TDL – Eine Architekturbeschreibungssprache für Postpassoptimierungen und -analysen

Daniel Kästner

6. August 1999

Zusammenfassung

Gegenstand dieses Artikels ist TDL, eine Hardware- und Assemblybeschreibungssprache, die als Ausgangspunkt für Postpassanalysen und -optimierungen eingesetzt wird. Deren Eingabe liegt in Form von Assemblercode bzw. textuellen Darstellungen compilerspezifischer Zwischenformate vor. Die Postpassorientierung erlaubt es, effizienzsteigernde Transformationen an compilierter, bereits im Einsatz befindlicher Software vorzunehmen und ermöglicht hardwareabhängige Programmanalysen, z. B. zur Erstellung von Laufzeitgarantien für Echtzeitsysteme. Eine wichtige Anforderung ist es, die für Analysen und Optimierungen benötigten Informationen in kurzer und prägnanter Form darzustellen und für die Anwendungen in effizienter Form zugänglich zu machen. TDL erlaubt die Modellierung komplexer Abhängigkeiten, wie sie in vielen irregulären Architekturen auftreten. Durch verschiedene Abstraktionsmechanismen wird der Einsatz derselben Spezifikation für unterschiedliche Zielanwendungen unterstützt. Jede Beschreibung wird auf semantische Konsistenz überprüft; der Benutzer wird somit frühzeitig über Fehler und Inkonsistenzen in der Maschinenbeschreibung informiert.

1 Einführung

Viel Compiler für digitale Signalprozessoren erzielen aufgrund der oftmals irregulären Prozessorarchitektur nur eine unbefriedigende Codequalität[21, 24]. Die Erzeugung effizienten Codes für digitale Signalprozessoren erfordert hochoptimierende Verfahren, die an spezifische Hardwareeigenschaften des Zielprozessors angepaßt sein müssen. Viele dieser Optimierungen können als Postpassoptimierungen, d. h. nach der Erzeugung einer vorläufigen Codesequenz ausgeführt werden. Hierzu zählen z. B. integrierte Instruktionsanordnung und Registerzuweisung, Optimierungen für Mode-Setzungen, sowie auch Cache-bezogene Codeoptimierungen [12, 22]. Die Realisierung solcher Optimierungen ist zum einen im Rahmen von Compilern möglich, eine andere Möglichkeit besteht jedoch in der Entwicklung eines eigenständigen Werkzeugs. Zentrale Idee diesen Ansatzes ist, die Analyse und Qualitätsverbesserung von bereits erzeugtem Ausgangscode zu ermöglichen. Programmoptimierung ist damit für existierende Programme möglich, selbst wenn deren Quellcode nicht mehr zur Verfügung steht. Um eine Effizienzsteigerung zu erreichen, ist es somit nicht erforderlich, neue Software zu entwickeln; die bestehende Software kann weiterverwendet werden. Auf diese Weise können die Kosten für eine Effizienzsteigerung stark reduziert werden, bzw. eine Leistungssteigerung überhaupt erst ermöglicht werden, da der Umstieg auf neue Software oder ihre Neuentwicklung in der Regel mit sehr hohen Kosten verbunden ist.

Zur Entwicklung eines solchen Tools müssen einige Entwurfsentscheidungen getroffen werden. Eine wichtige Anforderung ist, daß eine Integration in existierende Systeme, also in beliebige Übersetzungsumgebungen leicht möglich sein soll. Durch die Postpassorientierung wird dies bereits in natürlicher Weise unterstützt. Ein anderer Aspekt ergibt sich aus den Anwendungsbereichen digitaler Signalprozessoren. Viele digitale Signalprozessoren werden in eingebetteten Systemen eingesetzt. Für Programme innerhalb eingebetteter Systeme muß oft die Einhaltung vorgegebener Zeitschranken bei der Programmausführung garantiert werden. Die Berechnung genauer Zeitschranken wird bei modernen Architekturen durch die Verwendung von Caches und Pipelines erschwert. Um das Zeitverhalten dieser Architekturkomponenten bei der Laufzeitberechnung zu berücksichtigen, sind komplexe, hardwareabhängiger Programmanalysen erforderlich [8]. Die Resultate solcher Analysen können auch zur Effizienzsteigerung verwendet werden, z. B. im Rahmen eines cache-sensitiven Taskscheduling [5]. Daher muß das zu entwickelnde Werkzeug nicht nur Optimierungen durchführen, sondern auch die Integration von Programmanalysen in einfacher Weise ermöglichen. Aufgrund der Architekturvielfalt digitaler Signalprozessoren sowie der schnellen Entwurfszyklen für neue Architekturen besteht eine weitere wesentliche Anforderung an ein solches Optimierungs- und Analysewerkzeug in der Retargierbarkeit [24]. Das bedeutet, daß eine Anpassung an beliebige Zielarchitekturen und beliebige Assemblersprachen mit geringem Aufwand möglich sein muß.

