Ausnutzung von Conditional Instructions in VLIW DSP-Compilern

Rainer Leupers
Universität Dortmund
Lehrstuhl Informatik 12
44221 Dortmund
email: leupers@ls12.cs.uni-dortmund.de
WWW: ls12-www.cs.uni-dortmund.de/~leupers

Abstract – In diesem Beitrag stellen wir eine neuartige Compiler-Optimierungstechnik vor, welche vor allem bei der Programmierung von kontrollintensiven Applikationen auf VLIW DSPs Anwendung findet. Die Technik beruht auf der optimierten Ausnutzung sog. conditional instructions, welche bei den Befehlssätzen neuerer DSPs vorkommen, zur Implementierung von if-then-else Statements. Während die "klassische" Implementierung mittels bedingter Sprünge aufgrund von Pipeline-Konflikten erhebliche Performanceverluste im Maschinencode bewirken kann, ist die Verwendung von conditional instructions häufig effizienter. Die vorgestellte Technik wählt für jedes (evtl. geschachtelte) if-then-else Statement im Source Code die schnellste Implementierung aus. Hierzu werden Abschätzungen der Ausführungszeit sowie ein auf dynamischer Programmierung basierendes Verfahren eingesetzt. Experimentelle Ergebnisse für einen TI 'C62xx demonstrieren die Leistungsfähigkeit der Optimierungstechnik[1].

1 Einleitung

Bei Prozessoren zur digitalen Signalverarbeitung ist gegenwärtig ein Trend zu hochgradig parallelen Befehlssätzen zu beobachten. Diese werden mittels sehr breiter Befehlswörter ("VLIW") realisiert, welche die parallele Ausführung mehrerer Maschinenbefehle pro Befehlszyklus erlauben. Beispiele hierfür sind der Texas Instruments C62xx [1] und der Philips Trimedia TM1000 [2], welche 8 bzw. 5 Befehle pro Zyklus ausführen können.

Dieser Trend ist nicht nur unter dem Aspekt der Performancesteigerung durch Parallelisierung von Befehlen zu betrachten, sondern auch als Schritt in Richtung "compiler-freundlicher" Prozessorarchitekturen. Bekanntlich erzeugen derzeitige Compiler für klassische fixed-point DSPs wie TI TMS320C5x oder Motorola 56k relativ schlechten Code im Vergleich zu handgeschriebenem Assemblercode [3]. Dies liegt in erster Linie an der irregulären Architekur solcher DSPs, auf die Standard-Compileroptimierungen nur begrenzt anwendbar sind. Daher wurden in den vergangenen Jahren neue Techniken entwickelt, die speziell zur Optimierung von datenflußdominierten Anwendungen für solche fixed-point DSPs geeignet sind. Übersichten hierzu finden sich in [4, 5].

[1] Publikation: DSP Deutschland, Munich/Germany, Oct 1998

Dagegen verfügen VLIW DSPs über eine vergleichsweise reguläre Architektur, welche für bekannte Compilertechniken besser zugänglich ist. Allerdings treten für VLIW DSPs auch neue Compilerprobleme auf. Zur Performancesteigerung verfügen derartige Prozessoren über tiefe Befehlspipelines, welche ihre Wirksamkeit allerdings nur bei einer relativ geringen Anzahl von Sprüngen im Programm voll entfalten können. Aufgrund des Pipelinings führt ein Sprung zu einer Verzögerung der Programmausführung ("pipeline stalls") von mehreren Befehlszyklen. Bei einem TI C62xx bspw. verursacht ein Sprung bis zu 5 Stall-Zyklen, d.h. er führt zu einem Performanceverlust von maximal $5 \cdot 8 = 40$ Befehlen.

Dieses Problem tritt insbesondere bei kontrolldominierten Anwendungen auf, bei denen der Source Code eine große Anzahl von if-then-else (ITE) Statements (und damit implizit auch viele Sprünge) enthält. Wie kürzlich in einer Erweiterung des DSPStone-Projektes gezeigt wurde [6, 7], produzieren derzeitige DSP-Compiler nicht nur bei datenflußdominierten sondern auch bei kontrolldominierten Anwendungen Assemblercode von unzureichender Qualität. Dies ist insbesondere deshalb ein Problem, weil für kontrolldominierte Funktionen im Gegensatz zu vielen datenflußdominierten (wie digitale Filter und FFT) i.a. keine Assembler-Libraries verfügbar sind. Speziell bei kontrolldominierten Anwendungen und VLIW DSPs sind somit Architekturmaßnahmen und Compileroptimierungen notwendig, um den durch den Einsatz von Compilern verursachten Code-Overhead in Größe und Geschwindigkeit gering zu halten.

