Microbenchmarks

Stoyan Stefanov

@stoyanstefanov

perf.now() 2022

The plan

1. CPU: instruction counting
2. Memory: leak detection

Audience

- Single Page Applications
- And other assorted JavaScript-heavy sites/apps

About me

- Engineer on the WebPageTest.org team
- ex-Yahoo, ex-Facebook
- PerfPlanet Calendar

CPU

CPU

- Network/latency-bound
- ...vs CPU-bound

Too much **!@#$%** JavaScript

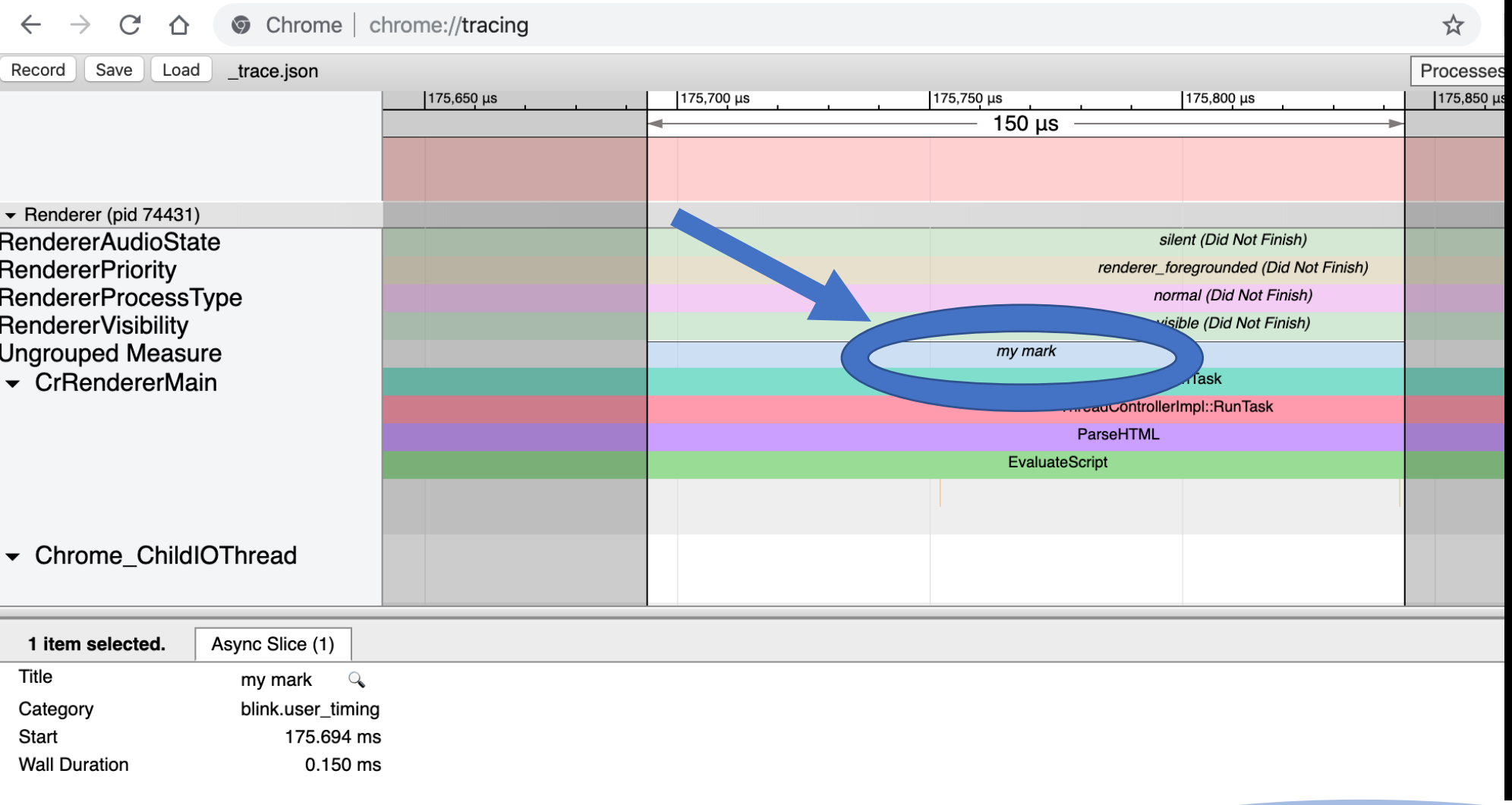
CPU instruction count

- No timings
- Because timing are not stable
- CPU instructions can be stunningly stable
- Per-component analysis

```
<script>
  performance.mark('my mark start');

  // go nuts here...

  performance.mark('my mark end');
  performance.measure(
    'my mark',
    'my mark start',
    'my mark end');
  );
</script>
```



chrome://tracing

trace.json


```
er_timing","name":"my mark start","tts":2945014,"ticount":1368702583,"args":{"data":
er_timing","name":"my mark end","tts":2957481,"ticount":1399566054,"args":{"data":{"
er_timing","name":"my mark","id":"0x11b01232","tts":2957502,"ticount":1399588143,"ar
```

```
import pup from 'puppeteer';

const browser = await pup.launch();
const page = await browser.newPage();
await page.tracing.start(
  {path: 'trace.json'});
await page.goto('test.html');
await page.tracing.stop();
await browser.close();
```

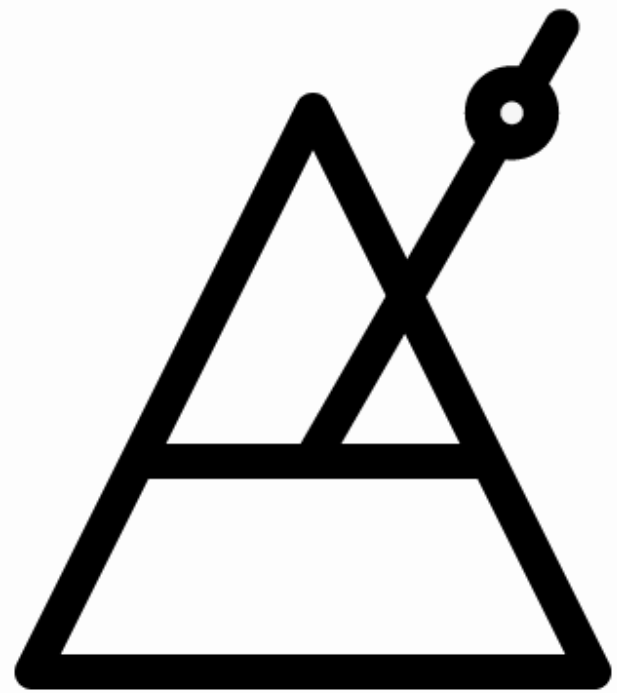
If only there was a tool

to just give me the results...

github.com/stoyan/ticr

ticr

thread instruction counter



$ npx ticr -h

```
Usage: ticr [options] [command]

CLI for thread instruction counting

Options:
  -V, --version          output the version number
  -m, --marker <char>    marker to look for in the trace file (default: "testmarker")
  -t, --trace <char>     name/path to write the trace file (default: "trace.json")
  -u, --url <char>       URL of a test page (default:
                         "file:///home/s/.npm/_npx/..../ticr/examples/sandbox.html")
  --chrome <char>        path to the Chrome executable
  --runs <int>           How many times to run the test. Each run closes and opens the browser again
                         (default: 3)
  --report-runs <char>   Options: lowest, median, all. (default: "lowest")
  -o, --options          Dump the program options (default: false)
  -h, --help             display help for command

Commands:
  ab <a.js> <b.js>       Run an A/B test by providing URLs to test page (--url option), an a.js and
a b.js
                         JavaScript files
  sandbox                Print out the contents of a sandbox.html to toy with.
                         Example use: `ticr sandbox > sandbox.html`
  support                Tests if `ticount` is supported by the browser
  results                Parses a trace.json to show results
```

