Stackdriver Profiler displays profiling data by using flame graphs. For a conceptual introduction, see Flame graphs (/profiler/docs/concepts-flame).

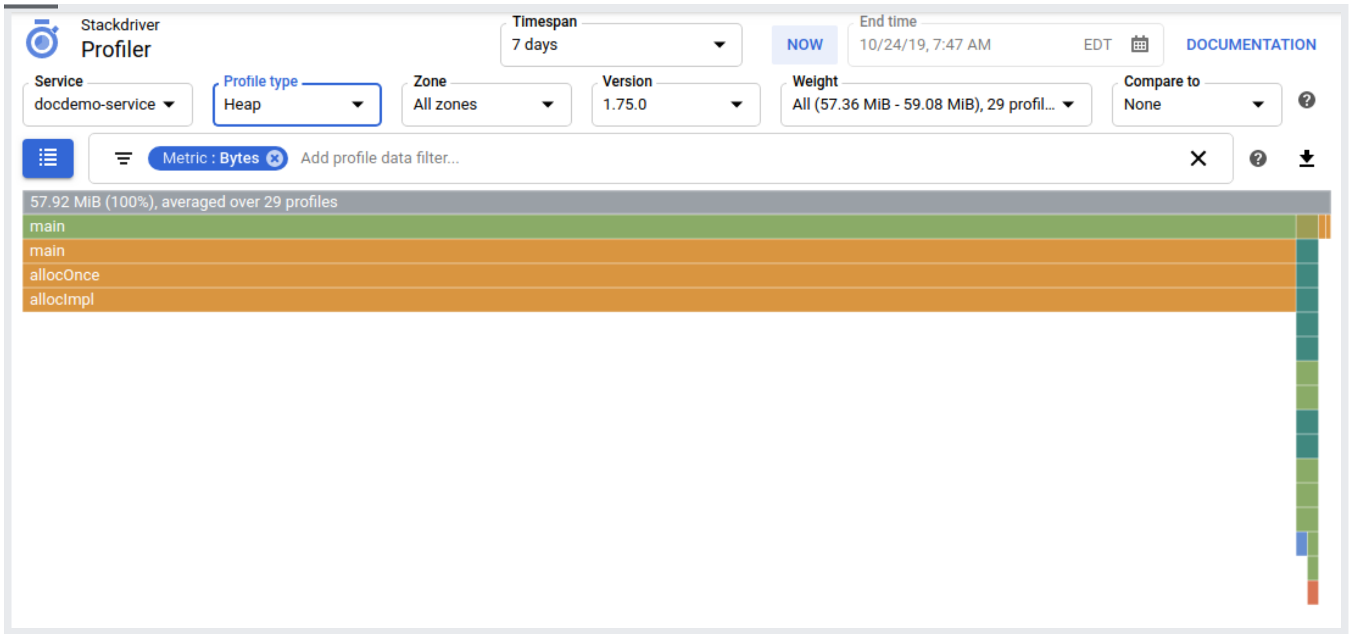
The flame graph is composed of *frames*. Each frame represents a function in the service:

- The width of a frame corresponds to its consumption of the metric being analyzed. The top frame, for example, represents the entire service and consumes 100% of the metric being analyzed. Therefore, this frame is shown as full width.
- 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. In a call stack, a change in block color means a transition from one package to another. You can change change the way the frames are colored. For more information, see Color mode (/profiler/docs/filtering-profiles#color-filter).

The flame graph shows one or more call stacks with the values averaged over a set of profiles:

- Each individual profile represents data collected one time per minute from a single instance of the configured service in a single Compute Engine zone. The collection period for a profile varies with the profile type. See Profile collection (/profiler/docs/concepts-profiling#collection) for more information.
- A maximum of 250 profiles are averaged. If there are more than 250 profiles available, 250 of them are selected randomly as a sample set.
- The top frame, or "root", in a flame graph represents the entire service. Under the "root" frame, is another frame or set of frames making up the second row in the graph. Each of these color-coded frames is a top-level call made by the service. Under each of those colored function frames is another set of function frames, each of which is responsible for some part of the resource of the frame above it. The hierarchy of function frames in the graph represents the call sequence, and the width of a frame represents that function or method's contribution to the resource consumption.

