Zelltypen berücksichtigt werden. Im 2D-Systemansatz ergibt sich demnach die folgende Menge der geeigneten freien Rechtecke eines geforderten Hardware-Moduls:

Definition 5.1 (Menge der geeigneten freien Rechtecke)**.** *Sei E die Menge der gegebenen freien Rechtecke und $m_{req} \in M$ ein gefordertes Hardware-Modul mit der Menge der möglichen Positionen $X_{pos}(m_{req})$. Bei Verwendung einer heterogenen rekonfigurierbaren Architektur ist die Menge der für die Platzierung des geforderten Hardware-Moduls m_{req} geeigneten freien Rechtecke:*

$$\begin{aligned} E_{free}(m) = \big\{ (x_{eh}, x_{ev}, a_{eh}, a_{ev}) \mid\ & (x_{eh}, x_{ev}, a_{eh}, a_{ev}) \in E \wedge \\ & \exists (x_h, x_v) \in X_{pos}(m_{req}) : x_h \geq x_{eh} \wedge x_v \geq x_{ev} \wedge \\ & x_h + a_h(m_{req}) \leq x_{eh} + a_{eh} \wedge \\ & x_v + a_v(m_{req}) \leq x_{ev} + a_{ev} \big\} \end{aligned} \tag{5.1}$$

Das geforderte Hardware-Modul wird innerhalb der freien Flächen nicht mehr an einer festen Position, wie der unteren linken Ecke platziert, sondern an einer der möglichen Positionen des Hardware-Moduls.

Legt man die Menge der geeigneten freien Flächen bei der Modulplatzierung zugrunde, wird bei Platzierung eines geforderten Hardware-Moduls $m_{req} \in M$ anhand des First-Fit-Verfahrens das erste Rechteck $(x_{eh}, x_{ev}, a_{eh}, a_{ev}) \in E_{free}(m_{req})$ ausgewählt. Beim Best-Fit-Verfahren wird das Rechteck mit der geringsten Fläche $(a_{eh} \cdot a_{ev})$, und entsprechend wird beim Worst-Fit-Verfahren das Rechteck mit der größten Fläche gewählt. Innerhalb des gewählten Rechtecks wird in allen Verfahren das Hardware-Modul an einer der möglichen Positionen platziert.

Sieht man von den hier beschriebenen für heterogene Architekturen angepasste Verfahren (First-Fit, Best-Fit und Worst-Fit) ab, so gibt es bisher nur wenige Platzierungsverfahren, die Einschränkungen der Platzierung durch Heterogenität der Architektur berücksichtigen. Banerjee et al. beschreiben in [6] ein heuristisches Verfahren zur Offline-Platzierung von Hardware-Modulen auf heterogenen rekonfigurierbaren Architekturen. D. h., die Ausführungszeiten der Hardware-Module und funktionale Abhängigkeiten untereinander sind zur Entwurfszeit bekannt, so dass die Platzierung und der Ablauf in Form eines Optimierungsproblems dargestellt werden kann. Banerjee et al. stellten fest, dass die exakte Lösung des Optimierungsproblems schon für eine kleine Anzahl an Hardware-Modulen eine sehr hohe Rechenzeit erfordern, so dass ein heuristisches Verfahren entwickelt wurde, um eine hinreichend gute Lösung in akzeptabler Zeit zu finden. Das Verfahren ist auf die Offline-Platzierung beschränkt und kann im Zusammenhang mit der hier betrachteten Online-Platzierung von Hardware-Modulen, bei denen die Ausführungszeit und Anfragezeitpunkt unbekannt sind, nicht verwendet werden.

Grundlage der klassischen Verfahren (First-Fit, Best-Fit und Worst-Fit) ist die freie Platzierung von Objekten in Boxen mit dem untergeordneten Ziel, die externe Frag-

mentierung gering zu halten. Durch Heterogenität ist die Eigenschaft der freien Platzierung nicht mehr gegeben, da die Anzahl der möglichen Positionen der Hardware-Module stark eingeschränkt ist. Ziel eines Platzierungsalgorithmus sollte daher neben der Minimierung der Fragmentierung ebenso die Erhaltung der freien Positionen aller Hardware-Module sein. Aus diesem Grund ist ein neuartiger Platzierungsansatz entwickelt worden, welcher in den folgenden beiden Abschnitten im Einzelnen beschrieben wird.

5.1.2 SUP-Fit

Die in heterogenen rekonfigurierbaren Architekturen vorhandene Einschränkung durch die unregelmäßige Verteilung der möglichen Positionen der einzelnen Hardware-Module ist Grund für die Entwicklung eines für heterogene rekonfigurierbare Architekturen geeigneten Platzierungsverfahrens. Die in [E5] beschriebenen heterogenen Verfahren *SUP-Fit* (engl.: Static Utilization Probabilty Fit) und *RUP-Fit* (engl.: Run-time Utilization Probability Fit) sind nach heutigem Kenntnisstand die ersten Ansätze, die die Einschränkungen heterogener rekonfigurierbarer Architekturen bei der Platzierung von Hardware-Modulen zur Laufzeit berücksichtigen. In diesem Abschnitt wird auf das SUP-Fit-Verfahren eingegangen, welches auch als *Platzierung anhand statischer Zellgewichte* bezeichnet werden kann. Ein Großteil des Verfahrens wird dabei zur Entwurfszeit ausgeführt. Dabei ergeben sich folgende Schritte:

1. In Abhängigkeit von den möglichen Positionen der gegebenen Hardware-Module wird für jede Zelle der Architektur ein so genanntes *statisches Zellgewicht* berechnet. Das *statische Zellgewicht* einer Zelle deutet die Wahrscheinlichkeit an, mit der die Zelle bei Platzierung eines Hardware-Moduls belegt wird.

2. Für jede mögliche Position eines Hardware-Moduls $m \in M$ wird durch Mittelwertbildung der statischen Zellgewichte der verwendeten Zellen ein so genanntes *Positionsgewicht* bestimmt. Das Positionsgewicht einer möglichen Position ist ein Maß dafür, ob die infolge einer Platzierung belegten Zellen von vielen anderen Hardware-Modulen genutzt werden können.

3. Die möglichen Positionen der einzelnen Hardware-Module werden entsprechend der Positionsgewichte sortiert, beginnend mit der möglichen Position mit dem geringsten Gewicht.

Zur Laufzeit wird beim SUP-Fit-Verfahren die Verfügbarkeit der möglichen Positionen des geforderten Hardware-Moduls in der Reihenfolge der Positionsgewichte überprüft. Das Hardware-Modul wird an der freien möglichen Position mit dem geringsten Gewicht platziert. Die infolge der Platzierung belegten Zellen besitzen im Mittel die geringste Wahrscheinlichkeit von anderen Modulen genutzt zu werden.

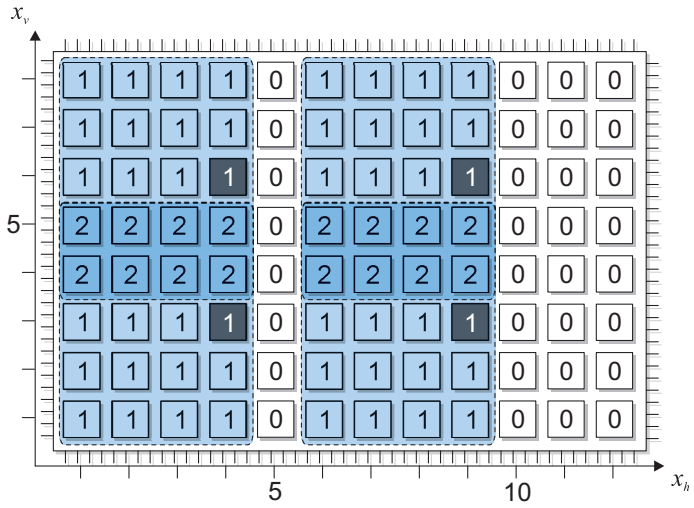


Abbildung 5.2: Beispiel der statischen Positionsüberlappung $o_{pos,st}(m_1, x_h, x_v)$ für das Hardware-Modul m_1 aus Abbildung 5.1.

Im Folgenden wird auf die für die Berechnung der Positionsgewichte erforderlichen Schritte im Einzelnen eingegangen. Ausgegangen wird dabei von der *statischen Positionsüberlappung* $o_{pos,st}(m, x_h, x_v)$.

Definition 5.2 (Statische Positionsüberlappung)**.** *Sei* $o_{pos,st}(m, x_h, x_v)$ *die* statische Positionsüberlappung*, wobei* $m \in M$, $x_h \in [1, N_{col}]$*, und* $x_v \in [1, N_{row}]$*. Der Wert der statischen Positionsüberlappung stellt die Anzahl der möglichen Positionen* $|X_{pos}(m, x_h, x_v)|$ *des Hardware-Moduls m dar, welche bei einer Modulplatzierung die Belegung der Zelle an Position* (x_h, x_v) *hervorrufen würde. Sei* $a_h(m)$ *die horizontale Flächenausdehnung und* $a_v(m)$ *die vertikale Flächenausdehnung des Hardware-Moduls* $m \in M$*. Die statische Positionsüberlappung* $o_{pos,st}(m, x_h, x_v)$ *ist demnach wie folgt definiert:*

$$
\begin{aligned}
& o_{pos,st}(m, x_h, x_v) = |X_{pos}(m, x_h, x_v)|, \text{ wobei} \\
& X_{pos}(m, x_h, x_v) = \big\{(i, j) \mid (i, j) \in X_{pos}(m) \wedge i \leq x_h < i + a_h(m) \wedge \\
& \qquad\qquad\qquad\qquad\qquad\qquad j \leq x_v < j + a_v(m)\big\}
\end{aligned}
\tag{5.2}
$$

Abbildung 5.2 zeigt ein Beispiel der statischen Positionsüberlappung des bereits in Abbildung 5.1 dargestellten Hardware-Moduls m_1.

Sei $p_{sel}(m)$ die *Auswahlwahrscheinlichkeit* eines Hardware-Moduls $m \in M$. Dabei beschreibt $p_{sel}(m)$ die Wahrscheinlichkeit mit der bei einer Platzierung das Hardware-Modul m gewählt wird, so dass $\sum_{m \in M} p_{sel}(m) = 1$ gilt. Im Folgenden sei zunächst angenommen, dass eine Gleichverteilung der Auswahlwahrscheinlichkeiten vorliegt ($p_{sel}(m) = 1/|M|$). Mittels der statischen Positionsüberlappung $o_{pos,st}(m, x_h, x_v)$ und

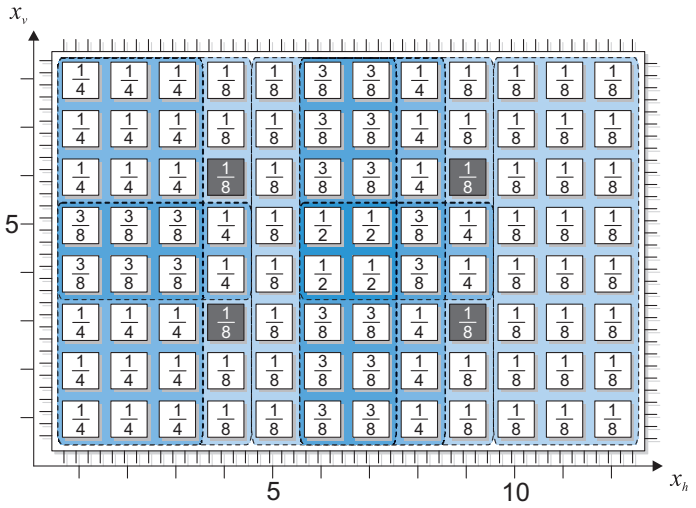


Abbildung 5.3: Beispiel der statischen Zellgewichte $w_{cell,st}(x_h,x_v)$ der Architektur und Hardware-Module $M = \{m_1, m_2\}$ aus Beispiel 5.1.

der Auswahlwahrscheinlichkeit $p_{sel}(m)$ lässt sich das *statische Zellgewicht* bestimmen.

Definition 5.3 (Statisches Zellgewicht)**.** *Gegeben sei eine heterogene rekonfigurierbare Architektur mit N_{col} Spalten und N_{row} Zeilen. Ferner sei gegeben die Menge der möglichen Positionen $X_{pos}(m)$, die entsprechende statische Positionsüberlappung $o_{pos,st}(m,x_h,x_v)$ und die Auswahlwahrscheinlichkeit $p_{sel}(m)$ der einzelnen Hardware-Module $m \in M$. Das* statische Zellgewicht *$w_{cell,st}(x_h,x_v)$ einer Zelle an horizontaler Position $x_h \in [1,N_{col}]$ und vertikaler Position $x_v \in [1,N_{row}]$ ist wie folgt definiert.*

$$w_{cell,st}(x_h,x_v) = \sum_{m \in M} \frac{p_{sel}(m) \cdot o_{pos,st}(m,x_h,x_v)}{|X_{pos}(m)|} \tag{5.3}$$

Das statische Zellgewicht lässt sich als Wahrscheinlichkeit, mit der die Zelle bei der Platzierung eines Hardware-Moduls belegt wird, interpretieren. Eine alternative Interpretation des statischen Zellgewichts ergibt sich unter der bereits getroffenen Annahme, dass die Auswahlwahrscheinlichkeiten der Hardware-Module gleichverteilt sind. Denn dann beschreibt das statische Zellgewicht den Anteil der möglichen Positionen aller Module, die eine Belegung der Zelle an Position (x_h,x_v) verursachen würden. Für die rekonfigurierbare Architektur mit den Hardware-Modulen $M = \{m_1,m_2\}$ aus Beispiel 5.1 sind die resultierenden statischen Zellgewichte in Abbildung 5.3 dargestellt. Das statische Zellgewicht der Zelle an Position $(6,4)$ ist $\frac{1}{2} = 0,5$, denn 2 von den 4 möglichen Positionen des Moduls m_1 und weitere 2 von den 4 möglichen Positionen des Moduls m_2 würden bei einer Modulplatzierung die Belegung der Zelle verursachen. Bei der benachbarten Zelle an Position $(5,4)$

ist das statische Zellgewicht $\frac{1}{8} = 0,125$, da die Zelle nur bei einer Platzierung des Hardware-Moduls m_2 an Position $(1,5)$ belegt werden kann.

Mithilfe der statischen Zellgewichte kann nun für jede mögliche Position der einzelnen Hardware-Module ein *statisches Positionsgewicht* bestimmt werden, das wie folgt definiert ist:

Definition 5.4 (Statisches Positionsgewicht)**.** *Gegeben sei eine heterogene rekonfigurierbare Architektur mit N_{col} Spalten und N_{row} Zeilen. Für jedes Hardware-Modul $m \in M$ sei die Menge der möglichen Positionen $X_{pos}(m)$ gegeben. Ferner sei $w_{cell,st}(x_h,x_v)$ das statische Zellgewicht der Zelle an Position (x_h,x_v). Das* statische Positionsgewicht *$w_{pos,st}(m,x_{ph},x_{pv})$ einer möglichen Position $(x_{ph},x_{pv}) \in X_{pos}(m)$ des Hardware-Moduls $m \in M$ stellt den quadratischen Mittelwert der statischen Zellgewichte der infolge einer Modulplatzierung belegten Zellen dar und ist demnach wie folgt definiert:*

$$w_{pos,st}(m,x_{ph},x_{pv}) = \sqrt{\frac{1}{a_h(m) \cdot a_v(m)} \sum_{x_h,x_v} w_{cell,st}(x_h,x_v)^2} \text{ , wobei}$$
$$x_h \in [x_{ph}, x_{ph} + a_h(m) - 1] \text{ , } \quad x_v \in [x_{pv}, x_{pv} + a_v(m) - 1] \tag{5.4}$$

Die statischen Positionsgewichte des Hardware-Moduls m_2 aus Beispiel 5.1 sind in Abbildung 5.4 dargestellt. Das Gewicht der Position $x = (11,1)$ ist mit $w_{pos,st}(m_2,x_h,x_v) = 0,125$ das geringste, denn die infolge einer Modulplatzierung belegten Zellen können ausschließlich von dem Hardware-Modul m_2 genutzt werden.

Nach Berechnung der Positionsgewichte werden die möglichen Positionen der einzelnen Hardware-Module entsprechend der statischen Positionsgewichte, beginnend mit dem geringsten Gewicht, sortiert. Die bisherigen Schritte werden einmalig zur Entwurfszeit ausgeführt. Zur Laufzeit werden die möglichen Positionen $X_{pos}(m_{req})$ des geforderten Hardware-Moduls $m_{req} \in M$ in der Reihenfolge der Positionsgewichte überprüft, so dass das geforderte Hardware-Modul an der freien möglichen Position mit dem geringsten Gewicht platziert wird. Sobald die Reihenfolge der Positionsgewichte bestimmt ist, haben diese selbst keinen Einfluss mehr auf die Platzierung, so dass die einzelnen Werte zur Laufzeit nicht gespeichert werden müssen, sondern lediglich die Reihenfolge der Positionsgewichte. Das SUP-Fit-Verfahren weist damit Ähnlichkeiten zum First-Fit-Verfahren auf. Der einzige Unterschied zum First-Fit-Verfahren besteht darin, dass beim SUP-Fit-Verfahren die Reihenfolge der möglichen Positionen anhand der Positionsgewichte festgelegt wird.

Eine effiziente Umsetzung des SUP-Fit-Verfahrens für den 2D-Systemansatz ist mithilfe der von Handa et al. in [38] definierten Datenstruktur möglich. Die Datenstruktur ist mit dem rechtsseitigen Zellzusammenhang (vgl. Definition 4.9) $\delta_r(i,j)$

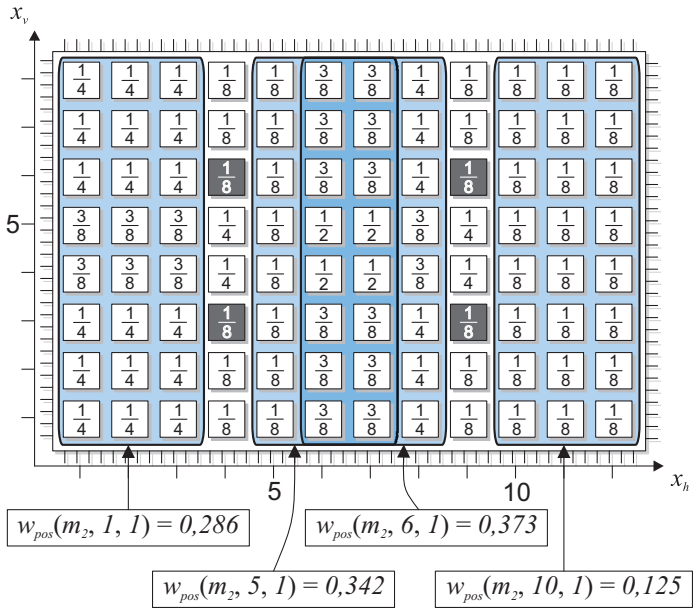


Abbildung 5.4: Beispiel der statischen Positionsgewichte $w_{pos,st}(m_2, x_{ph}, x_{pv})$ des Hardware-Moduls m_2 aus Beispiel 5.1.

vergleichbar und wird daher im Folgenden als *orthogonaler Zellzusammenhang* $\delta_o(i,j)$ bezeichnet.

