TECHNISCHE
UNIVERSITÄT
DARMSTADT

Auswahl und Implementierung von CGRA-Makrooperationen

Bachelorarbeit von

Johannes Wirth

Februar 2018

Erstgutachter:
Prof. Dr.-Ing. Andreas Koch

Zweitgutachter:
Dr.-Ing. Andreas Engel

Technische Universität Darmstadt
Department of Computer Science
Embedded Systems and Applications Group (ESA)

Auswahl und Implementierung von CGRA-Makrooperationen

Bachelorarbeit von Johannes Wirth
Eingereicht am 12.02.2018
Erstgutachter: Prof. Dr.-Ing. Andreas Koch
Zweitgutachter: Dr.-Ing. Andreas Engel

Eigenständigkeitserklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs.2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden. Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung überein.

Darmstadt, 12. Februar 2018

(Johannes Wirth)

Kurzfassung

Im Rahmen des UltraSynth-Projekts wird, in Zusammenarbeit mit mehreren Unternehmen, eine Toolchain zur Generierung der eingebetteten Implementierung für einen Coarse-Grain Reconfigurable Array (CGRA) aus einem mechatronischen System implementiert.

Das Systemmodell wird dabei vor dem Generieren der eigentlichen Implementierung als Datenflussgraph (DFG) dargestellt. Aufgrund des Aufbaus des CGRAs muss jede Operation vor der eigentlichen Berechnung die benötigten Daten laden und im Anschluss an die Berechnung das Ergebnis schreiben. Dies stellt vor allem bei einfachen Operationen einen erheblichen IO-Overhead dar. Ein Teil dieses Overheads kann durch das Ersetzen einer Gruppe von Operationen durch eine sogenannte Makro-Operation (Makro-Op) eingespart werden.

In Rahmen dieser Arbeit wurde die notwendige Funktionalität für das Ersetzen eines Subgraphen durch eine Makro-Op implementiert. Außerdem wurden zwei Analysen zum Suchen geeigneter Makro-Ops in einem DFG entwickelt: Eine erschöpfende Suche findet alle möglichen Ersetzungen, benötigt aber viel Arbeitsspeicher und Laufzeit. Die Heuristik schränkt den Suchraum auf vermutlich sinnvolle Makro-Ops ein, wodurch die Suche deutlich schneller läuft und weniger Speicher benötigt.

In der Auswertung wurden die beiden Analysen detaillierter verglichen: Die Heuristik läuft mindestens um Faktor 4 schneller; in vielen Fällen ist der Beschleunigungsfaktor sogar noch deutlich größer. Sie schränkt die gefundenen Ersetzungsmöglichkeiten bei realen DFGs kaum ein und die wenigen fehlenden Ergebnisse dürften in der Regel vernachlässigbar sein.

Außerdem wurden auch die von den Analysen vorgeschlagenen Ersetzungen untersucht: Die am besten bewerteten Ergebnisse sind meist Makro-Ops, deren Einsparung vergleichsweise gering ist, die aber häufig im DFG vorkommen. Dies hat den Vorteil, dass diese im CGRA unterstützte Makro-Op mehrmals verwendet werden kann. Auch das Ersetzen der Integratoren, die in jedem DFG vorkommen, erzielt meist gute Einsparungen. Ein CGRA mit dieser Makro-Op kann also für die Ausführung aller DFGs genutzt werden.

Inhaltsverzeichnis

1. Einleitung

Eingebettete Systeme spielen eine immer größere Rolle – unter anderem auch im Automobil- und Maschinenbau. In diesem Bereich werden sie beispielsweise für die Simulation von mechatronischen Systemmodellen (etwa eine Fahrzeugachse) eingesetzt, um modellbasierte Regler umzusetzen.

Solche eingebetteten Systeme bestehen aktuell häufig aus einer Kombination aus Central Processing Unit (CPU) und Field-Programmable Gate Array (FPGA), welche auf den konkreten Anwendungsfall zugeschnitten wurde.

Das Ziel des UltraSynth-Projekts [1] ist die Entwicklung einer kompletten Toolchain, die für ein mechatronisches System automatisch die eingebettete Implementierung erzeugt.

Der Ablauf dieser Toolchain ist in Abbildung 1.1 dargestellt und wird im Folgenden erläutert.

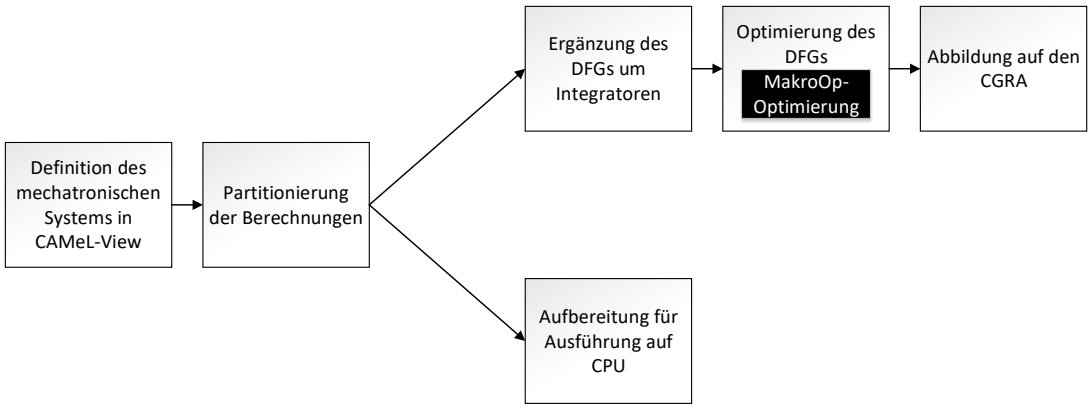

Abbildung 1.1.: UltraSynth Toolchain.

Zu Beginn wird das Modell im CAMeL-View-Programm [2] durch miteinander verbundene Funktionsblöcke dargestellt, welche periodisch ausgewertet werden. Diese Blöcke bestehen aus konstanten Parametern, Eingängen, Ausgängen, internen Zuständen und Hilfsvariablen. Der Folgezustand von Hilfsvariablen wird durch normale

Berechnungsvorschriften angegeben; für interne Zustände wird stattdessen die zeitliche Ableitung berechnet und gespeichert.

Im ersten Schritt werden die durch dieses System definierten Berechnungen partitioniert: Die performance-kritischen Teile werden in einen Datenflussgraph (DFG) (siehe Abschnitt 2.2) umgewandelt, um später auf dem FPGA-Teil ausgeführt zu werden. Der Rest wird für die Ausführung auf der eingebetteten CPU aufbereitet.

Nun wird der DFG um Integratoren ergänzt. Die Integratoren werden benutzt, um anhand der zeitlichen Ableitungen der internen Zustände den entsprechenden Wert für den nächsten Zeitschritt zu bestimmen. Dies ist beispielhaft in Abbildung 1.2 dargestellt: Auf der linken Seite befindet sich ein einfaches System vor der Ergänzung der Integratoren, welches nur den internen Zustand `val` besitzt. Dessen zeitliche Ableitung ist durch `val' = const - val` gegeben. Der gestrichelte Pfeil stellt den Übergang zwischen den Integrationszyklen dar. Diese implizite Abhängigkeit wurde auf der rechten Seite durch einen Integrator (hier Euler) aufgelöst. Auf den Wert des internen Zustands wird mithilfe der LOAD-Operation zugegriffen. Nach der eigentlichen Berechnung erfolgt der Integrationsschritt, welcher den Zustand für den nächsten Zyklus liefert. Der neue Zustand wird dann mithilfe der STORE-Operation gesichert. Nach dem Durchlauf des kompletten DFGs startet die Berechnung mit diesem aktualisiertem Zustand erneut.

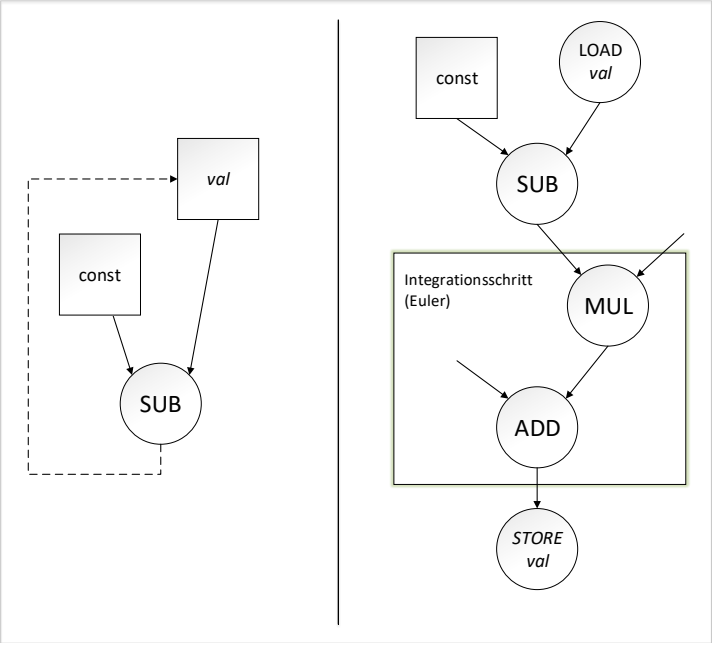

Abbildung 1.2.: Graph mit und ohne Integrator.

Nach der Ergänzung um Integratoren wird der DFG analysiert und optimiert. Hier findet auch die in dieser Arbeit beschriebene Makro-Operation (Makro-Op)-Optimierung

statt. Andere Optimierungen beziehungsweise Analysen sind beispielsweise Dead-Code-Elimination, Common-Subexpression-Elimination und Bitwidth-Analysis.

Der resultierende DFG wird auf einen Coarse-Grain Reconfigurable Array (CGRA) [3, 4] (siehe Abschnitt 2.1) abgebildet. Dafür legt ein Scheduler eine Zuordnung der verschiedenen Knoten auf die zur Verfügung stehenden Processing Elements (PEs) und die zeitliche Abfolge der Ausführung der Operationen auf den PEs fest. Im Gegensatz zur Abbildung auf einen FPGA ist die Komplexität deutlich geringer, was durch das niedrigere Level der Rekonfigurierbarkeit des CGRAs begründet ist. Ein FPGA ist auf Bit-Level rekonfigurierbar, ein CGRA nur auf Word-Level. Der Vorteil dieser eingeschränkten Rekonfigurierbarkeit besteht in der kleineren Menge an Rekonfigurationsinformationen und damit verbunden der kleineren Menge an Rekonfigurationsentscheidungen. Da das Ausführungsmodell des CGRAs sehr gut zum DFG passt, entsteht in diesem Fall trotzdem keine wesentliche Einschränkung der Flexibilität und der Performance. Zum jetzigen Zeitpunkt wird der CGRA dann meist mithilfe eines FPGAs emuliert – es ist aber zum Beispiel auch die Verwendung eines speziellen ASICs möglich. Diese Emulation ist daher auch kein Teil des UltraSynth-Projektes.

Die vorliegende Arbeit beschäftigt sich mit einer konkreten Optimierung des DFGs vor der Abbildung auf den CGRA. In Abbildung 1.3 ist der Aufbau eines PEs dargestellt. Es ist zu erkennen, dass vor der Ausführung einer Berechnung erst die benötigten Operanden in die Arithmetisch-logische Einheit (ALU) geladen werden müssen. Diese Operanden können entweder aus dem Registerfeld des PEs oder von außerhalb, also von einem anderen PE, stammen. Im Anschluss an die Berechnung werden die Ergebnisse in das Registerfeld zurückgeschrieben.

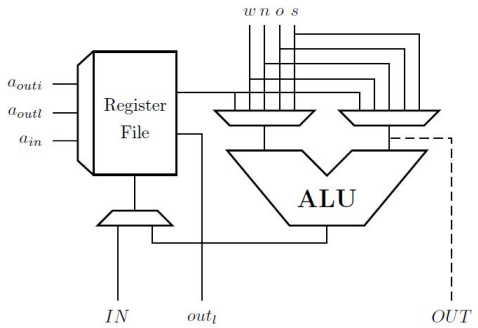

Abbildung 1.3.: Aufbau eines PEs (aus [4]).

Dies stellt vor allem bei kombinatorischen, aber auch bei sequentiellen Operationen einen erheblichen IO-Overhead dar. Die im weiteren Verlauf beschriebene Optimierung spart einen Teil dieses Overheads ein, indem sie mehrere zusammen-

hängende Operationsknoten zu einer Makro-Op zusammenfasst. Sie trägt daher den Namen Makro-Op-Optimierung. Durch das Zusammenfassen der Operationen müssen nur noch die Daten, welche von außerhalb der Makro-Op in diese hineinfließen, in die ALU geladen werden (analog müssen auch alle die Makro-Op verlassenden Daten in das Registerfeld geschrieben werden). Bei allen Daten, die nur innerhalb der Makro-Op zwischen den verschiedenen Teil-Operationen fließen, kann sowohl das Speichern durch die produzierende Operation als auch das erneute Laden durch die konsumierende Operation eingespart werden – es werden also mehrere Zugriffe auf das Registerfeld des PEs vermieden.

Um diese Optimierung einfach nutzbar zu machen, wurde der Toolchain des UltraSynth-Projekts Funktionalität für das Ersetzen einer Gruppe von Knoten durch eine Makro-Op hinzugefügt.

Damit der CGRA die entstandenen Makro-Ops ausführen kann, muss jeweils mindestens eine PE die entsprechende Implementierung enthalten. Das Ersetzen durch Makro-Ops verursacht daher drei offensichtliche Einschränkungen beziehungsweise Probleme:

Erstens kann eine Makro-Op nicht aus unbegrenzt vielen Teiloperationen bestehen, da die Implementierung der Operation ansonsten zu groß wird und das PE zu viel Fläche belegt. Auch die Komplexität der Teiloperationen spielt dabei eine Rolle: Bei einfacheren Operationen können mehr Teiloperationen zusammengefasst werden, als bei komplexen Operationen.

Desweiteren ist die maximale Taktfrequenz des CGRAs durch den längsten kritischen Pfad aller, in mindestens einer PE unterstützten Operationen bestimmt. Falls die Taktfrequenz in Folge der Ersetzung durch eine Makro-Op reduziert werden muss, kann dies negative Auswirkungen auf die Gesamtausführungszeit des DFGs haben. Dieses Problem kann vor allem bei langen Ketten von kombinatorischen Operationen auftreten.

Drittens ist es nicht sinnvoll, die Knoten des DFG in möglichst viele Gruppen einer bestimmten Größe einzuteilen und alle diese Gruppen durch Makro-Ops zu ersetzen. Die Menge an unterschiedlichen Makro-Ops, welche die PEs dann unterstützen müssten, würde deren Größe explodieren lassen.

Vor allem aus letzterem Grund folgt, dass eine Auswahl getroffen werden muss. Die gewählten Ersetzungen sollten eine möglichst große Laufzeitbeschleunigung bei akzeptabler Erhöhung der benötigten Logikressourcen bieten. Bei den auftretenden Größen des DFGs von Hunderten oder Tausenden Knoten ist dies händisch nicht möglich. Es wird daher eine Analyse benötigt, welche automatisch nach sinnvollen Makro-Ops sucht. Vor allem die oben beschriebenen Nachteile und Probleme einer Makro-Op kann diese Analyse allerdings nicht vollständig abschätzen. Daher schlägt

die Analyse nur mögliche Ersetzungen vor, welche dann durch den Anwender genauer untersucht werden können.

Im Folgenden werden die vorgenommenen Änderungen im Detail beschrieben. Die Arbeit ist dabei in folgende Teile gegliedert: In Kapitel 2 werden einige theoretische Grundlagen eingeführt und Besonderheiten im Rahmen dieser Arbeit beziehungsweise des UltraSynth-Projektes erläutert. Außerdem werden einige im weiteren Verlauf der Arbeit benötigten Begriffe definiert.

In Kapitel 3 werden einerseits Arbeiten vorgestellt, die für die Implementierung der Makro-Op-Optimierung notwendig sind. Dies betrifft vor allem detaillierte Informationen zum CGRA und zum zugehörigen Scheduler. Zum Anderen werden Ansätze betrachtet, die der Makro-Op-Optimierung ähnliche sind. Dabei liegt ein besonderer Fokus auf den unterschiedlichen Rahmenbedingungen und darauf, welche Ideen und Ansätze trotzdem auch im Rahmen des UltraSynth-Projekts anwendbar sind.

Die eigentliche Implementierung der Ersetzung und der Analysen, dabei aufgetretene Probleme und Schwierigkeiten und deren jeweilige Lösungen werden in Kapitel 4 behandelt.

In Kapitel 5 werden die Vor- und Nachteile der Optimierung analysiert. Ein besonderer Schwerpunkt liegt dabei einerseits auf den erzielten (zeitlichen) Einsparungen durch das Zusammenfassen der Operationen und andererseits auf dadurch entstehende negative Effekte. Auch die beiden implementierten Analysen werden hinsichtlich der gefundenen Makro-Ops, ihrer Laufzeit und ihres Speicherverbrauchs verglichen. Anschließend werden diese Ergebnisse in Kapitel 6 diskutiert und in Kontext gesetzt.

Abschließend werden in Kapitel 7 die wichtigsten Erkenntnisse zusammengefasst. Auch die Einschränkungen der aktuellen Implementierung werden betrachtet und mögliche Erweiterungs- und Verbesserungsmöglichkeiten diskutiert.

2. Theoretische Grundlagen

In diesem Kapitel werden die zum Verständnis der weiteren Arbeit essentiellen Grundlagen eingeführt: CGRAs, DFGs und Integratoren.

Ein besonderer Fokus liegt dabei auf den Besonderheiten und Einschränkungen, welche im Rahmen des UltraSynth-Projektes gelten und Voraussetzungen für die MakroOp-Optimierung sind, beziehungsweise Auswirkungen auf diese haben.

Außerdem werden einige Begriffe vorgestellt, welche im weiteren Verlauf der Arbeit häufig verwendet werden. Dies können sowohl allgemein benutzte Begriffe sein, welche der Vollständigkeit halber aufgeführt werden, als auch Begriffe die speziell für die Makro-Op-Ersetzung benötigt werden.

2.1. CGRA

Ein *Coarse-Grain Reconfigurable Array (CGRA)* ist eine rekonfigurierbare Hardwareeinheit, welche die Parallelisierung von Berechnungen auf Instruktionsebene erlaubt. Er besteht aus einer Menge von miteinander verbundenen *PEs*. Eine beispielhafte Struktur ist in Abbildung 2.1 dargestellt. Die PEs enthalten jeweils eine ALU und ein Registerfeld. Der Aufbau eines PEs ist in Abbildung 2.2 skizziert. Die PEs des CGRAs können sich dabei bezüglich der unterstützten Instruktionen unterscheiden. Die Gesamtmenge aller in mindestens einer PE unterstützten Instruktionen wird als *Instruktionssatz* des CGRAs bezeichnet.

In der Regel hat ein CGRA eine einheitliche *Datenpfadbreite*, welche die Bitbreite des Verbindungsnetzwerkes zwischen den PEs und die Verbindungsbreite innerhalb der PEs (zum Beispiel zwischen ALU und Registerfeld) angibt. Oft bestimmt die Datenpfadbreite auch die Operationsbreite der ALU.

Im Gegensatz zu einem FPGA ist ein CGRA auf Word-Level beziehungsweise Instruction-Level statt auf Bit-Level rekonfigurierbar. Durch den reduzierten Freiheitsgrad ist die Konfiguration kompakter und lässt sich zudem schneller erstellen. Für Letzteres wird ein auf den CGRA abgestimmter Scheduler verwendet, der den sogenannten Ablaufplan erstellt.

Der Scheduler bekommt als Eingabe in den meisten Fällen einen DFG. Jeder Operations-Knoten des DFGs wird beim Scheduling dann einer PE in Kombina-

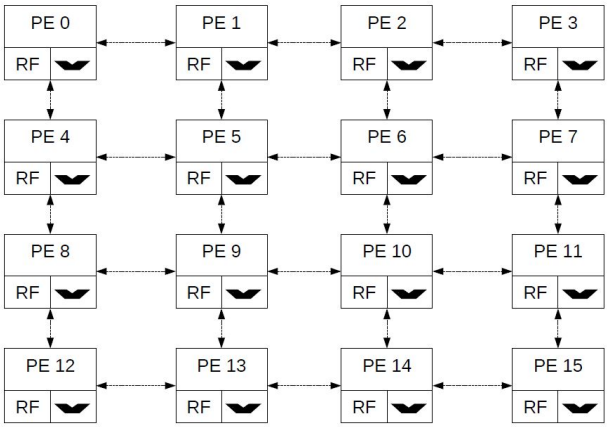

Abbildung 2.1.: Aufbau eines CGRAs (aus [3])

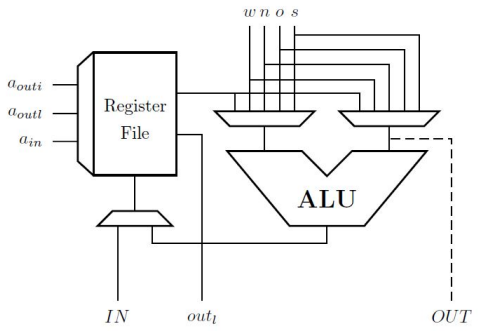

Abbildung 2.2.: Aufbau eines PEs (aus [4])

tion mit einem Zeitpunkt der Ausführung zugewiesen. Für jede PE wird also im Ablaufplan festgelegt, wann sie welche Operation ausführen muss und woher die Operanden kommen. Außerdem wird auch der Austausch der Daten zwischen den PEs konfiguriert. Diese Konfigurationsdaten werden auf den CGRA geladen und in sogenannten Kontextspeichern gehalten. Zur Laufzeit lädt dann jede PE taktweise ihre jeweilige Konfiguration.

Beim im UltraSynth-Projekt verwendeten CGRA besitzt die ALU jedes PEs genau zwei Eingangsports und einen Ausgangsport. Dies hat zur Folge, dass pro Takt maximal die doppelte Datenpfadbreite in die ALU geladen und maximal die einfache Datenpfadbreite aus dieser geschrieben werden kann. Bei größeren Datenmengen muss das Lesen beziehungsweise Schreiben auf mehrere Takte verteilt werden.

Eine Operation kann in jedem Takt mehrere Aktionen gleichzeitig ausführen, zum Beispiel das Laden der Eingänge parallel zum ersten Teil der Berechnung oder den letzten Teil der Berechnung parallel zum Schreiben. Vorraussetzung ist nur, dass die

jeweils benötigten Daten schon während des Taktes bereitstehen. Solange dies der Fall ist, können sich die Phasen beliebig überschneiden.

Da der CGRA eine synchrone Schaltung ist, benötigt er eine Taktfrequenz. Die maximale Taktfrequenz wird wie üblich aus dem längsten kritischen Pfad der Schaltung bestimmt. Dies ist hier in der Regel der längste kritische Pfad aller unterstützten Instruktionen. Die unterstützten Instruktionen können dabei sowohl kombinatorisch als auch sequentiell sein. Bei sequentiellen Instruktionen ist für die Taktfrequenz nur der kritische Pfad ihres längsten Takts relevant. Wenn sich die Berechnung mit dem Lesen beziehungsweise Schreiben überschneidet, fließt auch die dafür benötigte Zeit in den kritischen Pfad dieses Takts der Instruktion ein.

2.2. Datenflussgraphen

2.2.1. Allgemein

Ein Datenflussgraph (DFG) ist ein gerichteter Graph; seine Knoten stellen Operationen dar. Die Kanten bilden Datenabhängigkeiten oder Kontrollabhängigkeiten der Operationen untereinander ab. Für diese Arbeit sind nur die Datenabhängigkeiten interessant, daher werden im Folgenden nur diese Kanten betrachtet. Eingehende Kanten eines Knotens stellen Daten dar, die der Knoten für seine Operation benötigt. Eine ausgehende Kante steht für ein Ergebnis dieser Operation. Ein DFG wird verwendet, um komplexe Berechnungen übersichtlich darzustellen und Analysen sowie Optimierungen zu vereinfachen. Die Anzahl der ein- beziehungsweise ausgehenden Kanten je Knoten ist beliebig. Insbesondere sind auch Knoten ohne ein- oder ausgehende Kanten möglich – deren Operation kann dann beispielsweise ein Speicherzugriff sein.

2.2.2. Notation

Im Rahmen der Makro-Op-Optimierung werden einige Begriffe benötigt, die entweder nicht allgemein definiert sind oder mit verschiedenen Bedeutungen verwendet werden. Im Folgenden wird daher beschrieben, mit welcher Bedeutung diese Begriffe im Rest der Arbeit verwendet werden.

Ein *Knoten* des DFGs führt eine bestimmte *Operation* beziehungsweise *Instruktion* aus. Diese Operation kann zum Beispiel mathematischer oder logischer Natur sein. Für die Ausführung auf dem CGRA steht eine *Implementierung* in Verilog bereit. Als *Berechnung* wird die Operation ohne das Lesen der Operanden und das Schreiben der Ergebnisse bezeichnet.

Angenommen, Knoten A besitzt eine Kante zu Knoten B. Knoten A wird dann als *Vorgänger* von B bezeichnet, analog wird B als *Nachfolger* von A bezeichnet. Die

Kante stellt einen *Ausgang* der Operation von A und einen *Eingang* der Operation von B dar. Daten, welche über diese Kanten wandern, werden als *produzierte Daten* von A und als *konsumierte* oder *benötigte Daten* von B bezeichnet. Bezogen auf die Daten ist A die *Quelloperation* beziehungsweise der *Quellknoten*, B die *Zieloperation* oder der *Zielknoten*. Im Allgemeinen ist die Reihenfolge der Eingänge einer Operation relevant, eine Ausnahme bilden hier nur kommutative Operationen.

Eine Gruppe von Knoten wird auch als *Subgraph* bezeichnet. Die *Nachfolger* eines Subgraphen sind alle Nachfolger seiner Teilknoten, die nicht Teil des Subgraphen sind. Analog sind die *Vorgänger* eines Subgraphen alle Vorgänger seiner Teilknoten, die nicht in diesem enthalten sind. Als *Nachbarknoten* eines Subgraphen wird die Vereinigung seiner Vorgänger und Nachfolger bezeichnet.

Alle Knoten eines Subgraphen, die Daten von Knoten außerhalb des Subgraphen konsumieren, werden als *Eingangsknoten* bezeichnet. Analog werden alle Knoten, welche Daten für Knoten außerhalb produzieren, *Ausgangsknoten* genannt. Die Vorgängerknoten eines Subgraphen werden auch als *Operandenknoten* bezeichnet. Eingangsknoten, welche ausschließlich Daten von außerhalb des Subgraphen konsumieren, werden als *reine Eingangsknoten* bezeichnet. Diese Bezeichnungen sind in Abbildung 2.3 veranschaulicht.

