

UPHA

- eine Umgebung zur Programmierung hocheffizienter Anwendungen
für Digitale Signalprozessoren

Peter Bachmann

BTU Cottbus, Institut für Informatik

Lehrstuhl Programmiersprachen und Compilerbau

Postfach 10 13 44, D-03013 Cottbus

e-mail: pb@informatik.tu-cottbus.de

Tel.:(0355)693887, Fax: (0355)693830

Zusammenfassung

Mit der Programmierumgebung UPHA soll die Brücke zwischen einer maximalen Programmierunterstützung und der Sicherung einer hohen Effizienz der entwickelten Programme gebaut werden. Dazu kann die Programmentwicklung auf zwei Ebenen erfolgen. Die untere Ebene ist der jeweiligen Prozessorarchitektur stark angepaßt. Der Programmierer hat sich auf das Niveau der Maschinensprache zu begeben, ist aber von einigen schwer zu überschauenden Problemen, wie der Parallelisierung von Befehlssequenzen oder der Registerzuordnung, entlastet oder erhält dabei Unterstützung. Die obere Ebene ist problemorientiert und von der Prozessorarchitektur unabhängig. In beiden Ebenen erfolgt die Programmentwicklung über eine graphische Schnittstelle. Statt Programmtext zu erzeugen, füllt und manipuliert der Programmierer eine Datenbasis, aus der bei Bedarf Programmtext gewonnen werden kann. Zwischen den verschiedenen Ebenen und Repräsentationen eines Programmes kann automatisch transformiert werden.

1 Anliegen

Eine wesentliche Klasse von Digitalen Signalprozessoren sind für spezielle Anwendungen entwickelt, haben eine auf diese Anwendungen zugeschnittene Architektur, um diese Anwendungen auf einer kleinen Chipfläche mit geringem Energiebedarf und hocheffizient, d.h. einem Kompromiß von wenig Speicherbedarf und geringer Laufzeit, zu realisieren. Daraus folgt, daß bei der Programmierung der benötigten Algorithmen keine Effizienzverluste hinnehmbar sind. Die gängige industrielle Praxis ist daher die Programmierung im Assembler durch erfahrene Programmierer. Daneben entwickelt sich verstärkt der Wunsch nach Nutzung höherer Programmiersprachen. Favorisiert wird C. Prinzipiell ist es natürlich kein Problem, C-Compiler für DSP's bereitzustellen und tatsächlich existieren bereits eine Reihe solcher Compiler (Siehe z.B. [1]). Im Rahmen eines Praktikums bei der Thesys GmbH Erfurt wurde in Kooperation mit dem Lehrstuhl 12 des Instituts für Informatik der Universität Dortmund ein solcher Compiler auch für den GEPARD ([4]) entwickelt. Das schwer beherrschbare Defizit besteht in der mangelnden Effizienz der Zielprogramme. Oft wird deshalb bei der Nutzung von C von Hand "nachoptimiert", indem die Assemblerprogramme modifiziert werden (extrem hohe Fehlergefahr!) oder Funktionen vollständig durch von Hand programmierten Assemblercode ersetzt werden (Fehlergefahr geringer, aber vorhanden).

In vielen intensiven Untersuchungen bemüht man sich durchaus erfolgreich, die im Compilerbau vorhandenen Optimierungstechniken bei der Codeerzeugung auf die irregulären DSP-Architekturen anzupassen bzw. neue Techniken dafür zu entwickeln (Eine gute Übersicht findet man in [2]). Dabei ist immer die Hürde zu überwinden, daß die Sprache C auf andere Architekturen ausgerichtet ist. Die Nutzung höherer Programmiersprachen und die Compiler-technik hat ja unter anderem zur Entwicklung von Compiler-"freundlichen" Architekturen, zum Beispiel mit stapelnden Akkumulatoren geführt.

Hier wird der (natürlich auch schon bekannte, siehe z.B. [5]) Weg beschritten, angepasste Programmierinstrumente in Form von integrierten Programmierumgebungen zu schaffen. Der Hauptunterschied zu den bekannten Ansätzen besteht in einem bottom-up Vorgehen, indem

- auf einer unteren maschinennahen, aber vom Assembler abgehobenen, Ebene umfangreiche Programmierunterstützung angeboten wird, die eine hocheffiziente und trotzdem relativ sichere Programmierung erlaubt und
- auf einer oberen problemnahen Ebene, etwa im C-Niveau, entsprechende Programmierwerkzeuge bereitstellen, die die Mechanismen der unteren Ebene nutzen und auf ihnen aufbauen.

