Formale Methoden in der Codeerzeugung für digitale Signalprozessoren

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

ABSTRACT

Der Bereich HW/SW-Codesign für eingebettete Systeme umfaßt neben Methoden zur HW/SW-Partitionierung und Hardwaresynthese notwendigerweise auch Techniken zur Codeerzeugung für eingebettete programmierbare Prozessoren. Speziell im Falle von digitalen Signalprozessoren (DSPs) ist die Qualität verfügbarer Compiler unzureichend. Zur Vermeidung von aufwendiger Programmierung auf Assemblerebene sind daher neue DSP-spezifische Codeerzeugungstechiken notwendig. Dieser Beitrag stellt den Compiler Record vor, welcher für eine Klasse von DSPs Hochsprachenprogramme in Maschinencode übersetzt. Um den speziellen Anforderungen an Compiler für DSPs gerecht zu werden, werden teilweise formale Methoden eingesetzt. Wir stellen zwei solche für Record entwickelte Methoden vor, welche zur *Analyse von Prozessormodellen* sowie zur *Code-Kompaktierung* verwendet werden, und diskutieren deren praktische Anwendung.[1]

1 Einleitung

Im Gegensatz zu Allzweck-Systemen wie PCs und Workstations sind beim Entwurf eingebetteter Systeme oft *Echtzeit-Restriktionen* einzuhalten, wobei Chipfläche und/oder Leistungsaufnahme zu minimieren sind. Zur Einhaltung aller Echtzeit-Restriktionen ist meist ein Anteil von anwendungsspezifischer Hardware notwendig. Um kürzere Entwurfszeiten und höhere Flexibilität zu erreichen, werden die weniger zeitkritischen Teile des Systems durch Software realisiert, welche auf *eingebetteten Prozessoren* (RISCs, DSPs, Mikrocontroller) abläuft. Werden Prozessoren in einem Ein-Chip-Entwurf als Layout-Makrozellen instanziiert, so spricht man von *Prozessor-Cores*. Der Programmcode für Cores wird dabei in Form eines on-chip-ROMs implementiert. Die Codequalität wirkt sich somit unmittelbar auf die benötigte Chipfläche aus.

Aus diesen besonderen Randbedingungen entstehen neue Anforderungen an *Compiler* für eingebettete Prozessoren. Während bei Allzweck-Systemen die Übersetzungsgeschwindigkeit im Vordergrund steht, so ist bei eingebetteten Prozessoren die *Codequalität* (Größe und Ausführungsgeschwindigkeit) von zentraler Bedeutung. Schlechter Code führt zu unnötig großer Chipfläche und kann eine höhere Taktrate erforderlich machen, was sich wiederum ungünstig auf die Leistungsaufnahme auswirkt. Daher ist bei eingebetteten Prozessoren der Einsatz von relativ zeitaufwendigen Code-Optimierungstechniken gerechtfertigt.

In diesem Beitrag befassen wir uns mit Codeerzeugungstechniken für eine spezielle Klasse von eingebetteten Prozessoren, nämlich für *digitale Signalprozessoren* (DSPs). Aufgrund von sehr anwendungsspezifischen Befehlssätzen und irregulären Architekturen ist die Leistung heutiger Hochsprachen-Compiler für DSPs besonders schlecht. Der Overhead von compilergeneriertem Code gegenüber handgeschriebenem Assemblercode beträgt oft mehrere hundert

[1] Publikation: 5. GI/ITG/GMM Workshop "Methoden des Entwurfs und der Verifikation digitaler Systeme", Linz (Austria), April 1997

Prozent [1]. Der weitaus größte Anteil (ca. 90 %) der DSP-Software wird heute immer noch in Assemblersprachen entwickelt [2]. Der Aufwand für die Softwareentwicklung ist daher oft höher als der Aufwand zur Entwicklung der Hardwarekomponenten eines Systems. Neuere Forschungsaktivitäten befassen sich mit speziellen Codeerzeugungstechniken [3] mit dem Ziel, den SW-Flaschenhals im Entwurf DSP-basierter Systeme zu beseitigen und die Verwendung von Hochsprachen-Compilern für DSPs zu ermöglichen.

