Compilertechniken für VLIW DSPs

Rainer Leupers

Universität Dortmund
Informatik 12 (Embedded System Design)
D-44221 Dortmund
leupers@LS12.cs.uni-dortmund.de

Abstract – Für Hochleistungsanwendungen der digitalen Signalverarbeitung werden zunehmend VLIW-Prozessorarchitekturen eingesetzt. Zur effektiven Programmierung von VLIW DSPs werden leistungsfähige C-Compiler benötigt. In diesem Beitrag zeigen wir einige Schwachstellen derzeitiger Compiler für VLIW DSPs auf und stellen hierfür neue Codeoptimierungsverfahren vor. Diese beziehen sich auf die Ausnutzung von SIMD-Befehlen und conditional instructions, sowie effizientes Scheduling und Funktions-Inlining. Experimentelle Ergebnisse für einen bekannten VLIW DSP, den Texas Instruments C6201, zeigen den praktischen Nutzen. Des weiteren werden auch Frontend-Aspekte und maschinenunabhängige Codeoptimierungen kurz behandelt[1]

1. Einleitung

VLIW (*very long instruction word*) DSPs sind eine neue Klasse von digitalen Signalprozessoren für Applikationen mit sehr hohen Leistungsanforderungen. Durch ihr breites Befehlswort können mehrere parallele Funktionseinheiten im Prozessor weitgehend unabhängig voneinander angesteuert werden, wodurch bei entsprechender Taktfrequenz ein sehr hoher Durchsatz erzielt werden kann. Beispiele für neuere VLIW DSPs sind der Texas Instruments C62xx [1] und der Philips Trimedia [2], welche bis zu 8 bzw. 5 parallele Befehle pro Befehlszyklus bei Taktfrequenzen zwischen 150 und 300 MHz erlauben. Dies gestattet eine Rechenleistung bis zu 2400 MIPS, eine Größenordnung mehr als bei bisherigen DSP-Familien.

Nicht nur die Hardwarearchitektur sondern auch die Software-Entwicklungsumgebung ist bei VLIW DSPs anders als bei herkömmlichen DSPs, insbesondere im Bezug auf C-Compiler. Während frühere DSPs aufgrund des relativ schlechten Qualität des compilergenerierten Codes vorwiegend in Assembler programmiert wurden [3, 4], ist dies bei VLIW DSPs kaum noch mit vernünftigem Zeitaufwand möglich. Der Grund ist, dass die hohe Parallelität in VLIW DSPs besondere Programmiertechniken erfordert, welche dem gewöhnlichen sequentiellen Programmierschema widersprechen. Ein Beispiel hierfür ist *software pipelining*, eine Optimierungstechnik für Schleifen [5], bei der in jedem Schleifendurchlauf des Maschinencodes mehrere Iterationen des ursprünglichen C-Codes parallel ausgeführt werden. Die manuelle Programmierung von *software pipelining* ist sehr aufwendig, kann aber leicht automatisiert werden. In der Tat ist dies eine der zentralen Codeoptimierungen im TI C62xx C-Compiler.

Obwohl VLIW DSPs meist eine wesentlich regulärere Architektur aufweisen als klassische Fixed-Point-DSPs, welche für Compiler besser zugänglich ist, gibt es nach wie vor Probleme mit der Codequalität. Ein Grund liegt in der Abhängigkeit des generierten Maschinencodes vom C-Code: Äquivalente Programme können in der Sprache C bekanntlich auf unendlich viele verschiedene Weisen formuliert werden, und die Qualität des Codes kann dabei beträchtlich schwanken [6]. Ein anderer Grund liegt in den Spezialbefehlen von VLIW DSPs, welche durch Compiler schlecht ausgenutzt werden können, sowie in verschiedenen Architektureigenschaften wie z.B. der Auslegung des Datenpfades und des Befehls-Pipelinings.

In diesem Beitrag werden neue Compiler-Optimierungstechniken vorgestellt, welche die speziellen Eigenschaften von VLIW DSPs zur Steigerung der Codequalität ausnutzen: Codegenerierung mir SIMD-Befehlen (Abschnitt 2), Ausnutzung von *conditional instructions* (Abschnitt 3) und Scheduling (Abschnitt 4). Diese beziehen sich auf das *Backend* des Compilers, d.h. den maschinenspezifischen Teil. Daneben stellen wir eine neue Technik zum *function inlining* auf C-Code-Ebene vor (Abschnitt 5), sowie das LANCE-System, eine Plattform zur schnellen Entwicklung von C-Compilern (Abschnitt 6).

[1] Publication: DSP Deutschland, Munich, Oct 2000

2. Ausnutzung von SIMD-Befehlen

SIMD (*single instruction multiple data*) bezeichnet ein Architekturprinzip zur schnellen Verarbeitung von Vektordatenstrukturen: Der gleiche Befehl wird parallel auf verschiedene Datenelemente angewendet. Bei VLIW DSPs wurde dieses Prinzip auch auf der Befehlsebene realisiert. Abb. 1 zeigt ein Beispiel für den TI C6201. Die normalen 32-Bit-Datenregister werden hierzu in je zwei 16-Bit-Subregister aufgespalten, die paarweise addiert werden, wobei die beiden Resultate in das obere bzw. untere Subregister des 32-Bit-Zielregisters geschrieben werden. Der Vorteil solcher SIMD-Befehle (welche es auch für vier Subregister zu je 8 Bit gibt) ist eine bessere Ausnutzung der Prozessorressourcen. Sind nur 16-Bit-Daten zu verarbeiten, so wäre eine Berechnung auf der vollen 32-Bit-Breite des Datenpfades eine Verschwendung. Der ADD2-Befehl aus Abb. 1 nutzt dagegen einen 32-Bit-Addierer wie zwei parallele 16-Bit-Addierer und führt somit idealerweise zu einer Verdopplung der Ausführungsgeschwindigkeit.