Install

```
$ npm i ticr -g
$ ticr -h
```

A ticr’s purpose

```
$ ticr
    --chrome ~/chromium/982481/chrome-linux/chrome
    -u "https://example.org"
    -m testmarker
    -t trace.json
    --runs 3
    --report-run lowest
{
  "tic": 107145
}
```

TLA: TIC

Support

```
$ ticr support
`ticount` is NOT supported, see
https://github.com/stoyan/ticr for ideas

$ ticr support --chrome
~/chromium/982481/chrome-linux/chrome
`ticount` is supported
```

A/B

```
$ tickr ab a.js b.js
  -u "file:///home/s/ticr-main/examples/ab.html"
Results from script A (a.js)
{
  "tic": 131214
}
Results from script B (b.js)
{
  "tic": 395091
}
B uses 3.01 times more CPU instructions than A
```

A sandbox

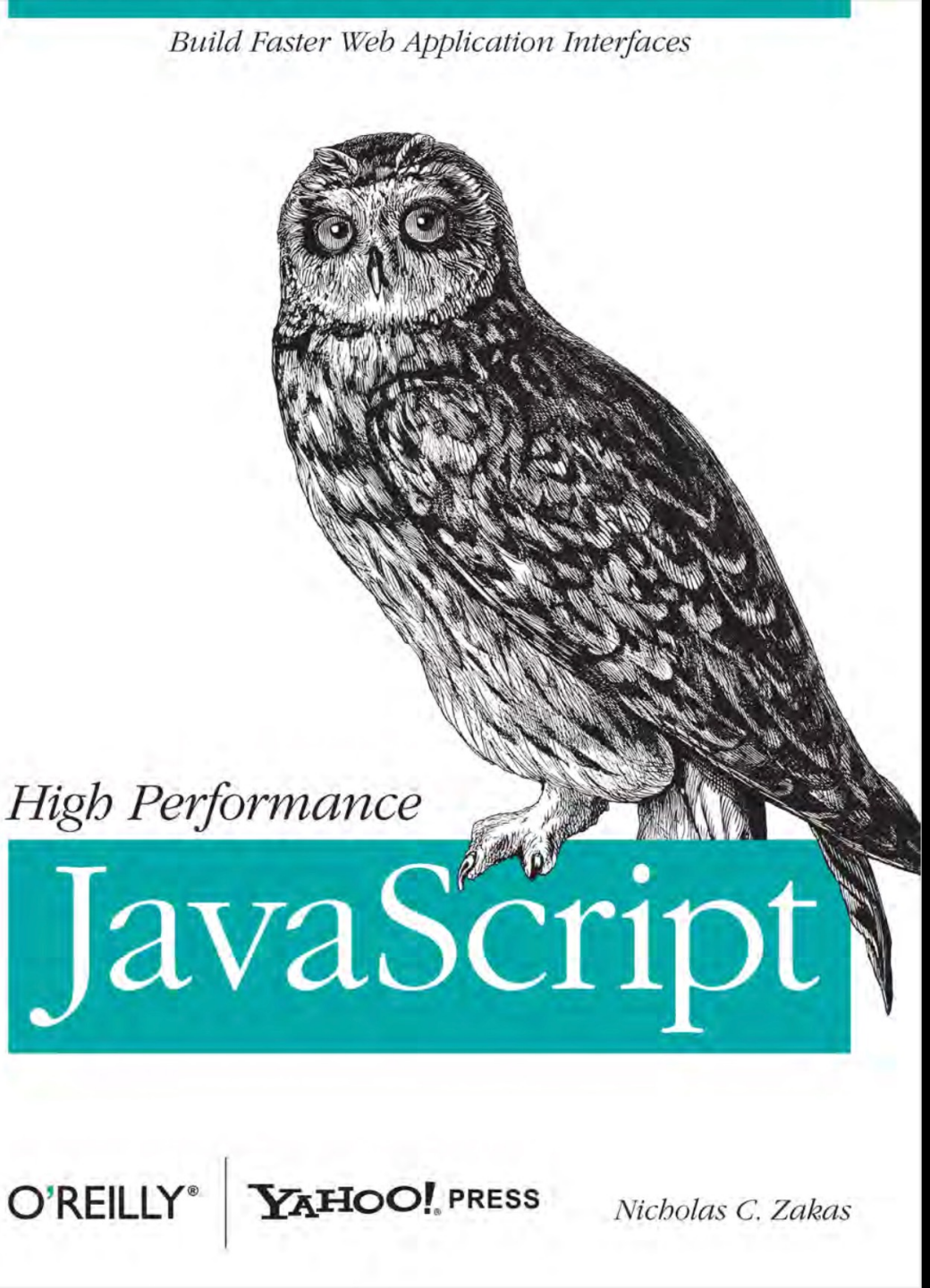
```
$ ticr sandbox
```

```
<!DOCTYPE html>
<html>
  <head>
    ticr's sandbox
  </head>
  <body>
    <!-- Keep this. Making it visible is the signal to Puppeteer that we're done with the test -->
    <div id="done" style="display: none;">done</div>

    <script>
      // we do the actual testing once the browser has had a chance to settle down
      window.onload = () => {
        // start
        performance.mark('testmarker start');
```

