Optimierende Compiler für DSPs: Was ist verfügbar ?

Rainer Leupers, Peter Marwedel
Universität Dortmund, Lehrstuhl Informatik 12
44221 Dortmund
email: leupers|marwedel@ls12.informatik.uni-dortmund.de

Abstract – Die Softwareentwicklung für eingebettete Prozessoren findet heute größtenteils noch auf Assemblerebene statt. Der Grund für diesen langfristig wohl unhaltbaren Zustand liegt in der mangelnden Verfügbarkeit von guten C-Compilern. In den letzten Jahren wurden allerdings wesentliche Fortschritte in der Codeoptimierung – speziell für DSPs – erzielt, welche bisher nur unzureichend in kommerzielle Produkte umgesetzt wurden. Dieser Beitrag zeigt die prinzipiellen Optimierungsquellen auf und faßt den Stand der Technik zusammen. Die zentralen Methoden hierbei sind komplexe Optimierungsverfahren, welche über die traditionelle Compilertechnologie hinausgehen, sowie die Ausnutzung der DSP-spezifischen Hardware-Architekturen zur effizienten Übersetzung von C-Sprachkonstrukten in DSP-Maschinenbefehle. Die genannten Verfahren lassen sich teilweise auch allgemein auf (durch Compiler generierte oder handgeschriebene) Assemblerprogramme anwenden.[1]

1 Einleitung

1.1 Methoden der DSP-Programmierung

Im Bereich des Entwurfs *eingebetteter Systeme* ist gegenwärtig ein Trend zum *prozessorbasierten Entwurf* zu beobachten. Prozessoren ermöglichen die *Wiederverwendbarkeit* von Hardware- und Software-Entwürfen, und sie garantieren ein hohes Maß an *Flexibilität* durch Programmierbarkeit.

Prozessorbasierter Entwurf erfordert Programmierwerkzeuge, d.h. Compiler, Assembler, Linker, Debugger und Profiler. Unter diesen spielen die Compiler eine herausragende Rolle, da sie die Verwendung von *Hochsprachen* (vorzugsweise C) zur Programmierung eingebetteter Prozessoren erlauben. Dies wiederum bewirkt einfachere Programmentwicklung, weniger Fehler und leichtere Portierbarkeit von Programmen.

[1]DSP Deutschland '97, München, Oktober 1997, ©MagnaMedia Verlag

Speziell im Bereich der digitalen Signalprozessoren ist die Leistungsfähigkeit heutiger C-Compiler im Bezug auf die Codequalität allerdings schwach, was für verschiedene DSPs und DSP-Anwendungsbereiche dokumentiert wurde [1]. Der Grund hierfür liegt darin, daß DSPs sehr spezifische Maschineninstruktionen haben. Diese Instruktionen sind effizient im Bezug auf DSP-Anwendungen, haben aber keine direkte Entsprechung in C-Programmen, so daß Compiler nur einen Teil der Fähigkeiten eines DSPs wirklich ausnutzen. Ist der durch einen Compiler generierte Code nicht akzeptabel (was für Echtzeitanwendungen meist der Fall ist), so ergeben sich folgende Möglichkeiten:

Schnellere Prozessoren: Der durch den Compiler verursachte Effizienzverlust wird durch die Verwendung eines schnelleren DSPs kompensiert. Dieser "naive" Ansatz erhöht allerdings die Kosten, den Chip-Flächenverbrauch sowie die Leistungsaufnahme und scheidet daher in der Praxis meist aus.

C-Erweiterungen: Man verwendet DSP-spezifische Erweiterungen der Programmiersprache. Der Intermetrics C-Compiler für den NEC 77016 DSP bspw. [2] bietet C-Konstrukte für Festkommazahlen, zyklische Speicheradressierung und Hardware-Schleifen. Diese ermöglichen eine bessere Ausnutzung der Maschineninstruktionen durch den Compiler, verringern aber die Portierbarkeit von Programmen.

Assemblerprogrammierung: Die derzeit am häufigsten verwendete Methode ist der (zumindest teilweise) Einsatz von Assemblerprogrammierung. Diese bietet die bestmögliche Ausnutzung der DSP-Hardware, macht aber die Programmentwicklung und die Wiederverwendung von Software bei einem Wechsel zu einem anderen Prozessor äußerst schwierig.

