GNU gprof

The GNU Profiler
(GNU Binutils)
Version 2.45

Jay Fenlason and Richard Stallman

This manual describes the GNU profiler, `gprof`, and how you can use it to determine which parts of a program are taking most of the execution time. We assume that you know how to write, compile, and execute programs. GNU `gprof` was written by Jay Fenlason. Eric S. Raymond made some minor corrections and additions in 2003.

Copyright © 1988-2025 Free Software Foundation, Inc.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Table of Contents

1 Introduction to Profiling 1

2 Compiling a Program for Profiling 3

3 Executing the Program 5

4 gprof Command Summary 7
- 4.1 Output Options 7
- 4.2 Analysis Options 10
- 4.3 Miscellaneous Options 12
- 4.4 Deprecated Options 13
- 4.5 Symspecs 14

5 Interpreting gprof's Output 15
- 5.1 The Flat Profile 15
- 5.2 The Call Graph 17
  - 5.2.1 The Primary Line 18
  - 5.2.2 Lines for a Function's Callers 19
  - 5.2.3 Lines for a Function's Subroutines 19
  - 5.2.4 How Mutually Recursive Functions Are Described 20
- 5.3 Line-by-line Profiling 22
- 5.4 The Annotated Source Listing 24

6 Inaccuracy of gprof Output 27
- 6.1 Statistical Sampling Error 27
- 6.2 Estimating `children` Times 28

7 Answers to Common Questions 29

8 Incompatibilities with Unix gprof 31

9 Details of Profiling 33
- 9.1 Implementation of Profiling 33
- 9.2 Profiling Data File Format 34
  - 9.2.1 Histogram Records 35
  - 9.2.2 Call-Graph Records 35
  - 9.2.3 Basic-Block Execution Count Records 36
- 9.3 gprof's Internal Operation 36
- 9.4 Debugging gprof 39

Appendix A GNU Free Documentation License 41

1 Introduction to Profiling

Profiling allows you to learn where your program spent its time and which functions called which other functions while it was executing. This information can show you which pieces of your program are slower than you expected, and might be candidates for rewriting to make your program execute faster. It can also tell you which functions are being called more or less often than you expected. This may help you spot bugs that had otherwise been unnoticed.

Since the profiler uses information collected during the actual execution of your program, it can be used on programs that are too large or too complex to analyze by reading the source. However, how your program is run will affect the information that shows up in the profile data. If you don't use some feature of your program while it is being profiled, no profile information will be generated for that feature.

Profiling has several steps:

- You must compile and link your program with profiling enabled. See Chapter 2 [Compiling a Program for Profiling], page 3.
- You must execute your program to generate a profile data file. See Chapter 3 [Executing the Program], page 5.
- You must run `gprof` to analyze the profile data. See Chapter 4 [`gprof` Command Summary], page 7.

The next three chapters explain these steps in greater detail.

Several forms of output are available from the analysis.

The *flat profile* shows how much time your program spent in each function, and how many times that function was called. If you simply want to know which functions burn most of the cycles, it is stated concisely here. See Section 5.1 [The Flat Profile], page 15.

The *call graph* shows, for each function, which functions called it, which other functions it called, and how many times. There is also an estimate of how much time was spent in the subroutines of each function. This can suggest places where you might try to eliminate function calls that use a lot of time. See Section 5.2 [The Call Graph], page 17.

The *annotated source* listing is a copy of the program's source code, labeled with the number of times each line of the program was executed. See Section 5.4 [The Annotated Source Listing], page 24.

To better understand how profiling works, you may wish to read a description of its implementation. See Section 9.1 [Implementation of Profiling], page 33.

2 Compiling a Program for Profiling

The first step in generating profile information for your program is to compile and link it with profiling enabled.

To compile a source file for profiling, specify the '`-pg`' option when you run the compiler. (This is in addition to the options you normally use.)

To link the program for profiling, if you use a compiler such as `cc` to do the linking, simply specify '`-pg`' in addition to your usual options. The same option, '`-pg`', alters either compilation or linking to do what is necessary for profiling. Here are examples:

```
cc -g -c myprog.c utils.c -pg
cc -o myprog myprog.o utils.o -pg
```

The '`-pg`' option also works with a command that both compiles and links:

```
cc -o myprog myprog.c utils.c -g -pg
```

Note: The '`-pg`' option must be part of your compilation options as well as your link options. If it is not then no call-graph data will be gathered and when you run `gprof` you will get an error message like this:

```
gprof: gmon.out file is missing call-graph data
```

If you add the '`-Q`' switch to suppress the printing of the call graph data you will still be able to see the time samples:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 44.12      0.07     0.07                             zazLoop
 35.29      0.14     0.06                             main
 20.59      0.17     0.04                             bazMillion
```

If you run the linker `ld` directly instead of through a compiler such as `cc`, you may have to specify a profiling startup file '`gcrt0.o`' as the first input file instead of the usual startup file '`crt0.o`'. In addition, you would probably want to specify the profiling C library, '`libc_p.a`', by writing '`-lc_p`' instead of the usual '`-lc`'. This is not absolutely necessary, but doing this gives you number-of-calls information for standard library functions such as `read` and `open`. For example:

```
ld -o myprog /lib/gcrt0.o myprog.o utils.o -lc_p
```

If you are running the program on a system which supports shared libraries you may run into problems with the profiling support code in a shared library being called before that library has been fully initialised. This is usually detected by the program encountering a segmentation fault as soon as it is run. The solution is to link against a static version of the library containing the profiling support code, which for `gcc` users can be done via the '`-static`' or '`-static-libgcc`' command-line option. For example:

```
gcc -g -pg -static-libgcc myprog.c utils.c -o myprog
```

If you compile only some of the modules of the program with '`-pg`', you can still profile the program, but you won't get complete information about the modules that were compiled without '`-pg`'. The only information you get for the functions in those modules is the total time spent in them; there is no record of how many times they were called, or from where. This will not affect the flat profile (except that the `calls` field for the functions will be blank), but will greatly reduce the usefulness of the call graph.

If you wish to perform line-by-line profiling you should use the `gcov` tool instead of `gprof`. See that tool's manual or info pages for more details of how to do this.

Note, older versions of `gcc` produce line-by-line profiling information that works with `gprof` rather than `gcov` so there is still support for displaying this kind of information in `gprof`. See Section 5.3 [Line-by-line Profiling], page 22.

It also worth noting that `gcc` implements a '`-finstrument-functions`' command-line option which will insert calls to special user supplied instrumentation routines at the entry and exit of every function in their program. This can be used to implement an alternative profiling scheme.

3 Executing the Program

Once the program is compiled for profiling, you must run it in order to generate the information that `gprof` needs. Simply run the program as usual, using the normal arguments, file names, etc. The program should run normally, producing the same output as usual. It will, however, run somewhat slower than normal because of the time spent collecting and writing the profile data.

The way you run the program—the arguments and input that you give it—may have a dramatic effect on what the profile information shows. The profile data will describe the parts of the program that were activated for the particular input you use. For example, if the first command you give to your program is to quit, the profile data will show the time used in initialization and in cleanup, but not much else.

Your program will write the profile data into a file called '`gmon.out`' just before exiting. If there is already a file called '`gmon.out`', its contents are overwritten. You can rename the file afterwards if you are concerned that it may be overwritten. If your system libc allows you may be able to write the profile data under a different name. Set the GMON_OUT_PREFIX environment variable; this name will be appended with the PID of the running program.

In order to write the '`gmon.out`' file properly, your program must exit normally: by returning from `main` or by calling `exit`. Calling the low-level function `_exit` does not write the profile data, and neither does abnormal termination due to an unhandled signal.

The '`gmon.out`' file is written in the program's *current working directory* at the time it exits. This means that if your program calls `chdir`, the '`gmon.out`' file will be left in the last directory your program `chdir`'d to. If you don't have permission to write in this directory, the file is not written, and you will get an error message.

Older versions of the GNU profiling library may also write a file called '`bb.out`'. This file, if present, contains an human-readable listing of the basic-block execution counts. Unfortunately, the appearance of a human-readable '`bb.out`' means the basic-block counts didn't get written into '`gmon.out`'. The Perl script `bbconv.pl`, included with the `gprof` source distribution, will convert a '`bb.out`' file into a format readable by `gprof`. Invoke it like this:

```
bbconv.pl < bb.out > bh-data
```

This translates the information in '`bb.out`' into a form that `gprof` can understand. But you still need to tell `gprof` about the existence of this translated information. To do that, include *bb-data* on the `gprof` command line, *along with* '`gmon.out`', like this:

```
gprof options executable-file gmon.out bb-data [yet-more-profile-data-files...] [> out
file]
```

4 gprof Command Summary

After you have a profile data file 'gmon.out', you can run gprof to interpret the information in it. The gprof program prints a flat profile and a call graph on standard output. Typically you would redirect the output of gprof into a file with '>'.

You run gprof like this:

```
gprof options [executable-file [profile-data-files...]] [> outfile]
```

Here square-brackets indicate optional arguments.

If you omit the executable file name, the file 'a.out' is used. If you give no profile data file name, the file 'gmon.out' is used. If any file is not in the proper format, or if the profile data file does not appear to belong to the executable file, an error message is printed.

You can give more than one profile data file by entering all their names after the executable file name; then the statistics in all the data files are summed together.

The order of these options does not matter.

4.1 Output Options

These options specify which of several output formats gprof should produce.

Many of these options take an optional *symspec* to specify functions to be included or excluded. These options can be specified multiple times, with different symspecs, to include or exclude sets of symbols. See Section 4.5 [Symspecs], page 14.

Specifying any of these options overrides the default ('-p -q'), which prints a flat profile and call graph analysis for all functions.

`-A[`*symspec*`]`
`--annotated-source[=`*symspec*`]`
: The '-A' option causes gprof to print annotated source code. If *symspec* is specified, print output only for matching symbols. See Section 5.4 [The Annotated Source Listing], page 24.

`-b`
`--brief`
: If the '-b' option is given, gprof doesn't print the verbose blurbs that try to explain the meaning of all of the fields in the tables. This is useful if you intend to print out the output, or are tired of seeing the blurbs.

`-B`
: The '-B' option causes gprof to print the call graph analysis.

`-C[`*symspec*`]`
`--exec-counts[=`*symspec*`]`
: The '-C' option causes gprof to print a tally of functions and the number of times each was called. If *symspec* is specified, print tally only for matching symbols.

If the profile data file contains basic-block count records, specifying the '`-l`' option, along with '`-C`', will cause basic-block execution counts to be tallied and displayed.

`-i`
`--file-info`
: The '`-i`' option causes `gprof` to display summary information about the profile data file(s) and then exit. The number of histogram, call graph, and basic-block count records is displayed.

`-I` *dirs*
`--directory-path=`*dirs*
: The '`-I`' option specifies a list of search directories in which to find source files. Environment variable *GPROF_PATH* can also be used to convey this information. Used mostly for annotated source output.

`-J[`*symspec*`]`
`--no-annotated-source[=`*symspec*`]`
: The '`-J`' option causes `gprof` not to print annotated source code. If *symspec* is specified, `gprof` prints annotated source, but excludes matching symbols.

`-L`
`--print-path`
: Normally, source filenames are printed with the path component suppressed. The '`-L`' option causes `gprof` to print the full pathname of source filenames, which is determined from symbolic debugging information in the image file and is relative to the directory in which the compiler was invoked.

`-p[`*symspec*`]`
`--flat-profile[=`*symspec*`]`
: The '`-p`' option causes `gprof` to print a flat profile. If *symspec* is specified, print flat profile only for matching symbols. See Section 5.1 [The Flat Profile], page 15.

`-P[`*symspec*`]`
`--no-flat-profile[=`*symspec*`]`
: The '`-P`' option causes `gprof` to suppress printing a flat profile. If *symspec* is specified, `gprof` prints a flat profile, but excludes matching symbols.

`-q[`*symspec*`]`
`--graph[=`*symspec*`]`
: The '`-q`' option causes `gprof` to print the call graph analysis. If *symspec* is specified, print call graph only for matching symbols and their children. See Section 5.2 [The Call Graph], page 17.

`-Q[`*symspec*`]`
`--no-graph[=`*symspec*`]`
: The '`-Q`' option causes `gprof` to suppress printing the call graph. If *symspec* is specified, `gprof` prints a call graph, but excludes matching symbols.

`-t`
`--table-length=`*num*
: The '`-t`' option causes the *num* most active source lines in each source file to be listed when source annotation is enabled. The default is 10.

`-y`
`--separate-files`
: This option affects annotated source output only. Normally, `gprof` prints annotated source files to standard-output. If this option is specified, annotated source for a file named '`path/`*`filename`*' is generated in the file '*`filename`*`-ann`'. If the underlying file system would truncate '*`filename`*`-ann`' so that it overwrites the original '*`filename`*', `gprof` generates annotated source in the file '*`filename`*`.ann`' instead (if the original file name has an extension, that extension is *replaced* with '`.ann`').

`-Z[`*symspec*`]`
`--no-exec-counts[=`*symspec*`]`
: The '`-Z`' option causes `gprof` not to print a tally of functions and the number of times each was called. If *symspec* is specified, print tally, but exclude matching symbols.

`-r`
`--function-ordering`
: The '`--function-ordering`' option causes `gprof` to print a suggested function ordering for the program based on profiling data. This option suggests an ordering which may improve paging, tlb and cache behavior for the program on systems which support arbitrary ordering of functions in an executable.

  The exact details of how to force the linker to place functions in a particular order is system dependent and out of the scope of this manual.

`-R` *map_file*
`--file-ordering` *map_file*
: The '`--file-ordering`' option causes `gprof` to print a suggested .o link line ordering for the program based on profiling data. This option suggests an ordering which may improve paging, tlb and cache behavior for the program on systems which do not support arbitrary ordering of functions in an executable.

  Use of the '`-a`' argument is highly recommended with this option.

The *map_file* argument is a pathname to a file which provides function name to object file mappings. The format of the file is similar to the output of the program `nm`.

