Ein retargierbarer Mikrocode-Compiler und seine Anwendung in Entwurfsverifikation und Architekturbewertung

Lothar Nowak, Peter Marwedel

Institut für Informatik und Praktische Mathematik, Universität Kiel
Olshausenstr. 40-60, D-2300 Kiel, W.Germany

Zusammenfassung

Die vorliegende Arbeit beschreibt einen retargierbaren Compiler. Ein solcher Compiler ist in der Lage für verschiedenen Zielmaschinen (Targets) Maschinencode zu erzeugen. Einsatzgebiete dieses Compilers sind vor allem: die Codeerzeugung für anwendungsspezifische Spezialprozessoren (speziell auch für Maschinen großer Befehlswortbreite), die Verifikation einer manuell erzeugten Hardwarestruktur gegenüber einer Verhaltensspezifikation und der Vergleich verschiedener Architekturkonzepte untereinander. Wesentliche Vorteile des Compilers ergeben sich aus der Tatsache, daß bereits dem Hardware-Designer eine Compiler-Unterstützung zur Verfügung steht. Schnittstellen-Probleme zwischen Compiler- und Hardware-Entwicklung entfallen, da für beide die gleiche Maschinenbeschreibung benutzt wird. So können beim Architekturentwurf verschiedene Konzepte hinsichtlich Laufzeit und Codedichte längerer Anwendungsprogramme verglichen werden.

1 Einleitung

Die Höchstintegration führt zur Fertigung immer komplexerer Architekturen. Dabei ist eine Tendenz zu anwendungsspezifischen Spezialprozessoren erkennbar. Es werden zur Zeit bereits Spezialprozesoren für Aufgaben entwickelt, die früher durch die Programmierung allgemeiner Prozessoren oder durch spezielle Schaltwerke gelöst wurden. Als Beispiele ließen sich Spezialprozessoren zur Unterstützung des Window-Managements von Workstations, Simulationsrechner, spezielle PROLOG-Maschinen usw. anführen.

Wegen der hohen Komplexität der Prozessoren bei gleichzeitig zum Teil relativ niedrigen Stückzahlen setzt eine ökonomische Fertigung leistungsfähige Entwurfswerkzeuge voraus. Diese Werkzeuge müssen es ermöglichen, innerhalb kurzer Zeit eine korrekte Implementierung einer vorgegebenen Spezifikation zu generieren. Diese Werkzeuge sollten insbesondere helfen, verschiedene Architekturkonzepte miteinander zu vergleichen. Sowohl Simulatoren wie auch Synthesewerkzeuge und Verifier dienen dem angegebenen Ziel.

Einer der Vorteile von Simulatoren liegt darin, daß sie dem Entwerfer noch weitgehende Freiheiten in der Implementierung lassen. Genau wie Verifier setzen sie jedoch einen Handentwurf der Implementierung voraus. Mit dem Ziel der Beschleunigung des Entwurfsprozesses wurde in den letzten Jahren eine ganze Reihe von Werkzeugen zur automatischen Synthese von Implementierungen aus einer vorgegebenen Spezifikation entwickelt. Wesentliche Entwurfsentscheidungen sind dabei bereits oberhalb von Gatter- bzw. Schalterebene zu treffen. So wurden denn auch Verfahren zur Synthese von Register-Transfer-Strukturen aus Spezifikationen des gewünschten Verhaltens entwickelt (siehe u.a. [Haf83, Mar86, Par84, Ros86]). Das gewünschte Verhalten liegt dabei gewöhnlich in imperativer Form als PASCAL-ähnliches Programm vor. Bei einem solchen Programm kann es sich z.B. um einen Befehlsinterpreter oder auch um ein Benutzerprogramm handeln.

Trotz der zum Teil langjährigen Vorarbeiten auf diesem Gebiet sind die generierten Strukturen häufig nicht so effizient wie manuell erstellte Entwürfe. Automatisch generierte Entwürfe sind meist nicht grundsätzlich zu verwerfen. Als initialer Startwert einer manuellen iterativen Verbesserung sind sie in der Regel gut geeignet. Das heißt, synthetisierte Strukturbeschreibungen müssen manuell abgeändert werden. Derartige manuelle Eingriffe beinhalten jedoch eine große Gefahr: Während automatisch generierte Strukturen mit hoher Wahrscheinlichkeit korrekt sind, führen manuelle Eingriffe vielfach zu Design-Fehlern. Da alle bisherigen Erfahrungen dafür sprechen, daß manuelle Entwurfsmodifikationen auch in Zukunft erforderlich sind, wurde für das MIMOLA-Hardware-Entwurfssystem [MIM87] ein Werkzeug entwickelt, welches die Korrektheit manueller Eingriffe überprüft.

