GCC gestern, heute und morgen – die Performance der GNU Compiler Collection

Geschwindigkeitsrausch

Jede neue GCC-Version verspricht schnelleren und besseren Code zu generieren. Mit der Ende April veröffentlichten Version 3.4.0 soll die Compiler-Suite auch das Kompilieren schneller erledigen. Das Linux-Magazin hat nachgemessen und mit Intels ICC verglichen. René Rebe

Unter Linux setzen C- und C++-Programmierer fast ausschließlich auf die GNU Compiler Collection [1]. Dabei entwickelt sich auch GCC weiter. Zu wissen, welche Neuerung Version 3.4 verspricht [2] und wie die Performance im Vergleich zu älteren GCCs und Intels ICC [3] abschneidet, hilft bei der Wahl des richtigen Werkzeugs und der besten Optimierungsoptionen.

Ein wichtiges Ziel der GCC-Entwickler ist, dass alle in der Suite enthaltenen Compiler die Programmiersprachen vollständig und korrekt unterstützen. Da sich die Sprachstandards weiterentwickeln, ist dieses Unterfangen schwerer, als man glauben mag. Die GCC-Entwickler feilen noch an Ansi C99 und C++98, auch an Objective-C und vor allem Ada arbeiten sie intensiv.

Viel Energie verwenden sie darauf, die Codegenerierung weiter zu optimieren. Jeder Benutzer möchte seinen Prozessor optimal ausnutzen – ein Compiler-Update ist dafür die billigste Lösung. Intel hat mit dem C++-Compiler die Messlatte für x86-spezifische Optimierungen sehr hoch gelegt, wie die Titelgeschichte des letzten Linux-Magazins bewiesen hat. Der GCC muss sich mit dieser Vorlage vergleichen lassen.

Neu in GCC 3.4

Gegenüber GCC 3.3 hat sich recht viel verändert. Für Entwickler wichtig sind neue und bessere Optimierungen (siehe Kasten „Neue Optimierung in GCC 3.4“), PCH (Precompiled Header), erneute Änderungen im ABI (Application Binary Interface) sowie die wohl umfangreichste Änderung: der von Grund auf neu geschriebene C++-Parser.

Im Laufe der Zeit wurde es immer schwieriger, dem in die Jahre gekommenen YACC-basierten Parser alle Ansi-C++-Features beizubringen, ihn komplett standardkonform weiterzuentwickeln und die Fehlermeldungen lesbar zu gestalten. Daher entschieden sich die GCC-Entwickler dazu, den Parser von Hand (ohne YACC-Hilfe) neu zu programmieren.

Das ABI beschreibt unter anderem auch, wie der Binärcode Funktionsparameter übergibt, Ergebnisse zurückliefert, wie Strukturen im Speicher liegen und wie sie ausgerichtet sind (Alignment). Anders als bei früheren Versionen, bei denen das ABI teils größere Korrekturen erfahren hat, gibt es jetzt in GCC 3.4 zum Glück fast nur kleinere Änderungen bei einigen Plattformen (Mips, Sparc, Alpha). Auf den meistgenutzten Plattformen sind kaum Inkompatibilitäten zu erwarten.

Precompiled Header

Ein probates Mittel gegen langwierige Compiler-Läufe ist nun auch in den GCC eingeflossen: Precompiled Header, PCH. Der Compiler speichert dabei den Inhalt von Headerdateien für C, C++ und Objective-C nach dem Parsen in einer separaten Datei »*.gch«, und zwar in einer internen Binär-Repräsentation. Bei späteren Läufen muss der Compiler nur diese internen GCC-Strukturen laden und nicht erneut alle Header parsen. Das beschleunigt vor allem das Übersetzen großer C++-Projekte stark.

Um Header einzeln zu kompilieren, ruft man den Compiler so auf, als solle er eine normale C- oder C++-Datei übersetzen. Das Kommando »g++ Threads.hh« übersetzt den Threads-Header und erzeugt die – recht große – Datei »Threads.hh.gch«. Der große Umfang der kompilierten Header liegt unter anderem daran, dass Header selbst wieder andere Header einbinden.

Das Vorkompilieren benötigt Zeit, auch beim Übersetzen der Implementierungsfiles entsteht Overhead beim Laden der Precompiled Header. Es ist daher ineffizient, jeden Header einzeln vorzukompilieren. Stattdessen können Entwickler eines Projekts viele Header zusammenfassen:

```
$ g++ -I. `ls *.hh` -o Most.hh.gch
```

Anschließend fügen sie diesen virtuellen Header den »CXXFLAGS« hinzu. Da die meisten Header durch »#define«- und »#ifdef«-Makros selbst dafür sorgen, dass sie nur ein Mal eingebunden werden, genügt die »-include«-Option des GCC. Die Abbildung 1 zeigt mehrere Messungen, bei denen GCC das GSMP-Paket (General Sound Manipulation Program, [7]) übersetzen musste. Die erste Zeile zeigt die Ergebnisse mit der GCC-Version 3.2.3, danach folgen drei Messungen mit GCC 3.4.0.