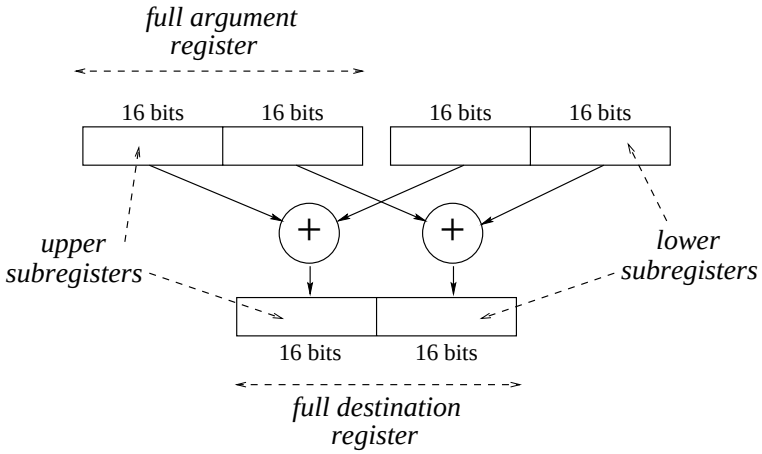


Abbildung 1. *SIMD-Befehl "ADD2" des TI C6201 VLIW DSPs*

Leider können existierende C-Compiler SIMD-Befehle nicht oder nur sehr umständlich ausnutzen. Zum einen geht dies über sogenante *compiler-known functions*, d.h. explizite Aufrufe der SIMD-Befehle im C-Code, oder direkt über Assemblerfunktionen. Darunter leidet allerdings die Portierbarkeit des C-Codes, und auch die Programmentwicklungszeit steigt, da der Programmierer teilweise praktisch auf Assemblerebene arbeiten muss. Wünschenswert wäre dagegen eine Technik, mit der SIMD-Befehle direkt von Standard-C aus ausgenutzt werden können.

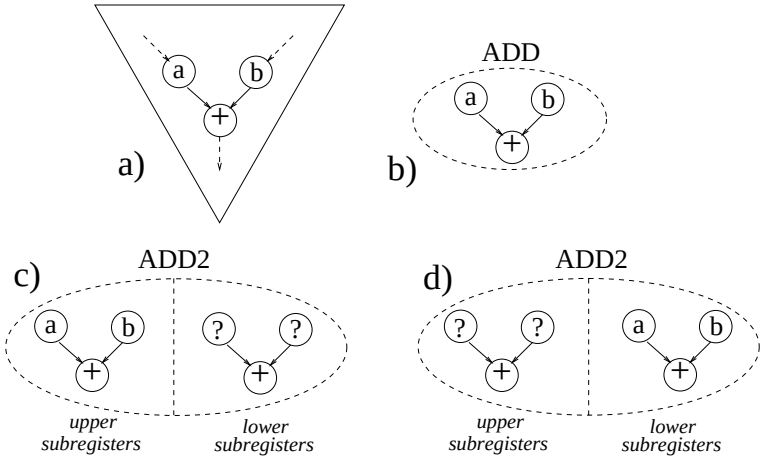


Abbildung 2. *Alternative Überdeckungen einer 16-Bit-Addition*

Um dies zu ermöglichen, kann man eine existierende Technik aus dem klassischen Compilerbau so erweitern, dass auch SIMD-Befehle erzeugt werden können. Dies wird in Abb. 2 skizziert.

Abb. 2 a) zeigt den Ausschnitt eines Datenflussgraphen (DFG). Dieser repräsentiert eine Berechnung auf C-Ebene, wobei

die Graphknoten Operationen und die Graphkanten Datenabhängigkeiten darstellen. Der gezeigte Teilgraph repräsentiert die Berechnung von ”a+b”. Die meisten Verfahren zur Codeerzeugung arbeiten mittels *tree pattern matching* [7, 8]. Hierbei werden DFGs durch Baummuster überdeckt, die jeweils einzelne Instruktionen darstellen. Somit käme als einzige Überdeckung für das Beispiel diejenige aus Abb. 2 b) in Betracht, d.h. eine ADD-Instruktion.

In unserer Technik werden dagegen alternative Überdeckungen betrachtet, z.B. die in Abb. 2 c) und d) gezeigten. Dort wird die zu überdeckende 16-Bit-Addition auf die oberen bzw. unteren Subregister des ADD2-Befehls abgebildet. Somit ergeben sich inkl. des normalen 32-Bit ADD-Befehls insgesamt drei mögliche Überdeckungen. Im Gegensatz zu herkömmlichen Techniken wird nun nicht sofort eine Überdeckung ausgewählt, sondern die Alternativen werden aufbewahrt, bis genügend Information über den gesamten DFG berechnet worden ist. Ob die Variante b) oder die SIMD-Varianten c) oder d) letztendlich möglich sind, hängt stark von dem Kontext ab, da ein ”halber” SIMD-Befehl für sich genommen natürlich kein gültiger Befehl ist. Die Auswahl der alternativen Überdeckungen wird in unserem Ansatz mit Hilfe von *Integer Linear Programming* vorgenommen.