Um den künftigen Spezialprozessoren eine ausreichende Flexibilität und Anwendungsbreite zu sichern, werden diese vermutlich in gewissem Umfang programmierbar sein. Dieser Aspekt der Programmierbarkeit scheint uns in bisherigen Entwurfswerkzeugen nicht ausreichend berücksichtigt zu sein. Die bisherigen Entwurfswerkzeuge machen keinen Unterschied zwischen Programmspeichern, Programmzählern und den übrigen Hardware-Bausteinen. Dadurch entfällt z.B. die Möglichkeit des Tests durch ein Selbsttestprogramm, wie es mit dem MIMOLA-System generiert werden kann [Krü88].

Generell ist eine programmierbare Rechnerstruktur bezüglich einer vorgegebenen Verhaltensbeschreibung korrekt, wenn sie sich so programmieren läßt, daß sie das vorgegebene Verhalten zeigt. Eine programmierbare Rechnerstruktur ist also bezüglich einer imperativen Spezifikation korrekt, wenn sich die Spezifikation in den Steuercode (d.h. das binäre Maschinenprogramm) der Struktur übersetzen läßt. Die Verifikation kann damit von einem Compiler vorgenommen werden, der ein imperatives Programm in den Code einer vorgegebenen Struktur übersetzen kann. Compiler, deren Zielmaschine durch eine Änderung der eingegebenen Maschine ausgewechselt werden kann, heißen im Englischen **"retargetable"**.

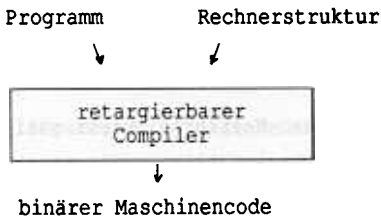


Abb. 1: Ein- und Ausgaben eines retargierbaren Compilers

Ist der Versuch der Codeerzeugung erfolgreich, so ist die Struktur korrekt in dem oben angegebenen Sinne. Mißlingt die Codeerzeugung, so ist entweder die Struktur inkorrekt oder der Codegenerator verfügt über zu wenig semantisches Wissen.

Die Simulation des Rechners allein ist keine Alternative zu dem aufgezeigten Weg. Da in der Regel keine vollständige Simulation möglich ist, kann so die Abwesenheit von Fehlern ohnehin nicht bewiesen werden. Außerdem ist zur Simulation programmierbarer Strukturen auch das binäre Maschinenprogramm erforderlich. Dieses muß bislang (fehleranfällig!) von Hand erstellt werden. Dadurch war es auch bislang kaum möglich, verschiedene Architekturkonzepte hinsichtlich Laufzeit und Codedichte längerer Testprogramme zu vergleichen.

Auch der Vergleich von initialer und modifizierter Struktur durch ein CAD-Systems reicht zur Verifikation nicht aus, da die Verhaltensbeschreibung mitbetrachtet werden muß. Ob eine Modifikation zulässig ist oder nicht, hängt im allgemeinen von der Verhaltensbeschreibung ab.

Durch den Einsatzbereich des retargierbaren Compilers ergibt sich die Forderung, daß Rechnerstruktur-Beschreibungen als Maschinenbeschreibung verwendet werden müssen. Der Compiler muß also aufgrund der Kenntnis der Hardware-Bausteine und deren Verbindungen untereinander Code erzeugen. Der Code steuert die Hardware dann direkt, ohne Interpretation durch ein weiteres Programm. Bei mikroprogrammierten Rechnern ist also der Mikrocode zu erzeugen. Folglich muß der benögtigte Compiler die Fähigkeiten eines Mikrocode-Compilers besitzen. Dazu gehört u.a. die Fähigkeit, Anweisungen zu parallelen Mikrobefehlen zusammenfassen zu können.

Bislang sind nur wenige Verfahren der retargierbaren Mikrocode-Erzeugung bekannt [Bab81,Mue83,Mue84,Veg82,Veg82a,Veg83]. Von diesen Verfahren wird nicht direkt die Rechnerstruktur-Beschreibung als Maschinenbeschreibung akzeptiert. Vielmehr wird

eine manuelle Erstellung von Tabellen vorausgesetzt, aufgrund derer bestimmte Steuercodes in der Codeerzeugung ausgewählt werden.

Für die Verifikation sind diese Verfahren nicht geeignet. Erstens ist die manuelle Tabellenerzeugung selbst fehleranfällig. Zweitens setzt die Tabellenerzeugung Compilerbau-Kenntnisse voraus, die bei Hardware-Designern nicht vorhanden sind.