Definition 5.5 (Orthogonaler Zellzusammenhang)**.** *Gegeben sei die Zellbelegung $b(i,j)$ der momentan platzierten Modulinstanzen. Wenn die Zelle an Position (i,j) frei ist ($b(i,j) = 1$), dann entspricht der Wert des orthogonalen Zellzusammenhangs $\delta_o(i,j)$ der Anzahl der nach oben verlaufenden aufeinander folgenden freien Zellen, ausgehend von der Zelle an Position (i,j). Wenn die Zelle an Position (i,j) belegt ist ($b(i,j) = 0$), dann ist der Wert von $\delta_o(i,j) < 0$, und der Betrag $|\delta_o(i,j)|$ entspricht der Anzahl der nach rechts verlaufenden aufeinander folgenden belegten Zellen, ausgehend von der Zelle an Position (i,j).*

Beispiel 5.2. *Gegeben sei eine heterogene rekonfigurierbare Architektur und die Menge der Hardware-Module aus Beispiel 5.1. Die Menge der momentan platzierten Modulinstanzen sei $C = \{c_1\}$, wobei die Modulinstanz c_1 auf dem Hardware-Modul $m_1 \in M$ basiert. Die Modulinstanz c_1 sei an der Position $(6,4) \in X_{pos}(m_1)$ platziert. Der entsprechende orthogonale Zellzusammenhang $\delta_o(i,j)$ ist in Abbildung 5.5 dargestellt.*

Nach dem Platzieren einer neuen Modulinstanz oder dem Löschen einer terminierten Modulinstanz ist es nicht notwendig, alle Werte des orthogonalen Zellzusammenhangs $\delta_o(i,j)$ neu zu berechnen, sondern nur die Werte der Zellen, die von der Modulinstanz belegt sind oder unterhalb der Modulinstanz liegen. Effiziente Methoden

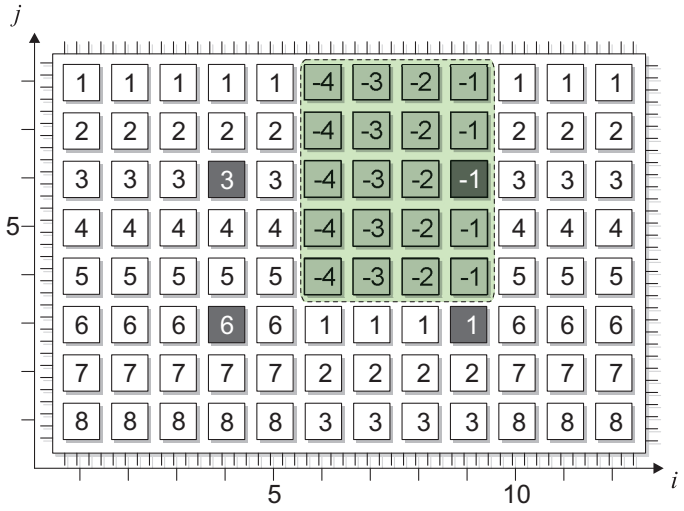


Abbildung 5.5: Beispiel des orthogonalen Zellzusammenhangs $\delta_o(i, j)$.

zur Aktualisierung von $\delta_o(i, j)$ beim Platzieren oder Löschen von Modulinstanzen sind in [38] beschrieben.

Eine einfache Umsetzung des SUP-Fit-Verfahrens ist in Algorithmus 5.1 abgebildet. Das Verfahren benötigt den orthogonalen Zellzusammenhang $\delta_o(i, j)$ und durchsucht die möglichen Positionen entsprechend der zur Entwurfszeit anhand der Positionsgewichte bestimmten Reihenfolge. Ausgehend von der gewählten möglichen Position (x_{ph}, x_{pv}) werden nun die nach rechts verlaufende Werte des orthogonalen Zellzusammenhangs überprüft. Wenn $a_h(m)$ aufeinander folgende Werte des orthogonalen Zellzusammenhangs mindestens den Wert von $a_v(m)$ haben, dann sind hinreichend viele freie zusammenhängende Zellen vorhanden, um die Platzierung des Hardware-Moduls m an der gewählten möglichen Position (x_{ph}, x_{pv}) zu ermöglichen.

Theorem 5.1. *Im 2D-Systemansatz hat das SUP-Fit-Verfahren anhand Algorithmus 5.1 eine Laufzeit von* $\mathcal{O}(N_{row} \cdot {N_{col}}^2)$.

Beweis. Anhand der Schleifenstruktur lässt sich ableiten, dass der Algorithmus maximal $|X_{pos}(m)| \cdot a_h(m)$ Schritte benötigt. Die Anzahl der möglichen Positionen $|X_{pos}(m)|$ für ein Hardware-Modul m ist genau dann maximal, wenn es wie in einer homogenen Architektur ohne Einschränkungen platziert werden kann, so dass zur Bestimmung der möglichen Positionen (4.1) verwendet werden kann. Die resultierende Anzahl der möglichen Positionen ist somit

$$|X_{pos}(m)| = (N_{row} - a_v(m) + 1) \cdot (N_{col} - a_h(m) + 1) \; . \tag{5.5}$$

Eingabe: Orthogonaler Zellzusammenhang $\delta_o(i,j)$ gemäß Definition 5.5 und gefordertes Hardware-Modul $m_{req} \in M$, Menge der möglichen Positionen $X_{pos}(m_{req}) = \{(x_{ph}(m_{req},n), x_{pv}(m_{req},n))\}$ mit $n \in [1, |X_{pos}(m_{req})|]$ sortiert nach $w_{cell,st}(m_{req}, x_{ph}(m_{req},n), x_{pv}(m_{req},n))$.

Ausgabe: Position $(x_h, x_v) \in X_{pos}(m_{req})$ des geforderten Hardware-Moduls $m_{req} \in M$ für den 2D-Systemansatz.

(1) $a_{temp} \leftarrow 0$; $n \leftarrow 1$; $x_h \leftarrow 0$; $x_v \leftarrow 0$
(2) **while** $n \leq |X_{pos}(m_{req})|$ **and** $a_{temp} < a_h(m_{req})$
(3) $i \leftarrow 0$
(4) **while** $i < a_h(m_{req})$ **and** $i = a_{temp}$
(5) **if** $\delta_o(x_{ph}(m_{req},n)+i, x_{pv}(m_{req},n)) \geq a_v(m_{req})$
(6) $a_{temp} \leftarrow a_{temp} + 1$
(7) **else**
(8) $a_{temp} \leftarrow 0$
(9) **end if**
(10) $i \leftarrow i + 1$
(11) **end while**
(12) **if** $a_{temp} = a_h(m_{req})$
(13) $x_h \leftarrow x_{ph}(m_{req},n)$; $x_v \leftarrow x_{pv}(m_{req},n)$
(14) **end if**
(15) **end while**

Algorithmus 5.1: Positionsbestimmung im 2D-Systemansatz anhand des SUP-Fit-Verfahrens.

Unter Verwendung von (5.5) ist die maximale Anzahl der Schritte von Algorithmus 5.1 genau dann erreicht, wenn $a_v(m) = 1$ und $a_h(m) = \lceil N_{col}/2 \rceil$ ist, so dass die resultierende Anzahl der Schritte

$$N_{row} \cdot \left(\left\lfloor \frac{N_{col}}{2} \right\rfloor + 1 \right) \cdot \left\lceil \frac{N_{col}}{2} \right\rceil$$

ist, was einer Laufzeit von $\mathcal{O}(N_{row} \cdot {N_{col}}^2)$ entspricht. □

Eine alternative Umsetzung des SUP-Fit-Verfahrens, welche ebenfalls auf dem orthogonalen Zellzusammenhang basiert, ist in Algorithmus 5.2 abgebildet. Das Verfahren überprüft zunächst, unabhängig von der Menge der möglichen Positionen

Eingabe: Orthogonaler Zellzusammenhang $\delta_o(i,j)$ gemäß Definition 5.5 und gefordertes Hardware-Modul $m_{req} \in M$, Menge der möglichen Positionen $X_{pos}(m_{req}) = \{(x_{ph}(m_{req},n), x_{pv}(m_{req},n))\}$ mit $n \in [1, |X_{pos}(m_{req})|]$ sortiert nach $w_{cell,st}(m_{req}, x_{ph}(m_{req},n), x_{pv}(m_{req},n))$.

Ausgabe: Position $(x_h, x_v) \in X_{pos}(m_{req})$ des geforderten Hardware-Moduls $m_{req} \in M$ für den 2D-Systemansatz.

(1) $a_{temp} \leftarrow 0$; $n \leftarrow 1$; $x_h \leftarrow 0$; $x_v \leftarrow 0$
(2) **for** $j \leftarrow 1, \ldots, N_{row} - a_v(m_{req}) + 1$
(3) **for** $i \leftarrow 1, \ldots, N_{col}$
(4) $b_{pos}(i,j) \leftarrow 0$
(5) **if** $\delta_o(i,j) \geq a_v(m_{req})$
(6) $a_{temp} \leftarrow a_{temp} + 1$
(7) **if** $a_{temp} \geq a_h(m_{req})$
(8) $b_{pos}(i - a_h(m_{req}) + 1, j) \leftarrow 1$
(9) **end if**
(10) **else**
(11) $a_{temp} \leftarrow 0$
(12) **end if**
(13) **end for**
(14) **end for**
(15) **while** $n \leq |X_{pos}(m_{req})|$ **and** $x_h = 0$ **and** $x_v = 0$
(16) **if** $b_{pos}(x_{ph}(m_{req},n), x_{pv}(m_{req},n)) = 1$
(17) $x_h \leftarrow x_{ph}(m_{req},n)$; $x_v \leftarrow x_{pv}(m_{req},n)$
(18) **end if**
(19) $n \leftarrow n + 1$
(20) **end while**

Algorithmus 5.2: Alternative Positionsbestimmung im 2D-Systemansatz anhand des SUP-Fit-Verfahrens.

$X_{pos}(m)$ des geforderten Hardware-Moduls m, an welchen Positionen eine rechteckige Fläche von freien zusammenhängenden Zellen vorhanden ist, die dem Flächenbedarf des Hardware-Moduls entspricht. Dementsprechend wird in den Schritten (2)-(14) die Variable $b_{pos}(i,j)$ auf 1 gesetzt, wenn eine rechteckige Fläche von freien zusammenhängenden Zellen, die der Größe $(a_h(m), a_v(m))$ entspricht, existiert. Existiert die freie Fläche nicht, dann wird $b_{pos}(i,j) = 0$ gesetzt. Nach Bestimmung der zum Flächenbedarf des Hardware-Moduls passenden freien Flächen wird in den Schritten (15)-(20) in der Reihenfolge der Positionsgewichte nach der ersten möglichen Position gesucht, welche deckungsgleich mit einer der zuvor bestimmten freien Flächen ist. Der Algorithmus ist beendet, wenn eine passende mögliche Position gefunden ist (Schritt (16)-(18)), oder alle möglichen Positionen durchsucht wurden und folglich die Platzierung des Hardware-Moduls nicht möglich ist.

Theorem 5.2. *Im 2D-Systemansatz hat das SUP-Fit-Verfahren anhand Algorithmus 5.2 eine Laufzeit von* $\mathcal{O}(N_{row} \cdot N_{col})$.

Beweis. Anhand der Schleifenstruktur lässt sich erkennen, dass der Algorithmus maximal $(N_{row} - a_v(m) + 1) \cdot N_{col} + |X_{pos}(m)|$ Iterationen benötigt. Die Anzahl der möglichen Positionen $|X_{pos}(m)|$ eines Hardware-Moduls m kann die Anzahl der Zellen einer Architektur nicht überschreiten, woraus folgt, dass der Algorithmus eine Laufzeit von $\mathcal{O}(N_{row} \cdot N_{col})$ hat. □

Obwohl Algorithmus 5.2 ein besseres Laufzeitverhalten aufweist, werden in jedem Fall $N_{col} \cdot (N_{row} - a_v(m) + 1)$ Schritte benötigt, um die freien Flächen zu bestimmen. Wenn also gilt, dass $\forall m \in M : |X_{pos}(m)| \cdot a_h(m) \leq N_{col} \cdot (N_{row} - a_v(m) + 1)$, dann verursacht Algorithmus 5.1 eine kürzere Laufzeit als Algorithmus 5.2.

Eine Erweiterung des SUP-Fit-Verfahrens stellt das RUP-Fit-Verfahren dar, welches im folgenden Abschnitt im Einzelnen beschrieben wird.

5.1.3 RUP-Fit

Das in diesem Abschnitt vorgestellte RUP-Fit-Verfahren kann auch als *Platzierung anhand dynamischer Zellgewichte* bezeichnet werden. Das Verfahren weist dabei eine Ähnlichkeit zum SUP-Fit-Verfahren auf, d. h., die Platzierung wird anhand von Positionsgewichten durchgeführt. Der wesentliche Unterschied ist jedoch, dass die Gewichte der freien Positionen $X_{free}(m) \subseteq X_{pos}(m)$ des zu platzierenden Hardware-Moduls m zur Laufzeit entsprechend der aktuellen Zellbelegung hergeleitet werden. Während beim SUP-Fit-Verfahren die Auswahlwahrscheinlichkeiten der Hardware-Module gleichverteilt sind, wird beim RUP-Fit-Verfahren von keiner festen Verteilung ausgegangen, sondern die Auswahlwahrscheinlichkeit wird durch ein *adaptives Auswahlgewicht* ersetzt, das wie folgt definiert ist:

Definition 5.6 (Adaptives Auswahlgewicht)**.** *Sei M die Menge der gegebenen Hardware-Module. N_{sel} markiert die Anzahl aller bereits angeforderten Hardware-Module, wobei $N_{sel}(m)$ die Anzahl der geforderten Platzierungen des Hardware-Moduls $m \in M$ ist. Die Konstante $\eta \in [0,1]$ sei eine* Adaptierungsrate*. Dann ist das* adaptive Auswahlgewicht *des Hardware-Moduls $m \in M$ wie folgt definiert:*

$$w_{sel}(m) = \frac{1}{1+\eta \cdot N_{sel}} \left(\frac{1}{|M|} + \eta \cdot N_{sel}(m) \right) \tag{5.6}$$

Mithilfe der Adaptierungsrate η lässt sich das Verhalten des adaptiven Auswahlgewichts beeinflussen. Wenn $\eta = 0$ ist, dann ist das adaptive Auswahlgewicht $\forall m \in M : w_{sel}(m) = 1/|M|$ und entspricht damit der im SUP-Fit-Verfahren getroffenen Annahme, dass eine Gleichverteilung der Modulauswahl vorliegt. Bei $\eta > 0$ sind die Werte des Auswahlgewichts zu Beginn ($N_{sel} = 0, \forall m \in M : N_{sel}(m) = 0$) mit $\forall m \in M : w_{sel}(m) = 1/|M|$ identisch. Mit jeder neuen Modulplatzierung verändert sich der Wert von $w_{sel}(m)$, so dass für eine große Anzahl an Platzierungsanfragen $w_{sel}(m) \approx (\eta \cdot N_{sel}(m))/(1+\eta \cdot N_{sel})$ ist und damit etwa dem Wert der relativen Häufigkeit entspricht. Mithilfe der Adaptierungsrate η kann daher festgelegt werden, wie schnell sich das adaptive Auswahlgewicht der relativen Häufigkeit annähert.

Anstelle der statischen Positionsüberlappung wird bei dem RUP-Fit-Verfahren die *dynamische Positionsüberlappung* verwendet, welche wie folgt definiert ist:

Definition 5.7 (Dynamische Positionsüberlappung)**.** *Gegeben sei die Menge der Hardware-Module M und die entsprechenden Mengen der freien Positionen $X_{free}(m) \subseteq X_{pos}(m)$ für eine heterogene rekonfigurierbare Architektur mit N_{col} Spalten und N_{row} Reihen. Die* dynamische Positionsüberlappung *$o_{pos,dyn}(m,x_h,x_v)$ entspricht der Anzahl der freien Position $|X_{free}(m)|$ des Hardware-Moduls $m \in M$, welche infolge einer Modulplatzierung die Belegung der Zelle an Position (x_h,x_v) verursachen würde, so dass*

$$\begin{aligned} &o_{pos,dyn}(m,x_h,x_v) = |X_{free}(m,x_h,x_v)|, \text{ wobei} \\ &X_{free}(m,x_h,x_v) = \big\{(i,j) \mid (i,j) \in X_{free}(m) \wedge i \le x_h < i + a_h(m) \wedge \\ &\qquad\qquad j \le x_v < j + a_v(m)\big\}. \end{aligned} \tag{5.7}$$

Die dynamische Zellüberlappung wird daher anhand der tatsächlich verfügbaren freien Positionen $X_{free}(m)$ des Hardware-Moduls m hergeleitet und ändert sich in Abhängigkeit der aktuellen Zellbelegung.

Während beim SUP-Fit-Verfahren die statischen Zellgewichte als Grundlage zur Berechnung der Positionsgewichte genutzt werden, dienen beim RUP-Fit-Verfahren die *dynamischen Zellgewichte* als Grundlage zur Berechnung der Positionsgewichte.

Definition 5.8 (Dynamisches Zellgewicht)**.** *Sei $X_{free}(m)$ die Menge der freien Positionen eines Hardware-Moduls $m \in M$ und $o_{pos,dyn}(m,x_h,x_v)$ die entsprechende dynamische Positionsüberlappung. Gegeben sei das adaptive Auswahlgewicht $w_{sel}(m)$ der einzelnen Hardware-Module. Dann ist das* dynamische Zellgewicht *$w_{cell,dyn}(x_h,x_v)$ wie folgt definiert:*

$$w_{cell,dyn}(x_h,x_v) = \sum_{m \in M_{free}} \frac{w_{sel}(m) \cdot o_{pos,dyn}(m,x_h,x_v)}{|X_{free}(m)|} \text{ , wobei}$$
$$M_{free} = \left\{ m \mid m \in M \wedge X_{free}(m) \neq \{\} \right\} \tag{5.8}$$

Das *dynamische Zellgewicht* $w_{cell,dyn}(x_h,x_v)$ lässt sich als die Wahrscheinlichkeit, mit der die Zelle an Position (x_h,x_v) unter Berücksichtigung der aktuellen Zellbelegung bei der Platzierung eines Hardware-Moduls belegt wird, interpretieren. Wenn eine Zelle an Position (x_h,x_v) das dynamische Zellgewicht $w_{cell,dyn}(x_h,x_v) = 0$ besitzt, dann wird sie entweder von einer bereits platzierten Modulinstanz genutzt, oder es existiert keine mögliche Position der Hardware-Module, die eine Belegung der Zelle verursachen würde. In [E5] wird beschrieben, wie zusätzlich Modul-Prioritäten bei der Berechnung des dynamischen Zellgewichts $w_{cell,dyn}(x_h,x_v)$ mit einbezogen werden. Auf diese Weise kann die Modulplatzierung dahingehend beeinflusst werden, dass Bereiche von möglichen Positionen hoch priorisierter Hardware-Module bei der Platzierung gemieden werden.

Analog zum statischen Positionsgewicht wird das *dynamische Positionsgewicht* einer freien Position ebenfalls anhand des quadratischen Mittelwerts der dynamischen Zellgewichte der Zellen, die infolge einer Platzierung belegt werden würden, bestimmt.

Definition 5.9 (Dynamisches Positionsgewicht)**.** *Gegeben sei das dynamische Zellgewicht $w_{cell,dyn}(x_h,x_v)$ einer heterogenen rekonfigurierbaren Architektur. Das* dynamische Positionsgewicht *$w_{pos,dyn}(m,x_{ph},x_{pv})$ einer freien Position $(x_{ph},x_{pv}) \in X_{free}(m)$ des Hardware-Moduls $m \in M$ stellt den quadratischen Mittelwert der dynamischen Zellgewichte der infolge einer Modulplatzierung belegten Zellen dar und ist wie folgt definiert:*

$$w_{pos,dyn}(m,x_{ph},x_{pv}) = \sqrt{\frac{1}{a_h(m) \cdot a_v(m)} \sum_{x_h,x_v} w_{cell,dyn}(x_h,x_v)^2} \text{ , wobei}$$
$$x_h \in [x_{ph}, x_{ph} + a_h(m) - 1] \text{ , } \quad x_v \in [x_{pv}, x_{pv} + a_v(m) - 1] \tag{5.9}$$

Ein gefordertes Hardware-Modul wird beim RUP-Fit-Verfahren ebenfalls an der freien Position mit dem geringsten dynamischen Positionsgewicht platziert. Im Gegensatz zum SUP-Fit-Verfahren weist das RUP-Fit-Verfahren keine Ähnlichkeit zum

First-Fit-Verfahren auf, da die Platzierung nicht mehr anhand einer festen Reihenfolge der möglichen Positionen geschieht. Beim RUP-Fit-Verfahren ändern sich die Positionsgewichte in Abhängigkeit der Zellbelegung. Das erfordert die Berechnung der dynamischen Positionsüberlappung und der resultierenden dynamischen Zellgewichte der einzelnen Zellen zur Laufzeit, so dass eine höhere Laufzeit als beim SUP-Fit-Verfahren benötigt wird.

Theorem 5.3. *Im 2D-Systemansatz hat das RUP-Fit-Verfahren eine Laufzeit von* $\mathcal{O}(|M| \cdot {N_{col}}^2 \cdot {N_{row}}^2)$.

