GNU gprofng

The next generation profiling tool for Linux
version 1.0 (last updated 22 February 2022)

Ruud van der Pas

This document is the manual for gprofng, last updated 22 February 2022.

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.
- 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 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 stopped 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. Outside of 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.exe.

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

This is an example that multiplies a 3000 by 2000 matrix with a vector of length 2000 using 2 threads:

```
$ ./mxv-pthreads.exe -m 3000 -n 2000 -t 2
mxv: error check passed - rows = 3000 columns = 2000 threads = 2
$
```

The program performs an internal check to verify the results are correct. The result of this check is printed, followed by 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.exe -m 3000 -n 2000 -t 1
```

This command produces the following output:

```
Creating experiment database test.1.er (Process ID: 2416504) ...
mxv: error check passed - rows = 3000 columns = 2000 threads = 1
```

We see the message that a directory with the name `test.1.er` has been created. The application then 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 with a hyphen (`-`) if used on the command line.

For example, with the `functions` command we request a list of the functions that have been executed and 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.     Incl.      Name
Total     Total
CPU sec.  CPU sec.
2.272     2.272      <Total>
2.160     2.160      mxv_core
0.047     0.103      init_data
0.030     0.043      erand48_r
0.013     0.013      __drand48_iterate
0.013     0.056      drand48
0.008     0.010      _int_malloc
0.001     0.001      brk
0.001     0.002      sysmalloc
0.        0.001      __default_morecore
0.        0.113      __libc_start_main
0.        0.010      allocate_data
0.        2.160      collector_root
0.        2.160      driver_mxv
0.        0.113      main
0.        0.010      malloc
0.        0.001      sbrk
```

As easy and simple as these steps are, we do have a first profile of our program! There are three columns. The first two contain the *Total CPU Time*, which is the sum of the user and system time. See Section 4.2 [Inclusive and Exclusive Metrics], page 45, 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 the sort metric can be changed by the user.

We then see three columns with the exclusive and inclusive CPU times, plus the name of the function.

The function with the name `<Total>` is not a user function, but is introduced by `gprofng` and is used to display the accumulated metric values. In this case, we see that the total CPU time of this job was `2.272` seconds.

With `2.160` seconds, function `mxv_core` is the most time consuming function. It is also a leaf function.

The next function in the list is `init_data`. Although the CPU time spent in this part is negligible, this is an interesting entry because the inclusive CPU time of `0.103` seconds is higher than the exclusive CPU time of `0.047` seconds. Clearly it is calling another function, or even more than one function. See Section 3.1.12 [The Call Tree], page 17, for the details how to get more information on this.

The function `collector_root` does not look familiar. It is one of the internal functions used by `gprofng collect app` and can be ignored. 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 is a very useful command to get more details on a function. See Section 3.1.15 [Information on Load Objects], page 20.

3.1.3 The Source Code View

In general, you would like to focus the tuning efforts on the most time consuming part(s) of the program. In this case that is easy, since 2.160 seconds on a total of 2.272 seconds is spent in function `mxv_core`. That is 95% of the total and it is time to dig deeper and look at the time distribution at the source code level.

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`.

Please note that the source code has to be compiled with the `-g` option in order for the source code feature to work. Otherwise the location can not be determined.

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

The slightly modified output is as follows:

```
Source file: <apath>/mxv.c
```

```
Object file: mxv-pthreads.exe (found as test.1.er/archives/...)
Load Object: mxv-pthreads.exe (found as test.1.er/archives/...)

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

   <lines deleted>
                                <Function: mxv_core>
   0.        0.             32. void __attribute__ ((noinline))
                                mxv_core (
                                uint64_t row_index_start,
                                uint64_t row_index_end,
                                uint64_t m, uint64_t n,
                                double **restrict A,
                                double *restrict b,
                                double *restrict c)
   0.        0.             33. {
   0.        0.             34.    for (uint64_t i=row_index_start;
                                        i<=row_index_end; i++) {
   0.        0.             35.       double row_sum = 0.0;
## 1.687     1.687          36.       for (int64_t j=0; j<n; j++)
   0.473     0.473          37.          row_sum += A[i][j]*b[j];
   0.        0.             38.       c[i] = row_sum;
                            39.    }
   0.        0.             40. }
```

The first three lines provide information on the location of the source file, the object file and the load object (See Section 4.6 [Load Objects and Functions], page 47).

