

Examining R Profiling Data: The **proftools** Package

Luke Tierney
University of Iowa

Riad Jarjour
University of Iowa

Abstract

This note introduces the **proftools** package for examining data collected by R's sampling profiler. **proftools** includes facilities for summarizing results at the function, call, and source line level; for filtering to narrow the focus to functions of primary interest; and for visualizing profiling data. Use of the package is illustrated with a small running example.

Keywords: **proftools**, profiling, R, **Rprof**.

1. Introduction

Profiling is a program analysis method for determining where a program run spends most of its execution time, and can be very helpful in guiding programmer effort for improving program performance. R includes a sampling based profiling mechanism that records information about calls on the stack at specified time intervals. If available, information about the specific source code lines active at the sampling point is recorded as well. Information about time spent in the garbage collector can also be collected. The collected profiling data is written to a file, by default the file **Rprof.out** in the current working directory. The function **summaryRprof**, available in the **base** package in R, provides a simple interface for examining this data. The **profvis** package ([Chang and Luraschi 2018](#)) provides a visualization and another representation of the data. The **proftools** package provides a much more extensive set of tools for summarizing, visualizing, and filtering this data.

2. Collecting Profile Data

The **proftools** package includes an example script **bootlm.R** in the **samples** directory that runs several examples from the **boot** package ([Canty and Ripley 2017](#)) and then fits a simple linear model. The file path can be obtained as

```
srcfile <- system.file("samples", "bootlmEx.R", package = "proftools")
```

The traditional way to collect profiling data in R is to call **Rprof** to start profiling, run the code to be profiled, and then call **Rprof** again to end profiling. For example, to profile the code in **bootlm.R**, and collect both source line and GC information, you could use

```
profout <- tempfile()
Rprof(file = profout, gc.profiling = TRUE, line.profiling = TRUE)
```

```
source(srcfile)
Rprof(NULL)
pd <- readProfileData(profout)
unlink(profout)
```

The **proftools** package provides the simpler alternative

```
pd <- profileExpr(source(srcfile))
```

By default **profileExpr** enables GC and source information to be collected. It also trims off stack information leading up to the **profileExpr** call.

RStudio (RStudio Team 2015) provides a **Profile** menu for collecting profile data. The data are stored in a cache directory. The function **readRStudioProfileCacheData** returns the data for the most recent profile run in the cache, or **NULL** if none is available.

3. Summary Functions

The most basic summary function is **funSummary** for summarizing profile results at the function level. It produces information similar to the result returned by R's **summaryRprof** but in a more usable form:

```
head(funSummary(pd), 10)
```

##	total.pct	gc.pct	self.pct	gcself.pct
## withVisible	100	11.3	0.00	0
## source	100	11.3	0.00	0
## eval	100	11.3	3.77	0
## statistic	81	6.6	0.94	0
## lapply	81	6.6	0.00	0
## FUN	81	6.6	1.89	0
## glm (bootlmEx.R:33)	47	2.8	0.94	0
## boot (bootlmEx.R:44)	38	2.8	0.00	0
## boot (bootlmEx.R:39)	38	1.9	0.00	0
## model.frame.default	31	4.7	2.83	0

The result returned by **funSummary**, and most summary functions, is a data frame, so **head** can be used to focus on the top entries.

By default, when source information is available results are summarized at the call level, so multiple calls to **boot** from different source lines are shown separately. This can be suppressed by supplying the **srcLines = FALSE** argument:

```
head(funSummary(pd, srcLines = FALSE), 10)
```

##	total.pct	gc.pct	self.pct	gcself.pct
## withVisible	100	11.3	0.0	0.0
## source	100	11.3	0.0	0.0
## eval	100	11.3	47.2	0.0
## boot	82	7.5	4.7	4.7

## statistic	81	6.6	7.5	0.0
## lapply	81	6.6	0.0	0.0
## FUN	81	6.6	27.4	0.0
## glm	47	2.8	8.5	0.0
## model.frame.default	31	4.7	36.8	0.0
## predict	27	1.9	8.5	0.0