Ausgangspunkt muß daher eine Spezifikation der Zielarchitektur und des Instruktionssatzes sein, die alle für Optimierungen und Analysen notwendigen Informationen enthält. Die Eingabe liegt in Form von Assemblercode bzw. textuellen Darstellungen compilerspezifischer Zwischenformate vor. Die Zielbeschreibung sollte daher die automatische Generierung eines Parsers für das jeweilige Eingabeformat gestatten. Weitere wichtige Anforderungen sind Flexibilität und Modularität. Es soll einerseits ein breites Spektrum an Zielarchitekturen unterstützt und die Durchführung unterschiedlicher Analysen und Optimierungen ermöglicht werden. Zum anderen sollte es möglich sein, sich auf diejenigen Spezifikationsteile zu beschränken, die für die jeweilige Zielanwendung erforderlich sind. Es existiert eine Vielzahl unterschiedlicher Hardwarebeschreibungssprachen; diese wurden jedoch nicht unmittelbar für den Einsatz in einem postpass-orientierten Analyse- und Optimierungswerkzeug konzipiert. Daher haben wir eine eigenständige Beschreibungssprache TDL (Target Description Language) entwickelt, die im folgenden vorgestellt werden soll.

Dieser Artikel ist wie folgt aufgebaut: in Kapitel 2 werden verwandte Arbeiten vorgestellt, Kapitel 3 befaßt sich mit der Konzeption von TDL. In Kapitel 4 wird das PROPAN-Framework als Anwendungsbeispiel von TDL vorgestellt, Kapitel 5 enthält eine Zusammenfassung und einen Ausblick auf geplante Erweiterungen.

2 Verwandte Arbeiten

Hardwarebeschreibungssprachen werden für eine Vielzahl unterschiedlicher Zielanwendungen eingesetzt: zur Architektursynthese, zur Hardwaresimulation, Codeerzeugung und zur Codeoptimierung. Innerhalb der letzten Jahre wurden zahlreiche Hardwarebeschreibungsformalismen enwickelt. Im Bereich der Prozessormodellierung und -simulation haben VHDL [16] und Verilog [23] große Verbreitung; bekannte Ansätze im Bereich der Codeerzeugung sind ISPS [2], MARIL [4], MIMOLA [18], oder nML [7]. Neuere Entwicklungen sind LISA [19], das auf die Erzeugung zyklengenauer Simulatoren für Architekturen mit komplexen Pipelines zielt und ISDL [9], das zur Erzeugung eines umfassenden Toolsets, darunter Assembler, Simulatoren und Codegeneratoren entwickelt wurde. Eine Ursache für die Entwicklung einer solch großen Anzahl unterschiedlicher Beschreibungssprachen ist das Problem, verschiedene Abstraktionsebenen bzw. verschiedene Sichten innerhalb derselben Spezifikation zu modellieren. Für verschiedene Anwendungsbereiche wurden daher unterschiedliche Formalismen entwickelt.

Um als Ausgangsbasis für assembly-basierte Codeanalysen und -transformationen verwendbar zu sein, muß die Maschinenbeschreibung die Eigenschaften der Hardware erfassen und ebenso die Definition der verwendeten Assemblersprache liefern. Die einzige uns bekannte Sprache in diesem Gebiet wurde innerhalb des SALTO-Systems entwickelt [3]. Ziel von SALTO ist es, aus einer gegebenen Assemblerdatei einen generischen Kontrollflußgraphen zu erzeugen, der als Ausgangspunkt für die Implementierung von maschinenabhängigen Analysen und Transformationen dient. Die Informationen über die Zielmaschine wird, ebenso wie die Spezifikation der Assemblersyntax, aus einer Maschinenbeschreibung extrahiert. Die Eigenschaften von Ressourcen und Instruktionen können durch Attribute spezifiziert werden; allerdings besteht keine Möglichkeit, benutzerdefinierte Attribute einzuführen. Eine detaillierte Spezifikation der Operationssemantik ist nicht vorgesehen, nur eine Grobklassifizierung von Kontrollflußoperationen und die Angabe von Sprungzielen ist möglich. Daher ist die Menge der Zielanwendungen, die auf SALTO aufsetzen, beschränkt; viele Analysen benötigen zusätzliche Informationen, die der Anwender außerhalb der eigentlichen Maschinenbeschreibung zur Verfügung stellen muß. Zudem ist kein Constraintmechanismus vorgesehen, der zur Behandlung komplexer Abhängigkeiten zwischen Operationen, wie sie häufig in irregulären Architekturen auftreten, erforderlich ist (vgl. hierzu Abschnitt 3.3). Eine semantische Überprüfung der Maschinenbeschreibung, um den Benutzer auf mögliche Inkonsistenzen innerhalb der Maschinenbeschreibung aufmerksam zu machen, wird nicht durchgeführt.

