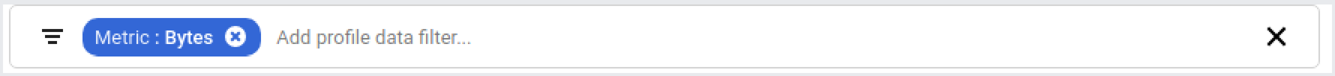
Profiler samples (/profiler/docs/samples) describes several Go language samples stored in a GitHub repository. You
oad, and then run these samples in the Cloud Shell, on Google Cloud, or on a local Linux box. The images on this page
ervice named **`docdemo-service`**. For information on how to generate your own data with the same configuration,
mple **`hotapp`** (/profiler/docs/samples#hotapp).

Stackdriver Profiler lets you add filters to control how the information in the selected profiles is displayed. For example, you can add a filter to hide particular frames or call stacks. Adding and removing filters doesn't change your set of selected profiles.

Each filter is specified by a predefined `FILTER-OPTION` that is paired with a user-defined `VALUE`:

Each filter that you add is displayed on the filter bar. In this example, there is one filter that displays `Metric : Bytes`:



Profiler automatically creates a filter with a `FILTER-OPTION` of `Metric` and a `VALUE` based on your selected profile type. You can change `VALUE` for some profile types. You cannot remove this filter.

To add a filter, use one of the following approaches:

- Click **Filters**, select an option from the list, and then enter the value.
- Click the gray text **Add profile data filter** in the filter bar, and then enter the filter option and value.
- For the **Focus**, **Show from frame**, and **Show stacks** filter options, you can also place your pointer on the frame, and then select the option from the frame tooltip.

To remove a filter, click **Close** on the filter.

When preparing the data to display, Profiler searches for matches between a frame and a filter. When a match occurs Profiler uses the `FILTER-OPTION` to determine what action to execute. A frame *matches* the filter when the frame's function name or the filename of the function's source contains

`VALUE`. A case-sensitive comparison is performed. For example, if the filter is `Hide frames : oo`, then frames with functions named `foo`, `foo1`, and `busyloop` match are hidden from the flame graph.

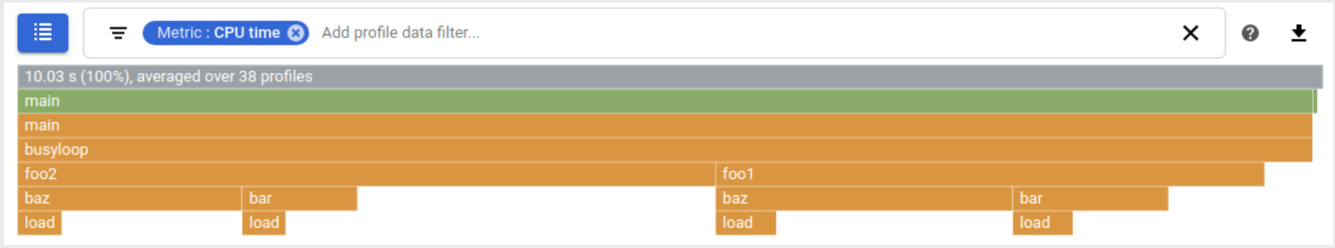
To set the profile type aggregation mode, use the **Metric** filter. For example, if you select a **Heap** profile type, you have the choice between visualizing the data in terms of **Bytes** and **Objects**.

The choices available for the **Metric** filter depend on the programming language and the selected **Profile type**:

- For **CPU time** profiles, the only choice is **CPU time**.
- For **Heap** profiles, the choices are:
  - **Bytes**
  - **Objects**
- For **Allocated Heap** profiles, the choices are:
  - **Total alloc bytes**
  - **Total alloc objects**
- For **Wall time** profiles, the choices are:
  - **Count**
  - **Wall time**
- For **Threads** profiles, the only choice is **Goroutine**.
- For **Contention** profiles, the choices are:
  - **Delay**
  - **Contentions**

For more information about types of profiling metrics, see Profiling concepts (/profiler/docs/concepts-profiling).