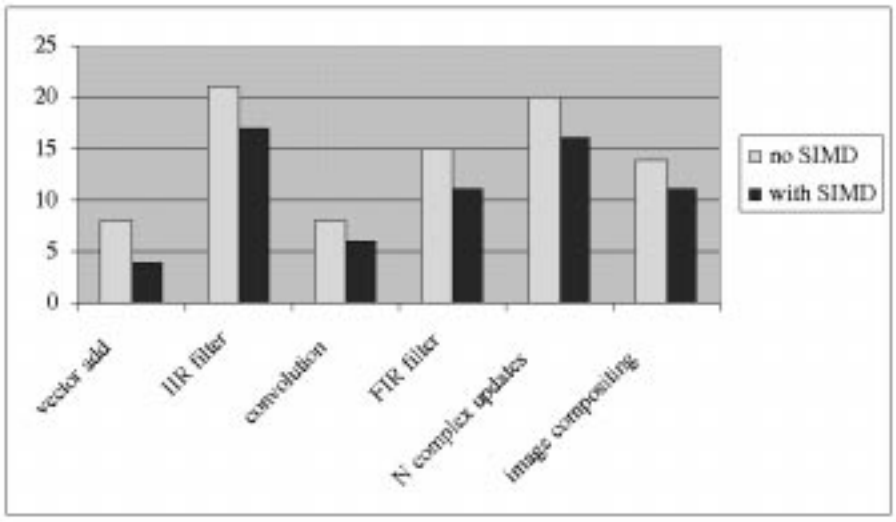


Abbildung 3. *Ergebnisse der Codeerzeugung mit SIMD-Befehlen für den TI C6201*

Abb. 3 zeigt experimentelle Ergebnisse für den TI C6201 DSP. Für einige Beispielcodes (u.a. aus DSPStone [3]) wird die Codegröße der inneren Schleifen angegeben, einmal ohne (links) und einmal mit Ausnutzung von SIMD-Befehlen. Der Vorteil der Ausnutzung von SIMD-Befehlen liegt bei 25 - 50 %. Dies gilt natürlich nur für solche Programmteile, die für SIMD prinzipiell geeignet sind, d.h. Vektoroperationen. Der wesentliche Vorteil unserer Technik liegt darin, dass SIMD-Befehle ohne Verwendung von C-Spracherweiterungen ausgenutzt werden. So lassen sich z.B. exakt die gleichen C-Sourcen auch für den Philips Trimedia compilieren, wobei ebenfalls SIMD-Befehle ausgenutzt werden.

3. Codeoptimierung mit Conditional Instructions

Neben den SIMD-Befehlen sind *conditional instructions* (CIs) ein weiteres Architekturmerkmal neuerer VLIW DSPs. CIs dienen zur Beschleunigung von kontrollintensiven Programmen mit if-then-else-Statements. Die Idee ist, dass jeder Maschinenbefehl eine Bedingung erhalten kann, welche zur Laufzeit entweder *true* oder *false* wird. Nur bei *true* wird der Befehl ausgeführt, während er sich bei *false* wie ein NOP verhält.

Der Einsatz von CIs ermöglicht die Reduzierung von Sprungbefehlen im Maschinencode, welche aufgrund des Befehlspipelinings bei VLIW DSPs eine erhebliche Verzögerung (z.B. bis zu 5 Zyklen beim TI C6201) bewirken können. Allerdings sind CIs nicht immer besser als Sprungbefehle. Der Grund ist in Abb. 4 für einen Prozessor mit zwei *issue slots* (d.h. parallelen Funktionseinheiten) dargestellt.

Hat ein if-then-else-Statement zwei Basisblöcke B_T und B_E im then- bzw. else-Zweig, so lassen sich bei der Verwendung von CIs im günstigsten Fall (Abb. 4 a) die Befehle aus B_T und B_E so parallelisieren, dass die Ausführungszeit nie länger ist als das Maximum von B_T und B_E. Andererseits kann es im Falle von Ressourcen-Konflikten (Abb. 4 b) auch zu einer starken Verlängerung der Ausführungszeit kommen, so dass anstelle von CIs besser bedingte Sprünge zur Umsetzung des if-then-else-Statements eingesetzt werden sollten.

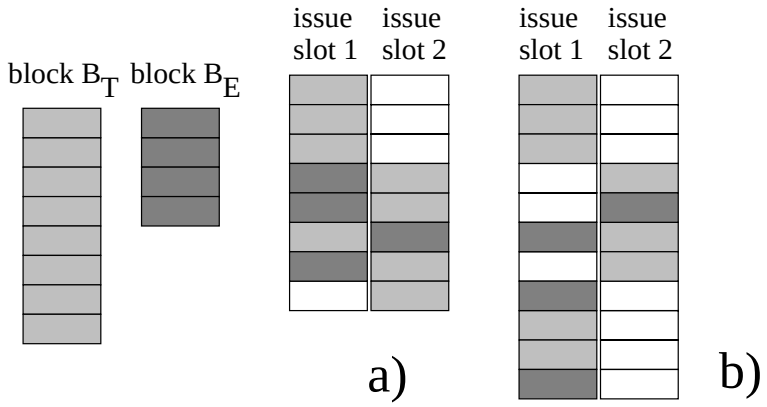


Abbildung 4. *Parallelisierung zweier bedingter Basisblöcke*

```
int A[N+1],B[N+1];

void f1(int c)
{
  if (c)
  {
    A[1] = 1;
     ...
    A[N] = N;
  }

  else
  {
    B[1] = 1;
     ...
    B[N] = N;
  }
}
```

Abbildung 5. *C-Beispiel zu conditional instructions*

Für den TI C6201 DSP wollen wir dies am Beispielprogramm in Abb. 5 näher verdeutlichen. Es handelt sich um ein einfaches if-then-else-Statement mit einer durch N parametrisierten Anzahl von Statements im then- und else-Zweig. Tabelle 1 gibt die Ausführungszeit (in Befehlszyklen) des entsprechenden durch den TI C-Compiler erzeugten Maschinencodes an, wobei einmal bedingte Sprünge und einmal CIs eingesetzt wurden. Für $N = 4$ sind CIs überlegen. Ab $N = 8$ werden bedingte Sprünge besser, und für $N = 16$ sind CIs bereits deutlich schlechter.

