Profiling mit gprof

Christof Fox, Christian Jung
Stand: 27.11.2003

Seminararbeit zur Vorlesung
„Verteilte und parallele Systeme II“
bei Prof. Rudolf Berrendorf

Fachbereich Informatik
Fachhochschule Bonn-Rhein-Sieg
Grantham-Allee 20
53757 Sankt Augustin

Inhaltsverzeichnis:

1. Einführung in gprof 3

2. Kompilieren für gprof und Ausführen von gprof 3

3. gprof Kommandos 4
   - 3.1 Ausgabeoptionen 4
   - 3.2 Optionen für die Analyse 6
   - 3.3 Weitere Optionen 7
   - 3.4 Ältere Optionen 7
   - 3.5 Symspecs 8

4. Interpretation des Outputs 8
   - 4.1 Flat-Profil 8
   - 4.2 Call-Graph 9
   - 4.3 Line-by-line Profiling 11
   - 4.4 Kommentierte Quellen Auflistung 12

5. Ungenauigkeiten von gprof´s Ausgabe 13
   - 5.1 Statistische Sampling Fehler 13
   - 5.2 Analyse der aufgerufenen „Kind-Methoden“ 14

6. Details des Profilen 15
   - 6.1 Implementation des Profilen 15
   - 6.2 Das Profile-Datenformat 16
     - 6.2.1 Histogramm 17
     - 6.2.2 CallGraph 17
     - 6.2.3 BasicBlockCount 17
   - 6.3 gprof´s Interne Operationen 18
   - 6.4 Debugging gprof 20

7. Literatur 20

1. Einführung in gprof

gprof ist ein Profiling-Programm. Mit ihm kann man die Verteilung von Laufzeiten eines Programms herausfinden. Außerdem lässt sich feststellen, welche Funktionen in einem Programm wie oft aufgerufen werden. Mit gprof ist es möglich herauszufinden, welche Funktionen eines Programms sehr viel Zeit bei der Ausführung benötigen und welche Funktionen wie oft aufgerufen werden.
gprof benutzt Informationen, die das zugehörige Laufzeitsystem während der Ausführung des Programms sammelt. Das Profiling-Programm zeigt die Methoden an, die während der Laufzeit benutzt werden. Hiermit kann man u.a. sehen, welche Methoden nicht durchlaufen werden.
Um gprof zu benutzen muss zuerst das Programm mit einer Profiling-Option kompiliert werden. Danach muss das kompilierte Programm ausgeführt werden. Nach dem Ausführen des Programms kann es mit gprof analysiert werden.

2. Kompilieren für gprof und Ausführen von gprof

Um die verwertbaren Informationen für gprof zu erhalten, muss das Programm mit einer speziellen Option kompiliert werden(„-pg“). Beispiel:

```
cc –o meinprog meinprog.c –pg
```

oder Kompilieren und Linken getrennt:

```
cc –c meinprog.c –pg
cc –o meinprog meinprog.o –pg
```

Wird das lauffähige Programm jetzt ausgeführt, erzeugt es eine Profiling-Datei („**gmon.out“)**, welche die im Programmlauf erfassten Informationen enthält. Die Informationen in gmon.out können dann von gprof ausgewertet werden. Das Programm wird wie üblich ausgeführt ohne spezielle Optionen für gprof. Es kann spezielle Argumente oder Eingaben, wie Dateinamen, etc. übergeben bekommen. Die einzige Änderung die bei der Ausführung des Programms im Profiling-Modus bemerkt wird, ist dass das Programm ein wenig langsamer ausgeführt wird, da noch eine Profiling-Datei geschrieben werden muss.
Die Profiling-Informationen werden bei jedem Ausführen des Programms überschrieben. Es gibt kein Weg eine andere Profiling-Datei zu wählen. Sollen verschiedene Profiling-Dateien gespeichert werden, empfiehlt es sich nach jedem Ausführen des Programms die Datei gmon.out umzubenennen. Geschrieben wird die gmon.out Datei in das aktuelle Verzeichnis. Dies bedeutet: Führt das Programm einen Ordnerwechsel aus, so wird die gmon.out in das Verzeichnis geschrieben, indem sich das Programm beim Beenden befindet. Die gmon.out-Datei wird erst erzeugt, wenn das Programm ordnungsgemäß beendet wird (main-Methode wird verlassen, oder exit wird aufgerufen). Wird das Programm durch andere Umstände unterbrochen oder mit _exit beendet so wird gmon.out nicht erzeugt oder überschrieben.
Werden nicht alle Module eines Programms mit der Profiling-Option kompiliert, so kann gprof nur Informationen zu den Methoden geben, welche mit –pg kompiliert wurden. Diese Möglichkeit ist sehr nützlich, um bestimmte Teile eines Programms zu untersuchen.
Um genauere Informationen über das Programm zu bekommen, welches geprüft werden soll, kann die Option –a bei der Kompilierung zusätzlich verwendet werden. Hierdurch wird die

Anzahl der Aufrufe der if-Anweisung, die Iterationen der do-Schleife etc. gezählt. Diese Option veranlasst, dass gprof die Zeit jeder Zeile im Source-Code angibt.

3. gprof Kommandos

Nachdem die Datei gmon.out erstellt wurde, kann gprof ausgeführt werden, um die Informationen der Profiling-Datei auszuwerten.

gprof [Optionen] [Ausführbare-Datei [Profiling-Dateien]] [> Outfile]

Wird keine Ausführbare-Datei angegeben sucht gprof nach a.out. Im Gegensatz zum Kompilieren ist es bei der Interpretation mit gprof möglich, eine oder mehrere Profiling-Dateien anzugeben. Wird keine Profiling-Datei angegeben wird nach gmon.out gesucht.

3.1 Ausgabeoptionen

Die folgenden Optionen veranlassen gprof, die Ausgabe in verschiedene Formate auszugeben:

-A[symspec]
--annotated-source[=symspec]
Kommentierung des Source-Codes. Bei Angabe eines symspec (s.u.) wird der Source-Code kommentiert, der im symspec definiert wurde.

-b
--brief
Ausgabe ohne Kommentare (erklärende Ausgabebeschreibung).

-C[symspec]
--exec-counts[=symspec]
Ausgabe der Anzahl der Funktionen und die Häufigkeit, wie oft diese aufgerufen wurden.

-i
--file-info
Es wird eine zusammengefasste Version ausgegeben. Histogramm, Call-Graph und die basic-block count records werden angezeigt.

-I dirs
--directory-path=dirs
Mit dieser Option kann man gprof's Suchverzeichnis für die Sourcen erweitern. Alternativ hierzu kann auch die GPROF_PATH-Variable erweitert werden.

