Toolumgebung für plattformbasierte DSPs der nächsten Generation

Matthias H. Weiss, Gerhard P. Fettweis
Systemonic AG
Am Waldschlößchen 1
01099 Dresden, Germany
{matthias.weiss, gerhard.fettweis}@systemonic.de

Markus Lorenz, Rainer Leupers, Peter Marwedel
Universität Dortmund
Lehrstuhl Informatik 12
44221 Dortmund
{lorenz, leupers, marwedel}@ls12.cs.uni-dortmund.de

Zusammenfassung

Digitale Signalprozessoren (DSPs) werden überall dort eingesetzt, wo bestimmte Performance- oder Aufwandsanforderungen nicht von Standardprozessoren erfüllt werden können. Beispiele sind Media- und Sprachcodecs, Modems, sowie Bild- und Spracherkennungen. Aufgrund der unterschiedlichen Performanceanforderungen dieser Anwendungen beginnen DSP-Hersteller nun, nicht *einen* DSP sondern eine *Plattformlösung* anzukündigen. Beispiele sind der StarCore von Motorola/Lucent oder der TigerSharc von Analog Devices. Am Lehrstuhl Mobile Nachrichtensysteme der TU Dresden ist hierbei die M3-DSP Plattform entstanden [Fettweis et al., 1998]. Eine Programmierumgebung für eine solche DSP-Plattform muß im Gegensatz zu konventionellen Ansätzen auch die Anforderungen der Plattform mit unterstützen. In diesem Paper stellen wir die Architektur einer solchen Toolumgebung für die M3-Plattform vor und demonstrieren die Anwendbarkeit anhand von Beispielen.

1 Einleitung

Softwaregetriebene Aufgaben der digitalen Signalverarbeitung werden heutzutage von *Digitalen Signalprozessoren* (DSPs) übernommen. Dies gilt insbesondere für das immer mehr an Bedeutung gewinnende Feld der Systeme auf einem Chip (*SoC: Systems on Chip*). Beispiele sind Audio-, Video-, Modem- oder Sprachapplikationen. In diesen SoC-Domänen können Systemaufgaben auf unterschiedliche SoC-Komponenten wie Mikrocontroller, zugeschnittene *applikationsspezifische integrierte Schaltkreise* (ASICs) oder DSPs partitioniert werden. Wenn hierbei die Anforderungen an den DSP steigen, die zugrundeliegene Architektur aber nicht erweiterbar ist, müssen wiederum Spezial-ASICs eingesetzt werden, obwohl eine DSP-Flexibilität benötigt wird. Deshalb werden insbesondere in SoC-Domänen DSP-Architekturen benötigt, die den jeweiligen Systemanforderungen angepaßt werden können. Diesen Ansprüchen kann nur eine DSP-Plattform genügen.

1.1 Anforderungen

Neue DSP Architekturen sind insbesondere in Applikationen mit hohen Datenraten wie Video- oder Graphik-Codecs, Spracherkennung oder Modems mit hohen Datenraten gefordert. Hierbei können die Datenraten durchaus in ähnlicher Größenordnung wie der Prozessortakt liegen. Die Verarbeitung ist hierbei nur durch Ausnutzung von Parallelität möglich. Dies kann einerseits durch die Nutzung *grobkörniger Parallelität* geschehen, indem bespielsweise unterschiedliche Algorithmenblöcke durch hartverdrahtete Hardwareblöcke realisiert werden. Der Datenfluß wird somit durch die Hardwareimplementierung nachgeahmt. Wenn sich hierbei allerdings die Anwendung ändert, muß auch die Hardware vollständig modifiziert werden.

Einen anderen Ansatz liefert die Nutzung *feinkörniger* Parallelität. Dies kann beispielsweise durch eine Prozessorarchitektur mit parallelen Datenpfaden geschehen. Eine solche software-programmierte Lösung kann sogar effizienter als eine hart verdrahtete Lösung sein [Kim et al., 1997]. Da hierbei Parallelität auf der Instruktionsebene genutzt wird, müssen die

Algorithmen auf einer gemeinsamen Architektur aufsetzen. Insbesondere für SoC-Domänen sind also Prozessorarchitekturen nötig, die *skalierbar* und *erweiterbar* sind.

1.2 DSP Architekturen - State of the Art

Ein Ansatz, DSP-Architekturen zu erweitern, liegt darin, den kompletten DSP-Kern zu duplizieren. Ein Beispiel hierfür stellt die Multicore-Lösung C80x von TI dar. Die Architektur umfaßt 4 DSP-Einheiten, 4 Speicher, und einen RISC-Kern. Die Verbindungen werden durch einen Crossbar-Schalter realisiert. Dieser Ansatz führt jedoch zu einem aufwendigen Kommunikationsnetzwerk *und* einem komplizierten Programmierparadigma. Auch TI hat dies erkannt und reduzierte die Folgearchitektur C82x auf 2 DSP-Kerne.

Einen anderen Weg nehmen Mediaprozessoren. Diese Prozessoren wie TI's C62x oder Philips' TriMedia sind typischerweise VLIW (*Very Long Instruction Word*) basiert, nutzen große Registerfiles und bieten gepackte Instruktionen an. Das Kommunikationsproblem ist hierbei mittels teurer globaler Registerfiles gelöst, die aber schon in TI's Dual-MAC-DSP C62x zweigeteilt werden mußten.

Zur Lösung dieses Kommunikationsproblems wurde in [Wittenburg et al., 1997] und [Trenas et al., 1998] vorgeschlagen, einen speziellen Matrizenspeicher einzusetzen. Da Speicher jedoch einen der kritischen Systemparameter darstellen, ist meist die Nutzung von Standardkomponenten wünschenswert. Hierfür wird von MicroUnity beispielsweise ein breiter Standardspeicher eingesetzt, was durch eine Gruppierung von Daten erreicht wird [Hansen, 1996]. Dabei nutzen verschiedene Datenelemente eine gemeinsame Adresse und können nur in Gruppen angesprochen werden. Die Verarbeitung basiert auf dem *Single Instruction Multiple Data* (SIMD) Prinzip. Um allerdings die Gruppenregister anzusprechen, nutzt MicroUnity's MediaProcessor wiederum eine spezielle Kommunikationseinheit. Soll die Architektur hierbei erweitert werden - durch Nutzung weiterer Parallelität oder neuer Algorithmen - muß die komplette Kommunikationseinheit neu entworfen werden.

