Visualisations for Function Time Profiles

Category:

1 Introduction

Modern program profilers record a wealth of performance information such as execution time, resource allocations, lock contentions, thread sleep and wakeup events, and more. Some of the most important information is the time used per-function along with the context of these functions' invocations. This information is critical to understand which 'areas' in a program are consuming the most time, and consequently which should be focused upon to reduce the runtime of a program.

While there are a number of visualisations for various performance aspects, such as thread wakefulness over time, network activity, and so on, there seems to be very few visualisations specialised for visualising per-function time while maintaining invocation context.

We present minor improvements to one of these existing visualisation (flame graphs [2]), as well as a new related visualisation, funky graphs, which complement flame-graphs by focusing on the aggregated cost of entire functions rather than the call-stacks.

Section section 2 details the structure of time profiling data, describes some transformations that can be applied to reduce the data set prior to visualisation, and underlines an important particularity of this type of data. Section section 3 describes an existing visualisation, flame graphs, which partially satisfies our requirements. Section section 4 identifies issues with flame graphs, describes our attempts to resolve these shortcomings, and presents results of these changes. Section section 5 introduces a new visualisation, named funky graphs, which are intended to complement flame graphs by providing an alternative view of the same data. Section section 6 discusses general issues discovered during implementation, and section section 7 presents our conclusions.

2 Background

There are many methods to record the execution time of a program, such as program instrumentation and periodic sampling. We limit our consideration to those producing a set of call-stacks and the amount of time spent in each of these call-stacks. A call-stack is defined to a sequence of frames, where each frame is an invocation of a function.

These call-stacks and their associated time value (their 'weight') can be reduced to a Rose Tree (a N-child tree where each node is associated with a datum) where each node in the tree corresponds to a frame, and the datum of that node is the amount of time recorded executing that particular call-stack. This reduction to a Rose Tree does not reduce the amount of data that will be displayed, but it exposes the hierarchical nature of the data set and it simplifies further reductions.

One such reduction is folding the frames of recursive functions. It is simple, effective, and nearly lossless information wise (one is rarely uninterested in seeing a deep call-stack from a merge-sort function).

Closely related to this, however, is an important complication: A function can be indirectly recursive. An example of this is a function `a` occasionally calling function `b`, which in turn calls `a`. Should we fold the child frame of function `a` (and consequently its child frames) into its parent frame? On one hand we wish to fold any recursion we can to simplify the Rose Tree; on the other hand this would erase possibly important contextual information. The decision ultimately lies in whether we are interested in the actual frame sequence (i.e. the call-stack) for our contextual information, or if we desire a 'bigger picture' view and are content with having entire functions as our contextual information.

This decision, whether to fold indirectly recursive function frames to their top-most frame, is one of the two primary differences between flame graphs and funky graphs: Flame graphs do not perform this lossy fold; funky graphs do.

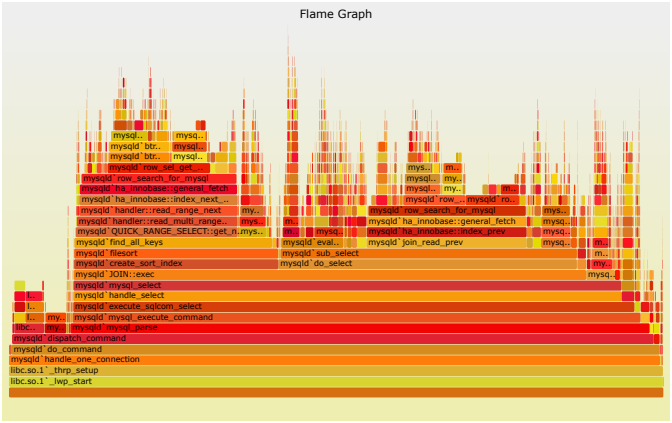


Fig. 1: Canonical example of a flame graph, by Gregg. [2]

Fig. 2: A close-up look at a flame graph node and its immediate children. The width of a node is equal to its inclusive weight. Child nodes are placed above their parent, and space proportional to the parent's exclusive time is reserved on the left. (Subset of flame graph in Figure Figure 1.)

