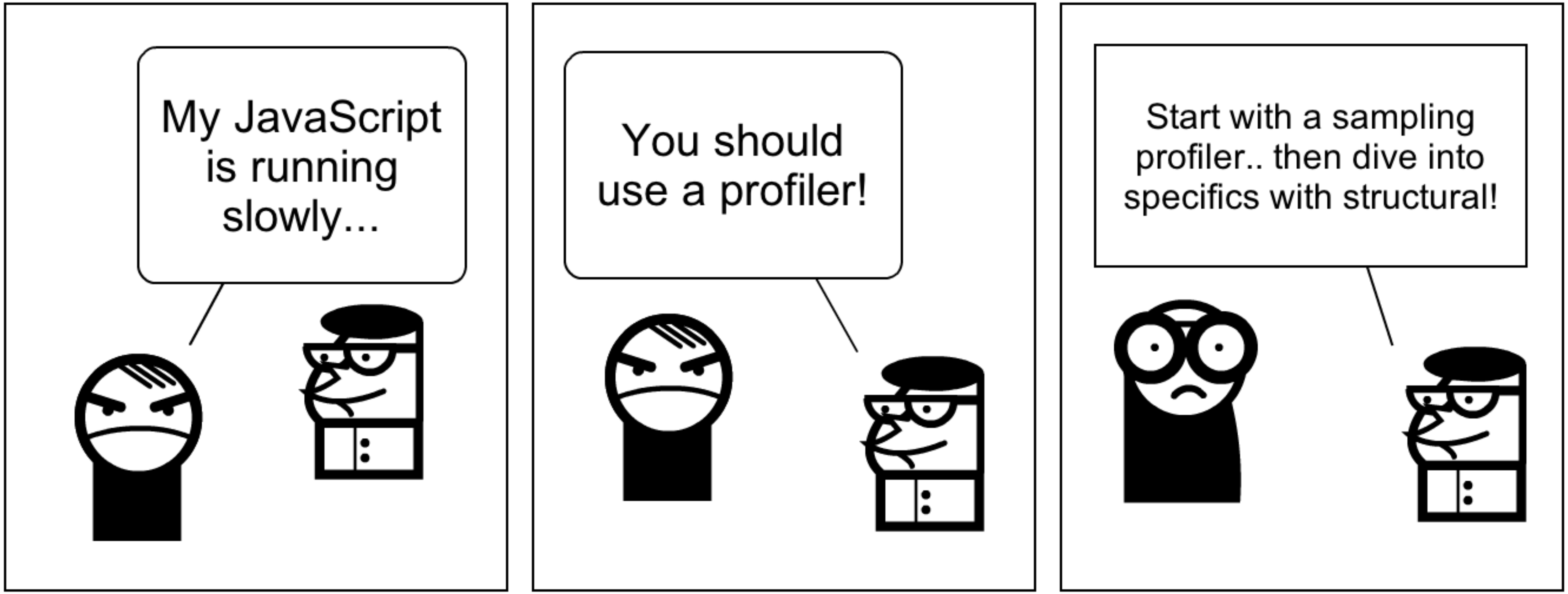
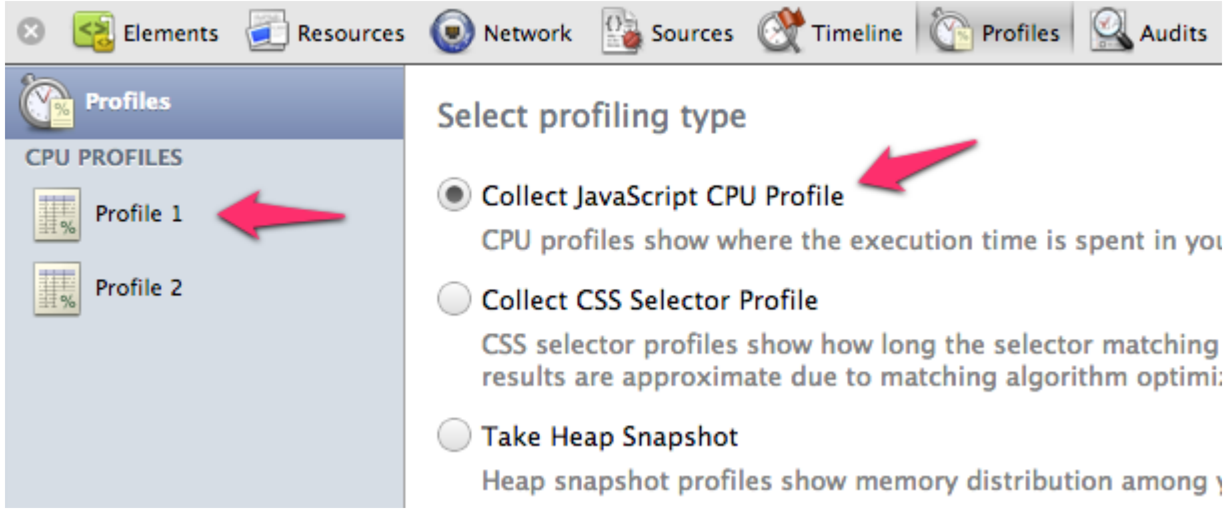
Structural and Sampling JavaScript Profiling

in Google Chrome


My JavaScript is running slowly...
You should use a profiler!
Start with a sampling profiler.. then dive into specifics with structural!

1. **Sampling**
    a. Measures samples



Select profiling type

● Collect JavaScript CPU Profile

CPU profiles show where the execution time is spent in you

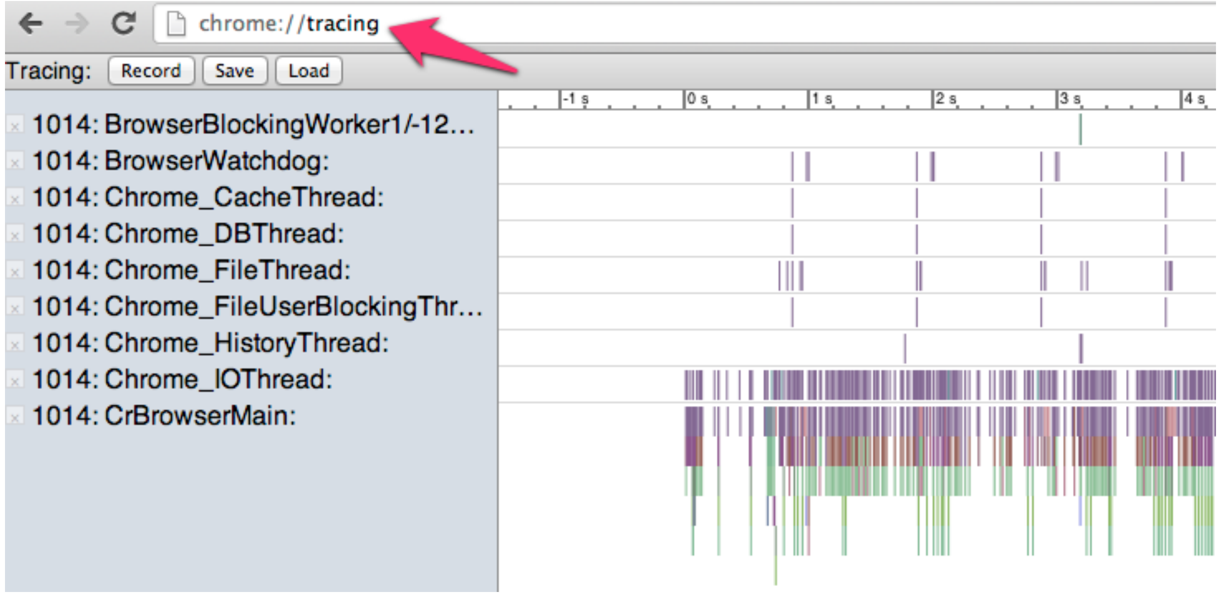
○ Collect CSS Selector Profile

CSS selector profiles show how long the selector matching results are approximate due to matching algorithm optimi

○ Take Heap Snapshot

Heap snapshot profiles show memory distribution among y

2. **Structural**
    a. Measures time
    b. aka, **instrumenting** / markers / inline



