Flexible Compiler-Techniken für anwendungsspezifische DSPs

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

Abstract–*Unzureichende Codequalität ist bekanntermaßen ein Hauptproblem derzeit verfügbarer Hochsprachen-Compiler für DSPs. Die Notwendigkeit neuer DSP-spezifischer Compilertechniken wird darüber hinaus durch den Trend zu anwendungsspezifischen DSPs verstärkt. Für diese sind nicht nur optimierende, sondern auch flexible Compiler erforderlich, die mit geringem Aufwand an veränderte Zielarchitekturen anpaßbar sind. In diesem Beitrag beschreiben wir einen Ansatz zur automatischen Erzeugung von Codegeneratoren aus hardwarenahen, leicht änderbaren Prozessormodellen. Des weiteren stellen wir DSP-spezifische Code-Optimierungstechniken vor, die auf eine bestmögliche Ausnutzung potentieller Parallelität abzielen.*[1]

1 Einleitung

Während für interaktive Computersysteme wie PCs und Workstations zufriedenstellende Software-Entwicklungswerkzeuge zur Verfügung stehen, konnte sich der Einsatz von Hochsprachen-Compilern für auf programmierbaren DSPs basierende Systeme in der industriellen Praxis bisher nicht durchsetzen. Eine neuere Studie über DSP-Softwareentwicklung bei Bell Northern Research [1] zeigt, daß etwa 90 % der Software noch immer in Assemblersprachen erstellt werden. Trotz der hinreichend bekannten Nachteile der Assemblerprogrammierung bzgl. Wartbarkeit, Fehleranfälligkeit und Portierbarkeit wird diese nach wie vor der Programmentwicklung in Hochsprachen vorgezogen. Ein Hauptgrund hierfür ist die unzureichende Codequalität von verfügbaren Compilern im Bezug auf Realzeitanforderungen. Dies wird von vielen industriellen Anwendern bestätigt, ebenso wie durch eine systematische Studie an der RWTH Aachen [2]. Die unzureichende Codequalität wird wiederum durch die Tatsache verursacht, daß DSPs während ihres Entwurfs traditionell als Prozessoren betrachtet wurden, die in Assembler programmiert werden *sollen*. Die Ausnutzung DSP-spezifischer Hardware-Charakteristika (z.B. Spezialregister, Speicheradressierung, Parallelität) wird im Standard-Compilerbau [3, 4] praktisch nicht behandelt, so daß der Versuch, Standard-Compilertechniken auf DSPs zu übertragen, im allgemeinen in sehr ineffizientem Code resultiert.

Ein zweiter Grund für die mangelnde Verbreitung von Hochsprachen-Compilern im DSP-Bereich liegt in der Tatsache, daß die *Entwicklung* eines Compilers nur für Prozessoren mit sehr hohen Stückzahlen, d.h. Standard-DSPs, lohnenswert ist. Zu beobachten ist jedoch ein Trend zu *anwendungsspezifischen* programmierbaren Prozessoren (*application-specific instruction-set processors, ASIPs*), mit vergleichsweise kleinen Stückzahlen und kurzen Lebensdauern am Markt [1]. ASIPs kommen typischerweise als Komponenten in *Ein-Chip-Systemen* zum Einsatz und liegen daher in Form von Layout-Makrozellen (*cores*) vor. Obwohl eine zunehmende Anzahl von Standard-DSPs verschiedener Hersteller (z.B. Texas Instruments, LSI Logic) ebenfalls in Form von cores verfügbar ist, sind speziell auf einen Anwendungsbereich abgestimmte ASIPs häufig die effizientere Alternative bzgl. Chipflächenverbrauch, Leistungsaufnahme und Verarbeitungsgeschwindigkeit. Da diese aber in der Regel firmen- oder sogar abteilungsspezifische Entwicklungen sind, und Ressourcen zur Compilerentwicklung meist nicht vorhanden sind, werden die Vorteile der Verwendung von ASIPs durch den Zwang zur aufwendigen Programmierung auf Maschinenebene erkauft.

Es ist zu erwarten, daß bessere Verfügbarkeit und bessere Codequalität von DSP-Compilern zur Produktivitätssteigerung zwingend notwendig werden. Wie bei interaktiven Systemen steigt auch bei DSP-Systemen (oder allgemeiner in *eingebetten Systemen*) der Anteil der Software – sowohl bzgl. Funktionalität als auch Entwicklungszeit – gegenüber dem der Hardware ste-

[1] DSP Deutschland '96, München, Oktober 1996,

tig an. Dies bedeutet, daß die Assemblerprogrammierung in Zukunft nicht mehr handhabbar sein wird. Des weiteren trägt bei Ein-Chip-Systemen die (in ROMs vorliegende) Software wesentlich zur Chipfläche bei, so daß die Codequalität sich indirekt auf die Chipausbeute auswirkt.

Neue, effektive Compiler-Techniken für eingebettete Prozessoren, speziell DSPs, sind daher in den letzten Jahren Gegenstand intensiver Forschung geworden [5]. Zur Zeit zeichnen sich folgende allgemeine Ansätze zur Lösung der obigen Probleme ab:

Berücksichtigung von Compiler-Anforderungen: Die Schnittstelle zwischen Hardware und Software im DSP-Bereich zeigt Schwachstellen, die im Workstation-Bereich zur Ersetzung von CISC-Prozessoren durch RISCs geführt haben: Spezielle, optimierte Machinenbefehle werden durch Compiler nicht oder nur unzureichend ausgenutzt, da die Umsetzung von Hochsprachen-Anweisungen in diese Befehle extrem schwierig ist. Werden die zu erwartenden Fähigkeiten von Compilern bereits während des Entwurfs eines DSPs berücksichtigt, so lassen sich derartige Ineffizienzen vermeiden. Dies erfordert jedoch eine engere Kooperation zwischen Hardwareentwicklern und Compilerbauern.

Spezielle Programmiersprachen: Die Diskrepanz zwischen Hochsprachen und DSP-Maschinenbefehlen kann durch den Einsatz von speziellen Programmiersprachen beseitigt werden. DSP-spezifische Sprachen wie SILAGE [6] oder DFL [7] haben allerdings keine große Akzeptanz gefunden, da die Umstellung vom de-facto-Standard "C" zu aufwendig erscheint. Vielversprechender sind daher *Erweiterungen* existierender Sprachen. Wie in [8] betont, müssen diese jedoch sorgfältig ausgewählt werden, um noch eine ausreichende Abstraktion von der Zielmaschine zu gewährleisten.