Neben hoher Codequalität ist *Retargierbarkeit* eine weitere wichtige Anforderung an Compiler für DSPs, d.h. die verwendeten Codeerzeugungstechniken sollten für eine ganze Klasse von *Zielprozessoren* einsetzbar sein. Diese Anforderung ergibt sich aus dem Trend zu anwendungsspezifischen Prozessoren ("ASIPs"), deren Befehlssatz auf einen sehr speziellen Anwendungsbereich zugeschnitten ist, und für die häufig überhaupt keine Compiler zur Verfügung stehen. Die detaillierte Architektur eines ASIPs steht teilweise auch nicht a priori fest, sondern wird erst während des HW/SW-Codesign-Prozesses für ein eingebettetes System ermittelt. Die Software-Komponenten des Systems werden erst anschließend auf den Befehlssatz des endgültigen ASIPs abgebildet. Retargierbare Compiler ermöglichen es, für verschiedene ASIPs bzw. für verschiedene Konfigurationen eines ASIPs den entsprechenden Maschinencode zu erzeugen ohne den Compiler selbst ändern zu müssen. Auf diese Weise kann der HW/SW-Tradeoff zwischen Prozessorarchitekturen und der Ausführungsgeschwindigkeit des Maschinencodes studiert werden.

Im folgenden Abschnitt stellen wir den an der Universität Dortmund entwickelten retargierbaren Compiler Record im Überblick vor, welcher zur Codeerzeugung für eine Klasse von DSPs bzw. ASIPs eingesetzt werden kann und teilweise formale Methoden verwendet. Die in Abschnitt 3 erläuterte *Befehlssatz-Extraktion* dient dazu, einen durch ein HDL-Modell spezifizierten Prozessor in ein internes Modell zu überführen, welches eine effiziente Codeerzeugung für den modellierten Zielprozessor gestattet. Um die bei DSPs besonders ausgeprägte *Befehlssatz-Parallelität* auszunutzen, wird in Record eine *Code-Kompaktierung* durchgeführt. Die in Abschnitt 4 diskutierte neuartige Kompaktierungstechnik ermöglicht die lokal optimale Ausnutzung potentieller Parallelität unabhängig vom Befehlsformat. Experimentelle Ergebnisse werden in Abschnitt 5 vorgestellt.

2 Das RECORD-Compilersystem

In der derzeitigen Version arbeitet Record für fixed-point DSPs mit fester Befehlslänge und Ein-Zyklus-Instruktionen. Der Zielprozessor, für den Maschinencode zu erzeugen ist, wird durch ein vom Benutzer (basierend auf einem User's Manual oder einem Schematic) erstelltes HDL-Modell an Record übergeben. Dieses Modell beinhaltet u.a. auch die konkrete Befehlswortlänge und das Befehlsformat. Während verwandte Arbeiten (z.B. CHESS [4], SPAM [5], CodeSyn [6]) meist sehr werkzeugspezifische Modellierungsformalismen verwenden, ermöglicht der HDL-basierte Ansatz in Record eine enge Anbindung des Compilers an HW-Entwurfsumgebungen. Des weiteren wird der Benutzer hierdurch von der Notwendigkeit befreit, sich von vornherein für einen bestimmten Abstraktionsgrad des Prozessormodells entscheiden zu müssen. Record unterstützt Verhaltensmodelle, RT-Modelle und teilweise auch Modelle auf Gatterebene. Aus Gründen der vereinfachten Modellierung verwenden wir die HDL MIMOLA [7] anstelle von VHDL. Abb. 1 zeigt als Beispiel das MIMOLA-Modell eines einfachen 8-Bit-Prozessors, welcher in diesem Fall als RT-Netzliste gegeben ist. Auf unterster Ebene werden alle Komponenten durch ihr Verhalten beschrieben. Dies geschieht durch Aufzählung *nebenläufiger Zuweisungen* an lokale Ports oder Speicher innerhalb jeder Komponente. Hierarchische Strukturen sind ebenso möglich wie auch komplexe Komponenten, z.B. Teil-Datenpfade mit lokalen Instruktionen.

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

PARTS                           -- Spezifikation von RT-Komponenten
 IM: MODULE InstrROM (IN adr: Byte; OUT ins: InstrFormat);
     BEHAVIOR IS
      VAR storage: ARRAY[0..255] OF InstrFormat; -- Befehlsspeicher
      BEGIN ins <- storage[adr]; END;

 PC, REG: MODULE Reg8bit (IN data: Byte; OUT outp: Byte);
          BEHAVIOR IS
           VAR R: Byte;   -- 8-Bit Programmzaehler und Datenregister
           BEGIN R := data; outp <- R; END;

 PCIncr: MODULE IncrementByte (IN data: Byte; OUT inc: Byte);
         BEHAVIOR IS      -- Incrementer fuer Programmzaehler
          BEGIN inc <- INCR data; END;

 RAM: MODULE Memory (IN data, adr: Byte; OUT outp: Byte; IN c: Bit);
      BEHAVIOR IS
       VAR storage: ARRAY[0..255] OF Byte; -- Speicher
       BEGIN
        IF c THEN storage[adr] := data;
        outp <- storage[adr];
       END;

