GCC-Interna im Detail

Übersetzerbau

Andreas Bauer

Seit nunmehr zwei Jahrzehnten hat die GNU Compiler Collection zahlreiche Erweiterungen und Verbesserungen erfahren und – den Entwicklern sei Dank – noch ist kein Ende in Sicht. Version 4 steht bereits mit Änderungen vor den Toren und wird für das Frühjahr 2005 erwartet.

Ohne Zweifel ist die GNU Compiler Collection, kurz: GCC, eines der am häufigsten eingesetzten Programmpakete auf Unix-Systemen, da Open-Source-Software oft im Quelltext ausgeliefert wird. Als frei verfügbarer Übersetzer für eine Vielzahl von Programmiersprachen (C, C++, Java, Fortran, Ada, Treelang, Objective-C) und Plattformen dient dazu auf den meisten Systemen die GCC.

Allerdings schrecken viele Anwender auf Grund der hohen Gesamtkomplexität des Systems (circa 150 MByte systemnaher Quelltext) davor zurück, sich mit den GCC-Interna auseinander zu setzen. So kommt es, dass viele Verbesserungen, die das System über die Jahre im Detail erfahren hat, zu Unrecht nur Insider wahrgenommen haben.

Um die Struktur und Arbeitsweise von GCC besser zu verstehen, ist es zunächst wichtig, sich die Abläufe im Inneren der Suite vor Augen zu führen, Abbildung 1 zeigt eine schematische Darstellung. Wesentlich ist dabei die Unterscheidung zwischen Front- und Back-End, da beispielsweise sprachabhängige Funktionen fast ausschließlich in den Quellen der jeweiligen Front-Ends verankert sind. Aufgabe des Back-Ends hingegen ist die Optimierung und Erzeugung von plattformspezifischem Code. Getreu der GNU-Tradition repräsentiert das Tripel „CPU-Vendor-OS“ diese, wobei „OS“ entweder ein „System“ oder „Kernel-System“ ist. So steht beispielsweise „i586-unknown-linux“ für Linux auf einer gängigen Intel-Architektur.

Alles wird abstrahiert

Diese Informationen liegen in den Machine Descriptions, Dateien, die eine algebraische Beschreibung der Eigenschaften von Hardware und Betriebssystem enthalten. Gemeint sind damit insbesondere die Zahl der verfügbaren Register, Eigenheiten bezüglich Funktionsaufrufen und andere Details, die allesamt die Aufrufkonvention sowie das ABI einer Plattform vorschreiben (siehe Kasten „Application Binary Interface“).

Im Klartext: das GCC-Back-End führt alle Programmoptimierungen nur auf abstrakter Ebene aus und bildet den erzeugten Zwischen-Code erst im letzten Schritt auf die eigentliche Zielplattform ab. Dazu transferiert es den Quelltext vom Front-End zunächst auf eine generalisierte Syntaxbaumrepräsentation, die Abstract Syntax Trees (AST) und übersetzt ihn anschließend in die Register Transfer Language (RTL). RTL kann man sich als Maschinensprache für eine virtuelle CPU vorstellen, die prin-

iX-TRACT

- Die Gnu Compiler Collection GCC optimiert plattformübergreifend auf der Register Transfer Language.
- GCC kann Code für hunderte verschiedener Plattformen erzeugen.
- GIMPLE und GENERIC erleichtern die Unterstützung neuer Sprachen in die freie Übersetzer-Suite.

zipiell über unendlich viele Register und verschiedene logische Operationen zum Umgang mit Daten verfügt. Das hat den Vorteil, dass alle Optimierungen flexibel, aber trotzdem irgendwie „Hardware-nah“ sind, denn von RTL zur tatsächlichen Maschine ist es kein weiter Weg.

Optimierungen verbessert

Lange Zeit war die Version 2.95 von GCC der Referenz-Compiler auf Linux- und BSD-Plattformen. Die Akzeptanz des Schrittes von 2.95 auf 3.0 war zunächst zögerlich, nicht zuletzt dadurch, dass Red Hat damals auf eigene Faust eine zum Teil nicht binärkompatible Zwischenversion (2.96) auslieferte. Trotzdem haben sich die aktuellen Releases mittlerweile weit etabliert, und viele Anwender kommen damit auf Grund von tiefgreifenden Architekturänderungen in den Genuss von Optimierungen, die früher undenkbar gewesen wären.