Die in [Fettweis et al., 1998] vorgestellte skalierbare M3-DSP Plattform basiert auf dem Gruppenprinzip. Durch Aufteilung der Architektur in sogenannte Streifen (*engl. slices*), Nutzung von SIMD-Eigenschaften und Einsatz einer modularen VLIW-Befehlssatzarchitektur kann diese Plattform an die jeweiligen Anwendungsbedürfnisse angepaßt werden.

1.3 Programmierumgebung

Die Anpassung der M3-Plattform erfolgt durch zwei wesentliche Merkmale:

- Skalierbarkeit
- Spezialisierbarkeit

Die Skalierbarkeit wird durch eine SIMD-Architektur erreicht. Dazu besitzt die Architektur gleichförmige Streifen, deren Anzahl je nach Anforderung variiert werden kann. Jedem Streifen ist auch ein Teil des Speichers zugeordnet, der als *Gruppenspeicher* ausgelegt ist [Weiss et al., 1999]. Im Gegensatz zu einem MIMD-Ansatz wie bei dem StarCore DSP erlaubt das SIMD-Verfahren eine durchgängige Skalierbarkeit, ohne den Instruktionssatz des DSPs ändern zu müssen. Dies ist eine wesentliche Voraussetzung für eine optimierte Toolumgebung. Eine Spezialisierung dagegen wird nur in den Datenpfaden vorgenommen, die in einer Hochsprache einfach durch neue Datentypen zur Verfügung gestellt werden können. Beispiele sind Galois-Operationen, komplexe Operationen etc. Diese Eigenschaften müssen von der Toolumgebung unterstützt werden.

Dieser Beitrag ist wie folgt strukturiert. In Kapitel 2 wird der Aufbau und die Architektur der M3-Plattform vorgestellt. Für diese Plattform werden in Kapitel 3 die Anforderungen an die Programmiertools abgeleitet. Es wird gezeigt, daß eine Trennung in C-Compiler und Smart Assembler für die Plattform-Architektur von Vorteil ist. Der C-Compiler selbst wird schließlich in Kapitel 4, der Smart Assembler in Kapitel 5 beschrieben.

2 M3-Plattform

Um den Anforderungen unterschiedlicher Applikationen genügen zu können, muß eine DSP-Architektur leicht anpaßbar sein. Dazu muß die Basis um weitere Datenpfade oder Funktionseinheiten erweiterbar sein. Hierfür haben wir die Methode der Orthogonalisierung eingesetzt. Sowohl Algorithmen als auch Architektur sind dazu in *Datentransfer* und *Datenmanipulation* unterteilt. Während im Datentransfer alle Speicher-Register, Register-Register oder Speicher-Speicher-Transfer enthalten sind schließt die *Datenmanipulation* Datenpfadfunktionalitäten wie verbundene oder unterteilte Multiplizierer/Addierer, Galois- oder Integerarithmetik [Drescher et al., 1998] oder die *Add Compare Select* (ACS) Funktionalität zur Unterstützung des Viterbi-Algorithmus ein. Dieser Ansatz erlaubt es, Algorithmen getrennt nach Datentransfer und Datenmanipulation zu behandeln.

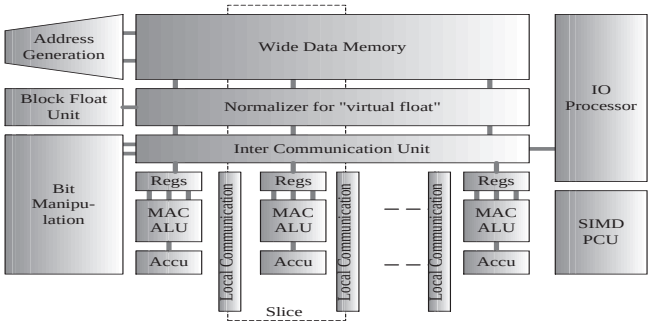

Abbildung 1: *Sliced Architecture*

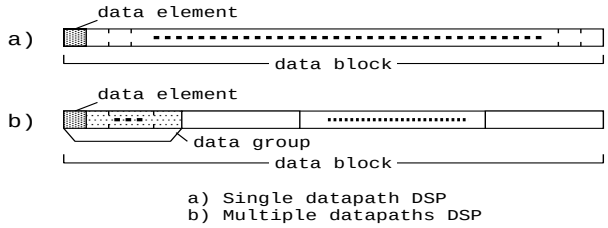

Abbildung 2: *Struktur des Gruppenspeicher*

2.1 Datentransfer and Datenmanipulation

Einer der wesentlichen Unterschiede zwischen unterschiedlichen DSP Architekturen ist die Art und Weise, wie Daten zwischen dem Speicher und den Verarbeitungseinheiten transportiert werden. Eine *Multiplizier/Addier* (MAC) Einheit benötigt beispielsweise drei Eingangswerte und generiert einen Ausgangswert. Wird nun diese MAC-Einheit dupliziert, werden sechs Eingangswerte benötigt und zwei Ausgangswerte generiert. Allgemein muß nun eine Möglichkeit geschaffen werden, nicht für jeden Wert einen eigenen Speicherzugriff durchzuführen. Allerdings ist die Suche nach einem geeigneten Registerfile eine grundlegende Entwurfsentscheidung um Leistungs- und Flächeneffizienz zu erreichen. So mußte TI das C6x-Registerfile zweiteilen, während im HiPAR DSP die Allgemeinheit des Registerfile [Wittenburg et al., 1997] reduziert werden mußte.