Ein erster retargierbarer Mikrocode-Compiler für das MIMOLA-System wurde ab etwa 1981 entwickelt und ab 1984 publiziert [Mar84, Mar85]. Aufgrund der Erfahrungen mit diesem Compiler konnte ein zweiter Compiler (**MSSQ**) [Now87] entwickelt werden, der den ersten in der Übersetzungsgeschwindigkeit um bis zu zwei Größenordnungen übertrifft. Diese Leistungssteigerung wurde durch eine Datenstruktur erreicht, die optimal an die Anwendung angepaßt ist. Der erste Compiler dagegen war zum großen Teil regelbasiert. Diese Datenstruktur und der darauf basierende Übersetzungsvorgang sollen im folgenden eingehend beschrieben werden.

2 Der retargierbare Mikrocode-Compiler MSSQ

2.1 Übersicht

Basis des Übersetzungsverfahrens ist der Connection-Operation-Graph (**CO-Graph**), der sowohl die Verbindungsstruktur der Hardware repräsentiert, als auch die zur Ausführung einzelner Operationen nötige Belegung des Instruktionswortes (den Maschinencode) enthält. Das Verfahren basiert im wesentlichen auf einem Pattern-Matching Algorithmus. Dieser versucht die durch das Programm gegebenen Zuweisungen (dargestellt als Baum) auf den durch die Hardware-Beschreibung gegebenen CO-Graphen abzubilden. Ist eine solche Teilstruktur gefunden, wird damit der gewünschte Datenfluß innerhalb der Hardware festgelegt. Die Auswahl der Flußkanten bestimmt die auszuwählenden Multiplexer-Eingänge und damit ihren Kontroll-Code. Die Auswahl der Operatoren bestimmt die Kontroll-Codes der Funktionseinheiten. Ressource-Konflikte werden auf Instruktionsfeld-Konflikte abgebildet und als solche erkannt. Die Darstellung solcher Feldbelegungen und die Konflikterkennung erfolgt anhand von I-Trees (s.u.) und ist wesentlich für die Praktikabilität des Verfahrens.

Der Übersetzungsprozeß zerfällt in drei Phasen: In der ersten Phase, der Preallocation-Phase, wird aus der MIMOLA-Hardwarebeschreibung der CO-Graph aufgebaut. In der zweiten und dritten Phase erfolgt dann die Übersetzung der einzelnen Zuweisungen (Allocation) sowie ihre Zuordnung zu den Instruktionen (Scheduling).

2.2 Definition von I-Trees

Die Teilbelegung von Instruktionsfeldern mit binären Werten und die Verknüpfungen

solcher Belegungen können formal durch I-Trees und darauf definierten Funktionen SET, MERGE und CUT beschrieben werden [Now87].

Der **I-Tree** ist eine baumartige Struktur mit folgenden Eigenschaften:

- Jeder Knoten ist einem Feld des Instruktionswortes zugeordnet. Er enthält genau eine mögliche Belegung dieses Feldes mit einem Bitstring aus '0','1' oder 'X'.
- Die Vater-Sohn Beziehung entspricht einer UND-Verknüpfung der betroffenen Instruktionsfelder: beide Belegungen sind gemeinsam gültig.
- Benachbarte Teilbäume stellen alternative Belegungen des Instruktionswortes (Versionen) dar.
- Jeder Pfad durch den I-Tree entspricht genau **einer** möglichen Belegung des Instruktionswortes. Felder die in einem solchen Pfad nicht auftreten, sind per Definition mit Dont-Cares belegt.
- ROOT(I-Tree) ist ein ausgezeichneter Knoten (die Wurzel). Sein Inhalt ist jedoch ohne Bedeutung.

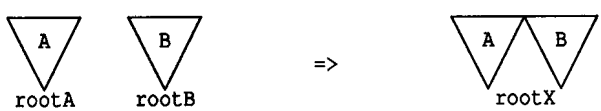
Auf diesen I-Trees sind die Funktionen SET, MERGE und CUT definiert:

SET ist eine Abbildung von Fields x Bitstring -> I-Tree, die ein Feld F des Instruktionswortes mit dem gewünschten Bitstring B belegt. Diese Belegung wird durch den I-Tree X dargestellt:

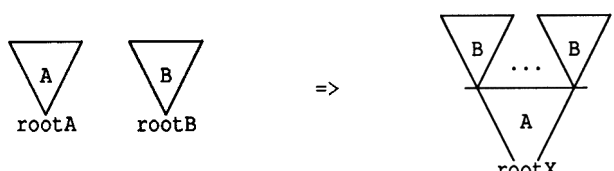
F:B

rootX

MERGE ist eine Abbildung von I-Tree x I-Tree -> I-Tree, die zwei I-Trees A,B überführt in den I-Tree X (die Alternativen A und B):



CUT ist eine Abbildung von I-Tree x I-Tree -> { I-Tree , {} }, die zwei I-Trees A,B überführt in den I-Tree X (den Schnitt von A und B). Dieser Schnitt kann auch leer sein.



