Reduzierung der Kommunikation in TTA-Verbindungsnetzen mittels Laufzeitanalyse

Nico Moser, Stefan Hauser, Carsten Gremzow

Technische Universität Berlin,
Institut für Technische Informatik und Mikroelektronik,
10587 Berlin
nico.moser@tu-berlin.de, hausers@cs.tu-berlin.de, carsten.gremzow@tu-berlin.de

Die meisten der derzeit verfügbaren Frameworks für die Synthese von applikationsspezifischen Prozessoren nutzen, genauso wie auch Hardware Software Co-Design Umgebungen, hardwareseitig RISC basierte Templates oder Erweiterungen für Standard RISC-Prozessoren. Eigenschaften wie Out-of-Order Execution oder Bypassing beschränken die Erweiterbarkeit jedoch. Unser Designansatz für Applikationsspezifische Prozessoren mit hoher Parallelität auf Befehlsebene basiert daher auf einer kontrollflussgesteuerten Datenflussarchitektur, die sowohl Eigenschaften von transport- als auch von operationsgesteuerten Architekturen kombiniert. Es soll gezeigt werden, wie mit Hilfe der dynamischen Analyse der abzubildenden Applikation die inheränte Parallelität und Flexibilität der *transport triggered architecture (TTA)* genutzt wird, um den Entwurfsraum zu untersuchen und die Synthese des Zielprozessors vorzubereiten. Weiterhin wird die Erweiterung der *TTA* um verteilte Registerfiles vorgestellt, deren Verwendung zu einer Entlastung bei der Ergebnisweiterleitung und somit der Bedingungen bei der Ablaufplanung führt. Die Analysetechniken und die eingeführten Architekturmerkmale führen zusammen genommen zu einer erhöhten Flexibilität bei der Codegenerierung und einem höheren Durchsatz, ohne eine Mehrbelastung des für *TTA* typischen Netzwerks.

1 Einführung

Die stetig steigende Komplexität von Applikationen in unterschiedlichen Anwendungsfeldern eingebetteter Systeme (z.B. Multimedia) erfordert nicht nur eine intensive Entwurfsraum-Untersuchung, wie es aus dem klassischen automatisierten Hardwareentwurf bekannt ist, sondern auch eine umfassende Unterstützung der Softwareanalyse und -generierung. Weithin als Lösung akzeptiert sind von Entwurfsautomatisierungs-Werkzeugen speziell angepasste Prozessoren (*application specific instruction set processor (ASIP)*). Dieser Ansatz setzt sich aus zwei wichtigen Komponenten zusammen: Die *ASIP* Architektur selbst und die dazugehörende Werkzeuge wie Compiler und Linker.

Zu den zentralen Eigenschaften einer ASIP Architektur lassen sich die Folgenden zählen: Erweiterbarkeit, Flexibilität und optimaler Gebrauch von Parallelität auf Befehlsebene. Wichtig dabei ist, dass all diese Eigenschaften einfach genug sein müssen, um eine vollautomatische Unterstützung durch die Werkzeugkette zu erhalten. Von geringerer Priorität erweisen sich Aspekte des traditionellen Prozessorentwurfs wie ein orthogonaler Befehlssatz und Lesbarkeit der Befehls- und Assemblerebene. Dafür gibt es mehrere Gründe: Zu allererst gibt es in einem automatischen Entwurfsfluss keine Notwendigkeit für einen Übersetzungszwischenschritt. Zweitens sind Referenzimplementierungen aktueller Anwendungen in Standard-Hochsprachen geschrieben und ihre Komplexität macht es unwahrscheinlich, mit vergleichbaren Implementierungen auf Assemblerebene zu arbeiten. Zudem ist Assemblerprogrammierung oftmals nur die letzte Möglichkeit, die Rechenleistung existierender Prozessorarchitekturen optimal auszunutzen - in Hinblick auf automatisierte Synthese von applikationsspezifischen Prozessoren und gleichzeitiger Konstruktion dazugehörender Softwarewerkzeuge jedoch obsolet.

Entwurfswerkzeuge sollten von ihren Analyse- und Optimierungsmöglichkeiten bei der Erzeugung von ASIP Strukturen und bei der Codegenerierung Nutzen ziehen. Wir erachten besonders die Möglichkeit der Analyse des Laufzeitverhaltens der Applikation als Vorbereitung der Architektursynthese für bedeutend. Zudem ist die Unterstützung mehrerer gebräuchlicher Programmiersprachen für eine erhöhte Akzeptanz bei Endanwendern zielführend.