Im Gegensatz zu einem globalen Registerfile-Ansatz beinhaltet unser Registerfile genau die Verbindungen, die für DSP-Anwendungen benötigt werden. Diese Verbindungen unterstützen die folgenden Datentransferklassen:

- *Vektor Datentransfer* (VDT),
- *Sliding Window Datentransfer* (SWDT), und
- *Shuffle Datentransfer* (SDT).

Jede dieser Klassen hat unterschiedliche Anforderungen an die Verbindungen innerhalb des Registerfiles. Um nun auch das Registerfile skalierbar zu gestalten, müssen alle diese Klassen unterstützt werden. Die Datenmanipulation kann dagegen zwischen den Algorithmen differieren. Daher erlauben wir, innerhalb unserer skalierbaren Architektur dedizierte Datenpfade einzufügen, um für unterschiedliche Algorithmen spezielle Arithmetik zur Verfügung zu stellen. Die Skalierbarkeit wird durch eine Gruppierung der Daten erreicht.

2.2 Das Gruppenprinzip

Um eine einfache *Adreßgeneriereinheit* (AGU) zur Verfügung zu stellen und trotzdem einen großen Datendurchsatz zu ermöglichen, haben wir das Prinzip von Datengruppen angewendet. Dazu wird ein Block von Eingangsdaten, z.B. ein Datenrahmen, nicht gleich in Elemente, sondern in Gruppen eingeteilt (s. Abbildung 2). In einer Gruppe sind mehrere Elemente zusammengefaßt, die nur gemeinsam als Gruppe adressiert werden können. Damit kann in unserer Architektur ein Speicherzugriff Daten für jeden Streifen zur Verfügung stellen. Mehrere Gruppen können im Registerfile zwischengespeichert und mit Hilfe von Softwarebefehlen umgeordnet werden. Hierfür steht spezielle Toolunterstützung zur Verfügung.

3 Toolumgebung

Die M3-Plattform ist flexibel in der Anzahl der Streifen und der Struktur der Datenpfade. Sie basiert jedoch auf einer gleichbleibenden Grundarchitektur, die einen Streifen enthält. Somit kann ein optimierender Compiler auf die feststehenden Teile der Zielarchitektur zugeschnitten werden. Für die Nutzung der flexiblen Teile - wie Nutzung der Parallelität und Spezialdatenpfade - wird ein *Smart Assembler* eingesetzt. Diese Toolarchitektur ist ähnlich dem TI C60x Ansatz [Instruments, 97], erlaubt darüber hinaus aber die Softwareentwicklung für eine DSP-Plattform.
Die Softwarearchitektur umfaßt damit die folgenden Teile:

- Optimierender C-Compiler:
  Dieser Compiler führt neben allen architekturunabhängigen Optimierungen die Abbildung von C-Code in Code für den Smart Assembler aus. Das Resultat ist schließlich Code, bei dem Instruktionsscheduling, Instruktionsauswahl und Registerallokation auf Streifenebene vorgenommen ist. Weiterhin ist die Allokation des Gruppenspeichers durchgeführt. Dies kann parametrisiert oder auf eine feste Gruppenbreite geschehen.

- Smart Assembler:
  Der Smart Assembler führt die Anpassung an den flexiblen Teil des DSP mittels Scheduling für die VLIW-basierte Befehlssatzarchitektur durch. Hat der Compiler parametrisierte Gruppen bestimmt, wird schließlich noch die Parallelisierung über die Streifen übernommen. Abschließend führt der Assembler die Erzeugung des Maschinencodes aus. Insbesondere für VLIW-basierte Prozessoren besteht hier allerdings das Problem der Explosion der Codegröße. Daher setzen wir ein Kompressionsverfahren namens TVLIW [Weiss and Fettweis, 1996] ein, das der Smart Assembler softwareseitig unterstützt.

4 M3 C-Compiler

Der C-Compiler basiert auf dem an der Universität Dortmund entwickelten Compilersystem LANCE*. Dieses umfaßt ein ANSI C Frontend sowie eine Bibliothek von verschiedenen maschinenunabhängigen Optimierungen, wie z.B. *constant propagation* und *dead code elimination*. Die Ausgabe von LANCE ist maschinenunabhängiger Zwischencode, welcher anschließend von einem maschinenspezifischen Codegenerator in Code für den Smart Assembler übersetzt wird.

Das Vorhandensein einer Reihe von Spezialregistern an den Ein- und Ausgängen der Funktionseinheiten macht die Anwendung von Standard-Compilertechniken problematisch. So war bei der Implementierung des Codegenerators für den M3-DSP dessen spezielle Datenpfad-Architektur (s. Abbildung 3) besonders zu berücksichtigen. Die sich hieraus ergebenden Prozessorinstruktionen werden nachfolgend aufgelistet.

Prozessorinstruktionen:

Accu = {Reg_A, Reg_B, Accu} {+, - } {Reg_A, Reg_B}
Accu = {Reg_A, Reg_B} * {Reg_B, Reg_C, Reg_D, Accu}
Accu = {Reg_A, Reg_B} * {Reg_B, Reg_C, Reg_D, Accu} {+, - }
{Reg_A, Reg_B, Accu}

Entsprechend der ersten Zeile stellt z.B. Accu = Reg_A + Reg_B eine gültige Instruktion dar. Zu beachten ist, daß die Argumente einer Instruktion in unterschiedlichen Eingangsregistern vorliegen müssen. So ist z.B. für die erste Instruktion die gleichzeitige Verwendung von Reg_A für beide Argumente der Addition nicht zulässig.

Weitere Prozessorinstruktionen sind Schreib- und Leseoperationen für Speicher (MEM) und Konstanten (INT) sowie Datentransporte zwischen Registern.