Die Funktion CUT verläuft in zwei Phasen: Entsprechend dem Distributivgesetz wird zunächst der I-Tree B an alle Blätter des I-Trees A kopiert. In der zweiten Phase werden alle inkompatiblen Pfade des neu entstandenen I-Trees X gelöscht. (Ein Pfad ist inkompatibel wenn er zwei sich widersprechende Teilbelegungen enthält.)

2.3 Die Grundidee des verwandten Verfahrens

Die Auswahl und Operation von Hardware-Modulen wird gewöhnlich durch die Belegung des Instruktionswortes bestimmt. Die Auswahl desselben Moduls mit verschiedenen Operationen führt zu einem Instruktionsfeld-Konflikt. Unter dieser Annahme können alle Ressource-Konflikte auf Instruktionsfeld-Konflikte zurückgeführt werden.

Ordnen wir also jeder Modul-Operation den entsprechenden I-Tree zu, z.B.:

itree('+_Alu') = SET (ctrfield_Alu , ctrcode('+')),

so können zwei Operationen op1,op2 nur dann gemeinsam ausgeführt werden, wenn ihre I-Trees kompatibel sind (d.h.: CUT(itree(op1),itree(op2)) <> {}).

Die Abbildung aller Ressource-Konflikte auf Instruktionsfeld-Konflikte ist möglich, falls keine Seiteneffekte auftreten. Störend in diesem Zusammenhang sind ("programmierbare") Bus-Konflikte, die eine Sonderbehandlung erfordern. Alle übrigen Seiteneffekte (z.B.: Operations-Auswahl einer unbeteiligten Funktionseinheit) können ignoriert werden.

2.3.1 Phase 1: Preallocation

Das Ziel dieser Phase ist der Aufbau des Connection-Operation-Graphen aus der MIMOLA Hardwarebeschreibung der Ziel-Architektur. Ein Ausschnitt einer solchen Beschreibung - die Deklaration einer ALU - ist in der folgenden Abbildung dargestellt.

```
MODULE Alu (IN in1,in2: word; OUT outp: word; FCT ctr: (1:0));
 BEGIN
  CASE ctr OF
   %00 :  outp <- in1 + in2     AFTER 10 ;
   %01 :  outp <- in2 - in1     AFTER 10 ;
   %10 :  outp <- in1           AFTER  5 ;
  END
 END;
```

Abb. 2: MIMOLA-Beschreibung einer ALU

Diese Modul-Beschreibung kann in einen **M-Graphen** überführt werden. Er enthält die Liste aller Modul-Operationen, ihrer Argumente und Codes. Alle Operations-Bäume sind mit einem gemeinsamen Knoten verbunden, der den Ausgang des Moduls repräsentiert. Speichernden Modulen werden zwei M-Graphen zugeordnet: einer enthält alle Lese-Operationen, der zweite alle Lade-Operationen.

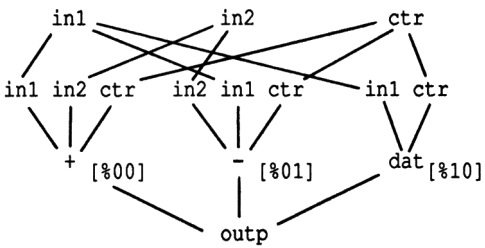
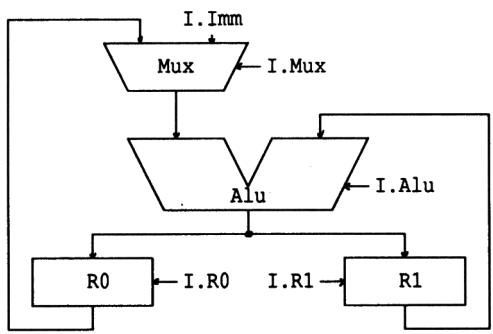


Abb. 3: Beispiel eines M-Graphen

Diese M-Graphen werden nun entsprechend der Hardware-Verbindungsstruktur (Netzliste) zu einem gemeinsamen Graphen (dem CO-Graphen) zusammengefaßt. Die folgenden Abbildungen zeigen eine Hardwarestruktur und den dazugehörigen CO-Graphen (die Interna der M-Graphen sind nicht dargestellt).



