GNU Compiler Collection

Von der Kathedrale zum Basar?

Februar 2006

- Robert Schuster
  robertschuster@fsfe.org

GCC – Geschichte

- C Compiler des GNU-Projekts
- ursprünglicher Autor Richard Stallman
- 23 Mai 1987: Version 1.0
- viele Releases in kurzer Zeit
- seit 1990 mit C++ Unterstützung
- 22 Februar 1992: Version 2.0

GCC – Geschichte

- 1997–1999: EGCS
- Juli 1999: GCC 2.95
- Juni 2001: Version 3.0
  - erstmals mit GCJ
- verbesserter Releasezyklus
  - +0.1 im Jahresrhythmus
  - +0.0.1 zur Stabilisierung in 2–3 Monatsabständen

GCC – Geschichte

- Mai 2005: GCC 4.0
  - Autovektorisierung, Tree-SSA, ...
- November 2005: GCC migriert zu Subversion

Was kann die?

- Frontends:
  - Ada, C, C++, Fortran, Java, Objective-C
  - daher GNU Compiler **Collection**
- unterstützt diverse Standards und Dialekte
- enthält Laufzeitbibliotheken
- Backends
  - so gut wie alles was Nullen und Einsen versteht :-)

unterstützte Plattformen

alpha*-*-*
alpha*-dec-osf*
alphaev5-cray-unicosmk*
arc-*-elf
arm-*-elf arm-*-coff arm-*-aout
xscale-*-*
avr
Blackfin
c4x
DOS
--freebsd*
h8300-hms
hppa*-hp-hpux*
hppa*-hp-hpux10
hppa*-hp-hpux11
--linux-gnu
i?86-*-linux*aout
i?86-*-linux*
i?86-*-sco3.2v5*
i?86-*-solaris2.10
i?86-*-udk
ia64-*-linux
ia64-*-hpux*
-ibm-aix
iq2000-*-elf
m32c-*-elf
m32r-*-elf
m6811-elf
m6812-elf
m68k-hp-hpux
mips-*-*
mips-sgi-irix5
mips-sgi-irix6
powerpc-*-darwin*
powerpc*-*-linux-gnu*
powerpcle-*-eabisim
powerpc-*-netbsd*
powerpc-*-eabisim
powerpc-*-eabi
powerpcle-*-eabi
s390-*-linux*
OS/2
s390x-*-linux*
s390x-ibm-tpf*
--solaris2*
sparc-sun-solaris2*
sparc-sun-solaris2.7
sparc-*-linux*
sparc64-*-solaris2*
sparcv9-*-solaris2*
--sysv*
vax-dec-ultrix
--vxworks*
x86_64-*-* amd64-*-*
xtensa-*-elf
xtensa-*-linux*
Microsoft Windows
powerpcle-*-elf powerpcle-*-sysv4
powerpc-*-elf powerpc-*-sysv4
powerpc*-*-* powerpc-*-sysv4

Bedeutung

- freier Compiler ist zentrales Element für GNU
  - eingebettet in die GNU Toolchain: GDB, gmake, glibc, autotools, ...
- erfüllt die Unabhängigkeit von der Plattform
  - portabel
  - cross-compiling
- kompiliert Anwendungen wie Apache, Linux kernel
- weite Verbreitung im Bereich eingebetteter Systeme

kommerzieller Erfolg

- 1987 Michael Tiemann
  - C++ Unterstützung
  - Backends für eingebettete Systeme
- 1989 Gründung von Cygnus Support (später Cygnus Solutions)
  - „Making free software affordable.“
  - Portierung der GCC für diverse Architekturen
  - Cygwin: „GNU für Windows“

kommerzieller Erfolg

- Erfolgsrezept: Egal für welche Plattform programmiert wird, die Bedienung des Compilers bleibt immer gleich
- 1999 Intel beauftragt CS mit der Entwicklung von GCC-Optimierungsstufen, die MMX und SSE verwenden
- Cygnus wird von Red Hat aufgekauft, tragende Beteiligung an GCC/GNU bleibt bestehen