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

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

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
                                    REG.outp   -> RAM.data;
END; -- STRUCTURE
```

Abbildung 1: *MIMOLA-Modell eines einfachen 8-Bit-Prozessors*

Abb. 2 zeigt das RECORD-System im Überblick. Das zu übersetzende Quellprogramm wird in der Datenflußsprache DFL [8] spezifiziert und wird durch ein Frontend in ein Kontroll/Datenflußgraphformat (CDFG) umgesetzt. Das MIMOLA-Modell des Zielprozessors wird analysiert, und der Befehlssatz des Prozessors wird extrahiert (siehe Abschnitt 3). Man erhält eine interne Darstellung des Zielprozessors in Form einer Menge von *RT-Mustern*.

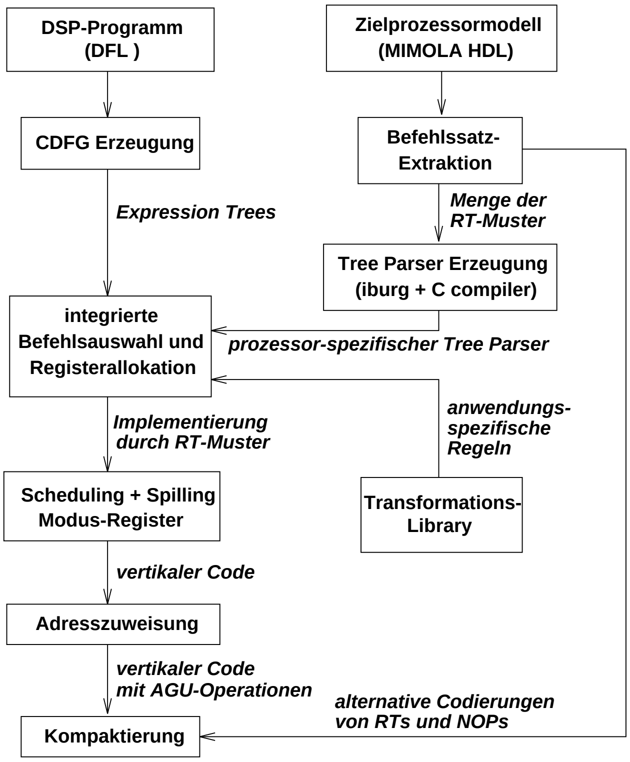

Abbildung 2: *Das RECORD-Compilersystem*

Jedes RT-Muster repräsentiert eine primitive Zuweisung auf RT-Ebene, die auf dem Zielprozessors ausgeführt werden kann. Zusätzlich liefert die Befehlssatzextraktion für jedes gefundene RT-Muster die Menge der zugehörigen Codierungen ("opcodes"), d.h. partielle Belegungen des Befehlswortes. Eine Besonderheit von DSPs ist, daß für jedes RT-Muster *alternative* Codierungen vorliegen können, wobei die Auswahl einer konkreten Codierung im Hinblick auf die Maximierung der Befehlssatz-Parallelität erfolgen muß. Wie für RT-Muster werden auch Codierungen für "no-operations" (NOPs) extrahiert, welche für die Code-Kompaktierung benötigt werden (Abschnitt 4).

Aus der Menge der RT-Muster wird mit Hilfe des Standard-Werkzeugs `iburg` [9] und eines C-Compilers ein prozessorspezifischer *Tree Parser* erzeugt. Das CDFG-Zwischenformat wird in *Ausdrucksbäume* zerlegt, für die der erzeugte Tree Parser jeweils eine optimale Befehlsauswahl (in Form von RT-Mustern) und Registerallokation berechnet. Hierbei kann optional eine benutzerdefinierte Library von *Ersetzungsregeln* berücksichtigt werden. Ersetzungsregeln (z.B. algebraische Regeln) werden verwendet, um semantisch äquivalente alternative Versionen für einen Ausdrucksbaum zu erzeugen, auf die jeweils der Tree Parser angewendet wird. Aus mehreren Alternativen wird diejenige ausgewählt, welche die geringste Anzahl von RT-Muster-Instanzen zur Implementierung benötigt. In der anschließenden Scheduling-Phase wird der benötigte Spill-Code für Register heuristisch minimiert. Ebenso wird Code erzeugt, welcher zur Programm-Laufzeit die geeigneten Zustände für die bei DSPs häufigen *Modus-Register* (z.B. Sign Extension Mode, Saturating Arithmetic) herstellt. Als Zwischenformat erhält man *vertikalen* Maschinencode. Während der *Adreßzuweisung* werden zusätzliche RTs in den vertikalen Code eingefügt, welche zur effizienten Berechnung von Adressen für Speicherzugriffe dienen. Hierbei kommen heuristische Techniken zum Einsatz [10], die auf die spezielle Architektur von Adreßwerken (AGUs) in DSPs abgestimmt sind. In der abschließenden Code-Kompaktierungsphase werden alle generierten RTs zu (parallelen) Maschineninstruktionen gepackt, wobei die potentielle Parallelität innerhalb von Basisblöcken jeweils optimal ausgenutzt wird. Das Endergebnis ist ausführbarer Maschinencode, welcher das DFL-Quellprogramm auf dem spezifizierten Zielprozessor realisiert.

3 Befehlssatz-Extraktion

