Eine effiziente Kontrollfluss-Repräsentation für die Erzeugung von Datenpfaden

Nico Kasprzyk, Andreas Koch, Ulrich Golze, Michael Rock

Technische Universität Braunschweig (E.I.S.)
Mühlenpfordstraße 23
D-38106 Braunschweig
Email: {kasprzyk, koch, golze, rock}@eis.cs.tu-bs.de

Kurzfassung: Rechenleistung wird klassisch durch schnellere oder zusätzliche Mikroprozessoren gesteigert. Dagegen beschleunigen *adaptive Rechner* Applikationen durch eine teilweise Auslagerung in konfigurierbare Hardware. Diese Arbeit führt eine neue Zwischendarstellung ein für die Übersetzung von C-Code in Programme für adaptive Rechner. Dabei werden Kontroll-und Datenfluss gegenüber anderer in der Hardware-Synthese verwendeten Darstellungen insofern besser dargestellt, als Optimierungen besser abschätzbar werden. Die Konvertierung in Hardware-Datenpfade wird durch die Bündelung von Datenfluss-Informationen einfacher und effizienter. Außerdem lassen sich bei Bedarf durch den Abbruch nicht spekulativer Berechnungen schnellere Datenpfade erzeugen.

1 Einleitung

Zur Lösung des steigenden Bedarfs an Rechenleistung sind rekonfigurierbare oder adaptive Architekturen eine vielversprechende Ergänzung zu herkömmlichen Prozessoren. Adaptive Computer bieten die Möglichkeit, Teile ihrer Hardware an den jeweiligen Algorithmus anzupassen. Oft unterstützt die rekonfigurierbare Einheit in solchen Systemen einen Standard-Prozessor, welcher u.a. Verwaltungsbefehle wie I/O-Operationen ausführt. Rechenintensive Programmteile dagegen werden in Hardware realisiert.

Adaptive Computer werden bisher noch nicht in großem Umfang verwendet. Das liegt unter anderem an ihrer schwierigen Programmierung auf einer niedrigen Beschreibungsebene. Neben Erfahrungen in der konventionellen Software-Programmierung muss ein Anwender auch fundierte Kenntnisse in Bereichen der System-Architektur und der Hardware-Synthese besitzen.

Zur Linderung dieses Problems können adaptive Rechenprogramme auch in einer höheren Sprache wie C formuliert werden [4,5,11]. Ein Compiler übernimmt dann die Partitionierung in Hardware (*HW*) und Software (*SW*), die Schnittstellengenerierung und die Planung der Hardware-Ausführung.

Als Nachfolger des von uns in Zusammenarbeit mit der Universität Berkeley und der Fa. Synopsys durchgeführten Projekts *A Nimble Compiler for Agile Hardware* [6] werden bisher bewährte Methoden übernommen.

Ein wichtiger Aspekt bei der automatischen Umsetzung einer Applikation in ein lauffähiges Programm für adaptive Hardware ist die Wahl einer geeigneten Zwischendarstellung, die allen Erfordernissen an die getrennte Behandlung von Software und Hardware sowie die Unterstützung der Hardware-Implementierung als Datenpfad entgegenkommt. Dieser Beitrag diskutiert die Eignung einer von uns entwickelten Darstellungsform von Kontrollflussgraphen.

2 Diskussion

Für die Erzeugung von Hardware aus Hochsprachen wie C werden verschiedene Zwischendarstellungen verwendet. Oft sind diese aus anderen Forschungsgebieten wie dem Parallelrechnen übernommen worden. Die Anforderungen dort sind denen bei der Generierung von Programmen für adaptive Rechner ähnlich.

Wichtig ist vor allem die unkomplizierte Behandlung von Datenflüssen in der Zwischendarstellung. Die Realisierung in der rekonfigurierbaren Logik (*RL*) ist insbesondere für datenflussdominierte Teile von Applikationen geeignet, daher ist eine Konvertierung in einen Datenflussgraphen von zentraler Beteutung.

