GNU gprofng

The next generation profiling tool for Linux
version 2.41 (last updated 3 July 2023)

Ruud van der Pas

This document is the manual for gprofng, last updated 3 July 2023.

Table of Contents

1 Introduction

The gprofng tool is the next generation profiler for Linux. It consists of various commands to generate and display profile information.

This manual starts with a tutorial how to create and interpret a profile. This part is highly practical and has the goal to get users up to speed as quickly as possible. As soon as possible, we would like to show you how to get your first profile on your screen.

This is followed by more examples, covering many of the features. At the end of this tutorial, you should feel confident enough to tackle the more complex tasks.

In a future update a more formal reference manual will be included as well. Since even in this tutorial we use certain terminology, we have included a chapter with descriptions at the end. In case you encounter unfamiliar wordings or terminology, please check this chapter.

One word of caution. In several cases we had to somewhat tweak the screen output in order to make it fit. This is why the output may look somewhat different when you try things yourself.

For now, we wish you a smooth profiling experience with gprofng and good luck tackling performance bottlenecks.

2 A Brief Overview of gprofng

Before we cover this tool in quite some detail, we start with a brief overview of what it is, and the main features. Since we know that many of you would like to get started rightaway, already in this first chapter we explain the basics of profiling with `gprofng`.

2.1 Main Features

These are the main features of the gprofng tool:

- Profiling is supported for an application written in C, C++, Java, or Scala.
- Shared libraries are supported. The information is presented at the instruction level.
- The following multithreading programming models are supported: Pthreads, OpenMP, and Java threads.
- This tool works with unmodified production level executables. There is no need to recompile the code, but if the '`-g`' option has been used when building the application, source line level information is available.
- The focus is on support for code generated with the `gcc` compiler, but there is some limited support for the `icc` compiler as well. Future improvements and enhancements will focus on `gcc` though.
- Processors from Intel, AMD, and Arm are supported, but the level of support depends on the architectural details. In particular, hardware event counters may not be supported. If this is the case, all views not related to these counters still ought to work though.
- Several views into the data are supported. For example, a function overview where the time is spent, but also a source line, disassembly, call tree and a caller-callees overview are available.
- Through filters, the user can zoom in on an area of interest.
- Two or more profiles can be aggregated, or used in a comparison. This comparison can be obtained at the function, source line, and disassembly level.
- Through a simple scripting language, and customization of the metrics shown, the generation and creation of a profile can be fully automated and provide tailored output.

2.2 Sampling versus Tracing

A key difference with some other profiling tools is that the main data collection command `gprofng collect app` mostly uses Program Counter (PC) sampling under the hood.

With *sampling*, the executable is interrupted at regular intervals. Each time it is halted, key information is gathered and stored. This includes

the Program Counter that keeps track of where the execution is. Hence the name.

Together with operational data, this information is stored in the experiment directory and can be viewed in the second phase.

For example, the PC information is used to derive where the program was when it was halted. Since the sampling interval is known, it is relatively easy to derive how much time was spent in the various parts of the program.

The opposite technique is generally referred to as *tracing*. With tracing, the target is instrumented with specific calls that collect the requested information.

These are some of the pros and cons of PC sampling verus tracing:

- Since there is no need to recompile, existing executables can be used and the profile measures the behaviour of exactly the same executable that is used in production runs.

  With sampling, one inherently profiles a different executable, because the calls to the instrumentation library may affect the compiler optimizations and run time behaviour.
- With sampling, there are very few restrictions on what can be profiled and even without access to the source code, a basic profile can be made.
- A downside of sampling is that, depending on the sampling frequency, small functions may be missed or not captured accurately. Although this is rare, this may happen and is the reason why the user has control over the sampling rate.
- While tracing produces precise information, sampling is statistical in nature. As a result, small variations may occur across seemingly identical runs. We have not observed more than a few percent deviation though. Especially if the target job executed for a sufficiently long time.
- With sampling, it is not possible to get an accurate count how often functions are called.

2.3 Steps Needed to Create a Profile

Creating a profile takes two steps. First the profile data needs to be generated. This is followed by a viewing step to create a report from the information that has been gathered.

Every gprofng command starts with `gprofng`, the name of the driver. This is followed by a keyword to define the high level functionality. Depending on this keyword, a third qualifier may be needed to further narrow down the request. This combination is then followed by options that are specific to the functionality desired.

The command to gather, or "collect", the performance data is called `gprofng collect app`. Aside from numerous options, this command takes the name of the target executable as an input parameter.

Upon completion of the run, the performance data can be found in the newly created experiment directory.

Unless explicitly specified otherwise, a default name for this directory is chosen. The name is `test.<n>.er` where <*n*> is the first integer number not in use yet for such a name.

For example, the first time `gprofng collect app` is invoked, an experiment directory with the name `test.1.er` is created. Upon a subsequent invocation of `gprofng collect app` in the same directory, an experiment directory with the name `test.2.er` will be created, and so forth.

Note that `gprofng collect app` supports an option to explicitly name the experiment directory. Aside from the restriction that the name of this directory has to end with '`.er`', any valid directory name can be used for this.

Now that we have the performance data, the next step is to display it.

The most commonly used command to view the performance information is `gprofng display text`. This is a very extensive and customizable tool that produces the information in ASCII format.

Another option is to use `gprofng display html`. This tool generates a directory with files in html format. These can be viewed in a browser, allowing for easy navigation through the profile data.

3 A Mini Tutorial

In this chapter we present and discuss the main functionality of `gprofng`. This will be a practical approach, using an example code to generate profile data and show how to get various performance reports.

3.1 Getting Started

The information presented here provides a good and common basis for many profiling tasks, but there are more features that you may want to leverage.

These are covered in subsequent sections in this chapter.

3.1.1 The Example Program

Throughout this guide we use the same example C code that implements the multiplication of a vector of length n by an m by n matrix. The result is stored in a vector of length m. The algorithm has been parallelized using Posix Threads, or Pthreads for short.

The code was built using the `gcc` compiler and the name of the executable is `mxv-pthreads`.

The matrix sizes can be set through the `-m` and `-n` options. The number of threads is set with the `-t` option. These are additional threads that are used in the multiplication. To increase the duration of the run, the computations are executed repeatedly.

This is an example that multiplies a 8000 by 4000 matrix with a vector of length 4000. Although this is a multithreaded application, initially we will be using 1 threads. Later on we will show examples using multiple threads.

```
$ ./mxv-pthreads -m 8000 -n 4000 -t 1
mxv: error check passed - rows = 8000 columns = 4000 threads = 1
$
```

The program performs an internal check to verify that the computed results are correct. The result of this check is printed, as well as the matrix sizes and the number of threads used.

3.1.2 A First Profile

The first step is to collect the performance data. It is important to remember that much more information is gathered than may be shown by default. Often a single data collection run is sufficient to get a lot of insight.

The `gprofng collect app` command is used for the data collection. Nothing needs to be changed in the way the application is executed. The only difference is that it is now run under control of the tool, as shown below:

```
$ gprofng collect app ./mxv-pthreads -m 8000 -n 4000 -t 1
```

This produces the following output:

```
Creating experiment directory test.1.er (Process ID: 2749878) ...
mxv: error check passed - rows = 8000 columns = 4000 threads = 1
```

We see a message that an experiment directory with the name `test.1.er` has been created. The process id is also echoed. The application completes as usual and we have our first experiment directory that can be analyzed.

The tool we use for this is called `gprofng display text`. It takes the name of the experiment directory as an argument.

If invoked this way, the tool starts in the interactive *interpreter* mode. While in this environment, commands can be given and the tool responds. This is illustrated below:

```
$ gprofng display text test.1.er
Warning: History and command editing is not supported on this system.
(gp-display-text) quit
$
```

While useful in certain cases, we prefer to use this tool in command line mode by specifying the commands to be issued when invoking the tool. The way to do this is to prepend the command(s) with a hyphen ('-') if used on the command line.

Since this makes the command appear to be an option, they are also sometimes referred to as such, but technically they are commands. This is the terminology we will use in this user guide, but for convenience the commands are also listed as options in the index.

For example, below we use the `functions` command to request a list of the functions that have been executed, plus their respective CPU times:

```
$ gprofng display text -functions test.1.er
```

```
$ gprofng display text -functions test.1.er

Functions sorted by metric: Exclusive Total CPU Time

Excl. Total   Incl. Total    Name
CPU           CPU
 sec.      %   sec.      %
9.367 100.00  9.367 100.00   <Total>
8.926  95.30  8.926  95.30   mxv_core
0.210   2.24  0.420   4.49   init_data
0.080   0.85  0.210   2.24   drand48
0.070   0.75  0.130   1.39   erand48_r
0.060   0.64  0.060   0.64   __drand48_iterate
0.010   0.11  0.020   0.21   _int_malloc
0.010   0.11  0.010   0.11   sysmalloc
0.      0.    8.926  95.30   <static>@0x47960 (<libgp-collector.so>)
0.      0.    0.440   4.70   __libc_start_main
0.      0.    0.020   0.21   allocate_data
```

```
0.      0.    8.926  95.30   driver_mxv
0.      0.    0.440   4.70   main
0.      0.    0.020   0.21   malloc
0.      0.    8.926  95.30   start_thread
```

As easy and simple as these steps are, we do have a first profile of our program!

There are five columns. The first four contain the "Total CPU Time", which is the sum of the user and system time. See Section 7.2 [Inclusive and Exclusive Metrics], page 59, for an explanation of "exclusive" and "inclusive" times.

The first line echoes the metric that is used to sort the output. By default, this is the exclusive CPU time, but through the `sort` command, the sort metric can be changed by the user.

Next, there are four columns with the exclusive and inclusive CPU times and the respective percentages. This is followed by the name of the function.

The function with the name `<Total>` is not a user function. It is a pseudo function introduced by `gprofng`. It is used to display the accumulated measured metric values. In this example, we see that the total CPU time of this job was 9.367 seconds and it is scaled to 100%. All other percentages in the same column are relative to this number.

With 8.926 seconds, function `mxv_core` takes 95.30% of the total time and is by far the most time consuming function. The exclusive and inclusive metrics are identical, which means that is a leaf function not calling any other functions.

The next function in the list is `init_data`. Although with 4.49%, the CPU time spent in this part is modest, this is an interesting entry because the inclusive CPU time of 0.420 seconds is twice the exclusive CPU time of 0.210 seconds. Clearly this function is calling another function, or even more than one function and collectively this takes 0.210 seconds. Below we show the call tree feature that provides more details on the call structure of the application.

The function `<static>@0x47960 (<libgp-collector.so>)` does odd and certainly not familiar. It is one of the internal functions used by `gprofng collect app` and can be ignored. Also, while the inclusive time is high, the exclusive time is zero. This means it doesn't contribute to the performance.

The question is how we know where this function originates from? There are several commands to dig deeper an get more details on a function. See Section 3.1.15 [Information on Load Objects], page 22.

3.1.3 The Source Code View

In general, the tuning efforts are best focused on the most time consuming part(s) of an application. In this case that is easy, since over 95% of the total

CPU time is spent in function `mxv_core`. It is now time to dig deeper and look at the metrics distribution at the source code level. Since we measured CPU times, these are the metrics shown.

The `source` command is used to accomplish this. It takes the name of the function, not the source filename, as an argument. This is demonstrated below, where the `gprofng display text` command is used to show the annotated source listing of function `mxv_core`.

Be aware that when using the `gcc` compiler, the source code has to be compiled with the `-g` option in order for the source code feature to work. Otherwise the location(s) can not be determined. For other compilers we recommend to check the documentation for such an option.

Below the command to display the source code of a function is shown. Since at this point we are primarily interested in the timings only, we use the `metrics` command to request the exclusive and inclusive total CPU timings only. See Section 3.1.5 [Display and Define the Metrics], page 14, for more information how to define the metrics to be displayed.

```
$ gprofng display text -metrics ei.totalcpu -source mxv_core test.1.er
```

The output is shown below. It has been somewhat modified to fit the formatting constraints and reduce the number of lines.

```
Current metrics: e.totalcpu:i.totalcpu:name
Current Sort Metric: Exclusive Total CPU Time ( e.totalcpu )
Source file: <apath>/mxv.c
Object file: mxv-pthreads (found as test.1.er/archives/...)
Load Object: mxv-pthreads (found as test.1.er/archives/...)

   Excl.     Incl.
   Total     Total
   CPU sec.  CPU sec.

   <lines deleted>
                              <Function: mxv_core>
                           43. void __attribute__ ((noinline))
                                    mxv_core (int64_t row_index_start,
                           44.               int64_t row_index_end,
                           45.               int64_t m,
                           46.               int64_t n,
                           47.               double **restrict A,
                           48.               double *restrict b,
                           49.               double *restrict c)
                           50. {
   0.        0.            50. {
   0.        0.            51.   for (int64_t i=row_index_start;
                                        i<=row_index_end; i++)
                           52.     {
   0.        0.            53.       double row_sum = 0.0;
## 4.613    4.613         54.       for (int64_t j=0; j<n; j++)
```

```
## 4.313     4.313          55.         row_sum += A[i][j] * b[j];
   0.        0.             56.       c[i] = row_sum;
                            57.     }
   0.        0.             58. }
```

The first line echoes the metrics that have been selected. The second line is not very meaningful when looking at the source code listing, but it shows the metric that is used to sort the data.

The next three lines provide information on the location of the source file, the object file and the load object (See Section 7.6 [Load Objects and Functions], page 62).

Function `mxv_core` is part of a source file that has other functions as well. These functions will be shown with the values for the metrics, but for lay-out purposes they have been removed in the output shown above.

The header is followed by the annotated source code listing. The selected metrics are shown first, followed by a source line number, and the source code. The most time consuming line(s) are marked with the `##` symbol. In this way they are easier to identify and find with a search.

What we see is that all of the time is spent in lines 54-55.

A related command sometimes comes handy as well. It is called `lines` and displays a list of the source lines and their metrics, ordered according to the current sort metric (See Section 3.1.9 [Sorting the Performance Data], page 16).

Below the command and the output. For lay-out reasons, only the top 10 is shown here and the last part of the text on some lines has been replaced by dots. The full text is '`instructions without line numbers`' and means that the line number information for that function was not found.

```
$ gprofng display text -lines test.1.er
```

```
Lines sorted by metric: Exclusive Total CPU Time

Excl. Total   Incl. Total    Name
CPU           CPU
 sec.      %   sec.      %
9.367 100.00  9.367 100.00   <Total>
4.613  49.25  4.613  49.25   mxv_core, line 54 in "mxv.c"
4.313  46.05  4.313  46.05   mxv_core, line 55 in "mxv.c"
0.160   1.71  0.370   3.95   init_data, line 118 in "manage_data.c"
0.080   0.85  0.210   2.24   <Function: drand48, instructions ...>
0.070   0.75  0.130   1.39   <Function: erand48_r, instructions ...>
0.060   0.64  0.060   0.64   <Function: __drand48_iterate, ...>
0.040   0.43  0.040   0.43   init_data, line 124 in "manage_data.c"
0.010   0.11  0.020   0.21   <Function: _int_malloc, instructions ...>
0.010   0.11  0.010   0.11   <Function: sysmalloc, instructions ...>
```

What this overview immediately highlights is that the third most time consuming source line takes 0.370 seconds only. This means that the inclu-

sive time is only 3.95% and clearly this branch of the code hardly impacts the performance.

