Verbesserte Hardware-Software-Partitionierung für Adaptive Computer

Nico Kasprzyk, Andreas Koch

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

Abstract: Rechenleistung wird klassisch durch schnellere oder zusätzliche Mikroprozessoren gesteigert. Dagegen beschleunigen *Adaptive Computersysteme* (ACS) Applikationen durch eine teilweise Auslagerung in konfigurierbare Hardware. Der Compiler COMRADE übersetzt C-Quelltexte in Programme für ein ACS. Er bestimmt hierbei automatisch Regionen für eine Hardware-Beschleunigung. Diese Arbeit behandelt ein erweitertes Verfahren für die Entscheidung zwischen Hardware- und Software-Ausführung. Dabei wird eine neue Auswahltechnik von Pfaden für die Hardware-Realisierung eingeführt. Dieses bearbeitet auch *geschachtelte* Schleifen und nutzt so die konfigurierbare Hardware besser aus.

1 Einleitung

Zur Lösung des steigenden Bedarfs an Rechenleistung sind rekonfigurierbare oder adaptive Architekturen eine vielversprechende Ergänzung zu herkömmlichen Prozessoren. Ein ACS kann Teile ihrer Hardware (HW) an den jeweiligen Algorithmus anpassen. Oft unterstützt eine rekonfigurierbare Logik (RL) in solchen Systemen einen Standard-Prozessor, welcher u.a. Verwaltungsbefehle wie I/O-Operationen ausführt. Rechenintensive Programmteile, die zugleich die Möglichkeit von feingranularer Parallelität bieten, werden auf der RL realisiert. Zum Ausnutzen der Vorteile von Berechnungen auf dem Prozessor und der RL müssen beide Komponenten Zugriff auf Systemressourcen wie Speicher haben.

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 HW-Synthese besitzen.

Zur Linderung dieses Problems können Programme auch in einer höheren Sprache wie C formuliert werden [CHW00][GS98][Ro02]. Ein Compiler übernimmt dann die Partitionierung in HW und Software (*SW*), die Schnittstellengenerierung und die Planung der HW-Ausführung.

Besonderes Augenmerk benötigt während der Übersetzung die automatische Partitionierung in HW und SW, da eine optimale Lösung aufgrund vieler zu beachtender Neben-

bedingungen nicht in vertretbarer Zeit zu berechnen ist. Deshalb werden zur Lösung dieser Probleme meist Heuristiken angewendet. Die Algorithmen sehen nur innere Schleifen für die HW-Ausführung vor, da sie den größten Laufzeitgewinn versprechen.

Diese Arbeit schlägt eine neue Auswahlstrategie von Operationen in Hochsprachenprogrammen vor, welche darüberhinaus auch ineinandergeschachtelte Schleifen bearbeiten kann. Es erweitert damit die bestehenden Pfad-basierten Lösungen [Ma96].

2 Diskussion

Durch die Verfügbarkeit von immer größeren rekonfigurierbaren Schaltkreisen ist auch die Implementierung von ganzen Schleifenverschachtelungen auf einer rekonfigurierbaren Logik möglich. Dadurch müssen nicht immer äußere in SW laufende Schleifen unterbrochen werden, um Daten an die HW zu übertragen und die in HW realisierten inneren Schleifen ausführen zu lassen.

Die dabei verwendeten Algorithmen basieren auf dem Aufstellen aller Pfaden in den inneren Schleifen und der Bewertung jedes einzelnen. Danach wird entschieden, welche der Pfade für die HW verwendet werden. Eines dieser Verfahren ist die Bildung von Hyperblöcken [Ma96]. Hierbei werden alle Pfade einer Region im Kontrollflussgraph (CFG) mit einem Eingang und mehreren Ausgängen aufgestellt. Die Bewertung der einzelnen Pfade und deren Zusammenfassen erzeugen eine Region, die gut optimiert und deren Ausführung gut geplant werden kann. Probleme der Strategie treten auf, wenn die Anzahl von Operationen und Kontrollflussverzweigungen in den bearbeiteten Schleifen sehr groß wird. Allein das Aufstellen der Pfade verlangt dann schon einen hohen Rechenaufwand. Möchte man darüber hinaus noch äußere Schleifen in die Betrachtung einbeziehen, so ist der Aufwand meist nicht mehr zu rechtfertigen. Jeder Pfad einer äußeren Schleife muss mit jedem Pfad einer inneren Schleife verknüpft werden. Das Konzept der Hyperblöcke wurde auch nie für andere als nur die innere Schleife entwickelt.