3 Classic Flame Graphs

Flame graphs were created by Brendan Gregg as a solution for visualising DTrace, a system-wide profiler, output. [2] Flame graphs can be described as a hybrid of icicle plots [4] and sunbursts [7], inheriting the stacked linear layout of icicle plots and sunbursts' size-partitioning of child nodes. A canonical example of a flame graph can be found in Figure Figure 1.

The hierarchical structure of a flame graph is effectively a 1-to-1 representation of a Rose Tree, where the visual width of a node is equal to its inclusive weight. Child nodes are sorted by name, and placed above their parents. A concrete example of this can be seen in Figure Figure 2).

Nodes are coloured by using a hash of their function's name as an index into a palette. This is intended both to aid in distinguishing adjacent nodes, and to assign a single colour to all nodes belonging to the same function.

The original implementation of flame graphs offered tool-tips which reported the exact total weight of the hovered node and the percentage of that weight against the total. Later implementations offered additional features such as click-to-zoom [3], and searching by name [5]. (Figure Figure 3.)

Flame graphs offer a compelling visualisation. They effectively convey the data's hierarchy (the context of a call-stack), and make efficient use of space. The use of length to encode node weight allows accurate estimation of quantitative information against a common scale (the width of graph) [9] [12], as well as pre-attentive identification of heavier nodes [9]. Finally, flame graphs offer a few quick and light-

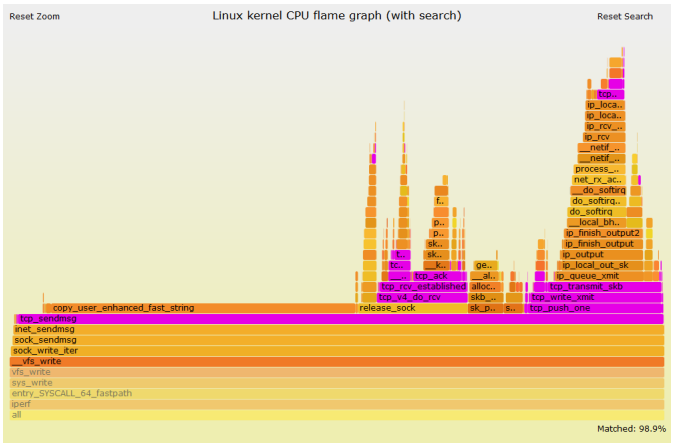
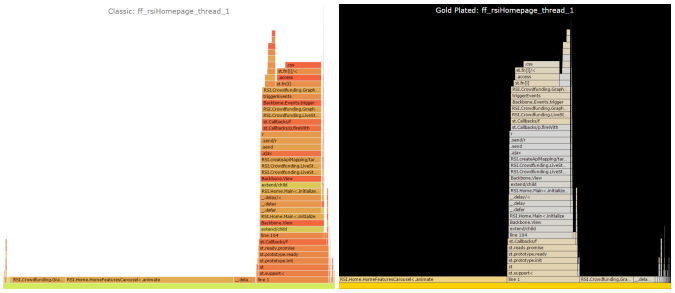


Fig. 3: An example of a zoomed flame graph presenting search results for functions matching "tcp_". Matching nodes have been coloured purple, allowing pre-attentive identification. The graph has also been 'zoomed' onto `__vfs_write`, which has been resized to the full width of the view port. The parent nodes of `__vfs_write` have been faded, but are still displayed to provide context. (Flame-graph by Gregg. [5])



(a) Spier's implementation in D3. [6]

(b) Our 'gold plated' flame graphs.

Fig. 4: A overview comparison of classic flame graphs, and our 'gold plated' flame graphs.

weight epistemic queries in the form of tool-tips (hover-queries) and click-to-zoom functionality. [8]

4 Gold Plating Flame Graphs

As compelling as they are, flame graphs have room for improvement. They suffer from minor visual design flaws, and they do a terrible job at conveying any kind of function-level aggregated data. This last is particularly disappointing as the identification of heavy functions is a fundamental task for performance analysis.

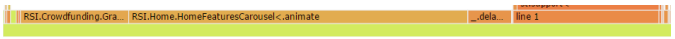
Spier's implementation of flame graphs [6] [1] was selected as a base upon which to experiment, allowing us to quickly iterate through possible improvements and modification to the original flame graph design. Flame graphs featuring our modifications are here-on referred to as 'gold plated'.