For example, the profiler graph of the consumed heap for the docdemo-service shows that the service calls different top-level frames, one of them is the Go runtime's `main` function. The width of these top-level frames show the majority of the heap is consumed by the call stack involving Go runtime's `main`, application-specific `main`, `allocOnce`, and `allocImpl`:



The root frame displays metric consumption information.

When you hold the pointer over a frame in the flame graph, a frame tooltip opens and displays additional information about the frame:

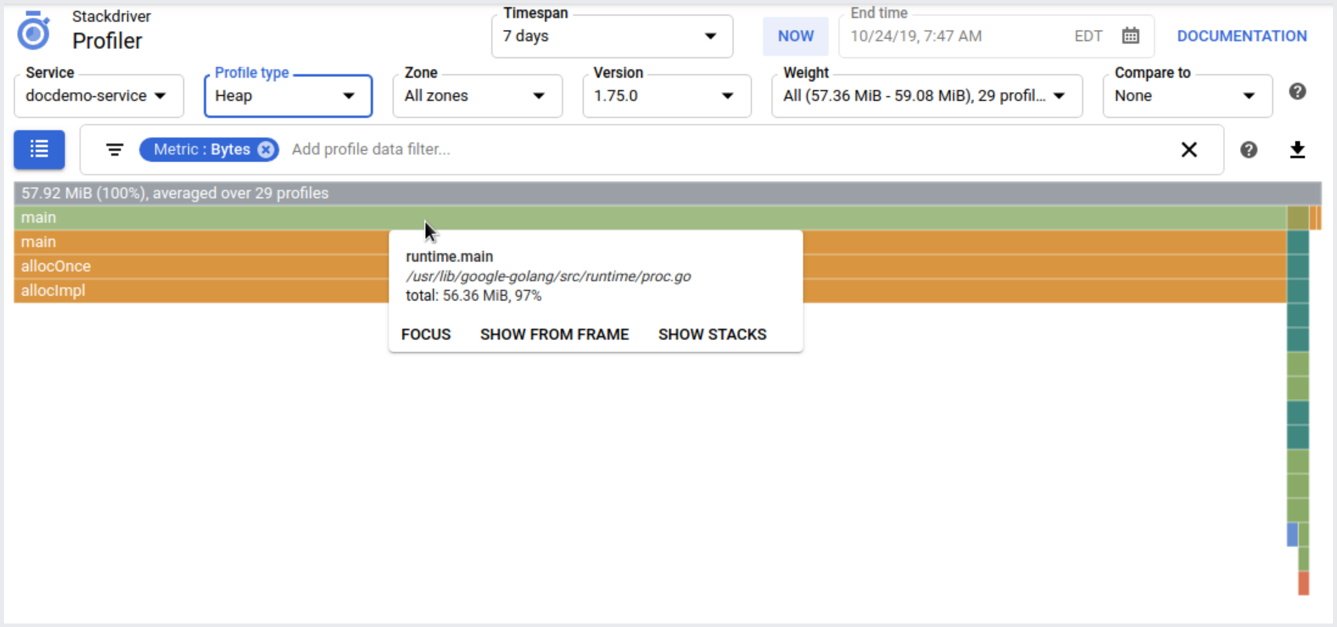
- Function name
- Source file location
- Metric consumption information

In the metric consumption information, the prefix **total** indicates that the metric-value and percentage are for a function and its children. In contrast, the prefix **self** indicates the metric-value and percentage are for the function with the consumption of its children excluded.

The dialog also includes buttons for three filtering actions:

- **Focus** (/profiler/docs/focusing-profiles)
- **Show from frame** (/profiler/docs/filtering-profiles#show-from-frame)
- **Show stacks** (/profiler/docs/filtering-profiles#stack-filters)

In this example, the tooltip shows that the `runtime.main` function is located in the `/usr/local/go/src/runtime/proc.go` file. The metric consumption information shows that `runtime.main` and its children consume 56.36 MiB of memory, or about 97% of the total memory consumed by the program:



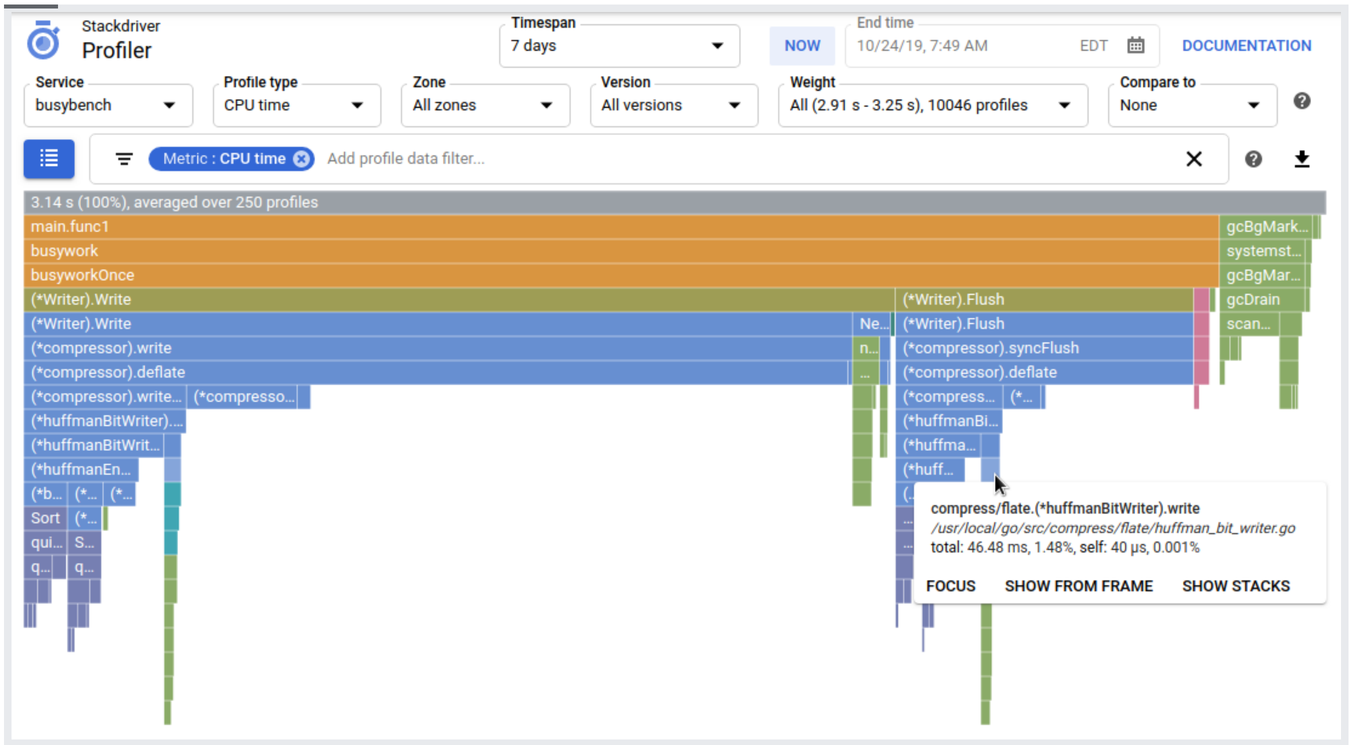
Timespan: 7 days
End time: 10/24/19, 7:47 AM EDT
Service: docdemo-service
Profile type: Heap
Zone: All zones
Version: 1.75.0
Weight: All (57.36 MiB - 59.08 MiB), 29 profil...
Compare to: None
Metric : Bytes
Add profile data filter...

57.92 MiB (100%), averaged over 29 profiles
main
main
allocOnce
allocImpl

runtime.main
/usr/lib/google-golang/src/runtime/proc.go
total: 56.36 MiB, 97%

When you select a frame, the flame graph is redrawn with that frame shown full width and the colors higher in the call stack are muted. Selecting on a frame doesn't change any settings, it only changes how the graph is displayed. To revert to the original view, select the root (top) frame.

In the following example, the pointer is on a tiny frame and it is difficult to view the call stack. The tooltip identifies `(*huffmanBitWriter).write` as the frame function:

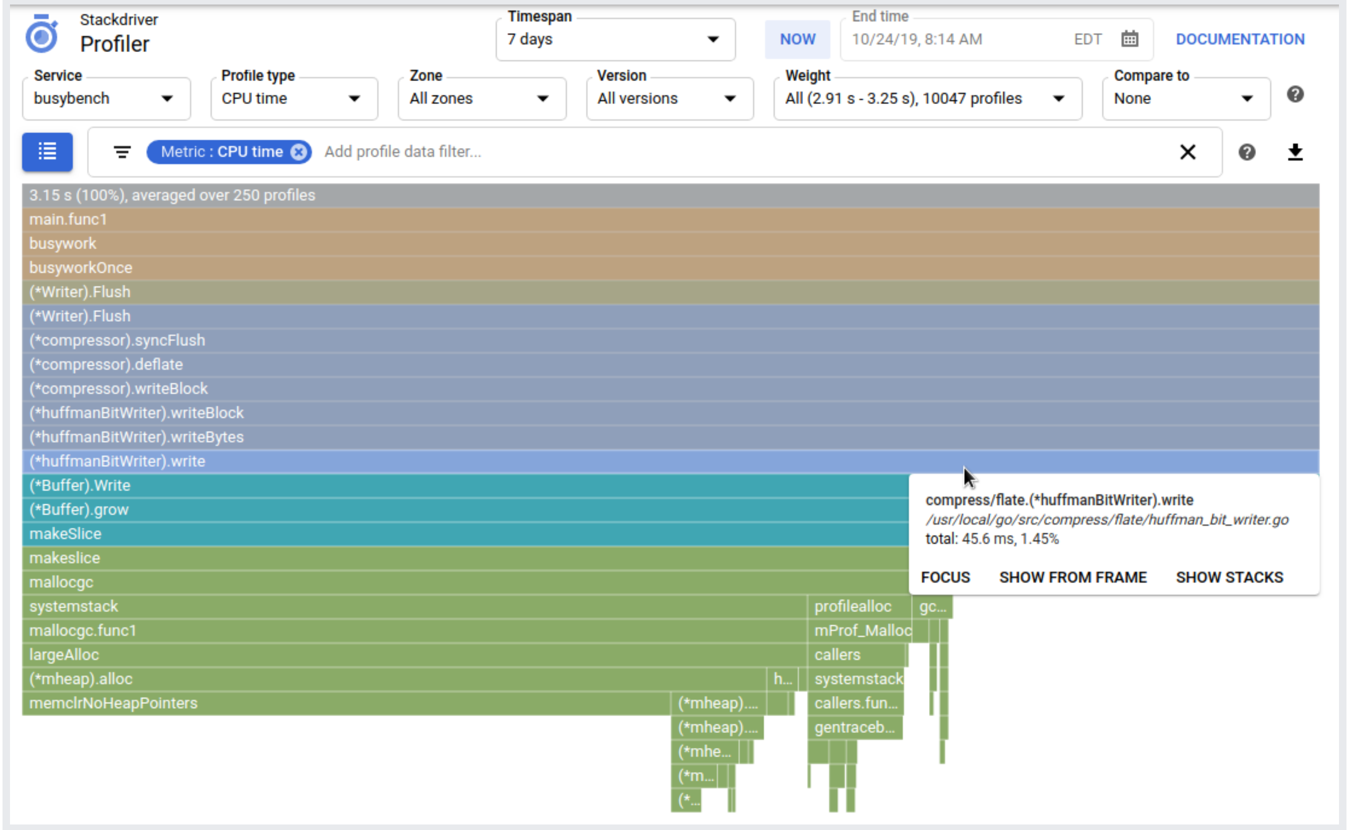


Timespan: 7 days
End time: 10/24/19, 7:49 AM EDT
Service: busybench
Profile type: CPU time
Zone: All zones
Version: All versions
Weight: All (2.91 s - 3.25 s), 10046 profiles
Compare to: None
Metric : CPU time

3.14 s (100%), averaged over 250 profiles

compress/flate.(*huffmanBitWriter).write
/usr/local/go/src/compress/flate/huffman_bit_writer.go
total: 46.48 ms, 1.48%, self: 40 µs, 0.001%

After you select the frame, the graph is redrawn, making the call stack more visible:



Timespan: 7 days
End time: 10/24/19, 8:14 AM EDT
Service: busybench
Profile type: CPU time
Zone: All zones
Version: All versions
Weight: All (2.91 s - 3.25 s), 10047 profiles
Compare to: None
Metric : CPU time

3.15 s (100%), averaged over 250 profiles

compress/flate.(*huffmanBitWriter).write
/usr/local/go/src/compress/flate/huffman_bit_writer.go
total: 45.6 ms, 1.45%

To restore a standard flame graph to the original state, select the root (top) frame. To restore a focused flame graph
l state, select the frame that displays the value of the **Focus** filter. For example, if your filter bar contains `Focus: So`
the frame with the label `Sort`.

- For information on how to suppress, or highlight, frames in the flame graph, see Using filters (/profiler/docs/filtering-profiles).
- 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 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)