Projekte zur Hardware-Synthese bauen häufig auf Zwischendarstellungen wie Kontroll-Datenfluss-Graphen (*CDFG*) oder mit Def-Use-Chains erweiterten

Kontrollfluss-Graphen (*CFG*) auf. Für unseren Anwendungsfall sind beide Formen zwar geeignet, konnten aber nicht allen Anforderungen gerecht werden. Programme in C lassen sich ohne größere Aufwendungen für Konvertierungen nicht in CDFGs darstellen. So fehlen z.B. Knoten, die Hochsprachen-Konstrukte wie `while`-Schleifen aufnehmen können.

Dieser Anforderung kommt eine Darstellung als CFG mehr entgegen. Der hierin fehlende Datenfluss wird durch eine Analyse der Zuweisungen und Verwendung von Variablen in den Blöcken des CFG ermittelt. Informationen, welche die gleiche Variable betreffen, werden miteinander verkettet (sogenannte Def-Use-Chains). Anwendung findet dieses Verfahren beispielsweise bei der Reduzierung der Bitbreite in Hardware-Implementierungen [12].

Hier zeigt sich, dass diese Darstellung gut geeignet ist, um Optimierungen durchzuführen, in denen der Datenfluss nicht modifiziert wird. Die Situation ändert sich aber, wenn die Darstellung auf Grund von gebräuchlichen Optimierungen wie dem Abrollen von Schleifen transformiert werden muss. In diesem Fall verändern sich auch die Datenabhängigkeiten, und die Verkettungen zwischen Variablen-Definitionen und -Verwendungen müssen wieder neu aufgebaut oder korrigiert werden. Außerdem sind die Datenflussabhängigkeiten nicht direkt in der Zwischendarstellung vorhanden, sondern müssen separat dazu verwaltet werden.

Wir möchten in dieser Arbeit eine erweiterte Darstellung des CFG, die Static-Single-Assignment-Form (SSA-Form) [1] motivieren. Diese zeichnet sich vor allem durch ihre gute Einbindung von Datenflussabhängigkeiten in den Kontrollfluss aus. Dazu gibt Abschnitt 3 einen kurzen Einblick in die Vorgehensweise des derzeit in der Entwicklung befindlichen Compilers COMRADE (Compiler für adaptive Systeme). In Abschnitt 4 wird die von uns entwickelte DFCSSA-Form (Data Flow Controlled SSA) beschrieben, und deren Vorteile für die Hardware-Synthese und speziell für die Programmerstellung für adaptive Rechner werden vorgestellt. Abschnitt 5 diskutiert den Mehraufwand für die DFCSSA-Form im Gegensatz zu einer einfachen SSA-Form.

3 Der Compiler COMRADE

Der COMRADE Compiler in **Abbildung 1** hat als Ziel, aus C-Quelltexten auf adaptiven Rechnern lauffähige Programme automatisch zu erzeugen. Er setzt auf dem Compiler-System SUIF2 der Universität Stanford [9] auf und erweitert es durch die DFCSSA-Form. Nach High Level-Optimierungen [2] wird der eingelesene C-Code in die DFCSSA-Form konvertiert. Auf dieser Darstellung wer-

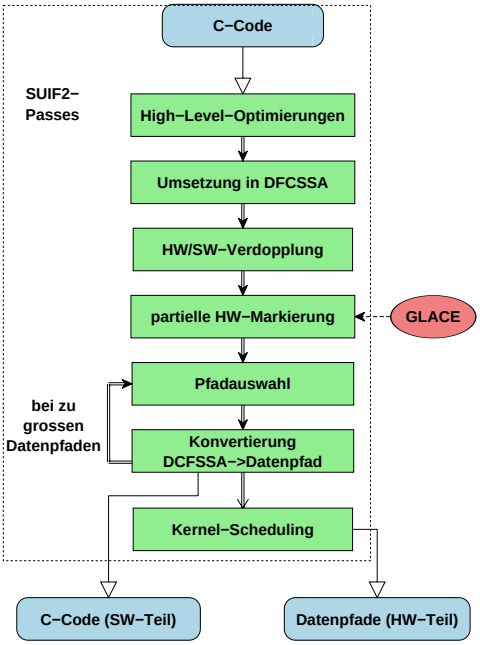

Abb. 1: Der Compiler-Fluss

den dann alle nachfolgenden Compiler-Durchläufe durchgeführt.