-J[symspec]
--no-annotated-source[=symspec]
Ausgabe ohne Kommentare. Ist ein symspec angegeben, werden alle Funktionen mit Kommentare ausgegeben, bis auf die im symspec definierten.

-L
--print-path
Da Dateinamen normalerweise mit unterdrückten Pfadangabe ausgegeben werden, kann man gprof anweisen diesen aus den Debuginformationen zu entnehmen und relativ zum aktuellen Pfad mit auszugeben.

-p[symspec]
--flat-profile[=symspec]
Ausgabe in einem Flat-Profil. Ist eine symspec definiert wird nur der im symspec definierte Bereich als Flat-Profile ausgegeben.

-P[symspec]
--no-flat-profile[=symspec]
Unterdrückt die Ausgabe eines einfachen Profils. Mit symspec kann man Funktionen hiervon ausschließen.

-g[symspec]
--graph[=symspec]
Ausgabe in einem Call-Profil. Ist eine symspec definiert, wird nur der im symspec definierte Bereich als Call-Profile ausgegeben.

-Q[symspec]
--no-graph[=symspec]
Unterdrückt die Ausgabe eines Call-Profils. Mit symspec können Funktionen hiervon ausgeschlossen werden.

-y
--separate-files
Diese Option wirkt sich nur auf kommentierte Sourcen aus. Normalerweise werden kommentierte Sourcen nur auf die Standardausgabe ausgegeben. Wenn diese Option angegeben ist, werden die angemerkten Sourcen in eine Datei der Endung „-ann“ ausgegeben.

-Z[symspec]
--no-exec-counts[=symspec]
Veranlasst gprof, keine Summierung der Funktionen und deren Anzahl an Aufrufen auszugeben. Mit symspec wird diese Option auf die dort angegebenen Patterns abgebildet.

--function-ordering
Veranlasst gprof, die Funktionen nach einer vorgeschlagenen Reihenfolge zu sortieren. Genaue Details der Sortierung sind systembezogen und können deshalb hier nicht für jedes System angeben werden.

--file-ordering map_file
Diese Option ordnet die Ausgabe nach Dateinamen. Die zusätzliche Benutzung der –a Option wird empfohlen. Das map_file-Argument ist ein Pfad zu einer Datei, welche Funktionsnamen auf Dateinamen abbildet. Das Format der Datei ist ähnlich dem des nm-Befehls.
Beispiel:
c-parse.o:00000000 T yyparse
c-parse.o:00000004 C yyerrflag
c-lang.o:00000000 T maybe_objc_method_name

c-lang.o:00000000 T print_lang_statistics
c-lang.o:00000000 T recognize_objc_keyword
c-decl.o:00000000 T print_lang_identifier
c-decl.o:00000000 T print_lang_type

-T
--traditional
Veranlasst gprof die Ausgabe im traditionellen BSD-Format auszugeben.

-w
--width=width
Verändert die Weite der Ausgabezeilen. Diese Option funktioniert nur in einem Call-Graph.

-x
--all-lines
Diese Option kennzeichnet alle Zeilen eines Basisblocks. Normalerweise werden nur die Zeilen gekennzeichnet, welche am Beginn eines Basisblockes stehen.

--no-demangle
Da gprof standardgemäß die C++ Symbolnamen bei der Ausgabe berücksichtig, kann man die Option mit --no-demangle ausschalten.

3.2 Optionen für die Analyse

-a
--no-static
Unterdrückt als static deklarierten Funktionen. Die Zeiten für diese Funktionen werden dann der aufrufenden Funktion beigerechnet.

-D
--ignore-non-functions
Ignoriert Symbole, welche nicht von Funktion angesprochen werden. Dies bewirkt ein lesbareres Profil. Diese Option muss vom Betriebssystem unterstützt werden (z.B. auf Solaris oder HPUX).

-k from/to
Erlaubt das Löschen von Argumenten auf dem Call-Graphen, welche der symspec zugeordnet werden können.

-l
--line
Diese Option schaltet das Line-by-Line-Profiling ein.

-m num
--min-count=num
Diese Option unterdrückt die Ausgabe von Symbolen, welche weniger als “num”-mal aufgerufen werden.

-n[symspec]
--time[=symspec]

Veranlasst gprof, nur die Funktionen im Call-Graph zu berücksichtigen, welche zu der symspec passen.

-N[symspec]
--no-times=[symspec]
Veranlasst gprof, nur die Funktionen im Call-Graph zu berücksichtigen, welche nicht zu der symspec passen.

-z
--display-unused-functions
Veranlasst gprof, alle Funktionen im Flat-Profil anzuzeigen, auch die, die nie aufgerufen werden und die, in denen das Programm keine Zeit verbraucht hat.

3.3 Weitere Optionen

-d[num]
--debug[=num]
Spezifiziert die Debugging-Option. Wird kein Wert für num eingetragen, werden alle Debugging-Optionen eingeschaltet.

-Oname
--file-format=name
Wählt das Format für die Profil-Datei aus. Formate sind: auto (standard), bsd, magic und prof (noch nicht unterstützt).

-s
--sum
Veranlasst gprof, die Informationen in der Profil-Datei zusammenzufassen und eine neue Profil-Datei (gmon.sum) zu erstellen.
So ist es möglich Daten mehrerer Profil-Dateien in einer (gmon.sum) zu verschmelzen.

-v
--version
Ausgabe der Versionsnummer.

3.4 Ältere Optionen

Diese Optionen wurden durch neuere ausgetauscht, welche mit symspecs umgehen können

-e function_name
Veranlasst gprof, Funktionen mit bestimmten Namen und deren Kinder nicht im Call-Graph anzuzeigen. Will man mehr als eine Funktion ausschließen, so muss man die Option öfters angeben. Pro –e Option kann eine Funktion ausgeschlossen werden.

-E function_name
Diese Option arbeitet ähnlich wie die –e Option, aber die Dauer, welche das Programm in einer Funktion verbringt wird nicht benutzt, um den prozentualen Anteil der Zeit zu berechnen.

-f function_name
Mit dieser Option lässt sich der Call-Graph auf eine Funktion und deren Kinder begrenzen. Diese Option kann öfters hintereinander benutzt werden.

-F function_name
Diese Option arbeitet ähnlich wie die –f Option, aber nur die Dauer, die in der Funktion und deren Kinder gebraucht wird, wird benutzt um die gesamte Zeit und den prozentualen Anteil zu berechnen. Diese Option kann öfters hintereinander verwendet werden.