Da die Erstellung hocheffizienter Programme höchste Priorität hat, wird sich dieses Ziel einschränkend auf die in den zwei Ebenen verfügbaren Programmier-Techniken auswirken. Trotzdem aber soll im angestrebten System der Forderung nach vollem C als Programmiersprache entsprochen werden. Damit werden Möglichkeiten des Prototyping und der Nachnutzung vorhandener Quellen eröffnet.

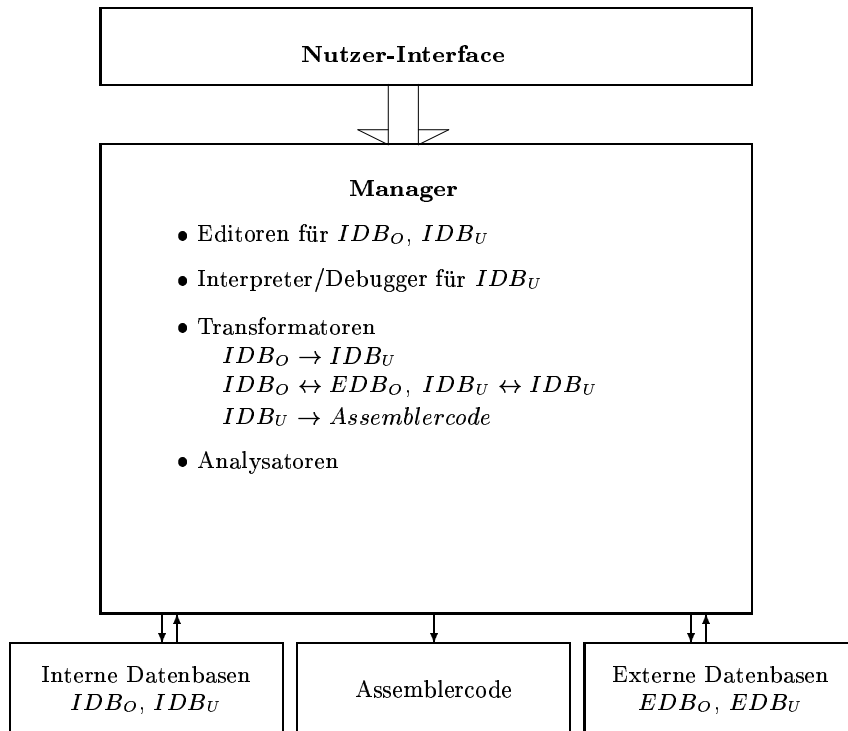
2 Struktur von UPHA

2.1 Allgemeines

UPHA wird als eine integrierte Programmierumgebung aufgebaut. In jeder Ebene wird der Programmierer über ein **Nutzer-Interface** bei der Programmentwicklung geleitet. Die Programmsynthese erfolgt mittels Editoren, die über das Interface genutzt werden. Nur wird nicht Text erzeugt, sondern es wird eine **interne Datenbasis (IDB)** aufgebaut, die das entstehende Programm repräsentiert. Der Nutzer kennt die Struktur dieser Datenbasis nicht, sondern kann sich mit Hilfe der Editoren über den Inhalt informieren, den Inhalt füllen und modifizieren. Weiter stehen ihm unterschiedliche **Analysewerkzeuge** zur Verfügung, die Informationen über das entstehende Programm (deklarierte Objekte wie Variable und Funktionen, verfügbare Typen, benutzte und freie Variable usw.) in jeder Phase, auch zum unfertigen Programmfragment, verfügbar machen. Es existiert eine automatische Transformation $IDB_O \rightarrow IDB_U$ der internen Datenbasis aus der oberen in die untere Ebene. Es ist damit möglich, ein Programm in der oberen Ebene zu entwickeln, in die untere Ebene zu transformieren und dann dort weiter zu bearbeiten, wenn zum Beispiel die erforderliche Effizienz durch die automatische Transformation nicht erreicht wurde. Es kann aber auch direkt in der unteren Ebene entwickelt werden.