ISDL [9] wurde konzipiert, um die automatische Generierung eines Codegenerators, Assemblers, Disassemblers und Instruktionssatzsimulators für die spezifizierte Zielarchitektur zu unterstützen. Da Assembler und Disassembler automatisch generiert werden, können vereinfachende Annahmen über die Struktur der Assemblersprache gemacht werden; eine vollständige Definition einer externen Eingabesprache ist daher nicht möglich. Die Menge der Ressourcentypen, die von ISDL unterstützt werden, ist beschränkt; es gibt keine Möglichkeit, zusätzliche, benutzerdefinierte Ressourcentypen einzuführen. Der derzeitigen Version von ISDL liegt ein vereinfachtes Piplinemodell zugrunde; der Schwerpunkt liegt auf VLIW-artigen Architekturen ohne komplexe Pipelines[1]. Es gibt keinen erweiterbaren Attributmechanismus innerhalb von ISDL, wodurch die Ausführung von Analysen auf der Basis von ISDL erschwert wird. Einige Informationen können nur innerhalb der Spezifikation der Operationssemantik dargestellt werden, so daß Analysen möglicherweise die komplette Semantikspezifikation jeder Operation interpretieren müssen.

nML [7] ist ein Maschinenbeschreibungsformalismus, der im Rahmen des retargierbaren Compilers CHESS [15] verwendet wird. Der Instruktionssatz wird als Attributgrammatik modelliert; jedes Terminalwort der Grammatik entspricht einer gültigen Instruktion. Zusätzlich zur Instruktionsgrammatik kann eine nML-Beschreibung Deklarationen für Speicherobjekte, Datentypen, Konstanten und Makros beinhalten. Ressourcendeklarationen sind jedoch nur für Speicherobjekte möglich; Deklarationen funktionaler Einheiten oder Caches sind nicht vorgesehen und es besteht keine Möglichkeit, zusätzliche, benzutzerdefinierte Ressourcentypen einzuführen. Abhängig von den durchzuführenden Programmanalysen werden solche Informationen jedoch benötigt, z. B. für die statische Cacheanalyse aus [8]. Zur Beschreibung von Instruktionen sind nur die Attribute `action` (Semantikbeschreibung), `syntax` (Assemblyrepräsentation) und `image` (Binärdarstellung) vordefiniert. Um Informationen über Quell- oder Zieloperanden zu ermitteln, muß die Semantikbeschreibung analysiert werden; durch die Einführung zusätzlicher Attribute zur Operandenbeschreibung würden Redundanzen eingeführt. Attribute werden in nML implizit deklariert, ohne einen Gültigkeitsbereich für Attributwerte festzulegen. Dadurch ist es nicht möglich, die Gültigkeit von Attributzuweisungen zu überprüfen; es wird keine semantische Analyse der Maschinenbeschreibung selbst durchgeführt. Desweiteren ist nML nicht zur für Postpassoptimierungen und -analysen konzipiert; eine Spezifikation der Syntax von Assemblerkommentaren, -direktiven oder von Instruktionsbegrenzern der

[1] In TDL wird ebenfalls ein vereinfachtes Pipelinemodell angenommen. Eine Modellierung komplexer Pipelines wird derzeig jedoch entwickelt.

Eingabesprache ist daher nicht vorgesehen.

3 TDL

TDL ist eine deskriptive Sprache zur Spezifikation der Hardwareressourcen und Instruktionssätzen von Zielprozessoren, sowie der verwendeten Assemblersprachen. Die primären Anwendungsgebiete von TDL sind assembler-basierte Postpassanalysen und -optimierungen. Diese Zielanwendungen erfordern die Kenntnis aller Eigenschaften der vorhandenen Hardwareressourcen, die den Analyse- bzw. Optimierungsprozeß beeinflussen können. Desweiteren müssen die Instruktionen des Prozessors erkannt werden und die Semantik der Operationen muß bekannt sein. Eine TDL-Spezifikation kann mehrere Teile umfassen: eine Spezifikation der Hardwareressourcen, eine Beschreibung des Instruktionssatzes, eine Constraintabschnitt und eine Assemblysektion. Abgesehen vom Ressourcenteil sind diese Abschnitte optional; eine gültige TDL-Beschreibung muß lediglich den Ressourcenteil enthalten. In diesem Fall kann sie als reine Hardwarebeschreibungssprache verwendet werden. Hierdurch wird eine Modularisierbarkeit der Maschinenbeschreibung realisiert; dies wird als wichtige Anforderung an Maschinenbeschreibungen angesehen und z. B. auch im Rahmen der CSDL-Sprachfamilie [6] angestrebt.

Aus einer TDL-Spezifikation wird zum einen ANSI C Code generiert, der zur Erzeugung und Initialisierung von Funktionen und Datenstrukturen dient, in denen alle relevanten Informationen über Hardwareressourcen und Instruktionssatz repräsentiert werden. Die hierfür erzeugten Dateien können in beliebige Zielanwendungen eingebunden werden und ermöglichen den generischen Zugriff auf architekturspezifische Eigenschaften. Desweiteren wird aus der TDL-Spezifikation ein Parser für die spezifizierte Eingabesprache generiert. Der generierte Parser liest die Eingabeprogramme und erzeugt einen generischen Kontrollflußgraphen im CRL-Format [14].