Beweis. Ein gefordertes Hardware-Modul wird beim RUP-Fit-Verfahren an der freien Position mit dem geringsten dynamischen Positionsgewicht platziert. Die Berechnung eines Positionsgewichts erfordert dabei $a_h(m) \cdot a_v(m)$ Schritte und wird maximal für jede mögliche Position durchgeführt, so dass insgesamt $|X_{pos}(m)| \cdot a_h(m) \cdot a_v(m)$ Schritte benötigt werden. Die maximale Anzahl an Schritten ist dann erreicht, wenn die Menge der möglichen Positionen aus (5.5) verwendet wird und für die Modulfläche die Annahme gilt, dass $a_h(m) = \lceil N_{col}/2 \rceil$ und $a_v(m) = \lceil N_{row}/2 \rceil$. Die maximale Anzahl der Schritte zur Berechnung der dynamischen Positionsgewichte ist somit:

$$\left(\left\lfloor \frac{N_{col}}{2} \right\rfloor + 1\right) \cdot \left\lceil \frac{N_{col}}{2} \right\rceil \cdot \left(\left\lfloor \frac{N_{row}}{2} \right\rfloor + 1\right) \cdot \left\lceil \frac{N_{row}}{2} \right\rceil$$

Die Berechnung der dynamischen Zellgewichte der einzelnen Zellen in (5.8) erfordert maximal $|M| \cdot N_{col} \cdot N_{row}$ Schritte. Entscheidend für die Laufzeit des RUP-Fit-Verfahrens ist die Berechnung der dynamischen Positionsüberlappung, bei der für jedes Hardware-Modul und jede Zelle die Anzahl der freien Positionen ermittelt werden muss.

Bei der Berechnung von $o_{pos,dyn}(m, x_h, x_v)$ wird für jede freie Position eines Hardware-Moduls $m \in M$ der Wert von $a_h(m) \cdot a_v(m)$ Zellen um 1 inkrementiert, so dass für $X_{free} = X_{pos}$ genau $|X_{pos}(m)| \cdot a_h(m) \cdot a_v(m)$ Schritte benötigt werden, was der Anzahl der Schritte zur Berechnung der dynamischen Positionsgewichte entspricht. Für alle Module in M ergeben sich somit $|M| \cdot |X_{pos}(m)| \cdot a_h(m) \cdot a_v(m)$ Schritte. Die maximale Anzahl der Schritte zur Berechnung der dynamischen Positionsüberlappung ist somit:

$$|M| \cdot \left(\left\lfloor \frac{N_{col}}{2} \right\rfloor + 1\right) \cdot \left\lceil \frac{N_{col}}{2} \right\rceil \cdot \left(\left\lfloor \frac{N_{row}}{2} \right\rfloor + 1\right) \cdot \left\lceil \frac{N_{row}}{2} \right\rceil$$

Das entspricht einer Laufzeit von $\mathcal{O}(|M| \cdot {N_{col}}^2 \cdot {N_{row}}^2)$. □

Vergleicht man die Laufzeit des SUP-Fit-Verfahrens aus Theorem 5.2 mit der des RUP-Fit-Verfahrens aus Theorem 5.3, so zeigen sich deutliche Unterschiede. Während die Laufzeit des SUP-Fit-Verfahrens unabhängig von der Anzahl der Hardware-Module $|M|$ ist und linear mit der Anzahl der Zellen skaliert, wächst die Laufzeit des

RUP-Fit-Verfahrens quadratisch mit der Anzahl der Zellen und linear mit der Anzahl der Hardware-Module. Bezüglich des 1D-Systemansatzes ergeben sich jedoch bei beiden Verfahren einige Vereinfachungen, die im folgenden Abschnitt erläutert werden.

5.1.4 Vereinfachungen im 1D-Systemansatz

Wie schon in Abschnitt 4.1.3 im Zusammenhang mit homogenen rekonfigurierbaren Architekturen gezeigt, ergeben sich ebenso bei heterogenen rekonfigurierbaren Architekturen einige Vereinfachungen, wenn man die zuvor beschriebenen Platzierungsverfahren auf den 1D-Systemansatz überträgt. Die in Abschnitt 5.1.1 beschriebenen für heterogene rekonfigurierbare Architekturen geeigneten Varianten des First-Fit- und Best-Fit-Verfahrens können direkt durch die in Abschnitt 4.2.3 beschriebenen Algorithmen umgesetzt werden. Die in Algorithmus 4.3 dargestellte Umsetzung des First-Fit-Verfahrens, sowie die in Algorithmus 4.5 beschriebene Umsetzung des Best-Fit-Verfahrens sind damit sowohl für homogene, als auch für heterogene rekonfigurierbare Architekturen geeignet. Beide Verfahren haben eine Laufzeit, die linear von der Anzahl der Spalten N_{col} abhängt.

Da im 1D-Systemansatz immer komplette Spalten belegt werden, variieren die statische Positionsüberlappung und die statischen Zellgewichte auch nur spaltenweise, so dass die resultierenden statischen Positionsgewichte nur von der horizontalen Position abhängen. Zur Laufzeit kann mithilfe des rechtsseitigen Zellzusammenhangs in einem Schritt festgestellt werden, ob die gewählte mögliche Position frei ist. Da das SUP-Fit-Verfahren zur Laufzeit mit dem First-Fit-Verfahren vergleichbar ist, kann das SUP-Fit-Verfahren ebenfalls mithilfe von Algorithmus 4.3 umgesetzt werden, so dass sich eine Laufzeit ergibt, die linear mit der Anzahl der Spalten N_{col} skaliert.

Beim RUP-Fit-Verfahren ist die Platzierung in drei Teile gegliedert. Zunächst wird die Bestimmung der dynamischen Positionsüberlappungen der einzelnen Hardware-Module vorgenommen, aus denen dann die dynamischen Zellgewichte hergeleitet werden. Mithilfe der Zellgewichte werden dann die Positionsgewichte ermittelt, anhand derer die Position des geforderten Moduls bestimmt wird. Wie bereits gezeigt, wird die Laufzeit des RUP-Fit-Verfahrens durch die Bestimmung der dynamischen Positionsüberlappung geprägt. Im 2D-Systemansatz werden dabei $|M| \cdot |X_{pos}(m)| \cdot a_h(m) \cdot a_v(m)$ Schritte benötigt. Durch die Einschränkungen der in der Platzierung werden im 1D-Systemansatz nur $|M| \cdot |X_{pos}(m)| \cdot a_h(m)$ Schritte zur Bestimmung der dynamischen Positionsüberlappung benötigt. Die maximale Anzahl der Schritte beträgt somit:

$$|M| \cdot \left(\left\lfloor \frac{N_{col}}{2} \right\rfloor + 1 \right) \cdot \left\lceil \frac{N_{col}}{2} \right\rceil$$

Daher ergibt sich eine Laufzeit von $\mathcal{O}(|M| \cdot {N_{col}}^2)$ für das RUP-Fit-Verfahren im 1D-Systemansatz. Im folgenden Abschnitt werden die einzelnen Platzierungsverfahren sowohl im 1D-Systemansatz als auch im 2D-Systemansatz anhand von Simulationen miteinander verglichen.

5.2 Simulative Analyse

In diesem Abschnitt werden die zuvor vorgestellten Platzierungsverfahren für heterogene rekonfigurierbare Architekturen mithilfe der in Abschnitt 3.3.1 beschriebenen Simulationsumgebung SARA miteinander verglichen. Es wurden Simulationen sowohl im 1D-Systemansatz als auch im 2D-Systemansatz mit dem First-Fit-, Best-Fit-, SUP-Fit- und RUP-Fit-Verfahren durchgeführt. Anhand der Simulationen wurde analysiert, welches der gegebenen Verfahren die höchste Leistungsfähigkeit im Zusammenhang mit heterogenen rekonfigurierbaren Architekturen hat. Im Vordergrund steht hierbei der direkte Vergleich des First-Fit-Verfahrens mit dem SUP-Fit-Verfahren. Beide Verfahren verwenden zur Laufzeit das gleiche Prinzip, d. h., das geforderte Hardware-Modul wird an der ersten gefundenen freien Position platziert. Lediglich die Suchreihenfolge der möglichen Positionen variiert in Abhängigkeit des Verfahrens. Beim First-Fit wird die erste freie Position in einer vorgegebenen Richtung (z. B. von links nach rechts) gesucht, wobei beim SUP-Fit die erste freie Position mit dem geringsten statischen Positionsgewicht gesucht wird. Im direkten Vergleich beider Verfahren lässt sich erkennen, welchen Einfluss die Suchreihenfolge auf die Platzierung hat.

Als Grundlage der Simulationen dient erneut eine hypothetische Anwendung, die aus zufällig angeforderten Systemkomponenten besteht. Dabei werden die bereits im Zusammenhang mit homogenen rekonfigurierbaren Architekturen verwendeten Systemkomponenten aus Tabelle 4.1 verwendet. Zusätzlich wird angenommen, dass die Systemkomponenten *Rijndael Encryption*, *Ethernet Switch* und *32-bit RISC-CPU* neben Logikzellen auch Speicherblöcke (BlockRAM) verwenden.

Als rekonfigurierbare Architekturen dienen erneut die Xilinx FPGAs XC2V2000, XC2V4000 und XC2V6000. Im Vergleich zu den Simulationen aus Abschnitt 4.3.2 werden diesmal Heterogenitäten durch Speicherblöcke und Verbindungsstrukturen berücksichtigt, so dass die möglichen Positionen der Hardware-Module ungleichmäßig verteilt sind. Die in Kalte et al. [E1] vorgestellte Kommunikationsinfrastruktur ermöglicht eine Platzierung von Hardware-Modulen in Abständen von 4 Spalten. Daher wird bei den Simulationen ebenfalls angenommen, dass sich aufgrund der Kommunikationsinfrastruktur Hardware-Module nur alle 4 Spalten platzieren lassen. Die Speicherblöcke (BlockRAM) eines Xilinx Virtex-II FPGAs sind in gesonderten Spalten angeordnet, wobei ein Speicherblock eine Höhe von 4 Logikzellen hat. Da die Modellierung nur gleichgroße Zellen zugrunde legt, wird ein Speicherblock in 4 einzelne Teilzellen unterteilt, um die Granularität der Logikzellen zu erreichen.

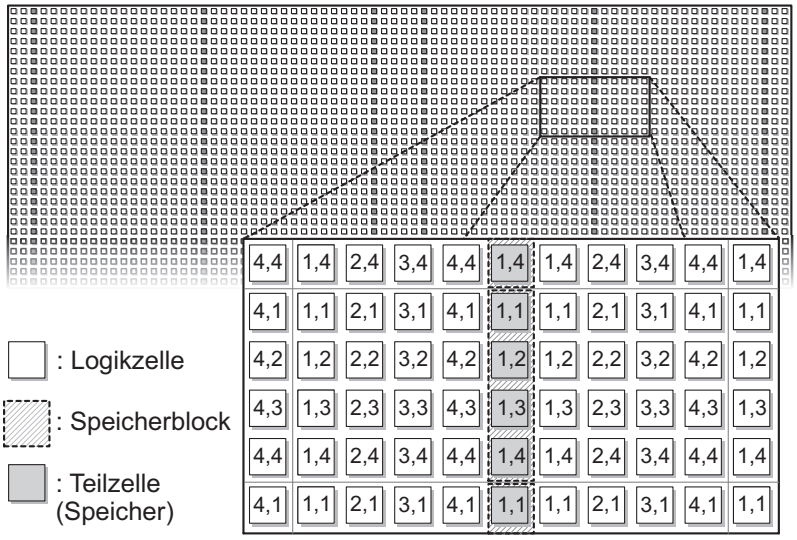


Abbildung 5.6: Zellanordnung des Xilinx Virtex-II XC2V4000 FPGAs.

Bei Belegung eines Speicherblocks wird automatisch die Belegung aller 4 Teilzellen angenommen. Auf diese Weise lässt sich die ursprüngliche Granularität der Speicherblöcke wiederherstellen.

Unter Berücksichtigung der durch die Speicherblöcke und die Kommunikationsinfrastruktur hervorgerufenen Heterogenität lassen sich die möglichen Positionen eines Hardware-Moduls wie folgt bestimmen. In Abbildung 5.6 sind die Zellen eines Xilinx Virtex-II XC2V4000 FPGAs dargestellt. Das FPGA besitzt insgesamt 6 BlockRAM-Spalten, die entsprechend der Abbildung verteilt sind. Die durch die Kommunikationsinfrastruktur vorgegebene Rasterung wird dadurch erzeugt, dass ein alle 4 Spalten wiederholendes Muster vorliegt. Ebenso wird ein Muster in vertikaler Richtung generiert, um die durch die BlockRAM-Zellen erzeugte Rasterung zu erhalten. Das resultierende Muster ist in Abbildung 5.6 angedeutet und dient zur Ermittlung der möglichen Positionen. Für jedes Hardware-Modul ergibt sich nun ein Muster, welches von der Position der vorsynthetisierten Implementierung, aus der das Hardware-Modul erzeugt wurde, abhängt. Die möglichen Positionen des Hardware-Moduls entsprechen den Positionen, in denen das gleiche Muster erneut auftaucht.

Wie bereits in den Simulationen aus Abschnitt 4.2 werden im 2D-Systemansatz zu jeder Systemkomponente drei verschiedene Modulvarianten mit den Seitenverhältnissen (1×2, 1×1 und 2×1) erzeugt. Im 1D-Systemansatz existiert zu jeder Systemkomponente nur ein Hardware-Modul. Mit der zuvor beschriebenen Methode ergibt sich bei Verwendung eines Xilinx XC2V4000 FPGAs für jedes Hardware-Modul die in Tabelle 5.1 abgebildete Anzahl der möglichen Positionen. Vergleicht man die Anzahl der möglichen Positionen mit der Anzahl der möglichen Positionen in Zusammenhang mit homogenen rekonfigurierbaren Architekturen (vgl. Tabelle 4.3), zeigt

Systemkomponente	Anzahl der möglichen Positionen für XC2V4000				
	2D (1×2)	2D (1×1)	2D (2×1)	2D (ges.)	1D
FIR-Filter	216	144	152	512	18
32-bit Divider	128	68	72	268	17
Digital Controller	120	64	54	238	17
Rijndael Encr.	48	28	32	108	4
3D-Graphic Accel.	27	26	15	68	8
Ethernet Switch	16	24	14	54	4
32-bit RISC CPU	14	11	14	39	4

Tabelle 5.1: Beispiel für die Anzahl der möglichen Positionen der Hardware-Module für den 1D- und 2D-Systemansatz bei Verwendung eines Xilinx XC2V4000 FPGAs.

sich, dass auch im 2D-Systemansatz die möglichen Positionen stark eingeschränkt sind. Ebenso sind die Unterschiede der Anzahl der möglichen Positionen im 1D- und 2D-Systemansatz nicht mehr so groß wie bei homogenen rekonfigurierbaren Architekturen.

In den Simulationen werden die Anwendungsklassen G und H betrachtet, die auf den Anwendungsklassen B und E aus Tabelle 4.4 basieren. Die Simulationen beinhalten 500 Komponentenanfragen, so dass für beide Anwendungsklassen $N_{sim} = 500/p_{req}$ gilt. Die Anfragewahrscheinlichkeiten der Anwendungsklassen G und H sind in Tabelle 5.2 abgebildet.

Anw.-Klasse	Parameter	XC2V2000	XC2V4000	XC2V6000
G	N_{sim}	3,33E+06	1,67E+06	1,11E+06
	p_{req}	1,50E-04	3,00E-04	4,50E-04
	$p_{sel}(d)$	$\propto 1/N_{res}(d)$		
	$t_{EXE}(d_i)$	$1\text{E-}04 \cdot N_{res}(d_i)$		
H	N_{sim}	6,67E+06	3,33E+06	2,00E+06
	p_{req}	7,50E-05	1,50E-04	2,50E-04
	$p_{sel}(d)$	$= 1/\lvert D \rvert$		
	$t_{EXE}(d_i)$	$5\text{E-}05 \cdot N_{res}(d_i)$		

Tabelle 5.2: Parameter der Anwendungsklassen für heterogene Architekturen.

Wie in den in Abschnitt 4.3.2 durchgeführten Simulationen werden fehlgeschlagene Modulplatzierungen abgewiesen. Zusätzlich sind jedoch auch Simulationen durchgeführt worden, bei denen fehlgeschlagene Modulplatzierungen nicht abgewiesen werden, sondern zu einem späteren Zeitpunkt erneut durchgeführt werden. Die Modulplatzierung und alle folgenden werden dabei so lange verzögert, bis durch das Zurücksetzen abgelaufener Modulinstanzen hinreichend viele Ressourcen frei werden, um die Platzierung zu ermöglichen. Bei der Platzierungsablaufplanung wird das FCFS-Verfahren zugrunde gelegt, so dass die Reihenfolge der Modulplatzierungen

nicht verändert wird. Wenn in der Zeit, in der eine Modulplatzierung verzögert wird, weitere Modulplatzierungen gefordert werden, werden diese ebenfalls verzögert. Daher entsteht eine Platzierungswarteschlange.

Da bei einer derartigen Handhabung fehlgeschlagener Modulplatzierungen keine Hardware-Module abgewiesen werden, kann die Metrik der Zellabweisung nicht angewendet werden. Eine vergleichbare Metrik ist die durchschnittliche Anzahl der Komponentenanfragen, die in der Platzierungswarteschlange sind, die im Folgenden als n_{delay} bezeichnet wird. Im Idealfall ist die Anzahl der Komponentenanfragen in der Platzierungswarteschlange immer 0, d. h., alle Komponentenanfragen führten unmittelbar zu einer erfolgreichen Modulplatzierung. Um die Leistungsfähigkeit der Platzierungsverfahren zu testen, sind die Simulationen derart ausgelegt, dass es zu verzögerten Modulplatzierungen kommen kann. Ziel des Platzierungsverfahrens ist es, den Wert von n_{delay} so gering wie möglich zu halten.

Im folgenden Abschnitt wird zunächst auf die Ergebnisse der Simulation im allgemeinen 2D-Systemansatz eingegangen.

5.2.1 2D-Systemansatz

Die in diesem Abschnitt betrachteten Simulationen wurden im 2D-Systemansatz durchgeführt, wobei die Platzierungszeiten und die Konfigurationszeiten der Modulinstanzen zunächst nicht berücksichtigt wurden. In den Simulationen wurden die Platzierungsverfahren First-Fit, Best-Fit, SUP-Fit und RUP-Fit für heterogene rekonfigurierbare Architekturen miteinander verglichen. Beim RUP-Fit-Verfahren wurde eine Adaptierungsrate von $\eta = 0,05$ angenommen. D. h., nach 20 Platzierungsanfragen entsprechen die adaptiven Auswahlgewichte (vgl (5.6)) der einzelnen Hardware-Module dem Mittelwert zwischen der relativen Häufigkeit der einzelnen Module und dem Wert $1/|M|$ ($\widehat{=}$ Wahrscheinlichkeit bei Gleichverteilung). Für die Anwendungsklassen G und H wurden für jedes FPGA jeweils 20 verschiedene *virtuelle Anwendungen* in Form von Listen mit 500 Komponentenanfragen erzeugt. Die einzelnen Platzierungsverfahren wurden dann mit jeder Liste getestet. Zunächst werden die Simulationen betrachtet, die bei fehlgeschlagener Modulplatzierung die geforderte Systemkomponente abweisen. In Tabelle 5.3 ist die mittlere Ressourcenauslastung aller Simulationen der einzelnen Anwendungsklassen abgebildet.

In der Mehrzahl der Simulationen brachte das RUP-Fit-Verfahren die größte Ressourcenauslastung hervor. Die Unterschiede in der mittleren Ressourcenauslastung der einzelnen Platzierungsverfahren nehmen mit der Größe des FPGAs zu. Die geringsten Abweichungen der mittleren Ressourcenauslastung ergaben sich beim XC2V2000 FPGA und der Anwendungsklasse H. Zwischen dem kleinsten Mittelwert (First-Fit-Verfahren: $27,91 \pm 1,36\%$) und dem größten Mittelwert (SUP-Fit-Verfahren: $28,96 \pm 1,24\%$) ergibt sich eine Differenz von nur $1,05\%$. Beim XC2V6000 FPGA in der Anwendungsklasse G hingegen entstanden die größten Unterschiede. Beim First-Fit-Verfahren wurde eine mittlere Ressourcenauslastung von

FPGA	Anw.-Klasse	Mittlere Ressourcenauslastung [%]			
		First-Fit	Best-Fit	SUP-Fit	RUP-Fit
XC2V2000	G	28,27 ±2,48	27,74 ±1,97	**30,65 ±2,28**	30,36 ±2,51
	H	27,91 ±1,36	28,10 ±1,38	**28,96 ±1,24**	28,80 ±1,25
XC2V4000	G	30,99 ±2,17	33,58 ±1,80	34,70 ±2,60	**35,93 ±2,58**
	H	30,50 ±1,06	32,83 ±1,15	34,07 ±1,30	**34,22 ±1,31**
XC2V6000	G	33,78 ±2,64	37,82 ±2,52	38,55 ±2,97	**41,17 ±3,15**
	H	36,95 ±1,18	40,10 ±1,11	41,27 ±1,28	**42,30 ±1,05**

Tabelle 5.3: Mittlere Ressourcenauslastung der Simulationen der Anwendungsklassen G und H mit Modulabweisung bei fehlgeschlagener Platzierung.