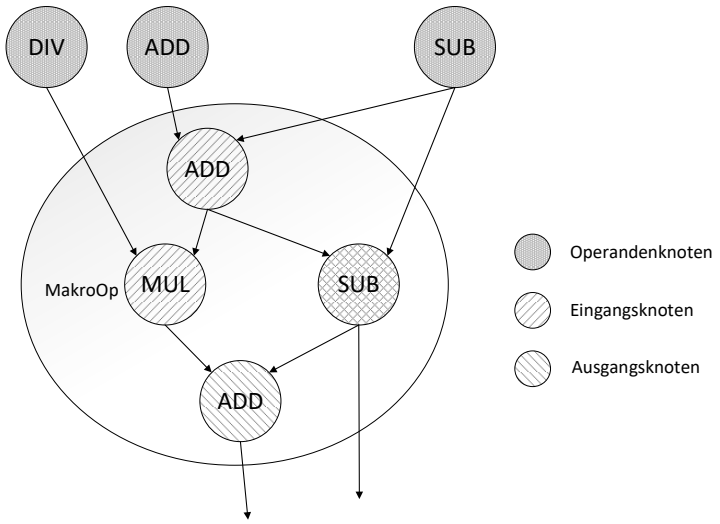

Abbildung 2.3.: Begrifflichkeiten

Eine *Makro-Op* ist ein Knoten, welcher einen Subgraphen des ursprünglichen DFGs ersetzt. Seine Operation muss daher alle Teiloperationen des ersetzten Subgraphen durchführen, um dieselbe Gesamtfunktionalität bereitzustellen. Die ersetzten

Knoten werden als *Teilknoten* oder *Teiloperationen* der Makro-Op bezeichnet. *Op-Graph* (Abschnitt 4.1.1) wird im Folgenden als Synonym für Makro-Op verwendet.

Eine einzelne Makro-Op kann mehrmals in einem DFG benutzt werden, wenn ihre Gesamtfunktionalität mehrmals in diesem benötigt wird. Eine einzelne Instanz einer Makro-Op wird als *Vorkommen* bezeichnet.

2.2.3. Besonderheiten und Einschränkungen

Im Rahmen des UltraSynth-Projekts gibt es einige Einschränkungen bezüglich der Knoten und der Struktur des DFGs. Diese vereinfachen die Ersetzung durch Makro-Ops teilweise deutlich oder machen die Ersetzung überhaupt erst möglich und sind daher für das Verständnis unverzichtbar. Diese Einschränkungen werden im Folgenden erläutert:

Der DFG kann keine Zyklen enthalten; es ist also nicht möglich, einen Knoten auf einem Pfad durch den Graphen mehrmals zu erreichen. Die Rückkopplung wird implizit durch den Integrationsschritt (Abschnitt 2.3) vollzogen.

Der Datenfluss findet dementsprechend nur in eine Richtung statt – die Darstellung von oben nach unten ist intuitiv, aber nicht notwendig.

Von der PE-Architektur und vom eingesetzten Scheduler werden aktuell nur Knoten mit maximal zwei Eingängen und einem Ausgang unterstützt. Alle Nicht-Makro-Op-Knoten im DFG richten sich daher nach dieser Einschränkung. Makro-Ops ergeben mit dieser geringen Anzahl allerdings kaum Sinn, daher gilt diese Einschränkung für die neu erstellten Knoten nicht. Zumindest der Scheduler muss daher vor dem Einsatz von Makro-Ops noch angepasst werden.

Die Operation eines Knotens beinhaltet neben deren Verilog-Implementierung auch Informationen über deren Zeitverhalten (Abschnitt 2.2.4) und die Ein- und Ausgabeformate. Die Formate beschreiben dabei sowohl die Bitbreite als auch die Interpretation der Daten, beispielsweise ob es sich um Gleitkommazahlen oder Ganzzahlen handelt.

Die beiden Eingangsformate eines Nicht-Makro-Op-Knotens besitzen aktuell immer die gleiche Bitbreite. Da dies an einigen Stellen eine Vereinfachung darstellt, wurde diese Einschränkung für die Arbeit als gegeben angenommen. Eine Anpassung für unterschiedlich breite Eingangsformate ist aber möglich.

2.2.4. Zeitverhalten

Die Gesamtlatenz einer Operation setzt sich aus drei Teilen zusammen: *Eingabelatenz*, *Berechnungslatenz* und *Ausgabelatenz*. Die Eingabelatenz beschreibt die Anzahl an Takten, welche benötigt werden, um alle Eingabedaten der Operation einzulesen. Bei Nicht-Makro-Op-Knoten ist dies die Anzahl an Takten, die das Lesen eines

Eingangs benötigt. Dies entspricht der Bitbreite des Eingangsformats geteilt durch die Datenpfadbreite des DFGs (aufgerundet).

Analog dazu entspricht die Ausgabelatenz der Bitbreite des Ausgangsformats geteilt durch die Datenpfadbreite. Die Berechnungslatenz ist die Anzahl an Takten, die die Operation für das Verarbeiten der Daten benötigt. Eine Berechnungslatenz von eins bedeutet, dass die Berechnung im selben Takt abgeschlossen ist, in dem sie begonnen wurde. Es beschreibt also eine kombinatorische Operation. Berechnungslatenzen größer als eins beschreiben dementsprechend sequentielle Operationen: Eine Berechnungslatenz von zwei bedeutet, dass die Berechnung im Takt nach ihrem Beginn abgeschlossen wird und so weiter.

Für die Gesamtlatenz der Operation müssen diese drei Latenzen aufaddiert werden. Dabei muss allerdings noch beachtet werden, dass sich die Phasen Einlesen und Berechnung beziehungsweise Berechnung und Schreiben überschneiden können. Eine Überschneidung von einem Takt bedeutet zum Beispiel, dass die Berechnung im letzten Takt des Einlesens startet. Bei einigen Operationen ist auch die Überschneidung von allen drei Phasen möglich. Außerdem ist können sich auch zwei oder alle drei Phasen vollständig überlappen. Dies ist in Abbildung 2.4 dargestellt. Eine teilweise Überschneidung ist im oberen Teil von Abbildung 2.5 zu sehen.

Abbildung 2.4.: Zeitverhalten kombinatorische Operation

Bei einigen sequentiellen Operationen mit Ein- beziehungsweise Ausgabedaten größer als die Datenpfadbreite ist aber auch eine größere Überschneidung an einem oder beiden Übergängen möglich. Ein Beispiel hierfür ist eine Additions-Operation, welche Daten mit der doppelten Datenpfadbreite verarbeitet. Deren Eingabe- und Ausgabelatenz beträgt daher jeweils zwei Takte. Die Implementierung kann entweder kombinatorisch sein, sie benötigt dann einen Addierer für die doppelte Datenpfadbreite.

Alternativ ist auch eine Implementierung mit einem Addierer für die einfache Datenpfadbreite denkbar. Diese ist dann sequentiell und hat eine Berechnungslatenz

von zwei Takten. Im ersten Takt werden die unteren Hälften der Eingänge aufaddiert und das Carry-Bit wird in einem Register gespeichert. Im zweiten Takt werden die oberen Hälften und das Carry-Bit addiert. Die Berechnung kann in diesem Fall schon im ersten Takt des Einlesens starten, da die untere Hälfte der Eingangsdaten in diesem gelesen wird. Die Überschneidung zwischen Eingabe und Berechnung beträgt also zwei Takte. Analog kann das Schreiben des Ergebnisses im ersten Takt der Berechnung stattfinden, da die untere Hälfte des Ergebnisses bereits vorliegt. Die Überschneidung zwischen Berechnung und Eingabe beträgt also ebenfalls zwei Takte.

Die beiden Möglichkeiten sind in Abbildung 2.5 dargestellt.

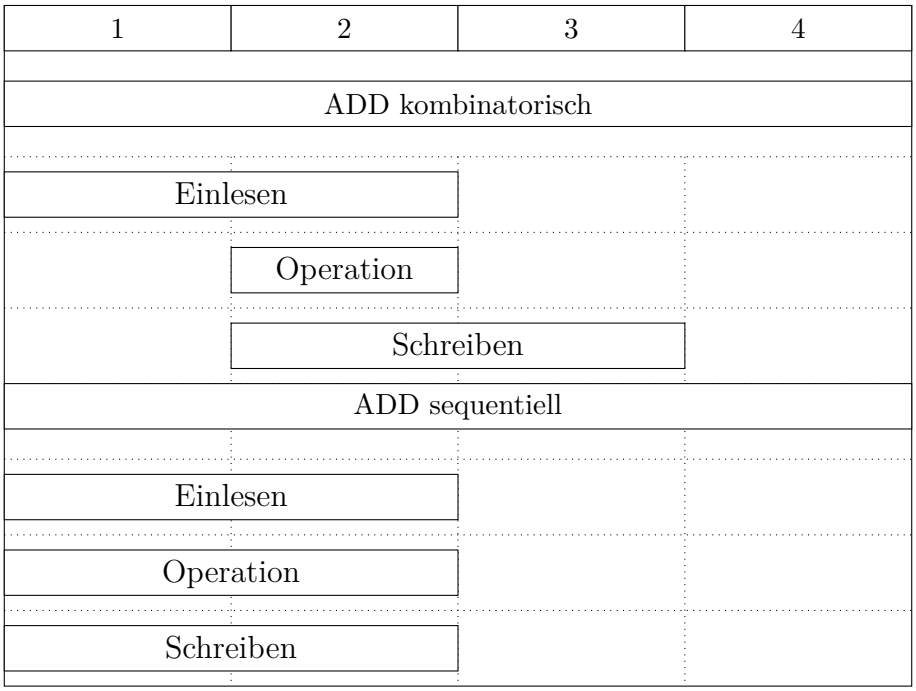

Abbildung 2.5.: Zeitverhalten Addition doppelte Datenpfadbreite

2.3. Integratoren: Euler & Heun

Vor der Makro-Op-Optimierung wird der DFG um Integratoren erweitert, um die zeitlichen Ableitungen aufzulösen. Die zeitlichen Ableitungen werden verwendet, um die Veränderung der internen Zustände des mechatronischen Systems zwischen zwei Integrationsschritten zu beschreiben. Im UltraSynth-Projekt können zwei verschiedene Integratoren eingesetzt werden: Euler und Heun. Um das System zu ver-

Evtl. umstellen? Zweimal um doof

Literaturverweise?

anschaulichen, sei ein einfaches System mit dem internen Zustand `val` und seiner zeitlichen Ableitung `val' = const - val` gegeben. Beide Integratoren bestimmen den neuen Wert des Zustands `val_neu` für den nächsten Integrationszyklus aus dem alten Wert `val_alt` und der zeitlichen Ableitung `val'`.

Die Formel des Integrators erster Ordnung, genannt Euler-Integrator, lautet `val_neu = val_alt + val' * dt`. `dt` ist eine Konstante und wird als Schrittweite des Integrators bezeichnet. Dies ist beispielhaft in Abbildung 2.6 dargestellt: Auf der linken Seite befindet sich das System vor der Ergänzung des Integrators. Der gestrichelte Pfeil stellt den Übergang zwischen den Integrationszyklen dar. Diese implizite Abhängigkeit wurde auf der rechten Seite durch den Euler-Integrator aufgelöst. Auf den Wert des internen Zustands wird mithilfe der LOAD-Operation zugegriffen. Nach der eigentlichen Berechnung der zeitlichen Ableitung erfolgt der Integrationsschritt, welcher den Zustand für den nächsten Zyklus liefert. Der neue Zustand wird dann mithilfe der STORE-Operation gesichert. Nach dem Durchlauf des kompletten DFGs startet die Berechnung mit diesem aktualisierten Zustand erneut.

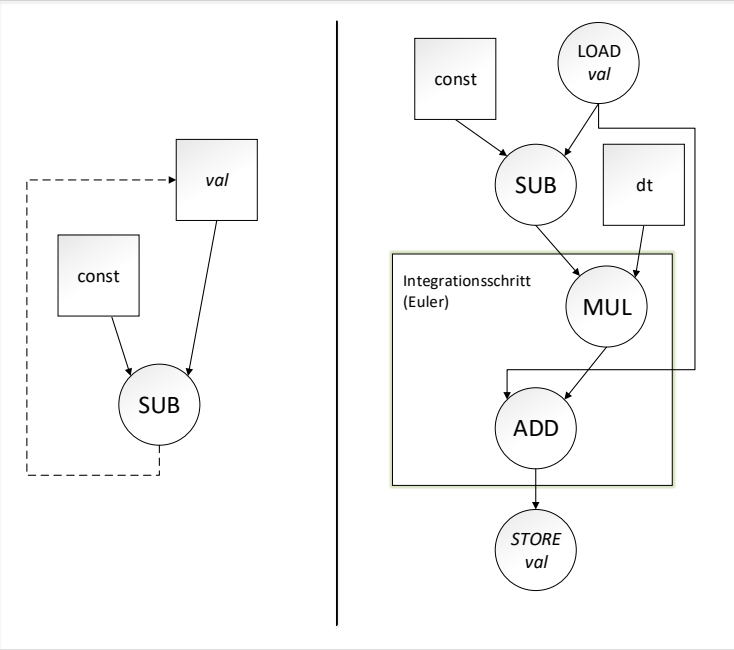

Abbildung 2.6.: Graph mit und ohne Euler-Integrator.

Der Integrator zweiter Ordnung, genannt Heun-Integrator, ist etwas komplizierter, denn er verwendet zusätzlich einen Nebenschritt. In diesem wird, wie beim Euler-Integrator, der neue Wert berechnet: `val_neu_nebenschritt = val_alt + val' * dt`. Für diesen Wert wird dann erneut die zeitliche Ableitung `val'_nebenschritt` bestimmt. Der neue Wert wird dann mithilfe des Mittelwerts aus den beiden Ableitungen bestimmt: `val_neu = val_alt + (val' + val'_nebenschritt) * 0.5`

`* dt`. In Abbildung 2.7 ist die Gegenüberstellung des Systems vor und nach der Ergänzung des Heun-Integrators dargestellt.

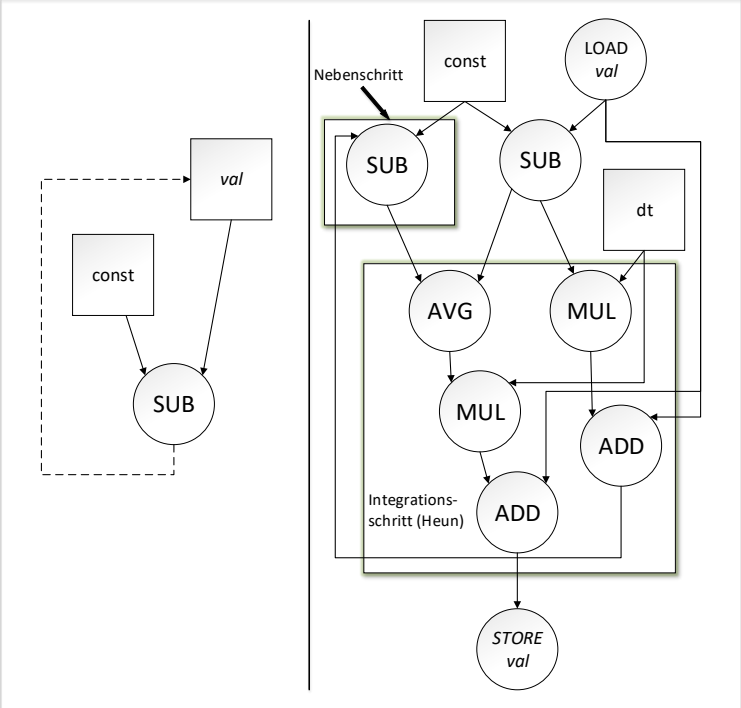

Abbildung 2.7.: Graph mit und ohne Heun-Integrator.

Beide Integratoren haben die Gemeinsamkeit, dass sie für jeden internen Zustand einmal in den DFG eingefügt werden. Der Integrationsschritt ist also in realen Systemen mehrfach vorhanden und daher eine möglicherweise lohnende Option für eine Makro-Op. Bei den Heun-Integratoren wird neben dem eigentlichen Integrationsschritt auch die komplette zeitliche Ableitung vervielfältigt. Auch dies stellt eine Möglichkeit für sinnvolle Makro-Ops dar, weil die Anzahl der Vorkommen durch den Nebenschritt des Heun-Integrators garantiert verdoppelt wird. Beides wird in Kapitel 5 und Kapitel 6 genauer betrachtet.

3. Verwandte Arbeiten

Die Makro-Op-Optimierung ähnelt auf den ersten Blick unter anderem bekannten Ansätzen aus der High-Level Synthese (HLS). Ein Beispiel hierfür ist das Operation Chaining. Dies beschreibt eine Optimierung, die mehrere aufeinanderfolgende Teiloperationen in einen Takt zusammenfasst, um etwa die Gesamtlatenz zu minimieren. Es hat Ähnlichkeiten zur Überlappung von Teiloperationen (siehe Abschnitt 2.2.4). Dieser und andere Ansätze wurden in zahlreichen Arbeiten allgemein oder unter einem bestimmten Aspekt betrachtet. Im Folgenden werden einige dieser Paper im Detail betrachtet und vor allem auf Gemeinsamkeiten und Unterschiede eingegangen. Daraus resultierend wird erörtert, welche der Ansätze und Konzepte ganz oder teilweise auf die Makro-Op-Optimierung übertragbar sind.

In [5] wird ein neuer Schedulingalgorithmus für das Operation Chaining in der HLS vorgestellt. Im Gegensatz zu den bekannten As-Soon-As-Possible (ASAP)- und As-Late-As-Possible (ALAP)-Schedulingalgorithmen versucht der Algorithmus von Zaretsky u. a., die Operationen möglichst gleichmäßig auf die Takte zu verteilen. Dementsprechend wird dieser Schedulingalgorithmus auch Balanced Scheduling genannt. Diese Technik ist im Rahmen der Makro-Op-Optimierung allerdings nicht anwendbar; es wird zwar eine Art Operation Chaining durchgeführt, allerdings wird kein Scheduler entwickelt. Stattdessen wird der bestehende Scheduler des UltraSynth-Projekts verwendet. Es wird dagegen nur ermittelt, bei welchen Subgraphen eine Ersetzung immer Vorteile bringt. Das Ersetzen durch eine Makro-Op verbirgt das Operation Chaining vor dem Scheduler, weshalb dieser das Operation Chaining nicht unterstützen muss.

In [6] und [7] wird eine Methode zum Verketten mehrerer Operationen in der HLS diskutiert. Das Operation Chaining in der HLS wurde bereits in vielen vorherigen Arbeiten betrachtet, allerdings immer für eine Centralized-Register-Architektur (CR-Architektur). Terada u. a. schlagen erstmals einen HLS-Algorithmus mit Operation Chaining für die Regular-Distributed-Register-Architektur (RDR-Architektur) vor. Bei einer CR-Architektur sind alle Register an einer einzelnen oder einigen wenigen Stellen des Chips konzentriert, bei der RDR-Architektur dagegen sind die Register gleichmäßig auf der kompletten Chipfläche verteilt.

In [6] werden immer nur maximal zwei Operationen verkettet, in [7] dagegen gibt es diese Einschränkung nicht mehr.

Auf den ersten Blick gibt es Ähnlichkeiten zwischen der RDR-Architektur und dem CGRA beim UltraSynth-Projekt: Jedes PE enthält ein Registerfeld, wodurch die Register gleichmäßig über den Chip verteilt sind. Vor allem aus zwei Gründen lassen sich die Ansätze der Arbeiten allerdings trotzdem nicht in größerem Stil auf das Makro-Op-Problem anwenden:

Die Register der RDR-Architektur und die Registerfelder des CGRAs erfüllen zwar denselben Zweck – das Speichern von Daten. Aber die vorgeschlagenen Algorithmen sind so speziell auf Register angepasst, dass sich die Vorgehensweise nicht ohne größere Änderungen auf Registerfelder abbilden lässt. Das Hauptproblem ist dabei die Begrenzung der Anzahl an gleichzeitig lesenden beziehungsweise schreibenden Zugriffen auf das Registerfeld.

Ein noch wichtigerer Unterschied ist allerdings, dass die Registerfelder und deren Ansteuerung im Rahmen der Makro-Op-Optimierung nicht geändert werden können – dies ist fest durch den CGRA vorgegeben. Auch die Bereitstellung der Daten aus den Registerfeldern für die Operationen kann nicht angepasst werden.

In [8] geht es um ein optimiertes Design für eine konkrete Operation: Fused Add-Multiply. Diese Studie steht hier stellvertretend für eine große Anzahl an Arbeiten, welche sich mit der Optimierung einer bestimmten Operation befassen. Für die Optimierung werden dabei immer anwendungsspezifische Einschränkungen und Eigenschaften verwendet. Daher sind die Ergebnisse auch selten auf andere Szenarien übertragbar, meist ist eine erneute händische Optimierung erforderlich. Dies kann zwar sehr zeitintensiv sein, aber in vielen Fällen lohnt sich der Aufwand und es sind dadurch zusätzliche Einsparungen möglich. Eine solche manuelle Optimierung ist auch im Rahmen der Makro-Op-Optimierung denkbar – statt einer automatisch generierten Verilog-Implementierung würde diese dann von Hand geschrieben.

[9] beschäftigt sich mit dem Operation Chaining in asynchronen Pipelines. Im Gegensatz zum Verketten in synchronen Schaltungen gibt es dort keinen Performancegewinn – es gibt schließlich keinen Takt, den es gut auszunutzen gilt. Allerdings entfällt durch das Verketten von zwei oder mehr Pipelinestufen die Handshake-Logik zwischen diesen. Dadurch wird sowohl die Fläche als auch die benötigte Energie der Handshake-Logik gespart. Dies erfolgt auf Kosten der Parallelität – es gibt weniger Pipelinestufen, die gleichzeitig ausgeführt werden können.

Bei der Makro-Op-Optimierung gibt es zwar einen globalen Takt, es handelt sich also nicht wirklich um eine asynchrone Pipeline. Da die einzelnen Operationen aber eine beliebige Anzahl an Takten benötigen dürfen, gibt es durchaus Gemeinsamkeiten zu der Situation bei den asynchronen Pipelines. Durch die Ausführung auf

dem CGRA kommt die Einsparung von Handshake-Logik nicht infrage – diese ist fest durch die CGRA-Struktur vorgegeben. Die Logik einer Makro-Op und damit auch die Handshakes zwischen den Teiloperationen sind zwar komplett durch ihre Verilog-Implementierung beeinflussbar, allerdings spart dies keine Chipfläche, da die fest vorgegebene Handshake-Logik des CGRA trotzdem benötigt wird und nicht weggelassen werden kann. Dass die Makro-Op-Optimierung jedoch eine Einsparung beim Energieverbrauch hervorbringt ist durchaus denkbar – immerhin müssen weniger Daten zwischen ALU und Registerfeld beziehungsweise zwischen den verschiedenen PEs transportiert werden.

Die folgenden beiden Arbeiten beschäftigen sich nicht direkt mit dem Zusammenfassen von Operationen, enthalten aber interessante Ansätze, welche auf die Suche nach möglichen Ersetzungen für Makro-Ops angewendet werden können.

In [10] geht es in erster Linie um ein gänzlich anderes Thema: das Zusammenlegen von mehreren konsekutiven Array- oder Listenoperationen. Dies hat in imperativen Programmiersprachen beispielsweise zur Folge, dass mehrere Schleifen kombiniert werden. In funktionalen Programmiersprachen dagegen werden mehrere Kombinatoren zusammengefasst, in anderen Programmierparadigmen gibt es vergleichbare Effekte. Durch dieses Zusammenlegen werden die Arrayzugriffe reduziert und dadurch wird die Ausführung beschleunigt. Diese Optimierung wird im Paper dabei allgemein als Weighted Subroutine Partition (WSP) Problem definiert. Für dieses NP-schwere Problem wird dann ein exakter Lösungsalgorithmus sowie eine schnelle Heuristik präsentiert.

Der Zusammenhang zur Makro-Op-Optimierung liegt in der allgemeinen Formulierung des WSP: Die Definitionen des Problems lassen sich einfach auf die Makro-Op-Optimierung abbilden, der Algorithmus wäre also prinzipiell auch hier einsetzbar. Allerdings liefert der vorgeschlagene Algorithmus keine einzelnen Subgraphen, welche zusammengefasst werden können, sondern eine komplette Partitionierung des Graphen. Dies ist im Rahmen der Makro-Op-Optimierung aber nicht sinnvoll, da jede einzelne Makro-Op Chipfläche in der ALU mindestens eines PEs benötigt. Die vollständige Partitionierung enthält potentiell so viele Makro-Ops, dass die benötigte Chipfläche schlicht zu groß wäre – die zeitlichen Einsparungen stehen in keinem Verhältnis zur dafür verwendeten Chipfläche.

[11] gibt einen Überblick über das Instruction-Set Extension Problem und die Lösungsmethoden und -ansätze aus diversen Arbeiten. Dieses Problem ist eigentlich eine Ebene höher als die Makro-Op-Optimierung angesiedelt: Es geht um die Auswahl von performanzkritischen Programmteilen, welche dann direkt in Hardware statt als Programm auf der CPU ausgeführt werden. Dies passiert im Rahmen des

UltraSynth-Projekts vor der Makro-Op-Optimierung. Die Ähnlichkeiten zur Suche nach Kandidaten für die Makro-Op-Ersetzung sind jedoch so groß, dass viele der Ideen und Ansätze zumindest teilweise darauf anwendbar sind. Vor allem die Teile zu Instruction Generation (das Generieren aller technisch möglichen Kandidaten für die Ersetzung) und Instruction Selection (das Auswählen einiger dieser Kandidaten auf Grundlage der Vor- und Nachteile) sind für die Makro-Op-Optimierung interessant. Mehrere Ideen aus diesen Kapiteln wurden verwendet: Für die Heuristik (siehe Abschnitt 4.2.5) wurde eine optionale zusätzliche Abbruchbedingung erstellt, welche Bereiche des Suchraums ausschließt, die mit großer Wahrscheinlichkeit keine sinnvollen Ergebnisse enthalten. Diese Technik wird auch als Pruning bezeichnet und von Galuzzi u. a. unter anderem in Abschnitt 4.3 beschrieben. Sowohl Heuristik als auch Vollständige Suche (siehe Abschnitt 4.2.4) enthalten eine weitere Abbruchbedingung, welche ebenfalls eine Art Pruning darstellt. Diese verhindert, dass Bereiche des Suchraums mehrmals durchlaufen werden. In Abschnitt 4.5 beschreiben die Autoren *template overlapping*, also das Überlappen mehrerer Makro-Ops beziehungsweise Vorkommen. In Abschnitt 4.2.3 wird erklärt, wie im Rahmen dieser Arbeit eine Überlappung ermittelt und ausgeschlossen wird. Galuzzi u. a. stellen als weitere Lösung die Replikation der Überlappungsbereiche vor: Die Knoten, welche in beiden Subgraphen liegen, werden dupliziert, um beide Ersetzungen zu ermöglichen. Auch das in Abschnitt 4.1.1 beschriebene Konzept der Konvexität der Subgraphen wird in [11] erläutert.

Die verbleibenden drei Arbeiten stellen den Ziel-CGRA, den zugehörigen Scheduler sowie weitere theoretische Grundlagen vor.

