Lecture 21 — Performance Case Studies, Performance Culture

Patrick Lam
patrick.lam@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

September 4, 2022

Case Study: Firefox Quantum

Some Firefox Perf Improvements, per Mike Conley

- don't animate out-of-view elements
- move db init off main thread
- keep better profiling data
- parallel painting for macOS
- lazily instantiate Search Service
- halve size of the blocklist
- refactor to reduce main-thread IO
- don't hold all frames of animated GIFs/APNGs in memory
- eliminate unnecessary hash table
- use more modern compiler

`https://mikeconley.ca/blog/2018/02/14/firefox-performance-update-1/`

We've seen this before:

- do less work (or do it sooner/later);
- use threads (move work off main thread);
- measure performance;

Which of the updates fall into which categories?

Some Firefox Perf Improvements, per Mike Conley

- don’t animate out-of-view elements
- move db init off main thread
- keep better profiling data
- parallel painting for macOS
- lazily instantiate Search Service
- halve size of the blocklist
- refactor to reduce main-thread IO
- don’t hold all frames of animated GIFs/APNGs in memory
- eliminate unnecessary hash table
- use more modern compiler

How?

- do less work (or do it sooner/later);
- use threads (move work off main thread);
- measure performance;

Tab Warming

https://mikeconley.ca/blog/2018/01/11/making-tab-switching-faster-in-firefox-with-tab-warming/.

"Maybe this is my Canadian-ness showing, but I like to think of it almost like coming in from shoveling snow off of the driveway, and somebody inside has *already made hot chocolate for you*, because they knew you'd probably be cold." — Mike Conley

How Can Tab Warming Work?

Before: Firefox requests paint of newly-active tab, and then waits for the result before switching.

Idea: reduce user-visible latency by predicting an imminent tab switch.

Q: How can we predict the future?

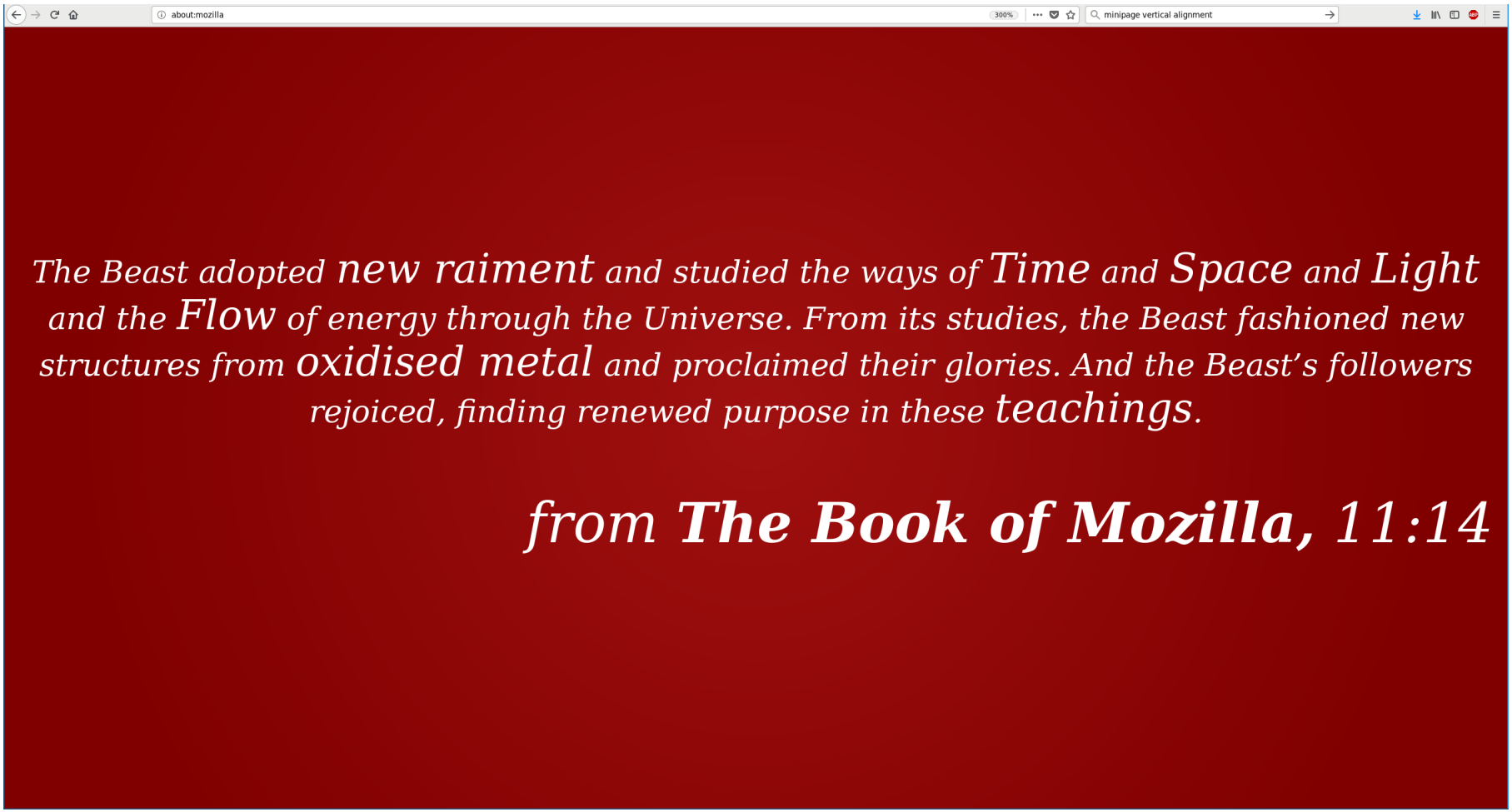
Q': How can we predict which tab will be switched to?