Der Rest dieser Arbeit gliedert sich wie folgt: Im nächsten Abschnitt wird auf vorhergehende sowie thematisch eng verbundene Arbeiten Bezug genommen. Im darauf Folgenden werden Details der erweiterten *TTA* erläutert. Anschließend werden Analysemethoden beschrieben. Im fünften Abschnitt werden Ergebnisse vorgestellt und im letzten Abschnitt wird ein Ausblick gegeben.

2 Vorhergehende Arbeiten

2.1 Architektur

Das hervorstechende Merkmal der *TTA* ist die hohe Parallelität auf Befehlsebene [2]. Erreicht wurde diese, indem bei der Grundstruktur der *TTA* Anlehnungen bei Datenflussarchitekturen genommen wurden. Es gibt ausschließlich eine Operation, die die Verbindungen zwischen den Funktionseinheiten und somit den Datenfluss steuert. Da für jede Transportoperation ein Datenbus erforderlich ist, ist das aus der Gesamtheit der Busse resultierende Verbindungsnetzwerk, welches zwar flexibel, aber hinsichtlich des Flächenverbrauches auch sehr teuer ist, die sensibelste Komponente der *TTA*. Weitere Untersuchungen [1] zielten deswegen auf die Entlastung und damit einhergehend Komplexitätsreduzierung des Verbindungsnetzwerkes mittels Ausnutzung funktionseinheitlokaler Techniken. Es gibt weiterhin Untersuchungen, die sich mit der Speicherarchitektur [5, 4] und der Verarbeitung von Konstanten in Operationen [7] beschäftigen.

2.2 Analyse Framework

Die *Low Level Virtual Machine (LLVM)* ist ein Übersetzer-Werkzeug, das Hilfsmittel zur lebenslangen Optimierung bereitstellt (Optimierung zur Übersetzungszeit, während des Linkens, zur Laufzeit). Grundlage der *LLVM* ist die Arbeit von Chris Lattner [6]. Neben der Möglichkeit zur Analyse während der Laufzeit, die vor allem auch durch den integrierten Interpreter unterstützt wird, besticht die *LLVM* auch durch Besonderheiten wie Datentypen mit

variabler Bitbreite. Das prädestiniert dieses Werkzeug für automatisierte Entwurfswerkzeuge sowohl im kommerziellen [11] wie auch im akademischen Bereich [12] .

Außerdem erzeugt die *LLVM* die Ausgangsbeschreibung für das *Low Level Intermediate Language Analyzer (LLILA)* Werkzeug [3]. *LLILA* wird entwickelt, um den globalen quantitativen Datenfluss von Applikationen zur Laufzeit zu analysieren und die Ergebnisse zur automatisierten Partitionierung der jeweiligen Applikation zu verwenden. Eine mögliche Zielarchitektur stellt beispielsweise die hier vorgestellte *TTA* dar.

3 TTA-ASIP Architektur

Einem synthetisierbaren applikationsspezifischen Prozessor sollte eine Architektur zugrunde liegen, deren Eigenschaften speziellen Anforderungen genügen. In erster Linie sollten das Parametrisier- und Erweiterbarkeit sein. Diese Eigenschaften sind zwar die Voraussetzung für den Entwurfsfluss, jedoch oftmals schlecht vereinbar mit anderen Eigenschaften der jeweiligen Architektur (zum Beispiel Datenpfadabhängigkeiten zwischen verschiedenen Segmenten einer Architektur). Inhärent gegebene Parallelität sollte nicht nur wegen der Spezialisierung sondern auch wegen der Nebenläufigkeit eine weitere Eigenschaft der Architektur sein. Aus diesem Grunde entschieden wir uns für den *synZEN* als Grundlage für die synthetisierbare Zielarchitektur, die viele Ähnlichkeiten mit der *TTA* hat.

3.1 Strukturüberblick

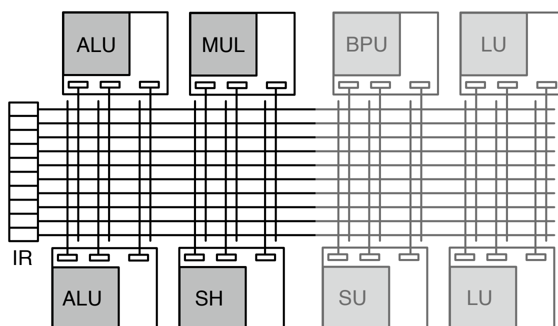


Abbildung 1: Schematischer *synZEN* Überblick: Links das Befehlsregister mit den Transportoperationen. Das Verbindungsnetzwerk befindet sich in der Mitte und die synZEN-Einheiten sind über und unter dem Netzwerk angeordnet.

