Diplomarbeit

Codegrößenreduktion eingebetteter Systeme durch kombiniertes In- und Exlining

Peter Imhoff
Lehrstuhl Informatik XII
Jens Wagner,
Prof. Dr. Peter Marwedel

Diplomarbeit am Fachbereich Informatik der Universität Dortmund

Codegrößenreduktion eingebetteter Systeme durch kombiniertes In- und Exlining

Peter Imhoff
Lehrstuhl Informatik XII
Jens Wagner,
Prof. Dr. Peter Marwedel

Peter Imhoff
Lentorfstr. 32
45307 Essen
peter.imhoff@epost.de

21. September 2003

Inhaltsverzeichnis

Kapitel 1

Einführung

In der heutigen Zeit gehören mobile Geräte zum Alltag. Längst ist es selbstverständlich geworden, zu jeder Zeit, an jedem Ort telefonieren, Musik hören oder seine Termine planen zu können. Die Anforderungen an mobile Geräte, die diesen Ansprüchen genügen, werden immer größer. Während sich die Geschwindigkeiten und Speicherkapazitäten der Chips in den letzten Jahrzehnten noch an Moore's und Joy's Gesetze hielten, d.h. sie verdoppelten sich etwa alle 18 Monate, konnten die Kapazitäten von Akkus nicht mit den steigenden Anforderungen Schritt halten. Auch wenn der Energiebedarf nicht proportional zur gestiegenen Speicherkapazität gewachsen ist, so bildet die Anforderung hoher Stand-By-Zeiten eine starke Restriktion bzgl. des Strombedarfs.

Abbildung 1.1: Wafer, Quelle: heise-news

Eine Möglichkeit, Energie zu sparen ist, Programme mit weniger Speicherzugriffen zu entwerfen, bzw. möglichst On-Chip-Speicher bei der Ausführung von Programmen zu benutzen. Weiterhin ist es allgemein günstiger Chipfläche zu sparen, da man so in der Fertigung mehr Chips pro Die (siehe Abbildung 1.1 unterbringen kann und so die Ausschussrate und damit die Stückkosten reduzieren kann.

Neben der Reduktion des Energieverbrauchs und der Fertigungskosten ist auch die Leistung, d.h. in diesem Fall die mögliche Beschleunigung eines Systems durch parallele Verarbeitung durch mehrere Prozessoren auf einem Chip, durch die Chipfläche eingeschränkt. Bei Netzwerkprozessoren z.B. geht die Zahl der Prozessoren linear in die Verarbeitungsgeschwindigkeit ein, da verschiedene Verbindungen parallel und unabhängig voneinander auf verschiedenen Prozessoren verarbeitet werden können [FW02]. Außerdem benötigen die Prozes-

soren Speicher für die Daten, die sie parallel verarbeiten wollen, der wie die Prozessoren selbst Chipfläche benötigt. Also kann die Reduktion des Speicherbedarfs und der damit verbundene geringere Chipflächenbedarf auch zur Geschwindigkeitssteigerung beitragen. Es gibt also zahlreiche Gründe, um den Speicherbedarf eines Programms zu reduzieren. Dafür gibt es verschiedene Optimierungen.

Für zahlreiche Optimierungen ist es vorteilhaft, wenn Funktionsaufrufe durch den Rumpf der Funktion ersetzt werden, d.h. Inlining durchgeführt wird. Das Inlining erleichtert, Datenabhängigkeiten zu erkennen und datenflussorientierte Optimierungen durchzuführen. Dazu zählen *Register Allocation, CSE Elimination, Constant Propagation* und *Dead Code Elimination*. Wenn man die Parameter einer Funktion im Rumpf ersetzt hat, kann man zum Beispiel direkt die Register benutzen, die die passenden Werte enthalten und kann sich so Befehle für das Kopieren in bestimmte Register sparen. Bei einer Funktion, die von mehreren Positionen aufgerufen wird, ist man sonst eingeschränkt. Bei konstanten Parametern können diese entsprechend behandelt werden, d.h. Berechnungen mit evtl. vorhandenen weiteren Konstanten können durch die konstanten Werte ersetzt werden. Durch das Ersetzen von Variablen durch Konstanten, kann es dazu kommen, dass bedingt ausgeführter Code einer Funktion nie ausgeführt wird, weshalb man diesen *Dead Code* anschließend eliminieren kann. Gleiche Befehlsfolgen, die durch das Inlining an den Positionen der Funktionsaufrufe entstehen und kein Optimierungspotential für die eben erwähnten datenflussabhängigen Optimierungen schaffen, sollen durch ein anschließendes Exlining eliminiert werden.

Beim Exlining versucht man gleiche Befehlsfolgen zu finden und zu Pseudo-Funktionen herauszuziehen. Dabei wird jedes Auftreten einer häufig vorkommenden Befehlsfolge durch einen Sprung zu einer Pseudo-Funktion, die diese Befehle enthält, ersetzt. Pseudo-Funktion deshalb, weil die Art, die Funktion aufzurufen, nicht einem üblichen Funktionsaufruf entspricht. Außerdem sollen im Exlining-Schritt, auch ohne das Inlining auftretende, gleiche Befehlsfolgen gefunden und ggf. zu Pseudo-Funktionen ausgelagert werden. In der Diplomarbeit soll gezeigt werden, ob und wie mit kombiniertem Inlining und Exlining die Codegröße eines Programms reduziert werden kann. Das Erstellen eines kompletten optimierenden Compilers würde aber den Zeitrahmen einer Arbeit sprengen. Ein freier Compiler bietet die Möglichkeit, bei offenem Quelltext, prinzipiell in jeder Phase des Compilers eingreifen zu können. Mit dem TriCore hat man einerseits ein eingebettetes System, für den die Betrachtung Codegrößenoptimierender Verfahren interessant sind. Andererseits gibt es für den TriCore bereits, neben kommerziellen Compilern, wie denen von Green Hill oder Tasking, auch eine freie HighTec GCC-Version, die neben dem bekannten GNU Compiler weitere nützliche Werkzeuge zur Analyse von übersetzten Dateien enthält. Während das Inlining noch auf der Quellcodeebene betrachtet wird, soll das Exlining auf der Ausgabe des GCC operieren. Dazu wird der vom GCC erzeugte Assemblercode in die **LLIR** [Eck01] überführt. Die **LLIR** (**L**ow **L**evel **I**ntermediate **R**epresantation) dient mir als Eingabe für das Exlining, da sie sich als geeignete Zwischendarstellung für weitere Optimierung anbietet. So betrachtet Robert Pyka in seiner Diplomarbeit [Pyk03] Bitlevel-Optimierungen für

die gleiche Architektur und benutzt für seine Optimierungen ebenfalls die **LLIR**. Daher teilen wir uns den von der Optimierung unabhängigen Teil, die Assembler-Ausgabe des GCC in die **LLIR** zu überführen und aus der **LLIR** nach Optimierungen wieder Assembler-Code zu erzeugen.

Kapitel 2

Stand der Technik

Bisherige Studien, die sich mit Inlining befasst haben, wie die Arbeit von P.P. Chang, S. A. Mahlke, W.Y. Chen und W.W. Hwu [CMCWH92] betrachten Inlining aus dem Blickwinkel von Geschwindigkeitsoptimierung. Dabei nimmt man im Allgemeinen eine Vergrößerung des übersetzten Programms in Kauf. Im oben erwähnten Forschungsbericht kam man auf eine durchschnittliche Vergrößerung der Programme um 16 %. Außerdem stellte sich heraus, dass man zur Wahl einer geeigneten Funktion zum Inlining nur eine Heuristik wählen kann, da die möglichen Einsparungen durch spätere Optimierungen nicht präzise vorhergesagt werden können, ohne das Inlining tatsächlich durchzuführen. Andere Analysen zum Thema Inlining, wie: “Inline Expansion: When And How?” [Ser97] oder “Flow-directed Inlining” [JW96], versuchen Laufzeitvorteile durch das Wegfallen von Funktionsaufrufen zu gewinnen.

Betrachtet man bisherige Arbeiten, die sich mit Exlining beschäftigen, so beinhalten diese hauptsächlich interaktive Werkzeuge, wie die Untersuchungen von Frank Vahid [Vah95b, Vah95a]. Es gibt zwar auch Forschungsberichte, die automatisiertes Exlining beschreiben, wie die von Nyström u.a. [NRS00], dort werden aber auch Basisblöcke als kleinste Einheit betrachtet.

Interaktive Werkzeuge, denen man Muster vorgeben kann, nach denen Befehlsfolgen gesucht werden können, wie sie z.B. in Frank Vahids: “Procedure exlining: A transformation for improved system and behavioral synthesis” [Vah95b] beschrieben werden, arbeiten ebenfalls mit Basisblöcken als kleinste Einheit. Dies ist bei Interaktivität auch gar nicht anders machbar, da Menschen nicht in der Lage wären, die große Zahl von Möglichkeiten, Befehlsfolgen zu Funktionen zusammenzufassen, zu bewerten und eine geeignete Auswahl zu treffen.

Eine Bewertung kürzerer Befehlsfolgen scheint auf den ersten Blick vielleicht nicht sinn-

voll, da das Exlining jeder Befehlssequenz einen Laufzeit-Overhead bedeutet, der für den Sprung zu der Subroutine benötigt wird. Geht man allerdings davon aus, dass die in [HP95] beschriebene 90 / 10 Regel gilt, d.h. ein Programm führt 90 % seiner Anweisungen in 10 % seines Codes aus, so ist der Verlust an Geschwindigkeit relativ gering, wenn man die vermeintlich am häufigsten ausgeführten Befehlsfolgen vom Exlining ausschließt.

In dieser Arbeit soll es neben dem Inlining auch um das Exlining gehen. Bisher hat sich keine Arbeit sowohl mit Inlining als auch mit Exlining zur Codegrößenreduktion beschäftigt. Die beiden Techniken scheinen auf den ersten Blick widersprüchlich zu sein. Es lässt sich aber insgesamt kleinerer Code erreichen, vorausgesetzt man findet eine geeignete Heuristik, um zu entscheiden, welche Funktion mittels Inlining hereingezogen werden soll.

Eine Behandlung kürzerer Befehlssequenzen war einerseits zu rechenaufwendig, andererseits schien der zeitliche Verlust durch den Aufruf-Overhead für sehr kurze Befehlsfolgen zu groß zu sein. Diesen Problemen wird im Algorithmus zum Exlining, der in dieser Diplomarbeit vorgestellt wird, zum einen mit einer effizienten Suche von Präfixgruppen begegnet, zum anderen bietet das Verfahren die Möglichkeit, Befehle, die eine zu große Schleifentiefe besitzen, vom Exlining auszuschließen, um den Zuwachs der Laufzeit durch häufige Sprünge zu kurzen Befehlsfolgen möglichst gering zu halten.

Kapitel 3

Grundlagen

Hier werden die nötigen Grundlagen für meine Arbeit erläutert. Dazu gehören der Aufbau eines Compilers, Inlining, das Exlining und die **B**urrows-**W**heeler-**T**ransformation, die dem Exlining zu Grunde liegt.

Für das Inlining werden die benötigten Werkzeuge betrachtet. Anfangs schien **SUIF** ein geeignetes Instrument zu sein, um Inlining durchzuführen. Eine genauere Betrachtung bewog mich jedoch zu einer eigenen schlanken Implementierung eines C-Frontends, das am Code nur die gewünschten Änderungen vornimmt, da **SUIF** teilweise eigenmächtig Typdefinitionen einführte, die wohl für andere Optimierungen genutzt werden sollten, aber zu größerem Code führten. Daher werden für das Inlining noch die Werkzeuge *bison* und *flex* beschrieben.

Für das Exlining werden neben der **B**urrows-**W**heeler-**T**ransformation auch Grundlagen über die **LLIR** und die Eigenschaften des TriCore erläutert.

3.1 Aufbau eines Compilers

Die folgende Beschreibung des Aufbaus eines Compilers stützt sich im Wesentlichen auf [ASU88]. Der Aufbau eines Compilers lässt sich in mehrere Phasen aufteilen:

Wie in Abbildung 3.1 dargestellt, arbeitet ein Compiler konzeptuell in mehreren Phasen. Als Eingabe erhält er zunächst den Programm-Quelltext.

- *Lexical Analyzer:* Auf diesem Quelltext wird eine lexikalische Analyse durchgeführt,

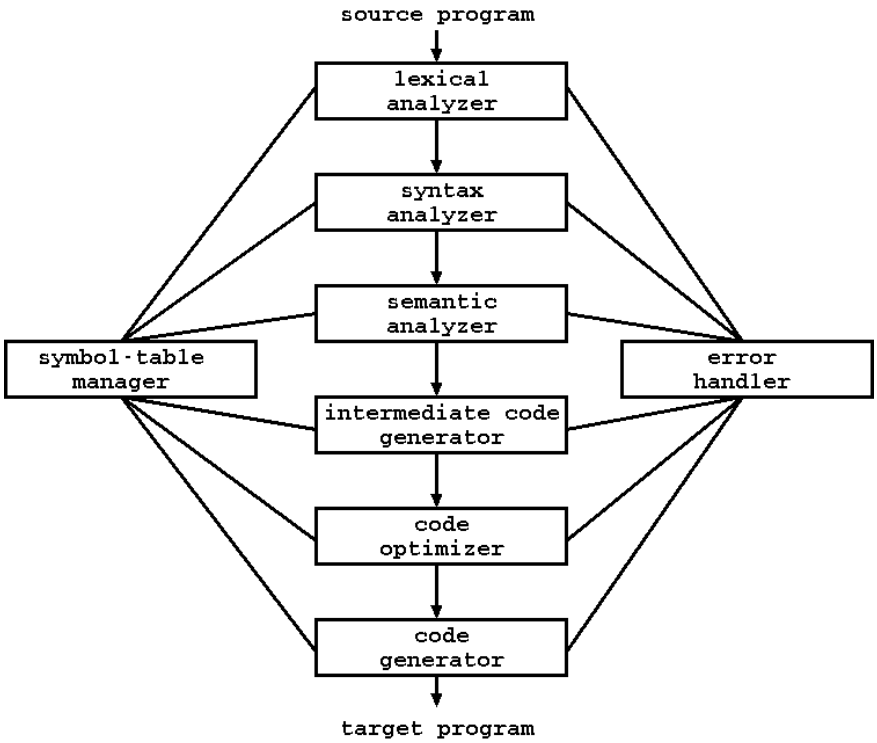

Abbildung 3.1: Compilerphasen

d.h. er wird in sogenannte *Tokens* aufgeteilt. Jeder *Token* stellt dabei eine logisch zusammenhängende Einheit dar, wie z.B. ein *Identifier*, ein *Keyword* oder *Operator*, der aus mehreren Zeichen besteht, wie der Zuweisungsoperator *":="*.

- *Syntax Analyzer:* Nach der Aufteilung in *Tokens* wird in der Syntaxanalyse eine hierarchische Darstellung, häufig in der Form eines Syntaxbaums, erstellt. Bei einem Syntaxbaum hat man an den inneren Knoten einen Datensatz mit einem Feld für den Operator und zwei weiteren Feldern für Zeiger auf die linken und rechten Kinder. Ein Blatt in einem Syntaxbaum ist dabei ein Datensatz, der einerseits den Token des Blattes identifiziert und andererseits Informationen über den Token enthält.

- *Semantic Analyzer:* In der semantischen Analyse-Phase wird überprüft, ob eine Folge von Tokens wirklich zu einer Sprache gehört. Dabei hilft es, sich vorzustellen, dass ein sogenannter *Parse Tree* erzeugt werden soll. Auch wenn ein Compiler diesen nicht tatsächlich erzeugt, muss es dennoch möglich sein, diesen eindeutig zu erzeugen, da sonst eine fehlerfreie Übersetzung nicht garantiert werden kann. Formal ist ein *Parse Tree* einer kontextfreien Grammatik ein Baum mit folgenden Eigenschaften:

  1. Die Wurzel wird mit dem Start-Symbol gekennzeichnet.
  2. Jedes Blatt wird mit einem *Token* oder ϵ bezeichnet.

3. Jeder innere Knoten wird mit einem nicht-Terminal-Zeichen gekennzeichnet.
4. Wenn A einen inneren Knoten bezeichnet, der mit einem nicht-Terminal-Zeichen gekennzeichnet wurde, und $X_1, X_2, \ldots, X_n$ sind die Bezeichner der Kinder dieses Knotens von links nach rechts, dann ist $A \rightarrow X_1 X_2 \ldots X_n$ eine Produktion. Dabei stehen die X_1 bis X_n jeweils für ein Terminal- oder nicht-Terminal-Zeichen. Als ein Sonderfall kann A ein einzelnes Kind haben, dass mit ϵ bezeichnet wird, wenn $A \rightarrow \epsilon$ eine Produktion der Grammatik ist.

- *Intermediate Code Generator:* Nach der syntaktischen und semantischen Analyse eines Programms erzeugen einige Compiler eine *Intermediate Code* Darstellung des Programms. Diesen *Intermediate Code* kann man sich als Code für eine abstrakte Maschine vorstellen. Der *Intermediate Code* muss dabei einfach zu erzeugen sein und natürlich auch leicht in einen konkreten Code für eine reale Maschine umzusetzen sein. Eine zwischenzeitliche Darstellung als *Intermediate Code* bietet sowohl den Vorteil, dass man bei Umstellung auf eine andere reale Maschine nicht den gesamten Compiler neu schreiben muss, als auch den Vorteil, dass man einfache Optimierungen bereits auf diesem Intermediate Code durchführen kann und so mit einer Optimierung besseren Code für verschiedene Plattformen erhält.

- *Code Optimizer:* Die Code Optimierungsphase versucht den Intermediate Code zu verbessern, d.h. schnelleren oder kompakteren Intermediate Code, also maschinenunabhängig besseren, Code zu erzeugen.

- *Code Generator:* Die letzte Phase eines Compilers besteht darin, aus der Zwischendarstellung aus der "Intermediate Code Generation"-Phase den Ziel-Code zu erzeugen. Der Ziel-Code besteht üblicherweise aus verschiebbarem Maschinencode oder Assembler-Code.

Die eben beschriebenen 6 Phasen interagieren mit dem *Symbol-Table Manager* und dem *Error Handler*.

- *Symbol Table Manager:* Eine wesentliche Aufgabe eines Compilers besteht darin, sich die in einem Programm benutzten Bezeichner und die zugehörigen Attribute, wie z.B. Typ oder Kontext, in dem die Bezeichner gültig sind, zu merken. Wenn im Quelltext in der lexikalischen Analyse ein Bezeichner gefunden wird, wird für ihn ein Eintrag in der Symboltabelle erzeugt. Zum Zeitpunkt der lexikalischen Analyse können aber noch nicht alle Attribute gefüllt werden. Z.B. erfolgt in Pascal die Angabe des Typs bei der Variablen-Deklaration erst nach der Benennung des Bezeichners, so dass beim Erstellen eines Eintrags für einen Bezeichner während der lexikalischen Analyse der Typ noch nicht eingetragen werden kann.

