Adaptive Rechensysteme – Architekturen und Werkzeuge

Dr. ing. Andreas Koch, TU Braunschweig, Abt. Entwurf integrierter Schaltungen, Braunschweig

Kurzfassung

Wir diskutieren verschiedene mögliche Hardware-Architekturen für adaptive Rechensysteme, die rekonfigurierbare Elemente als Recheneinheiten verwenden. Ihrer hohen Leistungsfähigkeit und Flexibilität steht eine schwierigere Anwendungsentwicklung gegenüber, deren Vereinfachung wesentlicher Teil unserer Arbeit ist. Die Erfahrungen aus der Entwicklung von Compilern und Floorplanning-Prototypen werden als Grundlage für die Konzeption eines neuen Entwurfsflusses beschrieben. Für dessen Realisierung wichtige Komponenten wie eine universelle Schnittstelle zu parametrisierten Modulgeneratoren und der Entwurf einer allgemeinen Basisbibliothek für die EDA-Forschung werden als erste Ergebnisse auf dem Weg zu seiner Implementierung vorgestellt.

1 Einleitung

Im Gegensatz zu einem konventionellen Rechner, bei dem variable Software auf fester Hardware ausgeführt wird, wird bei einem adaptiven Rechner auch die Hardware ganz oder teilweise an die aktuelle Problemstellung angepasst. Auf diese Weise lassen sich bei bestimmten Anwendungen nennenswerte Verkürzungen der Rechenzeit erreichen (in Extremfällen bis auf wenige Tausendstel [1]).

Konventionelle Prozessoren sind wegen ihrer starren internen Architektur als Kern eines adaptiven Rechners ungeeignet. Stattdessen werden die Recheneinheiten aus rekonfigurierbaren Field-Programmable Gate Arrays (FPGAs) aufgebaut. Diese wurden ursprünglich entwickelt, um für die Fertigung von Kleinserien oder Prototypen die Funktion diverser weniger hoch integrierter Bauelemente (z.B. diskrete PAL/PLA/GAL/74xx) zu einem einzelnen Chip zusammenzufassen, ohne die immensen Investitionen für die Fertigung eines anwendungsspezifischen ICs (ASICs) zu erfordern.

Heute ist aber das Fassungsvermögen von FPGAs in den Bereich der Millionen von Gattern gewachsen [2]. Damit lassen sich neben den vormals implementierten einfachen Steuerungsschaltungen (“glue logic”) nun auch prozessorartige Strukturen realisieren. Zum Vergleich: Der SPARC “Sunrise” RISC Prozessor benötigte lediglich 40.000 Gatter [3].

Den Leistungssteigerungen und der Flexibilität, die durch rekonfigurierbare Hardware erreicht werden können, steht aber eine deutlich kompliziertere Programmierung sowie die Beherrschung der Komplexität von variabler Software kombiniert mit variabler Hardware gegenüber.

Hier wird ein Überblick über aktuelle Arbeiten gegeben, die in Zusammenarbeit mit der UC Berkeley [4] sowie Synopsys Inc. durchgeführt werden. Die Forschung umfaßt sowohl Hardware-Architekturen als auch Software-Werkzeuge und hat die Lösung der eingangs angesprochenen Probleme als Ziel.

2 Architekturen

Nach den Erfahrungen in [5] eignen sich rekonfigurierbare Prozessoren am Besten für Anwendungen, in denen entweder sehr viele Berechnungen auf einer kleinen Datenmenge ausgeführt werden (z.B. Verschlüsselung) oder solche, die auf einer großen Datenmenge vergleichsweise einfache Berechnungen ausführen (z.B. Signal- und Bildverarbeitung).

Die kontrolldominierten Teile außerhalb der inneren Schleifen der Algorithmen werden in der Regel effizienter auf konventionellen Prozessoren ausgeführt. Aus diesem Grund sollte auch in einem adaptiven Rechensystem eine konventionelle CPU präsent sein, und sei es auch nur als eine der möglichen (Teil-)Konfigurationen der adaptiven Komponente. In diesem Text werden solche Architekturen als “hybrid” bezeichnet.