Das HDL-Modell eines Prozessors kann – je nach Abstraktionsgrad – Strukturanteile enthalten, welche oft eine bequeme Modellierung gestatten aber für die Codeerzeugung irrelevant sind. Für Prozessoren mit einem relativ hohen Grad an Befehlssatz-Parallelität (in DSPs typischerweise 3-6 RTs pro Instruktion) sind Netzlistenbeschreibungen häufig kompakter als komplette Aufzählungen aller verfügbaren Maschinenbefehle. Das von RECORD verwendete interne Prozessormodell besteht daher aus einer Aufzählung aller atomaren RT-Operationen sowie Informationen über die Parallelisierbarkeit von RTs. Beides wird aus dem HDL-Modell extrahiert. Ein RT-Muster ist gegeben durch ein Paar (d, e), wobei das *Ziel* d eine sequentielle Komponente (Register, Speicher) oder einen Port bezeichnet, dem ein *RT-Ausdruck* e zugewiesen wird. Ein RT-Ausdruck e ist induktiv definiert als:

- eine **binäre Konstante** $B \in \{0, 1, x\}^+$
- ein **Lesezugriff** $\mathit{read}(r)$ auf eine Komponente r (Register, Speicher, Port)
- ein **komplexer Ausdruck** $\mathit{op}(e_1, \ldots, e_k)$, wobei *op* ein Operator (`+, -, *, AND, OR, NOT, SHIFT, ...`) mit Stelligkeit k und $e_1, \ldots, e_k$ RT-Ausdrücke sind, oder
- ein **Unterbereichs-Ausdruck** $e' = e.(hi : lo)$, wobei e ein RT-Ausdruck ist und $(hi : lo)$ einen Bit-Indexbereich bezeichnet.

Jedes RT-Muster ist gekoppelt an eine *Ausführungsbedingung* ("RT-Condition"). Diese beschreibt die Anforderungen an den Maschinenzustand, unter dem die jeweilige RT-Operation ausgeführt wird. Der Maschinenzustand bezieht sich auf eine partielle Belegung des aktuellen Befehlswortes sowie – falls vorhanden – auf die Zustände von Modus-Registern. Eine RT-Condition ist formal definiert durch eine Boolesche Funktion

$$F : \{I_1, \ldots, I_n\} \to \{0, 1\}$$

wobei die Variablen $I_1, \ldots, I_n$ die Bits des Befehlswortes repräsentieren. Ein RT wird genau dann ausgeführt, wenn die zugehörige RT-Condition für die aktuelle Belegung der Zustandsvariablen den Wert 1 berechnet. Zwei RT-Operationen, die keine wechselseitigen Abhängigkeiten aufweisen, können parallel ausgeführt werden, wenn die Konjunktion ihrer RT-Conditions nicht konstant 0 ist.

3.1 Extraktion von RT-Mustern

Die Bestimmung der RT-Muster erfolgt durch Aufzählung aller Datentransportwege im Zielprozessor. Ausgehend von jedem Ziel (Register, Speicher, Port) wird ein internes Graphmodell der Prozessor-Netzliste rekursiv durchlaufen, wobei alle möglichen Datentransporte von einer oder mehreren Quellen zum Ziel bestimmt werden, die innerhalb eines Maschinenzyklus ausgeführt werden können (Abb. 3). Beim Erreichen von Komponenten mit mehreren Eingängen (z.B. ALUs, Multiplexer) verzweigt der Durchlauf für jeden Eingang. Falls das HDL-Modell Busse beinhaltet, so werden alle möglichen Bustreiber betrachtet. Für das in Abb. 1 gezeigte Modell werden die in Tabelle 1 (Spalte 1) aufgelisteten RT-Muster extrahiert.

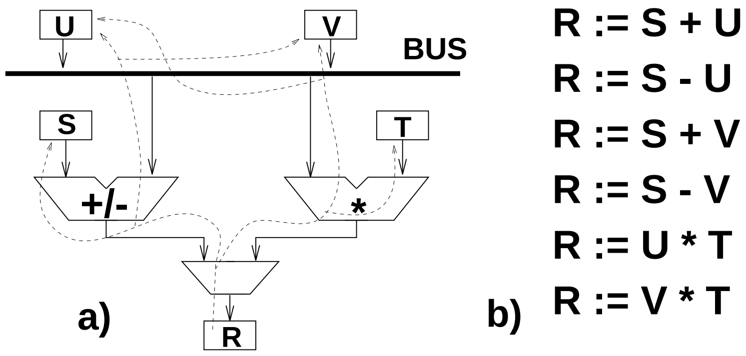

Abbildung 3: *Aufzählung von Datentransportwegen während der Befehlssatz-Exttraktion: a) Der Netzlistendurchlauf startet beim Zielregister R und verfolgt mögliche Datentransportwege zurück bis zu den Quellregistern S, T, U, V. b) Extrahierte RT-Muster mit Ziel R.*

RT-Muster	partielle Instruktion (Bits 20..0)
	`21111111111`
	`098765432109876543210`

PC.R := INCR PC.R	`xxxxxxxxxxxxxxxxxxxxx`
REG.R := inp	`xxxxxxxxxxxxxxxxx011x`