Data can also be summarized by call:

```
head(callSummary(pd), 10)
```

	total.pct	gc.pct	self.pct	gcself.pct
## withVisible -> eval	100	11.3	0.00	0
## source -> withVisible	100	11.3	0.00	0
## eval -> eval	100	11.3	0.94	0
## lapply -> FUN	81	6.6	0.00	0
## FUN -> statistic	81	6.6	0.94	0
## statistic -> glm (bootlmEx.R:33)	47	2.8	0.94	0
## eval -> boot (bootlmEx.R:44)	38	2.8	0.00	0
## eval -> boot (bootlmEx.R:39)	38	1.9	0.00	0
## boot (bootlmEx.R:44) -> lapply	38	2.8	0.00	0
## boot (bootlmEx.R:39) -> lapply	38	1.9	0.00	0

When source information is available in the profile data the `srcSummary` function can be used to summarize at the source line level; only lines appearing in the sample are included:

```
srcSummary(pd)
```

	total.pct	gctotal.pct	source
## bootlmEx.R:8	3.77	0.94	gp1 <- 1:table(as.numeric(d\$seri ...
## bootlmEx.R:10	0.94	0.00	m2 <- sum(d[-gp1,1] * f[-gp1])/s ...
## bootlmEx.R:12	0.94	0.94	ss2 <- sum(d[-gp1,1]^2 * f[-gp1] ...
## bootlmEx.R:16	6.60	2.83	boot(grav1, diff.means, R = 999, sty ...
## bootlmEx.R:33	47.17	2.83	lm.b <- glm(fit+resid[inds] ~ da ...
## bootlmEx.R:35	27.36	1.89	pred.b <- predict(lm.b, x.pred)
## bootlmEx.R:36	0.94	0.00	c(coef(lm.b), pred.b - (fit.pred ...
## bootlmEx.R:39	37.74	1.89	nuke.boot <- boot(nuke.data, nuke.fu ...
## bootlmEx.R:44	37.74	2.83	nuke.boot <- boot(nuke.data, nuke.fu ...
## bootlmEx.R:50	6.60	0.94	X <- matrix(rnorm(n * p), n, p)
## bootlmEx.R:51	0.94	0.00	y <- rnorm(n) + X[,1]
## bootlmEx.R:52	10.38	2.83	fit <- lm(y ~ X)

The function `annotateSource` can show the full files with profiling annotations.

A useful way to examine profile data is to look for hot execution paths. This approach sorts functions called at top level, i.e. at the bottom of the call stack, by the total amount of time spent in their top level calls; within each top level call to a function `f` the functions called by `f` are sorted by the amount of time spent in them within the top level call to `f`; and the process continues for higher level calls. The function `hotPaths` produces a hot path summary; the `total.pct` argument causes leaf functions in stack traces to be pruned back until the execution time percentage for each stack trace is at least `total.pct`:

```
hotPaths(pd, total.pct = 10.0)

## path                                total.pct self.pct
## source                             100.00     0.00
## . withVisible                       100.00     0.00
## . . eval                            100.00     0.00
## . . . eval                          100.00     0.00
## . . . . boot (bootlmEx.R:39)         37.74     0.00
## . . . . . lapply                     37.74     0.00
## . . . . . FUN                        37.74     0.00
## . . . . . . statistic                 37.74     0.00
## . . . . . . . glm (bootlmEx.R:33)    20.75     0.00
## . . . . . . . eval                   12.26    12.26
## . . . . . . . predict (bootlmEx.R:35) 16.04     0.00
## . . . . . . . predict.glm            16.04    16.04
## . . . . boot (bootlmEx.R:44)         37.74     0.00
## . . . . . lapply                     37.74     0.00
## . . . . . FUN                        37.74     0.00
## . . . . . . statistic                 37.74     0.00
## . . . . . . . glm (bootlmEx.R:33)    26.42     0.94
## . . . . . . . eval                   17.92    17.92
## . . . . . . . predict (bootlmEx.R:35) 11.32     0.94
## . . . . . . . predict.glm            10.38    10.38
## . . . . lm (bootlmEx.R:52)           10.38     0.00
```

Examining the result of `hotPaths` starting with high values of `total.pct` and then moving to lower values is a useful way to explore where the computational effort is concentrated. An alternative for limiting the depth to which stack traces are followed is provided by the `maxdepth` argument.