Der Lauf ohne Precompiled Header dauert 114 Sekunden. Der zweite Messwert ergibt sich bei »make CXXFLAGS = "-include $PWD/utility/include/Most.hh"«. Die Laufzeit verringert sich um 20 Sekunden. Allerdings ist diese Technik nicht immer sinnvoll, etwa wenn sich während der Entwicklungsarbeit der Inhalt einzelner Headerdateien ändert. Dann ist es jedes Mal nötig, den virtuellen Header neu zu erstellen.

Vorkompilierte Standard Template Library

Wer in C++ die STL (Standard Template Library) verwendet, kann die Übersetzungszeiten noch weiter verbessern. Der GCC kompiliert die STL-Header gleich bei der Installation und stellt sie als »bits/stdc++.h« bereit. Diesen virtuellen Header kann ein Check im Configure-Skript der eigenen Software den »CXXFLAGS« hinzufügen. Der Zeitgewinn überzeugt: Mit »make CXXFLAGS = "-include bits/stdc++.h"« ergibt sich der letzte Wert in Abbildung 1, der Make-Durchlauf spart 46 Sekunden gegenüber GCC 3.4.0 ohne PCH.

Die wenigsten Programmierer werden ihre STL modifizieren, sie müssen die Pre-Compiled-Fassung also niemals neu erzeugen. Hier lohnen sich PCH besonders – die STL wächst in der vorkompilierten Fassung auf satte 21 MByte.

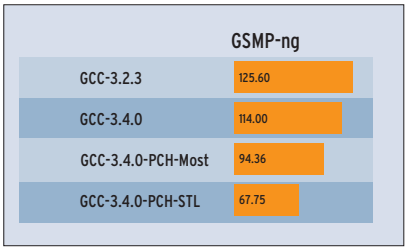

Abbildung 1: Ein Übersetzungslauf des GSMP-ng-Pakets ist mit GCC 3.4 generell schneller als mit GCC 3.2.3, durch Precompiled Header (PCH) reduziert sich die Laufzeit weiter.

Viele Neuerungen in GCC 3.4 betreffen den Parser, der nun nicht-Ansi-konformen C- oder C++-Code abweist. Die meisten dieser Standardverstöße treten glücklicherweise eher selten in Open-Source-Projekt auf.

Der neue Parser

Der Parser akzeptiert ungültige Semikola nicht mehr, etwa am Ende von Namensbereichen oder Funktionsdefinitionen:

```
namespace N {}; // Fehler
void f() {}; // Fehler
```

Auch Label am Ende eines Statements – ohne einen zugehörigen Ausdruck – gelten jetzt als Bug:

```
switch (i) {
  // ...
  default: // Fehler
}
```

Bei C++-Templates sucht der GCC jetzt Member-Variablen und -Methoden, die vom Template-Typ abhängen, bei der Deklaration nicht mehr in Basisklassen. Das ist standardkonform und für Spezialisierungen nötig. Der Programmierer muss dem Namen explizit »this->« voranstellen. Ein kleines Beispiel:

```
template <typename T> struct B {
  int m;
  int f ();
};
template <typename T> struct C : B<T> {
  void g () {
    m = 0;        // nicht mehr gefunden
    f ();         // nicht mehr gefunden
    this->m = 0;  // korrekt
    this->f ();   // korrekt
  }
};
```

Neue Optimierung in GCC 3.4

Die Entwickler haben GCC 3.4 einige neue Optimierungstechniken sowie bessere Implementierungen verpasst, um schnelleren Code zu erzeugen.

Inlining-Heuristiken: Der Compiler ersetzt Funktionsaufrufe durch den Code der gerufenen Funktionen. Bei C, Objective-C, C++ und Java wählt GCC jetzt Code für das Inlining zielsicherer aus.

Unit-at-a-Time Compilation: Optimiert den Code über Modulgrenzen hinweg.

Profile-Feedback: GCC nutzt die per Profiling gewonnenen Daten besser, zum Beispiel für Loop Unrolling. Das funktioniert jetzt auch mit mehreren gleichzeitig laufenden Prozessen eines Programms. Mit »make profiledbootstrap« ist es jetzt außerdem möglich, mit dieser Technik den Compiler selbst zu optimieren. Er soll dann schneller mit seiner Arbeit fertig werden.

Libstdc++: Die C++-Standardbibliothek ist in GCC 3.4 deutlich schneller als bisher. Durch Caching braucht formatierte Ein- und Ausgabe jetzt nur noch ein Drittel der Zeit von Version 3.2. Viele weitere Beschleunigungen in dieser Library erklärt Paolo Carlini in „Performance work in the libstdc++-v3" [6].

CFG-level Loop-Optimierer: Durch das Control-Flow-Graph-Projekt des GCC ist ein neuer Loop-Optimierer hinzugekommen, der auch die Techniken Loop Peeling und Loop Unswitching beherrscht (siehe Kasten „Optimierungstechniken").