Der EGCS-Vorfall

- 1997: GCC Maintainer strebt nach höherer Stabilität von gcc2
- neue Features werden nur zögerlich aufgenommen
- Vorwurf der Kathedralenbauweise (Raymond)
- es existieren viele separat gepflegte Erweiterungen
  - pgcc – Pentium-optimiert
  - neues Fortran Frontend
  - bessere Linux Unterstützung

Der EGCS-Vorfall

- Unmut über die Nichtaufname der Patches wächst
  - Flamewar
  - Fork von einem CVS-Snapshot
- Verbindung aller separat entwickelten Patches
- Veröffentlichung als „Enhanced GNU Compiler System“ (EGCS)
- EGCS wird 2 Jahre lang in einem Zweig des GCC-CVS entwickelt

Der EGCS-Vorfall

- Entwicklung von EGCS geht deutlich besser voran als die der GCC
  - neue Features werden eingebracht
  - EGCS überholt GCC schnell als bevorzugter Compiler
- Entwicklung der GCC wird gestoppt
- Einigung erzielt:
  - Gründung eines Steuerungskomitees
  - Hinwendung zu offenerem Entwicklungsmodell
  - EGCS wird zu GCC erklärt
- mit GCC 2.95 sind beide Projekte wieder vereint

Mission Statement (1999)

- Kooperation & Kommunikation zwischen Entwicklern wird gefördert
- bessere Zusammenarbeit mit den Anwendern
- Verfügbarkeit des Quellcodes zu jeder Zeit und für jede Person, jeder ist eingeladen sich zu engagieren
- offen zugängliche Mailinglisten
- Entwicklerfreundliche Werkzeuge und Verfahren (CVS, mehrere Maintainer)
- ein Steering Commitee vermittelt bei Interessenskonflikten

Steering Commitee

- EGCS Projekt schafft sich eine FSF-unabhängige Verwaltung
- keine Partei soll zu großen Einfluss über das Projekt gewinnen
- alle Beteiligten sind an der Langlebigkeit der Software interessiert

Steering Commitee

- Teilnehmer vertreten eine Benutzergruppe, nicht ihre jeweiligen Arbeitgeber/akademischen Institutionen
- November 1998: EGCS Projektleitung wird zum offiziellen Steering Commitee des GCC Projektes
- der Ablauf diverser Entwicklungsvorgänge wird schriftlich festgehalten

neue Philosophie

- größere Verbesserungen werden prinzipiell als gut angesehen
- destabilisierende und größere Änderungen sollen in einem separaten Zweig durchgeführt werden
- Ablauf des beisteuern von Patches geregelt
  - Codier-Stil
  - testen
  - ChangeLog

Eine weitere Neuausrichtung

- Unzufriedenheit mit langer Entwicklungszeit zwischen 2.95 und 3.0
- Einführung einer stufenbasierten Entwicklung
- Ziel
  - Veröffentlichungsfahrplan soll eingehalten werden können
  - Zunahme von Stabilität bei Annäherung an den Veröffentlichungstermin

Entwicklungsstufen – Stage 1

- jede Änderung am Compiler ist möglich
- größere Änderungen aus den Zweigen dürfen eingepflegt werden
- Release Manager koordiniert diese Arbeiten

Entwicklungsstufen

- Stage 2
  - nur noch kleinere Erweiterungen erlaubt
  - Release Manager kann Patches verhindern
- Stage 3
  - es sind nur noch Bug Fixes zugelassen