4.1 Flame Graph Flaws

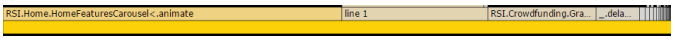
There are two main areas in which the visual design of flame graphs leaves to be desired: Sorting sibling nodes by name, and misuse of colour.

4.1.1 Sort by Name

Flame graphs sort sibling nodes by name, which can result in very light siblings often getting visually lost among their heavier brothers. While unimportant when only a few siblings get lost this way, this can become significant if the program exhibits a 'long tail' where there are many tiny siblings summing up to a notable amount (5% or more). (Figure Figure 5.)



(a) Sort by name, colour by mapping hash of function name to vector in RGB colour space. [6]



(b) Sort by weight, colour by mapping $min(\frac{weight_{function}}{weight_{zoomRoot}}, 1)$ onto the `b` axis of the CIE Lab colour space.

Fig. 5: Sorting and colouring in classic flame graphs (Figure 5a) vs 'gold plated' flame graphs (Figure 5b).

4.1.2 Colour by Name

As mentioned in section section 3, the colour of a node is a function of the hashed value of the node's function's name. This scheme poses several problems, the most egregious of which is that the canonical implementation does not hold the luminosity constant. This in turn means the legibility of the node's label could be negatively affected if there were insufficient contrast between the two. [11]

Even if the luminosity were to be held constant, the colour-encodes-name approach offers no real value other than as a contrast helper between adjacent nodes. The encoding is redundant (nodes have a label showing their function's name) but not predictable nor reversible, which eliminates any dual-encoding benefits to the user's comprehension. Finally the encoding suffers heavily from aliasing due to a large number of names being mapped onto a relatively small range of distinguishable colours.

4.2 Improvements - Putting Lipstick on a Flame Graph

Solving the sorting design flaw outlined in subsection 4.1 proved trivial. Making better use of the colour channel and modifying flame-graphs to convey some sense of function-level information, however, proved somewhat more challenging.

4.2.1 Sort by Node Weight

The sibling-sort issue was easily solved by ordering siblings by their weight. Though a trivial change, it dramatically improved the clarity of a sibling row's weight distribution and removed any ambiguity regarding the relative weights of similarly sized siblings. (Figure Figure 5.)

4.2.2 Colour by Function Weight

Our initial approach was to colour nodes by mapping $\frac{weight_{function}}{weight_{graph}}$ to a subset of `h` axis of CIE Lhs colour space (i.e. the yellow-blue portion). CIE Lhs was selected specifically for being a perceptually linear colour space. [10] Luminosity was held constant to ensure proper contast between the fill-colour of a node and its text label, and saturation was 'jittered' based on the hash of the function's name to help contrast adjacent nodes. (Figure Figure 6.)

By using low saturation colours and a yellow-blue scale to avoid colour-blindness, as suggested by Ware [10], it was hopped that this would result in the graphs generally transitioning from yellow (high weight) to blue (low weight) as one travelled from the root towards the leaves. Since colour is defined as a function of $weight(function)$, rather than $weight(node)$, it was also hoped that this would highlight (in yellow-er tones) any heavier functions appearing in the extremities of the graph.

This was partially successful, but suffered from two main problems. The first issue being that the saturation jittering affected the luminosity and chroma when the Lhs colour was reduced to 8-bit RGB for display. The second issue being that the strong contrast between yellow and blue gave the impression that there was some kind of categorical difference with the lighter nodes rather than simply being below 50% of the graph weight. This effect was accentuated in data-sets featuring large number of lightweight nodes, as the graph would suddenly become sharply divided between a small number of yellow-grey nodes and a majority of blue nodes.