Der Compiler wählt Operationen für eine spätere Hardware-Implementierung auf einer Abstraktionsebene aus, die vergleichbar mit der von C ist. Hierbei sind also noch Hochsprachen-Konstrukte wie `for`-Schleifen und `if`-Verzweigungen enthalten. Während eines Compiler-Schritts werden für alle atomaren Operationen wie Additionen hardware-relevante Daten wie Laufzeit und Flächenverbrauch bestimmt. Die Laufzeit dieser Einzel-Operationen wird durch Umformungen wie z.B. dem Abrollen von Schleifen oder dem Übersetzen von `switch`-Anweisungen in `if`-Anweisungen nicht verändert. Deshalb kann der Schritt zu diesem Zeitpunkt ausgeführt werden.

Um die Implementierbarkeit von SW-Operationen in HW zu bestimmen, stützt sich der Compiler auf Schätzwerte aus einer Modulgenerator-Bibliothek. Wir benutzen die bei uns entwickelte Bibliothek GLACE [10], auf die unter Benutzung der flexiblen Generatorschnittstelle FLAME-Schnittstelle zugegriffen wird [8]. Diese Kriterien werden zusammen mit Profiling-Informationen zur Auswahl von Regionen im CFG für die HW-Realisierung herangezogen. Zur Zeit verwenden wir noch dynamisches Profiling. Dieses soll in Zukunft durch ein statisches wie in [3,14] beschrieben ersetzt werden.

Für die Auswahl von HW-Regionen beschränken wir uns auf rechenintensive Schleifen. Nun lohnt es sich nicht immer, jede als geeignet ausgewählte Region komplett in Hardware auszuführen. Manchmal ist es sinnvoll, einzelne Pfade der Region für eine HW-Ausführung vorzusehen. Dieser Fall tritt z.B. ein, wenn auf einem Pfad der Region

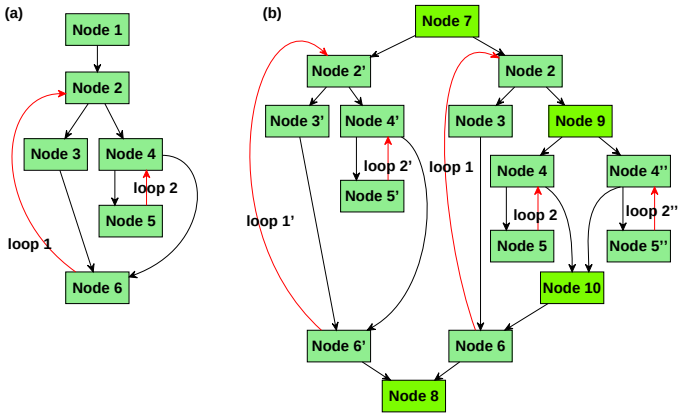

Abb. 2: Schleifenduplikation

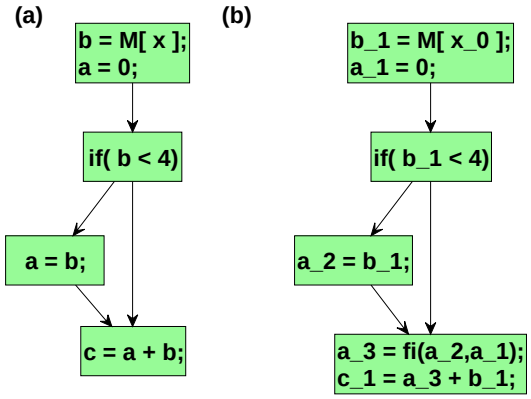

Abb. 3: CFG (a) und CFG in SSA-Form (b)

