ECE459: Programming for Performance Winter 2020

Lecture 19 — Performance Case Studies

Patrick Lam *2019-10-31*

Making Firefox Fast

Let's look at Mike Conley's Firefox Performance Updates,

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

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

We can categorize most of these updates into the categories we've seen before:

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

Which of the updates fall into which categories?

Tab warming

We continue by examining one particular update, *tab warming*, in detail:

`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 youâĂŹd probably be cold." — Mike Conley

Consider switching tabs. Previously, Firefox would request a paint of the newly-selected tab and wait for the rendering to be available before switching the tab.

The idea is to reduce user-visible latency by predicting an imminent tab switch. How do you know that the user is about to switch tabs? 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 werenâĂŹt interested in seeing the tab, we eventually throw the uploaded layers away.

The blog post does not report performance numbers (but bug 1430160 discusses how to collect them).

Firefox in general

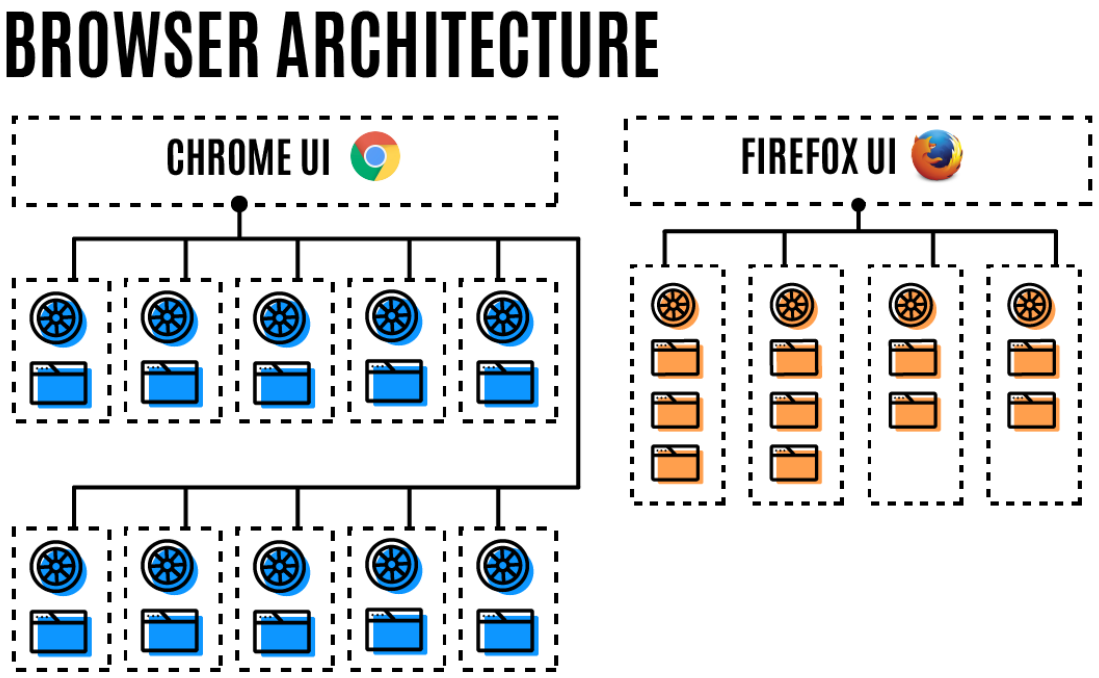
Try: “about:mozilla” in Firefox. On a Quantum Flow-enabled version, you’ll see

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 BeastâĂŹs followers rejoiced, finding renewed purpose in these teachings.

from The Book of Mozilla, 11:14

In 2017, Mozilla released Electrolysis (E10s[1]), which leverages multicore processors by using multiple OS-level processes. (Chrome has always done this, but Firefox attempts to also keep memory usage down[2].) Beyond internal architecture issues, handling Add-Ons (now WebExtensions) was perhaps the most challenging part of going multi-process.

Note the connection to different thread/process models. Chrome is one-process-per-tab, while Firefox multiplexes tabs across the 4 content processes (“hardware threads”, by analogy). Limiting the number of tabs also limits the memory consumption of the browser: we don’t have arbitrary numbers of renderer state.



Source: Ryan Pollock, “The search for the Goldilocks browser and why Firefox might be ‘just right’ for you”,
`https://medium.com/mozilla-tech/the-search-for-the-goldilocks-browser-and-why-firefox-may-be-just-right-for-you-1f520506aa35`

As a crude summary, Electrolysis works on splitting across processes while the newer Quantum Flow leverages multithreading and other improvements. Quantum Flow uses the Rust programming language and its “fearless concurrency” (in Rust-speak). Rust should probably be part of a future revision of the ECE 459 curriculum. But we’ll focus on Firefox here.