In jeder Ebene wird auch eine **externe Datenbasis (EDB_O, EDB_U)** in Textform geführt. Zwischen den internen und externen Datenbasen existieren automatische Transformationen in beiden Richtungen. Die externen Datenbasen haben den hauptsächlichen Zweck, als druckbare und lesbare Dokumentation zu dienen. Da sie in Textform vorliegen, lassen sie sich natürlich auch mit einem beliebigen Editor bearbeiten. Das ist aber nicht das Anliegen, da der Nutzer dann ja syntaktische Regeln beachten müsste. Von solchen Regeln ist er bei Nutzung des Interface völlig entlastet. Im Gegenteil, er bekommt kontextabhängig angezeigt, welche Programmiermöglichkeiten ihm an der entsprechenden Stelle zur Verfügung stehen.

Im Sinne der klassischen Compilertechnik realisiert die Transformation $EDB_O \rightarrow IDB_O$ ein Frontend (Parsing mit Generierung einer Zwischensprache) und die beiden Transformationen $IDB_O \rightarrow IDB_U \rightarrow Assemblercode$ ein Backend (Optimierung sowie Codeerzeugung). Die (wesentlichen) Erweiterungen bestehen darin, dass dem Programmierer Mittel zur Information über und Manipulation der als Zwischensprache fungierenden IDB_U verfügbar sind sowie Rücktransformationen $IDB \rightarrow EDB$ existieren. Die Hintereinanderausführung von $EDB \rightarrow IDB \rightarrow EDB$ ist zwar Bedeutungs-erhaltend, wird aber im allgemeinen zur syntaktischen Veränderung des Quellcodes, zum Beispiel zu typographischen Veränderungen, führen. Da hauptsächliche Zweck der externen Datenbasen eine lesbare und druckbare Dokumentation ist, ist die Transformation $EDB \rightarrow IDB$ nur ein Nebenprodukt, das insbesondere auf der unteren Ebene kaum Bedeutung besitzt.



2.2 Die untere Ebene

Bei der Konzeption der unteren Ebene ist die Effizienz der erzeugten Programme absolute Zielstellung, der alle anderen Teilziele wie schnelles Prototyping oder Verifizierbarkeit untergeordnet sind.

Der Programmierer wird mit den Aufgaben belastet, die bei einer Automatisierung Effizienzverlust mit sich bringen können. Das sind zum Beispiel

- die Planung der Zuordnung von Datenbereichen (Variablen) auf verschiedene Datenspeicher,
- die Indexrechnung bei speziellen Feldern,
- die Verwendung von Registern für globale Aufgaben wie Iterationszähler, Parameterübergabe, Stapelverwaltung (falls vorhanden), usw.,
- der Einsatz spezieller Instruktionen für arithmetische Operationen.

Auf der unteren Ebene muß der Programmierer also mit der Architektur des DSP vertraut sein, insbesondere mit den arithmetischen Operationen, den Registern und deren Verwendung. Er hat keine allgemeinen arithmetischen Operationen wie in höheren Programmiersprachen verfügbar, und findet kein allgemeines Typkonzept vor. Er muß in den Mitteln des DSP denken und arbeiten.

Dagegen ist eine Entlastung des Programmierers in den Aufgaben vorgenommen, die gegenwärtig bereits gut automatisierbar sind und dabei zu optimalen Ergebnissen führen. Das sind

- die Umsetzung von allgemeinen Steuerstrukturen wie Blöcke, Schleifen, Verzweigungen, Unterprogramm-Behandlung,
- Adressberechnungen durch Bereitstellung von mnemonischen Adressen für Variable,
- die Registerzuordnung und
- die Parallelisierung der Befehle sowie die Ablaufplanung.

Für die letzten beiden Punkte existieren optimale Algorithmen für Basisblöcke, die aber nur mit grossem Aufwand auf globalere Programmfragmente erweiterbar sind. Bei der Registerverwendung kann sich der Programmierer auf konkrete Register festlegen (zum Beispiel für Parameterübergabe) oder er verwendet symbolische Register, denen vom Allokator des UPHA automatisch konkrete Register zugeordnet werden.

Zum Teil ist wegen der irregulären Architektur bei den DSP-Befehlen die Registerverwendung stark eingeschränkt, zum Teil auf spezielle Register gebunden.