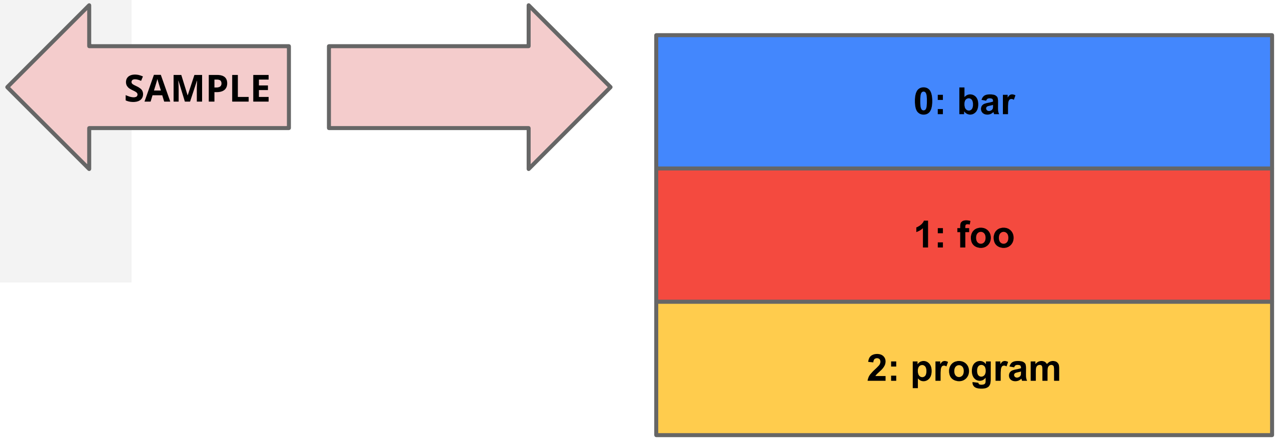
1014: BrowserBlockingWorker1/-12...
1014: BrowserWatchdog:
1014: Chrome_CacheThread:
1014: Chrome_DBThread:
1014: Chrome_FileThread:
1014: Chrome_FileUserBlockingThr...
1014: Chrome_HistoryThread:
1014: Chrome_IOThread:
1014: CrBrowserMain:

Sampling CPU Profilers

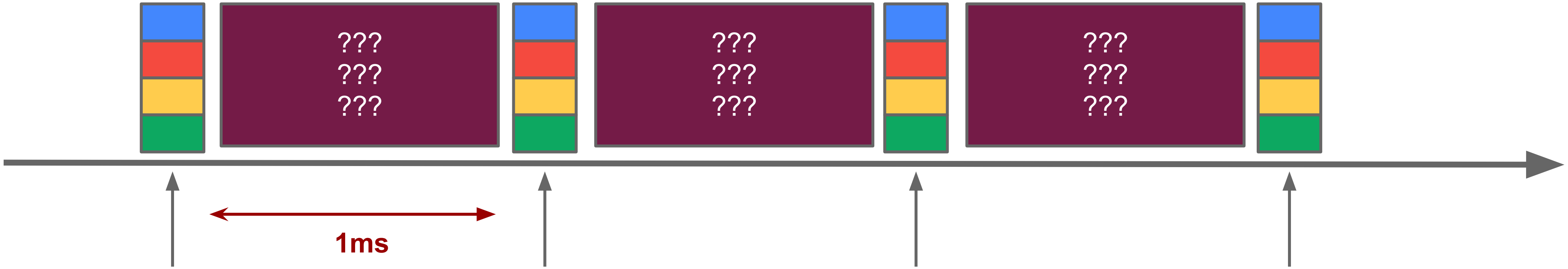
At a fixed frequency:

Instantaneously **pause the program** and **sample the call stack**

```
function foo() {
  bar();
}

function bar() {

}

foo();
```



Sampling CPU Profilers



Assumption: our sample is representative of the workload

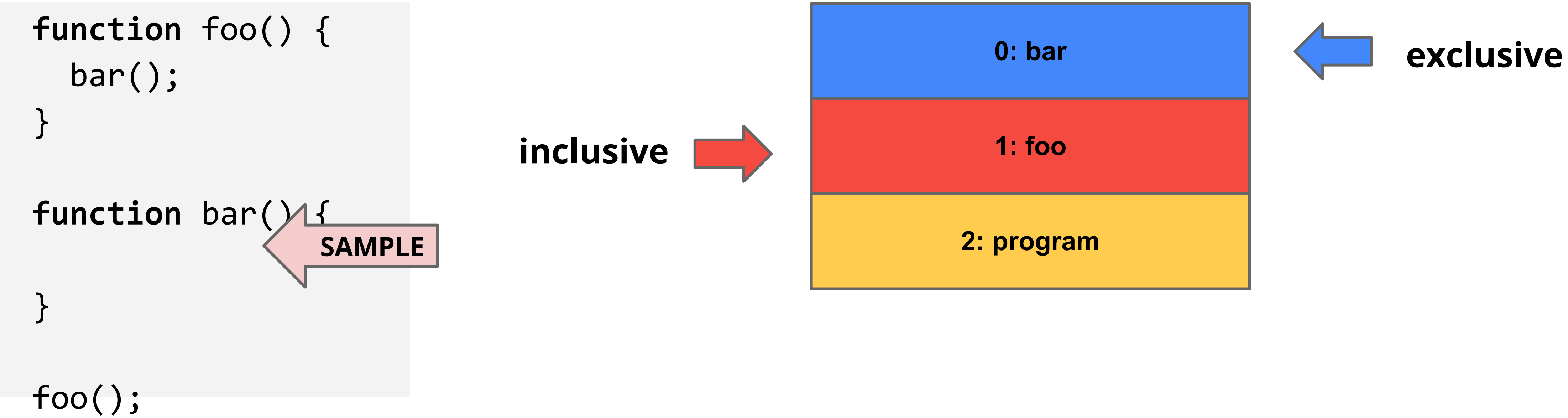
- data sampled on a ***1 ms*** interval in Chrome
- collect data for longer period of time
- ensure that your code is exercising the right code-paths

Sampling CPU Profilers

Samples are processed and outputs two data points per function:

1. Percentage of samples function was **leaf of a call stack**
   a. Analogous to **exclusive time**

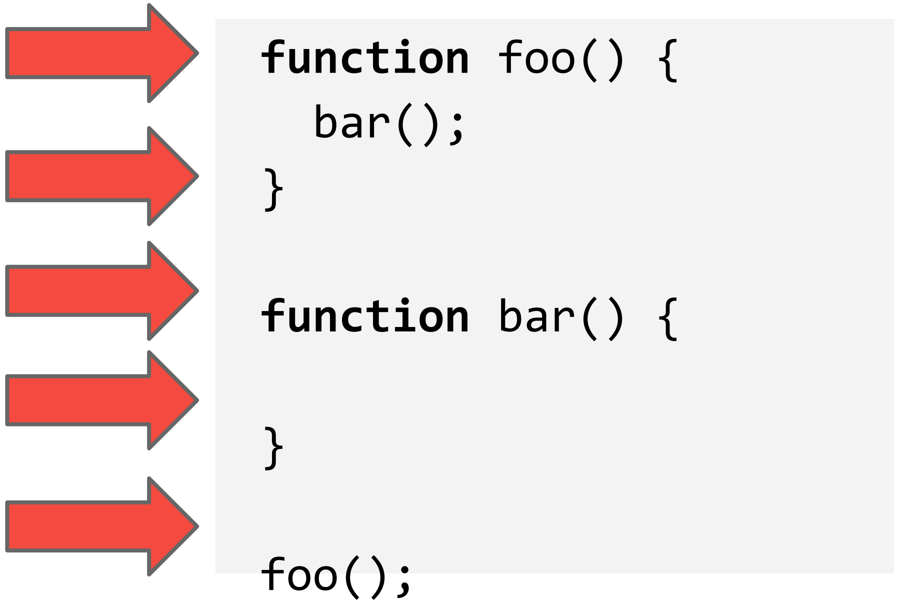
2. Percentage of samples function was **present in call stack**
   a. Analogous to **inclusive time**