Web Construction Pass: Dieser Schritt sollte in beinahe jedem Fall die Verteilung der vom übersetzten Programm verwendeten CPU-Register optimieren. Er ist per Default erst ab der Optimierungsstufe »-O3« eingeschaltet, da er das Debugging des Binärcodes sehr erschweren kann.

Der GCC verwendet für sein eigenes Memory-Management Garbage Collection. Die minimal allozierte Größe war aber noch aus den Zeiten der 200-MHz-CPUs auf 4 MByte gesetzt. Auf heutigen Rechnern mit viel Speicher führt das unnötigerweise zu wiederholtem Garbage Collecting während des Übersetzens. Deshalb ermittelt GCC seit Version 3.3 die Anfangsgröße dynamisch: ein Achtel des gesamten RAM, aber mindestens 4 MByte und maximal 128 MByte.

Für die kommenden GCC-Versionen sind wieder einige Änderungen geplant. So soll ein neuer Vektorisierer automatisch den Inhalt von Schleifen auf Vektoroperationen verteilen (siehe Kasten „Optimierungentechniken“).

Fortran 95 wird wohl der nächsten Major-Version hinzugefügt werden. Einer der Gründe ist, dass die Entwickler den vorhandenen G77 nicht für die aktuellen Veränderungen im Bereich der Optimierungen umschreiben möchten. Vor allem

Optimierungenstechniken

Moderne Compiler setzen eine ganze Menge Optimierungenstechniken ein. Wie sie wirken, lässt sich am besten anhand kleiner Codebeispiele nachvollziehen.

Peephole-Optimierungen

Sie kommen meist erst in einer späten Übersetzungsstufe zum Einsatz. Abhängig von der Architektur ersetzt der Compiler einzelne Instruktionen durch äquivalente Konstrukte, die der Prozessor in weniger Taktzyklen ausführt. Zum Beispiel ersetzt GCC die Multiplikation x·2 durch die Addition x+x oder das Initialisieren eines Registers durch eine Xor-Operation. Das Original in Maschinensprache setzt Register EBX auf null und enthält dazu einen Immediate-Wert, der ganze 4 Bytes groß ist:

```
movl  $0, %ebx
```

Die optimierte Variante berechnet den Xor-Wert des Registers mit sich selbst. Das ergibt ebenfalls null, allerdings muss die CPU dafür 4 Bytes weniger lesen:

```
xorl  %ebx, %ebx
```

Loop-Optimierungen

Diese wendet der Compiler auf einen Satz von Ausdrücken an, die eine Schleife ergeben, zum Beispiel For- und While-Schleifen in Hochsprachen. Loop-Optimierungen sind besonderes wichtig, da ein Programm typischerweise 90 Prozent seiner Ausführungszeit in 10 Prozent des Codes verbringt – im Inneren von Schleifen. Es gibt mehrere Optimierungstechniken für Schleifen, unter anderem Loop Unrolling, Loop Peeling und Loop Unswitching.

Loop Unrolling

Das Abrollen einer Schleife kann folgendermaßen aussehen:

```
n = 4;
for (i=0, i<n; i++)
  a[i] = a[i] * b + c;
```

Da die Schleife einen konstanten Iterationsraum aufweist (sie zählt immer von 0 bis 3), der zudem recht klein ausfällt, kann der Compiler alle Sprünge entfernen und die Iterationsschritte einzeln aufführen:

```
a[0] = a[0] * b + c;
a[1] = a[1] * b + c;
a[2] = a[2] * b + c;
a[3] = a[3] * b + c;
```

Bei größeren Schleifen ist Loop Unrolling nicht mehr sinnvoll, da die Cache-Effizient abnimmt und die Binaries riesig würden:

```
n = 30;
for (i=0; i<n; i++)
  a[i] = a[i] * b + c;
```

Partial Loop Unrolling

Dabei entrollt der Compiler die Schleifen nur teilweise. Als Kompromiss könnte er das Schleifeninnere dreifach hintereinander angeben – das Post-Inkrement (»i++«) erhöht den Schleifenzähler jetzt in jeder Anweisung. Wichtig ist, dass die Zahl der ausgerollten Anweisungen ein ganzzahliger Teiler der gewünschten Schleifendurchläufe ist:

```
for (i=0; i<30;) {
  a[i++] = a[i] * b + c;
  a[i++] = a[i] * b + c;
  a[i++] = a[i] * b + c;
}
```

Loop Peeling

Bei dieser Form des Loop Unrolling zieht der Compiler die ersten und/oder letzten Iterationen aus der Schleife heraus und platziert sie vor beziehungsweise nach der Schleife. Ein Beispiel:

```
for (i=0; i<n; i++) {
  if (i==0)
    x[i] = 0;
  else if (i==n)
    x[i] = n;
  else
    x[i] = x[i] * c;
}
```

Die erste If-Bedingung ist nur im ersten Schleifendurchlauf erfüllt, die zweite Bedingung nur im letzten. Das nutzt der Compiler, um die Codemenge in der Schleife zu reduzieren:

```
x[0] = 0;
for (i=1; i<n-1; i++) {
  x[i] = x[i] * c;
}
x[n] = n;
```