MEM = {Reg_A, Reg_B, Accu}
{Reg_A, Reg_B, Reg_C, Reg_D } = {INT, MEM}
{Reg_A, Reg_B} = {Accu}

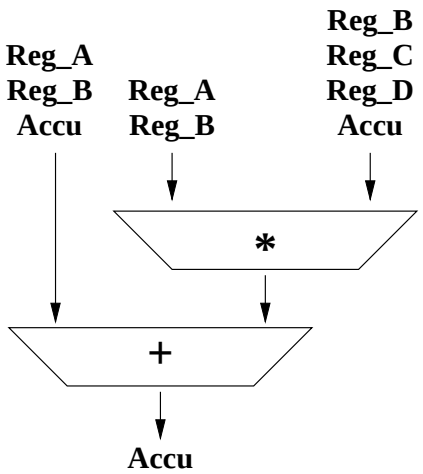

Abbildung 3: Datenpfad

Da der Speicher des M3-DSP als Gruppenspeicher realisiert ist, ist zu erwarten, daß hier die Anzahl der Speicherzugriffe einen wesentlich größeren Einfluß auf den Stromverbrauch des Prozessors besitzt, als bei Prozessoren, die mit einem

*http://LS12-www.cs.uni-dortmund.de/~leupers/

Speicherzugriff nur ein Datum adressieren. Klassische Verfahren zur Codeerzeugung basieren jedoch auf einem aus Ausdrucksbäumen bestehenden Zwischencode, bei denen gemeinsame Teilausdrücke i.d.R. im Speicher gehalten werden, was zusätzliche Speicherzugriffe und Instruktionen zur Folge hat.

Zur Ausschöpfung des gesamten Optimierungspotentials bzgl. Ausführungszeit und Stromverbrauch ist also, wie in Abbildung 4 verdeutlicht, ein Codegenerierungsverfahren erforderlich, das auf einer verallgemeinerten Graphdarstellung des Zwischencodes arbeitet.

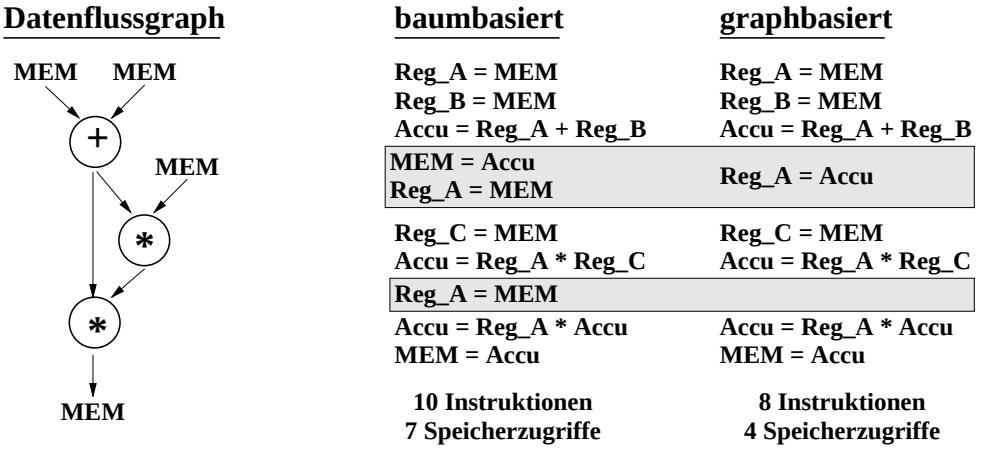

Abbildung 4: Vergleich baumbasierter und graphbasierter Verfahren

Dargestellt werden die Instruktionen, die ein baumbasiertes und ein graphbasiertes Verfahren für den M3-DSP aus dem gegebenen Datenflußgraphen generieren würde. Die Abarbeitung erfolgt bei beiden Verfahren zunächst gleich. Während beim baumbasierten Verfahren jedoch der in der dritten Instruktion berechnete gemeinsame Teilausdruck im Speicher abgelegt wird (s. Instruktion 4) und bei jeder Verwendung neu geladen werden muß (s. Instruktionen 5 und 8), kann beim graphbasierten Verfahren dieser Wert in Reg_A zwischengespeichert und wiederverwendet werden. Als Folge daraus zeigt sich, daß unter Anwendung des graphbasierten Verfahrens bereits bei diesem kleinen Beispiel eine Reduzierung der Instruktionenzahl um zwei und der Anzahl erforderlicher Speicherzugriffe um drei erreicht werden kann.

Da die Abbildung des graphbasierten Zwischencodes in optimalen Assemblercode bekannterweise ein NP-hartes Optimierungsproblem darstellt, wurde von uns ein genetischer Ansatz verfolgt, dessen Prinzip im nachfolgenden Unterabschnitt beschrieben wird. Die Effektivität dieses Verfahrens wird anhand einiger DSP-Algorithmen durch Vergleich mit einer baumbasierten Variante dieses Ansatzes demonstriert.

4.1 Realisierung des genetischen Codegenerators

Genetische Algorithmen [Holland, 1992] nehmen sich die Natur als Vorbild und lösen Optimierungsprobleme, indem sie den biologischen Evolutionsprozeß nachahmen. Dazu besteht in einem genetischen Algorithmus (GA) eine Population aus mehreren Individuen, die jedes für sich genommen jeweils eine potentielle Lösung des Optimierungsproblems darstellen. Die Repräsentation eines Individuums erfolgt mittels eines Chromosoms, das in einzelne Gene unterteilt ist. Durch die Gene werden die Variablen des Optimierungsproblems dargestellt, für die eine optimale Belegung (*Allele*) gefunden werden soll.

In unserem Ansatz werden die in der graphbasierten Zwischendarstellung des Quellprogramms vorhandenen Graphknoten[†] als Gene dargestellt. Mit der Belegung eines Gens werden alle zur Ausführung der entsprechenden Operation erforderlichen Informationen abgelegt. Dazu gehören u.a. verwendete Register, Ausführungszeitpunkt, ausgeführte Instruktion und die Bindung von Argumenten an Eingangsports der Funktionseinheit[‡].