In [3] und [4] werden der beim UltraSynth-Projekt verwendete CGRA beziehungsweise der zugehörige Scheduler im Detail beschrieben. Sowohl für das Umsetzen als auch das Nachvollziehen der Makro-Op-Optimierung sind einige technische Details der Hardware und des Schedulers unbedingt nötig. Die zum Verständnis notwendigen Teile aus den beiden Arbeiten sind in Abschnitt 2.1 zusammengefasst.

In [12] wird die Synthese eines CGRAs mit einem spezialisierten beziehungsweise erweiterten Instruktionssatz beschrieben und auf seine Leistungsfähigkeit hin analysiert. Essentiell dabei ist, dass nicht jede PE den kompletten Instruktionssatz abdeckt – der Flächenbedarf der PEs würde schnell explodieren – sondern neben den Basisinstruktionen nur einige wenige spezielle. Nur durch das Zusammenspiel aller PEs wird der gesamte benötigte Instruktionssatz erreicht. Die Analyse ergibt, dass dies bei typischen Anwendungen keine relevante Verschlechterung gegenüber einem CGRA mit uniformen PEs darstellt – vorausgesetzt, die einzelnen Instruktionssätze der PEs sind sinnvoll gewählt.

Die Erkenntnisse dieses Artikels bilden eine wichtige Grundlage für die Makro-Op-Optimierung. Ohne die Möglichkeit, die speziellen Makro-Ops auf verschiedene PEs zu verteilen, wären die Vorteile vermutlich minimal oder nicht existent.

4. Implementierung

In diesem Kapitel wird dargestellt, wie die Makro-Op-Optimierung in den Code-Rahmen des UltraSynth-Projekts eingefügt wurde. Die Erweiterungen lassen sich dabei grob in zwei Bereiche unterteilen:

Der erste Teil ist die Analyse eines DFGs, um mögliche Kandidaten für die Ersetzung zu identifizieren. Es wurden zwei verschiedene Analysen implementiert: eine erschöpfende Suche und eine Heuristik. Beide erwarten als Eingabe einen Graphen und liefern mögliche Ersetzungen. Dabei wird unter anderem auch auf die dabei aufgetretenen Probleme eingegangen.

Der zweite Teil ist das Ersetzen eines Subgraphen durch die entsprechende Makro-Op. Der Subgraph kann zum einen das Ergebnis einer der Analysen sein, aber auch die manuelle Definition durch den Benutzer ist möglich.

Um die Benutzung dieser Funktionen zu erleichtern, wurde ein einfaches Benutzerinterface entwickelt. Neben dem Durchsuchen des DFGs mithilfe einer der Analysen und dem Anzeigen der Ergebnisse erlaubt das Interface auch das Speichern und spätere Laden der Analyseergebnisse. Nach dem Analysieren oder Laden von Ergebnissen ist das Exportieren einzelner Makro-Ops im .dot-Format möglich. Außerdem kann eine Gruppe von disjunkten Vorkommen einer Makro-Op gewählt werden, die im DFG ersetzt wird. Das Ersetzen ist auch ohne Analyseergebnisse mit einem händisch definierten Subgraphen möglich.

Aus Gründen der besseren Nachvollziehbarkeit wird im Folgenden zuerst der zweite Teil, das Ersetzen einer Makro-Op, beschrieben.

4.1. Makro-Op-Ersetzung

Das Ersetzen eines Subgraphen des DFGs durch eine Makro-Op besteht aus vier Teilschritten: Zuerst werden alle für die übrigen Schritte notwendigen Eigenschaften des zu ersetzenden Subgraphen analysiert und in einem neuen Objekt gespeichert. Anschließend muss das Zeitverhalten der resultierenden Makro-Op analysiert werden, da der Scheduler diese Informationen benötigt. Diese beiden Schritte werden auch von den Analysen (siehe Abschnitt 4.2) verwendet, da die Latenz der Makro-Op für deren Bewertung (siehe Abschnitt 4.2.1) benötigt wird. Die Eigenschaften des

Subgraphen werden unter anderem für den Vergleich zweier Makro-Ops (siehe Abschnitt 4.2.2) benötigt. Dann wird die Verilog-Implementierung der Makro-Op aus den Implementierungen der Teiloperationen zusammengesetzt. Im letzten Schritt wird der DFG modifiziert, um die Ersetzung widerzuspiegeln: Die Teiloperationen des Subgraphen werden entfernt und die neu erstellte Makro-Op wird eingefügt und korrekt verknüpft.

Diese vier Schritte werden im Folgenden genauer erläutert.

4.1.1. Analyse des Subgraphen

Bei der Erstellung einer Makro-Op wird als erstes der Subgraph, der ersetzt werden soll, analysiert und relevante Informationen gespeichert. Dies hat den Vorteil, dass diese Daten für die weiteren Schritte direkt zur Verfügung stehen und der Subgraph nicht für jede einzelne Information erneut durchlaufen werden muss. Zuvor muss allerdings geprüft werden, ob dieser Subgraph überhaupt durch eine Makro-Op ersetzt werden darf. Hierfür wurde die Methode `makroOpAllowed()` in der `IDP`-Klasse implementiert.

Es gibt mehrere *Einschränkungen*, die festlegen, welche Subgraphen durch eine Makro-Op ersetzt werden können. Diese Einschränkungen gelten entweder allgemein oder nur für die aktuelle Implementierung. Die wichtigste allgemeine Voraussetzung an den Subgraphen ist die Konvexität: Kein Pfad von einem Teilknoten des Subgraphen zu einem anderen Teilknoten darf Knoten enthalten, die nicht zum Subgraphen gehören. Wenn dies der Fall wäre, würde ein Eingang der Makro-Op transitiv von einem ihrer Ausgänge abhängen. Dies würde das Scheduling unmöglich machen oder zumindest extrem verkomplizieren. Für die Überprüfung dieser Einschränkung werden die transitiven Vorgänger aller Teilknoten bestimmt. Sollte einer dieser transitiven Vorgänger ein Nachfolger des Subgraphen sein, ist der Subgraph nicht konvex und daher keine erlaubte Makro-Op.

Darüber hinaus werden aktuell bestimmte Knotentypen ausgeschlossen: Nicht erlaubt sind Knoten ohne Ein- oder Ausgänge (zum Beispiel Laden/Speichern oder Konstanten), sowie Knoten, deren Operation ein Vergleich ist. Letztere verwenden ein zusätzliches Status-Signal, um das Ergebnis des Vergleichs anzuzeigen. Sowohl das Fehlen von Ein- oder Ausgängen als auch das Status-Signal bereiten bei der Ersetzung durch eine Makro-Op Probleme. Nur unter bestimmten Bedingungen sind solche Makro-Ops überhaupt möglich. Das Fehlen von Ein- oder Ausgängen stellt beispielweise bei der aktuellen Implementierung der Laufzeitanalyse (siehe Abschnitt 4.1.2) ein Problem dar. Diese ermittelt die längsten Pfade zwischen den Ein- und Ausgängen der Makro-Op. Knoten, denen eines von beiden fehlt, werden daher inkorrekterweise für die Latenzberechnung ignoriert. Auch das Bestimmen von gleichen Makro-Ops (siehe Abschnitt 4.2.2) basiert auf den Ein- und Ausgän-

gen. Bei Operationen, welche ein Status-Signal ausgeben, gibt es eine Einschränkung aufgrund des Aufbaus des CGRAs: Pro Operation ist maximal ein Status-Ausgang erlaubt; eine Makro-Op könnte also maximal einen Vergleich beinhalten, der ein Status-Signal für außerhalb produziert. Dies gilt umgekehrt auch für Status-Eingänge der Makro-Op. Außerdem müssen sowohl interne als auch nach außen gehende oder von dort kommende Status-Signale in der Implementierung extra berücksichtigt werden. Daher werden beide Fälle aktuell ausgeschlossen – zumindest eine teilweise Unterstützung ist aber bei beiden möglich.

Wenn diese Voraussetzungen erfüllt sind, wird die Struktur des Subgraphen analysiert und relevante Informationen werden gespeichert. Für die Speicherung der Daten werden mehrere Klassen aus dem `makroop`-Package verwendet.

Die Klasse `SubOpNode` speichert alle benötigten Informationen eines Teilknotens der Makro-Op. Dies umfasst insbesondere folgende Daten:

- Die ID des Knotens im DFG

  Diese wird benötigt, um den Knoten eindeutig zu identifizieren, was vor allem für die Anpassung des Graphen notwendig ist. Außerdem wird sie benutzt, um die internen Verilog-Bezeichner in der Verilog-Implementierung der Makro-Op eindeutig dem Knoten zuzuordnen.

- Die Nachfolger des Knotens

  Dabei werden ausschließlich diejenigen Knoten gespeichert, welche ebenfalls Teil des Subgraphen sind. Sie werden unter anderem für den Vergleich von zwei Makro-Ops und für die Laufzeitanalyse benötigt.

- Die Vorgänger des Knotens

  Es werden ebenfalls nur die Vorgänger innerhalb des Subgraphen gespeichert. Diese Information ist zwar redundant zu den Nachfolgern, sie reduziert aber die Komplexität des Vergleichs und der Laufzeitanalyse.

- Die Operation des Knotens

  Dies umfasst neben der Verilog-Implementierung auch Informationen über verschiedene Latenzen und die Ein- und Ausgaben der Operation. Diese Informationen werden sowohl für die Laufzeitanalyse als auch für das Generieren der Verilog-Implementierung der Makro-Op benötigt.

- Zusätzliche Informationen über die Eingabedaten der Operation

  Dies umfasst immer, ob diese Daten von einer Teiloperation der Makro-Op oder von einem ihrer Operandenknoten produziert werden. Wenn die Quelle innerhalb der Makro-Op liegt, wird zusätzlich der Verilog-Bezeichner des

Ergebnisses der Quelloperation gespeichert. Dieser wird für die Generierung der Verilog-Implementierung benötigt. Ansonsten wird der Verilog-Bezeichner gespeichert, unter welchem diese Eingangsdaten der Makro-Op innerhalb derselben zur Verfügung stehen. Zusätzlich werden in diesem Fall auch die anderen Knoten der Makro-Op gespeichert, welche diese Eingabedaten benötigen.

Die Klasse `OpGraph` speichert Informationen der kompletten Makro-Op. Dies sind im Wesentlichen ihre Eingangs- und Ausgangsknoten. Zusätzlich werden auch alle in der Makro-Op enthaltenen Knoten gespeichert. Diese Information ist ebenfalls redundant, vereinfacht und beschleunigt aber unter anderem den Vergleich zweier Makro-Ops.

4.1.2. Laufzeitanalyse

Die Laufzeitanalyse berechnet die Latenz nach der Ersetzung eines Subgraphen durch eine Makro-Op. Außerdem werden auch die Ein- und Ausgabelatenzen der Makro-Op bestimmt.

Die Methode `analyzeLatencies()` der Klasse `OpGraph` führt die Laufzeitanalyse aus, der Zugriff auf die Ergebnisse ist dann durch `getLatency()`, `getInputLatency()` und `getOutputLatency` möglich.

Prinzipiell ist für die Latenz der längste Pfad durch die Makro-Op relevant. Die Länge des Pfades ist dabei nicht durch die Anzahl an Knoten bestimmt, sondern durch die Summe der Latenzen der Operationen auf dem Pfad.

Allerdings ist die Berechnung der Latenz der Makro-Op wesentlich komplexer, als einfach den längsten Pfad zu bestimmen. Da innerhalb der Makro-Op kein I/O stattfindet, müssen auf den Pfaden nur die reinen Berechnungslatenzen aufaddiert werden – ohne die Takte, welche vorher von den Teiloperationen für das Laden und Speichern der Werte benötigt wurden. Um eine optimale Latenz der Gesamtoperation zu erreichen, muss dabei aber eine mögliche Überlappung zweier aufeinanderfolgender Berechnungen berücksichtigt werden (siehe Abschnitt 2.2.4). Die mögliche Überlappung zweier aufeinanderfolgender Berechnungen ist dabei das Maximum der Eingabeüberlappung der ersten Berechnung und der Ausgabeüberlappung der zweiten Berechnung.

Das größere Problem hängt aber mit dem I/O-Verhalten der Makro-Op zusammen: Vor der Ersetzung wird jede Teiloperation auf einem eigenen PE ausgeführt; es stehen also jeweils zwei Eingabe-Ports und ein Ausgabe-Port zur Verfügung (Abschnitt 2.1). Es können also alle Eingaben parallel ausgeführter Operationen gleichzeitig gelesen werden. Analog gilt dies ebenfalls für die Ausgaben. Nach der Ersetzung wird die Makro-Op nur auf einer PE ausgeführt – ihr stehen daher insgesamt nur zwei Eingabe-Ports und ein Ausgabe-Port zur Verfügung. Besitzt die Makro-Op mehr als zwei Eingänge, können daher nicht mehr alle Daten parallel eingelesen

werden, sondern müssen in mehreren, aufeinanderfolgenden Takten gelesen werden. Analog gilt dies wiederum für die Ausgangsdaten. Um eine effiziente Makro-Op zu ermöglichen, ist also eine sinnvolle Reihenfolge für das Übertragen der Ein- und Ausgänge essentiell.

Auf den ersten Blick ist die Lösung trivial: Die Eingänge des längsten Pfads werden als erstes geladen, dessen Ausgänge als letztes geschrieben. Anschließend folgen die Ein- und Ausgänge des zweitlängsten Pfades und so weiter. Diese Lösung produziert auch in vielen Fällen gute Ergebnisse. Wenn allerdings zwei gleichlange Pfade vorliegen, verbessert es die Latenz, wenn derjenige bevorzugt wird, bei dem der Ausgang von weniger Eingängen abhängt. Dies verhindert, dass die Ausgangsknoten der Pfade ihre Operation gleichzeitig abschließen, aber nur ein Ergebnis direkt geschrieben werden kann, weil nur ein Ausgangsport zur Verfügung steht. Dies kann anhand des Beispiels in Abbildung 4.1 veranschaulicht werden: Die drei Zahlen in den Knoten beschreiben jeweils die Eingabeübelappung, die Berechnungslatenz und die Ausgabeüberlappung. Die Datenbreite entspricht immer der Datenpfadbreite des CGRAs, die Eingabe- und Ausgabelatenz beträgt also immer 1.

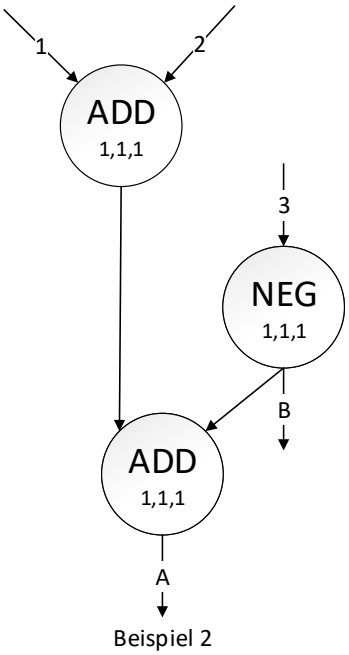

Abbildung 4.1.: Beispiel zur Laufzeitanalyse.

Es existieren vier Pfade: Von allen drei Eingängen jeweils zu Ausgang A und von Eingang 3 zu Ausgang B. Diese Pfade haben offensichtlich alle dieselbe Latenz. Da die Operationen alle kombinatorisch sind und sich überschneiden dürfen, beträgt

die Latenz jeweils 0 Takte. Die beiden Ergebnisse stehen also in dem Takt zum Schreiben bereit, in dem alle ihre benötigten Eingänge eingelesen wurden. Um die Gesamtlatenz der Makro-Op zu minimieren, ist es aber notwendig, Eingang 3 im ersten Takt einzulesen. Ausgang B kann dann nämlich ebenfalls im ersten Takt geschrieben werden, Ausgang A im zweiten Takt. Die Gesamtlatenz beträgt also 2. Wenn im Gegensatz dazu die Eingänge 1 und 2 im ersten Takt gelesen werden, kann keiner der Ausgänge im ersten Takt geschrieben, da beide noch auf Eingang 3 warten. Im zweiten Takt können aber nicht beide Ergebnisse geschrieben werden, da nur ein Ausgangs-Port zur Verfügung steht. Einer der Ausgänge kann also erst im dritten Takt geschrieben werden, die Gesamtlatenz beträgt somit 3.

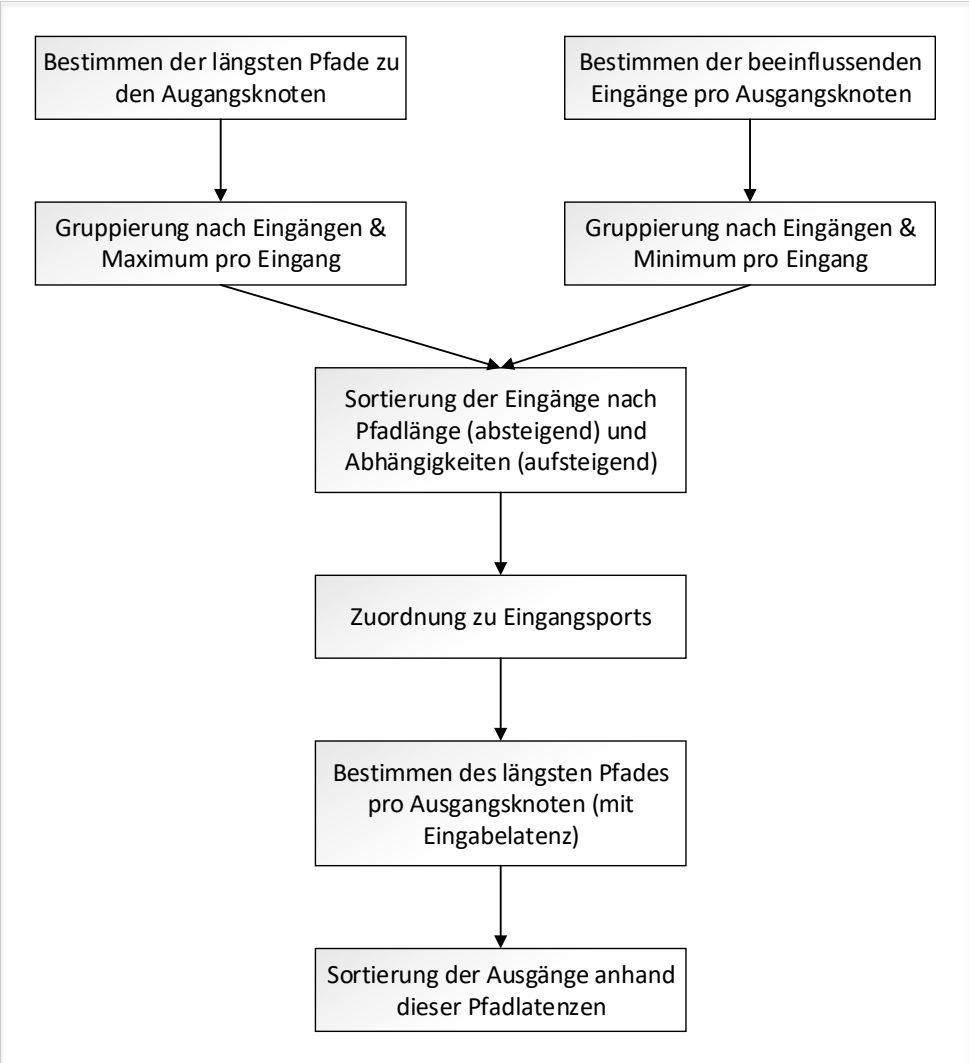

Abbildung 4.2.: Ablaufplan Laufzeitanalyse.

Um diesen und ähnliche Fälle gut zu lösen, wurde das im Folgenden beschriebene Verfahren gewählt. Der grobe Ablauf ist in Abbildung 4.2 dargestellt. Zum besseren Verständnis werden die einzelnen Schritte anhand des oben gegebenen Beispiels durchgeführt.

Im ersten Schritt wird, ausgehend von jedem Eingang der Makro-Op, der Subgraph durchlaufen. Dabei wird angenommen, dass alle Eingänge ab dem ersten Takt eingelesen werden. Beim Durchlauf wird in jedem Knoten der Makro-Op für alle Makro-Op-Eingänge die jeweilige Latenz des längsten Pfades vom Eingang zum Knoten gespeichert. Dabei wird wie oben beschrieben die mögliche Überlappung zweier Berechnungen berücksichtigt. Es werden also die Berechnungslatenzen auf jedem Pfad durch die Makro-Op bestimmt, jeder Ausgangsknoten speichert also seinen längsten Pfad zu jedem Eingang (sofern vorhanden). Im Beispiel haben die längsten Pfade beider Ausgangsknoten wie oben beschrieben eine Latenz von 0. Diese längsten Pfadlatenzen werden nun gesammelt und nach ihren Eingängen gruppiert. Für jeden Eingang wird der maximale Wert bestimmt.

Außerdem wird für jeden Ausgangsknoten die Anzahl der Makro-Op-Eingänge bestimmt, von denen der Ausgang abhängt. Dies sind im Beispiel drei Eingänge für Ausgang A und ein Eingang für Ausgang B. Diese Werte werden ebenfalls nach Eingängen gruppiert und der jeweils minimale Wert berechnet. Für jeden Eingang wurde nun also die Latenz seines längsten Pfades sowie die kleinste Anzahl an Eingangsabhängigkeiten aller seiner beeinflussten Ausgänge bestimmt. Im Beispiel betragen die Pfadlatenzen für alle Eingänge 0. Die Anzahl der Abhängigkeiten beträgt für Eingang 1 und 2 drei, für Eingang 3 eins.

Nun können die Eingänge nach diesen maximalen Latenzen sortiert werden, der Eingang mit der höchsten Latenz sollte zuerst eingelesen werden. Wenn für zwei Eingänge dieselbe höchste Latenz ermittelt wurde, wird zusätzlich die minimale Anzahl an Abhängigkeiten betrachtet. Der Eingang mit dem geringeren Wert wird dann zuerst eingelesen. Dies bewirkt, dass gleichzeitig bereitstehende Ausgangswerte, die dann nacheinander geschrieben werden müssten, vermieden werden. Durch diese Sortierung wird der Ausgangsknoten mit weniger Abhängigkeiten früher fertig, da die Makro-Op-Eingänge für seine Pfade früher bereitstehen und die längsten Pfade gleich lang waren. Im Beispiel sind die Latenzen gleich, die Sortierung erfolgt also anhand der Abhängigkeiten. Da für Eingang 3 hier der niedrigere Wert berechnet wurde, wird dieser als erstes einsortiert.

Daraus kann nun die Reihenfolge des Einlesens bestimmt werden: Für beide Eingangsports wird der Takt gespeichert, in dem dieser den nächsten Wert einlesen kann. Zu Beginn ist dies für beide Eingänge 1. Die sortierten Eingänge werden nun durchlaufen und jeweils einem der Eingangs-Ports der Makro-Op zugeordnet. Es wird immer der Port für den aktuellen Eingang gewählt, der in einem früheren Takt frei ist. Je nach Bitbreite des Eingangs ist dieser Port dann für einen oder mehrere

Takte belegt. Der Takt, in welchem dieser Port wieder verfügbar ist, wird daher um diese Einleselatenz erhöht. Im Beispiel wird also Eingang 3 im ersten Takt gelesen. Die Reihenfolge der anderen beiden Eingänge kann beliebig gewählt werden. Im Folgenden sei angenommen, dass Eingang 1 ebenfalls im ersten Takt gelesen wird und Eingang 2 im zweiten Takt.

Mit diesen so ermittelten Starttakten der Eingangsknoten wird nun, wie im ersten Schritt, der Subgraph ausgehend von jedem Makro-Op-Eingang durchlaufen. Es werden wiederum die Latenzen der längsten Pfade in den Ausgangsknoten gespeichert. Als Starttakt der Pfade wird nun allerdings der gerade berechnete Wert für diesen Eingang verwendet. Die Werte geben also an, in welchem Takt das Ergebnis dieses Knotens zum Schreiben bereitsteht. Im Beispiel ist dies für Ausgang B Takt 1, für Ausgang A Takt 2.

Im letzten Schritt wird die Reihenfolge der Ausgabedaten festgelegt. Dafür werden die Ausgangsknoten aufsteigend anhand ihres eben neu berechneten Endtaktes sortiert. Jeder Ausgangsknoten darf sein Ergebnis im kleinsten Takt schreiben, in welchem sowohl seine Berechnung abgeschlossen als auch das Schreiben des vorherigen Ausgabeknotens beendet ist. Die Gesamtlatenz der Makro-Op entspricht nun dem letzten Takt, in dem Ausgabedaten geschrieben werden. Im Beispiel wird also Ausgang B im ersten Takt geschrieben. Da die Ausgabelatenz eins beträgt, kann Ausgang A direkt im zweiten Takt geschrieben werden. Die Gesamtlatenz beträgt also wie erwartet 2.

Mithilfe der ermittelten Reihenfolge für die Eingänge und Ausgänge der Makro-Op können ihre Ein- und Ausgabelatenz leicht berechnet werden. Im Beispiel betragen sowohl Ein- als auch Ausgabelatenz zwei.

4.1.3. Verilog-Implementierung

Für das Generieren der Verilog-Implementierung der Makro-Op müssen im Wesentlichen nur die Implementierungen ihrer Teiloperationen zusammengefügt werden. Dabei ist allerdings zu beachten, dass die Verilog-Bezeichner aller Operationen identisch sind. Der erste Eingang heißt beispielsweise immer OP_A_I. Damit es nicht zu Konflikten kommt, werden alle Verilog-Bezeichner der Teiloperationen mit der ID des jeweiligen Knotens geprefixt. Nicht davon betroffen sind ausschließlich das Clock- und das Reset-Signal, da diese beiden in allen Teiloperationen gleich angesteuert werden müssen. Es besteht also kein Grund, diese beiden Signale der Teiloperationen unterscheidbar zu machen.

besseres Wort?

Zusätzlich muss noch sichergestellt werden, dass die benötigten Daten aller Teiloperationen unter diesen Bezeichnern mit Prefix zur Verfügung stehen. Dafür wird für jeden Eingang der Teiloperation eine Zuweisung mit dessen Verilog-Bezeichner als Ziel ergänzt. Die Zuweisungsquelle ist dabei entweder der Bezeichner eines Eingangs

der Makro-Op oder des Ausgangs einer anderen Teiloperation. Diese Information ist in den Daten zu den Eingängen der Operation in `SubOpNode` enthalten (siehe Abschnitt 4.1.1).

Außerdem wird für jeden Ausgang der Makro-Op eine Zuweisung mit dem Ausgang der entsprechenden Teiloperation als Quelle erstellt.

Wenn die Makro-Op mindestens eine sequentielle Teiloperation beinhaltet, sind weitere Ergänzungen notwendig. Es wird dann ein Taktzähler im Verilog-Code der Makro-Op ergänzt, der gestartet wird, sobald ihr Enable-Signal aktiviert wird. Für jede sequentielle Teiloperation wird dann mithilfe dieses Zählers das Enable-Signal der Teiloperation in ihrem – in der Laufzeitanalyse ermittelten – Starttakt gesetzt.