So verarbeitete GCC vor Version 2.x den Quelltext jeweils per Anweisung und übersetzte diese relativ knappen Programmteile in RTL. In GCC 2.95 stellten Funktionen, die natürlich aus mehreren Anweisungen bestehen, eine logische Übersetzungseinheit dar: Das Backend überführte also Funktionen in AST; dann kam das Function-Inlining (siehe Kasten „Inlining“) und schließlich übersetzt GCC den vorverarbeiteten Zwischen-Code für weitere, eher low-level ausgerichtete Optimierungen in RTL.

GCC 3.4 erweiterte diese Fähigkeit nochmals: einige Front-Ends, darunter C, C++, Objective-C und Java, verarbeiten jetzt ganze Dateien als eine Übersetzungseinheit und können somit weit reichendere Optimierungen anwenden, beispielsweise die Eliminierung von unerreichbaren Programmfragmenten. GCC realisiert dies durch die interne Erstellung eines aufwendigen Aufrufgraphen (Call Graph), der die Abhängigkeiten von Funktionen zueinander widerspiegelt. Aktivieren lässt sich diese erweiterte Analyse durch die Kommandozeile mit *-funit-at-a-time*, oder einfach durch Hinzufügen von *-O2*.

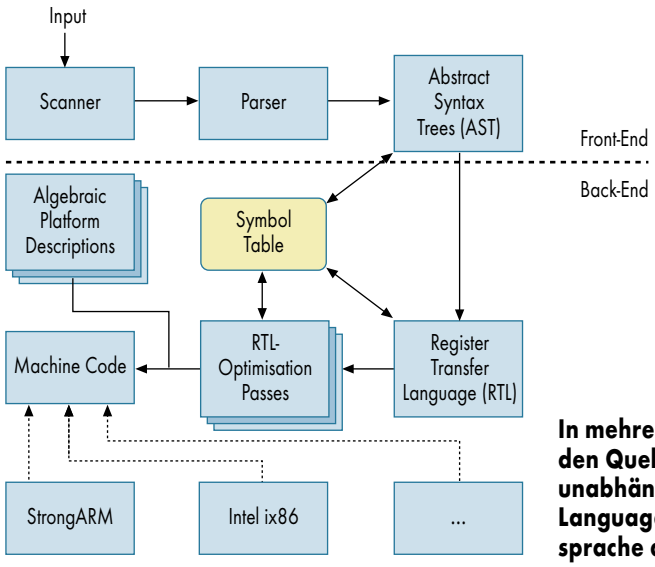
Sehr zur Freude von C/C++- und Objective-C-Programmierern hat Apple nun endlich auch die nötigen Patches zu den vorübersetzten Header-Dateien in GCC 3.4 einfließen lassen, die unter Mac OS X schon seit längerem zur Verfügung standen. Dieses Feature beschleunigt zwar nicht das übersetzte Programm, verkürzt allerdings die Übersetzungszeit erheblich, da der Compiler Header-Dateien, die er schon verarbeitet hat (sofern sie sich seitdem nicht mehr geändert haben), nicht erneut mit übersetzen muss.

Überhaupt hat sich mit der Einführung von GCC 3.x die C++-Unterstützung stark verbessert, was nicht zuletzt damit zusammenhängt, dass sich auch der C++-Standard selbst zunehmend stabilisiert hat. Am augenfälligsten ist vermutlich die Transition hin zur neuen „C++ Cross Vendor ABI“ (siehe Kasten). Mit dieser erzeugt GCC (genauer gesagt: G++) binärkompatible Dateien, die man zum Teil mit nicht von der freien Übersetzer-Suite erstellten Modulen linken kann. Das erhöht nicht nur die Flexibilität, sondern im Fall von G++ zusätzlich die Performanz, da das neue Modell viele Laufzeitüberprüfungen (Typen, Abbildung von Klassen im Speicher et cetera) wesentlich effizienter implementiert.