Der Strukturüberblick in Abbildung 1 zeigt den prinzipiellen Aufbau des *synZEN* mit den wichtigsten Komponenten. Auf der linken Seite der Abbildung ist das Befehlsregister zu sehen. Die Befehlsworte beinhalten Transportoperationen, die den Datenfluss im Verbindungsnetzwerk steuern. Das Netzwerk befindet sich rechts vom Befehlsregister und ist mit den synZEN-Einheiten, die sich ober- und unterhalb befinden, verbunden. Die synZEN-Einheiten dienen zur Datenverarbeitung und -haltung.

3.2 synZEN-Einheit

Die synZEN-Einheit (s. Abb. 2) besteht im Wesentlichen aus der eigentlichen Funktionseinheit und einem Interface-Wrapper.

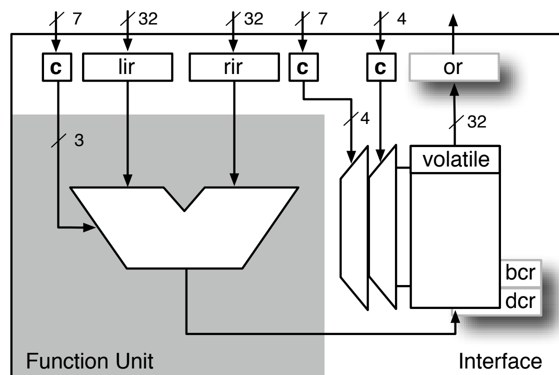


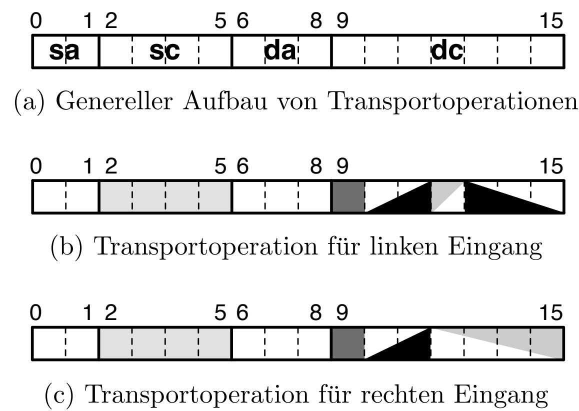
Abbildung 2: Unten links befindet sich die Funktionseinheit der synZEN-Einheit. Der Interface-Wrapper untergliedert sich in die beiden Eingangsregister (*lir,rir*), das Ergebnis-Registerspeicher und die Register für die verschiedenen Steuerungsbits (*c*). Ergänzend sind die virtuellen Register *bcr,dcr,or* abgebildet.

Funktionseinheit. Die Funktionseinheit, die in Abbildung 2 unten links dargestellt ist, ist das datenverarbeitende Element der synZEN-Einheit und somit der Teil, indem sich die unterschiedlichen synZEN-Einheiten am meisten unterscheiden. Die Funktionalität beinhaltet wie bereits in Abbildung 1 angedeutet: Sprungeinheit *BPU*, Lade- und Speichereinheit *LU,SU* sowie binäre Operationen *ALU,MUL,SH*. Diese Arbeit legt den Fokus der Untersuchungen auf die Binäroperationen, um die Anschaulichkeit zu bewahren und da sie für einige Analysen besonders geeignet sind. Aus diesem Grunde sind die synZEN-Einheiten, die keine Binäroperationen enthalten, in Abbildung 1 ausgegraut.

Ergebnis-Registerspeicher. Der auf der rechten Seite der Abbildung 2 zu sehende Registerspeicher, verfügt sowohl über einen Lese- wie auch einen Schreib-Port. Schreiboperationen werden über die Steuerbits *c*, die dem rechten Eingangsregister *rir* zugeordnet sind, adressiert, Leseoperationen über die dem Ausgang zugeordneten Steuerbits.

In der jetzigen Realisierung stehen 16 Adressen zur Verfügung. Drei der Adressen haben eine besondere Bedeutung. Die Erste ist reserviert für die synZEN-Einheit interne Rückkopplung (*bcr*), die das Netzwerk entlasten soll. Denselben Zweck verfolgt die zweite Adressreservierung (*dcr*), die allerdings eine direkte Kopplung zweier synZEN-Einheiten erreichen soll. Sowohl *dcr* als auch *bcr* sind somit keine realen Register sondern Teil des Registerfiles. Die dritte Adresse ist mit *volatile* gekennzeichnet, weil sie als Ergebnisspeicher für Operaionen zur Verfügung steht, die keine Ergbnisadresse angeben können. Alle drei Adressen haben somit einen Primärzweck außerhalb der normalen Ergebnisspeicherung und sind nur bedingt geeignet für Datenhaltung.

