

# Lecture 19 — Performance Case Studies

Patrick Lam  
patrick.lam@uwaterloo.ca

Department of Electrical and Computer Engineering  
University of Waterloo

October 31, 2019

# 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/>

- 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;

<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

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?

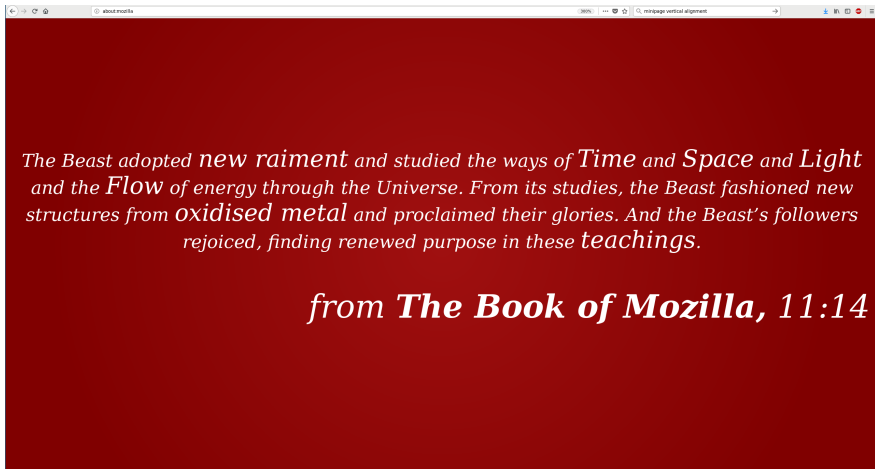
A: When the user has a mouse, then the mouse cursor will hover over the next tab.

*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.*

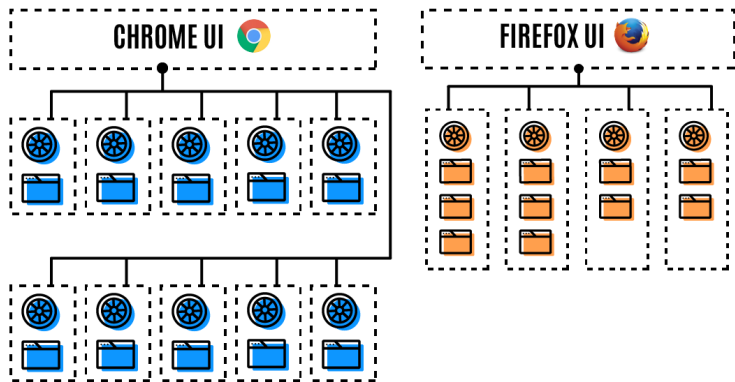
Blog post does not report performance numbers.





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

## BROWSER ARCHITECTURE



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.

Two different Firefox projects:

Electrolysis = split across processes

Quantum Flow = leverage multithreading  
(using Rust's “fearless concurrency”),  
plus other improvements.

## Steps:

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

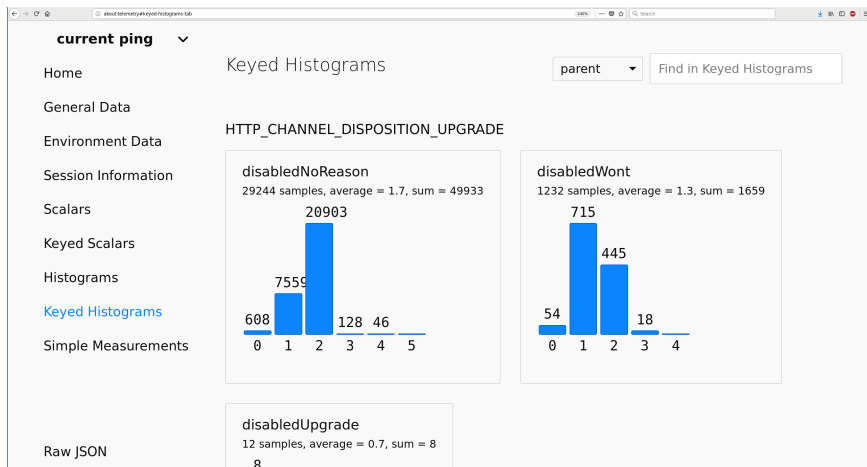
## In 6 months:

prioritized 895 bugs, fixed 369.

