Oracle Linux 10

Profiling For Performance Analysis With Gprofng

G14608-02
October 2025

Oracle Linux 10 Profiling For Performance Analysis With Gprofng,

G14608-02

Contents

8 Known Issues

Preface

Oracle Linux 10: Profiling For Performance Analysis With Gprofng describes how to install and use the `gprofng` tool to find performance bottlenecks in executable programs.

1

About `gprofng` and Profiling

Gprofng is a next generation application profiling tool that can be used to diagnose performance bottlenecks in software applications.

The tool can be used to profile programs compiled with toolchains released by Oracle Linux. These programs can be written using the C, C++, Java and Scala programming languages for the x86_64 and aarch64 processor architectures. The full extent of the data that can be collected differs between CPU models and types.

Oracle developed this tool and contributed it back upstream to the `binutils` project so that it's now part of the GNU binutils tools suite.

For more information, see https://sourceware.org/binutils/wiki/gprofng and the `gprofng(1)` manual page.

2 Installing `gprofng`

Install the `binutils-gprofng` package on Oracle Linux 10.

Before installing the `binutils-gprofng` package, enable the `ol10_addons` repository:

```
sudo dnf config-manager --enable ol10_addons
```

```
sudo dnf update -y
```

For more information, see Oracle Linux: Managing Software on Oracle Linux.

The `binutils-gprofng` package provides the gprofng profiling tool and its prerequisites on Oracle Linux systems.

1. Install the `binutils-gprofng` package.

   Use the `dnf` command to install the package:

   ```
   sudo dnf install -y binutils-gprofng
   ```

2. Verify that the `binutils-gprofng` package has installed successfully.

   Use the `gprofng` command to verify its presence:

   ```
   gprofng --version
   ```

The `binutils-gprofng` package is installed.

3 Getting Started With gprofng

Creating an experiment directory, capturing performance data, and inspecting the results.

Install the `binutils-gprofng` package. For more information, see Installing `gprofng`.

The gprofng profiling tool can be used to assist development teams seeking to optimize their code and improve application performance.

1. Set up the experiment directory.

   You can run the `gprofng` command inside any directory because it generates the necessary directory structure automatically.

   Consider creating a separate directory for the performance experiments. That directory can be stored anywhere, for example in the user home directory or as an unversioned subdirectory within a code project folder.

2. Collect performance data for a program.

   Use the `gprofng collect app` command to start the application and collect performance data while it runs:

   ```
   gprofng collect app /path/to/application -options
   ```

3. Review the performance data that has been captured.

   Use the `gprofng display text` command to analyze the performance data. By default, experiment results are stored in an experiment directory that follows the `test.n.er` naming pattern, where `n` is a numerical identifier for the test and `.er` is a required suffix.

   For example, to review the performance data stored in the `test.1.er` directory, run the following command:

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

An experiment directory has been created, performance data was captured, and the test directory in which that performance data is stored can be analyzed by using the `gprofng display` command.

4

`gprofng` Command Reference

This table provides information about the `gprofng` command.

Action	Command	Description
Collect performance data.	`gprofng collect app`	Collects performance data about a running application and stores it in the experiment directory.
Review performance results in a terminal.	`gprofng display text`	Displays performance data from the specified experiment directories in ASCII plain-text format.
Review performance data in a web browser.	`gprofng display html`	Generates a HTML structure from the specified experiment directories.
Review the source and disassembly code.	`gprofng display src`	Displays the source code interleaved with instructions.
Archive an experiment directory.	`gprofng archive`	Copies shared libraries, object files, and source code to the experiment directory for later analysis.
Open a graphical user interface for collecting and reviewing performance data.	`gprofng display gui`	If the `gprofng-gui` package is installed, starts a graphical user interface for collecting performance data and displaying experiments.

Collecting Performance Data

Use the `gprofng collect app` command to configure experiments and collect performance data.

When collecting performance data, it's possible to specify the experiment directory by using the `-O` option:

```
gprofng collect app -O experiment-directory-name.er /path/to/application -options
```

Note that `.er` is a required suffix for any experiment directory name.

For more information about collecting performance data, see Getting Started With gprofng.

Analyzing Performance Metrics

Use the `gprofng display text` command to review a range of performance data that has been collected in an experiment.