Übersetzen für fremde Ziele

Auf Grund des stark modularen Aufbaus der Übersetzer-Suite macht GCC eine wahrlich meisterhafte Figur, wenn es darum geht, Code für nicht native Zielarchitekturen zu erzeugen. Tatsächlich gibt es wohl kaum einen anderen Cross-Compiler, der für so viele unterschiedliche Plattformen Code generieren kann – sei es nun im freien oder kommerziellen Umfeld.

Je nach Version unterstützt GCC mehr als 20 Architekturen, darunter Sparc, Intel, AMD, PowerPC, ARM oder



In mehreren Stufen transferiert GCC den Quellcode über die hardwareunabhängige Register Transfer Language (RTL) in die Maschinensprache der Zielarchitektur (Abb. 1).

Application Binary Interface

Unter einem Application Binary Interface (ABI) versteht man allgemein eine Menge von Eigenschaften, die der Übersetzer statisch überprüft und berücksichtigt, damit er binärkompatiblen Code für eine Plattform erzeugen kann. Konkret sind damit Konventionen bezüglich Funktionsaufrufen gemeint, beispielsweise bestimmte Register zum Austausch von Argumenten, oder wie der Runtime Stack aufgebaut sein muss, damit Funktionen richtig darauf zugreifen können.

Wenn man über die ABI spricht, meint man dabei häufig die Unix System V ABI (für die Sprache C), die spezifiziert, wie Daten- und Systemstrukturen beschaffen sein müssen, damit man eigene Programme gegen existierende Bibliotheken linken kann. ABIs werden normalerweise von Gremien beschlossen, sind unter Umständen aber lediglich durch eine Anzahl von Werkzeugen (Übersetzer, Assembler, Linker et cetera) bestimmt, die auf einer gegebenen Plattform zur Verfügung stehen.

C++ Cross Vendor ABI

Ursprünglich entstand das so genannte „Cross Vendor Application Binary Interface“ für den 64-bittigen Intel Itanium. Angesichts der damaligen Kompatibilitätsprobleme zwischen verschiedenen C++-Übersetzern wählten die Entwickler diese als Neuanfang für einen plattformübergreifenden Standard. Das hatte leider den unangenehmen Seiteneffekt, dass sich alte C++-Bibliotheken mit G++-3.x-erzeugtem Code nicht mehr ohne weiteres linken ließen.

Da die Arbeiten an der C++-ABI auch heute noch nicht gänzlich abgeschlossen sind, lassen sich ähnliche Probleme in aktuellen Versionen gut durch Kommandozeilenparameter umgehen: *-fabi-version=1* legt fest, dass der Compiler die mit G++ 3.2 eingeführte ABI verwendet; *-fabi-version=0* hingegen wählt die ABI, die am dichtesten zum jeweiligen C++-Standard ist, der sich auch in Zukunft an manchen Stellen noch ändern kann. Mit der neuen Option *-Wabi* gibt der G++ Warnungen aus, wenn er erkennt, dass der erzeugte Code nicht mit dem Standard übereinstimmt.

VAX und mindestens ebensoviele Systemkerne, wie Linux, verschiedene BSDs oder Solaris. Rein kombinatorisch ergeben sich daraus hunderte von Plattformen, die sich als Tripel (oder Quadrupel) darstellen und konfigurieren lassen. Nötig ist dazu lediglich der Quelltext von GCC, sowie die GNU *binutils*, die den Linker und Assembler enthalten.

Je nach Anforderung benötigt man noch einige Bibliotheken, doch das hängt von der Beschaffenheit des zu übersetzenden Quelltextes ab. Hier ein konkretes Beispiel:

```
$ ./configure --target=\
    x86_64-unknown-linux \
  --disable-nls \
  --disable-shared \
  --without-headers \
  --enable-languages=c
$ make
```

Diese beiden Befehle übersetzen GCC für das 64-Bit-AMD-System *x86_64-unknown-linux*. Der neue GCC verwendet dann die passenden Machine Descriptions für AMD und erzeugt fortan 64-Bit-optimierten Code, ohne dass die Plattform physisch zur Verfügung stünde.