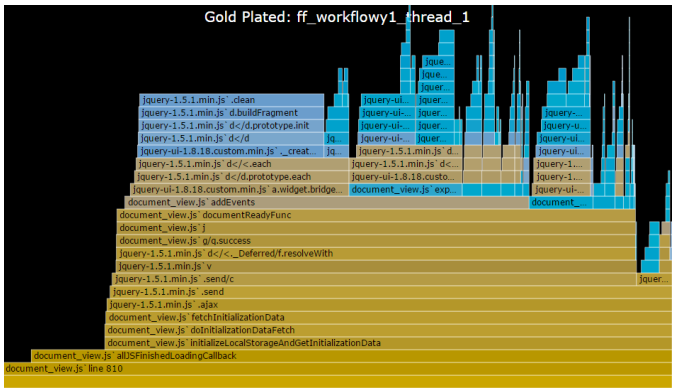


Fig. 6: Colouring nodes by their function's weight relative to the total weight. Function with near 100% of the total weight are yellow, fade to grey (50% of the total weight), and finally to blue (0% of the total weight). Luminosity is held constant, but saturation is jittered based on the function's name in an attempt to enhance contrast between adjacent nodes. Note that this graph has been fully folded. (See Figure Figure 9 for more on folding.)

Changing to the Lab colour space, and mapping directly onto a subset of `b` axis which could be precisely represented in 8 bits of RGB solved these two issues. Since saturation could no longer be independently manipulated in this colour space, contrasting between adjacent nodes was handled by changing the outline of each node to match the background colour. (Figure Figure 4b.)

The resulting mapping was usable, but its use of $weight_{(}graph)$ as the denominator interacted in an undesirable manner with the click-to-zoom feature. The colour function's 'domain' remained constant regardless of what node was 'focused' and caused the graph to be dominated by greys when zoomed into small sub-trees. Changing the domain function to $min(\frac{weight_{function}}{weight_{zoomRoot}}, 1)$ ensured the use of entire range at all times.

4.2.3 Hover Queries

Hover queries are among the quickest and cheapest (in terms of interactive and cognitive overhead) forms of epistemic interactions. [8] As such, the existing hover-functionality in flame-graphs seemed somewhat underexploited.

The tool-tips were expanded to display more information and, more importantly, hovering over a node causes all nodes belonging to the same function to be highlighted. (Figure Figure 7.) The 'highlighting' scheme uses three features suggested by Ware [9] to enable pre-attentive identification:

1. Outlining with a thick contrasting border
2. Increasing the weight of the label's font (line thickness)
3. Appearance of motion induced by font weight change

Magenta was selected as the outlining colour, as it lies on the CIE Lab's `a` axis and is therefore perceptually orthogonal to the `b` axis used for colouring nodes. [10]

The last feature, motion, was an unintended but beneficial side-effect of changing the font's weight, which caused the text to 'jump' slightly from the text's layout being shifted to accommodate the wider character glyphs.

4.3 Results - Lipstick Is Not Enough

The changes outlined above did result in notable improvements, but these were not significant as was hoped. Colouring by function weight did provide some ability to locate the nodes of heavy functions but was hampered by the distribution of weight among functions being heavily biased. This resulted in large amounts of dull yellows/greys throughout the graph, rather than the insightful highlights we were hoping for.

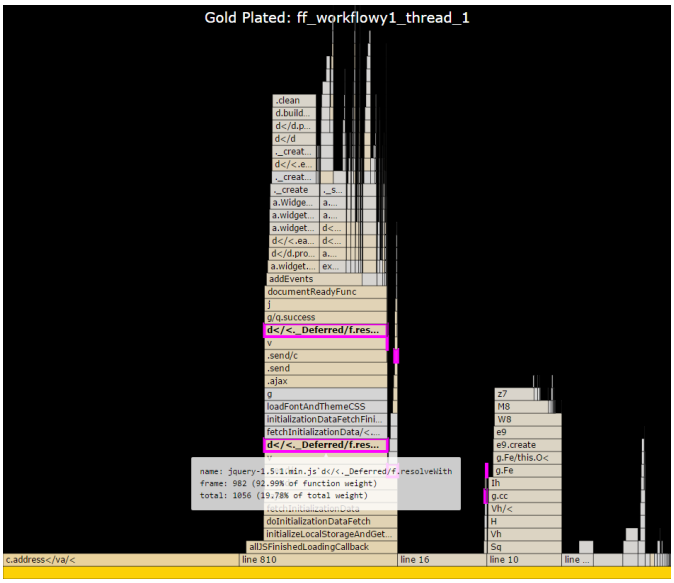


Fig. 7: Hovering over a node displays both additional information as a tool-tip, and highlights all other nodes belonging to the same function using a thick contrasting outline and a heavier font weight.