4. Filtering Profile Data

The hot path summary shows information associated with the source command itself that is not directly relevant to our analysis. The `filterProfileData` function can be used to select or omit certain functions, drop functions with small self or total times, narrow to a particular time interval, among others. For example, by selecting only stack traces that include calls to `withVisible` and then trimming off the leading four calls we can focus just on the work done in the sourced file:

```
filteredPD <- filterProfileData(pd, select = "withVisible", skip = 4)
```

The hot path summary for this reduced profile is

```
hotPaths(filteredPD, total.pct = 10.0)

## path                                total.pct self.pct
## boot (bootlmEx.R:39)                 37.74     0.00
## . lapply                             37.74     0.00
## . . FUN                              37.74     0.00
## . . . statistic                       37.74     0.00
```

```
## . . . . glm (bootlmEx.R:33)      20.75      0.00
## . . . . . eval                  12.26      0.94
## . . . . . . eval                 11.32      0.00
## . . . . predict (bootlmEx.R:35) 16.04      0.00
## . . . . . predict.glm           16.04      0.00
## . . . . . . predict.lm          15.09      0.00
## boot (bootlmEx.R:44)             37.74      0.00
## . lapply                         37.74      0.00
## . . FUN                         37.74      0.00
## . . . statistic                 37.74      0.00
## . . . . glm (bootlmEx.R:33)      26.42      0.94
## . . . . . eval                  17.92      0.00
## . . . . . . eval                 17.92      0.00
## . . . . predict (bootlmEx.R:35) 11.32      0.94
## . . . . . predict.glm           10.38      0.00
## lm (bootlmEx.R:52)              10.38      0.00
```

We can use the `focus` filter to further narrow our examination to stack frames containing calls to `glm` and also remove all calls preceding the first `glm` call from the selected stack frames. For this reduced data it also makes sense to follow the paths further by lowering `total.pct` to 5%:

```
glmPD <- filterProfileData(filteredPD, focus = "glm")
hotPaths(glmPD, total.pct = 5.0)

## path                                total.pct self.pct
## glm (bootlmEx.R:33)                 47.17      0.94
## . eval                             30.19      0.94
## . . eval                           29.25      0.00
## . . . glm.fit                       12.26      9.43
## . . . stats::model.frame            16.98      0.94
## . . . . model.frame.default         16.04      1.89
## . . . . . External2                 5.66      0.00
## . . . . . . na.omit                 5.66      0.00
## . . . . . . . na.omit.data.frame    5.66      0.94
## . model.matrix                     6.60      0.00
## . . model.matrix.default            6.60      0.94
## . structure                         6.60      0.00
## . . .getXlevels                    6.60      0.00
## . . . vapply                        5.66      0.00
```

5. Visualizing Profile Data

Call graphs annotated with profile information are a very popular way to view profiling results. `plotProfileCallGraph` uses the **graph** (Gentleman, Whalen, Huber, and Falcon 2017) and **Rgraphviz** (Hansen, Gentry, Long, Gentleman, Falcon, Hahne, and Sarkar 2017) packages from Bioconductor to render an annotated call graph. The default style for the graph is based on the style used in **Google's profiling library** but can be customized in a number of ways. A call graph for the full profile data is produced by

```
plotProfileCallGraph(pd)
```

and is shown in Figure 1.

By default the call graph size is limited to at most 30 nodes; nodes with lower total hit counts are dropped. The `maxnodes` argument can be used to adjust this limit. Also by default color is used to encode the total hit score for the function nodes. This can be suppressed with the `score = "none"` argument.

