DSP-Compiler: auf dem Weg zum Produktionstool ?

Möglichkeiten und Grenzen DSP-spezifischer Spracherweiterungen

Markus Willems, Michael Warmers, Vojin Zivojnovic
Lehrstuhl für Integrierte Systeme der Signalverarbeitung
RWTH Aachen
Templergraben 55, 52056 Aachen

1 Einleitung

Hochsprachen-Compiler für digitale Signalprozessoren (DSPs) gehören zu den wenigen Produkten der Signalverarbeitungs-Industrie, über die eine kontroverse Debatte bezüglich ihrer Nützlichkeit geführt wird. Fakt ist, daß heute für nahezu alle angebotenen DSPs Hochsprachen-Compiler zur Verfügung stehen. Unterstützt wird vor allem C, es existieren auch Compiler für C^{++} [1], Ada und Pascal [2]. Fakt ist aber auch, daß signifikante Anteile der DSP-Programmierung in Assembly erfolgen. Damit ergibt sich die provozierende Frage:

DSP-Compiler: Marketing-Tool, oder zukünftig auch Produktions-Tool?

Die Motivationen für eine Programmierung in einer Hochsprache sind allgemein bekannt:

- **Modellierungseffizienz:** die Beschreibung der Funktionalität erfolgt auf einer hohen Abstraktionsebene
- **Wiederverwendbarkeit**: getesteter Code kann auf der gleichen Architektur wiederverwendet werden
- **Wartbarkeit**: Änderungen und Erweiterungen sind wesentlich einfacher einzufügen als im Assembly-Code
- **Testen und Debuggen**: die Hochsprache erlaubt effizientes Testen und Debuggen,
- **Portierbarkeit**: der Code kann auf andere Architekturen übertragen werden

Jeder einzelne Aspekt führt also zu einer Erhöhung der **Produktivität** des Entwicklers.

Die grundsätzliche Aufgabe des Compilers ist die korrekte Übersetzung der Funktionalität aus der Hochsprache in die Maschinensprache. Darüber hinaus bestehen applikationsabhängig verschiedene Anforderungen an den Compiler. Solange Speicherbedarf und Laufzeit keine kritischen Größen darstellen, steht die schnelle Compilierung der Funktionalität im Vordergrund, und tatsächlich sind viele der in der Literatur angesprochenen Optimierungen Compilezeit-Optimierungen.

Eine Abkehr von diesem Prinzip der Compilezeit-Optimierung wird erst dann notwendig, wenn die zugrundeliegende Hardware zur Abarbeitung des generierten Codes unter den gegebenen Randbedingungen nicht mehr in der Lage ist. Hier stellen die Signalprozessoren einen Extremfall dar: der Anwendungsbereich erfordert im allg. die schnelle Bearbeitung numerisch intensiver Aufgaben. Dabei ist aus Kosten- und Leistungsverbrauchs-Gründen ein minimaler Speicherbedarf anzustreben, da der auf dem Chip angeordnete Speicher eine knappe Ressource ist und die Erweiterung um externen Speicher zusätzliche Kosten verursacht. Somit stellt sich für den Compiler die Aufgabe, extrem

- **laufzeiteffizienten** und
- **speichereffizienten** Code zu generieren.

Die Compilierzeit tritt durch diese Anforderungen in den Hintergrund.

Es ist bekannt, daß state-of-the-art Compiler bzgl. dieser beiden Anforderungen sehr oft hinter dem Notwendigen zurückbleiben. Daher gehört weiterhin, trotz der Verfügbarkeit von Compilern, die Programmierung in Assembly zum Handwerk des DSP-Programmierers.

Mit DSPstone wurde erstmals eine Methodik vorgestellt, die eine Erweiterung dieser qualitativen Aussagen durch quantitative ermöglicht [3]. Die Methodik basiert auf folgendem Prinzip.

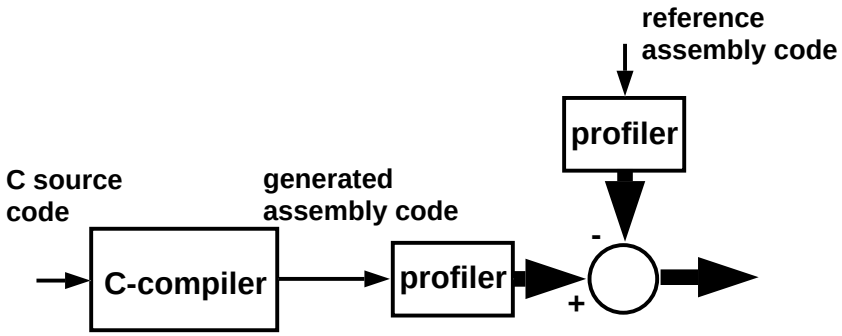

Abbildung 1

Für verschiedene funktionale Kerne wird der C-Quellcode compiliert und bzgl. des Speicherbedarfes und der Anzahl der notwendigen Instruktionen mit dem optimalen Assembly-Code verglichen. Diese Methodik gibt dem Anwender Auskunft über den zu erwartenden Verlust bei der Verwendung des C-Compilers, verglichen mit der optimalen Assembly-Version.
Wird mit DSPstone die Frage nach dem 'Wieviel?' beantwortet, so erlaubt der Vergleich von Compiler-generiertem und optimalem Assembly-Code auch eine Aussage zum 'Warum?'. Die Behandlung des 'Warum?' sowie Ansätze zur Verbesserung der Compiler-Effizienz sind Gegenstand der folgenden Abschnitte.