Eine Möglichkeit ist die Implementierung von ITE-Statements ohne Veränderung des Kontrollflusses, d.h. ohne Modifikation des Programmzählers, da in diesem Fall die oben erwähnten Stall-Zyklen vermieden werden. Eine Reihe von Architekturmaßnahmen hierfür wird in [8] genannt. Eine davon (*conditional instructions*) wurde in neueren VLIW DSPs implementiert. Hierbei handelt es sich um eine Verallgemeinerung bedingter Sprungbefehle. Eine conditional instruction ist ein Befehl, dessen Ausführung an eine zur Programmlaufzeit ausgewertete Bedingung geknüpft ist.

Aus Sicht eines Compilers oder eines Assemblerprogrammierers führt die Verfügbarkeit von conditional instructions dazu, daß zwei alternative Möglichkeiten zur Implementierung von ITE-Statements verfügbar sind: die "klassische" Variante mit bedingten Sprüngen und eine Variante, welche conditional instructions ausnutzt. Speziell bei geschachtelten ITE-Statements im Source Code ist es aber keineswegs offensichtlich, welche dieser Varianten den besseren Code produziert. Bei der Benutzung der Variante mit conditional instructions entstehen Assemblerprogramme, bei denen in jedem Befehlszyklus verschiedene Kontrollpfade des Programms quasi gleichzeitig ausgeführt werden. Diese Programme sind teilweise äußerst schlecht lesbar, so daß conditional instructions bei der manuellen Assemblerprogrammierung vermutlich nur restriktiv eingesetzt werden.

In diesem Beitrag stellen wir daher eine neue Compilertechnik vor, welche die Implementierung beliebig geschachtelter ITE-Statements systematisch optimiert. Das Ziel dabei ist die Minimierung der worst case-Ausführungszeit einer Funktion. Die Minimierung erfolgt dadurch, daß basierend auf einer Abschätzung der Ausführungszeit von Programmblöcken die jeweils schnellere der beiden ITE-Implementierungen ausgewählt wird.

Diese Optimierungstechnik arbeitet auf einer Zwischenrepräsentation (*intermediate representation*, IR) von ANSI C Programmen, welche mittels eines C-Frontends erzeugt wird. Die IR besteht aus drei Klassen von Statements:

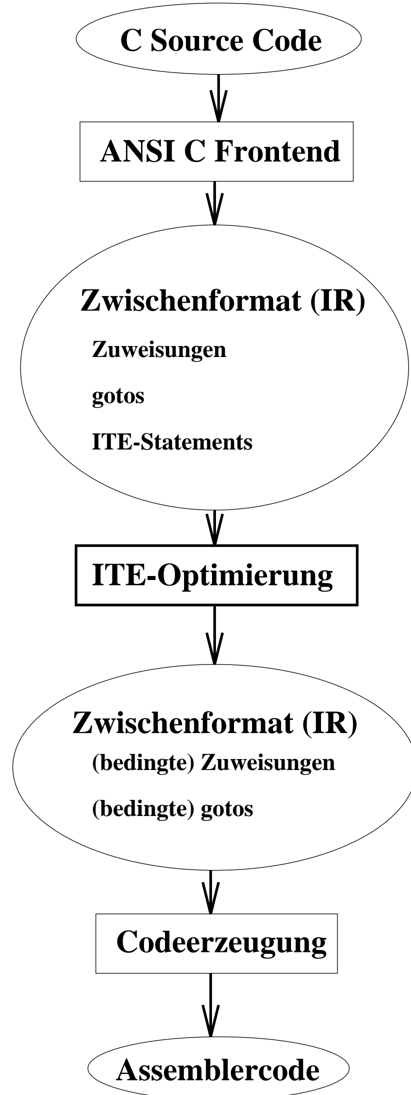

Abbildung 1: *Ablauf der Optimierung von ITE-Statements*

1. **Zuweisungen** in Form von Drei-Adreß-Code (z.B. "a = b + c").

