Schneller Code statt schnelle Compiler
Neuartige Code-Optimierungen für digitale Signalprozessoren

Rainer Leupers

DSPs werden heute noch zeitaufwendig in Assembler programmiert, da die verfügbaren C-Compiler relativ schlechten Code generieren. Die Hauptursache hierfür sind komplexe, anwendungsspezifische Befehlssätze, welche die Erzeugung von effizientem Maschinencode für die Compiler stark erschweren. Dieser Artikel beschreibt Wege und Techniken zur Produktivitätssteigerung in der DSP-Softwareentwicklung mittels innovativer Compiler-Optimierungstechniken.[1]

Compiler für eingebettete Systeme

Eingebettete Systeme bestehen heute oft zu einem großen Teil aus Software, welche auf eingebetteten Prozessoren, z.B. RISCs, Mikrocontroller und DSPs abläuft. Der Systementwurf mit programmierbaren Prozessoren bietet gegenüber der Verwendung von ASICs den Vorteil höherer Flexibilität und besserer Wiederverwendbarkeit von Systemkomponenenten. Der Anteil softwarebasierter eingebetteter Systeme wird daher weiterhin zunehmen.

Während im PC- und Workstation-Bereich zur Softwareentwicklung fast ausschließlich Compiler (meist für C oder C++) eingesetzt werden, überwiegt im eingebetteten Bereich heute noch die zeitaufwendige und fehleranfällige Programmierung in Assembler. Abb. 1 zeigt das qualitative Verhältnis der Verwendung von C und Assembler zur Programmierung eingebetteter DSPs und Mikrocontroller in einer Studie von Paulin [1]. Der Grund für die Vorherrschaft von Assembler ist, daß der durch einen Compiler generierte Code langsamer (und auch größer) ist als handgeschriebener Assemblercode. Ist die Programmspeichergröße begrenzt und müssen Echtzeitanforderungen erfüllt werden, so ist die Assemblerprogrammierung trotz ihrer Nachteile oft die einzige Alternative um effizienten Maschinencode zu erhalten. Angesichts der steigenden Komplexität eingebetteter Systeme und Prozessoren ist dies langfristig ein unhaltbarer Zustand. Um kurze Entwicklungszeiten zu erreichen ist es notwendig, auch im eingebetteten Bereich den Schritt von der Assemblerprogrammierung zur Verwendung von C-Compilern zu vollziehen. Hierzu muß allerdings zunächst die Codequalität der Compiler verbessert werden.

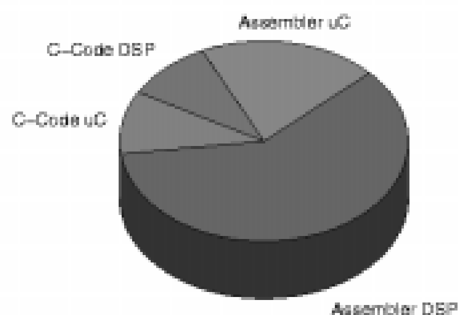

Abbildung 1: *Verhältnis von C und Assembler*

C-Compiler für DSPs

Verbesserungbedarf besteht insbesondere im Bereich der digitalen Signalverarbeitung. Hier steht zwar eine Reihe von DSP-Prozessoren mit spezieller Hardwareunterstützung für rechenintensive Applikationen zur Verfügung, jedoch lassen die zugehörigen C-Compiler in der Codequalität sehr zu wünschen übrig. Empirische Studien zeigen, daß der Overhead von compilergeneriertem Code im Vergleich zu handgeschriebenem Code für typische DSP-Algorithmen mehrere 100 % betragen kann [2, 3], was für den DSP-Bereich inakzeptabel ist.

Die Ursache für die schlechte Codequalität von C-Compilern für DSPs wird offenbar, wenn man deren Assembler-Befehlssatz genauer betrachtet. Auf einem Texas Instruments TMS320C25 DSP beispielsweise führt der Befehl "MPYA *+" drei Operationen (Addition, Multiplikation und Adreßmodifikation) parallel in einem einzigen Befehlszyklus aus. Ein solcher komplexer "multiply-accumulate"-Befehl ist zwar äußerst nützlich zur effizienten Realisierung digitaler Filter, jedoch gibt es hierzu keine Entsprechung in der Programmiersprache C, die nur eine sequentielle Befehlsabarbeitung kennt. Der Compiler muß also erst mühsam herausfinden, wie sich die abstrakten C-Befehle am besten in die sehr speziellen DSP-Befehle übersetzen lassen, was

[1] Publication: Elektronik, Nr. 22, WEKA Verlag, Munich/Germany, 1999

unter anderem eine Umordnung der Befehle im ursprünglichen Programm erfordern kann. Abhilfe schaffen könnte hier eine besser auf DSP-Applikationen abgestimmte Programmiersprache, jedoch möchten die meisten Anwender nicht auf die vielfältigen Vorteile einer standardisierten Sprache wie C nicht verzichten. Daher bleibt nur die Möglichkeit, die Compiler leistungsfähiger zu machen.