Die folgende Abschnitte geben eine Überblick über den Compiler COMRADE, führen eine erweiterte pfadbasierte Auswahl von Blöcken zur HW-Ausführung ein und belegen die Verwendbarkeit der vorgestellten Pfadauswahl.

3 Der Compiler COMRADE

Der Compiler COMRADE in **Abb. 1** hat als Ziel, aus C-Quelltexten auf einem ACS lauffähige Programme automatisch zu erzeugen. Die Zielarchitektur ist in [Ko00] beschrieben. Er setzt auf dem Compiler-System SUIF2 der Universität Stanford [La99] auf und erweitert es durch die DFCSSA-Form [Ro02]. Nach High-Level-Optimierungen [Ap98][BGS94] wird der eingelesene C-Quelltext in die DFCSSA-Form konvertiert. Auf dieser Darstellung werden dann alle nachfolgenden Compiler-Durchläufe durchgeführt.

Der Compiler wählt Operationen für eine spätere HW-Implementierung auf einer hohen Abstraktionsebene aus. Während eines Compiler-Schritts werden für alle atomaren Operationen wie Additionen HW-relevante Daten wie Laufzeit und Flächenverbrauch

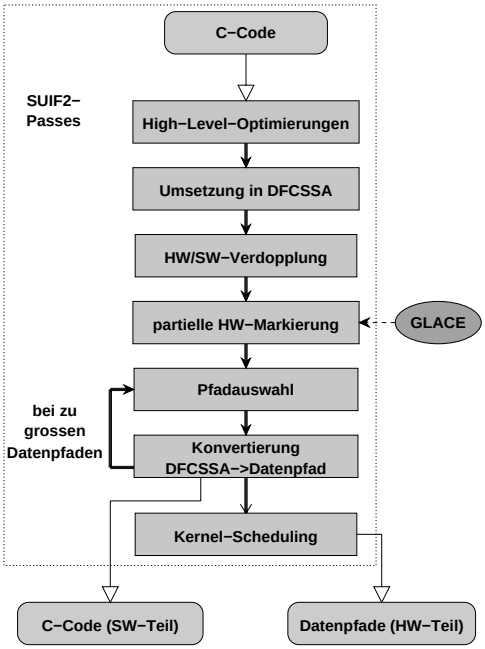

Abbildung 1: Der Compiler-Fluss in COMRADE

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 [La99], auf die unter Benutzung der flexiblen Generatorschnittstelle FLAME zugegriffen wird [KK00]. 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 [BL96][WL94] 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 HW 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 I/O-Anweisungen des Betriebssystems liegen, die nicht in HW ausgeführt werden können. Desweiteren kann es während des Programmlaufs 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. Die Behandlung dieser Ausnahme ist bisher noch nicht implementiert.

Um die Entscheidung später treffen zu können, werden alle Kandidaten für eine HW-Ausführung im folgenden Schritt dupliziert. Dazu werden diese Bereiche im CFG verdoppelt (analog zu GarpCC [CHW00]). Durch eine zusätzlich eingefügte Verzweigung kann

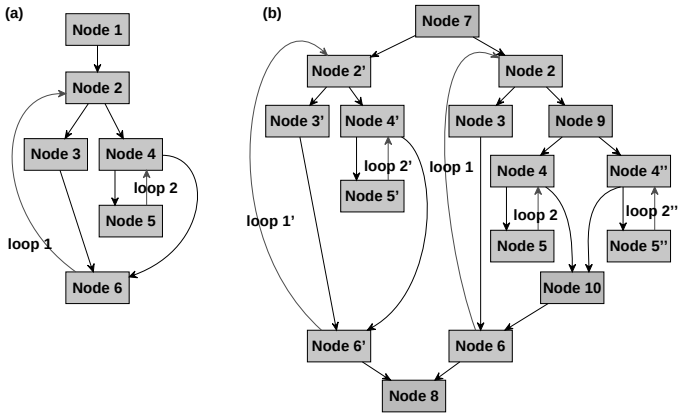

Abbildung 2: Schleifenduplikation

auch zur Laufzeit noch zwischen HW- und SW-Ausführung gewählt werden. Dadurch ist es möglich, auf die ursprüngliche SW-Version zurückzugreifen, wenn sich während der Übersetzung oder Laufzeit zeigt, dass eine HW-Ausführung nachteilig ist.

