ES6 Module Loading Performance (Worklog)

Publicly visible, but please note that this is my raw lab note and not intended for wide distribution.

ksakamoto@chromium.org

2022

Oct 13: WebBundles v2 PoC benchmark

Context: http://bit.ly/webbundle-v2

Compare three different ways to pass payload bytes from SameOriginWebBundleURLLoader to WebURLLoaderClient.

Benchmark

https://github.com/google/webbundle/tree/main/webbundle-bench
Generated with `--depth 4 --branches 4`

Chromium Patches

- Baseline: Existing "v1" (NetworkService-based) implementation
- DataPipe: Creates a data pipe and passes its consumer end to WebURLLoaderClient::DidStartLoadingResponseBody.
- DidReceiveData: Calls WebURLLoaderClient::DidReceiveData from SameOriginWebBundleURLLoader::OnData. Doesn't work for RawResource (DCHECKs in FetchManager::Loader::DidFinishLoading).
- BytesConsumer: Creates a SharedBufferBytesConsumer and passes it to WebURLLoaderClient.

Results

On Z840 workstation, average of 3 runs.

Patch	ImportDuration (ms)
Baseline	339
DataPipe	337

DidReceiveData	159
BytesConsumer	168

Using a data pipe per subresource has a large performance cost. The BytesConsumer layer adds some overhead, but the result looks pretty good.

2021

Mar 30: Setting up a new benchmark environment

Benchmark

https://github.com/irori/js-module-benchmark/tree/webbundle
Usage:

```
# Install dependencies
$ npm install

# Generate test cases
$ ./build.sh

# Run benchmarks. It runs the browser in headless mode and
# output results to stdout.
$ node run_benchmark.js --browser ~/chromium/src/out/Release/chrome
```

Chromium binaries (for Linux)

- Chromium-r867396-linux.zip: Chromium ToT as of 2021-03-30. NetworkService based implementation.
- Chromium-r830437-scopes-linux.zip: Chromium ToT as of 2020-11-24 (just before we switch to NetworkService based implementation), plus scopes= attribute support
- Chromium-subresource-wbn-poc1-linux.zip: Built from this patch. Bundled resources are intercepted in ResourceFetcher.
- Chromium-subresource-wbn-poc2-linux.zip: Built from this patch.This intercepts module script requests in the Modulator class.

Diagrams showing at which layer each implementation intercepts bundled requests.