[†] Die Knoten des Graphen entsprechen den auszuführenden Operationen, Kanten stellen Datenabhängigkeiten dar.

[‡] Dies spielt für kommutative Operationen eine wichtige Rolle, da die vorhandenen Spezialregister nicht beliebig miteinander kombiniert werden dürfen.

Der Ablauf des genetischen Algorithmus ist in Abbildung 5 skizziert. In der Initialisierungsphase werden alle Individuen der Anfangspopulation initialisiert und anschließend mittels einer Bewertungsfunktion bewertet. Die Bewertung der einzelnen Individuen ergibt sich dabei aus der Anzahl der benötigten Instruktionen. Zur Differenzierung von Individuen mit gleicher Instruktionenanzahl wird als weiteres Bewertungskriterium die Anzahl der zusätzlich erforderlichen Speicherzugriffe (*Spills*) betrachtet. Auf Basis der durchgeführten Bewertung werden dann in der Selektionsphase die Individuen ausgewählt, die ihre Gene in die nächste Generation vererben dürfen. Danach werden die Gene zweier Individuen mittels Crossover zu neuen Individuen rekombiniert und anschließend einer Mutation unterzogen. Da mit Durchführung des Crossover jedoch ungültige Lösungen generiert werden können, wird in der Mutationsphase zusätzlich eine Korrektheitsüberprüfung aktueller Genbelegungen durchgeführt. Wird z.B. festgestellt, daß zu einem bestimmten Zeitpunkt ein auf dem Gen kodiertes Register nicht verwendet werden darf, wird ebenfalls eine Mutation ausgeführt. Solange die Abbruchbedingung (hier: max. Anzahl zu simulierender Generationen) nicht erfüllt ist, dient die anschließende Bewertung wiederum als Grundlage für die Selektion (in Schritt drei). Im anderen Fall terminiert der Algorithmus, und das bisher beste Individuum wird ermittelt.

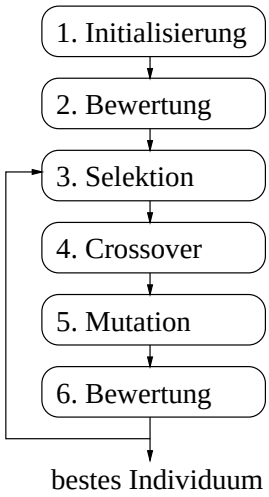

Abbildung 5: Ablauf des GA

Das der Initialisierungsphase zugrundliegende Prinzip ist wesentlich für das Verständnis dieses Verfahrens und wird deswegen im folgenden detaillierter beschrieben.

Zur Initialisierung eines Individuums wird eine Variante des *List-Scheduling* [Baker, 1974] verwendet. Während beim List-Scheduling die Operationen, die zum aktuellen Zeitpunkt ausgeführt werden können, bzgl. eines heuristischen Auswahlkriteriums nach Prioritäten geordnet und aufgrund dieser Sortierung zugewiesen werden, erfolgt hier eine probabilistische Auswahl. Der Ablauf der Initialisierungsphase wird iterativ in den folgenden Schritten durchgeführt, solange noch nicht behandelte Operationen vorhanden sind:

1. Aus der Menge aller aktuell ausführbaren Operationen wird probabilistisch die nächste auszuführende Operation ausgewählt (*Instruktionsscheduling*).
2. Für die im vorherigen Schritt ausgewählte Operation wird eine *Instruktionsauswahl* vorgenommen. An dieser Stelle ist es mittels Pattern-Matching auf einfache Weise möglich, komplexe Operationen wie z.B. MAC-Operationen zu berücksichtigen. Wird eine Operation durch mehrere Prozessorinstruktionen überdeckt, erfolgt eine probabilistische Zuordnung der Operation zu einer Prozessorinstruktion.
3. Mit Kenntnis der aktuellen Registerbelegung können nun für die ausgewählte Operation die zu verwendenden Eingangs- und Ausgangsregister bestimmt werden (*Registerallokation*). Dazu wird zunächst die Menge der zu gültigen Lösungen führenden Allele (hier: Register, Speicher) bestimmt. Analog zur Instruktionsauswahl erfolgt wiederum eine probabilistische Auswahl. Erforderliche Datentransporte oder Spills werden hierbei an geeigneter Stelle eingefügt.

Das zuvor beschriebene Prinzip zur Initialisierung der Individuen wird in leicht abgewandelter Form ebenfalls in der Mutationsphase angewendet. Hier dient es dann zur Durchführung einer Mutation und zur Korrektheitsüberprüfung aktueller Genbelegungen.

Nach Beendigung des Optimierungsprozesses wird für die beste gefundene Lösung anhand der generierten Informationen, wie Ausführungsreihenfolge der Operationen und verwendete Register, der Code in C/C++-ähnlicher Syntax ausgegeben und dient dann als Eingabe für den Smart Assembler.

4.2 Ergebnisse

Tabelle 1 stellt experimentelle Ergebnisse für einige DSP-Algorithmen einerseits für das (klassische) baumbasierte Verfahren und andererseits unter Anwendung des graphbasierten Verfahrens gegenüber. Es ergibt sich unter Anwendung der graphbasierten Variante eine Reduzierung der Instruktionenzahl um bis zu 33 % gegenüber der baumbasierten Variante. Die Anzahl der benötigten Spills konnte sogar um bis zu 64 % reduziert werden.