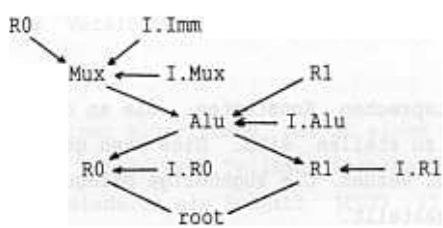


Abb. 4 : Hardwarestruktur und zugehöriger CO-Graph

Anschließend werden auf dem CO-Graphen lokale Transformationen durchgeführt, die der Behandlung von Bussen und konverser Operationen dienen. Außerdem werden in diesem Schritt **via**-Operationen generiert: Dies sind Operationen, die unter einer speziellen Eingangsbelegung die Daten eines anderen Eingangs unverändert weitergeben. Als Beispiel sei die Addition einer '0' oder die AND-Operation mit einem Bitstring von '1' genannt. Solche Operationen können in Verbindung mit ihrem Neutralen Element zum Durchschalten von Daten verwandt werden. Sie werden als zusätzliche Versionen in den CO-Graphen aufgenommen. Die Forderung nach einer speziellen Eingangs-Belegung sei im folgenden als **Precondition** bezeichnet und im Graphen durch ein '!' gekennzeichnet.

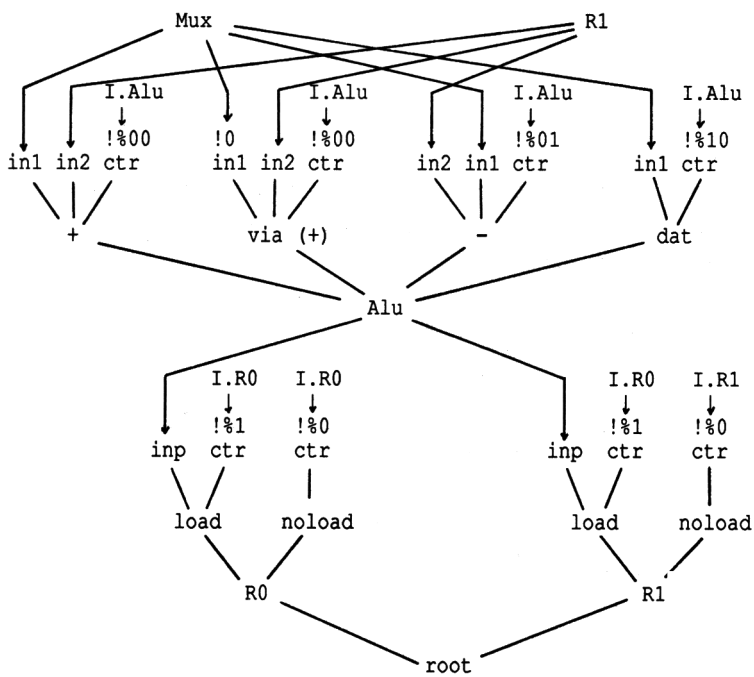


Abb. 5 CO-Graph mit Preconditions

Die Preconditions entsprechen Konstanten, die an dem betreffenden Punkt von der Hardware zur Verfügung zu stellen sind. Dies kann gewöhnlich durch eine geeignete Programmierung erreicht werden. Die zugehörige Belegung des Instruktionswortes wird durch einen I-Tree dargestellt.

Bei der Berechnung dieser I-Trees sind drei Fälle zu unterscheiden:
- Direkt über der Precondition befindet sich ein Instruktionsfeld:
  SET(inst_field,precond_value) liefert den I-Tree.
- Direkt über der Precondition befindet sich eine hartverdrahtete Konstante:
  Paßt diese, so ist der I-Tree = {X}, sonst = {}.
- Ansonsten wird ein Weg durch den CO-Graphen gesucht, der über dat/via-Operationen zu Instruktionsfeldern oder hartverdrahteten Konstanten führt. Der I-Tree ergibt sich dann als Schnitt (CUT) aller entlang dieses Weges gefundenen I-Trees. Gibt es mehrere solcher Wege, so sind die Alternativen durch MERGE miteinander zu verknüpfen.

Kann eine Precondition nicht erfüllt werden (I = {}), so ist die entsprechende Operation niemals ausführbar: der gesamte Operations-Baum kann entfernt werden. Sind jedoch alle Preconditions einer Operation erfüllt, so sind die entsprechenden Eingänge für die folgenden Schritte ohne Bedeutung und werden entfernt. Der zugehörige I-Tree wird dem Operations-Knoten zugeordnet ({...} seien I-Tree Knoten):

