Code Generierung für HS-LooPo

(Programmierpraktikum am Lehrstuhl Prof. C. Lengauer, Ph. D.)

Michael Claßen

Sommersemester 2004

Inhaltsverzeichnis

1 Einführung

Im Rahmen des *LooPo*-Projekts sollen verschiedene Techniken zur automatischen Parallelisierung von Schleifen im Polytopmodell untersucht werden. Die für die unterschiedlichen Phasen der Schleifenparallelisierung vorgesehenen Module wurden dabei in C bzw. C++ implementiert, wobei der Austausch von Daten über Textdateien erfolgt.

1.1 HS-LooPo

Bei der Arbeit an und mit *LooPo* wurde die Erfahrung gemacht, dass es in C/C++ oft relativ umständlich ist, schnell kleine Tools zu schreiben, um eine neue mathematische Methode umzusetzen. Daraus entstand die Idee, neue Teile von *LooPo* in der funktionalen Programmiersprache *Haskell* zu schreiben, die dafür bekannt ist, gute Möglichkeiten zur Abstrahierung von mathematischen Problemen bereitzustellen und über eine sehr komfortable Syntax zu verfügen. Das so entstandene Projekt *HS-LooPo* soll nach und nach auch zur vollen Funktionalität des in C/C++ *LooPo* Projektes erweitert werden.

1.2 Zielcode Generierung

In *HS-LooPo* fehlte bislang eine Möglichkeit, für ein fertig parallelisiertes Schleifenprogramm ausführbaren Zielcode zu generieren. Diese Funktionalität bereitzustellen, ist die Zielsetzung dieses Programmierpraktikums. Dabei sollen möglichst viele durch die verwendete Parallelisierungstechnik auftretende Spezialfälle behandelt werden. Insbesondere soll es auch möglich sein, Zielcode für Programme zu generieren, auf welche die sogenannte *tiling*-Technik angewendet wurde, bei der die Dimensionalität des Schleifensatzes des Zielprogrammes erhöht wird.

Im Rahmen dieses Programmierpraktikums soll auf die Generierung von Code verzichtet werden, der für die Kommunikation der Prozessoren untereinander notwendig ist. Das heißt es werden ausschließlich die durch die jeweiligen verwendeten Parallelisierungsschritte transformierten Schleifensätze für die Berechnungs-Statements des ursprünglichen Quellprogramms generiert. Die parallel auszuführbaren Schleifen werden dabei als solche markiert.

2 Verwendete Hilfsmittel

Für die Realisierung eines Moduls in *HS-LooPo* zur Generierung von ausführbarem Zielcode soll zwar grundsätzlich beachtet werden, dass möglichst wenige neue Abhängigkeiten zu äußeren Tools bzw. zum bisherigen in C/C++ implementierten *LooPo*-Projekt entstehen. Andererseits hat es auch keinen Sinn, "das Rad neu zu erfinden", wenn es schon vorhandene Tools gibt, welche eine gewünschte Funktionalität bereitstellen. Im Folgenden werden kurz die verwendeten Hilfsmittel beschrieben.

2.1 Loopo

Das Quellprogramm und die darauf angewandten Transformationen liegen im *LooPo*-Projekt als Daten vor, die in Form von Textdateien von den verschiedenen Modulen erstellt bzw. eingelesen und verwendet werden. Diese Dateiformate werden auch von *HS-LooPo* verwendet, um eine einheitliche Schnittstelle für den Austausch von Daten zwischen den beiden Projekten zu ermöglichen.

Für die Generierung des Zielcodes werden dabei hauptsächlich Daten aus den folgenden Dateien benötigt:

- ispc-Datei
  Diese Datei enthält zu jedem Statement aus dem ursprünglichen (nicht transformierten) Quellprogramm den zugehörigen Indexraum, d.h. eine Beschreibung der aufgezählten Punkte im Schleifensatz, die in Form von Ungleichungssystemen vorliegt.

- trafo-Datei
  Die trafo-Datei enthält die auf das Quellprogramm angewandten Transformationen, die in Form von Transformationsmatrizen für die einzelnen Statements vorliegen.

- trc-Datei
  Diese Datei enthält zusätzliche Bedingungen, die den Quell-Indexraum beschränken,

auf den die Transformationen angewandt werden. Diese Einschränkungen des Indexraumes sind notwendig, wenn man stückweise affine Funktionen als schedule/allocation verwendet, die nur auf einem bestimmten Teilbereich gültig sind.