- *Error Handler:* Während jeder Phase kann es zu einem oder mehreren Fehlern kommen. Ein Compiler, der bei dem ersten Fehler abbricht, wäre nicht sehr hilfreich. Daher muss jede Phase versuchen, mit einem Fehler umgehen zu können, damit soviel wie möglich übersetzt werden kann und somit möglichst viele Fehler auf einmal angezeigt werden können. Bei einem vergessenen Semikolon möchte man ein C-Programm nicht komplett neu übersetzten müssen, um dann erst das nächste vergessene Semikolon angezeigt zu bekommen. In der lexikalischen Analyse werden Fehler angezeigt, wenn Zeichen übrig bleiben, die kein Token der Sprache bilden. Ein großer Anteil der Fehler, die vom Compiler ermittelt werden können, werden während der Syntaxanalyse und der semantischen Analyse entdeckt.

In einem realen Compiler müssen die einzelnen Phasen nicht so fein unterteilt werden. Oft werden die Phasen in *Front End* und *Back End* aufgeteilt.

- Das *Front End* umfasst die Phasen, die hauptsächlich von der Programmiersprache abhängig und im wesentlichen unabhängig von der Zielplattform sind. Das *Front End* umfasst also die lexikalische Analyse, die syntaktische Analyse, die semantische Analyse, die Erzeugung der Symboltabelle bis hin zur Erzeugung von *Intermediate Code*, der als Schnittstelle zwischen *Front End* und *Back End* dient.
- Das *Back End* beinhaltet die Phasen, die abhängig von der Zielplattform sind. Es arbeitet auf der Zwischendarstellung, die die Ausgabe des *Front End* darstellt, ist damit also unabhängig von der Programmiersprache, die der Compiler übersetzen soll.

Gewöhnlich kann man so mit einem *Front End* für eine Eingabesprache mit verschiedenen *Back Ends* Compiler für verschiedene Zielplattformen erstellen. Umgekehrt ist der Weg nicht ganz so leicht, da die Sichtweisen verschiedener Programmiersprachen feine Unterschiede haben. Im Idealfall könnte man mit einem *Back End* für eine Zielplattform mit mehreren *Front Ends* Compiler für verschiedene Programmiersprachen erzeugen. Der *Intermediate Code* dient dabei als Schnittstelle zwischen *Front End* und *Back End*.

3.2 Inlining

Ein Programm wird üblicherweise modularisiert, d.h. man versucht, ein Problem in Teilprobleme zu zerlegen. Funktionen zu definieren und aufzurufen stellt ein natürliches Mittel zur Modularisierung in Hochsprachen dar. Für immer mehr Programmierer stellt heute eine Objektorientierte Sprache, wie z.B. das weit verbreitete und populäre Java von

Sun, den Einstieg in das Programmieren dar. Beim Objektorientierten Entwurf denkt man häufig nicht in erster Linie an die Laufzeit, wenn man sich für die Wahl einer Funktion entscheidet. So gilt es bspw. als unschön, Variablen in einer Klasse *public*, also global zu deklarieren. Stattdessen werden häufig sogenannte *Getter* und *Setter* für das Lesen und Schreiben von Membervariablen definiert. Überträgt man diese Gewohnheiten auf die Programmiersprache C, so wird es häufiger ähnliche Konstrukte geben, d.h. man kapselt die Daten in einer Struktur und stellt Funktionen zur Verfügung, die auf solch einer Struktur arbeiten. Dies hat einerseits den Vorteil, dass man Fehler besser lokalisieren kann, da nur die Memberfunktionen bzw. die Funktionen, die mit der Struktur arbeiten, die Daten verändern, andererseits hat man den Nachteil, dass es häufiger zu Funktionsaufrufen kommt.

Allgemein ist es nicht immer optimal, wie eine Funktion gewählt wird. Wenn eine Funktion zwar häufig auftritt, wie z.B. das Vertauschen zweier Werte, der Overhead für einen Funktionsaufruf aber größer ist als die Funktion selbst, ist es günstiger, den Rumpf der Funktion mit den ersetzten Parametern an Stelle des Funktionsaufrufs zu setzen. In anderen Fällen wird vielleicht eine Funktion häufiger mit den selben Parametern aufgerufen. Zieht man diese Funktionen zuerst herein, kann man anschließend evtl. längere Funktionen herausziehen bzw. das von anderen Optimierungen nicht ausgeschöpfte bzw. nicht ausschöpfbare Optimierungspotential in den Befehlssequenzen zu Funktionen herausziehen.

3.3 Exlining

Beim Exlining versucht man, gleiche Befehlsfolgen in einem Programm zu finden, zu einer Funktion zu extrahieren und die Vorkommen der Befehlsmuster durch den Funktionsaufruf zur ausgelagerten Befehlssequenz zu ersetzen. Dabei darf die Semantik des Programms nicht verändert werden.

3.4 Burrows-Wheeler-Transformation

Die **B**urrows-**W**heeler-**T**ransformation wurde zuerst 1994 von Michael Burrows und David Wheeler unter dem Namen “A Block-Sorting Lossless Data Compression Algorithm” veröffentlicht. Da ich nicht die Reversibilität der Burrows-Wheeler Transformation nutze, möchte ich an dieser Stelle auch nur den Teil genauer erläutern, der für mein Ziel nützlich ist. Für eine ausführliche Darstellung der **B**urrows-**W**heeler-**T**ransformation siehe [MW94]. Bei der **B**urrows-**W**heeler-**T**ransformation geht man wie folgt vor: Ausgehend von einer zu kompaktierenden Zeichenfolge $\boldsymbol{x}^T = (x_0 x_1 \cdots x_{n-1})$ macht man sich

zunutze, dass Folgen gleicher Zeichen sich z.B. durch Run-Length-Encoding- Verfahren gut kompaktieren lassen. Man bildet zunächst eine $n \times n$-Matrix $M = (m_{i,j}) : \ i, j \in \{0, 1, \ldots, n-1\}$ Dabei enthält die i-te Zeile der Matrix die um i nach links rotierte Zeichenfolge $\boldsymbol{row_i}^T = (x_i x_{i+1} \cdots x_{n-1} x_0 \cdots x_{i-1})$. Jede im Text an Position i vorkommende Zeichenfolge beginnt so in Zeile i, d.h. jede im Text vorkommende Zeichenfolge kommt in irgendeiner Zeile vor. Anschließend werden die Zeichenfolgen $\boldsymbol{row_i}^T$ sortiert. Ergebnis des Sortierens ist es, dass alle im Text vorkommenden, gleichen Zeichenfolgen in aufeinander folgenden Zeilen auftreten. Diese Eigenschaft wird für das im Algorithmus zum Auffinden von Exline-Komponenten (siehe Abschnitt 4.2.1) beschriebene Suchen gleicher Befehlsfolgen vorteilhaft sein.

3.5 Low Level Intermediate Representation

Hier wird ein kleiner Überblick über die Klassen der **LLIR** [Eck01] gegeben, die der Implementierung des Exlining zugrunde liegen. Es handelt sich also nicht um eine vollständige Beschreibung der **LLIR**, deren Funktionsumfang an dieser Stelle den Rahmen sprengen würde. Bei der **LLIR** handelt es sich um eine Sammlung von Klassen, die nicht nur, wie der Name vermuten lässt, eine Zwischendarstellung eines Programms repräsentieren, sondern auch zahlreiche Funktionen auf dieser Datenstruktur zur Verfügung stellt. Dabei werden das Parsen von Input-Files, das Erzeugen der entsprechenden Objekt-Hierarchie, Lebenszyklus-Analysen und das Schreiben entsprechender Ausgabedateien unterstützt.

Die Hierarchie einer **LLIR**-Datei kann dabei als Instanzen von Klassen, die Funktionen, Basisblöcke, Instruktionen, Operationen und Register repräsentieren beschrieben werden (siehe Abbildung 3.3)

Im wesentlichen hat man als Container für die weiteren Klassen die *LLIR*-Klasse. Sie enthält mindestens eine Instanz der Klasse *LLIR_Function*, die eine Funktion repräsentiert. Ein *LLIR_Function*-Objekt wiederum enthält mindestens ein *LLIR_BB*-Objekt, das einen Basisblock repräsentiert. Ein Basisblock besteht seinerseits aus *LLIR_Instruction*-Objekten, die Mikro-Instruktionen entsprechen. Ein *LLIR_Instruction*-Objekt beinhaltet ein oder mehrere *LLIR_Operation*-Objekte, die Mikro-Operationen entsprechen, die parallel ausgeführt werden können. Jede Operation verweist auf *LLIR_Parameter*-Objekte, von denen es vier verschiedene Typen gibt:

1. Es kann sich um ein *LLIR_Register* handeln, wobei es je Funktion maximal eine Instanz von *LLIR_Register* als Repräsentant desselben Registers gibt. Verschiedene Operationen innerhalb einer Funktion, die dasselbe Register benutzen, verweisen auf die selbe Instanz eines *LLIR_Register*-Objektes.

2. Es kann sich um eine Konstante handeln. In diesem Fall beinhaltet das *LLIR_Parameter*-Objekt den Wert der Konstante als *int*.

3. Es kann sich um einen Operator handeln, der z.B. einen Postinkrement-Operator repräsentiert, der einen Parameter nach dem Lesen für eine Operation und nach Ausführen der Operation um eins erhöht.

4. Schließlich kann es sich um ein Label handeln, das durch seine Zeichenkette in *LLIR_Parameter* repräsentiert wird.

Als Basisklasse haben alle gerade aufgeführten Klassen noch die Klasse *LLIR_TaggedElement*, das mit verweisen auf Instanzen der Klasse *LLIR_Pragma* jedem Objekt Pragmas zuordnen kann. Die Pragmas können neben der Speicherung von *.pragma*-Anweisungen für beliebige Kommentare, die man einem Objekt zuordnen möchte, genutzt werden. Die Implementierung des Exlining arbeitet hauptsächlich auf einer Liste der *LLIR_Operation*-Objekte, die die Befehle eines zuvor in die **LLIR** eingelesenen Assembler-Programms repräsentieren. Jedes Objekt der **LLIR** kennt seinen Vater. So war es bei der Implementierung des Exlining einfach möglich, die von der **LLIR** zur Verfügung gestellten Befehle zur Bestimmung der Schleifentiefe eines Basisblocks zu nutzen.

3.6 SUIF

SUIF steht für Stanford University Intermediate Format. Dabei handelt es sich um ein Compiler System zur Unterstützung von Compilerforschung. Das primäre Ziel für dieses Compiler System besteht darin, möglichst einfach benutzbar zu sein und für alle Phasen des Compilers die gleiche Zwischendarstellung bzw. Intermediate Format zu benutzen, damit man möglichst jede Optimierung in jeder Phase zu jedem Zeitpunkt in Verbindung mit den anderen betrachten kann. Das **SUIF** System ist als eine Menge von Compiler-Läufen organisiert, die auf einen Kern aufsetzen, der das Intermediate Format definiert. Die Compiler-Läufe sind als einzelne Programme implementiert, die den Kern, der in der SUIF-Bibliothek enthalten ist, benutzen. So scheint **SUIF** für mich die richtige Wahl zu sein, um Inlining durchzuführen. Es gibt zwei verschiedene Entwicklungszweige bei **SUIF**. Einen älteren Zweig, der mit der Version 1.3.0.5 eine stabile Ausgangsbasis darstellt und einen Entwicklerzweig, der unabhängig davon weiterentwickelt wird, sich daher aber auch noch in der Testphase befindet und dementsprechend Fehler aufweist. Da ich für das Inlining keinen entscheidenden Vorteil in der aktuell entwickelten Version bzgl. der von mir benötigten Funktionalität sehe, wird der ältere Zweig gewählt Bald stellte sich heraus, dass beide Zweige den Nachteil haben, dass sie nicht den *storage class modifier inline* unterstützen. Auch wenn viele Compiler, wie z.B. auch der bekannte **GNU** C-Compiler diesen *storage class modifier* unterstützen und er damit de facto zum Standard gehört, gehört er

dennoch nicht zum kostenlos veröffentlichtem ANSI C Standard und wird zur Zeit auch von keinem anderen Werkzeug unterstützt, das in der Lage wäre, C-Code in eine Zwischendarstellung einzulesen. Außerdem müsste ein solches Werkzeug in der Lage sein, den eingelesenen Code dahingehend zu ändern, dass Funktionsdefinitionen und die zugehörigen Funktionsdeklarationen nach Bedarf mit dem *storage class modifier inline* versehen werden können oder nicht. Schließlich müsste der so modifizierte C-Code wieder ausgegeben werden können, um als Eingabe für einen C-Compiler dienen zu können. Alternativ erscheint eine direkte Durchführung des Inlining als zu zeitaufwendig. Das Werkzeug, das für das Inlining benötigt wird, muss also einerseits in der Lage sein, das Schlüsselwort *inline* zu verarbeiten, wenn es bereits im Quelltext vorkommt, und es andererseits vor Funktionsdeklarationen bzw. den zugehörigen Funktionsdefinitionen selbstständig nach einer bestimmten Bewertungsfunktion einfügen können. Die übliche Vorgehensweise, *inline* mittels *#define inline /* nichts */* und vorgeschaltetem Präprozessor verschwinden zu lassen und so für **SUIF** unsichtbar zu machen kann nur insoweit helfen, als dass **SUIF** so die Eingabe verarbeiten kann. Die Ausgabe, die **SUIF** anschließend erzeugt, enthält aber nicht mehr die ursprünglichen Informationen darüber, welche Funktionen vorher bereits *inline* deklariert waren, da diese Informationen auf diese Art bereits durch den Präprozessor herausgefiltert würden. Also muss man auf anderem Weg an diese Information gelangen.

3.7 bison/flex

Auf den ersten Blick erscheint es vielleicht einfach, nach dem Schlüsselwort *inline* zu suchen und die zugehörige Funktion zu bestimmen. Will man jedoch allgemein Funktionsdeklarationen und -definitionen finden, so kommt man mit einfachem Pattern-Matching nicht zum gewünschten Ziel. So suche ich nach einem Weg, möglichst schnell, ein meinen Anforderungen genügendes Werkzeug zu schaffen. Dabei sollen mir die im Folgenden beschriebenen **GNU** Utilities *Bison* und *Flex* helfen.

Will man herausfinden, welche Funktionen *inline* deklariert sind, so muss man den Source-Code wenigstens soweit parsen, dass man eine Funktionsdeklaration und eine Funktionsdefinition erkennen kann, und für die dabei gelesenen Funktionsnamen eine Symboltabelle anlegt. In der Symboltabelle benötigt man Einträge, wie Position der Deklaration bzw. Definition im Programmtext, eine Boolesche Variable, die darüber Auskunft gibt, ob die Funktion *inline* deklariert ist oder nicht. Bei dieser Aufgabe sollen mir der Scanner-Generator *Flex* und der Parser-Generator *Bison* helfen. *Flex* steht für “fast lexical analyzer generator”. Es ist ein Werkzeug, mit dem man sogenannte *Scanner* erzeugen kann. *Scanner* sind Programme, die Zeichenmuster in einem Text erkennen. Als Eingabe für *Flex* dient eine Beschreibung, die im Wesentlichen aus Paaren von regulären Ausdrücken und C-Code besteht. Der jeweilige C-Code wird immer in dem Fall ausgeführt, dass ein regulärer Ausdruck im Text gefunden wird. Als Ausgabe von *Flex* erhält man die Datei ’lex.yy.c’. Die

Datei 'lex.yy.c' enthält die Definition der Funktion *yylex*, die als Hilfsfunktion für den mit *Bison* erzeugtem Parser dient. Die Funktion *yylex* liest die Eingabe und liefert den nächsten *Token-Typ* zurück. So wird die Eingabe in *Tokens* zerlegt. Damit der mit *Bison* erzeugte Parser mit den Rückgabewerten der mit *Flex* generierten Funktion *yylex* etwas anfangen kann, müssen die möglichen *Token-Typen* in *Bison* definiert und *Flex* mittels einer von *Bison* erzeugten Header-Datei zur Verfügung gestellt werden.

Bei *Bison* handelt es sich um einen Mehrzweck Parser-Generator, der aus einer Beschreibung einer kontextfreien LALR(1) Grammatik [GJ98] ein C-Programm zum Parsen dieser Grammatik erzeugt. Eine Grammatik-Beschreibung für *Bison* besteht im Wesentlichen aus vier Abschnitten:

1. C-Deklarationen
2. Bison-Deklarationen
3. Grammatik-Regeln
4. Zusätzlicher C-Code

Sowohl *Bison* als auch *Flex* stehen unter der GNU General Public License. Mehr Informationen zu diesen freie erhältlichen Werkzeugen findet man auf den Webseiten des GNU Projekts.

Es gibt bereits eine Grammatik-Beschreibung für die Programmiersprache C bzw. C++ für *Bison* und eine Beschreibung für den nötigen Scanner in für *Flex* geeignetem Format von Jim Roskind. Dabei handelt es sich jedoch nicht um die Beschreibung eines voll funktionstüchtigen C-Parsers, da weder eine Symboltabelle noch die zugehörigen Gültigkeitsbereiche verwaltet werden. Außerdem setzt die Verwendung der Grammatik bereits vom Präprozessor verarbeitete Dateien voraus. Auf den ersten Blick erscheint es einfach, nach Funktionen zu suchen und zu betrachten ob sie *inline* deklariert sind oder nicht. Mit einer einfachen Mustererkennung kann man zwar das Schlüsselwort *inline* leicht finden, nicht aber allgemeine Funktionsdeklarationen, die ohne das Schlüsselwort auskommen. Man kommt also nicht trivial um die Verwendung einer Grammatik zur Erkennung von Funktionsdeklarationen und -Definitionen herum. Also passe ich die Aktionen der Regeln sowohl für den Scanner als auch für den Parser nach meinen Anforderungen an. Dies stellt sich aufwendiger dar als zunächst erwartet, da die gegebene Grammatik keine Verwaltung von Symboltabellen und Scopes beinhaltet. Damit der erzeugte Parser korrekt arbeiten kann, müssen Typdefinitionen und Bezeichner durch eine Symboltabelle unterschieden werden. Dies erfordert ein zeitaufwendiges Einarbeiten in die über 2000 Zeilen starke Grammatik-Beschreibung für *Bison* und die Anpassungen und Implementierungen für die Symboltabelle. **SUIF** kann genutzt werden, um weitere Informationen über die

Funktionen zu gewinnen. Damit **SUIF** die Eingabe verarbeiten kann, muss vorher das Schlüsselwort *inline* mittels Präprozessor entfernt werden.

3.8 Eigenschaften des TriCore

An dieser Stelle möchte ich einen kleinen Überblick über die von mir betrachtete Architektur, den TriCore von Infineon, geben. Dabei werden die Befehle genauer erläutert, die für das Exlining von Befehlssequenzen benutzt werden. Das Exlining selbst wird im Abschnitt 4.2.1 beschrieben.