3.1.4 The Disassembly View

The source view is very useful to obtain more insight where the time is spent, but sometimes this is not sufficient. The disassembly view provides more details since it shows the metrics at the instruction level.

This view is displayed with the `disasm` command and as with the source view, it displays an annotated listing. In this case it shows the instructions with the metrics, interleaved with the source lines. The instructions have a reference in square brackets (`[` and `]`) to the source line they correspond to.

We again focus on the tmings only and set the metrics accordingly:

```
$ gprofng display text -metrics ei.totalcpu -disasm mxv_core test.1.er
```

```
Current metrics: e.totalcpu:i.totalcpu:name
Current Sort Metric: Exclusive Total CPU Time ( e.totalcpu )
Source file: <apath>/src/mxv.c
Object file: mxv-pthreads (found as test.1.er/archives/...)
Load Object: mxv-pthreads (found as test.1.er/archives/...)

   Excl.     Incl.
   Total     Total
   CPU sec.  CPU sec.

   <lines deleted>
                     43. void __attribute__ ((noinline))
                              mxv_core (int64_t row_index_start,
                     44.                int64_t row_index_end,
                     45.                int64_t m,
                     46.                int64_t n,
                     47.                double **restrict A,
                     48.                double *restrict b,
                     49.                double *restrict c)
                     50. {
                         <Function: mxv_core>
   0.        0.          [50]   401d56:  mov    0x8(%rsp),%r10
                     51.   for (int64_t i=row_index_start;
                                         i<=row_index_end; i++)
   0.        0.          [51]   401d5b:  cmp    %rsi,%rdi
   0.        0.          [51]   401d5e:  jg     0x47
   0.        0.          [51]   401d60:  add    $0x1,%rsi
   0.        0.          [51]   401d64:  jmp    0x36
                     52.     {
                     53.       double row_sum = 0.0;
                     54.       for (int64_t j=0; j<n; j++)
                     55          row_sum += A[i][j] * b[j];
   0.        0.          [55]   401d66:  mov    (%r8,%rdi,8),%rdx
```

```
   0.        0.           [54]   401d6a:  mov    $0x0,%eax
   0.        0.           [53]   401d6f:  pxor   %xmm1,%xmm1
   0.110     0.110        [55]   401d73:  movsd  (%rdx,%rax,8),%xmm0
   1.921     1.921        [55]   401d78:  mulsd  (%r9,%rax,8),%xmm0
   2.282     2.282        [55]   401d7e:  addsd  %xmm0,%xmm1
## 4.613     4.613        [54]   401d82:  add    $0x1,%rax
   0.        0.           [54]   401d86:  cmp    %rax,%rcx
   0.        0.           [54]   401d89:  jne    0xffffffffffffffea
                     56.       c[i] = row_sum;
   0.        0.           [56]   401d8b:  movsd  %xmm1,(%r10,%rdi,8)
   0.        0.           [51]   401d91:  add    $0x1,%rdi
   0.        0.           [51]   401d95:  cmp    %rsi,%rdi
   0.        0.           [51]   401d98:  je     0xd
   0.        0.           [53]   401d9a:  pxor   %xmm1,%xmm1
   0.        0.           [54]   401d9e:  test   %rcx,%rcx
   0.        0.           [54]   401da1:  jg     0xffffffffffffffc5
   0.        0.           [54]   401da3:  jmp    0xffffffffffffffe8
                     57.     }
                     58. }
   0.        0.           [58]   401da5:  ret
```

For each instruction, the timing values are given and we can immediately identify the most expensive instructions. As with the source level view, these are marked with the **##** symbol.

It comes as no surprise that the time consuming instructions originate from the source code at lines 54-55. One thing to note is that the source line numbers no longer appear in sequential order. This is because the compiler has re-ordered the instructions as part of the code optimizations it has performed.

As illustrated below and similar to the `lines` command, we can get an overview of the instructions executed by using the `pcs` command.

Below the command and the output, which again has been restricted to 10 lines. As before, some lines have been shortened for lay-out purposes.

```
$ gprofng display text -pcs test.1.er
```

```
PCs sorted by metric: Exclusive Total CPU Time

Excl. Total   Incl. Total    Name
CPU           CPU
 sec.      %   sec.      %
9.367 100.00  9.367 100.00   <Total>
4.613  49.25  4.613  49.25   mxv_core + 0x0000002C, line 54 in "mxv.c"
2.282  24.36  2.282  24.36   mxv_core + 0x00000028, line 55 in "mxv.c"
1.921  20.51  1.921  20.51   mxv_core + 0x00000022, line 55 in "mxv.c"
0.150   1.60  0.150   1.60   init_data + 0x000000AC, line 118 in ...
0.110   1.18  0.110   1.18   mxv_core + 0x0000001D, line 55 in "mxv.c"
0.040   0.43  0.040   0.43   drand48 + 0x00000022
0.040   0.43  0.040   0.43   init_data + 0x000000F1, line 124 in ...
0.030   0.32  0.030   0.32   __drand48_iterate + 0x0000001E
```

```
    0.020   0.21  0.020   0.21   __drand48_iterate + 0x00000038
```

What we see is that the top three instructions take 94% of the total CPU time and any optimizations should focus on this part of the code..

3.1.5 Display and Define the Metrics

The metrics shown by `gprofng display text` are useful, but there is more recorded than displayed by default. We can customize the values shown by defining the metrics ourselves.

There are two commands related to changing the metrics shown: `metric_list` and `metrics`.

The first command shows the currently selected metrics, plus all the metrics that have been stored as part of the experiment. The second command may be used to define the metric list.

This is the way to get the information about the metrics:

```
$ gprofng display text -metric_list test.1.er
```

This is the output:

```
Current metrics: e.%totalcpu:i.%totalcpu:name
Current Sort Metric: Exclusive Total CPU Time ( e.%totalcpu )
Available metrics:
Exclusive Total CPU Time: e.%totalcpu
Inclusive Total CPU Time: i.%totalcpu
                    Size: size
              PC Address: address
                    Name: name
```

This shows the metrics that are currently used, the metric that is used to sort the data and all the metrics that have been recorded, but are not necessarily shown.

In this case, the current metrics are set to the exclusive and inclusive total CPU times, the respective percentages, and the name of the function, or load object.

The `metrics` command is used to define the metrics that need to be displayed.

For example, to swap the exclusive and inclusive metrics, use the following metric definition: `i.%totalcpu:e.%totalcpu`.

Since the metrics can be tailored for different views, there is also a way to reset them to the default. This is done through the special keyword `default` for the metrics definition (`-metrics default`).

3.1.6 Customization of the Output

With the information just given, the function overview can be customized. For sake of the example, we would like to display the name of the function

first, only followed by the exclusive CPU time, given as an absolute number and a percentage.

Note that the commands are parsed in order of appearance. This is why we need to define the metrics *before* requesting the function overview:

```
$ gprofng display text -metrics name:e.%totalcpu -functions test.1.er
```

```
Current metrics: name:e.%totalcpu
Current Sort Metric: Exclusive Total CPU Time ( e.%totalcpu )
Functions sorted by metric: Exclusive Total CPU Time

Name                                      Excl. Total
                                          CPU
                                           sec.      %
 <Total>                                  9.367 100.00
 mxv_core                                 8.926  95.30
 init_data                                0.210   2.24
 drand48                                  0.080   0.85
 erand48_r                                0.070   0.75
 __drand48_iterate                        0.060   0.64
 _int_malloc                              0.010   0.11
 sysmalloc                                0.010   0.11
 <static>@0x47960 (<libgp-collector.so>) 0.      0.
 __libc_start_main                        0.      0.
 allocate_data                            0.      0.
 driver_mxv                               0.      0.
 main                                     0.      0.
 malloc                                   0.      0.
 start_thread                             0.      0.
```

This was a first and simple example how to customize the output. Note that we did not rerun our profiling job and merely modified the display settings. Below we will show other and also more advanced examples of customization.

3.1.7 Name the Experiment Directory

When using `gprofng collect app`, the default names for experiments work fine, but they are quite generic. It is often more convenient to select a more descriptive name. For example, one that reflects conditions for the experiment conducted, like the number of threads used.

For this, the mutually exclusive `-o` and `-O` options come in handy. Both may be used to provide a name for the experiment directory, but the behaviour of `gprofng collect app` is different.

With the '`-o`' option, an existing experiment directory is not overwritten. Any directory with the same name either needs to be renamed, moved, or removed, before the experiment can be conducted.

This is in contrast with the behaviour for the '`-O`' option. Any existing directory with the same name is silently overwritten.

Be aware that the name of the experiment directory has to end with `.er`.

3.1.8 Control the Number of Lines in the Output

The `limit <n>` command can be used to control the number of lines printed in various views. For example it impacts the function view, but also takes effect for other display commands, like `lines`.

The argument `<n>` should be a positive integer number. It sets the number of lines in the (function) view. A value of zero resets the limit to the default.

Be aware that the pseudo-function `<Total>` counts as a regular function. For example `limit 10` displays nine user level functions.

3.1.9 Sorting the Performance Data

The `sort <key>` command sets the key to be used when sorting the performance data.

The key is a valid metric definition, but the visibility field (See Section 7.3 [Metric Definitions], page 59) in the metric definition is ignored, since this does not affect the outcome of the sorting operation. For example if the sort key is set to `e.totalcpu`, the values will be sorted in descending order with respect to the exclusive total CPU time.

The data can be sorted in reverse order by prepending the metric definition with a minus ('`-`') sign. For example `sort -e.totalcpu`.

A default metric for the sort operation has been defined and since this is a persistent command, this default can be restored with `default` as the key (`sort default`).

3.1.10 Scripting

The list with commands for `gprofng display text` can be very long. This is tedious and also error prone. Luckily, there is an easier and elegant way to control the output of this tool.

Through the `script` command, the name of a file with commands can be passed in. These commands are parsed and executed as if they appeared on the command line in the same order as encountered in the file. The commands in this script file can actually be mixed with commands on the command line and multiple script files may be used. The difference between the commands in the script file and those used on the command line is that the latter require a leading dash ('`-`') symbol.

Comment lines in a script file are supported. They need to start with the '`#`' symbol.

3.1.11 A More Elaborate Example

With the information presented so far, we can customize our data gathering and display commands.

As an example, we would like to use `mxv.1.thr.er` as the name for the experiment directory. In this way, the name of the algorithm and the number of threads that were used are included in the name. We also don't mind to overwrite an existing experiment directory with the same name.

All that needs to be done is to use the '`-O`' option, followed by the directory name of choice when running `gprofng collect app`:

```
$ exe=mxv-pthreads
$ m=8000
$ n=4000
$ gprofng collect app -O mxv.1.thr.er ./$exe -m $m -n $n -t 1
```

Since we want to customize the profile and prefer to keep the command line short, the commands to generate the profile are put into a file with the name `my-script`:

```
$ cat my-script
# This is my first gprofng script
# Set the metrics
metrics i.%totalcpu:e.%totalcpu:name
# Use the exclusive time to sort
sort e.totalcpu
# Limit the function list to 5 lines
limit 5
# Show the function list
functions
```

This script file is specified as input to the `gprofng display text` command that is used to display the performance information stored in experiment directory `mxv.1.thr.er`:

```
$ gprofng display text -script my-script mxv.1.thr.er
```

This command produces the following output:

```
# This is my first gprofng script
# Set the metrics
Current metrics: i.%totalcpu:e.%totalcpu:name
Current Sort Metric: Exclusive Total CPU Time ( e.%totalcpu )
# Use the exclusive time to sort
Current Sort Metric: Exclusive Total CPU Time ( e.%totalcpu )
# Limit the function list to 5 lines
Print limit set to 5
# Show the function list
Functions sorted by metric: Exclusive Total CPU Time

Incl. Total   Excl. Total    Name
CPU           CPU
 sec.      %   sec.      %
9.703 100.00  9.703 100.00   <Total>
9.226  95.09  9.226  95.09   mxv_core
```

```
0.455   4.69  0.210   2.17   init_data
0.169   1.75  0.123   1.26   erand48_r
0.244   2.52  0.075   0.77   drand48
```

In the first part of the output the comment lines in the script file are echoed. These are interleaved with an acknowledgement message for the commands.

This is followed by a profile consisting of 5 lines only. For both metrics, the percentages plus the timings are given. The numbers are sorted with respect to the exclusive total CPU time. Although this is the default, for demonstration purposes we use the `sort` command to explicitly define the metric for the sort.

While we executed the same job as before and only changed the name of the experiment directory, the results are somewhat different. This is sampling in action. The numbers are not all that different though. It is seen that function `mxv_core` is responsbile for 95% of the CPU time and `init_data` takes 4.5% only.

3.1.12 The Call Tree

The call tree shows the dynamic structure of the application by displaying the functions executed and their parent. The CPU time attributed to each function is shown as well. This view helps to find the most expensive execution path in the program.

This feature is enabled through the `calltree` command. For example, this is how to get the call tree for our current experiment:

```
$ gprofng display text -calltree mxv.1.thr.er
```

This displays the following structure:

```
Functions Call Tree. Metric: Attributed Total CPU Time

Attr. Total    Name
CPU
 sec.      %
9.703 100.00   +-<Total>
9.226  95.09     +-start_thread
9.226  95.09     |  +-<static>@0x47960 (<libgp-collector.so>)
9.226  95.09     |    +-driver_mxv
9.226  95.09     |      +-mxv_core
0.477   4.91     +-__libc_start_main
0.477   4.91       +-main
0.455   4.69         +-init_data
0.244   2.52         |  +-drand48
0.169   1.75         |    +-erand48_r
0.047   0.48         |      +-__drand48_iterate
0.021   0.22         +-allocate_data
0.021   0.22         |  +-malloc
0.021   0.22         |    +-_int_malloc
```

```
0.006   0.06          |      +-sysmalloc
0.003   0.03          |        +-__default_morecore
0.003   0.03          |          +-sbrk
0.003   0.03          |            +-brk
0.001   0.01         +-pthread_create
0.001   0.01           +-__pthread_create_2_1
```

At first sight this may not be what is expected and some explanation is in place.

The top function is the pseudo-function `<Total>` that we have seen before. It is introduced and shown here to provide the total value of the metric(s).

We also see function `<static>@0x47960` in the call tree and apparently it is from `libgp-collector.so`, a library that is internal to `gprofng`. The `<static>` marker, followed by the program counter, is shown if the name of the function cannot be found. This function is part of the implementation of the data collection process and should be hidden to the user. This is part of a planned future enhancement.

In general, if a view has a function that does not appear to be part of the user code, or seems odd anyhow, the `objects` and `fsingle` commands are very useful to find out more about load objects in general, but also to help identify an unknown entry in the function overview. See Section 7.6 [Load Objects and Functions], page 62.

Another thing to note is that there are two main branches. The one under `<static>@0x47960` and the second one under `__libc_start_main`. This reflects the fact that this is a multithreaded program and the threaded part shows up as a separate branch in the call tree.

The way to interpret this structure is as follows. The program starts under control of `__libc_start_main`. This executes the main program called `main`, which at the top level executes functions `init_data`, `allocate_data`, and `pthread_create`. The latter function creates and executes the additional thread(s).

For this multithreaded part of the code, we need to look at the branch under function `start_thread` that calls the driver code for the matrix-vector multiplication (`driver_mxv`), which executes the function that performs the actual multiplication (`mxv_core`).