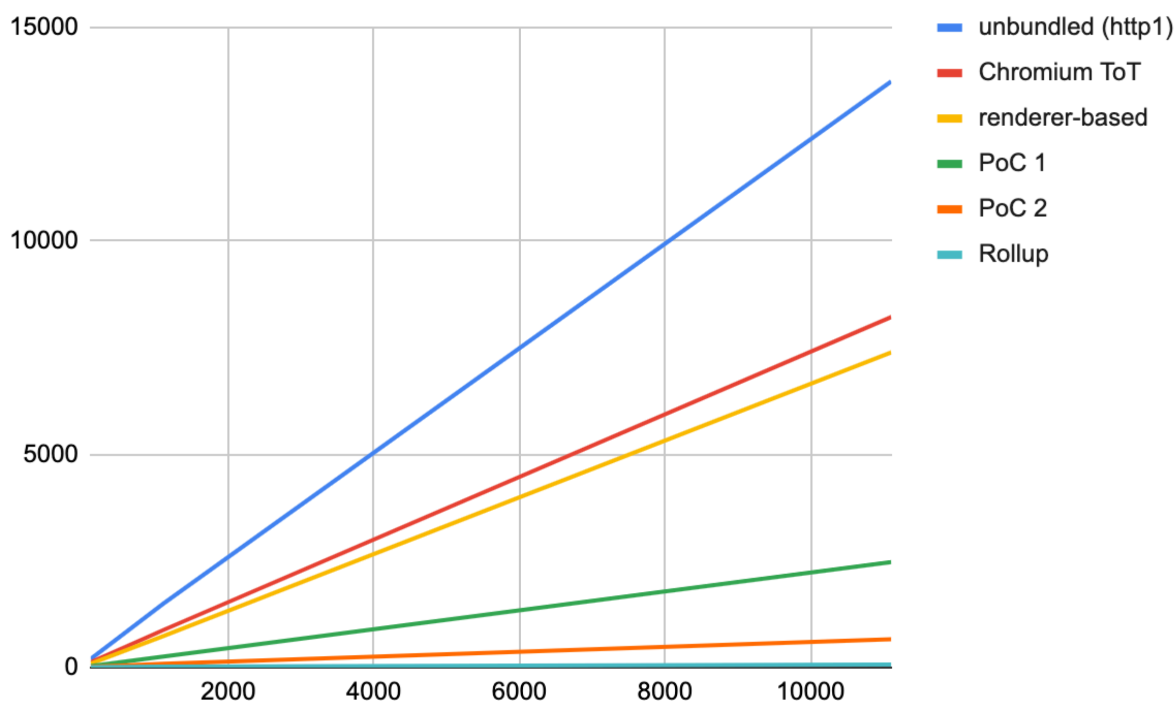
Preliminary results

On Z840 workstation.

spreadsheet

#modules	unbundled (http1)	Chromium ToT	renderer-based	PoC 1	PoC 2	Rollup
110	199	123	91	36	16	9
1110	1504	881	736	258	86	22
11110	13743	8215	7387	2470	663	72

(unit: milliseconds)



2020

Nov 6: Subresource WBN performance - Renderer based vs. NetworkService based

Compared the performance of subresource loading, between the currently landed implementation (renderer-based) and the new design (NetworkService-based).

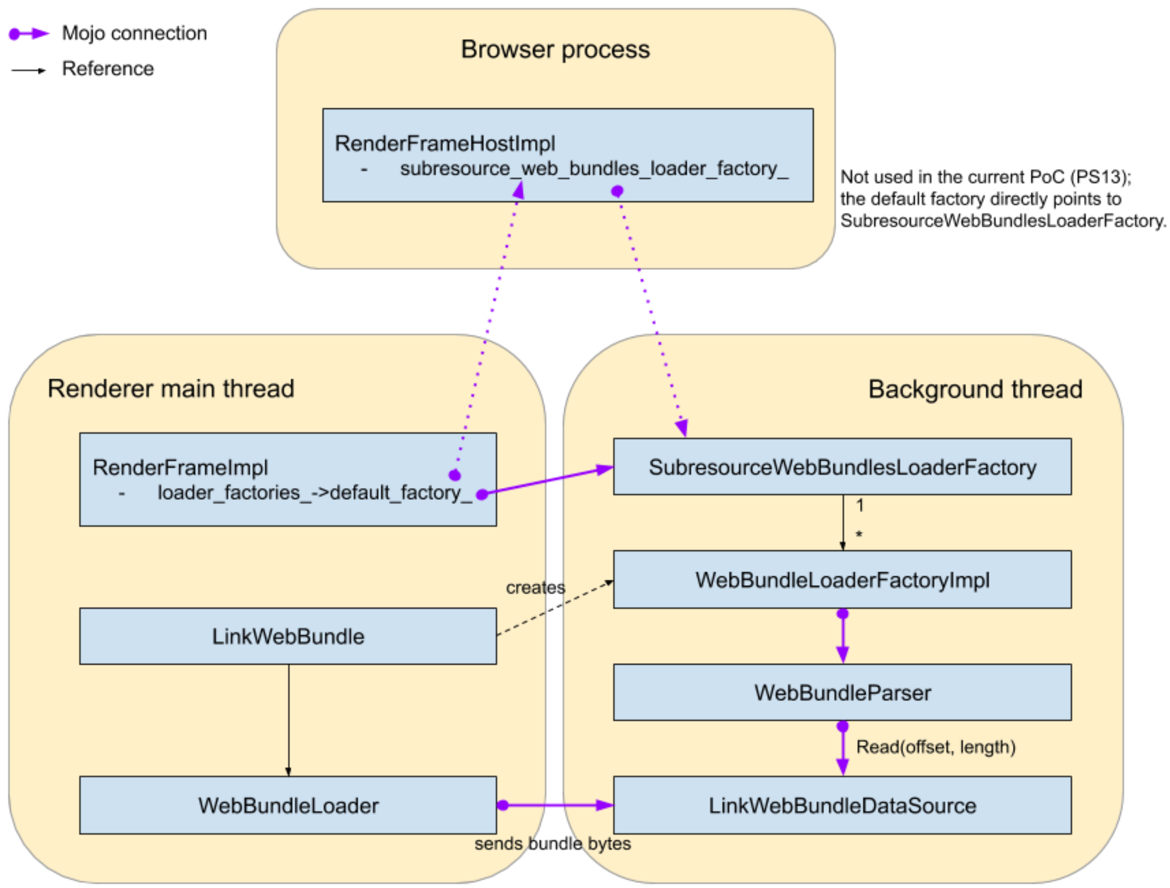
	NS based (crrev.com/c/2497394/8)	Renderer based (Chromium ToT #822482)
Moment.js (104 modules)	60ms	70ms