Beim TriCore handelt es sich um einen 32-Bit Mikrocontroller-DSP Architektur, die für den Einsatz in eingebetteten Echtzeit-Systemen entworfen wurde. In einem kompakten programmierbaren Kern bietet die Architektur des Befehlssatzes die Echtzeitfähigkeiten eines Mikrocontrollers, stellt Befehle zur digitalen Signalverarbeitung zur Verfügung und hat eine RISC load/store Architektur.

Hier eine Liste über Eigenschaften des TriCore nach Herstellerangaben:

- 32-Bit Architektur
- 4-GByte virtueller oder physikalischer Daten-, Programm- und I/O-Adressraum
- 16-Bit und 32-Bit Befehle, um die Codegröße zu reduzieren
- Die meisten Befehle werden innerhalb eines Taktzyklus ausgeführt.
- Sprungbefehle benötigen 1 bis 3 Taktzyklen. Es wird eine Sprungvorhersage benutzt.
- Eine kurze Interrupt-Latenzzeit mit schnellem automatischen Kontextwechsel.
- Vorgesehene Schnittstelle für applikationsspezifische Co-Prozessoren
- Single Instruction Multiple Data (SIMD) Befehle (2 * 16-Bit oder 4 * 8-Bit Werte)
- ...

Bei einer RISC-Architektur wie dem TriCore hat man häufig keinen einfachen Sprung und Rücksprungbefehl, der als Seiteneffekt nur die Rücksprungadresse auf dem Stack ablegt. So gibt es zwar *call*- und *ret*-Befehle, diese haben aber den Seiteneffekt, dass ein Teil der Register sowie die beiden Status-Register $PCXI$ und PSW beim *call*-Aufruf gesichert und beim *ret*-Befehl wiederhergestellt werden. Dies schafft allgemein den Vorteil, dass nicht vor

jedem Sprung viele Register jeweils durch einzelne Befehle gesichert und nachher durch einzelne Befehle wiederhergestellt werden müssen. Für das Exlining bedeutet es, dass nicht jede Befehlsfolge einfach mit zwei solcher *call*- und *ret*-Befehle ausgelagert werden darf. An dieser Stelle wird ein kurzer Überblick über die Register, die dem TriCore zur Verfügung stehen, und deren Verhalten beim *call* bzw. *ret*-Aufrufen gegeben. Im Anschluss daran werden die wichtigsten Sprungbefehle des TriCore, die für das Exlining in Betracht kommen erläutert.

Register des TriCore

Die Architektur des TriCore besitzt jeweils 16 Datenregister $D0$ bis $D15$ sowie 16 Adressregister $A0$ bis $A15$. Darüberhinaus besitzt der TriCore zwei Statusregister $PCXI$ und PSW, die Informationen über vorige Berechnungen und Schutzinformationen enthalten. Schließlich gibt es noch den Programmzähler PC, der auf den nächsten auszuführenden Befehl zeigt. Die folgende Tabelle gibt einen kleinen Überblick:

Address	Data	System
A15 (Implicit Base Address)	D15 (Implicit Data)	PCXI
A14	D14	PSW
A13	D13	PC
A12	D12	
A11 (Return Address)	D11	
A10 (Stack Return)	D10	
A9 (Global Address reg.)	D9	
A8 (Global Address reg.)	D8	
A7	D7	
A6	D6	
A5	D5	
A4	D4	
A3	D3	
A2	D2	
A1 (Global Address reg.)	D1	
A0 (Global Address reg.)	D0	

Tabelle 3.1: Register des TriCore

Von vielen Befehlen auf dem TriCore gibt es sowohl eine 16 Bit- als auch eine 32 Bit-Variante. Benutzt ein Befehl das Datenregister $D15$, so gibt es dafür im allgemeinen eine 16 Bit-Variante. Gleiches gilt für Befehle, die das Adressregister $A16$ benutzen. Auch beim *call*-Befehl gibt es eine 16 Bit- und eine 32 Bit-Variante abhängig davon, wie weit PC-relativ gesprungen wird.

3.8.1 Sprungbefehle des TriCore

Um weitere Sprünge ermöglichen zu können, wird bei relativen Sprüngen jede Adresse mit 2 multipliziert, bevor sie 32Bit-vorzeichenerweitert auf den aktuellen Programmzähler (PC) addiert wird. Die kleineren 16 Bit Befehle bieten natürlich nicht so viel Platz, um lange Adressen anzugeben. Daher liegen ihre Zieladressen in einem Bereich ± 256 Bytes während die längeren 32 Bit Sprungbefehle mit ihren 24 Bit Adressinformationen Sprünge mit einer Reichweite von bis zu 16 MByte relativ zum aktuellen Programmzähler durchführen können.

Syntax	*call* disp24 (B)
	call disp8 (SB)
Beschreibung	Der Wert des Programmzählers PC wird auf den auf 32 Bit vorzeichenerweiterten zweifachen Wert von disp24 gesetzt. Die Ziel-Adresse liegt in einem Bereich ± 16 MByte relativ zum aktuellen Befehlszähler.
	Der Wert des Programmzählers PC wird auf den auf 32 Bit vorzeichenerweiterten zweifachen Wert von disp8 gesetzt. Die Ziel-Adresse liegt in einem Bereich ± 256 Bytes relativ zum aktuellen Befehlszähler.
Operation	$ret\_addr = PC + 4;$ $PC = PC + sign\_ext(disp24 * 2)$ Save upper context; $A[11] = ret\_addr;$
	$ret\_addr = PC + 2;$ $PC = PC + sign\_ext(disp8 * 2)$ Save upper context; $A[11] = ret\_addr;$

Der TriCore untestützt einen Schnellen Kontextwechsel. Dazu hat er Befehle, die gleichzeitig 8 Datenregister 6 Adressregister und 2 Statusregister in einer dafür vorgesehenen Liste, der CSA (Context Save Area), sichern können.

Beim *call*-Befehl wird der sogenannte Upper Context (s. Tabelle 3.2) in einer freien Context Save Area gesichert. Zum Upper Context gehören die Datenregister $D8$ bis $D15$, die Adressregister $A10$ bis $A15$ sowie das Statusword PSW und das Statusregister $PCXI$.

Will man noch weiter springen, so muss man absolut mit *calla* oder indirekt mit *calli* springen.

Lower Context	Upper Context
D7	D15 (Implicit Data)
D6	D14
D5	D13
D4	D12
A7	A15
A6	A14
A5	A13
A4	A12
D3	D11
D2	D10
D1	D9
D0	D8
A3	A11 (RA)
A2	A10 (SP)
Saved PC	PSW
PCXI (Link Word)	PCXI (Link Word)

Tabelle 3.2: Lower und Upper Context

Syntax	*calla* disp24 (B)
Beschreibung	Springt zur Adresse, die aus disp24 wie folgt berechnet wird: $PC = \{disp24\,[23\colon 20]\,, 7'b0000000, disp24\,[19\colon 0]\,, 1'b0\}$
Operation	$ret\_addr = PC + 4$; $PC = \{disp24[23 : 20], 7'b0000000, disp24[19 : 0], 1'b0\}$ Save upper context; $A\,[11] = ret\_addr$;

Syntax	*calli* Aa (RR)
Beschreibung	Springt zur Adresse, die durch den Inhalt von Register Aa bestimmt wird. Parallel zum Sprung wird der Upper Context in einer freien Context Save Area gesichert und die Rücksprungadresse in A11 gesichert.
Operation	$ret\_addr = PC + 4$; $PC = \{Aa[31 : 1], 1'b0\}$ Save upper context; $A\,[11] = ret\_addr$;

Wie bereits erwähnt, wird bei den *call*-Befehlen der Upper Context gesichert. Daher kann nicht für alle Befehlsfolgen, die mit Exlining herausgezogen werden sollen, ein *call* $[i|a]$-Befehl zusammen mit dem *ret*-Befehl benutzt werden. Ausgeschlossen sind alle Befehle, die

ein Register aus dem Upper Context definieren, das nach der Befehlsfolge noch benutzt wird. Für mehr Details der Exlining-Varianten auf dem TriCore, siehe Abschnitt 4.2.3.

Auf dem TriCore gibt es einen unbedingten Sprung, der sich nur die Rücksprungadresse merkt, ohne parallel den upper context zu sichern. Die Befehle dafür lauten *jl* bzw. *jla*.

Syntax	*jl* disp24 (B)
Beschreibung	Der Wert des Programmzählers PC wird analog zum *call* Befehl auf den auf 32 Bit vorzeichenerweiterten zweifachen Wert von disp24 gesetzt. Die Ziel-Adresse liegt in einem Bereich ± 16 MByte relativ zum aktuellen Befehlszähler.
Operation	$ret\_addr = PC + 4;$ $PC = PC + sign\_ext(disp24 * 2)$ $A[11] = ret\_addr;$

Syntax	*jla* disp24 (B)
Beschreibung	Springt zur Adresse, die aus disp24 analog zum *calla*-Befehl berechnet wird: $PC = \{disp24[23:20], 7'b0000000, disp24[19:0], 1'b0\}$
Operation	$A[11] = PC + 4;$ $PC = \{disp24[23:20], 7'b0000000, disp24[19:0], 1'b0\}$

Bei den *jump and link*-Befehlen wird der Upper Context also nicht automatisch mitgesichert.

Mit dem nun beschriebenen *ji*-Befehl kann man wieder indirekt an die Rücksprungadresse zurückspringen.

Syntax	*ji* Aa (RR)
Beschreibung	Springt indirekt an die Adresse, die durch Adressregister Aa bestimmt wird: PC = A[a][31:1], 1'b0
Operation	$A[11] = PC + 4;$ $PC = \{disp24[23:20], 7'b0000000, disp24[19:0], 1'b0\}$

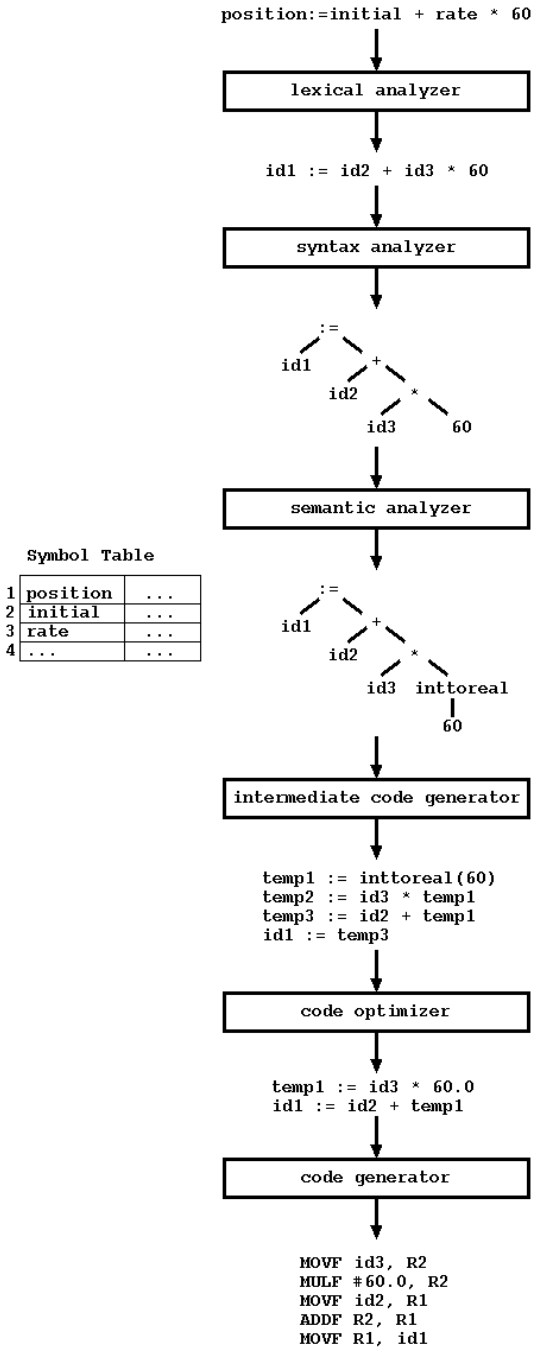

Abbildung 3.2: Übersetzung einer Befehlszeile

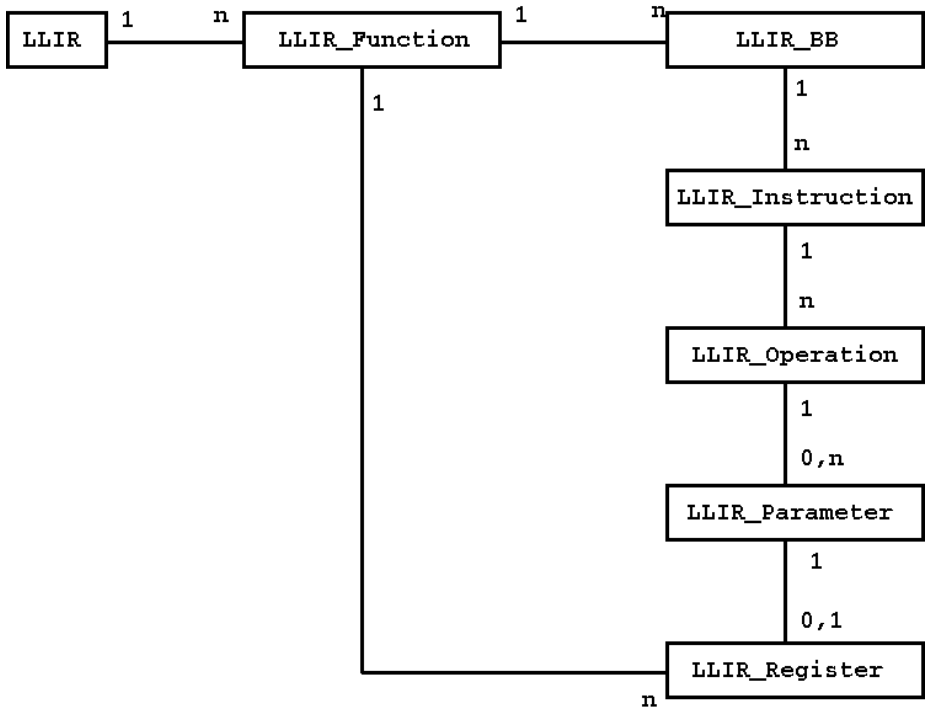

Abbildung 3.3: Low Level Intermediate Representation,
Quelle: Beschreibung der **LLIR**-Klassen [Eck01]

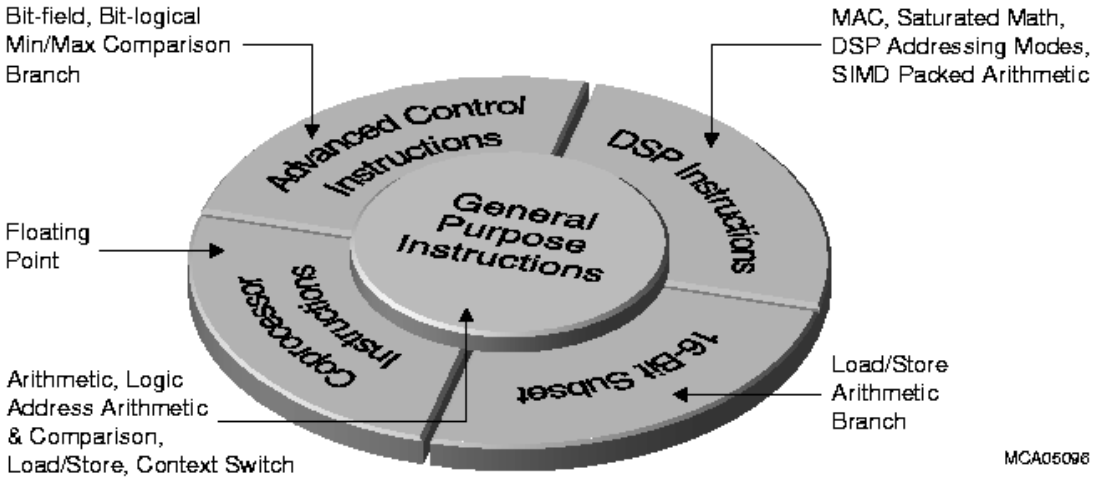

Abbildung 3.4: TriCore Overview

Kapitel 4

Algorithmus zur Codegrößenreduktion

Nun sollten die nötigen Grundlagen für die folgenden Abschnitte über Inlining und Exlining vorhanden sein. Zuerst ein kleiner Überblick, über die hier verwendete *Toolchain* (siehe Abbildung 4.1).

1. *Präprozessor:* Als Eingabe dient eine C-Quelldatei, die hier Programm.c genannt wird. Der Präprozessor sorgt dafür, dass die Ausgabe *Programm.preprocessed.c* keine Kommentare, Makros oder sonstige Definitionen, sondern reines C enthält.

2. *Inliner:* Der Inliner, der noch im Abschnitt 4.1.1 genauer beschrieben wird, liest die Datei *Program.preprocessed.c* und fügt vor bestimmte ausgewählte Funktionsdeklarationen bzw. -definitionen das Schlüsselwort *inline* ein.

3. *Compiler:* Der Compiler erhält als Eingabe die vom Inliner erzeugte Datei *Programm.inlined.c* und führt das Inlining für die entsprechend markierten Funktionen durch. Im Zusammenhang damit werden auch weitere Optimierungen durchgeführt, die durch das Inlining begünstigt werden sollen. Die Ausgabe ist die Assembler-Datei *Programm.inlined.s*.

4. *gcc2llir:* Das von Robert Pyka erstellte Werkzeug *gcc2llir* liest die Datei *Programm.inlined.s* ein und erzeugt daraus die Datei *Programm.inlined.llir*, die die **LLIR**-Repräsentation des Programms enthält (zur **LLIR** siehe Abschnitt 3.5).

5. *Exliner:* Der Exliner, der noch im Abschnitt 4.2.1 genauer beschrieben wird, benutzt die zuvor erzeugte **LLIR**-Darstellung des Programms, findet Gruppen gleicher Befehlsfolgen, wenn welche vorhanden sind, und lagert diese ggf. zu Pseudo-Funktionen aus. Das Ergebnis kann in die Datei *Programm.exlined.llir* ausgegeben werden.

6. *llir2gcc:* In der tatsächlichen Implementierung erzeugt der *Exliner* direkt die Datei *Programm.exlined.s*, da das Exlining als Post-Pass Optimierung hier das letzte Glied in der *Toolchain* darstellt und die **LLIR**-Repräsentation nicht weiter benötigt wird. Hier wird das Werkzeug *llir2gcc* explizit erwähnt, um den Ablauf der *Toolchain* zu verdeutlichen. *llir2gcc* konvertiert, wie der Name vermuten lässt, die **LLIR**-Repräsentation *Programm.exlined.llir* nach *Programm.exlined.s*.