Eingangsschnittstelle. Neben dem Ergebnis-Registerspeicher gehört der Eingang zur Kommunikationsschnittstelle. Dieser besteht im Wesentlichen aus den Eingangsregistern für den linken (*lir*) und den rechten (*rir*) Eingang sowie den jeweils dazugehörigen Steuerbits (*c*)(in Abbildung 2 oben), aber auch aus Dekodiereinheiten, die die Steuerbits auswerten und Signale weitergeben.



(a) Genereller Aufbau von Transportoperationen

(b) Transportoperation für linken Eingang

(c) Transportoperation für rechten Eingang

Abbildung 3: Die Transportoperationen: (3a) zeigt den generellen Aufbau, (3b) eine Transportoperation mit den spezifischen Steuerbits für den linken und (3c) für den rechten Eingang einer synZEN-Einheit.

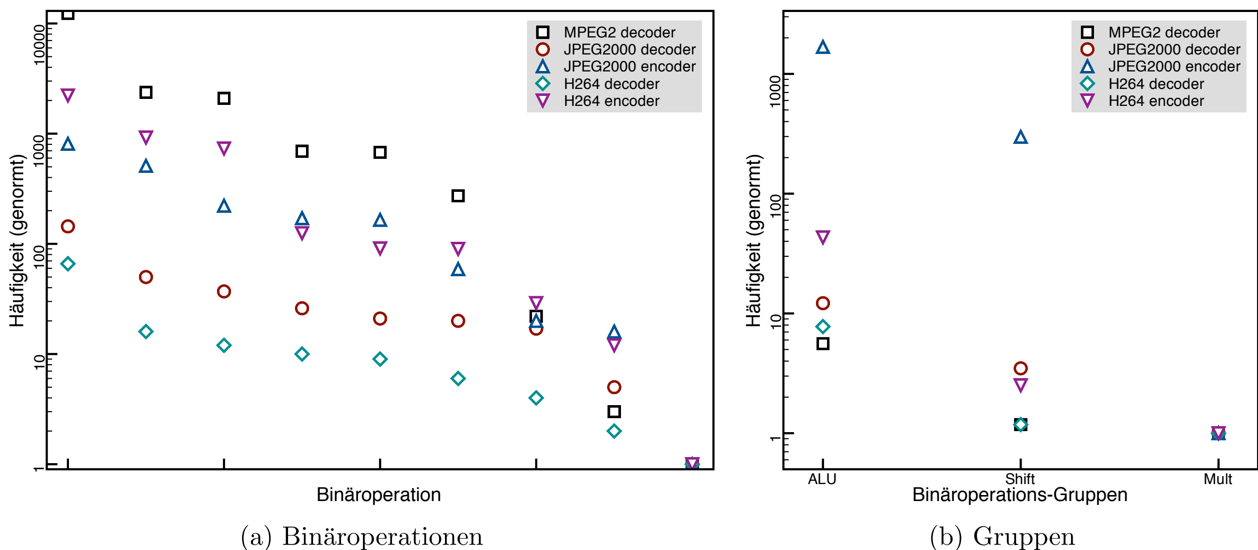
3.3 Befehlsregister und Verbindungsnetzwerk

Für jeden Datentransportbus gibt es im Befehlswort exakt eine Transportoperation. Die Busse werden in Abbildung 1 durch horizontale Linien dargestellt. Von der Anzahl der Busse hängt somit auch die maximale Anzahl von Daten ab, die pro Takt zwischen den synZEN-Einheiten transportiert werden können.