More examples

```
$ which ticr
$ cp -r ~/.nvm/.../ticr/examples/ .
```



CHAPTER 3

DOM Scripting

Stoyan Stefanov

DOM scripting is expensive, and it's a common performance bottleneck in rich web applications. This chapter discusses the areas of DOM scripting that can have a negative effect on an application's responsiveness and gives recommendations on how to im-

A/B

```
$ tickr ab layNothing.js layInner.js
  -u "file:///home/s/ticr-main/examples/ab.html"
Results from script A (layNothing.js)
{
  "tic": 131214
}
Results from script B (layInner.js)
{
  "tic": 395091
}
B uses 3.01 times more CPU instructions than A}
```

layNothing.js vs layInner.js

```
function layNothing() {}

function layInner() {
  document.getElementById('done').innerHTML =
    'Well done!';
}
```

layInner vs layOut

```
function layInner() {
  document.getElementById('done').innerHTML =
    'Well done!’;
}
```

```
function layOut() {
  document.getElementById('done').innerHTML =
    'Well done!’;
  return document.body.offsetHeight;
}
```

layInner vs layOut

```
$ tickr ab layInner.js layOut.js

Results from script A (layInner.js)
{ "tic": 396842 }
Results from script B (layOut.js)
{ "tic": 681785 }

B uses 71.8% more CPU instructions than A
```

layInner vs layOutVisible

```
$ tickr ab layInner.js layOutVisible.js

Results from script A (layInner.js)
{
  "tic": 394661
}
Results from script B (layOutVisible.js)
{
  "tic": 4209669,
  "layouts": 1
}
B uses 10.67 times more CPU instructions than A
```

Events in order of badness

- ScheduleStyleRecalculation:
  - touching the DOM
- UpdateLayoutTree:
  - request calculated style
- InvalidateLayout + Layout:
  - request calculated style in visible DOM

DOM vs string appendage

```
function innerHTMLa() {
  for (let count = 0; count < 100; count++) {
    document.getElementById('output').innerHTML +=
      Math.random();
  }
}
function innerHTMLb() {
  let content = '';
  for (let count = 0; count < 100; count++) {
    content += Math.random();
  }
  document.getElementById('output').innerHTML += content;
}
```

innerHTMLa vs innerHTMLb 100x

```
Results from script A (innerHTMLa.js)
{
  "tic": 19550307
}
Results from script B (innerHTMLb.js)
{
  "tic": 969369
}
A uses 20.17 times more CPU instructions than B
```

innerHTMLa vs innerHTMLb 10x

```
Results from script A (innerHTMLa.js)
{
  "tic": 2042749
}
Results from script B (innerHTMLb.js)
{
  "tic": 419271
}
A uses 4.87 times more CPU instructions than B
```

getElementById vs querySelector

```
Results from script A (gela.js)
{
  "tic": 197247
}
Results from script B (gelc.js)
{
  "tic": 196310
}
A uses 0.48% more CPU instructions than B
```