I/O-Anweisungen des Betriebssystems liegen, die nicht in HW ausgeführt werden können. Desweiteren kann es während des Laufs eines Programms vorkommen, dass eine im Prinzip geeignete HW-Region wegen unpassender Eingabedaten in Einzelfällen nicht auf der HW ausgeführt werden kann. Darüber kann zur Übersetzungszeit aber nicht vorab entschieden werden.

Deshalb werden alle Kandidaten für eine HW-Ausführung im folgenden Schritt dupliziert. Dazu werden diese Bereiche im CFG verdoppelt (analog zu GarpCC [4]). Durch eine zusätzlich eingefügte Verzweigung kann auch zur Laufzeit noch zwischen HW- und SW-Ausführung gewählt werden. Ein Beispiel für die Duplikation ist in **Abbildung 2** zu sehen. Hier werden zwei ineinander geschachtelte Schleifen dupliziert. Dadurch ist es möglich, auf die ursprüngliche SW-Version zurückzugreifen, wenn sich während der Übersetzung oder Laufzeit zeigt, dass eine Hardware-Ausführung nachteilig ist.

Außerdem kann man erkennen, dass nicht nur größtmögliche Regionen dupliziert werden. Im Beispiel wird sowohl nur die innere Schleife, als auch die gesamte Region mit beiden Schleifen dupliziert. So ergibt sich die Freiheit, die HW-Realisierung nur der inneren Schleife zu benutzen, wenn die zweite Region bespielsweise zu groß für die Ziel-RL ist. Während dieses Compiler-Schritts werden auch Blöcke in den CFG eingefügt, in denen zu einem späteren Zeitpunkt die Schnittstellen zwischen HW- und SW-Teilen der Applikation generiert werden.

Nach der Bearbeitung aller Regionen setzt ein auf der DFCSSA-Form basierender Algorithmus den CFG dann in den Datenpfad für die Ziel-RL um. Die erzeugten Datenpfade werden einem Scheduling unterzogen und anschließend mit speziell an adaptive Rechner angepassten Werkzeuge weiterverarbeitet [7].

4 Darstellung in SSA-Form

Die Anforderungen an eine Zwischendarstellung, welche sich für die Erzeugung von DFGs eignet, sind vielfältig. In unserem Fall müssen wir die Programmiersprache C so gut wie möglich abbilden sowie Kontroll- und Datenfluss darstellen können. Desweiteren sollten Änderungen am Kontrollfluss sofort Rückschlüsse auf Änderungen des Datenflusses erlauben. Eine ständig wiederholte Analyse des Datenflusses wie in der Def-Use-Chain-Darstellung sollte dabei vermieden werden. Als Lösung schlagen wir die Benutzung der SSA-Form von CFGs vor, in der Datenfluss direkt im CFG vorhanden ist.

Bei der Umwandlung einer Applikation in die SSA-Form wird für jede Variablen-Definition ein neuer Variablennamen eingeführt. Die Zuweisungen der Variablen `a` in **Abbildung 3** werden beispielsweise zu den Definitionen von `a_1` und `a_2`. Hierbei kann es Kollisionen geben, wenn sich Definitionen einer Variablen von verschiedenen Wegen durch den CFG an einem Knoten treffen. An dieser Stelle wird eine sogenannte Fi-Anweisung eingeführt, welcher die verschiedenen Definitionen wieder zu einer vereint. Im Beispiel wird die Definition von `a_3` durch eine Fi-Anweisung erzeugt.