Der Schlüssel zur Entwicklung besserer DSP-Compiler ist der Einsatz von neuen, relativ zeitaufwendigen Optimierungstechniken. Normalerweise wird von Compilern eine hohe Übersetzungsgeschwindigkeit erwartet, d.h. Tausende von Source-Code-Zeilen sollen möglichst in wenigen Sekunden in Assemblercode übersetzt werden. Dies macht Sinn z.B. bei der Entwicklung sehr komplexer Software auf Workstations. Im Gegensatz dazu liegt die Codegröße bei eingebetteten Systemen oft nur im Kilobyte-Bereich, und die Software muß darüberhinaus auch nicht oft neu compiliert werden. Bedenkt man, daß die Zeit für die Logiksynthese einer Schaltung mittels eines ASIC-Design-Tools oft Stunden oder gar Tage beträgt, so spricht nichts dagegen, auch einen C-Compiler in der höchsten Optimierungsstufe im Extremfall "über Nacht" laufen zu lassen, wenn dadurch letztendlich die ROM-Größe in einem Massenprodukt oder die Leistungsaufnahme eines Handys entscheidend gesenkt werden können.

Heutige DSP-Compiler realisieren diese Idee nur unzureichend und bauen stattdessen nach wie vor auf herkömmlichen, d.h. schnellen aber in ihrer Wirkung beschränkten Code-Optimierungstechniken auf. Anhand zweier konkreter Techniken wollen wir in diesem Artikel zeigen, wie sich die Codequalität für DSPs steigern läßt, wenn die Übersetzungsgeschwindigkeit nicht länger im Vordergrund steht. Diese Beispiele sind stellvertretend für eine ganze Reihe neuer DSP-spezifischer Optimierungen, welche in den letzten Jahren von verschiedenen universitären und industriellen Forschungsgruppen entwickelt wurden. Hierzu zählen z.B. Techniken zur Ausnutzung der speziellen Speicher-Adressierungsarten von DSPs, die Behandlung von Spezialregistern (wie Akkumulatoren) und die Verteilung von Programmvariablen auf mehrere Speicherbänke zur Erhöhung der Zugriffsbandbreite. Eine Übersicht wichtiger Techniken ist in [4] enthalten.

Ausnutzung von SIMD-Befehlen

Die Befehlssätze neuerer DSPs, wie der Texas Instruments 'C6x oder der Philips Trimedia, bieten eine direkte Unterstützung für Multimedia-Anwendungen. Da die Wortbreite dieser DSPs bei 32 Bit liegt, viele Multimedia-Anwendungen jedoch nur 8 oder 16 Bit Genauigkeit erfordern, lassen sich die 32-Bit-Datenpfade dieser DSPs in zwei Teile zu 16 Bit oder vier Teile zu 8 Bit "aufspalten". Diese Idee ist auch in Intels MMX-Architektur realisiert. Mit einer solchen *single instruction multiple data* (SIMD) Architektur lassen sich z.B. zwei getrennte 16-Bit-Additionen in einem einzigen Befehl ausführen. Es ergibt sich somit eine wesentlich bessere Ausnutzung der Prozessor-Ressourcen. Abb. 2 verdeutlicht dies anhand des ADD2-Befehls des TI 'C6x, welcher für die Argumente und Resultate jeweils die oberen und die unteren 16 Bit eines vollen 32-Bit-Registers verwendet.