Loop Unswitching

Eine weitere Optimierungstechnik entfernt bedingte Sprünge aus der Schleifen, sodass der Prozessor die Sprünge nur noch einmal statt n-mal ausführen muss:

```
for (i=0; i<30; i++) {
  a[i] = x[i] * b + c;
  if (w)
    y[i] = 0;
}
```

Statt die unveränderliche Bedingung »w« in jedem Durchlauf zu testen, genügt eine If-Anweisung vor der Schleife:

```
if (w) {
  for (i=0; i<30; i++) {
    a[i] = x[i] * b + c;
    y[i] = 0;
  }
} else {
  for (i=0; i<30; i++)
    a[i] = x[i] * b + c;
}
```

Inlining

Hierbei kopiert der Compiler kleine Funktionen oder Methoden direkt in den Aufrufer und spart so mindestens das Aufrufen und Zurückkehren. Oft werden auch temporäre Objekte überflüssig sowie Konstanten besser zusammengefasst. Das reduziert ebenfalls Code.

Unit-at-a-Time

Erlaubt es dem Compiler, Optimierungen nach dem Parsen durchzuführen. Wenn er mehrere Quelldateien auf einmal übersetzen muss, darf der Compiler auch über Dateigrenzen hinweg optimieren. Bei der Codegenerierung stehen dann detailliertere Informationen bereit, die präzisere Optimierung und das Vermeiden von Dead Code erlaubt (Code, den das Programm nie ausführt).

Vektorisierung

Für automatische Vektorisierung rollt der Compiler normalerweise Schleifen ab und ersetzt Operationen durch ihr SIMD-Äquivalent (Single Instruction, Multiple Data: Die Prozessorhersteller nennen ihre Techniken MMX, SSE, Altivec, VIZ ...). Diese Vektoroperationen verarbeiten in einem Schritt gleichzeitig mehrere Werte. Intels ICC erzeugt beispielsweise aus folgender Schleife

```
for (i=0; i<400; i++) {
   r[i] = m[i] + v[i];
}
```

eine abgerollte Schleife mit zwölf SSE-Instruktionen, die 16 Additionen pro Iteration verarbeitet. Eine konventionell abgerollte Schleife schafft mit zwölf Instruktionen gerade mal vier Additionen pro Iteration.

Apple hat Interesse daran, eine weniger komplizierte Assembler-Syntax einzuführen. Sie soll dem Programmierer lästige Aufgaben abnehmen, beispielsweise „Register and Memory Clobber“, die jeder Programmierer bislang selbst ermitteln musste.

Benchmarking

Bei so vielen Veränderungen in den letzten Jahren ist die Frage besonders spannend, wie stark sich das unter realistischen Bedingungen auf die Übersetzungszeiten und die resultierenden Binaries auswirkt. Bei CPU-lastigen Benchmarks ist der SPEC CPU2000 verbreitet. Der Bench ist jedoch weder frei noch besonders kostengünstig einsetzbar, daher kamen für diesen Artikel nur Open-Source-Tools zum Einsatz. Damit ist es auch möglich, die Ergebnisse selbst nachzuprüfen.

Leider war keine fertige, für CPU- und Compiler-Messungen geeignete Open-Source-Benchmarksuite zu finden. Die bekannten Benches sind stark I/O-lastig (Iozone, Dbench). Aus Anlass dieses Artikels hat sich der Autor dazu entschlossen, einen CPU-lastigen Test zu entwickeln: Openbench [5]. Bei ihm steuern mehrere Shellskripte die Einzeltests, siehe Kasten „Ausgewählte Benchmarks“. Jedes Paket musste eine genau definierte Menge von Daten verarbeiten, um reproduzierbare Benchmark-Ergebnisse zu erzielen.

Für die nächste Zeit ist geplant, die Openbench-Ergebnisse auf Punktwerte umzustellen, ähnlich beim SPEC CPU. Damit wäre es möglich, ein mittleres Gesamtergebnisse zu berechnen. Mehr Punkte entsprächen größerer Leistung, die Ergebnisse wären intuitiv erfassbar. Weitere Teil-Benchmarks sollen hinzukommen, zum Beispiel Ogg/Vorbis. Dabei ist geplant, die Compiler mit mehr modernem C++-Code zu konfrontieren.

Testumgebung

Als Testgeräte dienten ein PC mit AMD-Athlon-Prozessor XP 2500+, 512 MByte PC333-RAM und VIA-Chipsatz KT133 sowie ein Apple I-Book 2 mit Motorolas PowerPC G3, 800 MHz Takt und 640 MByte RAM. Auf beiden Maschinen mussten mehrere GCC-Versionen unter Rock-Linux den Benchmark-Parcours meistern: GCC 2.95.3, 3.0.4, 3.2.3, 3.3.3, 3.4.0 sowie 3.5-20040523 (Pre-Release). Auf dem AMD-Prozessor kam zusätzlich Intels ICC zum Einsatz.