REG.R := IM.storage[PC.R].(20:13)	`xxxxxxxxxxxxxxxxx001x`
REG.R := RAM.storage[IM.storage[PC.R].(12:5)]	`xxxxxxxxxxxxxxxxx1x1x`
REG.R := REG.R - inp	`xxxxxxxxxxxxxxxxx0101`
REG.R := REG.R - IM.storage[PC.R].(20:13)	`xxxxxxxxxxxxxxxxx0001`
REG.R := REG.R - RAM.storage[IM.storage[PC.R].(12:5)]	`xxxxxxxxxxxxxxxxx1x01`
REG.R := REG.R + inp	`xxxxxxxxxxxxxxxxx0100`
REG.R := REG.R + IM.storage[PC.R].(20:13)	`xxxxxxxxxxxxxxxxx0000`
REG.R := REG.R + RAM.storage[IM.storage[PC.R].(12:5)]	`xxxxxxxxxxxxxxxxx1x00`
RAM.storage[IM.storage[PC.R].(12:5)] := REG.R	`xxxxxxxxxxxxxxxx1xxxx`
outp ← REG.R	`xxxxxxxxxxxxxxxxxxxxx`

Tabelle 1: *Extrahierte RT-Muster für* `SimpleProcessor`

3.2 Extraktion von RT-Conditions

Problematischer als die Extraktion von RT-Mustern ist die Bestimmung der RT-Conditions. Da neben dem Datenpfad auch der Controller Bestandteil des HDL-Modells ist, müssen die lokalen Kontrollsignale für jede Prozessorkomponente zu den primären Quellen für Kontrollsignale (Befehlsspeicher, Modus-Register) zurückverfolgt werden. Hierbei sind unter Umständen komplexe kombinatorische Schaltungen, z.B. Befehlsdecoder, zu betrachten, welche wiederum auf verschiedenen Abstraktionsniveaus (RT-Ebene, Gatterebene) modelliert sein können (Abb. 4).

Die Extraktion der RT-Conditions kann auf die Manipulation Boolescher Funktionen zurückgeführt werden und wird in zwei Phasen durchgeführt:

1. **Extraktion lokaler RT-Conditions:** Für alle Zuweisungen innerhalb einer Komponente werden die notwendigen Belegungen der lokalen Kontrollports der Komponente bestimmt. Diese Belegung wird in der MIMOLA HDL mittels (evtl. verschachtelten) IF- und CASE-Konstrukten und Booleschen Operatoren (AND, OR, NOT, ...) ausgedrückt. Repräsentieren z.B. die Booleschen Variablen $C_1, \ldots, C_m$ die Bits eines lokalen Kontrollports (oder eines lokalen Modus-Registers), so ist die lokale RT-Condition eine Funktion $F(C_1, \ldots, C_m)$ dieser Variablen.

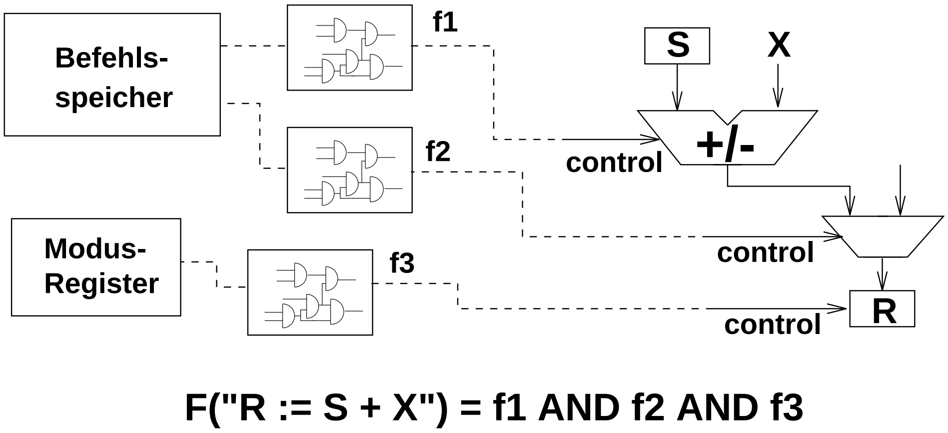

Abbildung 4: *Extraktion von RT-Conditions: Die lokalen Kontrollsignale für die am RT-Muster "R := S + X" beteiligten Komponenten müssen gleichzeitig eingestellt werden. Die lokalen Kontrollsignale können beliebige Boolesche Funktionen (hier: f1, f2, f3) der Instruktionsbits und Modus-Register-Bits sein. Im Beispiel erhält man die Ausführungsbedingung F als Konjunktion von f1, f2, f3.*

2. **Substitution:** Um die "globalen" RT-Conditions (in Abhängigkeit von den primären Kontrollsignal-Quellen) zu erhalten, werden die lokalen Kontrollsignale $C_1, \ldots, C_m$ durch geeignete Funktionen substituiert. Diese ergeben sich aus der Verdrahtung der lokalen Kontrollports mit dem Befehlsspeicher sowie der evtl. Decodierlogik. Somit kann jedes Signal C_i als Funktion f_i der Variablen $I_1, \ldots, I_n$ aufgefaßt werden, und man erhält die globale RT-Condition durch die Substitution $F|_{C_i = f_i}$, welche für alle C_i durchgeführt wird.

Die Implikanten einer RT-Condition F können als alternative (partielle) Belegungen des Befehlwortes angesehen werden, unter denen ein bestimmter Register-Transfer stattfindet. Spalte 2 in Tabelle 1 zeigt die extrahierten Belegungen für das Beispiel aus Abb. 1 in Form von partiellen Instruktionen.