2. **Sprünge** der Form "goto *label*".

3. **ITE-Statements** der Form $(cond, B_T, B_E)$, wobei die Bedingung *cond* ein Boolescher Ausdruck ist, und die Blöcke B_T (then-Zweig) und B_E (else-Zweig) Statement-Blöcke sind, welche wiederum ITE-Statements enthalten dürfen.

Dabei wird vorausgesetzt, daß Programme "strukturiert" sind, d.h. es gibt im Source Code keine "goto"-Sprünge in die then- oder else-Zweige von ITE-Statements. Die vorgestellte Optimierungstechnik ersetzt die ITE-Statements in der IR durch äquivalente Statement-Blöcke, welche nur aus (evtl. bedingten) Zuweisungen und Sprüngen bestehen. Aus der so modifizierten IR kann nun Assemblercode generiert werden (Abb. 1).

In den folgenden Abschnitten erfolgt eine genauere Beschreibung des Konzepts der conditional instructions sowie der Optimierungstechnik. Im Anschluß daran stellen wir experimentelle

Ergebnisse für einen TI TMS320C62xx vor, welche die Leistungsfähigkeit der vorgestellten Technik aufzeigen.

2 Conditional instructions

Eine conditional instruction ist ein Paar (C, I), wobei C eine Boolesche Variable und I eine "normale" Instruktion ist. Die Instruktion I wird nur dann *effektiv* ausgeführt (d.h. sie schreibt einen Wert in den Speicher oder ein Register), wenn C zum Zeitpunkt der Ausführung von I "true" ist. Ansonsten verhält I sich wie ein NOP-Befehl, wobei allerdings auch Prozessorressourcen, z.B. ALUs, reserviert werden müssen. Die Bedingung C ist typischerweise das Resultat eines Vergleichs und wird in einem (symbolischen oder physikalischen) Register R gespeichert. C ist "false", wenn $R = 0$, und ansonsten "true".

Im folgenden verwenden wir die Notation "[C] I" für eine conditional instruction (C, I). Die Notation "[!C] I" bezeichnet eine negierte Bedingung, d.h. I wird nur bei $C =$ "false" ausgeführt.

Prozessoren wie der ARM RISC oder der Analog Devices ADSP-210x realisieren bereits ein einfaches Schema für conditional instructions, bei denen Bedingungen in ALU-Flags gespeichert sind. Da diese jedoch bei vielen Befehlen überschrieben werden, sind die Bedingungen hier eher "flüchtiger" Natur, was zu sehr begrenzten Einsatzmöglichkeiten für conditional instructions führt.

In modernen VLIW DSPs wie dem TI TMS320C6x oder Philips Trimedia TM1000 dagegen werden Bedingungen in den normalen Registerfiles gespeichert und können somit "beliebig" lange aufbewahrt werden. Dies wiederum gestattet die bedingte Ausführung großer Programmblöcke unter Einsatz von conditional instructions. Die Entscheidung, ein bestimmtes ITE-Statement mittels bedingter Sprünge oder bedingter Ausführung von Instruktionen zu implementieren, kann große Auswirkungen auf die Ausführungsgeschwindigkeit des Assemblercodes haben. Wir bezeichenen diese beiden Varianten im folgenden als c-jump und c-exec.

Der Einsatz von c-exec kann die Ausführungszeit auf zweierlei Weise reduzieren. Zum einen werden durch den Verzicht auf bedingte Sprünge die o.g. Stall-Zyklen der Pipeline vermieden. Zum anderen entstehen größere sprungfreie Blöcke (sog. "Basisblöcke"). Hierdurch wiederum gibt es bessere Möglichkeiten zur Parallelisierung von Befehlen, was insbesondere für VLIW DSPs von Bedeutung ist.

Zum Thema Ausnutzung von conditional instructions gibt es erst sehr wenige Veröffentlichungen, z.B. [9, 10], welche allerdings nur lokale Optimierungsverfahren vorstellen. Im Gegensatz dazu behandelt die hier vorgestellte Technik beliebig verschachtelte ITE-Statements und berücksichtigt auch den Overhead durch die dabei evtl. einzufügenden zusätzlichen Instruktionen.