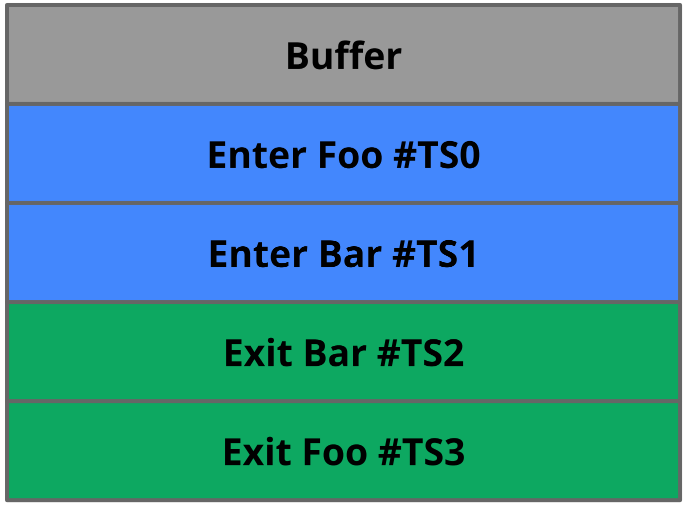
```
function foo() {
  bar();
}

function bar() {

}

foo();
```

Structural CPU Profilers

*Functions are instrumented to record **entry** and **exit** times.*



```
function foo() {
  bar();
}

function bar() {

}

foo();
```



Buffer

Enter Foo #TS0

Enter Bar #TS1

Exit Bar #TS2

Exit Foo #TS3

Structural execution trace

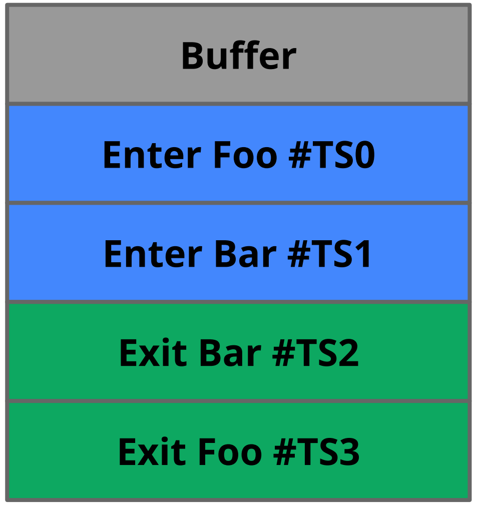
Structural CPU Profilers

Buffer is processed and outputs three data points per function:

1. **Inclusive Time**
   a. Time function was running for ***including*** **time spent inside children**.

2. **Exclusive Time**
   a. Time function was running for ***excluding*** **time spent inside children**.

3. **Call Count**
   a. Number of times the function was called.



Buffer

Enter Foo #TS0

Enter Bar #TS1

Exit Bar #TS2

Exit Foo #TS3

Structural execution trace

JavaScript optimization: the quest to ***minimize the inclusive time*** *of a function.* *

aka, including time spent inside children

Which should I use? ... Both!

	Sampling	Structural / Instrumenting
Time	Approximate	Exact
Invocation count	Approximate	Exact
Overhead	Small	High(er)
Accuracy * **	Good - Poor	Good - Poor
Extra code / instrumentation	No	Yes

- Instrumenting profilers requires that you.. instrument your code:
    - Fine-grained control over what is being traced, but requires that you know what to trace
    - Platform code / API's out of reach

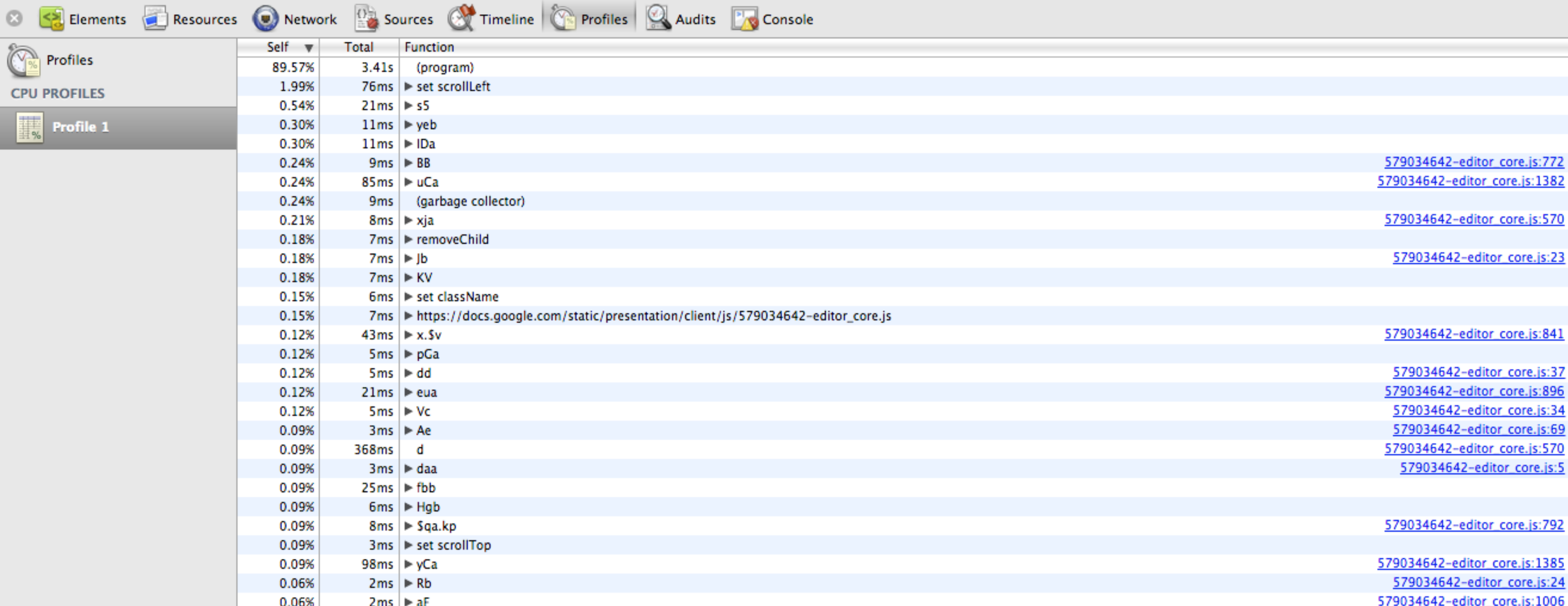
- Sampling profilers require no instrumentation, but:
    - Are an approximation of what is happening in your application
    - May miss or hide some code-paths

P.S. It's not either, or... you need both!

Sampling CPU Profiling in Chrome

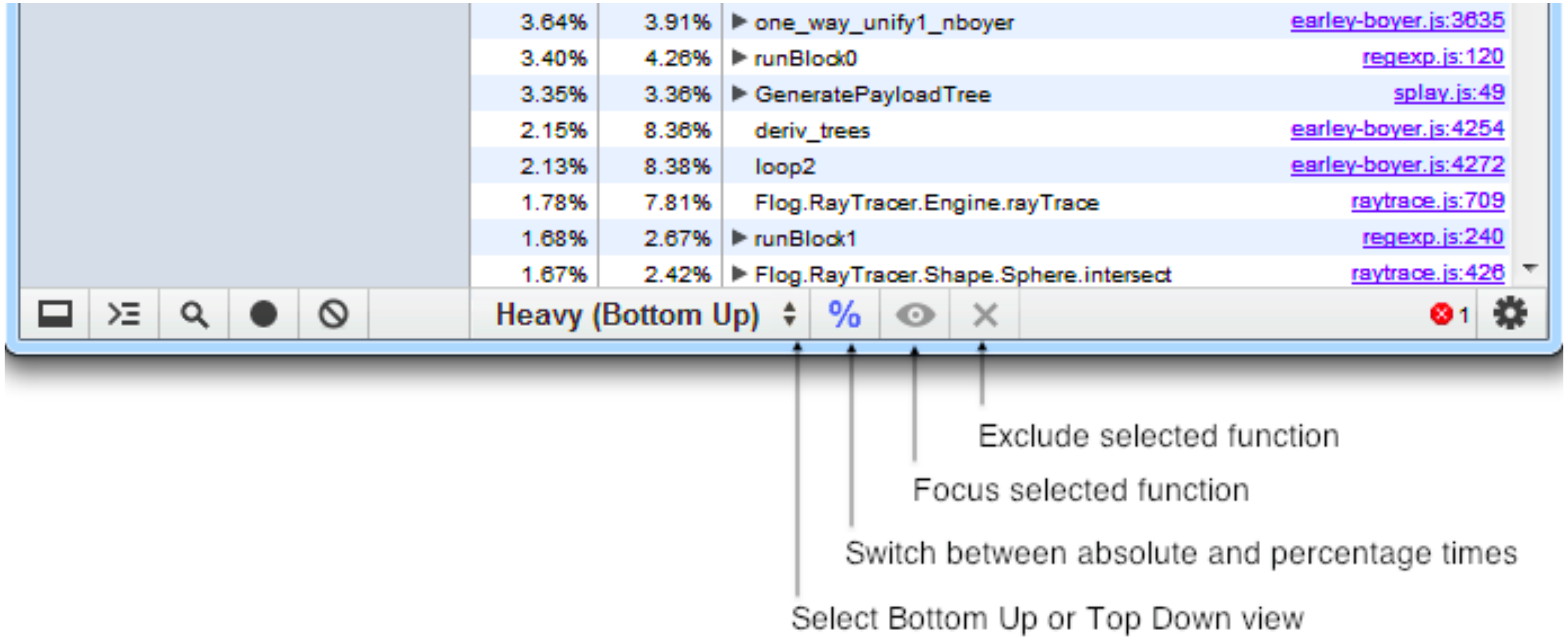
Built-in sampling CPU profiler in ... Profiles tab in Developer Tools!