7. *Assembler / Linker:* Hier kommt wie bereits als *Präprozessor* und *Compiler* der **GNU** C-Compiler für den TriCore von HiTec zum Einsatz. Aus der Datei *Programm.exlined.s* werden *Programm.exlined.o* bzw. *Programm.exlined.out* erzeugt.

Später kann man sich mit dem Werkzeug *tricore-size* die Größe der übersetzten Programme anzeigen lassen. Die Korrektheit sowie die Laufzeit der von den In- und Exlining-Stufen bearbeiteten Programme lassen sich mit einem Simulator (*tricore-tsimb*) und dem Debugger (*tricore-gdb*) überprüfen. Mehr zu den Ergebnissen in Abschnitt 5.

4.1 Algorithmus zum Inlining

4.1.1 Algorithmus zur Wahl von Inline-Funktionen

Auf den ersten Blick scheint es wenig Erfolg versprechend zu sein, mit Inlining, also mit dem Ersetzen eines kurzen Funktionsaufrufs durch den Inhalt bzw. den Parameter-ersetzten Rumpf der Funktion, Platz sparen zu können.

Wie bereits in Kapitel 2 erwähnt, haben sich frühere Studien schon mit Inlining beschäftigt, bisher aber nur aus dem Blickwinkel der Laufzeitoptimierung.

Dort wurden aber bereits Situationen aufgeführt, in denen Inlining auch Platz sparen kann. Das Inlining kann für die Platzoptimierung dann sinnvoll sein, wenn die hereingezogene Funktion zwar größer als der Overhead für einen Funktionsaufruf ist, das Inlining aber andere statische Optimierungen begünstigt, so dass insgesamt der Code durch das Inlining kleiner werden kann. Es ging hier nicht darum, das Inlining neu zu erfinden, da Inlining eine bekannte Technik ist. So unterstützt der TriCore-GCC von HiTec bereits die Compileroption *-finline-functions*, mit der Funktionsaufrufe, die nach der Heuristik des GCC lohnenswert erscheinen, durch den durch die eingesetzten Parameter modifizierten Rumpf der Funktion ersetzt werden. Untersuchungen mit einigen Beispielprogrammen haben jedoch gezeigt, dass im Allgemeinen Programme, die mit dieser Option des GCC übersetzt werden, größer werden als ohne diese Option übersetzte Programme. Dies entsprach auch

den Ergebnissen, die andere Studien über das Inlining gehabt haben. Die mit der Option *-finline-functions* übersetzten Programme blieben auch nach einem anschließend durchgeführtem Exlining meist größer, als ohne diese Optimierung übersetzte Programme. Dies soll hier nicht heißen, dass die Heuristik des GCC schlecht ist, sondern nur zeigen, dass die Heuristik nicht den Platz optimiert und folglich hier nicht benutzt werden kann.

Als weiteres Ergebnis der anderen Analysen hatte sich herausgestellt, dass man nicht präzise vorhersagen kann, wie sich das Inlining auf die später folgenden Optimierungen auswirkt. Daraus folgte einerseits, dass eine Heuristik gewählt werden musste, da es keine effiziente exakte Lösung gab [CMCWH92], andererseits durfte die Heuristik nicht der des GCC entsprechen, da diese allgemein zu größerem Code führte. Der GCC bietet aber nicht nur die Option *-finline-functions*, sondern erlaubt auch das Schlüsselwort *inline* vor Funktionsdeklarationen bzw. Funktionsdefinitionen, mit dem man einzelne Funktionen markieren kann. So bietet es sich an, das Inlining auf die Art zu realisieren, dass Funktionen durch die Heuristik ausgewählt werden und anschließend der Sourcecode entsprechend modifiziert ausgegeben wird, d.h. man benötigt ein Werkzeug, das die Funktionen im C-Quellcodes lokalisiert, bewertet, das Schlüsselwort *inline* vor eine evtl. vorhandene Deklaration und die Definition der Funktion schreibt, und den so modifizierten Sourcecode als Ergebnis liefert. Die so modifizierte C-Quelldatei soll dem GCC als Eingabe dienen können. Zu diesem Zweck arbeitete ich mich zunächst in **SUIF** ein (siehe Abschnitt 3.6). Wie dort bereits beschrieben bot **SUIF** zwar eine sehr umfassende, aber nicht die für die Betrachtung des Inlining notwendige Funktionalität, zumal der C-Code auch ohne die Wahl jeglicher Optimierungsphase durch **SUIF** modifiziert wurde, was in einigen Fällen zu einem größeren übersetzten Programm führte, so dass selbst eine Erweiterung von **SUIF** um das Schlüsselwort *inline* nicht das gewünschte Ergebnis gebracht hätte. Schließlich sollte Inlining betrachtet werden und nicht die Modifikation von C-Code durch **SUIF** bewertet werden. Dies sei hier nur aufgeführt, um zu motivieren, warum die Entscheidung, ein neues C-Frontend mit minimaler Funktionalität zu implementieren, getroffen wurde. Nur wenn die C-Quelltexte sich nur um die eingefügten Schlüsselworte *inline* unterscheiden, sind die Unterschiede, die durch ein vorangestelltes Inlining entstehen messbar.

Wie bereits in Abschnitt 3.7 erwähnt, gibt es eine kommentierte C-Grammatik für *Bison* von Jim Roskind, die für wissenschaftliche Zwecke benutzt und verändert werden darf, sowie eine kommentierte Beschreibungsdatei für den Scannergenerator *Flex*.

Damit die Grammatik benutzt werden konnte, musste man die Regeln noch so erweitern, dass eine Symboltabelle verwaltet wurde, damit einfache Bezeichner von Typdefinitionen unterschieden werden konnten. Nach jeder erkannten Regel wurde zum aktuell erkannten Symbol die Start- und End-Position in der eingelesenen Datei festgehalten. Handelte es sich bei dem Regel um eine Funktionsdefinition, so wurde zusätzlich zum Symbolnamen noch die Eigenschaft, dass es sich um eine Funktion handelt gespeichert. Zusätzlich dazu merkt man sich, ob die Funktion *static* und / oder *inline* definiert ist. Nach dem Erkennen einer Funktionsdefinition wird nach einem Symbol mit gleichem Namen, der möglichen

Funktionsdeklaration gesucht und die Eigenschaft *Funktion* gesetzt. Die Start-Position der Funktionsdefinition bzw. Funktionsdeklaration bildet im Fall, dass die Funktion nicht *inline* deklariert war die Position des einzufügenden Schlüsselwortes *inline*. Bei bereits vorhandenem *inline* wird die entsprechende Position festgehalten, so dass das Werkzeug prinzipiell die Möglichkeit unterstützt, *inline* deklarierte Funktionen nicht *inline* zu deklarieren.

Im wesentlichen arbeitet der *Inliner* in drei Phasen:

1. *Parsen:* Parsen einer C-Datei, die hier *Programm.preprocessed.c* genannt wird. Dabei wird das C-Programm in eine Sequenz von Bereichen im Quelltext zerlegt. Für Funktionsdeklarationsbereiche und Funktionsdefinitionsbereiche merkt man sich die Position für ein vorhandenes oder ein mögliches *inline*.

2. *Setze Inline-Flags:* Nach erfolgreichem Parsen hat man nun die Möglichkeit, für bestimmte Funktionen die Eigenschaft *inline* zu setzen. Zur Bewertung dient in einer ersten Implementierung die Zahl der nicht leeren Zeilen des Rumpfes einer Funktion und die Zahl der Referenzen auf das Funktionssymbol. Dabei ist die Zeilenzahl parametrisierbar.

3. *Schreiben der modifizierten C-Datei:* Schließlich kann man die Sequenz der Codestücke gemäß den modifizierten Inline-Flags ausgeben und erhält *Programm.inlined.c*.

Auch wenn man im Allgemeinen nicht präzise sagen kann, wann es sich lohnt, eine Funktion *inline* zu deklarieren, so gibt es doch Fälle, die ein solches Vorgehen geradezu fordern:

- Funktionen, für die der Aufruf mehr Befehle umfasst, als die Funktion selbst. Dem wird versucht, mit dem Kostenmaß der nicht leeren Zeilenzahl des Rumpfes Rechnung zu tragen, was zugegebener Maßen sehr unpräzise ist. Die Implementierung eines komplexeren Analyse-Werkzeugs war neben dem Entwickeln des Exlining-Werkzeugs im Zeitrahmen der Diplomarbeit nicht mehr möglich.

- Funktionen, die *static* deklariert sind und höchstens einmal aufgerufen werden. So spart man den Funktionsaufruf-Overhead. Betrachtet man auf einmal den gesamten Quelltext, gilt dies auch für nicht explizit *static* deklarierte Funktionen. In der Implementierung der Heuristik wird dazu die Zahl der lookups auf ein Symbol während des Parsens in der Symboltabelle aktualisiert und anschließend in der Bewertung benutzt.

Es gibt aber auch Fälle, in denen Inlining nicht durchgeführt werden kann. Siehe dazu auch [CMCWH92]:

1. Bei einer variablen Zahl von Argumenten.
2. Wenn der Aktivierungsstack zu groß würde (bei Aufrufen aus rekursiven Funktionen).
3. Wenn der Rumpf der Funktion nicht bekannt ist (externe Funktionen).
4. Wenn an der Aufrufenden Stelle nicht feststeht, welche Funktion in Frage kommt (Aufruf mit Zeiger auf Funktion).

Die Implementierung des Inliners braucht diese Fälle nicht zu berücksichtigen, da diese Arbeit bereits vom anschließend aufgerufenen Compiler erledigt wird. Der vorletzte Fall wird dabei vom Inliner grundsätzlich nicht erzeugt, da nur Funktionen, deren vollständige Definition vorliegt, berücksichtigt werden.

4.1.2 Laufzeit des Inliners

Die Laufzeit des Inliners hängt im wesentlichen linear von der Eingabelänge ab, da *Bison* für die benutzte Grammatik lediglich einen Schiebe-Reduziere-Konflikt für den *if then else*-Fall erzeugt. Dieser Konflikt kann mit den bekannten Methoden aus [ASU88] beseitigt werden. Man muss natürlich noch das Einfügen und Suchen in der Symboltabelle berücksichtigen. In der ersten Implementierung ist dies als Liste implementiert worden, so dass insgesamt eine quadratische Laufzeit bzgl. der Programmlänge n auftritt. Mit der Implementierung der Symboltabelle als Hashtable und nach der Beseitigung des *if then else* Schiebe-Reduziere-Konflikts, kann der Overhead für das Parsen aber insgesamt auf $O(n)$ gebracht werden. Die Bewertung der Funktionen im Anschluss an das Parsen und das entsprechende Setzen des Inline-Flags kann bei der Implementierung der Symboltabelle mit einer Hashtable ebenfalls in Linearzeit abhängig von der Zahl der gefundenen Funktionsdefinitionen erfolgen. So kann der benötigte Zeitbedarf des Inliners als $O(n)$ bzgl. der Länge n des Programms angegeben werden.

4.1.3 Optimierungen, die vom Inlining profitieren

Wie bereits im Abschnitt 1 erwähnt, können andere bekannte Optimierungen vom Inlining profitieren:

- *Register Allocation:* Ersetzen von Parametern im Funktionsrumpf durch Variablen, die bereits vor dem ursprünglichen Funktionsaufruf in Registern gehalten werden, brauchen so nicht umkopiert, sondern können direkt vom modifiziertem Code benutzt werden.

- *CSE Elimination:* Gemeinsame Teilausdrücke, die sowohl in der ursprünglich aufgerufenen Funktion sowie vor dem Aufruf berechnet werden, können erkannt und eliminiert werden.

- *Constant Propagation:* Konstanten, die einer Funktion als Parameter übergeben werden, können direkt im Rumpf ersetzt werden. Dadurch kann es zu Vereinfachungen in Berechnungen kommen. Andere Ausdrücke können in der Folge selbst konstant und durch die berechnete Konstante ersetzt werden.

- *Dead Code Elimination:* Durch die Übergabe von Konstanten können bedingt ausgeführte Programmabschnitte evtl. als *Dead Code*, d.h. Code, der nicht ausgeführt werden kann, erkannt und eliminiert werden.

4.2 Algorithmus zum Exlining

4.2.1 Algorithmus zum Auffinden von Exline-Gruppen

Um gleiche Befehlsfolgen aus einem Programm zu Funktionen zusammenfassen zu können, müssen diese erst einmal gefunden werden.

Die Eingabe für den Algorithmus besteht aus der **LLIR** (siehe Abschnitt 3.5) des vom Compiler erzeugten Programms. Man betrachtet die Befehlsfolge als Zeichenfolge, wobei gleiche Befehle, d.h. Befehle, die sowohl im Opcode als auch in ihren Parametern gleich sind, durch gleiche Zeichen repräsentiert werden.

1. Wie bei der **B**urrows-**W**heeler-**T**ransformation wird zunächst eine $(n \times n) - Matrix$, $n = \sharp\,(Befehle)$ angelegt. In der i-ten Zeile, $i = 0, 1, \ldots, n-1$, steht die um i nach links rotierte Befehlsfolge. Um die Zeilen zu repräsentieren wird ein Vektor $\boldsymbol{t} = \begin{pmatrix} 0 \\ 1 \\ \vdots \\ n-1 \end{pmatrix}$ benutzt.

2. Sortiere die Matrix, d.h. die Verweise auf Zeilen der Matrix in Vektor $\boldsymbol{t}$. Anschließend liegen gleiche Befehlsfolgen in aufeinander folgenden Zeilen der Matrix bzw. werden durch aufeinanderfolgende Zeilen in $\boldsymbol{t}$ repräsentiert.

3. Suche Paare von Befehlsfolgen mit möglichst langem, gleichen Präfix. Bei dem Vergleich zweier Präfixe muss man beachten:

(a) Man darf nicht über Funktionsende-Befehle hinweg vergleichen, da solche Befehle nicht in Funktionen ausgelagert werden können.

(b) Ebensowenig werden Sprungziele und Sprünge innerhalb von Befehlsfolgen ausgelagert. Die Rückkehr von solchen Sprüngen lässt sich nicht allgemein berücksichtigen bzw. würde einen zu großen Overhead bedeuten. Daher werden bei der Sortierung gleiche Befehle, nach der der Eigenschaft, ob sie Sprung und ob sie Sprungziel sind sortiert. Die Probleme, die bei Sprüngen aus einem Präfix heraus auftreten, lassen sich evtl. durch ein vorher durchgeführtes Inlining umgehen.

(c) Es darf keine Überlappung im Programm vorkommen, d.h. ein Präfix darf nicht in den Anfang des anderen reichen. Da die Relation, ob zwei Präfixe sich an Stelle j überlappen nicht transitiv ist, kann nach dieser Eigenschaft nicht sortiert werden.

(d) Über vom Algorithmus als *benutzt* markierte Befehle darf nicht hinweg verglichen werden, da ein Befehl nur in einer auszulagernden Gruppe gleichzeitig vorkommen darf. Analog zur Sprungzieleigenschaft wird bei Gleichheit nach der Eigenschaft, ob sie benutzt sind sortiert.

Sind die aktuell verglichenen Präfixe bis Position k gleich und die bis jetzt angefallenen Kosten, d.h. der von den Befehlen im Präfix benötigte Platzbedarf größer als der benötigte Overhead zum Exlining der aktuellen Präfixe, so werden die Komponenten der k-Partition von i und j verschmolzen. Dabei wird ein Präfixpaar durch das Tripel (k, i, j) repräsentiert, wobei k der Zahl der Befehle des Präfixes entspricht, i und j entspricht den Zeilennummern in $\boldsymbol{t}$. Für alle Präfixlängen k werden Partitionen p_k der Zeilennummern in $\boldsymbol{t}$ verwaltet.

- $merge\left(p_k, find\left(p_k, i\right), find\left(p_k, j\right)\right)$

Im *merge*-Schritt werden dabei auch die Einsparpotentiale und die Zahl der Präfixe der k-Komponenten aktualisiert. Dabei merkt man sich die Komponente mit dem bisher größten Einsparpotential. Muss man den Vergleich zweier Zeilen i und j an Position k wegen Überlappung abbrechen, so muss Zeile i auch mit der auf j folgenden Zeile verglichen werden. In allen anderen Fällen braucht Zeile i nicht mit weiteren Zeilen verglichen werden.

4. Nun kann man einfach die teuerste k-Komponente benutzen, d.h. die zugehörigen Befehle werden als benutzt markiert.

5. Das ganze wiederholt man maximal $n - 2$ mal, da bei jedem benutzen einer Gruppe mindestens 2 Zeilen des Vektors $\boldsymbol{t}$ wegfallen. Dies wird solange wiederholt bis keine Gruppe mit positivem Kostensenkungspotential mehr übrig ist.

Unter 3 werden einige Bedingungen erwähnt, die man beim Vergleich zweier Zeilen der Matrix beachten muss.

Warum darf man nicht über Funktionsende-Befehle hinweg vergleichen (siehe 3a)? Wie in der Grafik 4.2 beschrieben, darf man zwei Präfixe nicht über einen Befehl vergleichen, der Funktionsende ist. In der Grafik wird gezeigt, wie sich der Ablauf ändert, wenn man versucht, eine Befehlsfolge, wie M U S X T E R mit Exlining aus einer Funktion f zu einer Funktion $.ex$ herauszuziehen. Dabei entspricht X dem Funktionsende-Befehl. Wie man sieht, würde nach durchgeführtem Exlining nach den Befehlen M U S nicht H U R Z sondern D E F ausgeführt.

Warum darf man nicht über ein Sprungziel hinweg vergleichen (siehe 3b)? Wie in der Grafik 4.3 zu erkennen, würde nach Exlining von M U S .l:L E bei einem Sprung aus $.f$: zu $.l$: nach der Befehlsfolge L E nicht, wie vor dem Exlining, D E F und anschließend H U R Z ausgeführt, sondern nur L E H U R Z.

Warum sortiert man nach der Eigenschaft Sprungziel (siehe 3b)?

Man sortiert Vektor $\boldsymbol{t}$, um anschließend nur aufeinanderfolgende Zeilen der Matrix, die durch den sortierten Vektor repräsentiert werden, vergleichen zu müssen. Angenommen, die Eigenschaft **Sprungziel** wäre keine Sortiereigenschaft:

i)	M	U	S	T	E	R	A
j)	M	U	**S**	T	E	R	B
k)	M	**U**	S	T	E	R	C
l)	M	U	S	T	E	R	D
m)	M	U	S	T	E	R	E

Ohne Sortierung nach der Eigenschaft **Sprungziel** müsste zuerst Zeile i) mit Zeile j), dann mit Zeile k) und schließlich mit Zeile l) verglichen werden, um **MUSTER** zu finden.

Mit der Sortiereigenschaft **Sprungziel** ergibt sich folgendes Bild:

k)	M	**U**	S	T	E	R	C
j)	M	U	**S**	T	E	R	B
i)	M	U	S	T	E	R	A
l)	M	U	S	T	E	R	D
m)	M	U	S	T	E	R	E

Nun müssen nur noch jeweils benachbarte Zeilen verglichen werden, um die Befehlsfolge **MUSTER** zu finden.