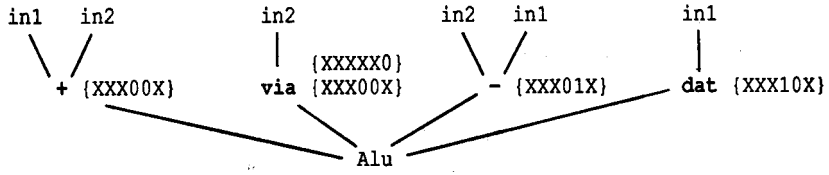
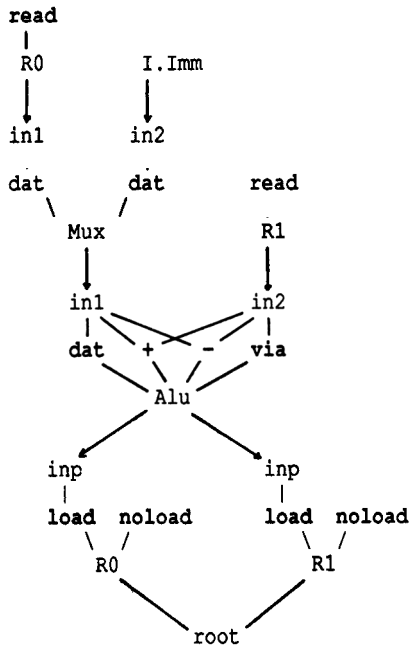


Abb. 6 : M-Graph mit eingetragenen I-Trees

2.3.2 Phase 2: Assignment Allocation

Das Ziel dieser Phase ist die Zuordnung der Operatoren einer Zuweisung zu den vorhandenen Funktionseinheiten sowie die Erzeugung des dazugehörigen Codes. Da hier jedoch nur einzelne Zuweisungen (und keine Instruktionen) betrachtet werden, werden in dieser Phase wiederum nur Teilbelegungen des Instruktionswortes erzeugt, gegebenenfalls jedoch mehrere Versionen. Ihre Darstellung erfolgt weiterhin durch I-Trees.

Die Zuordnung geschieht durch einen Algorithmus, der zu einem vorgegebenen Baum (die Zuweisung des Programms) einen ähnlichen Teilbaum innerhalb des CO-Graphen sucht. Der gesuchte Code ergibt sich wiederum als Schnitt (CUT) aller darin gefundenen I-Trees. Gibt es mehrere solcher Teilbäume, ist die MERGE-Funktion anzuwenden. Das Ergebnis (der zugehörige I-Tree) wird der allokierten Zuweisung zugeordnet. Damit sind **alle** Versionen dieser Zuweisung allein durch ihren I-Tree repräsentiert.



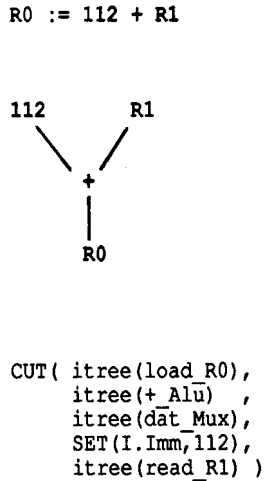


Abb. 7 : Allokation einer Zuweisung

Kann eine Zuweisung nicht allokiert werden, liefert der Algorithmus die mögliche Fehler-Ursache und den möglichen Fehler-Ort. Gegebenenfalls wird der u.U. zu komplexe Ausdruck durch das Einfügen von Hilfszellen-Zuweisungen aufgebrochen und eine erneute Allokation versucht.

2.3.3 Phase 3: **Assignment Scheduling**

In dieser Phase liegt dem System eine Menge allokierter Zuweisungen vor, die in geeigneter Weise einzelnen Instruktionen zuzuordnen sind. Die Zuweisungen selbst stehen in zwei Relationen zueinander: Sie können **datenabhängig** oder **antidaten-abhängig** voneinander sein (s.a. [Mar85]). Entsprechend der daraus resultierenden Nachfolger-Relationen '>' und '>=' kann eine Halbordnung aufgestellt werden, die über die Zuordnung der Zuweisungen zu einzelnen Instruktionen entscheidet.

Mehrere Zuweisungen (xyz) können gemeinsam in eine Instruktion gepackt werden, wenn ihre I-Trees kompatibel sind. Zusätzlich muß jedoch gewährleistet sein, daß alle unbeteiligten Speicher und Register nicht geladen werden. Sei noop(xyz) die dazugehörige Menge, so kann dies anhand der Kompatibilität mit dem I-Tree itree(noop(xyz)) überprüft werden. Insgesamt gilt:

Sei P die Menge aller Zuweisungen des Programms, $m_i \in M \subseteq P$ {i=1,..,n}, dann ist M parallel (in einer Instruktion) ausführbar, g.d.w:

a) $\forall\, m_i, m_j \in M :$
$\quad \exists\, i,j \in \{1,..,n\} : \; m_i > m_j$
$\forall\, m_i, m_j \in M : \; \exists\, x \in P\text{-}M :$
$\quad (x > m_i \wedge m_j > x) \vee (x > m_i \wedge m_j >= x) \vee (x >= m_i \wedge m_j > x)$
(die Halbordnung bleibt erhalten)

b) itree(M) = CUT(itree(m_1), .., itree(m_n <> {}

c) itree(inst) = CUT(itree(M), itree(noop(M))) <> {}

Das Ergebnis dieser Phase ist eine Liste aller gepackten Instruktionen. Die Instruktionen liegen in der Form von I-Trees vor, enthalten also noch Versionen. Jeder Pfad innerhalb dieser I-Trees repräsentiert eine mögliche Version des Maschinencodes.