- instantaneously pauses your code and samples the call stack



Self ▼	Total	Function	
89.57%	3.41s	(program)	
1.99%	76ms	▶ set scrollLeft	
0.54%	21ms	▶ s5	
0.30%	11ms	▶ yeb	
0.30%	11ms	▶ IDa	
0.24%	9ms	▶ BB	579034642-editor_core.js:772
0.24%	85ms	▶ uCa	579034642-editor_core.js:1382
0.24%	9ms	(garbage collector)	
0.21%	8ms	▶ xja	579034642-editor_core.js:570
0.18%	7ms	▶ removeChild	
0.18%	7ms	▶ Jb	579034642-editor_core.js:23
0.18%	7ms	▶ KV	
0.15%	6ms	▶ set className	
0.15%	7ms	▶ https://docs.google.com/static/presentation/client/js/579034642-editor_core.js	
0.12%	43ms	▶ x.$v	579034642-editor_core.js:841
0.12%	5ms	▶ pGa	
0.12%	5ms	▶ dd	579034642-editor_core.js:37
0.12%	21ms	▶ eua	579034642-editor_core.js:896
0.12%	5ms	▶ Vc	579034642-editor_core.js:34
0.09%	3ms	▶ Ae	579034642-editor_core.js:69
0.09%	368ms	d	579034642-editor_core.js:570
0.09%	3ms	▶ daa	579034642-editor_core.js:5
0.09%	25ms	▶ fbb	
0.09%	6ms	▶ Hgb	
0.09%	8ms	▶ $qa.kp	579034642-editor_core.js:792
0.09%	3ms	▶ set scrollTop	
0.09%	98ms	▶ yCa	579034642-editor_core.js:1385
0.06%	2ms	▶ Rb	579034642-editor_core.js:24
0.06%	2ms	▶ aF	579034642-editor_core.js:1006

Sampling CPU profiling in Chrome



3.64%	3.91%	▶ one_way_unify1_nboyer	earley-boyer.js:3635
3.40%	4.26%	▶ runBlock0	regexp.js:120
3.35%	3.36%	▶ GeneratePayloadTree	splay.js:49
2.15%	8.36%	deriv_trees	earley-boyer.js:4254
2.13%	8.38%	loop2	earley-boyer.js:4272
1.78%	7.81%	Flog.RayTracer.Engine.rayTrace	raytrace.js:709
1.68%	2.67%	▶ runBlock1	regexp.js:240
1.67%	2.42%	▶ Flog.RayTracer.Shape.Sphere.intersect	raytrace.js:426

Exclude selected function

Focus selected function

Switch between absolute and percentage times

Select Bottom Up or Top Down view

Demo: V8 Benchmark Suite

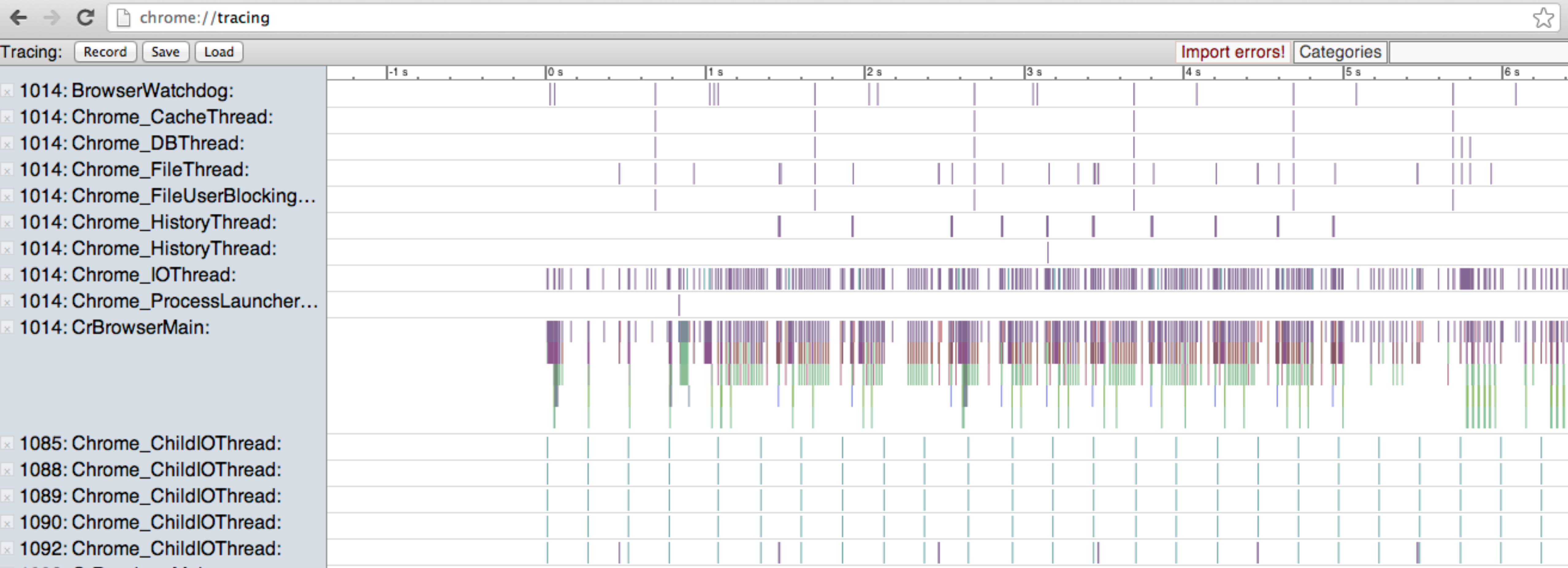
- **Heavy** (bottom up view): functions by impact on performance + ability to examine the calling paths to each
- **Tree** (top down view): overall picture of the calling structure, starting at the top of the call stack
- Use "**Focus selected function**" to zero in on just the code you care about

Chrome Developer Tools: CPU Profiling

Structural CPU Profiling in Chrome

chrome://tracing is a power user structural profiler

- built for intrusive profiling of Chrome's internals
- most of this can and should be hidden for JavaScript profiling


chrome://tracing
Tracing:
Record
Save
Load
Import errors!
Categories
-1 s
0 s
1 s
2 s
3 s
4 s
5 s
6 s
1014: BrowserWatchdog:
1014: Chrome_CacheThread:
1014: Chrome_DBThread:
1014: Chrome_FileThread:
1014: Chrome_FileUserBlocking...
1014: Chrome_HistoryThread:
1014: Chrome_HistoryThread:
1014: Chrome_IOThread:
1014: Chrome_ProcessLauncher...
1014: CrBrowserMain:
1085: Chrome_ChildIOThread:
1088: Chrome_ChildIOThread:
1089: Chrome_ChildIOThread:
1090: Chrome_ChildIOThread:
1092: Chrome_ChildIOThread:

How to use chrome://tracing to profile JavaScript...