Die Extraktion von RT-Conditions erfordert umfangreiche Manipulationen (Verknüpfung, Substitution, Erfüllbarkeitstest) von Booleschen Funktionen. Zur effizienten Ausführung dieser Manipulationen werden in RECORD *Binary Decision Diagrams* (BDDs) eingesetzt, wobei hier auf ein Standard-OBDD-Paket [11] zurückgegriffen wird. Hierdurch läßt sich die gesamte Befehlssatz-Extraktion auch für komplexe Zielprozessoren effizient durchführen. Die extrahierten RT-Muster werden während der Codeerzeugung zur Codeselektion verwendet. Im Zusammenhang mit der Erzeugung von Parsern für Ausdrucksbäume aus den RT-Mustern erhält man eine vollautomatische Prozedur zur Generierung eines prozessorspezifischen Codeselektors aus einem HDL-Modell. Die extrahierten RT-Conditions werden in der im folgenden beschriebenen Code-Kompaktierung verwendet.

4 Code-Kompaktierung

Durch die Codeerzeugung wird eine Menge von RTs bestimmt, welche das gewünschte Programmverhalten auf dem jeweiligen Zielprozessor realisieren. Aus Komplexitätsgründen wird meist darauf verzichtet, die Parallelisierbarkeit von RTs schon während der Codeerzeugung zu überprüfen. Die Ausnutzung potentieller Parallelität auf Befehlsebene ist allerdings gerade bei DSPs eine wesentliche Optimierungsquelle. Daher muß nach der Codeerzeugung eine *Kompaktierung* erfolgen, welche die generierten RTs einzelnen Kontrollschritten in einem Schedule zuordnet, wobei das Ziel eine Minimierung der Schedule-Länge ist. Die Parallelisierbarkeit von RTs wird eingeschränkt durch *Vorrangsvorschriften*, welche sich durch die

Codeerzeugung ergeben, sowie durch *Konflikte* zwischen RTs. Ein Konflikt liegt vor, wenn die partiellen Instruktionen zweier RTs inkompatibel sind. Dies schließt sowohl Befehlswort- als auch Ressourcenkonflikte ein.

Unter diesen Bedingungen ist das Problem der optimalen Code-Kompaktierung NP-hart. Verschiedene Heuristiken zur Kompaktierung wurde im Bereich der *Mikroprogrammierung* entwickelt [12]. Da diese jedoch in erster Linie für VLIW-artige Rechner mit horizontalen Befehlsformaten entworfen worden sind, lassen sich diese auf DSPs nicht unmittelbar anwenden. Bei DSPs müssen zusätzliche Randbedingungen in Betracht gezogen werden:

1. Es wird Maschinencode von **sehr hoher Qualität** benötigt, d.h. es sollte möglichst wenig vom Optimum abgewichen werden. Die zur Kompaktierung benötigte Rechenzeit ist allerdings "unkritisch".

2. Im Gegensatz zu reinen VLIW-Rechnern sind die Möglichkeiten zur Parallelisierung von RTs eher beschränkt. Um eine möglichst geringe Befehlswortbreite zu erreichen, sind Befehlsformate bei DSPs oft **stark codiert**, d.h. es lassen sich nur ganz bestimmte Kombinationen von RTs parallel ausführen.

3. Für jedes RT-Muster können **alternative Codierungen** vorliegen. Die Konflikte zwischen RTs sind somit nicht statisch vorgegeben, sondern hängen von der Auswahl einer bestimmten Codierung ab. Bei einem TI TMS320C2x DSP bspw. existieren mehr als 20 verschiedene Codierungen für "auto-increment"-Befehle auf Adreßregistern.

4. Es können **Seiteneffekte** auftreten. Die Auswahl einer Codierung für ein RT kann die Aktivierung eines anderen RTs implizieren, falls sich deren Mengen von alternativen Codierungen überlappen. Je nach Kontext können Seiteneffekte ausgenutzt oder toleriert werden, während unerwünschte Seiteneffekte unbedingt vermieden werden müssen.