Insgesamt existiert also keine wirklich befriedigende Compilerunterstützung für DSPs. Eine Lösung dieses Problems wäre der Einsatz von besseren *Optimierungstechniken* in Compilern. Ein Compiler ist *optimierend*, wenn er sich bei der Codeerzeugung nicht ausschließlich an der Struktur des Programm-Quelltextes orientiert, sondern die speziellen (für den Programmierer nicht offensichtlichen) Eigenschaften eines Programms analysiert und ausnutzt, um besseren Code zu erzeugen.

In einem früheren Beitrag [3] haben wir den Compiler-Prototyp RECORD vorgestellt, wobei der Schwerpunkt auf dem Aspekt der *Retargierbarkeit* von Compilern lag, d.h. auf der automatischen Anpassung an neue Prozessoren. Die hierauf erhaltenen Reaktionen haben bestätigt, daß ein großer Bedarf an besseren DSP-Compilern besteht. In der Tat wurden in den letzten Jahren bedeutende Fortschritte erzielt aber bisher zu wenig in die Praxis umgesetzt. Das Ziel dieses Beitrags ist es daher, einen allgemeinen und intuitiven Überblick über neue Optimierungstechniken für DSP-Compiler zu geben. Dieses Gebiet befindet sich derzeit im Stadium der Forschung und Entwicklung, so daß keine Verweise auf fertige Compilerprodukte gegeben werden können. Viele der vorgestellten Techniken lassen sich aber – unabhängig vom Compiler – auf relativ einfache Art und Weise als effektive "Zusatzoptimierungen" implementieren.

1.2 Wer braucht optimierende Compiler ?

Arbeiten an optimierenden Compilern für DSPs finden – außer im Universitätsbereich – derzeit hauptsächlich in drei Industriezweigen statt:

Systemhäuser: Die Auswirkungen schlechter Compiler für eingebettete Prozessoren (z.B. zu große on-chip Programmspeicher) sind für Systementwickler unmittelbar sichtbar. Die Compiler-Performance hat direkte Auswirkungen auf die Chipausbeute, so daß optimierende Compiler neben effizienten Hardwaresynthese-Tools von großer Bedeutung sind. Beispielsweise werden bei Philips hochoptimierende Compiler im Bereich *digital audio broadcasting* (DAB) entwickelt und eingesetzt [4].

Halbleiterhersteller: Die Verfügbarkeit eines (guten) C-Compilers kann ein entscheidendes Verkaufsargument für einen DSP sein. Einige DSP-Hersteller (z.B. Analog Devices) bieten Derivate des GNU C Compilers für ihre Produkte an. Andere (z.B. Texas Instruments) setzen auf Eigenentwicklungen. SGS Thomson verfügt über einen optimierenden C-Compiler für einen MPEG-2 VLIW DSP [5].

CAD-Hersteller: Im Bereich der CAD-Werkzeuge findet eine Ausrichtung zur Entwurfsunterstützung für komplette VLSI-*Systeme* statt, d.h. unter Verwendung von komplexen vordefinierten *building blocks* oder *cores*. Inzwischen ist eine Vielzahl von Prozessoren [8] in Form von VHDL/Verilog-Modellen oder technologiespezifischen Layout-Makrozellen verfügbar (siehe z.B. die neuen "DesignWare" Produkte von Synopsys). Dies bedeutet, daß zukünftig Compiler in CAD-Frameworks integriert werden müssen. Unter Beteiligung von Synopsys werden z.Zt. Compileroptimierungstechniken für Motorola und Texas Instruments DSPs entwickelt [6]. Die Integration von CAD und Compilern wird auch bei Cadence betrachtet [7].

2 Ansatzpunkte der Codeoptimierung

Codeoptimierungen können auf verschiedenen *Abstraktionsebenen* ansetzen. Für die oberste Ebene ist bereits der Programmierer verantwortlich, da die konkrete Implementierung eines DSP-Algorithmus Einfluß auf die erzielbare Codequalität hat.

2.1 Standard-Optimierungen

Eine Reihe von Optimierungstechniken ist bereits seit langem im Compilerbau bekannt [9]. Hierzu zählen *common subexpression elimination*, *strength reduction* und *constant folding*, wodurch redundante Rechenoperationen eliminiert werden. Solche Techniken operieren auf dem Quellprogramm und sind daher prinzipiell in allen Compilern einsetzbar. DSP-spezifische Eigenheiten können allerdings hier nicht ausgenutzt werden.