3 Implementierungen für ITE-Statements

In diesem Abschnitt werden die beiden Implementierungsvarianten c-jump und c-exec für ein gegebenes ITE-Statement $S = (cond, B_T, B_E)$ besprochen und analysiert. Wir betrachten zunächst den einfachen Fall, daß B_T und B_E keine ITE-Statements enthalten und somit Basisblöcke sind.

3.1 c-jump

Die Standardimplementierung mittels bedingter Sprünge sieht in Pseudocode folgendermaßen aus:

```
                 c := evaluate(cond)
            [c]  goto then_label
                 B_E
                 goto join_label
then_label:      B_T
join_label:      ...
```

Die Bedingung wird in ein Register c ausgewertet, und abhängig von deren Wert wird entweder B_T oder B_E ausgeführt. Anschließend fließt der Kontrollfluß bei der S folgenden Instruktion wieder zusammen.

Bezeichnet $T(B)$ die Ausführungszeit eines Blocks B und Z die Anzahl der Stall-Zyklen bei einem Sprung (einschließlich der Zeit für den Sprung selbst), so ist die Ausführungszeit für das Statement S bei $c =$ "true" gegeben durch $Z + T(B_T)$ und bei $c =$ "false" durch $2Z + T(B_E)$, da im letzteren Fall zwei Sprungbefehle verarbeitet werden. Die worst case-Ausführungszeit ist somit $\max(Z + T(B_T), 2Z + T(B_E))$.

3.2 c-exec

Bei Verwendung von conditional instructions sieht die Implementierung wie folgt aus:

```
        c := evaluate(cond)
  [c]   B_T
  [!c]  B_E
```

Die Notation "[c] B" bezeichnet dabei die bedingte Ausführung aller Instruktionen in Block B. Abstrakt gesehen verwandelt das c-exec Schema Kontrollabhängigkeiten im Programm in Datenabhängigkeiten. Die then- und else-Zweige des ITE-Statements werden dabei aneinandergehängt ("konkateniert"). Die Ausführungszeit ergibt sich somit als $T(B_T \circ B_E)$, wobei "$\circ$" die Konkatentation von Blöcken bezeichnet.

Insgesamt ist c-exec also genau dann schneller als c-jump, wenn

$$T(B_T \circ B_E) < \max(Z + T(B_T), 2Z + T(B_E))$$

Da bei der Ausführung typischer Programmblöcke auf VLIW DSPs nicht alle Ressourcen belegt werden, ist $T(B_T \circ B_E)$ i.a. wesentlich kleiner als $T(B_T) + T(B_E)$. Daher ist das c-exec Schema oft schneller. Andererseits ist klar, daß dies nicht notwendigerweise der Fall ist, so daß die schnellste Implementierung jeweils sorgfältig ausgewählt werden muß.

3.3 c-jump mit Precondition

Bisher haben wir nur "innere" ITE-Statements $S = (c, B_T, B_E)$ betrachtet, bei denen B_T und B_E Basisblöcke sind. Sind die Instruktionen in einem Block B Zuweisungen oder Sprünge, so kann eine Bedingung c für diese Instruktionen I einfach durch Bildung einer conditional instruction "[c] I" berücksichtigt werden.

Enthält jedoch bspw. der Block B_T wiederum ein ITE-Statement $S' = (c', B'_T, B'_E)$ so ist dessen Implementierung komplizierter, denn sowohl B'_T als auch B'_E dürfen (unabhängig von c') nur dann ausgeführt werden, wenn $c =$ "true" ist. Die Ausführungsbedingung für B'_T ist $c \wedge c'$, und die für B'_E ist $c \wedge NOT(c')$. In diesem Fall bezeichnen wir c als *Precondition* von S'.

Um das korrekte Verhalten eines Programms bei Verwendung von c-exec zu bewahren, müssen Preconditions an innere ITE-Statements weitergegeben werden. Dies läßt sich durch Einfügen zusätzlicher Instruktionen erreichen.