There are two things worth noting for the call tree feature:

- This is a dynamic tree and since sampling is used, it most likely looks slighlty different across seemingly identical profile runs. In case the run times are short, it is worth considering to use a high resolution through the '`-p`' option. For example use '`-p hi`' to increase the sampling rate.
- In case hardware event counters have been enabled (See Section 3.4 [Profile Hardware Event Counters], page 34), these values are also displayed in the call tree view.

3.1.13 More Information on the Experiment

The experiment directory not only contains performance related data. Several system characteristics, the profiling command executed, plus some global performance statistics are stored and can be displayed.

The `header` command displays information about the experiment(s). For example, this is command is used to extract this data from for our experiment directory:

```
$ gprofng display text -header mxv.1.thr.er
```

The above command prints the following information. Note that some of the lay-out and the information has been modified. Directory paths have been replaced `<apath>` for example. Textual changes are marked with the '<' and '>' symbols.

```
Experiment: mxv.1.thr.er
No errors
No warnings
Archive command ‘ /usr/bin/gp-archive -n -a on --outfile
                     <apath>/archive.log <apath>/mxv.1.thr.er’

Target command (64-bit): ’./mxv-pthreads -m 8000 -n 4000 -t 1’
Process pid 2750071, ppid 2750069, pgrp 2749860, sid 2742080
Current working directory: <apath>
Collector version: ‘2.40.00’; experiment version 12.4 (64-bit)
Host ‘<the-host-name>’, OS ‘Linux <version>’, page size 4096,
                     architecture ‘x86_64’
  4 CPUs, clock speed 2294 MHz.
  Memory: 3506491 pages @  4096 = 13697 MB.
Data collection parameters:
  Clock-profiling, interval = 997 microsecs.
  Periodic sampling, 1 secs.
  Follow descendant processes from: fork|exec|combo

Experiment started <date and time>

Experiment Ended: 9.801216173
Data Collection Duration: 9.801216173
```

The output above may assist in troubleshooting, or to verify some of the operational conditions and we recommend to include this command when generating a profile.

Related to this command there is a useful option to record comment(s) in an experiment. To this end, use the '`-C`' option on the `gprofng collect app` tool to specify a comment string. Up to ten comment lines can be included. These comments are displayed with the `header` command on the `gprofng display text` tool.

The `overview` command displays information on the experiment(s) and also shows a summary of the values for the metric(s) used. This is an example how to use it on the newly created experiment directory:

```
$ gprofng display text -overview mxv.1.thr.er
```

```
Experiment(s):

Experiment      :mxv.1.thr.er
  Target        : './mxv-pthreads -m 8000 -n 4000 -t 1'
  Host          : <hostname> (<ISA>, Linux <version>)
  Start Time    : <date and time>
  Duration      : 9.801 Seconds

Metrics:

  Experiment Duration (Seconds): [9.801]
  Clock Profiling
    [X]Total CPU Time - totalcpu (Seconds): [*9.703]

Notes: '*' indicates hot metrics, '[X]' indicates currently enabled
       metrics.
       The metrics command can be used to change selections. The
       metric_list command lists all available metrics.
```

This command provides a dashboard overview that helps to easily identify where the time is spent and in case hardware event counters are used, it shows their total values.

3.1.14 Control the Sampling Frequency

So far we did not go into details on the frequency of the sampling process, but in some cases it is useful to change the default of 10 milliseconds.

The advantage of increasing the sampling frequency is that functions that do not take much time per invocation are more accurately captured. The downside is that more data is gathered. This has an impact on the overhead of the collection process and more disk space is required.

In general this is not an immediate concern, but with heavily threaded applications that run for an extended period of time, increasing the frequency may have a more noticeable impact.

The `-p` option on the `gprofng collect app` tool is used to enable or disable clock based profiling, or to explicitly set the sampling rate. This option takes one of the following keywords:

`off` Disable clock based profiling.

`on` Enable clock based profiling with a per thread sampling interval of 10 ms. This is the default.

`lo` Enable clock based profiling with a per thread sampling interval of 100 ms.

`hi` Enable clock based profiling with a per thread sampling interval of 1 ms.

`value` Enable clock based profiling with a per thread sampling interval of *value*.

It may seem unnecessary to have an option to disable clock based profiling, but there is a good reason to support this. By default, clock profiling is enabled when conducting hardware event counter experiments (See Section 3.4 [Profile Hardware Event Counters], page 34). With the `-p off` option, this can be disabled.

If an explicit value is set for the sampling, the number can be an integer or a floating-point number. A suffix of '`u`' for microseconds, or '`m`' for milliseconds is supported. If no suffix is used, the value is assumed to be in milliseconds.

For example, the following command sets the sampling rate to 5123.4 microseconds:

```
$ gprofng collect app -p 5123.4u ./mxv-pthreads -m 8000 -n 4000 -t 1
```

If the value is smaller than the clock profiling minimum, a warning message is issued and it is set to the minimum. In case it is not a multiple of the clock profiling resolution, it is silently rounded down to the nearest multiple of the clock resolution. If the value exceeds the clock profiling maximum, is negative, or zero, an error is reported.

Note that the `header` command echoes the sampling rate used.

3.1.15 Information on Load Objects

It may happen that the function view shows a function that is not known to the user. This can easily happen with library functions for example. Luckily there are three commands that come in handy then.

These commands are `objects`, `fsingle`, and `fsummary`. They provide details on load objects (See Section 7.6 [Load Objects and Functions], page 62).

The `objects` command lists all load objects that have been referenced during the performance experiment. Below we show the command and the result for our profile job. Like before, some path names in the output have been shortened and replaced by the `<apath>` symbol that represents an absolute directory path.

```
$ gprofng display text -objects mxv.1.thr.er
```

The output includes the name and path of the target executable:

```
<Unknown> (<Unknown>)
<mxv-pthreads> (<apath>/mxv-pthreads)
```

```
<libdl-2.28.so> (/usr/lib64/libdl-2.28.so)
<librt-2.28.so> (/usr/lib64/librt-2.28.so)
<libc-2.28.so> (/usr/lib64/libc-2.28.so)
<libpthread-2.28.so> (/usr/lib64/libpthread-2.28.so)
<libm-2.28.so> (/usr/lib64/libm-2.28.so)
<libgp-collector.so> (/usr/lib64/gprofng/libgp-collector.so)
<ld-2.28.so> (/usr/lib64/ld-2.28.so)
<DYNAMIC_FUNCTIONS> (DYNAMIC_FUNCTIONS)
```

The `fsingle` command may be used to get more details on a specific entry in the function view, say. For example, the command below provides additional information on the `pthread_create` function shown in the function overview.

```
$ gprofng display text -fsingle pthread_create mxv.1.thr.er
```

Below the output from this command. It has been somewhat modified to match the display requirements.

```
+ gprofng display text -fsingle pthread_create mxv.1.thr.er
pthread_create
        Exclusive Total CPU Time: 0.     (  0. %)
        Inclusive Total CPU Time: 0.001 (  0.0%)
                            Size:   258
                      PC Address: 8:0x00049f60
                     Source File: (unknown)
                     Object File: (unknown)
                     Load Object: /usr/lib64/gprofng/libgp-collector.so
                    Mangled Name:
                         Aliases:
```

In this table we not only see how much time was spent in this function, we also see where it originates from. In addition to this, the size and start address are given as well. If the source code location is known it is also shown here.

The related `fsummary` command displays the same information as `fsingle`, but for all functions in the function overview, including `<Total>`:

```
$ gprofng display text -fsummary mxv.1.thr.er
```

```
Functions sorted by metric: Exclusive Total CPU Time

<Total>
        Exclusive Total CPU Time: 9.703 (100.0%)
        Inclusive Total CPU Time: 9.703 (100.0%)
                            Size:     0
                      PC Address: 1:0x00000000
                     Source File: (unknown)
                     Object File: (unknown)
                     Load Object: <Total>
                    Mangled Name:
```

```
                        Aliases:

mxv_core
        Exclusive Total CPU Time: 9.226 ( 95.1%)
        Inclusive Total CPU Time: 9.226 ( 95.1%)
                            Size:    80
                      PC Address: 2:0x00001d56
                     Source File: <apath>/src/mxv.c
                     Object File: mxv.1.thr.er/archives/mxv-pthreads_ss_pf53V__5
                     Load Object: <apath>/mxv-pthreads
                    Mangled Name:
                        Aliases:

          ... etc ...
```

3.2 Support for Multithreading

In this chapter the support for multithreading is introduced and discussed. As is shown below, nothing needs to be changed when collecting the performance data.

The difference is that additional commands are available to get more information on the multithreading details, plus that several filters allow the user to zoom in on specific threads.

3.2.1 Creating a Multithreading Experiment

We demonstrate the support for multithreading using the same code and settings as before, but this time 2 threads are used:

```
$ exe=mxv-pthreads
$ m=8000
$ n=4000
$ gprofng collect app -O mxv.2.thr.er ./$exe -m $m -n $n -t 2
```

First of all, in as far as gprofng is concerned, no changes are needed. Nothing special is needed to profile a multithreaded job when using `gprofng`.

The same is true when displaying the performance results. The same commands that were used before work unmodified. For example, this is all that is needed to get a function overview:

```
$ gprofng display text -limit 5 -functions mxv.2.thr.er
```

This produces the following familiar looking output:

```
Print limit set to 5
Functions sorted by metric: Exclusive Total CPU Time

Excl. Total   Incl. Total    Name
CPU           CPU
```

```
 sec.      %   sec.      %
9.464 100.00  9.464 100.00   <Total>
8.961  94.69  8.961  94.69   mxv_core
0.224   2.37  0.469   4.95   init_data
0.105   1.11  0.177   1.88   erand48_r
0.073   0.77  0.073   0.77   __drand48_iterate
```

3.2.2 Commands Specific to Multithreading

The function overview shown above shows the results aggregated over all the threads. The interesting new element is that we can also look at the performance data for the individual threads.

The `thread_list` command displays how many threads have been used:

```
$ gprofng display text -thread_list mxv.2.thr.er
```

This produces the following output, showing that three threads have been used:

```
Exp Sel Total
=== === =====
  1 all     3
```

The output confirms there is one experiment and that by default all threads are selected.

It may seem surprising to see three threads here, since we used the `-t 2` option, but it is common for a Pthreads program to use one additional thread. Typically, there is one main thread that runs from start to finish. It handles the sequential portions of the code, as well as thread management related tasks. It is no different in the example code. At some point, the main thread creates and activates the two threads that perform the multiplication of the matrix with the vector. Upon completion of this computation, the main thread continues.

The `threads` command is simple, yet very powerful. It shows the total value of the metrics for each thread.

```
$ gprofng display text -threads mxv.2.thr.er
```

The command above produces the following overview:

```
Objects sorted by metric: Exclusive Total CPU Time

Excl. Total    Name
CPU
 sec.      %
9.464 100.00   <Total>
4.547  48.05   Process 1, Thread 3
4.414  46.64   Process 1, Thread 2
0.502   5.31   Process 1, Thread 1
```

The first line gives the total CPU time accumulated over the threads selected. This is followed by the metric value(s) for each thread.

From this it is clear that the main thread is responsible for a little over 5% of the total CPU time, while the other two threads take 47-48% each.

This view is ideally suited to verify if there are any load balancing issues and also to find the most time consuming thread(s).

While useful, often more information than this is needed. This is where the thread selection filter comes in. Through the `thread_select` command, one or more threads may be selected. See Section 7.5 [The Selection List], page 60, how to define the selection list.

Since it is most common to use this command in a script, we do so as well here. Below the script we are using:

```
# Define the metrics
metrics e.%totalcpu
# Limit the output to 5 lines
limit 5
# Get the function overview for thread 1
thread_select 1
functions
# Get the function overview for thread 2
thread_select 2
functions
# Get the function overview for thread 3
thread_select 3
functions
```

The definition of the metrics and the output limit have been shown and explained earlier. The new command to focus on is `thread_select`.

This command takes a list (See Section 7.5 [The Selection List], page 60) to select specific threads. In this case, the individual thread numbers that were obtained earlier with the `thread_list` command are selected.

This restricts the output of the `functions` command to the thread number(s) specified. This means that the script above shows which function(s) each thread executes and how much CPU time they consumed. Both the exclusive timings and their percentages are given.

Note that technically this command is a filter and persistent. The selection remains active until changed through another thread selection command, or when it is reset with the '`all`' selection list.

This is the relevant part of the output for the first thread:

```
Exp Sel Total
=== === =====
  1 1       3
Functions sorted by metric: Exclusive Total CPU Time
```

```
Excl. Total    Name
CPU
 sec.      %
0.502 100.00   <Total>
0.224  44.64   init_data
0.105  20.83   erand48_r
0.073  14.48   __drand48_iterate
0.067  13.29   drand48
```

As usual, the comment lines are echoed. This is followed by a confirmation of the selection. The first table shows that one experiment is loaded and that thread 1 out of the three threads has been selected. What is displayed next is the function overview for this particular thread. Due to the `limit 5` command, there are only five functions in this list.

Clearly, this thread handles the data initialization part and as we know from the call tree output, function `init_data` executes the 3 other functions shown in this profile.

Below are the overviews for threads 2 and 3 respectively. It is seen that all of the CPU time is spent in function `mxv_core` and that this time is approximately the same for both threads.

```
# Get the function overview for thread 2
Exp Sel Total
=== === =====
  1 2       3
Functions sorted by metric: Exclusive Total CPU Time

Excl. Total    Name
CPU
 sec.      %
4.414 100.00   <Total>
4.414 100.00   mxv_core
0.      0.     <static>@0x48630 (<libgp-collector.so>)
0.      0.     driver_mxv
0.      0.     start_thread

# Get the function overview for thread 3
Exp Sel Total
=== === =====
  1 3       3
Functions sorted by metric: Exclusive Total CPU Time

Excl. Total    Name
CPU
 sec.      %
4.547 100.00   <Total>
4.547 100.00   mxv_core
0.      0.     <static>@0x48630 (<libgp-collector.so>)
0.      0.     driver_mxv
0.      0.     start_thread
```

When analyzing the performance of a multithreaded application, it is sometimes useful to know whether threads have mostly executed on the same core, say, or if they have wandered across multiple cores. This sort of stickiness is usually referred to as *thread affinity*.

Similar to the commands for the threads, there are several commands related to the usage of the cores, or *CPUs* as they are called in `gprofng` (See Section 7.7 [The Concept of a CPU in gprofng], page 62).

Similar to the `thread_list` command, the `cpu_list` command displays how many CPUs have been used. The equivalent of the `threads` threads command, is the `cpus` command, which shows the numbers of the CPUs that were used and the metric values for each one of them. Both commands are demonstrated below.

```
$ gprofng display text -cpu_list -cpus mxv.2.thr.er
```

This command produces the following output:

```
+ gprofng display text -cpu_list -cpus mxv.2.thr.er
Exp Sel Total
=== === =====
  1 all     4
Objects sorted by metric: Exclusive Total CPU Time

Excl. Total    Name
CPU
 sec.      %
9.464 100.00   <Total>
4.414  46.64   CPU 2
2.696  28.49   CPU 0
1.851  19.56   CPU 1
0.502   5.31   CPU 3
```

The first table shows that there is only one experiment and that all of the four CPUs have been selected. The second table shows the exclusive metrics for each of the CPUs that have been used.