In der gegenwärtigen Version von UPHA wurden folgende Konzeption zur Struktur der internen Datenbasis auf der unteren Ebene realisiert:

- **Anweisungen** sind Basisblöcke, Blöcke, Iterationen, Verteiler, Unterprogramm-Rufe, Aussprünge und Rücksprünge.
- **Basisblöcke** sind azyklische Graphen, deren Knoten atomare Instruktionen des Prozessors sind und deren Kanten beschreiben, wie Ergebnisse aus dem jeweiligen Knoten in andere Knoten einfließen. Jede Kante repräsentiert somit ein zunächst virtuelles Register aus einer gewissen Registerklasse, dem bei der Allokation (siehe unten) ein reales Register zugeordnet wird.
- **Blöcke** (block) sind Sequenzen von Anweisungen, die in der entsprechenden Reihenfolge abgearbeitet werden.
- **Iterationen** (loop) sind Blöcke, die entweder unbeschränkt oder solange durchlaufen werden, bis ein angegebene Register leer ist.
- **Verteiler** (switch) sind Sequenzen von Anweisungen, in die (als einzige) in Abhängigkeit eines angegebenen Registerinhalts eingesprungen wird.
- **Unterprogramm-Rufe** (routine-call) aktivieren ein Unterprogramm und erlauben, hinter die Aktivierung zurückzukehren.
- **Aussprünge** (break) erlauben einen Block, eine Iteration oder einen Verteiler in Abhängigkeit eines Registerinhalts vor dessen Ende zu verlassen und hinter die entsprechende Struktur zu verzweigen. Es ist möglich, einen Ausprung über mehrere Strukturen hinweg zu veranlassen.
- **Rücksprünge** (return) verzweigen aus einem Unterprogramm hinter die Aufrufstelle zurück.

Die Struktur ist relativ arm, erlaubt aber (man kann das leicht nachprüfen) alle in Programmiersprachen gängigen Steuerstrukturen nachzubilden. Durch die disziplinierte Verwendung von Sprüngen wird erstens eine gute Strukturierung des Programms erreicht, zweitens wird die Analyse des Programms erleichtert und drittens kann eine effiziente Abbildung in die Maschinsprache gefunden werden.

Für die **Registerzuordnung** im Basisblock wurde die folgende algorithmische Idee verfolgt:

Vorbereitung: Alle Kanten des azyklischen Graphen werden mit virtuellen Registernamen markiert. Jeder solcher virtueller Registername N symbolisiert eine Menge R_N von aktuellen Registern. Damit wird der irregulären Registerstruktur des GEPARD Rechnung getragen. Fast alle Operationen verlangen für Operanden und Resultate spezielle Arten von Registern. Im Spezialfall kann die einem virtuellen Registernamen zugeordnete Menge nur aus einem Register bestehen (zum Beispiel dem P-Register im GEPARD). Die Zuordnung von virtuellen Registernamen zu den Kanten erfolgt wahlweise automatisch über die Art des Operationsknotens, von dem Kanten abgehen bzw. zu dem Kanten münden. Sie kann aber vom Nutzer eingeschränkt werden. Bei der Zuordnung von aktuellen Registern zu virtuellen Registernamen wird berücksichtigt, dass gewisse Register für globale Aufgaben reserviert sind.

Schritt 1: Über den Kanten des Graphen wird eine Verträglichkeitsrelation gebildet. Zwei Kanten sind verträglich, falls

- sie im gleichen Knoten münden (dann beziehen sie sich aufs gleiche Resultat) oder
- sie auf einem gemeinsamen Pfad im Graphen liegen (dann besteht eine Abarbeitungsreihenfolge zwischen beiden).

Schritt 2: Es wird eine Überdeckung der Kantenmenge mit einer minimalen Menge \mathcal{C} von Verträglichkeitsklassen und eine injektive Funktion $\varphi : \mathcal{C} \rightarrow R$ von \mathcal{C} in die Menge R der verfügbaren aktuellen Register so bestimmt, daß für jedes $C \in \mathcal{C} : \varphi(C) \in R_C$ gilt, wobei

$$R_C = \bigcap \{ R_N \mid N \text{ ist virtueller Registername einer Kante aus } C \}.$$

Wenn eine solche Registerzuordnung durch die Funktion φ nicht gefunden werden kann, so geht man über zu

Schritt 3: Es wird durch das Einfügen von Hilfskanten und das Zwischenspeichern von Registerinhalten in Hilfsvariable (beides vergrößert die Verträglichkeitsrelation) versucht, verschiedene Verträglichkeitsklassen zu verschmelzen. Das Verschmelzen von zwei Verträglichkeitsklassen C und C' ist dann sinnvoll, wenn $R_C \cap R_{C'} \neq \emptyset$. Dann kann für beide ein Register benutzt werden.