Function `mxv_core` is part of a source file that has other functions as well. These functions will be shown, but without timing information. They have been removed in the output shown above.

This 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 find.

What we see is that all of the time is spent in lines 36-37.

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 15).

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.

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

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

Excl.     Incl.  Name
Total     Total
CPU sec.  CPU sec.
2.272     2.272  <Total>
1.687     1.687  mxv_core, line 36 in "mxv.c"
0.473     0.473  mxv_core, line 37 in "mxv.c"
0.032     0.088  init_data, line 72 in "manage_data.c"
0.030     0.043  <Function: erand48_r, instructions without line numbers>
0.013     0.013  <Function: __drand48_iterate, instructions without ...>
0.013     0.056  <Function: drand48, instructions without line numbers>
0.012     0.012  init_data, line 77 in "manage_data.c"
0.008     0.010  <Function: _int_malloc, instructions without ...>
0.003     0.003  init_data, line 71 in "manage_data.c"
```

What this overview immediately highlights is that the next most time consuming source line takes 0.032 seconds only. With an inclusive time of 0.088 seconds, it is also clear that this branch of the code does not impact 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. This is when the disassembly view comes in. It is activated 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.

This is what we get for our example:

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

```
Source file: <apath>/mxv.c
Object file: mxv-pthreads.exe (found as test.1.er/archives/...)
Load Object: mxv-pthreads.exe (found as test.1.er/archives/...)

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

   <lines deleted>
                          32. void __attribute__ ((noinline))
                              mxv_core (
                              uint64_t row_index_start,
                              uint64_t row_index_end,
                              uint64_t m, uint64_t n,
                              double **restrict A,
                              double *restrict b,
                              double *restrict c)
                          33. {
```

```
                                     <Function: mxv_core>
    0.         0.                    [33]   4021ba:  mov    0x8(%rsp),%r10
                                34.    for (uint64_t i=row_index_start;
                                            i<=row_index_end; i++) {
    0.         0.                    [34]   4021bf:  cmp    %rsi,%rdi
    0.         0.                    [34]   4021c2:  jbe    0x37
    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.                    [37]   4021c5:  mov    (%r8,%rdi,8),%rdx
    0.         0.                    [36]   4021c9:  mov    $0x0,%eax
    0.         0.                    [35]   4021ce:  pxor   %xmm1,%xmm1
    0.002      0.002                 [37]   4021d2:  movsd  (%rdx,%rax,8),%xmm0
    0.096      0.096                 [37]   4021d7:  mulsd  (%r9,%rax,8),%xmm0
    0.375      0.375                 [37]   4021dd:  addsd  %xmm0,%xmm1
 ## 1.683      1.683                 [36]   4021e1:  add    $0x1,%rax
    0.004      0.004                 [36]   4021e5:  cmp    %rax,%rcx
    0.         0.                    [36]   4021e8:  jne    0xffffffffffffffea
                                38.        c[i] = row_sum;
    0.         0.                    [38]   4021ea:  movsd  %xmm1,(%r10,%rdi,8)
    0.         0.                    [34]   4021f0:  add    $0x1,%rdi
    0.         0.                    [34]   4021f4:  cmp    %rdi,%rsi
    0.         0.                    [34]   4021f7:  jb     0xd
    0.         0.                    [35]   4021f9:  pxor   %xmm1,%xmm1
    0.         0.                    [36]   4021fd:  test   %rcx,%rcx
    0.         0.                    [36]   402200:  jne    0xffffffffffffffc5
    0.         0.                    [36]   402202:  jmp    0xffffffffffffffe8
                                39.    }
                                40. }
    0.         0.                    [40]   402204:  ret
```

For each instruction, the timing values are given and we can exactly which ones are the most expensive. As with the source level view, the most expensive instructions are market with the `##` symbol.

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:

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

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

Excl.     Incl.      Name
Total     Total
CPU sec.  CPU sec.
2.272     2.272  <Total>
1.683     1.683  mxv_core + 0x00000027, line 36 in "mxv.c"
0.375     0.375  mxv_core + 0x00000023, line 37 in "mxv.c"
0.096     0.096  mxv_core + 0x0000001D, line 37 in "mxv.c"
```

```
0.027     0.027  init_data + 0x000000BD, line 72 in "manage_data.c"
0.012     0.012  init_data + 0x00000117, line 77 in "manage_data.c"
0.008     0.008  _int_malloc + 0x00000A45
0.007     0.007  erand48_r + 0x00000062
0.006     0.006  drand48 + 0x00000000
0.005     0.005  __drand48_iterate + 0x00000005
```