As also echoed in the output, the data is sorted with respect to the exclusive CPU time, but it is very easy to sort the data by the CPU id by using the `sort` command:

```
$ gprofng display text -cpu_list -sort name -cpus mxv.2.thr.er
```

With the `sort` added, the output is as follows:

```
Exp Sel Total
=== === =====
  1 all     4
Current Sort Metric: Name ( name )
Objects sorted by metric: Name
```

```
Excl. Total    Name
CPU
 sec.      %
9.464 100.00   <Total>
2.696  28.49   CPU 0
1.851  19.56   CPU 1
4.414  46.64   CPU 2
0.502   5.31   CPU 3
```

While the table with thread times shown earlier may point at a load imbalance in the application, this overview has a different purpose.

For example, we see that 4 CPUs have been used, but we know that the application uses 3 threads only. We will now demonstrate how filters can be used to help answer the question why 4 CPUs are used, while the application has 3 threads only. This means that at least one thread has executed on more than one CPU.

Recall the thread level timings:

```
Excl. Total    Name
CPU
 sec.      %
9.464 100.00   <Total>
4.547  48.05   Process 1, Thread 3
4.414  46.64   Process 1, Thread 2
0.502   5.31   Process 1, Thread 1
```

Compared to the CPU timings above, it seems very likely that thread 3 has used more than one CPU, because the thread and CPU timings are the same for both other threads.

The command below selects thread number 3 and then requests the CPU utilization for this thread:

```
$ gprofng display text -thread_select 3 -sort name -cpus mxv.2.thr.er
```

The output shown below confirms that thread 3 is selected and then displays the CPU(s) that have been used by this thread:

```
Exp Sel Total
=== === =====
  1 3       3

Objects sorted by metric: Exclusive Total CPU Time

Excl. Total    Name
CPU
 sec.      %
4.547 100.00   <Total>
2.696  59.29   CPU 0
1.851  40.71   CPU 1
```

The results show that this thread has used CPU 0 nearly 60% of the time and CPU 1 for the remaining 40%.

To confirm that this is the only thread that has used more than one CPU, the same approach can be used for threads 1 and 2:

```
$ gprofng display text -thread_select 1 -cpus mxv.2.thr.er
Exp Sel Total
=== === =====
  1 1       3
Objects sorted by metric: Exclusive Total CPU Time

Excl. Total    Name
CPU
 sec.      %
0.502 100.00   <Total>
0.502 100.00   CPU 3

$ gprofng display text -thread_select 2 -cpus mxv.2.thr.er
Exp Sel Total
=== === =====
  1 2       3
Objects sorted by metric: Exclusive Total CPU Time

Excl. Total    Name
CPU
 sec.      %
4.414 100.00   <Total>
4.414 100.00   CPU 2
```

The output above shows that indeed threads 1 and 2 each have used a single CPU only.

3.3 View Multiple Experiments

One thing we did not cover sofar is that `gprofng` fully supports the analysis of multiple experiments. The `gprofng display text` tool accepts a list of experiments. The data can either be aggregated across the experiments, or used in a comparison.

The default is to aggregate the metric values across the experiments that have been loaded. The `compare` command can be used to enable the comparison of results.

In this section both modes are illustrated with an example.

3.3.1 Aggregation of Experiments

If the data for multiple experiments is aggregrated, the `gprofng display text` tool shows the combined results. For example, below is the script to show the function view for the data aggregated over two experiments, drop the first experiment and then show the function view fo the second experiment only. We will call it `my-script-agg`.

```
# Define the metrics
metrics e.%totalcpu
# Limit the output to 5 lines
limit 5
# Get the list with experiments
experiment_list
# Get the function overview for all
functions
# Drop the first experiment
drop_exp mxv.2.thr.er
# Get the function overview for exp #2
functions
```

With the exception of the `experiment_list` command, all commands used have been discussed earlier.

The `experiment_list` command provides a list of the experiments that have been loaded. This may be used to get the experiment IDs and to verify the correct experiments are loaded for the aggregation.

Below is an example that loads two experiments and uses the above script to display different function views.

```
$ gprofng display text -script my-script-agg mxv.2.thr.er mxv.4.thr.er
```

This produces the following output:

```
# Define the metrics
Current metrics: e.%totalcpu:name
Current Sort Metric: Exclusive Total CPU Time ( e.%totalcpu )
# Limit the output to 5 lines
Print limit set to 5
# Get the list with experiments
ID Sel     PID Experiment
== === ======= ============
 1 yes 1339450 mxv.2.thr.er
 2 yes 3579561 mxv.4.thr.er
# Get the function overview for all
Functions sorted by metric: Exclusive Total CPU Time

Excl. Total     Name
CPU
  sec.      %
20.567 100.00   <Total>
19.553  95.07   mxv_core
 0.474   2.30   init_data
 0.198   0.96   erand48_r
 0.149   0.72   drand48

# Drop the first experiment
Experiment mxv.2.thr.er has been dropped
```

```
# Get the function overview for exp #2
Functions sorted by metric: Exclusive Total CPU Time

Excl. Total     Name
CPU
  sec.      %
11.104 100.00   <Total>
10.592  95.39   mxv_core
 0.249   2.24   init_data
 0.094   0.84   erand48_r
 0.082   0.74   drand48
```

The first five lines should look familiar. The five lines following echo the comment line in the script and show the overview of the experiments. This confirms two experiments have been loaded and that both are active. This is followed by the function overview. The timings have been summed up and the percentages are adjusted accordingly.

3.3.2 Comparison of Experiments

The support for multiple experiments really shines in comparison mode. In comparison mode, the data for the various experiments is shown side by side, as illustrated below where we compare the results for the multithreaded experiments using two and four threads respectively.

This feature is controlled through the `compare` command.

The comparison mode is enabled through `compare on` and with `compare off` it is disabled again. In addition to '`on`', or '`off`', this command also supports the '`delta`' and '`ratio`' keywords.

This is the script that will be used in our example. It sets the comparison mode to '`on`':

```
# Define the metrics
metrics e.%totalcpu
# Limit the output to 5 lines
limit 5
# Set the comparison mode to differences
compare on
# Get the function overview
functions
```

Assuming this script file is called `my-script-comp`, this is how it is used to display the differences:

```
$ gprofng display text -script my-script-comp mxv.2.thr.er mxv.4.thr.er
```

This produces the output shown below. The data for the first experiment is shown as absolute numbers. The timings for the other experiment are shown as a delta relative to these reference numbers:

```
mxv.2.thr.er  mxv.4.thr.er
```

```
Excl. Total   Excl. Total     Name
CPU           CPU
 sec.      %    sec.      %
9.464 100.00  11.104 100.00   <Total>
8.961  94.69  10.592  95.39   mxv_core
0.224   2.37   0.249   2.24   init_data
0.105   1.11   0.094   0.84   erand48_r
0.073   0.77   0.060   0.54   __drand48_iterate
```

This table is already helpful to more easily compare (two) profiles, but there is more that we can do here.

By default, in comparison mode, all measured values are shown. Often profiling is about comparing performance data. It is therefore sometimes more useful to look at differences or ratios, using one experiment as a reference.

The values shown are relative to this difference. For example if a ratio is below one, it means the reference value was higher.

In the example below, we use the same two experiments used in the comparison above. The script is also nearly identical. The only change is that we now use the '`delta`' keyword.

As before, the number of lines is restricted to 5 and we focus on the exclusive timings plus percentages. For the comparison part we are interested in the differences.

This is the script that produces such an overview:

```
# Define the metrics
metrics e.%totalcpu
# Limit the output to 5 lines
limit 5
# Set the comparison mode to differences
compare delta
# Get the function overview
functions
```

Assuming this script file is called `my-script-comp2`, this is how we get the table displayed on our screen:

```
$ gprofng display text -script my-script-comp2 mxv.2.thr.er mxv.4.thr.er
```

Leaving out some of the lines printed, but we have seen before, we get the following table:

```
mxv.2.thr.er  mxv.4.thr.er
Excl. Total   Excl. Total     Name
CPU           CPU
 sec.      %   delta      %
9.464 100.00  +1.640 100.00   <Total>
8.961  94.69  +1.631  95.39   mxv_core
0.224   2.37  +0.025   2.24   init_data
0.105   1.11  -0.011   0.84   erand48_r
0.073   0.77  -0.013   0.54   __drand48_iterate
```

It is now easier to see that the CPU times for the most time consuming functions in this code are practically the same.

It is also possible to show ratio's through the `compare ratio` command. The first colum is used as a reference and the values for the other columns with metrics are derived by dividing the value by the reference. The result for such a comparison is shown below:

```
mxv.2.thr.er  mxv.4.thr.er
Excl. Total   Excl. Total CPU    Name
CPU
 sec.      %      ratio      %
9.464 100.00  x   1.173 100.00   <Total>
8.961  94.69  x   1.182  95.39   mxv_core
0.224   2.37  x   1.111   2.24   init_data
0.105   1.11  x   0.895   0.84   erand48_r
0.073   0.77  x   0.822   0.54   __drand48_iterate
```

Note that the comparison feature is supported at the function, source, and disassembly level. There is no practical limit on the number of experiments that can be used in a comparison.

3.4 Profile Hardware Event Counters

Many processors provide a set of hardware event counters and `gprofng` provides support for this feature. See Section 7.8 [Hardware Event Counters Explained], page 63, for those readers that are not familiar with such counters and like to learn more.

In this section we explain how to get the details on the event counter support for the processor used in the experiment(s), and show several examples.

3.4.1 Getting Information on the Counters Supported

The first step is to check if the processor used for the experiments is supported by `gprofng`. The `-h` option on `gprofng collect app` will show the event counter information:

```
$ gprofng collect app -h
```

In case the counters are supported, a list with the events is printed. Otherwise, a warning message will be issued.

For example, below we show this command and the output on an Intel Xeon Platinum 8167M (aka "Skylake") processor. The output has been split into several sections and each section is commented upon separately.

```
Run "gprofng collect app --help" for a usage message.

Specifying HW counters on 'Intel Arch PerfMon v2 on Family 6 Model 85'
(cpuver=2499):
```

```
-h {auto|lo|on|hi}
       turn on default set of HW counters at the specified rate
-h <ctr_def> [-h <ctr_def>]...
-h <ctr_def>[,<ctr_def>]...
       specify HW counter profiling for up to 4 HW counters
```

The first line shows how to get a usage overview. This is followed by some information on the target processor. The next five lines explain in what ways the `-h` option can be used to define the events to be monitored.

The first version shown above enables a default set of counters. This default depends on the processor this command is executed on. The keyword following the `-h` option defines the sampling rate:

`auto` Match the sample rate of used by clock profiling. If the latter is disabled, Use a per thread sampling rate of approximately 100 samples per second. This setting is the default and preferred.

`on` Use a per thread sampling rate of approximately 100 samples per second.

`lo` Use a per thread sampling rate of approximately 10 samples per second.

`hi` Use a per thread sampling rate of approximately 1000 samples per second.

The second and third variant define the events to be monitored. Note that the number of simultaneous events supported is printed. In this case we can monitor four events in a single profiling job.

It is a matter of preference whether you like to use the `-h` option for each event, or use it once, followed by a comma separated list.

There is one slight catch though. The counter definition below has mandatory comma (`,`) between the event and the rate. While a default can be used for the rate, the comma cannot be omitted. This may result in a somewhat awkward counter definition in case the default sampling rate is used.

For example, the following two commands are equivalent. Note the double comma in the second command. This is not a typo.

```
$ gprofng collect app -h cycles -h insts ...
$ gprofng collect app -h cycles,,insts ...
```

In the first command this comma is not needed, because a comma (“`,`”) immediately followed by white space may be omitted.

This is why we prefer the this syntax and in the remainder will use the first version of this command.

The counter definition takes an event name, plus optionally one or more attributes, followed by a comma, and optionally the sampling rate. The output section below shows the formal definition.

```
<ctr_def> == <ctr>[[~<attr>=<val>]...],[<rate>]
```

The printed help then explains this syntax. Below we have summarized and expanded this output:

`<ctr>` The counter name must be selected from the available counters listed as part of the output printed with the `-h` option. On most systems, if a counter is not listed, it may still be specified by its numeric value.

`~<attr>=<val>`
This is an optional attribute that depends on the processor. The list of supported attributes is printed in the output. Examples of attributes are "user", or "system". The value can given in decimal or hexadecimal format. Multiple attributes may be specified, and each must be preceded by a `~`.

`<rate>`
The sampling rate is one of the following:

`auto` This is the default and matches the rate used by clock profiling. If clock profiling is disabled, use '`on`'.

`on` Set the per thread maximum sampling rate to ~100 samples/second

`lo` Set the per thread maximum sampling rate to ~10 samples/second

`hi` Set the per thread maximum sampling rate to ~1000 samples/second

`<interval>`
Define the sampling interval. See Section 3.1.14 [Control the Sampling Frequency], page 21, how to define this.

After the section with the formal definition of events and counters, a processor specific list is displayed. This part starts with an overview of the default set of counters and the aliased names supported *on this specific processor*.

```
Default set of HW counters:

  -h cycles,,insts,,llm

Aliases for most useful HW counters:

 alias    raw name                   type units regs description

 cycles   unhalted-core-cycles   CPU-cycles 0123 CPU Cycles
```

```
insts    instruction-retired        events 0123 Instructions Executed
llm      llc-misses                 events 0123 Last-Level Cache Misses
br_msp   branch-misses-retired      events 0123 Branch Mispredict
br_ins   branch-instruction-retired events 0123 Branch Instructions
```

The definitions given above may or may not be available on other processors.

The table above shows the default set of counters defined for this processor, and the aliases. For each alias the full "raw" name is given, plus the unit of the number returned by the counter (CPU cycles, or a raw count), the hardware counter the event is allowed to be mapped onto, and a short description.

The last part of the output contains all the events that can be monitored:

```
Raw HW counters:

    name                                type      units regs description

    unhalted-core-cycles                     CPU-cycles 0123
    unhalted-reference-cycles                    events 0123
    instruction-retired                          events 0123
    llc-reference                                events 0123
    llc-misses                                   events 0123
    branch-instruction-retired                   events 0123
    branch-misses-retired                        events 0123
    ld_blocks.store_forward                      events 0123
    ld_blocks.no_sr                              events 0123
    ld_blocks_partial.address_alias              events 0123
    dtlb_load_misses.miss_causes_a_walk          events 0123
    dtlb_load_misses.walk_completed_4k           events 0123

    <many lines deleted>

    l2_lines_out.silent                          events 0123
    l2_lines_out.non_silent                      events 0123
    l2_lines_out.useless_hwpf                    events 0123
    sq_misc.split_lock                           events 0123
```

As can be seen, these names are not always easy to correlate to a specific event of interest. The processor manual should provide more clarity on this.

3.4.2 Examples Using Hardware Event Counters

The previous section may give the impression that these counters are hard to use, but as we will show now, in practice it is quite simple.

With the information from the `-h` option, we can easily set up our first event counter experiment.

We start by using the default set of counters defined for our processor and we use 2 threads:

```
$ exe=mxv-pthreads
$ m=8000
$ n=4000
$ exp=mxv.hwc.def.2.thr.er
$ gprofng collect app -O $exp -h auto ./$exe -m $m -n $n -t 2
```

The new option here is `-h auto`. The `auto` keyword enables hardware event counter profiling and selects the default set of counters defined for this processor.

As before, we can display the information, but there is one practical hurdle to take. Unless we like to view all metrics recorded, we would need to know the names of the events that have been enabled. This is tedious and also not portable in case we would like to repeat this experiment on another processor.

This is where the special `hwc` metric comes very handy. It automatically expands to the active set of events used.