For example, the following screenshot shows the CPU consumption of a program:



Here, you can see that the `busyloop` routine calls `foo1` and `foo2`, both of which call various other routines. You can add filters to further restrict the graph only to the data of interest.

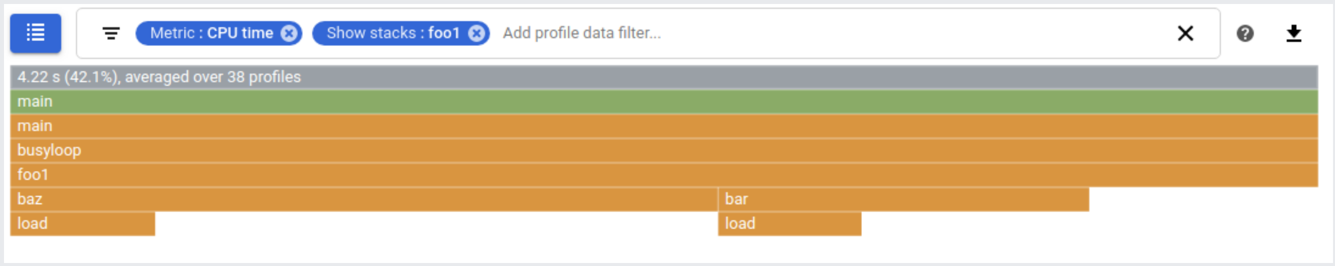
When you use the **Focus** filter, you select a single *function* and the flame graph displays the code paths that flow into, and out of, that specific function.

For details how to focus the graph and interpret the results, see Focusing the graph (/profiler/docs/focusing-profiles).

To display all call stacks that contain a frame that matches the filter value and to hide all other call stacks, use the **Show stacks** filter. The graph shows the callers and callees of the function, that is, everything that calls the matching function, and everything it calls.

This filter performs a case sensitive substring test. A match occurs if the frame function contains the filter value.

To restrict the CPU-usage graph from the previous example to show only the call stacks that involve the function `foo1`, set a **Show stacks** filter for `foo1`:

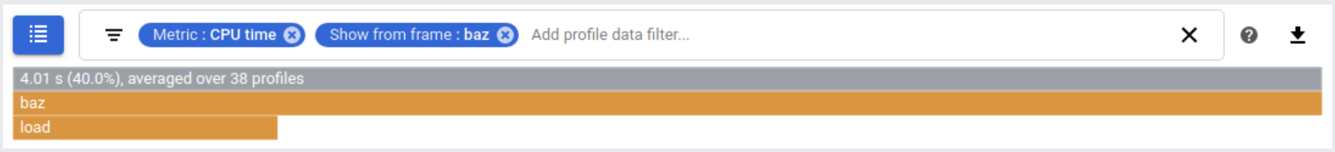


Hide all call stacks that contain a frame that matches the filter value. This filter is often useful when you want to hide uninteresting stacks. For example, with Java applications, adding a `Hide stacks: unsafe.park` filter is common.

This filter performs a case sensitive substring test. A match occurs if the frame function contains the filter value.

To display all call stacks, starting from the frame that matches the filter value and to hide all other call stacks, use the **Show from frame** filter. The resulting graph shows the call stacks from the named function down. This filter is useful if your function is called from many places, and you want to see the total consumption attributable to it.

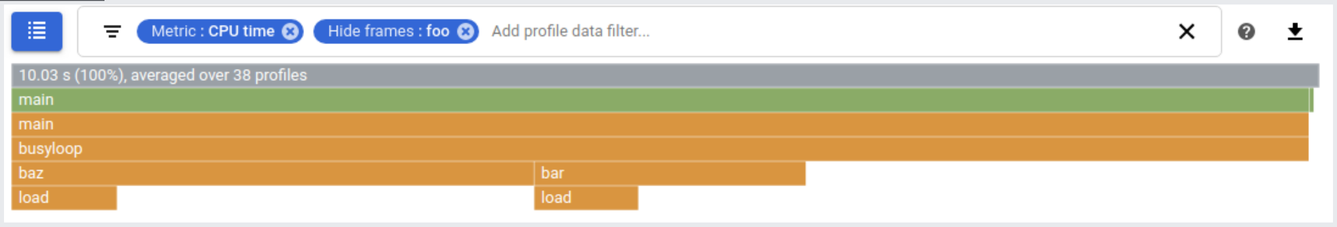
For example, to only show calls originating from the `baz` function, set a **Show from frame** filter for **`baz`**:


Metric : CPU time
Show from frame : baz
Add profile data filter...
4.01 s (40.0%), averaged over 38 profiles
baz
load

This filter performs a case sensitive substring test. A match occurs if the frame function contains the filter value.

To hide from view all frames that match the filter value, use the **Hide frames** filter. The graph shows the callers of the function, and any callees of the function are collected together. This filter is useful for removing irrelevant frames from the graph.

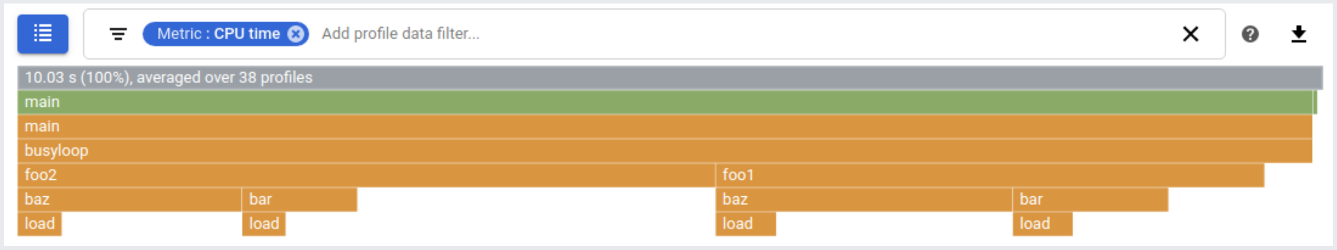
For example, to hide the frames for both `foo1` and `foo2`, set a **Hide frames** filter for `foo`. Both `foo1` and `foo2` match, so both are removed from the graph. Because both of them call the `bar` and `baz` routines, the data for each of those functions is aggregated together.


Metric : CPU time
Hide frames : foo
Add profile data filter...
10.03 s (100%), averaged over 38 profiles
main
main
busyloop
baz
bar
load
load

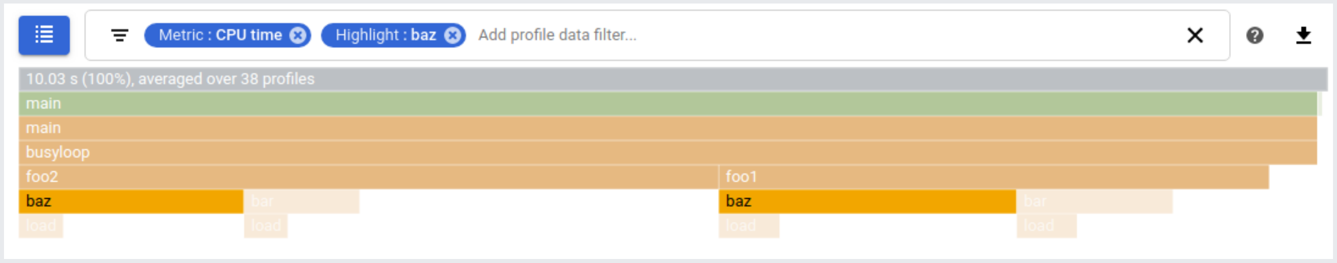
This filter performs a case sensitive substring test. A match occurs if the frame function contains the filter value.

To highlight all frames whose function names match the filter value, use the **Highlight** filter. The function remains in normal color mode, but the call sequences are colored in more subdued tones.