Der Zeitaufwand für die genetische Optimierung[§] (hier bis zu 28 s) ist höher als bei herkömmlichen Verfahren und hängt im wesentlichen von der Anzahl der zu betrachtenden Graphknoten ab. Allerdings ist die zu erzielende Codequalität

[§] Die Optimierung erfolgte hier über 200 Generationen mit einer Populationsgröße von 60 Individuen, von denen jeweils 30 unverändert in die nächste Generation übernommen wurden.

source	# Instruktionen baumbasiert	graphbasiert	%	# Spills baumbasiert	graphbasiert	%	# Graph-knoten	Rechenzeit (s) graphb. Verfahren
DCT	82	64	22	50	30	40	34	24
lattice	81	54	33	50	18	64	46	28
IIR	36	29	19	14	6	57	26	15
complex mult	15	12	20	4	2	50	12	7

Tabelle 1: Experimentelle Ergebnisse zur Codeerzeugung

bei eingebetteten Prozessoren wesentlich wichtiger einzustufen als die Übersetzungsgeschwindigkeit. Außerdem lassen sich in genetischen Algorithmen viele Vorgänge parallelisieren, da viele Berechnungen unabhängig voneinander durchgeführt werden können. So ließe sich durch eine Verteilung der Berechnungen auf mehrere Prozessoren eine drastische Verkürzung der Rechenzeit erzielen.

Da die Codegenerierung in mehreren Iterationen (Generationen) durchgeführt wird, stehen zu einem Zeitpunkt getroffene Entscheidungen immer wieder auf dem Prüfstand und setzen sich letztendlich nur dann durch, wenn diese unter Berücksichtigung aller anderen Entscheidungen zu einem guten Ergebnis führen. Auf diese Weise wird für die Teilaufgaben Instruktionsscheduling, Instruktionsauswahl und Registerallokation eine einheitliche Betrachtungsweise (*Phasenkopplung*) erreicht. Eine Erweiterung dieses Verfahren ist auf einfache Weise möglich. So kann z.B. eine Speicherlayout-Optimierung einfach in das bestehende Konzept integriert werden, indem mit jedem Gen eine weitere Information (Position der Argumente im Speicher) abgelegt wird. Hierbei ist zu erwarten, daß der zusätzlich erforderliche Rechenaufwand sehr gering ausfallen wird, da neben der Verwaltung der Variablenpositionen im Speicher lediglich zusätzlich erforderliche Konsistenzüberprüfungen anfallen. Im Gegensatz dazu wäre bei klassischen Verfahren, ein separater Optimierungsschritt notwendig. Dies würde allerdings aufgrund der großen Abhängigkeit des Speicherlayouts und der Codegenerierung ineffizienteren Code bzgl. der Anzahl von Speicherzugriffen und Instruktionen erwarten lassen.

Durch die Kombination von Compiler und Smart Assembler wird eine flexible Möglichkeit geschaffen, Code für unterschiedliche Architektur-Merkmale zu generieren. Da jedoch Abhängigkeiten zwischen den Aufgaben des Compilers und des Smart Assemblers bestehen, wird das vorhandene Optimierungspotential bislang noch nicht voll ausgenutzt. So soll in einem nächsten Schritt das Verfahren so erweitert werden, daß auch die parallelen Streifen der M3-Plattform ausgenutzt werden.

5 M3 Smart Assembler

Der Smart Assembler übernimmt nun die Anpassung auf den flexiblen Teil der M3-Plattform. Die Eingabesprache ist C/C++ basiert, die auch dem Assemblerprogrammierer erlaubt, Hochsprachenelemente wie Arrayindizierung mittels Schleifenvariablen oder neue Datentypen, die mittels *operator overloading* programmiert werden können, zu nutzen. Dazu werden Optimierungsverfahren wie Softwarepipelining [Chao et al., 1997] oder Schleifentransformationen [Weiss et al., 1998] eingesetzt (s.u.).

Ist die Anpassung an den variablen Teil beendet, kann der Smart Assembler ein Scheduling des Codes durchführen, um die Möglichkeiten der orthogonalen VLIW-Architektur nun vollständig nutzen zu können. In [Araujo et al., 1996] oder [Leupers et al., 1998] wurden Verfahren angegeben, um durch Optimierungen der Adreßrechnungen eine Steigerung der Codequalität zu erlangen. Dies wird in Kapitel 5.2 gezeigt. In Kapitel 5.3 wird schließlich gezeigt, wie diese Methoden auch auf eine VLIW-basierte Befehlssatzarchitektur angewendet werden können.

5.1 Transformationen am Beispiel Lattice FIR

Um DSP-Algorithmen auf einen DSP der M3-Plattform abzubilden, müssen zwei wesentliche Phasen durchlaufen werden:

1. Die benötigte Anzahl paralleler Streifen muß festgelegt und
2. die eigentliche Algorithmenabbildung auf genau diese Architektur durchgeführt werden.

Dadurch kann erreicht werden, daß unsere Tools nicht auf eine spezifische Architektur festgelegt sind, sondern inhärent mitskalieren können. Nutzt man den Smart Assembler Stand-Alone, kann die Anzahl der Streifen aus Phase 1 ermittelt werden. Ist die Anzahl der Streifen bekannt, kann der Smart Assembler als Teil der Toolkette in Verbindung mit dem C-Compiler zur Codeerzeugung aus Phase 2 genutzt werden.

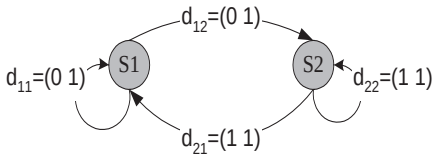

Abbildung 6: Reduzierter Abhängigkeitsgraph *(RAG) des Lattice FIR Algorithmus*

In [Weiss et al., 1998] sind Verfahren angegeben worden, um eine Standard-DSP Architektur zu erweitern und zu programmieren. Dieselben Verfahren können nun auch auf die streifenbasierte DSP-Architektur angewendet werden. Damit erreichen wir eine Softwarekompatiblität zu einer Standard-DSP Architektur wie in [U.Walter et al., 1999] angegeben.