With this, it is very easy to display the event counter values. Note that although the regular clock based profiling was enabled, we only want to see the counter values. We also request to see the percentages and limit the output to the first 5 lines:

```
$ exp=mxv.hwc.def.2.thr.er
$ gprofng display text -metrics e.%hwc -limit 5 -functions $exp
```

```
Current metrics: e.%cycles:e+%insts:e+%llm:name
Current Sort Metric: Exclusive CPU Cycles ( e.%cycles )
Print limit set to 5
Functions sorted by metric: Exclusive CPU Cycles

Excl. CPU     Excl. Instructions  Excl. Last-Level   Name
Cycles        Executed            Cache Misses
 sec.      %                  %                 %
2.691 100.00  7906475309 100.00   122658983 100.00   <Total>
2.598  96.54  7432724378  94.01   121745696  99.26   mxv_core
0.035   1.31   188860269   2.39       70084   0.06   erand48_r
0.026   0.95    73623396   0.93      763116   0.62   init_data
0.018   0.66    76824434   0.97       40040   0.03   drand48
```

As we have seen before, the first few lines echo the settings. This includes a list with the hardware event counters used by default.

The table that follows makes it very easy to get an overview where the time is spent and how many of the target events have occurred.

As before, we can drill down deeper and see the same metrics at the source line and instruction level. Other than using `hwc` in the metrics definitions, nothing has changed compared to the previous examples:

```
$ exp=mxv.hwc.def.2.thr.er
$ gprofng display text -metrics e.hwc -source mxv_core $exp
```

This is the relevant part of the output. Since the lines get very long, we have somewhat modified the lay-out:

```
   Excl. CPU Excl.        Excl.
   Cycles    Instructions Last-Level
    sec.     Executed     Cache Misses
                                        <Function: mxv_core>
   0.                0          0   32. void __attribute__ ((noinline))
                                        mxv_core(...)
   0.                0          0   33. {
   0.                0          0   34.   for (uint64_t i=...) {
   0.                0          0   35.     double row_sum = 0.0;
## 1.872     7291879319   88150571   36.     for (int64_t j=0; j<n; j++)
   0.725      140845059   33595125   37.        row_sum += A[i][j]*b[j];
   0.                0          0   38.     c[i] = row_sum;
                                     39.    }
   0.                0          0   40. }
```

In a smiliar way we can display the event counter values at the instruction level. Again we have modified the lay-out due to page width limitations:

```
$ exp=mxv.hwc.def.2.thr.er
$ gprofng display text -metrics e.hwc -disasm mxv_core $exp
```

```
   Excl. CPU Excl.        Excl.
   Cycles    Instructions Last-Level
    sec.     Executed     Cache Misses
                                           <Function: mxv_core>
   0.                0          0  [33] 4021ba: mov   0x8(%rsp),%r10
                                    34.   for (uint64_t i=...) {
   0.                0          0  [34] 4021bf: cmp   %rsi,%rdi
   0.                0          0  [34] 4021c2: jbe   0x37
   0.                0          0  [34] 4021c4: ret
                                    35.       double row_sum = 0.0;
                                    36.       for (int64_t j=0; j<n; j++)
                                    37.         row_sum += A[i][j]*b[j];
   0.                0          0  [37] 4021c5: mov   (%r8,%rdi,8),%rdx
   0.                0          0  [36] 4021c9: mov   $0x0,%eax
   0.                0          0  [35] 4021ce: pxor  %xmm1,%xmm1
   0.002      12804230     321394  [37] 4021d2: movsd (%rdx,%rax,8),%xmm0
   0.141      60819025    3866677  [37] 4021d7: mulsd (%r9,%rax,8),%xmm0
   0.582      67221804   29407054  [37] 4021dd: addsd %xmm0,%xmm1
## 1.871     7279075109  87989870  [36] 4021e1: add   $0x1,%rax
   0.002      12804210      80351  [36] 4021e5: cmp   %rax,%rcx
   0.                0          0  [36] 4021e8: jne   0xffffffffffffffea
                                    38.       c[i] = row_sum;
   0.                0          0  [38] 4021ea: movsd %xmm1,(%r10,%rdi,8)
   0.                0          0  [34] 4021f0: add   $0x1,%rdi
```

```
0.                0          0  [34] 4021f4: cmp   %rdi,%rsi
0.                0          0  [34] 4021f7: jb    0xd
0.                0          0  [35] 4021f9: pxor  %xmm1,%xmm1
0.                0          0  [36] 4021fd: test  %rcx,%rcx
0.                0      80350  [36] 402200: jne   0xfffffffffffffc5
0.                0          0  [36] 402202: jmp   0xfffffffffffffe8
                                 39.   }
                                 40. }
0.                0          0  [40]  402204:  ret
```

So far we have used the default settings for the event counters. It is quite straightforward to select specific counters. For sake of the example, let's assume we would like to count how many branch instructions and retired memory load instructions that missed in the L1 cache have been executed. We also want to count these events with a high resolution.

This is the command to do so:

```
$ exe=mxv-pthreads
$ m=8000
$ n=4000
$ exp=mxv.hwc.sel.2.thr.er
$ hwc1=br_ins,hi
$ hwc2=mem_load_retired.l1_miss,hi
$ gprofng collect app -O $exp -h $hwc1 -h $hwc2 $exe -m $m -n $n -t 2
```

As before, we get a table with the event counts. Due to the very long name for the second counter, we have somewhat modified the output.

```
$ gprofng display text -limit 10 -functions mxv.hwc.sel.2.thr.er
```

```
Functions sorted by metric: Exclusive Total CPU Time
Excl.     Incl.     Excl. Branch  Excl.                 Name
Total     Total     Instructions  mem_load_retired.l1_miss
CPU sec.  CPU sec.                Events
2.597     2.597     1305305319    4021340               <Total>
2.481     2.481     1233233242    3982327               mxv_core
0.040     0.107       19019012       9003               init_data
0.028     0.052       23023048      15006               erand48_r
0.024     0.024       19019008       9004               __drand48_iterate
0.015     0.067       11011009       2998               drand48
0.008     0.010              0       3002               _int_malloc
0.001     0.001              0          0               brk
0.001     0.002              0          0               sysmalloc
0.        0.001              0          0               __default_morecore
```

When using event counters, the values could be very large and it is not easy to compare the numbers. As we will show next, the `ratio` feature is very useful when comparing such profiles.

To demonstrate this, we have set up another event counter experiment where we would like to compare the number of last level cache miss and the

number of branch instructions executed when using a single thread, or two threads.

These are the commands used to generate the experiment directories:

```
$ exe=./mxv-pthreads
$ m=8000
$ n=4000
$ exp1=mxv.hwc.comp.1.thr.er
$ exp2=mxv.hwc.comp.2.thr.er
$ gprofng collect app -O $exp1 -h llm -h br_ins $exe -m $m -n $n -t 1
$ gprofng collect app -O $exp2 -h llm -h br_ins $exe -m $m -n $n -t 2
```

The following script has been used to get the tables. Due to lay-out restrictions, we have to create two tables, one for each counter.

```
# Limit the output to 5 lines
limit 5
# Define the metrics
metrics name:e.llm
# Set the comparison to ratio
compare ratio
functions
# Define the metrics
metrics name:e.br_ins
# Set the comparison to ratio
compare ratio
functions
```

Note that we print the name of the function first, followed by the counter data. The new element is that we set the comparison mode to `ratio`. This divides the data in a column by its counterpart in the reference experiment.

This is the command using this script and the two experiment directories as input:

```
$ gprofng display text -script my-script-comp-counters \
  mxv.hwc.comp.1.thr.er \
  mxv.hwc.comp.2.thr.er
```

By design, we get two tables, one for each counter:

```
Functions sorted by metric: Exclusive Last-Level Cache Misses

                              mxv.hwc.comp.1.thr.er  mxv.hwc.comp.2.thr.er
Name                          Excl. Last-Level       Excl. Last-Level
                              Cache Misses           Cache Misses
                                                         ratio
 <Total>                      122709276              x   0.788
```

```
 mxv_core                    121796001            x   0.787
 init_data                      723064            x   1.055
 erand48_r                      100111            x   0.500
 drand48                         60065            x   1.167

Functions sorted by metric: Exclusive Branch Instructions

                             mxv.hwc.comp.1.thr.er  mxv.hwc.comp.2.thr.er
Name                         Excl. Branch           Excl. Branch
                             Instructions           Instructions
                                                      ratio
 <Total>                     1307307316             x 0.997
 mxv_core                    1235235239             x 0.997
 erand48_r                     23023033             x 0.957
 drand48                       20020009             x 0.600
 __drand48_iterate             17017028             x 0.882
```

A ratio less than one in the second column, means that this counter value was smaller than the value from the reference experiment shown in the first column.

This kind of presentation of the results makes it much easier to quickly interpret the data.

We conclude this section with thread-level event counter overviews, but before we go into this, there is an important metric we need to mention.

In case it is known how many instructions and CPU cycles have been executed, the value for the IPC ("Instructions Per Clockcycle") can be computed. See Section 7.8 [Hardware Event Counters Explained], page 63. This is a derived metric that gives an indication how well the processor is utilized. The inverse of the IPC is called CPI.

The `gprofng display text` command automatically computes the IPC and CPI values if an experiment contains the event counter values for the instructions and CPU cycles executed. These are part of the metric list and can be displayed, just like any other metric.

This can be verified through the `metric_list` command. If we go back to our earlier experiment with the default event counters, we get the following result.

```
$ gprofng display text -metric_list mxv.hwc.def.2.thr.er
```

```
Current metrics: e.totalcpu:i.totalcpu:e.cycles:e+insts:e+llm:name
Current Sort Metric: Exclusive Total CPU Time ( e.totalcpu )
Available metrics:
         Exclusive Total CPU Time: e.%totalcpu
         Inclusive Total CPU Time: i.%totalcpu
             Exclusive CPU Cycles: e.+%cycles
             Inclusive CPU Cycles: i.+%cycles
  Exclusive Instructions Executed: e+%insts
  Inclusive Instructions Executed: i+%insts
```

```
Exclusive Last-Level Cache Misses: e+%llm
Inclusive Last-Level Cache Misses: i+%llm
 Exclusive Instructions Per Cycle: e+IPC
 Inclusive Instructions Per Cycle: i+IPC
 Exclusive Cycles Per Instruction: e+CPI
 Inclusive Cycles Per Instruction: i+CPI
                             Size: size
                       PC Address: address
                             Name: name
```

Among the other metrics, we see the new metrics for the IPC and CPI listed.

In the script below, we use this information and add the IPC and CPI to the metrics to be displayed. We also use a the thread filter to display these values for the individual threads.

This is the complete script we have used. Other than a different selection of the metrics, there are no new features.

```
# Define the metrics
metrics e.insts:e.%cycles:e.IPC:e.CPI
# Sort with respect to cycles
sort e.cycles
# Limit the output to 5 lines
limit 5
# Get the function overview for all threads
functions
# Get the function overview for thread 1
thread_select 1
functions
# Get the function overview for thread 2
thread_select 2
functions
# Get the function overview for thread 3
thread_select 3
functions
```

In the metrics definition on the second line, we explicitly request the counter values for the instructions (`e.insts`) and CPU cycles (`e.cycles`) executed. These names can be found in output from the `metric_list` command above. In addition to these metrics, we also request the IPC and CPI to be shown.

As before, we used the `limit` command to control the number of functions displayed. We then request an overview for all the threads, followed by three sets of two commands to select a thread and display the function overview.

The script above is used as follows:

```
$ gprofng display text -script my-script-ipc mxv.hwc.def.2.thr.er
```

This script produces four tables. We list them separately below, and have left out the additional output.

The first table shows the accumulated values across the three threads that have been active.

```
Functions sorted by metric: Exclusive CPU Cycles

Excl.         Excl. CPU     Excl.  Excl.   Name
Instructions  Cycles        IPC    CPI
Executed       sec.      %
7906475309    2.691 100.00  1.473  0.679   <Total>
7432724378    2.598  96.54  1.434  0.697   mxv_core
 188860269    0.035   1.31  2.682  0.373   erand48_r
  73623396    0.026   0.95  1.438  0.696   init_data
  76824434    0.018   0.66  2.182  0.458   drand48
```

This shows that IPC of this program is completely dominated by function `mxv_core`. It has a fairly low IPC value of 1.43.

The next table is for thread 1 and shows the values for the main thread.

```
Exp Sel Total
=== === =====
  1 1       3
Functions sorted by metric: Exclusive CPU Cycles

Excl.         Excl. CPU     Excl.  Excl.   Name
Instructions  Cycles        IPC    CPI
Executed       sec.      %
473750931     0.093 100.00  2.552  0.392   <Total>
188860269     0.035  37.93  2.682  0.373   erand48_r
 73623396     0.026  27.59  1.438  0.696   init_data
 76824434     0.018  18.97  2.182  0.458   drand48
134442832     0.013  13.79  5.250  0.190   __drand48_iterate
```

Although this thread hardly uses any CPU cycles, the overall IPC of 2.55 is not all that bad.

Last, we show the tables for threads 2 and 3:

```
Exp Sel Total
=== === =====
  1 2       3
Functions sorted by metric: Exclusive CPU Cycles

Excl.         Excl. CPU     Excl.  Excl.   Name
Instructions  Cycles        IPC    CPI
Executed       sec.      %
3716362189    1.298 100.00  1.435  0.697   <Total>
3716362189    1.298 100.00  1.435  0.697   mxv_core
         0    0.      0.    0.     0.      collector_root
         0    0.      0.    0.     0.      driver_mxv
```

```
Exp Sel Total
=== === =====
  1 3       3
Functions sorted by metric: Exclusive CPU Cycles

Excl.          Excl. CPU     Excl.  Excl.   Name
Instructions   Cycles        IPC    CPI
Executed        sec.      %
3716362189     1.300 100.00  1.433  0.698   <Total>
3716362189     1.300 100.00  1.433  0.698   mxv_core
         0     0.      0.    0.     0.      collector_root
         0     0.      0.    0.     0.      driver_mxv
```

It is seen that both execute the same number of instructions and take about the same number of CPU cycles. As a result, the IPC is the same for both threads.

3.5 Java Profiling

The `gprofng collect app` command supports Java profiling. The `-j on` option can be used for this, but since this feature is enabled by default, there is no need to set this explicitly. Java profiling may be disabled through the `-j off` option.

The program is compiled as usual and the experiment directory is created similar to what we have seen before. The only difference with a C/C++ application is that the program has to be explicitly executed by java.

For example, this is how to generate the experiment data for a Java program that has the source code stored in file `Pi.java`:

```
$ javac Pi.java
$ gprofng collect app -j on -O pi.demo.er java Pi < pi.in
```

Regarding which java is selected to generate the data, `gprofng` first looks for the JDK in the path set in either the `JDK_HOME` environment variable, or in the `JAVA_PATH` environment variable. If neither of these variables is set, it checks for a JDK in the search path (set in the PATH environment variable). If there is no JDK in this path, it checks for the java executable in `/usr/java/bin/java`.

In case additional options need to be passed on to the JVM, the `-J <string>` option can be used. The string with the option(s) has to be delimited by quotation marks in case there is more than one argument.

The `gprofng display text` command may be used to view the performance data. There is no need for any special options and the same commands as previously discussed are supported.

The `viewmode` command See Section 7.4 [The Viewmode], page 60, is very useful to examine the call stacks.