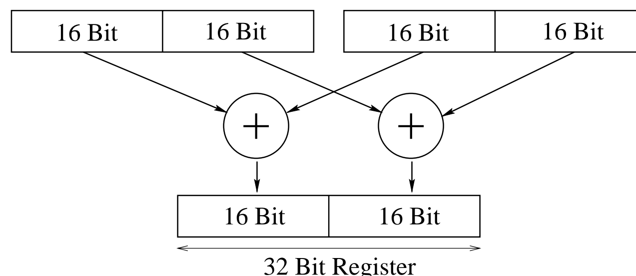

Abbildung 2: *TI 'C6x Befehl ADD2*

Der Einsatz von ADD2 anstelle von separaten 32-Bit-Additionen für eine 16-Bit-Vektoraddition verringert die benötigte Rechenzeit um 50 %. Leider sind heutige Compiler nicht in der Lage, diese SIMD-Befehle bei der Übersetzung von C-Code (ohne Hilfestellung durch spezielle Sprachkonstrukte) auszunutzen, wodurch sich potentiell ein hoher Verlust an Codequalität ergibt.

Warum ist die Ausnutzung von SIMD-Befehlen schwierig ? Um dies zu erläutern, ist es hilfreich, auf eine graphische Programmdarstellung zurückzugreifen. Wir betrachten eine einfache C-Schleife für die Addition zweier 16 Bit-Vektoren:

```
void f(short* A,short* B,short* C)
{ int i;
  for (i = 0; i < N; i += 2)
  { A[i]   = B[i] + C[i];
    A[i+1] = B[i+1] + C[i+1];
  }
}
```

Diese Schleife wurde einmal "abgerollt", um die mögliche Parallelität sichbar zu machen. Abb. 3 stellt die beiden Zuweisungen im Schleifenkörper in Form eines Datenflußgraphen (DFG) dar, welcher aus zwei Bäumen besteht. Herkömmliche Techniken zur Codegenerierung arbeiten stets nur auf einzelnen Bäumen. Hierbei wird ein Baum jeweils mittels der verfügbaren Instruktionen überdeckt. Da die beiden 16-Bit-Teiladditionen des ADD2-Befehls für sich genommen jeweils keine gültigen Befehle darstellen, bleibt nur die Möglichkeit, beide Bäume getrennt mit 32-Bit-Befehlen zu überdecken. Hierzu werden jeweils zwei 32-Bit-Load-Befehle, eine 32-Bit Addition und ein 32-Bit Store-Befehl benötigt. Dies ist zwar für jeden einzelnen Baum optimal, insgesamt jedoch nicht.

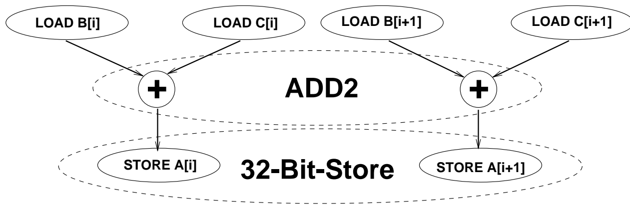

Abbildung 3: *Datenflußgraph der Vektoraddition*

Um den ADD2-Befehl auszunutzen, müßte der Compiler den DFG als Ganzes betrachten, denn mittels ADD2 lassen sich, wie in Abb. 3 angedeutet, die Additionen in beiden Bäumen gleichzeitig überdecken. Ebenso lassen sich auch die Load- und Store-Befehle paarweise durch 32-Bit-Befehle überdecken. Der Grund warum dies in existierenden Compilern nicht geschieht liegt in der Problemkomplexität. Während man einzelne Bäume effizient und optimal überdecken kann, ist die optimale Überdeckung allgemeiner DFGs ein NP-hartes Problem, dessen Lösung im schlechtesten Fall exponentielle Rechenzeit erfordert. Für große DFGs ist dies nicht praktikabel.

Die Ausnutzung von SIMD-Befehlen läßt sich aber in vernünftiger Zeit erreichen, wenn man einen Kompromiß zwischen Baum- und DFG-Überdeckung realisiert. Die einzelnen Bäume in einem DFG werden zunächst auf herkömmliche Weise getrennt überdeckt. Dabei werden allerdings noch Alternativen bzgl. der Operanden- und Ergebnisregister offengehalten. Für eine 16-Bit-Addition z.B. wird noch nicht festgelegt, ob das Ergebnis letztlich in einem vollen 32-Bit-Register, oder in einem (oberen oder unteren) 16-Bit-Teilregister abgelegt wird. Diese Entscheidung wird erst später getroffen, wenn die alternativen Überdeckungen aller Bäume des DFGs berechnet worden sind. Es verbleibt dann nur noch die Aufgabe, unter den Alternativen diejenigen Kombinationen auszuwählen, welche (a) zu einer gültigen Lösung (d.h. zu korrektem Code) und (b) zu einer bestmöglichen Ausnutzung der SIMD-Befehle führen. Dies ist zwar immer noch ein NP-hartes Problem, der Suchraum ist jedoch durch die vorherige Baumüberdeckung bereits stark eingeschränkt.