FPGA	Anw.-Klasse	Mittlere Zellabweisung [%]			
		First-Fit	Best-Fit	SUP-Fit	RUP-Fit
XC2V2000	G	28,65 ±2,81	29,05 ±3,39	**26,30 ±3,29**	26,39 ±3,33
	H	36,76 ±1,84	37,07 ±1,89	**35,53 ±1,64**	36,23 ±1,78
XC2V4000	G	19,77 ±4,03	16,92 ±4,11	16,17 ±4,30	**14,55 ±3,82**
	H	28,79 ±2,25	25,21 ±2,00	23,74 ±2,03	**23,40 ±2,07**
XC2V6000	G	18,19 ±3,05	14,25 ±3,19	14,70 ±2,73	**11,43 ±2,89**
	H	27,70 ±2,25	24,00 ±2,24	23,91 ±2,19	**21,72 ±2,50**

Tabelle 5.4: Mittlere Zellabweisung der Simulationen der Anwendungsklassen G und H mit Modulabweisung bei fehlgeschlagener Platzierung.

$33,78 \pm 2,64\%$ und beim RUP-Fit-Verfahren eine mittlere Ressourcenauslastung von $41,17 \pm 3,15\%$ erzielt, was eine Differenz von $7,39\%$ ergibt. Ein Grund, dass die Unterschiede der Ressourcenauslastung mit der Größe zunehmen, liegt in der zunehmenden Anzahl der möglichen Positionen. Insbesondere beim XC2V2000 ist die Anzahl der möglichen Positionen derart eingeschränkt, dass z. B. insgesamt nur 6 mögliche Positionen aller Hardware-Module der größten Systemkomponente *32-bit RISC-CPU* existieren. Daher ist der Einfluss des Platzierungsverfahrens entsprechend gering. Die geringe Anzahl der möglichen Positionen spiegelt sich ebenfalls in der hohen mittleren Zellabweisung wieder.

Die Ergebnisse der Analyse der mittleren Zellabweisung sind in Tabelle 5.4 dargestellt und bestätigen die Ergebnisse aus der Analyse der Ressourcenauslastung. Auch hier zeigt sich, dass im direkten Vergleich das Platzierungsverfahren RUP-Fit über-

FPGA	Anw.-Klasse	Mittlere Ressourcenauslastung [%]			
		First-Fit	Best-Fit	SUP-Fit	RUP-Fit
XC2V2000	G	50,89 ±3,40	51,20 ±3,48	**51,46 ±3,64**	51,41 ±3,43
	H	48,89 ±1,79	**49,36 ±1,81**	49,17 ±1,79	49,31 ±1,75
XC2V4000	G	45,30 ±4,04	46,25 ±4,30	**46,82 ±4,64**	46,74 ±4,54
	H	46,65 ±2,09	47,72 ±2,21	**48,22 ±2,21**	47,97 ±2,26
XC2V6000	G	47,58 ±3,85	49,65 ±4,51	50,58 ±4,90	**50,68 ±4,71**
	H	50,38 ±1,27	54,69 ±1,79	**56,59 ±1,89**	56,01 ±1,80

Tabelle 5.5: Mittlere Ressourcenauslastung der Simulationen der Anwendungsklassen G und H mit Platzierungsverzögerung bei fehlgeschlagener Platzierung.

wiegend die geringste Zellabweisung hervorbrachte. Die Unterschiede in der mittleren Zellabweisung der einzelnen Platzierungsverfahren nehmen auch mit der Größe des FPGAs zu. Die größten Unterschiede in der mittleren Zellabweisung sind erneut bei den Simulationen mit dem XC2V6000 FPGA zu beobachten.

Wie zuvor erwähnt, weisen das First-Fit- und das SUP-Fit-Verfahren das gleiche Laufzeitverhalten auf, d. h., Hardware-Module werden an der ersten gefundenen freien Position platziert. Die Verfahren unterscheiden sich lediglich in der Reihenfolge, in der die möglichen Positionen der Hardware-Module zur Laufzeit durchsucht werden. Vergleicht man die Ressourcenauslastungen und die Zellabweisungen der beiden Platzierungsverfahren, zeigt sich ferner, dass das SUP-Fit-Verfahren im Vergleich zum First-Fit-Verfahren im Mittel eine bessere Ressourcenauslastung und eine geringere mittlere Zellabweisung hervorbringt. Die durchgeführten Simulationen belegen, dass die Suchreihenfolge der möglichen Position der einzelnen Hardware-Module einen großen Einfluss auf die Leistungsfähigkeit eines Platzierungsverfahrens hat.

In Tabelle 5.5 ist die mittlere Ressourcenauslastung der Simulationen dargestellt, die keine Abweisungen zulassen und bei fehlgeschlagener Modulplatzierung die entsprechenden Platzierungen verzögern. In der Mehrzahl der Simulationen hat das SUP-Fit-Verfahren die größte mittlere Ressourcenauslastung hervorgerufen. Im Vergleich zu der Ressourcenauslastung bei den Platzierungsverfahren mit Modulabweisung (Tabelle 5.3) ist die mittlere Ressourcenauslastung bei den Platzierungsverfahren mit Modulverzögerung größer. Dadurch, dass keine Hardware-Module abgewiesen, sondern solange verzögert werden, bis hinreichend viele freie Ressourcen verfügbar sind, werden mehr Hardware-Module in der gleichen Zeit platziert. Auf diese Weise ergibt sich die erkennbare Differenz in der mittleren Ressourcenauslastung im Vergleich zu Tabelle 5.3.

FPGA	Anw.-Klasse	Mittlere Anz. d. Platzierungsverzög. (n_{delay})			
		First-Fit	Best-Fit	SUP-Fit	RUP-Fit
XC2V2000	G	15,70 ±12,31	16,25 ±12,16	**14,84 ±11,44**	16,78 ±13,71
	H	**7,04 ±4,47**	7,95 ±4,91	7,15 ±4,69	9,07 ±6,23
XC2V4000	G	12,68 ±13,16	9,51 ±10,65	8,99 ±10,28	**8,27 ±9,32**
	H	8,19 ±4,29	6,07 ±3,17	**5,60 ±3,04**	6,10 ±2,93
XC2V6000	G	14,64 ±14,23	9,24 ±8,65	9,81 ±8,19	**8,84 ±8,87**
	H	26,80 ±13,37	18,66 ±10,31	**15,10 ±8,61**	15,71 ±9,18

Tabelle 5.6: Mittelwert der Anzahl der Komponentenanfragen in der Platzierungswarteschlange aller Simulationen der Anwendungsklassen G und H.

Die Unterschiede der mittleren Ressourcenauslastungen der einzelnen Platzierungsverfahren sind jedoch geringer. Lediglich beim XC2V6000 FPGA zeigt sich, dass sowohl das SUP-Fit-Verfahren als auch das RUP-Fit-Verfahren eine deutlich höhere mittlere Ressourcenauslastung als die des First-Fit-Verfahrens oder die des Best-Fit-Verfahrens hervorbrachte. Bei Betrachtung der Mittelwerte der Anzahl der Komponentenanfragen in der Platzierungswarteschlange in Tabelle 5.6 zeigt sich, dass in der Mehrzahl der Simulationen erneut das SUP-Fit-Verfahren den geringsten Wert verursachte. Im direkten Vergleich der Platzierungsverfahren First-Fit und SUP-Fit ist beim First-Fit-Verfahren der Mittelwert von n_{delay} lediglich beim XC2V2000 und Anwendungsklasse H mit $7,04 \pm 4,47$ etwas geringer als beim SUP-Fit-Verfahren mit $7,15 \pm 4,69$. Beim XC2V6000 und der Anwendungsklasse H zeigt sich jedoch ein großer Unterschied bezüglich der mittleren Anzahl der Komponentenanfragen in der Platzierungswarteschlange, denn das SUP-Fit-Verfahren liegt mit einem Mittelwert von $15,10 \pm 8,61$ deutlich unter dem Mittelwert des First-Fit-Verfahrens von $26,80 \pm 13,37$.

In den durchgeführten Simulationen im 2D-Systemansatz brachten die Platzierungsverfahren RUP-Fit und SUP-Fit in der Regel eine bessere Ressourcenauslastung als die für heterogene Architekturen angepassten Platzierungsverfahren First-Fit und Best-Fit hervor. Ob sich diese Ergebnisse auch im 1D-Systemansatz bestätigen, wird im folgenden Abschnitt diskutiert.

5.2.2 1D-Systemansatz

In den Simulationen im 1D-Systemansatz wurden sowohl die Platzierungszeit als auch die Konfigurationszeit der einzelnen Modulinstanzen berücksichtigt. Es wurden erneut die Platzierungsverfahren First-Fit, Best-Fit, SUP-Fit und RUP-Fit miteinan-

FPGA	Mittlere Platzierungszeit [s]			
	First-Fit	Best-Fit	SUP-Fit	RUP-Fit
XC2V2000	5,81E-06 ± 33,0E-09	9,43E-06 ± 40,0E-09	**5,58E-06 ± 20,0E-09**	21,57E-06 ± 1,25E-06
XC2V4000	8,38E-06 ± 61,0E-09	14,07E-06 ± 79,0E-09	**8,20E-06 ± 30,0E-09**	38,58E-06 ± 2,79E-06
XC2V6000	10,16E-06 ± 77,0E-09	16,87E-06 ± 91,0E-09	**9,88E-06 ± 45,0E-09**	40,20E-06 ± 4,13E-06

Tabelle 5.7: Mittelwert der Platzierungszeit aller Simulationen im 1D-Systemansatz der Anwendungsklasse H.

der verglichen. Wie bei den im vorherigen Abschnitt dargestellten Simulationen wurde beim RUP-Fit-Verfahren eine Adaptierungsrate von $\eta = 0,05$ angenommen. Die Platzierungszeiten des First-Fit- und Best-Fit-Verfahrens wurden anhand der in Abschnitt 4.2.3 beschriebenen Methode approximiert. Da das SUP-Fit-Verfahren das gleiche Laufzeitverhalten wie das First-Fit-Verfahren hat, lässt sich zur Abschätzung der Platzierungszeit des SUP-Fit-Verfahrens ebenfalls die in Abschnitt 4.2.3 beschriebene Approximation der Platzierungszeit des First-Fit-Verfahrens verwenden. Die Platzierungszeit des RUP-Fit-Verfahrens geschah ebenfalls anhand der Zyklenapproximation der einzelnen im Algorithmus verborgenen Iterationen. Um die Güte der Approximation zu verifizieren, wurde das RUP-Fit-Verfahren ebenfalls auf dem eingebetteten IBM PowerPC 405 des Xilinx Virtex-II Pro XC2VP20 implementiert. Wie im Abschnitt 4.2.3 beschrieben, wurde die Ausführungszeit des RUP-Fit-Verfahrens mithilfe des TBR-Registers des PowerPCs gemessen. Dabei wurden Platzierungen von unterschiedlich großen Hardware-Modulen und verschiedenen Zellbelegungen für einen XC2V4000 FPGA durchgeführt. Als Grundlagen dienten die in Tabelle 5.1 aufgeführten Hardware-Module für den 1D-Systemansatz.

Die durchschnittliche gemessene Ausführungszeit der PowerPC-Implementierung des RUP-Fit-Verfahrens betrug $42,44 \pm 5,59 \cdot 10^{-6}$ s. Die entsprechende durchschnittliche approximierte Ausführungszeit des RUP-Fit-Verfahrens ergab $42,40 \pm 5,58 \cdot 10^{-6}$ s. Der Mittelwert der Differenz zwischen Zyklenapproximation und Zyklenmessung lag bei $46,11 \pm 23,52 \cdot 10^{-9}$ s. Die Approximation der Ausführungszeit des RUP-Fit-Verfahrens entspricht damit bis auf eine geringe Abweichung in der Größenordnung von Nanosekunden der realen Ausführungszeit.

Es wurden erneut für die Anwendungsklassen G und H und für jedes FPGA jeweils 20 *virtuelle Anwendungen* mit je 500 Komponentenanfragen erzeugt. Im Folgenden werden die Simulationen betrachtet, die bei fehlgeschlagener Modulplatzierung die geforderte Systemkomponente abweisen. In Tabelle 5.7 sind die durchschnittlichen Platzierungszeiten der Anwendungsklasse H für die verschiedenen FPGAs abgebildet. Das First-Fit- und das SUP-Fit-Verfahren ermöglichen die schnellste Modulplatzierung, wobei im direkten Vergleich das SUP-Fit-Verfahren sogar eine geringfügig besser Platzierungszeit als das First-Fit-Verfahren aufweist.

FPGA	Anw.-Klasse	Mittlere Ressourcenauslastung [%]			
		First-Fit	Best-Fit	SUP-Fit	RUP-Fit
XC2V2000	G	27,29 ±2,81	28,29 ±2,39	**31,93 ±2,39**	31,90 ±2,58
	H	28,95 ±1,20	29,08 ±1,15	30,93 ±1,23	**30,99 ±1,26**
XC2V4000	G	34,94 ±3,07	36,46 ±3,23	35,44 ±3,14	**37,72 ±3,36**
	H	35,53 ±1,62	35,88 ±1,65	36,06 ±1,80	**36,51 ±1,71**
XC2V6000	G	33,46 ±2,40	36,05 ±2,86	37,53 ±3,11	**38,31 ±3,18**
	H	36,25 ±1,10	36,94 ±1,15	37,60 ±1,08	**37,86 ±1,10**

Tabelle 5.8: Mittlere Ressourcenauslastung der Simulationen der Anwendungsklassen G und H mit Modulabweisung bei fehlgeschlagener Platzierung.

FPGA	Anw.-Klasse	Mittlere Zellabweisung [%]			
		First-Fit	Best-Fit	SUP-Fit	RUP-Fit
XC2V2000	G	29,71 ±3,32	28,56 ±3,20	24,65 ±2,96	**24,59 ±2,78**
	H	34,53 ±2,06	34,33 ±2,03	31,73 ±1,80	**31,64 ±1,79**
XC2V4000	G	14,72 ±4,00	12,65 ±3,52	12,95 ±3,27	**10,95 ±3,43**
	H	17,56 ±1,50	16,90 ±1,50	16,63 ±1,43	**15,91 ±1,61**
XC2V6000	G	19,27 ±3,00	15,59 ±2,80	13,59 ±3,06	**12,74 ±2,85**
	H	26,99 ±2,05	25,80 ±2,18	24,12 ±2,24	**23,78 ±2,12**

Tabelle 5.9: Mittlere Zellabweisung der Simulationen der Anwendungsklassen G und H mit Modulabweisung bei fehlgeschlagener Platzierung.

Das Best-Fit-Verfahren benötigte im Mittel in etwa doppelt so viel Zeit wie das SUP-Fit-Verfahren. Das Platzierungsverfahren mit der längsten mittleren Platzierungszeit war das RUP-Fit-Verfahren, welches beim größten FPGA (XC2V6000) im Durchschnitt $40,20 \cdot 10^{-6} \pm 4,13 \cdot 10^{-6}$ *s* für die Platzierung eines Hardware-Moduls brauchte. Im Vergleich zur benötigten Konfigurationszeit der Hardware-Module (vgl. Abschnitt 4.2.4) ist die Platzierungszeit jedoch deutlich geringer und wirkt sich daher nicht entscheidend auf die Ressourcenauslastung aus.

In Tabelle 5.8 ist die mittlere Ressourcenauslastung der Simulationen mit Modulabweisung bei fehlgeschlagener Platzierung dargestellt. In den meisten durchgeführten Simulationen ergab das RUP-Fit-Verfahren die höchste Ressourcenauslastung. Im Vergleich zum 2D-Systemansatz sind die mittleren Ressourcenauslastungen der einzelnen Platzierungsverfahren nicht mehr so stark von der Größe des

FPGA	Anw.-Klasse	Mittlere Ressourcenauslastung [%]			
		First-Fit	Best-Fit	SUP-Fit	RUP-Fit
XC2V2000	G	51,18 ±4,09	51,19 ±4,11	51,26 ±4,28	**51,27 ±4,26**
	H	48,88 ±1,83	48,88 ±1,83	48,90 ±1,84	**48,90 ±1,84**
XC2V4000	G	45,58 ±5,45	45,58 ±5,44	45,56 ±5,49	**45,62 ±5,50**
	H	**45,10 ±2,13**	45,10 ±2,13	45,08 ±2,11	45,08 ±2,11
XC2V6000	G	45,86 ±3,92	46,03 ±3,98	**46,40 ±4,27**	46,38 ±4,28
	H	50,33 ±1,79	50,47 ±1,83	**51,04 ±1,99**	51,03 ±1,98

Tabelle 5.10: Mittlere Ressourcenauslastung der Simulationen der Anwendungsklassen G und H mit Platzierungsverzögerung bei fehlgeschlagener Platzierung.

FPGAs abhängig. Lediglich die mittleren Ressourcenauslastungen der Simulationen des XC2V2000 sind etwas geringer als die der anderen FPGAs. Die gleiche Tendenz ist auch bei der in Tabelle 5.9 abgebildeten mittleren Zellabweisung der Simulationen mit Modulabweisung bei fehlgeschlagener Platzierung zu beobachten. Das RUP-Fit-Verfahren verursachte in der Regel die geringste mittlere Zellabweisung. Im direkten Vergleich des SUP-Fit- und des First-Fit-Verfahrens lässt sich wie schon in den Simulationen des 2D-Systemansatzes erkennen, dass die Reihenfolge der möglichen Positionen bei der Platzierung eine entscheidende Rolle spielt. Auch hier verursachte das SUP-Fit-Verfahren eine geringere Zellabweisung als das First-Fit-Verfahren. Der größte Unterschied ergab sich bei den Simulationen des XC2V6000 und der Anwendungsklasse G, bei dem das First-Fit-Verfahren eine mittlere Zellabweisung von $19,27 \pm 3,00\%$ und das SUP-Fit-Verfahren eine mittlere Zellabweisung von $13,59 \pm 3,06\%$ verursachte.

Im Folgenden wird auf die Simulationen mit Platzierungsverzögerung bei fehlgeschlagener Modulplatzierung eingegangen. In Tabelle 5.10 ist die mittlere Ressourcenauslastung der Simulationen der Anwendungsklassen G und H dargestellt, bei denen die Abweichungen der mittleren Ressourcenauslastung der einzelnen Platzierungsverfahren sehr gering sind. Die größten Unterschiede des Mittelwerts ergaben sich bei den Simulationen mit dem XC2V6000 FPGA und der Anwendungsklasse H, bei der die mittlere Ressourcenauslastung des First-Fit-Verfahrens $50,33 \pm 1,79\%$ betrug und die des SUP-Fit-Verfahrens $51,04 \pm 1,99\%$, was eine Differenz von nur $0,71\%$ darstellt. Deutlicher vielen die Unterschiede der in Tabelle 5.11 abgebildeten mittleren Anzahl der Komponentenanfragen in der Platzierungswarteschlange aus. In der Mehrzahl der durchgeführten Simulationen verursachte das RUP-Fit-Verfahren die geringste mittlere Anzahl an Komponentenanfragen in der Plat-

FPGA	Anw.-Klasse	Mittlere Anz. d. Platzierungsverzög. (n_{delay})			
		First-Fit	Best-Fit	SUP-Fit	RUP-Fit
XC2V2000	G	9,83 ±8,58	9,71 ±8,44	**9,06 ±7,72**	9,06 ±7,69
	H	3,63 ±2,16	3,62 ±2,16	3,53 ±2,04	**3,52 ±2,04**
XC2V4000	G	2,21 ±1,56	1,98 ±1,28	2,03 ±1,38	**1,85 ±1,36**
	H	1,04 ±0,42	**0,99 ±0,39**	1,04 ±0,52	0,99 ±0,49
XC2V6000	G	11,29 ±10,47	10,31 ±10,01	8,38 ±8,55	**8,16 ±8,73**
	H	14,44 ±7,94	13,80 ±7,52	10,88 ±6,14	**10,76 ±6,10**

Tabelle 5.11: Mittelwert der Anzahl der Komponentenanfragen in der Platzierungswarteschlange aller Simulationen der Anwendungsklassen G und H.

zierungswarteschlange. Der größte Unterschied ergab sich bei den Simulationen mit dem XC2V6000 und der Anwendungsklasse H, bei denen der Mittelwert des RUP-Fit-Verfahrens bei $10,76 \pm 6,10$ und der Mittelwert des First-Fit-Verfahrens bei $14,44 \pm 7,94$ lag.