Abbildung 1 zeigt verschiedene Weisen, auf die ei-

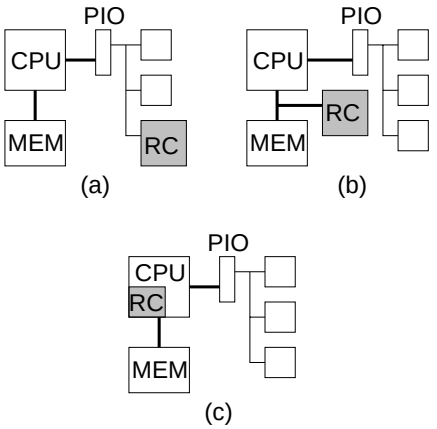

Abbildung 1: Systemintegration der rekonfigurierbaren Komponente

ne rekonfigurierbare Komponente (RC) in ein adaptives Rechensystem integriert werden kann. In Abbildung 1.a wird die RC an den Peripheriebus (PIO, konkret z.B. PCI, SBus, etc) des Rechners angeschlossen. Dieser Ansatz wird häufig in älteren Systemen [6] [7] [8] oder in solchen, in denen die RC bevorzugt für Logikemulation eingesetzt wird [9], realisiert. Auch heute noch wird wegen der leichten Realisierbarkeit (klar definierte Schnittstelle, breite Anwenderbasis) diese Architektur verwendet. Wegen der langen Kommunikationslatenzzeiten und der beschränkten Datentransferraten ist sie aber nicht optimal, da die Anbindung an den konventionellen Wirtsrechner zu lose ist.

Eine engere Anbindung ist in Abbildung 1.b skizziert. Hier ist die RC direkt an den Prozessor-Speicherbus angeschlossen. Diese Klasse von Architektur wurde beispielsweise in [10] [11] verwendet. Neben einem verbesserten Datendurchsatz und verkürzten Latenzzeiten vereinfacht sich auch die Implementierung der Schnittstelle in der RC, da hier ein einfaches Speicherzugriffsprotokoll implementiert werden kann. Bei leistungsfähigeren Systemen muß allerdings die Kohärenz von eventuell vorhandenen Caches sichergestellt werden. Im einfachsten Fall ist dies durch Definition von nicht-cachebaren Speicherregionen für die Kommunikation realisierbar.

Die ideale Anbindung in Bezug auf kurze Latenz und hohe Bandbreite ist in Abbildung 1.c dargestellt. Hierbei ist die RC direkt auf dem CPU-Die untergebracht und kann so direkt über die on-chip Busse kommunizieren. Einige Umsetzungen dieses Ansatzes [12] [13] [14] sind aber durch das Fehlen eines RC-eigenen Speicherzugangs behindert, da nur entweder die CPU oder die RC auf den gemeinsamen Speicher zugreifen können. Abhilfe schaffen hier entsprechende Speicherprotokolle, z.B. Zuteilung nach dem Round-Robin Verfahren [15], mittels eines paketorientierten Protokolls [16], oder die Integration von ausreichend RC-eigenem Speicher auf den “Die” [17]. Leider ist derzeit keine dieser Implementierungen in einer verwendbaren Form verfügbar.

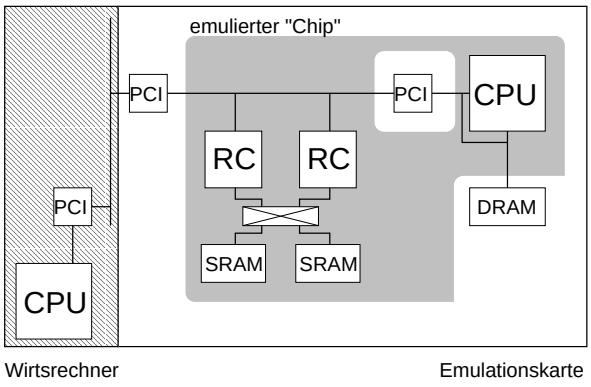

Abbildung 2: Reale und emulierte (schattiert) adaptive Rechnerarchitektur

Da aber die Simulation eines kompletten adaptiven Rechensystems bei weitem zu zeitaufwendig ist, verwenden wir für unsere bisherige Arbeit daher eine Emulation von Architektur (c) in Form einer TSI-Telsys “ACE2” Erweiterungskarte (Abbildung 2, [18]), die mittels des PCI-Busses für Ein-/Ausgabe und Steuerung an einen Wirtsrechner angebunden ist. Die schattierten Teile spiegeln die Komponenten (CPU, RC, Speicher) wieder, die mittelfristig auf einem einzelnen Chip verfügbar sein werden. Im Ganzen bildet die Karte so einen kompletten adaptiven Rechner in Hybridarchitektur nach.