2 Mikroskopische Analyse

Die Analyse der Unterschiede zwischen Compiler-generiertem und optimalen Code wird im folgenden exemplarisch für das FIR-Filter dargestellt. Natürlich können die Ergebnisse für dieses Beispiel nicht beliebig übertragen werden, doch die gefundenen Ineffizienzen erlauben einen konkreten Einblick in die Ineffizienzen des Compiliervorgangs. Als Beispielarchitektur dient der NEC7701x. Dieser Festkomma-Prozessor mit einer 16-bit Busstruktur bietet 2 getrennte Datenspeicher, auf die parallel zugegriffen werden kann, sowie 8 universell verwendbare, 40-bit breite Register .
Die Funktionalität des FIR-Algorithmus wird beschrieben durch

$$y(k) = \sum_{i=0}^{N-1} x(k-i)h(i)$$

Dabei ist X = {x(0), x(1),...} die Sequenz der Eingangswerte, und H={h(0),...,h(N-1)} der Koeffizienten-Vektor. Zu jedem Zeitpunkt k wird ein neuer Eingangswert x(k) in die Berechnung aufgenommen. Das entspricht einer Verschiebung der Datenwerte relativ zu den Filterkoeffizienten um jeweils einen Wert und dem Herausfallen von x(k-N) aus den relevanten Datenwerten.
Ein C-Code unter Verwendung der Zeiger-Adressierung lautet:

```
y = 0;
for (i=0; i < (N-1); i++){
   y    += *coeff-- * *data;
   *data-- = *old_data--;
}
y  += *coeff-- * *data;
*data = x0;
```

Die Initialisierung der Koeffizienten und Datenwerte ist im untersuchten Code nicht mehr erfaßt.
Zunächst soll der optimale Assembly-Code vorgestellt werden (für N=16). Dabei sind Eingangsdaten sowie Ergebnisse im Integer-Format zu interpretieren:

```
clr(r4)          r1= *dp1%%      r5= *dp5%%;
rep 14;
      r4= r4+ r1h* r5h    r1= *dp1%%      r5= *dp5%%;
r4= r4+ r1h* r5h    r1= *dp1        r5= *dp5%%;
r4= r4+ r1h* r5h    r3= *dp2;
r4= r4 SLL r0l      *dp1= r3h;
*dp4= r4h;
```

Es liegt eine nahezu maximale Parallelisierung vor, neben jeder arithmetischen Operation werden eine oder zwei Ladeoperationen parallel durchgeführt. Das gleichzeitige Durchführen von zwei Ladeoperationen ist nur möglich, wenn die Operanden in unterschiedlichen Speicherbänken angeordnet sind. Dieses erlaubt im vorliegenden Beispiel insbesondere die Ausführung der inneren Schleife als wiederholte *mac*-Operation. Das parallel stattfindende Laden benutzt die Modulo-Adressierung (..%%).
Da, wie angenommen, die Eingangsdaten als Integer vorliegen, der NEC7701x aber eine Fractional-Hardware besitzt, wird eine explizite Verschiebung des Endergebnisses notwendig (Zeile 6). Dieses liegt an der Interpretation des Bitmusters: die Fractional Arithmetik nimmt den Dezimalpunkt links vom SignBit an, während die Integer-Arithmetik vom Dezimalpunkt rechts vom LSB ausgeht. Die Anzahl der zu verschiebenden Stellen steht in einem Register, damit ergibt sich die Möglichkeit der Parallelisierung einer weiteren Ladeoperation.

Der vom Compiler generierte Assembly-Code ergibt sich zu:

```
1        CLR (r5);
2        LOOP 15 {
3          dp5 = r0l;
4          r0 = r0- 1              r1 = *dp6;
5          r2 = *dp7--;
6          r1 = r1h* r2h;
7          r1 = r1 SLL 15;
8          r5 = r5+ r1       r1 = *dp5;
9          *dp6-- = r1h;
10       }
11       r0 = *dp6;
12       r1 = *dp7;
13       r0 = r0h* r1h;
14       r0 = r0 SLL 15;
15       r5 = r5+ r0;
16       r0 = *ANON3:Y
;
```

Eine geringere Parallelisierung als im optimalen Code ist offensichtlich. Dieses hat mehrere Gründe:

- Daten sind nicht in verschiedenen Speicherbereichen angeordnet, damit entfällt die Möglichkeit der Parallisierung, z.B. der Zeilen 4 und 5. Der Compiler benutzt prinzipiell nicht den *X*-Speicher zur Ablage von Daten. Damit sind parallele Ladeoperationen ausgeschlossen. Zu dieser ineffizienten Speicherzuweisung besteht aber keine zwingende Notwendigkeit. Dem Compiler steht bereits die Information bzgl. der Konfiguration des Prozessors zur Verfügung, daher könnte er prinzipiell die Vorteilhaftigkeit alternativer Speicherzuweisungen erkennen. Hier wird deutlich, daß die DSP-Architektur durch die fehlende Orthogonalität (d.h. in diesem Fall, nicht alle Speicherplätze sind alternativ zu gleichen Kosten nutzbar) eine zusätzliche Komplexität in den Compilierungs-Vorgang einbringt, die derzeit von keinem Compiler in geeigneter Weise berücksichtigt wird.
- Die Linksverschiebung (Zeilen 7 und 14) wird nicht durch den Wert in einem Register angegeben, sondern absolut. Damit ist kein paralleles Abarbeiten möglich. Eine Zuweisung des absoluten Wertes zu einem Register außerhalb der Schleife würde (N-1) - 1 = 14 Instruktionen einsparen, denn damit könnte Operation 8.2. parallel zu Zeile 7 ausgeführt werden, und damit Operation 9 parallel zu Operation 8.1. Offenbar führt der Compiler DSP-spezifische Schleifenoptimierungen, die sich aus der Parallisierbarkeit von Operationen ergeben, nicht durch.
- Die Möglichkeit der Modulo-Adressierung wird nicht ausgenutzt. Stattdessen erfolgt in den Zeilen 8 und 9 das Umladen von Speicherbelegungen. Prinzipiell aber wäre der Compiler mittels einer aufwendigen Buchhaltung in der Lage, zu erkennen, daß die in C als Kopieroperation ausgedrückte Funktion durch eine Modulo-Adressierung realisierbar ist: zu Beginn der Schleife ist bekannt, daß *data* und *old_data* um 1 versetzt auf dasselbe Datenfeld zeigen. Innerhalb der Schleife werden die Dekrementierungen parallel fortgesetzt. Ein Überschreiben der Werte, wie es die Modulo-Operation verlangt, ist erlaubt.

  Offensichtlich führt dieser Ansatz für den allgemeinen Fall zu einer hohen Komplexität des Compiliervorgangs. Keiner der in DSPstone untersuchten Compiler erkennt die Möglichkeit der Modulo-Adressierung bei einer Programmierung in ANSI-C.
- Die Operation *y += *coeff * *data* zerlegt der Compiler zunächst in eine Multiplikation (Zeile 6), eine Linksverschiebung zur Abbildung des Ergebnisses auf den Datentyp Integer, sowie eine anschließende Addition von 2 Integer-Zahlen. Durch dieses sequentielle Vorgehen ist sichergestellt, daß exakt die in C spezifizierte Integer-Arithmetik eingehalten wird.

Wird im Gegensatz dazu allerdings die *mac*-Operation verwendet, Zwischenergebnisse im Fractional-Format beibehalten und lediglich das Endergebnis auf Integer abgebildet, so ergeben sich Unterschiede zum vom Compiler realisierten Fall nur, wenn die sequentielle Bearbeitung aufgrund eines Overflows ein fehlerhaftes Ergebnis liefert. Dieser Fall muß aber prinzipiell durch den Programmierer durch eine korrekte Skalierung ausgeschlossen werden. Daher können die Zeilen 6-8 sowie 13-15 jeweils mittels einer *mac* Operation realisiert werden, ohne daß sich die Endergebnisse in der Genauigkeit unterscheiden.

Die bisher betrachteten Ineffizienzen sind also nicht prinzipieller Natur, denn die zur Optimierung notwendigen Informationen sind in der C-Formulierung bereits vorhanden und mit unterschiedlich hohem Aufwand extrahierbar. Der Compiler nutzt diese Informationen allerdings nicht aus.

Ein zusätzlicher Aspekt der Einschränkung wird hier nur indirekt deutlich. Da die NEC-Architektur auf einer Fractional-Darstellung der Festkomma-Werte basiert (d.h. der Wertebereich ist [-1,1)), ist in Assembly die effiziente Formulierung im Fractional-Bereich möglich. Da, wie oben angesprochen, eine Skalierung der Koeffizienten vorgenommen werden muß, kann diese die Zahlen auch direkt in den Fractional-Bereich überführen. Eine Multiplikation kann daher bei Verwendung der Fractional-Arithmetik nie zu einem Überlauf führen. Dieses vereinfacht die Anforderungen an die Skalierung wesentlich. Aus C heraus ist die Ausnutzung der Fractional-Darstellung nicht möglich, daher muß der Compiler immer wieder eine Verschiebung einfügen, um auf die Integer-Darstellung zurückzukehren. Das wiederum erfordert eine wesentlich restriktivere Skalierung. Für das FIR-Filter würde eine Programmierung mit fractional-Zahlen eine Einsparung von einer Instruktion im optimalen Assembly-Code bedeuten, der Compiler-generierte könnte auf 15 Instruktionen verzichten. Nicht erfaßt wird bei dieser Betrachtung der erzielbare Genauigkeitsgewinn durch die wegfallende Skalierung. Für eine genauere Diskussion der Auswirkung einer Fractional-Programmierung auf Genauigkeit und Skalierungsanforderungen wird auf [4] verwiesen.

Damit sind die beiden prinzipiellen Einflußgrößen für Compiler-Ineffizienzen bereits an diesem Beispiel identifizierbar:

- **nicht durchgeführte mögliche Optimierungen**: der Informationsgehalt des Quell-Codes wird nicht ausgenutzt
- **Restriktionen der Eingangssprache**: der Wortschatz der Sprache ist unzureichend

3 Spracherweiterungen

Das Konzept der Spracherweiterungen basiert auf den beiden identifizierten Gründen für die Compiler-Ineffizienz und den daraus abgeleiteten folgenden Aussagen:

- C ist ungeeignet, spezifische DSP-Eigenschaften zu adressieren.
- es ist für einen Compiler aufgrund der Komplexität der Aufgabe nicht möglich, den kompletten Informationsgehalt des Hochsprachenprogramms auszunutzen. Dieses liegt zum einen an der nicht akzeptierbaren Zunahme der Compilierzeit, zum anderen an der fehlenden Marktakzeptanz für hochoptimierende Compiler zu einem hohen Preis.