Zu den wichtigen Standardverfahren zählen auch *Schleifenoptimierungen* wie das Verschieben von schleifeninvariantem Code (*code motion*). Fortgeschrittene Techniken wie *loop folding* ermöglichen eine bessere Ausnutzung paralleler Befehle (siehe 2.5). Die Ersetzung der Schleife

```
for (i=1; i<=10; i++)
{ a[i] = x[i] + 7;
  b[i] = a[i] * 2; }
```

durch

```
a[1] = x[1] + 7;
for (i=2; i<=10; i++)
{ a[i] = x[i] + 7;
  b[i-1] = a[i-1] * 2; }
b[10] = a[10] * 2;
```

eliminiert die Abhängigkeit der Berechnung von `b[i]` von `a[i]` durch eine "Faltung" der Berechnung von `b[i]` um die Schleifengrenzen. Hierdurch kann die parallele Ausführung der Addition und der Multiplikation (*multiply-accumulate*) ermöglicht werden.

2.2 Die interne Programmdarstellung

Programme können als eine Folge von *Basisblöcken* dargestellt werden. Ein Basisblock ist eine Folge von Zuweisungen, zwischen denen jeweils keine Verzweigung stattfindet. Der Datenfluß in einem Basisblock kann durch *Ausdrucksbäume* repräsentiert werden wie in Abb. 1 veranschaulicht. Ausdrucksbäume bilden die Grundlage nahezu aller Codeerzeugungstechniken im Compilerbau. Der Grund dafür ist, daß für Bäume *optimaler* Code effizient generiert werden kann [9]. Der Nachteil ist allerdings, daß Optimalität nur innerhalb jedes einzelnen Baums erzielt wird.

Durch Zusammenfassung von identischen Variablen oder Berechnungen in den Ausdrucksbäumen eines Blocks kommt man zu einer verallgemeinerten Blockdarstellung in Form eines *gerichteten azyklischen Graphen* (GAG) (Abb. 1 c). Im Gegensatz zu Ausdrucksbäumen gibt es für GAGs keine effizienten Verfahren zur optimalen Codeerzeugung. Allerdings bieten GAGs ein höheres Optimierungspotential, da die durch die Programm-Statements vorgegebene "künstliche" Trennung von Berechnungen aufgehoben wird. Neuere Verfahren betrachten daher Codeerzeugung für GAGs, bspw. mittels einer *branch-and-bound*-Prozedur

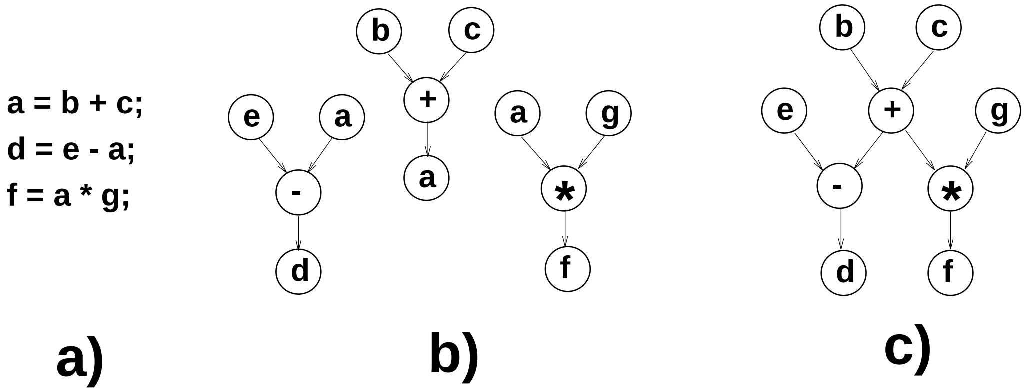

Abbildung 1: *a) Basisblock, b) Baumdarstellung, c) GAG-Darstellung*

[10]. Hierdurch lassen sich erhebliche Einsparungen in der Codegröße von Basisblöcken erzielen, allerdings auf Kosten höherer Übersetzungszeiten. Eine weitere wichtige Technik ist die geschickte Aufspaltung von GAGs in Ausdrucksbäume derart, daß ein Verlust an Codequalität größtenteils vermieden wird [11]. Damit lassen sich die Vorteile von GAGs und Ausdrucksbäumen kombinieren.