- qual- sowie slp-Datei
  Die qual- sowie die slp-Datei enthalten jeweils für jedes Ziel-Statement Informationen über die Art (quality) der im Zielraum verwendeten Schleifen, bzw. darüber, von welchen Schleifen das entsprechende Statement im Quellprogramm umgeben war (surrounding loops).

2.2 CLooG

Ein wesentlicher Bestandteil der automatischen Codegenerierung ist das Erzeugen der Schleifensätze für das Zielprogramm aus den transformierten Indexräumen, die in Form von Ungleichungssystemen vorliegen. Diese Ungleichungssysteme kann man mathematisch als Polyeder ansehen, deren Punkte in der richtigen Reichenfolge aufgezählt werden müssen.

Der ursprüngliche Ansatz, die Schleifen "per Hand", d.h. von Haskell aus direkt aus den Ungleichungssystemen zu generieren, stellte sich als ungünstig heraus. Für die Generierung der Ziel-Schleifensätze müssen nämlich die Zielpolyeder so "gemischt" werden, dass auch für den Fall, dass sich mehrere Polyeder überlappen, die Punkte in der richtigen Reihenfolge aufgezählt werden. Dieses richtige Aufzählen der Punkte der Polyeder wird auch als *scanning* bezeichnet.

Glücklicherweise existiert für das scanning von Polyedern und das Generieren der dazugehörigen Schleifensätze schon ein gutes Werkzeug: *CLooG* [2]. Es wird an einer französischen Universität entwickelt und basiert auf dem Algorithmus von Quilleré [3].

Neben der Möglichkeit, auch Vereinigungen von Polyedern zu mischen und die so entstandenen Schleifensätze zu generieren, bietet *CLooG* auch zwei andere wesentliche Vorteile:

- *CLooG* erzeugt erheblich effizienteren Code, als man es per Hand ohne enormen Aufwand erreichen könnte. So werden z.B. Schleifen teilweise abgerollt und komplizierte Kontrollausdrücke in den Schleifenköpfen reduziert.

- Mit Hilfe von *CLooG* kann man ein Problem umgehen, das entsteht, wenn man nichtunimodulare Transformationen auf die Polyeder des Quellprogramms anwendet. Diese so entstandenen Polyeder können gestreckt sein und würden so dazu führen, dass mehr Punkte als im Quellprogramm aufgezählt würden. *CLooG* bietet die Möglichkeit, mittels Gleichungen (die sich aus den Transformationen ergeben) die Beziehung zwischen Quell- und Ziel-Indexraum auszudrücken, ohne dass man die Polyeder explizit transformieren muss.

Um eine zu starke Kopplung zu vermeiden, wird *CLooG* nicht mittels Aufrufen von Bibliotheksfunktionen verwendet, sondern als Binary, dessen Ein- und Ausgabe jeweils in Form von Textdateien vorliegt, die von *HS-LooPo* aus verwaltet werden.

3 Mathematische Grundlagen

Bei der Parallelisierung im Polytopmodell liegen die Beschreibungen der Schleifensätze für jedes Statement in Form von Ungleichungssystemen vor. Da alle angewandten Transformationen jeweils als affin lineare Abbildungen betrachtet werden können, lassen diese sich auch als Matrizen darstellen.

Da an einigen Stellen auch die inversen Abbildungen benötigt werden, wird in *LooPo* (genauer gesagt, in dem Modul *targgen*) jeweils die Transformationsmatrix in eine invertierbare Form gebracht, indem die linear abhängigen Zeilen aus der Matrix entfernt und durch passende Zeilen aus der Einheitsmatrix ersetzt werden.

Um die ursprüngliche Matrix rekonstruieren zu können, werden zusätzliche zu der invertierbar gemachten Matrix ein Bitvektor abgespeichert, der die Zeilen in der Matrix markiert, die unverändert übernommen (benutzt) worden sind. Dieser Vektor wird hier als *used vector* bezeichnet.

Außerdem werden die Linearkombinationen abgespeichert, die angeben, wie man aus den benutzten (nicht ersetzten) Zeilen der Matrix die ersetzten Zeilen kombinieren kann.

3.1 Ungleichungssystem für Compute-Statements

Die Informationen aus der sog. trafo-Datei werden, zusammen mit den Daten aus den anderen, oben beschriebenen Dateien benutzt, um die Indexräume der Statements im Zielprogramm zu beschreiben. Diese Beschreibung erfolgt gemäß dem Eingabeformat von *CLooG* in Form eines Ungleichungssystems für jedes Ziel-Statement.