Ein Beispiel für die Duplikation ist in **Abb. 2** zu sehen. Hier werden zwei ineinander geschachtelte Schleifen dupliziert. 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 Werkzeugen weiterverarbeitet [Ha98].

4 Hardware-Auswahl

Die HW-Auswahl gliedert sich in drei getrennte Schritte. Im **ersten Schritt** werden alle Schleifen wie im vorigen Abschnitt beschrieben dupliziert, für die aufgrund von Profiling-Daten eine Beschleunigung durch die HW-Ausführung zu erwarten ist. Damit wird die Duplikation auf einige ineinandergeschachtelte Schleifen reduziert.

Die Duplikation aller Schleifen in einem Programm würde die Laufzeit des Compilers stark erhöhen. Außerdem kann schon in diesem Schritt für viele Schleifen bestimmt werden, dass sie nicht in HW realisiert werden. Das erste Kriterium sind Profiling-Daten. Nur sehr oft ausgeführte Schleifen sind auch für eine HW-Ausführung lohnend.

Ein weiteres Kriterium sind enthaltene Funktionsaufrufe. Wenn eine Schleife einen Funktionsaufruf enthält, so kann dieser nicht direkt in HW realisiert werden. Ist die Defi-

nition der aufgerufenen Funktion dem Compiler bekannt (im Quelltext), so sollte sie an die aufrufende Stelle eingefügt werden. Ein profile-basiertes Inlining ermöglicht in diesem Zusammenhang, dass nur Funktionsdefinitionen in andere Funktionen eingefügt werden, wenn es sich auf Grund von Laufzeitinformationen wirklich lohnt. Durch das Inlining können größere Regionen für eine HW-Realisierung gefunden werden.

Zwischen diesem und dem folgenden Schritt der HW-Auswahl wird die Markierung aller Operatoren mit HW-relevanten Daten in den duplizierten Regionen durch einen Compilerschritt vorgenommen. Der **zweite Schritt** sucht nach Pfaden, welche sich für die Implementierung eignen. Dieser Schritt ist Thema des nächsten Abschnitts.

Der **dritte Schritt** der HW-Auswahl beurteilt alle HW-Regionen durch den zu erwartenden Laufzeitgewinn gegenüber der SW-Realisierung. Dabei wird davon ausgegangen, dass die SW-Ausführung während der HW-Ausführung suspendiert wird. Der Laufzeitgewinn bestimmt sich hauptsächlich durch die Anzahl der parallel ausführbaren Operatoren in der HW. Außerdem spielt auch der Kommunikationsfaktor zwischen Prozessor und RC eine Rolle. Fällt der Laufzeitgewinn zu klein aus, wird die SW-Realisierung gewählt.

4.1 Pfadauswahl

Bei der Pfadauswahl werden alle für eine eventuelle Ausführung in HW vorgesehen, duplizierten Regionen des CFG bearbeitet. Dabei müssen verschiedene Faktoren berücksichtigt werden, welche die Ausführung in der HW beeinträchtigen.

Eine feste Grenze sind hierbei die auf der RL verfügbaren Ressourcen, welche aus der Modulbibliothek GLACE entnommen werden können. Möchte man eine Region als HW realisieren, so sollte diese Grenze natürlich nicht überschritten werden.

Zum Zeitpunkt der Pfadauswahl sind die Datenpfade noch nicht platziert und verdrahtet. Es steht bisher nur der Ressourcenverbrauch der einzelnen Operationen zur Verfügung. Deshalb wird von einem Zuwachs des Ressourcenverbrauchs von 40% von der vorliegenden Ressourcengröße bis zur fertigen Implementierung ausgegangen. Sollte sich dieser Wert als zu klein nach der HW-Implementierung herausstellen, so wird er bei einem nochmaligen Ausführen der Datenpfadbearbeitung im Compiler korrigiert (Schleife in Abb. 1).