Flexibilität: Der Einsatz von Compilern auch für ASIPs mit geringen Stückzahlen ist möglich, wenn Compiler hinreichend flexibel sind, d.h. für eine Vielzahl möglicher Prozessorarchitekturen innerhalb einer definierten Architekturklasse Code erzeugen können. Ist das "Grundgerüst" des Compilers einmal erstellt, so kann er mit vergleichsweise geringem Aufwand an neue Zielarchitekturen angepaßt werden. Derartige Compiler werden als *retargierbar* bezeichnet. Hierbei ist es wichtig, daß das Retargieren durch den *Benutzer* erfolgen kann, was z.B. für den (durchaus flexiblen) GNU C Compiler [9] nicht der Fall ist. Ist das Retargieren sehr schnell möglich, so kann der Compiler sogar zur Optimierung der Zielarchitektur beitragen, da dann die Auswirkungen verschiedener Hardwaremaßnahmen auf die Codequalität mit Hilfe des Compilers sehr schnell studiert werden können (Stichwort: Hardware-Software Codesign).

Spezifische Optimierungstechniken: DSPs besitzen eine Reihe von Hardware-Merkmalen, die in Allzweck-Prozessoren nicht vorkommen. Über die oben erwähnten Spracherweiterungen hinaus ist es daher notwendig, *neue* Code-Optimierungstechniken zu entwickeln, die den Anforderungen von DSPs gerecht werden. Das hohe Potential neuer Optimierungstechniken resultiert aus den besonderen Randbedingungen im DSP-Bereich: Während im Standard-Compilerbau hohe Anforderungen vor allem an die *Übersetzungsgeschwindigkeit* gestellt werden, steht bei DSPs die *Codequalität* klar im Vordergrund. Dies rechtfertigt den Einsatz von relativ aufwendigen Optimierungsalgorithmen.

Dieser Beitrag befaßt sich mit den beiden letztgenannten Punkten. Innerhalb des ESPRIT-Projekts 9138 ("CHIPS") arbeiten wir an Methoden zur Entwicklung von *durch den Benutzer retargierbaren* und *optimierenden* Compilern für DSPs. Im folgenden Abschnitt stellen wir zunächst den von uns realisierten Ansatz zur Retargierbarkeit vor, und vergleichen diesen mit bestehenden Ansätzen. Das Retargieren besteht aus der Modellierung des Zielprozessors sowie der automatischen Erzeugung eines prozessorspezifischen Codegenerators aus dem Modell. In Abschnitt 3 wird die Verwendung des generierten Codegenerators für die Abbildung von DSP-Hochsprachenprogrammen in verfügbare Maschinenbefehle erläutert. In Abschnitt 4 stellen wir neue Methoden zur Codeoptimierung vor, die insbesondere auf die Ausnutzung von potentieller Parallelität in Maschinenprogrammen abzielen.

2 Retargierung

Kernstück eines Compilers ist der *Codegenerator*, der die Abbildung einer Zwischenrepräsentation eines Programms auf verfügbare Maschinenbefehle vornimmt. Um leichte Retargierbarkeit zu gewährleisten, so muß die gewählte Implementierung möglichst *maschinenunabhängig* sein. Dies widerspricht jedoch i.a. den Anforderungen bzgl. Codequalität und auch Übersetzungsgeschwindigkeit.

In unserem Ansatz (Abb. 1) versuchen wir, diesen Zielkonflikt durch *automatische* und *maschinenun-*

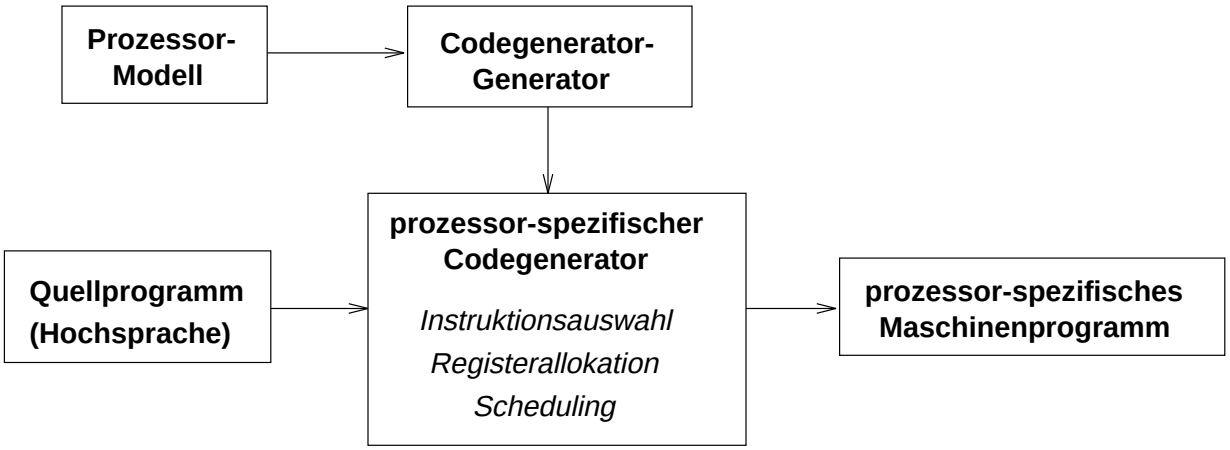

Abbildung 1: *Prinzip der retargierbaren Codeerzeugung*

abhängige Erzeugung von *maschinenspezifischen* Codegeneratoren zu beseitigen. Ausgehend von einem formalen, vom Benutzer erstellten Prozessormodell wird ein prozessorspezifischer Codegenerator automatisch erzeugt. Dieser führt die eigentliche Übersetzung des Quellprogramms in ein ausführbares Maschinenprogramm durch. Dieser Ansatz gewährleistet Retargierbarkeit, da der Benutzer außer dem Formalismus zur Prozessormodellierung keine Details über den Codegenerator selbst kennen muß. Der gewöhnlich mit der Retargierbarkeit verbundene Verlust an Codequalität und Übersetzungsgeschwindigkeit wird weitgehend vermieden, da der erzeugte Codegenerator auf die modellierte Zielmaschine abgestimmt und somit hinreichend effizient ist.