Allein durch die Konvertierung in die SSA-Form ergeben sich Vorteile für die spätere HW-Implementierung. Durch das Einführen eines neuen Variablen-Namens an jeder Stelle, an der sich eine Zuweisung befindet, können Lebenszeiten von Variablen verkürzt werden. Im Pro-

```
for( i=0; i< 10; i++ ) {
  a[i] = 0;
}
for( i=0; i<10; i++ ) {
  b[i] = 0;
}
```

Abb. 4: Beispielprogramm 1

```
i = 0;
a = 0;
do {
  if(i<15)
    if(i<10) a = 1;
  else     a = 2;
  c = a+1;
  i=i+1;
} while( i < 100 );
```

Abb. 5: Beispielprogramm 2

grammstück von **Abbildung 4** wird beispielsweise die Variable `i` in zwei unabhängigen Schleifen gebraucht. Durch die Umwandlung in die SSA-Form kann nun sehr leicht durch den Compiler erkannt werden, dass diese Schleifen unabhängig voneinander ausführbar sind. Da hier auch keine weiteren Datenabhängigkeiten bestehen, könnten sie auch parallel ausgeführt werden.

4.1 Die DFCSSA-Form

Für den Einsatz in COMRADE wurde die einfache SSA-Form noch um einige Eigenschaften erweitert, um sie besser an die speziellen Anforderungen anzupassen.

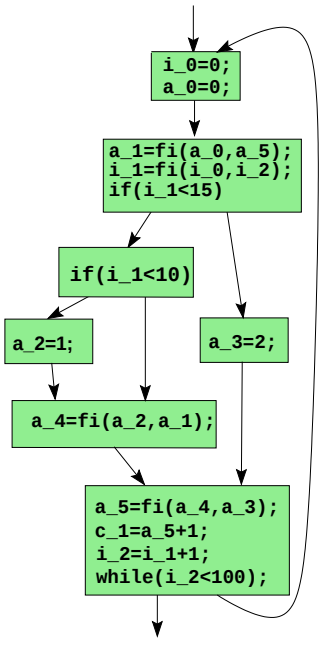

Abb. 6: Einfache SSA-Darstellung

Die erste Änderung betrifft die Platzierung der Fi-Anweisungen. In der einfachen SSA-Form werden diese immer dann an sich vereinende Kontrollfluss-Kanten gesetzt, wenn sich hier zwei Variablen-*Definitionen* treffen. Im Gegensatz dazu werden in der DFCSSA-Form die Fi-Anweisungen nur dann gesetzt, wenn bei der *Benutzung* einer Variablen zwei Definitionen zur Verfügung stehen würden. Eine Übersetzung des Programms aus **Abbildung 5** würde in einer normalen SSA-Form wie in **Abbildung 6** dargestellt aussehen. An allen zusammenführenden Kanten sind hier Fi-Anweisungen eingefügt, obwohl die definierten Variablen (`a_4`) im weiteren Verlauf nicht mehr benötigt werden. Dies wird durch die DFCSSA-Form vermieden. Überflüssige Fi-Anweisungen treten hier nicht mehr auf (**Abbildung 7**).

Durch das Erzeugen von Fi-Anweisungen nur bei Bedarf kann sich deren Anzahl aber auch wieder erhöhen. Dies kann im Beispiel an der mehrmaligen Benutzung der Variablen `a` erkannt werden. Hier haben die Fi-Anweisungen für die Definitionen von `i_1` und `i_3` dieselben Argumente. Dieses auf den ersten Blick unnötige Erzeugen von Anweisungen hat Vorteile für nachfolgende Optimierungen auf der DFCSSA-Form. Auf diese wird in dieser Arbeit nicht weiter eingegangen. Außerdem hat dies keinen Einfluss auf die erzeugte Hardware (Anzahl Multiplexer), da Fi-Anweisungen mit gleichen Argument-Listen vor der Konvertierung in den DFG wieder zusammengefasst werden.

4.2 Partielle Bearbeitung

Wie vorher beschrieben, sind nur Teile des gesamten Kontrollflussgraphen dazu gedacht, später in der adaptiven Hardware implementiert zu werden. Der Software-Teil soll durch übliche C-Compiler weiterverarbeitet werden können.