Die endgültige DFG-Überdeckung erfolgt dann mittels ganzzahliger linearer Optimierung, für die schnelle Software-Werkzeuge existieren. Mit Hilfe dieses "hybriden" Verfahrens ist es zum ersten Mal möglich, SIMD-Befehle für reinen ANSI C-Code (ohne zusätzliche Sprachkonstrukte oder Assembler-Libraries) auszunutzen, wodurch eine einfachere Programmierung und eine bessere Portierbarkeit des Codes erreicht wird. Der Zeitaufwand ist zwar beträchtlich höher als bei anderen Compilertechniken, liegt aber immer noch im akzeptablen Bereich (bis zu 30 CPU-Sekunden bei DFGs normaler Größe). Experimente für die TI 'C6x und Philips Trimedia DSPs ergaben Verbesserungen in der Codequalität um bis zu 75 % gegenüber dem herkömmlichen baumbasierten Verfahren.

Optimales Funktions-Inlining

Die Verwendung von Funktionen in C-Code dient häufig der besseren Modularisierung eines Programms. Die Verwendung vieler "kleiner" Funktionen bedeutet aber stets einen Verlust an Ausführungsgeschwindigkeit, da Funktionsaufrufe einen Overhead in Form von Parameterübergabe und Retten von Registerinhalten bewirken. Bei Prozessoren mit Befehls-Pipelining unterbrechen Funktionsaufrufe zudem dem noch linearen Kontrollfluß eines Programms, was wiederum das temporäre Anhalten der Pipeline erforderlich macht.

Eine bekannte Optimierung ist daher das Inlining von Funktionen. Hierbei werden Funktionsaufrufe durch Kopien des entsprechenden Funktionskörpers ersetzt, d.h. die Funktionen werden praktisch zu Makros wie im folgenden Beispiel:

```
/* Ohne Inlining */

int f(int x)
{
  return x * 2 + 1;
}

int g(int x)
{
  return f(x >> 1);
}

/* Mit Inlining von f */

int g(int x)
{
  return (x >> 1) * 2 + 1;
}
```

Hierdurch wird im allgemeinen die Ausführungsgeschwindigkeit gesteigert, allerdings zu Lasten einer höheren Codegröße, da die Funktionen evtl. vielfach dupliziert werden. Um das Inlining durch den Programmierer steuern lassen zu können, kennen viele C-Compiler ein "inline"-Schlüsselwort bei Funktionsdefinitionen. Ebenso beherrschen viele Compiler das automatische Inlining. Hierbei werden diejenigen Funktionen vom Compiler selbst zum Inlining ausgewählt, die ein gewisses Kriterium erfüllen, z.B. eine gegenüber dem Aufruf-Overhead als kurz eingeschätzte Ausführungszeit. Auch hier ist das primäre Ziel normalerweise, den zur Optimierung benötigten Zeitaufwand

möglichst gering zu halten. Daher ist die Wirksamkeit solcher Techniken nur lokaler Natur.

Bei eingebetteten Systemen möchte der Softwareentwickler durch das Inlining jedoch möglichst mehr erreichen, nämlich ein *insgesamt* geschwindigkeitsoptimales Inlining der in einer Applikation enthaltenen Funktionen unter dem Gesichtspunkt einer *beschränkten Codegröße* für die Applikation. Das erfordert eine zeitaufwendige globale Analyse des Source Codes, und dies wird durch herkömmliche Compiler-Techniken nicht geleistet. Insbesondere bei großen Applikationen ist es für den Programmierer nur sehr schwer einzuschätzen, welche Funktionen für das Inlining auszuwählen sind, so daß die Geschwindigkeit unter Einhaltung der maximalen Codegröße optimal ist.

Abhilfe schafft hier das folgende Verfahren. Ist ein C-Code mit N Funktionen $(f_1, \ldots, f_N)$ gegeben, so werden zunächst mittels eines normalen Compilers und eines Profilers folgende Daten (ohne jegliches Inlining) ermittelt:

1. Die *Größe* $B(f_i)$ (in Bytes) des compilierten Maschinencodes für jede Funktion f_i
2. Die Anzahl $D(f_i)$ der *dynamischen* (d.h. zur Laufzeit ausgeführten) Aufrufe jeder Funktion f_i
3. Die Anzahl $S(f_j, f_i)$ der *statischen* (d.h. im C-Code vorhandenen) Aufrufe von f_i durch f_j für jedes Paar von Funktionen.