Die GCC-3.1-Serie blieb bewusst unberücksichtigt, weil die Entwickler sie wegen ABI-Änderungen umbenannt haben: Sie heißt jetzt Version 3.2 und diese ist in den Benches enthalten. Die Benchmark-Zeiten in den Abbildungen 2a und 2b sind aus drei Läufen gemittelt. Die Abbildungen 3a und 3b zeigen die Übersetzungszeiten (User und System), jeweils für die Intel/AMD-Plattform und für Apple/PowerPC.

Jede Spalte in den Abbildungen zeigt einen einzelnen Benchmark-Test (siehe Kasten „Ausgewählte Benchmarks“). In den Zeilen sind die verglichenen Compiler, deren Versionen und die verwendeten Optionen aufgeführt. Die Messwerte sind Laufzeit oder Übersetzungszeit in Sekunden, kürzere Balken und niedrigere Werte kennzeichnen daher bessere Ergebnisse.

Optimierungsstufen

Die einzelnen Compiler-Versionen mussten zeigen, wie stark sich Optimierungsstufen allgemein und die architekturspezifischen Optionen im Speziellen auswirken. Auf dem Athlon-Prozessor produzierte der GCC einmal generischen x86-Code und zusätzlich eine Athlon-optimierte Fassung, siehe Abbildungen 2a und 3a. Dazu dient – je nach Compiler-Version – der Schalter »-march=i686«, »-march=athlon« oder »-march=athlon-xp«. Für die PowerPC-Plattform (durch G3 gekennzeichnet, Abbildungen 2b und 3b) kam die Option »-mcpu=750« zum Einsatz. ▶

Mehrere GCCs parallel installieren

Für den Vergleichstest war es nötig, mehrere GCC-Versionen gleichzeitig zu installieren. Dabei sind einige Kleinigkeiten zu beachten. Die GCC-Programme sollten ein eindeutiges Kürzel erhalten, sodass man sie ohne »PATH«-Manipulationen auswählen und unterscheiden kann. Das gelingt per Configure-Option: Mit »./configure --program-suffix=340« lautet das GCC-Kommando »gcc340«. Die verschiedenen Versionen der Compiler müssen ihre Bibliotheken und Include-Dateien noch in einem eigenem Verzeichnis ablegen: »--enable-version-specific-runtime-libs«.

Um nur den C- und C++-Compiler des GCC zu übersetzen, gibt es je nach Version zwei Varianten. Vor Version 3 galt beispielsweise für die 2.95er:

```
./configure --prefix=/opt/gcc ➥
  --enable-version-specific-runtime-libs ➥
  --enable-threads=posix
rm -rf libobjc ibchill libf2c
make LANGUAGES="c c++" bootstrap
make LANGUAGES="c c++" install
```

Mit Version 3 ändert sich das Konfigurationsverfahren:

```
./configure --prefix=/opt/gcc ➥
  --enable-version-specific-runtime-libs ➥
  --enable-threads=posix ➥
  --enable-__cxa_atexit ➥
  --enable-languages="c,c++"
make bootstrap
make install
```

Für CVS-Checkouts und Snapshots ist »--disable-checking« wichtig, da sonst intensive GCC-interne Selbsttests die Buildtime-Benchmarks stark verfälschen. Ein Skript, das automatisch die hier verwendeten Compiler übersetzt, ist in Openbench [5] enthalten.

Per Default verwenden die Läufe als Optimierungsstufe »-O2«. Bei GCC 3.4 sind zusätzlich die Stufen »-O4« und »-O5« aufgeführt. Der Compiler selbst kennt diese Optionen nicht; in der Stufe »-O4« kam schlicht »-O2 -mfpmath = sse« zum Einsatz und statt »-O5« ist die Optionensammlung in Wahrheit: »-O2 -funroll-loops -fomit-frame-pointer -foptimize-sibling-calls -finline-all-stringops -ftracer -funit-at-a-time -funswitch-loops«. Auf dem Athlon-Prozessor diente Intels ICC 8.0.055 als Referenz (Abbildungen 2a und 3a, untere drei Zeilen).

Die Unterschiede je nach Optimierungsstufe fielen auf dem G3-Prozessor recht gering aus. Apple empfiehlt für das Mac OS X eine eigene Sammlung von Optimierungsoptionen. Für Linux adaptiert und als »-O6« bezeichnet (Abbildungen 2b und 3b) kamen sie zusätzlich zum Einsatz: »-O3 -funroll-loops -fstrict-aliasing -fsched-interblock -falign-loops = 16 -falign-jumps = 16 -falign-functions = 16 -falign-jumps-max-skip = 15 -falign-loops-max-skip = 15 -malign-natural -ffast-math -mpowerpc-gpopt -fstrict-aliasing -mcpu = 750 -mtune = 750«.

Zum Teil überraschende Ergebnisse

Einige Ergebnisse überraschen. Zum Beispiel galt GCC 2.95.3 als schneller beim Übersetzen als alle Version-3-Compiler. Interessanterweise trifft dies für den PowerPC nicht zu, dort ist GCC 3.0.4 tendenziell der schnellste im Testfeld. Außerdem zeigt sich, dass die x86-Prozessoren vermutlich mehr Aufmerksamkeit der GCC-Entwickler genießen. Die PowerPC-Ergebnisse in der Stufe »-O2« unterscheiden sich nur geringfügig voneinander.