To analyze performance data collected for each function, use the `-functions` option with the `gprofng display text` command:

```
gprofng display text -functions experiment-directory-name.er


Functions sorted by metric: Exclusive Total CPU Time

Excl. Total   Incl. Total    Name
CPU           CPU
 sec.      %   sec.      %
5.554 100.00  5.554 100.00   <Total>
5.274  94.95  5.274  94.95   mxv_core
0.140   2.52  0.270   4.86   init_data
0.090   1.62  0.110   1.98   erand48_r
0.020   0.36  0.020   0.36   __drand48_iterate
0.020   0.36  0.130   2.34   drand48
0.010   0.18  0.010   0.18   _int_malloc
0.      0.    0.280   5.05   __libc_start_main
0.      0.    0.010   0.18   allocate_data
0.      0.    5.274  94.95   collector_root
0.      0.    5.274  94.95   driver_mxv
0.      0.    0.280   5.05   main
0.      0.    0.010   0.18   malloc
0.      0.    5.274  94.95   start_thread
```

To limit the number of functions displayed, use the `-limit` option as follows:

```
gprofng display text -limit 5 -functions experiment-directory-name.er


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

Excl. Total   Incl. Total    Name
CPU           CPU
 sec.      %   sec.      %
5.775 100.00  5.775 100.00   <Total>
5.494  95.15  5.494  95.15   mxv_core
0.126   2.18  0.267   4.63   init_data
0.068   1.17  0.104   1.80   erand48_r
0.038   0.66  0.142   2.45   drand48
```

To list all the metrics that have been collected in an experiment directory, use the `-metric_list` option without any other options:

```
gprofng display text -metric_list experiment-directory-name.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
```

After you have established which metrics have been collected in an experiment directory, select the displayed metrics by using the `-metrics` options, separating each with a `:` character:

```
gprofng display text -metrics name:i.%totalcpu:e.%totalcpu -limit 10 -
functions experiment-directory-name.er


Current metrics: name:i.%totalcpu:e.%totalcpu
Current Sort Metric: Exclusive Total CPU Time ( e.%totalcpu )
Print limit set to 10
Functions sorted by metric: Exclusive Total CPU Time

Name                Incl. Total   Excl. Total
                    CPU           CPU
                     sec.      %   sec.      %
 <Total>            5.775 100.00  5.775 100.00
 mxv_core           5.494  95.15  5.494  95.15
 init_data          0.267   4.63  0.126   2.18
 erand48_r          0.104   1.80  0.068   1.17
 drand48            0.142   2.45  0.038   0.66
 __drand48_iterate  0.036   0.62  0.036   0.62
 _int_malloc        0.013   0.22  0.008   0.14
 sysmalloc          0.005   0.09  0.003   0.05
 brk                0.002   0.03  0.002   0.03
 __default_morecore 0.002   0.03  0.      0.
```

To sort the performance data according to a specific metric, for example `name`, add the `-sort` option:

```
gprofng display text -metrics name:i.%totalcpu:e.%totalcpu -sort name -limit
10 -functions experiment-directory-name.er


Current metrics: name:i.%totalcpu:e.%totalcpu
Current Sort Metric: Exclusive Total CPU Time ( e.%totalcpu )
Current Sort Metric: Name ( name )
Print limit set to 10
Functions sorted by metric: Name

Name                Incl. Total   Excl. Total
                    CPU           CPU
                     sec.      %   sec.      %
 <Total>            5.775 100.00  5.775 100.00
 __default_morecore 0.002   0.03  0.      0.
 __drand48_iterate  0.036   0.62  0.036   0.62
 __libc_start_main  0.280   4.85  0.      0.
 _int_malloc        0.013   0.22  0.008   0.14
 allocate_data      0.013   0.22  0.      0.
 brk                0.002   0.03  0.002   0.03
 collector_root     5.494  95.15  0.      0.
```

```
drand48            0.142   2.45  0.038   0.66
driver_mxv         5.494  95.15  0.      0.
```

Use the `-disasm` option and specify the function name to review metrics at an instruction or assembly level:

```
gprofng display text -metrics e.totalcpu -disasm function-name experiment-directory-name.er
```

For more information, see https://sourceware.org/binutils/wiki/gprofng and the `gprofng(1)` manual page.

Working With gprofng-gui

Use the `gprofng display gui` command to open a GUI front-end for the `gprofng` command, if the `gprofng-gui` package is installed.

To review an experiment containing performance data in a graphical user interface, use the `gprofng display gui` command, which is provided by the `gprofng-gui` package:

```
gprofng display gui experiment1.er
```

If more than one experiment is specified, they're aggregated together.