Bei N Funktionen gibt es 2^N mögliche Inlining-Kombinationen, denn jede einzelne Funktion kann (oder kann nicht) "ge-inlined" werden. Gesucht ist diejenige Kombination, die zu maximaler Ausführungsgeschwindigkeit unter Einhaltung eines Limits L der Codegröße führt. Man hat es also wieder mit einem exponentiellen Problem zu tun. Dieses Problem kann man jedoch näherungsweise (in den allermeisten Fällen) recht effizient mit Hilfe eines *Branch-and-Bound*-Algorithmus lösen.

Die Grundidee ist, daß eine Minimierung der dynamischen Funktionsaufrufe tendenziell auch die Ausführungsgeschwindigkeit minimiert. Wird z.B. die Funktion f_i "ge-inlined", so sinkt die Anzahl der Funktionsaufrufe offensichtlich um $D(f_i)$. Allerdings wächst hierdurch die Codegröße um $B(f_i)$ multipliziert mit der Anzahl der statischen Aufrufe von f_i. Die hohe Problemkomplexität ergibt sich dadurch, daß die Gesamtcodegröße auch stets davon abhängt, welche der übrigen Funktionen f_j, die f_i aufrufen, für das Inlining ausgewählt werden. Man kann also nicht entscheiden, ob das Inlining von f_i einen Vorteil bringt und außerdem nicht das Limit verletzt, ohne gleichzeitig anderen Funktionen "im Auge" zu behalten Hier setzt nun der Branch-and-Bound-Algorithmus an. Mit Hilfe der Werte $B(f_i)$, $D(f_i)$ und $S(f_j, f_i)$ kann man für jede Funktion f_i durch die Berechnung von *unteren Schranken* entscheiden, ob das Inlining einen Vorteil bringt, ohne alle anderen Funktionen zu betrachten. Ein großer Teil des Suchraums wird dadurch frühzeitig und ohne Verlust der Optimalität "abgeschnitten".

Das Verfahren wurde für einen in C beschriebenen GSM Sprach- und Kanalcodierer und einem TI 'C6x als Zielarchitektur experimentell ausgewertet. Die Applikation besteht aus gut 7000 Zeilen C-Code mit 126 Funktionen, von denen mittels Profiling 26 Funktionen als mögliche Kandidaten für das Inlining bestimmt wurden. Die ursprüngliche Codegröße ohne Inlining betrug ca. 68 KB bei einer (durch Simulation gemessenen) Ausführungsgeschwindigkeit von 27.400.547 Befehlszyklen. Nun wurde geprüft, welche Beschleunigung durch Funktions-Inlining erzielt werden kann, wenn die maximal zulässige Codegröße um 25 % gegenüber dem ursprünglichen Wert gesteigert wird. Es ergab sich eine Reduktion um 33 % auf 18.235.114 Zyklen bei einer Codegröße von rund 83 KB. Dank des Branch-and-Bound-Algorithmus konnte die riesige Zahl von 2^{26} Möglichkeiten innerhalb von nur knapp 6 CPU-Minuten vollständig untersucht werden. Es ergab sich eine Auswahl von 11 bestimmten Funktionen aus den 26 möglichen Kandidaten, die in diesem Fall das Optimum darstellten.

Wie bei der oben beschriebenen Technik zur Ausnutzung von SIMD-Befehlen bringt eine im Vergleich zu normalen Compilern relativ zeitaufwendige Code-Optimierung auch hier eine wesentliche Verbesserung der Codequalität.

Wollen Sie einen Compiler entwickeln ?

Verwenden Sie für den Entwurf eingebetteter Systeme einen In-House-Prozessor oder einen anwendungsspezifischen Prozessor eines Drittherstellers, für den noch kein Compiler zur Verfügung steht ? Falls ja, so stellt sich möglicherweise mittelfristig die Frage nach dem Aufwand zur Entwicklung eines C-Compilers zur effizienteren Programmierung als nur mittels Assembler. Dabei ist zu beachten, daß die prozessorspezifischen Code-Optimierungen nur ein Teil eines Compilers sind. Dieser muß zunächst ein Frontend zur Analyse des C-Codes und zur Erzeugung eines Zwischenformates besitzen und sollte auch prozessorunabhängige Standardoptimierungen durchführen. Diese Komponenten machen bereits einen erheblichen Teil des Entwicklungsaufwandes aus. Hilfe leisten hier kommerzielle Werkzeuge zur automatisierten Compiler-Generierung (z.B. Cosy [5] der Firma ACE).