2.3 Phasenkopplung

Traditionell wird die Codeerzeugung zerlegt in die Phasen *Befehlsauswahl*, *Registerallokation* und *Scheduling*. Für diese Einzelphasen existieren im Standard-Compilerbau zufriedenstellende Techniken, z.B. *pattern matching* und *Graph-Färbung*. Die strikte Trennung dieser Phasen bewirkt allerdings bei DSPs, welche typischerweise eine sehr irreguläre Befehlssatz- und Registerarchitektur aufweisen, einen großen Verlust an Codequalität: Findet die Befehlsauswahl getrennt von der Registerallokation statt, so werden Spezialregister schlecht ausgenutzt. Die Trennung von Befehlsauswahl und Scheduling führt zu einer mangelnden Ausnutzung potentieller Parallelität.

Neue DSP-spezifische Optimierungsverfahren setzen daher auf *Phasenkopplung*, d.h. verschiedene Phasen werden simultan ausgeführt. Die negativen Auswirkungen der wechselseitigen Abhängigkeiten zwischen den Phasen werden somit vermindert. Die optimale Kopplung von Befehlsauswahl und Registerallokation auf Ausdrucksbäumen läßt sich durch den Einsatz von *pattern matching mit dynamischer Programmierung* realisieren [12]. Diese Technik wurde in mehreren neuen DSP-Compilerprojekten erfolgreich verwendet.

Ein Verfahren zur Kopplung von Befehlsauswahl und Scheduling wird in [13] beschrieben. Die Idee ist, die Befehlsauswahl teilweise zu verzögern, bis genügend Informationen über das Scheduling verfügbar sind. Z.B. verfügt der TI TMS320C2x DSP über die Multiplikationsbefehle "MPY" (*multiply*) und "MPYA" (*multiply-accumulate*). Welcher dieser Befehle für eine Multiplikation sinnvollerweise auszuwählen ist, kann erst während des Schedulings entschieden werden: Kann eine Addition gleichzeitig ausgeführt werden, so ist "MPYA" insgesamt schneller. Ansonsten muß "MPY" gewählt werden, um unerwünschte Seiteneffekte zu vermeiden. Ein nicht-optimierender Compiler wählt stets "MPY", um auf der sicheren Seite zu bleiben.

2.4 Speicheradressierung

DSPs verfügen typischerweise nur über sehr wenige Adressierungsarten, so daß die Codequalität – im Gegensatz zu Mikrocontrollern – durch die Auswahl von Adressierungsarten nur gering beeinflußt wird. Möglichkeiten zur Codeoptimierung ergeben sich in erster Linie dadurch, daß DSPs parallele *post-modify*-Operationen (z.B. *auto-increment*) auf Adreßregistern sowie das parallele Umschalten zwischen verschiedenen Adreßregistern unterstützen. Für die Codequalität ist es daher von großer Bedeutung, wie die Programmvariablen im Speicher *angeordnet* sind und wie die verfügbaren Adreßregister den Speicherzugriffen im Programm *zugeordnet* sind.

Nicht-optimierende Compiler legen die Variablen typischerweise einfach in der Reihenfolge ihrer Deklaration im Speicher ab, wodurch sich ein Overhead von Instruktionen für die Adreßberechnung ergibt. Es wurden daher Algorithmen entwickelt, die für ein gegebenes Programm eine (fast) optimale Anordnung von Variablen im Speicher berechnen [14, 15]. Solche Verfahren ermöglichen eine sehr hohe Ausnutzung von parallelen post-modify-Operationen. Erweiterungen dieser Verfahren beziehen auch evtl. verfügbare *Indexregister* in die Optimierung ein [16, 17]. Gegenüber einer nicht-optimierten Variante lassen sich hierdurch ca. 70 % der Instruktionen für die Adreßberechnung einsparen.