Effiziente Codegenerator-Generatoren sind seit Ende der 80er Jahre aus dem Compilerbau bekannt [10]. Diese erfordern als Eingabe ein Prozessormodell in Form einer *Baum-Grammatik*, welche den Befehlssatz des Prozessors repräsentiert. Beschreibungen in Form von Baum-Grammatiken sind jedoch für den Anwender nur bedingt geeignet, da ihre Erstellung recht aufwendig ist, und diese außerdem sehr weit von den im Hardwareentwurf verbreiteten Beschreibungssprachen entfernt sind. Mit Hilfe einer neuartigen Technik ("*Befehlssatzextraktion*") schließen wir die Lücke zwischen hardwarenahen Prozessormodellen und Baum-Grammatiken.

2.1 Prozessormodellierung

Retargierbarkeit – basierend auf formalen Prozessormodellen – war bereits sehr früh Forschungsgegenstand im Compilerbau [11]. Hieraus resultierende Prozessor-Beschreibungssprachen waren sehr werkzeugspezifisch und beschrieben Prozessoren ausschließlich durch ihr *Verhalten*, d.h. durch den (evtl. um einige Informationen ergänzten) Befehlssatz. Verhaltensmodelle in spezifischen Sprachen finden auch in neueren Ansätzen zur DSP-Codeerzeugung Verwendung [12, 13]. Im Bereich Hardwareentwurf kommen allerdings oft *Strukturmodelle* (z.B. RT-Netzlisten) zum Einsatz. Gegenüber Verhaltensmodellen erlauben Strukturmodelle, insbesondere bei Parallelität auf Befehlssatzebene, eine benutzerfreundlichere Modellierung. Des weiteren erleichtern Strukturmodelle *lokale Änderungen* im Modell, bspw. Hinzufügen oder Entfernen einzelner RT-Komponenten, was insbesondere beim HW/SW Codesign von ASIPs nützlich ist. RT-Strukturmodelle enthalten allerdings wesentlich mehr Details (z.B. Busse, Multiplexer) als für die Codeerzeugung eigentlich benötigt werden.

Ein früherer retargierbarer Compiler (MSSQ, [14]) unserer Forschungsgruppe basierte auf RT-Strukturmodellen und erreichte somit eine enge Kopplung von Codeerzeugung und Hardwareentwurf. Realistische Anwendungsstudien [15] zeigten jedoch, daß in der Praxis sowohl Struktur- als auch Verhaltensmodelle (sowie Mischformen) benötigt werden: Der jeweils geeignetste Prozessor-Modellierungsstil hängt von der einzelnen Anwendung – speziell vom Befehlsformat – ab. Für die Codeerzeugung sind allerdings Verhaltensmodelle am besten geeignet.

Des weiteren ist es wünschenswert, dem Benutzer eine universelle *Hardwarebeschreibungssprache* (anstelle eines werkzeugspezifischen Formates) für die Prozessormodellierung zur Verfügung zu stellen. Die erlaubt eine relativ leichte Modellierung von Prozessoren sowie eine natürliche Anbindung an bestehende CAD-Werkzeuge, bspw. zur Synthese oder Simulation. In unserem Ansatz verwenden wir MIMOLA [16], eine an PASCAL angelehnte Hardwarebeschreibungssprache zur Model-

lierung von programmierbaren Prozessoren. Mögliche Abstraktionsebenen in MIMOLA-Modellen reichen von der Verhaltensebene über die RT-Ebene bis zur Gatterebene. Somit wird eine Vielzahl von Prozessor-Modellierungsstilen unterstützt. MIMOLA enthält viele Elemente von VHDL, weist aber keine starke Typisierung auf und erlaubt somit prägnantere Modelle. Existierende Konverter können aus einer Untermenge von VHDL äquivalente MIMOLA-Modelle erzeugen.

Ein MIMOLA-Prozessormodell besteht aus einer *hierarchischen Netzliste* von Prozessorkomponenten (*Modulen*). Jedes Modul wird durch sein Verhalten beschrieben, oder besitzt wiederum eine interne Modulstruktur. Verhaltensmodule sind typischerweise logische Gatter, RT-Komponenten (Register, RAMs, ROMs, Multiplexer, ALUs, Decoder) oder komplexe Komponenten (Controller, ALUs mit Pipelinestufe). Im Gegensatz zu den meisten anderen Ansätzen sind MIMOLA-Modelle *integriert*, d.h. sie beschreiben sowohl Datenpfad als auch Controller des Prozessors. Unsere derzeitige Compiler-Version unterstützt Modelle von fixed-point DSPs mit fester Instruktionslänge und beliebigem Instruktionsformat. Abb. 2 zeigt als Beispiel das MIMOLA-Modell eines sehr einfachen 8-bit Prozessors.

2.2 Befehlssatzextraktion

Prozessormodellierung in Hardwarebeschreibungssprachen bietet den Vorteil hoher Benutzerfreundlichkeit durch Vielseitigkeit. Andererseits ist es für die Codeerzeugung günstig, ein einheitliches internes Modell zu verwenden, welches von unnötigen Details abstrahiert. Wir führen daher eine *Extraktion des Befehlssatzes* auf dem Prozessormodell durch, um ein für die Codeerzeugung geeignetes internes Modell zu erhalten. Die Funktionalität der Befehlssatzextraktion ist in Abb. 3 dargestellt.