Die größte Steigerung erzielt der Wechsel von GCC 2.95.3 auf Version 3.4 beim GnuPG-Test, der Code läuft immerhin 15 Prozent schneller (Abbildung 2b). Nur der massive Einsatz von Optimierungsoptionen bringt deutlich mehr Geschwindigkeit. Der Tramp-3D-Test läuft in der Variante »GCC-3.4.0-g3-O6« immerhin 30 Prozent schneller als mit der Optimierung »-O2«.

Auf x86 (Abbildung 2a) sieht es besser aus: Die Laufzeiten werden bei den

	Botan	Bzip2	GnuPG	Gzip	Libmad	OpenSSL	Tramp-3D
GCC-2.95.3	67.48	91.16	10.26	18.42	40.06	1.96	
GCC-2.95.3-Athlon	65.40	91.31	9.37	17.68	28.45	1.96	
GCC-3.0.4	58.00	87.36	10.56	21.51	39.22	1.86	
GCC-3.0.4-Athlon	53.48	84.27	8.98	19.52	27.63	1.88	
GCC-3.2.3	50.66	86.61	9.88	21.27	39.77	1.67	6.99
GCC-3.2.3-Athlon	49.56	89.26	9.53	15.45	26.54	1.67	7.16
GCC-3.3.3	63.65	90.96	10.43	21.01	39.31	1.70	6.41
GCC-3.3.3-Athlon	61.60	90.22	9.63	15.45	26.66	1.69	6.22
GCC-3.4.0	62.71	95.57	8.88	20.04	40.50	1.69	5.79
GCC-3.4.0-Athlon	60.94	94.38	9.38	18.95	26.44	1.66	5.94
GCC-3.4.0-Athlon-04	59.14	86.38	8.91	17.76	27.51	1.67	5.97
GCC-3.4.0-Athlon-05	52.79	87.00	9.46	17.25	26.66	1.62	5.00
GCC-3.5-20040523	75.01	90.47	8.91	19.10	42.70	1.67	5.90
GCC-3.5-20040523-Athlon	72.34	90.05	8.71	20.40	28.90	1.66	5.84
ICC	58.11	79.51	9.42	19.72	37.22	1.58	3.17
ICC-xK	57.15	79.45	10.13	17.61	37.68	1.58	3.16
ICC-xK-03	60.55	79.57	9.93	19.68	26.27	1.58	3.31

Abbildung 2a: Die Laufzeiten der einzelnen Benchmark-Programme auf einem Athlon-Prozessor unterscheiden sich je nach Compiler-Version und Optimierungsstufe teils recht stark. Der Tramp3D-Test ließ sich mit älteren Compilern nicht übersetzen, daher sind diese Felder leer.

	Botan	Bzip2	GnuPG	Gzip	Libmad	OpenSSL	Tramp-3D
GCC-2.95.3		194.29	27.31	50.06	77.66	4.21	
GCC-2.95.3-G3		188.05	27.04	52.57	77.07	4.03	
GCC-3.0.4	100.74	198.29	24.25	54.71	68.42	4.04	
GCC-3.0.4-G3	100.67	189.32	24.96	44.12	68.70	4.02	
GCC-3.2.3	102.58	192.35	23.00	53.41	69.66	4.05	16.36
GCC-3.2.3-G3	100.74	182.71	24.50	46.36	69.55	4.03	16.57
GCC-3.3.3	103.22	192.25	23.55	51.92	68.32	4.23	17.07
GCC-3.3.3-G3	103.00	185.33	25.30	48.43	68.00	4.20	17.06
GCC-3.4.0	98.72	190.81	23.12	50.34	72.69	4.21	16.56
GCC-3.4.0-G3	98.36	179.84	27.64	48.98	72.55	4.17	16.54
GCC-3.4.0-G3-05	98.98	183.71	23.10	46.95	69.91	4.08	16.50
GCC-3.4.0-G3-06	99.11	188.10	22.82	44.54	68.58	4.07	11.45
GCC-3.5-20040523	113.64	189.85	23.23	51.48	71.46	4.02	15.46
GCC-3.5-20040523-G3	113.08	179.25	24.54	52.75	71.39	3.97	15.55

Abbildung 2b: Die gleichen Benchmarks wie aus Abbildung 2a, nur diesmal auf einem Power-Prozessor (in einem Apple-Notebook) gemessen. Intels ICC unterstützt im Gegensatz zum GCC diese CPU nicht, daher fehlen die entsprechenden Zeilen.