3 Entwurfsfluß

Eines der großen Hemmnisse beim praktischen Einsatz vom adaptiven Prozessoren ist die Komplexität ihrer Programmierung. Während konventionelle Rechner in relativ abstrakten Hochsprachen programmiert werden können, werden noch heute die meisten An-

wendungen für adaptive Rechner rein mit den Methoden des Hardware-Entwurfs erstellt. Es kommen dabei zwar teilweise auch Syntheseverfahren zum Einsatz, die Schaltungen aus einer Hardware-Beschreibungssprache wie zum Beispiel Verilog generieren, die Entwicklungskomplexität ist aber immer noch wesentlich höher als bei der Programmierung eines konventionellen Rechners. Sollen adaptive Rechner eine weitere Verbreitung finden, müssen sie für Anwender, beispielsweise aus dem Bereich der Signalverarbeitung, die in der Regel neue Verfahren in MATLAB oder C implementieren, leichter zugänglich gemacht werden.

3.1 Compiler

Diese Problematik soll durch neu entwickelte Compiler entschärft werden. Als Eingabe muß ein Programm in einer geeigneten Hochsprache akzeptiert werden, das dann automatisch auf in Hardware auslagerbare Teile (in der Regel Schleifen) hin untersucht wird. Die Entscheidung, Operationen auszulagern, wird im Einzelfall auch beeinflußt von der Abwägung des Platzbedarfs ihrer Implementierung auf der RC, dem zusätzlichen Zeitaufwand für Kommunikation (Übertragen der Startparameter und Ergebnisse) sowie dem Rekonfigurationsverhalten während des gesamten Programmlaufes. Es ist diese letztgenannte Eigenschaft, die Compiler für adaptive Rechner von High-Level-Synthese Werkzeugen [19] für ASICs unterscheidet. Da FPGAs in Bezug auf Schaltungsdichte wesentlich ineffizienter sind als ASICs, muß für eine effiziente Nutzung die Siliziumfläche des FPGAs mehrfach wiederverwendet werden. Die Bestimmung der zeitliche Abfolge des Ein- und Auslagerns von Programmfunktionen in die RC ist gerade auch wegen der damit verbundenen zusätzlichen Verzögerung von essentieller Bedeutung.

Erste Prototypen für solche Compiler existieren bereits [20] [21]. In beiden Fällen wird als Ausgangssprache C verwendet, die konkreten Vorgehensweise sind jedoch unterschiedlich. Während sich [21] an expliziten Hinweisen des Benutzers auf Hardware-Eigenschaften orientiert (Zuordnung einer Variable an Speicher oder RC-Register, Bitbreite von Operanden, Auslagerung von Prozeduren oder einzelnen Ausdrücken, parallele Ausführung), versucht [20] diese Parameter vollautomatisch zu extrahieren. Auf diese Art kann ohne Benutzerintervention beliebiger C Code übersetzt werden. Sollte sich dabei keine geeignete Hardware-Abbildung realisieren lassen, so wird das gesamte Programm auf der konventionellen CPU des Hybridrechners ausgeführt (ohne Geschwindigkeitsgewinn, aber immer noch lauffähig). Programme können so inkrementell auf einen immer höheren Hardware-Anteil hin modifiziert werden. Um die oben geforderte leichtere Benutzbarkeit adaptiver Rechensysteme zu fördern, verfolgen wir daher die Weiterentwicklung von [20].

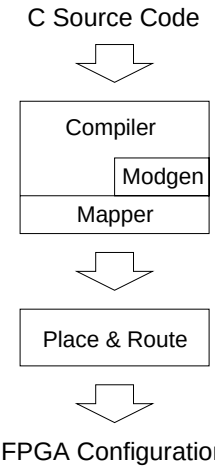

Abbildung 3: Architektur des aktuellen Compiler-Prototyps [20]

Dieser Compiler (Abbildung 3) beinhaltet einfache Modulgeneratoren für primitive Funktionen (Logik, Addierer, ...) sowie einen “Mapper”, der die erstellten Datenpfade auf das Ziel-FPGA abbildet. Komplexe Funktionen oder ein effizientes Floorplanning (siehe nächster Abschnitt) werden jedoch noch nicht unterstützt.

3.2 FPGA-spezifische Entwurfswerkzeuge

Nach der Generierung geeigneter Hardware-Architekturen durch den Compiler gilt es, diese möglichst effizient auf die Strukuren des als RC genutzten FPGAs abzubilden. Die bisher dafür verwendeten Werkzeuge, die auf die Verarbeitung kleinerer Schaltungen ausgelegt sind, haben mit dem Wachstum des FPGA-Fassungsvermögens nicht Schritt halten können und bearbeiten größere Probleme nur sehr ineffizient. Die reguläre Struktur von Datenpfaden geht bei Verwendung dieser einfachen Plazierungs- und Verdrahtungsprogramme verloren, auch konzep-

tionell zusammengehörende Schaltungselemente (z.B. Bit-Slices) werden willkürlich, oft mittels Simulated Annealing, auf dem Chip plaziert (Abbildung 4.a). Dieses Vorgehen führt zu einer Verlangsamung der Schaltung.