Dies gilt für alle Algorithmen, deren Indexfunktion affine Rekurrenzgleichungen darstellen. Hierdurch kann also der *Datentransfer* unabhängig von der *Datenmanipulation* bestimmt werden. Ein Beispiel hierfür stellt der Lattice FIR Algorithmus, wie er beispielsweise in [ETSI, 1994] genutzt wird, dar:

$$
\begin{array}{l}
\textbf{for} \quad i_1 = 0 \textbf{ to } I_1, \textbf{for} \quad i_2 = 0 \textbf{ to } I_2 \\
\quad S_1 : \; y_1[i_1, i_2 + 1] = F_1(y_1[i_1, i_2], x_1[i_2 + 1], y_2[i_1 - 1, i_2]) \\
\quad S_2 : \; y_2[i_1, i_2 + 1] = F_1(y_2[i_1 - 1, i_2], x_1[i_2 + 1], y_1[i_1, i_2]) \\
\textbf{end for}, \textbf{end for}
\end{array}
$$

wobei $F_1(a, b, c) = add(a, mult(b, c))$ gilt. Für diesen Algorithmus kann ein *Reduzierter Abhängigkeitsgraph* (RAG) mit den Abhängigkeiten $d_{S_1,S_2} = (i_1, i_2)$ wie in Abbildung 6 dargestellt, abgeleitet werden.

Geht man nun von einer Architektur mit M Streifen aus, kann jeder Speicherzugriff M Elemente zur Verfügung stellen. Da der Zustand S_1 keine Abhängigkeiten in Richtung i_2 hat, können M unabhängige Streifen zu Berechnung des Zustandes S_1 genutzt werden. Damit können in Iteration $i_2 = 0$ die Gruppen

$$Y_1{}^{\text{Gr-1.1}} = y_1[m \in (0, .., M-1), i_2] \text{ und } Y_2{}^{\text{Gr-1.1}} = y_2[m \in (-1, .., M-2), i_2]$$

gelesen werden. Diese Gruppen mit Ergebnissen werden in Zustand S_1 zur Berechung einer neuen Gruppen:

$$Y_1{}^{\text{Gr-1.2}} = y_1[m \in (0, .., M-1), i_2 + 1],$$

während in Zustand S_2 schon die neue Gruppe

$$Y_2{}^{\text{Gr-1.2}} = y_2[m \in (0, .., M-1), i_2 + 1]$$

berechnet wird. In der nächsten Iteration $i_2 = 1$ werden nun die Gruppen

$$Y_1{}^{\text{Gr-2.1}} = y_1[m \in (0, .., M-1), i_2 + 1] \text{ und } Y_2{}^{\text{Gr-2.1}} = y_2[m \in (-1, .., M-2), i_2 + 1]$$

benötigt. Da hierbei gilt: $Y_1{}^{\text{Gr-2.1}} = Y_1{}^{\text{Gr-1.2}}$, können Daten im *Vektor Datatransfer Mode* gelesen werden. Allerdings sind die Gruppen $Y_2{}^{\text{Gr-2.1}}$ und $Y_2{}^{\text{Gr-1.2}}$ unterschiedlich angeordnet. Daher muß die Gruppe $Y_2{}^{\text{Gr-1.2}}$ um ein Element nach rechts geschoben werden. Dies wird durch den *Sliding Window Datatransfer Modus* erreicht. Innerhalb des RAG kann dies aus der Abhängigkeit $d_{S_1,S_2} = (1, 1)$ abgeleitet werden. Damit müssen nur zwei Gruppen gelesen werden, um alle Datenpfade beschäftigen zu können. Dies ermöglicht einen hohen Datendurchsatz, wenn die Latenz des Speicherzugriffs hinter der Berechung "versteckt" werden kann.

5.2 Scheduling

Die vorhergehenden Phasen liefern Code für eine vorgegebene Streifenanzahl M. Weiterhin wurden durch den Compiler schon Register allokiert, Ressourcen gebunden und Speicher reserviert. Offen hingegen ist noch das Scheduling des Codes über die Streifen, was insbesondere in einer VLIW-Architektur zu einer erheblichen Erhöhung der Performance führen kann.

Für die Parameter `I1 = 9, I2=160 und M=16` könnte vor dem Scheduling Zwischencode der folgenden Form vorliegen. (Hierbei bezeichnet `Rx(SIMD)` eine parallele Operation auf Register Rx, `SWDT` eine Sliding Window Datentransfermode):

Tabelle 2: Benchmarks für den M^3-DSP vs. TI's C6x, HiPAR, und Butterfly DSP

Algorithms	M^3 DSP @100 Mhz	TI's C60 @ 200 Mhz	HiPAR @ 100 Mhz	Butterfly DSP @ 50 Mhz
1024 complex point FFT (Radix-2)	2200 cycles/22 μs	20815 cylces/104 μs	42 μs	54 μs
complex FIR, 32 coeff.,100 samples	1204 cycles/12 μs	6410 cycles/32 μs	N.A.	N.A.
Lattice FIR, 8 coeff.,128 samples	268 cycles/3 μs	1546 cycles/8 μs	N.A.	N.A.
BCH code(216,124,25)	244 cycles/2.4 μs	N.A.	N.A.	N.A.

```
    for i2 = 0 to 8 {
        Lesen: Gruppe x1(Koffizienten) aus Speicher nach R4(SIMD));
        for i1 = 159 to 0 STEP 16 {
            I1: Lesen: Gruppe y1(i1) aus Speicher nach R1(SIMD);
            I2: Lesen: Gruppe y2(i1) aus Speicher nach R2(SIMD);
            I3: Ausrichten: SWDT(R1,R3);
            I4: Rechne: Accu(SIMD) = MAC( R1(SIMD) + R4 * R2(SIMD));
            I5: Speichere: Accu(SIMD) als Gruppe y1(i1) in Speicher;
            I6: Rechne: Accu(SIMD) = MAC( R2(SIMD) + R4 * R1(SIMD));
            I7: Speichere: Accu(SIMD) als Gruppe y2(i1) in Speicher;
            I8: Verschiebe: R3(SIMD) = R1(SIMD);
            }
    }
```