Betrachten wir nun mögliche Überlappungen (siehe 3c). Angenommen, man hat eine Befehlsfolge, wie

A B A B A B A B D

Wenn anschließend der Vektor $\boldsymbol{t}$ sortiert wird, hat man ab einer bestimmten Zeile folgendes Bild:

	0	1	2	3	4	5	6	7	8
	A	B	A	B	A	B	A	B	D
$\boldsymbol{x}$	A	B	A	B	A	B	A	B	D
$\boldsymbol{y}$	A	B	A	B	A	B	D		
$\boldsymbol{z}$	A	B	A	B	D				

Beim Vergleich der aufeinanderfolgenden Zeilen könnte man zunächst vermuten, dass man drei mal die Befehlsfolge A B A B zu einer Funktion herausziehen könnte. In drei aufeinanderfolgenden Zeilen werden Präfixe $\boldsymbol{x}$, $\boldsymbol{y}$ und $\boldsymbol{z}$ repräsentiert, deren Position im Quelltext hier ist:

$$\overbrace{AB\underbrace{AB}_{\boldsymbol{y}}}^{\boldsymbol{x}}\overbrace{\underbrace{AB}_{\boldsymbol{y}}AB}^{\boldsymbol{z}}D$$

Nun kann man sofort erkennen, dass man nur $\boldsymbol{x}$ und $\boldsymbol{z}$ gleichzeitig herausziehen kann, da sich $\boldsymbol{x}$ und $\boldsymbol{y}$ überlappen aber auch $\boldsymbol{y}$ und $\boldsymbol{z}$. Es wäre vorteilhaft, könnte man wie nach der Eigenschaft Sprungziel auch nach der Eigenschaft Überlappt sortieren. Wie man an dem kleinen Beispiel leicht sehen kann, ist die Relation $\boldsymbol{x}$ überlappt $\boldsymbol{y}$ nicht transitiv:

$\boldsymbol{x}$ überlappt $\boldsymbol{y}$, $\boldsymbol{y}$ überlappt $\boldsymbol{z}$ aber $\boldsymbol{x}$ überlappt nicht $\boldsymbol{z}$.

Daher ist die Eigenschafft Überlappt keine Halbordnung und man kann nicht danach sortieren. Glücklicherweise kommen derartige Überlappungen nicht sehr häufig vor. Erwähnenswert wären hier höchsten mögliche NOP-Blöcke, d.h. mit leeren Befehlen gefüllte Bereiche, die aber mit einem einmaligen Durchlauf erkannt, markiert und damit von weiteren langen Vergleichen ausgeschlossen werden können.

Kommen wir nun zur Eigenschaft Benutzt (siehe 3d). Befehle, die einmal durch den Algorithmus zum Exlining ausgewählt werden, dürfen nicht mehrfach in verschiedenen Präfixgruppen landen. Angenommen, man hätte bereits zehn mal die Befehlsfolge **STERBE** gefunden und zehn mal die Folge **STERCE**. Beide Präfixgruppen mit je zehn Präfixen würden von Algorithmus als **benutzt** markiert, da sie nicht mehr in den folgenden Zeilen der Matrix benutzt werden dürften:

j)	M	U	**S**	**T**	**E**	**R**	**C**
k)	M	U	**S**	**T**	**E**	**R**	**B**
i)	M	U	S	T	E	R	A
l)	M	U	S	T	E	R	D
m)	M	U	S	T	E	R	E

Mit der Sortierung müssen nur aufeinanderfolgende Zeilen verglichen werden, um noch benutzbare Vorkommen von MUSTER zu finden.

4.2.2 Laufzeit

Kommen wir nun zur Laufzeit des vorgestellten Algorithmus. Sei die Eingabegröße N die Zahl der Befehle und koste der Vergleich von zwei Befehlen $O(1)$. Dann kostet das Sortieren der Verweise in $\boldsymbol{t}$ $O(N^2 \log N)$ Befehlsvergleiche. Nun müssen jeweils 2 aufeinander folgende Zeilen in der sortierten Matrix verglichen werden, also $O(N^2)$ viele Befehlsvergleiche und *merge*- und *find*- Operationen auf den k-Partitionen, die jeweils $O(\log N)$ Vergleichsoperationen benötigen. Mit Pfadkompression [Güt92, Seiten 134-140] benötigen n $merge-$, $find-$Operationen $O(n \cdot G(n))$, (Es gilt $G(n) \leq 5,\ n \leq 2^{65536}$) [AHU74] Beim Vergleich zweier Zeilen kann man die Präfixkosten aktuell halten. Bei der *merge*-Operation kann man die Zahl der Präfixe in der k-Komponente aktualisieren, und abhängig davon und abhängig vom benötigten maximalen Overhead für einen Funktionsaufruf die Kosten einer Komponente in $O(1)$ aktualisieren.

Die Kosten bzw. das Kostensenkungspotential $c(K_{k,n,i})$ einer Komponente $K_{k,n,i},\ k \cong \sharp Befehle,\ n \cong \sharp Präfixe$ berechnen sich wie folgt: $c(K_k) = n \cdot (c_p - c_{call}) - (c_p + c_{ret})$ Dabei seien die Kosten des aktuellen Präfixes c_p, $c_p = \sum_{1 \leq j \leq k} c(I_j)$, $c(I_j)$ entspricht den Kosten des j-ten Befehls des Präfixes, c_{call} entspricht den (teuersten) Kosten, die für einen Funktionsaufruf anfallen. c_{ret} entspricht den Kosten der zugehörigen Rückkehr von der mittels Exlining herausgezogenen Funktion.

Insgesamt hat man $O(N^2 \log N)$ Vergleiche zur Berechnung des Kostensenkungspotentials der k-Komponente, die das größte Einsparpotential verspricht. Da bei der Benutzung einer Komponente mindestens 2 Zeilen der Matrix anschließend nicht mehr benutzt werden können, da sie bereits von einer Komponente mit mindestens zweifachem Vorkommen der selben Befehlsfolge benutzt wurden, kann man dies höchstens $O(N)$ mal durchführen, bis keine k-Komponente mehr zu benutzen ist. Also maximal $O(N^3 \log N)$. Diese Aussage stimmt nur für die Annahme, dass man Überlappungen von gleichen Befehlsfolgen in aufeinander folgenden Zeilen vernachlässigen kann. Dies ist dann der Fall, wenn man zu Beginn in $O(N)$ Schritten Blöcke von n gleichen Befehlen, wie sie z.B. als NOP-Blöcke vorkommen können, als benutzt markiert und diese, falls erlaubt, zu Schleifen auslagert.

4.2.3 Exliningvarianten auf dem TriCore

Im Folgenden werden verschiedene Möglichkeiten, Befehlsfolgen in Funktionen herauszuziehen und die dafür notwendigen bzw. hinreichenden Bedingungen beschrieben.

Exlining mit *call* und *ret*

Da jeweils der Upper Context (siehe Tabelle 3.2), der zum Zeitpunkt des *call*-Aufrufs gesichert wird, beim *ret*-Befehl wiederhergestellt wird, dürfte nur eine stark eingeschränkte Menge von Befehlsfolgen mit *call* und *ret* zu echten Funktionen herausgezogen werden. Die von den 16 Bit-Varianten der Befehle implizit benutzten Register $A15$ und $D15$ gehören zum Upper Context, daher werden diese Register bevorzugt verwendet. Außerdem gehört auch das Statuswort PSW zum Upper Context, so dass zwischen einem Befehl, der das Statuswort setzt und einem Befehl, der das Statuswort benutzt, kein *ret*-Befehl liegen darf. Somit ist die Zahl der Befehlsfolgen, die durch das Exlining mit *call*- und *ret*-Befehlen ausgelagert werden können, zusätzlich eingeschränkt.

Ideal wäre ein unbedingter Sprung, der sich nur die Rücksprungadresse merkt, ohne parallel den Upper Context zu sichern. Dafür gibt es auf dem TriCore den *jl*- bzw. den *jla*-Befehl.

Allgemein darf mit den Befehlen *call* und *ret* nur dann ausgelagert werden, wenn innerhalb der auszulagernden Befehlsfolge kein Register aus dem Upper Context definiert wird, das nach der auszulagernden Befehlsfolge benutzt wird, ohne vorher erneut definiert worden zu sein. Die neuen Ergebnisse in den Registern aus der Upper Context Save Area gehen beim *ret*-Befehl verloren, da er die alten Werte wiederherstellt.

Man kann im wesentlichen drei Fälle unterscheiden. Dabei heiße die auszulagernde Befehlsfolge *exseq*.

1. Wird kein Register aus dem Upper Context in *exseq* definiert, kann man *exseq* mit *call* und *ret* auslagern.

2. Wird mindestens ein Register aus dem Upper Context in *exseq* definiert, aber innerhalb dieser Befehlsfolge nicht mehr benutzt, kann man davon ausgehen, dass es nach *exseq* mindestens einen Pfad auf dem Kontrollflußgraphen gibt, auf dem das definierte Register benutzt wird, da man sonst eine dd-Anomalie (siehe [Rie97, Seiten 316-327]) hätte, d.h. ein Wert wird in ein Register geschrieben, ohne dass der alte Wert vorher gelesen wurde. Unter der Voraussetzung, dass es keine solche dd-Anomalie gibt, kann man eine derartige Befehlsfolge nicht mit *call* und *ret* auslagern.

3. Werden alle Register aus dem Upper Context, die innerhalb von *exseq* definiert werden, auch innerhalb *exseq* gelesen und nach *exseq* auf keinem möglichen Pfad im Kontrollflussgraphen gelesen bevor sie erneut definiert werden, so läßt sich *exseq* mit *call* und *ret* auslagern.

Exlining mit *jla* und *ji*

Wenn die auszulagernde Befehlsfolge *exseq* ein Register aus dem Upper Context (s. Tabelle 3.2) definiert, das erst später benutzt wird, kann man, wie beschrieben, nicht mit *call* und *ret* auslagern. Es gibt aber eine weitere Möglichkeit, eine Befehlsfolge mit einem Sprungbefehl und einem Rücksprungbefehl auszulagern. Der TriCore besitzt dazu die Befehle *jla* (Jump and Link Absolute) und *ji* (Jump Indirect). Der erste der beiden Sprungbefehle springt nicht nur zu einem angegebenen Label sondern speichert auch die Adresse des nächsten Befehls im Adressregister $A11$. Mit dem Befehl *ji %A11* kann man also wieder indirekt an die Rücksprungadresse hinter den *jla*- Befehl springen. Im Gegensatz zum *call*-Befehl werden hierbei keine Register aus dem Upper Context gesichert. Daher kann man diese Methode auch dann nutzen, wenn man in *exseq* Register aus dem Upper Context definiert, die nach der Ausführung von *exseq* noch benötigt werden, oder das Prozessor-Statuswort (PSW) setzt. Da man im allgemeinen davon ausgehen kann, dass sich im Adressregister $A11$ bereits eine Rücksprungadresse befindet, die noch benötigt wird, muss man vor dem Ausführen des *jla*-Befehls den Inhalt aus $A11$ sichern. Dies kann auf drei verschiedene Arten geschehen:

1. In einem noch freien Adressregister z.B. $A7$.
2. In einem noch freien Datenregister z.B. $D14$.
3. An einer bestimmten Adresse im Hauptspeicher.

An einem kleinen geläufigen Beispiel möchte ich zunächst zeigen, wie man Exlining mit den eben beschriebenen Befehlen durchführen kann.

Eine einfache C-Funktion zum Vertauschen zweier Arrayelemente sei hier kurz angegeben:

```
void test__swap(int a[], int l, int r){
    int i;
    i = a[l];
    a[l] = a[r];
    a[r] = i;
}
```

Übersetzt man die C-Funktion mit dem tricore-gcc nach Assembler-Code, so ergibt sich innerhalb der Funktion vier mal die folgende Befehlssequenz:

```
mov   %d0, %d15
sha   %d15, %d0,2
mov   %d0, %d15
ld.w  %d15,[%a10] 12
add   %d1, %d0, %d15
```

Da das Register $D15$ aus dem Upper Context definiert und gelesen wird, sieht man nicht sofort, ob man mit *call* und *ret* Exlining durchführen kann. Außerdem verändert der add-Befehl den Inhalt des Statusworts PSW, das ebenfalls zum Upper Context gehört. Angenommen der Inhalt des Registers $D15$ und des Statusworts PSW wird nach dieser Befehlssequenz nicht mehr benötigt, so kann man die Befehlssequenz *exseq* jeweils durch den nun folgenden Befehl ersetzen:

```
call .EX_L1
```

Einmal muss die Befehlsfolge *exseq* natürlich noch im Assemblercode vorkommen und mit *ret* abgeschlossen werden, um an die beim *call*-Aufruf gespeicherte Rücksprungadresse zu gelangen.

```
    ...
.EX_L1:
    mov   %d0, %d15
    sha   %d15, %d0,2
    mov   %d0, %d15
    ld.w  %d15,[%a10] 12
    add   %d1, %d0, %d15
    ret
```

Würde der Inhalt des Registers $D15$ nach der Befehlsfolge *exseq* noch benötigt, z.B. weil eine kürzere Befehlsfolge ohne den *add*-Befehl sehr häufig vorkommt, müsste man eine der drei anderen Varianten benutzen, um *exseq* mittels *jl* und *ji* auszulagern. Das Adressregister $A11$ muss dazu, wie folgt, vor dem Sprung mit *jl* gesichert werden:

zu 1) Angenommen das Adressregister $A7$ ist frei, was es in diesem Fall tatsächlich war, so kann man wie folgt den Inhalt vom Adressregister $A11$ im Daten $A7$ sichern:

```
mov.aa %a7, %a11
jl     .EX_L1
mov.aa %a11, %a7
```

zu 2) Angenommen das Datenregister $D14$ ist frei, so kann man wie folgt $A11$ nach $D14$ sichern und nach dem Sprung wiederherstellen:

```
mov.d  %d14, %a11
jl     .EX_L1
mov.a  %a11, %d14
```

zu 3) Wenn kein Register mehr frei ist, aber ein Register in *exseq* definiert wird, bevor es gelesen wird, kann man es benutzen, um die Rücksprungadresse im Hauptspeicher zu sichern. Selbst wenn es kein freies Register mehr gibt und in *exseq* ein Register aus dem Upper Context definiert wird, das außerhalb von *exseq* gelesen wird, so bleibt schließlich noch die Möglichkeit den Inhalt des Registers $A11$ an einem festen Platz im Speicher zu sichern. Nach dem Sprung mit *jli .EX_L1* darf man das Adressregister $A11$ nur mit Hilfe eines noch freien Adressregisters wiederherstellen. Dabei verändern die Befehle *movh.a*, *st.a* und *ld.a* den Status nicht.

```
movh.a %a15, HI:EXMEM
st.a   [%a15] LO:EXMEM, %a11
jl     .EX_L1
...
movh.a %a15, HI:EXMEM
ld.a   %a11, [%a15] LO:EXMEM
```

Nun kann die ausgelagerte Befehlsfolge nicht mehr mit *ret* abgeschlossen werden, da ja zuvor durch den *jl*-Befehl kein Upper Context gesichert wurde. Daher ersetzt man *ret* durch einen indirekten Sprung an die durch den *jla*-Befehl im $A11$-Register gesicherte Rücksprungadresse, unabhängig davon welche der drei vorigen Methoden man benutzt hat. Die resultierende Pseudo-Funktion sieht dann wie folgt aus:

```
    ...
.EX_L1:
    mov   %d0, %d15
    sha   %d15, %d0,2
    mov   %d0, %d15
    ld.w  %d15,[%a10] 12
    add   %d1, %d0, %d15
    ji    %a11
```

Warum habe ich gerade die drei Methoden so ausführlich beschrieben? Einerseits wollte ich mit realen Codefragmenten zeigen, dass man tatsächlich meist mit wenigen Befehlen Exlining durchführen kann, andererseits erkennt man hier, dass man auf der RISC-Architektur des TriCore einen Geschwindigkeitsvorteil erlangen kann, je nachdem, welche der drei Methoden alternativ zu *call* und *ret* gewählt werden muss. Wie es zu vermuten war, ist es günstiger, wenn man das Register $A11$ in ein anderes freies Register sichern kann. Beim Platzbedarf sieht es ähnlich aus. Der TriCore hat von vielen Befehlen nicht nur eine 32 Bit sondern auch eine 16 Bit Variante. Von den Befehlen *st.a* und *ld.a* gibt es nur eine 32-Bit Variante. Anders sieht es da bei den *mov*-Befehlen aus. Diese müssen nur vor Labeln ggf. zu 32 Bit Länge aufgefüllt werden, da Sprungziele auf geraden Adressen liegen. Neben zu erwartenden Vorteilen im Platzbedarf kann man auch einen Vorteil im Energiebedarf der Varianten mit den *mov*-Befehlen sehen, da der Zugriff auf den Hauptspeicher, bei der Ausführung von Programmen die meiste Energie benötigt (siehe [Gru02]) und die Zahl der Befehle, die für das Sichern benötigt werden, geringer ist. Da der Overhead sowohl in Zeit- Platz und Energiebedarf der letzten der drei Varianten am größten ist, werde ich in einer ersten Implementierung Präfixgruppen, die sich nur auf diese Art herausziehen lassen, nicht extrahieren. Sollte sich herausstellen, dass es häufig große Gruppen sehr langer Präfixe geben sollte, die ein entsprechendes Exlining lohnenswert erscheinen lassen, kann diese Möglichkeit entsprechend berücksichtigt werden.