Die Extraktion des Befehlssatzes wird ausgeführt, indem sämtliche Transportwege zwischen Registern und Speichern im Datenpfad (und damit alle möglichen Mikrooperationen) aufgezählt werden. Gleichzeitig werden die *binären Befehlscodierungen* (d.h. partielle Instruktionen) der extrahierten Operationen bestimmt. Dies dient einerseits dem Zweck, nur *gültige* Mikrooperationen zu extrahieren, d.h. solche, die aufgrund der Befehlswortrestriktionen erlaubt sind. Des weiteren bilden die binären Codierungen die Grundlage für die (unten beschriebene) *Code-Kompaktierung*, da sich Befehlskonflikte in den Codierungen widerspiegeln. Die Analyse der binären Codierungen kann bei stark codierten Befehlsformaten recht aufwendig sein und kann auf die Manipulation von Booleschen Funktionen zurückgeführt werden. Wir greifen daher auf die aus der Logiksynthese und -verifikation als effizient bekannten *Binary Decision Diagrams* (BDDs) zurück [17]. Unsere bisherigen experimentellen Ergebnisse zeigen, daß durch den Einsatz von BDDs der Aufwand für die Befehlssatzextraktion durchaus akzeptabel ist: Für kleinere ASIPs werden auf einer SPARC-20 Workstation einige Sekunden benötigt. Für ein komplexeres DSP-Modell (Texas Instruments TMS320C25), bestehend aus 1300 Zeilen MIMOLA-Code, wurden etwa 2 Minuten CPU-Zeit (SPARC-20) gemessen.

Unabhängig vom gewählten Modellierungsstil erzeugt die Befehlssatzextraktion ein einheitliches Verhaltensmodell des Zielprozessors, wobei für die Codeerzeugung irrelevante Details eliminiert werden. Die extrahierten Mikrooperationen werden als *Baum-Muster* repräsentiert. Jedes Muster repräsentiert eine primitive Prozessoroperation, welche (in einem Maschinenzyklus) Werte aus Registern, Speichern oder Ports liest, eine Berechnung ausführt und das Resultat in ein Register, einen Speicher oder einen Port schreibt.

2.3 Erzeugung des Codegenerators

Die extrahierten Baum-Muster lassen sich unmittelbar als Regeln einer Baum-Grammatik darstellen. Wir verwenden das Standardwerkzeug `iburg` [18], um aus einer so erzeugten Baum-Grammatik einen maschinenspezifischen, ausführbaren *Baum-Parser* zu erhalten (Abb. 4). Aus der spezifizierten Baum-Grammatik erzeugt `iburg` (ähnlich zu `yacc`) C-Quellcode, welcher mit einem gewöhnlichen Compiler übersetzt werden kann. Der CPU-Zeitaufwand zur Erzeugung eines Baum-Parsers mittels `iburg` liegt typischerweise im Sekundenbereich. Die Extraktion des Befehlssatzes und die Erzeugung des entsprechenden Baum-Parsers müssen für jeden Zielprozessor nur einmal durchgeführt werden.

3 Codeerzeugung

Wir verwenden eine Zwischenrepräsentation des zu übersetzenden Programms in Form von Datenflußbäumen. Die Instruktionsauswahl für Datenflußbäume durch Baum-Parsen kann als *Pattern Matching* zwischen Datenflußbäumen und Mikrooperationen aufgefaßt werden. Dies wird in Abb. 5 beispielhaft verdeutlicht.

Instruktionsauswahl durch Baum-Parsen ist *optimal* sowohl bzgl. der Ausführungsgeschwindigkeit (linear in der Baumgröße) als auch in der Anzahl generierter Mikrooperationen. Darüber hinaus machen zwei spezielle Eigenschaften Baum-Parser besonders attraktiv für die DSP-Codeerzeugung:

```
MODULE SimpleProcessor (IN inp:(7:0); OUT outp:(7:0));
STRUCTURE IS                    -- aeusseres Modul ist Netzliste
TYPE InstrFormat = FIELDS       -- 21-bit horizontales Befehlsformat
                     imm:       (20:13);
                     RAMadr:     (12:5);
                     RAMctr:        (4);
                     mux:         (3:2);
                     alu:         (1:0);
                   END;
     Byte = (7:0); Bit = (0);  -- skalare Typen

PARTS                           -- instantiiere RT-Verhaltens-Module
 IM: MODULE InstrROM (IN adr: Byte; OUT ins: InstrFormat);
     BEHAVIOR IS
      VAR storage: ARRAY[0..255] OF InstrFormat;
      CONBEGIN ins <- storage[adr]; CONEND;

 PC, REG: MODULE Reg8bit (IN data: Byte; OUT outp: Byte);
          BEHAVIOR IS
           VAR R: Byte;
           CONBEGIN R := data; outp <- R; CONEND;

 PCIncr: MODULE IncrementByte (IN data: Byte; OUT inc: Byte);
         BEHAVIOR IS
          CONBEGIN inc <- INCR data; CONEND;

 RAM: MODULE Memory (IN data, adr: Byte; OUT outp: Byte; FCT c: Bit);
      BEHAVIOR IS
       VAR storage: ARRAY[0..255] OF Byte;
       CONBEGIN
        CASE c OF 0: NOLOAD storage; 1: storage[adr] := data; END;
        outp <- storage[adr];
       CONEND;

 ALU: MODULE AddSub (IN d0, d1: Byte; OUT outp: Byte; FCT c: (1:0));
      BEHAVIOR IS
       CONBEGIN                  -- "%" bezeichnet Binaerzahlen
        outp <- CASE c OF %00: d0 + d1; %01: d0 - d1; %1x: d1; END;
       CONEND;

  MUX: MODULE Mux3x8 (IN d0,d1,d2: Byte; OUT outp: Byte; FCT c: (1:0));
       BEHAVIOR IS
        CONBEGIN outp <- CASE c OF 0: d0;  1: d1; ELSE d2; END; CONEND;

CONNECTIONS -- Verbindungsliste
 -- Controller:                  -- Datenpfad:
 PC.outp       -> IM.adr;         IM.ins.imm ->  MUX.d0;
 PC.outp       -> PCIncr.data;    inp        -> MUX.d1;   -- primaerer input
 PCIncr.inc    -> PC.data;        RAM.outp   -> MUX.d2;
 IM.ins.RAMadr -> RAM.adr;        MUX.outp   -> ALU.d1;
 IM.ins.RAMctr -> RAM.c;          ALU.outp   -> REG.data;
 IM.ins.alu    -> ALU.c;          REG.outp   -> ALU.d0;
 IM.ins.mux    -> MUX.c;          REG.outp   -> outp;     -- primaerer output

END; -- STRUCTURE
```