3.5 Symspecs

Viele Optionen erlauben das ein- oder ausschließen von Funktionen mittels einem symspec (Symbol Beschreibung) mit folgender Syntax:

Dateiname_mit_einem_Punkt
| Methodenname_ohne_Punkt
| Zeilennummer
| ([beliebiger_Dateiname]`:'(beliebiger_Methodenname|Zeilennummer))

Beispiele:

Test.c → Markiert die komplette Test.c-Datei. Durch den Punkt interpretiert gprof diesen symspec als Datei. Hat der Dateiname keinen Punkt, muss ein „:“ hinter der Datei angefügt werden (Bsp: „test:“)

main → Markiert die komplette main-Methode. Da es von der main-Methode mehrere Instanzen geben kann, sollte eine genauerer symspec mit Dateiname und Methodenname oder Zeilennummer angegeben werden (Bsp: Test.c:main oder Test.c:122).

4. Interpretation des Outputs

gprof beherrscht verschiedene Formate für die Ausgabe. Das einfachste Format für die Ausgabe(Dateiinformationen, Ausführungszähler, Funktions- und Dateisortierung) wurde bereits in oberem Kapitel ausführlich beschrieben. Weitere Formate sind:

4.1 Flat-Profil

Das Flat-Profil zeigt die komplette Zeit an, die das Programm in jeder Funktion verbringt. Wird die –z Option nicht benutzt, dann werden Funktion, die anscheinend nicht aufgerufen werden oder anscheinend keine Zeit verbrauchen, nicht angezeigt. Dies kann geschehen, wenn die Funktion nicht lange genug läuft um in das Programmzählerhistogramm aufgenommen zu werden. Diese Funktion ist nicht von denen unterscheidbar, die nie aufgerufen werden.

Beispiel eines Flat-Profils eines kleinen Programms:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.34      0.02     0.02     7208     0.00     0.00  open
 16.67      0.03     0.01      244     0.04     0.12  offtime
 16.67      0.04     0.01        8     1.25     1.25  memccpy
 16.67      0.05     0.01        7     1.43     1.43  write
 16.67      0.06     0.01                             mcount
  0.00      0.06     0.00      236     0.00     0.00  tzset
  0.00      0.06     0.00      192     0.00     0.00  tolower
  0.00      0.06     0.00       47     0.00     0.00  strlen
  0.00      0.06     0.00       45     0.00     0.00  strchr
  0.00      0.06     0.00        1     0.00    50.00  main
  0.00      0.06     0.00        1     0.00     0.00  memcpy
  0.00      0.06     0.00        1     0.00    10.11  print
  0.00      0.06     0.00        1     0.00     0.00  profil
  0.00      0.06     0.00        1     0.00    50.00  report
...
```

Die Funktionen sind folgendermaßen sortiert:

1. Zeit die zur Abarbeitung benötigt wird.
2. nach Anzahl, wie oft diese Aufgerufen wurden
3. Alphabetisch nach Name

Die Funktionen mcount und profil gehören nicht zu den Methoden des Programms, sondern zum Profiling-Programm und werden deshalb in jedem flat-Profil angezeigt. Das Problem beim Profiling ist, wie hier im Beispiel, die Zeit, in der ein Sample (Each sample count) gemacht wurde. Diese beträgt in unserem Beispiel 0,01 Sekunden. So sind die Zeitangaben in unserem Beispiel nicht sehr verlässlich, da die Zeiten sehr klein sind. Würde man z. B. noch einmal den Test durchlaufen, so könnte man für die Funktion open eine Laufzeit von 0.03 Sekunden oder 0.01 erhalten. Die Samplingrate ist in diesem Fall zu niedrig. Eine genauere Beschreibung zu diesem Problem finden Sie im Kapitel Statistische Sampling Fehler.

Bedeutung der einzelnen Spalten:

% time	→Prozentuale Angabe der insgesamt benötigten Zeit
cumulative seconds	→Summe der Zeit in Sekunden
Self seconds	→Zeit pro Funktion in Sekunden
Calls	→Anzahl, wie oft die Funktion aufgerufen wurde. Wurde diese Methode nie aufgerufen, so bleibt dieses Feld leer.
Self ms/call	→durchschnittliche Zeit in Millisekunden pro Aufruf der Funktion
Total ms/call	→durchschnittliche Zeit in Millisekunden pro Aufruf der Funktion und seines nachfolgenden Aufrufes.
Name	→Name der Funktion

4.2 Call-Graph

Der Call-Graph zeigt einen Abschnitt für jede Funktion. Die Bezugsfunktion ist ausgerückt (im Beispiel: start, main, report).
Über der Bezugsfunktion stehen die Funktionen, die diese aufrufen (im Beispiel: Unter [2]: „parents“: start). Darunter stehen die Funktionen, die von der Bezugsfunktion aufgerufen werden (im Beispiel: Unter [2]: „child“: report). Die Profiling-Funktion mcount wird in dieser Darstellung nicht angezeigt.

Beispiel eines Call-Graphen (Profiling-Datei ist die gleiche wie die im Flat-Profil):