For example, this is how one can see the native call stacks. For lay-out purposes we have restricted the list to the first five entries:

```
$ gprofng display text -limit 5 -viewmode machine -calltree pi.demo.er
```

```
Print limit set to 5
Viewmode set to machine
Functions Call Tree. Metric: Attributed Total CPU Time

Attr.      Name
Total
CPU sec.
1.381      +-<Total>
1.171        +-Pi.calculatePi(double)
0.110        +-collector_root
0.110        |  +-JavaMain
0.070        |    +-jni_CallStaticVoidMethod
```

Note that the selection of the viewmode is echoed in the output.

4 The gprofng Tools

Several tools are included in gprofng. In subsequent chapters these are discussed in detail. Below a brief description is given, followed by an overview of the environment variables that are supported.

4.1 Tools Overview

The following tools are supported by gprofng:

`gprofng collect app`
: Collects the performance data and stores the results in an experiment directory. There are many options on this tool, but quite often the defaults are sufficient. An experiment directory is required for the subsequent analysis of the results.

`gprofng display text`
: Generates performance reports in ASCII format. Commandline options, and/or commands in a script file are used to control the contents and lay-out of the generated report(s).

`gprofng display html`
: Takes one or more experiment directories and generates a directory with HTML files. Starting from the index.html file, the performance data may be examined in a browser.

`gprofng display src`
: Displays the source code, interleaved with the disassembled instructions.

`gprofng archive`
: Archives an experiment directory by (optionally) including source code and object files, as well as the shared libraries that have been used.

4.2 The gprofng.rc file with default settings

The `gprofng.rc` file is used to define default settings for the `gprofng display text` and `gprofng display src` tools, but the user can override these defaults through local configuration files.

There are three files that are checked when the tool starts up. The first file has pre-defined settings and comes with the installation, but through a hidden file called `.gprofng.rc`, the user can (re)define the defaults:

These are the locations and files that are checked upon starting the above mentioned tools:

1. The system-wide filename is called `gprofng.rc` and is located in the top level `/etc` directory.

   If gprofng has been built from the source, this file is in subdirectory `etc` in the top level installation directory.

2. The user's home directory may have a hidden file called `.gprofng.rc`.
3. The directory where `gprofng display text` (or `gprofng display src`) is invoked from may have a hidden file called `.gprofng.rc`.

The settings of each file override the settings of the file(s) read before it. Defaults in the system-wide file are overruled by the file in the user home directory (if any) and any settings in the `.gprofng.rc` file in the current directory override those.

Note that the settings in these files only affect the defaults. Unlike the commands used in a script file, they are not commands for the tools.

The `.gprofng.rc` configuration files can contain the `addpath`, `compare`, `dthresh`, `name`, `pathmap`, `printmode`, `sthresh`, and `viewmode` commands as described in this user guide.

They can also contain the following commands, *which cannot be used on either the command line, or in a script file*:

`dmetrics` *`metric-spec`*
: Specify the default metrics to be displayed or printed in the function list. The syntax and use of the metric list is described in section Section 7.3 [Metric Definitions], page 59. The order of the metric keywords in the list determines the order in which the metrics are presented.

  Default metrics for the `callers-callees` list are derived from the function list default metrics by adding the corresponding attributed metric before the first occurrence of each metric name in the list.

`dsort` *`metric-spec`*
: Specify the default metric by which the function list is sorted. The sort metric is the first metric in this list that matches a metric in any loaded experiment, subject to the following conditions:

  - If the entry in *metric-spec* has a visibility string of an exclamation point ('`!`'), the first metric whose name matches is used, regardless of whether it is visible.
  - If the entry in *metric-spec* has any other visibility string, the first visible metric whose name matches is used.

  The syntax and use of the metric list is described in section Section 7.3 [Metric Definitions], page 59. The default sort metric for the `callers-callees` list is the attributed metric corresponding to the default sort metric for the function list.

`en_desc {on | off | =`*`regex`*`}`
: Set the mode for reading descendant experiments to '`on`' (enable all descendants) or '`off`' to disable all descendants. If '`=`'*regex* is used, enable data from those experiments whose executable name matches the regular expression.

The default setting is '`on`' to follow all descendants. In reading experiments with descendants, any sub-experiments that contain little or no performance data are ignored by `gprofng display text`.

4.3 Filters

Various filter commands are supported by `gprofng display text`. Thanks to the use of filters, the user can zoom in on a certain area of interest. With filters, it is possible to select one or more threads to focus on, define a window in time, select specific call stacks, etc.

While already powerful by themselves, filters may be combined to further narrow down the view into the data.

It is important to note that filters are *persistent*. A filter is active until it is reset. This means that successive filter commands increasingly narrow down the view until one or more are reset.

An example is the following:

```
$ gprofng display text -thread_select 1 -functions \
                       -cpu_select 2 -functions ...
```

This command selects thread 1 and requests the function view for this thread. The third (`cpu_select 2`) command *adds* the constraint that only the events on CPU 2 are to be selected. This means that the next function view selects events that were executed by thread 1 and have been running on CPU 2.

In contrast with this single command line, the two commands below look similar, but behave very differently:

```
$ gprofng display text -thread_select 1 -functions ...
$ gprofng display text -cpu_select 2 -functions ...
```

The first command displays the function view for thread 1. The second command shows the function view for CPU 2 for *all* threads that have been running on this CPU.

As the following example demonstrates, things get a little more tricky in case a script file is used. Consider the following script file:

```
thread_select 1
functions
cpu_select 2
functions
```

This script file displays the function view for thread 1 first. This is followed by those functions that were executed by thread 1 *and* have been run on CPU 2.

If however, the script should behave like the two command line invocations shown above, the thread selection filter needs to be reset before CPU 2 is selected:

```
thread_select 1
functions
# Reset the thread selection filter:
thread_select all
cpu_select 2
functions
```

In general, filters behave differently than commands or options. In particular there may be an interaction between different filter definitions.

For example, as explained above, in the first script file the `thread_select` and `cpu_select` commands interact.

For a list of all the predefined filters see [Predefined Filters], page 55.

4.4 Supported Environment Variables

Various environment variables are supported. We refer to the man page for gprofng(1) for an overview and description (See Section A.1 [Man page for gprofng], page 67).

5 Performance Data Collection

The `gprofng collect app` command is used to gather the application performance data while the application executes. At regular intervals, program execution is halted and the required data is recorded. An experiment directory is created when the tool starts. This directory is used to store the relevant information and forms the basis for a subsequent analysis with one of the viewing tools.

5.1 The `gprofng collect app` command

This is the command to collect the performance information for the target application. The usage is as follows:

```
$ gprofng collect app [OPTION(S)] TARGET [TARGET_ARGUMENTS]
```

Options to the command are passed in first. This is followed by the name of the target, which is typically a binary executable or a script, followed by any options that may be required by the target.

6 View the Performance Information

Various tools to view the performance data stored in one or more experiment directories are available. In this chapter, these will all be covered in detail.

6.1 The `gprofng display text` Tool

This tool displays the performance information in ASCII format. It supports a variety of views into the data recorded. These views can be specified in two ways and both may be used simultaneously:

- Command line options start with a dash ('`-`') symbol and may take an argument.
- Options may also be included in a file, the "script file". In this case, the dash symbol should *not* be included. Multiple script files can be used on the same command line.

While they may appear as an option, they are really commands and this is why they will be referred to as *commands* in the documentation.

As a general rule, *the order of options matters* and if the same option, or command, occurs multiple times, the rightmost setting is selected.

6.1.1 The `gprofng display text` Commands

The most commonly used commands are documented in the man page for this tool (See Section A.3 [gprofng display text], page 74). In this section we list and describe all other commands that are supported.

Commands that List Experiment Details

`experiment_ids`
: For each experiment that has been loaded, show the totals of the metrics recorded, plus some other operational characteristics like the name of the executable, PID, etc. The top line contains the accumulated totals for the metrics.

`experiment_list`
: Display the list of experiments that are loaded. Each experiment is listed with an index, which is used when selecting samples, threads, or LWPs, and a process id (PID), which can be used for advanced filtering.

`cpu_list`
: Display the total number of CPUs that have been used during the experiment(s).

`cpus`
: Show a list of CPUs that were used by the application, along with the metrics that have been recorded. The CPUs are rep-

resented by a CPU number and show the Total CPU time by default.

Note that since the data is sorted with respect to the default metric, it may be useful to use the `sort name` command to show the list sorted with respect to the CPU id.

`GCEvents`

This commands is for Java applications only. It shows any Garbage Collection (GC) events that have occurred while the application was executing..

`lwp_list`

Displays the list of LWPs processed during the experiment(s).

`processes`

For each experiment that has been loaded, this command displays a list of processes that were created by the application, along with their metrics. The processes are represented by process ID (PID) numbers and show the Total CPU time metric by default. If additional metrics are recorded in an experiment, these are shown as well.

`samples`

Display a list of sample points and their metrics, which reflect the microstates recorded at each sample point in the loaded experiment. The samples are represented by sample numbers and show the Total CPU time by default. Other metrics might also be displayed if enabled.

`sample_list`

For each experiment loaded, display the list of samples currently selected.

`seconds`

Show each second of the profiling run that was captured in the experiment, along with the metrics collected in that second. The seconds view differs from the samples view in that it shows periodic samples that occur every second beginning at 0 and the interval cannot be changed.

The seconds view lists the seconds of execution with the Total CPU time by default. Other metrics might also be displayed if the metrics are present in the loaded experiments.

`threads`

Show a list of threads and their metrics. The threads are represented by a process and thread pair and show the Total CPU time by default. Other metrics might also be displayed by default if the metrics are present in the loaded experiment.

`thread_list`
: Display the list of threads currently selected for the analysis.

The commands below are for use in scripts and interactive mode only. They are not allowed on the command line.

`add_exp` *exp-name*
: Add the named experiment to the current session.

`drop_exp` *exp-name*
: Drop the named experiment from the current session.

`open_exp` *exp-name*
: Drop all loaded experiments from the session, and then load the named experiment.

Commands that Affect Listings and Output

`dthresh` *value*
: Specify the threshold percentage for highlighting metrics in the annotated disassembly code. If the value of any metric is equal to or greater than *value* as a percentage of the maximum value of that metric for any instruction line in the file, the line on which the metrics occur has a '`##`' marker inserted at the beginning of the line. The default is 75.

`printmode {text | html |` *single-char*`}`
: Set the print mode. If the keyword is `text`, printing will be done in tabular form using plain text. In case the `html` keyword is selected, the output is formatted as an HTML table.

  Alternatively, *single-char* may be used in a delimiter separated list, with the single character *single-char* as the delimiter.

  The printmode setting is used only for those commands that generate tables, such as `functions`. The setting is ignored for other printing commands, including those showing source and disassembly listings.

`sthresh` *value*
: Specify the threshold percentage for highlighting metrics in the annotated source code. If the value of any metric is equal to or greater than *value* (as a percentage) of the maximum value of that metric for any source line in the file, the line on which the metrics occur has a '`##`' marker inserted at the beginning of the line. The default is 75.

Predefined Filters

The filters below use a list, the selection list, to define a sequence of numbers. See Section 7.5 [The Selection List], page 60. Note that this selection is persistent, but the filter can be reset by using '`all`' as the *selection-list*.

`cpu_select` *`selection-list`*
: Select the CPU ids specified in the *selection-list*.

`lwp_select` *`selection-list`*
: Select the LWPs specified in the *selection-list*.

`sample_select` *`selection-list`*
`thread_select` *`selection-list`*
: Select a series of threads, or just one, to be used in subsequent views. The *selection-list* consists of a sequence of comma separated numbers. This may include a range of the form '`n-m`'.

Commands to Set and Change Search Paths

`addpath` *`path-list`*
: Append *path-list* to the current setpath settings. Note that multiple `addpath` commands can be used in `.gprofng.rc` files, and will be concatenated.

`pathmap` *`old-prefix new-prefix`*
: If a file cannot be found using the path list set by `addpath`, or the `setpath` command, one or more path remappings may be set with the `pathmap` command.

  With path mapping, the user can specify how to replace the leading component in a full path by a different string.

  With this command, any path name for a source file, object file, or shared object that begins with the prefix specified with *old-prefix*, the old prefix is replaced by the prefix specified with *new-prefix*. The resulting path is used to find the file.

  For example, if a source file located in directory `/tmp` is shown in the `gprofng display text` output, but should instead be taken from `/home/demo`, the following `pathmap` command redefines the path:

  ```
  $ gprofng diplay text -pathmap /tmp /home/demo -source ...
  ```

  Note that multiple `pathmap` commands can be supplied, and each is tried until the file is found.

`setpath` *`path-list`*
: Set the path used to find source and object files. The path is defined through the *path-list* keyword. It is a colon separated list of directories, jar files, or zip files. If any directory has a colon character in it, escape it with a backslash ('\').

  The special directory name `$expts`, refers to the set of current experiments in the order in which they were loaded. You can abbreviate it with a single '`$`' character.

  The default path is '`$expts:.`.' which is the directories of the loaded experiments and the current working directory.

Use `setpath` with no argument to display the current path.

Note that `setpath` commands *are not allowed .gprofng.rc configuration files.*

7 Terminology

Throughout this manual, certain terminology specific to profiling tools, or `gprofng`, or even to this document only, is used. In this chapter this terminology is explained in detail.

7.1 The Program Counter

The *Program Counter*, or PC for short, keeps track where program execution is. The address of the next instruction to be executed is stored in a special purpose register in the processor, or core.

The PC is sometimes also referred to as the *instruction pointer*, but we will use Program Counter or PC throughout this document.

7.2 Inclusive and Exclusive Metrics

In the remainder, these two concepts occur quite often and for lack of a better place, they are explained here.

The *inclusive* value for a metric includes all values that are part of the dynamic extent of the target function. For example if function `A` calls functions `B` and `C`, the inclusive CPU time for `A` includes the CPU time spent in `B` and `C`.

In contrast with this, the *exclusive* value for a metric is computed by excluding the metric values used by other functions called. In our imaginary example, the exclusive CPU time for function `A` is the time spent outside calling functions `B` and `C`.

In case of a *leaf function*, the inclusive and exclusive values for the metric are the same since by definition, it is not calling any other function(s).

Why do we use these two different values? The inclusive metric shows the most expensive path, in terms of this metric, in the application. For example, if the metric is cache misses, the function with the highest inclusive metric tells you where most of the cache misses come from.

Within this branch of the application, the exclusive metric points to the functions that contribute and help to identify which part(s) to consider for further analysis.

7.3 Metric Definitions

The metrics displayed in the various views are highly customizable. In this section it is explained how to construct the metrics definition(s).

The `metrics` command takes a colon (‘`:`’) separated list, where each item in the list consists of the following three fields: <*flavor*><*visibility*><*metric-name*>.

The <*flavor*> field is either '`e`' for "exclusive", and/or '`i`' for "inclusive". The <*metric-name*> field is the name of the metric and the <*visibility*> field consists of one ore more characters from the following table:

`.` Show the metric as time. This applies to timing metrics and hardware event counters that measure cycles. Interpret as '`+`' for other metrics.

`%` Show the metric as a percentage of the total value for this metric.

`+` Show the metric as an absolute value. For hardware event counters this is the event count. Interpret as '`.`' for timing metrics.

`!` Do not show any metric value. Cannot be used with other visibility characters. This visibility is meant to be used in a `dmetrics` command to set default metrics that override the built-in visibility defaults for each type of metric.