Als Abhilfe müssen neue Ansätze geschaffen werden, die auch größere prozessorartige Strukturen geschickter implementieren. Entscheidend für das Erzielen verbesserter Ergebnisse sind der Einsatz von Modulgeneratoren [22] [23] und ein Floorplanning der Datenpfad-Komponente des adaptiven Prozessors [24].

Die Modulbibliothek umfasst eine breite Palette von Schaltungselementen, die durch Wahl entsprechender Parameter (z.B. Bitbreiten, Datentypen, ...) automatisch an die Erfordernisse der abzubildenden Architektur angepasst wird. Dabei stellen die Generatoren neben der logischen Funktion (Netzliste) des Moduls in der Regel auch ein optimal auf dem RC-FPGA plaziertes Layout bereit. Ein solcher Entwurfsfluß ist in Abbildung 4.b skizziert. Einzelne Module werden jetzt effizient erzeugt, eine Optimierung über die Modulgrenzen hinweg findet aber noch nicht statt.

Diese Art der Verfeinerung ist Aufgabe des Floorplanners, der für den Aufbau des gesamten Datenpfadteils des adaptiven Prozessors verantwortlich ist (Abbildung 4.c). Er wählt gegebenenfalls aus der Vielzahl von möglichen Implementierungsalternativen, die von den Modulgeneratoren angeboten werden, eine aufeinander abgestimmte Teilmenge aus und fügt diese unter Ausnutzung von Regularität [25] und Hierarchie zusammen. In einem bereits entwickelten Prototypen [24] konnten so Geschwindigkeitssteigerungen um bis zu 30% und Reduktionen der Compilezeit auf bis zu 16% des konventionellen Ansatzes erreicht werden.

3.3 Neue Architektur

Nach den vielversprechenden Experimenten mit den Prototypen werden die so gewonnenen Erfahrungen in die Realisierung eines neuen Entwurfsflusses eingebracht. Dabei werden Compiler, Floorplanning und Modulgenerierung in einer neuartigen Weise kombiniert.

Anstatt daß Compiler und Floorplanner auf unterschiedliche Modulgeneratoren zurückgreifen, steht nun beiden Werkzeugen die gleiche vielseitige und leicht erweiterbare Modulbibliothek zur Verfügung. Auf die Realisierung von Sonderfällen durch Generierung in-