For example, here is a graph with no highlighting:



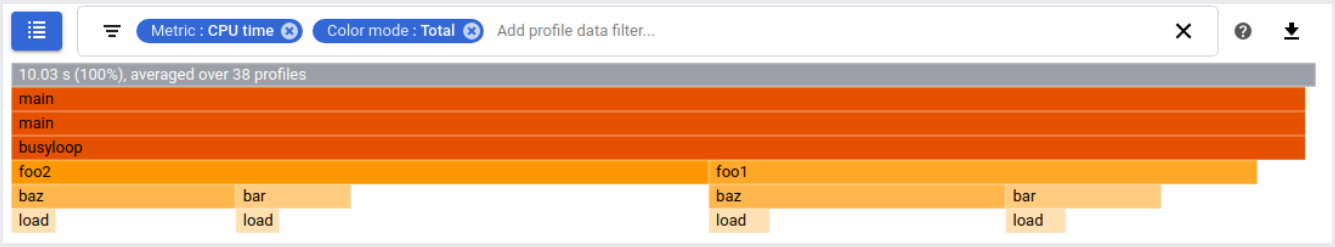
Here's the same graph with highlighting requested for the `baz` function:


Metric : CPU time
Highlight : baz
Add profile data filter...
10.03 s (100%), averaged over 38 profiles
main
main
busyloop
foo2
foo1
baz
baz

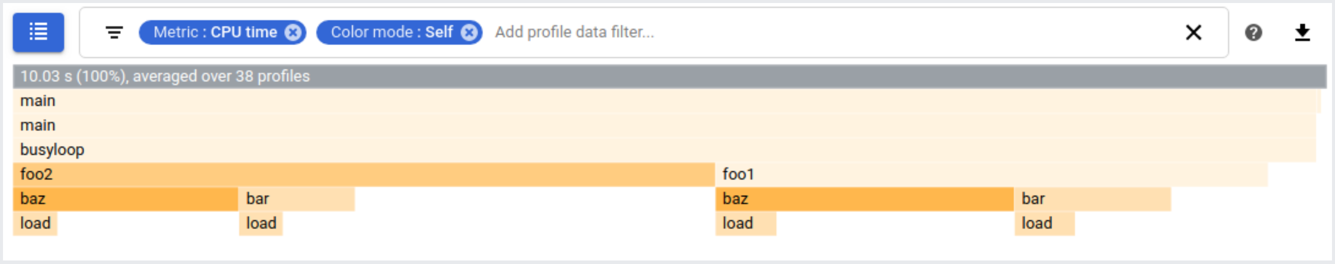
This filter performs a case sensitive substring test. A match occurs if the frame function contains the filter value.

By default, the frame color corresponds, where possible, to the function's package. If package information is unavailable, as with Node.js, the names of the source files are used to color the function blocks. With the default setting, a change in a call stack frame color means a transition from one package to another. The default option corresponds to the **Color mode** filter with a value of **Name**.

To color the frames in the flame graph by the consumption of a function and its children, add a **Color mode** filter with the value of **Total**. If a function is called through multiple call stacks, then the color is determined by the metric consumption for all call stacks. For example, the `main` and `busyloop` are colored red. These two frames consume the most CPU time. The frames labeled `foo2` and `baz` are a deep orange, while the frame labeled `foo1` is a lighter orange. The frames for `bar` and `load` are the lightest. This flame graph illustrates that `foo2` consumes more CPU time than `foo1` but less than `busyloop`:



To color the frames in the flame graph by the function's metric consumption but exclude the metric consumption of its children, add a **Color mode** filter with the value of **Self**. For example, this filter shows that the `baz` function consumes more CPU time than any other function:



- For information on focusing the graph on a single function, see Focusing the graph (/profiler/docs/focusing-profiles).
- For information on comparing profiles collected by different deployments of your service, see Comparing profiles (/profiler/docs/comparing-profiles).
- To learn how to download your profile data, see Downloading profiles (/profiler/docs/downloading-profiles).
- For information on using the Profiler agent to collect profiling data for your services, see:

- Profiling Go applications (/profiler/docs/profiling-go)
- Profiling Java applications (/profiler/docs/profiling-java)
- Profiling Node.js applications (/profiler/docs/profiling-nodejs)
- Profiling Python applications (/profiler/docs/profiling-python)
- Profiling applications running outside Google Cloud (/profiler/docs/profiling-external)