Bei den bisherigen Simulationen wurde bei einer fehlgeschlagenen Modulplatzierung die entsprechende Komponentenanfrage abgewiesen, oder die Platzierung wurde so lange verzögert, bis hinreichend viele freie Ressourcen zur erfolgreichen Modulplatzierung verfügbar waren. Ein alternativer Ansatz ist die in Abschnitt 4.3 beschriebene Verwendung von Defragmentierungsverfahren. Das Konzept der partiellen Kompaktierung ist jedoch nur für homogene rekonfigurierbare Architekturen geeignet, denn die Grundvoraussetzung für das Verfahren ist die beliebige Platzierbarkeit eines Hardware-Moduls. Bezüglich heterogener rekonfigurierbarer Architekturen ist diese Voraussetzung jedoch nicht gegeben, so dass gesonderte Verfahren verwendet werden müssen, die im folgenden Abschnitt beschrieben werden.

5.3 Defragmentierung

Im Zusammenhang mit homogenen rekonfigurierbaren Architekturen wurde anhand der Simulationsergebnisse in Abschnitt 4.2 gezeigt, dass das fortlaufende Platzieren und Entfernen von Modulinstanzen eine externe Fragmentierung verursacht. Um die Platzierung eines Hardware-Moduls auch bei hoher externer Fragmentierung zu ermöglichen, können in homogenen rekonfigurierbaren Architekturen Defragmentierungsverfahren wie die partielle Kompaktierung eingesetzt werden.

In heterogenen rekonfigurierbaren Architekturen ergibt sich mit der Zeit ebenfalls eine externe Fragmentierung, die mithilfe von Defragmentierungsverfahren verringert werden kann. Voraussetzungen zur Realisierung einer Defragmentierung sind

geeignete Methoden zur Umplatzierung von bereits platzierten Modulinstanzen zur Laufzeit unter Berücksichtigung der Erhaltung der internen Registerzustände. In [E7] ist ein Verfahren beschrieben, welches eine solche Umplatzierung von Hardware-Modulen zur Laufzeit ermöglicht. Bei dem Verfahren werden die Kontexte von Logikzellen und Speicherblöcken erhalten, so dass das Verfahren für den Einsatz in heterogenen rekonfigurierbaren Architekturen geeignet ist.

Wegen der unterschiedlichen und ungleichmäßig verteilten möglichen Positionen der Hardware-Module können in heterogenen rekonfigurierbaren Architekturen die entsprechenden Modulinstanzen nicht beliebig spaltenweise verschoben werden. Jedoch basieren die bisherigen Algorithmen (z. B. [25, 26, 79]) zur Bestimmung neuer Positionen der bestehenden Modulinstanzen, sowie das in Abschnitt 4.3.1 vorgestellte Konzept der partiellen Kompaktierung, auf der Annahme, dass die Modulinstanzen an beliebige Positionen umplatziert werden können. Die Algorithmen sind daher nicht ohne Weiteres für heterogene Architekturen geeignet.

Im folgenden Abschnitt wird ein Defragmentierungsverfahren beschrieben, welches für heterogene Architekturen im 1D-Systemansatz geeignet ist. Das Verfahren markiert nach heutigem Kenntnisstand damit den ersten Ansatz eines Defragmentierungsverfahrens in Zusammenhang mit heterogenen rekonfigurierbaren Architekturen.

5.3.1 Partielle Verdrängung

Das hier beschriebene heuristische Defragmentierungsverfahren, welches im Folgenden *partielle Verdrängung* genannt wird, wurde in [E7] vorgestellt und dient zur ereignisgesteuerten Defragmentierung im 1D-Systemansatz. D. h., das Verfahren versucht die geforderte Platzierung eines nicht ohne Weiteres platzierbaren Hardware-Moduls durch Umplatzieren der vorhandenen Modulinstanzen zu ermöglichen. Grund für das Fehlschlagen der Platzierung ist der Mangel an freien möglichen Positionen des geforderten Hardware-Moduls. Bei homogenen Architekturen ist die Modulplatzierung nach Defragmentierung genau dann möglich, wenn die Anzahl der freien Spalten mindestens der benötigten Anzahl der Spalten des Hardware-Moduls entspricht (vgl. (4.15)). Daher lässt sich anhand der Anzahl der freien Spalten einfach feststellen, ob eine Platzierung überhaupt möglich ist. Bei heterogenen rekonfigurierbaren Architekturen gilt diese Defragmentierungsbedingung nicht. Zwar muss die Anzahl der freien Spalten ebenfalls mindestens der Anzahl der benötigten Spalten des geforderten Hardware-Moduls entsprechen, jedoch lässt sich nicht ohne Weiteres bestimmen, ob die bereits platzierten Modulinstanzen derart umplatziert werden können, dass eine freie mögliche Position des geforderten Hardware-Moduls entsteht.

Auch bei Erfüllung der Bedingung (4.15) führt das hier beschriebene Verfahren der partiellen Verdrängung nicht notwendigerweise zu einer gültigen Lösung. Ebenso kann bei Existenz einer gültigen Lösung nicht garantiert werden, dass diese auch gefunden wird. Dennoch wird sich anhand der späteren Analysen zeigen, dass das

Verfahren in vielen Fällen eine Platzierung des geforderten Hardware-Moduls durch Umplatzieren der vorhanden Modulinstanzen ermöglicht. Das Verfahren berücksichtigt dabei den Aspekt der Minimierung der benötigten Umplatzierungszeit der Modulinstanzen, so dass das geforderte Hardware-Modul mit geringer Verzögerung platziert werden kann. Ein zentraler Bestandteil des Verfahrens ist die Verwendung des in Abschnitt 5.1.2 vorgestellten Platzierungsverfahrens SUP-Fit.

Bei der partiellen Verdrängung werden zunächst die Modulinstanzen ermittelt, die umplatziert werden müssen, um die Platzierung des geforderten Hardware-Moduls $m_{req} \in M$ zu ermöglichen. Dabei werden die folgenden Schritte, die ebenfalls in Algorithmus 5.3 dargestellt sind, virtuell durchgeführt:

1. Zunächst wird die i-te mögliche Position $(x_{ph}(m_{req}, i), 1) \in X_{pos}(m_{req})$ des geforderten Hardware-Moduls $m_{req} \in M$ ausgewählt. Sei $C_{use}(x_{ul}, x_{ur}) \subseteq C$ die Menge der Modulinstanzen, welche Zellen zwischen der Spalte an der horizontalen Position $x_{ul} \in [1, N_{col} - 1]$ und der Spalte an der horizontalen Position $x_{ur} \in [x_{ul}, N_{col}]$ verwenden. Die Menge C_{use} ist demnach wie folgt definiert:

$$\begin{aligned} C_{use}(x_{ul}, x_{ur}) &= \bigcup_{n=x_{ul}}^{x_{ur}} c(n) \text{ , wobei} \\ c(n) &= \{c \mid c \in C \wedge n \geq x_h(c) \wedge n < x_h(c) + a_h(m(c))\} \end{aligned} \tag{5.10}$$

   Dann entspricht

$$C_{int} = C_{use}(x_{ph}(m_{req}, i), x_{ph}(m_{req}, i) + a_h(m_{req}) - 1) \tag{5.11}$$

   der Menge der *überschneidenden Modulinstanzen*, welche Zellen verwenden, die infolge einer Platzierung des geforderten Hardware-Moduls $m_{req} \in M$ an der gewählten möglichen Position $(x_{ph}(m_{req}, i), 1) \in X_{pos}(m_{req})$ belegt werden würden.

2. Nach Bestimmung der Menge der überschneidenden Modulinstanzen C_{int} werden alle Modulinstanzen in C_{int} aus C entfernt ($C \leftarrow C \setminus C_{int}$). Da nun hinreichend viele freie Zellen für die Platzierung von m_{req} verfügbar sind, wird eine neue Modulinstanz mit dem geforderten Hardware-Moduls m_{req} an der gewählten möglichen Position $(x_{ph}(m_{req}, i), 1) \in X_{pos}(m_{req})$ erzeugt.

3. Nun wird versucht, die zuvor entfernten Modulinstanzen der Menge C_{int} anhand des SUP-Fit-Verfahrens neu zu platzieren. Wenn die Platzierung aller zuvor entfernten Modulinstanzen fehlschlägt, wird die originale Belegung der Modulinstanzen wiederhergestellt ($C \leftarrow C_{org}$) und mit der nächsten möglichen Position des geforderten Hardware-Moduls fortgefahren ($i \leftarrow i + 1$). Wenn die neue Platzierung der zuvor entfernten Modulinstanzen erfolgreich ist, dann ist eine gültige Lösung der Defragmentierung gefunden.

4. Nach dem Finden einer gültigen Lösung kann der Algorithmus beendet werden. Jedoch besteht zusätzlich die Möglichkeit nach weiteren Lösungen zu suchen, um am Ende, nachdem alle möglichen Positionen des Hardware-Moduls m_{req} betrachtet wurden, die Lösung zu wählen, die den geringsten Rekonfigurationsaufwand verursacht. Demnach beschreibt C_{defrag} die Menge der umzuplatzierenden Modulinstanzen, die für die Platzierung des Hardware-Moduls m_{req} an Position $(x_{ph}(m_{req}, i_{best}), 1) \in X_{pos}(m_{req})$ von ihrer ursprünglichen Position verdrängt werden. Dabei wird eine Umplatzierungszeit von T_{min} benötigt.

Erst nachdem die Menge der Modulinstanzen C_{defrag} und die entsprechenden Positionen ermittelt sind, wird die Defragmentierung anhand der Umplatzierungsmechanismen der rekonfigurierbaren Architektur durchgeführt. Falls keine gültige Lösung gefunden wurde, dann ist die Menge $C_{defrag} = \{\}$, so dass die Platzierung des geforderten Hardware-Moduls anhand der partiellen Verdrängung nicht möglich ist.

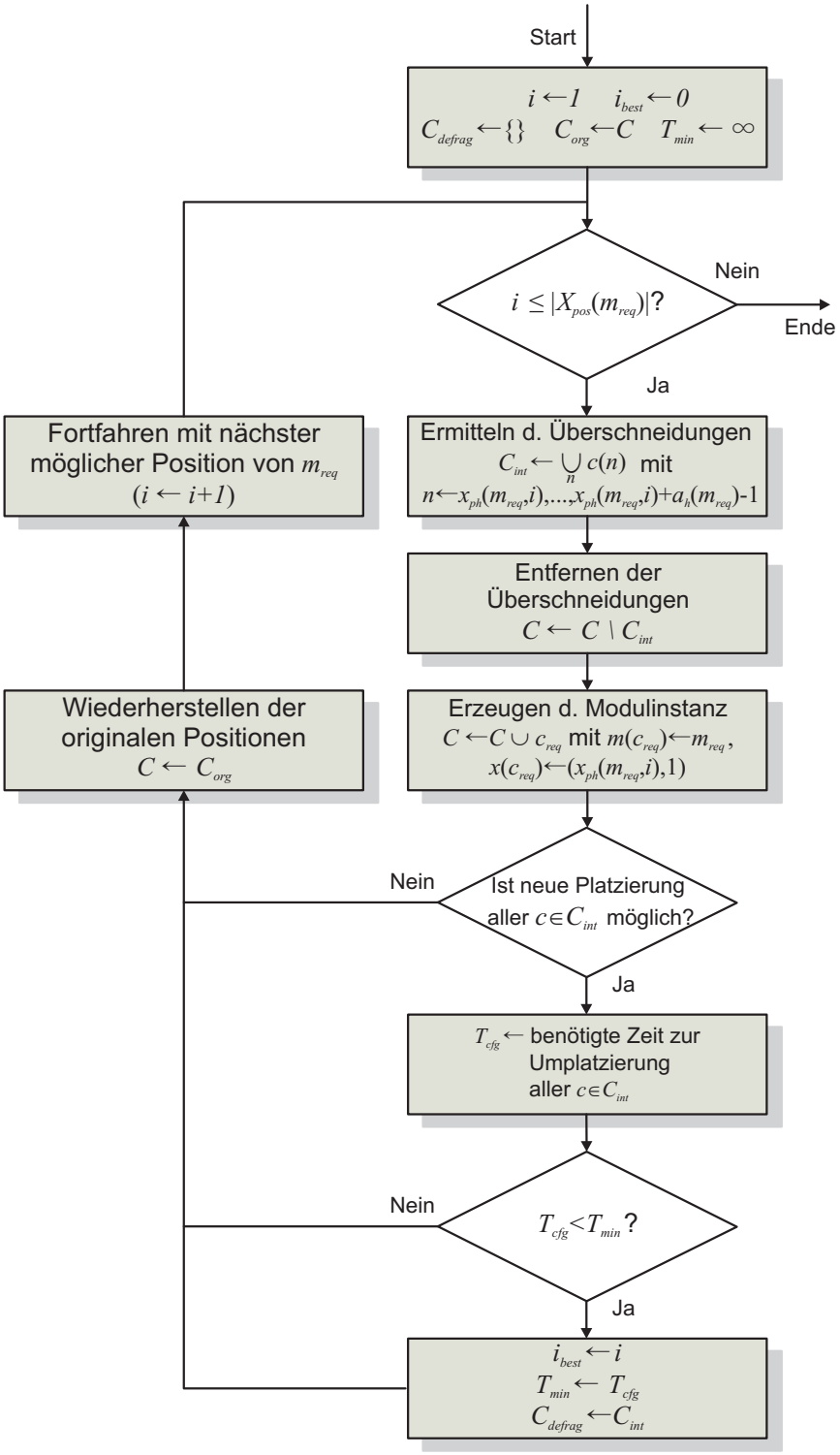
Mithilfe von Simulationen wird im folgenden Abschnitt untersucht, welchen Einfluss die Defragmentierung zur Laufzeit anhand der partiellen Verdrängung auf die Ressourcenauslastung hat.

5.3.2 Simulative Analyse

Die Ergebnisse der in Abschnitt 4.3.2 dargestellten und analysierten Simulationen im 1D-Systemansatz haben gezeigt, dass die Defragmentierung zur Laufzeit in homogenen rekonfigurierbaren Architekturen zu einer Verbesserung der Platzierung führen kann, obwohl die Umplatzierung vorhandener Modulinstanzen einen zusätzlichen Konfigurationsaufwand verursacht. In heterogenen rekonfigurierbaren Architekturen ist die Anzahl der möglichen Positionen wesentlich geringer, so dass die Flexibilität der Platzierung insbesondere im 1D-Systemansatz stark eingeschränkt ist.

In diesem Abschnitt wird anhand von Simulationen untersucht, inwieweit eine Defragmentierung zur Laufzeit mit der Methode der partiellen Verdrängung die Ressourcenauslastung oder die Zellabweisung verbessern kann. Es werden drei verschiedene Methoden zur Handhabung fehlgeschlagener Modulplatzierungen miteinander verglichen. D. h., es werden Simulationen mit Abweisung der Komponentenanfrage bei fehlgeschlagener Platzierung durchgeführt, ebenso wie Simulationen mit ereignisgesteuerter Defragmentierung anhand partieller Verdrängung (PV) und Abweisung der Komponentenanfrage bei fehlgeschlagener Defragmentierung. Zusätzlich werden Simulationen mit Verzögerung der Modulplatzierung bei fehlgeschlagener Platzierung betrachtet.

Ebenfalls wie in Abschnitt 4.3.2 unterscheiden sich die Simulationen in den Verhältnissen der Ausführungszeit zur Konfigurationszeit t_{EXE}/t_{CFG} der einzelnen Modulinstanzen. Um solch konstante Verhältnisse zu erlangen, eignet sich nur die Anwendungsklasse *G* (vgl. Tabelle 5.2), bei der die Ausführungszeit der Module - ebenso wie die Konfigurationszeit - von der Anzahl der benötigten Zellen abhängt. Die in



Algorithmus 5.3: Defragmentierung anhand partieller Verdrängung.

FPGA	Methode bei fehlg. Platzierung	Mittlere Ressourcenauslastung [%]						
		t_{EXE}/t_{CFG} ≈5	t_{EXE}/t_{CFG} ≈10	t_{EXE}/t_{CFG} ≈20	t_{EXE}/t_{CFG} ≈30	t_{EXE}/t_{CFG} ≈40	t_{EXE}/t_{CFG} ≈50	t_{EXE}/t_{CFG} ≈100
XC2V2000	Abweisung	26,54 ±1,74	29,13 ±1,79	30,15 ±2,26	30,83 ±2,20	30,95 ±2,23	31,00 ±2,26	31,31 ±2,34
	Defrag. PV	28,41 ±1,82	30,66 ±1,78	31,60 ±2,33	32,09 ±2,46	32,23 ±2,41	32,30 ±2,33	32,58 ±2,40
	Verzög.	**38,28 ±1,02**	**46,69 ±2,29**	**49,25 ±3,26**	**49,76 ±3,62**	**49,98 ±3,87**	**50,06 ±3,96**	**50,26 ±4,20**
XC2V4000	Abweisung	14,86 ±1,49	31,23 ±2,24	33,58 ±3,12	34,28 ±2,94	34,45 ±2,78	34,55 ±2,85	34,74 ±3,08
	Defrag. PV	22,25 ±2,00	34,95 ±2,82	36,90 ±3,46	37,38 ±3,60	37,66 ±3,64	37,63 ±3,59	37,92 ±3,62
	Verzög.	**25,64 ±1,84**	**41,67 ±3,27**	**44,21 ±5,13**	**44,46 ±5,37**	**44,52 ±5,36**	**44,58 ±5,38**	**44,64 ±5,37**
XC2V6000	Abweisung	7,40 ±0,84	26,10 ±1,83	35,20 ±2,94	35,71 ±3,01	35,95 ±2,88	36,14 ±2,84	36,78 ±3,05
	Defrag. PV	12,92 ±1,44	29,76 ±2,18	36,59 ±2,88	37,17 ±2,89	37,30 ±3,01	37,60 ±2,97	37,89 ±3,11
	Verzög.	**17,54 ±1,06**	**32,69 ±1,62**	**42,78 ±2,68**	**44,28 ±3,37**	**44,82 ±3,75**	**45,02 ±3,89**	**45,45 ±4,18**

Tabelle 5.12: Mittlere Ressourcenauslastung der Simulationen in Abhängigkeit des Verhältnisses der Ausführungszeit zur Konfigurationszeit t_{EXE}/t_{CFG}.

den Simulationen erzeugten virtuellen Anwendungen bestehen daher aus einer Liste mit 500 Komponentenanfragen, die auf der Anwendungsklasse *G* basieren. Die Platzierung der Hardware-Module wurde anhand des SUP-Fit-Verfahrens durchgeführt.

In Tabelle 5.12 ist die mittlere Ressourcenauslastung der verschiedenen Simulationen und deren Verhältnis der Ausführungszeit zur Konfigurationszeit t_{EXE}/t_{CFG} dargestellt. Mit zunehmendem Verhältnis von t_{EXE}/t_{CFG} nimmt auch die mittlere Ressourcenauslastung zu, wobei ab einem Verhältnis von $t_{EXE}/t_{CFG} > 20$ die mittlere Ressourcenauslastung nur noch geringfügig zunimmt. Vergleicht man die Simulationen mit Abweisung der Komponentenanfrage bei fehlgeschlagener Platzierung (Abweisung) mit den Simulationen mit Defragmentierung anhand partieller Verdrängung (Defrag. PV), dann zeigt sich, dass in den meisten Simulationen die mittlere Ressourcenauslastung durch Defragmentierung um $1\% - 2\%$ verbessert werden konnte. Die Simulationen mit Verzögerung der Modulplatzierung (Verzög.) haben eine deutlich größere mittlere Ressourcenauslastung hervorgebracht. So beträgt etwa der Unterschied zwischen der mittleren Ressourcenauslastung der Simulationen mit Abweisung und der Simulationen mit Verzögerung bis zu $19,06\%$ (XC2V2000, $t_{EXE}/t_{CFG} \approx 50$).