Three.js (333 modules)	227ms	271ms
DevTools frontend	FCP: 616ms LCP: 894ms (timeline)	FCP: 685ms LCP: 994ms (timeline)

(Test environment: MacBook Pro 2018)

Jun 11: Subresource WBN loading architecture

The diagram below illustrates current design of Subresource WBN loading PoC:



Most of the WebBundle-specific processes are offloaded to a background sequence, except for WebBundleLoader which implements ThreadableLoaderClient and pushes loaded WBN bytes to LinkWebBundleDataSource.

We could optimize a bit more in the background thread, by removing the Mojo interface layer between WebBundleLoaderFactoryImple and WebBundleParser, and between WebBundleParser and LinkWebBundleDataSource. However, a trace of three.js shows that the renderer main thread is busy (see below). So optimization in the background thread may not improve benchmark performance.

Jun 2: Measuring the new approach of Subresource WBN

- Mojo parser: Use data_decoder::WebBundleParser for WBN parsing. It runs in-process, but adds mojo message serialization overhead. CL
- Mojo URLLoaderFactory with IPC: Use SubresoureWebBundlesLoaderFactory. All subresource loads go through the browser process (straw idea 1 of design sketch). CL
- Mojo Parser + Mojo URLLoaderFactory with IPC: Both of the above. CL
- Mojo URLLoaderFactory w/o IPC: Use SubresoureWebBundlesLoaderFactory, but without IPC router. CL
- Mojo URLLoaderFactory (bg thread) w/o IPC: Run SubresoureWebBundlesLoaderFactory in a background thread. CL (old) Improved CL

	Three.js	DevTools FCP	DevTools LCP
Horo@'s PoC (baseline)	113 ms	596 ms	947 ms
Mojo Parser	146 ms	640 ms	978 ms
Mojo URLLoaderFactory with IPC	399 ms	1092 ms	1778 ms
Mojo Parser + Mojo URLLoaderFactory with IPC	432 ms	1100 ms	1741 ms
Mojo URLLoaderFactory w/o IPC	400 ms	1081 ms	1747 ms

Mojo URLLoaderFactory (bg thread) w/o IPC	~~306 ms~~ 259 ms	~~918 ms~~ 851 ms	~~1375 ms~~ 1317 ms

Observations:

- Using WBN parser via the mojo interface adds some overhead (~0.1ms / resource).
- Using mojom::URLLoaderFactory adds more overhead. The DevTools case was even slower than the WBN navigation case.
- With or without IPC did not have a significant performance impact when SubresoureWebBundlesLoaderFactory is used. Trace suggests that going through up to URLLoader layer adds several overheads:
  - More postTask hops on the renderer main thread
    - 4 main-thread tasks per resource, where ResourceFetcher-level intercept was 2 tasks per resource
  - Although the default factory is directly set to the Subresource WBN factory, render-browser IPCs still happen (safe-browsing?)
  - Overhead of using mojom::URLLoaderFactory interface (e.g. data pipe creation)

Apr 17: Subresource WBN performance report

Published here.

Apr 6: Module script optimization for Subresource WBN, continued

The experimental patch used in the Apr 2 measurement had a bug where WBN resource is fetched/parsed twice.

Here's a result of re-running the devtools-frontend measurement with the fixed patch, and using a well-ordered WebBundle (resources requested earlier comes first).

spreadsheet

	FCP (ms)	LCP (ms)
Subresource WBN	636	921
Subresource WBN + WebBundleModuleScriptFetcher	504	787
WebPack (w/ code splitting) (*1)	599	966

(*1) CSS and JSON resources are fetched from the network.

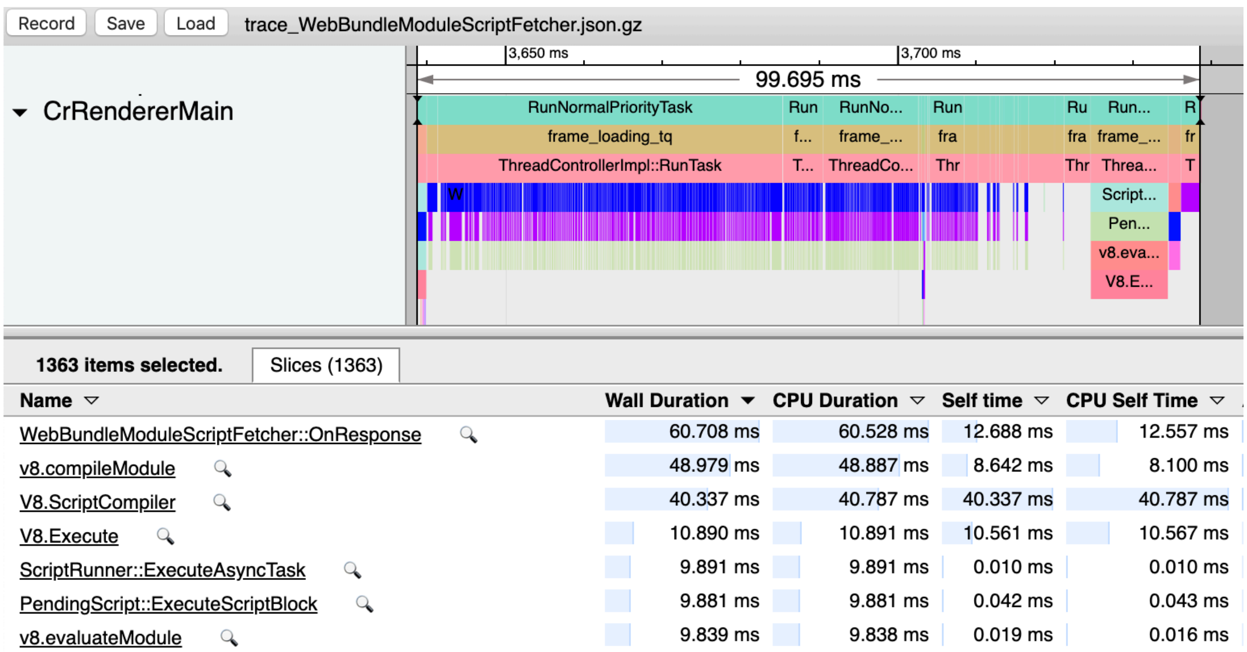
Apr 2: Subresource WBN: Use ModuleScriptFetcher to intercept module script requests

[Time breakdown for Subresource WBN](#) suggests that `ResourceFetcher` has considerable overhead. Can we cut this by intercepting requests at a higher layer?