Sei $S = (cond, B_T, B_E)$ ein beliebiges ITE-Statement mit der in einem Register gespeicherten Precondition p. Dann wird das folgende Implementierungsschema verwendet:

```
             [p]  c := evaluate(cond)
             [!p] c := 0
             [c]  goto then_label
             [p]  B_E
                  goto join_label
then_label:       B_T
join_label:       ...
```

Dies Korrekheit ergibt sich folgendermaßen. Das gesamte Statement S darf nur für $p =$ "true" ausgeführt werden. Dies gilt auch für die Auswertung von *cond*, damit Seiteneffekte vermieden werden. Nach Ausführung von "`[!p] c := 0`" ist $c = p \wedge cond$. Für $c =$ "true" wird zu B_T verzweigt. Ansonsten ist entweder p oder c "false". Daher wird p als Precondition an Block B_E weitergereicht, so daß dieser nur unter der Bedingung $p \wedge NOT(cond)$ ausgeführt wird.

3.4 c-exec mit Precondition

Beim Einsatz von c-exec mit Precondition wird das folgende Implementierungsschema verwendet:

```
[p]  c := evaluate(cond)
[p]  d := !c
[!p] c := 0
[!p] d := 0
[c]  B_T
[d]  B_E
```

Auch hier wird die Bedingung nur für $p =$ "true" ausgewertet. In diesem Fall wird auch die Bedingung d auf $!c$ gesetzt. Bei $p =$ "false" werden sowohl c als auch d auf "false" gesetzt. Nach Ausführung dieser zusätzlichen Instruktionen, welche man als "setup"-Instruktionen ansehen kann, enthalten c und d die korrekten Ausführungsbedingungen $p \wedge cond$ und $p \wedge NOT(cond)$ für B_T bzw. B_E.

Abb. 2 verdeutlicht die beiden alternativen Implementierungen für ITE-Statements an einem Code-Beispiel.

4 Abschätzungsverfahren

Zur Auswahl der schnellsten Implementierung eines ITE-Statements $S = (cond, B_T, B_E)$ werden Kostenabschätzungen sowohl für c-jump als auch für c-exec berechnet. Wie aus den

```
C Source              nur c-jump          nur c-exec             c-jump/c-exec

if (a > 10)                 cmpgt a,10,R1       cmpgt a,10,R1          cmpgt a,10,R1
{ d = b + c;          [R1]  jmp L1        [R1]  add b,c,d        [R1]  jmp L1
  if (d > 13)               add f,g,h     [R1]  cmpgt d,13,R2          add f,g,h
  { h = d + e;              jmp L2        [R1]  not R2,R3              jmp L2
  }                   L1:   add b,c,d     [!R1] mov 0,R2         L1:   add b,c,d
  else                      cmpgt d,13,R2 [!R1] mov 0,R3               cmpgt d,13,R2
  { i = d - e;        [R2]  jmp L3        [R2]  add d,e,h        [R2]  add d,e,h
  }                         sub d,e,i     [R3]  sub d,e,i        [!R2] sub d,e,i
}                           jmp L2        [!R1] add f,g,h        L2:
else                  L3:   add d,e,h
{ h = f + g;          L2:
}
```

Abbildung 2: *Illustration der* c-jump *und* c-exec *ITE-Implementierungen*

o.g. Implementierungsschemata hervorgeht, hängt die Existenz einer Precondition für jedes ITE-Statement S' innerhalb von S von der für S gewählten Implementierung ab. Bei c-jump mit Precondition z.B. erhält nur der else-Zweig eine Precondition, während bei c-exec immer beide Zweige eine Precondition erhalten.

Das Vorhandensein einer Precondition beeinflußt aufgrund der notwendigen "setup"-Instruktionen die worst case-Ausführungszeit von S'. Da diese wiederum die Ausführungszeit von S beeinflußt, ergibt sich eine zyklische Abhängigkeit bei der Auswahl der schnellsten Implementierung für S.

Um diesen Zyklus aufzubrechen, verwenden wir einen Zwei-Phasen-Ansatz, der auf dem Prinzip der *dynamischen Programmierung* beruht. In der ersten Phase werden zunächst die Kostenabschätzungen für alle möglichen Fälle berechnet. In der zweiten Phase werden dann auf Basis der Abschätzungen die jeweils besten ITE-Implementierungen bestimmt.