1. **You*** must instrument your JavaScript code.

```
function foo() {
  console.time("foo");
  bar();
  console.timeEnd("foo");
}

function bar() {
  console.time("bar");
  console.timeEnd("bar");
}

foo();
```

Some types of instrumentation:

- Manual
- Compiler / automatic tool
- Runtime instrumentation (ex. Valgrind)

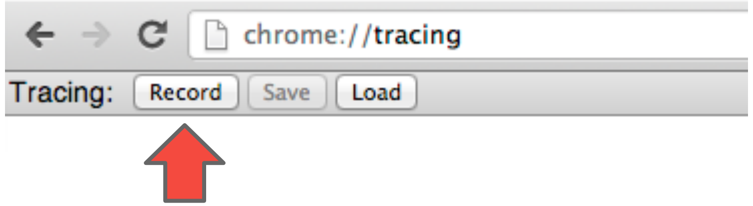
"Trace macros are very low overhead. When tracing is ***not turned on, trace macros cost at most a few dozen clocks.*** *When running, trace macros cost a few thousand clocks at most.*

Arguments to the trace macro are evaluated only when tracing is on --- if tracing is off, the value of the arguments don't get computed."

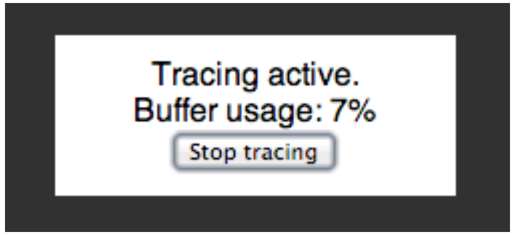
WARNING: console.time and console.timeEnd spam the developer tools console. Keep it closed.

How to use chrome://tracing to profile JavaScript...

2. Start recording a trace



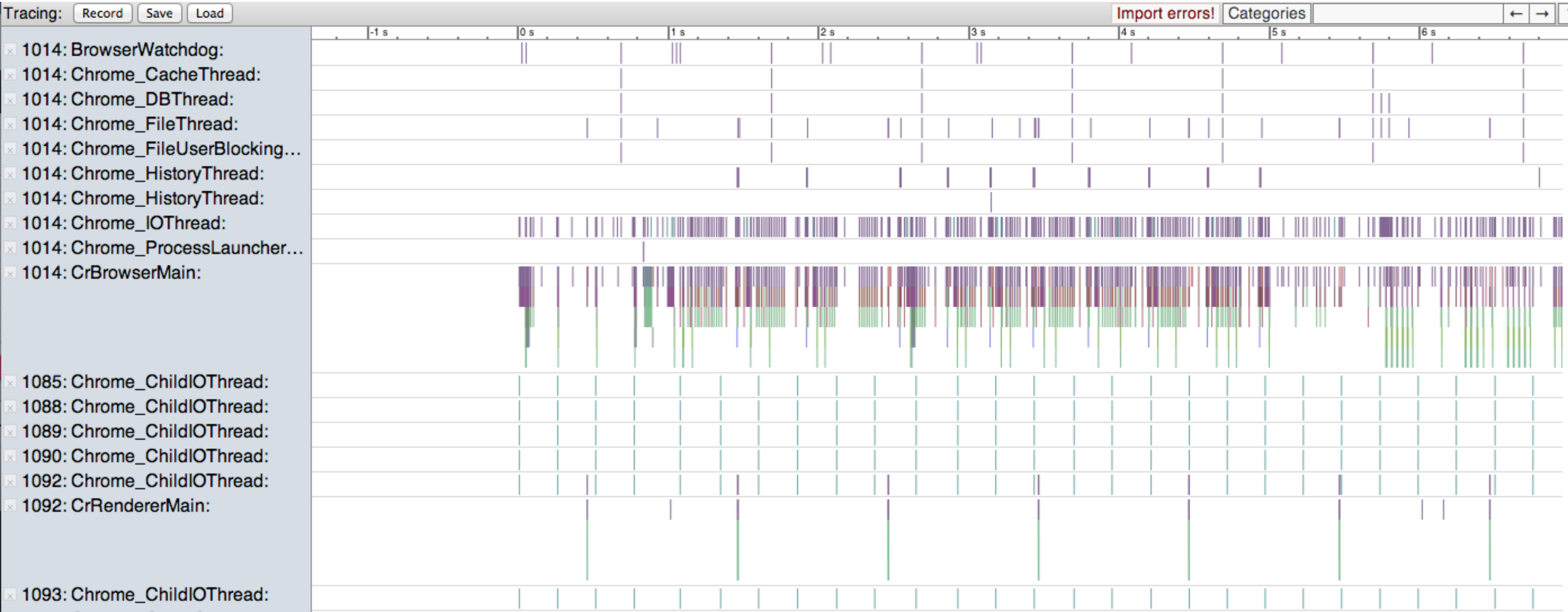
3. Interact with your application...
4. Head back, hit **stop tracing**


Tracing active.
Buffer usage: 7%

Record on the order of a few to dozens of seconds of profiling data...

How to use chrome://tracing to profile JavaScript...

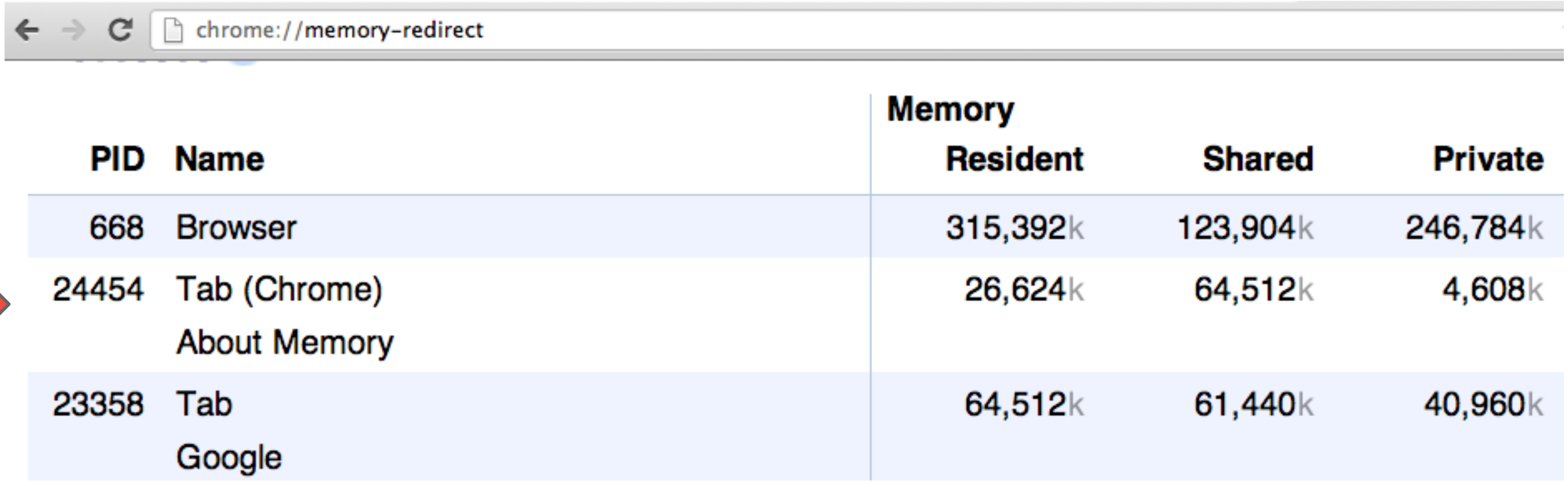
5. Behold the noise!


Tracing:
Record
Save
Load
Import errors!
Categories
←
→
-1 s
0 s
1 s
2 s
3 s
4 s
5 s
6 s
1014: BrowserWatchdog:
1014: Chrome_CacheThread:
1014: Chrome_DBThread:
1014: Chrome_FileThread:
1014: Chrome_FileUserBlocking...
1014: Chrome_HistoryThread:
1014: Chrome_HistoryThread:
1014: Chrome_IOThread:
1014: Chrome_ProcessLauncher...
1014: CrBrowserMain:
1085: Chrome_ChildIOThread:
1088: Chrome_ChildIOThread:
1089: Chrome_ChildIOThread:
1090: Chrome_ChildIOThread:
1092: Chrome_ChildIOThread:
1092: CrRendererMain:
1093: Chrome_ChildIOThread:

How to use chrome://tracing to profile JavaScript...