5 Funky Graphs

Flame graphs are fundamentally geared towards visualising weighted call-stacks, and while effective at their domain, it has proven rather difficult to modify them to present function-aggregated information without compromising their effectiveness at visualising weighed call-stacks. This prompted the creation of complementary visualisation specialised for displaying function-aggregated information: The funky (**FUN**ction **HIE**rarchy) graph. (Figure Figure 8.)

A funky graph visualises a set of functions, the rose tree of callers for each function, and the rose tree of callees for each function. Stated more formally, a funky graph is visualisation of a set of pairs of two rose trees sharing a common root. Stated less formally, a funky graph visualises for each function both all the functions it calls, and all the functions that call it.

Since funky graphs are derived from gold plated flame graphs they inherit the latter's epistemic and interactive features, such as search-by-name, hover highlights, tool-tips, and click-to-zoom.

5.1 Layout & Structure

A funky graph is constructed by assigning a pair of fully-folded flame graphs to every function in a profile, one for their callees and one for their callers. Figure 9 provides helpful visual comparison of a fully folded flame graph, and the equivalent funky graph for a single function.

This set of bi-flame-graphs is ordered by their owning function's inclusive weight, and assigned a width proportional to their function's exclusive weight. Assigning a width to root-nodes based on exclusive weight allows for the identification of heavy leaf functions, while sorting by inclusive weight maintains consistency throughout the hierarchy.

5.1.1 Constructing a Function's Bi-Flame-Graph

Each function generates a two flame graph sharing a common root node. The first flame graph details the callees of the function, while the second details the callers of the function. The callees flame graph is position above the function's root and coloured with shades of yellow (just as a normal flame-graph). The callers flame graph is 'flipped' vertically, placed below the root, and is coloured with shades of blue (exploiting the categorical difference noted in subsubsection 4.2.2).

The rose tree for a Callees flame graph is constructed by fully folding any frames of the given function `a`, and then constructing a rose

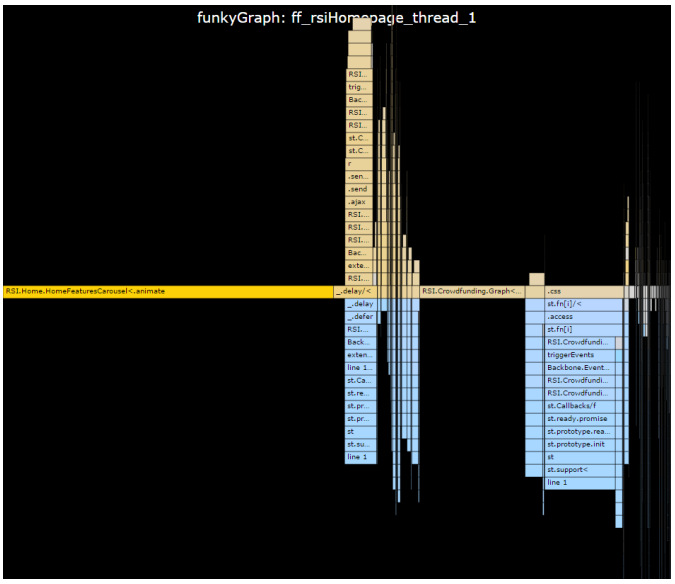
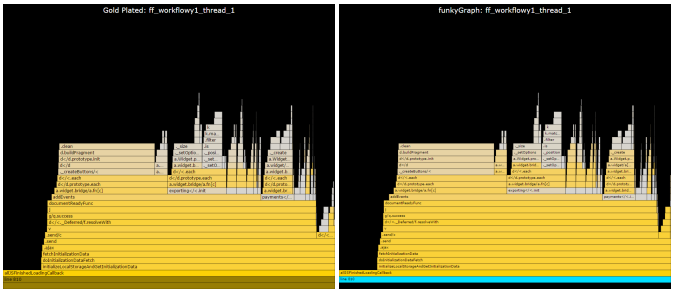


Fig. 8: An example of a funky graph.