Insbesondere im Schritt 2 entsteht ein kombinatorisches Problem, dessen optimale Lösung NP-vollständige Algorithmen verlangt. Gegenwärtig werden hier heuristische Algorithmen eingesetzt, die auf einer relativ "regulären" Struktur der Verträglichkeitsklassen basieren. Dies kann auch günstig im Schritt 3 eingesetzt werden. Auf Details muß aus Platzgründen verzichtet werden.

Der Vorteil des skizzierten Vorgehens besteht darin, daß unter der gegebenen Nebenbedingung der aktuell verfügbaren Register ein maximal möglicher Parallelitätsgrad erhalten wird. Dies wird bei der **Ablaufplanung** genutzt, der folgende Idee zugrunde liegt:

Die Knoten im Basisblock sind entsprechend ihrer Kantenbeziehung partiell geordnet. Alle minimalen Elemente in dieser Ordnung können als erste parallel abgearbeitet werden. Da im allgemeinen, bedingt durch die irreguläre Architektur von DSP's, nicht beliebige Operationen parallelisierbar sind, existieren zusätzliche Einschränkungen. Abstrahiert können diese Einschränkungen als eine Menge \mathcal{E} von Untermengen von atomaren Instruktionen modelliert werden. Es gilt $E \in \mathcal{E}$, falls alle Instruktionen aus E parallelisierbar sind. Bei der Ablaufplanung startet man nun mit der Menge M aller minimalen Knoten und bildet $\mathcal{K} = \{M \cap E \mid E \in \mathcal{E}\}$. Für $K \in \mathcal{K}$ ist K ein Kandidat für ein zu bildendes Instruktionswort, in dem alle atomaren Instruktionen aus K zusammengefaßt sind und deshalb parallel bearbeitet werden können. Zu jedem solchen Kandidaten ist nun für die Restmenge der Knoten ein optimaler Ablaufplan nach gleichem Prinzip rekursiv zu erstellen und dann der Kandidat mit dem günstigsten Ablaufplan auszuwählen. Dieses Verfahren kann man dadurch effizienter gestalten, ohne daß der kürzeste Ablaufplan verloren geht, indem als Kandidaten aus der Menge \mathcal{K} nur die maximalen Mengen betrachtet werden, für die also keine echte Obermenge in \mathcal{K} enthalten ist. Schränkt man sich weiter auf Kandidatenmengen mit maximaler Kardinalität ein, so wird der Algorithmus linear und man erhält im schlechtesten Fall einen doppelt so langen Ablaufplan wie der kürzeste.

Für die Registerzuordnung und Ablaufplanung sind globale Informationen zum Datenfluss erforderlich, die über **Analysatoren** bereitgestellt werden. Solche Informationen sind natürlich auch für den Programmierer nützlich und können über das Interface erstellt und abgefragt werden. Dazu gehören:

- korrekte Initialisierung von Variablen bzw. Registern,
- belegte und freie Variable bzw. Register,
- Pfade, auf denen Variable bzw. Register nicht verändert werden.

Es ist nicht vorgesehen, auf der unteren Ebene automatisch Transformationen der internen Datenbasis durchzuführen, um Optimierungen zu ermöglichen wie

- Elimination von nicht benutzten Operationen (death code elimination),
- Elimination doppelter Berechnungen (common subexpressions),
- Herausziehen von schleifeninvarianten Berechnungen (move of loop-invariant computations) oder
- Umwandlung von Multiplikationen mit induktiven Variablen in Schleifen (strength reduction).

Es ist das Prinzip gewahrt, dass der Programmierer die interne Datenbasis immer so vorfindet, wie er diese entwickelt hat. Trotzdem sollen die Analysatoren aber Situationen aufdecken, die zu solchen Optimierungen Anlass geben und dies dem Programmierer mitteilen.