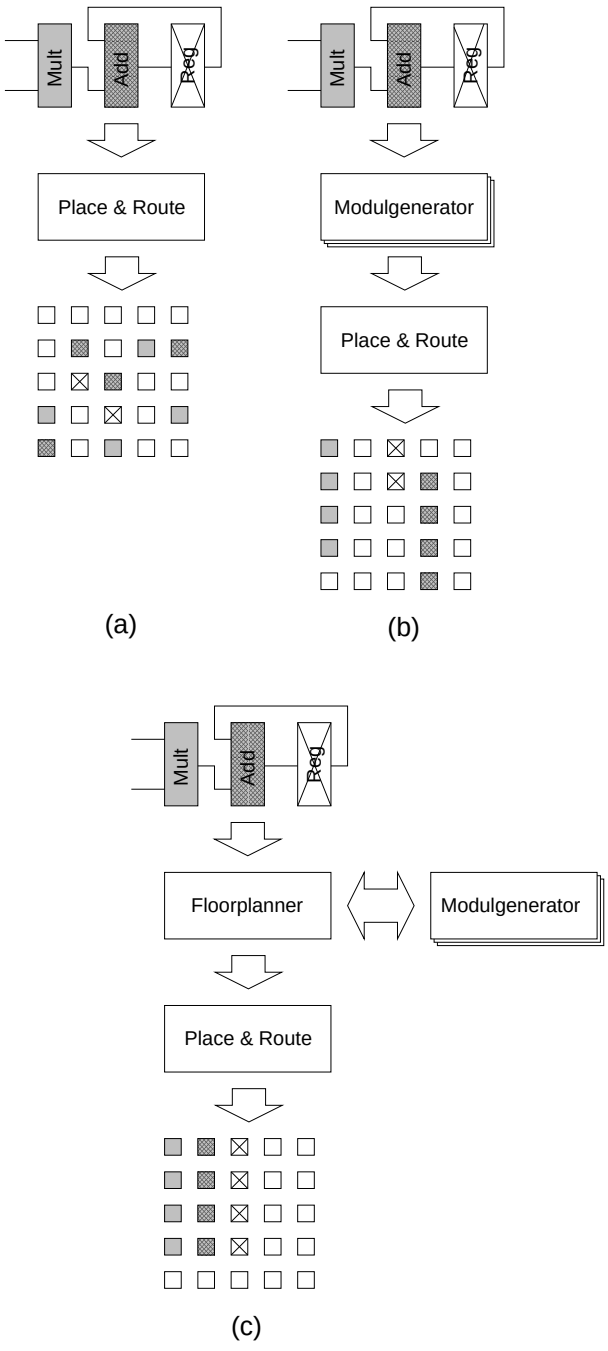

Abbildung 4: Entwicklung FPGA-spezifischer Entwurfsflüsse

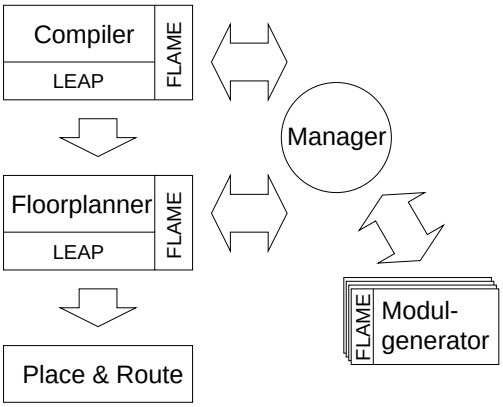

Abbildung 5: Architektur des zukünftigen Entwurfsflusses

Sicht	Inhalt
behavior	Funktion, logische Schnittstelle
synthesis	Zeitverhalten, Fläche, Leistung, Ansteuerung und physik. Schnittstelle
structure	Regularität, Hierarchie
topology	Layout-Formen, Verdrahtungsdichte
netlist	FPGA-unabhängige Netzliste
mapped	FPGA-abhängige Netzliste
placed	FPGA-abhängige Plazierung
routed	FPGA-abhängige Verdrahtung
simulation	Modell für Verhaltenssimulation

Tabelle 1: FLAME Sichten

nerhalb des Compilers kann verzichtet werden. Ebenso entfällt das rudimentäre Compiler-Mapping. Stattdessen wird ein vollständiges Floorplanning wie in Abbildung 4.c und [24] beschrieben durchgeführt.

4 FLAME

Als standardisierte Schnittstelle in die Modulbibliothek wurde FLAME, das Flexible API for Module-based Environments definiert. Die Klienten (z.B. Compiler und Floorplanner) führen über den FLAME Manager einen interaktiven, aber vollautomatischen Dialog mit der Bibliothek. Dabei werden durch schrittweise Verfeinerung von Anfragen die Eigenschaften der gewünschten Zellen eingegrenzt. Es werden jeweils nur solche Daten berechnet und ausgetauscht, die für den aktuellen Bearbeitungsschritt notwendig sind. Beispielsweise sind in der Phase der Datenflußgraph-Überdeckung durch den Compiler lediglich Informationen über die Funktionen der einzelnen generierbaren Zellen erforderlich, nicht aber beispielsweise fertig verdrahtete Layouts. Ein solches Vorgehen trägt zu verkürzten Compile-Zeiten bei.

Neben dieser aktiven Schnittstelle beinhaltet FLAME ein alle Phasen des Entwurfsflusses abdeckendes Datenmodell, das die zu einer Phase gehörenden Daten in sogenannten Sichten gruppiert (Tabelle 1).

Da die Modulgeneratoren bereits während der Compile-Phase des Entwurfsflusses zur Verfügung stehen, kann schon bei der Bewertung verschiedener Zielarchitekturen auf beliebig präzise Charakteristika der in Frage kommenden Zellen zugegriffen werden. So werden die, auch beim Compiler-Prototypen noch recht ungenauen, Abschätzungen, zum Beispiel von Zeit- und Flächenbedarf, durch die realen Angaben ersetzt.

Das gesamte Datenmodell wurde weitgehend FPGA-unabhängig gehalten und erlaubt so die Implementierung portabler Klienten, die leicht auf unterschiedliche Zieltechnologien optimieren können. Beispielsweise werden sonst FPGA-spezifische Eigenschaften wie die Eigenschaften von Speicher- oder Tri-State-Elementen und der Flächenbedarf einer Zelle einheitlich abstrahiert.