Transportoperationen. In Abbildung 3a ist eine Transportoperation in ihrem typischen Aufbau dargestellt. Eine Transportoperation besteht im Wesentlichen aus einem Quell- und einem Zielteil, die wiederum unterteilt sind in jeweils einen Adress- und einen Steuerteil. Prinzipiell ist der Teil, der Daten für die Quelle enthält, kleiner als der Zielanteil. Zum Einen werden für die Quelle weniger Steuerbits benötigt, zum Anderen sind die meisten synZEN-Einheiten dyadisch ausgelegt oder gar rein konsumtiv, sodass es im Verbindungsnetz mehr Senken als Quellen gibt. Der Adressanteil kann je nach Implementierung variieren, der Steueranteil ist wie im Folgenden beschrieben festgelegt. Wie den Abbildungen 3b und 3c zu entnehmen ist, unterscheiden sich die Steuerbits nur von Bit 12 bis 15, die Restlichen beinhalten die gleichen Angaben. Bit 2 bis 5 (Steuerbits des Quellteils) der Transportoperation beinhalten die Adresse des Ergebnisregisterfiles der Quell-synZEN-Einheit. Bit 9 ist das *permanent*-Bit, das dafür Sorge trägt, dass der Wert im Eingangsregister wiederverwendet werden kann. Bits 10 bis 11 wählen aus, ob der Operand aus dem Verbindungsnetzwerk, dem *dcr* Register einer verkoppelten synZEN-Einheit oder aus dem *bcr* Register der Adressierten genommen werden soll. Bei den restlichen vier Bits unterscheidet sich die Belegung je nach Eingang. Die Steuerbits 12 bis 15 bei dem rechten Eingang (Abbildung 3c) adressieren das Ergebnis-Registerspeicher zum Abspeichern des von der Funktionseinheit erzeugten Resultats. Beim linken Eingang dient Bit 12 dafür Steuerbits des rechten Einganges als Immediate-Wert zu interpretieren und die Bits 13 bis 15 werden als Operationscode an die Funktionseinheit weitergegeben.

4 Analyse und Optimierung

Für die Laufzeitanalyse haben wir Benchmarks ausgewählt, deren Umfang dem aktueller Problemstellungen entspricht, die einem wichtigen Einsatzgebiet applikationsspezifischer Pro-



(a) Binäroperationen

(b) Gruppen

Abbildung 4: Die zur Laufzeit gemessenen Häufigkeiten: (4a) zeigt die Häufigkeiten der neun untersuchten Binäroperationen, (3b) zeigt die Häufigkeiten in Gruppen zusammengefasster Binäroperationen.

zessoren entspringen (Multimedia) und deren Quellcode in den Sprachen C/C++ vorliegt [10, 8, 9]. Die Laufzeitanalyse selbst erfolgt mittels Profiling auf Basisblock-Ebene gekoppelt mit den jeweils zu untersuchenden Elementen der Applikation (Befehle, Datenabhängigkeiten etc.).

Analyse der Häufigkeit von Instruktionen. Eine essentielle Analyse stellt für uns die Untersuchung der Aufrufhäufigkeit der einzelnen Binäroperationen dar. Bei der ursprünglichen *TTA* war durch den Datenflusscharakter und die Simplizität der Transportoperationen vorgegeben, dass die einzelnen Funktionseinheiten elementare Funktionen abbilden und ohne Steuereingabevektoren auskommen. Für die hier benutzten Benchmarkapplikationen würde sich daraus folgendes Problem ergeben: Wie in Abbildung 4a zu sehen ist, gibt es eine hohe Diskrepanz zwischen den am wenigsten häufig benutzten Operationen (für jeden Benchmark jeweils ganz rechts im Graphen) und denen, die am häufigsten gebraucht werden (ganz links). Die Aufrufhäufigkeit unterscheidet sich mitunter um vier Größenordnungen. Bei begrenzten Hardwareressourcen hätte das zur Folge, dass eine notwendige eins zu eins Abbildung jeder Operation zu Funktionseinheiten führen würde, die nur selten benutzt werden und somit nicht ausgelastet sind.

Ein etwas anderes Bild ergibt sich, wenn man die Operationen gruppiert und die Aufrufhäufigkeit auf Gruppenbasis miteinander vergleicht. Wie in Abbildung 3b zu sehen, beträgt der Unterschied im Normalfall nur bis zu einer Größenordnung. Lediglich bei dem JPEG2000 Encoder Benchmark ist die Diskrepanz sehr viel größer und in diesem Fall wäre es zu überlegen, die Multiplikationsoperationen in Abfolgen von Shift- und Additionsbefehlen zu zerlegen und auf die entsprechenden Operationsgruppen zu verteilen.