```
granularity: each sample hit covers 2 byte(s) for 20.00% of 0.05 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.05                 start [1]
                0.00    0.05       1/1           main [2]
                0.00    0.00       1/2           on_exit [28]
                0.00    0.00       1/1           exit [59]
-----------------------------------------------
                0.00    0.05       1/1           start [1]
[2]    100.0    0.00    0.05       1         main [2]
                0.00    0.05       1/1           report [3]
-----------------------------------------------
                0.00    0.05       1/1           main [2]
[3]    100.0    0.00    0.05       1         report [3]
                0.00    0.03       8/8           timelocal [6]
                0.00    0.01       1/1           print [9]
                0.00    0.01       9/9           fgets [12]
                0.00    0.00      12/34          strncmp <cycle 1> [40]
                0.00    0.00       8/8           lookup [20]
                0.00    0.00       1/1           fopen [21]
                0.00    0.00       8/8           chewtime [24]
                0.00    0.00       8/16          skipspace [44]
-----------------------------------------------
[4]     59.8    0.01        0.02       8+472     <cycle 2 as a whole> [4]
                0.01        0.02     244+260         offtime <cycle 2> [7]
                0.00        0.00     236+1           tzset <cycle 2> [26]
```

Bedeutung der einzelnen Spalten:

Index →Laufende Nummer zur besseren Orientierung

% time →Prozentueller Anteil an der Gesamtlaufzeit pro Funktion

self →tatsächliche Zeit pro Funktion

children →Die komplette Zeit, die das Programm in einer Methode verbringt. Diese sollte identisch mit der Zahl in der seconds-Spalte im Flat-Profil sein.

called →Anzahl, wie oft eine Funktion aufgerufen wird. Ruft eine Funktion sich selber auf (rekursiv), so gibt es 2 Zahlen, welche mit einem „+“ getrennt werden. Die erste zählt die rekursiven Aufrufe, die zweite die nicht rekursiven. Im Beispiel wird die Funktion report einmal von der main-Funktion aufgerufen ([3]).

name →Name der Funktion

Spezielle Anzeigen für die Zeile der aufrufenden Funktionen (über der Bezugsfunktion):
Der called-Wert enthält zwei Angaben. Die erste Angabe ist die Anzahl der Aufrufe, wie oft die Funktion aufgerufen wird. Die zweite Angabe gibt die Anzahl der nicht rekursiven Aufrufe an.
Hat eine Bezugsfunktion keine aufrufenden Funktionen, so wird über der Bezugsfunktion eine <spontaneous> Zeile eingefügt (siehe Beispiel: main).
Ist die aufrufende Funktion Teil einer rekursiven Schleife, so wird die Anzahl der Zyklen nach dem Namen aufgeführt.

Spezielle Anzeigen für die Zeile der Subroutinen(unter der Bezugsfunktion):
Es wird für jede Subroutine eine Zeile unter der Bezugszeile angelegt.
Der called-Wert enthält zwei Angaben. Die erste Angabe ist die Anzahl der Aufrufe, wie oft die Bezugsfunktion aufgerufen wird. Die zweite Angabe gibt die Anzahl der nicht rekursiven Aufrufe an.

Ist die aufrufende Funktion Teil einer rekursiven Schleife, so wird die Anzahl der Zyklen nach dem Namen aufgeführt.
Ein wichtiger Punkt von gprof ist die Erkennung von Zyklen. Ein Zyklus besteht, wenn eine Funktion eine andere aufruft und diese dann direkt oder indirekt die aufrufende Funktion wieder aufruft (Bsp.: A ruft B auf und B ruft wieder A auf).
Wichtig ist dies für die Zeitangabe der Subroutinen der Funktionen. So müsste die Zeit der Subroutinen von A die Zeit der Subroutinen von B beinhalten. Diese aber besteht wiederum u. a. aus der Subroutine A. Nun stellt sich die Frage, wie viel Zeit von A hinzu addiert werden soll. Gelöst hat gprof dieses Problem durch eine extra Zeile für den Zyklus, dessen gesamte Zeit aus den einzelnen Zeiten der Funktion besteht, die am Zyklus beteiligt sind.
Außerdem werden Funktionen eines Zyklus mit <cycle x> in der name-Spalte markiert, wobei x für die Zyklusnummer steht. gprof nummeriert die Zyklen der Reihe nach durch, so ist erkennbar, welche Funktionen einen Zyklus bilden.
Für die Zeit der Subroutinen werden nur Funktionen berücksichtigt, die nicht zum Zyklus gehören.

4.3 Line-by-line Profiling

Mit der –l Option kann gprof line-by-line Profiling aktivieren. Hier wird ein Histogramm nicht nur zu den Funktionen erstellt, sondern es bezieht sich auf die einzelnen Zeilen des Quellcodes. Das Programm sollte zusätzlich mit der –g Option kompiliert werden, um Debugging-Symbole zu erstellen, die Quellcodezeilen ausfindig machen können.

Das FlatProfile ist hier die nutzvollste Art der Ausgabe. Das CallGraphProfile ist hier nicht so sinnvoll, da die neueste Version von gprof keine CallGraph-Pfeile unterstützt, die von einer beliebigen Zeile der aufrufenden Methode in die aufzurufende Methode zeigen.
Der CallGraph zeigt jedoch jede Zeile von Code, die eine Funktion aufgerufen hat, mit der Anzahl der Aufrufe.

ohne line-by-line Profiling:

Flat profile:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 30.77      0.13     0.04     6335     6.31     6.31  ct_init

                  Call graph (explanation follows)

granularity: each sample hit covers 4 byte(s) for 7.69% of 0.13 seconds

index % time    self  children    called     name

        0.00    0.00       1/13496   name_too_long
        0.00    0.00      40/13496   deflate
        0.00    0.00     128/13496   deflate_fast
        0.00    0.00   13327/13496   ct_init
[7]0.0  0.00    0.00   13496         init_block
```

hier berichtet ct_init von 4 Treffern im Histogramm, und init_block wurde 13327 mal aufgerufen.

mit line-by-line Profiling:

Die vier ct_init Histogrammtreffer werden aufgeteilt auf vier Zeilen mit den zugehörigen Zeilennummern des Programmcodes (349, 351, 382, 385). Im CallGraph werden die 13327 Aufrufe von ct_init zu init_block in einen Aufruf von Zeile 396, 3071 Aufrufe von Zeile 384, 3730 Aufrufe von Zeile 385 und 6525 Aufrufe von Zeile 387 verteilt.

Flat profile:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self
 time   seconds   seconds    calls  name
  7.69      0.10     0.01           ct_init (trees.c:349)
  7.69      0.11     0.01           ct_init (trees.c:351)
  7.69      0.12     0.01           ct_init (trees.c:382)
  7.69      0.13     0.01           ct_init (trees.c:385)
```

Call graph (explanation follows)

```
granularity: each sample hit covers 4 byte(s) for 7.69% of 0.13 seconds

  % time    self  children    called        name

            0.00    0.00          1/13496   name_too_long (gzip.c:1440)
            0.00    0.00          1/13496   deflate (deflate.c:763)
            0.00    0.00          1/13496   ct_init (trees.c:396)
            0.00    0.00          2/13496   deflate (deflate.c:727)
            0.00    0.00          4/13496   deflate (deflate.c:686)
            0.00    0.00          5/13496   deflate (deflate.c:675)
            0.00    0.00         12/13496   deflate (deflate.c:679)
            0.00    0.00         16/13496   deflate (deflate.c:730)
            0.00    0.00        128/13496   deflate_fast (deflate.c:654)
            0.00    0.00       3071/13496   ct_init (trees.c:384)
            0.00    0.00       3730/13496   ct_init (trees.c:385)
            0.00    0.00       6525/13496   ct_init (trees.c:387)
[6]   0.0   0.00    0.00      13496         init_block (trees.c:408)
```

4.4 Kommentierte Quellen Auflistung

Gprof´s –A Option erstellt eine Liste mit dem Programmcode, bei der jede Funktion mit der Anzahl der Aufrufe gekennzeichnet sind. Falls gprof den Quellcode nicht finden kann, ist es vielleicht auch nötig, die –I Option mit anzugeben.

Mit „gcc ... –g –pg –a„ wird der Code zusätzlich mit BasicBlockCounts erweitert. Hier wird es gprof ermöglicht festzuhalten, wie oft jede Zeile eines Codes aufgerufen wurde.
Beispiel:

```
 1 ulg updcrc(s, n)
 2     uch *s;
 3     unsigned n;
 4 {
 5     register ulg c;
 6
 7     static ulg crc = (ulg)0xffffffffL;
 8
 9     if (s == NULL) {
10         c = 0xffffffffL;
```

```
11    } else {
12        c = crc;
13        if (n) do {
14            c = crc_32_tab[...];
15        } while (--n);
16    }
17    crc = c;
18    return c ^ 0xffffffffL;
19 }
```

updcrc hat 5 BasicBlocks. Der erste ist die Funktion selbst. Die if Anweisung in Zeile 9 generiert zwei weitere. Ein vierter wird von der if Anweisung in der 13. Zeile erzeugt. Und der Inhalt der do Anweisung generiert den fünften Block. Der Compiler könnte noch weitere für Spezialfälle generieren.

Ein für BasicBlocks erweitertes Programm kann mit der Option gprof –l –A analysiert werden. Es wird auch empfohlen, die –X Option hinzuzufügen, welche ermöglicht, dass jede Zeile mindestens einmal gezeigt wird.
Hier steht die Kommentierte Quellen Auflistung für das obere Beispiel:

```
                    ulg updcrc(s, n)
                      uch *s;
                      unsigned n;
                2 ->{
                       register ulg c;

                       static ulg crc = (ulg)0xffffffffL;