Predicting the future

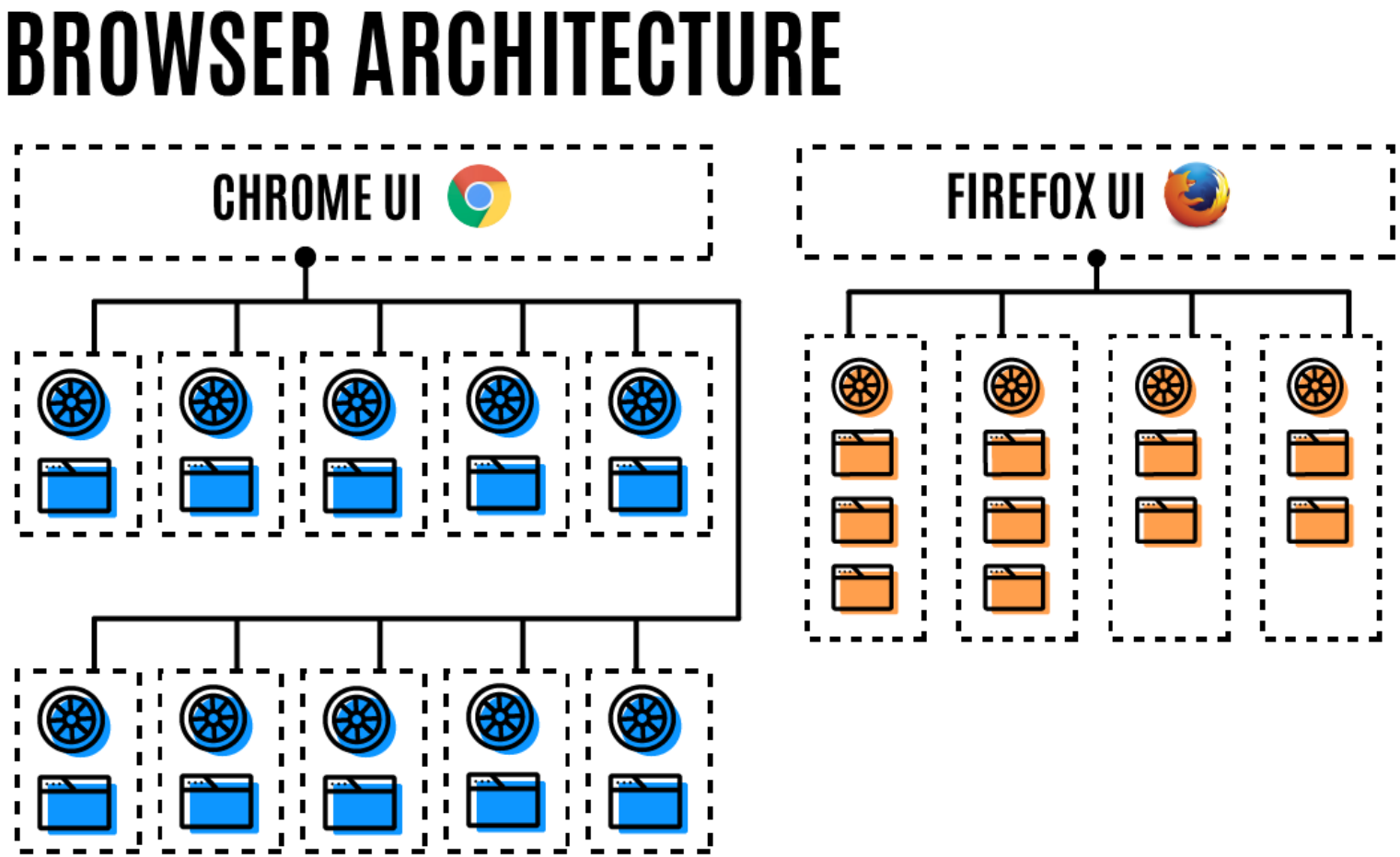
A: When the user has a mouse, then the mouse cursor will hover over the next tab.