(a) A full-folded gold plated flame graph.

(b) A funky graph.

Fig. 9: Funky graphs are effectively flame graphs applied to entire functions rather than a set of call-stacks. Funky graphs are constructed by creating a pair of fully-folded flame graphs for each function. The first flame graph (above the root, yellow) details all the functions a given function invokes, while the second flame graph (below the root, blue) details the functions which call it.

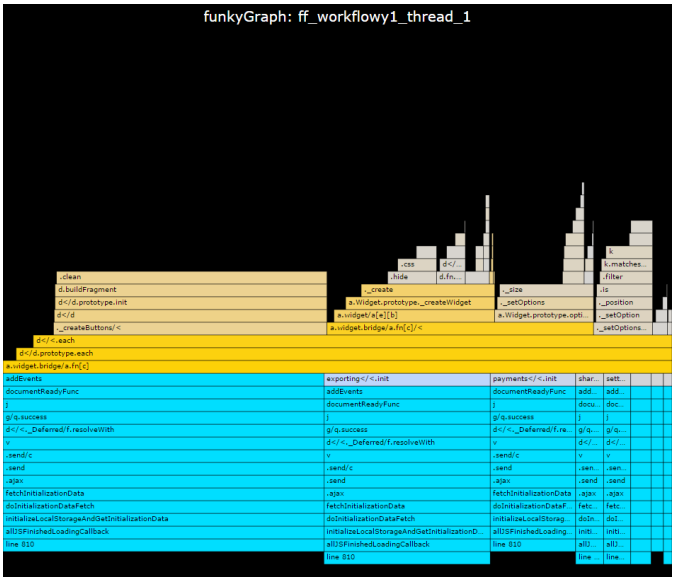


Fig. 10: A funky graph zoomed onto a function (`a.widget.bridge/a.fn[c]`) with multiple invokers (blue). The width of each invoker is proportional how much they contributed to their children's invocation. Note that the call-stacks of the parents have also been folded.

from the sub-stacks of the remaining frames of `a`.

The rose tree for a Callers flame graph is constructed in two steps. First, a set of new weighed call-stacks is constructed by taking creating a 'reversed' call-stack for every frame of the function in question (post full-fold, as with Callees), where the weight of the reversed call-stack is equal to the inclusive weight of the frame. The rose tree is then constructed from this set of weighed 'reversed' call-stacks.

An example of a fully assembled bi-flame-graph for a function can be found in Figure 10.

5.2 Cross-Graph Epistemic Interaction

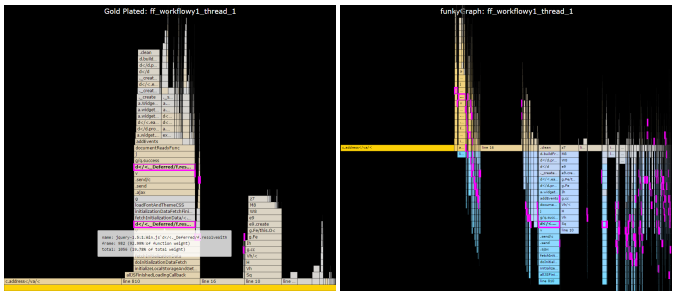
Since the width of a root-node in a funky graphs is defined by the function's exclusive weight, it is difficult to identify heavy non-leaf functions. This is the primary reason why funky graphs complement rather than supersede flame graphs. In our implementation the two graphs are placed side-by side as in Figures Figure 9, Figure 11, and Figure 12.

Epistemic actions such as hovering and searching are performed simultaneously on both graph (Figure Figure 11). This is has proven useful for a number of tasks, such as locating a heavy leaf function in the funky graph and quickly locating the corresponding nodes in the flame graph.