In Tabelle 5.13 ist die mittlere Zellabweisung der Simulationen mit Defragmentierung und der Simulationen mit Abweisung der Komponentenanfrage bei fehlge-

FPGA	Methode bei fehlg. Platzierung	Mittlere Zellabweisung [%]						
		t_{EXE}/t_{CFG} ≈5	t_{EXE}/t_{CFG} ≈10	t_{EXE}/t_{CFG} ≈20	t_{EXE}/t_{CFG} ≈30	t_{EXE}/t_{CFG} ≈40	t_{EXE}/t_{CFG} ≈50	t_{EXE}/t_{CFG} ≈100
XC2V2000	Abweisung	31,04 ±2,75	27,42 ±3,14	25,98 ±3,12	25,28 ±2,92	25,13 ±3,00	25,01 ±3,11	24,65 ±2,96
	Defrag. PV	**29,99 ±3,16**	**26,40 ±2,97**	**24,88 ±2,98**	**24,26 ±2,95**	**24,07 ±2,95**	**23,90 ±3,06**	**23,51 ±2,90**
XC2V4000	Abweisung	42,30 ±3,45	17,28 ±3,71	14,34 ±3,56	13,49 ±3,31	13,30 ±3,44	13,25 ±3,47	12,95 ±3,27
	Defrag. PV	**38,60 ±3,52**	**14,75 ±3,70**	**11,55 ±3,13**	**10,85 ±3,10**	**10,45 ±3,02**	**10,45 ±3,00**	**10,13 ±3,27**
XC2V6000	Abweisung	62,47 ±2,26	28,32 ±3,96	15,49 ±3,27	14,53 ±3,30	14,35 ±3,16	14,16 ±3,34	13,59 ±3,06
	Defrag. PV	**59,20 ±2,34**	**26,33 ±3,89**	**13,93 ±3,46**	**13,07 ±3,28**	**12,86 ±3,07**	**12,50 ±3,28**	**12,23 ±3,14**

Tabelle 5.13: Mittlere Zellabweisung der Simulationen in Abhängigkeit des Verhältnisses der Ausführungszeit zur Konfigurationszeit t_{EXE}/t_{CFG}.

schlagener Modulplatzierung dargestellt. Der Vergleich beider Verfahren bestätigt das Ergebnis der Analyse der mittleren Ressourcenauslastung, d. h., die Simulationen mit Defragmentierung verursachten eine geringere mittlere Zellabweisung als die Simulationen ohne Defragmentierung. Allgemein zeigt sich, dass die mittlere Zellabweisung mit steigendem Verhältnis der Ausführungszeit zur Konfigurationszeit t_{EXE}/t_{CFG} abnimmt. Ab einem Verhältnis von $t_{EXE}/t_{CFG} > 20$ ist die Abnahme der mittleren Zellabweisung nur noch gering.

In Tabelle 5.14 ist für jede Simulation der Mittelwert der Verzögerung vom Zeitpunkt der Komponentenanfrage (t_{REQ}) bis zum Beginn der Ausführung der entsprechenden Modulinstanz (t_{BOE}) aller erfolgreich platzierten Modulinstanzen abgebildet. Da der Beginn der Ausführung einer Modulinstanz, die aufgrund einer erfolgreichen Defragmentierung platziert wurde, durch die benötigte Zeit der Defragmentierung verzögert wird, ist der Mittelwert der Verzögerung $t_{BOE} - t_{REQ}$ bei den Simulationen mit Defragmentierung etwas größer als der entsprechende Mittelwert bei den Simulationen mit Abweisung und ohne Defragmentierung.

Die mittlere Konfigurationszeit der Modulinstanzen der Simulationen mit Abweisung liegt bei $2,58 \cdot 10^{-3} \pm 0.23 \cdot 10^{-3}$ s, so dass der Mittelwert der Verzögerung $t_{BOE} - t_{REQ}$ bei den Simulationen mit Defragmentierung und den Simulationen mit Abweisung überwiegend durch die Konfigurationszeit der Modulinstanz geprägt wird. Bei den Simulationen mit Verzögerung der Modulplatzierung bei fehlgeschlagener Platzierung ist die mittlere Verzögerung $t_{BOE} - t_{REQ}$ deutlich größer.

Zusammenfassend lässt sich feststellen, dass sich mithilfe der Defragmentierung anhand partieller Verdrängung die Ressourcenauslastung gegenüber dem Ansatz ohne Defragmentierung steigern lässt. Jedoch entsteht durch die Defragmentierung eine

FPGA	Methode bei fehlg. Platzierung	Mittlere Verzögerung $t_{BOE}(d_i)$-$t_{REQ}(d_i)$ [s]						
		t_{EXE}/t_{CFG} ≈5	t_{EXE}/t_{CFG} ≈10	t_{EXE}/t_{CFG} ≈20	t_{EXE}/t_{CFG} ≈30	t_{EXE}/t_{CFG} ≈40	t_{EXE}/t_{CFG} ≈50	t_{EXE}/t_{CFG} ≈100
XC2V2000	Abweisung	**5,6E-03 ±3,4E-04**	**3,8E-03 ±1,8E-04**	**3,1E-03 ±1,7E-04**	**2,9E-03 ±1,4E-04**	**2,8E-03 ±1,1E-04**	**2,8E-03 ±9,7E-05**	**2,7E-03 ±1,0E-04**
	Defrag. PV	6,0E-03 ±4,1E-04	4,0E-03 ±2,0E-04	3,3E-03 ±1,9E-04	3,0E-03 ±1,6E-04	2,9E-03 ±1,3E-04	2,9E-03 ±1,2E-04	2,8E-03 ±1,1E-04
	Verzög.	18,3E-03 ±9,7E-04	14,7E-03 ±1,6E-03	12,7E-03 ±1,8E-03	12,0E-03 ±1,8E-03	11,6E-03 ±1,8E-03	11,5E-03 ±1,8E-03	11,1E-03 ±1,9E-03
XC2V4000	Abweisung	**31,8E-03 ±2,7E-03**	**8,2E-03 ±1,0E-03**	**4,4E-03 ±4,1E-04**	**3,7E-03 ±2,4E-04**	**3,4E-03 ±1,9E-04**	**3,3E-03 ±1,7E-04**	**2,9E-03 ±1,3E-04**
	Defrag. PV	34,5E-03 ±2,6E-03	9,7E-03 ±1,1E-03	4,9E-03 ±4,7E-04	4,0E-03 ±3,1E-04	3,7E-03 ±3,0E-04	3,5E-03 ±2,6E-04	3,1E-03 ±1,8E-04
	Verzög.	42,3E-03 ±1,4E-03	22,3E-03 ±3,2E-03	11,8E-03 ±3,2E-03	9,9E-03 ±2,5E-03	9,2E-03 ±2,4E-03	8,7E-03 ±2,1E-03	8,2E-03 ±2,4E-03
XC2V6000	Abweisung	**57,7E-03 ±1,7E-03**	**26,7E-03 ±3,4E-03**	**6,5E-03 ±7,7E-04**	**4,6E-03 ±3,7E-04**	**4,0E-03 ±2,5E-04**	**3,7E-03 ±2,3E-04**	**3,2E-03 ±1,9E-04**
	Defrag. PV	58,6E-03 ±2,6E-03	28,6E-03 ±3,3E-03	7,1E-03 ±8,4E-04	4,9E-03 ±4,1E-04	4,3E-03 ±3,1E-04	3,9E-03 ±2,9E-04	3,3E-03 ±2,1E-04
	Verzög.	62,3E-03 ±2,0E-03	39,7E-03 ±1,8E-03	21,6E-03 ±3,0E-03	17,4E-03 ±3,5E-03	15,9E-03 ±3,7E-03	15,0E-03 ±3,9E-03	13,3E-03 ±4,0E-03

Tabelle 5.14: Durchschnittliche Verzögerung $t_{BOE}(d_i) - t_{REQ}(d_i)$ der Simulationen in Abhängigkeit des Verhältnisses der Ausführungszeit zur Konfigurationszeit t_{EXE}/t_{CFG}.

zusätzliche Verzögerung, so dass der Mittelwert der Verzögerung $t_{BOE} - t_{REQ}$ ebenfalls größer ist als bei dem Ansatz ohne Defragmentierung. Bei Verzögerung der Modulplatzierung wurde eine wesentlich höhere mittlere Ressourcenauslastung erzielt, jedoch ebenso ein wesentlich größerer Mittelwert der Verzögerung $t_{BOE} - t_{REQ}$.

5.4 Zusammenfassung

Im Gegensatz zu homogenen rekonfigurierbaren Architekturen sind die Hardware-Module in heterogenen rekonfigurierbaren Architekturen nicht mehr beliebig platzierbar, sondern erfordern die Berücksichtigung der Anordnung der einzelnen Zelltypen und der Unregelmäßigkeiten in der Kommunikationsinfrastruktur. Durch diese Heterogenität entstehen zu jedem Hardware-Modul eine eingeschränkte Menge an unregelmäßig verteilten möglichen Positionen. In diesem Kapitel wurden die erforderlichen Anpassungen der klassischen homogenen Platzierungsverfahren, wie z. B. Best-Fit, beschrieben. Ziel der Platzierung sollte aber nicht nur, wie in homogenen rekonfigurierbaren Architekturen üblich, die Berücksichtigung der externen Fragmentierung sein, sondern ebenso die Erhaltung der freien möglichen Positionen aller im System vorhandenen Hardware-Module. Einen ersten Ansatz zur gezielten Berück-

sichtigung der möglichen Positionen aller Hardware-Module stellen die hier vorgestellten effizienten Platzierungsverfahren SUP-Fit und RUP-Fit dar. Prinzip beider Verfahren ist die Verwendung von Positionsgewichten, die aus den Zellgewichten der einzelnen Zellen berechnet werden. Das SUP-Fit Verfahren verwendet statische Positionsgewichte und zeichnet sich durch eine lineare Laufzeit aus. Zur Laufzeit verhält sich das SUP-Fit-Verfahren ähnlich dem klassischen First-Fit-Verfahren, d. h., die möglichen Positionen des geforderten Hardware-Moduls werden in einer festen Reihenfolge durchsucht, bis eine freie Position gefunden ist. Der Unterschied zum First-Fit-Verfahren besteht in der Suchreihenfolge der möglichen Positionen, die zur Entwurfszeit anhand der statischen Positionsgewichte bestimmt wird. Das RUP-Fit-Verfahren hat eine höhere Zeitkomplexität, die sich daraus ergibt, dass die Positionsgewichte in Abhängigkeit der derzeitigen Zellbelegung aktualisiert werden. Das SUP-Fit- und das RUP-Fit-Verfahren wurden anhand von Simulationen mit den für heterogene rekonfigurierbare Architekturen angepassten Platzierungsverfahren Best-Fit und First-Fit verglichen. Die Analyse der durchgeführten Simulationen ergab, dass im direkten Vergleich sowohl im 2D- als auch im 1D-Systemansatz die Platzierungsverfahren RUP-Fit und SUP-Fit in der Regel die größte mittlere Ressourcenauslastung hervorbrachten.

Ebenso wie in homogenen rekonfigurierbaren Architekturen kann die Leistungsfähigkeit eines Platzierungsverfahrens durch Verwendung geeigneter Defragmentierungsverfahren zur Laufzeit verbessert werden. Existierende Defragmentierungsverfahren basieren jedoch auf der Annahme, dass die vorhandenen Modulinstanzen an beliebige Positionen umplatziert werden können, und sind daher wegen der Einschränkungen durch die unterschiedlichen und unregelmäßig verteilten möglichen Positionen der einzelnen Hardware-Module nicht für heterogene rekonfigurierbare Architekturen geeignet. Die hier vorgestellte Defragmentierung anhand partieller Verdrängung markiert einen ersten Ansatz eines heuristischen Defragmentierungsverfahrens für heterogene rekonfigurierbare Architekturen. Dabei versucht das Verfahren, die geforderte Platzierung eines nicht ohne Weiteres platzierbaren Hardware-Moduls durch Umplatzieren der bereits vorhanden Modulinstanzen zu ermöglichen. Das Verfahren wählt eine mögliche Position des geforderten Hardware-Moduls und versucht neue Positionen für die Modulinstanzen zu finden, welche die für die Platzierung benötigten Zellen verwenden und damit blockieren. Als Platzierungsverfahren zur Bestimmung der neuen Positionen wird das SUP-Fit-Verfahren verwendet. Simulative Analysen ergaben, dass sich mithilfe der partiellen Verdrängung die mittlere Ressourcenauslastung steigern lässt. In [E7] wird gezeigt, dass sich das Verfahren mit heutigen Xilinx Virtex-II FPGAs umsetzen lässt.

6 Zusammenfassung und Ausblick

Das Konzept der partiellen rekonfigurierbaren Architekturen ermöglicht die gezielte Veränderung eines Teils der Architektur, während der verbleibende Teil seine bisherige Funktion unverändert fortsetzt. Auf diese Weise lassen sich Systemkomponenten bei Bedarf zur Laufzeit platzieren und entfernen. In dieser Arbeit wurden die zur Umsetzung von partieller Rekonfiguration benötigten Methoden zur Ressourcenverwaltung anhand von Simulationen untersucht. Die Analyse mittels Simulationen ermöglichte dabei die Berücksichtigung real existierender Architekturen. Exemplarisch wurde die Xilinx Virtex-II FPGA Serie betrachtet. Die simulative Analyse wurde anhand der im Rahmen dieser Arbeit entwickelten Simulationsumgebung SARA durchgeführt. Diese basiert auf dem DMC-Modell, welches die drei Ebenen Systemkomponente, Hardware-Modul und Modulinstanz definiert und deren Zusammenhang beschreibt. Mithilfe der Simulationsumgebung SARA können die verschiedenen Systemansätze und deren Platzierungsverfahren bezüglich Metriken, wie z. B. Ressourcenauslastung, interne Fragmentierung und relative Verfügbarkeit verglichen werden.

Die Platzierung eines Hardware-Moduls zur Laufzeit unterteilt sich in die Verwaltung der auf der rekonfigurierbaren Architektur vorhandenen freien Flächen und in die Bestimmung einer freien Position aus der Menge der möglichen Positionen des Hardware-Moduls. Für die Verwaltung der freien Flächen existieren verschiedene Verfahren, wie z. B. die Bestimmung und Erhaltung maximaler freier Rechtecke. Die Bestimmung der Position ergibt sich durch Auswahl eines geeigneten freien Rechtecks und der Platzierung des Hardware-Moduls innerhalb des gewählten Rechtecks. Die Auswahl eines geeigneten freien Rechtecks entspricht dabei einem Online-Packungsproblem. Dementsprechend können bei der Platzierung bekannte Verfahren, wie z. B. First-Fit und Best-Fit, verwendet werden.

Mithilfe der Simulationsumgebung SARA wurden die verschiedenen Systemansätze für homogene rekonfigurierbare Architekturen miteinander verglichen. Der 2D-Systemansatz gestattet eine beliebige Platzierung von Hardware-Modulen in horizontaler und vertikaler Richtung. Durch diese hohe Flexibilität entsteht mit der Zeit eine größere externe Fragmentierung als im 1D-Systemansatz, bei dem die Hardware-Module nur in horizontaler Richtung platziert werden können. Im 1D-Systemansatz umfasst die vertikale Flächenausdehnung der Hardware-Module immer die komplette Höhe der rekonfigurierbaren Architektur, so dass die interne Fragmentierung des

1D-Systemansatzes in den betrachteten Simulationen größer war als die des 2D-Systemansatzes. Im unmittelbaren Vergleich erwies sich jedoch der 1D-Systemansatz bei den durchgeführten Simulationen als der Ansatz mit der größten mittleren Ressourcenauslastung und der geringsten mittleren Zellabweisung. Die externe Fragmentierung hat damit einen größeren Einfluss auf die Leistungsfähigkeit eines Systemansatzes als die interne Fragmentierung. Einerseits weist der 1D-Systemansatz eine Leistungsfähigkeit auf, die nicht signifikant schlechter, sondern in vielen Fällen sogar besser ist als die des 2D-Systemansatzes. Andererseits ist der 1D-Systemansatz im Gegensatz zum 2D-Systemansatz mit heutigen rekonfigurierbaren Architekturen realisierbar. Der ebenfalls in der Arbeit erwähnte Systemansatz mit fester Aufteilung ist zwar ebenso mit heutigen rekonfigurierbaren Architekturen realisierbar. Jedoch hat der Ansatz in allen betrachteten Simulationen eine deutlich geringere Ressourcenauslastung hervorgerufen als der 1D-Systemansatz.

Neben dem Vergleich der Systemansätze wurde der Einfluss der Platzierungszeit und der Konfigurationszeit der Hardware-Module auf die Ressourcenauslastung analysiert. Die Platzierungszeit entspricht dabei der zur Bestimmung der Position eines geforderten Hardware-Moduls benötigten Rechenzeit, die von dem gewählten Platzierungsalgorithmus abhängt. Die hierbei betrachteten Simulationen basierten auf den 1D-Systemansatz, wobei der in Xilinx Virtex-II/Pro FPGAs eingebettete Prozessor IBM PowerPC 405 als Plattform für die Ausführung der Platzierungsalgorithmen verwendet wurde. Die resultierenden Platzierungszeiten der untersuchten Platzierungsalgorithmen waren dabei so gering, dass keine signifikante Auswirkung auf die mittlere Ressourcenauslastung zu beobachten war. Die Konfigurationszeit entspricht der Zeit, die für die partielle Rekonfiguration der gegebenen Architektur anhand der Konfigurationsdaten des zu platzierenden Hardware-Moduls benötigt wird. Anhand von Simulationen konnte gezeigt werden, dass die Konfigurationszeit hingegen einen entscheidenden Einfluss auf die resultierende mittlere Ressourcenauslastung hat. Bezüglich eines Systementwurfs ergibt sich demnach die Forderung, die Konfigurationszeit so gering wie möglich zu halten.

Das im Zusammenhang mit Speichermanagement bekannte Konzept der Defragmentierung lässt sich auch auf partiell rekonfigurierbare Hardware übertragen. Durch die Umplatzierung von Modulinstanzen zur Laufzeit kann unter bestimmten Voraussetzungen eine fehlgeschlagene Modulplatzierung dennoch ermöglicht werden. Wichtige Zielgröße bei der Umplatzierung ist ebenfalls die benötigte Konfigurationszeit zur Umplatzierung der Modulinstanzen. Mit der Methode der partiellen Kompaktierung wurde ein entsprechendes Defragmentierungsverfahren vorgestellt, welches den für eine geforderte Modulplatzierung benötigten Rekonfigurationsaufwand minimiert. Trotz des erhöhten Konfigurationsbedarfs, der sich durch die Umplatzierung der Modulinstanzen ergibt, konnte anhand von Simulationen gezeigt werden, dass sich die mittlere Ressourcenauslastung durch das beschriebene Defragmentierungsverfahren steigern lässt.

Im Gegensatz zu homogenen rekonfigurierbaren Architekturen sind die Hardware-Module in heterogenen rekonfigurierbaren Architekturen nicht mehr uneingeschränkt platzierbar, sondern erfordern die Berücksichtigung der Anordnung der einzelnen Zelltypen und der Unregelmäßigkeiten in der Kommunikationsinfrastruktur. Somit ergibt sich für jedes Hardware-Modul eine Menge von ungleichmäßig verteilten möglichen Positionen. Übergeordnetes Ziel der Platzierung sollte daher nicht nur die Berücksichtigung der externen Fragmentierung sein, sondern ebenso die Erhaltung der freien möglichen Positionen aller im System vorhandenen Hardware-Module. Zu diesem Zweck sind im Rahmen dieser Arbeit die Platzierungsverfahren SUP-Fit und RUP-Fit entwickelt worden, welche zu den wichtigsten Ergebnissen dieser Arbeit zählen. Das Prinzip der Verfahren beruht auf der Bestimmung von Zellgewichten, aus denen eine Gewichtung der möglichen Positionen der einzelnen Hardware-Module hergeleitet wird. Bei einer Modulplatzierung wird die Auswahl der resultierenden Position anhand der entsprechenden Gewichte bestimmt. Die vorgestellten Verfahren stellen einen neuen Platzierungsansatz dar, der auf die Anforderungen von heterogenen rekonfigurierbaren Architekturen abgestimmt ist. Die Platzierungsverfahren SUP-Fit und RUP-Fit wurden anhand von Simulationen mit den für heterogene Architekturen angepassten Platzierungsverfahren Best-Fit und First-Fit verglichen. Die simulative Analyse ergab, dass im direkten Vergleich die Platzierungsverfahren RUP-Fit und SUP-Fit in der Regel die größte mittlere Ressourcenauslastung hervorbrachten.