Arguments about arguments

```
function a() {
  return arguments[0] + arguments[1];
}
function b(a1, a2) {
  return a1 + a2;
}
function c(...argh) {
  return argh[0] + argh[1];
}
```

arguments[] is more expensive

```
Results from script A (a.js)
{
  "tic": 145817
}
Results from script B (b.js)
{
  "tic": 141066
}
A uses 3.37% more CPU instructions than B
(…rest uses 5.39% more CPU instructions named args)
```

HTML Collections

- Array-like and “live”, so they are slow
- Make them real arrays instead

```
function collecta() {
  let all = document.getElementsByTagName('div');
  for (let a = 0, len = all.length; a < len; a++) {
    i++;
  }
}
```

HTML Collections

```
function collectb() {
  let all = Array.from(
    document.getElementsByTagName('div’)
  );
  for (let a = 0, len = all.length; a < len; a++) {
    i++;
  }
}
```

Array.areYouKiddingMe()?

```
Results from script A (collecta.js)
{
  "tic": 741704
}
Results from script B (collectb.js)
{
  "tic": 3944751
}
B uses 5.32 times more CPU instructions than A
```

HTML Collections

```
function toArray(col) {
  const a = [];
  for (let i = 0, len = col.length; i < len; i++) {
    a[i] = col[i];
  }
  return a;
}
// only 2.2x worse compared to over 5x
```

Enough examples

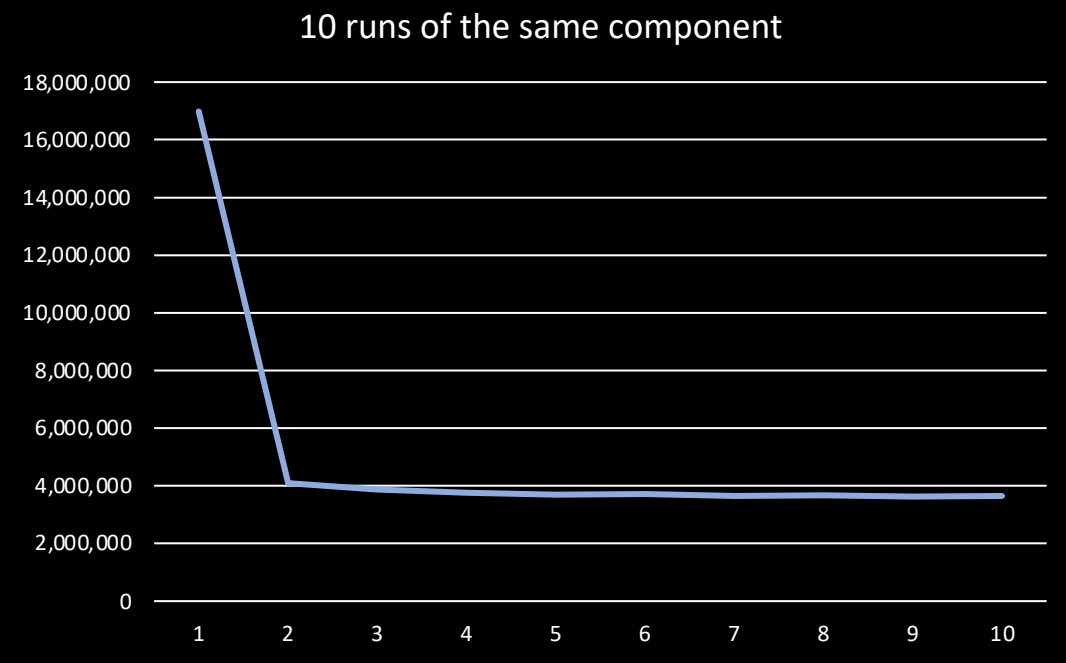
- You can play with these and more in the repo
- But I am looking forward to more findings coming from this community
- Explore CSS' effects on the CPU too?