Die FLAME-Definition selber setzt keine speziellen Anforderungen an die für die Implementierung von Klienten und Generatoren verwendeten Programmiersprachen und Betriebssysteme voraus. FLAME kann in beliebigen Umgebungen zum Einsatz kommen: Von monolithisch integrierten Systemen (bei denen alle Klienten und die Bibliothek ein einzelnes ausführbare Programm bilden) bis hin zu über das Internet verteilten Systemen (bei denen Klienten und Bibliothek über das Netz kommunizieren) sind alle Kombinationen möglich.

Die Verwendung der FLAME-Schnittstelle sorgt für die leichte Erweiterbarkeit des Entwurfsflusses sowohl auf Klienten- als auch auf Generatorseite: Die Integration eines neuen Generators in das System braucht nicht mehr zu erfordern, als die Generatorprogramme in einem entsprechenden Verzeichnis abzulegen. Von diesem Moment an können die Generatorfähigkeiten automatisch von allen bestehenden Klienten genutzt werden. Umgekehrt kann auch jederzeit ein weiterer Klient in den Entwurfsfluß eingebunden werden, alle bis-

herigen Modulgeneratoren können sofort angesprochen werden.

5 LEAP

Sowohl bei der Implementierung der verschiedenen Prototypen als auch bei der Entwicklung von EDA-Werkzeugen im allgemeinen hat sich gezeigt, daß viele Gemeinsamkeiten zwischen den einzelnen Programmen bestehen. So finden sich zum Beispiel fast immer graph-basierte Datenstrukturen mit den entsprechenden Operationen. Auch verschiedene Optimierungsverfahren sind häufig wiederkehrende Unterprogramme.

Für die Qualitätsbewertung von EDA-Algorithmen ist in der Regel eine konkrete Implementierung mit anschließender praktischer Erprobung durch Benchmarking erforderlich. Diese Programmierarbeiten nehmen häufig einen nennenswerten Teil der Forschungsarbeit in Anspruch. Das ist insbesondere dann der Fall, wenn der Algorithmus in einen bestehenden Entwurfsfluß eingebunden werden muß und daher bestimmte Ein- und Ausgabedatenformate gefordert sind. Gerade bei der Anbindung an industrielle Werkzeuge mit ihren komplexen Formaten (z.B. Verilog, VHDL, EDIF) wird so mehr Zeit auf die Erstellung der Testumgebung als auf die Implementierung des eigentlichen Algorithmus verwendet.

Um diesem Zustand abzuhelfen wird für die Implementierung unseres neuen Entwurfsflusses (Abbildung 5) konsequent von vorneherein eine leicht wiederverwendbare Basisbibliothek entworfen. Die Library for EDA-Algorithm Prototyping (LEAP) soll viele der gängigen Verfahren bereits enthalten und so die Erprobung neuer Algorithmen wesentlich erleichtern. Für den Entwurf von LEAP werden moderne objektorientierte Entwurfsmethoden (OOD [27], UML [28], “design patterns” [29]) unterstützt von den ensprechenden Werkzeugen für Computer-Aided Software Engineering (CASE) eingesetzt. Auch wird wie schon für FLAME auf eine weitgehend Programmiersprachen-unabhängige Konzeption geachtet. Der aktuelle LEAP-Entwurf umfaßt folgende Komponenten:

Hierarchische Graphen mit den üblichen Durchlaufoperationen sowie Funktionen zum Auflösen der Hierarchie für die Ausführung konventioneller (flacher) Graphalgorithmen. Standardverfahren wie längster Pfad, Durchlauf der Breite und Tiefe nach, ... werden bereitgestellt.

Optimierungsverfahren , sowohl heuristisch (Simulated Annealing [30], Tabu-Search [31]) als auch optimal (Constraint-based [32], Linear/Integer Linear Programming [33]). LEAP stellt Basisfunktionen bereit, die vom Benutzer lediglich um problemspezifische Operationen erweitert werden müssen. Für die Heuristiken können dies beispielsweise die Lösungsbewertung oder das Ausführen eines “Zuges” sein. Im I/LP Bereich werden Probleme allgemein aufgebaut und dann automatisch in das Format des verwendeten “Solver”-Programmes übersetzt. Umgekehrt werden auch die vom Solver gelieferten Ergebnisse wieder in die allgemeine LEAP-interne Datenstruktur übersetzt. Auf diese Weise können Algorithmen unabhängig von den eigentlichen Lösungsverfahren erprobt werden.