6. Find your page's process ID in **chrome://memory**

24454

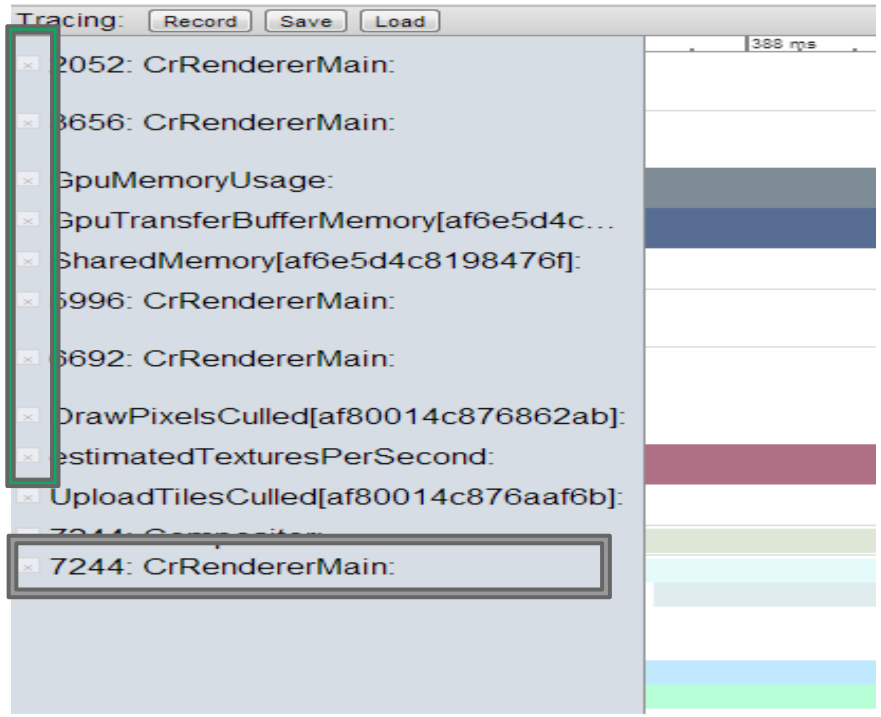


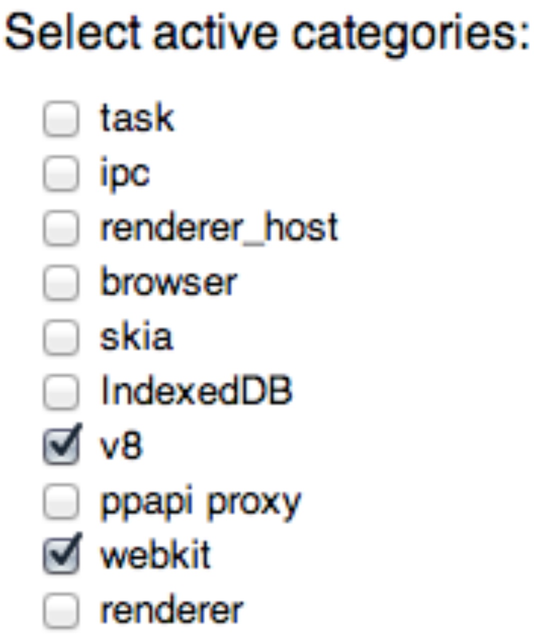
PID	Name	Memory Resident	 Shared	 Private
668	Browser	315,392k	123,904k	246,784k
24454	Tab (Chrome) About Memory	26,624k	64,512k	4,608k
23358	Tab Google	64,512k	61,440k	40,960k

How to use chrome://tracing to profile JavaScript...

7. **Filter** for the signal

- remove unnecessary threads and components
- click on "Categories" in top right, and filter down the list



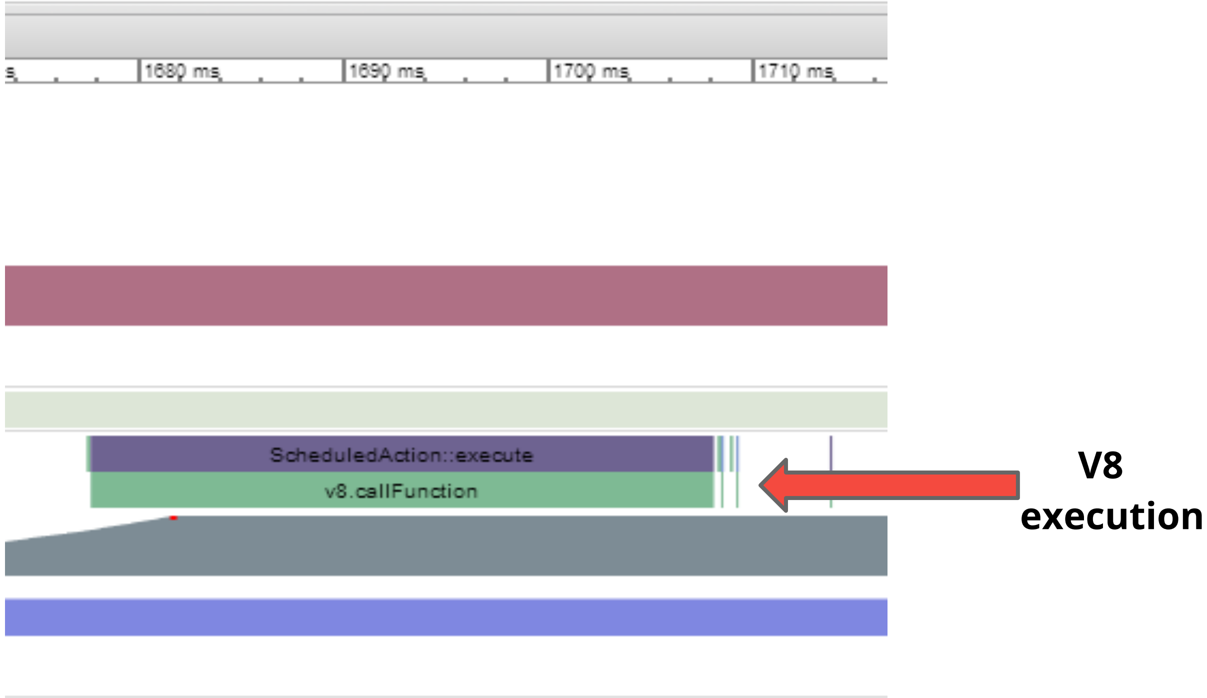


Select active categories:

☐ task
☐ ipc
☐ renderer_host
☐ browser
☐ skia
☐ IndexedDB
☑ v8
☐ ppapi proxy
☑ webkit
☐ renderer

How to use chrome://tracing to profile JavaScript...

8. Inspect the trace timeline, isolate your code...



V8 execution

W

A S D

Remember your Quake keys?

A - pan left
D - pan right
W - zoom in
S - zoom out

? - help

Let's do a walkthrough...

Hands on profiling...

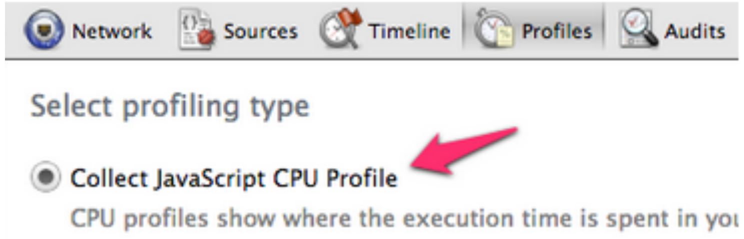
Let's assume the following scenario, with known exclusive run times...

```
function gameloop(timestamp) {
  A();
  requestAnimationFrame(gameloop);
}

function A() {
  spinFor(2);   // loop for 2 ms
  B();          // Calls C
}

...

function D() {
               // Called by C
  spinFor(2);  // loop for 2 ms
}
```

Function	Exclusive Run Time
A()	2 ms
B()	8 ms
C()	1 ms
D()	2 ms
Total	**13 ms**

Hands on profiling...