Diese Funktionalität wird von der `getImplementation()`-Methode der Klasse `MakroOp` im Package `operator` bereitgestellt. Die aktuelle Version generiert nicht in allen Fällen Verilog-Code, der die in der Laufzeitanalyse ermittelte Latenz erreicht. Dies hat den Grund, dass ihr nur der Verilog-Code der Teiloperationen zur Verfügung steht, aber keine Informationen über dessen Aufbau. Einige Operationen verwenden beispielsweise Schieberegister an den Ein- oder Ausgängen, die für den Verilog-Code der Makro-Op entfernt werden müssten.

Schieberegister, um geplante Ausgabetakte aus Laufzeitanalyse zu erreichen

4.1.4. Anpassung des Graphen

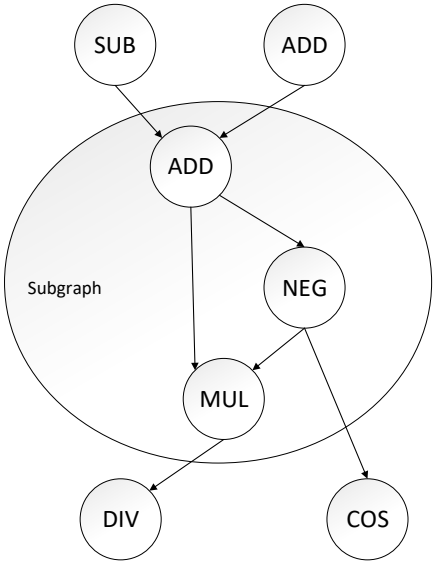

Abbildung 4.3.: Der Subgraph vor der Ersetzung

Um einen Subgraphen durch eine Makro-Op zu ersetzen, muss zuerst ein neu-

er Knoten erstellt werden. Der Knoten muss dann mit den entsprechenden Ein- und Ausgabeformaten der Makro-Op initialisiert werden. Da der Scheduler für die Registerfeld-Adressierung die Reihenfolge der Eingänge der Operation verwendet, ist es notwendig, dass die in der Laufzeitanalyse berechnete optimale Reihenfolge verwendet wird.

Im nächsten Schritt werden die ein- und ausgehenden Datenabhängigkeiten der Makro-Op in den Graphen eingefügt – wiederum ist die Reihenfolge aus der Laufzeitanalyse zu beachten. Kontrollabhängigkeiten können nicht existieren, da alle Operationen mit Status-Signalen ausgeschlossen wurden (siehe Abschnitt 4.1.1). Bei den Eingängen ist zusätzlich wichtig, dass Abhängigkeiten auch dann nicht mehrfach eingetragen werden, wenn mehrere Teiloperationen der Makro-Op diese Daten benötigen. Abschließend werden die Teilknoten des Subgraphen aus dem Graphen entfernt.

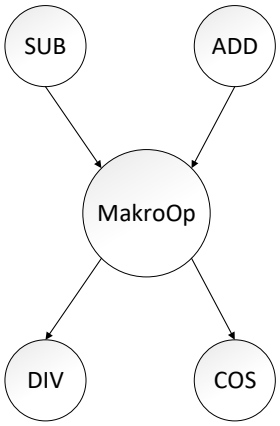

Abbildung 4.4.: Die Makro-Op nach der Ersetzung

4.2. Analyse des Graphen

Für die Suche nach ersetzbaren Makro-Op-Vorkommen wurden zwei Algorithmen implementiert: Der erste Algorithmus ist eine erschöpfende Suche, die alle möglichen Kombinationen bis zu einer konfigurierbaren Maximalgröße durchsucht und bewertet. Er wird im Folgenden als *Vollständige Suche* bezeichnet. Aufgrund seiner Komplexität ist dieser bei realistischen Graphengrößen mit mehreren hundert oder tausend Knoten kaum anwendbar. Daher wurde zusätzlich eine Heuristik entwickelt, welche den Suchraum auf wahrscheinlich sinnvolle Möglichkeiten einschränkt und geringe Ungenauigkeiten (siehe Abschnitt 5.4) bei der Bewertung in Kauf nimmt, um die Analysezeit deutlich zu reduzieren.

Im Folgenden werden zuerst drei Punkte erläutert, die bei diesen beiden und allen vergleichbaren Algorithmen relevant sind: Die Bewertung der Ersetzungen, der Vergleich von zwei Makro-Ops und das Zwischenspeichern der Ergebnisse. Im Anschluss werden die beiden implementierten Algorithmen im Detail vorgestellt.

4.2.1. Bewertung der Ergebnisse

Für die Bewertung der Ergebnisse wäre die Gesamtausführungszeit des DFGs auf dem CGRA das Kriterium erster Wahl. Allerdings kann dieser Wert aktuell nicht bestimmt werden, da der verwendete Scheduler ausschließlich Operationen mit maximal zwei Eingängen und einem Ausgang unterstützt. Annähernd alle sinnvollen Makro-Ops brechen diese Einschränkung.

Daher werden für die Bewertung der Ergebnisse folgende zwei Faktoren betrachtet: Die erwartete zeitliche Einsparung durch die Ersetzung pro Ausführung in Takten und die Anzahl der Vorkommen dieser Makro-Op im vorliegenden DFG. Hierbei ist zu beachten, dass sich in vielen Fällen mehrere Vorkommen einer Makro-Op überschneiden. Dies ist beispielhaft in Abbildung 4.5 dargestellt: Es sind zwei Vorkommen einer Makro-Op markiert, wobei das Ergebnis einer Addition jeweils in den rechten Eingang der Multiplikation fließt. Die beiden Vorkommen überschneiden sich allerdings im Additions-Knoten, können also nicht beide ersetzt werden. Daher darf für die Bewertung nur die maximale Anzahl an überlappungsfreien Vorkommen in Betracht gezogen werden.

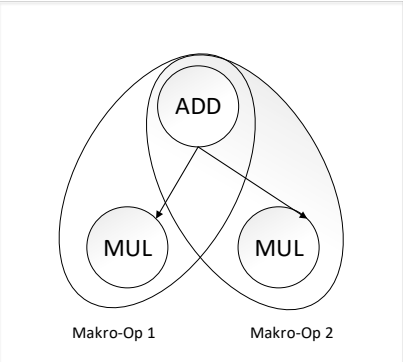

Abbildung 4.5.: Überlappung von Vorkommen.

Die *Gesamtbewertung* einer Makro-Op ist dann das Produkt dieser beiden Werte. Die Einsparung einer einzelnen Ersetzung entspricht `Latenz des Subgraphen - Latenz der Makro-Op`. Die Berechnung der Latenz der Makro-Op ist in Abschnitt 2.2.4 beschrieben. Die Latenz des Subgraphen wird durch den längsten Pfad im Subgraphen bestimmt. Die Länge dieses Pfades entspricht dann der minimalen Latenz des Subgraphen. Die Latenz des Subgraphen entspricht also der Anzahl an

Takten, nach denen der Subgraph frühestens komplett abgearbeitet sein kann. Diese minimale Zeit wird aber nur erreicht, wenn der CGRA beliebig viele Operationen parallel ausführen kann. Dies ist in der Realität nicht der Fall, da der CGRA nur eine begrenzte Zahl an PEs besitzt. Je nach Struktur des Subgraphen wird daher dessen Parallelität nicht vollständig ausgenutzt und die Ausführung benötigt mehr Takte. Die erwartete zeitliche Einsparung ist also eine Worst-Case-Abschätzung.

Es gibt noch einige weitere Faktoren, welche für die Bewertung von Makro-Ops relevant sind. Dazu zählt zum Beispiel der kritische Pfad der Makro-Op und daraus folgend der Einfluss auf die Taktfrequenz des kompletten CGRAs. Diese Faktoren werden aktuell nicht automatisch berücksichtigt; der Benutzer muss daher die vorgeschlagenen Makro-Ops manuell synthetisieren und entscheiden, ob die Ersetzung sinnvoll ist.

4.2.2. Gleichheit von Makro-Ops

Im Rahmen der Makro-Op-Optimierung muss zwischen zwei Arten von Gleichheit unterschieden werden: Die Gleichheit von zwei Makro-Ops und die Gleichheit von zwei Vorkommen einer Makro-Op. Letzteres ist selbstverständlich die stärkere Form, da die Gleichheit der Makro-Ops schon vorausgesetzt wird. Auf die Graphentheorie übertragen, ist die Gleichheit von zwei Makro-Ops dann gegeben, wenn die Struktur der beiden Subgraphen gleich ist. Für die Gleichheit der Vorkommen ist es im Gegensatz dazu notwendig, dass dieselben Subgraphen betrachtet werden. Die Gleichheit von zwei Makro-Ops wird unter anderem von den Analysen beim Speichern der Ergebnisse (siehe Abschnitt 4.2.3) benötigt, um festzustellen, ob zwei Subgraphen dieselbe Struktur besitzen und daher durch dieselbe Makro-Op ersetzt werden können. Die Gleichheit von zwei Vorkommen wird ebenfalls von den Analysen verwendet, um festzustellen, ob der aktuelle Subgraph schon betrachtet wurde.

Die Gleichheit von zwei Vorkommen ist einfach zu bestimmen. Jeder Knoten des DFGs besitzt eine eindeutige ID. Um zu prüfen, ob zwei Knoten identisch sind, müssen also nur ihre IDs verglichen werden. Um zu bestimmen, ob zwei Vorkommen identisch sind, muss dann nur geprüft werden, ob die enthaltenen Knoten paarweise identisch sind. Dies ist in der `exactEquals()`-Methode der Klasse `OpGraph` implementiert.

Der Vergleich von zwei Makro-Ops ist wesentlich komplexer, da festgestellt werden muss, ob die Struktur der beiden Subgraphen gleich ist. Da diese Funktionalität bei der Analyse häufig verwendet wird, ist eine effiziente Implementierung unerlässlich. Diese befindet sich in der `equals()`-Methode der Klasse `OpGraph`.

Das grundlegende Vorgehen sieht folgendermaßen aus: Zwei Teilknoten sind sich ähnlich, wenn ihre Vorgänger ähnlich sind und ihre Operation übereinstimmt. Falls

ein Vorgänger des Knotens außerhalb der Makro-Op liegt, kann dieser dabei ignoriert werden. Die Ähnlichkeit von reinen Eingangsknoten der Makro-Ops ist daher direkt bestimmbar: Sie sind ähnlich, wenn ihre Operation gleich ist. Derartige Knoten müssen immer vorhanden sein. Durch die Informationen, welche durch das Vergleichen der reinen Eingangsknoten gewonnen wurden, können neue Knoten, die nur Daten von außerhalb der Makro-Ops oder von bereits betrachteten Knoten konsumieren, verglichen werden. Dieses Vorgehen lässt sich so lange fortsetzen, bis alle Knoten der beiden Makro-Ops verglichen wurden. Im Folgenden werden die beiden zu vergleichenden Makro-Ops als Makro-Op A und Makro-Op B bezeichnet. Der Ablauf ist in Abbildung 4.6 grob skizziert und wird im Rest des Abschnitts detailliert beschrieben.

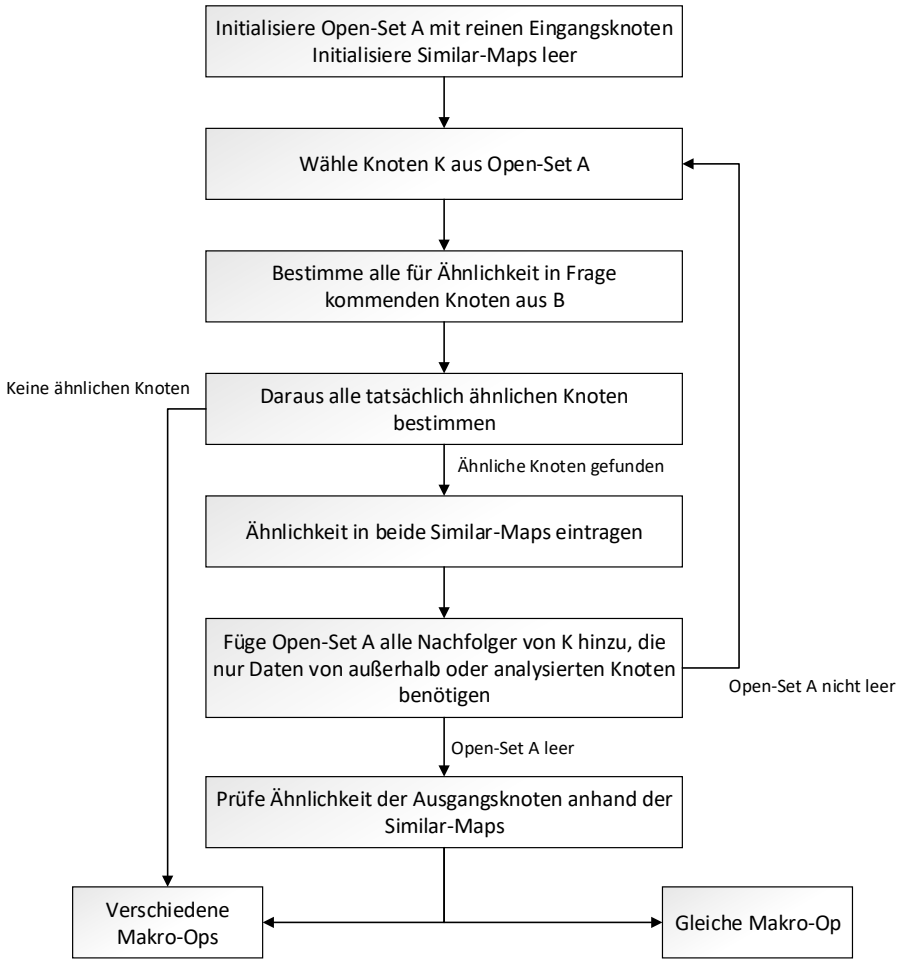

Abbildung 4.6.: Ablaufplan Gleichheit.

Für die Umsetzung werden folgende Datenstrukturen geführt: Eine Similar-Map

genannte Hash-Map speichert für jeden Teilknoten der Makro-Op eine Liste von ähnlichen Knoten der anderen Makro-Op. Diese Listen sind zu Beginn des Vergleichs leer und werden nach und nach gefüllt. Die Similar-Map wird zweimal geführt: einmal aus Sicht von Makro-Op A, einmal aus Sicht von Makro-Op B. Außerdem speichert eine Open-Set genannte Menge alle Knoten der Makro-Op A, welche aktuell zum Vergleich herangezogen werden können. Sie wird mit allen reinen Eingangsknoten von Makro-Op A initialisiert.

Es wird nun ein Knoten aus dem Open-Set entfernt und analysiert. Zuerst werden alle für die Ähnlichkeit in Frage kommenden Knoten aus B bestimmt. Dies sind alle reinen Eingabeknoten von B, wenn der neue betrachtete Knoten ein reiner Eingabeknoten von A ist. Ansonsten werden aus der Similar-Map von A alle Knoten von B ermittelt, welche mindestens einem Vorgänger des betrachteten Knotens ähnlich sind. Deren Nachfolger werden für die Ähnlichkeit in Betracht gezogen. Aus diesen in Frage kommenden Knoten werden nun diejenigen bestimmt, die dem betrachteten Knoten tatsächlich ähnlich sind. Hierfür muss nur überprüft werden, ob die Operationen der Knoten gleich sind und die Eingänge denselben Ursprung (innerhalb oder außerhalb der Makro-Op) haben. Alle als ähnlich eingestuften Knoten werden in der Similar-Map von A in die Liste des analysierten Knotens eingetragen und der Knoten aus A wird im Gegenzug in die Listen der ähnlichen Konten in der Similar-Map von B eingetragen. Wurde kein Knoten als ähnlich ermittelt, wird der komplette Vergleich abgebrochen, denn die beiden Makro-Ops sind verschieden. Ansonsten werden alle Knoten aus A ermittelt, welche durch die Bearbeitung des Knotens nun ebenfalls ins Open-Set von A eingefügt werden dürfen. Dies sind alle Nachfolger des Knotens, die ausschließlich Daten von außerhalb der Makro-Op oder von bereits analysierten Knoten konsumieren.

Sollte das Open-Set von A nicht leer sein, wird ein neuer Knoten aus diesem entfernt und wie beschrieben analysiert. Sobald das Open-Set von A leer ist, wird mit Teil zwei des Vergleichs fortgefahren:

Dort wird nun geprüft, ob die beiden Similar-Maps für alle Ausgangsknoten der jeweiligen Makro-Op mindestens einen ähnlichen Knoten aus der anderen Makro-Op enthalten. Wenn dies der Fall ist, sind die beiden Makro-Ops gleich.

Das schrittweise Vorgehen und die Verwendung der Hash-Maps ist im Vergleich zum trivialen, rekursiven Vorgehen deutlich effizienter. Dies liegt zum einen an der Vermeidung von Mehrfachbestimmung der Ähnlichkeit von Knotenpaaren, zum anderen an der Möglichkeit, direkt abbzubrechen, sobald ein Unterschied gefunden wurde. Da der Vergleich trotzdem noch aufwendig ist, aber unter anderem wegen der Verwendung von Hash-Maps zum Zwischenspeichern der Ergebnisse (Abschnitt 4.2.3) häufig gebraucht wird, wurden zu Beginn des Vergleiches mehrere Tests hinzugefügt, um den komplizierten Teil in eindeutigen Fällen wegzulassen:

Als erstes wird geprüft, ob die Anzahl an Knoten insgesamt, die Anzahl der Ein-

gangsknoten und die Anzahl der Ausgangsknoten jeweils übereinstimmen. Wenn eine dieser Bedingungen nicht zutrifft, können die Makro-Ops nicht gleich sein. Im Anschluss wird geprüft, ob die Makro-Ops identisch sind. Dieser Test ist wie oben beschrieben schnell und impliziert automatisch die Gleichheit. Um in allen anderen, nicht trivialen Fällen die Analyse für jedes beliebige Paar nur einmal durchführen zu müssen, werden die Ergebnisse außerdem gespeichert. Hierfür enthält jeder `OpGraph` eine Hash-Map, welche `OpGraphen` auf Wahrheitswerte abbilden. Sobald einmal der Vergleich von zwei Makro-Ops durchgeführt wurde, wird das Ergebnis in dieser Hash-Map in beiden `OpGraphen` gespeichert.

4.2.3. Zwischenspeichern der Analyseergebnisse

Das Zwischenspeichern der Ergebnisse ist aus den folgenden vier Gründen nicht trivial:

Erstens muss die Bewertung der einzelnen Ergebnisse während der Analyse des DFGs erfolgen und ebenfalls gespeichert werden. Dies ist notwendig, damit Ersetzungen ohne Taktersparnis nicht gespeichert werden; der benötigte Speicherplatz würde ansonsten explodieren. Außerdem können die Ergebnisse so direkt sortiert gespeichert werden. Nach der Analyse können dann einfach die besten Ergebnisse betrachtet werden.

Zum zweiten muss für die Bewertung einer Makro-Op auch die Anzahl der Vorkommen im DFG betrachtet werden (siehe Abschnitt 4.2.1). Daher ist es notwendig, für jede Makro-Op die Vorkommen zu speichern. Ein simples Mitzählen ist allerdings nicht ausreichend, da bei der Analyse ein Vorkommen mehrmals erreicht werden kann, es darf aber nicht doppelt gezählt werden. Zusätzlich überschneiden sich die Vorkommen einer Makro-Op häufig, aber es darf nur mit der maximalen Anzahl an sich nicht überlappenden Vorkommen gerechnet werden.

Darüber hinaus dürfen keine Ergebnisse mit niedriger, aber positiver Ersparnis verworfen werden. Durch die Mitbewertung der Anzahl der Vorkommen kann auch eine Makro-Op mit – einzeln betrachtet – geringer Ersparnis insgesamt einen großen Effekt haben, wenn sie häufig auftritt. Daraus folgt, kombiniert mit der Größe der DFGs, dass die Ergebniszahl schnell mehrere tausend erreichen kann. Bei jedem neuen Fund ist ein Durchsuchen der bisherigen Ergebnisse nach der gleichen Makro-Op nötig. Diese Operation muss also gut mit der Ergebnisanzahl skalieren. Dasselbe gilt für das Einfügen neuer Makro-Ops. Auch das nachträgliche Umsortieren von bereits gefundenen Makro-Ops muss ausreichend schnell möglich sein, da sich die Bewertung bei jedem neuen Vorkommen der Makro-Op ändert.

Zuletzt können bei der implementierten Heuristik unterschiedliche Einsparungspotenziale für eine Makro-Op auftreten. Da die Einsparungspotenziale immer Worst-Case gerechnet werden, ist die größte berechnete Einsparung die beste Näherung.

Um diesen Umstand sinnvoll zu berücksichtigen, sollte für die Ersparnis einer Makro-Op immer die größte bisher berechnete Einzelbewertung verwendet werden.

Aus diesen Gründen ist eine komplexe Datenstruktur zum Zwischenspeichern der Ergebnisse nötig. Diese ist in der Klasse `ResultMap` im Package `makroop` implementiert. Sie besteht aus zwei Hash-Maps, genannt Makro-Op-Map und Rating-Map. Erstere enthält als Schlüssel Makro-Ops (siehe Abschnitt 4.1.1) und bildet diese auf die entsprechenden Suchergebnisse ab. Diese Suchergebnisse beinhalten neben den Vorkommen der jeweiligen Makro-Op im DFG auch die Gesamtbewertung und die bisher beste Einzelbewertung. Dies ist notwendig, da die Vorwärtssuche verschiedene Einsparungen für eine Makro-Op berechnen kann (siehe Abschnitt 4.2.5). Da dies Worst-Case-Abschätzungen sind, sollte für die Berechnung der Gesamtbewertung immer der beste bisher gesehene Wert verwendet werden. Jedes dieser Vorkommen ist dabei eine Menge von Knoten. Um schnell die maximale Anzahl an nicht überlappenden Vorkommen bestimmen zu können, werden dabei jeweils alle größten Teilmengen ohne Überlappung gespeichert. Diese Hash-Map kann nicht durch Attribute im jeweiligen `OpGraph-Objekt` (siehe Abschnitt 4.1.1) ersetzt werden, weil die Eigenschaften nicht nur der aktuell betrachteten Makro-Op zugeordnet werden sollen, sondern allen äquivalenten Makro-Ops (siehe Abschnitt 4.2.2).

Die Rating-Map ordnet den vorkommenden Gesamtbewertungen alle Makro-Ops zu, welche aktuell mit dieser Bewertung eingestuft werden. Sie bildet also eine Zahl auf eine Menge von Makro-Ops ab. Diese Hash-Map ist notwendig, um nach Abschluss der Analyse einen effizienten Zugriff auf die bestbewerteten Ergebnisse zu ermöglichen. Dazu werden die höchsten Gesamtbewertungen durch Sortieren der Schlüsselmenge ermittelt, auf die entsprechend bewerteten Makro-Ops kann dann direkt zugegriffen werden.

Mit der Methode `addResult(OpGraph, int, Set<ANode>)` wird den Suchalgorithmen eine Möglichkeit zum Hinzufügen neuer Ergebnisse bereitgestellt. Die Methode erwartet dabei die Makro-Op-Repräsentation desselben als `OpGraph`-Objekt, die errechnete Bewertung und das Vorkommen als Menge von Knoten als Argumente. Zum Einfügen des Vorkommens muss erst ermittelt werden, ob bereits mindestens ein Vorkommen dieser Makro-Op gefunden wurde. Dies ist genau dann der Fall, wenn die Makro-Op-Map die Makro-Op bereits enthält.

Wenn dies nicht der Fall ist, wird in der Makro-Op-Map ein neuer Eintrag mit den entsprechenden Attributen angelegt. Die errechnete Bewertung wird dabei sowohl für die Einzelbewertung als auch für die aktuelle Gesamtbewertung der Makro-Op verwendet. Außerdem wird der Menge, die dieser Bewertung in der Rating-Map zugeordnet ist, die Makro-Op hinzugefügt.

Wenn die Makro-Op-Map die Makro-Op bereits enthält, werden die zugeordneten

Werte gelesen. Diese werden wie folgt aktualisiert: Die beste Einzelbewertung der Makro-Op wird neu gesetzt, falls die neu berechnete Bewertung größer ist als der aktuelle Wert. Das neu gefundene Vorkommen wird zu den bisherigen ergänzt. Aus Performancegründen wird dabei die mögliche Überlappung mit anderen Vorkommen nicht beachtet. Stattdessen wird einfach die Anzahl an Vorkommen benutzt, um die Gesamtbewertung der Makro-Op zu aktualisieren. Die Anzahl der überlappungsfreien Vorkommen wird erst beim Auslesen der besten Ergebnisse bestimmt. Außerdem wird die Rating-Map aktualisiert, indem die Makro-Op aus der Menge der alten Gesamtbewertung entfernt und in die Menge der neuen Gesamtbewertung ergänzt wird. Diese beiden Schritte werden auch durchgeführt, wenn sich die beste Einzelbewertung im vorherigen Schritt geändert hat.

Zum Extrahieren der besten Ergebnisse stellt die Klasse die Methode `toList(int)` zur Verfügung. Sie bekommt als einziges Argument eine Zahl, die bestimmt, wie viele Ergebnisse extrahiert werden, erzeugt eine Liste mit den besten Ergebnissen und gibt diese zurück.

Zu Beginn werden alle ermittelten Bewertungen durch Sortieren der Schlüsselmenge der Rating-Map ermittelt. Diese ermittelten Bewertungen werden nun nacheinander abgearbeitet. Dafür werden aus der Rating-Map alle Makro-Ops mit dieser Gesamtbewertung ermittelt. Dabei muss allerdings beachtet werden, dass die Gesamtbewertungen während der Analyse falsch bestimmt wurden, da die Gesamtzahl der Vorkommen dieser Makro-Op anstelle der Anzahl überlappungsfreier Vorkommen verwendet wurde. Daher wird nun diese Anzahl der überlappungsfreien Vorkommen bestimmt. Falls diese sich von der Gesamtzahl der Vorkommen unterscheidet, wird die Makro-Op nicht der Ergebnisliste hinzugefügt, sondern anhand der korrekten Gesamtbewertung in der Rating-Map eingeordnet. Falls die Gesamtbewertung nicht schon in der Menge der ermittelten Bewertungen enthalten ist, wird sie in diese passend einsortiert. Nur falls die Makro-Op nicht umsortiert werden musste, weil ihre Gesamtbewertung korrekt war, wird sie in die Ergebnisliste aufgenommen. Dies gilt insbesondere auch dann, wenn die Makro-Op schon vorher umsortiert wurde und jetzt erneut betrachtet wird. Dies wird solange fortgesetzt, bis die Ergebnisliste die geforderte Größe erreicht hat.

4.2.4. Vollständige Suche

Diese Analyse durchsucht den DFG nach allen möglichen Ersetzungen durch eine Makro-Op.

Sie kann durch Aufruf der Methode `makroOpExhaustiveSearch()` in der `IDP`-Klasse des Packages `graph` gestartet werden und liefert eine sortierte Liste der besten Ersetzungsmöglichkeiten. Sie lässt sich über die Eigenschaften `minsize`, `maxsize` und `resultsize` der Klasse `IDP` beeinflussen. Die ersten beiden schränken die Größe (in

Knoten) möglicher Makro-Ops ein, `resultsize` bestimmt die Größe der sortierten Ergebnisliste.