Dieses Ungleichungssystem setzt sich zusammen aus 4 Teilen:

- ispc
  Die Ungleichungen aus dem Quellprogramm, die dessen Schleifensätze beschreiben. Zu beachten ist dabei, dass jeweils nur die Ungleichungen für die umgebenden Schleifen (beschrieben durch die *slp-Datei*) verwendet werden.

- space-time-mapping
  Gleichungen (ausgedrückt als zwei gegengesetzte Ungleichungen), welche die Transformation zwischen Quell-Indexraum und den Zielkoordinaten beschreiben. Diese ergeben sich aus den jeweiligen inversen Transformationsmatrizen aus der trafo-Datei folgendermaßen:

  $$\left(\; -E \;\middle|\; iT \; \right) \times \left(\begin{array}{c} isVars \\ coordVars \end{array} \right) = 0$$

  E: Einheitsmatrix

  iT: inverse Trafo-Matrix

  $isVars$: Variablen im Quell-Indexraum

  $coordVars$: Variablen im Ziel-Indexraum

  mit:
  #Spalten(E) = #Variablen im Quell-Indexraum,
  #Zeilen(E) = #Zeilen(iT),
  #Spalten(iT) = #Variablen im Ziel-Indexraum

  Durch diese Beschreibung des Zielkoordinaten-Ungleichungssystems werden auch nicht unimodulare Transformationen durch *CLooG* korrekt behandelt.

- linCombEqs
  Diese Gleichungen (als Ungleichungen ausgedrückt) sind zusätzlich zu den space-time-

mapping Gleichungen notwendig. Man stellt damit die Gleichungen der Transformationsmatrix wieder her, die entfernt worden sind, um die Invertierbarkeit der Matrix zu gewährleisten. Für die Rekonstruktion dieser Gleichungen wird auf die oben genannten "used-Vektoren" aus der trafo-Datei zurückgegriffen.

- trc
  Die trc-Ungleichungen schränken den Quell-Indexraum auf den Bereich ein, auf dem die aktuelle Transformation gültig ist, z.B. für den Fall, dass man stückweise affine Funktionen als Transformationen hat, die nur auf bestimmten Teilbereichen gelten.

- tiling
  Dieser Teil des Ungleichungssystems wird nur benutzt, wenn man die Option für das Anwenden der Tiling-Funktion aktiviert hat. Die Gleichungen (wieder als gegengesetzte Ungleichungen dargestellt), definieren die Beziehung zwischen der globalen Tile-Nummer (globale Zeit und Realprozessor-Nummer) und den absoluten Koordinaten im Zielraum.

Damit ergibt sich der Aufbau des Ungleichungssystems für ein Statement wie folgt:

<table>
<tr><td colspan="3">Variablen</td><td>Parameter</td><td>Const</td></tr>
<tr><td>is-Vars</td><td>tileNr-Vars</td><td>coord-Vars</td><td></td><td>1</td></tr>
<tr><td>ispc-Vars</td><td colspan="2">0</td><td>ispc-Params</td><td>ispc-Const</td></tr>
<tr><td>$-E$</td><td>0</td><td>T^{-1}-Vars</td><td>T^{-1}-Params</td><td>T^{-1}-Const</td></tr>
<tr><td colspan="2">0</td><td>linCombEqs-Vars</td><td>linCombEqs-Params</td><td>linCombEqs-Const</td></tr>
<tr><td>0</td><td colspan="2">tiling-Vars</td><td>tiling-Params</td><td>tiling-Const</td></tr>
<tr><td>trc-Vars</td><td colspan="2">0</td><td>trc-Params</td><td>trc-Const</td></tr>
</table>

Diese Tabelle ist also so zu verstehen, dass die Einträge dieser Tabelle M jeweils für die linken Seiten der Ungleichung $M \times \vec{x} \geq 0$ stehen. Dabei ist $\vec{x}$ ein Vector aus den is-, tileNr- und coord-Variablen, den Parametern und der Konstanten 1.

3.2 Scatterfunktion für Compute-Statements

Wie bereits erwähnt, liegt ein großer Vorteil bei der Benutzung von *CLooG* für die Codegenerierung darin, dass auch das Mischen von Indexräumen verschiedener Statements möglich ist, wobei auch mehrere Statements von einem gemeinsamen Schleifensatz umgeben sein können.