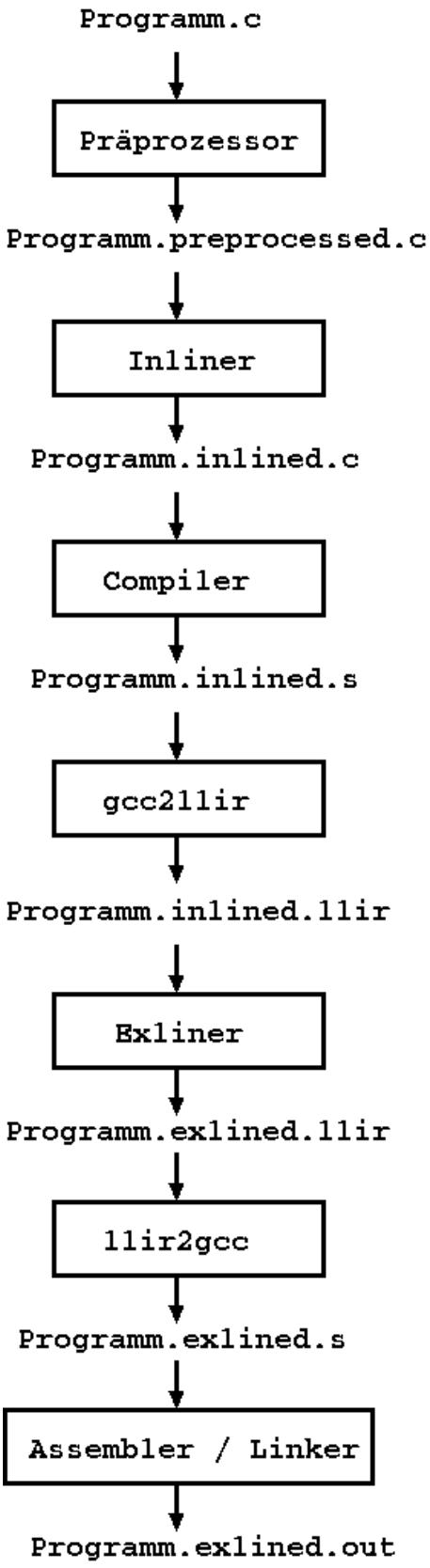

Abbildung 4.1: Toolchain zum In- und Exlining

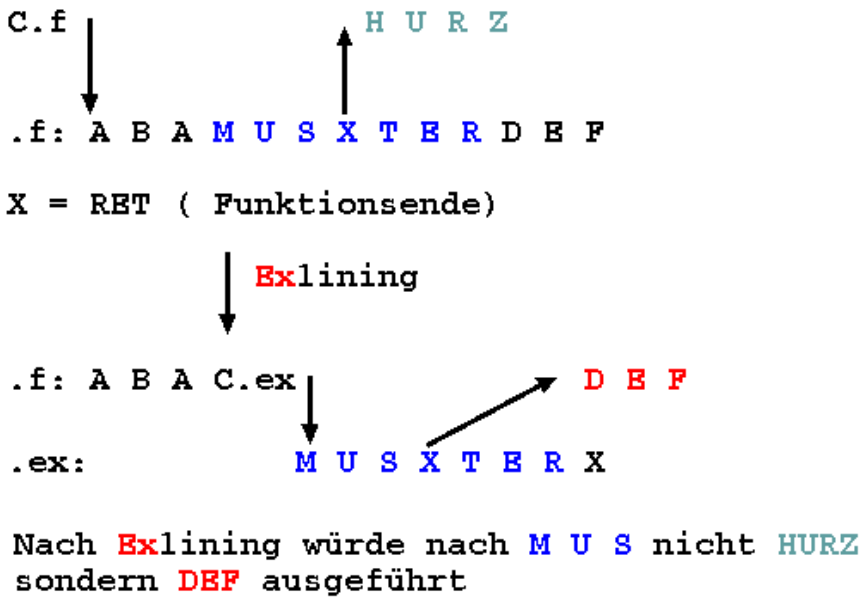

Abbildung 4.2: zu 3a): Nicht über Funktionsende

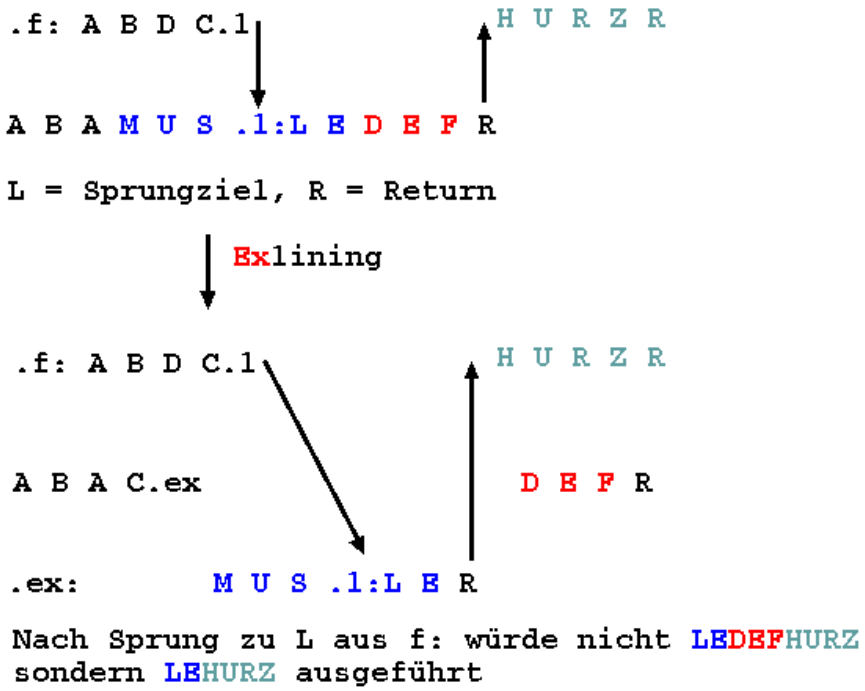

Abbildung 4.3: zu 3b): Nicht über Sprungziel

Kapitel 5

Ergebnisse für den TriCore

Nach der Vorstellung der Algorithmen für das Inlining und das Exlining, kommen wir nun zu den Benchmarks für den TriCore.

Um die Korrektheit der durch den Inliner und Exliner modifizierten Programme zu überprüfen fand zunächst ein Test mit einer Auswahl von ca. 30 Testprogrammen statt, die im Vergleich jeweils das gewünschte gleiche Ergebnis lieferten. Neben diesen ersten Funktionstests mussten natürlich weitere Tests erfolgen. Dazu wurden einige Beispiele aus NetBench [MMSH01], aus CommBench [WF00], eine Software-Implementierung von Fließkomma-Arithmetik nach IEC/IEEE Standard für binäre Fließkomma-Arithmetik von John R. Hauser sowie Code aus einer GSM-Bibliothek betrachtet.

5.1 Benchmarks

5.1.1 GSM

GSM steht heute für Globales System für Mobilkommunikation. Es stellt den Weltweit akzeptierten Standard für Mobilkommunikation dar.

5.1.2 SoftFloat

Bei der Implementierung des IEC/IEEE Standards für binäre Fließkomma-Arithmetik in Software handelt es sich um einen Plattform-Fließkommatest.

5.1.3 NetBench

Bei NetBench [MMSH01] handelt es sich um eine Testsuite für Netzwerkprozessoren, die einen Querschnitt über kommerzielle Applikationen für Netzwerkprozessoren enthält.

- *CRC (Cyclic Redundancy Check):* Hierbei handelt es sich um ein Programm aus dem Micro-Level Bereich. Es berechnet eine Prüfsumme wie sie in ISO 3309 [IS 84] beschrieben und bspw. vom Ethernet-Protokoll benutzt wird.
- *DRR (Deficit-Round Robin):* Es handelt sich dabei um ein Scheduling Verfahren, das in vielen heutiger Switches zum Einsatz kommt.
- *ROUTE:* Route implementiert das IPv4 Routing gemäß RFC 1812 [Bak95].
- *MD5 (Message Digest):* Dabei handelt es sich um eine kryptografische Prüfsumme für ausgehende Pakete, die beim Ziel überprüft wird. Die Implementierung stammt von der RSA Data Security, Inc. [RDS]

5.1.4 CommBench

Bei CommBench [WF00] handelt es sich ebenso, wie bei NetBench, um einen Benchmark für Netzwerkprozessoren. Zum Teil überdecken sich die Applikation. So gehört das oben beschriebene *DRR* ebenfalls zum CommBench.

- *FRAG:* Frag ist eine Applikation zum Aufteilen von IP-Paketen. IP-Pakete werden in mehrere Fragmente aufgeteilt, für die einige Header-Felder angepasst werden müssen. Der größte Teil der Applikation besteht aus der Berechnung von Prüfsummen, die von allen Programmen durchgeführt werden müssen, die IP-Pakete verarbeiten und sie nicht nur unverändert weiterleiten.
- *REED:* Bei Reed handelt es sich um eine Implementierung des Reed-Solomon Forward Error Correction Scheme, das Redundanz zu Daten hinzufügt, um eine gewisse Anzahl von Übertragungsfehlern korrigieren zu können (siehe [RF89])

5.2 Bewertung

Um das Inlining und Exlining zu bewerten, werden die oben aufgeführten Programme mit verschiedenen Optimierungsparametern übersetzt und mit, ausschließlich mit dem *tricore-gcc* übersetzten, Programmen verglichen. Programme, die ausschließlich mit dem *tricore-gcc*

übersetzt wurden, werden zur Abkürzung im Folgenden als GCC-Version bezeichnet. In den Tabellen werden Spalten, die sich auf die GCC-Version beziehen, mit *gcc* gekennzeichnet. Spalten, die die Version mit durchgeführtem In- und Exlining beschreiben, werden teilweise mit *inex* markiert.

Um möglichst genau die direkten Auswirkungen zu sehen, wird die Größe des Textsegmentes der übersetzten Objekt-Files verglichen. Hier soll nicht mit einem Kostenmaß gearbeitet werden, das auf der **LLIR** arbeitet, da bei manchen Befehlen erst nach der Übersetzung klar ist, ob die 16-Bit Variante oder die 32-Bit Variante eines Befehls benutzt wurde. So müsste man bei der Sequenz, die als Pseudo-Funktion dazu kommt, die 32-Bit-Variante addieren, dürfte aber nur jeweils die 16-Bit-Variante subtrahieren, wenn man keine zu optimistischen Aussagen treffen wollte.

Die Toolchain ist noch nicht so weit, dass man damit die kompletten Standard-Bibliotheken übersetzen kann. So gehören die Größen der fertigen Programme mit den statisch dazu gebunden Bibliotheken zwar zu den Ergebnissen. Daran soll man aber nur erkennen, dass der Vergleich der Text-Segmente der Objectfiles objektiv ist, da sich die Größenunterschiede absolut auf die Größen des Textsegmentes des fertigen Programms übertragen. Die in den folgenden Tabellen angegebenen prozentualen Veränderungen beziehen sich also auf die Größen der Textsegmente der Objectfiles im Vergleich zu den GCC-Versionen der Programme.

Die Programme wurden jeweils mit verschiedenen Optimierungsparametern übersetzt. Zu den verschiedenen Optimierungsoptionen gehören einerseits die verschiedenen Optimierungsstufen des *tricore-gcc*. Dabei wurden die Optionen $-O1$, $-O2$ und $-O3$ betrachtet. Die Betrachtung von Programmen ohne Optimierungen des *tricore-gcc* macht keinen Sinn, da einerseits die Programme bereits mit der ersten Optimierungsstufe wesentlich kleiner werden als ohne, andererseits wird keine Funktion mittels Inlining eingefügt, wenn keine Optimierungsstufe aktiviert wurde. Eine korrekte Implementierung ohne Optimierung wäre zu fehleranfällig und scheint auch wenig sinnvoll.

Die ersten beiden Optimierungsstufen sollen im wesentlichen den Code kleiner machen, wobei bei der ersten Optimierungsstufe lediglich lokale Optimierungen durchgeführt werden sollen. Ohne die erste Optimierungsstufe landen nur Variablen in Registern, die explizit mit *register* Deklariert wurden. Die zweite Optimierungsstufe schaltet nahezu alle Optimierungen an, die nicht auf Kosten des Platzbedarfs Geschwindigkeit optimieren. Die dritte Optimierungsstufe enthält zusätzlich die Option *-finline-functions* und nimmt für Geschwindigkeitsgewinn ein größeres Programm in Kauf. Allgemein kann man sich weder bei $-O2$- noch bei $-O3$-Optimierung darauf verlassen, dass das Programm anschließend noch korrekt arbeitet. Die übersetzten Ergebnisse müssen überprüft werden.

Der Inliner hat als möglichen Parameter die maximale Zeilenanzahl der Funktionen, die *inline* deklariert werden dürfen. Dem Exliner kann man die Kosten für den Overhead eines

Pseudo-Funktionsaufrufs sowie die maximal zulässige Schleifentiefe eines Befehls, der mit Exlining in eine Pseudo-Funktion wandern darf übergeben.

Der Overhead eines Pseudo-Funktionsaufrufs wurde konstant auf 10 Byte gesetzt, was den 2 Mov-Operationen zur Sicherung der Rücksprungadresse im Register $D14$ und dem Sprung mit jl Rechnung tragen soll.

Da das Exlining durch die Einführung von Pseudo-Funktionen allgemein zu einer höheren Laufzeit führt, werden auch die Laufzeiten in der Zahl der benötigten Taktzyklen verglichen, die vom Simulator mit angegeben werden. Interessant ist dabei insbesondere die Auswirkung der verschiedenen maximalen Schleifentiefen für das Exlining.

5.2.1 GSM

Beim *GSM*-Beispiel handelt es sich um Quelltext, der viele Makro-definitionen enthielt. Der Sourcecode enthielt zwar einige *static* deklarierte Funktionen, diese wurden aber mehrfach aufgerufen, so dass keine Funktion vom Inliner *inline* deklariert wurde. An diesem Beispiel kann man also die Auswirkungen des Exlining direkt erkennen. Die Benutzung von Makros, was im wesentlichen einer Funktionsdeklaration als *static inline* entspricht, bietet für das Exlining das größte Optimierungspotential. Wie man in Tabelle 5.1 erkennen kann, erhält man den kleinsten Code, wenn man zuvor im Compiler die zweite Optimierungsstufe -O2 wählt und für das Exlining die maximale Schleifentiefe (mld= max loop depth) auf den höchsten gewählten Wert zwei setzt. Im Vergleich mit der kleinsten GCC-Version des übersetzten Programms sieht man einen Platzvorteil der Codegröße von 15,3% (siehe Abbildung 5.3).

Gleichzeitig kann man an diesem Beispiel aber auch die Auswirkungen einer hoch gewählten maximalen Schleifentiefe für den Exliner(siehe Abschnitt 4.2.1) auf die Laufzeit erkennen. Die Laufzeit wächst im Vergleich zu der kleinsten GCC-Version des Programms um 6,9%.

Schließt man mit der Wahl einer maximalen Schleifentiefe von Null, alle Befehle, die sich innerhalb einer Schleife befinden, vom Exlining aus, so wirkt sich das hier positiv auf die Laufzeit aus. Wählt man bei Optimierungsstufe -O2 *mld=0*, so spart man immer noch 12,3% bei der Codegröße, kann den Laufzeitverlust aber auf verschwindend geringe 0,4% reduzieren.

Betrachtet man die Optimierungsstufe -O3, die in erster Linie auf Geschwindigkeit optimiert, kommt man in diesem Beispiel bei *mld=0* auf einen Platzvorteil von 14,4% gegenüber der schnellsten und auf 12,3% gegenüber der kleinsten GCC-Version des Programms. Gleichzeitig benötigt man aber nur nur 0,4% mehr Taktzyklen zur Ausführung des Programms als die schnellste GCC-Version.

5.2.2 SOFTFLOAT

Bei dem Beispiel *SOFTFLOAT* spart man gegenüber der kleinsten GCC-Version 2,7% Codegröße und benötigt dafür 0,1% mehr Laufzeit. Dabei wurde der Parameter *bodyline=0* als maximale Zeilenanzahl für den Inliner (siehe Abschnitt 4.1.1) und *mld=1* für den Exliner gewählt. An diesem Beispiel erkennt man den Einfluss des Parameters *bodyline* für den Inliner. Je größer dieser gewählt wird, um so schneller wird das Programm. Wird der Wert jedoch zu groß gewählt, wirkt sich das negativ auf die Größe des resultierenden Programms aus.

Obwohl die Zeilenzahl ein sehr ungenauer Parameter ist, kann man hier mit der Optimierungsstufe -O2, der Wahl von *bodyline=10* und einer maximalen Schleifentiefe *mld=1* ein Programm erreichen, das 1,2% schneller als das schnellste, ausschließlich vom GCC mit Optimierungsstufe -O3 übersetzte Programm ist. Dabei spart man außerdem gegenüber dieser langsameren, schnellsten GCC-Version 6,0% Codegröße. Wählt man die maximale Schleifentiefe für den Exliner hier mit *mld=2* zu groß, wird das Programm langsamer als die schnellste GCC-Version (siehe Tabelle 5.2).

5.2.3 FRAG

Bei diesem Beispiel kann durch den Inliner kein Platz gespart werden, da das Programm weder eine *static* deklarierte Funktion enthält, die nur einmal aufgerufen wird, noch enthält sie eine so kleine Funktion, dass der Aufrufoverhead größer wäre als die Funktion selbst.

Nach der dritten Optimierungsstufe -O3 konnte der Exliner keine gemeinsamen Befehlssequenzen ermitteln, so dass die Änderung der Codegröße um 4 Byte lediglich durch zusätzlich durch den gcc2llir eingeführte labels und dadurch bedingte Alignments auf gerade Adressen zu erklären ist. Der Assembler-Code unterscheidet sich ansonsten nicht.

Die kleinste Version (Parameter *(-O1, mld=1, bodyline=0)* spart gegenüber der kleinsten GCC-Version des Programms 1,0% Codegröße und benötigt 0,2% mehr Laufzeit.

5.2.4 REED

Beim *REED*-Beispiel spart die kleinste Version gegenüber der kleinsten GCC-Version lediglich 0,5% Codegröße, wird dabei aber nicht langsamer (siehe 5.4).

Bemerkenswert dabei ist, dass die Einsparungen ausschließlich durch den Inliner hervor-

gerufen werden, der zwei *static* deklarierte Funktionen, die nur einmal aufgerufen werden, *inline* deklariert hat, so dass diese im Platzbedarf einer Makrodefinition entsprechen. Bei den, mit der ersten Optimierungsstufe -O1 übersetzten, Programmen sind keine gemeinsamen Befehlssequenzen vorhanden, die der Exliner zu einer Pseudo-Funktion herausziehen könnte.

Das Dekodieren konnte mit dem Simulator nicht erfolgreich getestet werden. Dieser brach schon bei den ausschließlich mit dem GCC übersetzten Versionen nach Dekodieren von 873kB Referenzdaten ab, so dass eine reine Angabe der Größen bzw. Laufzeiten nicht sinnvoll bzw. nicht möglich ist.

5.2.5 CRC

Beim *CRC*-Beispiel liegen die Einsparungen bei 0,6%, bei einer Vergrößerung des Zeitbedarfs um 0,02% (siehe Tabellen 5.5 und 5.6).

5.2.6 DRR/ROUTE

Für das *ROUTE*-Beispiel, das in seinem Quelltext *DRR*(siehe Abschnitt 5.1.3) nutzt, können leider keine sinnvollen Laufzeiten ermittelt werden, da in einer Initialisierungsroutine mehr Speicher angefordert wird, als der Simulator zur Verfügung stellt. So bricht das Programm bereits die Initialisierung ab. Ein Laufzeitenvergleich macht daher keinen Sinn.

5.2.7 MD5

Beim *MD5-Beispiel* werden 0,5% Codegröße bei gleichzeitigem Laufzeitzuwachs von 0,01% gespart (siehe Tabelle 5.7).

Die letzteren Beispiele waren insgesamt klein, so dass die relativ kleinen Einsparungen nicht so stark bewertet werden dürfen. Auf die jeweils kleinste GCC-Version bezogen bringen die jeweils kleinsten Varianten mit In- und Exlining eine durchschnittliche Einsparung von 4,1% Codegröße (siehe Tabelle 5.9 und Abbildung 5.3) bei einem durchschnittlichem Laufzeitzuwachs von nur 0,1% (siehe Tabelle 5.8 und Abbildung 5.2).