3.1.5 Display and Define the Metrics

The default metrics shown by `gprofng display text` are useful, but there is more recorded than displayed. 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 metrics in use, plus all the metrics that have been stored as part of the experiment. The second command may be used to define the metric list.

In our example we get the following values for the metrics:

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

```
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 currently in use, 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 default metrics are set to the exclusive and inclusive total CPU times, plus 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 display the exclusive total CPU time, both as a number and a percentage, use the following metric definition: `e.%totalcpu`

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

3.1.6 A First Customization of the Output

With the information just given, we can customize the function overview. For sake of the example, we would like to display the name of the function first, 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>            2.272 100.00
 mxv_core           2.160  95.04
 init_data          0.047   2.06
 erand48_r          0.030   1.32
 __drand48_iterate  0.013   0.57
 drand48            0.013   0.57
 _int_malloc        0.008   0.35
 brk                0.001   0.04
 sysmalloc          0.001   0.04
 __default_morecore 0.      0.
 __libc_start_main  0.      0.
 allocate_data      0.      0.
 collector_root     0.      0.
 driver_mxv         0.      0.
 main               0.      0.
 malloc             0.      0.
 sbrk               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.

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. You either need to explicitly remove an existing directory first, or use a name that is not in use yet.

This is in contrast with the behaviour for the `-O` option. Any existing (experiment) 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 overviews, including the function view, but it 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 4.3 [Metric Definitions], page 45) in the metric definition is ignored since this does not affect the outcome of the sorting operation. For example if we set the sort key 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.

3.1.10 Scripting

As is probably clear by now, 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 more elegant way to control the behaviour 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.

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 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, to reflect the name of the algorithm and the number of threads that were used in the experiment, we select `mxv.1.thr.er` as the name of the experiment directory. All we then need to do is to add the `-O` option followed by this name on the command line when running `gprofng collect app`:

```
$ exe=mxv-pthreads.exe
$ m=3000
$ n=2000
$ gprofng collect app -O mxv.1.thr.er ./$exe -m $m -n $n -t 1
```

The commands to generate the profile are put into a file that we simply call `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 then specified as input to the `gprofng display text` command that is used to display the performance information stored in `mxv.1.thr.er`:

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

The command above 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.      %
2.272 100.00  2.272 100.00   <Total>
2.159  95.00  2.159  95.00   mxv_core
0.102   4.48  0.054   2.37   init_data
0.035   1.54  0.025   1.10   erand48_r
0.048   2.11  0.013   0.57   drand48
```

In the first part of the output, our comment lines in the script file are shown. 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.

It is now immediately clear that function `mxv_core` is responsbile for 95% of the CPU time and `init_data` takes 4.5% only.

This is also where we see sampling in action. Although this is exactly the same job we profiled before, the timings are somewhat different, but the differences are very small.

3.1.12 The Call Tree

The call tree shows the dynamic hierarchy of the application by displaying the functions executed and their parent. It helps to find the most expensive path in the program.