Anwendungen

Der Vorgänger des hier beschriebenen Codegenerators wurde benutzt, um den SAMP-Prozessor [Now86,Now88] zu entwickeln. Dieser Prozessor sollte u.a. für die Ausführung von PASCAL-Programmen optimiert werden. Typische Ausschnitte aus PASCAL-Programmen wurden daher als Entwurfsspezifikation vorgegeben. Mittels einer älteren Version des MIMOLA-Syntheseverfahrens wurde sodann eine initiale Hardwarestruktur erzeugt. Mit dem Vorläufer [Mar84] des hier beschriebenen Compilers wurde anschließend die Hardware optimiert. Schließlich wurde der Prozessor in TTL realisiert. Er kommt ohne zentralen Takt aus und besitzt eine dezentrale Ansteuerung der Funktionseinheiten. Seine Leistung entspricht in etwa der einer klassischen 3-MIPS-Maschine. Mit einer Befehlswortbreite von 142 Bit gehört er zu den "very long instruction word"-Maschinen (VLIWs). Maschinen dieser Art können nur mittels Compilern wie MSSQ programmiert werden. Ohne solche Compiler wären also diese Architekturen unbrauchbar.

Zur Zeit wird auf dem SAMP ein PROLOG-Interpreter implementiert. Mittels MSSQ wird dazu ein in MIMOLA geschriebener Interpreter der abstrakten Warren-Maschinen in den Befehlscode des SAMP übersetzt. Erste Simulationen lassen eine Leistung im unteren Bereich der Leistung spezieller PROLOG-Maschine erwarten. Dieses Beispiel zeigt, wie mittels MSSQ abstrakte Architekturen durch eine konkrete Maschine realisiert werden können.

Als weitere Anwendung der Mikrocode-Erzeugung im MIMOLA-System wurde Code für einen

Spezialprozessor generiert. Dieser aus AMD-Bitslice-Bausteinen aufgebaute Prozessor dient der Beschleunigung des LOGE-Hardware-Entwurfssystems [Kle86].

Die Leistungsfähigkeit des beschriebenen Compilers konnte im Rahmen des Compilerwettbewerbs auf dem zwanzigsten Mikroprogrammierungs-Workshop (MICRO-20) gezeigt werden: MSSQ war der einzige Compiler, der für die vorgegebene Hardware [Lin87] Mikrocode erzeugen konnte. Diese Hardware enthält 4 Speicher verschiedener Größe, 3 ALU's, sowie eine große Anzahl verschiedener Register und Multiplexer. Das Instruktionswort hat eine Breite von 240 Bits. Für die Retargierung des Compilers (die vollständige MIMOLA-Beschreibung der Hardware) wurden nur 5 Tage benötigt.

4 Zusammenfassung

Es wurde ein retargierbarer Compiler beschrieben. Anwendungsmöglichkeiten des Compilers liegen in der Codeerzeugung für anwendungsspezifische Spezialprozessoren und VLIW-Maschinen, in der Verifikation manuell erzeugter Hardwarestrukturen und beim Vergleich verschiedener Architekturkonzepte untereinander. Abstrakte Maschinen können so auf einer konkreten Hardware realisert werden. Bereits beim Architekturentwurf können verschiedene Konzepte hinsichtlich Laufzeit und Codedichte längerer Anwendungsprogramme verglichen werden.

Das Codegenerations-System benutzt in den Bereichen Allokation und Scheduling neuartige Verfahren. So wurde besonderes Gewicht auf die Behandlung alternativer Lösungen (Versionen) gelegt. Sie werden in kompakter Form durch die Instruktionswort-Belegungen repräsentiert und als I-Trees dargestellt. Spezielle Operationen, die auf diesen I-Trees definiert sind, erlauben eine schnelle Überprüfung von Ressource-Konflikten. Die Repräsentation aller Versionen durch einen I-Tree eröffnet in der Scheduling-Phase die Möglichkeit, die Auswahl einer bestimmten Version so spät wie möglich zu treffen ("delayed binding" [FLS82]). Neuartig ist die Betrachtung von NOOP-Versionen (s.o.) in der Kompaktierung. Sie stellen Seiteneffekte dar, die in retargierbaren Codegeneratoren zu berücksichtigen sind. Alle klassischen Kompaktierungsverfahren nehmen auf sie keine Rücksicht.