We can obtain a more readable graph by filtering. For example, to examine the `glm.fit` call and its callees we can use

```
plotProfileCallGraph(filterProfileData(pd, focus = "glm.fit"))
```

The result is shown in Figure 2.

A printed version of the call graph, similar to the call graph produced by `gprof` (Graham, Kessler, and Mckusick 1982), can be obtained with `printProfileCallGraph`. For the sub-graph of the `glm.fit` calls, for example,

```
printProfileCallGraph(filterProfileData(pd, focus = "glm.fit"))
```

produces the printed representation shown in Figure 3.

Another visualization sometimes used is a **flame graph**. A flame graph for the original data is produced by

```
flameGraph(pd)
```

The result is shown in Figure 4. A flame graph of the filtered data is produced by

```
flameGraph(filteredPD)
```

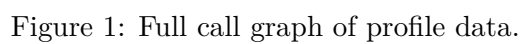
and shown in Figure 5

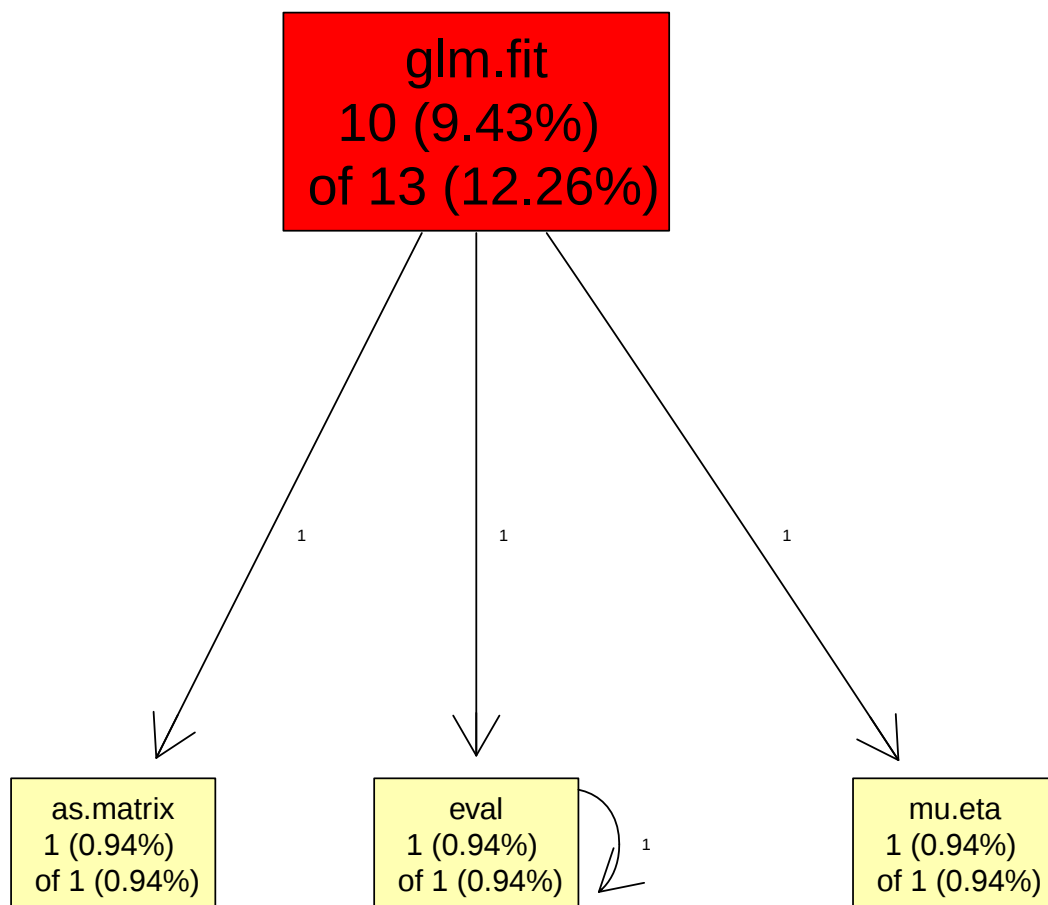
The vertical positions of rectangles in these flame graphs represent call depth on the stack. The widths of the rectangles represent the amount of time spent in a call at a particular call or set of calls at a particular depth. The `order` argument determines the ordering of call rectangles at a particular level within a call at the lower level. The default order is `"hot"`; it uses the hot path ordering with the call with the largest amount of time first. This produces a visual representation of the hot path summary. The `"alpha"` ordering orders the calls alphabetically. Specifying `order = "time"` shows the calls in the order in which they occurred:

```
flameGraph(pd, order = "time")
```