Die Zuordnung von Adreßregister zu Speicherzugriffen ist wesentlich zur *Adressierung von Array-Elementen* in Programmschleifen. Array-Zugriffe in DSP-Programmen folgen häufig einem sehr regulären Schema. Bei geschickter Zuordnung von Adreßregistern zu Array-Zugriffen können diese daher ebenfalls durch post-modify-Operationen auf Adreßregistern (anstelle von expliziten Index-Berechnungen) nachgebildet werden. Z.B. läßt sich für die Schleife

```
for (i=2; i<=N; i++)
{ ref A[i+1]
  ref A[i]
  ref A[i+2]
  ref A[i-1]
  ref A[i+1]
  ref A[i]
  ref A[i-2] }
```

(dabei steht `ref A[..]` für einen Zugriff auf ein Element eines Arrays `A`) eine Zuordnung zu 3 Adreßregistern derart finden, daß *ausschließlich* parallele auto-increment/decrement-

Operationen benötigt werden, so daß sich kein Overhead für die Adreßberechnung im Schleifenrumpf ergibt. Verfahren hierzu finden sich in [18, 19]. In der Ausführungszeit von Schleifen lassen sich damit 30-50 % gegenüber nicht-optimierenden Compilern einsparen.

Bei DSPs mit parallel zugreifbaren Speicherbänken (z.B. X- und Y-Memory in Motorola 56k DSPs) ist außerdem die *Zuordnung von Variablen zu Speicherbänken* wichtig. Während derzeitige Compiler häufig alle Variablen nur einer Bank zuweisen, wurden Verfahren entwickelt, welche diese Zuweisung gezielt anhand von Scheduling-Informationen vornehmen [20]. Auch dies ist ein Beispiel für Phasenkopplung.

2.5 Parallele Befehle

Parallele Befehle, d.h. Befehle, die mehrere Operationen "gleichzeitig" ausführen, sind bei DSPs zur Performance-Steigerung sehr verbreitet. Insbesondere neuere DSPs wie der TI TMS320C6x weisen eine sehr hohe "VLIW-artige" Parallelität auf Befehlsebene auf. Die Analyse von im Rahmen des DSPStone-Projektes [1] generierten Assemblercodes zeigt allerdings, daß derzeitige C-Compiler für DSPs sehr beschränkte Fähigkeiten zur Ausnutzung von parallelen Befehlen haben. Potentielle Parallelität wird im wesentlichen nur bei der Speicheradressierung ausgenutzt, während parallele "moves" und arithmetische Operationen vernachlässigt werden. Der Grund hierfür ist ein *starres Übersetzungsschema*, welches zwar C-Pointer-Arithmetik (wie "`*ptr++`") erkennen und effektiv in auto-increment-Befehle auf Adreßregistern transformieren kann, aber weitere Möglichkeiten zur Parallelisierung nicht überprüft.

Die effektive Parallelisierung von Assemblerbefehlen wird vor allem dadurch erschwert, daß eine *Neuanordnung von Befehlen* notwendig sein kann. Man betrachte den C-Ausdruck

```
a * b + c * d - i * (e * f + g * h)
```

Die Befehlsauswahl und Registerallokation für diese Berechnung würde auf einem TI 'C25 DSP den folgenden symbolischen Assemblercode ergeben:

```
(1)  LT g         // T-Register := g
(2)  MPY h        // P-Register := T-Register * h
(3)  PAC          // Accumulator := P-Register
(4)  LT e         // T-Register := e
(5)  MPY f        // P-Register := T-Register * f
(6)  APAC         // Accumulator := Accumulator + P-Register
(7)  SACL temp    // temp := Accumulator
(8)  LT c         // T-Register := c
(9)  MPY d        // P-Register := T-Register * d
(10) PAC          // Accumulator := P-Register
(11) LT a         // T-Register := a
(12) MPY b        // P-Register := T-Register * b
```

```
(13) APAC        // Accumulator := Accumulator + P-Register
(14) LT temp     // T-Register := temp
(15) MPY i       // P-Register := T-Register * i
(16) SPAC        // Accumulator := Accumulator - P-Register
```

Verschiedene Befehlspaare können nun zu jeweils einem einzigen Befehl zusammengefaßt werden. Z.B. können die Befehle (6) und (9) zu `MPYA d` kombiniert werden, allerdings unter der Voraussetzung, daß Befehl (8) nun vorher ausgeführt wird. Durch solche Parallelisierungen läßt sich insgesamt die Codelänge für den obigen C-Ausdruck von 16 auf 12 Befehle verringern, d.h. um immerhin 25 %.