Ein weiterer wichtiger Faktor für die Auswahl von Pfaden in den für HW vorgesehen Regionen ist die Ausführungswahrscheinlichkeit p. Diese ist eine Zahl zwischen 0 und 1 und berechnet sich für einen Block des CFG aus $p=pe/pe_{\max}$ mit der Ausführungsfrequenz des aktuellen Blocks pe und der maximalen Ausführungsfrequenz $pe_{\max}$. Für den Block mit der maximalen Ausführungsfrequenz gilt demnach $p = 1.0$. Die Ausführungswahrscheinlichkeit ist wichtig, wenn der Verbrauch von Ressourcen einer HW-Region die verfügbaren Ressourcen der Ziel-RL übersteigt. In diesem Fall sind nicht alle Operatoren in HW realisierbar und Teile müssen nach bestimmten Kriterien aus der HW ausgeschlossen werden. Es sollten nur die Pfade gewählt werden, welche einen hohen Laufzeitgewinn durch die HW-Realisierung versprechen. Das sind in erster Linie Pfade, auf denen Blöcke mit hohen Ausführungswahrscheinlichkeiten liegen.

Wie schon erwähnt, wird in diesem Schritt noch nicht der erwartete Laufzeitgewinn einer Region durch die HW-Realisierung berücksichtigt. Da die RL aber zur Beschleuni-

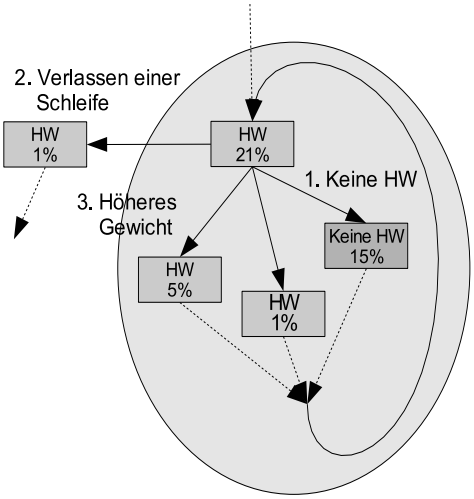

Abbildung 3: Auswahlkriterien für Pfade

gung der Programmausführung verwendet werden soll, wird derzeit in der Heuristik davon ausgegangen, dass eine HW-Realisierung jedes einzelnen Operators schneller ist als eine SW-Realisierung. Für die Erweiterung der Heuristik ist aber die Einbeziehung von Laufzeiten einzelner Operatoren geplant.

Ein Block kann weiterhin nicht in der Ziel-RL implementiert werden, wenn er Operatoren enthält, der nicht in HW ausführbar sind. Hierzu zählen zur Zeit insbesondere I/O-Operationen und Funktionsaufrufe, deren aufgerufene Funktionen nicht im bearbeiteten Quelltext enthalten sind.

4.2 Algorithmus

Jede der für die HW vorgesehen Regionen hat neben mehreren Austrittsblöcken nur einen Eintrittsblock. Beim Bearbeiten einer Region wird diese Block für Block beurteilt. Dabei sollen nach steigender Priorität folgende Forderungen beachtet werden (Abb. 3):

1. Es sollen nur Blöcke verwendet werden, die in HW realisierbar oder ohne Bedeutung für die HW-Ausführung sind. Blöcke, die z.B. nur die Zuweisung einer Konstanten an eine Variable besitzen, sind unbedeutend für die HW-Ausführung.
2. Der aktuelle Pfad soll auf den kürzesten Weg zu einem Ausgang der HW-Region führen. Dadurch wird verhindert, dass beispielsweise in verschachtelten Schleifen alle Pfade einer inneren Schleife der HW zugeordnet werden, aber der Pfad keinen Ausgang der Region enthält. Solche Blöcke wären nicht in HW realisierbar, da auf jeden Fall ein Ergebnis von der HW an die SW zurückgegeben werden muss.
3. Die Pfade sollen eine hohe Ausführungswahrscheinlichkeit haben. Dadurch wird gewährleistet, dass hauptsächlich Teile des Quelltexts in HW übersetzt werden, welche einen hohen Geschwindigkeitsgewinn erzielen. Nur für gelegentliche Unterbrechungen muss in SW gewechselt werden.