Figure 6 shows the `boot` calls preceding the data generation for the `lm` call and the `lm` call itself.

Default colors for flame graphs are based on the `rainbow` palette. Alternative colors can be specified by providing a `colormap` function. If space permits, the call labels are printed within the rectangles representing the calls. A method for the `identify` function is available that can be used to identify the individual calls when they are not visible. An option to this method is to request that all instances of the identified calls be outlined on the plot. The `identify` method uses the value returned by `flameGraph`; thus it would be used as



Figure 2: Call graph for `glm.fit` call.

Call graph

index	% time	% self	% children	name
[1]	12.26	9.43	2.83	glm.fit [1]
		0.94	0.00	as.matrix [2]
		0.00	0.94	eval [3]
		0.94	0.00	mu.eta [4]

[2]	0.94	0.94	0.00	glm.fit [1]
		0.94	0.00	as.matrix [2]

[3]	0.94	0.00	0.94	glm.fit [1]
		0.94	0.00	eval [3]
		0.94	0.00	eval [3]
		0.94	0.00	eval [3]

[4]	0.94	0.94	0.00	glm.fit [1]
		0.94	0.00	mu.eta [4]

Figure 3: Printed call graph.

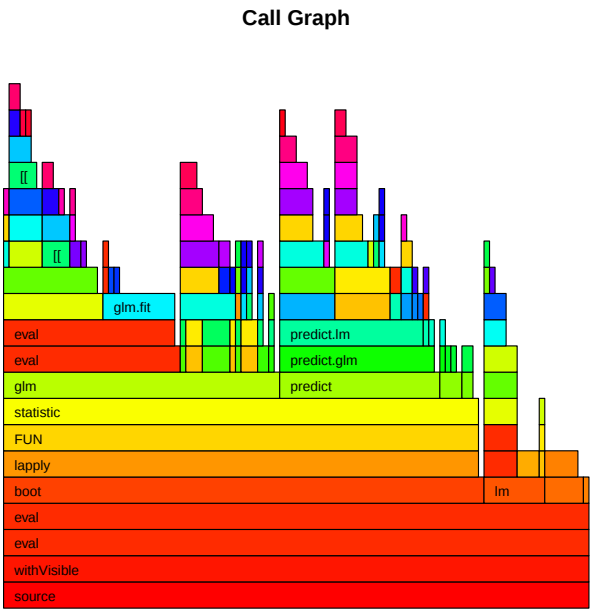


Figure 4: Flame graph visualizing hot paths for the full profile data.

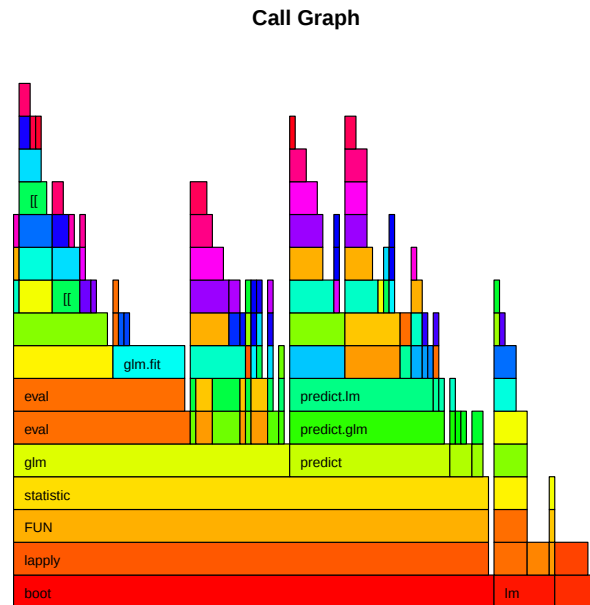


Figure 5: Flame graph of the filtered profile data.

```
fg <- flameGraph(pd)
identify(fg)
```