N	# Zyklen bei Sprüngen	# Zyklen bei CIs
4	14	11
8	17	18
16	25	62

Tabelle 1. *Ausführungszeiten bei Implementierung des Codes aus Abb. 5 mit Sprüngen und CIs*

Zur optimierten Auswahl von CIs bzw. bedingte Sprüngen für beliebig verschachtelte if-then-else-Statements haben wir ein auf *dynamischer Programmierung* beruhendes Verfahren entwickelt. Dieses setzt auf einer baumartigen Repräsentation eines verschachtelten if-then-else-Statements auf und wählt für jede Ebene entweder CIs oder bedingte Sprünge aus. Für Details des Verfahrens sei auf einen früheren DSP Deutschland-Beitrag verwiesen [9]. Wir wollen hier aus Platzgründen nur die Ergebnisse zeigen (Abb. 6). Das Diagramm zeigt die maximale Ausführungszeit von compilergeneriertem Code für 10 C-Beispielprogramme (rechts: TI C-Compiler, links: optimiertes Verfahren). Da mit Abschätzungen gearbeitet wird, ist unser Verfahren nicht in allen Fällen besser als der TI Compiler. Allerdings liegt die durchschnittliche Performanceverbesserung bei 7 %, mit einem Maximum von 27 %.

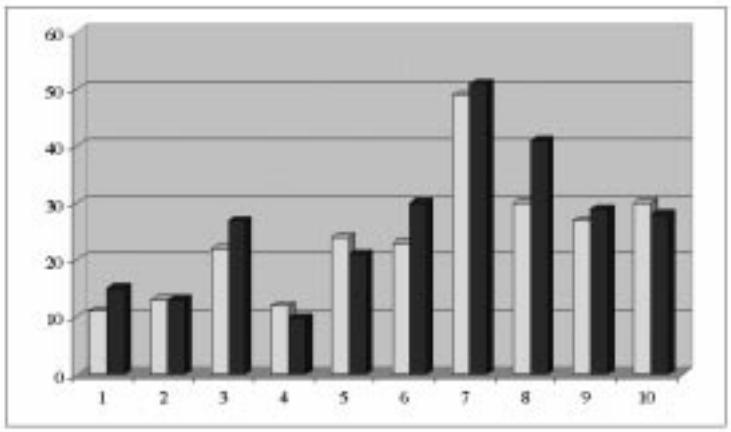


Abbildung 6. *Experimentelle Resultate zur Ausnutzung von Conditional Instructions für den TI C6201*

4. Scheduling für Clustered VLIW

Mit *clustered VLIW* werden Prozessorarchitekturen bezeichnet, deren Register in verschiedene Bänke aufgeteilt sind, und bei denen die Funktionseinheiten immer nur auf eine bestimmte Registerbank zugreifen können. Diese Funktionseinheiten bilden zusammen mit ihrer lokalen Registerbank einen Cluster. Beim TI C6201 z.B. gibt es zwei solche Cluster. Der Grund für die Einführung von Clustern liegt in der Reduktion der Anzahl der kostspieligen Read/Write-Ports der Registerbänke. Der Nachteil ist allerdings eine erschwerte Codeerzeugung, da die Codequalität wesentlich von der geeigneten Zuordnung von C-Operationen zu den Clustern abhängt. Um dies zu verdeutlichen, zeigt Abb. 7 den "geclusterten" Datenpfad des TI C6201. Die Cluster A und B haben jeweils ein 16×16-Bit-Registerfile und vier lokale Funktionseinheiten L, S, M und D.

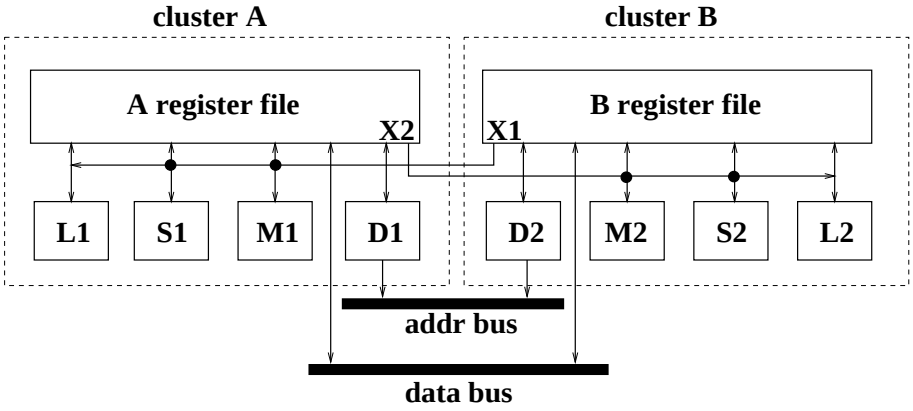
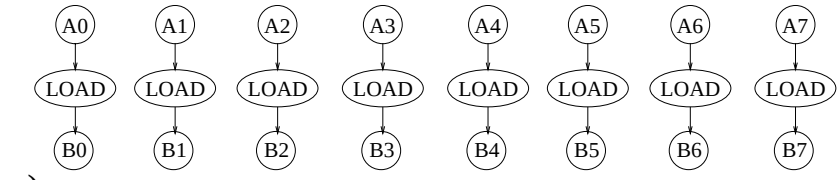


Abbildung 7. *TI C6201 Datenpfad mit Clustern A und B*

Das zentrale Problem bei *clustered VLIW*-DSPs ist die zyklische Abhängigkeit von Cluster-Zuordnung und Scheduling. Wird die Cluster-Zuordnung der Operationen vor dem Scheduling vorgenommen, so ist der Ressourcenbedarf noch nicht vollständig bekannt, und es können suboptimale Schedules resultieren. Wird umgekehrt zuerst das Scheduling durchgeführt, so kann es bei der anschließenden Cluster-Zuordnung zu unnötig vielen *copy*-Operationen kommen, die lediglich dazu dienen, Daten zwischen A und B hin und her zu transportieren. Diese *copy*-Operationen sind häufig ein Engpass beim Scheduling, denn beim TI C6201 müssen diese über die beiden *cross paths* X1 und X2 (siehe Abb. 7) laufen und belegen jeweils eine Funktionseinheit für einen Befehlszyklus.