Abbildung 2: *MIMOLA-Modell eines einfachen 8-bit Prozessors.*

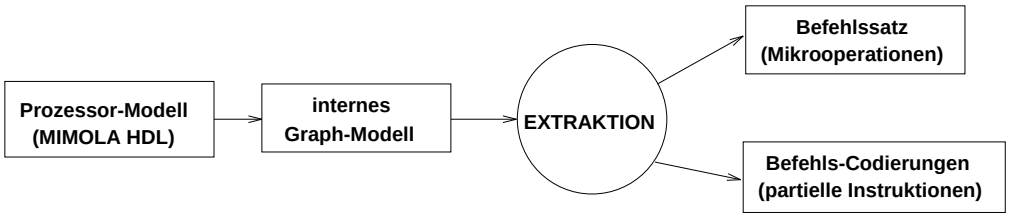

Abbildung 3: *Funktionalität der Befehlssatzextraktion*

Abbildung 4: *Erzeugung von Baum-Parsern aus extrahierten Befehlssätzen*

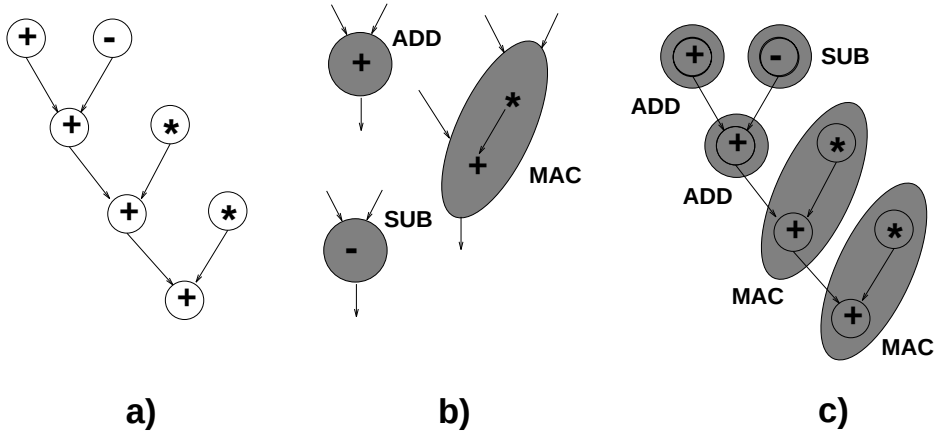

Abbildung 5: *Instruktionsauswahl durch Baum-Parsen: a) Datenflußbaum, b) verfügbare Instruktionsmuster, c) optimale Instruktionsauswahl*

- DSPs verfügen oft über Befehle, die mehrere aufeinanderfolgende Operation in einem Befehlszyklus ausführen (*chained operations*, bspw. multipy-accumulate, store-with-shift). Da die Regeln einer Baum-Grammatik solche Operationen explizit darstellen, ist deren optimale Ausnutzung garantiert.
- Während im Standard-Compilerbau die Registerallokation durch Lebenszeit-Analysen von Variablen dominiert wird, müssen bei DSP-Architekturen, die typischerweise irregulär im Datenpfad verteilte Spezialregister enthalten, die *Transportkosten* von und zu Registern in die Instruktionsauswahl einbezogen werden, d.h. Instruktionsauswahl und Registerallokation sollten für DSPs *simultan* behandelt werden. Durch Einführung von register-spezifischen Regeln können die Transportkosten während des Baum-Parsens implizit minimiert werden. Hierdurch wird die Registerallokation teilweise in die Instruktionsauswahl integriert.

Nach der Instruktionsauswahl wird wird eine Folge von Maschinenbefehlen (*vertikaler Code*) erzeugt. Wir verwenden eine auf [19] aufbauende Technik zur Minimierung des spill codes für Register mit beschränkter Kapazität. Anschließend wird der vertikale Code um Instruktionen erweitert, welche die benötigten Prozessor-Modi (z.B. sign/zero extension, shift mode) aktivieren.

Zur Verabschaulichung betrachten wir das folgenden Beipspielprogramm (`complex_multiply` aus dem DSPStone Benchmark Projekt [2]):

```
cr  = ar * br - ai * bi ;
ci  = ar * bi + ai * br ;
```

Abb. 6 zeigt den vertikalen Code, der für den TI TMS320C25 DSP erzeugt wird. Die Variablen `ar`, `ai`, `br`, `bi`, `cr`, `ci` sind an den Speicher `MEM` gebunden. Mit Hilfe der Spezialregister `TR`, `PR` und `ACCU` werden Real- und Imaginärteil des Resultats getrennt berechnet und im Speicher abgelegt.

4 Ausnutzung von Parallelität

Die Ausnutzung potentieller Parallalität im vertikalen Code geschieht durch zwei Optimierungsphasen. Die *Adreßzuweisung* berechnet ein geeignetes Layout von Programmvariablen im Speicher, um verfügbare Adreßgenerierungshardware möglichst effektiv auszunutzen. Basierend auf dem berechneten Speicherlayout werden Maschinenbefehle zur Adreßerzeugung eingefügt. Anschließend erfolgt die *Code-Kompaktierung*, welche den sequentiellen vertikalen Code in ausführbaren parallelen Code übersetzt.

```
TR = MEM[ar]      // TR = ar
PR = TR * MEM[br] // PR = ar * br
ACCU = PR         // ACCU = ar * br
TR = MEM[ai]      // TR = ai
PR = TR * MEM[bi] // PR = ai * bi
ACCU = ACCU - PR  // ACCU = ar * br - ai * bi
MEM[cr] = ACCU    // cr = ar * br - ai * bi
TR = MEM[ar]      // TR = ar
PR = TR * MEM[bi] // PR = ar * bi
ACCU = PR         // ACCU = ar * bi
TR = MEM[ai]      // TR = ai
PR = TR * MEM[br] // PR = ai * br
ACCU = ACCU + PR  // ACCU = ar * bi + ai * br
MEM[ci] = ACCU    // ci = ar * bi + ai * br
```