A third visualization that is sometimes used is a callee tree map. This is produced by

```
calleeTreeMap(pd)
```

and shown in Figure 7. A callee tree map shows a tree map (Shneiderman 1998) of the calls in the call tree. The tiling algorithm used depends on the `squarify` argument. If `squarify` is `TRUE` then the *squarified* algorithm (Bruls, Huizing, and van Wijk 2000) is used; otherwise, the longer side is partitioned. Again a method for `identify` is provided for `calleeTreeMap` objects. Clicking on rectangles in the plot returns a list of the call stacks for the identified rectangles.

6. Graphical User Interfaces

The function `writeCallgrindFile` can be used to write the profile data in Valgrind's `callgrind` format for use with the `kcachegrind` or `qcachegrind` graphical user interfaces available on Linux and Mac OS X. Figure 8 shows the `kcachegrind` interface for the example profile data written out using `writeCallgrindFile`. By default `writeCallgrindFile` assumes the common case where, as here, the profiled code is in a file run using `source` and source code

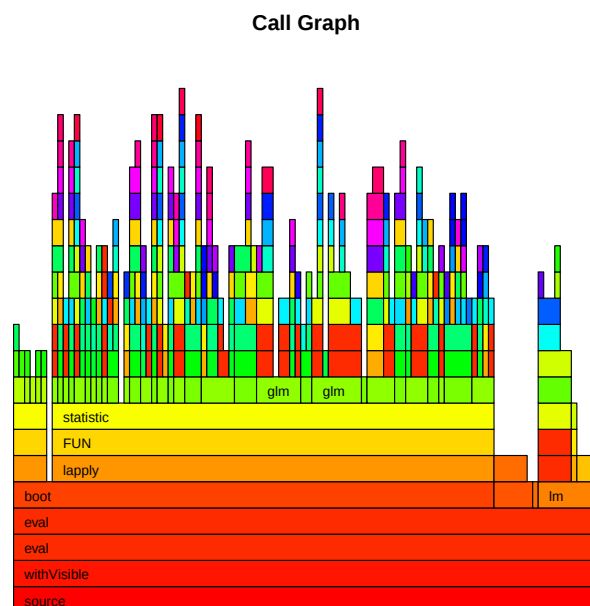


Figure 6: Time graph of the full profile data.

information is retained. If this is the case, `writeCallgrindFile` removes calls associated with the `source` call and adds a `<TOP>` entry; this persuades `kcachegrind` to show the top level line information.

Graphical user interfaces within R will be made available in the **proftoolsGUI** package. The current development version provides two interfaces, one based on **gWidgets2** (Verzani 2016) and one on **shiny** (Chang, Cheng, Allaire, Xie, and McPherson 2018). Figure 9 shows the shiny interface at the time of writing.

References

- Bruls M, Huizing K, van Wijk J (2000). “Squarified Treemaps.” In W de Leeuw, R van Liere (eds.), *Data Visualization 2000*, Eurographics, pp. 33–42. Springer Vienna. ISBN 978-3-211-83515-9. doi:10.1007/978-3-7091-6783-0_4. URL http://dx.doi.org/10.1007/978-3-7091-6783-0_4.
- Canty A, Ripley BD (2017). *boot: Bootstrap R (S-Plus) Functions*. R package version 1.3-20.
- Chang W, Cheng J, Allaire J, Xie Y, McPherson J (2018). *shiny: Web Application Framework for R*. R package version 1.1.0, URL <https://CRAN.R-project.org/package=shiny>.
- Chang W, Luraschi J (2018). *profvis: Interactive Visualizations for Profiling R Code*. R package version 0.3.5, URL <https://CRAN.R-project.org/package=profvis>.