Die beste Möglichkeit ist daher, Cluster-Zuordnung und Scheduling gemeinsam in einer Phase durchzuführen. Ein Beispiel zeigt Abb. 8. Teil a) zeigt einen sehr einfach strukturierten Datenflussgraphen, bei dem 8 parallele LOADs in Cluster B durchzuführen sind. Die Speicheradressen (Pointer) stehen in den Registern von Cluster A. Teil b) zeigt den vom TI C-Compiler hierfür generierten Assemblercode. Aufgrund von Architekturrestriktionen (bei zwei parallelen LOADs müssten die Pointer in verschiedenen Clustern stehen) werden alle LOADs sequentiell ausgeführt. Im Code in Abb. 8 b) dagegen



a)

```
LOAD *A4, B4
LOAD *A1, A8
LOAD *A3, A9
LOAD *A0, B0
LOAD *A2, B2
LOAD *A5, B5
LOAD *A7, A4
LOAD *A6, B6
NOP 1
MOVE A8, B1
MOVE A9, B3
MOVE A4, B7
```

b)

```
LOAD *A0, A8 || MOVE A1, B8
LOAD *B8, B1 || LOAD *A2, A9 || MOVE A3, B10
LOAD *B10, B3 || LOAD *A4, A10 || MOVE A5, B12
LOAD *B12, B5 || LOAD *A6, A11 || MOVE A7, B14
LOAD *B14, B7
MOVE A8, B0
MOVE A9, B2
MOVE A10, B4
MOVE A11, B6
```

c)

Abbildung 8. *Beispiel zum Scheduling für clustered VLIW*

werden *copy*-Operationen (oder MOVEs) ausgenutzt, um parallel zu den LOADs jeweils einen Pointer von A nach B zu kopieren. Hierdurch können die LOADs parallelisiert werden, und die Ausführungszeit wird von 12 Zyklen auf (optimale) 9 Zyklen reduziert. Der Preis hierfür ist eine höhere Codegröße (16 statt 12 Befehle) aufgrund der *copy*-Operationen.

Zur Berechnung guter Schedules für *clustered VLIW*-DSPs setzen wir eine Kombination von *Simulated Annealing* und *List Scheduling* ein. Das *List Scheduling* bestimmt den Schedule für eine gegebene Cluster-Zuordnung, während das *Simulated Annealing* aufgrund von Rückmeldungen des Schedulers die Cluster-Zuordnung selbst optimiert.

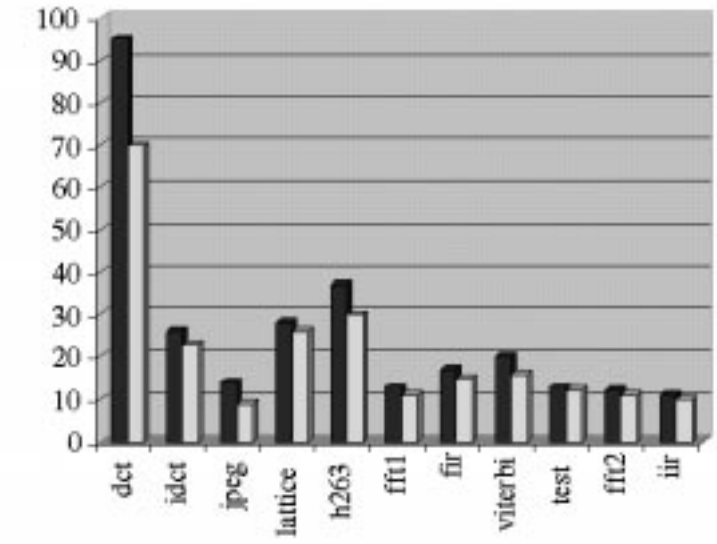


Abbildung 9. *Experimentelle Resultate zum Scheduling für den TI C6201*

Abb. 9 zeigt experimentelle Ergebnisse unseres Verfahrens (rechte Balken) für einige DSP-Routinen im Vergleich zum TI C-Compiler (linke Balken). Angegeben wird jeweils die Anzahl benötigter Taktzyklen. Durch die Kopplung von Cluster-Zuordnung und Scheduling lassen sich Geschwindigkeitssteigerungen bis über 20 % erzielen.

5. Optimales Funktions-Inlining

Inlining von Funktionen bedeutet das direkte Einfügen des Codes der aufgerufenen Funktionen anstelle des ursprünglichen Aufrufbefehls. Funktionen werden somit zu Makros auf C-Ebene. Inlining ist eine weitverbreitete Compilertechnik, die

eine modulare Programmierung mit vielen Funktionen unterstützt, und dabei den mit Funktionsaufrufen verbundenen Overhead (Retten von Registerinhalten, Parameterübergabe, etc.) vermeidet. Allerdings führt Inlining auch zu einer beträchtlichen Erhöhung der Codegröße. Daher werden in C-Compilern meist Heuristiken für das Inlining eingesetzt, welche nur "kleine" Funktionen als Inline-Kandidaten auswählen. Eine vorgegebene maximale Codegröße kann hiermit aber nicht garantiert werden. Ebensowenig kann die vorgegebene Programmspeichergröße optimal zur Performance-Steigerung durch Inlining ausgenutzt werden.