```
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
...
```

To create a *map_file* with GNU `nm`, type a command like *`nm --extern-only --defined-only -v --print-file-name program-name`*.

`-T`
`--traditional`
: The '`-T`' option causes `gprof` to print its output in "traditional" BSD style.

`-w` *width*
`--width=`*width*
: Sets width of output lines to *width*. Currently only used when printing the function index at the bottom of the call graph.

`-x`
`--all-lines`
: This option affects annotated source output only. By default, only the lines at the beginning of a basic-block are annotated. If this option is specified, every line in a basic-block is annotated by repeating the annotation for the first line. This behavior is similar to `tcov`'s '`-a`'.

`--demangle[=`*style*`]`
`--no-demangle`
: These options control whether C++ symbol names should be demangled when printing output. The default is to demangle symbols. The `--no-demangle` option may be used to turn off demangling. Different compilers have different mangling styles. The optional demangling style argument can be used to choose an appropriate demangling style for your compiler.

4.2 Analysis Options

`-a`
`--no-static`
: The '`-a`' option causes `gprof` to suppress the printing of statically declared (private) functions. (These are functions whose

names are not listed as global, and which are not visible outside the file/function/block where they were defined.) Time spent in these functions, calls to/from them, etc., will all be attributed to the function that was loaded directly before it in the executable file. This option affects both the flat profile and the call graph.

`-c`
`--static-call-graph`
: The '`-c`' option causes the call graph of the program to be augmented by a heuristic which examines the text space of the object file and identifies function calls in the binary machine code. Since normal call graph records are only generated when functions are entered, this option identifies children that could have been called, but never were. Calls to functions that were not compiled with profiling enabled are also identified, but only if symbol table entries are present for them. Calls to dynamic library routines are typically *not* found by this option. Parents or children identified via this heuristic are indicated in the call graph with call counts of '`0`'.

`-D`
`--ignore-non-functions`
: The '`-D`' option causes `gprof` to ignore symbols which are not known to be functions. This option will give more accurate profile data on systems where it is supported (Solaris and HPUX for example).

`-k` *from/to*
: The '`-k`' option allows you to delete from the call graph any arcs from symbols matching symspec *from* to those matching symspec *to*.

`-l`
`--line`
: The '`-l`' option enables line-by-line profiling, which causes histogram hits to be charged to individual source code lines, instead of functions. This feature only works with programs compiled by older versions of the `gcc` compiler. Newer versions of `gcc` are designed to work with the `gcov` tool instead.

  If the program was compiled with basic-block counting enabled, this option will also identify how many times each line of code was executed. While line-by-line profiling can help isolate where in a large function a program is spending its time, it also significantly increases the running time of `gprof`, and magnifies statistical inaccuracies. See Section 6.1 [Statistical Sampling Error], page 27.

`--inline-file-names`
: This option causes `gprof` to print the source file after each symbol in both the flat profile and the call graph. The full path to the file is printed if used with the '`-L`' option.

`-m` *num*
`--min-count=`*num*
: This option affects execution count output only. Symbols that are executed less than *num* times are suppressed.

`-n`*symspec*
`--time=`*symspec*
: The '`-n`' option causes `gprof`, in its call graph analysis, to only propagate times for symbols matching *symspec*.

`-N`*symspec*
`--no-time=`*symspec*
: The '`-n`' option causes `gprof`, in its call graph analysis, not to propagate times for symbols matching *symspec*.

`-S`*filename*
`--external-symbol-table=`*filename*
: The '`-S`' option causes `gprof` to read an external symbol table file, such as '`/proc/kallsyms`', rather than read the symbol table from the given object file (the default is `a.out`). This is useful for profiling kernel modules.

`-z`
`--display-unused-functions`
: If you give the '`-z`' option, `gprof` will mention all functions in the flat profile, even those that were never called, and that had no time spent in them. This is useful in conjunction with the '`-c`' option for discovering which routines were never called.

4.3 Miscellaneous Options

`-d[`*num*`]`
`--debug[=`*num*`]`
: The '`-d` *num*' option specifies debugging options. If *num* is not specified, enable all debugging. See Section 9.4 [Debugging gprof], page 39.

`-h`
`--help` The '`-h`' option prints command line usage.

`-O`*name*
`--file-format=`*name*
: Selects the format of the profile data files. Recognized formats are '`auto`' (the default), '`bsd`', '`4.4bsd`', '`magic`', and '`prof`' (not yet supported).

`-s`
`--sum` The '`-s`' option causes `gprof` to summarize the information in the profile data files it read in, and write out a profile data file called '`gmon.sum`', which contains all the information from the profile data files that `gprof` read in. The file '`gmon.sum`' may be one of the specified input files; the effect of this is to merge the data in the other input files into '`gmon.sum`'.

Eventually you can run `gprof` again without '`-s`' to analyze the cumulative data in the file '`gmon.sum`'.

`-v`
`--version`
The '`-v`' flag causes `gprof` to print the current version number, and then exit.

4.4 Deprecated Options

These options have been replaced with newer versions that use symspecs.

`-e` *function_name*
The '`-e` *function*' option tells `gprof` to not print information about the function *function_name* (and its children...) in the call graph. The function will still be listed as a child of any functions that call it, but its index number will be shown as '`[not printed]`'. More than one '`-e`' option may be given; only one *function_name* may be indicated with each '`-e`' option.

`-E` *function_name*
The `-E` *function* option works like the `-e` option, but time spent in the function (and children who were not called from anywhere else), will not be used to compute the percentages-of-time for the call graph. More than one '`-E`' option may be given; only one *function_name* may be indicated with each '`-E`' option.

`-f` *function_name*
The '`-f` *function*' option causes `gprof` to limit the call graph to the function *function_name* and its children (and their children...). More than one '`-f`' option may be given; only one *function_name* may be indicated with each '`-f`' option.

`-F` *function_name*
The '`-F` *function*' option works like the `-f` option, but only time spent in the function and its children (and their children...) will be used to determine total-time and percentages-of-time for the call graph. More than one '`-F`' option may be given; only one *function_name* may be indicated with each '`-F`' option. The '`-F`' option overrides the '`-E`' option.

Note that only one function can be specified with each `-e`, `-E`, `-f` or `-F` option. To specify more than one function, use multiple options. For example, this command:

```
gprof -e boring -f foo -f bar myprogram > gprof.output
```

lists in the call graph all functions that were reached from either `foo` or `bar` and were not reachable from `boring`.

4.5 Symspecs

Many of the output options allow functions to be included or excluded using *symspecs* (symbol specifications), which observe the following syntax:

```
    filename_containing_a_dot
  | funcname_not_containing_a_dot
  | linenumber
  | ( [ any_filename ] ':' ( any_funcname | linenumber ) )
```

Here are some sample symspecs:

'`main.c`' Selects everything in file '`main.c`'—the dot in the string tells `gprof` to interpret the string as a filename, rather than as a function name. To select a file whose name does not contain a dot, a trailing colon should be specified. For example, '`odd:`' is interpreted as the file named '`odd`'.

'`main`' Selects all functions named '`main`'.

Note that there may be multiple instances of the same function name because some of the definitions may be local (i.e., static). Unless a function name is unique in a program, you must use the colon notation explained below to specify a function from a specific source file.

Sometimes, function names contain dots. In such cases, it is necessary to add a leading colon to the name. For example, '`:.mul`' selects function '`.mul`'.

In some object file formats, symbols have a leading underscore. `gprof` will normally not print these underscores. When you name a symbol in a symspec, you should type it exactly as `gprof` prints it in its output. For example, if the compiler produces a symbol '`_main`' from your `main` function, `gprof` still prints it as '`main`' in its output, so you should use '`main`' in symspecs.

'`main.c:main`'
Selects function '`main`' in file '`main.c`'.

'`main.c:134`'
Selects line 134 in file '`main.c`'.

5 Interpreting gprof's Output

gprof can produce several different output styles, the most important of which are described below. The simplest output styles (file information, execution count, and function and file ordering) are not described here, but are documented with the respective options that trigger them. See Section 4.1 [Output Options], page 7.

5.1 The Flat Profile

The *flat profile* shows the total amount of time your program spent executing each function. Unless the '-z' option is given, functions with no apparent time spent in them, and no apparent calls to them, are not mentioned. Note that if a function was not compiled for profiling, and didn't run long enough to show up on the program counter histogram, it will be indistinguishable from a function that was never called.

This is part of a flat profile for a small program:

```
Flat profile:

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
...
```

The functions are sorted first by decreasing run-time spent in them, then by decreasing number of calls, then alphabetically by name. The functions 'mcount' and 'profil' are part of the profiling apparatus and appear in every flat profile; their time gives a measure of the amount of overhead due to profiling.

Just before the column headers, a statement appears indicating how much time each sample counted as. This *sampling period* estimates the margin of error in each of the time figures. A time figure that is not much larger than this is not reliable. In this example, each sample counted as 0.01 seconds, suggesting a 100 Hz sampling rate. The program's total execution time was 0.06 seconds, as indicated by the 'cumulative seconds' field. Since each

sample counted for 0.01 seconds, this means only six samples were taken during the run. Two of the samples occurred while the program was in the '`open`' function, as indicated by the '`self seconds`' field. Each of the other four samples occurred one each in '`offtime`', '`memccpy`', '`write`', and '`mcount`'. Since only six samples were taken, none of these values can be regarded as particularly reliable. In another run, the '`self seconds`' field for '`mcount`' might well be '`0.00`' or '`0.02`'. See Section 6.1 [Statistical Sampling Error], page 27, for a complete discussion.

The remaining functions in the listing (those whose '`self seconds`' field is '`0.00`') didn't appear in the histogram samples at all. However, the call graph indicated that they were called, so therefore they are listed, sorted in decreasing order by the '`calls`' field. Clearly some time was spent executing these functions, but the paucity of histogram samples prevents any determination of how much time each took.

Here is what the fields in each line mean:

`% time`
: This is the percentage of the total execution time your program spent in this function. These should all add up to 100%.

`cumulative seconds`
: This is the cumulative total number of seconds the computer spent executing this functions, plus the time spent in all the functions above this one in this table.

`self seconds`
: This is the number of seconds accounted for by this function alone. The flat profile listing is sorted first by this number.

`calls`
: This is the total number of times the function was called. If the function was never called, or the number of times it was called cannot be determined (probably because the function was not compiled with profiling enabled), the *calls* field is blank.

`self ms/call`
: This represents the average number of milliseconds spent in this function per call, if this function is profiled. Otherwise, this field is blank for this function.

`total ms/call`
: This represents the average number of milliseconds spent in this function and its descendants per call, if this function is profiled. Otherwise, this field is blank for this function. This is the only field in the flat profile that uses call graph analysis.

`name`
: This is the name of the function. The flat profile is sorted by this field alphabetically after the *self seconds* and *calls* fields are sorted.

5.2 The Call Graph

The *call graph* shows how much time was spent in each function and its children. From this information, you can find functions that, while they themselves may not have used much time, called other functions that did use unusual amounts of time.

Here is a sample call from a small program. This call came from the same `gprof` run as the flat profile example in the previous section.

```
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]
-----------------------------------------------
```

The lines full of dashes divide this table into *entries*, one for each function. Each entry has one or more lines.

In each entry, the primary line is the one that starts with an index number in square brackets. The end of this line says which function the entry is for. The preceding lines in the entry describe the callers of this function and the following lines describe its subroutines (also called *children* when we speak of the call graph).

The entries are sorted by time spent in the function and its subroutines.

The internal profiling function `mcount` (see Section 5.1 [The Flat Profile], page 15) is never mentioned in the call graph.

5.2.1 The Primary Line

The *primary line* in a call graph entry is the line that describes the function which the entry is about and gives the overall statistics for this function.

For reference, we repeat the primary line from the entry for function `report` in our main example, together with the heading line that shows the names of the fields:

```
index  % time    self  children called     name
...
[3]    100.0    0.00    0.05       1         report [3]
```

Here is what the fields in the primary line mean:

`index` Entries are numbered with consecutive integers. Each function therefore has an index number, which appears at the beginning of its primary line.

Each cross-reference to a function, as a caller or subroutine of another, gives its index number as well as its name. The index number guides you if you wish to look for the entry for that function.

`% time` This is the percentage of the total time that was spent in this function, including time spent in subroutines called from this function.

The time spent in this function is counted again for the callers of this function. Therefore, adding up these percentages is meaningless.

`self` This is the total amount of time spent in this function. This should be identical to the number printed in the `seconds` field for this function in the flat profile.

`children` This is the total amount of time spent in the subroutine calls made by this function. This should be equal to the sum of all the `self` and `children` entries of the children listed directly below this function.

`called` This is the number of times the function was called.

If the function called itself recursively, there are two numbers, separated by a '`+`'. The first number counts non-recursive calls, and the second counts recursive calls.

In the example above, the function `report` was called once from `main`.

`name` This is the name of the current function. The index number is repeated after it.

If the function is part of a cycle of recursion, the cycle number is printed between the function's name and the index number (see Section 5.2.4 [How Mutually Recursive Functions Are Described], page 20). For example, if function `gnurr` is part of

cycle number one, and has index number twelve, its primary line would be end like this:

```
gnurr <cycle 1> [12]
```

5.2.2 Lines for a Function's Callers

A function's entry has a line for each function it was called by. These lines' fields correspond to the fields of the primary line, but their meanings are different because of the difference in context.

For reference, we repeat two lines from the entry for the function `report`, the primary line and one caller-line preceding it, together with the heading line that shows the names of the fields:

```
index  % time    self  children called     name
...
                0.00    0.05       1/1           main [2]
[3]    100.0    0.00    0.05       1         report [3]
```

Here are the meanings of the fields in the caller-line for `report` called from `main`:

`self` An estimate of the amount of time spent in `report` itself when it was called from `main`.

`children` An estimate of the amount of time spent in subroutines of `report` when `report` was called from `main`.

The sum of the `self` and `children` fields is an estimate of the amount of time spent within calls to `report` from `main`.

`called` Two numbers: the number of times `report` was called from `main`, followed by the total number of non-recursive calls to `report` from all its callers.

`name and index number`

The name of the caller of `report` to which this line applies, followed by the caller's index number.

Not all functions have entries in the call graph; some options to `gprof` request the omission of certain functions. When a caller has no entry of its own, it still has caller-lines in the entries of the functions it calls.

If the caller is part of a recursion cycle, the cycle number is printed between the name and the index number.

If the identity of the callers of a function cannot be determined, a dummy caller-line is printed which has '`<spontaneous>`' as the "caller's name" and all other fields blank. This can happen for signal handlers.

5.2.3 Lines for a Function's Subroutines

A function's entry has a line for each of its subroutines—in other words, a line for each other function that it called. These lines' fields correspond to

the fields of the primary line, but their meanings are different because of the difference in context.

For reference, we repeat two lines from the entry for the function `main`, the primary line and a line for a subroutine, together with the heading line that shows the names of the fields:

```
index  % time    self  children called     name
...
[2]    100.0    0.00    0.05       1         main [2]
                0.00    0.05       1/1           report [3]
```

Here are the meanings of the fields in the subroutine-line for `main` calling `report`:

`self` An estimate of the amount of time spent directly within `report` when `report` was called from `main`.

`children` An estimate of the amount of time spent in subroutines of `report` when `report` was called from `main`.

The sum of the `self` and `children` fields is an estimate of the total time spent in calls to `report` from `main`.

`called` Two numbers, the number of calls to `report` from `main` followed by the total number of non-recursive calls to `report`. This ratio is used to determine how much of `report`'s `self` and `children` time gets credited to `main`. See Section 6.2 [Estimating `children` Times], page 28.

`name` The name of the subroutine of `main` to which this line applies, followed by the subroutine's index number.

If the caller is part of a recursion cycle, the cycle number is printed between the name and the index number.

5.2.4 How Mutually Recursive Functions Are Described

The graph may be complicated by the presence of *cycles of recursion* in the call graph. A cycle exists if a function calls another function that (directly or indirectly) calls (or appears to call) the original function. For example: if `a` calls `b`, and `b` calls `a`, then `a` and `b` form a cycle.

Whenever there are call paths both ways between a pair of functions, they belong to the same cycle. If `a` and `b` call each other and `b` and `c` call each other, all three make one cycle. Note that even if `b` only calls `a` if it was not called from `a`, `gprof` cannot determine this, so `a` and `b` are still considered a cycle.

The cycles are numbered with consecutive integers. When a function belongs to a cycle, each time the function name appears in the call graph it is followed by '`<cycle number>`'.

The reason cycles matter is that they make the time values in the call graph paradoxical. The "time spent in children" of `a` should include the time

spent in its subroutine b and in b's subroutines—but one of b's subroutines is a! How much of a's time should be included in the children of a, when a is indirectly recursive?

The way gprof resolves this paradox is by creating a single entry for the cycle as a whole. The primary line of this entry describes the total time spent directly in the functions of the cycle. The "subroutines" of the cycle are the individual functions of the cycle, and all other functions that were called directly by them. The "callers" of the cycle are the functions, outside the cycle, that called functions in the cycle.

Here is an example portion of a call graph which shows a cycle containing functions a and b. The cycle was entered by a call to a from main; both a and b called c.

```
index  % time    self  children called     name
-----------------------------------------
                1.77        0    1/1        main [2]
[3]     91.71   1.77        0    1+5    <cycle 1 as a whole> [3]
                1.02        0    3          b <cycle 1> [4]
                0.75        0    2          a <cycle 1> [5]
-----------------------------------------
                                 3          a <cycle 1> [5]
[4]     52.85   1.02        0    0      b <cycle 1> [4]
                                 2          a <cycle 1> [5]
                   0        0    3/6        c [6]
-----------------------------------------
                1.77        0    1/1        main [2]
                                 2          b <cycle 1> [4]
[5]     38.86   0.75        0    1      a <cycle 1> [5]
                                 3          b <cycle 1> [4]
                   0        0    3/6        c [6]
-----------------------------------------
```

(The entire call graph for this program contains in addition an entry for main, which calls a, and an entry for c, with callers a and b.)

```
index  % time    self  children called     name
                                            <spontaneous>
[1]    100.00      0     1.93    0      start [1]
                0.16     1.77    1/1        main [2]
-----------------------------------------
                0.16     1.77    1/1        start [1]
[2]    100.00   0.16     1.77    1      main [2]
                1.77        0    1/1        a <cycle 1> [5]
-----------------------------------------
                1.77        0    1/1        main [2]
[3]     91.71   1.77        0    1+5    <cycle 1 as a whole> [3]
                1.02        0    3          b <cycle 1> [4]
                0.75        0    2          a <cycle 1> [5]
                   0        0    6/6        c [6]
-----------------------------------------
                                 3          a <cycle 1> [5]
[4]     52.85   1.02        0    0      b <cycle 1> [4]
```

```
                                    2          a <cycle 1> [5]
                     0        0    3/6        c [6]
-----------------------------------------
                  1.77        0    1/1        main [2]
                                    2          b <cycle 1> [4]
[5]     38.86     0.75        0    1      a <cycle 1> [5]
                                    3          b <cycle 1> [4]
                     0        0    3/6        c [6]
-----------------------------------------
                     0        0    3/6        b <cycle 1> [4]
                     0        0    3/6        a <cycle 1> [5]
[6]      0.00        0        0    6      c [6]
-----------------------------------------
```

The `self` field of the cycle's primary line is the total time spent in all the functions of the cycle. It equals the sum of the `self` fields for the individual functions in the cycle, found in the entry in the subroutine lines for these functions.

The `children` fields of the cycle's primary line and subroutine lines count only subroutines outside the cycle. Even though `a` calls `b`, the time spent in those calls to `b` is not counted in `a`'s `children` time. Thus, we do not encounter the problem of what to do when the time in those calls to `b` includes indirect recursive calls back to `a`.

The `children` field of a caller-line in the cycle's entry estimates the amount of time spent *in the whole cycle*, and its other subroutines, on the times when that caller called a function in the cycle.

The `called` field in the primary line for the cycle has two numbers: first, the number of times functions in the cycle were called by functions outside the cycle; second, the number of times they were called by functions in the cycle (including times when a function in the cycle calls itself). This is a generalization of the usual split into non-recursive and recursive calls.

The `called` field of a subroutine-line for a cycle member in the cycle's entry says how many time that function was called from functions in the cycle. The total of all these is the second number in the primary line's `called` field.

In the individual entry for a function in a cycle, the other functions in the same cycle can appear as subroutines and as callers. These lines show how many times each function in the cycle called or was called from each other function in the cycle. The `self` and `children` fields in these lines are blank because of the difficulty of defining meanings for them when recursion is going on.

5.3 Line-by-line Profiling

`gprof`'s '`-l`' option causes the program to perform *line-by-line* profiling. In this mode, histogram samples are assigned not to functions, but to individual lines of source code. This only works with programs compiled with older

versions of the gcc compiler. Newer versions of gcc use a different program - gcov - to display line-by-line profiling information.

With the older versions of gcc the program usually has to be compiled with a '-g' option, in addition to '-pg', in order to generate debugging symbols for tracking source code lines. Note, in much older versions of gcc the program had to be compiled with the '-a' command-line option as well.

The flat profile is the most useful output table in line-by-line mode. The call graph isn't as useful as normal, since the current version of gprof does not propagate call graph arcs from source code lines to the enclosing function. The call graph does, however, show each line of code that called each function, along with a count.

Here is a section of gprof's output, without line-by-line profiling. Note that ct_init accounted for four histogram hits, and 13327 calls to init_block.

```
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
```

Now let's look at some of gprof's output from the same program run, this time with line-by-line profiling enabled. Note that ct_init's four histogram hits are broken down into four lines of source code—one hit occurred on each of lines 349, 351, 382 and 385. In the call graph, note how ct_init's 13327 calls to init_block are broken down into one call from line 396, 3071 calls from line 384, 3730 calls from line 385, and 6525 calls from 387.

```
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)
```

5.4 The Annotated Source Listing

`gprof`'s '`-A`' option triggers an annotated source listing, which lists the program's source code, each function labeled with the number of times it was called. You may also need to specify the '`-I`' option, if `gprof` can't find the source code files.

With older versions of `gcc` compiling with '`gcc ... -g -pg -a`' augments your program with basic-block counting code, in addition to function counting code. This enables `gprof` to determine how many times each line of code was executed. With newer versions of `gcc` support for displaying basic-block counts is provided by the `gcov` program.

For example, consider the following function, taken from gzip, with line numbers added:

```
 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` has at least five basic-blocks. One is the function itself. The `if` statement on line 9 generates two more basic-blocks, one for each branch of the `if`. A fourth basic-block results from the `if` on line 13, and the contents of the `do` loop form the fifth basic-block. The compiler may also generate additional basic-blocks to handle various special cases.