To compare two experiments containing performance data in a graphical user interface, use the `-c` option:

```
gprofng display gui -c experiment1.er experiment2.er
```

For more information about working with the `gprofng-gui` tool, see Using the Gprofng GUI.

5
Storing `gprofng` Options for Reuse

Use Scripts to save `gprofng display` options.

There can be many ways to customize the output of the `gprofng display text` command, and the `-script` option has been provided so that you can supply a text file containing options for use with other experiment directories. That can be a more straightforward way to reproduce performance views.

For example, to get a table with the inclusive and exclusive total CPU times with percentages, limited to the first 10 lines, create a script with the following content:

```
# Command to define the metrics
metrics name:i.%totalcpu:e.%totalcpu
# Limit the views to 10 lines
limit 10
# Display the function overview
functions
```

Note that each option has its own line in the script and no leading dash `-` character.

That script can then be used with the `gprofng display text` command as follows:

```
gprofng display text -script script-name experiment-directory-name.er
```

```
# Command to define the metrics
Current metrics: name:i.%totalcpu:e.%totalcpu
Current Sort Metric: Exclusive Total CPU Time ( e.%totalcpu )
# Limit the views to 10 lines
Print limit set to 10
# Display the function overview
Functions sorted by metric: Exclusive Total CPU Time

Name                Incl. Total   Excl. Total
                    CPU           CPU
                     sec.      %   sec.      %
 <Total>            5.775 100.00  5.775 100.00
 mxv_core           5.494  95.15  5.494  95.15
 init_data          0.267   4.63  0.126   2.18
 erand48_r          0.104   1.80  0.068   1.17
 drand48            0.142   2.45  0.038   0.66
 __drand48_iterate  0.036   0.62  0.036   0.62
 _int_malloc        0.013   0.22  0.008   0.14
 sysmalloc          0.005   0.09  0.003   0.05
 brk                0.002   0.03  0.002   0.03
 __default_morecore 0.002   0.03  0.      0.
```

6

Working With `gprofng` and Threaded Applications

Collect performance data for multithreaded applications.

By default, the performance data for a multithreaded application is aggregated over all threads. If the data for individual threads, or a set of threads, is needed, then filters can be used to analyze that information.

To list all the threads for which performance data has been captured, use the `-thread_list` and `-threads` options:

```
gprofng display text -thread_list -threads experiment-directory-name.er


Exp Sel Total
=== === =====
  1 all     3
Objects sorted by metric: Exclusive Total CPU Time

Excl. Total    Name
CPU
 sec.      %
5.775 100.00   <Total>
2.773  48.01   Process 1, Thread 3
2.722  47.13   Process 1, Thread 2
0.280   4.85   Process 1, Thread 1
```

In the example output you can see that the CPU times for three threads were used during that experiment. The thread number can be used to review information for both a specific thread or a group of threads.

To review the performance data for a specific thread, use the `-thread_select` option with a thread number:

```
gprofng display text -limit 5 -thread_select 1 -functions experiment-
directory-name.er


Print limit set to 5
Exp Sel Total
=== === =====
  1 1       3
Functions sorted by metric: Exclusive Total CPU Time

Excl. Total   Incl. Total    Name
CPU           CPU
 sec.      %   sec.      %
0.280 100.00  0.280 100.00   <Total>
0.126  44.84  0.267  95.37   init_data
0.068  24.20  0.104  37.01   erand48_r
```

```
0.038  13.52  0.142  50.53   drand48
0.036  12.81  0.036  12.81   __drand48_iterate
```

7

Using the Gprofng GUI

The `gprofng-gui` package provides an Oracle Java desktop application that can optionally function as a GUI front-end for the `gprofng` command.

It also provides a range of views and filters that can be used to customize and display the performance data that's been collected in experiments, and yield meaningful insights about where the performance bottlenecks are in an executable software program.

For more information, see http://savannah.gnu.org/projects/gprofng-gui/ and the `gp-display-gui(1)` manual page.

Installing gprofng-gui

Install the `gprofng-gui` package on Oracle Linux 10.

Before installing the `gprofng-gui` package, follow the steps in Installing `gprofng`.

Also install an Oracle Java runtime. For more information about installing Oracle Java SE, see https://docs.oracle.com/en/java/javase/index.html. On Oracle Cloud Infrastructure instances, follow the steps in Install Oracle Java SE on Oracle Linux.

The `gprofng-gui` package provides an optional graphical user interface for creating experiments and reviewing performance data.