3.1 Die Ressourcenspezifikation

In der Ressourcenspezifikation wird jede Hardwarekomponente eingeführt, die die durchzuführenden Analysen und Optimierungen beeinflussen kann. Dem Anwender steht eine Menge vordefinierter Ressourcentypen und vordefinierter Attribute zur Verfügung, die syntaktische und semantische Informationen beinhalten können. Der Benutzer kann neue Werte für die vordefinierten Attribute deklarieren und eigene Attribute mit explizit festgelegten Wertebereichen einführen. Aus Platzgründen kann hier nur eine Grobskizzierung gegeben werden; für Details sei auf [10] verwiesen.

Die Resourcenspezifikation wird durch das Schlüsselwort `Resources-Section` eingeleitet; zur Veranschaulichung des Aufbaus dient Abb. 1. Dem Benutzer stehen mehrere vordefinierte Ressourcentypen zur Verfügung: funktionale Einheiten(`FuncUnit`), Register- und Registermengen (`Register`), Speicher (`Memory`) und Caches (`Cache`). Für jeden dieser Ressourcentypen existiert eine Menge vordefinierter (zumeist optionaler) Attribute. Es wird angenommen, daß alle eingeführten funktionalen Einheiten parallel arbeiten können; Einschränkungen dieser Parallelität müssen im Constraintteil angegeben werden. Durch die Deklaration `FuncUnit ALU 2` wird beispielsweise als Ressourcentyp eine funktionale Einheit deklariert, die durch den eindeutigem Namen `ALU` identifiziert werden kann und von der genau zwei Instanzen mit identischen Eigenschaften existieren.

Zur Beschreibung von Registermengen stehen u.a. die Attribute `size`, `type` und `usage` zur Verfügung. In Abb. 1 wird eine Menge von 32 32-bit Registern definiert, die zur Speicherung von Zweierkomplementzahlen verwendet werden. Innerhalb der Zielanwendung kann diese Registermenge durch den eindeutigen Namen `gpr` identifiziert werden und die Assemblerrepräsentation der Register lautet `r0`, `r1`, ..., `r31`. Durch das Schlüsselwort `SetProperties` können Attributen bereits deklarierter Ressourcen eigenständige Werte

```
Resources-Section

...
FuncUnit ALU 2;
Register gpr "r%d" [0:31] size=32, type=signed;
SetProperties gpr[30] usage=SP;
RegisterAlias dreg "d%d" gpr mapping=[2:1],
             type=float<56,8>;
Memory DM type=data, align=16, access=32;
DefineAttribute Replacement {"LRU", "FILO"}
             associated to Cache;
Cache InstrCache assoc=2, size=256, linesize=32,
             type=instr, Replacement=LRU;
DefineResource MyResource MyAttrib=3;
DefineAttribute MyAttrib {Int}
             associated to MyResource;
...
```

Abbildung 1: Exemplarische Resourcenspezifikation

zugewiesen werden. In Abb. 1 wird auf diese Weise das Register `r30` als Stackpointer deklariert.

Für alle Ressourcen können Aliasbeziehungen deklariert werden. Hierdurch können unterschiedliche Sichten für bereits deklarierte Ressourcen eingeführt werden. Die Attributsetzungen dieser Sichten können sich unterscheiden. Im Beispiel aus Abb. 1 wird spezifiziert, daß jeweils zwei aufeinanderfolgende Integer-Register als ein Double-Register verwendet werden können. Die Assemblerrepräsentation dieser Register lautet `d0, d1, ..., d15` und ihr Inhalt wird standardmäßig als Gleitkommazahl mit 56 Bit Mantisse und 8-bit Exponent interpretiert.

Ein Beispiel für benutzerdefinierte Attribute ist die Deklaration eines Attributes, das die Ersetzungsstrategie von Caches spezifiziert. Benutzerdefinierte Attribute werden durch das Schlüsselwort `DefineAttribute` deklariert; darauf folgt der Name des Attributes, der zulässige Wertebereich und schließlich die Menge der Ressourcen, zu deren Beschreibung dieses Attribut verwendet werden kann. In 1 kann das Attribut `Replacement` die Werte `LRU` oder `FILO` annehmen; die restlichen in der Cache-Deklaration verwendeten Attribute sind vordefinierte Attribute.

Neben den vordefinierten Ressourcentypen kann der Benutzer mit dem Schlüsselwort `DefineResource` neue Ressourcentypen definieren; diese können durch benutzerdefinierte Attribute beschrieben werden. Im Beispiel aus Abb. 1 wird zur Beschreibung der Ressource `MyResource` ein Attribut `MyAttrib` eingeführt, dessen Wertebereich die Menge der ganzen Zahlen ist. Daneben besteht die Möglichkeit, vordefinierte Attribute dahingehend zu erweitern, daß sie auch für benutzerdefinierte Ressourcen verwendet werden können.