Vorteilhaft ist dafür also eine Darstellungsform, die erstens den Software-Teil weitestgehend unberührt lässt und zweitens die Vorteile der DFCSSA-Form für die in Hardware zu realisierenden Teile nutzt. Als Lösung wurde eine partielle Darstellung entwickelt. In dieser werden nur die für HW vorgesehenen Regionen des CFG in die DFCSSA-Form umgesetzt, wodurch überdies Zeit während der Übersetzung der Applikation gespart wird. Optimierungen können so auf diese Regionen beschränkt werden. Verschiedene Optimierungen sind darüber hinaus nur sinnvoll auf Hardware anwendbar (z.B. Bitbreitenreduktion).

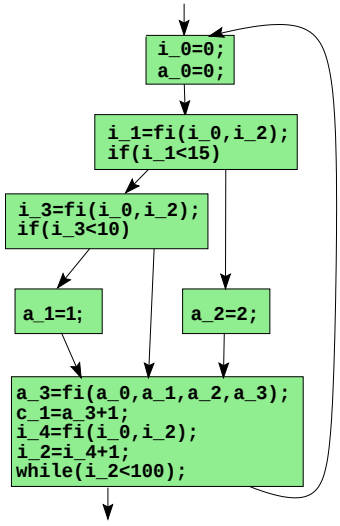

Abb. 7: DFCSSA-Darstellung

Außerdem kann der den Software-Teil bearbeitende Compiler auf den weitestgehend unveränderten Konstrukten des ursprünglichen Quellcodes arbeiten. Zusätzlich eingefügte Anweisungen während des Aufbaus und des Auflösens der DFCSSA-Form könnten den Software-Teil verlangsamen und ein Debugging noch weiter erschweren.

5 Umsetzen in einen DFG

Da die DFCSSA-Form implizit auch den Datenfluss in einem Programm darstellt, kann aus ihr einfach ein Datenflussgraph (DFG) erzeugt werden. Zuvor werden noch (wie in Abschnitt 4.1 beschrieben), Fi-Knoten mit gleichen Argument-Listen zusammengefasst.

Bei der Konvertierung in einen DFG werden die Variablen-Definitionen mit den Stellen verbunden, an deren sie benutzt werden. Der resultierende DFG enthält nun alle Operationen des CFG. Nur die für die SSA-Form neu eingefügten Fi-Anweisungen müssen separat behandelt werden.

Die Fi-Anweisungen beschreiben Stellen, an denen verschiedene Variablen-Definition zu einer zusammengefasst werden müssen. Hier bietet sich die Realisierung als Multiplexer an, welcher aus einer Menge von parallel berechneten Ergebnissen eines auswählt. So wird beispielsweise die in Abbildung 7 enthaltene Fi-Anweisung für `a_4` wie in **Abbildung 8** abgebildet.

Die Daten für die Eingänge des Multiplexers werden parallel berechnet. Nur eines der Ergebnisse an den Dateneingängen wird aber entsprechend dem Wert am Eingang `sel` ausgewählt. Steht diese Auswahl fest, so kann die Berechnung auf den anderen Datenpfaden zu den anderen Eingängen des Multiplexers abgebrochen werden. Ist beispielsweise `sel` nach 12 Takten berechnet und bestimmt das Schalten des Eingangs `a_1,` so kann bereits nach 15 Takten weitergerechnet werden. Das Ergebnis am Ausgang des Multiplexers ist nun stabil und die Berechnung zum Erzeugen von `a_4` wird unterbrochen oder verworfen.