Wenn man sich die Anteile der Laufzeiten der Programme an der aufsummierten Gesamtlaufzeit ansieht (siehe Abbildung 5.4 und Tabelle 5.10), so erkennt man, dass *FRAG* und

REED mit ihren großen Laufzeiten den durchschnittlichen Laufzeitverlust glätten, da ihre Code-Größenanteile verglichen mit ihren Laufzeiten sehr gering sind. Dies bestätigt und berücksichtigt aber letztlich nur, wie bereits in Abschnitt 2 erwähnt, die Gültigkeit der in [HP95] beschriebenen 90 / 10 Regel, d.h. ein Programm führt 90 % seiner Anweisungen in 10 % seines Codes aus. Auch wenn es sich hier um Benchmark-Programme handelt, ist diese Laufzeitverteilung also durchaus realistisch.

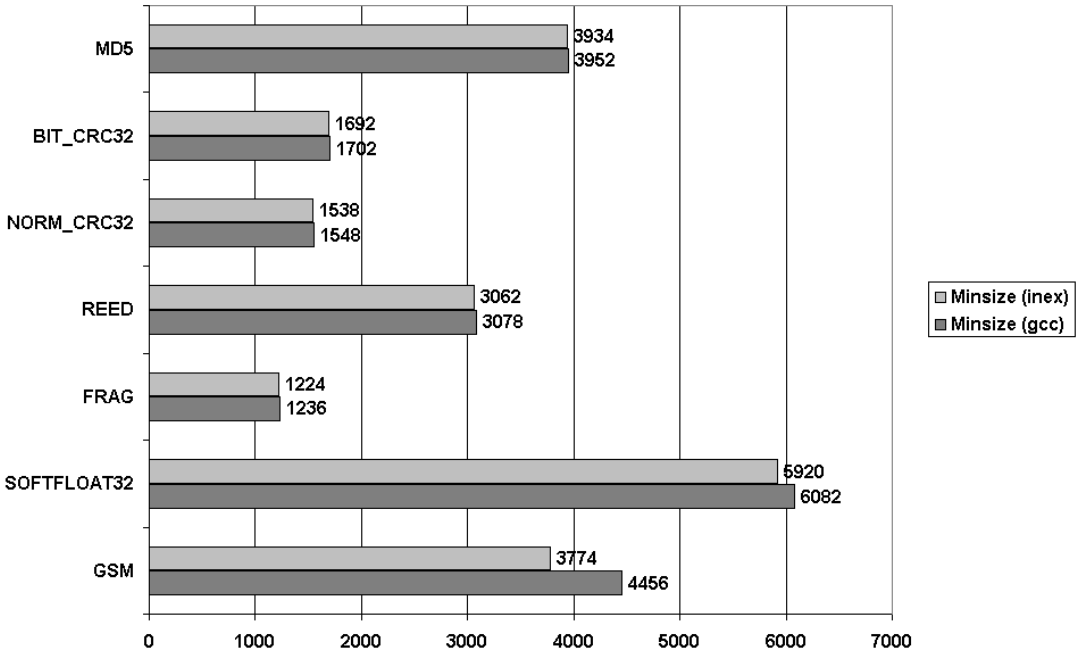

Abbildung 5.1: Größe der minimalen Programme

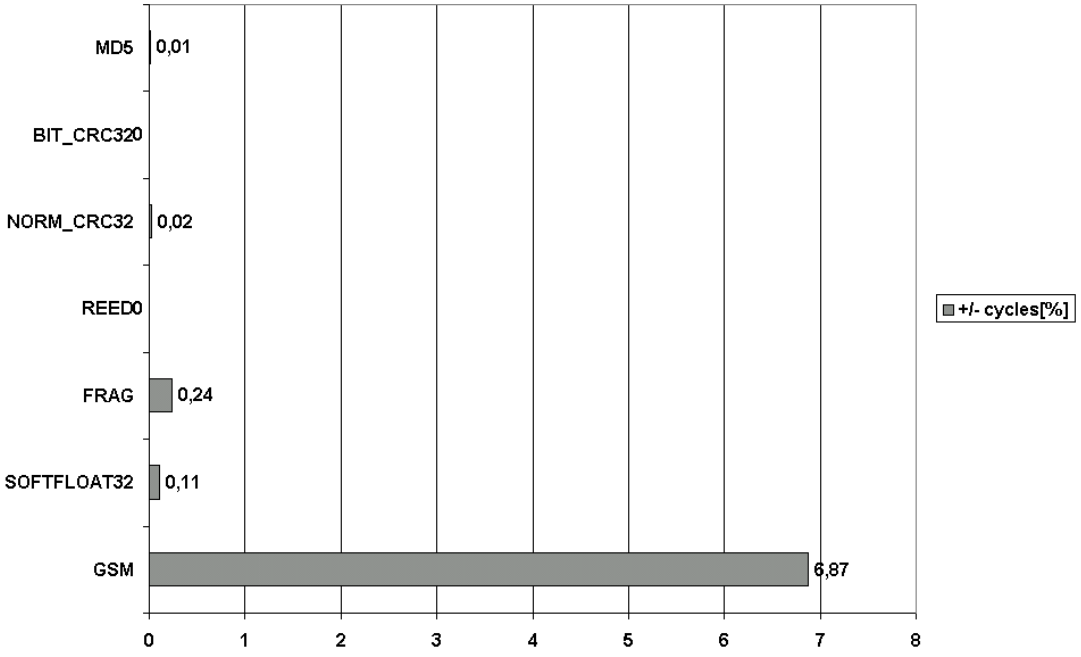

Abbildung 5.2: Laufzeitzuwachs der minimalen Programme

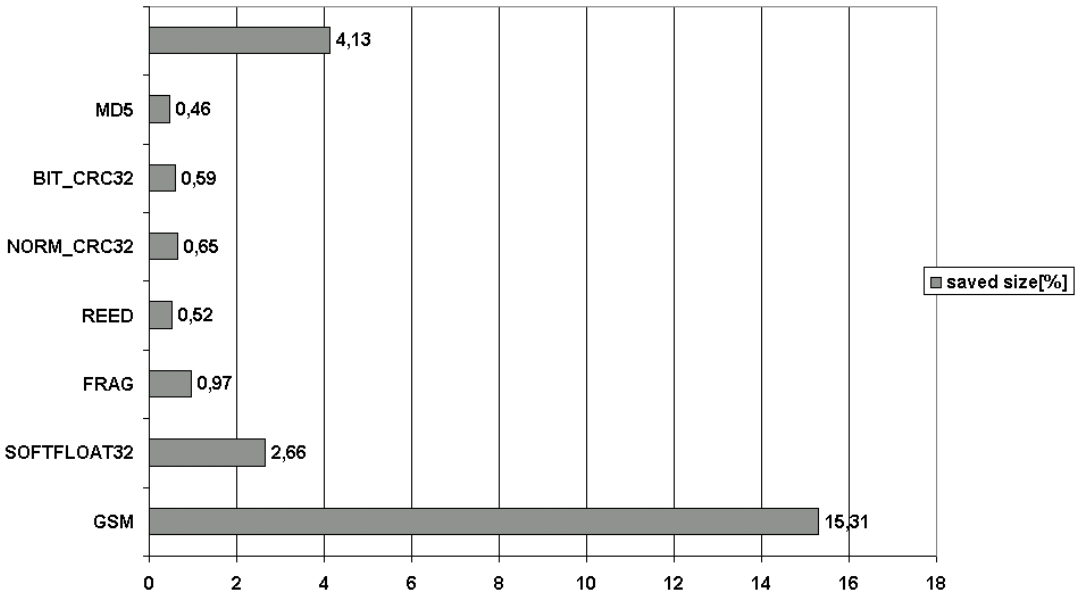

Abbildung 5.3: Platzersparnis der minimalen Programme

Mit In- und Exlining

	O3					
bodyline	0			5		
mld	2	1	0	2	1	0
size	3884	3898	4016	3884	3898	4016
cycles	67806	67554	**63674**	67806	67554	**63674**
bodyline	10			15		
mld	2	1	0	2	1	0
size	3884	3898	4016	3884	3898	4016
cycles	67806	67554	**63674**	67806	67554	**63674**
	O2					
bodyline	0			5		
mld	2	1	0	2	1	0
size	**3774**	3790	3908	**3774**	3790	3908
cycles	67856	67604	63724	67856	67604	63724
bodyline	10			15		
mld	2	1	0	2	1	0
size	**3774**	3790	3908	**3774**	3790	3908
cycles	67856	67604	63724	67856	67604	63724
	O1					
bodyline	0			5		
mld	2	1	0	2	1	0
size	5148	5162	5192	5148	5162	5192
cycles	79061	78809	78617	79061	78809	78617
bodyline	10			15		
mld	2	1	0	2	1	0
size	5148	5162	5192	5148	5162	5192
cycles	79061	78809	78617	79061	78809	78617

Ohne In- und Exlining

	O3	O2	O1
size	4564	**4456**	5362
cycles	**63444**	63494	78489

Tabelle 5.1: GSM-Benchmark

Mit In- und Exlining

	O3					
bodyline	0			5		
mld	2	1	0	2	1	0
size	6592	6592	6592	6592	6592	6592
cycles	187133	187133	187133	187133	187133	187133
bodyline	10			15		
mld	2	1	0	2	1	0
size	6782	6796	6796	6782	6796	6796
cycles	186752	**171264**	**171264**	186752	**171264**	**171264**
	O2					
bodyline	0			5		
mld	2	1	0	2	1	0
size	**5920**	**5920**	5928	6298	6298	6298
cycles	216782	216782	216542	216188	216188	216188
bodyline	10			15		
mld	2	1	0	2	1	0
size	6228	6252	6252	6228	6252	6252
cycles	196310	184822	184822	196310	184822	184822
	O1					
bodyline	0			5		
mld	2	1	0	2	1	0
size	5978	5978	5982	6368	6368	6386
cycles	218619	218619	218619	218579	218579	218387
bodyline	10			15		
mld	2	1	0	2	1	0
size	6298	6316	6342	6298	6316	6342
cycles	193012	189140	185076	193012	189140	185076

Ohne In- und Exlining

	O3	O2	O1
size	6650	**6082**	6108
cycles	**187125**	216547	218625

Tabelle 5.2: softfloat32-Benchmark

Mit In- und Exlining

	O3					
bodyline	0			5		
mld	2	1	0	2	1	0
size	1494	1494	1494	1494	1494	1494
cycles	**500742072**	**500742072**	**500742072**	**500742072**	**500742072**	**500742072**
bodyline	10			15		
mld	2	1	0	2	1	0
size	1494	1494	1494	1494	1494	1494
cycles	**500742072**	**500742072**	**500742072**	**500742072**	**500742072**	**500742072**
	O2					
bodyline	0			5		
mld	2	1	0	2	1	0
size	1248	1248	1262	1322	1322	1330
cycles	508142548	508142548	506942308	507342060	507342060	506142060
bodyline	10			15		
mld	2	1	0	2	1	0
size	1322	1322	1330	1322	1322	1330
cycles	507342060	507342060	506142060	507342060	507342060	506142060
	O1					
bodyline	0			5		
mld	2	1	0	2	1	0
size	**1224**	**1224**	1240	1314	1314	1350
cycles	509743152	509743152	508543152	509043119	509043119	507842927
bodyline	10			15		
mld	2	1	0	2	1	0
size	1314	1314	1350	1314	1314	1350
cycles	509043119	509043119	507842927	509043119	509043119	507842927

Ohne In- und Exlining

	O3	O2	O1
size	1490	1262	**1236**
cycles	**500742072**	506942308	508543153

Tabelle 5.3: frag-Benchmark

Mit In- und Exlining

	O3					
bodyline	0			5		
mld	2	1	0	2	1	0
size	3238	3238	3238	3238	3238	3238
cycles	**659342064**	**659342064**	**659342064**	**659342064**	**659342064**	**659342064**
bodyline	10			15		
mld	2	1	0	2	1	0
size	3238	3238	3238	3238	3238	3238
cycles	**659342064**	**659342064**	**659342064**	**659342064**	**659342064**	**659342064**
	O2					
bodyline	0			5		
mld	2	1	0	2	1	0
size	3212	3212	3220	3238	3238	3238
cycles	659342652	659342652	659342412	**659342064**	**659342064**	**659342064**
bodyline	10			15		
mld	2	1	0	2	1	0
size	3238	3238	3238	3238	3238	3238
cycles	**659342064**	**659342064**	**659342064**	**659342064**	**659342064**	**659342064**
	O1					
bodyline	0			5		
mld	2	1	0	2	1	0
size	**3062**	**3062**	**3062**	3110	3110	3128
cycles	723604320	723604320	723604320	723604286	723604286	723604094
bodyline	10			15		
mld	2	1	0	2	1	0
size	3110	3110	3128	3110	3110	3128
cycles	723604286	723604286	723604094	723604286	723604286	723604094

Ohne In- und Exlining

	O3	O2	O1
size	3262	3244	**3078**
cycles	**657313402**	657313750	723604331

Tabelle 5.4: reed_enc-Benchmark

Mit In- und Exlining

	O3					
bodyline	0			5		
mld	2	1	0	2	1	0
size	1904	1904	1914	1904	1904	1914
cycles	1463425	1463425	1463669	1463425	1463425	1463669
bodyline	10			15		
mld	2	1	0	2	1	0
size	1904	1904	1914	2028	2028	2050
cycles	1463425	1463425	1463669	1462930	1462930	**1462274**
	O2					
bodyline	0			5		
mld	2	1	0	2	1	0
size	**1538**	**1538**	1548	**1538**	**1538**	1548
cycles	1466362	1466362	1466106	1466362	1466362	1466106
bodyline	10			15		
mld	2	1	0	2	1	0
size	1640	1640	1650	1976	1976	1998
cycles	1465632	1465632	1465876	1462937	1462937	1462281
	O1					
bodyline	0			5		
mld	2	1	0	2	1	0
size	1600	1600	1630	1600	1600	1630
cycles	1557383	1557383	1557191	1557383	1557383	1557191
bodyline	10			15		
mld	2	1	0	2	1	0
size	1670	1670	1700	2004	2004	2048
cycles	1557170	1557170	1556978	1553866	1553866	1553280

Ohne In- und Exlining

	O3	O2	O1
size	1916	**1548**	1630
cycles	**1463169**	1466106	1557191

Tabelle 5.5: norm_crc32-Benchmark

Mit In- und Exlining

	O3					
bodyline	0			5		
mld	2	1	0	2	1	0
size	2324	2324	2334	2324	2324	2334
cycles	8150656	8150656	8150900	8150656	8150656	8150900
bodyline	10			15		
mld	2	1	0	2	1	0
size	2324	2324	2334	2432	2432	2454
cycles	8150656	8150656	8150900	8150258	8150258	**8149602**
	O2					
bodyline	0			5		
mld	2	1	0	2	1	0
size	**1692**	**1692**	1702	**1692**	**1692**	1702
cycles	13394635	13394635	13394379	13394635	13394635	13394379
bodyline	10			15		
mld	2	1	0	2	1	0
size	1904	1904	1914	2380	2380	2402
cycles	8154358	8154358	8154602	8150265	8150265	8159609
	O1					
bodyline	0			5		
mld	2	1	0	2	1	0
size	1756	1756	1786	1756	1756	1786
cycles	13485656	13485656	13485464	13485656	13485656	13485464
bodyline	10			15		
mld	2	1	0	2	1	0
size	1938	1938	1968	2422	2422	2466
cycles	8246196	8246196	8246004	8241597	8241597	8241011

Ohne In- und Exlining

	O3	O2	O1
size	2324	**1702**	1786
cycles	**8150398**	13394379	13485464

Tabelle 5.6: bit_crc32-Benchmark

Mit In- und Exlining

	O3					
bodyline	0			5		
mld	2	1	0	2	1	0
size	4710	4710	4720	4710	4710	4720
cycles	2793437	2793437	2793181	2793437	2793437	2793181
bodyline	10			15		
mld	2	1	0	2	1	0
size	4710	4710	4720	5150	5150	5164
cycles	2793437	2793437	2793181	2791639	2791639	**2790983**
	O2					
bodyline	0			5		
mld	2	1	0	2	1	0
size	4140	4140	4150	4140	4140	4150
cycles	2798419	2798419	2798163	2798419	2798419	2798163
bodyline	10			15		
mld	2	1	0	2	1	0
size	4282	4282	4292	4552	4552	4566
cycles	2796689	2796689	2796433	2794096	2794096	2793440
	O1					
bodyline	0			5		
mld	2	1	0	2	1	0
size	**3934**	**3934**	3964	**3934**	**3934**	3964
cycles	2914477	2914477	2914285	2914477	2914477	2914285
bodyline	10			15		
mld	2	1	0	2	1	0
size	4052	4052	4082	4348	4348	4392
cycles	2913464	2913464	2913272	2910562	2910562	2909976

Ohne In- und Exlining

	O3	O2	O1
size	4754	4156	**3952**
cycles	**2751879**	2757273	2914285

Tabelle 5.7: MD5-Benchmark

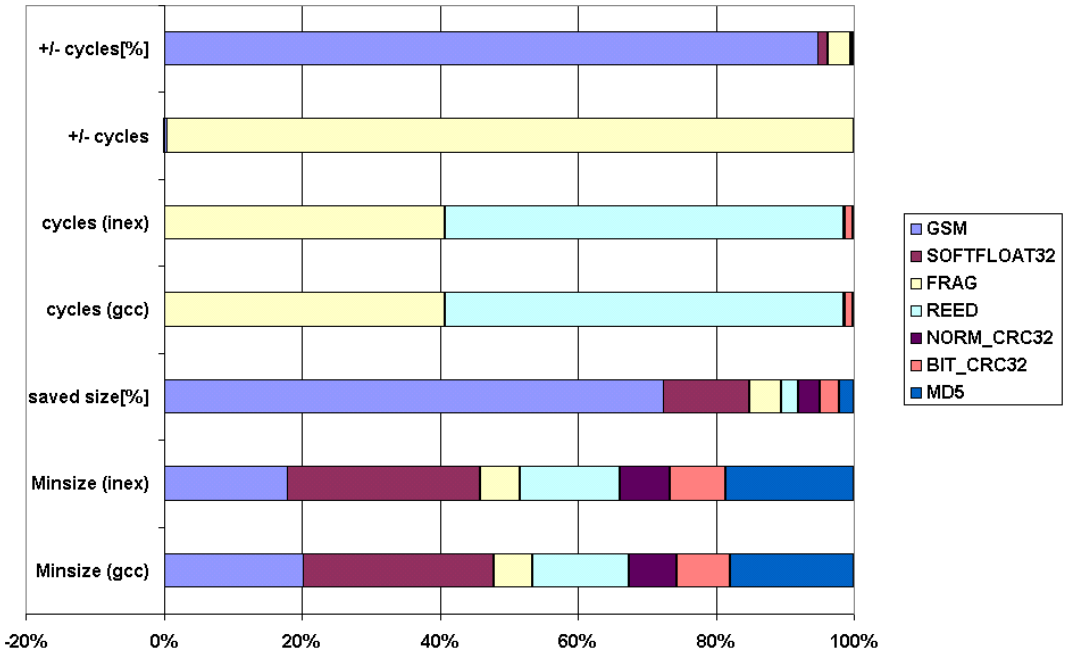

Abbildung 5.4: Größen- und Laufzeitanteile der minimalen Programme

program	minsize (gcc)	cycles	minsize (inex)	± cycles	±%
GSM	4456	63.494	3774	+4.362	+6,9%
SOFTFLOAT32	6082	216.547	5920	+235	+0,01%
FRAG	1236	508.543.153	1224	+1.199.999	+0,2%
REED	3078	723.604.331	3062	-11	-0,00%
NORM_CRC32	1548	1.466.106	1538	+256	+0,02%
BIT_CRC32	1702	13.394.379	1692	+256	+0,00%
MD5	3952	2.914.285	3934	+192	+0,01%
Total:	**22054**	**1.250.202.295**	**21144**	**+ 1.205.289**	**+0,10%**