3.2 Die Instruktionssatzspezifikation

Das TDL zugrundeliegende Ausführungsmodell ist ein VLIW[2]-Modell. Eine Instruktion kann aus mehreren parallel auszuführenden Operationen bestehen. Jede Operation wird einer Ressource zugeordnet, auf der sie ausgeführt wird. Durch diese Zuordnung wird implizit die Parallelisierbarkeit von Operationen erfaßt: alle Operationen die unterschiedlichen Ressourcentypen zugeordnet sind, können parallel ausgeführt werden. Die Anzahl gleichzeitig ausführbarer Operationen desselben Ressourcentyps wird durch die Anzahl der

[2] *Very Large Instruction Word*

```
OpNT guardnt "if %s" {guard = true, src3 = "$1" in {gpr}}{}{}{} |
             ""      {guard = false} {}{}{};
DefineOp GIAdd "%s = %!(guardnt) %s + %s" {dst = "$1" in {gpr},
                     src1 = "$2" in {gpr}, src2 = "$3" in {gpr},
                     {ALU(latency=1, slots=0, exectime=1);},
                     {if (guard == true)
                         { if (src3<0> == 0b1)
                              { dst = src1+src2; } }
                      else { dst = src1+src2; } };
```

Abbildung 2: Auszug aus einer Instruktionssatzspezifikation

Instanzen dieses Typs bestimmt. Die Deklaration `FuncUnit ALU 2` beispielsweise impliziert, daß stets zwei ALU-Operationen gleichzeitig ausgeführt werden können. Bei vielen Architekturen liegt kein völlig orthogonaler Instruktionssatz vor, d. h. die Parallelität verschiedener funktionaler Einheiten ist in der Regel eingeschränkt. Solche Einschränkungen der Parallelität zwischen unterschiedlichen Ressourcentypen oder auch zwischen Operationen können im Constraint-Abschnitt modelliert werden.

Die Beschreibung des Instruktionssatzes (`Instructionset-Section`) ist in Form einer Attributgrammatik gegeben. Jede Operation enspricht einer Produktion dieser Grammatik. Faktorisierbare Instruktionsteile, wie z. B. Adressierungsmodi, können durch eigene Produktionen beschrieben werden. Die Terminale der Grammatik entsprichen der Assemblyrepräsentation der Operationen des Zielprozessors. Der Attributmechanismus ermöglicht es, wichtige Eigenschaften von Operationen, die bei Analysen und Optimierungen berücksichtigt werden müssen, zu identifizieren. Hierzu zählen unter anderem die Assemblydarstellung jeder Instruktion, die Menge der Quell- und Zieloperanden, sowie eine Beschreibung des Zeitverhaltens. Auf diese Weise kann aus einer auf TDL basierenden Zielanwendung auf häufig benötigte Informationen direkt zugegriffen werden, ohne die detaillierte Semantikbeschreibung der Operation interpretieren zu müssen. Ein Beispiel für eine Operationsdefinition innerhalb der Instruktionssatzspezifikation ist in Abb. 2 gegeben.

Jede Operationsdeklaration wird durch das Schlüsselwort `DefineOp` eingeleitet; Nichtterminale, die innerhalb der Instruktionssatzspezifikation verwendet werden, jedoch keine eigenständigen Operationen darstellen, werden mittels `OpNT` deklariert. Jeder Operation und jedem Nichtterminal wird ein eindeutiger Name und eine Assemblyrepräsentation zugeordnet. Anschließend werden innerhalb von drei Attributgruppen die wesentlichen Eigenschaften der Operationen spezifiert. Jede Attributgruppe ist in geschweiften Klammern eingeschlossen, alle Attributsetzungen sind durch Kommata getrennt.

In der ersten Attributgruppe wird die Menge der Quell- und Zieloperanden einer Instruktion spezifiziert; dabei wird ihre Position in der Assemblydarstellung der Operation und das Speicherelement (Speicher oder Register), in dem der betreffende Operand enthalten ist, angegeben. Durch die Attributdefinition `dst1 = "$1" in {gpr}` in Abb. 2 wird festgelegt, daß der Zieloperand der Operation ein beliebiges Register der Registermenge `gpr` sein kann. Alle verwendeten Ressourcen müssen zuvor im Ressourcenteil deklariert worden sein; Teil dieser Deklaration ist immer auch die Definition der Assemblysyntax für Verweise auf die entsprechenden Ressourcen innerhalb der Assemblydarstellung der Operationen. Durch die Angabe der für den Operand verwendeten Ressource wird somit auch die syntaktische Repräsentation innerhalb des Instruktionswortes festgelegt. Im Beispiel aus Abb. 2 bedeutet dies, daß im Assemblercode `r0, r1, ..., r31` zulässige Darstellungen für das Zielregister sind. Durch `"$1"` wird ausgedrückt, daß die Assemblerrepräsentation dieses Operanden in der Assemblerdarstellung der Operation (`"%s = %!(guard) %s + %s"`) an der Position des ersten Formatstrings (eingeleitet durch '%') erwartet wird. Die Bedeutung der Formatstrings ist analog zu C [13] definiert.