Abbildung 6: *Vertikaler Code für* `complex_multiply` *und TMS320C25*

4.1 Adreßzuweisung

Heutige DSPs verfügen über gesonderte Hardware zur Erzeugung von Speicheradressen, die parallel zum Datenpfad arbeitet. Aufgrund der hohen Geschwindigkeitsanforderungen an DSPs ist oft nur ein restriktiver Adressierungsmechanismus implementiert, der über *postmodify* Befehle auf Adreßregistern realisiert ist. Dies bedeutet, daß Adreßarithmetik nur *am Ende* eines Befehlszyklus stattfinden kann. Unterscheidet sich die im nächsten Befehlszyklus benötigte Adresse nur um +1/-1 von der aktuellen Adresse, so kann die nächste Adresse auf vielen DSPs (z.B. Motorola 56xxx, TI TMS320C2x) *kostenlos*, d.h. parallel zu anderen Mikrooperationen, mittels *autoincrement/decrement*-Operationen berechnet werden. Diese Eigenschaft kann ausgenutzt werden, indem eine geeignete *Permutation* von Variablen im Speicher berechnet wird, die von der Variablen-Zugriffsfolge im Programm abhängt. Dies wird in Abb. 7 veranschaulicht. Die Variablenmenge $V = \{a, b, c, d\}$ werde zugegriffen in der Reihenfolge

$$S = (b, d, a, c, d, a, c, b, a, d, a, c, d)$$

Werden die Variablen – wie in gewöhnlichen Compilern – in lexikographischer oder Deklarationsreihenfolge im Speicher abgelegt (Abb. 7 a), so können autoincrement-Operationen nur unzureichend ausgenutzt werden. Addition oder Subtraktion von Werten ungleich 1/-1 zu/von einem Adreßregister (AR) verursachen allerdings jeweils eine zusätzliche, nicht parallelisierbare Instruktion. Setzt man die Kosten für solche Instruktionen mit 1 und die Kosten für autoincrement/decrement-Operationen mit 0 (d.h. parallelisierbar) an, so ergibt sich für das "naive" Layout in Abb. 7 a) ein Kostenwert von 9.

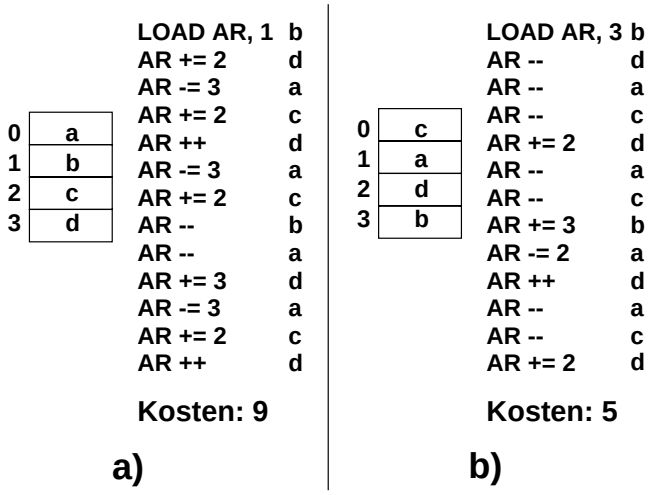

Abbildung 7: *Effekt der Adreßzuweisung: a) Speicherlayout in lexikographischer Reihenfolge, b) optimiertes Layout*

Wird dagegen ein auf die Variablen-Zugriffsfolge S optimiertes Layout berechnet (Abb. 7 b), so lassen sich die Kosten für reine Adreßoperationen auf 5 senken, da die Adreßarithmetik größtenteils auf autoincrement/decrement reduziert wird. Zur Berechnung von guten Speicherlayouts verwenden wir graphbasierte Verfahren [20], welche auf Bartleys [21] und Liaos [22] Arbeiten aufbauen. Hierbei wird eine prozessorabhängige Anzahl k von Adreßregistern ausgenutzt, ebenso wie evtl. verfügbare Index- oder Modify-Register. Durch optimierte Adreßzuweisung lassen sich gegenüber naiver Zuweisung ca. 70 % der für Adreßarithmetik benötigten Instruktionen einsparen. Im realen Maschinencode bedeutet dies eine Einsparung von bis zu 20 %. Die CPU-Zeit zur Adreßzuweisung ist in der Regel vernachlässigbar gering.

Für das obige Beispiel `complex_multiply` wird unter Verwendung eines Adreßregisters `AR[0]` die folgende Adreßzuweisung berechnet:

$$0 \leftrightarrow \texttt{ci}, \quad 1 \leftrightarrow \texttt{br}, \quad 2 \leftrightarrow \texttt{ai},$$

$$3 \leftrightarrow \texttt{bi}, \quad 4 \leftrightarrow \texttt{cr}, \quad 5 \leftrightarrow \texttt{ar}$$

Abb. 8 zeigt den um die hieraus resultierenden Adreßoperationen erweiterten vertikalen Maschinencode.

4.2 Kompaktierung

Obwohl die Code-Kompaktierung, d.h. die Parallelisierung unabhängiger, kompatibler Mikroinstruktionen, bereits seit den frühen 80er Jahren Forschungsgegenstand ist, ist die Ausnutzung von ILP in DSP-Compilern immer noch unzureichend. Bekannte heu-

```
ARP = 0                 // init AR pointer
AR[0] = 5               // AR = 5 (ar)
TR = MEM[AR[ARP]]
AR[ARP] -= 4            // AR = 1 (br)
PR = TR * MEM[AR[ARP]]
ACCU = PR
AR[ARP] ++              // AR = 2 (ai)
TR = MEM[AR[ARP]]
AR[ARP] ++              // AR = 3 (bi)
PR = TR * MEM[AR[ARP]]
ACCU = ACCU - PR
AR[ARP] ++              // AR = 4 (cr)
MEM[AR[ARP]] = ACCU
AR[ARP] ++              // AR = 5 (ar)
TR = MEM[AR[ARP]]
AR[ARP] -= 2            // AR = 3 (bi)
PR = TR * MEM[AR[ARP]]
ACCU = PR
AR[ARP] --              // AR = 2 (ai)
TR = MEM[AR[ARP]]
AR[ARP] --              // AR = 1 (br)
PR = TR * MEM[AR[ARP]]
ACCU = ACCU + PR
AR[ARP] --              // AR = 0 (ci)
MEM[AR[ARP]] = ACCU
```

Abbildung 8: *Vertikaler TMS320C25 Code mit Adreßoperationen*

ristische Kompaktierungsalgorithmen [23] liefern gute Ergebnisse für VLIW-Befehlsformate, sind aber für DSPs nur bedingt geeignet. Dies hat nach unserer Erfahrung zwei Hauptursachen:

1. DSPs besitzen aufgrund der begrenzten Fläche für on-chip-ROMs meist **stark codierte Befehlsformate**, die vergleichsweise wenig Raum für Kompaktierung lassen. Heuristische Verfahren mit begrenztem Suchraum entdecken mögliche Parallelität daher höchstens durch Zufall.