This feature is enabled through the `calltree` command. This is how to get this 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.      Name
Total
CPU sec.
2.272      +-<Total>
2.159        +-collector_root
2.159        |  +-driver_mxv
2.159        |    +-mxv_core
0.114        +-__libc_start_main
0.114          +-main
0.102            +-init_data
0.048            |  +-drand48
0.035            |    +-erand48_r
0.010            |      +-__drand48_iterate
0.011            +-allocate_data
0.011            |  +-malloc
0.011            |    +-_int_malloc
0.001            |      +-sysmalloc
0.001            +-check_results
0.001              +-malloc
0.001                +-_int_malloc
```

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

First of all, function `collector_root` is internal to `gprofng` and should be hidden to the user. This is part of a planned future enhancement.

Recall that 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 4.6 [Load Objects and Functions], page 47.

Another thing to note is that there are two main branches. The one under `collector_root` and the second one under `__libc_start_main`. This reflects the fact that we are executing a parallel program. Even though we only used one thread for this run, this is still executed in a separate path.

The main, sequential part of the program is displayed under `main` and shows the functions called and the time they took.

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 to 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 32), 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 actually command executed, and some global performance statistics can be displayed.

The `header` command displays information about the experiment(s). For example, this is the command 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. The textual changes are marked with the `<` and `>` symbols.

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

Target command (64-bit): ’./mxv-pthreads.exe -m 3000 -n 2000 -t 1’
Process pid 30591, ppid 30589, pgrp 30551, sid 30468
Current working directory: <cwd>
Collector version: ‘2.36.50’; experiment version 12.4 (64-bit)
Host ‘<hostname>’, OS ‘Linux <version>’, page size 4096,
     architecture ‘x86_64’
  16 CPUs, clock speed 1995 MHz.
  Memory: 30871514 pages @  4096 = 120591 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: 2.293162658
Data Collection Duration: 2.293162658
```

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

Related to this command there is a useful option to record your own 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 our newly created experiment directory:

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

```
  Experiment(s):

  Experiment      :mxv.1.thr.er
    Target        : './mxv-pthreads.exe -m 3000 -n 2000 -t 1'
    Host          : <hostname> (<ISA>, Linux <version>)
    Start Time    : <date and time>
    Duration      : 2.293 Seconds

  Metrics:

    Experiment Duration (Seconds): [2.293]
    Clock Profiling
      [X]Total CPU Time - totalcpu (Seconds): [*2.272]

  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 talk about 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>.

One may wonder why there is an option to disable clock based profiling. This is because by default, it is enabled when conducting hardware event counter experiments (See Section 3.4 [Profile Hardware Event Counters], page 32). 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.

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 list contains 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 4.6 [Load Objects and Functions], page 47).

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, the (long) 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.exe> (<apath>/mxv-pthreads.exe)
<librt-2.17.so> (/usr/lib64/librt-2.17.so)
<libdl-2.17.so> (/usr/lib64/libdl-2.17.so)
<libbfd-2.36.50.20210505.so> (<apath>/libbfd-2.36.50 <etc>)
<libopcodes-2.36.50.20210505.so> (<apath>/libopcodes-2. <etc>)
<libc-2.17.so> (/usr/lib64/libc-2.17.so)
<libpthread-2.17.so> (/usr/lib64/libpthread-2.17.so)
<libm-2.17.so> (/usr/lib64/libm-2.17.so)
<libgp-collector.so> (<apath>/libgp-collector.so)
<ld-2.17.so> (/usr/lib64/ld-2.17.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 `collector_root` function shown in the function overview.

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

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

```
collector_root
  Exclusive Total CPU Time: 0.    (  0. %)
  Inclusive Total CPU Time: 2.159 ( 95.0%)
            Size:   401
      PC Address: 10:0x0001db60
     Source File: <apath>/dispatcher.c
     Object File: mxv.1.thr.er/archives/libgp-collector.so_HpzZ6wMR-3b
     Load Object: <apath>/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: 2.272 (100.0%)
  Inclusive Total CPU Time: 2.272 (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: 2.159 ( 95.0%)
  Inclusive Total CPU Time: 2.159 ( 95.0%)
            Size:     75
      PC Address: 2:0x000021ba
     Source File: <apath>/mxv.c
     Object File: mxv.1.thr.er/archives/mxv-pthreads.exe_hRxWdccbJPc
     Load Object: <apath>/mxv-pthreads.exe
    Mangled Name:
         Aliases:

          ... etc ...
```

3.2 Support for Multithreading

In this chapter we introduce and discuss the support for multithreading. 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 parallel environment, 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 we use 2 threads:

```
$ exe=mxv-pthreads.exe
$ m=3000
$ n=2000
$ gprofng collect app -O mxv.2.thr.er ./$exe -m $m -n $n -t 2
```

First of all, note that we did not change anything, other than setting the number of threads to 2. 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 we used before work unmodified. For example, this is all that is needed to get a function overview:

```
$ gpprofng display text -limit 10 -functions mxv.2.thr.er
```

This produces the following familiar looking output:

```
Print limit set to 10
Functions sorted by metric: Exclusive Total CPU Time

Excl.     Incl.      Name
Total     Total
CPU sec.  CPU sec.
2.268     2.268      <Total>
2.155     2.155      mxv_core
0.044     0.103      init_data
0.030     0.046      erand48_r
0.016     0.016      __drand48_iterate
0.013     0.059      drand48
0.008     0.011      _int_malloc
0.003     0.003      brk
0.        0.003      __default_morecore
0.        0.114      __libc_start_main
```

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. This is typically the thread that runs from start to finish and handles the sequential portions of the code, as well as takes care of managing the threads.

It is no different in our 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. To make it easier to interpret the data, we modify the metrics to include percentages:

```
$ gprofng display text -metrics e.%totalcpu -threads mxv.2.thr.er
```

The command above produces the following overview:

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

Excl. Total    Name
CPU
 sec.      %
2.258 100.00   <Total>
1.075  47.59   Process 1, Thread 3
1.070  47.37   Process 1, Thread 2
0.114   5.03   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 5% of the total CPU time, while the other two threads take 47% each.

This view is ideally suited to verify if there 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 4.5 [The Selection List], page 46, 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 10 lines
limit 10
# 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 limiter has been shown and explained before and will be ignored. The new command we focus on is `thread_select`.

This command takes a list (See Section 4.5 [The Selection List], page 46) to select specific threads. In this case we simply use the individual thread numbers that we obtained with the `thread_list` command earlier.

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 timings and their percentages are given.

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

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

Excl. Total    Name
CPU
 sec.      %
0.114 100.00   <Total>
0.051  44.74   init_data
0.028  24.56   erand48_r
0.017  14.91   __drand48_iterate
0.010   8.77   _int_malloc
0.008   7.02   drand48
0.      0.     __libc_start_main
0.      0.     allocate_data
0.      0.     main
0.      0.     malloc
```

As usual, the comment lines are echoed. This is followed by a confirmation of our selection. We see that indeed thread 1 has been selected. What is displayed next is the function overview for this particular thread. Due to the `limit 10` command, there are ten entries in this list.

Below are the overviews for threads 2 and 3 respectively. We see 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.      %
1.072 100.00   <Total>
1.072 100.00   mxv_core
0.      0.     collector_root
0.      0.     driver_mxv

# 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.      %
1.076 100.00   <Total>
1.076 100.00   mxv_core
0.      0.     collector_root
0.      0.     driver_mxv
```

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 4.7 [The Concept of a CPU in gprofng], page 48).

In order to have some more interesting data to look at, we created a new experiment, this time using 8 threads:

```
$ exe=mxv-pthreads.exe
$ m=3000
$ n=2000
$ gprofng collect app -O mxv.8.thr.er ./$exe -m $m -n $n -t 8
```

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 CPU numbers that were used and how much time was spent on each of them. Both are demonstrated below.

```
$ gprofng display text -metrics e.%totalcpu -cpu_list -cpus mxv.8.thr.er
```

This command produces the following output:

```
Current metrics: e.%totalcpu:name
Current Sort Metric: Exclusive Total CPU Time ( e.%totalcpu )
Exp Sel Total
=== === =====
  1 all    10
Objects sorted by metric: Exclusive Total CPU Time

Excl. Total    Name
CPU
 sec.      %
2.310 100.00   <Total>
0.286  12.39   CPU 7
0.284  12.30   CPU 13
0.282  12.21   CPU 5
0.280  12.13   CPU 14
0.266  11.52   CPU 9
0.265  11.48   CPU 2
0.264  11.44   CPU 11
0.194   8.42   CPU 0
0.114   4.92   CPU 1
0.074   3.19   CPU 15
```

What we see in this table is that a total of 10 CPUs have been used. This is followed by a list with all the CPU numbers that have been used during the run. For each CPU it is shown how much time was spent on it.

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 10 CPUs have been used, but we know that the application uses 9 threads only. This means that at least one thread has executed on more than one CPU. In itself this is not something to worry about, but warrants a deeper investigation.

Honesty dictates that next we performed a pre-analysis to find out which thread(s) have been running on more than one CPU. We found this to be thread 7. It has executed on CPUs 0 and 15.

With this knowledge, we wrote the script shown below. It zooms in on the behaviour of thread 7.