Für häufig auftretende Operationskomponenten können Nichtterminale mit eigenen

Produktionen eingeführt werden; wird ein solches Nichtterminal `N` in der Syntaxdefinition einer Operation verwendet, wird es in Klammern eingeschlossen und durch ein vorangehendes `%!` markiert (`%!(N)`). In obigem Beispiel wird ein optionaler Guard spezifiziert; an der Position von `%!(guard)` kann eine der Zeichenfolgen `if r0, if r1, ..., if r31` oder aber die leere Zeichenkette auftreten. Im letzteren Fall ist die Ausführung der Operation nicht durch einen Guard geschützt, was durch die Attributsetzung `guard=false` dargestellt wird. Wurde ein Guard spezifiziert, (`guard=true`) stellt dieser einen zusätzlichen Quelloperand dar, assoziiert mit dem Attribut `src3`, der ein Elementregister des Registerfiles `gpr` sein muß.

Innerhalb der zweiten Attributgruppe wird die Instruktionsausführung durch einen Reservation-Table Mechanismus beschrieben; dieser ist in der derzeitigen Version auf die Modellierung einfacher Pipelines beschränkt, soll aber in späteren TDL-Versionen zur Modellierung komplexer Pipelines erweitert werden. Jeder Operation wird eine Menge von Alternativressourcen (i.a. funktionalen Einheiten) zugeordnet, auf der sie ausgeführt werden kann. Da das Zeitverhalten jeder Operation von der ausführenden Ressource abhängt, wird es in TDL stets in Abhängigkeit der jeweiligen Hardwareressource spezifiziert. Hierfür stehen die Attribute `exectime, latency, slots` zur Verfügung. Das Attribut `exectime` gibt die zur Ausführung der Operation benötigte Zeitspanne an, `latency` das minimale Zeitintervall, das vor der nächsten Dateneingabe zur verwendeten funktionalen Einheit verstreichen muß und `slots` gibt die Anzahl der Delay-Slots der Operation an.

Die letzte Attributgruppe beschreibt die detaillierte Semantik der definierten Operation; diese kann ebenfalls als Attribut angesehen werden. Die Semantik der Operationen wird durch eine erweiterte Registertranssprache (RTL) spezifiziert. Die Semantik einer Operation wird als Änderung des Maschinenzustandes beschrieben, der durch die Ausführung der Operation verursacht wird. Komponenten des Maschinenzustandes sind die Speichereinheiten des Prozessors, d. h. Speicherzellen, sowie Register. Zulässige RTL-Anweisungen sind Zuweisungen, bedingte Anweisungen, sowie Schleifenanweisungen; die Ausdrucksgrammatik entspricht im wesentlichen der von C [13].

Durch einen Funktionsmechanismus innerhalb dieser Registertransfersprache wird zum einen eine Modularisierung und Straffung der Semantikspezifikation ermöglicht und zum anderen ein Abstraktionsmechanismus realisiert. Dem Benutzer wird eine Menge vordefinierter atomarer Funktionen (Operatoren) zur Verfügung gestellt, die einfachen Schaltfunktionen (z. B. `iadd, isub, ...`) entsprechen. Daneben kann der Benutzer eigene RTL-Funktionen definieren. Ist der Funktionskörper für eine Zielanwendung nicht von Interesse, kann eine RTL-Funktion als Black-Box betrachtet werden; der Funktionskörper wird dann ignoriert. Auf diese Weise kann dieselbe Maschinenspezifikation für Anwendungen, die unterschiedliche Detailgenauigkeit in der Semantikbeschreibung erfordern, verwendet werden. In Abb. 2 wird durch die beschriebene Operation die Summe zweier Zweierkomplementzahlen berechnet, falls kein Guard existiert oder das erste Bit des als Guard verwendeten Registers den Wert 1 hat.

3.3 Constraint- und Assemblyspezifikation

Im Constraintabschnitt kann eine Menge von Nebenbedinungen spezifiziert werden, die bei Codetransformationen berücksichtigt werden müssen, um Korrektheit zu bewahren. Dies umfaßt z. B. Einschränkungen von Parallelverarbeitungskapazitäten der Zielmaschinen oder Abhängigkeiten zwischen Instruktionsanordnung und Registerverteilung. Bei dem digitalen Signalprozessor Analog Devices ADSP-2106x [1] ist z. B. die Parallelausführung einer ALU- und einer Multipliziereroperation nur möglich, falls alle Operanden in bestimmten Registern liegen. Befindet sich ein Operand in einem anderen Register, als zur Parallelisierung erforderlich, liegen gültige Operationen vor; diese können dann jedoch nicht parallel ausgeführt werden [12, 11]. Dies kann in TDL wie folgt spezifiziert werden:

```
op1 || op2 => op1.src1 in GroupA && op1.src2 in GroupB &&
              op2.src1 in GroupC && op2.src2 in GroupD
```

Die Assemblyspezifikation befaßt sich mit der Gesamtsyntax der als Eingabe verwendeten Assemblersprache, wie z. B. Instruktionsbegrenzer, Kommentare, Makros und Direktiven.