## Key tool:

“Quantum Flow Engineering Newsletter”.





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/>



- 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.

Firefox sends pings:

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

Presumably compressed JSON to Mozilla servers.

```
{
  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
}
```

- 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

Lower Level = Faster?

So far, we've only seen C—we haven't seen anything complex.

C is low level, which is good for learning what's really going on.

Writing compact, readable code in C is hard.

Common C sights:

- **#define macros**

- **void\***

C++11 has made major strides towards readability and efficiency (it provides light-weight abstractions).

## 1 Sorting

## 2 Vectors vs. Lists



Sort a bunch of integers.

In **C**, usually use `qsort` from `stdlib.h`.

---

```
void qsort (void* base, size_t num, size_t size,  
           int (*comparator) (const void*, const void*));
```

---

- A fairly ugly definition (as usual, for generic C functions)

---

```
#include <stdlib.h>

int compare(const void* a, const void* b)
{
    return (*((int*)a) - (*((int*)b));
}

int main(int argc, char* argv[])
{
    int array[] = {4, 3, 5, 2, 1};
    qsort(array, 5, sizeof(int), compare);
}
```

---

- This looks like a nightmare, and is more likely to have bugs.

C++ has a sort with a much nicer interface<sup>1</sup>...

---

```
template <class RandomAccessIterator>
void sort (
    RandomAccessIterator first ,
    RandomAccessIterator last
);

template <class RandomAccessIterator, class Compare>
void sort (
    RandomAccessIterator first ,
    RandomAccessIterator last ,
    Compare comp
);
```

---

---

<sup>1</sup>...nicer to use, after you get over templates (they're useful, I swear).

---

```
#include <vector>
#include <algorithm>

int main(int argc, char* argv[])
{
    std::vector<int> v = {4, 3, 5, 2, 1};
    std::sort(v.begin(), v.end());
}
```

---

**Note:** Your compare function can be a function or a functor.  
By default, sort uses operator< on the objects being sorted.

- Which is less error prone?
- Which is **faster**?

[Shown: actual runtimes of qsort vs sort]

The C++ version is **twice** as fast. Why?

- The C version just operates on memory—it has no clue about the data.
- We're throwing away useful information about what's being sorted.
- A C function-pointer call prevents inlining of the compare function.

OK. What if we write our own sort in C, specialized for the data?

[Shown: actual runtimes of custom sort vs sort]

- The C++ version is still faster (although it's close).
- However, this is quickly going to become a maintainability nightmare.
  - Would you rather read a custom sort or 1 line?
  - What (who) do you trust more?

Abstractions will not make your program slower.

They allow speedups and are much easier to maintain and read.

Let's throw Java-style programming (or at least collections) into the mix and see what happens.



1 Sorting

2 Vectors vs. Lists

1. Generate **N** random integers and insert them into (sorted) sequence.

**Example:** 3 4 2 1

- 3
- 3 4
- 2 3 4
- 1 2 3 4

2. Remove **N** elements one at a time by going to a random position and removing the element.

**Example:** 2 0 1 0

- 1 2 4
- 2 4
- 2
- 

For which **N** is it better to use a list than a vector (or array)?

## ■ Vector

### ■ Inserting

- $O(\log n)$  for binary search
- $O(n)$  for insertion (on average, move half the elements)

### ■ Removing

- $O(1)$  for accessing
- $O(n)$  for deletion (on average, move half the elements)

## ■ List

### ■ Inserting

- $O(n)$  for linear search
- $O(1)$  for insertion

### ■ Removing

- $O(n)$  for accessing
- $O(1)$  for deletion

Therefore, based on their complexity, lists should be better.

[Shown: actual runtimes of vectors and lists]

**Vectors** dominate lists, performance wise. Why?

- Binary search vs. linear search complexity dominates.
- Lists use far more memory.  
**On 64 bit machines:**
  - Vector: 4 bytes per element.
  - List: At least 20 bytes per element.
- Memory access is slow, and results arrive in blocks:
  - Lists' elements are all over memory, hence many cache misses.
  - A cache miss for a vector will bring a lot more usable data.

- Don't store unnecessary data in your program.
- Keep your data as compact as possible.
- Access memory in a predictable manner.
- Use vectors instead of lists by default.
- Programming abstractly can save a lot of time.