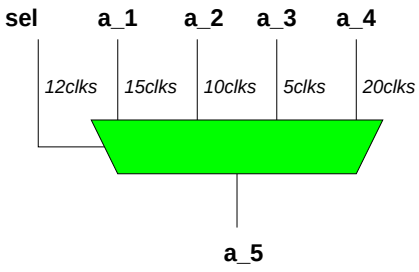

Abb. 8: Realisierung von Fi-Anweisungen als Multiplexer

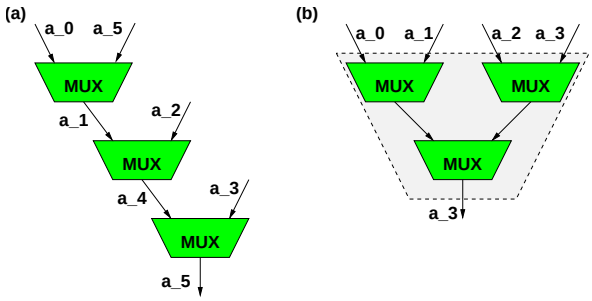

Abb. 9: Multiplexer-Implementierung

5.1 Zusammenfassen von Fi-Anweisungen

Eine spezielle Eigenschaft der von uns entwickelten Form ist das Zusammenfassen von Fi-Anweisungen direkt vor die Benutzung der dadurch neu erzeugten Variablen-Definition. Diese Vorgehensweise kann die Laufzeit von Multiplexern in der endgültigen Hardware reduzieren und erspart außerdem zusätzliche Optimierungen.

Wird das Beispiel aus Abbildung 6 in den DFG übersetzt, ergibt sich eine Multiplexer-Kaskade aus drei Multiplexern (**Abbildung 9**a). Leicht erkennbar ist jedoch, dass man diese drei Multiplexer aber auch so verschalten könnte, dass nur die Laufzeit von zwei statt drei Multiplexern benötigt wird (Abbildung 9b). Diese Optimierung ist hierbei nur durch eine weitere Analyse durchführbar.

Bei der DFCSSA-Form müssen aber nicht drei getrennte Multiplexer behandelt werden. Für deren Fi-Anweisungen wird dort nur vorgegeben, dass ein Multiplexer mit vier Eingängen benötigt wird. Als interne Realisierung wird man hier aber eine bessere Lösung als eine Kaskade wählen. Sie entspricht eher einer Realisierung wie in Abbildung 9b und verringert so die Laufzeit im generierten DFG um 33%.

6 Zusätzlicher Platzverbrauch

Neben den Vorteilen der DFCSSA-Form gegenüber der einfachen SSA-Form besteht ein Nachteil im vermehrten Einfügen von Fi-Anweisungen. Dass dieser Nachteil durch das Zusammenfassen gleicher Fi-Anweisungen vor der DFG-Erzeugung leicht behoben wird, soll an einigen Benchmarks gezeigt werden.

In **Abbildung 10** wird das Anwachsen der in der Zwischendarstellung enthaltenen Anweisungen gezeigt. Dabei sind Anweisungen alle in der Programmiersprache C vorhandenen Zuweisungen sowie alle Konstrukte für Schleifen, Verzweigungen, Sprünge und Marken. Die erste Spalte benennt das untersuchte Programm. In der zweiten Spalte wird die Anzahl der Anweisungen in diesem Programm ohne Fi-Anweisungen angegeben. In den nächsten

Programm	#Anw. ohne SSA	SSA-Form #Fi-Anw.	DFCSSA #Fi-Anw.	DFCSSA #Fi-Anw., opt	Zuwachs unopt.	Zuwachs opt.
adpcm	171	12	10	7	-16%	-41%
pegwit	2154	163	221	135	26%	-17%
jpeg	5027	469	697	441	48%	-6%
fft	104	16	19	5	19%	-31%
wavelet	312	30	51	25	70%	-17%

Abb. 10: Aufwand für die DFCSSA-Form

beiden Spalten sind die Anzahlen der hinzugefügten Fi-Anweisungen in der einfachen SSA- und der DFCSSA-Form aufgeführt. Die folgende Spalte zeigt die Anzahl von Fi-Anweisungen nach dem Zusammenfassen. Die folgenden beiden Spalten geben den prozentualen Mehraufwand der DFCSSA-Form vor und nach der Optimierung von Fi-Anweisungen an.