Diese drei Forderungen stellen sicher, dass als erstes alle Blöcke in der HW enthalten sind, welche für die Datenübergabe zwischen HW und SW nötig sind. Das sind die Eintritts- und Austrittsblöcke einer Region. Weiterhin wächst der in HW realisierte Teil in den am meisten verwendeten Blöcken einer Region.

Der Algorithmus (Abb. 4) beginnt mit der Suche nach Pfaden durch die HW-Regionen mit dem Eintrittsblock. Dazu werden alle Nachfolger eines Blocks betrachtet (Zeile 12). Es dürfen keine Nachfolger zum aktuellen Pfad hinzugefügt werden, die nicht in HW ausführbar sind (Forderung 1). Alle anderen Nachfolger bekommen ein Gewicht zugeordnet. Dieses ergibt sich für einen Nachfolger S aus $g_S=p_S+b_S$. Der Term p_S ist dabei die Ausführungswahrscheinlichkeit eines Blocks. Durch die Addition des Terms b_S wird erreicht, dass Schleifenausgänge bei der Pfadsuche bevorzugt werden. Er ergibt sich aus:

$$b_S = \begin{cases} 1, \text{ wenn S eine Schleife verlässt} \\ 0 \text{ sonst} \end{cases}$$

Somit ist sichergestellt, dass Forderung 2 eingehalten wird. Der Nachfolger des aktuellen Blocks mit dem höchsten Gewicht wird dem aktuellen Pfad hinzugefügt. Die Suche nach neuen Nachfolgern wird unterbrochen, wenn der Nachfolger im SW-Teil liegt oder durch

```
1   hwBlocks: set of blocks;
2   func searchPaths (region, block, collectBlocks, resources)
3   if (ist block nicht in HW realisierbar) then
4     kein Pfad gefunden
5   else if (ist block in HW realisierbar) then
6     addiere aktuelle Ressourcen zu resources
7   fi
8   if (benötigte Ressourcen übersteigen durch RL angebotene) then
9     kein Pfad gefunden
10  fi
11  füge block zu hwBlocks hinzu
12  bestSucc = ermittele besten Nachfolger von block;
13  füge alle anderen begehbaren Nachfolger zu collectBlocks hinzu
14  if (bester Nachfolger existiert) then
15    searchPaths (region, bestSucc, collectBlocks, resources)
16  else
17    while (noch Blöcke in collectBlocks) do
18      bestNode = bester Block aus collectBlocks & nicht in hwBlocks;
19      saveHwBlocks = hwBlocks;
20      hwBlocks zurücksetzen;
21      newCollectBlocks = collectBlocks;
22      newResources = resources;
23      searchPaths (region, bestNode, newCollectBlocks,
            newResources);
24      if (Suche nach neuem Pfad erfolgreich) then
25        collectBlocks = newBollectNodes;
26        resources = newResources;
27      else
28        hwBlocks = saveHwBlocks;
29      fi
30    done
31  fi
32  Pfadsuche war erfolgreich
```

Abbildung 4: Algorithmus für die Pfadsuche

Hinzunahme des Nachfolgers zu den bisher gewählten Pfaden die auf der Ziel-RL verfügbaren Ressourcen nicht mehr ausreichend sind (Zeile 8).

Während der Suche des aktuellen Pfads wird eine Liste angelegt, welche die Nachfolger enthält, die während der aktuellen Pfadsuche aufgrund der zu niedrigen Gewichte nicht besucht wurden (Zeile 13). Wurden außerdem die Ressourcen auf der Ziel-RL nicht ausgeschöpft, so beginnt eine weitere Pfadsuche an dem Block dieser Liste mit dem höchsten Gewicht (Zeile 18). Die so erzeugten neuen Pfade werden den schon bestehenden hinzugefügt, soweit sie keine Ressourcenüberschreitung darstellen (Zeile 24).