Callee Tree Map

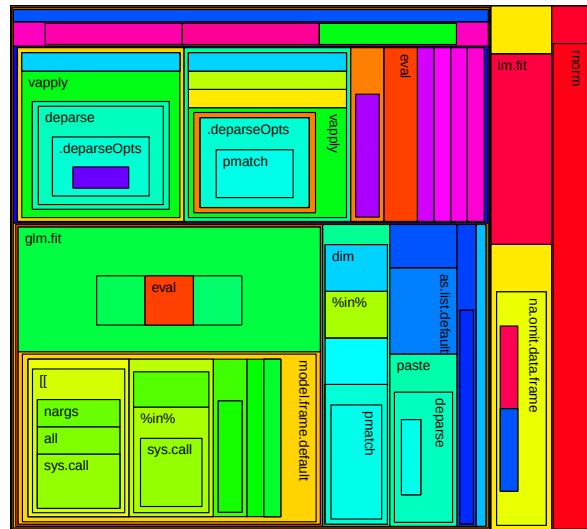


Figure 7: Call tree map of the full profile data.

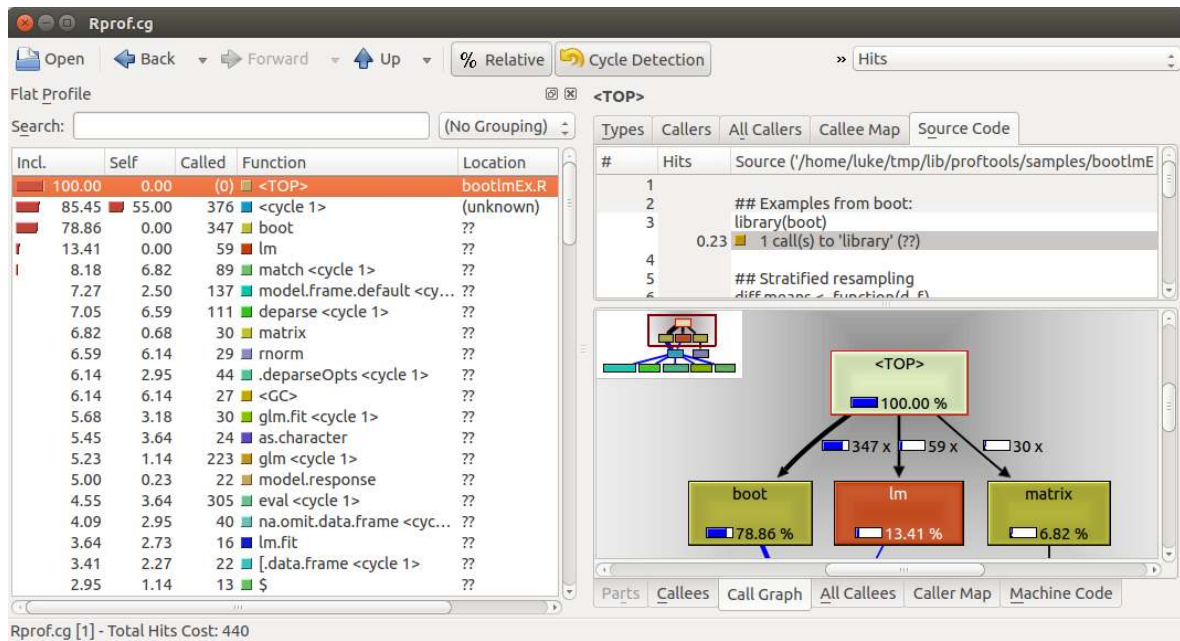


Figure 8: kcacheGrind interface for examining profile data for the example.