An idea: Run on every diff

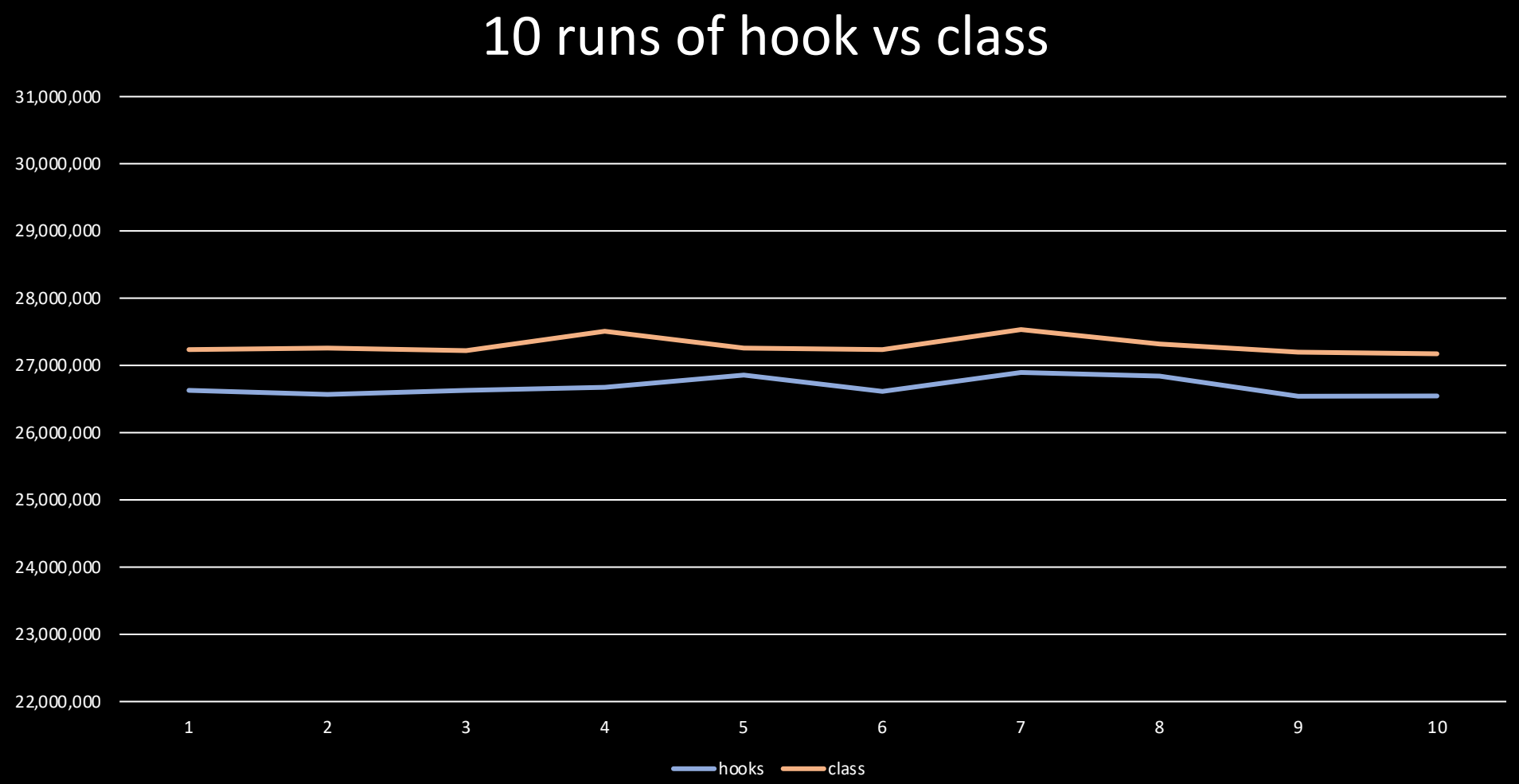
- Source control hook
- Before/after profiling data
- Profile components in isolation
- Using “style guide” or ”design system” examples

Devil’s in the details

- Think about what you’re measuring: browsers do optimize code they see more than once
- Async work not captured



Strong signal



Supporting info

- Write more markers to the trace, e.g. DOM count, React commits
- Count layouts (reflows)
- Take screenshots

Available now? Good news!