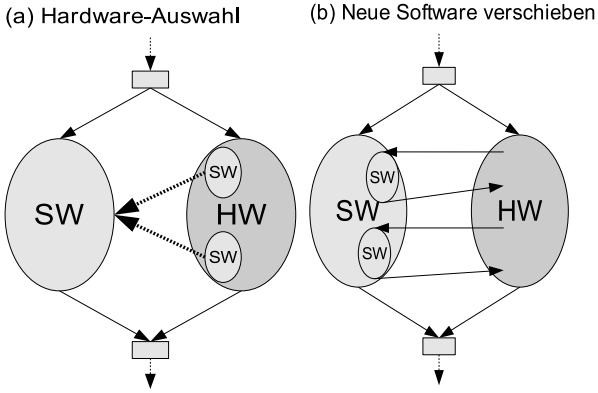

Abbildung 4: Auswahl von HW-Blöcken

Blöcke einer Region, welche nicht für die HW-Realisierung ausgesucht wurden, werden der SW hinzugefügt (Abb. 4). Der Prozessor führt diese Teile als Dienstleister für die HW aus. Der Vorteil liegt darin, dass keine komplizierten Verbindungen zu bestehenden SW-Regionen aufgebaut werden müssen.

Durch die Vorgehensweise entstehen Datenabhängigkeiten zwischen Operationen in der HW und in der neu erzeugten SW. Diese Abhängigkeiten beeinflussen die Laufzeit der HW-Region, da jeweils ein Datenaustausch zwischen HW und SW stattfinden muss. Dieser Faktor beeinflusst wie auch der schon erwähnte Laufzeitgewinn die endgültige Realisierung einer Region in HW oder SW.

5 Resultate

Der Algorithmus wurde an einigen Benchmarks auf seine Verwendbarkeit hin betrachtet. Die Ergebnisse in Abb. 5 zeigen, dass durch den Algorithmus ein Großteil der in ausgewählten Benchmarks findbaren Schleifen in HW realisiert werden kann. In der Tabelle gibt Spalte 1 den Programmnamen an. In den Hauptspalten zwei bis vier wird die Anzahl von ausgewählten Regionen für eine mögliche HW-Realisierung (Kandidaten) gegenüber der Anzahl von verfügbaren Regionen mit jeweils ein, zwei oder mehr enthaltenen Schleifen aufgeführt. Dabei müssen die Schleifen nicht unbedingt vollständig verschachtelt sein. Abb. 6 zeigt die beiden Möglichkeiten von Schleifenverschachtelungen, wenn eine Region beispielsweise 3 Schleifen enthält. Zu beachten ist, dass aufgrund der besonderen

Programm	Regionen mit 1 Schleife	Regionen mit 2 Schleifen	Regionen mit >2 Schleifen	Anzahl der HW-Blöcke in Kand.	Verwendete Ressourcen der Regionen in CLBs		
	HW-Kand /Total	HW-Kand /Total	HW-Kand /Total	Durchschn.	Durch-schnitt	Min	Max
versatility	9/15	2/4	0/1	100%	99,2	32	288
adpcm	1/2	0/0	0/0	100%	1040	1040	1040
capacity	17/17	2/2	2/3	72,8%	50,8	16	224
g721	3/4	0/1	0/0	96,1%	128	32	320
pegwit	43/71	4/10	0/1	95,6%	84,9	16	608
versatility (inlined)	12/13	3/3	5/5	100%	117,1	32	384

Abbildung 5: Ergebnisse der Pfadauswahl

Art von Schleifenduplikation in COMRADE einige der Schleifen aus Regionen mit nur einer Schleife auch in Regionen mit mehreren Schleifen enthalten sind.

Die fünfte Spalte gibt an, wieviel Blöcke der für eine HW-Realisierung vorgesehenen Regionen für die HW-Ausführung verwendet werden. Werden beispielsweise 20 von 40 Blöcken des CFG einer für HW vorgesehen Region für die HW-Realisierung gewählt, so würden also 50% der Blöcke verwendet.

In der letzten Hauptpalte ist der durchschnittliche, minimale und maximale Verbrauch an Ressourcen (CLBs) der Kandidaten für das in unserer Forschungsgruppe verwendete ACS [Ko00] mit Virtex 1000 zu sehen. Beachtet werden muss die Vergrößerung der Ergebnisse um 40-60%, da noch keine Verdrahtungskapazitäten abgeschätzt wurden.

Die ersten fünf Programme wurden ohne Inlining bearbeitet. Erkennbar ist der geringe Prozentsatz verschachtelter Schleifen. Viele der Schleifen werden trotzdem schon ohne Inlining von Funktionen für die HW-Realisierung vorgesehen.