> *Assuming a sufficiently long delay between hover and click, the tab switch should be perceived as instantaneous. If the delay was non-zero but still not long enough, we will have nonetheless shaved that time off in eventually presenting the tab to you.*
> *And in the event that we were wrong, and you weren't interested in seeing the tab, we eventually throw the uploaded layers away.*

Blog post does not report performance numbers.


about:mozilla
300%
minipage vertical alignment
The Beast adopted new raiment and studied the ways of Time and Space and Light and the Flow of energy through the Universe. From its studies, the Beast fashioned new structures from oxidised metal and proclaimed their glories. And the Beast's followers rejoiced, finding renewed purpose in these teachings.
from The Book of Mozilla, 11:14

Electrolysis (2017): multiple OS-level processes.
(Think about threading models).



Chrome vs Firefox

Chrome: 1-process-per-tab.
Firefox: 4 shared content processes.

Firefox uses less memory (has less render state).

Electrolysis challenges:
internal architecture, and add-ons.

Electrolysis vs Flow

Two different Firefox projects:

Electrolysis = split across processes
Quantum Flow = leverage multithreading
(using Rust's "fearless concurrency"),
plus other improvements.

Quantum Flow

Steps:

1. Measure slowness & prioritize
2. Gather help
3. Fix all (well, some of) the things!

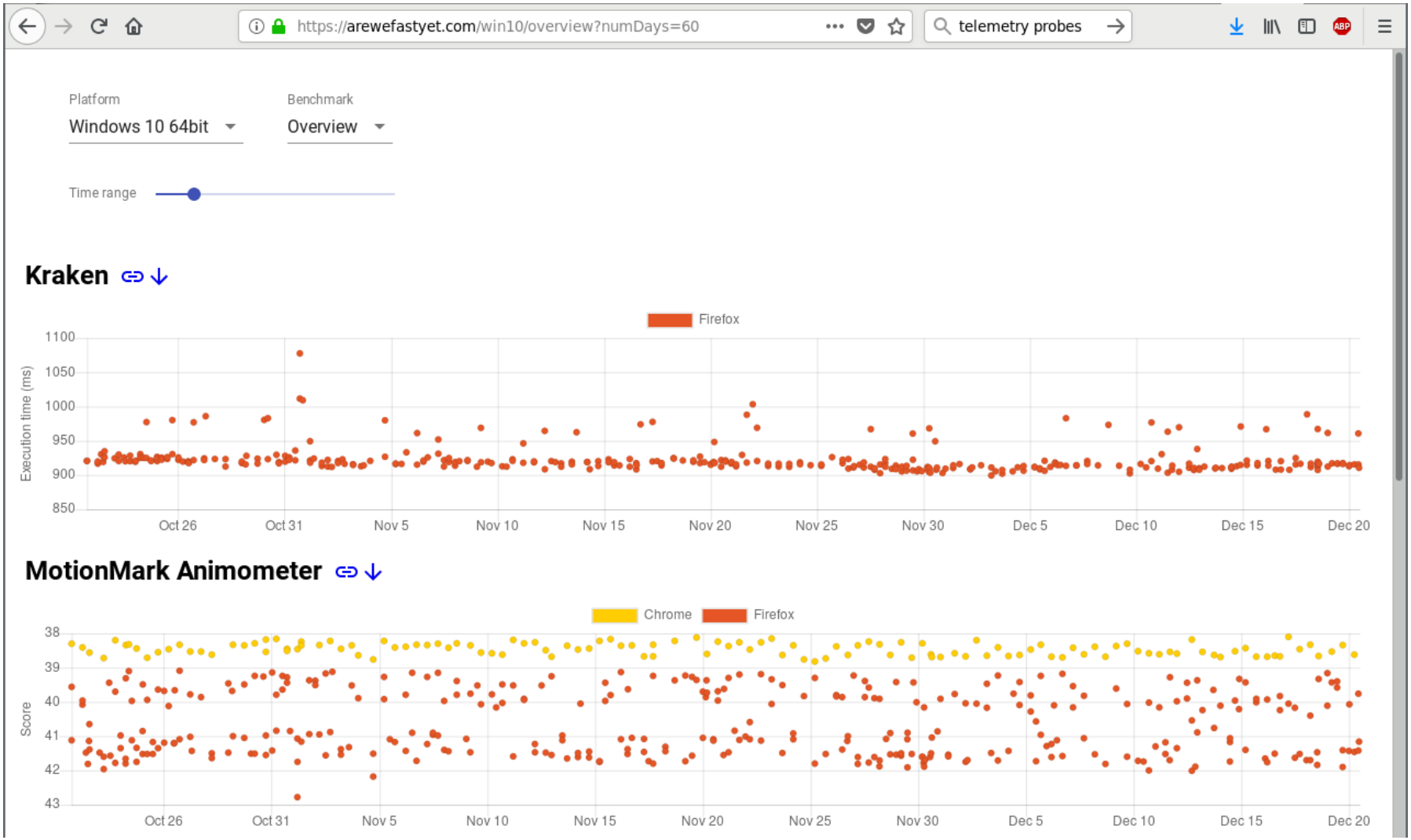
In 6 months:

prioritized 895 bugs, fixed 369.

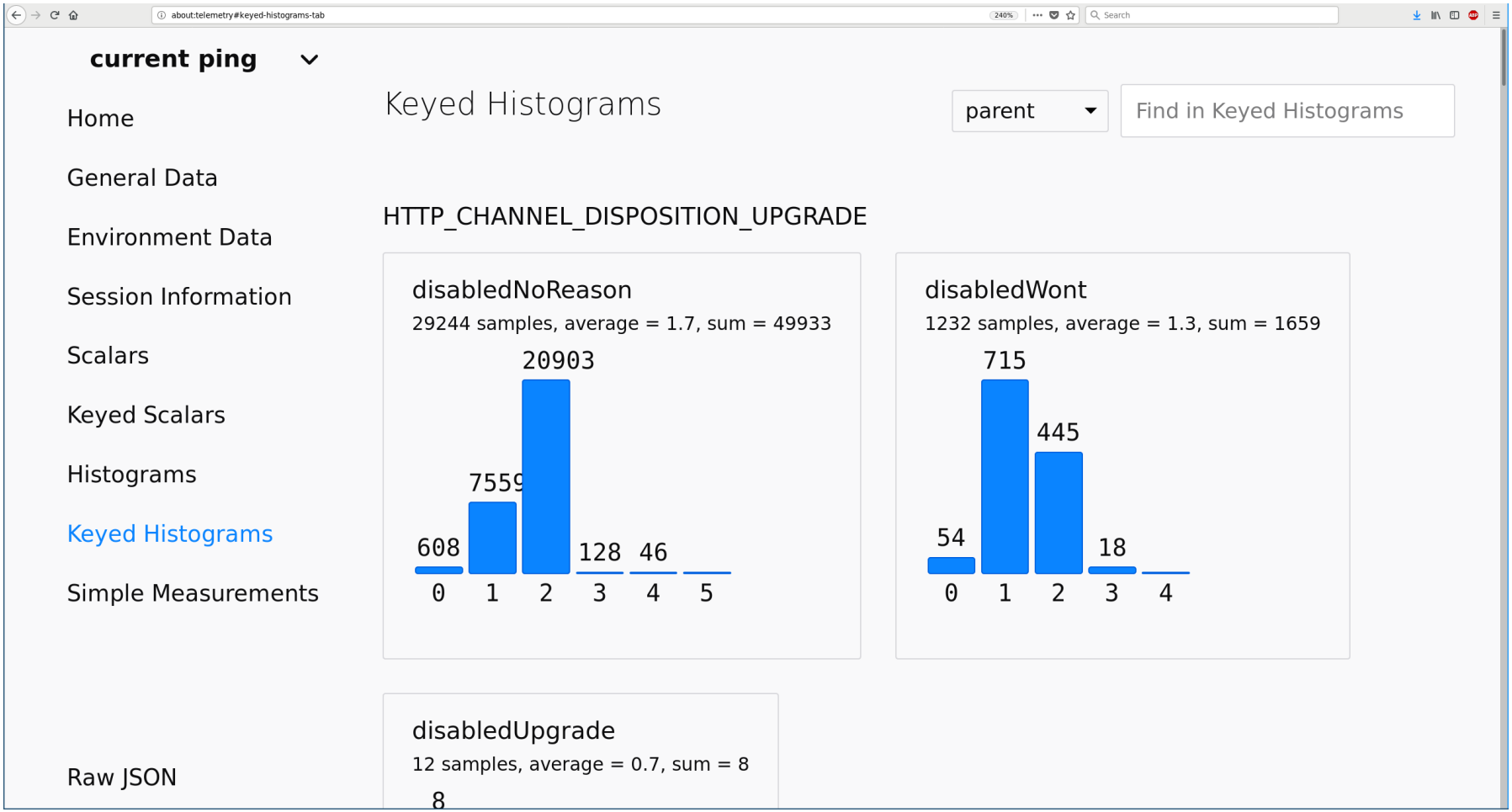
Key tool:

“Quantum Flow Engineering Newsletter”.

arewefastyet dot com



about:telemetry



About Telemetry

Idea: Ask questions first, act second.

Collect data about Firefox usage, then start hacking.

100s of GBs of anonymous metrics/day,
publicly available.

Analogous to CPU profiling, but massively distributed.

- collected much less often than CPU profiling data, but at much broader scope.

`https://telemetry.mozilla.org/`

Example Telemetry Questions

- Is Firefox the user's default browser? (69% yes)
- Does e10s make startup faster? (no, slower)
- Which plugins tend to freeze the browser on load? (Silverlight and Flash)

Can also see evolution of data over time.

Devs can propose new probes;
reviewed for data privacy plus normal code review.

ping!

Firefox sends pings:

- “main ping” every 24 hours;
- upon shutdown;
- upon environment change;
- upon abnormal shutdown.

Presumably compressed JSON to Mozilla servers.

common ping structure

```
{
  type: <string>, // "main", "activation", "optout", ...
  id: <UUID>, // a UUID that identifies this ping
  creationDate: <ISO date>, // the date the ping was generated
  version: <number>, // the version of the ping format

  application: {
    architecture: <string>, // build architecture, e.g. x86
    buildId: <string>, // "20141126041045"
    // etc
  },

  clientId: <UUID>, // optional
  environment: { ... }, // optional, not all pings contain
  payload: { ... }, // actual payload data for this ping type
}
```

Ping contents

1. Scalars (counts, booleans, strings)
2. Histograms = bucketed data (like grade distributions)

Both scalars and histograms can be keyed,
e.g. how often searches happen for which search engines.

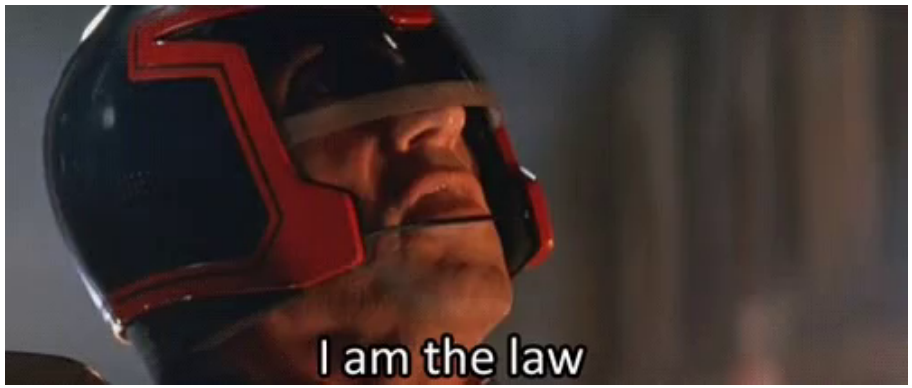
Part II

Laws of Performant Software

Laws of Performant Software

Suppose you want to write fast programs...
And you like checklists and handy rules.

If so, you are in luck, because there is Crista's Five Laws of Performant Software!



Rule 1: Knowledge is Power

1. Programming language $<<$ Programmers' awareness of performance.

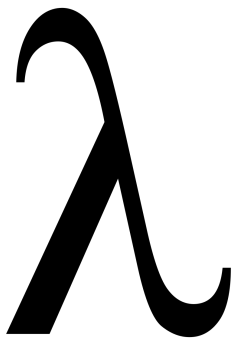
There is no programming language that is magic, whether good or evil.