Heuristische Kompaktierungsalgorithmen, welche alle diese Randbedingungen beachten, sind bisher nicht bekannt. Heutige kommerzielle C-Compiler für DSPs machen daher nur sehr eingeschränkt Gebrauch von potentieller Parallelität [1], wodurch sich wiederum ein deutlicher Verlust an Codequalität ergibt.

Aufgrund dieser Beobachtungen wurde für Record ein exaktes, formales Kompaktierungsverfahren entwickelt, welches auf *Integer Linear Programming* (ILP) beruht und für nicht allzu große Basisblöcke eines Programms eine optimale Kompaktierung in akzeptabler Zeit berechnet. ILP bietet die Möglichkeit, mathematisch exakte und leicht verifizierbare Formulierungen von Problemen mit heterogenen Randbedingungen zu erstellen. Daher wird ILP neuerdings auch in der High-Level-Synthese eingesetzt [13]. Record berechnet für jede Instanz des Kompaktierungsproblems automatisch ein entsprechendes 0/1-ILP-Modell. Dieses wird dann mit Standard-ILP-Solvern (z.B. OSL von IBM) gelöst. Aus der berechneten Belegung der 0/1-Entscheidungsvariablen wird dann der eigentliche Schedule abgeleitet. Die dreifach indizierten Entscheidungsvariablen $x_{i,j,t}$ codieren dabei die folgende Information

$$x_{i,j,t} = 1 \quad \Leftrightarrow \quad \text{RT Nr. } i \text{ wird in Kontrollschritt } t \text{ mit Codierung Nr. } j \text{ gescheduled.}$$

und beinhalten somit auch Codierungsauswahl für jeden RT. Zusätzliche Entscheidungsvariablen dienen zur Vermeidung von unerwünschten Seiteneffekten. Hierzu werden spezielle "NOP"-Befehle verwendet, die sich – zusammen mit ihren Codierungen – als Nebenprodukt der Befehlssatz-Extraktion ergeben. Falls ein Register in einem bestimmten Kontrollschritt nicht überschrieben werden darf, so wird das Scheduling eines NOPs für dieses Register in dem Kontrollschritt durch ein ILP-Constraint erzwungen. Weitere Constraints repräsentieren die Vorrangsvorschriften zwischen RTs sowie mögliche Konflikte. Die zu minimierende Zielfunktion gibt die Anzahl benötigter Kontrollschritte für eine Belegung der Entscheidungsvariablen an. Das genaue formale ILP-Modell wird in [14] beschrieben.

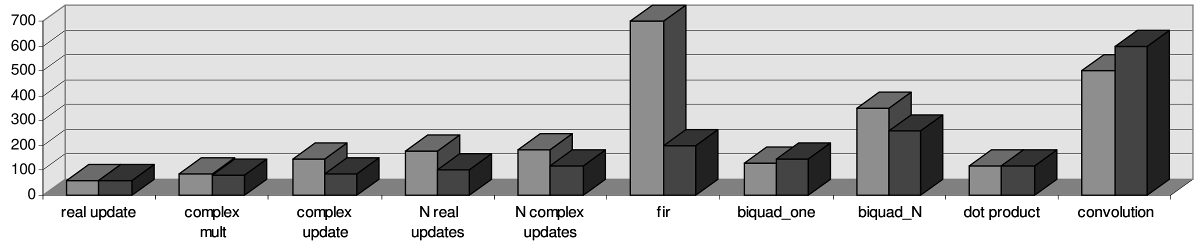

Abbildung 5: *Codequalität: relative Codegröße (in %) für den TI TMS320C25 DSP*

5 Ergebnisse

Der RECORD-Compiler wurde auf einer Workstationumgebung implementiert und für verschiedene Zielprozessoren (ASIPs und ein Texas Instruments TMS320C25 Standard-DSP) getestet. Tabelle 2 zeigt für die Retargierung benötigte Laufzeiten. Diese beinhalten die Befehlssatz-Extraktion sowie die Erzeugung des Codeselektors. Die Ergebnisse zeigen, daß

Ziel-Prozessor	Anzahl der RT-Muster	Laufzeit SPARC-20 CPU sec
demo	439	356
ref	1703	84
manocpu	207	6.3
tanenbaum	232	11.7
bass boost	89	3.7
TMS320C25	356	165

Tabelle 2: *Laufzeiten zur Retargierung von* RECORD

das Retargieren typischerweise innerhalb von wenigen CPU-Minuten durchgeführt werden kann. Bei Zielprozessoren mit nicht von vornherein festgelegter Architektur wird somit das *HW/SW-Codesign auf Prozessorebene* unterstützt: Durch Re-Compilieren eines Programms auf verschiedene programmierbare Zielarchitekturen können die Auswirkungen von HW-Änderungen auf den erzeugten Code studiert werden.