Analyse der Häufigkeit von Operationsmustern Operationsmuster symbolisieren direkten Datenfluss zwischen zwei Operationen. Die Analyse dieser Muster kann zur Identifizie-

Benchmark	gleiche Operationen in %	gleiche Gruppe in %	Häufigkeit von Operationsmustern zwischen zwei Gruppen in %					
			Shift→ALU	ALU→Shift	MUL→ALU	ALU→MUL	MUL→Shift	Shift→MUL
mpeg2 dec	28.8	38.5	5.2	9.2	7.3	39.4	0.5	0
jpeg2000 dec	12.5	33.8	7.8	31.1	0	0	0	19.5
jpeg2000 enc	14.6	39.6	7.4	51.6	0	0	0	0
h264 dec	24.9	46.5	8.5	18.2	9.1	14.2	0	0.6
h264 enc	28.7	32.8	3.7	34.4	0.6	27.9	0	0.5

Tabelle 1: Die zweite Spalte zeigt den prozentualen Anteil von Operationsmustern mit zwei gleichen Operationen, die dritte Spalte den prozentualen Anteil an Mustern mit Operationen aus der selben Operationsgruppe. Die restlichen Spalten zeigen den prozentualen Anteil von Operationsmustern mit den jeweils im Tabellenkopf ausgewiesenen beteiligten Operationen.

rung stark verbundener Operationen führen, was wiederum die Designentscheidung für eine Rückkopplung innerhalb einer synZEN-Einheit (*back-coupling register, bcr*) bzw. der direkten Kopplung zweier synZEN-Einheiten (*direct-coupling register, dcr*) zur Folge haben kann. In Tabelle 1 sind diese Muster aufgelistet.

In der zweiten Spalte sind aus Referenzgründen die Werte für Operationsmuster angegeben, die aus zwei gleichen Operationen bestehen. Anhand der in der dritten Spalte aufgetragenen Zahlen ist zu erkennen, dass der Anteil an Datenbewegungen, die innerhalb einer synZEN-Einheit zurückgeführt werden können, teilweise deutlich steigt, wenn man Operationen gruppiert und den funktionalen Teil der Einheiten so komplex gestaltet, dass er in der Lage ist, diese Operationsgruppen abzubilden.

Die weiteren Spalten zeigen die Konnektivität zwischen unterschiedlichen Operationsgruppen. Hier ist ebenfalls ein erheblicher Anstieg gegenüber der Einzelbetrachtung von Operationen zu verzeichnen, auch wenn diese hier nicht aufgeführt sind. Auffallend ist die Varianz der Wichtigkeit einzelner Verbindungen je nach Applikation. Zudem fällt auf, dass manche Kombinationen nahezu gar keine Rolle spielen (siehe zweite Spalte von rechts in Tabelle 1)

Befehls-Diversifizierung Es gibt mehrere Faktoren die für eine Diversifizierung bei den zu implementierenden Befehlen sprechen. Zum einen basiert der von der *LLVM* erzeugte Code auf ein paar wenigen Befehlen, mit deren Hilfe alle Operationen abgebildet werden und der somit nach der Analyse Raum für Erweiterungen hinsichtlich der Hardwareimplementierung lässt. Der Operationscodevector, der den synZEN-Einheiten übergeben wird, hat zudem genug freie Zustände, um jeder Operationsgruppe weitere Befehle hinzuzufügen. Weiterhin können Spezialbefehle eine Verringerung von Transportoperationen bedeuten.

In Tabelle 2 ist dafür ein Beispiel angegeben. Da es nicht möglich ist den synZEN-Einheiten am linken Eingang einen kleinen Immediate-Wert zu übergeben (wie dem rechten Eingang), sind Operationen, die einen Immediate-Wert auf dem linken Eingang erwarten, nur mit einer erhöhten Zahl an Transportoperationen zu bedienen. Dem kann man teilweise entgehen, indem

Benchmark		SUB 0	SHL 1	
mpeg2 dec	2.02%	0	0.01%	2.02%
jpeg2000 dec	5.27%	0.72%	3.11%	1.44%
jpeg2000 enc	8.51%	6.07%	2.26%	0.19%
h264 dec	7.08%	5.26%	1.30%	0.52%
h264 enc	1.20%	1.14%	0.04%	0.01%

Tabelle 2: Die hellgraue Spalte zeigt die prozentuale Benutzung des linken Eingangs der synZEN-Einheit mit konstanten Werten, die dunkelgraue Spalte zeigt die gleiche Benutzung nach Identifizierung und Ausgliederung zweier spezieller Instruktionen.

#	#Einh.	#Busse	Ziele/Bus	Quellen/Bus	Slices	längster Pfad	Synthesezeit
1			4	2	6%	13,4ns	0m37.684s
2	8	16	8	4	12%	15,0ns	1m34.687s
3			11	6	19%	18,8ns	3m40.005s
4			6	3	14%	12,6ns	2m12.813s
5	12	24	8	4	17%	15,3ns	2m10.118s
6			12	6	29%	19,2ns	8m53.159s

Tabelle 3: Syntheseergebnisse für generische Verbindungsnetzwerkvarianten mit einer unterschiedlichen Anzahl von angeschlossenen synZEN-Einheiten, Bussen sowie Verbindungen zwischen Bussen und Quellen und Zielen.

man häufig gebrauchte Operationen, die diesem Schema entsprechen, identifiziert und diesen eigene Befehle zuordnet. In Tabelle 2 ist abgebildet, wie mittels der beschriebenen Maßnahme, der Anteil der Instruktionen teilweise stark reduziert wird. Die beiden identifizierten Befehle, die zu einer Reduzierung bis auf ein Vierundvierzigstel führen, sind die Negation (in der Tabelle mit *SUB 0* gekennzeichnet) und das Schieben einer 1 nach links (*SHL 1*).