All the major programming languages allow you to write programs that perform well or badly.

High level languages give you lots of options...
Do I use an array? A vector? A list?

What do they do behind the scenes?
Is there a better way?

Rule 1 Asterisk



Some languages lend themselves better to parallelization than others.

A language may force a certain way of thinking, based on its rules (e.g., functional programming languages).

But there is no reason why the way of thinking can’t be applied in another language.

Rule 2: Nonlinear Output

2. $d(f^\tau(x), f^\tau(y)) > e^{\alpha\tau} d(x, y)$ **or small details matter.**

If two versions of the code are x and y, the difference between the performance outcomes $f(x), f(y)$ is much larger than the difference between the code.

Did you fix a memory leak? The addition of one `free()` call is a single line code change but can, in the long run, have a dramatic impact on performance.

Don't overlook the small stuff!

Rule 3: Establish Boundaries

3. corr(performance degradation, unbounded resource usage) > 0.9.

There is a very high correlation between performance degradation and unbounded use of resources.

Often times we focus on functionality: the software must have the following 847 251 features!

But if you want a program that scales you need to think in terms of operation, not functionality.

Rule 3: Establish Boundaries

Resources need to be limited.

If there aren't hard limits, eventually a resource will be exhausted.

If the program starts threads, use a thread pool and the thread pool should have a fixed size.

Is there a cache? It needs a maximum size.

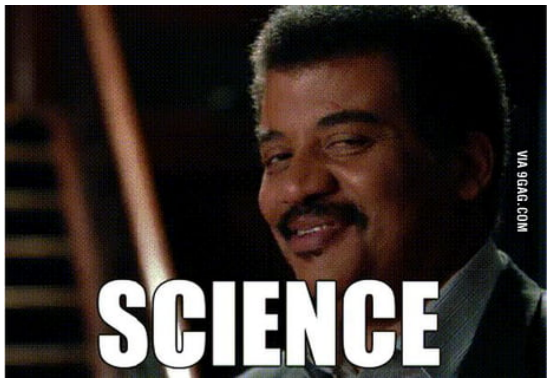
If you need to read input, don't use a function that reads an entire line (of arbitrary length).

Rule 3: Establish Boundaries

Furthermore your program needs design effort given to what happens when resources are exceeded.

So you decide to set a request queue size; once that queue is full, further requests are rejected in some well-understood manner.

Rule 4: Do Experiments



If you want your code to be faster you have to know why it is slow.

It’s okay not to know the answers, but not knowing how to find out is a problem.

Don’t guess; measure.

Rule 5: You Can’t Outrun the Fork

5. N*bad != good.

No amount of nodes, cores, memory, etc, will save you from poorly-written code.

Throwing more hardware at the problem is expensive and ineffective in the long term.

Bad code is still bad no matter how much hardware it runs on.

Part III

Performance Tips

Magnitude

Understand the order of magnitude that matters.

Perhaps you are writing code where 100 CPU cycles matters.

If so, a function that will acquire a lock through some sort of shared-memory interlocked instruction then this instruction is performance murder.

And it might be even worse if lock acquisition fails and you get blocked for 100 000 cycles trying to acquire the lock...

On the other hand, if you have a network intensive code then 100 cycles do not matter even the smallest bit because it is lost in the noise of the network.

Worst Case Scenario

Plan for the worst case scenario (not the nuclear apocalypse).

You can end up paying many more orders of magnitude in cost than you expected under ordinary circumstances.

No doubt you have seen these sorts of things as well and it is an awful user experience.

The spinning beach ball of death and the “Not Responding” added to the title bar are all symptoms of this situation.

Acknowledge Asynchrony

A lot of things are asynchronous without declaring themselves to be.

Acknowledge Asynchrony

Suppose that Jordan is writing code to return a list of fonts to be used in the UI.

The code checks a local font cache, and if that is already initialized, returns the fonts found in the cache in a List object.

If the cache is not initialized, however, the data needs to be loaded, perhaps from the printer.

Morgan intends to use this list of fonts in the UI of the program but is unaware that in certain circumstances this call will take much longer.

Acknowledge Asynchrony

Supposing that the API just returns a `List<Font>`, it is not obvious to Morgan where these come from.

And in local testing on the dev machine with nothing on it and nothing else doing and the printer is always turned on.

Thus even if the cache is not initialized it is “fast enough”.