Soll dennoch nicht auf die Vorteile einer Hochsprachenprogrammierung verzichtet werden, lautet die Konsequenz:
Erlaube dem Programmierer die Mitteilung DSP-spezifischer Optimierungsmöglichkeiten, ohne dabei eine hardwarespezifische Programmierung vorzunehmen.

Dieser Anspruch führt auf zwei Ansätze, die im folgenden näher vorgestellt werden sollen.

3.1 Applikationsorientierter Ansatz

Dieser Ansatz orientiert sich an den typischen Applikationen, die auf einem DSP ausgeführt werden sollen.
Eine typische Applikation auf dem DSP ist beispielsweise die ausführlich behandelte FIR-Filterung. Eine applikationsbezogene Erweiterung muß eine Operation FIR zur Verfügung stellen, deren Operanden der Daten- und der Koeffizientenvektor sind. Es handelt sich also um einen funktional orientierten Ansatz.
Derzeit existieren zu jedem Prozessor umfangreiche Assembly-Bibliotheken, die genau diese Funktionalitäten enthalten, die hier als Spracherweiterungen vorgeschlagen werden. Wo liegt also der Vorteil der Erweiterungen? Messungen zeigen, daß die Verwendung von Assembly-Bibliotheken zu einem Laufzeit-Verlust von ca. 50% - 100% relativ zum optimalen Code führt. Dieses liegt insbesondere daran, daß die Assembly-Funktionalität generisch beschrieben werden muß, um für eine Vielzahl von Applikationen einsetzbar zu sein. Im Gegensatz dazu

erfolgt die Compilierung der Erweiterungen kontextabhängig. Der Compiler kann also unterschiedlichen, an die Umgebung angepaßten Assembly-Code generieren. Darüber hinaus entfällt die Interface-Problematik.
Weitere Möglichkeiten bietet die Einführung neuer Datentypen, die sich nicht an der Hardware orientieren, sondern an der zu beschreibenden Aufgabenstellung. Hier sind z.B. die Einführung komplexer Zahlen sowie Vektoren zu erwähnen. Der Compiler kann aus der Kenntnis der Datentypen die auf diesen Datentypen auszuführenden Operationen effizienter umsetzen als nach der künstlichen Abbildung auf in C definierte Operationen.
Der entscheidende Vorteil dieses Ansatzes ergibt sich aus der gemeinsam erzielbaren Steigerung sowohl der Implementierungseffizienz als auch der Modellierungseffizienz. Dabei verstehen wir unter der Modellierungseffizienz die Möglichkeit, ein Problem effizient zu beschreiben. Die Implementierungseffizienz beschreibt die Qualität des Compiler-generierten Codes.
Es existieren bereits einige Konzepte, die dieser Definition der applikationsorientierten Erweiterungen entsprechen. Numerical C von Analog Devices [5] beispielsweise beinhaltet den Datentyp 'complex', sowie den Operator 'sum', der die Summation von Array-Elementen durchführt. Beispielhaft soll die Sprache DSPL [6] betrachtet werden

3.1.1 Ein Beispiel: DSPL

DSPL ist eine speziell für die Programmierung von DSPs entworfene Hochsprache, die nicht die komplette ANSI-C-Syntax umfaßt, aber DSP-Spracherweiterungskonzepte enthält, die der applikationsspezifischen Definition entsprechen und so ebenfalls in C integrierbar sind. Unterstützt werden die Prozessoren TMS320C1x/C2x.
Exemplarisch wird hier das *ACCUMULATE*-Statement betrachtet, das eine effiziente Beschreibung von Skalarprodukten und FIR-Filteroperationen erlaubt. Die Beschreibung einer kompletten FIR-Filterung kann in DSPL mittels einer einzigen Instruktion vorgenommen werden:

ACCUMULATE SCALPRO (ident) AND UPDATE (data)
    *y := coeff * data*
END ACCUMULATE

Diese Funktionalität erlaubt die automatische Skalierung der Filterkoeffizienten, automatische Rundung und Überlaufdetektion. Die Skalierung der Filterkoeffizienten, die im Fractional-Format vorliegen müssen, wird zur Compile-Zeit ausgeführt. *ident* wird im Deklarationsteil des Programms festgelegt und enthält die Information, in welchem Modus die Akkumulation durchgeführt werden muß. Dieses umfaßt die Festlegung, ob automatische Skalierung vorgenommen werden soll, ob mit einer Sättigungslogik gearbeitet werden soll und ob eine Rundung durchgeführt werden soll.
In DSPL wird die Compilierung des ACCUMULATE Statements auf eine minimale Ausführungszeit ausgelegt, daher werden im Assembly-Code keine Schleifen eingefügt. Zu Lasten des Programmspeicherverbrauchs erzeugt der Compiler sequentiell abzuarbeitenden Code. Ein Vergleich der Ergebnisse für die Realisierung des FIR-Filters in DSPL, ANSI-C und optimalem Assembly-Code für den TMS320C25 ist in Abbildung 2 dargestellt.
Offensichtlich erfodert das Ausnutzen der neuen Funktionalität zunächst eine Einarbeitung in die erweiterte Syntax. Die Modellierungseffizienz wird jedoch erhöht, da die Filterung bzw. das Skalarprodukt direkt als solche identifizierbar sind und nicht in eine Schleifenstruktur abgebildet werden müssen.