Um die Entwicklung von Compiler-Prototypen zu vereinfachen, hat der Lehrstuhl für Technische Informatik der Universität Dortmund das Compilersystem LANCE implementiert. Dieses beinhaltet ein ANSI C-Frontend und eine Reihe von Standardoptimierungen. Das vom Frontend generierte Zwischenformat ist prozessorunabhängig und für alle Optimierungen einheitlich. Es wird über eine Funktionsschnittstelle in Form einer C++ Library zugegriffen. Zusätzliche prozessorspezifische Werkzeuge, z.B. Codegenerato-

Abbildung 4: *Benutzerinterface des LANCE-Systems*

ren, lassen sich daher auf einfache Weise integrieren und über die graphische Oberfläche (Abb. 4) bedienen. Der Einsatz von LANCE ermöglicht es, sich bei der Compilerentwicklung auf das wesentliche zu konzentrieren und schnell neue Code-Optimierungen an praktischen Beispielen ausprobieren zu können. Das System läuft auf SUN/Solaris und PC/Linux-Plattformen und kann per Internet heruntergeladen werden [6].

Ausblick

Der Einsatz von C-Compilern gewinnt auch bei den eingebetteten Systemen zunehmend an Bedeutung. Dies gilt insbesondere für DSPs, deren stark anwendungsspezifische Befehlssätze die Assemblerprogrammierung und auch Portierung von Software sehr zeitaufwendig machen. Die meisten der heute erhältlichen Compiler für DSPs sind aufgrund ihrer schlechten Codequalität nur sehr begrenzt in der Praxis einsetzbar. Um dieses Problem zu beseitigen und somit eine höhere Produktivität in der DSP-Softwareentwicklung zu erzielen, muß die Codequalität der Compiler entscheidend verbessert werden. Zahlreiche Forschungsergebnisse der letzten Jahre zeigen, daß dies durchaus möglich ist, wenn man von der althergebrachten Forderung nach sehr schnellen Compilern absieht und stattdessen höhere Laufzeiten für fortgeschrittene Code-Optimierungen in Kauf nimmt.

Neben den optimierenden DSP-Compilern werden in Zukunft vermutlich auch *retargierbare* Compiler wichtig werden. Solche Compiler erhalten als Eingabe nicht nur den zu übersetzenden C-Code, sondern auch ein (z.B. in der Hardware-Beschreibungssprache VHDL spezifiziertes) Modell des Zielprozessors. Sind gewisse Eigenschaften des Prozessors wie die Anzahl der Register oder der Funktionseinheiten parametrisierbar, was heute schon bei einigen DSPs der Fall ist, so kann der Compiler einfach durch eine entsprechende Änderung des Prozessormodells auf verschiedene Prozessorarchitekturen angepaßt werden. Der Compiler optimiert somit nicht nur den Code, sondern hilft auch bei der Bestimmung der bestmöglichen Prozessorarchitektur für eine gegebene Anwendung. Diese Idee, welche unter das Stichwort "Hardware/Software Codesign" fällt, wird bereits durch einige kommerzielle Werkzeuge (z.B. Chess [7]) unterstützt.

Literatur

[1] P. Paulin, M. Cornero, C. Liem, et al.: *Trends in Embedded Systems Technology*, in: M.G. Sami, G. De Micheli (eds.): *Hardware/Software Codesign*, Kluwer Academic Publishers, 1996

[2] V. Zivojnovic, J.M. Velarde, C. Schläger, H. Meyr: *DSPStone – A DSP-oriented Benchmarking Methodology*, Int. Conf. on Signal Processing Applications and Technology (ICSPAT), 1994

[3] M. Levy: *C compilers for DSPs flex their muscles*, EDN Access, Juni 1997

[4] R. Leupers: *Retargetable Code Generation for Digital Signal Processors*, Kluwer Academic Publishers, 1997

[5] WWW: www.ace.nl

[6] WWW: ls12-www.cs.uni-dortmund.de/~leupers

[7] WWW: www.retarget.com