Die zusätzlichen Konfigurationsparameter dienen lediglich dazu, die Abhängigkeiten der Werkzeugkette zu minimieren. So deaktiviert der NLS-Schalter die Internationalisierung der *binutils* und *disable-shared* legt fest, dass GCC nur statisch linken darf, denn die installierten Systembibliotheken sind zum AMD-erzeugten Code in der Regel nicht binärkompatibel und somit unbrauchbar. *without-headers* weist GCC an, sich nicht in Abhängigkeit von einer bestimmten C-Systembibliothek zu begeben und *enabled-languages* legt schließlich die Sprachunterstützung fest – in diesem Fall nur C.

Diese Fähigkeit, die der GCC solch eine Flexibilität verleiht, legt man ihr oft auch als größtes Manko zur Last. In vielen Vergleichen mit anderen Übersetzern am Markt ist GCC stets das Schlusslicht, was die Übersetzungszeit anbelangt. Dazu muss man aber wissen, dass beim Erzeugen von Code bei GCC intern wesentlich mehr passiert als bei Konkurrenzprodukten, denn GCC führt alle Optimierungen zunächst abstrakt auf den Zwischenrepräsentationen AST und RTL aus und entscheidet erst ganz zuletzt, welche der parallel generierten Instruktionen sich tatsächlich eignen, auf die Zielplattform abgebildet zu werden.

So übersetzt GCC beispielsweise einen einfachen Funktionsaufruf nicht geradewegs in eine CALL-Instruktion, sondern erzeugt mindestens drei unterschiedlich optimierte Patterns. Erst am Ende entscheidet die Beschreibung der ABI, der CPU und andere Faktoren, welches im Code landet. Intuitiv dauert das natürlich dreimal länger, garantiert aber ein Maximum an Flexibilität und Portabilität.

AST als Schnittstelle

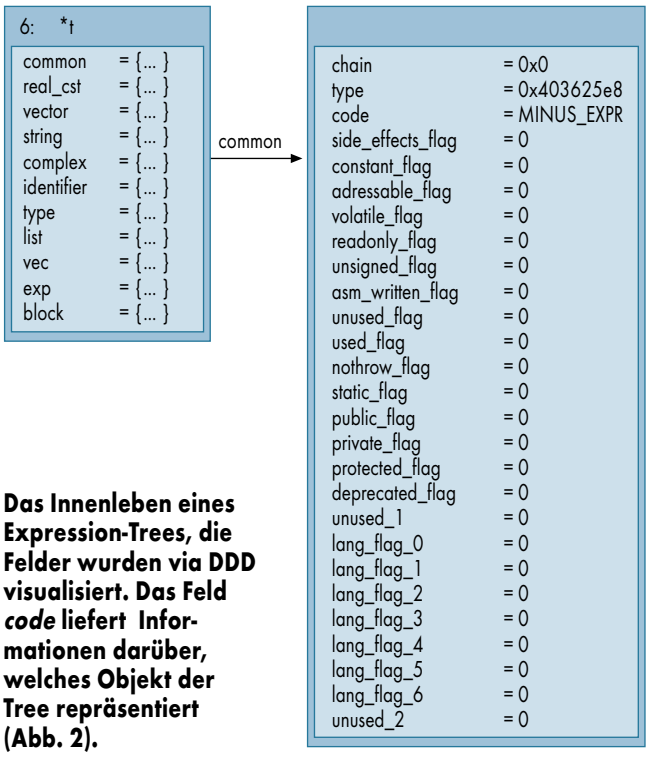
Nicht nur das portable Back-End, sondern auch die Vielzahl an Front-Ends unterstreichen die Adaptionsfähigkeit von GCC. Neben den eingangs erwähnten existieren zahlreiche, unabhängige Sprachimplementierungen, die allesamt auf GCC zur nativen Code-Generierung aufsetzen. Das Integrationsschema über AST hin zu RTL und schließlich zum Assembler ist dabei sprachübergreifend stets dasselbe. Mit anderen Worten: AST bildet die Hauptschnittstelle zwischen Front- und Back-End.

Traditionell galt: GCC-basierte Sprachimplementierungen, die nicht irgendwie C-artig ausfielen, wie beim objektorientierten Java, mussten zum Teil große Anstrengungen unternehmen, um neue repräsentative Trees einzuführen, die den AST um passende Fähigkeiten erweiterten.