                2 ->   if (s == NULL) {
                1 ->     c = 0xffffffffL;
                1 ->     } else {
                1 ->     c = crc;
                1 ->      if (n) do {
            26312 ->          c = crc_32_tab[...];
26312,1,26311 ->      } while (--n);
                      }
                2 ->   crc = c;
                2 ->   return c ^ 0xffffffffL;
                2 -> }
```

In diesem Beispiel wurde die Funktion zweimal aufgerufen und somit wurde je einmal jede if Klammerung besucht. Der Inhalt der do Schleife wurde 26312 mal aufgerufen. Die while Schleife wurde 26312 mal aufgerufen. Einmal wurde sie beendet und 26311 verwies sie zurück zum Begin der Schleife.

5. Ungenauigkeiten von gprof´s Ausgabe

5.1 Statistische Sampling Fehler

Die Laufzeitanalyse von gprof basiert teilweise auf einem Samplingverfahren, und beinhaltet somit statistische Ungenauigkeiten. Wenn eine Funktion nur eine sehr kurze Zeit läuft, und das Samplingverfahren diese nur einmal erkennen sollte, dann kann es sehr gut sein, dass die Funktion gar nicht oder sogar zweimal erkannt wird.

Die gezählte Anzahl der Aufrufe einer Methode und der BasicBlockCounts werden im Gegensatz dazu nicht durch Sampling sondern durch Zählen ermittelt, und haben somit keine Ungenauigkeiten.
Die Sampling-Periode wird am Anfang des FaltProfiles ausgegeben. Sie gibt an, wie oft Samples genommen werden.

Der auftretende Fehleranteil kann errechnet werden.
Für n Samples ist der erwartete Fehleranteil die Quadratische Wurzel von n. Beispielsweise gilt für eine Sampling-Periode von 0.01 Sekunden und eine Laufzeit von 1 Sekunde für eine Funktion, dass n 100 Samples entspricht (1 Sekunde / 0.01 Sekunden) und somit 10 Samples der Fehler sind. Der erwartete Fehlerwert für die Funktion ist nun 0,1 Sekunden, oder 10% der untersuchten Zeit.
Falls die Laufzeit der Funktion 100 Sekunden beträgt, und n beträgt 10000 Samples, ist der erwartete Fehler 100 Samples und somit 1% der Gesamtlaufzeit.
Normalerweise befindet sich der Fehler im Durchschnitt von mehreren Durchläufen.

Dies bedeutet nicht, dass eine kleine Laufzeit keine Informationen liefert. Wenn die Gesamtlaufzeit sehr lang ist, kann eine kurze Laufzeit einer Methode sagen, dass diese verhältnismässig bedeutungslos für die Laufzeit des kompletten Programms ist.
Normalerweise bedeutet das, dass es sich nicht lohnt diese zu optimieren.

Ein weiterer Weg ist es, Daten von mehreren Durchläufen mit der –s Option zu kombinieren:

1. Einmal das Programm laufen lassen
2. Gebe „mv gmon.out gmon.sum“ ein
3. Das Programm erneut laufen lassen
4. Zusammenfügen der neuen Daten mit gmon.sum:
   gprof –s executable-file gmon.out gmon.sum
5. Wiederhole die letzten beiden Schritte
6. Analysiere die Daten:
   gprof executable-file gmon.sum > output-file

5.2 Analyse der aufgerufenen „Kind-Methoden“

Einige der Angaben im CallGraph sind Schätzungen. Zum Beispiel die Kindmethoden-Zeitwerte und alle Zeitwerte in Aufrufzeilen und Subroutinenzeilen. Es gibt keine direkten Angaben in den erstellten Daten. Stattdessen schätzt gprof diese Zeiten.
Es ist egal wer eine bestimmte Funktionen aufruft, und somit ist die aufgerufene Funktion unabhängig von der aufrufenden Methode ist.
Wenn eine Methode 5 Sekunden benötigt, und 2/5 aller Aufrufe von einer Funktion a gekommen sind, dann bekommt a´s Kindmethodenzeit 2 Sekunden aufsummiert.

Diese Schätzung reicht für viele Programme, jedoch gibt es Ausnahmen:

Angenommen eine Methode würde je nach ihren mitgegeben Argumenten längere oder kürzere Zeit brauchen. Gprof würde die insgesammte Laufzeit der Methode gleichmässig an die aufrufenden Methoden verteilen, auch wenn diese jeweils anders beeinflussende Argumente übergeben würden.

6. Details des Profilen

6.1 Implementation des Profilen

Je nachdem, wie man den Compiler mit Argumenten steuert, werden beim Aufruf von Funktionen eines Programms Informationen abgespeichert, wodurch ermittelt wird, von wo bzw. wie oft eine Funktion in einem Programm aufgerufen wurde.
Durch die Option –pg wird beispielsweise die Funktion mcount oder je nach Betriebssystem _mcount oder __mcount benutzt.

Diese mcount Funktion, enthalten in der Profiling Bibliothek, speichert eine CallGraph-Tabelle in Hauptspeicher, in der die Funktionen und deren Elternfunktionen gespeichert werden. Dies wird erreicht, indem der „StackFrame“ nach den Adressen der Funktion und der Wiedergabeadressen der Elternfunktionen durchsucht wird.
Mcount ist eine kurze Routine, die in Assemblersprache programmiert ist. Diese ruft __mcount_internal mit den Argumenten – frompc und selfpc auf (eine normale C Funktion). Diese Funktion ist für die Erhaltung der CallGraph-Tabelle im Speicher verantwortlich. Hier werden frompc und selfpc gespeichert und die Anzahl, wie oft diese Aufrufe transferiert wurden. Selfpc ist dabei die Referenz auf die aufgerufene Funktion und frompc die Referenz auf den Aufrufer.

GCC Version 2 hat ein magische Funktion (__builtin_return_address), die einer generischen mcount Funktion das Auslesen von Informationen aus dem Stack ermöglicht. Auf einigen Systemen wie dem SPARC ist diese Funktion sehr Ressourcen einnehmend, und somit wird hier eine assemblersprachige Version von mcount genutzt.

Um Informationen über die Anzahl von Aufrufen zu bekommen, wird eine Routine einer speziellen C Bibliothek genutzt. Diese wir mittels der –pg Option eingebunden.

Profilen beinhaltet das Betrachten des Programmcounters zur Laufzeit, und dabei werden die ermittelten Daten in einem Histogramm abgespeichert. Normalerweise wird der Programm-counter alle 100 mal pro Sekunde abgerufen, dies kann aber je nach System variieren.
Dies kann auf zwei Arten realisiert werden:
Die meisten UNIX-ähnlichen System haben ein profil() SystemCall, dieser legt ein Array durch den Kernel an und einen Skalierungsfaktor der angibt wie die Programmadresse in das Array gemapt werden. Während das zu analysierende Programm läuft, wird bei jedem weiterschalten der Systemuhr der Programmcounter analysiert und der Wert des entsprechenden Slots des Arrays im Speicher wird inkrementiert. Da dies durch den Kernel geschieht, der sowieso den Prozess, wegen des Schaltens der Systemuhr, unterbrechen muss, wird somit wenig Systemoverhead erzeugt.

Von einigen Systemen, wie beispielsweise Linux 2.0 (und früheren) wird profil() nicht unterstützt. Auf diesen Systemen gibt es Abläufe, die es ermöglichen den Kernel in periodischen Zeitabständen anzusteuern (normalerweise über setitimer()), um den

Programmcounter zu analysieren, und das Array im Speicher an den entsprechenden Stellen zu inkrementieren.
Hier wird jedoch mehr Overhead benötigt, da ein Signal von der benötigten Methode geschickt werden muss, jedes mal wenn ein Sample genommen wird. Weiterhin ist diese Methode ungenauer, da das Signal, welches geschickt wird, eine Verspätung hinzufügt.