Standard-Ein-Ausgabeformate für Entwurfsdaten. LEAP wird dafür zunächst VHDL nutzen. Da die Übersetzung von VHDL in einen der benutzerdefinierten Graphen stark anwendungsabhängig ist, wird dem Entwickler eine vollständig elaborierte Zwischenschicht in IIR [34] zur Verfügung gestellt. Diese kann dann nach Bedarf ausgewertet werden.

Neben der einfachen Verwendung und Erweiterbarkeit ist LEAP auch auf eine effiziente Ausführung der Algorithmen ausgelegt. Es wird zum Beispiel ein einheitlicher “Memoization” (caching) Mechanismus bereitgestellt, mit dem einmal berechnete Daten sofort wiederverwendet werden können falls sich die Berechnungsgrundlage nicht geändert hat.

Um den LEAP-eigenen Implementierungsaufwand zu begrenzen, werden wo immer möglich bereits existierende Bibliotheken eingebunden. So wird beispielsweise für die Simulated Annealing-Heuristik “EBSA” [30] verwendet und das VHDL-Frontend wird mittels “SAVANT” [35] realisiert.

6 Ergebnisse

Aus den durch die verschiedenen Prototypen gewonnenen Erfahrungen wurde die Struktur eines verbesserten Entwurfsflusses für adaptive Rechensysteme definiert.

Zu seiner praktischen Umsetzung wurde als erstes konkretes Ergebnis mit FLAME eine mächtige universelle Schnittstelle zum Zugriff auf parametrisierbare Modulbibliotheken geschaffen. Für FLAME steht eine umfangreiche Dokumentation [36] sowie eine Referenzimplementierung in Java [37] zur Verfügung. Durch LEAP wird ein weiterer Baustein für die schnellere Erstellung von EDA-Werkzeugen und die einfachere Erprobung von neuen Algorithmen bereitgestellt werden.

7 Zusammenfassung

Nach Untersuchung verschiedener Hardware-Architekturen für adaptive Rechensysteme wird eine Lösung favorisiert, die im Idealfall eine rekonfigurierbare Komponente mit eigenem Speicher und eine konventionelle CPU auf einem Chip vereint. Da ein solcher Baustein derzeit aber noch nicht erhältlich ist, wird eine Emulation mit Hilfe der "ACE2" Erweiterungskarte vorgeschlagen, die eine vergleichbare Funktionalität bereitstellt.

Die Anwenderakzeptanz von adaptiven Rechnern hängt wesentlich von der Handhabbarkeit und Leistungsfähigkeit ihres Entwurfsflusses ab. Wir haben durch Untersuchung der verfügbaren Compiler sowie durch eigene Arbeiten auf dem Gebiet der Datenpfadsynthese einen Entwurfsfluß konzipiert, der die Schwächen der bisherigen Ansätze korrigieren sollte.

Als erstes Ergebnis der konkreten Realisierung dieses neuen Flusses wurde mit FLAME ein entscheidendes Bindeglied zwischen dem Compiler/Floorplanner sowie den Modulgeneratoren geschaffen. Der nächste Meilenstein wird die Implementierung des LEAP-Entwurfes einer allgemeinen Basisbibliothek für alle Arten von EDA-orientierter Forschung sein.

Literatur

[1] Ratha, N., Jain, A., Rover, T., "Fingerprint Matching on Splash 2", in *Splash 2 – FPGAs in Custom Computing Machines*, eds. Buell, D., Arnold, M., Kleinfelder, W., IEEE Press, 1996

[2] Xilinx Inc., "Virtex 2.5V Field Programmable Gate Arrays", *Advance Product Specification*, 01/1999

[3] SUN Inc., "SPARC Processors: Milestones, Fun Facts, Accolades", `http://www.sun.com/microelectronics/sparc/SPARCfacts.html`, 1999

[4] The BRASS Group, "Berkeley Reconfigurable Architectures, Systems, and Software", `http://www.cs.berkeley.edu/Research/Projects/brass/`, Berkeley, 1999

[5] Buell, D., Arnold, J., "The Promise and the Problems", in *Splash 2 – FPGAs in Custom Computing Machines*, eds. Buell, D., Arnold, M., Kleinfelder, W., IEEE Press, 1996

[6] Koch, A., Golze, U., "Practical Experiences with the SPARXIL Co-Processor", *Proc. Asilomar Conference on Signals, Systems, and Computers*, 11/1997

[7] Arnold, J. et al, "The Splash 2 Processor and Applications", *Proc. ICCD 1993*, CS Press, 1993

[8] Vuillemin, J. et al, "Programmable Active Memories: Reconfigurable Systems Come of Age", *IEEE Trans. VLSI Systems*, 3/1996