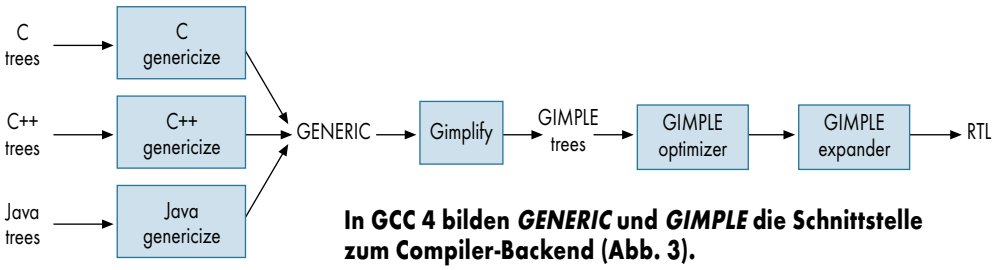
Allgemein sind Trees in *gcc/tree.def* und *gcc/tree.h* definiert, lassen sich aber durch Makros erweitern. Ein Tree ist dabei lediglich ein Zeiger, der auf einen Knoten im AST weist. Jeder Knoten wiederum hat einen „tree code“ assoziiert, der darüber Aufschluss gibt, welche Art von Objekt er repräsentiert. Selbsterklärende Beispiele für Trees sind *INTEGER_TYPE*, *ARRAY_TYPE* oder *PLUS_EXPR*, was einem arithmetischen Plus-Ausdruck entspricht, also $z <- x + y$.

Ein Baum ist ein Baum

Beim Tree selbst handelt es sich um eine C-Union. Auf die Felder hat man im Front-End über Makros Zugriff. So gibt beispielsweise *TREE_CODE* den absoluten Code eines Trees wieder und über *TREE_OPERAND* würde man die Operanden einer *MINUS_EXPR* erhalten (siehe Abbildung 2).



Das Innenleben eines Expression-Trees, die Felder wurden via DDD visualisiert. Das Feld *code* liefert Informationen darüber, welches Objekt der Tree repräsentiert (Abb. 2).



In GCC 4 bilden *GENERIC* und *GIMPLE* die Schnittstelle zum Compiler-Backend (Abb. 3).

Eine Übersicht über die Makros findet sich in den GCC Manuals [1] und in den Dateien, die auch die Trees selbst definieren. Wer sehr viel mit GCC arbeitet, wird deren Inhalt bald auswendig kennen.

Neue Trees, beispielsweise für neue Datentypen oder Programmausdrücke, sind an sich schnell via *DEFTREECODE* erzeugt: *DEFTREECODE(PLUS_EXPR, „plus_expr“, ’2’, 2)*. Das erste Argument ist der Code, das zweite der Typ und das dritte legt fest, ob es sich um einen Ausdruck oder eine Konstante, Deklaration oder andere Arten handelt. Das letzte Argument ist optional und bestimmt, wieviel Speicher für Operanden zur Verfügung stehen muss.

Jeder Teil des AST muss entweder eine direkte Entsprechung in RTL haben oder aber in andere, schon existierende Trees überführbar sein. Bei komplexen Neudefinitionen fällt somit ein großer Berg Implementierungsarbeit an, weil sich diese nicht immer aus bestehenden Trees adäquat aufbauen lassen.

Ein Blick in die Zukunft

Da RTL im Endeffekt nichts anderes als eine generalisierte Maschinensprache ist, lag es auf der Hand, an genau diesem Punkt Abhilfe zu schaffen. Die kommende Version von GCC hat sich zum Ziel gesetzt, nicht nur die Integration von Front-Ends zu erleichtern und zu standardisieren, sondern auch zusätzliche Optimierungen einzubauen, die man bisher auf der niedrigen Abstraktionsebene von RTL nicht durchführen konnte oder wollte.