Für jedes ITE-Statement S können genau vier Fälle eintreten, so daß in der ersten Phase vier Kostenwerte für S berechnet werden:

- $T^N_{c-jump}(S)$: Ausführungszeit bei Implementierung durch c-jump ohne Precondition
- $T^N_{c-exec}(S)$: Ausführungszeit bei Implementierung durch c-exec ohne Precondition
- $T^P_{c-jump}(S)$: Ausführungszeit bei Implementierung durch c-jump mit Precondition
- $T^P_{c-exec}(S)$: Ausführungszeit bei Implementierung durch c-exec mit Precondition

Die Berechnung dieser Werte beginnt bei den innersten ITE-Statements, deren then- und else-Zweige Basisblöcke sind, die nur aus Zuweisungen und Sprüngen bestehen. Für solche Statements werden zur Abschätzung Einheitskosten angenommen. Anschließend werden jeweils die Kostenwerte für die nächsthöhere Schachtelungsebene berechnet, wobei die Kosten "innerer" ITE-Statements mit Hilfe der in Abschnitt 3 angegebenen Analyse abgeschätzt werden. Hierbei ist es wichtig, auch die Ausführungszeit $T(B_T \circ B_E)$ von konkatenierten Blöcken geeignet abzuschätzen. Dies erfolgt mittels eines empirisch bestimmten Faktors, welcher die mögliche Parallelisierung der Statements in B_T und B_E reflektiert.

Source	Anzahl ITE-Statements	max. Schachtelungstiefe	Anzahl IR-Befehle
adapt_quant	4	3	16
adapt_predict1	3	1	29
adapt_predict2	6	2	44
diff_comp	2	1	22
outp_conv	4	2	34
code_adj1	5	5	19
code_adj2	17	9	86
code_adj3	17	5	95
detect_pos	7	3	45
find_mv	4	4	45

Tabelle 1: *Benchmark-Charakteristika*

5 Auswahl der schnellsten Implementierungen

Zur Bestimmung der schnellsten Implementierung eines ITE-Statements S ist es offenbar notwendig, die Werte $T_{c-jump}(S)$ und $T_{c-exec}(S)$ zu vergleichen und den kleineren auszuwählen. Hierbei ist es jedoch zunächst nicht klar, ob die jeweiligen T^N- oder T^P-Werte zu betrachten sind, d.h. ob eine Precondition zu berücksichtigen ist. Dies wird nun in der zweiten Phase mit Hilfe der dynamischen Programmierung bestimmt.

Bei der Implementierungsauswahl wird mit dem äußersten ITE-Statement S^* begonnen. Für dieses Statement kann keine Precondition existieren, so daß es ausreicht, die Werte $T^N_{c-jump}(S^*)$ und $T^N_{c-exec}(S^*)$ zu vergleichen um die schnellste Implementierung zu bestimmen. Bei $T^N_{c-jump}(S^*) < T^N_{c-exec}(S^*)$ wird c-jump gewählt, sonst c-exec.

Der Trick ist nun, daß diese Auswahl gleichzeitig festlegt, ob die ITE-Statements S' innerhalb von S^* eine Precondition haben oder nicht. Wird bspw. S mittels c-exec implementiert, so hat S' in jedem Fall eine Precondition, und es genügt der Vergleich $T^P_{c-jump}(S') < T^P_{c-exec}(S')$. Dies läßt sich wiederum für alle ITE-Statements innerhalb von S' fortsetzen, so daß letztendlich die jeweils schnellsten Implementierungen über alle Schachtelungsebenen bestimmt werden können.

6 Experimentelle Ergebnisse

Die vorgestellte Technik wurde für einen TI TMS320C62xx VLIW DSP experimentell ausgewertet. Hierzu wurden 10 "kontrollintensive" C-Benchmarks (größtenteils aus DSPStone [3]) betrachtet, deren Charakteristika in Tabelle 1 genannt sind.

Ausgehend von der Zwischenrepräsentation (IR) der C-Programme (siehe Abschnitt 1) wurde bei der Auswertung folgendermaßen vorgegangen: Die ITE Statements wurden mittels der oben beschriebenen Technik ersetzt, und aus der modifizierten IR wurde sequentieller, symbolischer Assemblercode für den C62xx generiert. Dieser Code wurde anschließend mittels des TI "assembly optimizers" in parallelen Maschinencode übersetzt. Daneben wurde der C Source Code direkt mittels des TI C6x ANSI-C Compilers übersetzt, welcher zur Codeerzeugung die gleichen Techniken verwendet wie der assembly optimizer. Hierdurch wurde sichergestellt, daß die gemessenen Unterschiede in der Codequalität tatsächlich aus der unterschiedlichen Implementierung von ITE-Statements resultieren. Die Ergebnisse zeigt Tabelle

source	nur c-jump	nur c-exec	optimiert	TI C-Compiler
adapt_quant	21	11	11	15
adapt_predict1	12	13	13	13
adapt_predict2	26	21	22	27
diff_comp	9	12	12	10
outp_conv	26	30	24	21
code_adj1	32	23	23	30
code_adj2	57	173	49	51
code_adj3	39	244	30	41
detect_pos	28	27	27	29
find_mv	27	30	30	28