Dies ist insbesondere bei Embedded Systems mit begrenztem *on-chip*-Programmspeicher problematisch. Besser geeignet wäre ein Verfahren, welches die Inline-Funktionen so auswählt, dass eine maximale Performancesteigerung für eine vorgegebene maximale Codegröße erzielt wird. Dies lässt sich mittels eines *branch-and-bound*-Algorithmus erreichen. Für ein gegebenes C-Programm mit n Funktionen $f_1, \ldots, f_n$ stellen wir die Teilmenge der Inline-Funktionen durch den *Inline-Vektor* $IV = (b_1, \ldots, b_n)$, mit $b_i \in \{0, 1\}$, dar. Die Belegung $b_i = 1$ bedeutet, dass f_i eine Inline-Funktion ist, während $b_i = 0$ die Funktion f_i vom Inlining ausschließt. Mit $S(IV)$ bezeichnen wir für einen Inline-Vektor IV die (geschätzte) Gesamt-Codegröße, die natürlich von der ausgewählten Menge der Inline-Funktionen abhängt. Mit $D(IV)$ bezeichnen wir die Anzahl der *dynamischen Funktionsaufrufe* für IV, d.h. die Anzahl der Funktionsaufrufe zur Laufzeit. Setzt man voraus, dass die Minimierung der dynamischen Funktionsaufrufe eine Minimierung der Programmlaufzeit bewirkt, so ist derjenige Inline-Vektor IV gesucht, der $D(IV)$ minimiert und für den $S(IV)$ einen vorgegebenen Wert (z.B. die feste Programmspeichergröße) nicht überschreitet. Mit Hilfe des *branch-and-bound*-Algorithmus und der Berechnung von oberen und unteren Schranken lässt sich der optimale Vektor IV recht effizient berechnen, obwohl es 2^n Möglichkeiten gibt.

Wir haben das Verfahren anhand eines GSM-Coders und den TI C6201 DSP experimentell ausgewertet. Hierbei wurde der GSM-C-Code zunächst ohne Inlining mittels des TI C-Compilers compiliert, wobei sich eine Codegröße von 67.820 Bytes bei einer Laufzeit von 27.400.547 Zyklen ergab. Anschließend wurde die maximal zulässige Codegröße in Schritten zu je 5 % bis zu einem Maximum von 150 % erhöht, wobei jeweils die optimale Menge von Inline-Funktionen bestimmt wurde. Abb. 10 zeigt, wie sich die Anzahl der dynamischen Funktionsaufrufe $D(IV)$ in Abhängigkeit von der maximalen Codegröße ändert. Alle Angaben sind in Prozent. Der erste Balken bezieht sich auf die Ausgangsversion ohne Inlining (Codegröße = Laufzeit = 100 %).

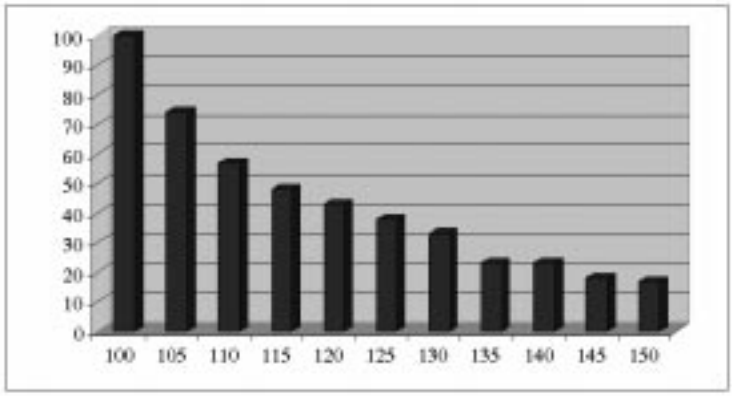


Abbildung 10. *Relative Anzahl dynamischer Funktionsaufrufe für verschiedene Codegrößen-Limits*

Wie zu erwarten, fällt $D(IV)$ monoton mit steigender Codegröße, da mehr Inlining zur einer Reduktion der Funktionsaufrufe führt. Dies gilt jedoch nicht exakt auch für die Laufzeit, welche in Abb. 11 dargestellt ist. Hier ergibt sich das absolute Minimum für ein Limit von 125 % gegenüber der Ausgangsversion ohne Inlining. Der Grund ist, dass $D(IV)$ lediglich als Approximation der Laufzeit verwendet wird, während die tatsächliche Laufzeit auch von anderen Faktoren abhängt. Insgesamt lässt sich zum Preis eines um 25 % vergrößerten Codes die Laufzeit durch Inlining um 33 % reduzieren. Der TI-Compiler dagegen kommt nur auf 3 % bei einem Codegrößenzuwachs von 6 %. Weiterer evtl. verfügbarer Programmspeicherplatz wird zur Laufzeitreduktion nicht ausgenutzt.