Im Zusammenwirken mit der Transformation $IDB_U \rightarrow \text{Assemblercode}$ erstellen Analysatoren Aussagen zur Programmlänge und Programmlaufzeit pro Programmfragment bzw. das gesamte Programm. Für letzteres werden erforderliche Informationen über die zu erwartende Dimension der zu bearbeitenden Daten vom Nutzer abgefragt. Solche Aussagen sind bei der Implementierung insbesondere dann wichtig, wenn die DSP-Cores für Echtzeittests noch nicht verfügbar sind.

Ein **Interpreter** kann jede Instanz von IDB_U auswerten. Damit kann die korrekte Funktionalität des entwickelten Programms geprüft werden. Allerdings sind damit keine Aussagen zum Zeitverhalten verbunden. Wie es bei Debuggern üblich ist, kann mittels Interpreter auch eine schrittweise Abarbeitung erfolgen oder es kann bei gesetzten Unterbrechungspunkten gehalten werden. Die Ansicht und Modifikation interner Zustände (Speicher, Register) muss möglich sein.

2.3 Die obere Ebene

Die obere Ebene ist die geeignete Plattform für das Prototyping. Sie erlaubt die problemorientierte Formulierung der Algorithmen. Auch auf dieser Ebene ist ein Interpreter/Debugger verfügbar. Dabei ist es offen, ob dieser nach einer Transformation $IDB_O \rightarrow IDB_U$ den Interpreter der unteren Ebene nutzt oder

eigenständig die interne Datenbasis der oberen Ebene auswertet. Entsprechendes gilt für die Analytoren, wobei bei der Transformation in die untere Ebene globale Optimierungen ausgeführt werden. Das bedeutet, dass eine transformierte Instanz der internen Datenbasis auf der unteren Ebene nicht notwendig die Struktur des Urbildes nur verfeinert. Rückbezüge sind deshalb im allgemeinen nur bedingt möglich.

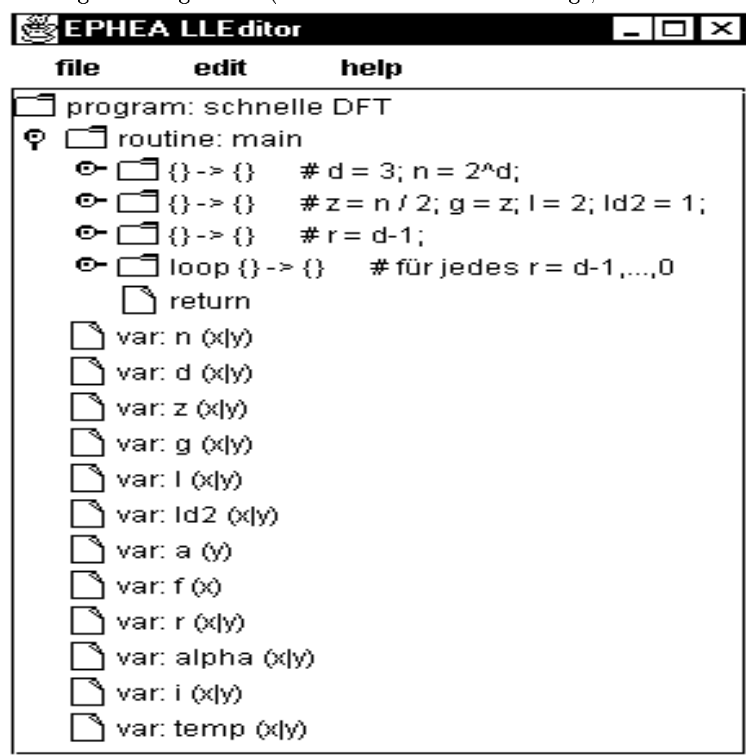
Die obere Ebene kann verschiedene Ausprägungen besitzen. Primär wird sie so konzipiert, dass eine effiziente Transformation in die untere Ebene möglich ist. Das wird zu Restriktionen bei der Programmgestaltung führen. Als externe Datenbasis wird stark reglementierter C-Quelltext verwendet (als C- bezeichnet), angereichert mit speziellen Pragmas, die Zusatzinformationen für eine gute Transformation in die untere Ebene liefern. Um andererseits vorhandene Programme, aus Standardwerkzeugen (etwa MATLAB) generierte Programme oder als Prototypen entwickelte Programme nutzen zu können, wird auch ANSI-C als Programmiersprache zugelassen. Das kann zwar zu erheblichen Effizienzverlusten führen, die aber nach Transformation der Programme in die untere Ebene mit den dort verfügbaren Instrumenten nachgebessert werden können.