[9] The Dini Group, "DN250k10 PCI-hosted ASIC Prototyping Engine", `http://www.dinigroup.com`, 1999

[10] Bright Star Engineering Inc.,"ipEngine-1 Hardware Reference Manual", `http://www.brightstareng.com/pub/ie1hwman.pdf`, 1999

[11] Heeb, B., Pfister, C., "Chameleon: A Workstation of a Different Colour", *Proc. 2nd FPL* , 1992

[12] Athanas, P., Silverman, H., "Processor Reconfiguration through Instruction Set Metamorphosis" *Computer*, Vol. 26. No 3, 1993

[13] Wittig, R., "OneChip: An FPGA Processor with Reconfigurable Logic", *M.A.Sc. Thesis*, Univ. Toronto, 1995

[14] Kastrup, B., Bink. A., Hoogerbrugge, J., "ConCISe: A Compiler-Driven CPLD-Based Instruction Set Accelerator", *Proc. FCCM*, Napa, 1999

[15] Triscend Inc., "Triscend E5 Configurable Processor Family", `http://www.triscend.com/products/sse5cpsu.pdf`, 11/1998

[16] Garverick, T., Rupp, C., “National Semiconductor NAPA1000”, http://www.nsc.com/appinfo/milaero/napa1000/index.html, 1999

[17] DeHon, A. et al, “HSRA: High-Speed, Hierarchical Synchronous Reconfigurable Array”, *Proc. 7th Symp. on FPGAs*, Monterey, 1999

[18] TSI-Telsys Inc., “ACE2card Hardware Manual”, 1998

[19] Ernst, R. et al, “The COSYMA environment for hardware/software cosynthesis of small embedded systems”, *Microprocessors and Microsystems*, 20(3):159-166, 1996

[20] Callahan, T., Wawrzynek, J, “Instruction Level Parallelism for Reconfigurable Computing”, *Proc. FPL’98*, LNCS #148, 1998

[21] Gokhale, M., Stone, J., “NAPA C: Compiling for a Hybrid RISC/FPGA Architecture”, *Proc. IEEE Symp. on FCCMs*, Napa, 1998

[22] Dittmer. J., Sadewasser, H. Dittmer, J., “Parametrisierbare Modulgeneratoren für die FPGA-Familie Xilinx XC4000”, *Diplomarbeit*, TU Braunschweig 1996

[23] Hutchings, B. et al., “A CAD Suite for High-Performance FPGA Design”, *Proc. IEEE Symp. on FCCMs*, Napa, 1999

[24] Koch, A., “Regular Datapaths on Field-Programmable Gate Arrays”, *Doktorarbeit*, TU Braunschweig, 1997

[25] Koch, A., “Module Compaction in FPGA-based Regular Datapaths”, *Proc. 33rd DAC*, 1996

[26] Koch, A., “Enabling Automatic Module Generation for FCCM Compilers”, *Proc. IEEE Symp, on FCCMs*, Napa, 1999

[27] Booch, G., “Object-Oriented Analysis and Design with Applications, 2nd ed.”, Addison-Wesley, 1994

[28] Rumbaugh, J., Jacobson, I., Booch, G., “The Unified Modeling Language Reference Manual”, Addison-Wesley, 1999

[29] Gamma, E., Helm, R., Johnson, R., Vlissides, J., “Design Patterns”, Addison-Wesley, 1995

[30] Frost, R., “EBSA C Library Documentation”, *User Manual*, San Diego Super Computing Center (SDSC) 1993

[31] Glover, F., Laguna, M., “Tabu Search”, Kluwer, 1997

[32] Barth, P., “A Davis-Putnam Based Enumeration Algorithm for Linear Pseudo-Boolean Optimization”, *Memo MPI-I-95-2-003*, Max-Planck-Institut für Informatik, Saarbrücken 1995

[33] Williams, H.P., “Model Building in Mathematical Programming”, Wiley, 1994

[34] Willis, J. et al, “Internal Intermediate Representation (IIR) Specification”, ftp://ftp.ftlsystems.com/pub/Documents/iirA4pdf.Z. 1999

[35] Wilsey, P.A., “SAVANT: An Extensible Intermediate for VHDL”, http://www.ececs.uc.edu/~paw/savant/, 1999

[36] Koch, A., “FLAME: A Flexible API for Module-based Environments – User’s Guide and Manual”, *http://www.icsi.berkeley.edu/~akoch/research.html#FLAME*, Berkeley, 1998

[37] Koch, A., “FLAME/Java”, http://www.icsi.berkeley.edu/~akoch/research.html#FLAME, Berkeley, 1998