3.1.2 Beurteilung des applikationsorientierten Ansatzes

Ein wesentliches Problem des applikationsorientierten Ansatzes stellt die Definition der relevanten Funktionalitäten dar. Wenn eine neues Problem nicht effizient unter Ausnutzung der definierten Funktionalitäten formuliert werden kann, bleibt nur die Programmierung in C. Der Programmierer hat die Möglichkeit, auf dem Abstraktionsniveau der Hochsprache weiterzuarbeiten und kann auf die Erlernung eines Prozessormodells verzichten.
Das Konzept der applikationsorientierten Erweiterungen von C scheint gerechtfertigt, wenn man dem Umstand Rechnung tragen will, daß auch die DSP-Architektur applikationsbezogen aufgebaut ist. Bestes Beispiel ist die Unterstützung des Bit-Reversed-Adressierungs-Modus, der allein für die effiziente Implementierung der FFT eingefügt wurde. Es ist daher nicht einzusehen, warum diese Eigenschaft nicht auch auf die Programmierungsebene erweitert werden sollte.

3.2 Architekturspezifischer Ansatz

Der Ansatz der architekturspezifischen Spracherweiterungen löst sich von dem bisher verfolgten Ansatz, die Formulierung eines Problems unabhängig von der Zielarchitektur durchzuführen. Er basiert auf der Erfahrung, daß im allgemeinen die Architekturklasse (µP, DSP oder GPP) bereits zu Beginn eines Projekts feststeht. Daher soll der

Entwickler die Möglichkeit erhalten, seine architekturspezifischen Kenntnisse in die Programmierung einzubringen, ohne damit eine prozessorspezifische Programmierung vorzunehmen.
Mit dieser Anforderung ist zunächst die Identifikation der allgemeinen Eigenschaften der DSP-Architekturen notwendig, die dann auf ein abstraktes Prozessormodell führen, das Grundlage der Programmierung der Klasse der DSPs werden soll. Im folgenden sollen exemplarisch der Ansatz sowie die Konsequenzen für die Hochsprache C diskutiert werden.
Die Speicherorganisation basiert auf einer Harvard- oder modifizierten Harvard-Architektur. Das erlaubt in der Regel nicht nur eine Trennung von Programm- und Datenworten, sondern auch das getrennte Ablegen von Datenworten in verschiedene Speicherbereiche. Damit sollten die Spracherweiterungen einen Hinweis auf die geeignete Ablage von Daten in bestimmten Speicherbereichen zulassen. Weiterhin kann für DSPs i.a. nicht von einer homogenen Speicherarchitektur ausgegangen werden, vielmehr wird ein Teil des Speichers auf dem Chip, ein anderer extern angeordnet sein, was sich zumindest in unterschiedlichen Zugriffszeiten ausdrücken kann. Somit sollte es dem Entwickler möglich sein, zu spezifizieren, ob Daten möglichst intern anzuordnen sind. Dieses Konzept der Speicherinformation stellt keine fundamentale Erweiterung von C dar, denn schon jetzt kann, z.B. durch den 'storage class specifier' *register*, innerhalb einer Deklaration dem Compiler mitgeteilt werden, daß es vorteilhaft ist, eine Variable möglichst lange im Register zu halten.
Weiterhin besitzen nahezu allen modernen DSP-Architekturen die Möglichkeit der Modulo-Adressierung. Der Benutzer sollte daher die Möglichkeit erhalten, explizit eine Modulo-Adressierung vorzunehmen. Dieses stellt ebenfalls keine prinzipielle Neuerung für die Sprache dar, denn die Inkrementierung/Dekrementierung von Zeigern ist ein zentrales Element von C. Als Unterschied ergibt sich, daß bei der Definition eines Feldes mitgeteilt werden muß, ob für dieses Feld eine zirkulare Adressierung vorzusehen ist. Allerdings ist die Anzahl der möglichen Ringspeicheradressierungen oft durch die Architektur begrenzt. Der Compiler erhält dann die Aufgabe, einen anderen Adressierungsmodus zu realisieren. I.a. ist es einfacher, aus der Information,eine Modulo-Adressierung vornehmen zu können, eine alternative Darstellung zu finden, als beliebige alternative Darstellungen auf ihre Eignung zur Modulo-Adressierung zu untersuchen.
Bit-Reversed-Adressierung ist ein weiterer Adressierungsmodus, der von der Klasse der DSPs unterstützt wird. Dieser Modus ist insbesondere durch die FFT motiviert, für die damit eine effiziente Inplace-Verarbeitung möglich wird.
Ein DSP besitzt keine allgemeine Wortbreite für Speicher, Bus und Register, sondern vielmehr eine heterogene Struktur. Es muß damit dem Benutzer möglich sein, zu jedem Zeitpunkt seine Anforderungen an die Darstellung einer Variablen anfügen zu können.
Es existieren bereits einige Ansätze der architekturspezifischen Erweiterungen. Numerical C erlaubt die explizite Zuweisung einer Variablen zum Daten- oder Programmspeicherbereich. Das gleiche gilt für den Tartan-Compiler für die TI-Fließkomma-Familie, der eine direkte Unterstützung der Modulo-Adressierung erlaubt. Beispielhaft soll das Konzept der architekturspezifischen ‘Language Extensions’ für die NEC7701x-Familie betrachtet werden.