	Botan	Bzip2	GnuPG	Gzip	Libmad	Linux	OpenSSL	Tramp-3D
GCC-2.95.3	205.21	3.13	26.44	1.36	5.48	262.54	85.34	
GCC-2.95.3-Athlon	200.39	3.26	24.79	1.34	5.42		85.04	
GCC-3.0.4	305.17	3.79	40.49	1.60	6.72	350.15	94.24	
GCC-3.0.4-Athlon	318.92	3.83	34.29	1.58	5.93		94.06	
GCC-3.2.3	349.17	4.44	36.36	1.38	7.57	417.76	107.69	789.05
GCC-3.2.3-Athlon	374.07	4.38	37.43	2.11	7.31		112.53	789.92
GCC-3.3.3	228.14	4.61	38.40	1.55	6.61	376.80	105.21	215.46
GCC-3.3.3-Athlon	225.48	4.66	33.99	2.28	6.55		109.83	214.77
GCC-3.4.0	299.64	5.42	37.92	1.53	7.82	322.35	108.96	81.71
GCC-3.4.0-Athlon	222.74	5.37	34.83	2.23	7.58		108.11	84.53
GCC-3.4.0-Athlon-04	227.33	6.53	39.83	2.51	8.77		117.31	88.64
GCC-3.4.0-Athlon-05	241.30	9.01	45.44	3.34	10.36		122.90	90.68
GCC-3.5-20040523	257.24	7.39	45.09	1.99	8.83	393.48	126.61	109.31
GCC-3.5-20040523-Athlon	239.94	7.16	41.16	2.59	8.26		125.47	141.90
ICC	305.21	4.98	25.06	1.50	6.61		113.53	174.22
ICC-xK	379.07	5.88	29.46	1.65	6.95		120.03	217.16
ICC-xK-03	280.17	6.05	28.02	1.66	6.91	2.94	124.79	224.75

Abbildung 3a: Die Zeit, die ein Compiler für seine Arbeit benötigt, schwankt je nach Version und Optimierungsstufe. Zusätzlich zu den in Abbildung 2a und 2b aufgeführten Programmen mussten GCC und ICC hier auch den Linux-Kernel übersetzen (ohne Athlon-spezifische Optimierungen).

	Botan	Bzip2	GnuPG	Gzip	Libmad	Linux	OpenSSL	Tramp-3D
GCC-2.95.3		11.84	86.59	3.77	15.79	615.54	257.15	
GCC-2.95.3-G3		11.86	79.62	5.07	16.01		266.24	
GCC-3.0.4	603.79	10.57	76.09	3.38	15.12	722.62	262.08	
GCC-3.0.4-G3	599.51	10.41	69.71	4.53	15.23		255.15	
GCC-3.2.3	673.50	14.21	112.21	4.53	19.02	929.67	330.81	2386.78
GCC-3.2.3-G3	667.75	13.95	95.31	6.42	18.88		321.87	2335.55
GCC-3.3.3	604.52	14.93	114.39	4.68	18.17	800.39	301.30	847.61
GCC-3.3.3-G3	608.27	14.63	96.73	6.54	18.26		296.48	845.93
GCC-3.4.0	603.28	15.79	106.97	4.37	20.53	671.59	296.80	202.03
GCC-3.4.0-G3	571.97	15.41	93.98	6.48	20.68		290.83	201.07
GCC-3.4.0-G3-05	620.02	25.36	127.33	10.30	29.73		332.39	226.16
GCC-3.4.0-G3-06	609.13	25.36	130.60	10.62	30.13		334.63	242.06
GCC-3.5-20040523	652.51	20.50	133.21	5.83	21.99	837.20	349.38	265.73
GCC-3.5-20040523-G3	653.44	19.60	112.36	7.75	22.09		341.63	267.99

Abbildung 3b: Die Übersetzungszeiten schwanken auch auf dem PowerPC-Gerät. Interessant: GCC 2.95.3 arbeitet auf dieser Plattform nicht immer schneller als die neueren Versionen. Architekturspezifische Optimierungen für den G3-Prozessor kamen auch hier beim Übersetzen des Linux-Kernels nicht zum Einsatz.

Benchmarks OpenSSL, Tramp-3D und GnuPG mit jeder Release kürzer. Auch Athlon-XP-spezifische Optimierungen bringen reproduzierbare Gewinne, zum Beispiel in Libmad und GnuPG. Bei anderen Tests – etwa Gzip – verbessern sich die Laufzeiten nicht weiter oder nehmen bei Bzip2 sogar wieder zu.

GCC arbeitet immer schneller

Auch bei den Übersetzungszeiten (Abbildung 2b) ist eine Tendenz zu Gunsten der aktuellen GCC-Versionen erkennbar. Gerade bei umfangreichem Sourcecode oder Template-lastigem C++ (Botan und Tramp-3D) arbeitet GCC nach den lahmem 3.0 und 3.2 in Version 3.3 und 3.4 wieder deutlich schneller.

Die Messwerte beim Übersetzen von Tramp-3D mit GCC 3.2.3 fallen stark aus dem Rahmen. GCC braucht sehr lange, um die Templates zu parsen. Wahrscheinlich spielen auch die beschriebenen konservativen Garbage-Collector-Einstellungen dieser Version eine Rolle. Aber selbst den umfangreichen Linux-Kernel übersetzt diese GCC-Version nur wenig langsamer als andere GCCs.