[This patch](#) adds `WebBundleModuleScriptInterceptor` which is registered in `Modulator` and intercepts module requests whose URLs are in-scope of the WebBundle. Intercepted requests are handled by `WebBundleModuleScriptFetcher` which returns response body from the WBN.

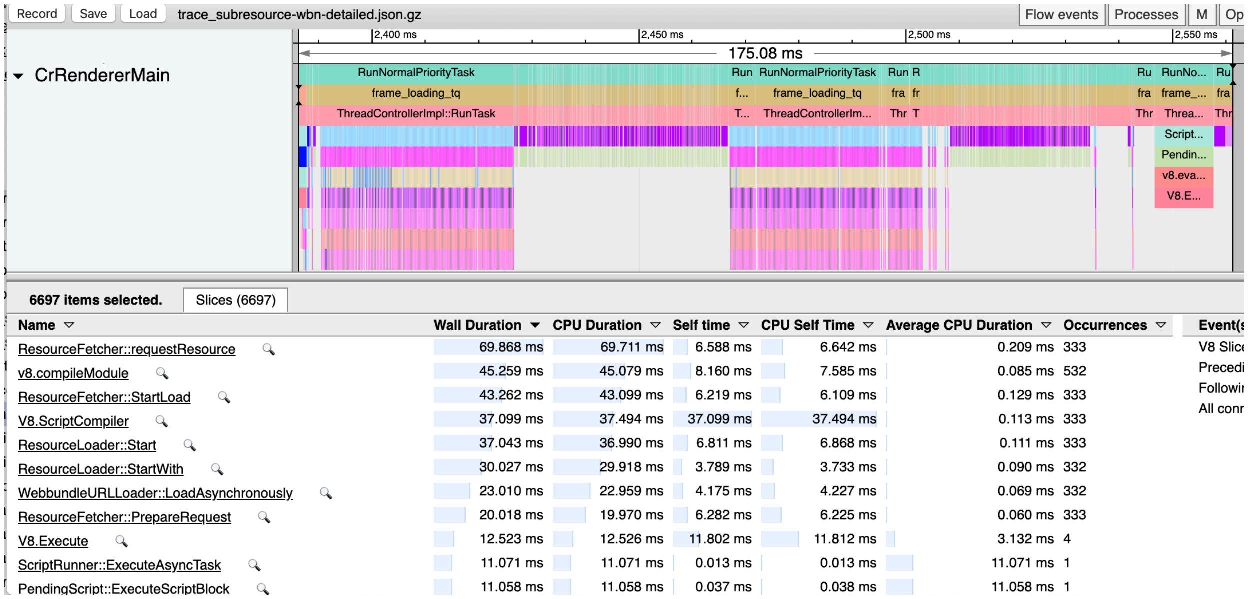
Results: Three.js

Here's a [trace](#) of [Three.js test](#), with this patch:



Name	Wall Duration	CPU Duration	Self time	CPU Self Time
WebBundleModuleScriptFetcher::OnResponse	60.708 ms	60.528 ms	12.688 ms	12.557 ms
v8.compileModule	48.979 ms	48.887 ms	8.642 ms	8.100 ms
V8.ScriptCompiler	40.337 ms	40.787 ms	40.337 ms	40.787 ms
V8.Execute	10.890 ms	10.891 ms	10.561 ms	10.567 ms
ScriptRunner::ExecuteAsyncTask	9.891 ms	9.891 ms	0.010 ms	0.010 ms
PendingScript::ExecuteScriptBlock	9.881 ms	9.881 ms	0.042 ms	0.043 ms
v8.evaluateModule	9.839 ms	9.838 ms	0.019 ms	0.016 ms

For comparison, here's a [trace](#) of the same test, with the original subresource-WBN PoC patch:



- The time it takes for `import("three.js")` is reduced from **175ms** to **100ms**.
  - This not only eliminated almost all of the resource fetching cost, but also reduced scheduling overheads, as many module scripts are compiled within a single `RunTask`.
- Since ResourceFetcher is completely bypassed, module scripts do not show up in the devtools network tab, while they are still visible in the Sources tab.