2. Mikrooperationen auf DSPs besitzen oft eine Anzahl von **alternativen Codierungen**, d.h. verschiedene partielle Instruktionen repräsentieren den gleichen Register-Transfer. Alternative Codierungen vergrößern den Suchraum für die Kompaktierung und können *unerwünschte Seiteneffekte* verursachen.

Um diesen Besonderheiten gerecht zu werden, verwenden wir ein *exaktes* Verfahren, welches für beliebige Befehlsformate inklusive alternativer Codierungen optimal kompaktierten Code erzeugt. Da das Kompaktierungsproblem NP-vollständig ist, lassen sich nur Probleme beschränkter Größe optimal bearbeiten. Um

dennoch ein praxisrelevantes Verfahren zu erhalten, verwenden wir eine Formulierung des Kompaktierungsproblems als ganzzahliges Optimierungsproblem (*Integer Programming, IP*) welches mit Standard-Tools gelöst werden kann. IP ist das Problem der Berechnung einer Belegung von n ganzzahligen Lösungsvariablen $(z_1, \ldots, z_n)$, so daß eine Zielfunktion $f(z_1, \ldots, z_n)$ maximiert (minimiert) wird unter der Nebenbedingung

$$A \cdot (z_1, \ldots, z_n)^T \leq (\geq) B$$

für eine Matrix A und einem Vektor B. IP erlaubt eine elegante, mathematisch exakte und leicht verifizierbare Formulierung von Optimierungsproblemen unter heterogenen Randbedingungen. Eine Reihe von erfolgversprechenden IP-basierten Ansätzen zur Lösung von Problemen im VLSI-Entwurf wurde in den letzten Jahren vorgestellt [24, 25, 26, 27]).

Für ein gegebenes Kompaktierungsproblem (beschrieben durch eine Menge von Mikrooperationen, Datenabhängigkeiten und Kompatibilitätsrestriktionen) wird eine äquivalente IP-Formulierung automatisch erzeugt. Zur Lösung verwenden wir das Standardwerkzeug OSL (*Optimization Subroutine Library*) von IBM. Die IP-Nebenbedingungen stellen sicher, daß Abhängigkeiten ebenso wie Inkompatibilitäten zwischen den Mikrooperationen während der Kompaktierung beachtet werden. Weitere Nebenbedingungen dienen der Vermeidung von unerwünschten Seiteneffekten. Das formale Modell wird in [28] vorgestellt. Aufgrund einer allgemeinen Problemformulierung läßt sich das Kompaktierungsverfahren unmittelbar auf eine breite Klasse von DSPs anwenden und gewährleistet so Retargierbarkeit.

Abb. 9 zeigt den erzeugten parallelen TMS320C25 Maschinencode für das `complex_multiply` Beispiel. Die CPU-Zeit zur IP-basierten Kompaktierung beträgt in diesem Beispiel ca. 10 CPU-Sekunden. Die Codequalität (16 Instruktionen) entspricht sowohl der manuell erstellten als auch der TI-Compiler-generierten Maschinenprogramme [2]. Für komplexe DSPs, wie TMS320C2x oder Motorola DSP56k lassen sich Programmblöcke bis zu einer Länge von 50 in akzeptabler Zeit (bis zu einigen Minuten) kompaktieren. Der erhöhte Zeitaufwand gegenüber heuristischer Kompaktierung wird – vor allem bei laufzeitkritischen Anwendungen – durch die garantierte Optimalität des Kompaktierungsergebnisses aufgewogen.

```
ARP = 0                                  // LARP 0
AR[0] = 5                                // LARK AR0,5
TR = MEM[AR[ARP]]                        // LT *
AR[ARP] -= 4                             // SBRK 4
PR = TR * MEM[AR[ARP]] || AR[ARP] ++ // MPY *+
ACCU = PR || TR = MEM[AR[ARP]]
 || AR[ARP] ++                           // LTP *+
PR = TR * MEM[AR[ARP]] || AR[ARP] ++ // MPY *+
ACCU = ACCU - PR                         // SPAC
MEM[AR[ARP]] = ACCU || AR[ARP] ++    // SACL *+
TR = MEM[AR[ARP]]                        // LT *
AR[ARP] -= 2                             // SBRK 2
PR = TR * MEM[AR[ARP]] || AR[ARP] -- // MPY *-
ACCU = PR || TR = MEM[AR[ARP]]
 || AR[ARP] --                           // LTP *-
PR = TR * MEM[AR[ARP]] || AR[ARP] -- // MPY *-
ACCU = ACCU + PR                         // APAC
MEM[AR[ARP]] = ACCU                      // SACL *
```

Abbildung 9: *Kompaktierter TMS320C25 Code. "||" bezeichnet parallele Ausführung.*

5 Zusammenfassung

Die Einführung von Hochsprachen-Compilern für DSPs auch im Bereich von zeitkritischer Software erfordert neue, DSP-spezifische Techniken zur Codeoptimierung. Für anwendungsspezifische DSPs mit kleinen Stückzahlen ist außerdem Flexibilität in Form von *retargierbaren* Compilern von zentraler Bedeutung. Ein neuer Forschungszweig im Bereich VLSI-Entwurf konzentriert sich derzeit auf diese Problemstellungen [5].