Kurzum, mit GCC 4 kommen zwei neue Zwischendarstellungen ins Spiel, *GENERIC* und *GIMPLE* (siehe Abbildung 3). Erstere ist eine Art größter gemeinsamer Teiler von praktischen AST-Repräsentationen und wurde weit gehend durch die Java-eigenen Tree-Definitionen inspiriert, die sich als universell einsetzbar erwiesen. Sie vereinfacht die Integration eines Front-Ends erheblich, denn plötzlich steht eine wesentlich höhere Zahl vordefinierter Trees zur Verfügung, sowie eine einheitliche Schnittstelle zu den folgenden Optimierungen. Auch das manuelle Hinzufügen von fehlerträchtigen RTL-Anweisungen entfällt mit dieser Schnittstelle.

Weiter reichende Optimierungen

Bei GIMPLE handelt es sich um ein Derivat der von der McGill University entwickelten Sprache SIMPLE für den eigenen McCAT-Übersetzer. Dabei ist GIMPLE eine C-artige Drei-Adressform, die alle wichtigen Informationen wie Typdefinitionen erhält und sich auf einfache Art und Weise analysieren lässt [2]. Beispielsweise musste GCC für einen Ausdruck wie

```
a = b + c – d;
```

bisher lediglich einen einzelnen Tree erzeugen. Die neue GIMPLE-basierte Variante wandelt diesen Ausdruck in die Drei-Adressform um und generiert somit zwei Trees:

```
t1 = b + c;
a  = t1 – d;
```

Diese Umformung selbst stellt noch keinen großen Vorteil dar, jedoch gibt es eine ganze Reihe von Algorithmen, die genau diese einfache Drei-Adressform zur Grundlage haben. Das heißt, durch GIMPLE steht dem GCC-Projekt nun eine große Anzahl von neuartigen Optimierungen zur Verfügung, die sich durch Vorarbeiten an SIMPLE und McCAT schon über viele Jahre in der Praxis bewährt haben. Diese sind auf einer Ebene anwendbar, die weit über dem bisherigen Abstraktionsniveau der RTL liegt.

Zudem sollten sich mit relativ geringem Aufwand bestehende Front-Ends an GENERIC/GIMPLE anpassen lassen, da sich die Schnittstellen zwar geändert haben, diese aber im Prinzip weiterhin Tree-basiert sind. Die tiefgreifenden Umstrukturierungen, die GIMPLE mit sich bringt, sind zum großen Teil im Back-End vergraben und betreffen somit den Benutzer nicht direkt. (avr)

ANDREAS BAUER

ist wissenschaftlicher Mitarbeiter am Lehrstuhl Prof. Broy der Technischen Universität München. Er hat die Optimierung von indirekten Endaufrufen für Intel-Architekturen in die GCC 3.4 integriert.

Literatur

[1] GCC Online Documentation. Function Attributes, gcc.gnu.org/onlinedocs/gcc/Function-Attributes.html

[2] Laurie J. Hendren, Chris Donawa, Maryam Emami, G. Gaung R. Gao, Justiani, and Bhama Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. Lecture Notes in Computer Science, no. 457, S. 406-420, Springer-Verlag, August 1992

Inlining

Bei Funktionsaufrufen speichern Anwendungen oft den Inhalt der Register auf dem Runtime Stack zwischen. Allerdings ist bei sehr kurzen Unterprogrammen die dazu benötigte Zeit größer als die eigentliche Ausführungszeit, was das Inlining verhindern soll. Dazu fügt der Übersetzer die Instruktionen der aufzurufenden Funktion einfach zum aufrufenden Teil hinzu, so dass kein Zugriff auf den Stack erfolgen muss. Beeinflussen lässt sich diese Eigenschaft außerdem durch die explizite Attributierung von Funktionen und Funktionsdeklarationen, wie die beiden folgenden Beispiele demonstrieren:

```
// Diese Funktion immer inlinen.
int foo (float a, int b) __attribute__
((always_inline));

// Diese Funktion niemals inlinen.
static int bar (int a) __attribute__
((noinline));
```

Bei den Attributen handelt es sich um GCC-eigene Features, die eng mit dem Back-End verzahnt sind. Sie können aber bei richtiger Anwendung Programme spürbar beschleunigen. Eine Übersicht über weitere Funktionsattribute liefert die GCC-Online-Dokumentation [1].