1. Install the `gprofng-gui` package.

   Use the `dnf` command to install the package:

   ```
   sudo dnf install -y gprofng-gui
   ```

2. Verify that the `gprofng-gui` package has installed successfully.

   Use the `gprofng` command to verify its presence:

   ```
   gprofng display gui --version
   ```

The `gprofng-gui` package is installed.

Getting Started With gprofng-gui

Create new experiments by using the GUI front-end for `gprofng`.

Install the `gprofng-gui` package. For more information, see Installing gprofng-gui.

The `gprofng-gui` tool provides a GUI front-end for software engineers and testers to use for collecting performance data for in-development applications.

1. Open a new `gprofng-gui` window.

Use the `gprofng` command to start the application:

```
gprofng display gui
```

2. Click **File**, then **Profile Application**.
3. Configure a new experiment in the **Profile Application** window.
   a. Set the application path, command line options, and working directory in the **Specify Application to Profile** section.
   b. Set the experiment file name and experiment directory in the **Specify Experiment** section.
   c. Configure the data that `gprofng` collects during the experiment in the **Data to Collect** tab.
4. Start the experiment by clicking **Run** in the **Profile Application** window.

   Switch to the **Output** tab to see the live results of the experiment as the performance data is collected.
5. After the experiment completes, you can open that experiment in the GUI. You can end the experiment early by clicking **Terminate** in the **Profile Application** window.

Performance data that was collected for a binary executable while the experiment was running has been saved for later review and comparison.

Reviewing Experiments With gprofng-gui

Review previous experiments by using the GUI front-end for `gprofng`.

Before you begin, create an experiment by using the `gprofng` command. For more information, see Getting Started With gprofng.

Or, create an experiment by using the `gprofng-gui` tool. For more information, see Getting Started With gprofng-gui.

The `gprofng-gui` provides a wide range of views and filters that are useful for reviewing performance data captured in software profiling experiments.

1. Open a new `gprofng-gui` window.

   Use the `gprofng` command to start the application:

   ```
   gprofng display gui
   ```
2. Click **File**, then **Open Experiment**.

   For experiments that were created recently, you might also see them listed under **Open Recent Experiment** in the **File** menu.
3. If you already know the experiment directories ahead of time, you can optionally open an experiment in a single step.

   Use the `gprofng` command to start the application with the experiment provided as an option:

   ```
   gprofng display gui experiment1.er
   ```

   If more than one experiment is specified, they're aggregated together.

An experiment has now been opened for review purposes in the main window for the `gprofng-gui` tool.

Comparing Experiments With gprofng-gui

Compare experiments by using the GUI front-end for `gprofng`.

Before you begin, create at least two experiments by using the `gprofng` command. For more information, see Getting Started With gprofng.

Or, create at least two experiments by using the `gprofng-gui` tool. For more information, see Getting Started With gprofng-gui.

The `gprofng-gui` provides a wide range of views and filters that are useful for comparing performance data captured across several software profiling experiments.

1. Open a new `gprofng-gui` window.

   Use the `gprofng` command to start the application:

   ```
   gprofng display gui
   ```

2. Click **File**, then **Compare Experiments**.
3. In the **Compare Experiments** window, specify two experiments, then click **OK**.

   You can add more than two experiments by clicking **More**. You can also switch the comparison order by clicking **Reverse**.
4. If you already know the experiment directories ahead of time, you can optionally start comparing experiments in a single step.

   Use the `gprofng` command to start the application, with the relevant experiments listed after specifying the `-c` option:

   ```
   gprofng display gui -c experiment1.er experiment2.er
   ```

Two or more experiments have now been opened for comparison purposes in the main window for the `gprofng-gui` tool.

gprofng-gui Views Reference

This table lists the views that are displayed by default in the **Views** menu when at least one experiment is open in the `gprofng-gui` main window.

View	Purpose
Callers-Callees	Displays calling relationships between functions run during the experiment, along with performance metrics.
Call Tree	Displays a tree node graph for calling relationships between functions run during the experiment.
Disassembly	Displays instructions for a selected function run during the experiment, along with performance metrics.

View	Purpose
Experiments	Lists the experiments that are displayed in the main window, along with any notes or comments that have been added to them.
Flame Graph	Displays a customizable graph for the call tree with call stacks on one axis and functions on the other. The width of the functions visually indicates the amount of CPU time used.
Functions	Lists the functions run during the experiment, along with CPU performance metrics. Click the function name to see more detailed information in a separate viewing pane.
Processes	Lists processes that were created during the experiment, along with performance metrics. Use filters for diagnostic purposes.
Source/Disassembly	Displays source code for a selected function, with performance metrics for each line.
Threads	List threads that were created during the experiment, along with performance metrics. Use filters for diagnostic purposes.
Timeline	Displays color-coded performance behavior throughout the duration of the experiment, based on the performance data that was collected.