[1] `https://blog.mozilla.org/blog/2017/06/13/faster-better-firefox/`
[2] `https://medium.com/mozilla-tech/the-search-for-the-goldilocks-browser-and-why-firefox-may-be-just-right-for-you-1f520506aa`

Quantum Flow

Here's a retrospective of the Quantum Flow project:

```
https://ehsanakhgari.org/blog/2017-09-21/quantum-flow-engineering-newsletter-25
```

To sum up, they formed a small team and did the following.

1. Measure slowness: gather information, instrument Firefox, collect profiling data and measurements. Prioritize issues.
2. Gather help: convince other teams to pitch in with perf improvements. Examples: front-end team (reduce flushes, timers); layout team (reflow performance).
3. Fix all the things! (Or at least the most important ones).

Given the short timeline they gave themselves (6 months) and the limited resources, an important part of their work was convincing others to help. They triaged 895 bugs and fixed 369 of them. The weekly Quantum Flow Engineering Newsletter was a key motivational tool.

After the project wound down, they aimed to distribute responsibility for perf improvements across the entire project.

Firefox Telemetry

Firefox's Telemetry feature collects lots of information from Firefox users. Idea: collect data before hacking away at things. Firefox collects hundreds of gigabytes of anonymous metrics per day while browsing and makes it all available to the public. One can view this as an analogy of CPU profiling on a massively distributed context. This data is collected much less often than CPU profiling data but at a much broader scope.

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

If you are running Firefox and want to see what it is collecting:

```
about:telemetry
```

You can view distributions of telemetry probes (in the form of histograms). You can also make your own dashboard based on Firefox Telemetry data and Mozilla has infrastructure for their developres to formulate and evaluate their own queries.

Example 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 see evolution of data over time.

Firefox developers can propose new telemetry probes which are reviewed for data privacy[3] as well as through normal code review channels.

[3] Mozilla Data Collection Practices: `https://wiki.mozilla.org/Firefox/Data_Collection`

Pings. Firefox phones the data home using so-called "pings". Firefox sends a "main ping" every 24 hours, upon shutdown, environment change, and crash. There are other types of pings as well. Pings get sent either by Firefox or by a helper program, Pingsender, when Firefox isn't running. Presumably they are sent over the network as compressed JSON to a central server.

Here's the common ping structure:

```
{
  type: <string>, // "main", "activation", "optout", "saved-session", ...
  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, currently 4

  application: {
    architecture: <string>, // build architecture, e.g. x86
    buildId: <string>, // "20141126041045"
    name: <string>, // "Firefox"
    version: <string>, // "35.0"
    displayVersion: <string>, // "35.0b3"
    vendor: <string>, // "Mozilla"
    platformVersion: <string>, // "35.0"
    xpcomAbi: <string>, // e.g. "x86-msvc"
    channel: <string>, // "beta"
  },

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

Pings contain scalars (counts, booleans, strings) and histograms. A histogram collects bucketed data (think grade distributions). Both scalars and histograms can be keyed, e.g. how often searches happen for which search engines.

Case Study: Is Lower Level Always Faster?

There's a lot of support for the idea that code written in lower level languages (e.g., choosing C rather than C++) means that your code will be faster. Is that always the case? Language elitism aside – not always!

C++11 has made major strides towards readability and efficiency—it provides light-weight abstractions. We'll look at a couple of examples.

Sorting. Our goal is simple: we'd like to sort a bunch of integers. In C, you would usually just use qsort from `stdlib.h`.

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

This is a fairly ugly definition (as usual, for generic C functions). How ugly is it? Let's look at a usage example.

```
#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 than what we'll see next.

C++ has a sort with a much nicer interface[4]:

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

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

It is, in fact, easier to use:

```
#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. (Don't know what functors are? In C++, they're functions with state.) By default, `sort` uses `operator<` on the objects being sorted.

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

The second question is empirical. Let's see. We generate an array of 2 million ints and sort it (10 times, taking the average).

- qsort: 0.49 seconds
- C++ sort: 0.21 seconds

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?

- Custom C sort: 0.29 seconds

[4] . . . well, nicer to use, after you get over templates.

Now the C++ version is still faster (but it's close). But, 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?

Lesson

Abstractions will not make your program slower.

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

Vectors vs Lists

Consider two problems.

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)?

Complexity analysis. As good computer scientists, let's analyze the complexity.

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.

Reality. OK, here's what happens.

```
$ ./vector_vs_list 50000
Test 1
======
vector: insert 0.1s   remove 0.1s   total 0.2s
list:   insert 19.44s   remove 5.93s   total 25.37s
Test 2
======
vector: insert 0.11s   remove 0.11s   total 0.22s
list:   insert 19.7s   remove 5.93s   total 25.63s
Test 3
======
vector: insert 0.11s   remove 0.1s   total 0.21s
list:   insert 19.59s   remove 5.9s   total 25.49s
```

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.

So, here are some tips for getting better performance.

- 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.
- Often, telling the compiler more gives you better code.
- Data structures can be critical, sometimes more than complexity.
- **Low-level code != Efficient**.
- Think at a low level if you need to optimize anything.
- Readable code is good code—different hardware needs different optimizations.

References