3.4 Konsistenzüberprüfung

Jede TDL-Beschreibung wird auf semantische Konsistenz hin überprüft. Sowohl die Eigenschaften von Hardwareressourcen als auch von Operationen werden durch Attribute beschrieben. Jedes Attribut ist mit einer Menge von Hardwareressourcen assoziiert; die Menge der möglichen Attributwerte ist explizit definiert. Dies gilt sowohl für die vordefinierten Attribute, als auch für zusätzliche, vom Benutzer eingeführte Attribute. Die Zuweisung eines ungültigen Wertes an ein Attribut oder die Verwendung eines Attributes zur Spezifikation einer Hardwareressource, mit der es nicht assoziiert ist, verursachen Fehlermeldungen. Andere semantische Überprüfungen gewährleisten die Deklariertheit von Ressourcen, verhindern das Überschreiten von Registermengengrößen, überprüfen die Typkorrektheit von RTL-Spezifikationen, etc.

3.5 Modellierte Architekturen

Mit TDL wurde eine Prozessorspezifikation für den digitalen Signalprozessor Analog Devices ADSP-2106x [1] entwickelt. Desweiteren wird eine TDL-Beschreibung des Siemens TriCore DSP erstellt, die als Ausgangspunkt einer cache-sensitiven Worst-Case Laufzeitvorhersage verwendet wird. Eine Modellierung des Philips TriMedia TM1000 [20] Prozessors ist ebenfalls in Arbeit.

4 Das PROPAN-Framework

Basierend auf TDL wird derzeit ein System PROPAN (Postpass Retargetable Optimizer and Analyzer) zur postpass-orientierten Analyse und Optimierung von Programmen aus dem Bereich der eingebetten Systeme entwickelt. Die Eingabe des PROPAN-Systems besteht aus einem Assemblerprogramm, das optimiert bzw. analysiert werden soll, sowie aus einer TDL-Beschreibung der Zielmaschine. Die TDL-Spezifikation wird auf zwei verschiedene Arten genutzt. Zum einen wird aus einer gegebenen Spezifikation ANSI C Code generiert, der zur Erzeugung und Initialisierung von Funktionen und Datenstrukturen dient, in denen alle relevanten Informationen über Hardwareressourcen und Instruktionssatz repräsentiert werden. Desweiteren wird aus der TDL-Spezifikation ein Parser für die spezifizierte Eingabesprache generiert. Der generierte Parser liest die Eingabeprogramme und erzeugt einen Basiskontrollflußgraphen. Dieser wird durch Sprungziel- und Schleifenerkennungsalgorithmen verfeinert und in der Zwischensprache `CRL` (Control Flow Representation Language [14]) dargestellt. Das `CRL`-Format dient als Schnittstelle für alle nachfolgenden Analyse- und Optimierungsalgorithmen. Kernstück der Optimierungen ist ein generisches Verfahren zur integrierten Instruktionsanordnung, Registerzuweisung und Allokation funktionaler Einheiten auf der Basis ganzzahliger linearer Programmierung [12, 11]. Desweiteren wird die Anbindung mittels des Programmanalysatorgenerators PAG [17] generierter Analysatoren unterstützt.

5 Zusammenfassung und Ausblick

In diesem Artikel wurde die Hardware- und Assemblybeschreibungssprache TDL als Ausgangspunkt für Postpassanalysen und -optimierungen vorgestellt. Diese können z. B. zur

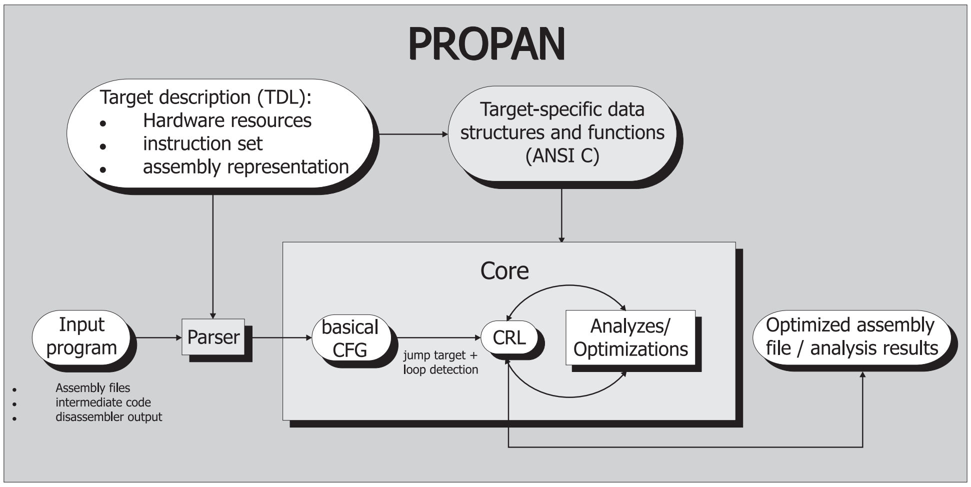

Abbildung 3: Das PROPAN-System

Durchführung effizienzsteigernder Transformationen an compilierter, bereits im Einsatz befindlicher Software sowie zur Berechnung hardwareabhängiger Programmanalysen eingesetzt werden. TDL erlaubt es, in kurzer und prägnanter Form die für Analysen und Optimierungen bedeutenden Informationen bereitzustellen. Der enthaltene Constraint-Mechanismus erlaubt es, auch komplexe Abhängigkeiten zu modellieren, wie sie in vielen irregulären Architekturen auftreten. Durch das enthaltene Alias-Konzept sowie funktionale Abstraktion kann dieselbe TDL-Beschreibung als Basis für Anwendungen mit unterschiedlichen Anforderungen verwendet werden. Jede Beschreibung wird auf semantische Konsistenz überprüft; der Benutzer wird somit frühzeitig über Fehler und Inkonsistenzen in der Maschinenbeschreibung informiert. Geplante Erweiterungen sind die Modellierung komplexer Pipelines, sowie die TDL-basierte Erzeugung von Hardwaresimulatoren und Codeselektoren.