A program augmented for basic-block counting can be analyzed with '`gprof -l -A`'. The '`-x`' option is also helpful, to ensure that each line of code is labeled at least once. Here is `updcrc`'s annotated source listing for a sample `gzip` run:

```
                    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 this example, the function was called twice, passing once through each branch of the `if` statement. The body of the `do` loop was executed a total of 26312 times. Note how the `while` statement is annotated. It began execution 26312 times, once for each iteration through the loop. One of those times (the last time) it exited, while it branched back to the beginning of the loop 26311 times.

6 Inaccuracy of gprof Output

6.1 Statistical Sampling Error

The run-time figures that `gprof` gives you are based on a sampling process, so they are subject to statistical inaccuracy. If a function runs only a small amount of time, so that on the average the sampling process ought to catch that function in the act only once, there is a pretty good chance it will actually find that function zero times, or twice.

By contrast, the number-of-calls and basic-block figures are derived by counting, not sampling. They are completely accurate and will not vary from run to run if your program is deterministic and single threaded. In multi-threaded applications, or single threaded applications that link with multi-threaded libraries, the counts are only deterministic if the counting function is thread-safe. (Note: beware that the mcount counting function in glibc is *not* thread-safe). See Section 9.1 [Implementation of Profiling], page 33.

The *sampling period* that is printed at the beginning of the flat profile says how often samples are taken. The rule of thumb is that a run-time figure is accurate if it is considerably bigger than the sampling period.

The actual amount of error can be predicted. For n samples, the *expected* error is the square-root of n. For example, if the sampling period is 0.01 seconds and `foo`'s run-time is 1 second, n is 100 samples (1 second/0.01 seconds), sqrt(n) is 10 samples, so the expected error in `foo`'s run-time is 0.1 seconds (10*0.01 seconds), or ten percent of the observed value. Again, if the sampling period is 0.01 seconds and `bar`'s run-time is 100 seconds, n is 10000 samples, sqrt(n) is 100 samples, so the expected error in `bar`'s run-time is 1 second, or one percent of the observed value. It is likely to vary this much *on the average* from one profiling run to the next. (*Sometimes* it will vary more.)

This does not mean that a small run-time figure is devoid of information. If the program's *total* run-time is large, a small run-time for one function does tell you that that function used an insignificant fraction of the whole program's time. Usually this means it is not worth optimizing.

One way to get more accuracy is to give your program more (but similar) input data so it will take longer. Another way is to combine the data from several runs, using the '`-s`' option of `gprof`. Here is how:

1. Run your program once.
2. Issue the command '`mv gmon.out gmon.sum`'.
3. Run your program again, the same as before.
4. Merge the new data in '`gmon.out`' into '`gmon.sum`' with this command:

   ```
   gprof -s executable-file gmon.out gmon.sum
   ```

5. Repeat the last two steps as often as you wish.

6. Analyze the cumulative data using this command:

```
gprof executable-file gmon.sum > output-file
```

6.2 Estimating `children` Times

Some of the figures in the call graph are estimates—for example, the `children` time values and all the time figures in caller and subroutine lines.

There is no direct information about these measurements in the profile data itself. Instead, `gprof` estimates them by making an assumption about your program that might or might not be true.

The assumption made is that the average time spent in each call to any function `foo` is not correlated with who called `foo`. If `foo` used 5 seconds in all, and 2/5 of the calls to `foo` came from `a`, then `foo` contributes 2 seconds to `a`'s `children` time, by assumption.

This assumption is usually true enough, but for some programs it is far from true. Suppose that `foo` returns very quickly when its argument is zero; suppose that `a` always passes zero as an argument, while other callers of `foo` pass other arguments. In this program, all the time spent in `foo` is in the calls from callers other than `a`. But `gprof` has no way of knowing this; it will blindly and incorrectly charge 2 seconds of time in `foo` to the children of `a`.

We hope some day to put more complete data into '`gmon.out`', so that this assumption is no longer needed, if we can figure out how. For the novice, the estimated figures are usually more useful than misleading.

7 Answers to Common Questions

How can I get more exact information about hot spots in my program?

Looking at the per-line call counts only tells part of the story. Because `gprof` can only report call times and counts by function, the best way to get finer-grained information on where the program is spending its time is to re-factor large functions into sequences of calls to smaller ones. Beware however that this can introduce artificial hot spots since compiling with '`-pg`' adds a significant overhead to function calls. An alternative solution is to use a non-intrusive profiler, e.g. oprofile.

How do I find which lines in my program were executed the most times?

Use the `gcov` program.

How do I find which lines in my program called a particular function?

Use '`gprof -l`' and lookup the function in the call graph. The callers will be broken down by function and line number.

How do I analyze a program that runs for less than a second?

Try using a shell script like this one:

```
for i in `seq 1 100`; do
  fastprog
  mv gmon.out gmon.out.$i
done

gprof -s fastprog gmon.out.*

gprof fastprog gmon.sum
```

If your program is completely deterministic, all the call counts will be simple multiples of 100 (i.e., a function called once in each run will appear with a call count of 100).

8 Incompatibilities with Unix gprof

GNU `gprof` and Berkeley Unix `gprof` use the same data file '`gmon.out`', and provide essentially the same information. But there are a few differences.

- GNU `gprof` uses a new, generalized file format with support for basic-block execution counts and non-realtime histograms. A magic cookie and version number allows `gprof` to easily identify new style files. Old BSD-style files can still be read. See Section 9.2 [Profiling Data File Format], page 34.
- For a recursive function, Unix `gprof` lists the function as a parent and as a child, with a `calls` field that lists the number of recursive calls. GNU `gprof` omits these lines and puts the number of recursive calls in the primary line.
- When a function is suppressed from the call graph with '`-e`', GNU `gprof` still lists it as a subroutine of functions that call it.
- GNU `gprof` accepts the '`-k`' with its argument in the form '`from/to`', instead of '`from to`'.
- In the annotated source listing, if there are multiple basic blocks on the same line, GNU `gprof` prints all of their counts, separated by commas.
- The blurbs, field widths, and output formats are different. GNU `gprof` prints blurbs after the tables, so that you can see the tables without skipping the blurbs.

9 Details of Profiling

9.1 Implementation of Profiling

Profiling works by changing how every function in your program is compiled so that when it is called, it will stash away some information about where it was called from. From this, the profiler can figure out what function called it, and can count how many times it was called. This change is made by the compiler when your program is compiled with the '`-pg`' option, which causes every function to call `mcount` (or `_mcount`, or `__mcount`, depending on the OS and compiler) as one of its first operations.

The `mcount` routine, included in the profiling library, is responsible for recording in an in-memory call graph table both its parent routine (the child) and its parent's parent. This is typically done by examining the stack frame to find both the address of the child, and the return address in the original parent. Since this is a very machine-dependent operation, `mcount` itself is typically a short assembly-language stub routine that extracts the required information, and then calls `__mcount_internal` (a normal C function) with two arguments—`frompc` and `selfpc`. `__mcount_internal` is responsible for maintaining the in-memory call graph, which records `frompc`, `selfpc`, and the number of times each of these call arcs was traversed.

GCC Version 2 provides a magical function (`__builtin_return_address`), which allows a generic `mcount` function to extract the required information from the stack frame. However, on some architectures, most notably the SPARC, using this builtin can be very computationally expensive, and an assembly language version of `mcount` is used for performance reasons.

Number-of-calls information for library routines is collected by using a special version of the C library. The programs in it are the same as in the usual C library, but they were compiled with '`-pg`'. If you link your program with '`gcc ... -pg`', it automatically uses the profiling version of the library.

Profiling also involves watching your program as it runs, and keeping a histogram of where the program counter happens to be every now and then. Typically the program counter is looked at around 100 times per second of run time, but the exact frequency may vary from system to system.

This is done is one of two ways. Most UNIX-like operating systems provide a `profil()` system call, which registers a memory array with the kernel, along with a scale factor that determines how the program's address space maps into the array. Typical scaling values cause every 2 to 8 bytes of address space to map into a single array slot. On every tick of the system clock (assuming the profiled program is running), the value of the program counter is examined and the corresponding slot in the memory array is incremented. Since this is done in the kernel, which had to interrupt the process anyway to handle the clock interrupt, very little additional system overhead is required.

However, some operating systems, most notably Linux 2.0 (and earlier), do not provide a `profil()` system call. On such a system, arrangements are made for the kernel to periodically deliver a signal to the process (typically via `setitimer()`), which then performs the same operation of examining the program counter and incrementing a slot in the memory array. Since this method requires a signal to be delivered to user space every time a sample is taken, it uses considerably more overhead than kernel-based profiling. Also, due to the added delay required to deliver the signal, this method is less accurate as well.

A special startup routine allocates memory for the histogram and either calls `profil()` or sets up a clock signal handler. This routine (`monstartup`) can be invoked in several ways. On Linux systems, a special profiling startup file `gcrt0.o`, which invokes `monstartup` before `main`, is used instead of the default `crt0.o`. Use of this special startup file is one of the effects of using '`gcc ... -pg`' to link. On SPARC systems, no special startup files are used. Rather, the `mcount` routine, when it is invoked for the first time (typically when `main` is called), calls `monstartup`.

If the compiler's '`-a`' option was used, basic-block counting is also enabled. Each object file is then compiled with a static array of counts, initially zero. In the executable code, every time a new basic-block begins (i.e., when an `if` statement appears), an extra instruction is inserted to increment the corresponding count in the array. At compile time, a paired array was constructed that recorded the starting address of each basic-block. Taken together, the two arrays record the starting address of every basic-block, along with the number of times it was executed.

The profiling library also includes a function (`mcleanup`) which is typically registered using `atexit()` to be called as the program exits, and is responsible for writing the file '`gmon.out`'. Profiling is turned off, various headers are output, and the histogram is written, followed by the call-graph arcs and the basic-block counts.

The output from `gprof` gives no indication of parts of your program that are limited by I/O or swapping bandwidth. This is because samples of the program counter are taken at fixed intervals of the program's run time. Therefore, the time measurements in `gprof` output say nothing about time that your program was not running. For example, a part of the program that creates so much data that it cannot all fit in physical memory at once may run very slowly due to thrashing, but `gprof` will say it uses little time. On the other hand, sampling by run time has the advantage that the amount of load due to other users won't directly affect the output you get.

9.2 Profiling Data File Format

The old BSD-derived file format used for profile data does not contain a magic cookie that allows one to check whether a data file really is a `gprof` file. Furthermore, it does not provide a version number, thus rendering changes

to the file format almost impossible. GNU `gprof` uses a new file format that provides these features. For backward compatibility, GNU `gprof` continues to support the old BSD-derived format, but not all features are supported with it. For example, basic-block execution counts cannot be accommodated by the old file format.

The new file format is defined in header file '`gmon_out.h`'. It consists of a header containing the magic cookie and a version number, as well as some spare bytes available for future extensions. All data in a profile data file is in the native format of the target for which the profile was collected. GNU `gprof` adapts automatically to the byte-order in use.

In the new file format, the header is followed by a sequence of records. Currently, there are three different record types: histogram records, call-graph arc records, and basic-block execution count records. Each file can contain any number of each record type. When reading a file, GNU `gprof` will ensure records of the same type are compatible with each other and compute the union of all records. For example, for basic-block execution counts, the union is simply the sum of all execution counts for each basic-block.

9.2.1 Histogram Records

Histogram records consist of a header that is followed by an array of bins. The header contains the text-segment range that the histogram spans, the size of the histogram in bytes (unlike in the old BSD format, this does not include the size of the header), the rate of the profiling clock, and the physical dimension that the bin counts represent after being scaled by the profiling clock rate. The physical dimension is specified in two parts: a long name of up to 15 characters and a single character abbreviation. For example, a histogram representing real-time would specify the long name as "seconds" and the abbreviation as "s". This feature is useful for architectures that support performance monitor hardware (which, fortunately, is becoming increasingly common). For example, under DEC OSF/1, the "uprofile" command can be used to produce a histogram of, say, instruction cache misses. In this case, the dimension in the histogram header could be set to "i-cache misses" and the abbreviation could be set to "1" (because it is simply a count, not a physical dimension). Also, the profiling rate would have to be set to 1 in this case.

Histogram bins are 16-bit numbers and each bin represent an equal amount of text-space. For example, if the text-segment is one thousand bytes long and if there are ten bins in the histogram, each bin represents one hundred bytes.

9.2.2 Call-Graph Records

Call-graph records have a format that is identical to the one used in the BSD-derived file format. It consists of an arc in the call graph and a count indicating the number of times the arc was traversed during program execution. Arcs are specified by a pair of addresses: the first must be within

caller's function and the second must be within the callee's function. When performing profiling at the function level, these addresses can point anywhere within the respective function. However, when profiling at the line-level, it is better if the addresses are as close to the call-site/entry-point as possible. This will ensure that the line-level call-graph is able to identify exactly which line of source code performed calls to a function.