Neben allen bisher ermittelten Ergebnissen in einer ResultMap (Abschnitt 4.2.3) führt die Suche ebenfalls den aktuell betrachteten Subgraphen als Hash-Set von Knoten.

Die eigentliche Analyse findet in der Methode `exhaustiveVisitNode()` statt. Diese erwartet als Argumente die ResultMap, den aktuellen Subgraphen und den zu besuchenden Knoten. Diese Methode wird von der Hauptmethode nacheinander für jeden in Frage kommenden Knoten (siehe Einschränkungen in Abschnitt 4.1.1) aufgerufen.

Die Methode fügt den Knoten zuerst dem Subgraphen hinzu. Danach erstellt sie ein `OpGraph`-Objekt (Abschnitt 4.1.1) für diesen Subgraphen, falls er sich in der durch `minsize` und `maxsize` spezifizierten Größenordnung befindet. Die Bewertung dieses `OpGraphen` wird berechnet, indem die Latenz der Makro-Op von der Latenz des Subgraphen subtrahiert wird. Wenn diese Bewertung größer als null ist, wird der aktuelle Subgraph mit dieser Bewertung in die ResultMap eingetragen.

Wenn der Subgraph kleiner als `maxsize` ist, wird die Analyse fortgesetzt. Hierzu werden alle in Frage kommenden (siehe Abschnitt 4.1.1) Nachbarknoten des Subgraphen bestimmt. Es erfolgt dann ein rekursiver Aufruf für jeden dieser Nachbarknoten. Zur Überprüfung der Konvexität des Subgraphen wird hier nur der neue Knoten betrachtet, denn der vorherige Subgraph muss schon konvex sein. Dafür müssen allerdings sowohl die transitiven Nachfolger des Knotens als auch seine transitiven Vorgänger überprüft werden: Der Knoten darf dem Subgraphen nicht hinzugefügt werden, falls einer der transitiven Vorgänger in den Nachfolgern des Subgraphen enthalten ist oder einer der transitiven Nachfolger in den Vorgängern des Subgraphen.

Vor dem Verlassen der Methode wird der betrachtete Knoten aus der Menge entfernt, um den Zustand des Aufrufers wiederherzustellen.

Ein großes Problem dieser Vollständigen Suche ist das mehrfache Besuchen desselben Vorkommens, da dies die benötigte Analysezeit deutlich erhöht. Um das Problem zu reduzieren, gibt die ResultMap beim Einfügen eines neuen Vorkommens Rückmeldung, ob exakt dieses Vorkommen bereits vorhanden war. Ist dies der Fall, wurde der aktuelle Subgraph – und damit auch alle größeren Subgraphen, die diesen Subgraphen enthalten – bereits analysiert. Daher wird die Analyse beim aktuellen Subgraphen nicht fortgesetzt, sondern der betrachtete Knoten wird direkt aus der Menge entfernt und die Methode verlassen.

Diese Abbruchbedingung betrifft allerdings nur die Makro-Ops mit einer positiven Bewertung; die negativ bewerteten werden nicht gespeichert. Da Letztere al-

lerdings einen wesentlichen Teil aller Möglichkeiten darstellen, werden optional alle Vorkommen mit negativer Bewertung in einer zusätzlichen Menge gesammelt. Wenn das aktuell betrachtete Vorkommen in dieser Menge bereits enthalten ist, wird die Analyse ebenfalls nicht fortgesetzt. Dies ist optional, da durch das Speichern der negativ bewerteten Makro-Ops der Speicherverbrauch deutlich ansteigt. Mithilfe der Boolean-Variable `skipAdditionalBreakCondition` der Klasse `IDP` kann dies deaktiviert werden: Wenn sie den Wert **`true`** hat, wird die Menge nicht geführt und die Abbruchbedingung nicht geprüft.

4.2.5. Heuristik

Die Vollständige Suche hat zwei wesentliche Probleme bezüglich ihrer Laufzeit:

Erstens ist der Suchraum schon bei moderaten Graphengrößen zu groß (vergleiche Kapitel 5), denn die Anzahl der betrachteten Subgraphen erreicht trotz der frühzeitigen Abbruchbedinungen schnell mehrere Millionen. Die Anzahl der relevanten Makro-Ops mit einer positiven Bewertung ist nur ein Bruchteil hiervon. Bei größeren DFGs ist die Laufzeit dann so groß, dass eine sinnvolle Verwendung der Vollständigen Suche nicht mehr möglich ist.

Zweitens wurde mithilfe eines CPU-Zeit-Samplings festgestellt, dass ein wesentlicher Teil der benötigten Zeit auf die Laufzeitanalyse (Abschnitt 4.1.2) verwendet wird. Diese ist im allgemeinen Fall nicht trivial, da die optimale Reihenfolge der Ein- und Ausgänge der Makro-Op bestimmt werden muss.

Beim Analysieren der besten Ergebnisse der Vollständigen Suche fällt eine Gemeinsamkeit vieler Ergebnisse auf, welche zur Einschränkung des Suchraums genutzt werden kann:

Fast alle guten Ersetzungen bestehen aus langen Ketten von Operationen, die sich, wenn überhaupt, im unteren Teil etwas auffächern. Breite Makro-Ops treten praktisch gar nicht auf. Diese Beobachtung lässt sich mit der I/O-Begrenzung einer Makro-Op erklären. Da der Makro-Op nur zwei Eingangsports und ein Ausgangsport zur Verfügung stehen, schneiden Ersetzungen schlecht ab, die zum Beispiel für ihre optimale Laufzeit viele Eingänge gleichzeitig benötigen würden. Vor der Ersetzung stellt dies kein Problem dar, weil die Operationen auf mehrere PEs verteilt sind und damit jede Operation genügend Eingangs- und Ausgangsports zur Verfügung hat. Am besten schneiden daher Makro-Ops ab, die allgemein wenige Eingänge und Ausgänge besitzen. Wenn die verschiedenen Pfade von den Ein- zu den Ausgängen unterschiedliche Latenzen aufweisen, wirkt sich dies ebenfalls positiv aus. Die begrenzte Zahl an Ports spielt dann nur eine untergeordnete Rolle, da die Ein- bzw. Ausgänge ohne Laufzeiteinbußen in aufeinanderfolgenden Takten angeordnet werden können.

Der Suchraum lässt sich daher ohne Verlust von wesentlichen Ergebnissen deutlich einschränken: Statt wie bei der Vollständigen Suche alle Nachbarn des aktuellen Subgraphen in Betracht zu ziehen, werden bei der Heuristik nur die Nachfolger des aktuellen Subgraphen betrachtet. Die langen Ketten aus Operationen werden dabei trotzdem gefunden. Breite Makro-Ops stellen wie oben beschrieben im Allgemeinen keine guten Ersetzungen dar, weshalb durch deren Verlust keine relevanten Ergebnisse verloren gehen. Ein detaillierter Vergleich der gefundenen Ergebnisse beider Analysen findet sich in Kapitel 5.

Diese Einschränkung auf die Nachfolger des aktuellen Subgraphen birgt einen weiteren Vorteil: Das Bestimmen der optimalen Reihenfolge der Eingänge der Makro-Op ist wesentlich einfacher, da die Makro-Op nur nach unten erweitert wird. Dadurch hängen alle Ausgänge der Makro-Op vom zuerst betrachteten Knoten ab. Dessen Eingänge als erstes einzulesen, führt also in jedem Fall zu einer optimalen Reihenfolge. Wenn die Makro-Op eine Kette ist – das heißt alle Knoten der Makro-Op besitzen maximal einen Nachfolger, der ebenfalls in der Makro-Op enthalten ist – ist die Reihenfolge der restlichen Eingänge ebenfalls offensichtlich: Die Eingänge werden in der Reihenfolge eingelesen, in der ihre Knoten zur Makro-Op hinzugefügt wurden.

Wenn sich die Kette allerdings irgendwann aufspaltet – das heißt ein Knoten der Makro-Op hat mindestens zwei Nachfolger innerhalb der Makro-Op – ist nicht mehr offensichtlich, wessen Eingänge zuerst eingelesen werden müssen. Das Anordnen anhand der Hinzufügungsreihenfolge zur Makro-Op erzeugt dann nicht in allen Fällen optimale Ergebnisse. Dies kann aber leicht behoben werden, indem alle Hinzufügungsreihenfolgen von der Analyse abgedeckt werden, denn mindestens eine Reihenfolge liefert das optimale Ergebnis. Für die endgültige Bewertung dieser Makro-Op wird einfach der beste erzielte Wert verwendet. Dies bedeutet, dass ein in der Vollständigen Suche verwendetes Abbruchkriterium geändert werden muss: Dort wurde ein Suchzweig abgebrochen, sobald ein bereits in der ResultMap enthaltenes Vorkommen gefunden wurde. Um nun die verschiedenen Hinzufügungsreihenfolgen abzudecken, darf nur abgebrochen werden, wenn das Vorkommen bereits gefunden wurde und die aktuell berechnete Bewertung schlechter als die beste bisher gesehene Bewertung ist oder dieser entspricht.

Mit dieser Einschränkung lässt sich auch das zweite oben beschriebene Problem lösen. Die auf die Laufzeitanalyse verwendete Zeit kann deutlich verringert werden, indem die Reihenfolge der Eingänge wie beschrieben direkt während der Analyse bestimmt wird. Auch die restlichen Teile der Laufzeitanalyse können parallel zur Analyse berechnet werden, um zusätzlich Zeit zu sparen.

Auf Basis dieser Überlegungen wurde eine Heuristik entwickelt, welche über die Methode `makroOpForwardSearch()` der Klasse `IDP` zugänglich ist. Bezogen auf die

Einschränkung des Suchraums wird sie im Folgenden auch als Vorwärtssuche bezeichnet. Ähnlich wie bei der Vollständigen Suche findet auch hier die eigentliche Analyse in der Methode `forwardSearchVisitNode()` statt. Diese wird ebenfalls für jeden in Frage kommenden Knoten des DFG aufgerufen und sucht dann nach möglichen Makro-Ops, die mit diesem Knoten beginnen. Dieser Knoten wird im Folgenden als aktueller Startknoten bezeichnet.

Vergleichbar zur Vollständigen Suche wird ebenfalls eine ResultMap mit allen bisher ermittelten Ergebnissen und der aktuelle Subgraph als Menge von Knoten geführt, wobei eine sortierte Liste der besten Ergebnisse zurückgeliefert wird. Auch die drei Parameter (`minsize`, `maxsize`, `resultsize`) können benutzt werden, um das Verhalten zu beeinflussen.

Zudem führt die Vorwärtssuche zusätzliche Datenstrukturen, um – wie oben beschrieben – die Laufzeiten parallel zur Analyse zu bestimmen. Hierfür werden zwei Hash-Maps benötigt: Die Before-Map bildet Knoten auf Zahlen ab. Die Zahl ist dabei der Takt, in welchem der jeweilige Knoten sein Ergebnis ohne Ersetzung durch eine Makro-Op geschrieben hat, wenn der aktuelle Startknoten in Takt 1 mit dem Einlesen begonnen hat. Die Hash-Map enthält also die Latenzen der Pfade vom aktuellen Startknoten zu den jeweiligen als Schlüssel in der Hash-Map vorhandenen Knoten. Für die Latenz des Subgraphen (Abschnitt 4.1.2) muss also der maximale Wert aller Ausgangsknoten des Subgraphen verwendet werden.

Für das Bestimmen der Latenz der Makro-Op wird die zweite Hash-Map verwendet, im Folgenden Finished-Map genannt. Diese bildet Knoten auf Paare aus Zahlen ab. Sie speichert dabei für jeden Knoten die Latenz vom Startknoten nach der Ersetzung durch eine Makro-Op bis zum Abschluss seiner Berechnung. Außerdem speichert sie die mögliche Ausgangsüberlappung der Operation. Mithilfe der Finished-Map kann dann die Latenz der Makro-Op bestimmt werden. Beide Hash-Maps werden auch für die Berechnung der jeweiligen Werte der Nachfolgerknoten benötigt.

Die Verwendung der beiden Hash-Maps wird im Folgenden am in Abbildung 4.7 dargestellten Beispiel illustriert. Die drei Zahlen in den Knoten beschreiben jeweils die Eingabeübelappung, die Berechnungslatenz und die Ausgabeüberlappung. Die Datenbreite entspricht immer der Datenpfadbreite des CGRAs, die Eingabe- und Ausgabelatenz beträgt also immer 1.

Die Before-Map enthält folgende Einträge, wenn der aktuell betrachtete Subgraph alle drei Knoten enthält: Dem Additions-Knoten ist der Endtakt 1 zugeordnet. Dem SQR-Knoten ist der Endtakt 5 zugeordnet, da er eine Gesamtlatenz von 4 besitzt und im zweiten Takt starten konnte. Dem Negations-Knoten ist der Endtakt 6 zugeordnet, da er im sechten Takt startet und seine Gesamtlatenz 1 beträgt.

Die Finished-Map ordnet in dieser Situation den Knoten folgende Werte zu: Dem Additions-Knoten wird eine Latenz von 1 zugewiesen, da er im ersten Takt startet

Abbildung 4.7.: Beispielgraph für die Vorwärtssuche.

und eine Berechnungslatenz von eins besitzt, die sich allerdings mit dem Einlesen überschneidet. Seine mögliche Ausgangsüberlappung beträgt ebenfalls 1. Dem SQR-Knoten ist eine Latenz von 3 zugeordnet, da seine Berechnung erst im zweiten Takt starten kann (seine Eingangsüberlappung beträgt 0) und seine Berechnungslatenz 2 beträgt. Die mögliche Ausgangsüberlappung beträgt 0. Dem Negations-Knoten ist eine Latenz von 4 zugeordnet. Er kann im vierten Takt mit der Berechnung starten, da die gespeicherte Ausgangsüberlappung seines Vorgängers 0 beträgt und seine Berechnung einen Takt dauert. Die ihm zugeordnete Ausgangsüberlappung beträgt 1. Alle Werte sind in Tabelle 4.1 gesammelt.

Knoten	Before-Map	Finished-Map
ADD	1	(1,1)
SQR	5	(3,0)
NEG	6	(4,0)

Tabelle 4.1.: Inhalte der Hash-Maps.

Zusätzlich zu diesen beiden Hash-Maps wird noch je eine Menge von Knoten für die Ausgangsknoten und die Operandenknoten des aktuellen Subgraphen geführt. Beide Mengen sind für das Erstellen einer Makro-Op notwendig (Abschnitt 4.1.1) und müssten ansonsten für jede mögliche Makro-Op neu bestimmt werden.

Die Methode `forwardSearchVisitNode()` wird von der Hauptmethode mit dem jeweils aktuellen Startknoten und den beschriebenen Datenstrukturen aufgerufen.

Zusätzlich bekommt sie noch für beide Eingangsports der Makro-Op den ersten Takt übergeben, in welchem dieser Port unbelegt ist. Der niedrigere Wert wird im Argument `lowerinput`, der höhere Wert im Argument `higherinput` übergeben. Beim Aufruf aus der Hauptmethode ist dies jeweils eins. Ihr Ablauf ist in Abbildung 4.8 überblicksmäßig dargestellt und wird im Folgenden detailliert beschrieben.

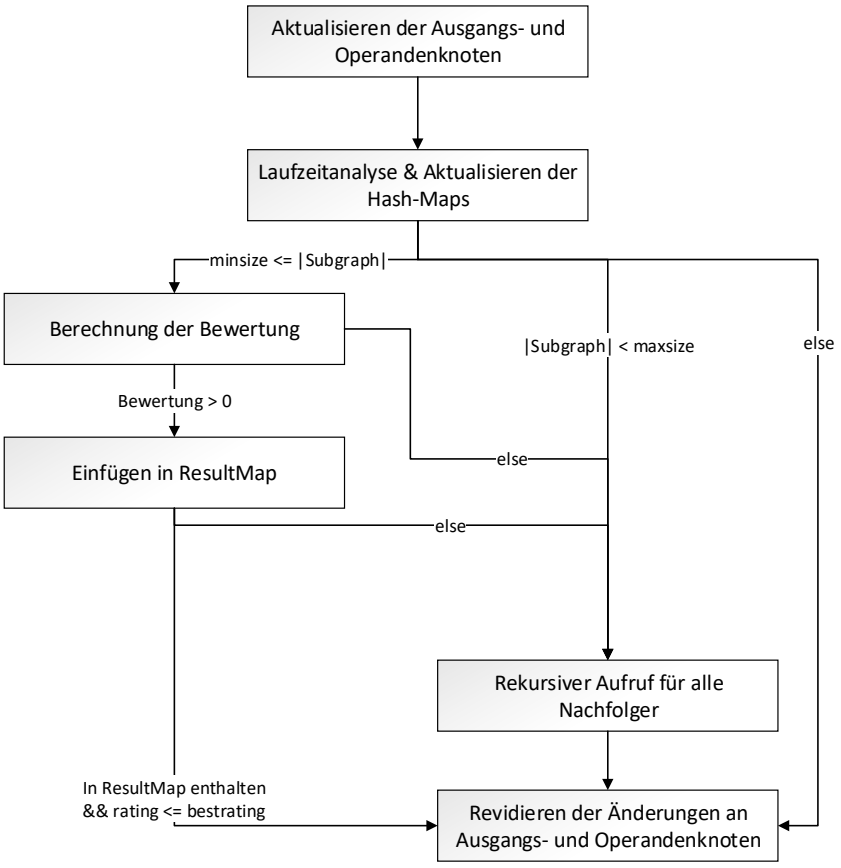

Abbildung 4.8.: Ablaufplan Vorwärtssuche.

Zu Beginn der Methode wird der neue Knoten der Menge von Ausgangsknoten hinzugefügt, da keiner seiner Nachfolger bereits in der Makro-Op enthalten sein kann. Außerdem wird jeder seiner Vorgänger, der nun keine Nachfolger außerhalb der Makro-Op mehr besitzt, aus der Menge entfernt. Alle entfernten Knoten werden in einer lokalen Liste gespeichert. Um den vorherigen Zustand wiederherzustellen, werden vor dem Verlassen der Methode alle Knoten in dieser Liste wieder der Menge von Ausgangsknoten hinzugefügt und der aktuelle Knoten wird aus dieser entfernt.

Ebenfalls zu Beginn der Methode werden der Menge der Operandenknoten alle Vorgänger des neuen Knotens hinzugefügt, welche nicht zur Makro-Op gehören und

noch nicht in der Menge enthalten waren. Die hinzugefügten Knoten werden ebenfalls in einer lokalen Liste gespeichert und vor dem Verlassen der Methode wieder aus der Menge entfernt.

Für das Aktualisieren der Before-Map wird zuerst der größte Wert aller Vorgänger des aktuellen Knotens aus dieser ermittelt, wobei für Vorgängerknoten außerhalb der Makro-Op der Wert 0 verwendet wird. Dies ist der Takt, in welchem der aktuelle Knoten mit seiner Berechnung starten kann. Er wird um die Gesamtlatenz des Knotens erhöht und für diesen in die Hash-Map eingefügt.

Um die Finished-Map zu aktualisieren, muss erst die Belegung der Eingangsports auf den aktuellen Stand gebracht werden. Dafür wird für jeden lokal der Operandenmenge hinzugefügten Knoten der `lowerinput`-Wert um die Eingabelatenz des aktuellen Knotens erhöht. Nach jeder Erhöhung werden `lowerinput` und `higherinput` bei Bedarf getauscht. Anschließend muss der Takt ermittelt werden, in welchem die aktuelle Teiloperation beginnen kann. Dies ist dann der Fall, wenn alle benötigten Daten bereitstehen. Dafür müssen sowohl alle direkt benötigten Eingänge der Makro-Op eingelesen sein, als auch alle internen Vorgänger ihre Berechnung abgeschlossen haben. Für beide Werte muss auch die jeweils mögliche Überlappung berücksichtigt werden. Der Takt, in dem alle Eingänge der Makro-Op bereitstehen, lässt sich aus `higherinput` bestimmen. Dessen Wert muss um eins reduziert werden, da er den ersten freien Takt angibt, aber der letzte belegte Takt benötigt wird. Außerdem muss noch die Eingangs-Überlappung des aktuellen Knotens subtrahiert werden. Für den anderen Wert können die Werte aller internen Vorgänger aus der Finished-Map betrachtet werden. Es wird das Wertepaar gesucht, bei welchem die Latenz vom Startknoten minus die mögliche Überlappung maximal ist. Der aktuelle Knoten kann dann im größeren von diesen beiden Werten starten, wobei dieser Wert plus seine Berechnungslatenz den Takt angibt, in dem er beendet ist. Der Finished-Map wird nun ein neuer Eintrag für den aktuellen Knoten hinzugefügt, mit dem Takt, in dem er beendet ist, und seiner Ausgangsüberlappung als Wertepaar.

Der restliche Ablauf der `forwardSearchVisitNode`-Methode ist nun ähnlich der entsprechenden Methode der Vollständigen Suche: Falls die aktuelle Größe des Subgraphen im Bereich zwischen `minsize` und `maxsize` liegt, wird die Bewertung berechnet. Dies erfolgt allerdings nicht mit den in Abschnitt 4.1.2 beschriebenen Methoden, sondern anhand der vorher berechneten Werte.

Für die Latenz des Subgraphen wird der größte Wert aller Ausgangsknoten aus der Before-Map ermittelt.

Für die Latenz der Makro-Op werden alle Wertepaare von Ausgangsknoten aus der Finished-Map gesammelt und anhand der Differenz von Latenz und Überlappung aufsteigend sortiert. Diese werden nun nacheinander betrachtet, um den Takt zu ermitteln, in dem der letzte Ausgang der Makro-Op geschrieben wurde. Dabei muss beachtet werden, dass die Makro-Op nur einen Ausgangsport besitzt. Für je-

des Paar aus Latenz und Überlappung wird daher der Takt berechnet, in welchem mit dem Schreiben dieses Ergebnisses begonnen werden kann. Dies ist der kleinste Takt, in dem sowohl das Ergebnis der Teiloperation bereitsteht (Latenz minus Überlappung), als auch das Ergebnis des vorherigen Wertepaares fertig geschrieben wurde. Zu diesem Starttakt des Schreibens wird die Ausgabelatenz addiert, um den Takt zu berechnen, ab welchem der Ausgangsport wieder frei ist. Nach dem Abarbeiten aller Paare kann nun die Bewertung der Makro-Op durch die Differenz von der Latenz des Subgraphen und dem gerade ermittelten letzten Ausgabetakt der Makro-Op ermittelt werden.

Falls die Bewertung größer null ist, wird der `OpGraph` (Abschnitt 4.1.1) der Makro-Op erstellt, wobei die Operandenmenge und die Ausgangsmenge verwendet werden. Dieser wird in die ResultMap eingefügt.

Falls die ResultMap exakt diesen Subgraphen schon enthält, muss die Analyse nicht weiter fortgesetzt werden, da alle größeren Subgraphen, die diesen Subgraphen enthalten, bereits analysiert wurden. Im Gegensatz zur Vollständigen Suche muss die Analyse allerdings trotzdem fortgesetzt werden, falls die neu berechnete Bewertung der Makro-Op besser ist als alle vorher berechneten. Denn wenn der aktuelle Subgraph nun besser bewertet wird, trifft dies eventuell auch auf die weiteren Subgraphen im Suchzweig zu.

Die Analyse wird außerdem nur fortgesetzt, wenn der Subgraph aktuell echt kleiner ist als `maxsize`. Die Methode wird mit allen in Frage kommenden (siehe Abschnitt 4.1.1) Nachfolgern des Subgraphen rekursiv aufgerufen. Da der Knoten ein Nachfolger des Subgraphen ist, müssen für die Überprüfung der Konvexität nur die transitiven Vorgänger des Knotens auf Übereinstimmung mit den Nachfolgern des Subgraphen überprüft werden.

Um die Laufzeit der Heuristik weiter zu beschleunigen, wurde außerdem eine weitere optionale Abbruchbedingung definiert: Die Analyse wird nur fortgesetzt, wenn die Ergänzung des Subgraphen auf `maxsize` Knoten noch eine positive Bewertung erreichen kann. Um dies zu überprüfen, wird angenommen, dass sich die Bewertung pro hinzugefügtem Knoten maximal um eins verbessern kann. Diese Annahme gilt in vielen – aber nicht allen Fällen – weshalb diese Abbruchbedingung optional ist. Mithilfe der Boolean-Variable `skipAdditionalBreakCondition` der Klasse `IDP` kann dies deaktiviert werden: Wenn sie den Wert **`true`** hat, wird die Abbruchbedingung nicht geprüft. Die Ergebnisse der Heuristik mit und ohne diese letzte Abbruchbedingung wurden ebenfalls in Kapitel 5 evaluiert.

5. Auswertung

In diesem Kapitel werden die implementierten Analysen evaluiert. Einerseits werden die Analysen (Vollständige Suche und Heuristik) und ihre Varianten hinsichtlich ihrer Ergebnisse, ihrer Laufzeiten und ihres Speicherverbrauchs verglichen. Andererseits werden die Einsparungspotenziale der gefundenen Ergebnisse betrachtet.

Die beiden Analysen, jeweils mit und ohne optionale Abbruchbedingung, wurden dafür auf einem Testsystem mit diversen DFGs angewendet. Die verwendeten Graphen mit ihren Knotenzahlen bei Euler- beziehungsweise Heun-Integratoren sind in Tabelle 5.1 aufgeführt. Das Testsystem verwendet einen Intel i5-6600K Prozessor mit vier Kernen und einem Basistakt von 3,50 GHz (Turbo 3,90 GHz) und 16 GByte DDR4-RAM. Für jeden DFG wurden dabei die Werte drei bis neun als maximale Größe einer Makro-Op verwendet, die minimale Größe war auf zwei festgelegt. Für die Betrachtung der Laufzeit und des Speicherverbrauchs wurde jeder Analyselauf dreimal durchgeführt und anschließend der Median der Werte gewählt, um einzelne Ausreißer auszusortieren.

Drei weitere Beispielgraphen (Woodpecker, Katze_greifer und BeamHcs) konnten für die Evaluation nicht verwendet werden, da ihre Ausführungszeit schon bei kleinen Maximalgrößen länger als zwölf Stunden betrug. Dies liegt nicht an ihrer Knotenzahl – alle drei sind kleiner als LinearDriveControl – sondern an der großen Anzahl an Vorkommen einiger weniger Makro-Ops. Wie in Abschnitt 4.2.3 beschrieben, arbeitet die aktuelle Implementierung bei großen Vorkommenzahlen nur schnell, wenn sich entweder fast alle Vorkommen überschneiden oder kaum Überschneidungen auftreten. Bei den drei oben genannten Graphen ist dies nicht der Fall und daher benötigt das Bestimmen der Überlappungen sehr viel Zeit.

Name des DFG	Knotenzahl Euler	Knotenzahl Heun
TwoMassesBouncing	106	159
Verlade	132	214
DoublePendulum	393	456
LinearDriveControl	3614	4405

Tabelle 5.1.: Knotenanzahl der DFGs.

In den folgenden Abschnitten werden die genannten Kriterien betrachtet und veranschaulicht.