```
# Define the metrics
metrics e.%totalcpu
# Limit the output to 10 lines
limit 10
functions
# Get the function overview for CPU 0
cpu_select 0
functions
# Get the function overview for CPU 15
cpu_select 15
functions
```

From the earlier shown threads overview, we know that thread 7 has used `0.268` seconds of CPU time..

By selecting CPUs 0 and 15, respectively, we get the following function overviews:

```
# Get the function overview for CPU 0
Exp Sel Total
=== === =====
  1 0      10
Functions sorted by metric: Exclusive Total CPU Time

Excl. Total    Name
CPU
 sec.      %
0.194 100.00   <Total>
0.194 100.00   mxv_core
0.      0.     collector_root
0.      0.     driver_mxv

# Get the function overview for CPU 15
Exp Sel Total
=== === =====
  1 15     10
Functions sorted by metric: Exclusive Total CPU Time

Excl. Total    Name
CPU
 sec.      %
0.074 100.00   <Total>
0.074 100.00   mxv_core
0.      0.     collector_root
0.      0.     driver_mxv
```

This shows that thread 7 spent `0.194` seconds on CPU 0 and `0.074` seconds on CPU 15.

3.3 Viewing 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.

Mention `experiment_list`

3.3.1 Aggregation of Experiments

By default, the data for multiple experiments is aggregrated and the display commands shows these combined results.

For example, we can aggregate the data for our single and dual thread experiments. Below is the script we used for this:

```
# Define the metrics
metrics e.%totalcpu
# Limit the output to 10 lines
limit 10
# Get the list with experiments
experiment_list
# Get the function overview
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 is is used to verify we are looking at the experiments we intend to aggregate.

```
$ gprofng display text -script my-script-agg mxv.1.thr.er mxv.2.thr.er
```

With the command above, we get the following output:

```
# Define the metrics
Current metrics: e.%totalcpu:name
Current Sort Metric: Exclusive Total CPU Time ( e.%totalcpu )
# Limit the output to 10 lines
Print limit set to 10
# Get the list with experiments
ID Sel   PID Experiment
== === ===== ============
 1 yes 30591 mxv.1.thr.er
 2 yes 11629 mxv.2.thr.er
# Get the function overview
Functions sorted by metric: Exclusive Total CPU Time

Excl. Total    Name
CPU
```

```
 sec.      %
4.533 100.00   <Total>
4.306  94.99   mxv_core
0.105   2.31   init_data
0.053   1.17   erand48_r
0.027   0.59   __drand48_iterate
0.021   0.46   _int_malloc
0.021   0.46   drand48
0.001   0.02   sysmalloc
0.      0.     __libc_start_main
0.      0.     allocate_data
```

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. For example, the total accumulated time is indeed 2.272 + 2.261 = 4.533 seconds.

3.3.2 Comparison of Experiments

The support for multiple experiments really shines in comparison mode. This feature is enabled through the command `compare on` and is disabled by setting `compare off`.

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 one and two threads respectively:

```
$ gprofng display text -compare on -functions mxv.1.thr.er mxv.2.thr.er
```

This produces the following output:

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

mxv.1.thr.er  mxv.2.thr.er  mxv.1.thr.er  mxv.2.thr.er
Excl. Total   Excl. Total   Incl. Total   Incl. Total    Name
CPU           CPU           CPU           CPU
 sec.          sec.          sec.          sec.
2.272         2.261         2.272         2.261          <Total>
2.159         2.148         2.159         2.148          mxv_core
0.054         0.051         0.102         0.104          init_data
0.025         0.028         0.035         0.045          erand48_r
0.013         0.008         0.048         0.053          drand48
0.011         0.010         0.012         0.010          _int_malloc
0.010         0.017         0.010         0.017          __drand48_iterate
0.001         0.            0.001         0.             sysmalloc
0.            0.            0.114         0.114          __libc_start_main
0.            0.            0.011         0.010          allocate_data
0.            0.            0.001         0.             check_results
0.            0.            2.159         2.148          collector_root
0.            0.            2.159         2.148          driver_mxv
```