3.2.1 Ein Beispiel: Spracherweiterungen für den NEC7701x

Für den NEC7701x wird von Intermetrics das bisher umfangreichste Konzept der DSP-bezogenen Spracherweiterungen für die Sprache C angeboten. Es handelt sich dabei im wesentlichen um ein architekturbezogenes Erweiterungskonzept. Dieses soll hier nicht im Detail vorgestellt werden (dazu wird auf [7] verwiesen), vielmehr sollen Handhabung und sich ergebende Konsequenzen für den Compiler-generierten Code anhand des FIR-Beispiels diskutiert werden.
Wie bereits bei der Diskussion im Abschnitt 2 identifiziert, ergaben sich bei der ANSI-C-Programmierung Ineffizienzen insbesondere aus der nicht optimalen Speicherzuweisung. Offensichtlich ist es vorteilhaft, Daten- und Koeffizientenvektoren in unterschiedlichen Speicherbänken abzulegen. Weiterhin ist es vorteilhaft, auf den Datenvektor mittels der Modulo-Adressierung zuzugreifen, da so ein Umspeichern vermieden werden kann. Die Typdeklaration als Integer wurde als Grund dafür identifiziert, daß die mac Operation nicht genutzt werden konnte.
Die C-Erweiterungen ermöglichen es nun, einer Variablen explizit mitzuteilen, in welcher Speicherbank sie abzulegen ist. Es handelt sich also im Unterschied zu den im vorherigen Abschnitt angeregten allgemeinen Spracherweiterungen um keine Empfehlung, sondern um eine feste Anweisung. Dieses geschieht durch eine Erweiterung des Typdeklarators um *_x*, *_y*. Natürlich wird bei der Verwendung der Erweiterungen vom Programmierer erwartet, daß er die Vorteilhaftigkeit der Zuordnung für bestimmte Operationen kennt.
Modulo-Adressierung wird ermöglicht durch die Bereitstellung eingebauter Funktionen, die vom Compiler direkt unterstützt werden. Es existieren 6 Funktionen, die auf den Datentyp *circ_buff* angewendet werden können:

- *_Init_cb(cb, ptr, stride, bound)* Initialisierung des Ringspeichers: Anfangsadresse (*ptr*), Inkrementierung/Dekrementierung (*stride*), Größe (*bound*)

- *_Get_cb (cb, type)* Ausgabe des Wertes, auf den der Ringspeicherzeiger gerade zeigt, auf den Datentypen *type* abgebildet.
- *_Next_cb(cb, type*), wie *_Get_cb(cb, type*), allerdings wird der Zeiger entsprechend der Einstellung in *stride* weitergeschoben.
- *_Tell_cb(cb), _Set_cb(cb, index*) Ausgabe und Setzen der derzeitigen Zeigerstellung.
- *_Set_stride(cb, stride)* Setzen der Inkrementierung/Dekrementierung auf einen neuen Wert.

Offensichtlich muß sich der Benutzer vor der Anwendung zunächst mit den Funktionalitäten vertraut machen. Die Programmierung erfordert keine hardwarespezifischen Kenntnisse, sondern lediglich ein Verstehen des abstrakten Ringspeichermodells.
Neben diesen Hinweisen für den Compiler, die geeignet sind, die Komplexität herabzusetzen, existieren auch einige Erweiterungen, die über das in C ausdrückbare hinausgehen. So kann eine Variable als *accum* deklariert werden, d.h. sie soll mit der maximalen Wortbreite von 40-bit dargestellt werden. Weiterhin wird der Datentyp *fractional* eingeführt, der eine Verarbeitung von Zahlen im Bereich [-1,1) erlaubt und damit der implementierten Arithmetik entspricht. Unter Ausnutzung dieser Erweiterungen ergibt sich folgender Code für das FIR-Filter:

```
y = 0;
data = _Next_cb (x_buff, fixed);
coeff = _Next_cb (c_buff, fixed);

for (i = 0; i < LENGTH - 2; i++)
  {
    y += data * coeff;
    data = _Next_cb (x_buff, fixed);
    coeff = _Next_cb (c_buff, fixed);
  }

y += data * coeff;
data = _Get_cb (x_buff, fixed);
coeff = _Next_cb (c_buff, fixed);

y += data * coeff;
_Get_cb (x_buff, fixed)= next;
```

Dabei wurde der Datentyp von *y* zu *accum* deklariert.
Der Assembly-Code lautet:

```
1  CLR (r5)                r0=*dp6%%     r1=*dp2%%;
2  REP  14;
3      r5=r5 + r1h*r0h     r0=*dp6%%     r1=*dp2%%;
4  r5=r5 + r1h*r0h         r0=*dp6%%     r1=*dp2;
5  r5=r5 + r1h*r0h;
6  r0=*_next:X;
```

Dieser Assembly-Code ist nahezu äquivalent zum optimalen.
Es muß erwähnt werden, daß das erzielte Resultat stark von der Formulierung des Algorithmus abhängt. So erkennt man, daß im C-Code der erste Modulo-Zugriff außerhalb der Schleife erfolgt, analog zur Realisierung im Assembly-Code. Unsere erste Formulierung, die eine Schleife der Größe 15 vorsah mit dem Zugriff auf Daten und Koeffizienten vor der Akkumulation, resultierte in einer loop Operation mit 2 Instruktionen (was fast einer Verdopplung der Laufzeit entspricht), wobei in der ersten Instruktion die Daten geladen wurden, und in der zweiten Instruktion die *mac* Operation durchgeführt wurde. Offensichtlich findet keine Schleifen-Optimierung statt. Es muß vielmehr eine Programmierung in C erfolgen, die sich sehr stark an die endgültige Assembly-Version anlehnt.