Verwendet man nun zur Beschreibung der Zielstatements ausschließlich das oben beschriebene Ungleichungssystem, so werden damit lediglich die Punkte im Ziel Indexraum eines Statements korrekt aufgezählt. Es wird aber noch nicht die Reihenfolge der Instanzen der verschiedenen Statements untereinander festgelegt für den Fall, dass sich mehrere Statements einen gemeinsamen Schleifensatz "teilen". Um diese Reihenfolge festzulegen, verwendet man zusätzliche Hilfsdimensionen.

Um nicht die Ungleichungssysteme mit zusätzlichen Dimensionen auffüllen zu müssen (welche die weitere Verarbeitung deutlich aufwändiger machen würden), bietet sich in *CLooG* der Mechanismus der sog. *scatter-Funktionen* an. Es lassen sich dazu affin lineare Funktionen angeben, die in Abhängigkeit der Dimensionen des Ungleichungssystems für ein bestimmtes Statement zusätzliche Zieldimensionen (genannt "scatter-Dimensionen") beschreiben, durch die man die gewünschte zeitliche Anordnung der Instanzen der Statements untereinander erzwingen kann.

Im Wesentlichen wird für die Codegenerierung in unserem Fall nur eine zusätzliche Dimension benötigt, um die Reihenfolge der Operationen festzulegen. Für die (hier nicht implementierte) Generierung des Codes für die Kommunikation benötigte man allerdings noch einige andere Hilfsdimensionen, bzw. müsste man die Dimensionen aus dem jeweiligen Ungleichungssystem durch Scatterfunktionen anders in der Zeit anordnen.

Da wir uns aber in dem vorliegenden Fall nur auf die Generierung des Zielcodes *ohne Kommunikation* beschränken, können wir auf diese zusätzlichen Erweiterungen verzichten und müssen für die Scatterfunktionen nur unterscheiden, ob man *Tiling* verwendet oder nicht. Es ergeben sich dann folgende Tabellen:

- ohne *Tiling*:

scatterDims		ineqDims		Const
coords'	stmtNr	ispc	coords	1
E	0	0	$-E$	0
0	1	0	0	$-stmtNr$

- mit *Tiling*:

scatterDims			ineqDims			Const
tileNr'	coords'	stmtNr	ispc	tileNr	coords	1
E	0	0	0	$-E$	0	0
0	E	0	0	0	$-E$	0
0	0	1	0	0	0	$-stmtNr$

Dabei steht E jeweils für die Einheitsmatrix mit der passenden Größe, die sich jeweils aus der Anzahl der Dimensionen für die absoluten Koordinaten (`coords`) bzw. für die Tile-Nummern (`tileNr`) ergibt

Wie man sieht, werden die Dimensionen des Quell-Indexraumes nur für die Generierung der Ungleichungen für die Zielschleifensätze der Statements verwendet, nicht aber für das Positionieren letzterer untereinander.

Da *CLooG* sowohl Code für die scatter-Dimensionen als auch für die restlichen Dimensionen aus dem Ungleichungssystem generiert, können die Quell-Indizes aus den ispc-Dimensionen z.B. für Array-Zugriffe verwendet werden (ohne eine sonst notwendige Funktion, welche die Zielkoordinaten in passende Koordinaten im Quell-Indexraum umwandelt, um Arrays richtig zu indizieren).

4 Implementierung

Die Implementierung dieses Moduls zur Code Generierung in *HS-LooPo* liegt in Form eines Haskell-Moduls vor, das auch eine main-Funktion besitzt, so dass man es als eigenständig ausführbares Programm laufen lassen kann.

Der Ablauf der Code Generierung erfolgt dabei in mehreren Phasen:

- Vorbereitung der Daten
  Zunächst werden alle benötigten Daten aus den von *LooPo* erzeugten Zwischendateien ausgelesen und in eine Form gebracht, die für die weitere Verarbeitung geeignet ist. So werden aus den als Listen unterschiedlicher Länge ($\#Quell-$ bzw. $\#Ziel-Statements$) vorliegenden Rohdaten eine Liste generiert, deren Einträge jeweils die für jedes Ziel-Statement benötigten Daten enthalten.

- Generierung der Eingabedaten für *CLooG*
  Pro Statment wird jeweils ein Ungleichungssytem zur Beschreibung des Zielschleifensatzes erstellt. Dabei wird wahlweise Tiling verwendet. Außerdem werden die jeweiligen scatter-Funktionen erzeugt und die verschiedenen Dimensionen mit passenden Namen versehen.