5.1. Laufzeit

Für die Laufzeit der Analysen wurde vor dem Start und nach dem Ende die aktuelle Zeit mithilfe der Methode `System.nanoTime()` ermittelt. Alle Werte sind in Kapitel A aufgeführt. Es ist jeweils der Median (in Sekunden) der drei Durchläufe angegeben, Ausreißer traten allerdings auch keine auf. Im Folgenden werden beispielhaft einige Werte veranschaulicht.

Die Laufzeit der verschiedenen Analysen auf den Beispielgraphen ist in Abbildung 5.1 dargestellt. Um auch die kleinen Werte gut darstellen zu können, wurde eine logarithmische Skalierung gewählt. Die dargestellten Laufzeiten gehören zu den Durchläufen mit einer Maximalgröße der Makro-Ops von neun.

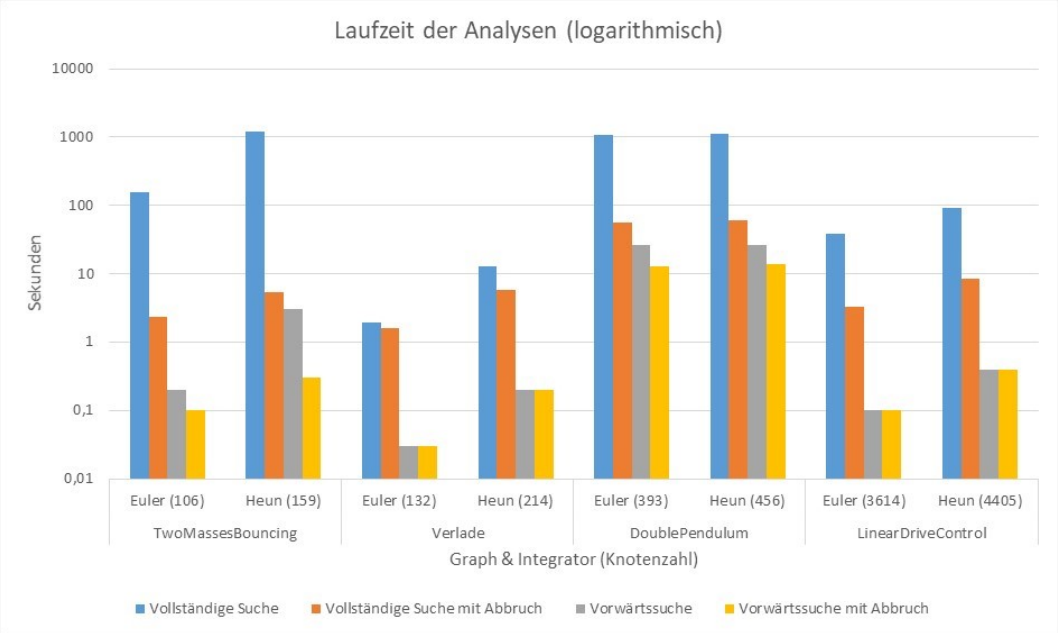

Abbildung 5.1.: Laufzeiten der Analysen für Maximalgröße neun (logarithmische Skala).

In Abbildung 5.2 werden die Laufzeiten für die verschiedenen Maximalgrößen von drei bis neun beispielhaft an DoublePendulum mit Heun-Integratoren veranschaulicht; wieder ist eine logarithmische Skalierung gewählt.

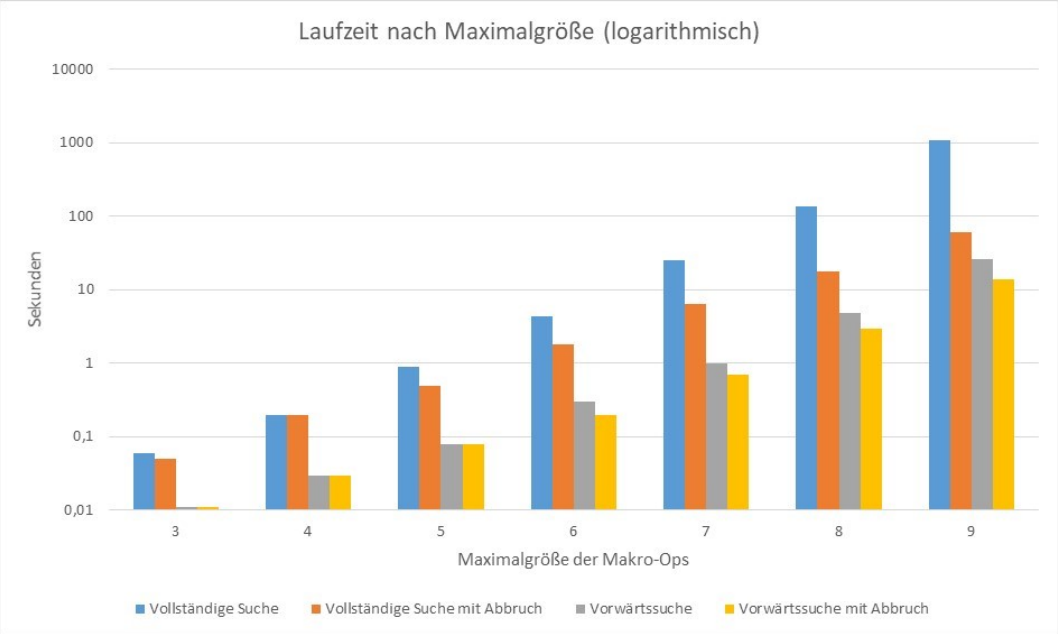

Abbildung 5.2.: Laufzeiten der Analysen für DoublePendulum (logarithmische Skala).

5.2. Speicherverbrauch

Der Speicherverbrauch der Analysen während der Ausführung wurde mithilfe von `ManagementFactory.getMemoryPoolMXBeans()` ermittelt. Für alle MemoryPools vom Typ Heap wurde dann mithilfe von `getPeakUsage()` der höchste Wert während des letzten Durchlaufs ermittelt und aufsummiert. In Abbildung 5.3 wird der benötigte Hauptspeicher (in Gigabyte) der verschiedenen Analysen auf den Beispielgraphen dargestellt. Die Maximalgröße der Makro-Ops war dabei wieder neun. Es ist jeweils der Median der drei Ausführungen angegeben. Bei einigen wenigen Fällen gab es große Unterschiede zwischen einem Durchlauf und den anderen beiden. In diesem Fall wurde dieser Ausreißer für den Durchschnitt nicht berücksichtigt.

In Abbildung 5.4 wird der Speicherverbrauch des DoublePendulum-DFGs mit Heun-Integratoren für die verschiedenen Maximalgrößen veranschaulicht.

5.3. Gefundene Ersetzungsmöglichkeiten

Für den Vergleich der Ergebnisse werden jeweils die zehn am besten bewerteten Funde der Analysen betrachtet. Alle Ergebnisse sind in Kapitel B aufgeführt.

Der Vergleich der ermittelten Ergebnisse von Vollständiger Suche mit und ohne Abbruchbedingung liefert keine Unterschiede: Beide Varianten liefern bei den Bei-

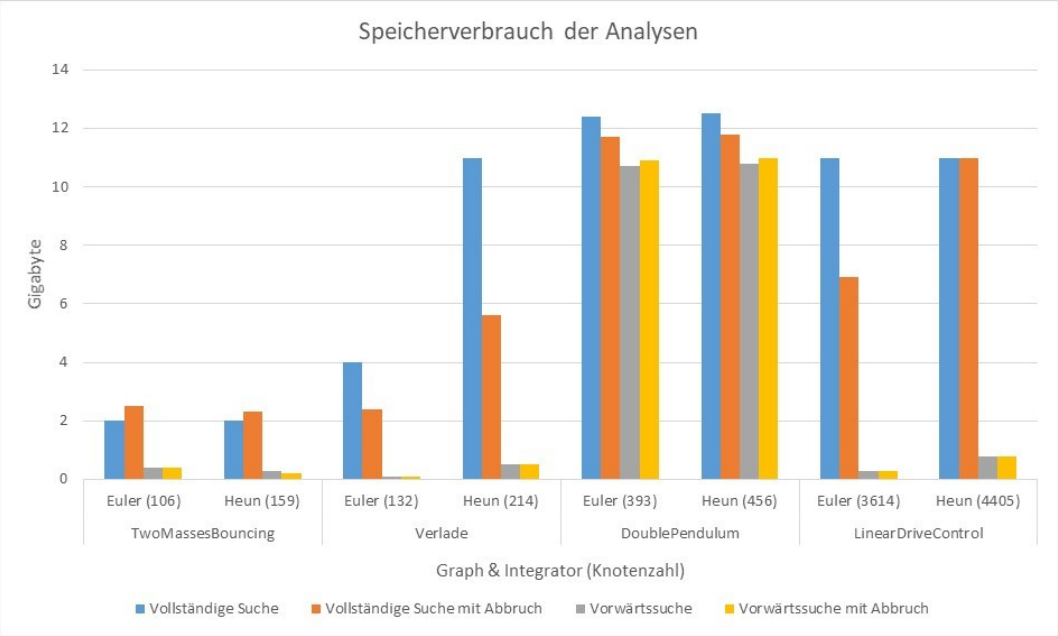

Abbildung 5.3.: Speicherverbrauch der Analysen für Maximalgröße neun.

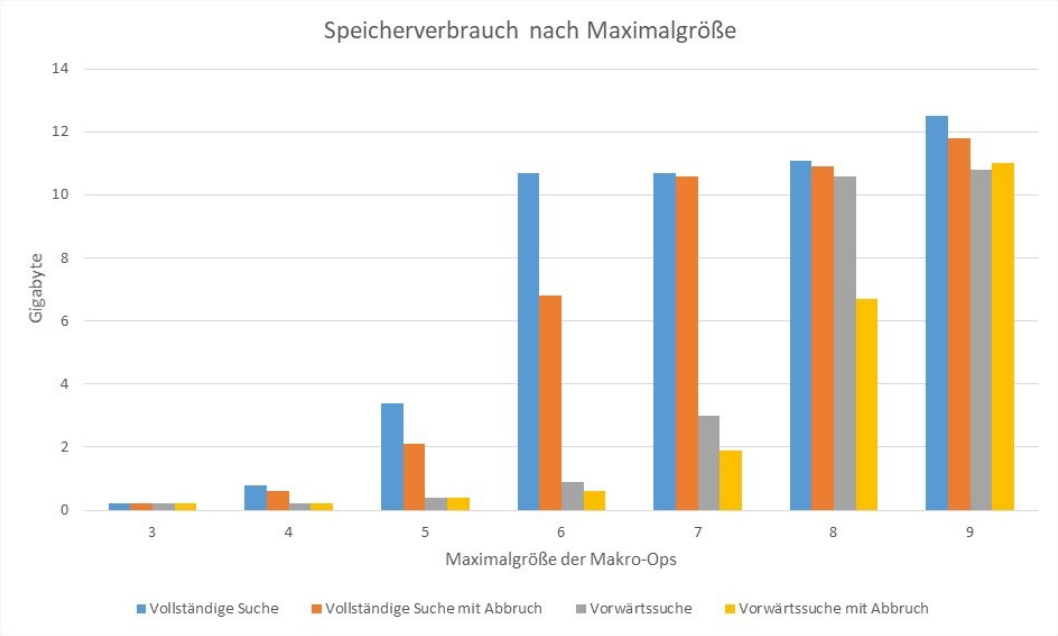

Abbildung 5.4.: Speicherverbrauch der Analysen für DoublePendulum.

spielgraphen dieselben Ergebnisse. Dasselbe gilt auch für die beiden Varianten der Vorwärtssuche. Im Folgenden werden daher nur noch die Ergebnisse von Vollstän-

diger Suche und Vorwärtssuche detaillierter verglichen.

Dort treten dann auch einige Unterschiede auf: Die Vollständige Suche liefert einzelne Ersetzungen, die die Vorwärtssuche nicht ermitteln kann. Diese Unterschiede werden allerdings immer geringer, je weniger Ergebnisse betrachtet werden. Bei einer Beschränkung auf fünf Ergebnisse treten bei den Beispielgraphen gar keine Unterschiede auf. Die Vollständige Suche findet also nur einzelne zusätzliche Ergebnisse, die auch nicht zu den höchstbewerteten Ersetzungen zählen. In allen Fällen liefert die Vorwärtssuche stattdessen Ersetzungen, die gleich oder nur minimal schlechter bewertet werden. Diese Ergebnisse tauchen bei der Vollständigen Suche weiter hinten in der Ergebnisliste auf. Die Unterschiede werden im Folgenden für den DoublePendulum-DFG beschrieben, sind jedoch bei den anderen Beispielgraphen vergleichbar. Bei Betrachtung der besten zehn Ergebnisse treten bei DoublePendulum ausschließlich mit Heun-Integratoren und einer Maximalgröße von acht Unterschiede auf. Diese Unterschiede sind in Tabelle 5.2 veranschaulicht. Die prozentuale Angabe in der zweiten Spalte setzt die Einzelbewertung in Relation zur Latenz des Subgraphen vor der Ersetzung. Nur das achte Ergebnis wurde von der Vorwärtssuche nicht gefunden. Stattdessen rückt das letzte in der Tabelle aufgeführte Ergebnis in die besten zehn, welches bei der Vollständigen Suche auf Platz elf liegt. Es hat dieselbe Bewertung wie das nicht gefundene Ergebnis. Nur beim DFG LinearDriveControl hat das nachrückende Ergebnis eine schlechtere Bewertung (16 statt 18).

Wenn man die vier Beispielgraphen – mit Euler- und Heun-Integratoren – für die jeweiligen Maximalgrößen von drei bis neun betrachtet, ergibt dies insgesamt `2 * 4 * 7 * 10 = 560` Ergebnisse. Die Vorwärtssuche findet nur sechs dieser Ergebnisse nicht, also ungefähr 1%.

5.4. Einsparung durch Ersetzungen

Wie in Abschnitt 4.2.1 beschrieben, wäre das ideale Bewertungskriterium einer Makro-Op der Unterschied hinsichtlich der Ausführungszeit des DFGs auf dem CGRA vor der Ersetzung und danach. Dies ist allerdings aktuell nicht möglich, da der eingesetzte Scheduler noch keine Operationen mit mehr als zwei Eingängen beziehungsweise mehr als einem Ausgang unterstützt. Daher kann für die Bewertung nur die Einsparung in Takten durch eine Ersetzung und die Anzahl der nicht überlappenden Vorkommen im DFG betrachtet werden. Beispielhaft sind in Tabelle 5.3 die jeweils drei besten Ergebnisse für alle Beispielgraphen mit Heun-Integratoren und Maximalgröße neun angegeben. In der letzten Spalte ist angegeben, ob die Makro-Op ein Euler- oder Heun-Integrator ist, beziehungsweise ein Teil derselben.

Um auch die Unterschiede zwischen DFGs mit Euler- und Heun-Integratoren sowie zwischen den verschiedenen Maximalgrößen zu veranschaulichen, sind die drei

Gesamtbewertung	Einzel-bewertung (prozentual)	Von Vor-wärtssuche gefunden
21	1 (20%)	Ja
20	10 (23%)	Ja
20	10 (40%)	Ja
20	10 (31%)	Ja
20	10 (20%)	Ja
20	10 (40%)	Ja
20	10 (29%)	Ja
18	9 (32%)	Nein
18	9 (31%)	Ja
18	9 (24%)	Ja
18	9 (24%)	Ja

Tabelle 5.2.: Vergleich der Ergebnisse für DoublePendulum mit Heun-Integratoren bei Maximalgröße acht.

besten Ergebnisse des DoublePendulum-Graphen für beide Integratoren und Maximalgrößen drei, sechs und neun in Tabelle 5.4 gegeben.

Graph	Gesamt-bewertung	Einzel-bewertung (prozentual)	Anzahl Operationen	Integrator
TwoMassesBouncing	24	6 (32%)	6	
	20	5 (28%)	5	
	20	10 (36%)	9	
Verlade	25	1 (20%)	2	Heun (Teil)
	24	12 (33%)	9	
	24	12 (33%)	9	
DoublePendulum	22	11 (35%)	9	
	22	11 (33%)	9	
	22	11 (33%)	9	
LinearDriveControl	55	1 (20%)	2	Heun (Teil)
	32	8 (38%)	9	
	30	10 (38%)	9	

Tabelle 5.3.: Beste Ergebnisse mit Heun-Integratoren und Maximalgröße neun.

Maximal-größe	Integrator	Gesamt-bewertung	Einzel-bewertung (prozentual)	Anzahl Operationen	Integrator
3	Euler	14	2 (7%)	3	
		14	2 (8%)	2	
		13	1 (20%)	2	Euler
	Heun	21	1 (20%)	2	Heun (Teil)
		18	1 (20%)	2	
		16	4 (33%)	3	
6	Euler	14	2 (7%)	3	
		14	2 (8%)	2	
		13	1 (20%)	2	Euler
	Heun	21	1 (20%)	2	Heun (Teil)
		18	1 (20%)	2	
		16	8 (35%)	6	
9	Euler	14	2 (7%)	3	
		14	2 (8%)	2	
		13	1 (20%)	2	Euler
	Heun	22	11 (35%)	9	
		22	11 (33%)	9	
		22	11 (33%)	9	

Tabelle 5.4.: Beste Ergebnisse DoublePendulum.

6. Diskussion

In diesem Kapitel werden die in Kapitel 5 präsentierten Ergebnisse diskutiert und interpretiert. Zuerst werden dabei die verschiedenen Analysen anhand ihrer Laufzeit, ihres Speicherverbrauchs und ihrer Ergebnisse verglichen. Im Anschluss wird versucht, das mögliche Einsparungspotenzial der gefundenen Ersetzungsmöglichkeiten einzuschätzen.

Wie in Abschnitt 5.3 beschrieben, liefert die Vollständige Suche mit und ohne zusätzliche Abbruchbedingung dieselben Ergebnisse. Dies entspricht der Erwartung, da die Abbruchbedingung nur bei bereits betrachteten Makro-Ops aktiv wird. Sie verkleinert also nicht den Suchraum, sondern verhindert nur, dass Suchzweige mehrfach analysiert werden. Wie in Abbildung 5.3 dargestellt benötigt die Variante mit Abbruchbedingung beim TwoMassesBouncing-DFG mehr Arbeitsspeicher, was an der zusätzlichen Liste mit bereits besuchten Makro-Ops liegt. Bei den größeren Graphen ist dies dann umgekehrt: In vielen Fällen spart die Abbruchbedingung sogar Speicher. Das ist damit zu erklären, dass in diesen Fällen das frühere Abbrechen so viel Speicher einspart, dass es auch den zusätzlichen Verbrauch durch die Liste ausgleicht oder sogar übertrifft. Hinsichtlich der Laufzeit ist die Version mit Abbruchbedingung in allen Fällen schneller, meist sogar deutlich. Dies ist gut in Abbildung 5.1 und Abbildung 5.2 zu erkennen. Dabei ist zu beachten, dass die Ersparnis deutlich größer ist, als es auf den ersten Blick wirkt. Die Verwendung der zusätzlichen Abbruchbedingung ist also immer sinnvoll und der leicht erhöhte Speicherverbrauch bei kleinen Graphen im Gegensatz zur Laufzeiteinsparung vernachlässigbar.

Die beiden Varianten der Vorwärtssuche liefern bei den verwendeten Beispielgraphen ebenfalls immer gleiche Ergebnisse. Wie in Abschnitt 4.2.5 beschrieben sollte dies meistens so sein. Es sind zwar theoretische Beispiele denkbar, bei denen die zusätzliche Abbruchbedingung sinnvolle Makro-Ops auslässt, allerdings sind diese in realen DFGs sehr unwahrscheinlich. Hinsichtlich des Speicherverbrauchs unterscheiden sich die beiden Versionen erwartungsgemäß nur in vernachlässigbarem Ausmaß. Auch die Laufzeit der beiden Varianten ist meist vergleichbar. Nur bei TwoMasses-Bouncing mit Heun-Integratoren und bei DoublePendulum spart die Abbruchbedingung Zeit. Ob die in einigen Fällen reduzierte Laufzeit die – theoretisch vorhandene – Möglichkeit, sinnvolle Makro-Ops auszulassen, rechtfertigt, hängt vom konkreten Anwendungsfall ab.

Beim Vergleich zwischen Vorwärtssuche und Vollständiger Suche fällt auf, dass erstere einige mögliche Makro-Ops nicht findet (siehe Abschnitt 5.3). Die Anzahl und Relevanz der ausgelassenen Ersetzungen sollte aber annähernd immer vernachlässigbar sein. Immerhin wurden bei den Beispielgraphen die besten fünf Ergebnisse immer auch von der Vorwärtssuche gefunden, bei den besten zehn Ergebnissen fehlen lediglich 1%. Im Gegensatz dazu benötigt die Vorwärtssuche, wie in Abbildung 5.3 dargestellt, in allen Fällen weniger Arbeitsspeicher, in den meisten Fällen sogar deutlich. Noch eindeutiger ist der Unterschied bei den Laufzeiten: Hier ist die Vollständige Suche fast immer um mindestens Faktor zehn langsamer (siehe Abbildung 5.1). Ausnahme ist nur der DoublePendulum-DFG, bei dem der Faktor aber auch mindestens 4 beträgt. Die Verwendung der Vorwärtssuche sollte sich also in fast allen Fällen lohnen, außer die Vollständigkeit der Ergebnisse ist unbedingt notwendig.

Abgesehen davon fällt noch auf, dass der LinearDriveControl-DFG im Vergleich zu den anderen Graphen für seine Größe relativ schnell analysiert wird. Auch der Speicherverbrauch ist niedriger, als bei seiner Knotenzahl anzunehmen wäre. Dies lässt sich damit erklären, dass neben der reinen Knotenzahl auch die Operationen der Knoten relevant sind: Von seinen 4405 Knoten (mit Heun-Integratoren) werden bei LinearDriveControl aktuell weniger als 25% für die Makro-Op-Ersetzung in Betracht gezogen. Die restlichen Knoten sind vor allem Vergleiche und Operationen ohne Ein- oder Ausgang. Im Gegensatz dazu werden beim DoublePendulum mit Heun-Integratoren von seinen 456 Knoten über 50% in Betracht gezogen. Die absolute Anzahl der relevanten Knoten ist zwar bei LinearDriveControl trotzdem deutlich größer, allerdings gibt es deutlich weniger zusammenhängende Subgraphen aus in Frage kommenden Knoten.

Das Einsparunspotenzial der gefundenen Ersetzungsmöglichkeiten ist in Abschnitt 5.4 sowie in Tabelle 5.3 und Tabelle 5.4 beschrieben. Die besten drei Ergebnisse sind hierbei oft Makro-Ops, die eine vergleichsweise geringe Einzelbewertung aufweisen. Die Einsparung bei der Ersetzung eines Vorkommens beträgt also nur wenige Takte, sie treten allerdings häufig im DFG auf. Einzelne Ausnahmen sind Makro-Ops mit wenigen Vorkommen aber großer Einzelbewertung. Vor allem bei den kleineren Maximalgrößen enthalten die besten drei Ergebnisse oft eine Makro-Op, welche den Integrator – beziehungsweise bei Heun-Integratoren einen Teil des Integrators – ersetzt. Dieser Teil des Heun-Integrators ist die Kombination aus Multiplikation und Addition, also exakt der Euler-Integrator (siehe Abschnitt 2.3). Das hat den Vorteil, dass diese Makro-Ops auch in anderen DFGs auftauchen, sogar unabhängig vom verwendeten Integrator. Es kann dann derselbe CGRA – mit um diese Makro-Op erweitertem Instruktionssatz – für alle Graphen eingesetzt werden. Bei größeren Maximalgrößen tauchen diese ebenfalls auf, oft allerdings erst in den besten fünf oder

zehn Ergebnissen.

Neben der garantierten zeitlichen Einsparung durch eine Ersetzung ist noch Folgendes zu beachten: Wie in Abschnitt 4.2.1 beschrieben, basiert die Bewertung auf einer Worst-Case-Abschätzung für die Latenz des Subgraphen vor der Ersetzung. Die ermittelte Latenz wird nur erreicht, wenn alle prinzipiell gleichzeitig ausführbaren Operationen auch gleichzeitig auf verschiedenen PEs geschedult wurden. Im Gegensatz dazu hat die Makro-Op dann allerdings den Vorteil, dass sie nur eine PE belegt und die frei gewordenen PEs daher vom Scheduler anderweitig verwendet werden können. Wenn die Operationen nicht parallel geschedult werden, liegt die tatsächliche Einsparung höher als die vorberechnete.

7. Zusammenfassung & Ausblick

In dieser Arbeit wurde die Verwendung von Makro-Ops im Rahmen des UltraSynth-Projektes vorgestellt und diskutiert. Neben der eigentlichen Ersetzung eines Subgraphen durch eine Makro-Op wurden dabei vor allem zwei Analysen, eine erschöpfende Suche und eine Heuristik entwickelt. Diese sollen den Entwickler bei der Auswahl geeigneter Ersetzungsmöglichkeiten unterstützen. Die Heuristik erfordert dabei eine deutlich reduzierte Laufzeit bei kaum reduzierten Ergebnissen. Die wenigen fehlenden Ergebnisse sollten in der Regel vernachlässigbar sein. Die gefundenen Ersetzungen reduzieren die Latenz der Makro-Op durchschnittlich um circa 20% im Vergleich zu der Latenz vor der Ersetzung. Viele der Ersetzungsmöglichkeiten tauchen mehrmals in einem DFG auf, die Implementierung der Makro-Op kann also wiederverwendet werden. Auch die (teilweise) Ersetzung der in jedem DFG vorkommenden Integratoren ist möglich. Dies hat den Vorteil, dass die Makro-Op-Implementierung in allen DFGs verwendet werden kann.

Die aktuelle Implementierung unterliegt einigen Einschränkungen. Durch deren Beseitigung lassen sich die Ergebnisse noch weiter verbessern: Beispielsweise sind die für die Teilknoten unterstützten Operationen eingeschränkt (siehe Abschnitt 4.1.1). Ausgeschlossen werden aktuell Knoten ohne Ein- oder Ausgänge (zum Beispiel Laden und Speichern), Vergleiche und Nicht-Operations-Knoten (beispielsweise Konstanten). Alle diese Knotentypen könnten zumindest unter bestimmten Bedingungen auch in Makro-Ops erlaubt werden. Vor allem die Knoten ohne Ein- oder Ausgänge sollten einen deutlich positiven Effekt haben, da sie nur wenige der ohnehin knappen IO-Ports belegen.

Eine andere Einschränkung der Implementierung liegt im Generieren der Verilog-Implementierung (siehe Abschnitt 4.1.3): Dies funktioniert aktuell nicht mit allen Operationen, da nur die komplette Implementierung der Teiloperationen vorliegt, aber keine Informationen über deren Aufbau. Zum Beispiel müssen Schieberegister an den Ein- beziehungsweise Ausgängen entfernt werden, um die in der Laufzeitanalyse bestimmten Latenzen einhalten zu können.