Code, der mit Intels C-Compiler ICC übersetzt wurde, führt nicht in allen Fällen das Testfeld an (Abbildung 2a). Gerade GnuPG scheint nicht sein Spezialgebiet zu sein, dort ist ICC so langsam wie der alte GCC 2.95.3 und wird mit Optimierung eher noch langsamer. Allerdings investieren die neuen GCC-Versionen auch deutlich mehr Zeit in die Übersetzung von GnuPG (Abbildung 3a), obwohl der ICC in anderen Benches am längsten über den Code nachdenkt.

ICC nicht immer vorn

Interessant ist, dass ICC in der getesteten Version 8.0.055 weder für OpenSSL noch für Botan korrekten Code erzeugt. Bei beiden Projekten melden die integrierten Test-Targets Fehler. Auf Anfrage teilte Intel mit, das Problem sei in der Version 8.0.66 (pe067) behoben.

Die Ergebnisse der Optimierungsstufen »-O4«, »-O5« und »-O6« belegen, dass Optimierungsorgien mit langen Optionslisten nicht immer zum Ziel führen, vielmehr müssen sie auf die CPU und das zu optimierende Programm abgestimmt sein. Während Botan, Tramp-3D oder GnuPG mit »-O5« von den gewählten Optionen profitieren, bremst »-O5« den GnuPG (Abbildung 2a) und »-O6« verlangsamt Bzip2 (Abbildung 2b).

Generell sind von Compiler-Tricks keine großen Performancesprünge zu erwarten. Bei ineffizienten Algorithmen und schlechtem Datenlayout können sie nur wenig verbessern. Beim Denken wie beim neu und besser Schreiben bleibt der Programmierer auf sich gestellt. Klar ist auch: Je mehr der Compiler den Code optimiert, desto länger muss der Entwickler auf das Ergebnis warten.

Warum die Codegenerierung auf PowerPC (Risc-Architektur) so viel weniger von maschinenspezifischen Optimierungen profitiert als auf x86 (Cisc), könnten künftige Untersuchungen des Autors mit Ultra-Sparc-Rechnern (Risc) zeigen. In sein Openbench-Projekt wird er weitere Tests integrieren. Über Vorschläge und Mitwirkende würde er sich besonders freuen. *(fjl)*

Ausgewählte Benchmarks

In die Messwerte für diesen Artikel sind folgende Einzeltests eingeflossen:

Gzip: Das Standard-Kompressionsprogramm musste ein 64 MByte großes Tar-Archiv und eine 16 MByte große Binärdatei ver- und entpacken. Als Kompressionslevel kamen »-1« und »-9« zum Einsatz (entspricht ungefähr dem Test 164.gzip in der SPEC CPU2000).

Bzip2: Äquivalent zu Gzip mit denselben Daten (ungefähr SPEC CPU2000 256.bzip2).

OpenSSL: Bei der freien SSL-Implementierung dient »openssl speed« als Benchmark.

GnuPG: Hier arbeitet das Kommando »make check« als Benchmark. Zusätzlich musste GnuPG das bei Gzip erwähnte 64-MByte-Tar-Archiv und die 16-MByte-Binärdatei ver- und wieder entschlüsseln.

Libmad: Der freie MP3-Decoder wird mit drei MP3-Dateien konfrontiert: Jazz (4,5 MByte), OpenBSD-Release-Song für Version 3.4 (6,7 MByte) und Version 3.5 (9,9 MByte).

Botan: Wie OpenSSL handelt es sich um eine freie Bibliothek von Verschlüsselungsfunktionen, allerdings ist sie inklusive Templates in recht modernem C++ implementiert. Auch hier dient »make check« als Benchmark.

Tramp-3D v3: Ein sehr Template-intensives numerisches C++-Programm.

Linux: Das Kompilieren des Kernels 2.6.6 (»vmlinux«, ohne Module) dient nur als zusätzlicher Indikator der Übersetzungszeiten. Als Architektur verwendet der Benchmark »defconfig«, daher gibt es keine Athlon- oder G3-optimierten Ergebnisse für die Build-Time.

Infos

[1] GCC: [http://gcc.gnu.org/]
[2] Änderungen in GCC 3.4: [http://gcc.gnu.org/gcc-3.4/changes.html]
[3] Intels C++-Compiler: [http://www.intel.com/software/products/compilers/clin/]
[4] SPEC: [http://www.spec.org]
[5] Openbench: [http://www.rocklinux-consulting.de/oss/openbench/]
[6] GCC Developers' Summit: [http://www.gccsummit.org/2004/]
[7] The General Sound Manipulation Program: [http://gsmp.rocklinux-consultign.de/]

Der Autor

René Rebe studiert Technische Informatik an der TFH-Berlin und kennt Linux leider erst seit 1997.

Er ist einer der Hauptentwickler bei Rock-Linux und beim Sound-Tool GSMP, arbeitet aber auch an vielen andere Projekten mit, zum Beispiel bei Sane.