Both the <*flavor*> and <*visibility*> strings may have more than one character. If both strings have more than one character, the <*flavor*> string is expanded first. For example, `ie.%user` is first expanded to `i.%user:e.%user`, which is then expanded into `i.user:i%user:e.user:e%user`.

7.4 The Viewmode

There are different ways to view a call stack in Java. In `gprofng`, this is called the *viewmode* and the setting is controlled through a command with the same name.

The `viewmode` command takes one of the following keywords:

`user` This is the default and shows the Java call stacks for Java threads. No call stacks for any housekeeping threads are shown. The function list contains a function `<JVM-System>` that represents the aggregated time from non-Java threads. When the JVM software does not report a Java call stack, time is reported against the function `<no Java callstack recorded>`.

`expert` Show the Java call stacks for Java threads when the Java code from the user is executed and machine call stacks when JVM code is executed, or when the JVM software does not report a Java call stack. Show the machine call stacks for housekeeping threads.

`machine` Show the actual native call stacks for all threads.

7.5 The Selection List

Several commands allow the user to specify a sequence of numbers called the *selection list*. Such a list may for example be used to select specific threads from all the threads that have been used when conducting the experiment(s).

A selection list (or "list" in the remainder of this section) can be a single number, a contiguous range of numbers with the start and end numbers separated by a hyphen ('-'), a comma-separated list of numbers and ranges, or the `all` keyword that resets the filter. Lists must not contain spaces.

Each list can optionally be preceded by an experiment list with a similar format, separated from the list by a colon (:). If no experiment list is included, the list applies to all experiments.

Multiple lists can be concatenated by separating the individual lists by a plus sign.

These are some examples of various filters using a list:

`thread_select 1`
: Select thread 1 from all experiments.

`thread_select all:1`
: Select thread 1 from all experiments.

`thread_select 1:all`
: Select all the threads from the first experiment loaded.

`thread_select 1:2+3:4`
: Select thread 2 from experiment 1 and thread 4 from experiment 3.

`cpu_select all:1,3,5`
: Selects cores 1, 3, and 5 from all experiments.

`cpu_select 1,2:all`
: Select all cores from experiments 1 and 2.

Recall that there are several list commands that show the mapping between the numbers and the targets.

For example, the `experiment_list` command shows the name(s) of the experiment(s) loaded and the associated number. In this example it is used to get this information for a range of experiments:

```
$ gprofng display text -experiment_list mxv.?.thr.er
```

This is the output, showing for each experiment the ID, the PID, and the name:

```
ID Sel     PID Experiment
== === ======= ============
 1 yes 2750071 mxv.1.thr.er
 2 yes 1339450 mxv.2.thr.er
 3 yes 3579561 mxv.4.thr.er
```

7.6 Load Objects and Functions

An application consists of various components. The source code files are compiled into object files. These are then glued together at link time to form the executable. During execution, the program may also dynamically load objects.

A *load object* is defined to be an executable, or shared object. A shared library is an example of a load object in `gprofng`.

Each load object, contains a text section with the instructions generated by the compiler, a data section for data, and various symbol tables. All load objects must contain an ELF symbol table, which gives the names and addresses of all the globally known functions in that object.

Load objects compiled with the -g option contain additional symbolic information that can augment the ELF symbol table and provide information about functions that are not global, additional information about object modules from which the functions came, and line number information relating addresses to source lines.

The term *function* is used to describe a set of instructions that represent a high-level operation described in the source code. The term also covers methods as used in C++ and in the Java programming language.

In the `gprofng` context, functions are provided in source code format. Normally their names appear in the symbol table representing a set of addresses. If the Program Counter (PC) is within that set, the program is executing within that function.

In principle, any address within the text segment of a load object can be mapped to a function. Exactly the same mapping is used for the leaf PC and all the other PCs on the call stack.

Most of the functions correspond directly to the source model of the program, but there are exceptions. This topic is however outside of the scope of this guide.

7.7 The Concept of a CPU in gprofng

In gprofng, there is the concept of a CPU. Admittedly, this is not the best word to describe what is meant here and may be replaced in the future.

The word CPU is used in many of the displays. In the context of gprofng, it is meant to denote a part of the processor that is capable of executing instructions and with its own state, like the program counter.

For example, on a contemporary processor, a CPU could be a core. In case hardware threads are supported within a core, a CPU is one of those hardware threads.

To see which CPUs have been used in the experiment, use the `cpu` command in `gprofng display text`.

7.8 Hardware Event Counters Explained

For quite a number of years now, many microprocessors have supported hardware event counters.

On the hardware side, this means that in the processor there are one or more registers dedicated to count certain activities, or "events". Examples of such events are the number of instructions executed, or the number of cache misses at level 2 in the memory hierarchy.

While there is a limited set of such registers, the user can map events onto them. In case more than one register is available, this allows for the simultaenous measurement of various events.

A simple, yet powerful, example is to simultaneously count the number of CPU cycles and the number of instructions excuted. These two numbers can then be used to compute the *IPC* value. IPC stands for "Instructions Per Clockcycle" and each processor has a maximum. For example, if this maximum number is 2, it means the processor is capable of executing two instructions every clock cycle.

Whether this is actually achieved, depends on several factors, including the instruction characteristics. However, in case the IPC value is well below this maximum in a time critical part of the application and this cannot be easily explained, further investigation is probably warranted.

A related metric is called *CPI*, or "Clockcycles Per Instruction". It is the inverse of the CPI and can be compared against the theoretical value(s) of the target instruction(s). A significant difference may point at a bottleneck.

One thing to keep in mind is that the value returned by a counter can either be the number of times the event occured, or a CPU cycle count. In case of the latter it is possible to convert this number to time.

This is often easier to interpret than a simple count, but there is one caveat to keep in mind. The CPU frequency may not have been constant while the experimen was recorded and this impacts the time reported.

These event counters, or "counters" for short, provide great insight into what happens deep inside the processor. In case higher level information does not provide the insight needed, the counters provide the information to get to the bottom of a performance problem.

There are some things to consider though.

- The event definitions and names vary across processors and it may even happen that some events change with an update. Unfortunately and this is luckily rare, there are sometimes bugs causing the wrong count to be returned.

  In `gprofng`, some of the processor specific event names have an alias name. For example `insts` measures the instructions executed. These aliases not only makes it easier to identify the functionality, but also provide portability of certain events across processors.

- Another complexity is that there are typically many events one can monitor. There may up to hundreds of events available and it could require several experiments to zoom in on the root cause of a performance problem.
- There may be restrictions regarding the mapping of event(s) onto the counters. For example, certain events may be restricted to specific counters only. As a result, one may have to conduct additional experiments to cover all the events of interest.
- The names of the events may also not be easy to interpret. In such cases, the description can be found in the architecture manual for the processor.

Despite these drawbacks, hardware event counters are extremely useful and may even turn out to be indispensable.

7.9 What is <apath>?

In most cases, `gprofng` shows the absolute pathnames of directories. These tend to be rather long, causing display issues in this document.

Instead of wrapping these long pathnames over multiple lines, we decided to represent them by the `<apath>` symbol, which stands for "an absolute pathname".

Note that different occurrences of `<apath>` may represent different absolute pathnames.

8 Other Document Formats

This chapter is applicable when building gprofng from the binutils source.

This document is written in Texinfo and the source text is made available as part of the binutils distribution. The file name is `gprofng.texi` and can be found in subdirectory `gprofng/doc` of the top level binutils directory.

The default installation procedure creates a file in the `info` format and stores it in the documentation section of binutils. This source file can however also be used to generate the document in the `html` and `pdf` formats. These may be easier to read and search.

To generate this documentation file in a different format, go to the directory that was used to build the tools. The make file to build the other formats is in the `gprofng/doc` subdirectory.

For example, if you have set the build directory to be <*my-build-dir*>, go to subdirectory <*my-build-dir*>*/gprofng/doc*.

This subdirectory has a single filed called `Makefile` that can be used to build the documentation in various formats. We recommend to use these commands.

There are four commands to generate the documentation in the `html` or `pdf` format. It is assumed that you are in directory `gprofng/doc` under the main directory <*my-build-dir*>.

`make html` Create the html file in the current directory.

`make pdf` Create the pdf file in the current directory.

`make install-html`
: Create and install the html file in the binutils documentation directory.

`make install-pdf`
: Creat and install the pdf file in the binutils documentation directory.

For example, to install this document in the binutils documentation directory, the commands below may be executed. In this notation, <*format*> is one of `html`, or `pdf`:

```
$ cd <my-build-dir>/gprofng/doc
$ make install-<format>
```

The binutils installation directory is either the default `/usr/local` or the one that has been set with the `--prefix` option as part of the `configure` command. In this example we symbolize this location with `<install>`.

The documentation directory is `<install>/share/doc/gprofng` in case `html` or `pdf` is selected and `<install>/share/info` for the file in the `info` format.

Some things to note:

- For the `pdf` file to be generated, the `texi2dvi` tool is required. It is for example available as part of the `texinfo-tex` package.
- Instead of generating a single file in the `html` format, it is also possible to create a directory with individual files for the various chapters. To do so, remove the use of `--no-split` in variable `MAKEINFOHTML` in the make file in the `<my-build-dir/gprofng/doc` directory.

Appendix A The gprofng Man Pages

In this appendix the man pages for the various gprofng tools are listed.

A.1 Man page for `gprofng`

NAME

gprofng - The driver for the gprofng application profiling tool

SYNOPSIS

`gprofng` [*option(s)*] *action* [*qualifier*] [*option(s)*] *target* [*options*]

DESCRIPTION

This is the driver for the gprofng tools suite to gather and analyze performance data.

The driver executes the *action* specified. An example of an action is '`collect`' to collect performance data. Depending on the action, a *qualifier* may be needed to further define the command. The last item is the *target* that the command applies to.

There are three places where options are supported. The driver supports options. These can be found below. The *action*, possibly in combination with the *qualifier* also supports options. A description of these can be found in the man page for the command. Any options needed to execute the target command should follow the target name.

For example, to collect performance data for an application called `a.out` and store the results in experiment directory '`mydata.er`', the following command may be used:

```
$ gprofng collect app -o mydata.er a.out -t 2
```

In this example, the action is '`collect`', the qualifier is '`app`', the single argument to the command is `-o mydata.er` and the target is `a.out`. The target command is invoked with the '`-t 2`' option.

If gprofng is executed without any additional option, action, or target, a usage overview is printed.

OPTIONS

`--version`
: Print the version number and exit.

`--help`
: Print usage information and exit.

ENVIRONMENT

The following environment variables are supported:

'GPROFNG_MAX_CALL_STACK_DEPTH'
Set the depth of the call stack (default is 256).

'GPROFNG_USE_JAVA_OPTIONS'
May be set when profiling a C/C++ application that uses dlopen() to execute Java code.

'GPROFNG_ALLOW_CORE_DUMP'
Set this variable to allow a core file to be generated; otherwise an error report is created on /tmp.

'GPROFNG_ARCHIVE'
Use this variable to define the settings for automatic archiving upon experiment recording completion.

'GPROFNG_ARCHIVE_COMMON_DIR'
Set this variable to the location of the common archive.

'GPROFNG_JAVA_MAX_CALL_STACK_DEPTH'
Set the depth of the Java call stack; the default is 256; set to 0 to disable capturing of call stacks.

'GPROFNG_JAVA_NATIVE_MAX_CALL_STACK_DEPTH'
Set the depth of the Java native call stack; the default is 256; set to 0 to disable capturing of call stacks (JNI and assembly call stacks are not captured).

NOTES

The gprofng driver supports the following commands.

Collect performance data:

`gprofng collect app`
Collect application performance data.

Display the performance results:

`gprofng display text`
Display the performance data in ASCII format.

`gprofng display html`
Generate an HTML file from one or more experiments.

Miscellaneous commands:

`gprofng display src`
Display source or disassembly with compiler annotations.

`gprofng archive`
Include binaries and source code in an experiment directory.

It is also possible to invoke the lower level commands directly, but since these are subject to change, in particular the options, we recommend to use the driver.

SEEALSO

gp-archive(1), gp-collect-app(1), gp-display-html(1), gp-display-src(1), gp-display-text(1)

Each gprofng command also supports the `--help` option. This lists the options and a short description for each option.

For example this displays the options supported on the `gprofng collect app` command:

```
$ gprofng collect app --help
```

The user guide for gprofng is maintained as a Texinfo manual. If the `info` and `gprofng` programs are correctly installed, the command `info gprofng` should give access to this document.

COPYRIGHT

A.2 Man page for `gprofng collect app`

NAME

gprofng collect app - Collect performance data for the target program

SYNOPSIS

`gprofng collect app` [*option(s)*] *target* [*option(s)*]

DESCRIPTION

Collect performance data on the target program. In addition to Program Counter (PC) sampling, hardware event counters and various tracing options are supported.

For example, this command collects performance data for an executable called '`a.out`' and stores the data collected in an experiment directory with the name '`example.er`'.

```
$ gprofng collect app -o example.er ./a.out
```

OPTIONS

`--version`
: Print the version number and exit.

`--help`
: Print usage information and exit.

`-p {off|on|lo|hi|<value>}`
: Disable (off) or enable (on) clock-profiling using a default sampling granularity, or enable clock-profiling implicitly by setting the sampling granularity (lo, hi, or a specific value in ms). By default, clock profiling is enabled ('`-p on`').

`-h {<ctr_def>...,<ctr_n_def>}`
: Enable hardware event counter profiling and select the counter(s). To see the supported counters on this system, use the '`-h`' option without other arguments.

`-o <exp_name>`
: Specify the name for the experiment directory. The name has to end with '`.er`' and may contain an absolute path (e.g. `/tmp/experiment.er`).

`-O <exp_name>`
: This is the same as the '`-o`' option, but unlike this option, silently overwrites an existing experiment directory with the same name.

`-C <comment_string>`
: Add up to 10 comment strings to the experiment. These comments appear in the notes section of the header and can be retrieved with the `gprofng display text` command using the '`-header`' option.

`-j {on|off|<path>}`
: Controls Java profiling when the target is a JVM machine. The allowed values of this option are: enable (on), disable (off) Java profiling when the target program is a JVM, or set '`<path>`' to a non-default JVM. The default is '`-j on`'

  `on` Record profiling data for the JVM machine, and recognize methods compiled by the Java HotSpot virtual machine. Also record Java call stacks. The default is '`-j on`'.

  `off` Does not record Java profiling data. Profiling data for native call stacks is still recorded.

  `<path>` Records profiling data for the JVM, and use the JVM as installed in <*path*>.

`-J <jvm-options>`
: Specifies additional options to be passed to the JVM used. The *jvm-options* list must be enclosed in quotation marks if it contains more than one option. The items in the list need to be separated by spaces or tab. Each item is passed as a separate option to the JVM. Note that this option implies '`-j on`'.

`-t <duration>[m|s]`
: Collects data for the specified duration. The duration can be a single number, optionally followed by either '`m`' to specify minutes, or '`s`' to specify seconds, which is the default.

  The duration can also two numbers separated by minus (-) sign. If a single number is given, data is collected from the start of the run until the given time. If two numbers are given, data is collected from the first time to the second. If the second time is zero, data is collected until the end of the run. If two non-zero numbers are given, the first must be less than the second.

`-n`
: This is used for a dry run. Several run-time settings are displayed, but the target is not executed and no performance data is collected.