3.2.2 Beurteilung der architekturspezifischen Spracherweiterungen

Grundsätzlich ist es für den Compiler einfacher, die architekturspezifischen Informationen zu berücksichtigen, da diese häufig eine direkte Umsetzung in Assembly ermöglichen. Dieser Ansatz ist sicherlich allgemeingültiger, da allen Applikationen die Architekturinformation zur Verfügung steht. Für den Programmierer bedeutet dieser Ansatz eines Herabsetzung der Abstraktion, denn schon zur Programmierzeit muß das, allerdings abstrakte, Prozessormodell berücksichtigt werden.

. Der wesentliche Vorteil der architekturspezifischen Erweiterung liegt in der Allgemeinheit des Konzepts sowie in der geringeren notwendigen Compilerkomplexität.

3.3 Leistungsvergleich

Ein Vergleich der Laufzeiten und des Speicherbedarfs für die Implementierung des FIR-Filters unter Ausnutzung der verschiedenen Spracherweiterungskonzepte ist in der folgenden Abbildung dargestellt.

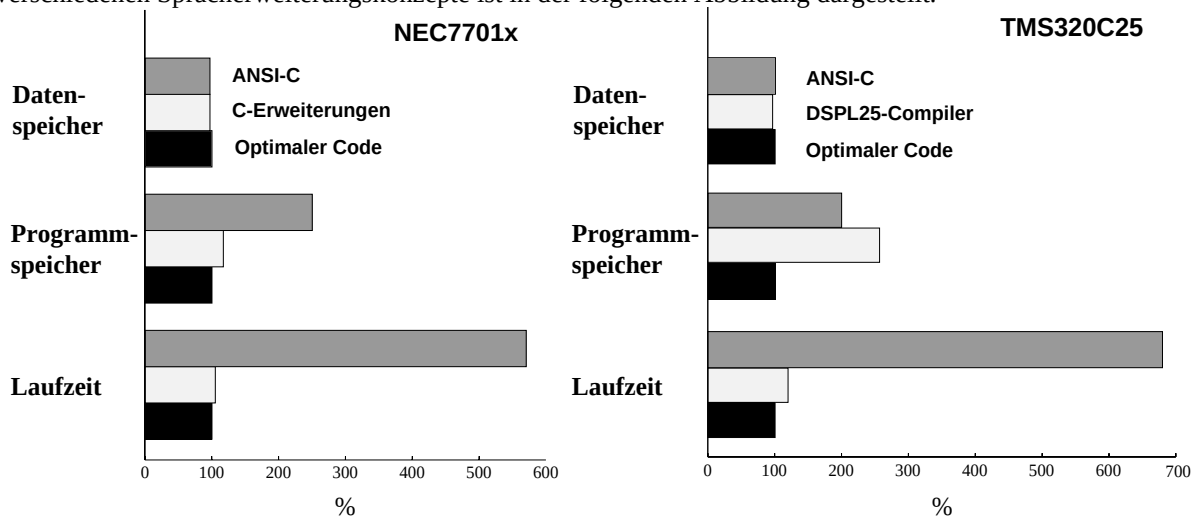

Abbildung 2

Die hervorragenden Ergebnisse für die Realisierungen des FIR-Filters kann natürlich nicht verallgemeinert werden. Innerhalb eines Projekts wurden von uns alle Kerne des DSPstone mit Spracherweiterungen realisiert. Dabei ergab sich in jedem Fall eine Verbesserung der erzielbaren Codequalitäten. Die Laufzeiten relativ zu denen der ANSI-C-programmierten lagen zwischen 19% (FIR-Filter) und 67% (n-real-update).
Im 1. Quartal ‘96 wird ein vollständiger Report zu DSPstone verfügbar sein, der die kompletten Benchmarking-Ergebnisse sowohl für ANSI-C als auch für die verfügbaren Spracherweiterungen umfaßt.

4 Spracherweiterungen: Hochsprache-Assembly?

Wie aus den vorangegangenen Abschnitten zu erkennen ist, erlauben Spracherweiterungen eine wesentlich verbesserte Codegenerierung verglichen mit der Programmierung in ANSI-C. Bleiben aber die Vorteile einer Hochsprachenprogrammierung dabei nicht auf der Strecke? Anhand der eingangs vorgestellten Motivation soll untersucht werden, inwieweit sich durch die Einführung der Spracherweiterungen Veränderungen ergeben.

- **Modellierungseffizienz**: die Vorteile der Hochsprache bleiben im wesentlichen erhalten. Der applikationsorientierte Ansatz kann die Modellierungseffizienz erhöhen, beim architekturorientierten Ansatz wird die Modellierungseffizienz durch die explizite Berücksichtigung des Prozessormodells herabgesetzt. Auf die Spracherweiterungen angepaßte Optimierungen sind notwendig, um nicht hardwarenah programmieren zu müssen.
- **Wartbarkeit**: die Programmstruktur wird durch die Einführung von Spracherweiterungen nicht beeinflußt. Daher sind Änderungen und vor allem das Debugging mit den Vorteilen der Hochsprache weiterhin durchführbar.
- **Wiederverwendbarkeit**: Für den Code bestehen die gleichen Möglichkeiten hinsichtlich der Einbindung in Funktionen wie in der Hochsprache, auch hier ergibt sich keine Veränderung.
- **Portierbarkeit**: Offensichtlich bestehen hier wesentliche Probleme, solange kein Standard festgelegt ist. Wie unsere Darstellungen zum abstrakten Prozessormodell gezeigt haben, ist es möglich, auch architekturspezifische Erweiterungen zu definieren, die prinzipiell für alle Signalprozessor-Architekturen ausgenutzt werden können. Die derzeitigen Insellösungen sind nicht geeignet, auf breite Akzeptanz zu stoßen. Im vorherigen Abschnitt haben wir gesehen, daß für den dort vorgestellten Ansatz die Programmierung der spezifischen Informationen sehr nah am Assembly-Modell erfolgen muß, um den optimalen Code zu erzielen. Dieses entspricht unserer Erfahrung, daß unterschiedliche Compiler eine unterschiedliche C-Beschreibung zur Generierung von effizientem Code erfordern. Damit ist eine Portierbarkeit des Codes zwar gewährleistet, eine Portierung der Effizienz jedoch nicht.