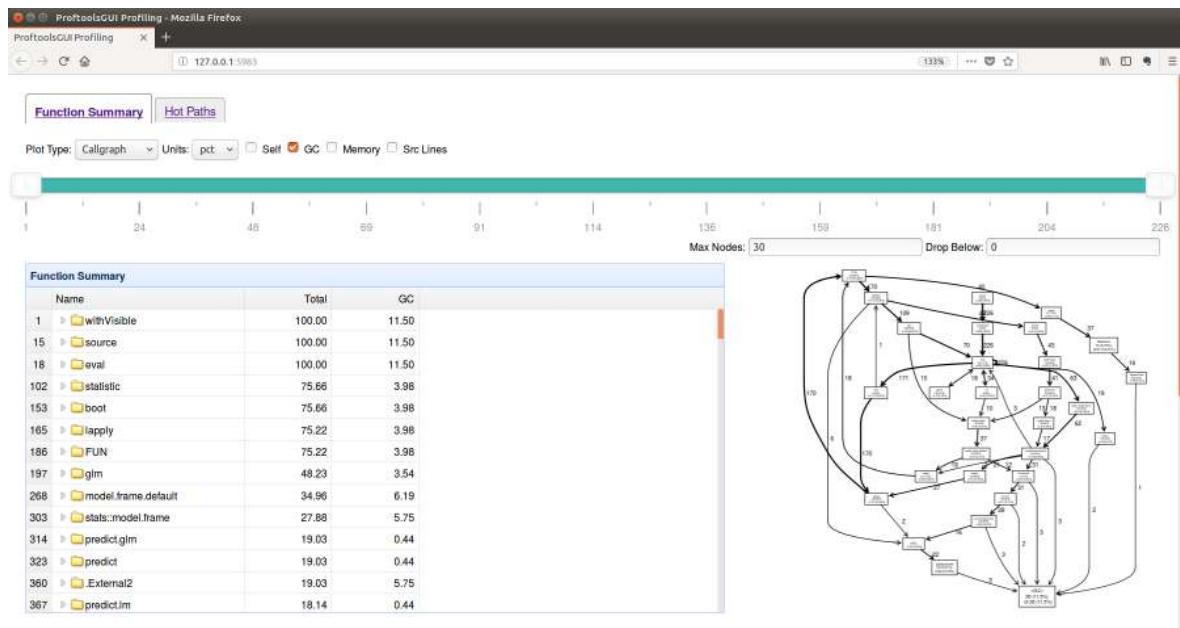


Figure 9: proftoolsGUI shiny interface for examining profile data for the example.

Gentleman R, Whalen E, Huber W, Falcon S (2017). *graph: graph: A package to handle graph data structures*. R package version 1.54.0.

Graham SL, Kessler PB, Mckusick MK (1982). “Gprof: A Call Graph Execution Profiler.” In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN ’82, pp. 120–126. ACM, New York, NY, USA. ISBN 0-89791-074-5. doi:10.1145/800230.806987. URL <http://doi.acm.org/10.1145/800230.806987>.

Hansen KD, Gentry J, Long L, Gentleman R, Falcon S, Hahne F, Sarkar D (2017). *Rgraphviz: Provides plotting capabilities for R graph objects*. R package version 2.20.0.

RStudio Team (2015). *RStudio: Integrated Development Environment for R*. RStudio, Inc., Boston, MA. URL <https://posit.co/>.

Shneiderman B (1998). “Treemaps for space-constrained visualization of hierarchies.” Human-Computer interaction lab University of Maryland. Available at: <http://www.cs.umd.edu/hcil/treemap-history/>. Accessed November 20, 2004.

Verzani J (2016). *gWidgets2: Rewrite of gWidgets API for Simplified GUI Construction*. R package version 1.0-7, URL <https://CRAN.R-project.org/package=gWidgets2>.

Affiliation:

Luke Tierney

Department of Statistics and Actuarial Science

Faculty of Statistics

University of Iowa

Iowa City, USA

E-mail: luke-tierney@uiowa.edu

URL: <https://homepage.stat.uiowa.edu/~luke/>

Riad Jarjour

Department of Statistics and Actuarial Science

Faculty of Statistics

University of Iowa

Iowa City, USA

E-mail: riad-jarjour@uiowa.edu