But in the real world, of course, the printer will be down and unicorns are nowhere to be found and it runs up against some hard timeout of 20 seconds.

Acknowledge Asynchrony

If Jordan's API now has a return type of `Task<List>` then it is obvious to Morgan that there is something asynchronous going on here some of the time.

With this information in hand, the UI won't wait for this task.

We might even convince Jordan to give us partial results (so we can draw the UI elements as backing data arrives) or at the very least, a progress bar.

Users love progress bars.

They are wildly inaccurate and not super informative even if they are accurate, but at least they give the user the impression that something is happening.

Part IV

Performance Culture

Would you work for him?

Performance Problem Check

The author's recommendations of warning signs that performance culture is off the rails:

- Answering the question, "how is the product doing on my key performance metrics," is difficult.
- Performance often regresses and team members either don't know, don't care, or find out too late to act.
- Blame is one of the most common responses to performance problems (either people, infrastructure, or both).

Performance Problem Check

The author's recommendations of warning signs that performance culture is off the rails:

- Performance tests swing wildly, cannot be trusted, and are generally ignored by most of the team.
- Performance is something one, or a few, individuals are meant to keep an eye on, instead of the whole team.
- Performance issues in production are common, and require ugly scrambles to address (and/or cannot be reproduced).

Human After All

As tempting as it is to say these are all technical issues, they are really human problems.

It would obviously be preferable to start with a good performance culture in the first place, but that is not always going to happen.

But suppose you are already in a hole. The usual course of action is to stop and say to yourself “I appear to be in a hole... I know, I’ll dig my way out!”.

The right thing to do would be stop digging, of course...

Change Comes From Within

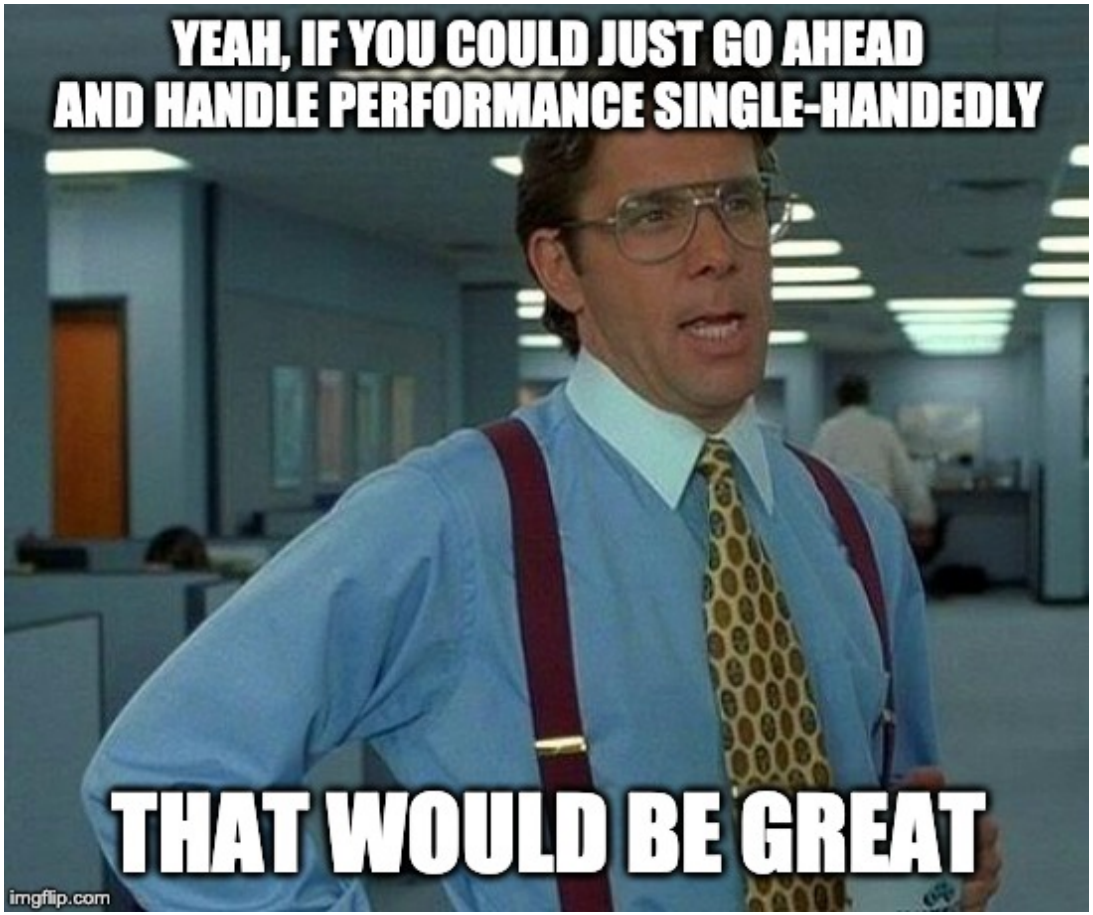
Change has to come from both the top down and the bottom up.