6. C-Frontends und Source-Level-Optimierung

Neben Techniken zur Codeoptimierung im Compiler-Backend benötigen C-Compiler auch ein Frontend, welches die Analyse des C-Quellcodes vornimmt und ein maschinenunabhängiges Zwischenformat (*intermediate representation*, IR) generiert. Verfügbare C-Frontends sind z.B. das GNU C Frontend [10], das im SUIF-System [11] integrierte EDG-Frontend

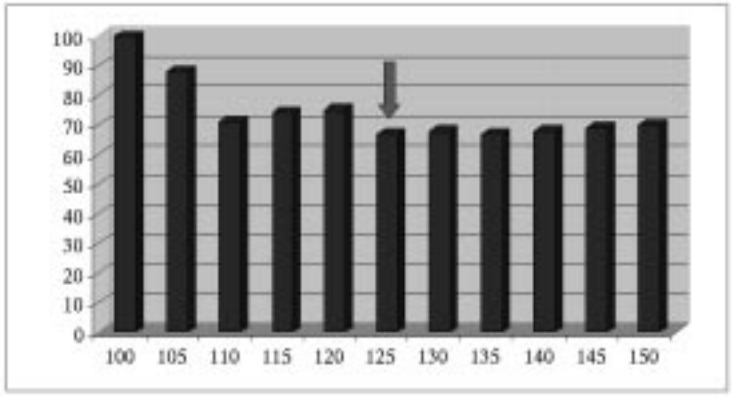


Abbildung 11. *Relative Ausführungsdauer für verschiedene Codegrößen-Limits (Optimum durch Pfeil gekennzeichnet)*

```
int main()
{
  static int A[16], B[16], C[16], D[16];

  register int *p_a = &A[0], *p_b = &B[0];
  register int *p_c = &C[0], *p_d = &D[0];
  register int i;

  for (i = 0; i < 16; i++)
    *p_d++ = *p_c++ + *p_a++ * *p_b++;

  return(0) ;
}
```

Abbildung 12. *C-Code für* `n_real_updates` *(aus DSPStone [3])*

[12], sowie das Frontend des lcc-Compilers [13]. Als ”Standardformat” für die IR hat sich *Drei-Adress-Code* durchgesetzt. Dieser besteht aus einfachen Zuweisungen mit jeweils höchstens drei Variablen (zwei Argumente, ein Resultat) sowie Labels und Sprungbefehlen.

Der Vorteil von Drei-Adress-Code ist seine einfache Struktur, welche die Implementierung von IR-Optimierungen und Compiler-Backends stark erleichtert. Leider geht die *Ausführbarkeit* des C-Codes nach der Übersetzung in die IR i.d.R. verloren, d.h. der IR-Code kann nicht mehr wie der C-Code compiliert und auf einem Host-Rechner ausgeführt werden. Im Gegensatz zu den o.g. C-Frontends kombiniert das IR-Format des an der Universität Dortmund entwickelten Compilersystems LANCE die Einfachheit von Drei-Adress-Code mit der Ausführbarkeit von C-Code. Dies wird dadurch erreicht, dass die LANCE IR eine Teilmenge von C ist. Ein Beispiel für die Umsetzung von C-Code in ausführbaren Drei-Adress-Code zeigen die Abb. 12 und 13. Der IR-Code ist nichts anderes als eine *low level*-Version des ursprünglichen C-Codes. Die ausführbare IR hat verschiedene Vorteile:

- Das C-Frontend sowie auch die IR-Optimierungen können leicht validiert werden. Hierzu werden der Originalcode und der generierte bzw. optimierte IR-Code mittels eines normalen C-Compilers compiliert, und die Ausgaben der compilierten Programme für Testeingaben werden verglichen. Das LANCE-System beinhaltet neben einem C-Frontend auch zahlreiche IR-Optimierungen, z.B. *constant folding*, *dead code elimination* und *common subexpression elimination*. Mit Hilfe der ausführbaren IR und einer umfangreichen Testsuite konnten diese Komponenten jeweils erfolgreich validiert werden.

- Auch backend-spezifische Compilerkomponenten können mittels der genannten Methode validiert werden. Das LANCE-System z.B. verfügt über ein Modul zur maschinenunabhängigen Generierung von Datenflussgraphen (vgl. Abschnitt 2) aus der IR, welche als Eingabe für Codegeneratoren dienen. Die generierten Datenflussgraphen können wie die IR in C-Syntax ausgegeben werden, wodurch auch die Graphen selbst compilierbar und ausführbar bleiben.

- Da die LANCE IR selbst wieder C-Code ist, lassen sich IR-Optimierungen als C-nach-C-Optimierungen einsetzen. Der

```
int main()
{
 char *t1;
 int t2;
 char *t3;
 int *t4;
 register int *p_a_7;

 /* ... */

 /*  *p_d++ = *p_c++ + *p_a++ * *p_b++ ; */

  t20 = p_c_9;
  t22 = (char *)p_c_9;
  t21 = t22 + 4;
  t23 = (int *)t21;
  p_c_9 = t23;
  t24 = p_a_7;
  t26 = (char *)p_a_7;
  t25 = t26 + 4;
  t27 = (int *)t25;
  p_a_7 = t27;
  t28 = p_b_8;
  t30 = (char *)p_b_8;
  t29 = t30 + 4;
  t31 = (int *)t29;
  p_b_8 = t31;
  t32 = *t24 * *t28;
  t33 = *t20 + t32;
  t34 = p_d_10;
  t36 = (char *)p_d_10;
  t35 = t36 + 4;
  t37 = (int *)t35;
  p_d_10 = t37;
  *t34 = t33;

/* ... */

}
```

Abbildung 13. *IR-Code für Schleifenkörper aus* `n_real_updates`

optimierte C-Code kann somit nach wie vor mittels eines beliebigen C-Compilers auf jeden Zielprozessor übersetzt werden, wobei die Codequalität allerdings gegenüber dem Originalcode verbessert wird.