Durch die eingeschränkte Anzahl an möglichen Positionen der Hardware-Module können Defragmentierungsverfahren für homogene rekonfigurierbare Architekturen nicht direkt für heterogene Architekturen genutzt werden. Aus diesem Grund ist ein neues Defragmentierungsverfahren für heterogene rekonfigurierbare Architekturen entwickelt worden. Wenn ein gefordertes Hardware-Modul nicht platziert werden kann, wählt das Verfahren eine derzeit belegte mögliche Position und versucht neue Positionen für die Modulinstanzen zu finden, welche die für die Platzierung benötigten Zellen verwenden. Die Bestimmung der neuen Positionen geschieht dabei mithilfe des SUP-Fit-Verfahrens. Anhand von Simulationen konnte gezeigt werden, dass die mittlere Ressourcenauslastung durch Verwendung des beschriebenen Verfahrens gesteigert werden kann.

Ein wichtiges Ergebnis der vorliegenden Arbeit ist die Erkenntnis, dass die Anzahl der möglichen Positionen von Hardware-Modulen in heterogenen Architekturen einen Einfluss auf die Ressourcenauslastung hat. Im Hinblick auf den Entwurf eines Systems mit heterogener rekonfigurierbarer Hardware ergibt sich daher die Forderung, dass bei der Synthese eines Hardware-Moduls die initiale Position derart gewählt wird, dass die Anzahl der daraus resultierenden möglichen Positionen maximiert wird. Ein weiteres Ziel bei dem Entwurf eines Systems mit heterogener rekonfigurierbarer Hardware ist die möglichst gleichmäßige Nutzung der vorhandenen Zellen, um eine hohe Ressourcenauslastung zu erzielen. Die möglichst gleichmäßige Nutzung der Zellen wird dann erreicht, wenn die Zellgewichte ausgeglichen

sind und keinen erheblichen Schwankungen unterliegen. Wenn etwa eine Zelle ein sehr niedriges statisches Zellgewicht hat, dann wird die Zelle von wenigen möglichen Positionen der gegebenen Hardware-Module genutzt. Mithilfe der Berechnung der statischen Zellgewichte kann beim Systementwurf die geringe oder sehr häufige Nutzung einer Zelle durch entsprechende Optimierung der initialen Positionen der Hardware-Module gezielt vermieden werden.

Die Mehrzahl der heutigen rekonfigurierbaren Architekturen weisen Heterogenitäten durch die Integration verschiedener Zelltypen und deren Verbindungen auf, was die Relevanz der in dieser Arbeit eingeführten heterogenen Platzierungsverfahren unterstreicht. Die hier erläuterte Datenstruktur zur Realisierung der entsprechenden Algorithmen diente vorrangig zum Nachweis, dass bis auf RUP-Fit alle vorgestellten Verfahren eine lineare Laufzeit besitzen. Durch Verwendung einer erweiterten Datenstruktur ist eine Optimierung der Laufzeit und des benötigten Speicherbedarfs der vorgestellten Algorithmen denkbar. Ferner kann die Berechnung der Positionsgewichte um verschiedene Faktoren erweitert werden. Wenn z. B. ein Hardware-Modul platziert werden soll, welches funktional von bereits platzierten Modulinstanzen abhängt, so kann etwa die Kommunikationskanallänge zu den entsprechenden Modulinstanzen in die Berechnung der Positionsgewichte einfließen. Unter Berücksichtigung der Kommunikationskanallänge ergeben sich bei der Platzierung zwei voneinander unabhängige Ziele: Einerseits versucht das Verfahren die Anzahl der freien möglichen Positionen der Hardware-Module zu maximieren. Andererseits wird die Kommunikationskanallänge von funktional abhängigen Modulinstanzen optimiert. Das auf diese Weise hervorgerufene Mehrzieloptimierungsproblem eröffnete neue Forschungsperspektiven.

Die formale Analyse der hier vorgestellten Platzierungs- und Defragmentierungsverfahren anhand von Methoden, wie z. B. der kompetitiven Analyse, ist ein offenes Problem. In dieser Arbeit wurde daher eine simulative Analyse der Systemansätze und deren Platzierungsverfahren verwendet. Die dafür entwickelte Simulationsumgebung SARA kann auf vielfältige Weise erweitert werden. Neue Systemansätze oder neue rekonfigurierbare Architekturen können auf einfache Weise in das bestehende System integriert und unter realen Bedingungen getestet werden. Die Simulationsumgebung SARA kann ebenso beim Systementwurf von partiell rekonfigurierbaren Architekturen genutzt werden, um Systemansätze und das Laufzeitverhalten der gewählten Platzierungsverfahren vor der Implementierung zu testen. Auf diese Weise lassen sich Abschätzungen darüber treffen, ob die Größe der gewählten Architektur für eine gegebene Anwendung ausreichend ist.

Literaturverzeichnis

[1] AHMADINIA, A. ; BOBDA, C. ; BEDNARA, M. ; TEICH, J. : A New Approach for On-line Placement on Reconfigurable Devices. In: *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS) - Reconfigurable Architectures Workshop (RAW)*, IEEE Computer Society, April 2004

[2] AHMADINIA, A. ; BOBDA, C. ; MAJER, M. ; TEICH, J. ; FEKETE, S. ; VEEN, J. van d.: DyNoC: A Dynamic Infrastructure for Communication in Dynamically Reconfigurable Devices. In: *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL)*. Tampere, Finland : IEEE Computer Society, August 2005, S. 153–158

[3] ALTERA CORP.: *Excalibur Device Overview*. San Jose, California, Mai 2002

[4] ALTERA CORP.: *Nios II Processor Reference Handbook*. San Jose, California, 2006

[5] ALTERA CORP.: *Stratix Device Handbook*. San Jose, California, 2006

[6] BANERJEE, S. ; BOZORGZADEH, E. ; DUTT, N. : Physically-Aware HW-SW Partitioning for Reconfigurable Architectures with Partial Dynamic Reconfiguration. In: *DAC '05: Proceedings of the 42nd annual conference on Design Automation*. New York, NY, USA : ACM Press, 2005, S. 335–340

[7] BAUMGARTEN, V. ; EHLERS, G. ; MAY, F. ; NÜCKEL, A. ; VORBACH, M. ; WEINHARDT, M. : PACT XPP - A Self-Reconfigurable Data Processing Architecture. In: *The Journal of Supercomputing* 26 (2003), Nr. 2, S. 167–184

[8] BAZARGAN, K. ; KASTNER, R. ; SARRAFZADEH, M. : Fast Template Placement for Reconfigurable Computing Systems. In: *IEEE Design and Test* 17 (2000), Nr. 1, S. 68–83

[9] BECKER, J. ; GLESNER, M. : A Parallel Dynamically Reconfigurable Architecture Designed for Flexible Application-Tailored Hardware/Software Systems in Future Mobile Communication. In: *The Journal of Supercomputing* 19 (2001), Nr. 1, S. 105–127

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

[11] BONDALAPATI, K. ; PRASANNA, V. K.: Mapping Loops onto Reconfigurable Architectures. In: *Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications (FPL)* Bd. 1482, Springer, September 1998 (Lecture Notes in Computer Science), S. 268–277

[12] BONDALAPATI, K. ; PRASANNA, V. K.: DRIVE: An Interpretive Simulation and Visualization Environment for Dynamically Reconfigurable Systems. In: *Field-Programmable Logic and Applications: 9th International Workshop (FPL)* Bd. 1673, Springer, 1999 (Lecture Notes in Computer Science), S. 31–40

[13] BREBNER, G. : The Swappable Logic Unit: A Paradigm for Virtual Hardware. In: *Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM)*. Washington, DC, USA : IEEE Computer Society, 1997, S. 77–86

[14] BREBNER, G. J.: A Virtual Hardware Operating System for the Xilinx XC6200. In: *FPL '96: Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*. London, UK : Springer-Verlag, 1996, S. 327–336

[15] BREBNER, G. J.; DIESSEL, O. : Chip-Based Reconfigurable Task Management. In: *Proceedings of the 11th International Conference on Field-Programmable Logic and Applications (FPL)* Bd. 2147, Springer, 2001 (Lecture Notes in Computer Science), S. 182–191

[16] CARRILLO, J. E.; CHOW, P. : The Effect of Reconfigurable Units in Superscalar Processors. In: *FPGA '01: Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*. New York, NY, USA : ACM Press, 2001, S. 141–150

[17] CASPI, E. ; CHU, M. ; HUANG, R. ; YEH, J. ; WAWRZYNEK, J. ; DEHON, A. : Stream Computations Organized for Reconfigurable Execution (SCORE). In: *Field-Programmable Logic: Proceedings of the The Roadmap to Reconfigurable Computing. 10th International Workshop on Field-Programmable Logic and Applications (FPL)* Bd. 1896, Springer, 2000 (Lecture Notes in Computer Science), S. 605–614

[18] CHAZELLE, B. : The Bottom-Left Bin Packing Heuristic: An Efficient Implementation. In: *IEEE Transactions on Computers* C-32 (1983), August, Nr. 8, S. 697–707

[19] COFFMAN, E. G.; GAREY, M. R.; JOHNSON, D. S.: Approximation Algorithms for Bin Packing: A Survey. In: HOCHBAUM, D. S. (Hrsg.): *Approximation Algorithms for NP-Hard Problems*. Boston, MA, USA : PWS Publishing Company, 1996, S. 46–93

[20] COMPTON, K. ; HAUCK, S. : Totem: Custom Reconfigurable Array Generation. In: *Proceedings of the 9th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM)*. Washington, DC, USA : IEEE Computer Society, 2001, S. 111–119

[21] COMPTON, K. ; HAUCK, S. : Reconfigurable Computing: A Survey of Systems and Software. In: *ACM Computing Surveys* 34 (2002), June, Nr. 2, S. 171–210

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

[23] DANNE, K. ; STÜHMEIER, S. : Off-line Placement of Tasks onto Reconfigurable Hardware Considering Geometrical Task Variants. In: *Proceedings of International Embedded Systems Symposium 2005 (IESS)*, 2005

[24] DEHON, A. : DPGA Utilization and Application. In: *FPGA '96: Proceedings of the 1996 ACM fourth international symposium on Field-Programmable Gate Arrays*. New York, NY, USA : ACM Press, 1996, S. 115–121

[25] DIESSEL, O. ; ELGINDY, H. ; MIDDENDORF, M. ; SCHMIDT, B. ; SCHMECK, H. : Dynamic Scheduling of Tasks on Partially Reconfigurable FPGAs. In: *IEE - Proceedings - Computer and Digital Techniques* 147 (2000), Nr. 3, S. 181–188

[26] DIESSEL, O. ; ELGINDY, H. A.: Run-Time Compaction of FPGA Designs. In: *Field-Programmable Logic and Applications, 7th International Workshop (FPL)* Bd. 1304, Springer, 1997 (Lecture Notes in Computer Science), S. 131–140

[27] EBELING, C. ; CRONQUIST, D. C.; FRANKLIN, P. : RaPiD – Reconfigurable Pipelined Datapath. In: HARTENSTEIN, R. W. (Hrsg.); GLESNER, M. (Hrsg.): *Field-Programmable Logic: Smart Applications, New Paradigms, and Compilers. 6th International Workshop on Field-Programmable Logic and Applications (FPL)* Bd. 1142. Darmstadt, Germany : Springer, 1996 (Lecture Notes in Computer Science), S. 126–135

[28] EDMONDS, J. ; GRYZ, J. ; LIANG, D. ; MILLER, R. J.: Mining for empty spaces in large data sets. In: *Theoretical Computer Science* 296 (2003), Nr. 3, S. 435–452

[29] EJNIOUI, A. ; DEMARA, R. F.: Area Reclamation Metrics for SRAM-based Reconfigurable Device. In: *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*. Las Vegas, USA : CSREA Press, June 27-30 2005, S. 196–202

[30] FEKETE, S. ; KÖHLER, E. ; TEICH, J. : Optimal FPGA module placement with temporal precedence constraints. In: *DATE '01: Proceedings of the conference on Design, automation and test in Europe*. Piscataway, NJ, USA : IEEE Computer Society, 2001, S. 658–667

[31] FERRANDI, F. ; SANTAMBROGIO, M. D.; SCIUTO, D. : A Design Methodology for Dynamic Reconfiguration: The Caronte Architecture. In: *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS) - Reconfigurable Architectures Workshop (RAW)*, IEEE Computer Society, 2005

[32] GARCIA, P. ; COMPTON, K. ; SCHULTE, M. ; BLEM, E. ; FU, W. : An Overview of Reconfigurable Hardware in Embedded Systems. In: *EURASIP Journal on Embedded Systems* 2006 (2006), S. Article ID 56320, 19 pages

[33] GEORGE, V. ; ZHANG, H. ; RABAEY, J. : The Design of a Low Energy FPGA. In: *ISLPED '99: Proceedings of the 1999 International Symposium on Low Power Electronics and Design*. New York, NY, USA : ACM Press, 1999, S. 188–193

[34] GERICOTA, M. G.; ALVES, G. R.; SILVA, M. L.; FERREIRA, J. M. M.: Online Defragmentation for Run-Time Partially Reconfigurable FPGAs. In: *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications (FPL)* Bd. 2438, Springer, 2002 (Lecture Notes in Computer Science), S. 302–311

[35] GOLDSTEIN, S. ; SCHMIT, H. ; MOE, M. ; BUDIU, M. ; CADAMBI, S. ; TAYLOR, R. ; LAUFER, R. : PipeRench: A Coprocessor for Streaming Multimedia Acceleration. In: *Proceedings of the 26th International Symposium on Computer Architecture*, 1999, S. 28–39

[36] HAGEMEYER, J. ; KETTELHOIT, B. ; PORRMANN, M. : Dedicated Module Access in Dynamically Reconfigurable Systems. In: *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS) - Reconfigurable Architectures Workshop (RAW)*. Rhodes, Greece : IEEE Computer Society, 2006

[37] HANDA, M. ; VEMURI, R. : Area Fragmentation in Reconfigurable Operating Systems. In: *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, CSREA Press, 2004, S. 77–83

[38] HANDA, M. ; VEMURI, R. : An Efficient Algorithm for Finding Empty Space for Online FPGA Placement. In: *Proceedings of the 41st Annual Conference on Design Automation (DAC)*. New York, NY, USA : ACM Press, 2004, S. 960–965

[39] HAUCK, S. ; FRY, T. W.; HOSLER, M. M.; KAO, J. P.: The Chimaera Reconfigurable Functional Unit. In: *Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM)*. Washington, DC, USA : IEEE Computer Society, 1997, S. 87–96

[40] HAUSER, J. R.; WAWRZYNEK, J. : GARP: A MIPS Processor with a Reconfigurable Coprocessor. In: *Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM)*. Washington, DC, USA : IEEE Computer Society, 1997, S. 12–21

[41] HÜBNER, M. ; SCHUCK, C. ; BECKER, J. : Elementary Block Based 2-Dimensional Dynamic and Partial Reconfiguration for Virtex-II FPGAs. In: *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS) - Reconfigurable Architectures Workshop (RAW)*. Rhodes, Greece : IEEE Computer Society, 2006

[42] IBM CORPORATION: *Instruction Set Simulator Version 1.3 User Guide 5th Edition*. Research Triangle Park, North Carolina, September 2002

[43] IEEE COMPUTER SCOCIETY: *IEEE Standard VHDL Language Reference Manual, Number IEEE Std 1076*. New York, New York, 2000

[44] KALTE, H. ; LEE, G. ; PORRMANN, M. ; RÜCKERT, U. : 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)*, IEEE Computer Society, 2005

[45] KALTE, H. ; PORRMANN, M. ; RÜCKERT, U. : A Prototyping Platform for Dynamically Reconfigurable System on Chip Designs. In: *Proceedings of the IEEE Workshop Heterogeneous reconfigurable Systems on Chip (SoC)*, 2002

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

[47] KALTE, H. ; PORRMANN, M. ; RÜCKERT, U. : System-on-Programmable-Chip Approach Enabling Online Fine-Grained 1D-Placement. In: *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS) - Reconfigurable Architectures Workshop (RAW)*. Santa Fé, New Mexico : IEEE Computer Society, 2004

[48] LANGE, S. ; MIDDENDORF, M. : Hyperreconfigurable Architectures for Fast Run Time Reconfiguration. In: *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Los Alamitos, CA, USA : IEEE Computer Society, 2004, S. 304–305

[49] LANGE, S. ; MIDDENDORF, M. : The Partition into Hypercontexts Problem for Hyperreconfigurable Architectures. In: *Proceedings of the 14th International Workshop on Field-Programmable Logic and Applications (FPL)* Bd. 3203, Springer, January 2004 (Lecture Notes in Computer Science), S. 251–260

[50] LANGEN, D. ; NIEMANN, J. C.; PORRMANN, M. ; KALTE, H. ; RÜCKERT, U. : Implementation of a RISC Processor Core for SoC Designs - FPGA Prototype vs. ASIC Implementation. In: *Proceedings of the IEEE–Workshop: Heterogeneous reconfigurable Systems on Chip (SoC)*, 2002

[51] LAUFER, R. ; TAYLOR, R. R.; SCHMIT, H. : PCI-PipeRench and the SwordAPI: A System for Stream-based Reconfigurable Computing. In: POCEK, K. L. (Hrsg.); ARNOLD, J. M. (Hrsg.): *IEEE Symposium on FPGAs for Custom Computing Machines*. Los Alamitos, CA : IEEE Computer Society Press, Apr. 1999, S. 200–208

[52] LODI, A. ; MARTELLO, S. ; MONACI, M. : Two-Dimensional Packing Problems: A Survey. In: *European Journal of Operational Research* 141 (2002), September, Nr. 2, S. 241–252

[53] LYSAGHT, P. ; STOCKWOOD, J. : A Simulation Tool for Dynamically Reconfigurable Field Programmable Gate Arrays. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 4 (1996), Nr. 3, S. 381–390

[54] MARESCAUX, T. ; BARTIC, A. ; VERKEST, D. ; VERNALDE, S. ; LAUWEREINS, R. : Interconnection Networks Enable Fine-Grain Dynamic Multitasking on FPGAs. In: *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications (FPL)* Bd. 2438, Springer, 2002 (Lecture Notes in Computer Science), S. 795–805

[55] MARSHALL, A. ; STANSFIELD, T. ; KOSTARNOV, I. ; VUILLEMIN, J. ; HUTCHINGS, B. : A reconfigurable arithmetic array for multimedia applications. In: *FPGA '99: Proceedings of the 1999 ACM/SIGDA seventh international*

symposium on Field programmable gate arrays. New York, NY, USA : ACM Press, 1999, S. 135–143

[56] MEI, B. ; VERNALDE, S. ; VERKEST, D. ; MAN, H. D.; LAUWEREINS, R. : ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In: *Proceedings of the 13rd International Conference on Field-Programmable Logic and Applications (FPL)* Bd. 2778, Springer, September 2003 (Lecture Notes in Computer Science), S. 61–70

[57] MIRSKY, E. ; DEHON, A. : MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources. In: POCEK, K. L. (Hrsg.); ARNOLD, J. (Hrsg.): *IEEE Symposium on FPGAs for Custom Computing Machines*. Los Alamitos, CA : IEEE Computer Society Press, 1996, S. 157–166

[58] MIYAMORI, T. ; OLUKOTUN, K. : A Quantitative Analysis of Reconfigurable Coprocessors for Multimedia Applications. In: POCEK, K. L. (Hrsg.); ARNOLD, J. M. (Hrsg.): *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '98)*, IEEE Computer Society Press, Apr. 1998, S. 2–11

[59] NAAMAD, A. ; LEE, D. T.; HSU, W.-L. : On the Maximum Empty Rectangle Problem. In: *Discrete Applied Mathematics* 8 (1984), July, Nr. 3, S. 267–277

[60] PELLERIN, D. ; THIBAULT, S. : *Practical FPGA Programming in C*. 1st edition. Prentice Hall, 2005

[61] RANDELL, B. : A Note on Storage Fragmentation and Program Segmentation. In: *Communications of the ACM* 12 (1969), Nr. 7, S. 365–ff.