```
0.            0.            0.114         0.114          main
0.            0.            0.012         0.010          malloc
```

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 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.

This feature is supported on the `compare` command. In addition to `on`, or `off`, this command also supports `delta`, or `ratio`.

Usage of one of these two keywords enables the comparison feature and shows either the difference, or the ratio, relative to the reference data.

In the example below, we use the same two experiments used in the comparison above, but as before, the number of lines is restricted to 10 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 10 lines
limit 10
# Set the comparison mode to differences
compare delta
# Get the function overview
functions
```

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

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

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

```
mxv.1.thr.er  mxv.2.thr.er
Excl. Total   Excl. Total     Name
CPU           CPU
 sec.      %   delta      %
2.272 100.00  -0.011 100.00   <Total>
2.159  95.00  -0.011  94.97   mxv_core
0.054   2.37  -0.003   2.25   init_data
0.025   1.10  +0.003   1.23   erand48_r
0.013   0.57  -0.005   0.35   drand48
0.011   0.48  -0.001   0.44   _int_malloc
0.010   0.44  +0.007   0.75   __drand48_iterate
```

```
0.001   0.04  -0.001   0.     sysmalloc
0.      0.    +0.      0.     __libc_start_main
0.      0.    +0.      0.     allocate_data
```

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

While in this case we used the delta as a comparison,

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 4.8 [Hardware Event Counters Explained], page 48, 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 20, 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, but we try to maximize the overlap across alias sets.

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

See Chapter 19 of the "Intel 64 and IA-32 Architectures Software
Developer's Manual Volume 3B: System Programming Guide"
```

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.exe
$ m=3000
$ n=2000
$ 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   0xffffffffffffffc5
    0.                0          0  [36] 402202: jmp   0xffffffffffffffe8
                                    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.exe
$ m=3000
$ n=2000
$ 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.exe
$ m=3000
$ n=2000
$ 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 Clockycle") can be computed. See Section 4.8 [Hardware Event Counters Explained], page 48. 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` commad 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 4.4 [The Viewmode], page 46, 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 Terminology

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

4.1 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.

4.2 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.

4.3 Metric Definitions

The metrics to be shown are highly customizable. In this section we explain the definitions associated with metrics.

The `metrics` command takes a colon (:) separated list with special keywords. This keyword consists of the following three fields: `<flavor><visibility><metric_name>`.

The <*flavor*> field is either an `e` for “exclusive”, or `i` for “inclusive”. The `<metric_name>` field is the name of the metric request. 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.

4.4 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.

4.5 The Selection List

Several commands allow the user to specify a subset of a list. For example, to select specific threads from all the threads that have been used when conducting the experiment(s).

Such 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. 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 1:1+2:2`
: Select thread 1 from experiment 1 and thread 2 from experiment 2.

`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, as listed by the `by exp_list` command.

4.6 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.

4.7 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, it could be one of those hardware threads.

4.8 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.

4.9 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.

5 Other Document Formats

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 `doc` under directory `gprofng` in the top level directory.

This file can be used to generate the document in the `info`, `html`, and `pdf` formats. The default installation procedure creates a file in the `info` format and stores it in the documentation section of binutils.

The probably easiest way to generate a different format from this Texinfo document is to go to the distribution directory that was created when the tools were built. This is either the default distribution directory, 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 `<dist>`.

The make file called `Makefile` in directory `<dist>/gprofng/doc` supports several commands to generate this document in different formats. We recommend to use these commands.

They create the file(s) and install it in the documentation directory of binutils, which is `<dist>/share/doc` in case `html` or `pdf` is selected and `<dist>/share/info` for the file in the `info` format.

To generate this document in the requested format and install it in the documentation directory, the commands below should be executed. In this notation, `<format>` is one of `info`, `html`, or `pdf`:

```
$ cd <dist>/gprofng/doc
$ make install-<format>
```

Some things to note:

- For the `pdf` file to be generated, the TeX document formatting software is required and the relevant commmands need to be included in the search path. An example of a popular TeX implementation is *TexLive*. It is beyond the scope of this document to go into the details of installing and using TeX, but it is well documented elsewhere.
- 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 `doc` directory.
- The make file also supports commands to only generate the file in the desired format and not move them to the documentation directory. This is accomplished through the `make <format>` command.

Index