Die **Produktivitát** kann also durch die Einführung von Spracherweiterungen wesentlich erhöht werden, wenn die Alternative eine (teilweise) Programmierung in Assembly ist. Umfassendere Abschnitte des Codes können in der Hochsprache formuliert werden, und eine evtl. Assembly-Programmierung bleibt auf wenige Funktionalitäten beschränkt

5 Zusammenfassung und Ausblick

Im vorliegenden Artikel wurden die Motivation, die Möglichkeiten sowie die Einschränkungen für die Einführung von DSP-spezifischen Spracherweiterungen behandelt. Ausgehend von den mittels DSPstone ermittelten Ergebnissen wurde gezeigt, welche Ineffizienzen sich derzeit im Compiliervorgang ergeben. Eine Analyse der Gründe führte zu der Erkenntnis, daß viele Optimierungen prinzipiell während des Compiliervorgangs bereits erkennbar sind, wegen der damit verbundenen Komplexität aber vom Compiler nicht durchgeführt werden. Dieses beruht zum einen auf der damit stark ansteigenden Compilierzeit, zum anderen auf den fehlenden finanziellen Anstrengungen zur Entwicklung hochoptimierender Compiler für DSPs.
Weiterhin erlaubt C als die untersuchte Hochsprache es nicht, die heterogene Struktur des DSP insbesondere durch die variierenden Wortbreiten zu unterstützen sowie hardware-spezifische Stärken auszunutzen.
Als ein Konzept zur Beseitigung dieser Ineffizienzen wurden DSP-spezifische Spracherweiterungen identifiziert. Ziel muß es sein, dem Entwickler die Möglichkeit zu geben, klassenspezifische Informationen in den Code einzubringen und so dem Compiler Hinweise zur effizienten Umsetzung der Funktionalität in die Maschinensprache zu geben. Dieses Konzept beruht auf der Idee, daß man einer Architektur, die für eine bestimmte Applikationsdomäne entwickelt wurde, auch eine spezielle Sprache, die auf diese Domäne zugeschnitten ist, zuweisen soll.
Das Beispiel der Erweiterung für den NEC7701x hat gezeigt, daß durch Spracherweiterungen signifikante Verbesserungen erzielt werden können. Die kompletten Ergebnisse unter Einbeziehung der Spracherweiterungen werden in Kürze in einem umfassenden Report herausgegeben. Dieser Report wird die Benchmarking-Ergebnisse aller relevanten derzeit verfügbaren Compiler sowohl für Festkomma- als auch für Fließkomma-Prozessoren enthalten.
Effiziente Compiler werden auf die Möglichkeit der Spracherweiterungen in absehbarer Zeit nicht verzichten können. Dieses erfordert jedoch die Bereitschaft, sich auf einen Sprachstandard zu verständigen. Insellösungen werden von den Anwendern nicht akzeptiert werden. Weiterhin ersetzen Spracherweiterungen nicht die Notwendigkeit guter Optimierungsstrategien, vielmehr sollten sie als Hilfsmittel verstanden werden, bestimmte Optimierungsrichtungen zu verfolgen. Erst das Zusammenspiel von Spracherweiterungen und angepaßten Optimierungsverfahren wird wesentliche Verbesserungen der Compiler-Ergebnisse bewirken, die dann zu einer breiteren Akzeptanz von Compilern im Anwenderbereich führen werden.

[1] S.Harbison,“Uses and Misuses of C^{++} in DSP Application Development“, in *Proc. of ICSPAT’94*, pp. 703 - 708, Oct. 94

[2] D.Crookes, D.Freeman and C.Gostling,“Improving DSP Software Productivity Using an Optimizing Pascal Compiler for the AT & T DSP32C*“, Microprocessing and Microprogramming*, pp. 239-242, 89

[3] V.Zivojnovic, J. Martinez, C.Schläger, and H.Meyr,“DSPstone: A DSP-oriented Benchmarking Methodology“, in *Proc. of ICSPAT ‘94*, Oct.94

[4] M.Willems,“Fractional vs. Integer Arithmetic - Comparing Scaling and Quantization Effects,“ Report, *Aachen Univ. of Technology*, Aug.95

[5] M.Hoffman and A.Zatsman,“Numerical C“, in *Proc. of DSPs in der Praxis*, Munich 93, pp.161-174, Oct.93

[6] A.Schwarte and H.Hanselmann,“The Programming Language DSPL“,in *Proc. Int. Conference on Intelligent Motion*, Munich 90, pp.268-280, Jun.90

[7] B.Krepp,“DSP Extensions to ANSI-C“,in *Proc. of ICSPAT’94*, pp.695-702, Oct.94