Literatur

[1] Analog Devices: *ADSP-2106x SHARC User's Manual*, 1995.

[2] Barbacci, M. R.: *Instruction Set Processor Specifications (ISPS): The Notation and Its Applications.* IEEE Transactions on Computers, C-30(1):24 – 40, Januar 1981.

[3] Bodin, F., Z. Chamski, E. Rohou und A. Seznec: *Functional Specification of SALTO: A Retargetable System for Assembly Language Transformation and Optimization. rev. 1.00 beta.* INRIA, Juni 1997.

[4] Bradlee, D. G., R. R. Henry und S. J. Eggers: *The Marion System for Retargetable Instruction Scheduling.* Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, Seiten 229–240, Juni 1991.

[5] Daniel Kästner, Stephan Thesing.: *Cache Sensitive Pre-Runtime Scheduling.* In: *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, Montreal, CA, Juni 1998.

[6] Davidson, J.W. und N. Ramsey: *Machine Descriptions to Build Tools for Embedded Systems.* In: *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, Seiten 172 – 188. Springer LNCS, Volume 1474, Juni 1998.

[7] Fauth, A., J. Van Praet und M. Freericks: *Describing Instruction Set Processors Using nML.* In: *Proceedings of the European Design and Test Conference*, Seiten 503 – 507. IEEE, 1995.

[8] Ferdinand, C.: *Cache Behavior Prediction for Real-Time Systems.* Doktorarbeit, Universität des Saarlandes, September 1997.

[9] Hadjiyiannis, G.: *ISDL: Instruction Set Description Language Version 1.0.* Technischer Bericht MIT RLE, April 1998.

[10] Kästner, D.: *TDL – A Hardware and Assembly Description Language.* Technischer Bericht TDL1, Transferbereich 14, Universität des Saarlandes, Juni 1999.

[11] Kästner, D. und M. Langenbach: *Integer Linear Programming vs. Graph-Based Methods in Code Generation.* Technischer Bericht Universität des Saarlandes, 1998.

[12] Kästner, D. und M. Langenbach: *Code Optimization by Integer Linear Programming.* In: Jähnichen, Stefan (Herausgeber): *Proceedings of the 8th International Conference on Compiler Construction CC99*, Seiten 122 – 136. Springer LNCS 1575, März 1999.

[13] Kernighan, B. W. und D. M. Ritchie: *The C Programming Language, Second Edition, ANSI C.* Prentice-Hall International, 1988.

[14] Langenbach, M.: *CRL – A Uniform Representation for Control Flow.* Technischer Bericht CRL1, Transferbereich 14, Universität des Saarlandes, November 1998.

[15] Lanneer, D., J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen und G. Goossens: *CHESS:Retargetable Code Generation For Embedded DSP Processors.* In: Marwedel, Peter und Gert Goossens (Herausgeber): *Code Generation for Embedded Processors*, Kapitel 5, Seiten 85–102. Kluwer, 1995.

[16] Lipsett, R., C. Schaefer und C. Ussery: *VHDL: Hardware Description and Design.* Kluwer Academic Publishers, 12. Auflage, 1993.

[17] Martin, F.: *Generation of Program Analyzers.* Doktorarbeit, Universität des Saarlandes, 1999.

[18] Nowak, L.: *Graph Based Retargetable Microcode Compilation in the MIMOLA Design System.* 20th Annual Workshop on Microprogramming, Seiten 126 – 132, 1987.

[19] Pees, S., A. Hoffmann, V. Zivojnovic und H. Meyr: *LISA - Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures.* Proceedings of the DAC, 1999.

[20] Philips Electronics North America Corporation: *TriMedia TM1000 Preliminary Data Book*, 1997.

[21] Saghir, M.A.R., P. Chow und C.G. Lee: *Exploiting Dual Data-Memory Banks in Digital Signal Processors.* Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, 1996.

[22] Sudarsanam, A., S. Malik, S. Tjiang und S. Liao: *Optimization of Embedded DSP Programs Using Post-Pass Data-Flow Analysis.* Proceedings of the ICASSP, 1997.

[23] Thomas, Donald und Philip Moorby: *The Verilog Hardware Description Language.* Kluwer Academic Publishers, 2. Auflage, 1995.

[24] Zivojnovic, V., J. M. Velarde, C. Schläger und H. Meyr: *DSPSTONE: A DSP-Oriented Benchmarking Methodology.* In: *Proceedings of the International Conference on Integrated Systems for Signal Processing*, Dallas, Oktober 1994.