In diesem Beitrag haben wir den an der Universität Dortmund verfolgten Ansatz zur retargierbaren Codeerzeugung für DSPs vorgestellt. Flexibilität wird durch die Verwendung von *externen*, durch den Benutzer editierbaren Prozessormodellen erreicht. Im Gegensatz zu anderen Ansätzen verwenden wir eine echte Hardwarebeschreibungssprache, wodurch eine geeignete Schnittstelle zu Hardware-Entwurfsumgebungen sichergestellt wird. Mögliche Abstraktionsebenen zur Prozessorbeschreibung reichen von der reinen Verhaltensebene bis zur RT-Ebene. Mit Hilfe der *Befehlssatzextraktion* schließen wir die Lücke zwischen hardwarenahen Prozessormodellen und fortgeschrittenen Techiken zur Codeerzeugung. Die extrahierten Befehlsmuster bilden eine von der Abstraktionsebene und konkreten Modellsyntax unabhängige Zwischenrepräsentation des Zielprozessors, für den Code erzeugt werden soll. Mit Hilfe von Standardwerkzeugen aus dem Compilerbau können automatisch effiziente auf Pattern Matching basierende Codegeneratoren erzeugt werden.

Ein zentraler Punkt in der DSP-spezifischen Codeoptimierung ist die Ausnutzung von potentieller Parallelität auf Befehlsebene. Zur Erhöhung von potentieller Parallelität verwenden wir Verfahren zur *Adreßzuweisung*, welche die spezifische Adreßerzeugungshardware von DSPs ausnutzen, indem sie – im Gegensatz zu

herkömmlichen Compilern – ein programmspezifisches Speicherlayout für Programmvariablen berechnen.

Die eigentliche Ausnutzung potentieller Parallelität findet in der *Kompaktierungsphase* statt. Parallelität wird von derzeitigen DSP-Compilern nicht oder nur unzureichend ausgenutzt. Wir verwenden ein exaktes formales Verfahren, welches auf DSPs zugeschnitten ist. Für DSPs mit stark codierten Befehlsformaten und restriktiver Parallelität ist bereits die Kompaktierung von kurzen Programmen ein nicht-triviales Problem. Unser Verfahren kompaktiert kleine bis mittlere Programmblöcke in akzeptabler Zeit und gewährleistet optimale Ausnutzung potentieller Parallelität für horizontale und codierte Befehlsformate.

Literatur

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

[2] V. Zivojnovic, J. Martinez, C. Schläger, H. Meyr: *DSPstone: A DSP-oriented Benchmarking Methodology*, Proc. ICSPAT, 1994

[3] A.V. Aho, R. Sethi, J.D. Ullman: *Compilers - Priciples, Techniques, and Tools*, Addison-Wesley, 1986

[4] R. Wilhelm, D. Maurer: *Compiler Design*, Addison-Wesley, 1995

[5] P. Marwedel, G. Goossens (Hrsg.): *Code Generation for Embedded Processors*, Kluwer Academic Publishers, 1995

[6] P. Hilfinger: *A High-Level Language and Silicon Compiler for Digital Signal Processing*, Custom Integrated Circuits Conference (CICC), 1985, pp. 213-316

[7] Mentor Graphics Corporation: *DSP Architect DFL User's and Reference Manual, V 8.2_6*, 1993

[8] M. Willems, M. Jersak, V. Zivojnovic: *DSP-bezogene Spracherweiterungen: Möglichkeiten und Grenzen*, Proc. DSP Deutschland, 1995, pp. 100-110

[9] R.M. Stallmann: *Using and Porting GNU CC* V2.4, Free Software Foundation, Cambridge/Massachusetts, 1993

[10] 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

[11] M.E. Conway: *Proposal for an* UNCOL, Comm. of the ACM, vol. 1, 1958

[12] D. Lanneer, J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, G. Goossens: *CHESS: Retargetable Code Generation for Embedded DSP Processors*, Kapitel 5 in [5]

[13] B. Wess: *Code Generation Based on Trellis Diagrams*, Kapitel 11 in [5]

[14] L. Nowak, P. Marwedel: *Verification of Hardware Descriptions by Retargetable Code Generation*, 26th Design Automation Conference (DAC), 1989, pp. 441-447

[15] R. Leupers, P. Marwedel: *Methods for Retargetable DSP Code Generation*, 7th IEEE Workshop on VLSI Signal Processing, 1994, pp. 127-136

[16] S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, D. Voggenauer: *The MIMOLA Language V4.1*, Forschungsbericht, Universität Dortmund, FB Informatik, September 1994

[17] R.E. Bryant: *Graph-based Algorithms for Boolean Function Manipulation*, IEEE Trans. on Computers, 40, no. 2, 1986, pp. 205-213

[18] C.W. Fraser, D.R. Hanson, T.A. Proebsting: *Engineering a Simple, Efficient Code Generator Generator*, ACM Letters on Programming Languages and Systems, vol. 1, no. 3, 1992, pp. 213-226

[19] G. Araujo, S. Malik: *Optimal Code Generation for Embedded Memory Non-Homogeneous Register Architectures*, 8th Int. Symp. on System Synthesis (ISSS), 1995, pp. 36-41

[20] R. Leupers, P. Marwedel: *Algorithms for Address Assignment in DSP Code Generation*, Int. Conference on Computer-Aided Design (ICCAD), 1996

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

[22] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, A. Wang: *Storage Assignment to Decrease Code Size*, ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 1995

[23] S. Davidson, D. Landskov, B.D. Shriver, P.W. Mallet: *Some Experiments in Local Microcode Compaction for Horizontal Machines*, IEEE Trans. on Computers, vol. 30, no. 7, 1981, pp. 460-477

[24] C. Gebotys, M. Elmasry: *Optimal VLSI Architectural Synthesis*, Kluwer Academic Publishers, 1992

[25] B. Landwehr, P. Marwedel, R. Dömer: *OSCAR: Optimum Simultaneous Scheduling, Allocation, and Resource Binding based on Integer Programming*, European Design Automation Conference (EURO-DAC), 1994

[26] H. Achatz: *Datenpfadsynthese mit Hilfe von ganzzahlige linearer Optimierung* (in German), Doctoral thesis, University of Passau, Germany, Shaker Verlag, 1995

[27] R. Niemann, P Marwedel: *Hardware/Software Partitioning Using Integer Programming*, European Design and Test Conference (ED & TC), 1996, pp. 473-479

[28] R. Leupers, P. Marwedel: *Time-constrained Code Compaction for DSPs*, IEEE Trans. on VLSI Systems, November 1996