Working With the Callers-Callees View

Use the **Callers-Callees** view provided by `gprofng-gui` to diagnose performance bottlenecks in a software program at a function call level.

The **Callers-Callees** view displays calling relationships between functions that were run while performance data was being collected, along with performance metrics.

The view is split into three horizontal panel. The middle panel displays the selected function. Other functions that make calls to the selected function are displayed in the upper panel, and any functions that are called by the selected function are displayed in the panel underneath.

A separate viewing pane displays extra information when a function is selected.

Working With the Call Tree View

Use the **Call Tree** view provided by `gprofng-gui` to diagnose performance bottlenecks in a software program at a function call level.

The **Call Tree** view displays a tree graph that reflects calling relationships between functions that were run while performance data was being collected, along with performance metrics.

Each function call is allocated a tree node that can be expanded and collapsed to display a list of related function calls, and those in turn are also allocated tree nodes that can be expanded and collapsed as needed. Selecting any function updates the **Selection Details** window with further details about that function and performance metrics.

To review the chain of function calls that has the most significant performance bottleneck, right-click any node, and then select **Expand Hottest Branch**.

Working With the Functions View

Use the **Functions** view provided by `gprofng-gui` to diagnose performance bottlenecks in a software program at function level.

The **Functions** view displays a listing of all the functions that were run while performance data was being collected during an experiment.

By default, functions are listed next to their CPU time usage data. Clicking on any function name displays further details in a separate viewing pane.

Double-clicking on a function name opens that function in a new **Source** view, which fills a separate viewing pane with a list of code lines annotated with performance metrics.

Functions can also be reviewed in the **Source/Disassembly** view to see a larger split pane view in the main window between source code and performance data annotations. Computationally expensive lines are highlighted with a different background color.

Working With the Timeline View

Use the **Timeline** view provided by `gprofng-gui` to diagnose performance bottlenecks in a software program that are triggered at a particular stage of execution or build up over time.

The **Timeline** view displays a color-coded timeline map of performance data that was collected while the experiment was running.

The **Time** axis displays the timeline, and on the other axis a series of bars represents each thread that was spawned during the experiment. The call stacks of the application are plotted as vertical bars. Each function or method that's called during the experiment is assigned a different color, and those colors are used in the vertical bars to represent those calls on the timeline.

The OS has its own unique bar at the top of the map area, and its background color indicates whether the application was still running in userspace or not.

Clicking anywhere within those bars highlights a particular point in the timeline, and further details are then displayed in a separate viewing pane.

The scale of the timeline can be changed by clicking on the **Zoom in** and **Zoom out** icons, or by highlighting a specific region in the **Time** axis with the mouse cursor and then pressing the `Enter` key.

Clicking the provided navigation arrows moves the timeline between threads and events within the timeline, and also updates any related viewing panes.

The **Timeline** view can also be customized to display other performance metrics over time, such as CPU usage and processes instead of threads. Use the **Group Data by** pull-down menu to select the performance metric to display.

Click the **Function Colors** button to change the colors that are assigned for any or all the functions that were called during the experiment. If all the functions are set to the same color except one, then it can make tracking the performance of that one specific function through the timeline more straightforward.

The **Timeline** view can also filter performance data so that only the most relevant threads or functions are displayed. Those filter options are provided in the **Tools** menu, or you can use the **Filter** icon within the **Timeline** view to apply a default selection of filters.

8

Known Issues

The following sections describe known issues in the latest gprofng release. For more information, see https://sourceware.org/binutils/wiki/gprofng#Known_Limitations.

Incorrect Source and Disassembly Percentages

The source and disassembly listings display all percentages as zero.

Internal gprofng Function Displayed in Function View

The `collector_root` function might be displayed in function view. That's an internal gprofng function that doesn't consume extra hardware resources.

Inactive gprofng-gui Help Menu Items

Various items aren't clickable in the `gprofng-gui` help menu. They relate to features that haven't been implemented in the graphical user interface.

Unable to Profile a Running Process by Using gprofng-gui

The "Profile a Running Process" feature hasn't been implemented in the graphical user interface, so it's not a clickable option.

For more information about how to create experiments for running programs, see `gprofng` Command Reference.