Zwischenbilanz

- 2003: Zachary Weinberg “A maintenance developers' view of GCC”
- stellt drei wesentliche Probleme fest:
  - unfertige Übergänge
  - mehrfache Implementierungen
  - unsaubere Modularität
- außerdem Verfahrensmängel und technische Hürden

unfertige Übergänge

- bestehendes internes API wird erweitert
- abhängige Module sollen auf die neuen Funktionen zurückgreifen
- dies geschieht aber nur für einen Teil der Module
- altes API kann nicht entfernt werden

mehrfache Implementierung

- generische Lösung vorhanden, ist aber nicht leistungsfähig genug
- Neuschreiben einer speziellen Variante ging schneller
- Wiederholung an mehreren Stellen
- von allgemeiner Lösung hätten alle Module profitiert
  - jetzt Zwischenstufe mit Eigenlösungen und gar keinen Lösungen

unsaubere Modularität

- Entwickler verschiedener Teilbereiche sprechen sich nicht ab
- internes APIs wird den jeweiligen Anwendungsbereich angepasst
- Implementierung einer Schnittstelle für Entwickler unklar, wenn er sich nicht mit verwendenden Modul auskennt

problematische Prozeduren

- jedes Patch muss getestet werden
  - Kompilierung der gesamten GCC (alle Frontends) kann extrem lange dauern (bis 12h)
  - richtige Versionen von Autoconf, DejaGNU
  - für Vergleich ist der Testlauf mit dem unmodifizierten Compiler notwendig
  - wird das Patch einer Revision unterzogen, muss der Testlauf wiederholt werden

technische Hürden

- nicht alle Makefiles sind für Parallelität ausgelegt
  - kein Vorteil für Mehrprozessorsysteme
- 15s für 'cvs update' (keine Änderungen)
- 'configure.ac' setzt veraltete autoconf-Version voraus

Stand 2006

- unfertige Übergänge
  - “peephole optimization”
  - neue: gcc.gnu.org/wiki/Partial%20Transitions
- mehrfache Implementierungen
  - Vereinheitlichung der RTL-Vereinfachungsfunktionen vorangeschritten, aber nicht vollständig
  - generischer Baum-zu-RTL Konverter kann whole-Function Bäume verarbeiten, wird aber nicht in jedem Frontend eingesetzt (kein Bedarf)

Stand 2006

- unsaubere Modularität
  - Übergang zu Strukturen mit Funktionszeigern anstelle von Macros zur Kopplung des Kerns an das Backend schreitet voran

Stand 2006

- Verfahren
  - Regeln zur Patchaufnahme sind nicht in allen Fällen streng: Patch für Java-Interpreter benötigt keinen kompletten Testlauf aller Compiler-Frontends
  - Patch wird auch aufgenommen, wenn bekannt ist, dass es Probleme verursacht (die aber im Anschluß behoben werden)
  - es wird weiterhin auf Einhaltung von Codierrichtlinien, üblichen Herangehensweisen und effizienten Code geachtet

Neuerungen

- GCC-Wiki
  - Beschreibung von speziellen Anwendungsfällen (BC-Kompilierung)
  - Pläne
  - Probleme
- Subversion anstelle von CVS
  - schnelleres 'update'
  - sauberes 'revert' und 'resolve'
  - einfachere Verzweigungen

Neuerungen

- automatisierte Tests
  - “nightly builds”
- GCC Summit
  - Förderung der RL-Kommunikation
  - öffentliches Diskutieren von Problemen & Plänen

Schlussgedanken

- GCC im Basarzeitalter angekommen
- kein Garant für problemlose Entwicklung
  - Umsetzen von Features ist spannender als das Beheben von Altlasten – auch für die Arbeitgeber
  - Compilertechnik ist schwierig
  - Backends für alte Systeme uninteressant

Fragen

- Wohin führt diese Entwicklung?
  - GCC irgendwann nicht mehr beherrschbar?
  - konzertiertes Clean-Up?
- Welche Projekte werden als nächstes mit ihrer Komplexität zu kämpfen haben (oder tun es schon)?
  - Qt, Java, Eclipse, KDE/Gnome, X.Org, Linux ...

Bye

Danke!

Some rights reserved ;–)