[62] RUPP, C. R.; LANDGUTH, M. ; GARVERICK, T. ; GOMERSALL, E. ; HOLT, H. ; ARNOLD, J. M.; GOKHALE, M. : The NAPA Adaptive Processing Architecture. In: POCEK, K. L. (Hrsg.); ARNOLD, J. M. (Hrsg.): *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '98)*, IEEE Computer Society Press, Apr. 1998, S. 28–37

[63] SCALERA, S. M.; VÁZQUEZ, J. R.: The Design and Implementation of a Context Switching FPGA. In: *Proceedings of the 6th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Napa Valley, CA : IEEE Computer Society, 1998, S. 78–85

[64] SCHMIT, H. ; WHELIHAN, D. ; TSAI, A. ; MOE, M. ; LEVINE, B. ; TAYLOR, R. R.: PipeRench: A Virtualized Programmable Datapath in 0.18 Micron Technology. In: *Proceedings of the 24th IEEE Custom Integrated Circuits Conferende (CICC)*, IEEE Computer Society, 2002, S. 63–66

[65] SEDCOLE, P. ; BLODGET, B. ; ANDERSON, J. ; LYSAGHT, P. ; BECKER, T. : Modular Partial Reconfiguration in Virtex FPGAs. In: *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL)*. Tampere, Finland : IEEE Computer Society, August 2005, S. 211–216

[66] SENG, S. P.; LUK, W. ; CHEUNG, P. Y. K.: Run-Time Adaptive Flexible Instruction Processors. In: *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications (FPL)* Bd. 2438. London, UK : Springer-Verlag, 2002 (Lecture Notes in Computer Science), S. 545–555

[67] SEPTIÉN, J. ; MECHA, H. ; MOZOS, D. ; TABERO, J. : 2D Defragmentation Heuristics for Hardware Multitasking on Reconfigurable Devices. In: *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS) - Reconfigurable Architectures Workshop (RAW)*. Rhodes, Greece : IEEE Computer Society, 2006

[68] SHOA, A. ; SHIRANI, S. : Run-time Reconfigurable Systems for Digital Signal Processing Applications: A Survey. In: *Journal of VLSI Signal Processing Systems* 39 (2005), Nr. 3, S. 213–235

[69] SIMMLER, H. ; LEVINSON, L. ; MÄNNER, R. : Multitasking on FPGA Coprocessors. In: *Field-Programmable Logic: Proceedings of the The Roadmap to Reconfigurable Computing. 10th International Workshop on Field-Programmable Logic and Applications (FPL)* Bd. 1896, Springer, 2000 (Lecture Notes in Computer Science), S. 121–130

[70] SLEATOR, D. D.; TARJAN, R. E.: Amortized efficiency of list update and paging rules. In: *Communications of the ACM* 28 (1985), Nr. 2, S. 202–208

[71] STEIGER, C. ; WALDER, H. ; PLATZNER, M. : Heuristics for Online Scheduling Real-time Tasks to Partially Reconfigurable Devices. In: *Proceedings of the 13rd International Conference on Field-Programmable Logic and Applications (FPL)* Bd. 2778, Springer, September 2003 (Lecture Notes in Computer Science), S. 575–584

[72] STEIGER, C. ; WALDER, H. ; PLATZNER, M. : Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-Time Tasks. In: *IEEE Transactions on Computers* 53 (2004), Nr. 11, S. 1393–1407

[73] TANENBAUM, A. S.: *Modern Operating Systems*. 2nd edition. Prentice-Hall, International, 2001

[74] TESSIER, R. ; BURLESON, W. : Reconfigurable Computing and Digital Signal Processing: A Survey. In: *Journal of VLSI Signal Processing Systems* 28 (2001), S. 7–27

[75] TODMAN, T. J.; CONSTANTINIDES, G. A.; WILTON, S. J. E.; MENCER, O. ; LUK, W. ; CHEUNG, P. Y. K.: Reconfigurable Computing: Architectures and Design Methods. In: *IEE Proceedings - Computers and Digital Techniques* 152 (2005), März, Nr. 2, S. 193–207

[76] TOWLES, B. ; DALLY, W. J.: Route Packets, Not Wires: On-Chip Inteconnectoin Networks. In: *38th Conference on Design Automation (DAC)*. Los Alamitos, CA, USA : IEEE Computer Society, 2001, S. 684–689

[77] TRIMBERGER, S. ; CARBERRY, D. ; JOHNSON, A. ; WONG, J. : A time-multiplexed FPGA. In: *Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM)*. Washington, DC, USA : IEEE Computer Society, 1997, S. 22–28

[78] ULLMANN, M. ; HÜBNER, M. ; GRIMM, B. ; BECKER, J. : 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)* Bd. 3203, Springer, January 2004 (Lecture Notes in Computer Science), S. 454–463

[79] VEEN, J. van d.; FEKETE, S. P.; MAJER, M. ; AHMADINIA, A. ; BOBDA, C. ; HANNIG, F. ; TEICH, J. : Defragmenting the Module Layout of a Partially Reconfigurable Device. In: *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*. Las Vegas, USA : CSREA Press, June 27-30 2005, S. 92–104

[80] VUILLEMIN, J. ; BERTIN, P. ; RONCIN, D. ; SHAND, M. ; TOUATI, H. ; BOUCARD, P. : Programmable Active Memories: Reconfigurable Systems Come of Age. In: *IEEE Transactions on VLSI Systems* 4 (1996), Nr. 1, S. 56–69

[81] WALDER, H. ; PLATZNER, M. : Non-preemptive multitasking on FPGA: Task placement and footprint transform. In: *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, CSREA Press, 2002, S. 24–30

[82] WALDER, H. ; STEIGER, C. ; PLATZNER, M. : Fast Online Task Placement on FPGAs: Free Space Partitioning and 2D-Hashing. In: *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS) - Reconfigurable Architectures Workshop (RAW)*, IEEE Computer Society, April 2003, S. 178b

[83] WIGLEY, G. ; KEARNEY, D. : The Management of Applications for Reconfigurable Computing using an Operating System. In: *Proceedings of the 7th Asia-Pacific conference on Computer Systems Architecture*. Darlinghurst, Australia, Australia : Australian Computer Society, Inc., 2002, S. 73–81

[84] WIRTHLIN, M. J.; HUTCHINGS, B. L.: A Dynamic Instruction Set Computer. In: *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*. Washington, DC, USA : IEEE Computer Society, 1995, S. 99–107

[85] XILINX INC.: *XC2000 Logic Cell Array Families*. San Jose, California, 1985

[86] XILINX INC.: *XC6200 Field Programmable Gate Arrays. Data Sheet*. San Jose, California, April 1997

[87] XILINX INC.: *Application Notes 151. Virtex Series Configuration Architecture User Guide*. San Jose, California, 2000

[88] XILINX INC.: *Virtex^{TM} 2.5 V Field Programmable Gate Arrays V2.5*. San Jose, California, April 2001

[89] XILINX INC.: *Virtex^{TM}-E 1.8 V Field Programmable Gate Arrays V2.3*. San Jose, California, July 2001

[90] XILINX INC.: *PowerPC Processor Reference Guide - Embedded Development Kit*. San Jose, California, September 2003

[91] XILINX INC.: *Application Notes 290. Two Flows for Partial Reconfiguration: Module Based or Difference Based*. San Jose, California, 2004

[92] XILINX INC.: *Virtex-II Platform FPGA User Guide V2.0*. San Jose, California, March 2005

[93] XILINX INC.: *Virtex-II Pro and Virtex-II Pro X FPGA User Guide V4.0*. San Jose, California, March 2005

[94] XILINX INC.: *MicroBlaze Processor Reference Guide V5.3*. San Jose, California, Okt. 2006

[95] XILINX INC.: *Virtex-4 User Guide V1.5*. San Jose, California, March 2006

[96] XILINX INC.: *Virtex-5 User Guide V1.2*. San Jose, California, July 2006

Eigene Veröffentlichungen

[E1] KALTE, H. ; KETTELHOIT, B. ; KOESTER, M. ; PORRMANN, M. ; RÜCKERT, U. : A System Approach for Partially Reconfigurable Architectures. In: *International Journal of Embedded Systems* 1 (2005), S. 274–290

[E2] KALTE, H. ; KOESTER, M. ; KETTELHOIT, B. ; PORRMANN, M. ; RÜCKERT, U. : A Comparative Study on System Approaches for Partially Reconfigurable Architectures. In: *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '04)*, CSREA Press, 2004, S. 70–76

[E3] KOESTER, M. : Efficient Utilization of Partially Reconfigurable Hardware. In: *Proceedings of the International Conference on Very Large Scale Integration of System-on-Chip (IFIP VLSI-SoC)*, 2005, S. 597–598. – PhD Forum Award

[E4] KOESTER, M. ; KALTE, H. ; PORRMANN, M. : Run-Time Defragmentation for Partially Reconfigurable Systems. In: *Proceedings of the International Conference on Very Large Scale Integration of System-on-Chip (IFIP VLSI-SoC)*, 2005, S. 109–115

[E5] KOESTER, M. ; KALTE, H. ; PORRMANN, M. : Task Placement for Heterogeneous Reconfigurable Architectures. In: *Proceedings of the IEEE 2005 Conference on Field-Programmable Technology (FPT'05)*, IEEE Computer Society, December 11-14 2005, S. 43–50

[E6] KOESTER, M. ; KALTE, H. ; PORRMANN, M. : Defragmentation Algorithms for Partially Reconfigurable Hardware. In: *IFIP International Federation for Information Processing Series* (2006). – zur Veröffentlichung angenommen

[E7] KOESTER, M. ; KALTE, H. ; PORRMANN, M. : Relocation and Defragmentation for Heterogeneous Reconfigurable Systems. In: *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '06)*, CSREA Press, June 27-30 2006, S. 70–76

[E8] KOESTER, M. ; PORRMANN, M. ; RÜCKERT, U. : Placement-Oriented Modeling of Partially Reconfigurable Architectures. In: *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS 2005) - Reconfigurable Architectures Workshop (RAW)*, IEEE Computer Society, 2005

[E9] RANA, V. ; SANTAMBROGIO, M. ; SCIUTO, D. ; KETTELHOIT, B. ; KOESTER, M. ; PORRMANN, M. ; RÜCKERT, U. : Partial dynamic reconfiguration in a multi-FPGA clustered architecture based on Linux. In: *Proceedings of the 21st International Parallel and Distributed Processing Symposium (IPDPS) - Reconfigurable Architectures Workshop (RAW)*, IEEE Computer Society, 2007

Abbildungsverzeichnis

Tabellenverzeichnis

Glossar

α ..... Relative Verfügbarkeit: Verhältnis der Anzahl der Zellen des größten maximalen freien Rechtecks zur Anzahl aller freien Zellen.

δ_b ..... Beidseitiger Zellzusammenhang.

δ_o ..... Orthogonaler Zellzusammenhang.

δ_r ..... Rechtsseitiger Zellzusammenhang.

η ..... Adaptierungsrate des adaptiven Auswahlgewichts.

γ ..... Seitenverhältnis eines im 2D-Systemansatz synthetisierten Hardware-Moduls $m \in M$.

ν ..... Zellabweisung: Anzahl der benötigten Zellen der abgewiesenen Komponentenanfragen im Verhältnis zur Anzahl der benötigten Zellen aller Komponentenanfragen.

ϕ_{int} ..... Interne Fragmentierung: Verhältnis der Anzahl der ungenutzten Zellen innerhalb der Hardware-Module zur Anzahl der genutzten Zellen der Hardware-Module.

σ ..... Liste von Komponentenanfragen (Systemkomponenten) der virtuellen Anwendung in der Simulationsumgebung SARA.

σ_{PP} ..... Liste von Objekten eines Online-Packungsproblems.

υ ..... Ressourcenauslastung: Verhältnis der Anzahl aktiver Zellen zur Anzahl aller auf der Architektur vorhandenen Zellen.

a ..... Flächenbedarf eines Hardware-Moduls $m \in M$.

a_h ..... Horizontale Flächenausdehnung eines Hardware-Moduls $m \in M$.

a_v ..... Vertikale Flächenausdehnung eines Hardware-Moduls $m \in M$.

a_{eh} ..... Horizontale Flächenausdehnung eines freien Rechtecks.

a_{ev} ..... Vertikale Flächenausdehnung eines freien Rechtecks.

B ..... Box (Teilmenge) eines Online-Packungsproblems.

b ..... Zellbelegung der rekonfigurierbaren Architektur.

b_{FCFS} ..... Zellbelegung unter Berücksichtigung der Modulinstanzen, die in dem Zustand PLA, CFG oder EXE sind.

C ..... Menge der momentan platzierten Modulinstanzen.

c ..... Modulinstanz (kurz: Instanz): Ein auf der rekonfigurierbaren Architektur abgebildetes Hardware-Modul $m \in M$.

C_{defrag} .... Menge der Modulinstanzen, die sich komplett innerhalb des Defragmentierungssegments befinden.

C_{int} ..... Menge der überschneidenden Modulinstanzen.

c_{PLA} ..... Modulinstanz im Zustand $s = PLA$.

c_{TRM} ..... Modulinstanz im Zustand $s = TRM$.

C_{use} ..... Menge der Modulinstanzen, welche Zellen zwischen der Spalte x_{ul} und der Spalte x_{ur} verwenden.

D ..... Menge der Systemkomponenten.

d ..... Systemkomponente (kurz: Komponente): Abstrakte Spezifikation eines Hardware-Entwurfs.

d_i ..... Komponentenanfrage (Systemkomponente) in der Liste der Komponentenanfragen σ.

D_{rej} ..... Menge der abgewiesenen Komponentenanfragen einer Simulation.

D_{REQ} ..... Menge der unplatzierten Komponentenanfragen.

e ..... Freies Rechteck: Rechteckige Fläche von unbelegten Zellen der rekonfigurierbaren Architektur.

E_{all} ..... Menge aller freien Rechtecke.

e_{best} ..... Größtes maximales freies Rechteck der aktuellen Zellbelegung.

E_{free} ..... Menge der für die Platzierung eines geforderten Hardware-Moduls geeigneten freien Rechtecke.

E_{max} ..... Menge der maximalen freien Rechtecke.

$f_{SelectMap}$ .. Taktfrequenz der Konfigurationsschnittstelle.

k ..... Kommunikationsparameter einer Modulinstanz $c \in C$.

M ..... Menge der Hardware-Module.

m ..... Hardware-Modul (kurz: Modul): Vorsynthetisierte Implementierung einer Systemkomponente $d \in D$ für eine rekonfigurierbare Architektur.

M_{free} ..... Menge aller platzierbaren Hardware-Module.

M_{pla} ..... Menge der platzierbaren Module einer Systemkomponente $d \in D$.

N_σ ..... Anzahl der Komponentenanfragen (Systemkomponenten) in der Liste σ.

N_B ..... Anzahl der Boxen (Teilmengen) eines Online-Packungsproblems.

N_{cell} ..... Anzahl der verschiedenen Zelltypen der rekonfigurierbaren Architektur.

N_{col} ..... Anzahl der Spalten der rekonfigurierbaren Architektur.

N_{com} ..... Anzahl der verschiedenen Kommunikationsparameter einer Modulinstanz $c \in C$.

N_{cost} ..... Anzahl der verschiedenen Kostenparameter eines Hardware-Moduls $m \in M$.

N_{PP} ..... Anzahl der Objekte in der Liste σ_{PP} eines Online-Packungsproblems.

N_{PR} ..... Maximale Priorität einer Komponentenanfrage.

N_{res} ..... Anzahl der benötigten Zellen einer Systemkomponente $d \in D$.

N_{row} ..... Anzahl der Reihen der rekonfigurierbaren Architektur.

N_{seg} ..... Anzahl der Blöcke, in denen Hardware-Module im Systemansatz mit fester Aufteilung platziert werden können.

N_{sel} ..... Anzahl der geforderten Modulplatzierungen.

N_{sim} ..... Simulationslänge: Anzahl von Zeiteinheiten der Simulation in der Simulationsumgebung SARA.

N_{sys} ..... Dimension des zugrunde liegenden Systemansatzes.

N_{toprow} .... Anzahl der Reihen, die unmittelbar über mindestens einer Modulinstanz liegen.

o ..... Objekt in einem Online-Packungsproblem.

$o_{pos,dyn}$ .... Dynamische Positionsüberlappung: Anzahl der freien Positionen eines Hardware-Moduls $m \in M$, welche die Belegung der Zelle an Position (x_h, x_v) verursachen.

$o_{pos,st}$ ..... Statische Positionsüberlappung: Anzahl der möglichen Positionen eines Hardware-Moduls $m \in M$, welche die Belegung der Zelle an Position (x_h, x_v) verursachen.

p_{req} ..... Anfragewahrscheinlichkeit: Wahrscheinlichkeit einer Komponentenanfrage pro Zeiteinheit.

p_{sel} ..... Auswahlwahrscheinlichkeit: Wahrscheinlichkeit, mit der eine Systemkomponente bei einer auftretenden Komponentenanfrage ausgewählt wird.

q ..... Implementierungskosten eines Hardware-Moduls $m \in M$.

r ..... Anzahl der benötigten Zellen eines Hardware-Moduls $m \in M$.

S ..... Menge der Zustände einer Modulinstanz $c \in C$.

s ..... Aktueller Zustand einer Modulinstanz $c \in C$.

s_t ..... Treppe: Menge aller maximalen freien Rechtecke mit gemeinsamer unteren rechten Ecke.

$t_{BOC-EOP}$ .. Zeitspanne zwischen dem Platzierungsende und dem Konfigurationsbeginn einer Modulinstanz $c \in C$.

t_{BOC} ..... Konfigurationsbeginn: Zeitpunkt, in dem mit der Konfiguration des Hardware-Moduls begonnen wird.

t_{BOD} ..... Löschbeginn: Zeitpunkt, in dem mit dem Zurücksetzen der Zellen der Modulinstanz begonnen wird.

t_{BOE} ..... Ausführungsbeginn: Zeitpunkt, in dem die Ausführung einer Modulinstanz beginnt.

t_{BOP} ..... Platzierungsbeginn: Zeitpunkt, in dem die Platzierungsablaufplanung die Komponentenanfrage der Modulplatzierung übergibt.

t_{CFG} ..... Konfigurationszeit: Zeit, die für die Konfiguration eines Hardware-Moduls auf der rekonfigurierbaren Architektur benötigt wird.

t_{END} ..... Freigabezeitpunkt: Zeitpunkt, in dem die Modulinstanz aus der Menge der momentan platzierten Modulinstanzen C entfernt wird.

t_{EOC} ..... Konfigurationsende: Zeitpunkt, in dem die Konfiguration des Hardware-Moduls endet.

t_{EOD} ..... Löschende: Zeitpunkt, in dem alle Zellen der Modulinstanz zurückgesetzt sind.

t_{EOE} ..... Ausführungsende: Zeitpunkt, in dem die Ausführung einer Modulinstanz endet.

t_{EOP} ..... Platzierungsende: Zeitpunkt des Endes einer Modulplatzierung.

t_{EXE} ..... Ausführungszeit einer Systemkomponente in der Liste der Komponentenanfragen σ.

t_{PLA} ..... Platzierungszeit: Die für die Platzierung des Hardware-Moduls benötigte Zeit.

t_{RELOC} .... Umplatzierungszeit: Benötigte Zeit zur Umplatzierung einer Modulinstanz.

t_{REQ} ..... Platzierungsanfrage: Anfragezeitpunkt einer Komponentenanfrage.

T_{sim} ..... Simulationszeit: Dauer einer Simulation in der Simulationsumgebung SARA.

t_{sim} ..... Simulationstakt: Takt der virtuellen Anwendung in der Simulationsumgebung SARA.

$w_{cell,dyn}$ ... Dynamisches Zellgewicht.

$w_{cell,st}$ ..... Statisches Zellgewicht.

$w_{pos,dyn}$ ... Dynamisches Positionsgewicht: Quadratischer Mittelwert der dynamischen Zellgewichte der infolge einer Modulplatzierung belegten Zellen.

$w_{pos,st}$ ..... Statisches Positionsgewicht: Quadratischen Mittelwert der statischen Zellgewichte der infolge einer Modulplatzierung belegten Zellen.

w_{PR} ..... Prioritätswert einer Komponentenanfrage.

w_{sel} ..... Adaptive Auswahlgewicht eines Hardware-Moduls $m \in M$.

x ..... Position einer Modulinstanz $c \in C$.

x_h ..... Horizontale Position einer Modulinstanz $c \in C$.

x_v ..... Vertikale Position einer Modulinstanz $c \in C$.

x_{dl} ..... Position der Spalte, die die linke Grenze des Defragmentierungssegments markiert.

x_{dr} ..... Position der Spalte, die die rechte Grenze des Defragmentierungssegments darstellt.

x_{eh} ..... Horizontale Position eines freien Rechtecks.

x_{ev} ..... Vertikale Position eines freien Rechtecks.

X_{free} ..... Menge der freien Positionen eines Hardware-Moduls $m \in M$.

X_{pos} ..... Menge der möglichen Positionen eines Hardware-Moduls $m \in M$.