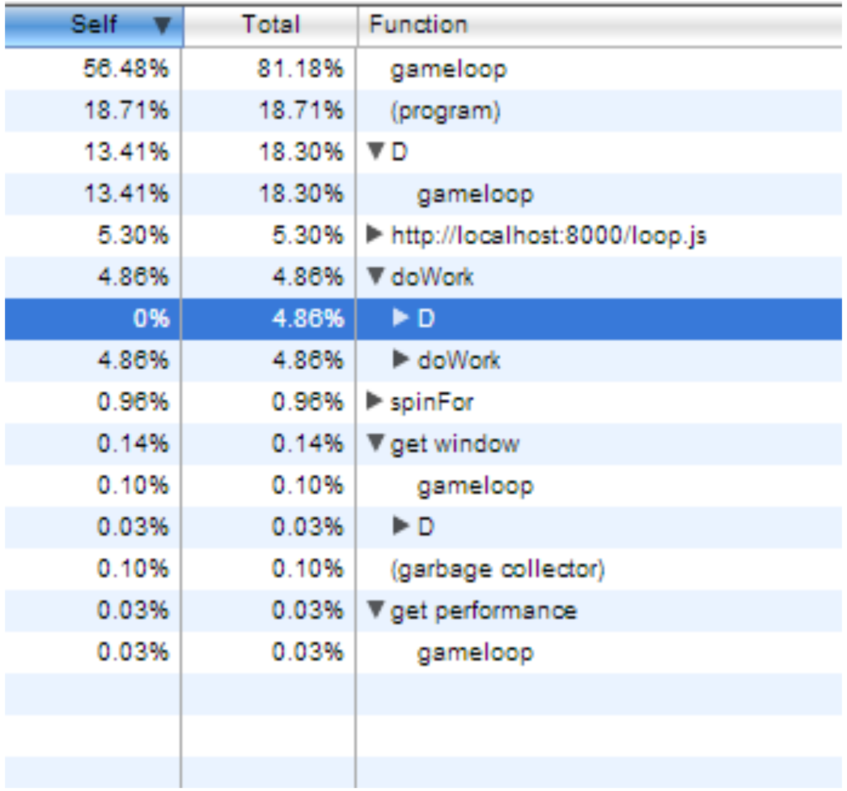
Open up Profiles tab in Developer Tools, hit start, record, stop...



Select profiling type

● Collect JavaScript CPU Profile

CPU profiles show where the execution time is spent in you



Self ▼	Total	Function
56.48%	81.18%	gameloop
18.71%	18.71%	(program)
13.41%	18.30%	▼ D
13.41%	18.30%	gameloop
5.30%	5.30%	▶ http://localhost:8000/loop.js
4.86%	4.86%	▼ doWork
0%	4.86%	▶ D
4.86%	4.86%	▶ doWork
0.96%	0.96%	▶ spinFor
0.14%	0.14%	▼ get window
0.10%	0.10%	gameloop
0.03%	0.03%	▶ D
0.10%	0.10%	(garbage collector)
0.03%	0.03%	▼ get performance
0.03%	0.03%	gameloop

Where is A(), B(), and C()?

spinFor() is only in 0.96 % of the samples?!

<facepalm>

A(), B(), C(), and *spinFor()* were optimized and ultimately **inlined into gameloop**!

</facepalm>

Inlining is a common compiler optimization

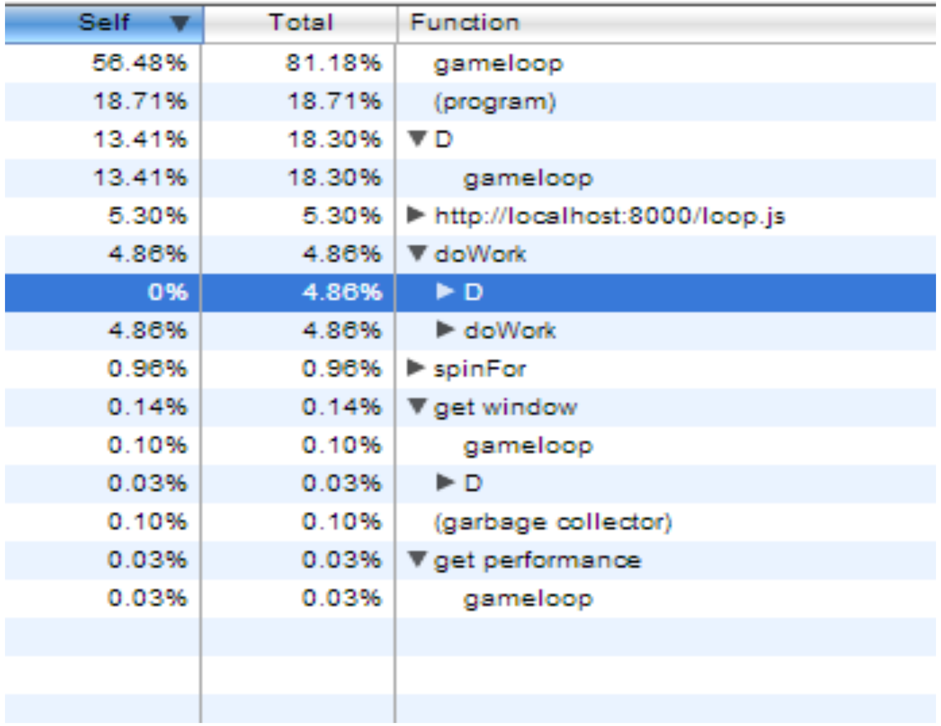
```
function gameloop(timestamp) {
 var x = 0;
 for (int i = 0; i < 10; i++) {
   x = A(x);
 }
}

function A(x) {
  return x + x;
}
```

```
function gameloop(timestamp) {
  var x = 0;
  for (int i = 0; i < 10; i++) {
    x = x + x;
  }
}
```

A() is erased when inlined into gameloop. **Erased functions cannot show up in sampling profiler capture.**

... Code in V8 != code in your source

Chrome Developer Tools (Sampling) Profiler



Self ▼	Total	Function
56.48%	81.18%	gameloop
18.71%	18.71%	(program)
13.41%	18.30%	▼ D
13.41%	18.30%	gameloop
5.30%	5.30%	▶ http://localhost:8000/loop.js
4.86%	4.86%	▼ doWork
0%	4.86%	▶ D
4.86%	4.86%	▶ doWork
0.96%	0.96%	▶ spinFor
0.14%	0.14%	▼ get window
0.10%	0.10%	gameloop
0.03%	0.03%	▶ D
0.10%	0.10%	(garbage collector)
0.03%	0.03%	▼ get performance
0.03%	0.03%	gameloop

*This trace **does not** resemble the application's actual execution flow or execution time.*

That's not to say that the sampling profiler is useless - to the contrary!

Hands on profiling...

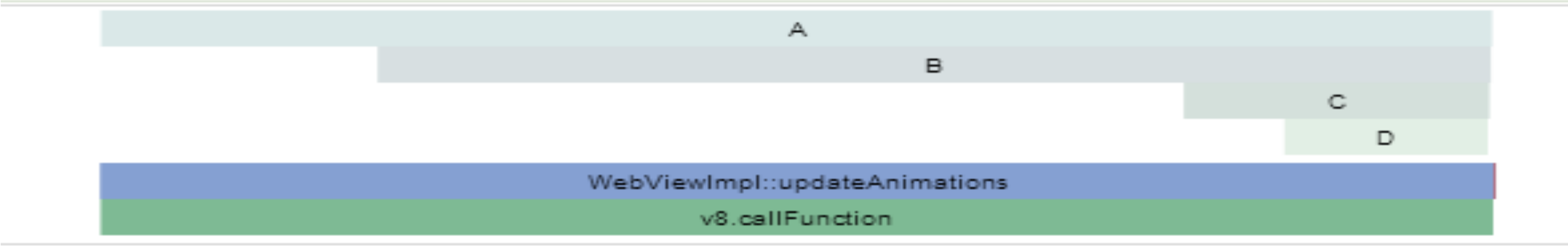
```
function A() {
  console.time("A");
  spinFor(2);     // loop for 2 ms
  B();            // Calls C
  console.timeEnd("A");
}

...

function D() {
  // Called by C
  console.time("D");
  spinFor(2);    // loop for 2 ms
  console.timeEnd("D");
}
```

Let's instrument our code with structural markers to help trace the actual execution path

P.S. The functions can still be inlined, but so will our console statements!