Auf diesen Zwischencode können nun Optimierungsverfahren wie *loop unrolling* oder *software pipelining* angewendet werden, um die vorhandenen Hardwareressourcen optimal auszunutzen. In diesem Schritt wird die Adreßberechnung jedoch noch nicht durchgeführt.

```
    Schleifen-Prolog
    for i1 = 159 to 0 STEP 16 {
        V1: I1(y1(i+1)) || I4 || I7(y2(i)) || I8;
        V2: I2(y2(i+1)) || I3 || I5(y1(i)) || I6;
        }
    Schleifen-Epilog
}
```

Die Adreßberechung kann nun auf diesem geschedulten Code aufsetzen und liefert durch die Ausnutzung von Postmodifikations-Addressierung und die Verwendung der Pointerregister `P1` und `P2`:

```
    Schleifen-Prolog
    for i1 = 159 to 0 STEP 16 {
        V1: I1(*P1++) || I4 || I7(*P2--) || I8;
        V2: I2(*P2++) || I3 || I5(*P1--) || I6;
        }
    Schleifen-Epilog
```

Durch die Adreßrechnung auf dem geschedulten Code sind für die Adreßpointer keine zusätzlichen Pipelineregister in der DSP-Hardware nötig. Das Ergebnis dieser Phase ist nun ein Programmcode mit performance-optimierten VLIW-Worten. Für den Einsatz in einer SoC-Domäne ist die hieraus resultierende Codegröße jedoch nicht tolerierbar.

5.3 Backend

Innerhalb der Backend-Phase durchläuft der geschedulte Code schließlich noch eine Codekomprimierungsphase, die beispielsweise ein 128 Bit breites VLIW-Wort auf ein 32 Bit breites TVLIW-Wort reduziert ([Weiss and Fettweis, 1996]). In Tabelle 2 sind Benchmarks für den M3-DSP mit 16 Streifen angegeben.

6 Zusammenfassung

Für die DSPs der nächsten Generation wird jetzt begonnen, nicht einen festen Kern, sondern eine Plattform-Lösung anzubieten. Ihre Akzeptanz wird neben der Leistungsfähigkeit der Architekturen eine geeignete Toolumgebung ausmachen. In diesem Beitrag haben wir die Toolkette für die $M3$-Plattform bestehend aus einem C-Compiler und einem Smart Assembler vorgestellt.

Der C-Compiler erzeugt optimierten Code für den Teil der $M3$-Plattform, der unabhängig von dem speziellen Zuschnitt immer gleich bleibt. Damit kann die Codeerzeugung auf diese feste Architektur erfolgen. Die Optimierung auf den variablen Teil der Plattform wird dagegen von einem Smart Assembler übernommen, der neben der eigentlichen Objektcodeerzeugung auch das Scheduling des Codes über die Streifen übernimmt. Die Leistungsfähigkeit dieses Ansatzes wurde anhand von verschiedenen DSP-Algorithmen gezeigt.

7 Literatur

Araujo, G. et al. (1996). Instruction set design for optimizations for address compution in DSP architectures. In *Proc. of Int.Sym. System Synthesis 96*. IEEE.

Baker, K. R. (1974). *Introduction to Sequencing and Scheduling*. Wiley, New York.

Chao, L, LaPaugh, A., and Sha, E. (1997). Rotation scheduling: A loop pipelining algorithm. *IEEE Trans. Computer-Aided Design*, 16(3):229–239.

Drescher, W., Mennenga, M., and Fettweis, G. (1998). An architectural study of a digital signal processor for block codes. In *Proc. of ICASSP '98*, volume 5, pages 3129–3133, Seattle, WA, USA.

ETSI (1994). *Recommendation: Full rate speech transcoding (GSM 6.10)*. ETSI.

Fettweis, G., M.Weiss, W.Drescher, U.Walther, F.Engel, and S.Kobayashi (1998). Breaking new grounds over 3000 MOPS: A broadband mobile multimedia modem DSP. In *Proc. of DSP'98*, pages 31–34, Muenchen, Germany.

Hansen, Craig (1996). MicroUnity's mediaprocessor architecture. *IEEE Micro*, 16(4):34–40.

Holland, J. H. (1992). *Adaption in Natural and Artificial Systems*. MIT Press.

Instruments, Texas (97). *TMS320C60 Instruction Set Manual*.

Kim, Kyosun, Karri, Ramesh, and Potkonjak, Miodrag (1997). Synthesis of application specific programmable processors. In *Proc. of DAC '97*, pages 353–358.

Leupers, Rainer, Basu, Anupam, and Marwedel, Peter (1998). Optimized array index computation in dsp programs. In *ASP-DAC '98*, Yokohama/Japan.

Trenas, M.A., Lopez, Juan, and L-Zapata, Emilio (1998). A memory system supporting the efficient SIMD computation of the two dimensional DWT. In *Proc. of ICASSP '98*.

U.Walter, F.Tischer, and G.P.Fettweis (1999). New DSPs for next generation mobile communications. In *IEEE Globecom '99*.

Weiss, M., Engel, F., and Fettweis, G. P. (1999). A New Scalable DSP Architecture for System on Chip (SoC) Domains. In *ICASSP'99*, Phoenix, Arizona.

Weiss, M. H. et al. (1998). Designing performance enhanced digital signal processors using loop transformations. In *Proc. of PACT 98-Workshop on Reconfigurable Computing*, pages 90–4.

Weiss, Matthias H. and Fettweis, Gerhard P. (1996). Dynamic codewidth reduction for VLIW instruction set architectures in digital signal processors. In *3rd. Int. Workshop in Signal and Image Processing (IWSIP '96)*, pages 517–520. IEEE.

Wittenburg, J.P. et al. (1997). HiPAR-DSP: A parallel VLIW RISC processor for real time image processing applications. In *Proc. of ICA3PP '97*, pages 155–162.