- Erzeugung des Zielschleifensatzes
  Die vorliegenden Daten werden gemäß dem spezifizierten Eingabeformat formatiert und als eine Datei abgespeichert, die an *CLooG* übergeben wird. Die generierte Ausgabe wird ebenfalls in einer Datei abgelegt.

- Nachbearbeitung der Ausgabe von *CLooG*
  Schließlich wird noch eine weitere Datei erzeugt, die zusätzliche C-Präprozessor `#define`-Ausdrücke enthält, die dafür sorgen, dass für die Platzhalter, die von *CLooG* verwendet werden, der eigentliche Code für die Statements eingesetzt wird. Außerdem werden für jedes Ziel-Statement Zuweisungen generiert, die aus dem generierten Zielschleifensatz Variablen auswählen und so umbenennen, dass sie von den jeweiligen Quell-Statements verwendet werden können. Dies ist notwendig, da durch das Mischen der Indexräume eine Variable im generierten Schleifensatz für mehrere unterschiedliche Variablen im Quellraum stehen kann, falls das Quellprogramm nicht perfekt verschachtelt war.

4.1 Datenstrukturen

Folgende Datenstrukturen wurden verwendet, um die jeweils benötigten Daten für die einzelnen Phasen zu repräsentieren:

- CloogDS
  In der Datenstruktur *CloogDS* werden alle Daten zusammengefasst, die für die spätere Repräsentation der Berechnungsstatements für *CLooG* benötigt werden. So z.B. die internen Datenstrukturen, die in *HS-LooPo* die Indexräume oder die Transformationen repräsentieren.

- CloogStmtData
  In *CloogStmtData* sind die Daten für die Generierung *eines* Ziel-Statements für *CLooG* aufgeführt und liegen in einer für die weitere Verarbeitung aufbereiteten Form vor.

- CloogStatement
  Der Datentyp *CloogStatement* stellt eine abstrakte Repräsentation eines Statments dar, mit dem von *CLooG* erwarteten Ungleichungssystem zur Beschreibung der aufzuzählenden Punkte im Zielraum sowie den für die Scatterfunktion benötigten Daten.

- CloogStmts
  Diese Datenstruktur faßt alle für die Eingabe von *CLooG* benötigten abstrakten Statement Beschreibungen zusammen und beinhaltet zusätzliche Daten um in der Nachbearbeitung die korrekten Bezeichnernamen für die Schleifenvariablen und Parameter zu erzeugen.

4.2 verwendete Funktionen

Es folgt eine kurze Übersicht über die wichtigsten verwendeten Funktionen:

- `createComputeStmtData :: CloogDS -> [CloogStmtData]`
  Diese Funktion erzeugt aus den rohen Daten eine Liste von `CloogStmtData` Datenstrukturen, die Informationen für jedes Ziel-Statement getrennt aufführt. Siehe dazu die Phase "Vorbereitung der Daten".

- `createComputeStatement :: CloogStmtData -> CloogStatement`
  Hier wird eine abstrakte Repräsentation eines Ziel-Statements erstellt, d.h. es wird das Ungleichungssystem für die Beschreibung des aufzuzählenden Schleifensatzes berechnet. Siehe dazu Kapitel 3.1 "Ungleichungssystem für Compute-Statements".

- `genScatterFct :: Integer -> CloogStatement -> String`
  Hier wird aus der als `CloogStatement` vorliegenden Repräsentation eines Statements die entsprechende *scatter-Funktion* für *CLooG* erzeugt. Siehe dazu Kapitel 3.2 "Scatterfunktionen für Compute-Statements". Weil dabei die Anzahl der zusätzlichen scatter-Dimensionen für alle Statements gleich sein muss, wird ein zusätzlicher Parameter (`commonCoordsDim`) verwendet, der die Anzahl der gemeinsamen Dimensionen für die Koordinaten im Zielraum angibt, aus denen sich auch die Anzahl der scatter-Dimensionen ergibt.

- `getScatterNames :: CloogStmts -> [String]`
  Diese Funktion liefert eine Liste der Bezeichner, die im Zielprogramm für die Variablen der scatter-Dimensionen verwendet werden. Dabei beinhalten diese Bezeichner auch Informationen über die Art der Schleifen (parallel, sequentiell) und über deren Bedeutung (Zeit oder Prozessor, globale Tile-Koordinaten, absolute Koordinaten).