In dem Maße, wie es gelingt, Optimierungsverfahren zu entwickeln, die eine automatische Erzeugung hocheffizienten Codes aus C-Quelltext auch für digitale Signalprozessoren ermöglichen, wird die untere Ebene als eigenständige Programmierplattform an Bedeutung verlieren. Sie spielt aber weiterhin die Rolle eines Backend im Prozess der Compilierung.

2.4 Stand

Da das Hauptanliegen eine Programmierumgebung für hocheffiziente Anwendungen ist, wurden die Arbeiten mit der unteren Ebene begonnen. Die interne Datenbasis ist als abstrakte Datenstruktur konzipiert und als JAVA-Klasse implementiert. Demzufolge werden auch andere Implementierungen in JAVA durchgeführt.

Arbeitsfähig ist die erste Version des Nutzer-Interface und der Editor für die interne Datenbasis [3]. Einen Eindruck von der Oberfläche kann man sich in den nachfolgenden drei Bildern verschaffen. Dabei sind Ausschnitte unterschiedlichen Detaillierungsgrades eines Programms zur schnellen diskreten Fouriertransformation angezeigt. Im ersten Bild ist nur die Grundstruktur der Hauptroutine einschließlich der deklarierten Variablen zu erkennen. Hinter dem Nummerzeichen ist über Kommentare die Bedeutung der entsprechenden Anweisung angegeben. Zusätzlich sind Informationen über die eingehenden und ausgehenden Register vorgesehen (in diesem Bild vernachlässigt, da noch nicht voll implementiert).



Im zweiten Bild sind die Iterationen aufgeblättert.

```

EPHEA LLE editor
file edit help
program: schnelle DFT
  routine: main
    {}->{} # d = 3; n = 2^d;
    {}->{} # z = n / 2; g = z; l = 2; ld2 = 1;
    {}->{} # r = d-1;
    loop {}->{} # für jedes r = d-1,...,0
      {}->{} # if(r < 0) break;
      {}->{} # alpha = 0;
      loop {}->{} # für jedes alpha = 0,...,z-1 (z=2^r)
        {}->{} # if(alpha>=z) break;
        {}->{} {10, 11, 12, 13} # i0=@f[alpha*=0]; i2=@f[alpha*l+ld2=ld2]; i1=i3=1;
        {}->{} {14, 15, 16, 17} # i4=@a[*z=0]; i6=@a[*z+g=g]; i5=i7=z;
        {}->{} # i = 0;
        loop {10, 11, 12, 13, 14, 15, 16, 17}->{} # für jedes i = 0,...,ld2-1
          {}->{} # if(i>=ld2) break;
          {}->{} {10, 12, 14, 16}->{} # calculation
          {}->{} {10, 11, 12, 13, 14, 15, 16, 17}->{10, 12, 14, 16} # modify indices: i0* i2* i4* i6*
          {}->{} # ++i;
          {}->{} # ++alpha;
        {}->{} # z = z/2; ld2 = l; l = l*2;
      {}->{} # --r;
    return
  
```

Schließlich kann man im dritten Bild den Basisblock sehen, der den Körper der innersten Iteration darstellt. Allerdings wurde hierbei noch eine sequentielle Darstellung gewählt. Ein graphischer Editor für Basisblöcke ist in Arbeit, aber noch nicht voll funktionsfähig.

```

EPHEA LLE editor
file edit help
  {}->{} {14, 15, 16, 17} # i4=@a[*z=0]; i6=@a[*z+g=g]; i5=i7=z;
  {}->{} # i = 0;
  loop {10, 11, 12, 13, 14, 15, 16, 17}->{} # für jedes i = 0,...,ld2-1
    {}->{} # if(i>=ld2) break;
    loop {10, 12, 14, 16}->{} # calculation
      LOAD-X {10} -> {A1}
      LOAD-X {12} -> {C}
      LOAD-Y {14} -> {A2}
      LOAD-Y {16} -> {A3}
      MUL {C, A2} -> {P, PL, PX2, PH, PH1}
      MV {PL} -> {A2}
      ADD {A1, A2} -> {A2}
      MUL {C, A3} -> {P, PL, PX2, PH, PH1}
      STORE-X {A2, 10} -> {}
      MV {PL} -> {A3}
      ADD {A1, A3} -> {A3}
      STORE-X {A3, 12} -> {}
    {}->{} {10, 11, 12, 13, 14, 15, 16, 17}->{10, 12, 14, 16} # modify indices: i0* i2* i4* i6*
    {}->{} # ++i;
    {}->{} # ++alpha;
  {}->{} # z = z/2; ld2 = l; l = l*2;
  