Der wesentlichste Unterschied zu anderen Verfahren besteht jedoch in der Art der Retargierung. So wird die Zielmaschinen-Abhängigkeit anhand der reinen Hardware-Strukturbeschreibung dargestellt. Damit werden Schnittstellen-Probleme zwischen verschiedenen CAD-Werkzeugen weitgehend vermieden.

MSSQ löst das Problem, zu einer gegebenen Maschine und einem gegebenen Programm den zugehörigen Maschinencode zu generieren, für VLIW-Maschinen. Es ist geplant, ein entsprechendes Werkzeug auch für programmierbare systolische Arrays wie dem ISA [Lan86] zu entwickeln.

Literatur

[Bab81] T. Baba, H. Hagiwara : The MPG System : A Machine-Independent Efficient Microprogram Generator, IEEE Trans.Comp., Vol. C-30, 6(1981), S.373-395

[FLS82] J.A. Fisher, D. Landskov, B.D Shriver : Microcode Compaction: the State of the Art, Technical Report, TR 82-3-3, University of Southwestern Louisiana, Lafayette, 1982

[Haf83] L. Hafer, A.C. Parker : A Formal Method for the Specification, Analysis and Design of Register-Transfer Level Digital Logic, IEEE Trans. on CAD Vol. CAD-2, 1(1983), S. 4-18

[Kle86] M. Klein: Entwurfs eines mikroprogrammierbaren Coprozessors für die die effiziente Abwicklung von Algorithmen des logischen Entwurfs, (Diplomarbeit), FB Elektrotechnik, Universität Kaiserslautern, 1986

[Lan86] H.W. Lang : The Instruction Systolic Array, A Parallel Architecture for VLSI, Integration, Vol. 4, 1986

[Lin87] J.L. Linn : A Suite of Microarchitectures for Evaluating Microcode Compilers, ACM SIGMICRO Newsletter, Vol. 18, 4(1987), S. 34-48

[Mar84] P. Marwedel : A Retargetable Compiler For A High-Level Microprogramming Language, ACM SIGMICRO Newsletter, Vol. 15, 4(1984), S. 267-274

[Mar85] P. Marwedel : Ein Software-System zur Synthese von Rechnerstrukturen und zur Erzeugung von Mikrocode, Habilitationsschrift, Institut für Informatik und Praktische Mathematik, Universität Kiel, 197 Seiten, 1985

[MIM87] K.Kelle, G.Krüger, P.Marwedel, L.Nowak, L.Terasa, F.Wosnitza: Werkzeuge des MIMOLA-Hardware-Entwurfssystems, Bericht Nr. 8707, Institut für Informatik und Praktische Mathematik, Universität Kiel, Juni 1987

[Mue83] R.A. Mueller, J. Varghese : Flow Graph Machine Models in Microcode Synthesis, 16th Ann. Microprogramming Workshop (MICRO-16), 1983, S.159-167

[Mue84] R.A. Mueller, J. Varghese, V.H. Allan: Global Methods in the Flow Graph Approach to Retargetable Microcode Generation, 17th Ann. Microprogramming Workshop (MICRO-17),1984, S. 275-284

[Now86] L. Nowak : SAMP: Entwurf und Realisierung eines neuartigen Rechnerkonzeptes, (Dissertation), Institut für Informatik und Praktische Mathematik, Universität Kiel, 1986

[Now87] L.Nowak : Graph Based Retargetable Microcode Compilation in the MIMOLA Design System, Proc. 20th Ann. Workshop on Microprogramming (Micro-20), Colorado Springs, Dez.1987, S. 126-132

[Now88] L.Nowak : SAMP: A General Purpose Processor Based on a Self-Timed VLIW Structure, Proc. 10. GI/ITG - Fachtagung: "Architektur und Betrieb von Rechensystemen", Paderborn, Mar.1988

[Par84] A.C. Parker : Automated Synthesis of Digital Systems, IEEE Design and Test of Computers, Vol. 1, 4(1984), S. 75-81

[Ros86] W. Rosenstiel, R. Camposano : The Karlsruhe DSL Synthesis System, in: D. Borrione (Hsg.) : From H.D.L. Descriptions to Guaranteed Correct Circuit Designs, North Holland, 1987

[Veg82] S.R. Vegdahl : Local Code Generation and Compaction in Optimizing Microcode Compilers (Dissertation), Bericht CMUCS-82-153, Carnegie-Mellon Universität, Pittsburgh, 1982

[Veg82a] S.R. Vegdahl : Phase Coupling and Constant Generation in an Optimizing Microcode Compiler, 15th Ann.Microprogramming Workshop (MICRO-15), 1982 S. 125-133

[Veg83] S.R. Vegdahl : A New Perspective on the Classical Microcode Compaction Problem, SIGMICRO Newsletter, Vol. 14, 1 (1983), S. 11-14