Tabelle 2: *Worst case-Ausführungszeiten*

2. Die Spalten 2 und 3 zeigen die Ausführungszeiten (in Anzahl Befehlszyklen) bei *ausschließlicher* Verwendung von c-jump bzw. c-exec. Die Spalten 4 und 5 zeigen die Ausführungszeiten bei Anwendung der Optimierungstechnik bzw. bei Verwendung des TI C6x ANSI-C Compilers.

In den meisten Fällen wurde durch die Optimierungstechnik – aufgrund eines flexibleren Einsatzes des c-exec Schemas gegenüber dem TI Compiler – schnellerer Code generiert. In einigen Fällen ist der "optimierte" Code langsamer als unter ausschließlicher Benutzung von c-jump bzw. c-exec. Der Grund dafür sind Ungenauigkeiten bei der Abschätzung der Ausführungszeiten. Die beiden größeren und tiefer verschachtelten Source Codes ("code_adj2" and "code_adj3") zeigen allerdings, daß eine starre Benutzung von c-jump oder c-exec ungünstig ist, da sich das Optimum i.a. zwischen diesen beiden Extremfällen befindet.

7 Zusammenfassung

Conditional instructions sind eine wichtige Architektureigenschaft neuerer VLIW DSPs, welche in Compilern bisher nur unzureichend ausgenutzt werden. Sie ermöglichen häufig eine wesentlich effizientere Implementierung von bedingten Programmanweisungen als die klassische Variante mit bedingten Sprüngen. Allerdings dürfte die optimale Ausnutzung von conditional instructions bei verschachtelten Bedingungen aufgrund der schlechteren Lesbarkeit der Programme sogar für geübte Assemblerprogrammierer schwierig sein. Die in diesem Beitrag vorgestellte Technik optimiert die Implementierung von if-then-else Statements, wobei die ersten experimentellen Ergebnisse recht vielversprechend sind. Mögliche Verbesserungen betreffen vor allem die Genauigkeit der Abschätzungsfunktionen zur Bestimmung der Ausführungszeit von Programmblöcken. Statt der hier verwendeten schnellen und einfachen Funktionen könnten auch genauere, auf einem Scheduling-Algorithmus basierende Funktionen eingesetzt werden.

Literatur

[1] Texas Instruments: TMS320C62xx CPU and Instruction Set Reference Guide, URL http://www.ti.com/sc/c6x, 1997

[2] Philips: URL http://www.trimedia.philips.com, 1998

[3] 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

[4] P. Marwedel, G. Goossens (Hrsg.): *Code Generation for Embedded Processors*, Kluwer Academic Publishers, ISBN 0-7923-9577-8, 1995

[5] R. Leupers: *Retargetable Code Generation for Digital Signal Processors*, Kluwer Academic Publishers, ISBN 0-7923-9958-7, 1997

[6] M. Willems, V. Zivojnovic: *DSP Compilers: Product Quality for Control Dominated Applications ?*, ICSPAT, 1996

[7] M. Willems, V. Zivojnovic, H. Meyr: *DSP-Compiler: Produktqualität für kontrolldominierte Anwendungen ?*, DSP Deutschland, 1996

[8] P. Marwedel: *Implementation of IF-statements in the TODOS microarchitecture synthesis system*, IFIP Transactions on Synthesis for Control Dominated Circuits (A-22), 1993, pp. 249-262

[9] S.A. Mahlke, D.C. Lin, W.Y. Chen, et al.: *Effective Compiler Support for Predicated Execution Using the Hyperblock*, 25th Ann. Symp. on Microarchitecture (MICRO-25), 1992

[10] R. Gupta, D.A. Berson, J.Z. Fang: *Path Profile Guided Partial Dead Code Elimination Using Predication*, Int. Conf. on Parallel Architectures and Compilation Techniques (PACT), 1997