5 Implementierung und Ergebnisse

Die Laufzeitanalyse wird mit Hilfe eines Analysepasses der *LLVM* 2.4 durchgeführt. Das Werkzeug zur Generierung der Verbindungsnetzstruktur, das in Java geschrieben ist, liest die Beschreibung des zu erzeugenden Verbindungsnetzes im XML Format ein und gibt synthesefähigen VHDL-Code wieder aus.

Die Synthese des Netzwerkes erfolgt mit Hilfe des Xilinx Werkzeuges XST in der Version 9.2i für den Xilinx FPGA Spartan 3A DSP 3400 (xc3sd3400a-4-fg676). Das System, auf dem die Syntheseergebnisse erzeugt wurden, ist ein Intel Xeon 3.0 GHz mit 4MB Cache, 32 GB Arbeitsspeicher und mit einem Linuxkernel 2.6.24.

Um die prinzipielle Realisierbarkeit des Verbindungsnetzwerkes zu überprüfen, wurden für mehrere mögliche Szenarien die XML-Beschreibungen generiert, anschließend dem Javawerkzeug übergeben und der damit erzeugte VHDL-Code synthetisiert. Der Tabelle 3 können die Syntheseergebnisse entnommen werden. Grundsätzlich lassen sich folgende Aussagen treffen: Sowohl eine steigende Anzahl von Verbindungen (Ziele/Bus, Quellen/Bus) als auch eine steigende Anzahl an Komponenten lassen den Ressourcenbedarf (Fläche, Zeit) ansteigen. Inter-

essant ist dabei die Tatsache, dass anscheinend eine sehr engmaschige Verbindung bei relativ wenig Komponenten einen höheren Ressourcenbedarf hervorruft(#3), als ein Szenario, bei dem es zwar mehr Komponenten gibt, die jedoch lediglich in moderater Ausprägung miteinander verknüpft sind (#5). Die, im Gegensatz zu der Synthesezeit von #5, wider Erwarten hohe Synthesezeit von #4 liegt vermutlich an durchgeführten Optimierungen, die aufgrund der ungünstigen Anzahl von Quellen und Zielen angestoßen wurden.

Eine synZEN-Einheit die eine ALU mit acht verschiedenen Operationen sowie die Schnittstellen inklusive der Register beinhaltet, belegt 2% der Slices. Die Kommunikationsstruktur außerhalb des Verbindungsnetzwerkes ist somit vernachlässigbar. Eine intensive Nutzung der beiden *coupling* Verfahren zur Vermeidung engmaschiger Verdrahtung würde unmittelbar Hardwarekosten einsparen und Leistungssteigerung zur Folge haben.

Für die Zuweisung der Funktionalitäten auf die realisierbaren synZEN-Einheiten wird eine Kostenfunktion bemüht, die die Aufrufhäufigkeit der einzelnen Operationen respektive derer Operationsgruppen auswertet. Initial muss jeder Operation(sgruppe), die Verwendung findet, eine synZEN-Einheit zugewiesen werden, um die Grundfunktionalität zu gewährleisten. Außerdem wird initial die maximale Anzahl der Instanzen (NoI_{max}) einer Operationsgruppe (n) ermittelt, die der Anzahl der Instanzen bei höchstmöglicher Granularität entspräche.

$$NoI_{max}(n) = \frac{Freq_{normed}(n)}{MIN(Freq_{normed})}$$

Ist die Gesamtanzahl von NoI_{max} kleiner als die zu vergebene Anzahl von synZEN-Einheiten, werden die NoI_{max} mit einem Faktor multipliziert, der diesen Umstand aufhebt.