Eine allgemeine Methode zur Ausnutzung paralleler Befehle ist die *Code-Kompaktierung*, für die sowohl heuristische als auch optimale Verfahren entwickelt wurden. Diese verwenden eine feinkörnige Darstellung von Maschinenbefehlen in Form von einzelnen *Register-Transfers*. Das Grundprinzip ist, unter Beachtung der wechselseitigen Abhängigkeiten jeden auszuführenden Register-Transfer einem möglichst "frühen" Maschinenbefehle zuzuordnen. Für digitale Filteralgorithmen auf einem Analog Devices ADSP-21xx können durch Code-Kompaktierung 40-55 % der Befehle gegenüber rein sequentiellem Assemblercode eingespart werden [21].

3 Ergebnisse in der Praxis

Die beschriebenen Codeoptimierungstechniken für DSPs sind in verschiedenen (bisher nicht-kommerziellen) Compilern erfolgreich eingesetzt worden. Als Beispiele seien hier genannt:

- Für Audio-Dekompression auf einem VLIW DSP wurde ein Overhead in der Codegröße von 0-25 % (je nach Programmierstil) von compilergeneriertem Code gegenüber handgeschriebenem Assemblercode erzielt [5].
- Durch die im Record-Compiler verwendeten Optimierungsverfahren [22] konnte der Overhead in der Codegröße gegenüber dem TI TMS320C25 C-Compiler im Durchschnitt auf ca. 75 % halbiert werden.
- Bei Philips konnte für DAB-Algorithmen in vielen Fällen nahezu optimaler Code für anwendungsspezifische DSPs erzeugt werden [4].
- Im Rahmen des SPAM-Projektes [20] wurden – im wesentlichen durch Phasenkopplung – für einen Motorola DSP 56k teilweise Verringerungen der Codegröße von 50 % gegenüber handgeschriebenem Assemblercode erreicht.

4 Ausblick

Diese und andere Ergebnisse zeigen, daß in der Compilertechnologie für DSPs noch erhebliche Leistungssteigerungen möglich sind. Gegenüber der heute üblichen Assemblerprogrammierung bieten C-Compiler höhere Produktivität bei der Softwareentwicklung und bessere Portierbarkeit von Programmen. Höchstwahrscheinlich wird daher optimierenden C-Compilern zukünftig eine bedeutende Rolle in der Entwicklung von eingebetteter DSP-Software zukommen.

Um in absehbarer Zeit zu praktisch verwendbaren optimierenden C-Compilern für DSPs zu gelangen, sollten nach unserer Ansicht vor allem folgende Punkte beachtet werden:

1. Die meisten Forschungsarbeiten zur DSP-Codeoptimierung stammen z. Zt. aus dem Bereich *Design Automation*, oft ohne ausreichendes Hintergrundwissen aus dem Compilerbau. Dagegen werden DSPs im Bereich des Compilerbaus vernachlässigt. Vorteilhaft wäre stattdessen eine engere Kooperation zwischen diesen – traditionell strikt getrennten – Forschungszweigen. Des weiteren müssen neue Techniken natürlich auch von Compilerentwicklern umgesetzt werden.

2. Hohe Codequalität läßt sich hauptsächlich auf Kosten hoher Übersetzungszeit erreichen. Compiler sollten daher über eine Reihe von verschieden aufwendigen – z.B. über Kommandozeilenoptionen einstellbaren – Optimierungsstufen verfügen. Während der Entwicklungsphase eines Programms kann dann mit "niedriger" (und daher schneller) Optimierungsstufe gearbeitet werden, während abschließend ein evtl. sehr langsamer Compilerlauf mit aufwendiger Optimierung durchgeführt wird, um höchste Codequalität zu erzeugen.

3. Die Hauptschwierigkeit in der DSP-Codeoptimierung liegt darin, Hochsprachen-Konstrukte effizient auf die meist irreguläre Architektur eines DSPs abzubilden. Es wäre daher günstig, "compiler-freundliche" Prozessoren zu entwickeln. Der neue TI 'C6x DSP ist ein Schritt in diese Richtung. Dieser DSP ist zwar aufgrund seiner *Parallelität* manuell nicht einfach zu programmieren, er ist aber durch seine *Regularität* besser zugänglich für bewährte Standard-Optimierungstechniken. Zumindest nach Aussage von TI erreicht der 'C6x C-Compiler daher 70-80 % der Qualität von handgeschriebenem Code, was einen echten Fortschritt darstellen würde.

