Lazy Parsing CSS [public]

Owner: csharrison@chromium.org
Status: Started
Tracking bug: crbug.com/642722

Overview

Past research (restricted to chromium.org) has shown that for files with many (> 1000) rules, their utilization is very low (< 30%). Because we spend a lot of time parsing CSS in the critical path of page loads, it would be nice to avoid some of this work.

Goal

Improve CSS parsing speed without introducing too much technical complexity.

Proposal

I propose a v1 of lazily parsing StyleRule properties. The majority of the feature is implemented in this CL here. I'll explain what the CL does.

To implement lazy parsing we need to defer parsing work until a rule's properties are needed. This means we have to ensure that everything needed to parse the properties is persisted. This includes:

- **Tokens**: these can be copied without much cost and can be owned by the StyleRule.
  - **Token backing strings**: tokens have StringViews into the decoded bytes of the sheet text. This CL keeps those bytes persisted with the css resource, so we're safe.
  - **Token backing escape strings**: Escape strings are stored separately from the decoded bytes. These need to be stored separately.
- **CSS Parser context**: this is persisted in the StyleSheetContents, which will outlive the StyleRule. Note this context also has a raw pointer to a UseCounter. To avoid UAF we need to hold a reference to the StyleSheetContents.

That's all the context we need. The linked CL has the parser store the relevant information in a LazyPropertyParser, which is sent to and owned by the StyleRule via a new lazy constructor createLazy(). For shared state (like the parsing context), each LazyPropertyParser holds a reference to a LazyParsingState object.

Integration with Streaming Parser

timloh@ is working on a streaming parser for custom property input preservation. This work has a number of benefits to lazy parsing, including:

- Deferred tokenization as well as parsing, which leads to
- Much lower overhead, as we don't need to copy tokens to a separate vector, just pointers into the underlying string. The only overhead at parse time will be allocating the LazyPropertyParser on the Oilpan heap.

Caveats

Author Sheets Only

For simplicity, the current proposal is only enabled for author style sheets, where the decoded bytes are ensured to be persisted with the underlying css resource. Some extra effort would be required to enable this for e.g. UA style sheets (there may be slight memory regressions).

Speculative Gating on Sheet Size

The design also doesn't have any sort of gating on "large sheets", so we need to make sure that we don't regress performance with small sheets with good utilization. Ideally we can enable the feature for all sheets.

.lazyMatch::before { content: attr(lazytext) } problem

As rune@ mentions in the loading-dev thread, if a sheet has a property like this:

> "it means we need to create an invalidation set for lazytext (with invalidatesSelf()) to the RuleFeatureSet for the stylesheet, and also propagate it to the document-wide RuleFeatureSet. You'd want to do that cheaply, not a full collectFeatures(). Also, as StyleSheetContents may be cached and shared at both a resource level (even between documents) and text source level (within the same document), propagation may need to happen in mutliple documents/scopes."

This is problematic. Our v1 avoids this problem by not lazy parsing if ::before and ::after appear in the selector.

Local Results

Worst Case Performance

To test overhead from the feature, I instrumented the parsing code to parse every property after lazy parsing it. This overhead measured to be ~2-3% of total parse time, and 5-8% of property parsing time (on Linux). This means we should see wins on sheets that use less than 92-95% of their style rules. For this reason I am confident that we can enable this optimization for all style sheets.

Raw Parsing Wins

Initial benchmarking on Pixel C shows 45% improvement on parse time before first contentful paint on facebook.

Metrics to watch and add

- PageLoad.PaintTiming.ParseStartToFirstContentfulPaint
- Style.AuthorStyleSheet.ParseTime
- Style.UpdateTime
- In Progress:
    - PageLoad.CSSTiming.ParseTime.BeforeFirstContentfulPaint
    - PageLoad.CSSTiming.UpdateStyleTime.BeforeFirstContentfulPaint
    - Style.LazyCSS.PercentRulesParsed. This is a bit tricky because we want to know *when* to log this. Doing it with a pre-finalizer will miss many sheets. An alternative could be an enumeration histogram.
        - >= 0% rules parsed
        - >= 25% rules parsed
        - >= 50% rules parsed
        - …

Initial data from the Wild (Canary + Dev)

ParseStartToFirstContentfulPaint (android)

This is showing nonsignificant wins of ~10ms at the 50th %ile and ~24ms at the 95%ile. Hopefully more data will show true wins.

Parse and Update Time Before FCP (android)

The key metric to watch is the *aggregate* update + parse time before first contentful paint, which takes into account us regressing update time. So far this is showing wins on Android, with a 12-15% reduction at most percentiles. This amounts to a ~10ms win at the 50th %ile, and a ~37ms win at the 95%ile.

Rule usage % (all platforms)

After massaging the data into the right format, the current data looks like this:

rule usage %	CDF
[0, 10]	36.33%
(10, 25]	76.60%
(25, 50]	89.47%
(50, 75]	94.70%
(75, 90]	95.88%
(90, 100]	96.10%
100	100.00%

The main takeaway is the ~77% of css files use <= 25% of their rules, and ~90% use <= 50%.

Rule usage % (android)

Android data looks a bit different

rule usage %	CDF
[0, 10]	0.26
(10, 25]	0.54
(25, 50]	0.64
(50, 75]	0.67
(75, 90]	0.67
(90, 100]	0.68
100	1.00

The main takeaway is that we still have a big chunk of css files that don’t parse a lot of rules, but the data is bimodal, and we also have a lot (32%) of files which parse *all* of their rules. We should figure out if there’s a way of tagging these.

Significant Data from 14 days on Beta

ParseStartToFirstContentfulPaint

- Android: It looks like we have significant 2% improvement at the 50th %ile (~10ms), and a ~1.5% improvement at the 75th %ile (~13ms). Other %iles are still in noise.
- Desktop: Nothing of significance

Style.AuthorStyleSheet.ParseTime

- Android: Significant improvement of ~30% at all percentiles. This is ~170us at the 50th %ile and 20ms at the 95th %ile.
- Desktop: Significant improvement of ~37% at all percentiles except 95 (27.7% improvement). This is ~282us at the 50th %ile and 5ms at the 95th %ile.

Style.UpdateTime

Nothing of significance in desktop or android.

PageLoad.CSSTiming.ParseAndUpdate.BeforeFirstContentfulPaint

- Android: 15% improvement at the 50th %ile (~8ms), 16.4% improvement at the 75th %ile (~18ms), and 13% improvement at the 95th %ile (37ms).
- Desktop: 10-15% improvement at all percentiles. This is a ~3ms win at the 50th %ile and a 19ms win at the 95th %ile.