Die erzielte Codequalität wurde für den TMS320C25 DSP und eine Reihe von DSP-Benchmarkprogrammen [1] ausgewertet. Das Balkendiagramm in Abb. 5 zeigt die prozentuale Größe compilierter Programme (rechts: RECORD, links: Texas Instruments C-Compiler) im Vergleich zu handgeschriebenem Assemblercode (100 %). In den meisten Fällen ergibt sich eine höhere Qualität des von RECORD generierten Codes gegenüber dem TI C-Compiler. Eine Analyse des Codes zeigt, daß dies in erster Linie durch eine bessere Ausnutzung der Befehlssatz-Parallelität erreicht wird. Die Größe der mittels ILP kompaktierten Basisblöcke liegt dabei zwischen 10 und 60 RTs. Die für die Übersetzung benötigte CPU-Zeit liegt zwischen 3 und 120 SPARC-20 CPU-Sekunden. Für weniger komplexe Zielprozessoren als den TMS320C25 lassen sich allerdings auch wesentlich größere Blöcke in akzeptabler Zeit (bis zu einigen CPU-Minuten) optimal kompaktieren.

6 Zusammenfassung

Die Anforderungen an Compiler für eingebettete DSPs liegen in erster Linie im Bereich Retargierbarkeit und Codequalität. Eine komfortable Möglichkeit, Retargierbarkeit zu erreichen, ist, dem Compiler neben dem zu übersetzenden Programmcode ein HDL-Modell des

Zielprozessors zu übergeben. Aus diesem Modell können mit Hilfe von BDDs alle für die Codeerzeugung benötigten Aspekte extrahiert werden. Auf diese Weise wird eine bessere Kopplung von Compilern an HW-Entwurfsumgebungen erreicht. Über die Codeerzeugung hinaus hat die vorgestellte Befehlssatz-Extraktion auch Anwendungen in der Validation von Prozessormodellen, bspw. bei der Überprüfung, ob eine gegebene RT-Struktur einen gewünschten Befehlssatz implementiert.

Zur Sicherstellung ausreichender Codequalität sind neue Code-Optimierungstechniken erforderlich, welche – evtl. zu Lasten höherer Übersetzungszeiten – über den Bereich des Standard-Compilerbaus hinausgehen. Mit Hilfe des vorgestellten ILP-basierten lokalen Code-Kompaktierungsverfahrens wird sichergestellt, daß – innerhalb von Basisblöcken – keine Codequalitätsverluste durch mangelnde Ausnutzung von Befehlssatz-Parallelität auftreten. Eine sorgfältige Formulierung des Kompaktierungsproblems als ILP gewährleistet dabei die Anwendbarkeit für praxisrelevante Problemgrößen.

Literatur

[1] 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

[2] 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

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

[4] 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 [3]

[5] 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

[6] C. Liem, P. Paulin, M. Cornero, A. Jerraya: *Industrial Experience Using Rule-driven Retargetable Code Generation for Multimedia Applications*, 8th Int. Symp. on System Synthesis (ISSS), 1995, pp. 60-65

[7] S. Bashford, U. Bieker, B. Harking, et al.: *The MIMOLA Language V4.1*, Technischer Bericht, Universität Dortmund, FB Informatik, September 1994

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

[9] 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

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

[11] K.S. Brace, R.L. Rudell, R.E. Bryant: *Efficient Implementation of a BDD Package*, 27th Design Automation Conference (DAC), 1990, pp. 40-45

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

[13] H. Achatz: *Datenpfadsynthese mit Hilfe von ganzzahliger linearer Optimierung*, Dissertation, Universität Passau, Shaker Verlag, 1995

[14] R. Leupers, P. Marwedel: *Time-Constrained Code Compaction for DSPs*, IEEE Transactions on VLSI Systems, vol. 5, no. 1, 1997