9.2.3 Basic-Block Execution Count Records

Basic-block execution count records consist of a header followed by a sequence of address/count pairs. The header simply specifies the length of the sequence. In an address/count pair, the address identifies a basic-block and the count specifies the number of times that basic-block was executed. Any address within the basic-address can be used.

9.3 `gprof`'s Internal Operation

Like most programs, `gprof` begins by processing its options. During this stage, it may building its symspec list (`sym_ids.c:sym_id_add`), if options are specified which use symspecs. `gprof` maintains a single linked list of symspecs, which will eventually get turned into 12 symbol tables, organized into six include/exclude pairs—one pair each for the flat profile (INCL_FLAT/EXCL_FLAT), the call graph arcs (INCL_ARCS/EXCL_ARCS), printing in the call graph (INCL_GRAPH/EXCL_GRAPH), timing propagation in the call graph (INCL_TIME/EXCL_TIME), the annotated source listing (INCL_ANNO/EXCL_ANNO), and the execution count listing (INCL_EXEC/EXCL_EXEC).

After option processing, `gprof` finishes building the symspec list by adding all the symspecs in `default_excluded_list` to the exclude lists EXCL_TIME and EXCL_GRAPH, and if line-by-line profiling is specified, EXCL_FLAT as well. These default excludes are not added to EXCL_ANNO, EXCL_ARCS, and EXCL_EXEC.

Next, the BFD library is called to open the object file, verify that it is an object file, and read its symbol table (`core.c:core_init`), using `bfd_canonicalize_symtab` after mallocing an appropriately sized array of symbols. At this point, function mappings are read (if the '`--file-ordering`' option has been specified), and the core text space is read into memory (if the '`-c`' option was given).

`gprof`'s own symbol table, an array of Sym structures, is now built. This is done in one of two ways, by one of two routines, depending on whether line-by-line profiling ('`-l`' option) has been enabled. For normal profiling, the BFD canonical symbol table is scanned. For line-by-line profiling, every text space address is examined, and a new symbol table entry gets created every time the line number changes. In either case, two passes are made through the symbol table—one to count the size of the symbol table required, and

the other to actually read the symbols. In between the two passes, a single array of type `Sym` is created of the appropriate length. Finally, `symtab.c:symtab_finalize` is called to sort the symbol table and remove duplicate entries (entries with the same memory address).

The symbol table must be a contiguous array for two reasons. First, the `qsort` library function (which sorts an array) will be used to sort the symbol table. Also, the symbol lookup routine (`symtab.c:sym_lookup`), which finds symbols based on memory address, uses a binary search algorithm which requires the symbol table to be a sorted array. Function symbols are indicated with an `is_func` flag. Line number symbols have no special flags set. Additionally, a symbol can have an `is_static` flag to indicate that it is a local symbol.

With the symbol table read, the symspecs can now be translated into Syms (`sym_ids.c:sym_id_parse`). Remember that a single symspec can match multiple symbols. An array of symbol tables (`syms`) is created, each entry of which is a symbol table of Syms to be included or excluded from a particular listing. The master symbol table and the symspecs are examined by nested loops, and every symbol that matches a symspec is inserted into the appropriate syms table. This is done twice, once to count the size of each required symbol table, and again to build the tables, which have been malloced between passes. From now on, to determine whether a symbol is on an include or exclude symspec list, `gprof` simply uses its standard symbol lookup routine on the appropriate table in the `syms` array.

Now the profile data file(s) themselves are read (`gmon_io.c:gmon_out_read`), first by checking for a new-style '`gmon.out`' header, then assuming this is an old-style BSD '`gmon.out`' if the magic number test failed.

New-style histogram records are read by `hist.c:hist_read_rec`. For the first histogram record, allocate a memory array to hold all the bins, and read them in. When multiple profile data files (or files with multiple histogram records) are read, the memory ranges of each pair of histogram records must be either equal, or non-overlapping. For each pair of histogram records, the resolution (memory region size divided by the number of bins) must be the same. The time unit must be the same for all histogram records. If the above containts are met, all histograms for the same memory range are merged.

As each call graph record is read (`call_graph.c:cg_read_rec`), the parent and child addresses are matched to symbol table entries, and a call graph arc is created by `cg_arcs.c:arc_add`, unless the arc fails a symspec check against INCL_ARCS/EXCL_ARCS. As each arc is added, a linked list is maintained of the parent's child arcs, and of the child's parent arcs. Both the child's call count and the arc's call count are incremented by the record's call count.

Basic-block records are read (`basic_blocks.c:bb_read_rec`), but only if line-by-line profiling has been selected. Each basic-block address is matched to a corresponding line symbol in the symbol table, and an entry made in the

symbol's bb_addr and bb_calls arrays. Again, if multiple basic-block records are present for the same address, the call counts are cumulative.

A gmon.sum file is dumped, if requested (`gmon_io.c:gmon_out_write`).

If histograms were present in the data files, assign them to symbols (`hist.c:hist_assign_samples`) by iterating over all the sample bins and assigning them to symbols. Since the symbol table is sorted in order of ascending memory addresses, we can simple follow along in the symbol table as we make our pass over the sample bins. This step includes a symspec check against INCL_FLAT/EXCL_FLAT. Depending on the histogram scale factor, a sample bin may span multiple symbols, in which case a fraction of the sample count is allocated to each symbol, proportional to the degree of overlap. This effect is rare for normal profiling, but overlaps are more common during line-by-line profiling, and can cause each of two adjacent lines to be credited with half a hit, for example.

If call graph data is present, `cg_arcs.c:cg_assemble` is called. First, if '`-c`' was specified, a machine-dependent routine (`find_call`) scans through each symbol's machine code, looking for subroutine call instructions, and adding them to the call graph with a zero call count. A topological sort is performed by depth-first numbering all the symbols (`cg_dfn.c:cg_dfn`), so that children are always numbered less than their parents, then making a array of pointers into the symbol table and sorting it into numerical order, which is reverse topological order (children appear before parents). Cycles are also detected at this point, all members of which are assigned the same topological number. Two passes are now made through this sorted array of symbol pointers. The first pass, from end to beginning (parents to children), computes the fraction of child time to propagate to each parent and a print flag. The print flag reflects symspec handling of INCL_GRAPH/EXCL_GRAPH, with a parent's include or exclude (print or no print) property being propagated to its children, unless they themselves explicitly appear in INCL_GRAPH or EXCL_GRAPH. A second pass, from beginning to end (children to parents) actually propagates the timings along the call graph, subject to a check against INCL_TIME/EXCL_TIME. With the print flag, fractions, and timings now stored in the symbol structures, the topological sort array is now discarded, and a new array of pointers is assembled, this time sorted by propagated time.

Finally, print the various outputs the user requested, which is now fairly straightforward. The call graph (`cg_print.c:cg_print`) and flat profile (`hist.c:hist_print`) are regurgitations of values already computed. The annotated source listing (`basic_blocks.c:print_annotated_source`) uses basic-block information, if present, to label each line of code with call counts, otherwise only the function call counts are presented.

The function ordering code is marginally well documented in the source code itself (`cg_print.c`). Basically, the functions with the most use and the most parents are placed first, followed by other functions with the most use, followed by lower use functions, followed by unused functions at the end.

9.4 Debugging `gprof`

If `gprof` was compiled with debugging enabled, the '`-d`' option triggers debugging output (to stdout) which can be helpful in understanding its operation. The debugging number specified is interpreted as a sum of the following options:

2 - Topological sort
: Monitor depth-first numbering of symbols during call graph analysis

4 - Cycles
: Shows symbols as they are identified as cycle heads

16 - Tallying
: As the call graph arcs are read, show each arc and how the total calls to each function are tallied

32 - Call graph arc sorting
: Details sorting individual parents/children within each call graph entry

64 - Reading histogram and call graph records
: Shows address ranges of histograms as they are read, and each call graph arc

128 - Symbol table
: Reading, classifying, and sorting the symbol table from the object file. For line-by-line profiling ('`-l`' option), also shows line numbers being assigned to memory addresses.

256 - Static call graph
: Trace operation of '`-c`' option

512 - Symbol table and arc table lookups
: Detail operation of lookup routines

1024 - Call graph propagation
: Shows how function times are propagated along the call graph

2048 - Basic-blocks
: Shows basic-block records as they are read from profile data (only meaningful with '`-l`' option)

4096 - Symspecs
: Shows symspec-to-symbol pattern matching operation

8192 - Annotate source
: Tracks operation of '`-A`' option

Appendix A GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.
http://fsf.org/

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or

to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not

add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such

new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (C) *year* *your name*.
> Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled ‘‘GNU Free Documentation License’’.

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

> with the Invariant Sections being *list their titles*, with the Front-Cover Texts being *list*, and with the Back-Cover Texts being *list*.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.