- `generateCloogInput :: CloogStmts -> String`
  Hier wird aus den vorhandenen abstrakten Beschreibungen aller Statements die Eingabe für *CLooG* in dem geforderten Format erzeugt. D.h. es werden die Informationen über die Indexräume, die scatter-Dimensionen, sowie verwendete Parameter und deren Einschränkungen generiert.

- `runCloog :: String -> String -> String -> IO ()`
  Diese monadische Funktion dient zum Aufruf des *CLooG*-Executables mit einer (vorher generierten) Eingabedatei und einer Ausgabedatei, die den von *CLooG* erzeugten Zielschleifensatz enthält. Dies findet in der Phase "Erzeugung des Zielschleifensatzes" statt.

- `createDefinesFile :: String -> String -> String -> IO ()`
  Diese monadische Funktion generiert eine Datei, die für jedes Zielstatement den von *CLooG* generierten Platzhalter durch den Code für das Quell-Statement einsetzt, sowie

die von *CLooG* erzeugten Ziel-Schleifenvariablen auf die im Code des Quell-Statements verwendeten Variablen abbildet. Um den Code des Quell-Statements in der gewünschten Form zu erhalten, wird das *LooPo*-Modul *targout* mit einer angepassten Sprachbeschreibungsdatei verwendet.

4.3 Aufruf und Kommandozeilenparameter

Der Aufruf des Moduls erfolgt aus Haskell heraus über den Aufruf der `main`-Funktion oder von der Kommandozeile unter Angabe des Namens der ausführbaren Datei:

codeGen `tempDir projFile latVectors cloogExec targoutExec`

- tempDir
  Dieser String steht für das temporäre Verzeichnis, in dem sich die benötigten, von *LooPo* generierten Dateien befinden.

- projFile
  Dieser String steht für den Pfad der Projekt Datei, auf die sich die Endungen der restlichen Zwischendateien beziehen.

- latVectors
  Die für Tiling für das Aufzählen der Tiles benötigten Gitter-Vektoren (*lattice*-Vektoren) werden hier als eine Liste von Listen von Integer angegeben (in Haskell-Notation, wobei die ganze Liste in doppelten Anführungszeichen eingeschlossen wird). Soll Tiling nicht verwendet werden, gibt man die leere Liste (`"[]"`) an.

- cloogExec
  Dieser String steht für den Pfad des *CLooG*-Executables.

- targoutExec
  Dieser String steht für den Pfad des Executables des *LooPo*-moduls *targout*.

5 Zusammenfassung

Für die Code-Generierung in *HS-LooPo* wird *CLooG* verwendet, um die Zielschleifensätze aus den transformierten Indexräumen zu erzeugen. Dabei werden die scatter-Funktionen verwendet, um mit Hilfe zusätzlicher Hilfsdimensionen die korrekte Ausführungsreihenfolge der Instanzen der Ziel-Statements zu erhalten.

Bei der Generierung des Ziel-Codes mittels *CLooG* kann folgendes beobachtet werden:

- benötigte Zeit zur Code-Generierung:
  CLooG liefert in allen bisher verwendeten Beispielen die Zielschleifensätze in wenigen Sekunden. Allerdings werden an einigen Stellen Algorithmen verwendet (z.B. zum Sortieren von Ungleichungssystemen), die ein mindestens exponentielles Laufzeitverhalten aufweisen. Deshalb ist zu erwarten, dass für sehr komplizierte Quellprogramme die benötigte Zeit für die Code-Generierung stark anwachsen wird.

- Code-Größe des erzeugten Codes:
  Bei der Erzeugung des Zielschleifensatzes mit *CLooG* wird sehr viel Wert auf ein gutes Laufzeitverhalten des Ziel-Codes gelegt. Dabei werden z.T. Techniken angewendet,

die dazu führen, dass die Code-Größe des erzeugten Codes stark anwächst (so z.B. durch Duplizierung von Code wegen Fallunterscheidung für unterschiedliche Parameter-Wertebereiche). In einigen Beispielen werden Ziel-Code-Größen von über 7000 Zeilen bei einer Eingabe von nur 19 Zeilen erreicht.

Literatur

[1] `http://www.infosun.fmi.uni-passau.de/~loopo/`

[2] `http://www.prism.uvsq.fr/~cedb/bastools/cloog.html`

[3] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. International Journal of Parallel Programming, 28(5):469-498, october 2000.