Abgesehen von den Einschränkungen gibt es weitere Aspekte bei der Makro-Op-Ersetzung, die verbessert werden können: Das aktuelle Konzept der Überlappung von Teiloperationen deckt nicht alle möglichen Fälle ab. Voraussetzung ist aktuell, dass Operationen, deren Bitbreite größer als die Datenpfadbreite des DFGs ist, mit den weniger

signifikanten Teilworten ihrer Eingänge beginnen können. Das Übertragen der Daten in die ALU beginnt nämlich bei den am wenigsten signifikanten Teilworten. Bei manchen Operationen ist allerdings nur eine Überlappung möglich, wenn das Einlesen mit den signifikantesten Teilworten beginnt. Außerdem ist die Überlappung aktuell nur möglich, wenn die Operationen pro Takt genau Daten der Datenpfadbreite produzieren beziehungsweise konsumieren. Eine Überlappung mit einer Operation, die beispielsweise nur jeden zweiten Takt Daten produziert, ist aber prinzipiell auch denkbar. Mit genaueren Angaben, ob die Operation erst die signifikanten oder weniger signifikanten Teilworte benötigt und wie viele Takte zwischen den produzierten beziehungsweise benötigten Teilworten liegen, sind hier noch zusätzliche Einsparungen möglich. Für den ersten Fall ist zusätzlich eine Möglichkeit notwendig, die Reihenfolge des Einlesens der Teilworte zu konfigurieren. Ein weiterer Punkt ist das Beachten der Kommutativität der Eingänge bei vielen Operationen. Die Einsparung durch eine einzelne Ersetzung ändert sich dadurch zwar nicht, allerdings können potentiell mehr Subgraphen einer einzelnen Makro-Op-Implementierung zugeordnet werden. Es verbessert sich also nicht die Einzel-, aber die Gesamtbewertung der Makro-Op.

Außerdem ist auch eine weitere Verbesserung der Analysen möglich: Beide Analysen haben unter bestimmten Bedingungen Probleme bei Makro-Ops mit vielen Vorkommen (siehe Abschnitt 4.2.3). Wenn der DFG solche Makro-Ops enthält, steigt die Laufzeit stark an, da die Bestimmung der Anzahl an disjunkten Vorkommen sehr aufwendig ist. Da diese Makro-Ops potentiell große Einsparungen versprechen, ist eine Verbesserung der Algorithmen diesbezüglich wichtig. Vor allem bei der implementierten Heuristik ist das weitere Minimieren der Laufzeiten und des Speicherverbrauchs kombiniert mit der Verbesserung der Qualität der gefundenen Ergebnisse möglich. Dabei sind zum Beispiel, wie in Kapitel 3 beschrieben, die Ideen aus [11] denkbar. Dort wird ein Überblick über verschiedene Ansätze zur Lösung des Instruction-Set-Extension Problems gegeben. Diese können größtenteils auf die Makro-Op-Optimierung übertragen werden. Auch der Algorithmus aus [10] ist prinzipiell denkbar, wenn dieser so abgeändert werden kann, dass er keine komplette Partitionierung des DFGs liefert, sondern einzelne Ersetzungsmöglichkeiten. Andererseits ist auch das Einbeziehen weiterer Bewertungskriterien bei beiden Analysen möglich (siehe auch Abschnitt 4.2.1). Sobald der Scheduler entsprechend erweitert wurde, ist auf jeden Fall die Einbeziehung des Ergebnisses vom Scheduling, also die Latenz eines kompletten DFG-Durchlaufs, sinnvoll. Auch eine Berücksichtigung der maximal möglichen Taktfrequenz des CGRAs beziehungsweise deren Veränderung durch die Makro-Ops ist unbedingt notwendig.

A. Laufzeit und Speicherverbrauch der Analysen

Laufzeit

TwoMassesBouncing

	3	4	5	6	7	8	9
Vollständige Suche	0.01	0.05	0.2	0.8	4.6	24	153
Vollständige Suche mit Abbruch	0.01	0.03	0.08	0.2	0.5	1.0	2.3
Vorwärtssuche	0.01	0.01	0.01	0.02	0.04	0.08	0.2
Vorwärtssuche mit Abbruch	0.01	0.01	0.01	0.02	0.04	0.07	0.1

Tabelle A.1.: Laufzeiten TwoMassesBouncing mit Euler-Integratoren in Sekunden

	3	4	5	6	7	8	9
Vollständige Suche	0.02	0.08	0.3	2.0	16	119	1203
Vollständige Suche mit Abbruch	0.02	0.05	0.1	0.3	0.9	2.0	5.3
Vorwärtssuche	0.01	0.01	0.02	0.04	0.2	0.6	3.0
Vorwärtssuche mit Abbruch	0.01	0.01	0.02	0.03	0.05	0.1	0.3

Tabelle A.2.: Laufzeiten TwoMassesBouncing mit Heun-Integratoren in Sekunden

Verlade

	3	4	5	6	7	8	9
Vollständige Suche	0.01	0.03	0.08	0.2	0.4	0.8	1.9
Vollständige Suche mit Abbruch	0.01	0.03	0.07	0.2	0.4	0.7	1.6
Vorwärtssuche	0.01	0.01	0.01	0.02	0.02	0.02	0.03
Vorwärtssuche mit Abbruch	0.01	0.01	0.01	0.02	0.02	0.02	0.03

Tabelle A.3.: Laufzeiten Verlade mit Euler-Integratoren in Sekunden

	3	4	5	6	7	8	9
Vollständige Suche	0.04	0.09	0.2	0.6	1.7	4.3	13
Vollständige Suche mit Abbruch	0.03	0.08	0.2	0.5	1.2	2.3	5.7
Vorwärtssuche	0.01	0.02	0.03	0.04	0.07	0.1	0.2
Vorwärtssuche mit Abbruch	0.01	0.02	0.03	0.04	0.07	0.1	0.2

Tabelle A.4.: Laufzeiten Verlade mit Heun-Integratoren in Sekunden

DoublePendulum

	3	4	5	6	7	8	9
Vollständige Suche	0.04	0.2	0.8	4.6	25	130	1074
Vollständige Suche mit Abbruch	0.03	0.1	0.4	1.5	5.8	17	57
Vorwärtssuche	0.01	0.03	0.07	0.2	1.0	4.8	26
Vorwärtssuche mit Abbruch	0.01	0.02	0.06	0.2	0.7	2.9	13

Tabelle A.5.: Laufzeiten DoublePendulum mit Euler-Integratoren in Sekunden

	3	4	5	6	7	8	9
Vollständige Suche	0.06	0.2	0.9	4.3	25	139	1100
Vollständige Suche mit Abbruch	0.05	0.2	0.5	1.8	6.5	18	61
Vorwärtssuche	0.01	0.03	0.08	0.3	1.0	4.9	26
Vorwärtssuche mit Abbruch	0.01	0.03	0.08	0.2	0.7	3.0	14

Tabelle A.6.: Laufzeiten DoublePendulum mit Heun-Integratoren in Sekunden

LinearDriveControl

	3	4	5	6	7	8	9
Vollständige Suche	0.07	0.1	0.3	0.7	2.5	9.5	38
Vollständige Suche mit Abbruch	0.06	0.1	0.2	0.4	0.8	1.7	3.3
Vorwärtssuche	0.02	0.03	0.04	0.05	0.07	0.09	0.1
Vorwärtssuche mit Abbruch	0.03	0.03	0.04	0.05	0.06	0.09	0.1

Tabelle A.7.: Laufzeiten LinearDriveControl mit Euler-Integratoren in Sekunden

	3	4	5	6	7	8	9
Vollständige Suche	0.2	0.3	0.7	1.9	5.7	21	91
Vollständige Suche mit Abbruch	0.1	0.2	0.5	0.9	2.0	4.2	8.4
Vorwärtssuche	0.05	0.07	0.1	0.2	0.2	0.3	0.4
Vorwärtssuche mit Abbruch	0.05	0.07	0.1	0.1	0.2	0.3	0.4

Tabelle A.8.: Laufzeiten LinearDriveControl mit Heun-Integratoren in Sekunden

Speicherverbrauch

TwoMassesBouncing

	3	4	5	6	7	8	9
Vollständige Suche	0.1	0.2	0.7	0.8	1.0	1.5	2.0
Vollständige Suche mit Abbruch	0.1	0.1	0.2	0.3	0.9	1.0	2.5
Vorwärtssuche	0.1	0.1	0.1	0.1	0.1	0.2	0.4
Vorwärtssuche mit Abbruch	0.1	0.1	0.1	0.1	0.1	0.2	0.4

Tabelle A.9.: Speicherverbrauch TwoMassesBouncing mit Euler-Integratoren in Gigabyte

	3	4	5	6	7	8	9
Vollständige Suche	0.1	0.3	0.8	0.9	1.0	1.5	2.0
Vollständige Suche mit Abbruch	0.1	0.2	0.3	0.5	1.0	1.7	2.3
Vorwärtssuche	0.1	0.1	0.1	0.1	0.2	0.2	0.3
Vorwärtssuche mit Abbruch	0.1	0.1	0.1	0.1	0.1	0.2	0.2

Tabelle A.10.: Speicherverbrauch TwoMassesBouncing mit Heun-Integratoren in Gigabyte

Verlade

	3	4	5	6	7	8	9
Vollständige Suche	0.1	0.1	0.3	0.7	0.9	3.0	4.0
Vollständige Suche mit Abbruch	0.1	0.1	0.3	0.3	0.6	1.0	2.4
Vorwärtssuche	0.1	0.1	0.1	0.1	0.1	0.1	0.1
Vorwärtssuche mit Abbruch	0.1	0.1	0.1	0.1	0.1	0.1	0.1

Tabelle A.11.: Speicherverbrauch Verlade mit Euler-Integratoren in Gigabyte

	3	4	5	6	7	8	9
Vollständige Suche	0.1	0.3	0.9	2.3	2.6	8.0	11.0
Vollständige Suche mit Abbruch	0.1	0.3	0.7	0.9	1.5	4.3	5.6
Vorwärtssuche	0.1	0.1	0.1	0.1	0.2	0.3	0.5
Vorwärtssuche mit Abbruch	0.1	0.1	0.1	0.1	0.2	0.3	0.5

Tabelle A.12.: Speicherverbrauch Verlade mit Heun-Integratoren in Gigabyte

DoublePendulum

	3	4	5	6	7	8	9
Vollständige Suche	0.2	0.6	2.7	7.6	10.4	11.1	12.4
Vollständige Suche mit Abbruch	0.1	0.4	1.4	3.2	5.5	10.9	11.7
Vorwärtssuche	0.07	0.1	0.3	0.7	2.8	10.6	10.7
Vorwärtssuche mit Abbruch	0.1	0.1	0.2	0.6	1.8	6.7	10.9

Tabelle A.13.: Speicherverbrauch DoublePendulum mit Euler-Integratoren in Gigabyte

	3	4	5	6	7	8	9
Vollständige Suche	0.2	0.8	3.4	10.7	10.7	11.1	12.5
Vollständige Suche mit Abbruch	0.2	0.6	2.1	6.8	10.6	10.9	11.8
Vorwärtssuche	0.2	0.2	0.4	0.9	3.0	10.6	10.8
Vorwärtssuche mit Abbruch	0.2	0.2	0.4	0.6	1.9	6.7	11.0

Tabelle A.14.: Speicherverbrauch DoublePendulum mit Heun-Integratoren in Gigabyte

LinearDriveControl

	3	4	5	6	7	8	9
Vollständige Suche	0.3	0.5	1.0	2.5	8.1	9.1	11.0
Vollständige Suche mit Abbruch	0.2	0.4	0.6	1.2	2.5	4.3	6.9
Vorwärtssuche	0.2	0.2	0.2	0.2	0.3	0.3	0.3
Vorwärtssuche mit Abbruch	0.2	0.2	0.2	0.2	0.2	0.3	0.3

Tabelle A.15.: Speicherverbrauch LinearDriveControl mit Euler-Integratoren in Gigabyte

	3	4	5	6	7	8	9
Vollständige Suche	0.6	1.0	2.3	5.8	10.9	10.9	11.0
Vollständige Suche mit Abbruch	0.6	0.8	1.5	3.0	6.3	11.0	11.0
Vorwärtssuche	0.4	0.4	0.4	0.4	0.6	0.6	0.8
Vorwärtssuche mit Abbruch	0.4	0.4	0.4	0.4	0.6	0.6	0.8

Tabelle A.16.: Speicherverbrauch LinearDriveControl mit Heun-Integratoren in Gigabyte

B. Ergebnisse der Analysen

TwoMassesBouncing

Euler

Vollständige Suche

GB	EB (%)	Enthaltene Operationen
12	1 (20%)	MUL, ADD
8	4 (25%)	MUL, DIV, DIV
7	1 (20%)	ADD, MUL
6	3 (27%)	DIV, MUL, ADD
4	2 (15%)	DIV, ADD, DIV
4	2 (20%)	DIV, MUL
4	2 (33%)	SUB, ADD, MUL
4	1 (17%)	SUB, ADD, MUL
4	2 (22%)	MUL, MUL, ADD
4	2 (33%)	MUL, NEG, ADD

Tabelle B.1.: Ergebnisse der Vollständige Suche mit Maximalgröße 3

GB	EB (%)	Enthaltene Operationen
14	7 (32%)	DIV, ADD, DIV, ADD, MUL, MUL
12	6 (32%)	DIV, ADD, ADD, MUL, ADD, DIV
12	6 (32%)	NEG, DIV, DIV, ADD, MUL, ADD
12	1 (20%)	MUL, ADD
10	5 (29%)	DIV, MUL, ADD, DIV
10	5 (42%)	SUB, ADD, ADD, MUL, ADD, MUL
10	5 (28%)	DIV, ADD, MUL, ADD, DIV
10	5 (42%)	ADD, SUB, ADD, SUB, MUL, MUL
10	5 (28%)	DIV, ADD, DIV, ADD, MUL
10	5 (28%)	MUL, ADD, ADD, DIV, DIV

Tabelle B.2.: Ergebnisse der Vollständige Suche mit Maximalgröße 6

GB	EB (%)	Enthaltene Operationen
18	9 (38%)	NEG, MUL, ADD, DIV, MUL, ADD, DIV, ADD
16	8 (35%)	DIV, MUL, MUL, DIV, ADD, ADD, ADD
16	8 (35%)	NEG, MUL, DIV, MUL, ADD, DIV, ADD
14	7 (32%)	DIV, ADD, DIV, ADD, MUL, MUL
12	6 (32%)	DIV, ADD, ADD, MUL, ADD, DIV
12	6 (46%)	SUB, ADD, MUL, ADD, ADD, SUB, MUL
12	6 (32%)	NEG, MUL, ADD, DIV, DIV, ADD
12	1 (20%)	MUL, ADD
10	5 (29%)	DIV, MUL, ADD, DIV
10	5 (42%)	SUB, ADD, MUL, ADD, ADD, MUL

Tabelle B.3.: Ergebnisse der Vollständige Suche mit Maximalgröße 9

Vorwärtssuche

GB	EB (%)	Enthaltene Operationen
12	1 (20%)	MUL, ADD
8	4 (25%)	MUL, DIV, DIV
7	1 (20%)	ADD, MUL
6	3 (27%)	DIV, MUL, ADD
4	2 (15%)	DIV, ADD, DIV
4	2 (20%)	DIV, MUL
4	2 (33%)	SUB, ADD, MUL
4	1 (17%)	SUB, ADD, MUL
4	2 (22%)	MUL, MUL, ADD
4	2 (33%)	MUL, NEG, ADD

Tabelle B.4.: Ergebnisse der Vorwärtssuche mit Maximalgröße 3

GB	EB (%)	Enthaltene Operationen
14	7 (32%)	DIV, ADD, DIV, ADD, MUL, MUL
12	6 (32%)	DIV, ADD, ADD, MUL, ADD, DIV
12	6 (32%)	NEG, DIV, DIV, ADD, MUL, ADD
12	1 (20%)	MUL, ADD
10	5 (29%)	DIV, MUL, ADD, DIV
10	5 (42%)	SUB, ADD, ADD, MUL, ADD, MUL
10	5 (28%)	DIV, ADD, MUL, ADD, DIV
10	5 (42%)	ADD, SUB, ADD, SUB, MUL, MUL
10	5 (28%)	DIV, ADD, DIV, ADD, MUL
10	5 (28%)	MUL, ADD, ADD, DIV, DIV

Tabelle B.5.: Ergebnisse der Vorwärtssuche mit Maximalgröße 6

GB	EB (%)	Enthaltene Operationen
18	9 (38%)	NEG, MUL, ADD, DIV, MUL, ADD, DIV, ADD
16	8 (35%)	DIV, MUL, MUL, DIV, ADD, ADD, ADD
16	8 (35%)	NEG, MUL, DIV, MUL, ADD, DIV, ADD
14	7 (32%)	DIV, ADD, DIV, ADD, MUL, MUL
12	6 (32%)	DIV, ADD, ADD, MUL, ADD, DIV
12	6 (46%)	SUB, ADD, MUL, ADD, ADD, SUB, MUL
12	6 (32%)	NEG, MUL, ADD, DIV, DIV, ADD
12	1 (20%)	MUL, ADD
10	5 (29%)	DIV, MUL, ADD, DIV
10	5 (42%)	SUB, ADD, MUL, ADD, ADD, MUL

Tabelle B.6.: Ergebnisse der Vorwärtssuche mit Maximalgröße 9

Heun

Vollständige Suche

GB	EB (%)	Enthaltene Operationen
20	1 (20%)	MUL, ADD
13	1 (20%)	ADD, MUL
10	2 (33%)	ADD, MUL, ADD
8	2 (15%)	DIV, ADD, DIV
8	4 (25%)	MUL, DIV, DIV
8	1 (17%)	SUB, ADD, MUL
8	2 (22%)	MUL, MUL, ADD
8	2 (33%)	MUL, NEG, ADD
8	2 (17%)	DIV, DIV
6	3 (23%)	DIV, DIV, ADD

Tabelle B.7.: Ergebnisse der Vollständige Suche mit Maximalgröße 3

GB	EB (%)	Enthaltene Operationen
24	6 (32%)	NEG, DIV, DIV, ADD, MUL, ADD
20	5 (28%)	DIV, ADD, DIV, ADD, MUL
20	1 (20%)	MUL, ADD
16	4 (33%)	ADD, MUL, SUB, SUB, ADD, MUL
16	4 (36%)	ADD, ADD, MUL, SUB, MUL
16	4 (31%)	NEG, ADD, MUL, DIV, ADD
14	7 (32%)	DIV, MUL, MUL, ADD, DIV, SUB
14	7 (32%)	MUL, ADD, ADD, DIV, MUL, DIV
14	7 (32%)	DIV, ADD, DIV, ADD, MUL, MUL
13	1 (20%)	ADD, MUL

Tabelle B.8.: Ergebnisse der Vollständige Suche mit Maximalgröße 6

GB	EB (%)	Enthaltene Operationen
24	6 (32%)	NEG, MUL, ADD, DIV, DIV, ADD
20	5 (28%)	DIV, ADD, DIV, ADD, MUL
20	10 (36%)	DIV, SUB, MUL, MUL, DIV, ADD, ADD, MUL, ADD
20	10 (36%)	DIV, MUL, MUL, ADD, DIV, MUL, ADD, ADD, SUB
20	1 (20%)	MUL, ADD
20	10 (36%)	ADD, MUL, ADD, DIV, MUL, MUL, DIV, ADD, ADD
18	9 (36%)	NEG, MUL, DIV, ADD, SUB, MUL, ADD, DIV, ADD
18	9 (36%)	ADD, SUB, DIV, ADD, ADD, MUL, ADD, DIV, MUL
18	9 (33%)	DIV, MUL, MUL, ADD, DIV, MUL, ADD, SUB
18	9 (33%)	ADD, DIV, MUL, MUL, ADD, DIV, SUB, MUL, ADD

Tabelle B.9.: Ergebnisse der Vollständige Suche mit Maximalgröße 9

Vorwärtssuche

GB	EB (%)	Enthaltene Operationen
20	1 (20%)	MUL, ADD
13	1 (20%)	ADD, MUL
10	2 (33%)	ADD, MUL, ADD
8	2 (15%)	DIV, ADD, DIV
8	4 (25%)	MUL, DIV, DIV
8	1 (17%)	SUB, ADD, MUL
8	2 (22%)	MUL, MUL, ADD
8	2 (33%)	MUL, NEG, ADD
8	2 (17%)	DIV, DIV
6	3 (23%)	DIV, DIV, ADD

Tabelle B.10.: Ergebnisse der Vorwärtssuche mit Maximalgröße 3

GB	EB (%)	Enthaltene Operationen
24	6 (32%)	NEG, DIV, DIV, ADD, MUL, ADD
20	5 (28%)	DIV, ADD, DIV, ADD, MUL
20	1 (20%)	MUL, ADD
16	4 (33%)	ADD, MUL, SUB, SUB, ADD, MUL
16	4 (36%)	ADD, ADD, MUL, SUB, MUL
16	4 (31%)	NEG, ADD, MUL, DIV, ADD
14	7 (32%)	DIV, MUL, MUL, ADD, DIV, SUB
14	7 (32%)	MUL, ADD, ADD, DIV, MUL, DIV
14	7 (32%)	DIV, ADD, DIV, ADD, MUL, MUL
13	1 (20%)	ADD, MUL

Tabelle B.11.: Ergebnisse der Vorwärtssuche mit Maximalgröße 6

GB	EB (%)	Enthaltene Operationen
24	6 (32%)	NEG, MUL, ADD, DIV, DIV, ADD
20	5 (28%)	DIV, ADD, DIV, ADD, MUL
20	10 (36%)	DIV, SUB, MUL, MUL, DIV, ADD, ADD, MUL, ADD
20	10 (36%)	DIV, MUL, MUL, ADD, DIV, MUL, ADD, ADD, SUB
20	1 (20%)	MUL, ADD
20	10 (36%)	ADD, MUL, ADD, DIV, MUL, MUL, DIV, ADD, ADD
18	9 (36%)	NEG, MUL, DIV, ADD, SUB, MUL, ADD, DIV, ADD
18	9 (36%)	ADD, SUB, DIV, ADD, ADD, MUL, ADD, DIV, MUL
18	9 (33%)	DIV, MUL, MUL, ADD, DIV, MUL, ADD, SUB
18	9 (33%)	ADD, DIV, MUL, MUL, ADD, DIV, SUB, MUL, ADD

Tabelle B.12.: Ergebnisse der Vorwärtssuche mit Maximalgröße 9

Verlade

Euler

Vollständige Suche

GB	EB (%)	Enthaltene Operationen
14	1 (20%)	ADD, MUL
8	1 (20%)	MUL, ADD
6	3 (27%)	DIV, MUL, ADD
6	3 (27%)	DIV, SUB, MUL
4	4 (25%)	MUL, DIV, DIV
4	2 (33%)	ADD, ADD, MUL
4	4 (13%)	MUL, COS, MUL
4	4 (25%)	MUL, DIV, DIV
4	2 (20%)	DIV, MUL
4	2 (17%)	DIV, DIV

Tabelle B.13.: Ergebnisse der Vollständige Suche mit Maximalgröße 3

GB	EB (%)	Enthaltene Operationen
14	1 (20%)	ADD, MUL
8	8 (14%)	MUL, SIN, MUL, DIV, ADD, POW
8	8 (30%)	DIV, SUB, MUL, MUL, DIV, DIV
8	8 (19%)	DIV, MUL, POW, DIV, SUB, MUL
8	8 (20%)	MUL, POW, ADD, MUL, MUL, DIV
8	8 (19%)	DIV, SUB, MUL, MUL, DIV, POW
8	8 (30%)	MUL, DIV, SUB, MUL, DIV, DIV
8	4 (24%)	SUB, MUL, DIV, DIV
8	1 (20%)	MUL, ADD
8	8 (35%)	MUL, MUL, MUL, ADD, MUL, DIV

Tabelle B.14.: Ergebnisse der Vollständige Suche mit Maximalgröße 6

GB	EB (%)	Enthaltene Operationen
14	1 (20%)	ADD, MUL
12	12 (33%)	MUL, DIV, DIV, MUL, MUL, MUL, ADD, SUB, DIV
12	12 (33%)	SUB, DIV, MUL, MUL, MUL, MUL, DIV, DIV, ADD
12	12 (14%)	SUB, ADD, DIV, SIN, POW, MUL, MUL, MUL, POW
12	12 (32%)	DIV, DIV, DIV, DIV, SUB, SUB, MUL, MUL, MUL
12	12 (33%)	DIV, MUL, DIV, MUL, MUL, MUL, ADD, SUB, DIV
12	12 (22%)	MUL, SUB, MUL, DIV, POW, SUB, DIV, MUL, DIV
11	11 (17%)	ADD, POW, SIN, DIV, MUL, NEG, SUB, MUL, MUL
11	11 (33%)	DIV, SUB, MUL, DIV, MUL, ADD, ADD, MUL, DIV
11	11 (22%)	ADD, MUL, SUB, POW, DIV, DIV, DIV, SUB, MUL

Tabelle B.15.: Ergebnisse der Vollständige Suche mit Maximalgröße 9

Vorwärtssuche

GB	EB (%)	Enthaltene Operationen
14	1 (20%)	ADD, MUL
8	1 (20%)	MUL, ADD
6	3 (27%)	DIV, MUL, ADD
6	3 (27%)	DIV, SUB, MUL
4	4 (25%)	MUL, DIV, DIV
4	2 (33%)	ADD, ADD, MUL
4	4 (13%)	MUL, COS, MUL
4	4 (25%)	MUL, DIV, DIV
4	2 (20%)	DIV, MUL
4	2 (17%)	DIV, DIV

Tabelle B.16.: Ergebnisse der Vorwärtssuche mit Maximalgröße 3

GB	EB (%)	Enthaltene Operationen
14	1 (20%)	ADD, MUL
8	8 (14%)	MUL, SIN, MUL, DIV, ADD, POW
8	8 (30%)	DIV, SUB, MUL, MUL, DIV, DIV
8	8 (19%)	DIV, MUL, POW, DIV, SUB, MUL
8	8 (20%)	MUL, POW, ADD, MUL, MUL, DIV
8	8 (19%)	DIV, SUB, MUL, MUL, DIV, POW
8	8 (30%)	MUL, DIV, SUB, MUL, DIV, DIV
8	4 (24%)	SUB, MUL, DIV, DIV
8	1 (20%)	MUL, ADD
8	8 (35%)	MUL, MUL, MUL, ADD, MUL, DIV

Tabelle B.17.: Ergebnisse der Vorwärtssuche mit Maximalgröße 6

GB	EB (%)	Enthaltene Operationen
14	1 (20%)	ADD, MUL
12	12 (33%)	MUL, DIV, DIV, MUL, MUL, MUL, ADD, SUB, DIV
12	12 (33%)	SUB, DIV, MUL, MUL, MUL, MUL, DIV, DIV, ADD
12	12 (14%)	SUB, ADD, DIV, SIN, POW, MUL, MUL, MUL, POW
12	12 (32%)	DIV, DIV, DIV, DIV, SUB, SUB, MUL, MUL, MUL
12	12 (33%)	DIV, MUL, DIV, MUL, MUL, MUL, ADD, SUB, DIV
12	12 (22%)	MUL, SUB, MUL, DIV, POW, SUB, DIV, MUL, DIV
11	11 (17%)	ADD, POW, SIN, DIV, MUL, NEG, SUB, MUL, MUL
11	11 (33%)	DIV, SUB, MUL, DIV, MUL, ADD, ADD, MUL, DIV
11	11 (22%)	ADD, MUL, SUB, POW, DIV, DIV, DIV, SUB, MUL