Management needs to make performance a priority: ask questions, demand rigour...

While at the same time, people doing the development need to understand the performance of what they write, making practice improvements.

Both sides need to have zero tolerance for regression.

Run, Run Away


YEAH, IF YOU COULD JUST GO AHEAD
AND HANDLE PERFORMANCE SINGLE-HANDEDLY
THAT WOULD BE GREAT
imgflip.com

There Can’t Be One

If only one person is responsible for performance, this will not work. One person could not reasonably keep up with the rest of the team.

If the other developers don’t have performance in mind when writing, it is a big waste of time for the performance person.

If this isn’t convincing, think about unit tests.

Could you really “outsource” the writing of all unit tests to one person? And would code quality be the same if you did?

Management Matters

Managers need to budget time, reward the work, and encourage performance work.

Managers who don’t understand performance culture are likely to be caught by surprise and blame things.

Consider the story of Managers A and B (see the notes).

But this is a startup, you object, why does it matter, because we need the minimum viable product and we needed it yesterday.

And also, VCs like features.

Well maybe, but architecture matters.

There's time to fix things later, maybe, but you can really cripple your startup by choosing the wrong things early on.

Performance Tests

After a discussion of the why and the what, it is time to think about how to make it systematic.

If it is not systematic, it will not get done and it will be dropped when in a time crunch. And development teams are always in a time crunch!

This is why best practices have automated unit tests that run on every commit.

If something has significantly regressed performance, the build failed and the commit is no good.

This obviously requires that the tests be meaningful, testing effectively and not overly noisy.

Performance Tests

Performance tests may be a bit too large and difficult to run for every commit or every build.

Although it would be nice to know exactly which commit is the cause of the problem, you may want a test that runs overnight to see the long term trends.

And thus a 12-hour test is probably unsuitable for this.

The solution is, obviously, levels of test.

Zero Tolerance

A zero tolerance rules advocated: anything that significantly regresses performance is reverted and re-worked.

No exceptions, no questions, no pleading one's case, etc.

If exceptions are allowed, exceptions quickly become the rule, and promises to fix something later are procrastinated endlessly.

Obviously when code does more it should be expected to take longer, but then the test(s) should be adjusted as well.

Metrics

There are, roughly speaking, two categories of metrics:

- *Consumption*: measure resources consumed by running a test.
- *Observational*: measure the outcomes of running the test, from the view of outside the system.

Observational Metrics

It might seem like observational metrics are all that we care about.

End users care about the amount of time it takes to complete their work, or management/sales/marketing/C-levels care about the throughput and that's it.

This matters, sure, but it's not enough to know the outcome; the consumption metrics are necessary to break down why the total time is what it is.

Observational Metrics

Suppose a change is committed and the tests run and the total time it takes increases by 10%.

If that's all you know, you know things got worse but have no data as to why.

If you see that memory allocations are reported and you have now allocated more memory than before, at least you have a theory to start with.

It might not be that, it might be something else entirely, but it's a place to start.

Printf Debugging

Does this remind you of `printf` debugging? It's a little bit like that.

But then again, you want to use this sort of thing for long running tests anyway.

If the test is going to run for 4h it would be helpful to know where exactly in that four hours it went off the rails (on a crazy train!).

Variability

There's one more consideration and that is the variability between runs. In the ideal situation, there is consistent performance every run.

That is probably too much to ask for and there will be some natural variation due to nondeterminism in computers.

But a change that causes a wild change in the variance is also no good.

A test that sometimes finishes in $0.5x$ time and sometimes in $2x$ time has a high variance and this will not be acceptable.

Dig Deep

Be ready to dig deep: suppose a developer has introduced a change that is slow under rare circumstances.

Those rare circumstances are not triggered until *n* years later when a second developer introduces another change.

The second change will be identified, at least initially, as the “problem”.

There’s two ways to go about it; either fix the initial change or work around it.

It’s not about blame or not about fault; it’s about improving the performance of the software.