Mit Hilfe von NoI_{max} ist es nun möglich, nach jeder Zuweisung einer synZEN-Einheit zu einer der Operationsgruppen, einen Wert (V) für alle Operationsgruppen zu bestimmen, der einen Vergleich ermöglicht, um die nächste Zuweisung zu bestimmen. Die synZEN-Einheit wird der Operationsgruppe mit diesem Wert zugewiesen.

$$V_n = \frac{NoI_{max}(n) - NoI_{current}(n)}{NoI_{max}(n)} - \frac{\sum_0^i \frac{NoI_{max}(i) - NoI_{current}(i)}{NoI_{max}(i)}}{m-1} \quad | i \neq n, i \leq m$$

Die Kostenfunktion diente dazu die Werte in Tabelle 4 für eine *synZEN*-Implementierung zu ermitteln, die zwölf Einheiten zur Verfügung stellt. Wie eingangs erwähnt, kommt es bei Operationsgruppen mit geringer Aufrufhäufigkeit zu einer prozentualen Überversorgung, während Operationsgruppen mit einem sehr hohen Anteil eher unterversorgt sind.

6 Zusammenfassung und Ausblick

Die Syntheseergebnisse zeigen, dass es möglich ist, auf TTA-ASIP Basis einen stark parametrisierbaren, parallelen Prozessor zu entwerfen und auf FPGAs abzubilden. Weiterhin zeigen die Analyseergebnisse, dass Entwurfsentscheidungen wie *back-coupling* und *direct-coupling* vielversprechend sind und zur Entlastung des Verbindungsnetzwerkes beitragen können.

Im Weiteren wird zu untersuchen sein, wie generierter Code für diese Architekturen mit den für die Synthese gemachten Annahmen in Einklang zu bringen ist und ob beispielweise Befehlsmuster nur basisblockweise zu betrachten sind. Der spekulative Charakter der jetzigen Herangehensweise schränkt die Leistung des generierten Codes eventuell ein.

Als Gegenmaßnahme für die stark steigende Laufzeit entlang des längsten Pfades werden zudem Pipelinestufen in unterschiedlichen Ausprägungen untersucht werden.

Benchmark	ALU %	ALU #	SHIFT %	SHIFT #	MUL %	MUL #
mpeg2 decode	72	8	15.1	2	12.9	2
jpeg2000 decode	76.8	8	20.9	3	6.1	1
jpeg2000 encode	84.9	9	15.0	2	0.1	1
h264 decode	76.6	8	13.5	2	9.9	2
h264 encode	92.5	10	5.4	1	2.1	1

Tabelle 4: Für jede Operationsgruppe ist angegeben wie hoch der prozentuale Anteil an der Aufrufhäufigkeit bezogen auf Binäroperationen ist und wie sich das auf die Zuweisung von synZEN-Einheiten auswirkt, bei einer Implementierung mit zwölf Einheiten.

Literatur

[1] M. Arnold et al., *Data Transport Reduction in Move Processors.*, Third Annual Conference of the Advance School for Computing and Imaging, Heyen, The Netherlands, June 1997.

[2] H. Corporaal, *Design of transport triggered architectures*, 'Design Automation of High Performance VLSI Systems'. GLSV '94, Proceedings., Fourth Great Lakes Symposium on VLSI, 1994.

[3] C. Gremzow, *Compiled low-level virtual instruction set simulation and profiling for code partitioning and ASIP-synthesis in hardware/software co-design*, SCSC: Proceedings of the 2007 summer computer simulation conference, pp. 741-748, 2007

[4] J. Heikkinen et al., *Immediate Optimization for Compressed Transport Triggered Architecture Instructions*, in Proc. Int. Symp. on System-on-Chip, Tampere, Finland, Nov. 19-21 2003.

[5] I. Janssen, *Enhancing the Move Framework. Endianness Port and Immediates Handling.*, Master Thesis, May 2001.

[6] C. Lattner et al., *LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation*, Proceedings of CGO, 2004

[7] J.K. Tanskanen et al., *Parallel Memory Architecture for TTA Processor* , in Embedded Computer Systems: Architectures, Modeling, and Simulation: Proc. 7th Int. Workshop SAMOS VII., S. Vassiliadis, M. Bereković, T.D. Hämäläinen, Volume LNCS 4599, pp. 233–240. Springer-Verlag, Berlin, Germany, 2007.

[8] Official reference implementation of the JPEG-2000 Part-1 codec, *http://www.ece.uvic.ca/ mdadams/jasper/*

[9] H.264/AVC reference software, *http://iphome.hhi.de/suehring/tml/*

[10] MPEG Software Simulation Group, *MPEG-2 Encoder / Decoder, Version 1.2*, http://www.mpeg.org/MSSG, 1996

[11] AutoESL Design Technologies, Inc. *http://www.autoesl.com*

[12] TCE - TTA-based Codesign Environment *http://tce.cs.tut.fi/*