Tabelle B.18.: Ergebnisse der Vorwärtssuche mit Maximalgröße 9

Heun

Vollständige Suche

GB	EB (%)	Enthaltene Operationen
25	1 (20%)	ADD, MUL
16	1 (20%)	MUL, ADD
12	3 (27%)	DIV, SUB, MUL
10	2 (20%)	DIV, MUL
9	3 (27%)	DIV, MUL, ADD
9	1 (17%)	ADD, MUL, ADD
8	4 (13%)	MUL, COS, MUL
8	2 (33%)	ADD, ADD, MUL
8	4 (25%)	MUL, DIV, DIV
8	2 (17%)	DIV, DIV

Tabelle B.19.: Ergebnisse der Vollständige Suche mit Maximalgröße 3

GB	EB (%)	Enthaltene Operationen
25	1 (20%)	ADD, MUL
16	8 (14%)	MUL, MUL, SIN, DIV, ADD, POW
16	8 (20%)	MUL, POW, ADD, MUL, MUL, DIV
16	8 (19%)	DIV, SUB, MUL, MUL, DIV, POW
16	4 (24%)	SUB, MUL, DIV, DIV
16	1 (20%)	MUL, ADD
16	8 (35%)	MUL, MUL, MUL, ADD, MUL, DIV
16	8 (30%)	DIV, MUL, DIV, SUB, DIV, MUL
16	8 (18%)	DIV, DIV, MUL, SUB, DIV, POW
14	7 (18%)	ADD, DIV, POW, ADD, MUL, MUL

Tabelle B.20.: Ergebnisse der Vollständige Suche mit Maximalgröße 6

GB	EB (%)	Enthaltene Operationen
25	1 (20%)	ADD, MUL
24	12 (33%)	MUL, DIV, DIV, MUL, MUL, MUL, ADD, SUB, DIV
24	12 (33%)	SUB, DIV, MUL, MUL, MUL, DIV, DIV, ADD, MUL
24	12 (14%)	SUB, ADD, DIV, SIN, MUL, POW, MUL, MUL, POW
22	11 (17%)	ADD, DIV, MUL, MUL, COS, MUL, MUL, ADD, POW
22	11 (17%)	ADD, SIN, POW, DIV, MUL, NEG, SUB, MUL, MUL
22	11 (17%)	ADD, SIN, POW, MUL, MUL, SUB, MUL, MUL, DIV
22	11 (33%)	SUB, MUL, DIV, SUB, MUL, DIV, ADD, DIV, MUL
22	11 (17%)	ADD, SIN, POW, DIV, MUL, SUB, MUL, MUL, MUL
22	11 (33%)	MUL, ADD, DIV, MUL, MUL, DIV, SUB, DIV, ADD

Tabelle B.21.: Ergebnisse der Vollständige Suche mit Maximalgröße 9

Vorwärtssuche

GB	EB (%)	Enthaltene Operationen
25	1 (20%)	ADD, MUL
16	1 (20%)	MUL, ADD
12	3 (27%)	DIV, SUB, MUL
10	2 (20%)	DIV, MUL
9	3 (27%)	DIV, MUL, ADD
9	1 (17%)	ADD, MUL, ADD
8	4 (13%)	MUL, COS, MUL
8	2 (33%)	ADD, ADD, MUL
8	4 (25%)	MUL, DIV, DIV
8	2 (17%)	DIV, DIV

Tabelle B.22.: Ergebnisse der Vorwärtssuche mit Maximalgröße 3

GB	EB (%)	Enthaltene Operationen
25	1 (20%)	ADD, MUL
16	8 (14%)	MUL, MUL, SIN, DIV, ADD, POW
16	8 (20%)	MUL, POW, ADD, MUL, MUL, DIV
16	8 (19%)	DIV, SUB, MUL, MUL, DIV, POW
16	4 (24%)	SUB, MUL, DIV, DIV
16	1 (20%)	MUL, ADD
16	8 (35%)	MUL, MUL, MUL, ADD, MUL, DIV
16	8 (30%)	DIV, MUL, DIV, SUB, DIV, MUL
16	8 (18%)	DIV, DIV, MUL, SUB, DIV, POW
14	7 (18%)	ADD, DIV, POW, ADD, MUL, MUL

Tabelle B.23.: Ergebnisse der Vorwärtssuche mit Maximalgröße 6

GB	EB (%)	Enthaltene Operationen
25	1 (20%)	ADD, MUL
24	12 (33%)	MUL, DIV, DIV, MUL, MUL, MUL, ADD, SUB, DIV
24	12 (33%)	SUB, DIV, MUL, MUL, MUL, DIV, DIV, ADD, MUL
24	12 (14%)	SUB, ADD, DIV, SIN, MUL, POW, MUL, MUL, POW
22	11 (17%)	ADD, DIV, MUL, MUL, COS, MUL, MUL, ADD, POW
22	11 (17%)	ADD, SIN, POW, DIV, MUL, NEG, SUB, MUL, MUL
22	11 (33%)	SUB, MUL, DIV, SUB, MUL, DIV, ADD, DIV, MUL
22	11 (17%)	ADD, SIN, POW, DIV, MUL, SUB, MUL, MUL, MUL
22	11 (33%)	MUL, ADD, DIV, MUL, MUL, DIV, SUB, DIV, ADD
22	11 (17%)	ADD, SIN, POW, DIV, MUL, SUB, MUL, MUL

Tabelle B.24.: Ergebnisse der Vorwärtssuche mit Maximalgröße 9

DoublePendulum

Euler

Vollständige Suche

GB	EB (%)	Enthaltene Operationen
14	2 (7%)	MUL, POW, ADD
14	2 (8%)	POW, MUL
13	1 (20%)	ADD, MUL
12	2 (18%)	MUL, DIV, Compare
12	1 (20%)	MUL, ADD
8	4 (33%)	MUL, MUL, MUL
7	1 (14%)	DIV, Compare
6	3 (27%)	DIV, SUB, MUL
6	2 (7%)	POW, MUL, ADD
6	3 (33%)	MUL, MUL, ADD

Tabelle B.25.: Ergebnisse der Vollständige Suche mit Maximalgröße 3

GB	EB (%)	Enthaltene Operationen
14	2 (7%)	MUL, POW, ADD
14	2 (8%)	POW, MUL
13	1 (20%)	ADD, MUL
12	2 (18%)	MUL, DIV, Compare
12	4 (13%)	ADD, POW, MUL, MUL
12	1 (20%)	MUL, ADD
8	8 (21%)	MUL, MUL, MUL, COS, MUL, ADD
8	8 (19%)	DIV, SUB, MUL, MUL, DIV, POW
8	4 (24%)	SUB, MUL, DIV, DIV
8	4 (33%)	MUL, MUL, MUL

Tabelle B.26.: Ergebnisse der Vollständige Suche mit Maximalgröße 6

GB	EB (%)	Enthaltene Operationen
14	2 (7%)	ADD, MUL, POW
14	2 (8%)	MUL, POW
13	1 (20%)	ADD, MUL
12	4 (13%)	POW, ADD, MUL, MUL
12	12 (32%)	DIV, DIV, DIV, DIV, SUB, SUB, MUL, MUL, MUL
12	12 (24%)	ADD, MUL, MUL, MUL, ADD, DIV, DIV, SUB, POW
12	12 (24%)	ADD, MUL, MUL, MUL, DIV, DIV, SUB, SUB, POW
12	1 (20%)	MUL, ADD
12	12 (22%)	MUL, SUB, MUL, DIV, POW, SUB, DIV, MUL, DIV
12	2 (18%)	Compare, DIV, MUL

Tabelle B.27.: Ergebnisse der Vollständige Suche mit Maximalgröße 9

Vorwärtssuche

GB	EB (%)	Enthaltene Operationen
14	2 (7%)	MUL, POW, ADD
14	2 (8%)	POW, MUL
13	1 (20%)	ADD, MUL
12	2 (18%)	MUL, DIV, Compare
12	1 (20%)	MUL, ADD
8	4 (33%)	MUL, MUL, MUL
7	1 (14%)	DIV, Compare
6	3 (27%)	DIV, SUB, MUL
6	2 (7%)	POW, MUL, ADD
6	3 (33%)	MUL, MUL, ADD

Tabelle B.28.: Ergebnisse der Vorwärtssuche mit Maximalgröße 3

GB	EB (%)	Enthaltene Operationen
14	2 (7%)	MUL, POW, ADD
14	2 (8%)	POW, MUL
13	1 (20%)	ADD, MUL
12	2 (18%)	MUL, DIV, Compare
12	4 (13%)	ADD, POW, MUL, MUL
12	1 (20%)	MUL, ADD
8	8 (21%)	MUL, MUL, MUL, COS, MUL, ADD
8	8 (19%)	DIV, SUB, MUL, MUL, DIV, POW
8	4 (24%)	SUB, MUL, DIV, DIV
8	4 (33%)	MUL, MUL, MUL

Tabelle B.29.: Ergebnisse der Vorwärtssuche mit Maximalgröße 6

GB	EB (%)	Enthaltene Operationen
14	2 (7%)	ADD, MUL, POW
14	2 (8%)	MUL, POW
13	1 (20%)	ADD, MUL
12	4 (13%)	POW, ADD, MUL, MUL
12	12 (32%)	DIV, DIV, DIV, DIV, SUB, SUB, MUL, MUL, MUL
12	12 (24%)	ADD, MUL, MUL, MUL, ADD, DIV, DIV, SUB, POW
12	12 (24%)	ADD, MUL, MUL, MUL, DIV, DIV, SUB, SUB, POW
12	1 (20%)	MUL, ADD
12	12 (22%)	MUL, SUB, MUL, DIV, POW, SUB, DIV, MUL, DIV
12	2 (18%)	Compare, DIV, MUL

Tabelle B.30.: Ergebnisse der Vorwärtssuche mit Maximalgröße 9

Heun

Vollständige Suche

GB	EB (%)	Enthaltene Operationen
21	1 (20%)	ADD, MUL
18	1 (20%)	MUL, ADD
16	4 (33%)	MUL, MUL, MUL
14	2 (7%)	MUL, POW, ADD
14	2 (8%)	POW, MUL
12	2 (18%)	MUL, DIV, Compare
12	3 (27%)	DIV, SUB, MUL
12	2 (25%)	MUL, MUL
8	4 (13%)	MUL, COS, MUL
8	4 (25%)	MUL, DIV, DIV

Tabelle B.31.: Ergebnisse der Vollständige Suche mit Maximalgröße 3

GB	EB (%)	Enthaltene Operationen
21	1 (20%)	ADD, MUL
18	1 (20%)	MUL, ADD
16	8 (35%)	MUL, MUL, MUL, ADD, MUL, DIV
16	8 (19%)	DIV, SUB, MUL, MUL, DIV, POW
16	4 (24%)	SUB, MUL, DIV, DIV
16	4 (33%)	MUL, MUL, MUL
16	8 (30%)	DIV, MUL, DIV, DIV, SUB, MUL
16	8 (18%)	DIV, DIV, MUL, SUB, DIV, POW
14	7 (19%)	MUL, ADD, SIN, MUL, ADD, MUL
14	7 (18%)	DIV, DIV, MUL, SUB, POW

Tabelle B.32.: Ergebnisse der Vollständige Suche mit Maximalgröße 6

GB	EB (%)	Enthaltene Operationen
22	11 (35%)	DIV, MUL, DIV, MUL, SUB, ADD, MUL, MUL, ADD
22	11 (33%)	MUL, DIV, ADD, MUL, MUL, ADD, DIV, DIV, SUB
22	11 (33%)	MUL, DIV, MUL, DIV, MUL, SUB, ADD, DIV, ADD
22	11 (33%)	SUB, MUL, DIV, SUB, MUL, ADD, DIV, DIV, MUL
22	11 (25%)	MUL, MUL, ADD, ADD, ADD, DIV, ADD, MUL, POW
22	11 (25%)	DIV, ADD, SUB, MUL, ADD, MUL, ADD, MUL, POW
22	11 (35%)	DIV, MUL, MUL, DIV, SUB, MUL, MUL, ADD, ADD
22	11 (33%)	MUL, ADD, DIV, MUL, MUL, DIV, SUB, DIV, ADD
22	11 (33%)	DIV, SUB, MUL, DIV, MUL, ADD, ADD, DIV, MUL
21	1 (20%)	ADD, MUL

Tabelle B.33.: Ergebnisse der Vollständige Suche mit Maximalgröße 9

Vorwärtssuche

GB	EB (%)	Enthaltene Operationen
21	1 (20%)	ADD, MUL
18	1 (20%)	MUL, ADD
16	4 (33%)	MUL, MUL, MUL
14	2 (7%)	MUL, POW, ADD
14	2 (8%)	POW, MUL
12	2 (18%)	MUL, DIV, Compare
12	3 (27%)	DIV, SUB, MUL
12	2 (25%)	MUL, MUL
8	4 (13%)	MUL, COS, MUL
8	4 (25%)	MUL, DIV, DIV

Tabelle B.34.: Ergebnisse der Vorwärtssuche mit Maximalgröße 3

GB	EB (%)	Enthaltene Operationen
21	1 (20%)	ADD, MUL
18	1 (20%)	MUL, ADD
16	8 (35%)	MUL, MUL, MUL, ADD, MUL, DIV
16	8 (19%)	DIV, SUB, MUL, MUL, DIV, POW
16	4 (24%)	SUB, MUL, DIV, DIV
16	4 (33%)	MUL, MUL, MUL
16	8 (30%)	DIV, MUL, DIV, DIV, SUB, MUL
16	8 (18%)	DIV, DIV, MUL, SUB, DIV, POW
14	7 (19%)	MUL, ADD, SIN, MUL, ADD, MUL
14	7 (18%)	DIV, DIV, MUL, SUB, POW

Tabelle B.35.: Ergebnisse der Vorwärtssuche mit Maximalgröße 6

GB	EB (%)	Enthaltene Operationen
22	11 (35%)	DIV, MUL, DIV, MUL, SUB, ADD, MUL, MUL, ADD
22	11 (33%)	MUL, DIV, ADD, MUL, MUL, ADD, DIV, DIV, SUB
22	11 (33%)	MUL, DIV, MUL, DIV, MUL, SUB, ADD, DIV, ADD
22	11 (33%)	SUB, MUL, DIV, SUB, MUL, ADD, DIV, DIV, MUL
22	11 (25%)	MUL, MUL, ADD, ADD, ADD, DIV, ADD, MUL, POW
22	11 (25%)	DIV, ADD, SUB, MUL, ADD, MUL, ADD, MUL, POW
22	11 (35%)	DIV, MUL, MUL, DIV, SUB, MUL, MUL, ADD, ADD
22	11 (33%)	MUL, ADD, DIV, MUL, MUL, DIV, SUB, DIV, ADD
22	11 (33%)	DIV, SUB, MUL, DIV, MUL, ADD, ADD, DIV, MUL
21	1 (20%)	ADD, MUL

Tabelle B.36.: Ergebnisse der Vorwärtssuche mit Maximalgröße 9

LinearDriveControl

Euler

Vollständige Suche

GB	EB (%)	Enthaltene Operationen
30	2 (25%)	MUL, MUL
28	1 (20%)	ADD, MUL
26	2 (20%)	MUL, DIV
22	1 (33%)	AND, AND, AND
21	3 (27%)	MUL, DIV, ADD
18	2 (20%)	MUL, DIV
16	4 (29%)	MUL, MUL, DIV
12	3 (27%)	MUL, ADD, DIV
12	3 (33%)	MUL, MUL, ADD
12	3 (33%)	MUL, ADD, MUL

Tabelle B.37.: Ergebnisse der Vollständige Suche mit Maximalgröße 3

GB	EB (%)	Enthaltene Operationen
30	2 (25%)	MUL, MUL
28	1 (20%)	ADD, MUL
26	2 (20%)	MUL, DIV
22	1 (33%)	AND, AND, AND
21	7 (35%)	ADD, MUL, MUL, ADD, MUL, DIV
21	3 (27%)	MUL, DIV, ADD
18	2 (20%)	MUL, DIV
18	6 (32%)	MUL, MUL, MUL, DIV, ADD
16	4 (29%)	MUL, MUL, DIV
15	5 (31%)	DIV, SUB, MUL, ADD, ADD, MUL

Tabelle B.38.: Ergebnisse der Vollständige Suche mit Maximalgröße 6

GB	EB (%)	Enthaltene Operationen
30	2 (25%)	MUL, MUL
28	1 (20%)	ADD, MUL
26	2 (20%)	MUL, DIV
22	1 (33%)	AND, AND, AND
21	7 (35%)	ADD, MUL, MUL, ADD, MUL, DIV
21	3 (27%)	MUL, DIV, ADD
18	2 (20%)	MUL, DIV
18	9 (33%)	MUL, ADD, DIV, MUL, ADD, SUB, MUL, MUL, DIV
18	6 (32%)	MUL, MUL, MUL, DIV, ADD
18	9 (33%)	ADD, DIV, DIV, SUB, ADD, MUL, MUL, MUL

Tabelle B.39.: Ergebnisse der Vollständige Suche mit Maximalgröße 9

Vorwärtssuche

GB	EB (%)	Enthaltene Operationen
30	2 (25%)	MUL, MUL
28	1 (20%)	ADD, MUL
26	2 (20%)	MUL, DIV
22	1 (33%)	AND, AND, AND
21	3 (27%)	MUL, DIV, ADD
18	2 (20%)	MUL, DIV
16	4 (29%)	MUL, MUL, DIV
12	3 (27%)	MUL, ADD, DIV
12	3 (33%)	MUL, MUL, ADD
12	3 (33%)	MUL, ADD, MUL

Tabelle B.40.: Ergebnisse der Vorwärtssuche mit Maximalgröße 3

GB	EB (%)	Enthaltene Operationen
30	2 (25%)	MUL, MUL
28	1 (20%)	ADD, MUL
26	2 (20%)	MUL, DIV
22	1 (33%)	AND, AND, AND
21	7 (35%)	ADD, MUL, MUL, ADD, MUL, DIV
21	3 (27%)	MUL, DIV, ADD
18	2 (20%)	MUL, DIV
18	6 (32%)	MUL, MUL, MUL, DIV, ADD
16	4 (29%)	MUL, MUL, DIV
15	5 (31%)	MUL, ADD, MUL, ADD, DIV

Tabelle B.41.: Ergebnisse der Vorwärtssuche mit Maximalgröße 6

GB	EB (%)	Enthaltene Operationen
30	2 (25%)	MUL, MUL
28	1 (20%)	ADD, MUL
26	2 (20%)	MUL, DIV
22	1 (33%)	AND, AND, AND
21	7 (35%)	ADD, MUL, MUL, ADD, MUL, DIV
21	3 (27%)	MUL, DIV, ADD
18	2 (20%)	MUL, DIV
18	9 (33%)	MUL, ADD, DIV, MUL, ADD, SUB, MUL, MUL, DIV
18	6 (32%)	MUL, MUL, MUL, DIV, ADD
18	9 (33%)	ADD, DIV, DIV, SUB, ADD, MUL, MUL, MUL

Tabelle B.42.: Ergebnisse der Vorwärtssuche mit Maximalgröße 9

Heun

Vollständige Suche

GB	EB (%)	Enthaltene Operationen
55	1 (20%)	ADD, MUL
28	2 (20%)	MUL, DIV
27	1 (33%)	AND, AND, AND
20	2 (25%)	MUL, MUL
16	2 (25%)	MUL, MUL
14	1 (17%)	MUL, ADD, ADD
12	2 (7%)	ADD, MUL, SIN
12	3 (11%)	SIN, MUL, ADD
12	2 (18%)	DIV, MUL, SUB
12	3 (33%)	MUL, MUL, NEG

Tabelle B.43.: Ergebnisse der Vollständige Suche mit Maximalgröße 3

GB	EB (%)	Enthaltene Operationen
55	1 (20%)	ADD, MUL
28	2 (20%)	MUL, DIV
27	1 (33%)	AND, AND, AND
21	7 (35%)	ADD, MUL, MUL, ADD, MUL, DIV
20	2 (25%)	MUL, MUL
20	5 (33%)	MUL, ADD, ADD, MUL, ADD, MUL
20	5 (16%)	ADD, MUL, ADD, MUL, SIN
20	5 (33%)	ADD, MUL, MUL, ADD, MUL, ADD
18	3 (25%)	ADD, SUB, DIV, MUL
18	6 (30%)	MUL, ADD, ADD, DIV, MUL, MUL

Tabelle B.44.: Ergebnisse der Vollständige Suche mit Maximalgröße 6

GB	EB (%)	Enthaltene Operationen
55	1 (20%)	ADD, MUL
32	8 (38%)	MUL, MUL, ADD, MUL, ADD, ADD, ADD, MUL, ADD
30	10 (38%)	ADD, ADD, MUL, ADD, ADD, MUL, MUL, MUL, DIV
30	10 (38%)	MUL, ADD, ADD, MUL, ADD, MUL, MUL, DIV, ADD
28	7 (35%)	MUL, MUL, ADD, ADD, ADD, MUL, ADD, MUL
28	2 (20%)	MUL, DIV
27	9 (36%)	MUL, ADD, ADD, MUL, ADD, MUL, MUL, DIV
27	1 (33%)	AND, AND, AND
27	9 (36%)	ADD, ADD, MUL, ADD, MUL, ADD, MUL, DIV, MUL
27	9 (36%)	ADD, MUL, ADD, ADD, MUL, MUL, MUL, DIV

Tabelle B.45.: Ergebnisse der Vollständige Suche mit Maximalgröße 9

Vorwärtssuche

GB	EB (%)	Enthaltene Operationen
55	1 (20%)	ADD, MUL
28	2 (20%)	MUL, DIV
27	1 (33%)	AND, AND, AND
20	2 (25%)	MUL, MUL
16	2 (25%)	MUL, MUL
14	1 (17%)	MUL, ADD, ADD
12	2 (7%)	ADD, MUL, SIN
12	3 (11%)	SIN, MUL, ADD
12	2 (18%)	DIV, MUL, SUB
12	3 (33%)	MUL, MUL, NEG

Tabelle B.46.: Ergebnisse der Vorwärtssuche mit Maximalgröße 3

GB	EB (%)	Enthaltene Operationen
55	1 (20%)	ADD, MUL
28	2 (20%)	MUL, DIV
27	1 (33%)	AND, AND, AND
21	7 (35%)	ADD, MUL, MUL, ADD, MUL, DIV
20	2 (25%)	MUL, MUL
20	5 (33%)	MUL, ADD, ADD, MUL, ADD, MUL
20	5 (16%)	ADD, MUL, ADD, MUL, SIN
20	5 (33%)	ADD, MUL, MUL, ADD, MUL, ADD
18	3 (25%)	ADD, SUB, DIV, MUL
18	6 (30%)	MUL, ADD, ADD, DIV, MUL, MUL

Tabelle B.47.: Ergebnisse der Vorwärtssuche mit Maximalgröße 6

GB	EB (%)	Enthaltene Operationen
55	1 (20%)	ADD, MUL
32	8 (38%)	MUL, MUL, ADD, MUL, ADD, ADD, ADD, MUL, ADD
30	10 (38%)	ADD, ADD, MUL, ADD, ADD, MUL, MUL, MUL, DIV
30	10 (38%)	MUL, ADD, ADD, MUL, ADD, MUL, MUL, DIV, ADD
28	7 (35%)	MUL, MUL, ADD, ADD, ADD, MUL, ADD, MUL
28	2 (20%)	MUL, DIV
27	9 (36%)	MUL, ADD, ADD, MUL, ADD, MUL, MUL, DIV
27	1 (33%)	AND, AND, AND
27	9 (36%)	ADD, ADD, MUL, ADD, MUL, ADD, MUL, DIV, MUL
27	9 (36%)	ADD, MUL, ADD, ADD, MUL, MUL, MUL, DIV

Tabelle B.48.: Ergebnisse der Vorwärtssuche mit Maximalgröße 9

C. Inhalt der CD

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ALAP As-Late-As-Possible

ALU Arithmetisch-logische Einheit

ASAP As-Soon-As-Possible

CGRA Coarse-Grain Reconfigurable Array

CPU Central Processing Unit

CR-Architektur Centralized-Register-Architektur

DFG Datenflussgraph

FPGA Field-Programmable Gate Array

HLS High-Level Synthese

Makro-Op Makro-Operation

PE Processing Element

RDR-Architektur Regular-Distributed-Register-Architektur

WSP Weighted Subroutine Partition

Literatur

[1] *UltraSynth-Projekt Homepage*. URL: `http://ultrasynth.de/` (besucht am 29. 12. 2017).

[2] *iXtronics Homepage*. URL: `http://www.ixtronics.de/` (besucht am 29. 12. 2017).

[3] D. L. Wolf. "Design and Implementation of a Generic CGRA for Hardware-Synthesis on AMIDAR". Masterthesis. TU Darmstadt, Sep. 2015.

[4] T. Ruschke. "Design and implementation of a Scheduling Algorithm for a CGRA with regard to Routing Constraints". Masterthesis. TU Darmstadt, Sep. 2015.

[5] D. C. Zaretsky, G. Mittal, R. P. Dick und P. Banerjee. "Balanced Scheduling and Operation Chaining in High-Level Synthesis for FPGA Designs". In: *8th International Symposium on Quality Electronic Design (ISQED'07)*. März 2007, S. 595–601.

[6] K. Terada, M. Yanagisawa und N. Togawa. "A floorplan-driven high-level synthesis algorithm with operation chainings using chaining enumeration". In: *2014 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*. Nov. 2014, S. 248–251.

[7] K. Terada, M. Yanagisawa und N. Togawa. "A floorplan-driven high-level synthesis algorithm with multiple-operation chainings based on path enumeration". In: *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*. Mai 2015, S. 2129–2132.

[8] M. Nepolean und K. Sivasubramanian. "An optimized design for Fused add-multiply operation". In: *2016 Online International Conference on Green Engineering and Technologies (IC-GET)*. Nov. 2016, S. 1–5.

[9] G. Venkataramani und S. C. Goldstein. “Operation chaining asynchronous pipelined circuits”. In: *2007 IEEE/ACM International Conference on Computer-Aided Design*. Nov. 2007, S. 442–449.

[10] M. R. B. Kristensen, S. A. F. Lund, T. Blum und J. Avery. “Fusion of parallel array operations”. In: *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*. Sep. 2016, S. 71–85.

[11] C. Galuzzi und K. Bertels. “The Instruction-Set Extension Problem: A Survey”. In: *ACM Trans. Reconfigurable Technol. Syst.* 4.2 (Mai 2011).

[12] S. Döbrich und C. Hochberger. “Exploring Online Synthesis for CGRAs with Specialized Operator Sets”. In: *Int. J. Reconfig. Comput.* 2011 (Jan. 2011).