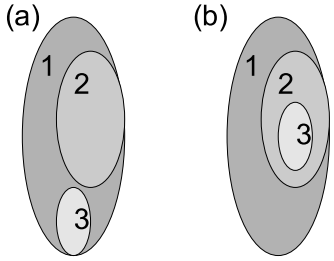

Abbildung 6: Verschachtelungen von 3 Schleifen

Durch Inlining können die für HW verwendbaren Regionen noch vergrößert werden. Das verhindert ein ständiges Rekonfigurieren der RL während der Programmausführung und nutzt auch Beschleunigungspotential von äußeren Schleifen. Das ist in der letzten Zeile der Tabelle gezeigt. Hier wurden alle Funktionen des versatility-Benchmarks in die main-Funktion eingefügt. Hier können bis zu neun Schleifen in einer Region auftreten. Trotz gleicher Anzahl von Regionen ist die Durchschnittsgröße dieser gestiegen.

Eine weitere Feststellung ist, dass es sich für eine große Menge der Schleifen in den Benchmarks aufgrund ihrer Größe lohnt, mehrere zu einer Konfiguration zusammenzu-

fassen. Dadurch kann die Häufigkeit der Rekonfigurationen reduziert werden. Für COMRADE wird zur Zeit an solch einem Rekonfigurations-Scheduling gearbeitet.

6 Zusammenfassung

Diese Arbeit stellte eine neue methodische Erweiterung zur Auswahl von HW-Regionen für die Programmerzeugung für Adaptve Computersysteme aus C-Quelltext vor. Gezeigt wurde, dass mehrere ineinandergeschachtelte Schleifen bearbeitet werden können, um damit Beschleunigungspotential durch die HW-Realisierung von äußeren Schleifen zu nutzen.

Die Benchmarks zeigten, dass ohne Bearbeitung der Programmstruktur in den C-Quelltexten schon die Vorteile der Pfadauswahl in ineinandergeschachtelten Schleifen benutzt werden können. Durch das Anwenden von Inlining sind aber noch bessere Ergebnisse erreichbar, was den Grund vor allem in der Größe der verfügbaren RL hat.

Literatur

[Ap98] Appel, A., Modern Compiler Implementation in C, Cambridge University Press, 1998

[BGS94] Bacon, D. F., Graham, S. L., Sharp O. J., Compiler Transformations for High-Performance Computing, ACM Computing Surveys 26(4), 1994

[BL96] Ball, T., Larus, J., Branch Prediction for free, Proc. of the Conf. on Prog. Language Design and Implementation, 1996

[CHW00] Callahan, T., Hauser, R., Wawrzynek, J., The GARP Architecture and C Compiler, IEEE Computer 33(4), 62-69, April 2000

[GS98] 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

[Ha98] 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

[KK00] Kasprzyk, N., Koch, A., Advances in Compiler Construction for Adaptive Computers, Conference on Parallel and Distributed Processing Techniques and Applications, 2000

[KK02] Koch, A., Kasprzyk, N., Module Generator-based Compilation for Adaptive Computing Systems, IEEE Intl. Symp. on FCCMs, Napa Valley (CA, USA), 2002

[Ko00] Koch, A., A Comprehensive Prototyping Platform for Hardware-Software Codesign, Workshop on Rapid Systems Prototyping, Paris, 2000

[La99] 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, 1999

[Li00] Li, Y.B., Harr, R., et al. Hardware-Software Co-Design of Embedded Reconfigurable Architectures, Proc. Design Automation Conference, 2000

[Ma96] Mahlke, S, Exploiting instruction level parallelism in the presence of conditional branches, PhD thesis, University of Illinois at Urbana-Champaign, 1996

[NK01] Neumann, T., Koch, A Generic Library for Adaptive Computing Environments, Workshop on Field-Programmable Logic and Applications, Belfast, 2001

[Ro02] Rock, M., Implementierung einer SSA-Form zur effizienten Erzeugung von Datenflussgraphen für Adaptive Rechner, Diplomarbeit, TU Braunschweig, 2002

[WL94] Wu, Y., Larus, J. R., Static Branch Frequency and Program Profile Analysis, In 27th IEEE/ACM Symposium on Microarchitecture (MICRO-27), 1994