`-F {off|on|=regex}`
: Control whether descendant processes should have their data recorded. To disable/enable this feature, use

'`off`'/'`on`'. Use '`=`'*regex* to record data on those processes whose executable name matches the regular expression. Only the basename of the executable is used, not the full path. If spaces or characters interpreted by the shell are used, enclose the *regex* in single quotes. The default is '`-F on`'.

`-a {off|on|ldobjects|src|usedldobjects|usedsrc}`
Specify archiving of binaries and other files. In addition to disable this feature (off), or enable archiving off all loadobjects and sources (on), the other op tions support a more refined selection.

All of these options enable archiving, but the keyword controls what exactly is selected: all load objects (ldobjects), all source files (src), the loadobjects asscoiated with a program counter (usedldobjects), or the source files associated with a program counter (usedsrc). The default is '`-a ldobjects`'.

`-S {off|on|<`*seconds*`>}`
Disable (off), or enable (on) periodic sampling of process-wide resource utilization. By default, sampling occurs every second. Use the <*seconds*> option to change this. The default is '`-S on`'.

`-y <`*signal*`>[,r]`
Controls recording of data with the signal named <*signal*>, referred to as the pause-resume signal. Whenever the given signal is delivered to the process, switch between paused (no data is recorded) and resumed (data is recorded) states.

By default, data collection begins in the paused state. If the optional '`r`' is given, data collection begins in the resumed state and data collection begins immediately.

SIGUSR1 or SIGUSR2 are recommended for this use, but any signal that is not used by the target can be used.

`-l <`*signal*`>`
Specify a signal that will trigger a sample of process-wide resource utilization. When the named <*signal*> is delivered to the process, a sample is recorded.

The signal can be specified using the full name, without the initial letters `SIG`, or the signal number. Note that the `kill` command can be used to deliver a signal.

If both the '`-l`' and '`-y`' options are used, the signal must be different.

`-s <option>[,<API>]`
: Enable synchronization wait tracing, where <*option*> is used to define the specifics of the tracing (on, off, <*threshold*>, or all). The API is selected through the setting for <*API*>: '`n`' selects native/Pthreads, '`j`' selects Java, and '`nj`' selects both. The default is '`-s off`'.

`-H {off|on}`
: Disable (off), or enable (on) heap tracing. The default is '`-H off`'.

`-i {off|on}`
: Disable (off), or enable (on) I/O tracing. The default is '`-i off`'.

NOTES

Any executable in the ELF (Executable and Linkable Format) object format can be used for profiling with gprofng. If debug information is available, gprofng can provide more details, but this is not a requirement.

SEEALSO

gprofng(1), gp-archive(1), gp-display-html(1), gp-display-src(1), gp-display-text(1)

The user guide for gprofng is maintained as a Texinfo manual. If the `info` and `gprofng` programs are correctly installed, the command `info gprofng` should give access to this document.

COPYRIGHT

A.3 Man page for `gprofng display text`

NAME

gprofng display text - Display the performance data in plain text format

SYNOPSIS

`gprofng display text` [*option(s)*] [*commands*] [-script *script-file*] *experiment(s)*

DESCRIPTION

Print a plain text version of the various displays supported by gprofng.

The input consists of one or more experiment directories. Through commands, the user controls the output.

There is a rich set of commands to control the display of the data. The '`NOTES`' section lists the most common ones. The gprofng user guide lists all the commands supported.

Commands specified on the command line need to be prepended with the dash ('-') symbol.

In this example, a function overview will be shown, followed by the source code listing of function '`my-func`', annotated with the performance metrics that have been recorded during the data collection and stored in experiment directory '`my-exp.er`':

```
$ gprofng display text -functions -source my-func my-exp.er
```

Instead of, or in addition to, specifying these commands on the command line, commands may also be included in a file called the *script-file*.

Note that the commands are processed and interpreted from left to right, *so the order matters*.

If this tool is invoked without options, commands, or a script file, it starts in interpreter mode. The user can then issue the commands interactively. The session is terminated with the `exit` command in the interpreter.

OPTIONS

`--version`
: Print the version number and exit.

`--help`
: Print usage information and exit.

`-script` *`script-file`*
: Execute the commands stored in the script file. This feature may be combined with commands specified at the command line.

NOTES

Many commands are supported. Below, the more common ones are listed in mostly alphabetical order, because sometimes it is more logical to swap the order of two entries.

`callers-callees`
: In a callers-callees panel, it is shown which function(s) call the target function (the *callers*) and what functions it is calling (the *callees*). This command prints the callers-callees panel for each of the functions, in the order specified by the function sort metric.

`calltree`
: Display the dynamic call graph from the experiment, showing the hierarchical metrics at each level.

`compare {on | off | delta | ratio}`
: By default, the results for multiple experiments are aggregated. This command changes this to enable the comparison of experiments for certain views (e.g. the function view). The first experiment specified is defined to be the reference. The following options are supported:

  `on`
  : For each experiment specified on the command line, print the values for the metrics that have been activated for the experiment.

  `off`
  : Disable the comparison of experiments. This is the default.

  `delta`
  : Print the values for the reference experiment. The results for the other experiments are shown as a delta relative to the reference (current-reference).

  `ratio`
  : Print the values for the reference experiment. The results for the other experiments are shown as a ratio relative to the reference (current/reference).

`disasm` *`function-name`*
: List the source code and instructions for the function specified. The instructions are annotated with the metrics used.

`fsingle` *`function-name`* `['n']`
: Write a summary panel for the specified function. The optional parameter *n* is needed for those cases where several functions have the same name.

`fsummary`
: Write a summary panel for each function in the function list.

`functions`
: Display a list of all functions executed. For each function the used metrics (e.g. the CPU time) ar shown.

`header`
: Shows several operational characteristics of the experiment(s) specified on the command line.

`limit` *n*
: Limit the output to *n* lines.

`lines`
: Write a list of source lines and their metrics, ordered by the current sort metric.

`metric_list`
: Display the currently selected metrics in the function view and a list of all the metrics available for the target experiment(s).

`metrics` *metric-spec*
: Define the metrics to be displayed in the function and callers-callees overviews.

  The *metric-spec* can either be the keyword '`default`' to restore the default metrics selection, or a colon separated list with metrics.

  The gprofng user guide has more details how to define metrics.

`name {short | long | mangled}[:{soname | nosoname}]`
: Specify whether to use the short, long, or mangled form of function names. Optionally, the load object that the function is part of can be included in the output by adding the *soname* keyword. It can also be ommitted (*nosoname*), which is the default.

  Whether there is an actual difference between these types of names depends on the language.

  Note that there should be no (white)space to the left and right of the colon ('`:`').

`overview`
: Shows a summary of the recorded performance data for the experiment(s) specified on the command line.

`pcs`
: Write a list of program counters (PCs) and their metrics, ordered by the current sort metric.

`sort` *metric-spec*
: Sort the function list on the *metric-spec* given.

  The data can be sorted in reverse order by prepending the metric definition with a minus ('`-`') sign.

  For example `sort -e.totalcpu`.

A default metric for the sort operation has been defined and since this is a persistent command, this default can be restored with `default` as the key (`sort default`).

`source` *`function-name`*
: List the source code for the function specified, annotated with the metrics used.

`viewmode {user | expert | machine}`
: This command is only relevant for Java programs. For all other languages supported, the viewmode setting has no effect.

  The following options are supported:

  `user` Show the Java call stacks for Java threads, but do not show housekeeping threads. The function view includes a function called '`<JVM-System>`'. This represents the aggregated time from non-Java threads. In case the JVM software does not report a Java call stack, time is reported against the function '`<no Java callstack recorded>`'.

  `expert` Show the Java call stacks for Java threads when the user Java code is executed, and machine call stacks when JVM code is executed, or when the JVM software does not report a Java call stack. Show the machine call stacks for housekeeping threads.

  `machine` Show the actual native call stacks for all threads. This is the view mode for C, C++, and Fortran.

SEEALSO

gprofng(1), gp-archive(1), gp-collect-app(1), gp-display-html(1), gp-display-src(1)

The user guide for gprofng is maintained as a Texinfo manual. If the `info` and `gprofng` programs are correctly installed, the command `info gprofng` should give access to this document.

COPYRIGHT

Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

A.4 Man page for `gprofng display html`

NAME

gprofng display html - Generate an HTML based directory structure to browse the profiles

SYNOPSIS

`gprofng display html` [*option(s)*] *experiment(s)*

DESCRIPTION

Process one or more experiments to generate a directory containing the `index.html` file that may be used to browse the experiment data.

OPTIONS

`--version`
: Print the version number and exit.

`--help`
: Print usage information and exit.

`--verbose {on|off}`
: Enable (‘`on`’) or disable (‘`off)`’ verbose mode. The default is ‘`off`’.

`--debug {on|s|m|l|xl|off}`
`-d {on|s|m|l|xl|off}`
: Control the printing of run time information to assist with troubleshooting, or further development of this tool. The keyword is case insensitive. A setting of ‘`on`’ gives a modest amount of information. The keywords ‘`s`’, ‘`m`’, ‘`l`’, and ‘`xl`’ give an increasing amount of information, while ‘`off`’ disables the printing of debug information. This is also the default.

  Note that currently ‘`on`’, ‘`s`’, ‘`m`’, and ‘`l`’ are equivalent. This is expected to change in future updates.

`---highlight-percentage` *value*
`-hp` *value*
: Set a percentage value in the interval [0,100] to select and color code source lines, as well as instructions, that are within this percentage of the maximum metric value(s). The default is 90 (%).

  A value of zero ‘`(-hp 0)`’ disables this feature.

`--output` *dirname*
`-o` *dirname*
: Use *dirname* as the directory name to store the HTML files in. The default name is '`display.<n>.html`' with <*n*> the first positive integer number not in use. An existing directory with the same name is not overwritten.

`--overwrite` *dirname*
`-O` *dirname*
: Use *dirname* as the directory name to store the HTML files in.

`--quiet {on|off}`
`-q {on|off}`
: Control the display of all warning, debug and verbose messages. If set to '`on`', the settings for verbose, warnings and debug are ignored. By default the quiet mode is disabled ('`-q off`').

`--warnings {on|off}`
`-w {on|off}`
: Enable ('`on`'), or disable ('`off`') run time warning messages from the tool. By default these are enabled.

NOTES

When setting a directory name for the HTML files to be stored in, make sure that umask is set to the correct access permissions.

Regardless of the setting for the warning messages, any warnings are accessible through the main `index.html` page.

SEEALSO

gprofng(1), gp-archive(1), gp-collect-app(1), gp-display-src(1), gp-display-text(1)

The user guide for gprofng is maintained as a Texinfo manual. If the `info` and `gprofng` programs are correctly installed, the command `info gprofng` should give access to this document.

COPYRIGHT

A.5 Man page for `gprofng display src`

NAME

gprofng display src - Display the source code, optionally interleaved with the disassembly of the target object

SYNOPSIS

`gprofng display src` [*option(s)*] *target_file*

DESCRIPTION

Display the source code listing, or source code interleaved with disassembly code, as extracted from the target file (an executable, shared object, object file, or a Java .class file).

For example, this command displays the source code and disassembly listing for a function called '`mxv_core`' that is part of object file '`mxv.o`':

```
$ gprofng display src -disasm mxv_core mxv.o
```

To list the source code and disassembly for all the functions in this file, use the following command:

```
$ gprofng display src -disasm all -1 mxv.o
```

The *target_file* is the name of an executable, a shared object, an object file (.o), or a Java .class file.

If no options are given, the source code listing of the *target_file* is shown. This is equivalent to '`-source all -1`'. If this information is not available, a message to this extent is printed.

OPTIONS

`--version`
: Print the version number and exit.

`--help`
: Print usage information and exit.

`-functions`
: List all the functions from the given object.

`-source` *item tag*
: Show the source code for *item* in *target_file*. The *tag* is used to differentiate in case there are multiple occurences with the same name. See the '`NOTES`' section for the definition of *item* and *tag*.

`-disasm` *item tag*
: Include the disassembly in the source listing. The default listing does not include the disassembly. If the source code

is not available, show a listing of the disassembly only. See the '`NOTES`' section for the definition of *item* and *tag*.

`-outfile` *filename*
: Write results to file *filename*. A dash (-) writes to stdout. This is also the default. Note that this option only affects those options included to the right of this option.

NOTES

Use *item* to specify the name of a function, or of a source or object file that was used to build the executable, or shared object.

The *tag* is an index used to determine which item is being referred to when multiple functions have the same name. It is required, but will be ignored if not necessary to resolve the function.

The *item* may also be specified in the form '`function‘file‘`', in which case the source or disassembly of the named function in the source context of the named file will be used.

The special *item* and *tag* combination '`all -1`', is used to indicate generating the source, or disassembly, for all functions in the *target_file*.

SEEALSO

gprofng(1), gp-archive(1), gp-collect-app(1), gp-display-html(1), gp-display-text(1)

The user guide for gprofng is maintained as a Texinfo manual. If the info and gprofng programs are correctly installed, the command `info gprofng` should give access to this document.

COPYRIGHT

A.6 Man page for `gprofng archive`

NAME

gprofng archive - Archive gprofng experiment data

SYNOPSIS

`gprofng archive` [*option(s)*] *experiment*

DESCRIPTION

Archive the associated application binaries and source files in a gprofng experiment to make it self contained and portable.

By default, the binaries are archived, but the application source files are not archived. Use this tool to change this and afterwards archive additional components.

OPTIONS

`--version`

Print the version number and exit.

`--help`

Print usage information and exit.

`-a {off|on|ldobjects|src|usedldobjects|usedsrc}`

Specify archiving of binaries and other files. In addition to disable this feature (off), or enable archiving off all loadobjects and sources (on), the other op tions support a more refined selection.

All of these options enable archiving, but the keyword controls what exactly is selected: all load objects (ldobjects), all source files (src), the loadobjects asscoiated with a program counter (usedldobjects), or the source files associated with a program counter (usedsrc). The default is '`-a ldobjects`'.

`-n`

Archive the named experiment only, not any of its descendants.

`-m` *regex*

Archive only those source, object, and debug info files whose full path name matches the given POSIX compliant *regex* regular expression.

`-q`

Do not write any warnings to stderr. Warnings are incorporated into the .archive file in the experiment directory. They are shown in the output of `gprofng display text`.

`-F`

Force writing or rewriting of the archive. This is ignored with the '`-n`' or '`-m`' option, or if this is a subexperiment.

`-d` *path*

The *path* is the absolute path path to a common archive, which is a directory that contains archived files. If the directory does not exist, then it will be created. Files are saved in the common archive directory, and a symbolic link is created in the experiment archive.

NOTES

Default archiving does not occur in case the application profiled terminates prematurely, or if archiving is disabled when collecting the performance data. In such cases, this tool can be used to afterwards archive the information, but it has to be run on the same system where the profiling data was recorded.

Some Java applications store shared objects in jar files. By default, such shared objects are not automatically archived. To archive shared objects contained in jar files, the addpath directive in an .er.rc file. The addpath directive should give the path to the jar file, including the jar file itself. The .er.rc file should be saved in the user home directory or parent of the experiment directory.

SEEALSO

gprofng(1), gp-collect-app(1), gp-display-html(1), gp-display-src(1), gp-display-text(1)

The user guide for gprofng is maintained as a Texinfo manual. If the info and gprofng programs are correctly installed, the command `info gprofng` should give access to this document.

COPYRIGHT

Index

C

D

E

F

G

H

I

J

L

M

O

P

S

T

V