If you're wondering... there is ~0.01 ms of overhead per console call

Let's zoom in on the execution trace in chrome://tracing...



Function	Entry Time	Exit Time	Inclusive Runtime	Exclusive Runtime
A()	0 ms	13 ms	13 ms	2 ms
B()	2 ms	13 ms	11 ms	8 ms
C()	10 ms	13 ms	3 ms	1 ms
D()	11 ms	13 ms	2 ms	2 ms

Hands on profiling conclusions...

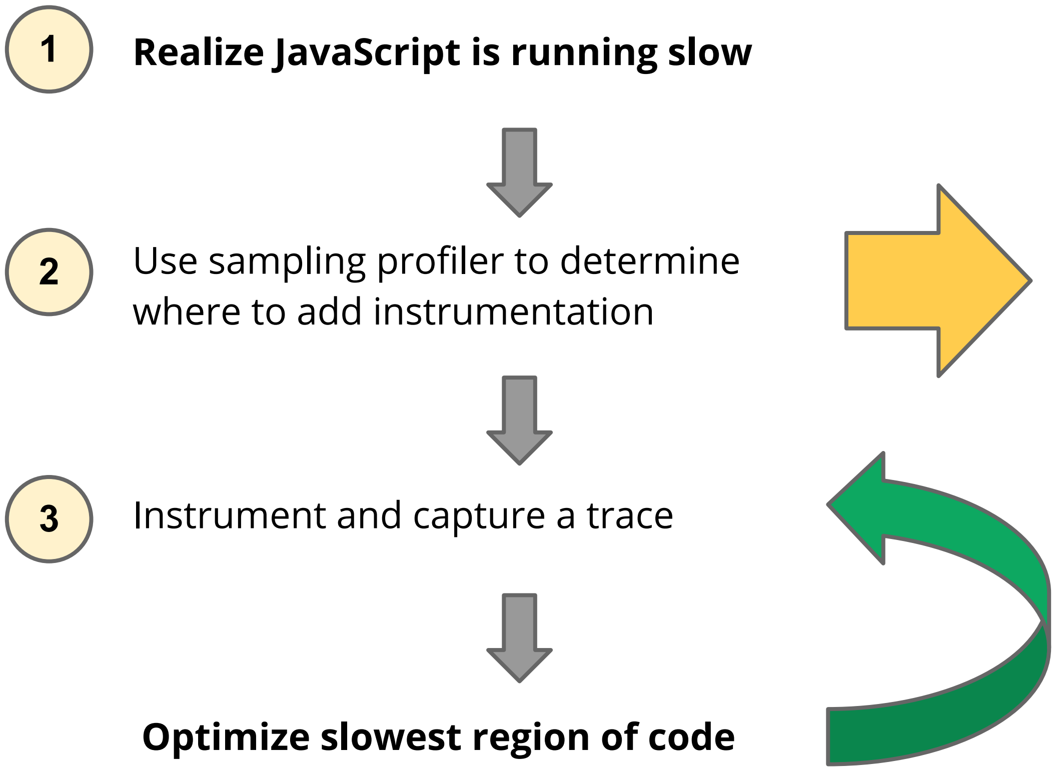
Sampling Profiler (Dev Tools)

- (in this case) did not present a clear picture of program execution flow or timings

Structural Profiler (chrome://tracing)

- Clearly showed program execution flow and timings
- Required additional instrumentation

Real-world profiling workflow

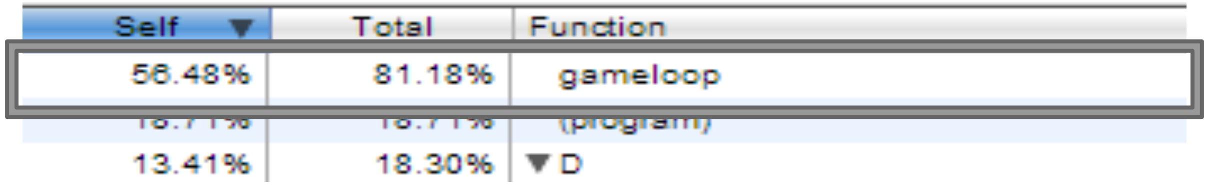


1 **Realize JavaScript is running slow**

2 Use sampling profiler to determine where to add instrumentation

3 Instrument and capture a trace

Optimize slowest region of code



Self ▼	Total	Function
56.48%	81.18%	gameloop
18.71%	18.71%	(program)
13.41%	18.30%	▼ D

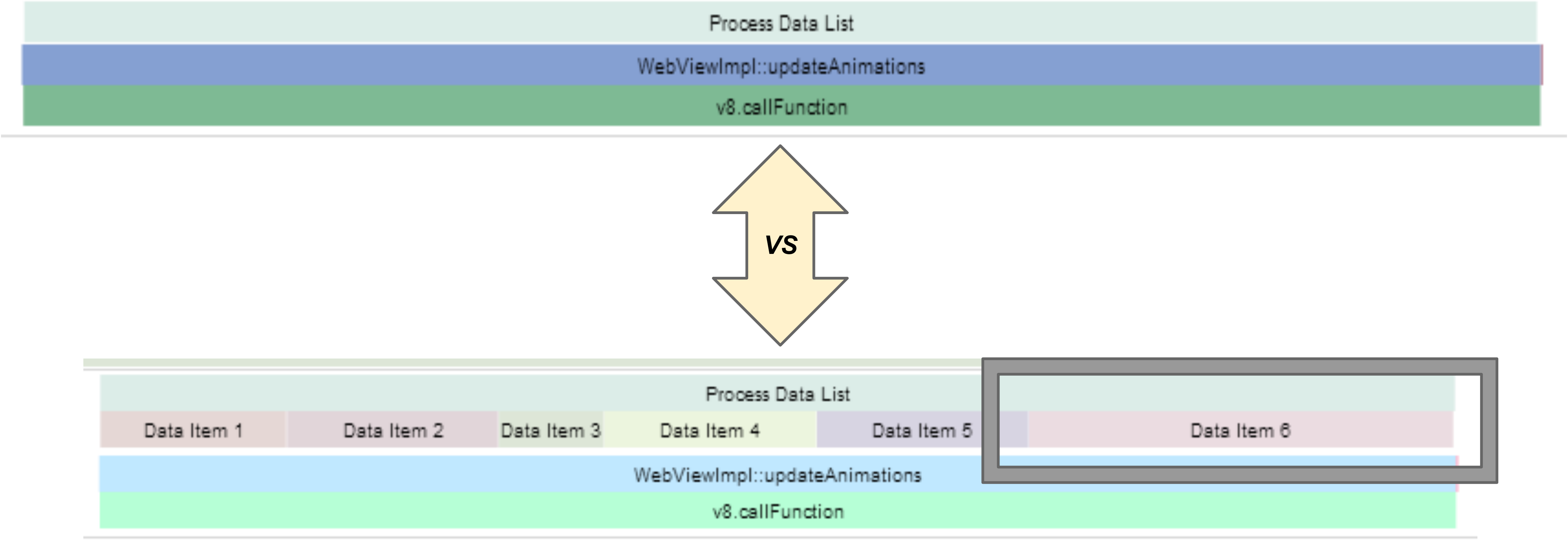
Rinse, lather, repeat...

A few closing tips...

- **start with the sampling profiler...**
- **learn the navigation keys** (WASD) for **chrome://tracing**
- **filter down the recorded trace** to **process ID** you care about
- **console.{time, timeEnd}** pairs can cross function boundaries
  - Start with a large area of code and narrow with a **binary search**!
- Recall that **V8** code **!=** your source code
  - *That is, it's not necessarily the same...*
- You can **save & load** both types of profiling runs
  - *Attach them to your tickets, save for later, etc.*

Think about the data being processed...

- Is one piece of data slower to process than the others?
- Experiment with naming time ranges based on data name



Planning for performance: allocate and follow a budget!!!

- **Budget**
  - Each module of your application should have a time budget
  - **Sum of all modules should be less than 16 ms for smooth apps**

- Track performance data daily (per commit?)
  - **Catch Budget Busters** right away

Oh, and one more thing...

Demo: determining frame rate in chrome://tracing

Questions!

http://goo.gl/OSYJo