```

Das Nutzerinterface ist so gestaltet, daß der Programmierer bei der Programmentwicklung weitgehende Unterstützung erhält. So werden ihm die verfügbaren Steuerstrukturen und atomaren Instruktionen angeboten, bei einer notwendigen Bezugnahme auf Variable werden diese angezeigt usw.

Kurz vor dem Abschluß stehen die Arbeiten am Interpreter für die interne Datenstruktur der unteren Ebene. Damit kann dann der programmierte Algorithmus getestet werden, ohne daß der zugehörige DSP verfügbar ist. Noch zu entwickeln ist die Transformation $IDB_U \rightarrow Assemblercode$.

Die Idee zu einer solchen Programmierumgebung entstand aus dem Wunsch der Thesys GmbH Erfurt, einen C-Compiler für den GEPARD verfügbar zu haben. In seinem Betriebspraktikum hat dort der Student M. Schölzel der BTU Cottbus das Backend eines C-Compilers implementiert, dessen Frontend am Lehrstuhl 12 des Instituts für Informatik an der Universität Dortmund entwickelt wurde. Gegenwärtig befindet sich dieser Compiler in der Testphase.

2.5 Ausbau

Schrittweise wird die hier vorgestellte Konzeption implementiert. Da dies gegenwärtig nur durch studentische Studien- und Diplomarbeiten bzw. über studentische wissenschaftliche Hilfskräfte erfolgen kann, ist der Arbeitsfortschritt bescheiden.

Ein weiterer Ausbau der Konzeption ist in den beiden Richtungen

- Variabilität bezüglich der DSP-Cores und
- durchgängige Entwurfs- und Implementierungsplattform für eingebettete Systeme.

angedacht.

In der ersten Richtung sollte durch Spezifikation des DSP-Cores eine weitgehend automatische Anpassung des UPHA an den entsprechenden Core erfolgen. Dazu müssen sowohl das Interface als auch die Komponenten des Managers adaptierbar sein. Diese Richtung entspricht der Entwicklung retargierbarer Compiler. Es liegt nahe, die Variabilität dahingehend auszunutzen, dass Aussagen zur Auswahl für die spezielle Anwendung geeigneter Cores, mindestens aber von Parametern (Anzahl spezieller Register) gemacht werden.

In der zweiten Richtung ist neben der Programmierung insbesondere die Spezifikation, Verifikation und Simulation von eingebetteten Systemen zu wichtigen Anwendungsklassen wie mixed-signal-Anwendungen einzubeziehen. Dabei müssen unterschiedliche Beschreibungstechniken in einem einheitlichen Rahmen gefasst werden. Es ist zu prüfen, wie weitere Werkzeuge, z.B. MATLAB, mit UPHA kooperieren können.

Literatur

- [1] R. Leupers, P. Marwedel, *Optimierende Compiler für DSPs: Was ist verfügbar?*, DSP Deutschland 1997, München, Oktober 1997.
- [2] P. Marwedel and G. Goosens, Eds., *Code Generation for Embedded Processors*. Boston, MA: Academic, 1995.
- [3] M. Müller, *Prototypische Implementierung der internen Datenbasis und des Editors für die untere Ebene einer Umgebung zur Programmierung hocheffizienter Anwendungen für digitale Signalprozessoren*, Studienarbeit am Lehrstuhl für Programmiersprachen und Compilerbau der BTU Cottbus, Juli 1999.
- [4] E. Ofner, R. Forsyth und A. Gierlinger, *GEPARD, ein parameterisierbarer DSP Kern für ASICs*, DSP Deutschland 1997, München, Oktober 1997.
- [5] M. Willems, *DSP-Programmierung: Manuelle Assembler-Programmierung oder mehr?*, DSP Deutschland 1999, München, September 1999.