Offenbar ist die Anzahl der Fi-Anweisungen in der unoptimierten DFCSSA-Form verhältnismäßig hoch. Doch wiegen für uns die Vorteile dieser Form den Mehraufwand, der sich in einem höherem Speicherbedarf während der Compilierung ausdrückt, auf. Außerdem ist zu sehen, dass die optimierte Darstellung sogar noch weniger zusätzliche Anweisungen als die einfache SSA-Form benötigt.

7 Zusammenfassung

In dieser Arbeit wurde eine neue Zwischendarstellung von CFGs für die Übersetzung von C-Quellcode in Programme für adaptive Rechner vorgestellt. Es wurde gezeigt, dass sie sich gut für die Erstellung von Hardware-Datenpfaden eignet und Optimierungen einfacher als bei anderen Darstellungen möglich sind.

Die DFCSSA-Form lässt während der Bearbeitung von Applikationen die Anzahl von Anweisungen gegenüber einer normalen SSA-Form stärker anwachsen. Da dies aber vorteilhaft für die Erstellung von Datenpfaden ist, ist dieser Aufwand akzeptabel.

Literatur

[1] Appel, A., Modern Compiler Implementation in C, Cambridge University Press, 1998

[2] Bacon, D. F., Graham, S. L., Sharp O. J., Compiler Transformations for High-Performance Computing, ACM Computing Surveys 26(4), 1994

[3] Ball, T., Larus, J., Branch Prediction for free, Proc. of the Conf. on Prog. Language Design and Implementation, 1993

[4] Callahan, T., Hauser, R., Wawrzynek, J., The GARP Architecture and C Compiler, IEEE Computer 33(4), 62-69, April 2000

[5] Gokhale, M.B., Stone, J.M., NAPA C: Compiling for a Hybrid RISC/FPGA Architecture, Proc. IEEE Symp. on FPGAs for Custom Computing Machines, 1998.

[6] Harr, R., The Nimble Compiler Environment for Agile Hardware“, Proc. ACS PI Meeting, http://www.dyncorp-is.com/darpa/meeting/acs98apr/Synopsys\%20for\%20WWW.ppt, Napa Valley (CA) 1998

[7] Kasprzyk, N., Koch, A., Advances in Compiler Construction for Adaptive Computers, The 7th International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2000), Las Vegas, USA, June 26-29, 2000

[8] Koch, A., Kasprzyk, N., Module Generator-based Compilation for Adaptive Computing Systems, IEEE Intl. Symp. on FCCMs, Napa Valley (CA, USA), 2002

[9] Monika Lam, An Overview of the SUIF2 System, ACM SIGPLAN ’99 Conference on Programming Language Design and Implementation, http://suif.stanford.edu/suif/suif2/doc-2.2.0-4/tutorial99.ps

[10] Neumann, T., Koch, A Generic Library for Adaptive Computing Environments, Workshop on Field-Programmable Logic and Applications, Belfast, 2001

[11] Li, Y.B., Harr, R., et al. Hardware-Software Co-Design of Embedded Reconfigurable Architectures, Proc. Design Automation Conference, 2000

[12] Mahlke, S., Ravindran, R., Schlansker, M., Schreiber, R., Sherwood, T., Bitwidth Cognizant Architecture Synthesis of Costum hardware Accelerators, IEEE TRANSACTIONS OF COMPUTER-AIDED DESIGN OF INTEGRATED CIRDUITS AND SYSTEMS, VOL. 20, NO. 11, 2001

[13] Rock, M., Implementierung einer SSA-Form zur effizienten Erzeugung von Datenflussgraphen für Adaptive Rechner, Diplomarbeit, TU Braunschweig, 2002

[14] Wu, Y., Larus, J. R., Static Branch Frequency and Program Profile Analysis, In 27th IEEE/ACM Symposium on Microarchitecture (MICRO-27), 1994