- Chrome 78+: look for `ticount/tidelta`
- Linux only

Now for the not-so-good news...

- Removed since Chrome 10X
- Works in v101, doesn’t in v104
- https://crbug.com/1355570

Memory Leaks

You Retweeted

Shaw
@shshaw

I keep calling `performance.now()` but my JavaScript isn't getting any faster. What gives?

7:14 AM · Jul 6, 2022 · Twitter Web App

42 Retweets **7** Quote Tweets **634** Likes

Memory Leaks

Did you know?

- Reporting API
- Your app can beacon back ”oom” and “unresponsive” browser crashes

How do you detect memory leaks

- Option A: you phone a friend
- Option B: take memory heap snapshots
  1. Load a starting page
  2. “navigate” to the target page/interaction
  3. navigate back to square 1
  4. Diff the snapshots from steps #1 and #3.
     If not the same, memory may have leaked.

If only there was a tool

to just give me the results...


if only...
SpeedCurve
performance
.now()

github.com/facebook/memlab

Memlab

- Command-line tool
- Uses Puppeteer to load a page and navigate forward-then-back
- Diffs heap snapshots

A scenario .js

```
function url() {
  return 'https://example.org/';
}
async function action(page) {
  await page.click('button[id="Next"]');
}
async function back(page) {
 await page.click('button[id="Home"]');
}
module.exports = {action, back, url};
```

$ memlab run --scenario ~/scenario.js

```
lgong-mbp:memlab-meta lgong$ memlab run --scenario ~/memlab/scenarios/detached-dom.js
page-load[6.5MB](baseline)[s1] > action-on-page[6.6MB](target)[s2] > revert[7MB](final)[s3]
```

1

```
total time: 50.5s
Memory usage across all steps:
8.0 _________
7.2 _________
6.3 _________
5.4 _________
4.5 _________
3.6 _________
2.8 _________
1.9 _________
1.0 _________
    1  2  3
```

```
MemLab found 1 leak(s)
```

2

```
--Similar leaks in this run: 1024--
--Retained size of leaked objects: 143.3KB--
```

3

```
[Window] (native) @35865 [10.7KB]
  --map (internal)--->  [system / Map] (hidden) @50289 [208 bytes]
  --prototype (internal)--->  [Window / http://localhost:3000] (object) @9821 [235.1KB]
  --leakedObjects (property)--->  [Array] (object) @168885 [148.5KB]
  --0 (element)--->  [Detached HTMLDivElement] (native) @168751 [140 bytes]
```

The problem

```
window.leakedObjects = [];

for (let i = 0; i < 1024; i++) {
  window.leakedObjects.push(
    document.createElement('div’)
  );
}
```

Results!

- Can be overwhelming
- Grouping of finding
- Custom analyzers

Results!

- Can be surprising
- Is this a memory leak?

```
let obj = {};
console.log(obj);
obj = null;
```

Wrapping up

Action items

- Let’s count CPU instructions to:
  - Find expensive parts/components of the app
  - Keep track as the app matures
  - And (in our free time) research better JS practices
- Let’s bring back thread instructions counting to Chrome (and hopefully other browsers)
- Adopt Reporting API to unearth OOM crashes
- Investigate where memory goes

Big thank-yous!

- Benoit Girard @b56girard
- Liang Gong
- Andrew Comminos @acomminos
- Eric Seckler

Thank you!

@stoyanstefanov

https://phpied.com

Web Performance Calendar

The speed geek's favorite time of year

https://calendar.perfplanet.com/contribute/

31st
Dec 2021

A Unified Theory of [illegible]

by Alex Russ[illegible]

[illegible]anner Hodges asked for what
[illegible]would like for the holidays: a unified theory of
[illegible]pose four key ingredients: Definition: What is
[illegible] beyond page speed? What, in particular, is “web performance”?
[illegible]rpose: What is the purpose of [...]

31st

Exposing mid-resource LCP values

by Yoav Weiss

Thank you!

@stoyanstefanov

https://phpied.com