Außer in zahlreichen Forschungsprojekten wird das LANCE-System auch in industriellen Compilerentwicklungsprojekten eingesetzt, welche am Informatik Centrum Dortmund (ICD) durchgeführt werden. Ein neueres Projekt befasst sich mit der Entwicklung eines C-Compilers für den OnDSP^{TM} der Firma Systemonic AG [14]. Hierbei handelt es sich um eine skalierbare DSP-Plattform mit SIMD-kontrollierten Datenpfaden (Abb. 14). Die Datenpfade sind über ein Netzwerk untereinander sowie mit einem breiten Speicher verbunden, aus dem jeweils ein breites Vektordatenwort geladen werden kann. Das Netzwerk erlaubt einen linearen Austausch der Daten, aber auch eine algorithmenspezifische parallele Kommunikation. Die Adressierung der Daten im Speicher erfolgt über eine einfache Adressgenerierung. Jeder Datenpfad beinhaltet ein lokales Registerfile sowie eine ALU mit Multiplikations-Akkumulationseinheit. Die Steuerung des DSPs geschieht über ein VLIW-Befehlswort. Über einen in [15] veröffentlichten patentierten Mechanismus wird dabei eine erhebliche Reduktion der notwendigen Instruktionsbandbreite erreicht. Dadurch wird die Parallelität und Flexibilität eines VLIW-Befehlsworts mit der Kompaktheit eines CISC-Befehlsworts kombiniert, was zu erheblichen Einsparungen bei Instruktionsspeichergröße und Leistungsverbrauch führt.

Der LANCE-basierte Compiler für den OnDSP arbeitet in der ersten Version zunächst nur auf einem der parallelen Datenpfade. Eine Erweiterung zur Ausnutzung der VLIW- und SIMD-Eigenschaften ist geplant.

Das LANCE ANSI C Frontend ist in C++ geschrieben und für Solaris/Linux/Win95-Plattformen per WWW verfügbar [16]. Das Gesamtsystem enthält außerdem eine Bibliothek von maschinenunabhängigen IR-Optimierungen, ein Interface für IR-Zugriff und -Manipulation, sowie ein Backend-Interface zur Erzeugung von Datenflussgraphen. Weitere Details zu LANCE und den einzelnen Codeoptimierungstechniken sind in [17] zu finden.

Literatur

[1] Texas Instruments: www.ti.com/sc/c6x, 2000

[2] Philips: www.trimedia.philips.com, 2000

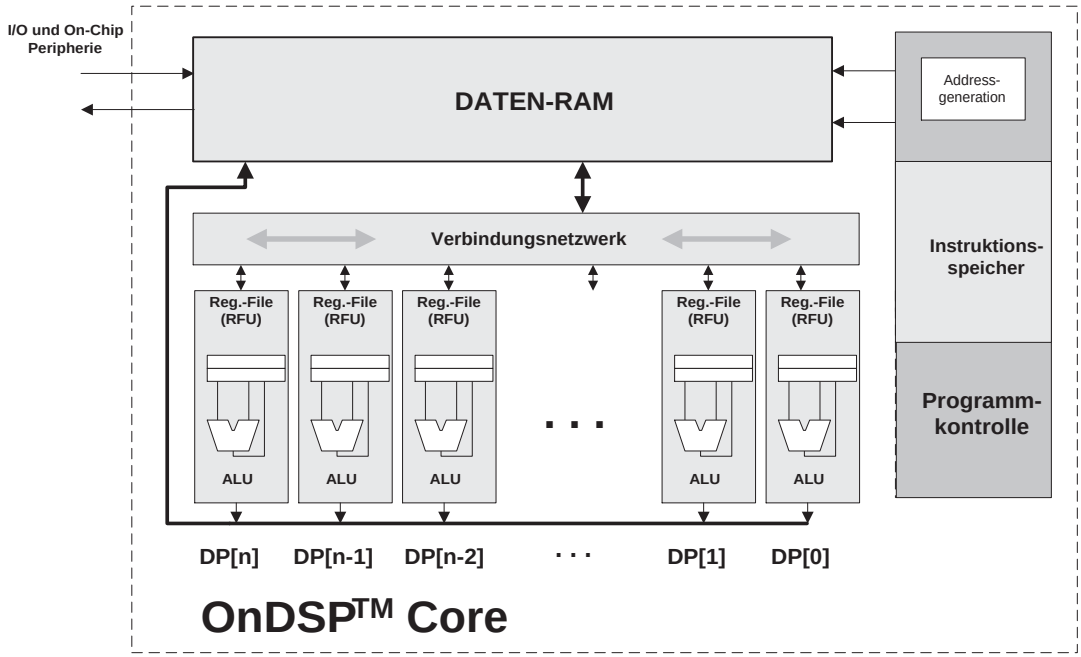


Abbildung 14. *Systemonic OnDSP^{TM}-Architektur*

[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] M. Levy: *C Compilers for DSPs flex their Muscles*, EDN Access, Issue 12, www.ednmag.com, 1997

[5] M. Lam: *Software Pipelining: An Effective Scheduling Technique for VLIW machines*, ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 1988

[6] M. Coors, O. Wahlen, H. Keding, O. Lüthje, H. Meyr: *TI C62x Performance Code Optimization*, DSP Germany, 2000

[7] 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

[8] 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

[9] R. Leupers: *Ausnutzung von Conditional Instructions in VLIW DSP-Compilern*, DSP Deutschland, 1998

[10] Free Software Foundation: www.gnu.org, 2000

[11] The Stanford Compiler Group: suif.stanford.edu, 2000

[12] Edison Design Group: www.edg.com, 2000

[13] C. Fraser, D. Hanson: *A Retargetable C Compiler: Design And Implementation*, Addison-Wesley, 1995, www.cs.princeton.edu/software/lcc

[14] Systemonic AG: www.systemonic.com, 2000

[15] M. Weiss, U. Walther, G. Fettweis: *A Structural Approach for Designing Performance Enhanced DSPs: A 1-MIPS GSM Full-Rate Case-Study*, Proc. IEEE ICASSP, 1997

[16] LANCE software: LS12-www.cs.uni-dortmund.de/∼leupers, 2000

[17] R. Leupers *Code Optimization Techniques for Embedded Processors – Methods, Algorithms, and Tools*, Kluwer Academic Publishers, erscheint 2000