Eine spezielle Startup-Routine alloziiert Speicher für das Histogramm und ruft auch die profil() Methode oder einen clock signal handler auf. Diese Routine (monstartup) kann auf verschiedene Arten aufgerufen werden. Auf Linux Systemen wird eine spezielle Startupdatei (gcrt.o) genutzt (anstatt crt.o), welche die Methode monstartup vor der main Methode des Programms aufruft. Auf SPARC Systemen werden diese Dateien nicht genutzt. Hier wird die monstartup Methode von der mcount Routine aufgerufen, die wiederum von der main Methode aufgerufen wird.

Durch die Option –a beim Kompilieren wird BasicBlockCounts aktiviert. Hier wird ein statisches Array von Zählern angelegt und mit 0 initialisiert. Nun wird in den ausführbaren Code eine Extra-Instruktion eingefügt, die im Array entsprechende Positionen hochzählt, wenn ein BasicBlock erreicht wird. Während des Kompilierens wird ein zweipaariges Array angelegt, in denen die Startadressen der BasicBlocks gespeichert werden. Dieses Array zählt nun die Aufrufe der BasicBlocks entsprechend zu den zugehörigen Adressen.

Die Profile-Bibliothek besitzt auch ein Methode mcleanup. Diese Methode nutzt atexit(), welche wiederum aufgerufen wird, wenn das Programm endet. Mcleanup schreib dann die gmon.out Datei. Das Profilen wird dann beendet. Verschiedene Header werden ausgegeben, ein Histogramm wird geschrieben, gefolgt von dem CallGraph und den BasicBlockCounts.

Die Ausgabe von gprof gibt keine Auskunft über Programmteile, die verknüpft sind mit I/O Operationen oder Swapping Bandweite, da nur während der Laufzeit des Programms der Programmcounter analysiert wird und Samples genommen werden. Zum Beispiel kann ein Teil des Programms soviel Daten erzeugen, so dass diese geswappt werden müssen, jedoch gibt gprof hier an, dass das Programm wenig Zeit genutzt hat.

6.2 Das Profile-Datenformat

Das alte BSD Datenformat besitzt keinen sogenannten Magic Cookie, welches die Überprüfung ermöglicht, ob es eine gprof-Datei ist oder nicht. Es besitzt auch keine Versionsnummer, deshalb ist es kaum möglich, das Format durch Formatierung wiederherzustellen. GNU gprof genutzt ein neues Format, welches diese Features unterstützt. Rückwärtskompatibel ist es möglich, das alte BSD Format zu unterstützen, jedoch werden dann nicht alle Features unterstützt, z.B. wird BasicBlockCount wird vom alten Format nicht unterstützt.

Das neue Datenformat ist in gmon_out.h definiert. Es besteht aus einem Header, der den Magic Cookie und die Versionsnummer beinhaltet, und weiteren freien Bytes für zukünftige Erweiterungen. Die Dateien werden im Native Format des Hosts abgelegt, somit passt sich gprof an die genutzte Byte-Reihenfolge automatisch an.

Im neuen Datenformat folgen nach dem Header eine Reihe von Einträgen. Momentan gibt es drei verschiedene Typen: Histogramm-Einträge, CallGraph-Pfeil-Einträge, und BasicBlockCounts-Einträge. Jede Datei kann beliebig viele Einträge der drei Arten haben.

Wenn eine Datei von GNU gprof gelesen wird, kontrolliert gprof ob alle Einträge desselben Typen kompatibel sind und errechnet die Vereinigung derer. Zum Beispiel ist die Vereinigung von BasicBlockCounts die Summe aller Ausführungen des gleichen Blocks.

6.2.1 Histogramm

Diese Einträge bestehen aus einem Header, gefolgt von einem Array aus Einträgen (bins). Dieser Header beinhaltet die Textsegmentweite, auf die sich das Histogramm bezieht, die Grösse des Histogramms in Bytes, die Rate der Profileuhr, und die physikalische Dimension, welche die Einträge repräsentiert.
Diese physikalische Dimension ist in zwei Teile unterteilt: ein bis zu 15 Buchstaben lange Namen und ein einzelner Buchstabe als Abkürzung.
Z.B.: Unter DEC OSF/1 kann die Funktion uprofile() ein Histogramm erzeugen, das „Instruktion-Cache-Misses“ aufzeichnet. In diesem Fall könnte die Dimension im Header auf „i-Cache-Misses“ gesetzt werden, und die Abkürzung auf „1“ (weil es sich um ein simples Zählen und nicht um eine physikalische Dimension handelt). Weiterhin würde die Profileuhrrate auf 1 gesetzt.

6.2.2 CallGraph

Diese Einträge entsprechen dem alten BSD-Format. Es beinhaltet Pfeile des CallGraphen und die Werte, wie oft diese, während das Programm lief, aufgerufen würden. Pfeile sind durch ein Paar von zwei Adressen definiert: Die erste muss in der aufrufenden Methode sein, die zweite in der aufzurufenden Methode. Wenn gprof in der Funktions-Ebene läuft, können diese Adressen irgendwo in die Methode zeigen. Wenn jedoch die Line-by-line-Ebene läuft, ist es besser wenn die Adressen möglichst nahe am Call-Site / Entry-Point liegen. Somit ist sichergestellt, dass man weiß, welche Zeile von der aufrufenden Methode angesprochen wurde.

6.2.3 BasicBlockCount

Diese bestehen aus einem Header gefolgt von einer Sequenz von Adresse/Zähler-Paaren. Der Header gibt letztlich nur die Länge der Sequenz an. Der Adressteil identifiziert den Basic-Block und der Zählerteil gibt an, wie oft dieser Block aufgerufen wurde. Irgendeine Adresse innerhalb dieses BasicBlocks kann dafür genutzt werden.

6.3 gprof´s interne Operationen

Wie die meisten Programme beginnt gprof damit seine Optionen auszuwerten. Falls Optionen spezifiziert wurden, die Symspecs nutzen, wird eine Symspecliste angelegt (sym_ids.c:sym_id_add). Aus einer Linkedlist der Sysmspecs wird eine 12 Symboltabelle erstellt, die in sechs Include/Exclude Paare aufgeteilt wird.
Sie besteht aus je einem Paar für Flat Profile (INCL_FLAT/EXCL_FLAT),
den CallGraph (INCL_ARCS/EXCL_ARCS),
dem Schreiben in den CallGraph (INCL_GRAPH/EXCL_GRAPH),
dem Zeitübertragen in den CallGraph (INCL_TIME/EXCL_TIME),
dem Kommentierten Quellen Auflisten (INCL_ANNO/EXCL_ANNO),
und der Zählerauflisten (INCL_EXEC/EXCL_EXEC).

Nun werden alle Symspec´s mit default_excluded_list in EXCL_TIME and EXCL_GRAPH eingefügt, und falls Line-by-line Profiling spezifiziert wurde auch in EXCL_FLAT.
Danach wird die BFD Bibliothek aufgerufen, um die Objektdatei zu öffnen, und deren Symboltabelle auszulesen (core.c:core_init) mit bfd_canonicalize_symtab. Speicherplatz für ein passendes Array wird alloziiert . Wenn die Option –file-odering angegeben wurde, werden die Funktionmappings gelesen, und wenn die Option –c angegeben wurde, wird der Core Text Space eingelesen.

Gprof´s eigene Symboltabelle wird nun erstellt. Dies ist auf zwei Arten möglich, abhängig davon, ob Line-by-line Profiling aktiviert worden ist oder nicht (-l). Beim normalen Profilen wird die BFD Symboltabelle gescannt. Bei Line-by-line Profiling wird jede Text space Adresse untersucht und immer ein neuer Eintrag in die Symboltabelle geschrieben, wenn die Zeilennummer sich ändert. Beide Arten benötigen zwei Durchgänge durch die Symboltabelle, eine um die Groesse für die Symboltabelle zu ermitteln, und einen zweiten, um die Symbole einzulesen. Zwischen den zwei Durchgängen wird ein Array mit dem Typen Sym und entsprechender Länge kreiert. Letztlich wird symtab.c:symtab_finalize aufgerufen, um die Tabelle zu ordnen und Duplikate (Einträge mit derselben Speicheradresse) zu entfernen.