6 Discussion

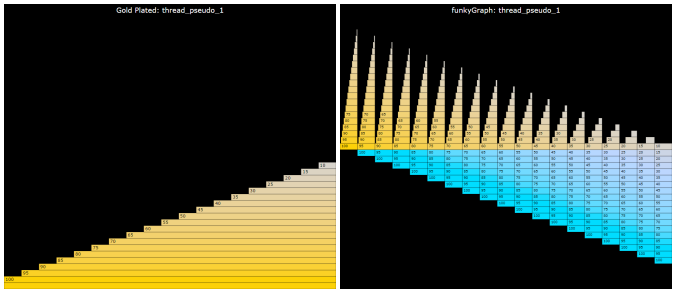
We initially had high hopes of using the colour channel to identify heavy functions split across multiple nodes a flame graph. While partially successful, the span of colours along the Lab `a` axis which are representable in RGB proved disappointingly small and undistinctive in practice. In particular the higher magnitudes of the `a` axis seemed to saturate to yellow more quickly than we expected. We are unsure if this is due to a perceptual illusion, or simply because our displays are poorly calibrated. (Figure 12.)

Relative quantitative data proved to be only reliably conveyed using length comparisons, which necessitated the creation of funky graphs to display function-aggregated weight. Unfortunately, funky-graphs are rather tricky to both implement and interpret when compared to flame-graphs. Features such as the asymmetric partitioning for root-nodes vs children in the bi-flame-graphs increase the density of information but impose a higher cognitive overhead to interpret.



(a) Gold plated flame graph. (b) Funky graph.

Fig. 11: Our implementation supports shared epistemic interactions between the two graphs. Here the cursor is hovered over a node in the flame graph, causing the corresponding nodes in graphs to be highlighted.



(a) Gold plated flame graph. (b) Funky graph.

Fig. 12: Test data demonstrating colouring in both gold plated flame graphs and funky graphs.

7 Conclusion

Most profiling visualisations for weighed call-stacks are rather rubbish. The best of the bunch so far is the flame graph, but it has difficulty conveying function-aggregated information. Despite our best attempts there does not seem to be any way of significantly improving their ability to do this. Pairing flame graphs with a visualisation specialised for displaying function-aggregated information proved to be much more successful.

Unfortunately, funky graphs are somewhat difficult to use without flame graphs, necessitating their pairing and effectively doubling the display footprint. A possible solution would be to allocate a smaller portion of the viewport to funky graphs, and only display a single bi-flame-graph at a time.

References

[1] M. Bostock, V. Ogievetsky, and J. Heer. D3: Data-driven documents. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2011.

[2] B. Gregg. Flame graphs. `www.brendangregg.com/flamegraphs.html`.

[3] B. Gregg. Flame graphs - click to zoom. `github.com/brendangregg/FlameGraph/pull/35`.

[4] J. M. L. J. B. Kruskal. Icicle plots: Better displays for hierarchical clustering. *The American Statistician*, 37(2):162–168, 1983.

[5] A. Mahieux. Flame graphs - search. `www.brendangregg.com/blog/2015-08-11/flame-graph-search.html`.

[6] M. Spier. d3 flame graph. `github.com/spiermar/d3-flame-graph`.

[7] J. Stasko, R. Catrambone, M. Guzdail, and K. McDonald. An evaluation of space-filling information visualisations for depicting hierarchical structures. *International Journal of Human-Computer Studies*, 53, 2000.

[8] C. Ware. Chapter eleven - visual thinking processes. In C. Ware, editor, *Information Visualization (Third Edition)*, Interactive Technologies, pages 375 – 423. Morgan Kaufmann, Boston, third edition edition, 2013.

[9] C. Ware. Chapter five - visual salience and finding information. In C. Ware, editor, *Information Visualization (Third Edition)*, Interactive Technologies, pages 139 – 177. Morgan Kaufmann, Boston, third edition edition, 2013.

[10] C. Ware. Chapter four - color. In C. Ware, editor, *Information Visualization (Third Edition)*, Interactive Technologies, pages 95 – 138. Morgan Kaufmann, Boston, third edition edition, 2013.

[11] C. Ware. Chapter three - lightness, brightness, contrast, and constancy. In C. Ware, editor, *Information Visualization (Third Edition)*, Interactive Technologies, pages 69 – 94. Morgan Kaufmann, Boston, third edition edition, 2013.

[12] R. M. William S. Cleveland. Graphical perception: Theory, experimentation, and application to the development of graphical methods. *Journal of the American Statistical Association*, 79(387):531–554, 1984.