Literatur

[1] M. Willems, V. Zivojnovic, H. Meyr: *DSP-Compiler: Produktqualität für kontrolldominierte Anwendungen ?*, DSP Deutschland '96, pp. 49-56

[2] Intermetrics: *NEC 77016 DSP C Compiler – Product Description*, Intermetrics Microsystems Software Inc., Cambridge (Mass.), 1996

[3] R. Leupers, P. Marwedel: *Flexible Compiler-Techniken für anwendungsspezifische DSPs*, DSP Deutschland '96, pp. 17-26, sowie: Design & Elektronik 23/96, MagnaMedia Verlag, München, 1996

[4] M. Strik, J. van Meerbergen, A. Timmer, J. Jess, S. Note: *Efficient Code Generation for In-House DSP Cores*, European Design and Test Conference (ED & TC), 1995, pp. 244-249

[5] C. Liem, A. Jerraya et al.: *An Embedded System Case Study: The Firmware Development Environment for a Multimedia Audio Processor*, 34th Design Automation Conference (DAC), 1997

[6] G. Araujo, S. Devadas, K. Keutzer et al.: *Challenges in Code Generation for Embedded Processors*, in: P. Marwedel, G. Goossens (eds.): *Code Generation for Embedded Processors*, Kluwer Academic Publishers, 1995

[7] J.A. Rowson, M. Hartoog, P. Reddy: *Generation of Software Tools from Processor Descriptions for Hardware/Software Codesign*, 34th Design Automation Conference (DAC), 1997

[8] World Wide Web: "http://www.eedesign.com/EEdesign/SoftCoretables.html" und "http://www.eedesign.com/EEdesign/HardCoretables.html"

[9] A.V. Aho, R. Sethi, J.D. Ullman: *Compilerbau*, Addison-Wesley, 1988

[10] S. Liao, S. Devadas, K. Keutzer et al.: *Code Optimization Techniques for Embedded DSP Microprocessors*, 32nd Design Automation Conference (DAC), 1995, pp. 599-604

[11] G. Araujo, S. Malik, M. Lee: *Using Register Transfer Paths in Code Generation for Heterogeneous Memory-Register Architectures*, 33rd Design Automation Conference (DAC), 1996

[12] A.V. Aho, M. Ganapathi, S.W.K Tjiang: *Code Generation Using Tree Matching and Dynamic Programming*, ACM Trans. on Programming Languages and Systems 11, no. 4, 1989, pp. 491-516

[13] R. Leupers, P. Marwedel: *Instruction Selection for Embedded DSPs with Complex Instructions*, European Design Automation Conference (EURO-DAC), Geneva (Switzerland), 1996, pp. 200-205

[14] D.H. Bartley: *Optimizing Stack Frame Accesses for Processors with Restricted Addressing Modes*, Software – Practice and Experience, vol. 22(2), 1992, pp. 101-110

[15] S. Liao, S. Devadas, K. Keutzer et al.: *Storage Assignment to Decrease Code Size*, Conference on Programming Language Design and Implementation (PLDI), 1995

[16] R. Leupers, P. Marwedel: *Algorithms for Address Assignment in DSP Code Generation*, Int. Conf. on Computer-Aided Design (ICCAD), San Jose (USA), 1996, pp. 109-112

[17] B. Wess, M. Gotschlich: *Constructing Memory Layouts for Address Generation Units Supporting Offset 2 Access*, Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP), 1997

[18] G. Araujo, A. Sudarsanam, S. Malik: *Instruction Set Design and Optimizations for Address Computation in DSP Architectures*, 9th Int. Symp. on System Synthesis (ISSS), 1996

[19] C. Liem, P.Paulin, A. Jerraya: *Address Calculation for Retargetable Compilation and Exploration of Instruction-Set Architectures*, 33rd Design Automation Conference (DAC), 1996

[20] A. Sudarsanam, S. Malik: *Memory Bank and Register Allocation in Software Synthesis for ASIPs*, Int. Conf. on Computer-Aided Design (ICCAD), 1995, pp. 388-392

[21] B. Wess: *Translating Expression DAGs into Optimized Code for non-homogeneous Register Machines*, 2nd Int. Workshop on Embedded Code Generation, Leuven/Belgium, March 1996

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