Die Symboltabelle muss ein kontinuierliches Array sein, weil die qsort Bibliotheksfunktion benutzt wird, und die Symbol-lookup-Routine (sym_ids.c:sym_id_parse) dieses benötigt, welche Symbole aufgrund ihrer Adresse mit einem Binary-Such-Algorithmus findet. Funktionssymbole sind mit einem isfunc gekennzeichnet. Zeilenummernsymbol haben kein spezielles Flag. Ein Symbol kann ein is_static Flag besitzen, um zu zeigen, dass es ein lokales Symbol ist.

Nachdem die Symboltabelle gelesen wurde, können die Symspecs nun in Syms transformiert werden (sym_ids:sym_id_parse), wobei ein Symspecs auf mehrere Symbole passen kann. Ein Array von Symboltabellen (sym) wird kreiert, bei dem jeder Eintrag eine Symboltabelle ist. Diese sind eingefügt nach einer bestimmten Reihenfolge. Die Haupttabelle und die Symspecs werden über verschachtelte Schleifen abgearbeitet, und jedes Symbol, das zu einem Symspec passt, wird in die entsprechende Symtabelle eingetragen. Diese wird zweimal durchgeführt. Einmal um die Anzahl der Symboltabellen zu ermitteln, und ein zweites mal, um die Tabellen zu generieren. Um nun zu schauen ob ein Symbol in eine Include- oder Exclude-Symspecliste geschrieben wird, nutzt gprof sein Standart-Lookup-Routine.

Nun werden die Profildatendateien gelesen (gmon_io.c:dmon_out_read), wobei mit Hilfe des Magic-Number-Tests kontrolliert wird, ob es ein neues oder altes Datenformat vorliegt.
Neue Histogrammeeinträge werden mit hist.c:hist_read_rec gelesen. Es wird Speicherplatz alloziiert, in den die Einträge geschrieben werden. Danach werden verschiedene Profiledaten eingelesen (oder Dateien mit verschiedenen Histogrammeinträgen). Die Startadresse, Endadresse, die Anzahl der Einträge und die Samplingrate muss zwischen den einzelnen Histgrammen übereinstimmen. Danach werden die zusätzlichen Histogramme in das Array im Speicher aufsummiert.

Jeder CallGraph-Eintrag wird gelesen (call_graph.c:cg_read_rec) und die Eltern- und Kindadressen werden in die Symboltabellen geschrieben. Weiterhin wird ein CallGraph „Pfeil“ durch cg_arcs.c:arc_add kreiert, nachdem dieser Pfeil mit einer Kontrolle der INCL_ARCS/EXCL_ARCS übereinstimmt. Nach dem Eintragen der „Pfeile“ werden Linkedlists angelegt mit den Pfeilen der Kinder und den Pfeilen der Eltern. Diese werden dann durch das Abarbeiten der Callcount-Einträge inkrementiert.

Wenn Line-by-line Profiling aktiviert wurde, werden nun die BasicBlocks eingelesen (basic_block.c:bb_read_rec). Jede BasicBlock-Adresse gehört zu einem Linesymbol in der Symboltabelle, und ein Eintrag wird in die entsprechenden bb_addr und bb_calls Arrays gemacht. Wenn mehrere BasicBlock-Einträge existieren, werden die einsprechenden Werte inkrementiert.

Die gmon.sum Datei wird, falls erwünscht weggeworfen (gmon.io.c:gmon_out_write).
Wenn Histogramme in den Daten vorhanden waren, werden diese durch Überarbeiten der Einträge in Symbol gewandelt (hist.c:hist_assign_samples). Wenn die Symboltabelle nach aufsteigenden Adressen sortiert ist, können diese einfach nacheinander betrachtet werden. Dabei werden die Einträge mit den INCL_FLAT/EXCL_FLAT geprüft. Abhängig von dem verschiedenen Skalierungsfaktoren kann ein Eintrag mehreren Symbolen zugeordnet werden. Dabei werden Teile des Counts an diese Symbole verteilt, proportional zum Grad des Überlappens. Diese Überlappungen sind eher selten bei normalem Profilen, können aber bei Line-by-line Profilen auftreten.

Wenn CallGraph-Daten vorhanden sind, wird cg_arcs.c:cg_assemble aufgerufen. Falls –c spezifiziert wurde, wir eine maschinenabhängige Routine (find_call) aufgerufen, die durch den Symbol Maschinencode scannt, um Subroutinenaufrufe zu finden, und diese zu den Call Graphen zu addieren und auf 0 zu initialisieren. Ein topologische Sortierung findet statt, mit depth-first Nummerierung der Symbole(cg_dfn.c:cg_dfn), so dass die Kinder immer kleinere Nummern haben als ihre Eltern, wonach ein Array von Zeiger in die Symboltabelle eingefügt wird, und diese nach numerischer Reihenfolge sortiert wird.(somit entsteht eine inverse topologische Sortierung). Alle, die eine gleiche topologische Nummer haben, sind Circles.

Nun werden zwei Durchgänge durch dieses Zeigerarray gemacht. Der erste, vom Ende zum Anfang (Eltern zu Kinder), errechnet den Anteil zum entsprechenden Elternteil und erstellt einen Printflag. Das Printflag gibt an, ob die Eltern gedruckt werden oder nicht, abhängig davon, ob die Kinder in INCL_GRAPH oder EXCL_GRAPH sind. Beim zweiten Durchgang von Anfang zum Ende (Kinder zu Eltern) werden die Zeiten des CallGraphs ermittelt, die wiederum mit INCL_TIME/EXCL_TIME geprüft werden. Nachdem die Printflags und Zeiten in der Symbolstruktur gespeichert sind, wird das topologische Array weggeschmissen und ein neues Array von Zeigern erstellt, dieses mal sortiert nach aufsteigender Zeit.

Letztlich werden die vom Benutzer gewünschten Daten ausgegeben. Die CallGraph-Daten (cg_print.c:cg_print) und die FlatProfile-Daten (hist.c:hist_print) können direkt ausgegeben. Die kommentierte Quellenauflistung (basic_blocks.c:print_annotated_source) nutzt die BasicBlock-Informationen, falls vorhanden, um jede Zeile des Codes mit CallCounts zu zählen, ansonsten werden nur die Funktions-CallCounts ausgegeben.

Cg_print.c gibt die Funktionsaufrufe an. Die Funktionen die am meisten genutzt werden und am meisten Eltern haben werden als erstes positioniert, gefolgt von Funktionen die am meissten genutzt werden und am Ende Funktionen die gar nicht genutzt werden.

6.4 Debugging gprof

Wenn Debugging mit der –d Option beim Kompilieren aktiviert wurde, dann wird Debugging-Output auf studout geschrieben. Die Debuggingnummer ist eine Summe der folgenden aufsummierten Optionen:

2 – Topologische Sortierung
Zeigt depth-first nummeriert, sortierte Symbole während der CallGragh-Analyse
4 - Cycles
Zeigt Symbol die als Kopf eine Cycles identifiziert werden
16 – Übereinstimmungen
Während die CallGraph-"Pfeile" gelesen werden, zeige jeden Pfeil und alle Aufrufe die übereinstimmen.
32 – CallGraph-"Pfeil" Sortierung
Details der Sortierung von individuellen Eltern/Kinden innerhalb jedes CallGraph-Eintrags
64 – Lesen der Histogramm- und CallGraph-Einträge
Zeigt Adressraum jedes Histogramms, wenn es gelesen wird, und jeden CallGraphPfeil
128 - Symboltabelle
Lesen, Analysieren und Sortieren der Symboltabelle der Objektdateien.
Bei Line-by-line Profilen werden auch die Zeilennummern aus den Speicheradressen angegeben
256 - Statischer CallGraph
Verfolgt Operationen der -c Option
512 - Symboltabellen und Pfeiltabellen lookups
Details der Lookup-Routine-Operation
1024 - CallGaph Verteilung
Zeigt wie sich Funktionszeiten über den CallGraph verteilen
2048 - Basic-Bocks
Zeigt Basic-Block-Einträge die aus Profiledaten gelesen wurden (-l Option)
4096 - Symspecs
Zeigt wie Symspecs auf Symbole bezogen werden
8192 - Kommentierte Quelle
Zeigt Operationen der -A Option

6. Literatur

- Fenlaso, Jay; Stallman, Richard: GNU gprof - The GNU Profiler, http://www.gnu.org/software/binutils/manual/gprof-2.9.1/html_mono/gprof.html, 2003
- Bruchhäuser, Gerrit: gprof, http://www.gnu.org/manual/gprof-2.9.1/gprof.html, 2003