Tabelle 5.8: Minimale Größen / Zugehörige Laufzeiten

program	minsize (gcc)	minsize (inex)	± size	±%
GSM	4456	3774	-682	-15,3%
SOFTFLOAT32	6082	5920	-162	-2,7%
FRAG	1236	1224	-12	-1,0%
REED	3078	3062	-16	-0,5%
NORM_CRC32	1548	1538	-10	-0,6%
BIT_CRC32	1702	1692	-10	-0,6%
MD5	3952	3934	-18	-0,5%
Total:	**22054**	**21144**	**-910**	**-4,1%**

Tabelle 5.9: Minimale Größen / Größenunterschiede

program	minsize (gcc)	quota [%]	minsize (inex)	quota[%]
GSM	4456	20,20	3774	17,85
SOFTFLOAT32	6082	27,58	5920	28,00
FRAG	1236	5,60	1224	5,79
REED	3078	13,96	3062	14,48
NORM_CRC32	1548	7,02	1538	7,27
BIT_CRC32	1702	7,72	1692	8,00
MD5	3952	17,92	3934	18,61
Total:	**22054**	100,00	**21144**	100,00

program	minsize (gcc) cycles	quota[%]	± minsize (inex) cycles	total cycles quota[%]
GSM	63.494	0,01	+4.362	0,01
SOFTFLOAT32	216.547	0,02	+235	0,02
FRAG	508.543.153	40,68	+1.199.999	40,73
REED	723.604.331	57,88	-11	57,82
NORM_CRC32	1.466.106	0,12	+256	0,12
BIT_CRC32	13.394.379	1,07	+256	1,07
MD5	2.914.285	0,23	+192	0,23
Total:	**1.250.202.295**	**100,00**	**+ 1.205.289**	**100,00**

Tabelle 5.10: Größen- und Laufzeitanteile der minimalen Programme

Mit In- und Exlining

	O3					
bodyline	0			5		
mld	2	1	0	2	1	0
size	26292	26308	26424	26292	26308	26424
cycles	67806	67554	**63674**	67806	67554	**63674**
bodyline	10			15		
mld	2	1	0	2	1	0
size	26292	26308	26424	26292	26308	26424
cycles	67806	67554	**63674**	67806	67554	**63674**
	O2					
bodyline	0			5		
mld	2	1	0	2	1	0
size	**26184**	26200	26316	**26184**	26200	26316
cycles	67856	67604	63724	67856	67604	63724
bodyline	10			15		
mld	2	1	0	2	1	0
size	**26184**	26200	26316	**26184**	26200	26316
cycles	67856	67604	63724	67856	67604	63724
	O1					
bodyline	0			5		
mld	2	1	0	2	1	0
size	27556	27572	27600	27556	27572	27600
cycles	79061	78809	78617	79061	78809	78617
bodyline	10			15		
mld	2	1	0	2	1	0
size	27556	27572	27600	27556	27572	27600
cycles	79061	78809	78617	79061	78809	78617

Ohne In- und Exlining

	O3	O2	O1
size	26972	**26864**	27768
cycles	**63444**	63494	78489

Tabelle 5.11: GSM-Benchmark Codegröße der ausführbaren Programme

Mit In- und Exlining

	O3					
bodyline	0			5		
mld	2	1	0	2	1	0
size	28720	28720	28720	28720	28720	28720
cycles	187133	187133	187133	187133	187133	187133
bodyline	10			15		
mld	2	1	0	2	1	0
size	28912	28924	28924	28912	28924	28924
cycles	186752	**171264**	**171264**	186752	**171264**	**171264**
	O2					
bodyline	0			5		
mld	2	1	0	2	1	0
size	**28048**	**28048**	28056	28428	28428	28428
cycles	216782	216782	216542	216188	216188	216188
bodyline	10			15		
mld	2	1	0	2	1	0
size	28356	28380	28380	28356	28380	28380
cycles	196310	184822	184822	196310	184822	184822
	O1					
bodyline	0			5		
mld	2	1	0	2	1	0
size	28108	28108	28112	28496	28496	28516
cycles	218619	218619	218619	218579	218579	218387
bodyline	10			15		
mld	2	1	0	2	1	0
size	28428	28444	28472	28428	28444	28472
cycles	193012	189140	185076	193012	189140	185076

Ohne In- und Exlining

	O3	O2	O1
size	28776	**28208**	28236
cycles	**187125**	216547	218625

Tabelle 5.12: softfloat32-Benchmark Codegröße der ausführbaren Programme

Mit In- und Exlining

	O3					
bodyline	0			5		
mld	2	1	0	2	1	0
size	25640	25640	25640	25640	25640	25640
cycles	**500742072**	**500742072**	**500742072**	**500742072**	**500742072**	**500742072**
bodyline	10			15		
mld	2	1	0	2	1	0
size	25640	25640	25640	25640	25640	25640
cycles	**500742072**	**500742072**	**500742072**	**500742072**	**500742072**	**500742072**
	O2					
bodyline	0			5		
mld	2	1	0	2	1	0
size	25392	25392	25408	25468	25468	25476
cycles	508142548	508142548	506942308	507342060	507342060	506142060
bodyline	10			15		
mld	2	1	0	2	1	0
size	25468	25468	25476	25468	25468	25476
cycles	507342060	507342060	506142060	507342060	507342060	506142060
	O1					
bodyline	0			5		
mld	2	1	0	2	1	0
size	**25368**	**25368**	25384	25460	25460	25496
cycles	509743152	509743152	508543152	509043119	509043119	507842927
bodyline	10			15		
mld	2	1	0	2	1	0
size	25460	25460	25496	25460	25460	25496
cycles	509043119	509043119	507842927	509043119	509043119	507842927

Ohne In- und Exlining

	O3	O2	O1
size	25636	25408	**25380**
cycles	**500742072**	506942308	508543153

Tabelle 5.13: frag-Benchmark Codegröße der ausführbaren Programme

Mit In- und Exlining

	O3					
bodyline	0			5		
mld	2	1	0	2	1	0
size	27384	27384	27384	27384	27384	27384
cycles	**659342064**	**659342064**	**659342064**	**659342064**	**659342064**	**659342064**
bodyline	10			15		
mld	2	1	0	2	1	0
size	27384	27384	27384	27384	27384	27384
cycles	**659342064**	**659342064**	**659342064**	**659342064**	**659342064**	**659342064**
	O2					
bodyline	0			5		
mld	2	1	0	2	1	0
size	27356	27356	27364	27384	27384	27384
cycles	659342652	659342652	659342412	**659342064**	**659342064**	**659342064**
bodyline	10			15		
mld	2	1	0	2	1	0
size	27384	27384	27384	27384	27384	27384
cycles	**659342064**	**659342064**	**659342064**	**659342064**	**659342064**	**659342064**
	O1					
bodyline	0			5		
mld	2	1	0	2	1	0
size	**27208**	**27208**	**27208**	27256	27256	27272
cycles	723604320	723604320	723604320	723604286	723604286	723604094
bodyline	10			15		
mld	2	1	0	2	1	0
size	27256	27256	27272	27256	27256	27272
cycles	723604286	723604286	723604094	723604286	723604286	723604094

Ohne In- und Exlining

	O3	O2	O1
size	27408	27388	**27224**
cycles	**657313402**	657313750	723604331

Tabelle 5.14: reed_enc-Benchmark Codegröße der ausführbaren Programme

Mit In- und Exlining

	O3					
bodyline	0			5		
mld	2	1	0	2	1	0
size	26336	26336	26348	26336	26336	26348
cycles	1463425	1463425	1463669	1463425	1463425	1463669
bodyline	10			15		
mld	2	1	0	2	1	0
size	26336	26336	26348	26460	26460	26484
cycles	1463425	1463425	1463669	1462930	1462930	**1462274**
	O2					
bodyline	0			5		
mld	2	1	0	2	1	0
size	**25972**	**25972**	25980	**25972**	**25972**	25980
cycles	1466362	1466362	1466106	1466362	1466362	1466106
bodyline	10			15		
mld	2	1	0	2	1	0
size	26072	26072	26084	26408	26408	26432
cycles	1465632	1465632	1465876	1462937	1462937	1462281
	O1					
bodyline	0			5		
mld	2	1	0	2	1	0
size	26032	26032	26064	26032	26032	26064
cycles	1557383	1557383	1557191	1557383	1557383	1557191
bodyline	10			15		
mld	2	1	0	2	1	0
size	26104	26104	26132	26436	26436	26480
cycles	1557170	1557170	1556978	1553866	1553866	1553280

Ohne In- und Exlining

	O3	O2	O1
size	26348	**25980**	26064
cycles	**1463169**	1466106	1557191

Tabelle 5.15: norm_crc32-Benchmark Codegröße der ausführbaren Programme

Mit In- und Exlining

	O3					
bodyline	0			5		
mld	2	1	0	2	1	0
size	26756	26756	26768	26756	26756	26768
cycles	8150656	8150656	8150900	8150656	8150656	8150900
bodyline	10			15		
mld	2	1	0	2	1	0
size	26756	26756	26768	26864	26864	26888
cycles	8150656	8150656	8150900	8150258	8150258	**8149602**
	O2					
bodyline	0			5		
mld	2	1	0	2	1	0
size	**26124**	**26124**	26136	**26124**	**26124**	26136
cycles	13394635	13394635	13394379	13394635	13394635	13394379
bodyline	10			15		
mld	2	1	0	2	1	0
size	26336	26336	26346	26812	26812	26836
cycles	8154358	8154358	8154602	8150265	8150265	8159609
	O1					
bodyline	0			5		
mld	2	1	0	2	1	0
size	26188	26188	26220	261881756	26188	26220
cycles	13485656	13485656	13485464	13485656	13485656	13485464
bodyline	10			15		
mld	2	1	0	2	1	0
size	26372	26372	26400	26856	26856	26900
cycles	8246196	8246196	8246004	8241597	8241597	8241011

Ohne In- und Exlining

	O3	O2	O1
size	26756	**26136**	26220
cycles	**8150398**	13394379	13485464

Tabelle 5.16: bit_crc32-Benchmark Codegröße der ausführbaren Programme

Mit In- und Exlining

	O3					
bodyline	0			5		
mld	2	1	0	2	1	0
size	29144	29144	29152	29144	29144	29152
cycles	2793437	2793437	2793181	2793437	2793437	2793181
bodyline	10			15		
mld	2	1	0	2	1	0
size	29144	29144	29152	29584	29584	29596
cycles	2793437	2793437	2793181	2791639	2791639	**2790983**
	O2					
bodyline	0			5		
mld	2	1	0	2	1	0
size	285724140	285724140	28584	28572	28572	28584
cycles	2798419	2798419	2798163	2798419	2798419	2798163
bodyline	10			15		
mld	2	1	0	2	1	0
size	28716	28716	28724	28984	28984	29000
cycles	2796689	2796689	2796433	2794096	2794096	2793440
	O1					
bodyline	0			5		
mld	2	1	0	2	1	0
size	**28368**	**28368**	28396	**28368**	**28368**	28396
cycles	2914477	2914477	2914285	2914477	2914477	2914285
bodyline	10			15		
mld	2	1	0	2	1	0
size	28484	28484	28516	28780	28780	28824
cycles	2913464	2913464	2913272	2910562	2910562	2909976

Ohne In- und Exlining

	O3	O2	O1
size	29188	28588	**28384**
cycles	**2751879**	2757273	2914285

Tabelle 5.17: MD5-Benchmark Codegröße der ausführbaren Programme

Kapitel 6

Ausblick

In der Diplomarbeit wurde gezeigt, dass es möglich ist, mit kombinierten Inlining und Exlining die Codegröße eines Programms um durchschnittlich 4,1% zu verringern, während gleichzeitig die Laufzeit durchschnittlich nur um 0,10% stieg (siehe Tabellen 5.9 und 5.8).

Der Inliner verarbeitet plattformunabhängig C-Code. In den überprüften Beispielen konnte durch ihn noch nicht viel gespart werden. Es ist denkbar, dass man durch ihn mehr sparen könnte, wenn er selbst feststellen könnte, wann eine Funktion *static* deklariert werden kann. Würden einfach aufgerufene Funktionen zuvor *static* deklariert, könnten diese auch *inline* deklariert werden, um den Aufruf-Overhead zu sparen. Ist die Funktion zuvor nicht *static* deklariert, ist sie auch nach anschließendem Inlining noch als Funktion im Object-File vorhanden. Dazu müssten alle Dateien und mögliche Referenzen betrachtet werden, wozu die aktuelle Implementierung nicht in der Lage ist. Dies wäre auch nur dann möglich, wenn man zur Übersetzungszeit bereits alle möglichen Referenzen aus anderen Programmdateien kennt bzw. kennen kann. Bei der Übersetzung von Bibliotheken, die erst von später geschriebenen Programmen benutzt werden sollen, wäre dies bspw. nicht der Fall.

Der Exliner arbeitet bei der Suche und dem Bewerten der Befehlssequenzen auf der **LLIR** und ist damit in den wesentlichen Teilen plattformunabhängig. Lediglich das tatsächliche Exlining, d.h. das Ersetzen der zuvor gefundenen gleichen Befehlssequenzen durch den plattformabhängigen Sprungbefehl zur ausgelagerten Pseudo-Funktion und die Rückkehr von dieser ausgelagerten Pseudo-Funktion sind TriCore-spezifisch.

Interessant wäre daher sicherlich die Betrachtung anderer Zielplattformen und ein Vergleich der erzielten Resultate. Nach der Überprüfung der Retargierbarkeit und der entsprechenden Einsparpotentiale, ist eine Erweiterung der **LLIR** um virtuelle Funktionen, die das Exlining in einer plattformabhängigen Implementierung durchführen, denkbar.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[AHU74] Aho, A. V. ; Hopcroft, J.E. ; Ullman, J.D.: *The Design and Analysis of Computer Algorithms*. Reading, Massachussetts : Addison-Wesley Publishing Company, 1974

[ASU88] Aho, A. V. ; Sethi, Ravi ; Ullman, Jeffrey D.: *Compilers, Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, März 1988

[Bak95] Baker, F. *Requirements for IP version 4 routers.* Request For Comment: 1812. Juni 1995

[CMCWH92] Chang, Pohua P. ; Mahlke, Scott A. ; Chen, William Y. ; mei W. Hwu, Wen: Profile-guided Automatic Inline Expansion for C Programs. In: *Software - Practice and Experience* 22 (1992), Nr. 5, S. 349–369

[Eck01] Eckart, Jörg: *Low Level Intermediate Representation.* : Informatik Centrum Dortmund e.V., 2001

[FW02] Franklin, Mark A. ; Wolf, Tilman: A Network Processor Performance and Design Model with Benchmark Parameterization. In: *Network Processor Design*, Morgan Kaufmann Publishers, September 2002

[GJ98] Grune, Dick ; Jacobs, Ceriel: *Parsing Techniques - A Practical Guide*. Ellis Horwood Limited, September 1998

[Gru02] Grunwald, Nils: *Energieminimierung eingebetteter Programme durch die dynamische Nutzung eines Scratchpad-Speichers*, Universität Dortmund, Diplomarbeit, 2002

[Güt92] Güting, Ralf H.: *Datenstrukturen und Algorithmen*. Stuttgart : B.G. Teubner, 1992

[HP95] Hennessy, John L. ; Patterson, David A.: *Computer Architecture: A Quantatative Approach*. Morgan Kaufmann Publishers, August 1995

[IS 84] *Processing Systems - Data Communications. High-Level Data Link Control Procedure - Frame Structure. IS 3309.* Oktober 1984

[JW96] Jagannathan, Suresh ; Wright, Andrew K.: Flow-directed Inlining. In: *SIGPLAN Conference on Programming Language Design and Implementation*, 1996, S. 193–205

[MMSH01] Memik, Gokhan ; Mangione-Smith, William H. ; Hu, Wendong: NetBench: A Benchmarking Suite for Network Processors. In: *ICCAD*, 2001, S. 39–

[MW94] M.Burrows ; Wheeler, D.J.: A Block-sorting Lossless Data Compression Algorithm / Systems Research Center. 130 Lytton Avenue, Palo Alto, California 94301, May 10 1994. – Forschungsbericht

[NRS00] Nyström, Sven-Olof ; Runeson, Johan ; Sjödin, Jan: Code Compression Techniques for Embedded Systems / Universität Uppsala. 2000. – Forschungsbericht

[Pyk03] Pyka, Robert: *Retargierbare Bitlevel Optimierungen für den Infineon Tricore Prozessor*, Universität Dortmund, Diplomarbeit, 2003

[RDS] RSA Data Security, Inc. *RSA Security Downloads.* http://www.rsasecurity.com/download

[RF89] Rao, T.R.N ; Fujiware, E.: *Error-Control Coding for Computer Systems.* Prentice Hall, Englewood Cliffs, NJ, 1989

[Rie97] Riedemann, Eike H.: *Testmethoden für sequentielle und nebenläufige Software-Systeme.* Teubner, 1997

[Ser97] Serrano, Manuel: Inline expansion: when and how? In: *Proceedings of the conference on Programming Languages, Implementation and Logic Programming.* Shouthampton, 1997, S. 15

[Vah95a] Vahid, Frank: Procedure Exlining: A New System-Level Specification Transformation. In: *Proc. of EuroDAC and VHDL'95*, 1995, S. 508–513

[Vah95b] Vahid, Frank: Procedure exlining: A transformation for improved system and behavioral synthesis. In: *International Symposium on System Synthesis*, 1995, S. 84–89

[WF00] Wolf, Tilman ; Franklin, Mark A.: CommBench - A Telecommunications Benchmark for Network Processors. In: *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software.* Austin, TX, April 2000, S. 154–162