Results: DevTools frontend

Obsolete, see Apr 6 measurement.

spreadsheet

	FCP (ms)	LCP (ms)
Subresource WBN	729	996
Subresource WBN + WebBundleModuleScriptFetcher (*1)	624	858
WebPack (w/ code splitting) (*2)	599	966

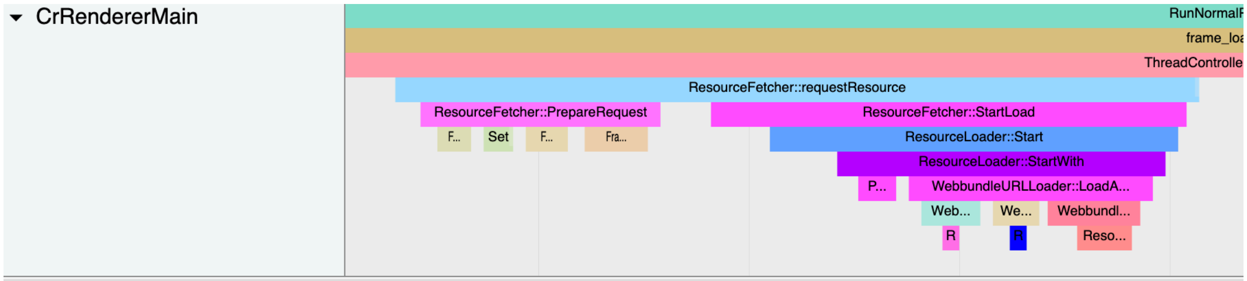
(*1) WBN file is fetched twice, by WebBundleModuleScriptInterceptor and WebBundleLoaderFactoryImpl. I plan to merge them to dedupe the work.
(*2) CSS and JSON resources are fetched from the network.

Mar 30: Subresource WBN: tracing inside ResourceFetcher::requestResource()

Added trace events to several functions called inside ResourceFetcher::requestResource().

- Patch
- Trace

The image below is a trace of a single module script request. WebbundleURLLoader::LoadAsynchronously() uses only 30% (58us / 191us) of the time inside ResourceFetcher::requestResource().



17 items selected. Slices (17)

Name	Wall Duration	CPU Duration	Self time	CPU Self Time	Average CPU Duration
ResourceFetcher::requestResource	0.191 ms	0.190 ms	0.021 ms	0.020 ms	0.190 ms
ResourceFetcher::PrepareRequest	0.057 ms	0.057 ms	0.017 ms	0.018 ms	0.057 ms
ResourceFetcher::StartLoad	0.113 ms	0.113 ms	0.016 ms	0.016 ms	0.113 ms
FrameFetchContext::PopulateResourceRequest	0.008 ms	0.008 ms	0.008 ms	0.008 ms	0.008 ms
SetReferrer	0.007 ms	0.006 ms	0.007 ms	0.006 ms	0.006 ms
FrameFetchContext::CanRequest	0.010 ms	0.010 ms	0.010 ms	0.010 ms	0.010 ms
FrameFetchContext::PrepareRequest	0.015 ms	0.015 ms	0.015 ms	0.015 ms	0.015 ms
ResourceLoader::Start	0.097 ms	0.097 ms	0.019 ms	0.019 ms	0.097 ms
ResourceLoader::StartWith	0.078 ms	0.078 ms	0.011 ms	0.011 ms	0.078 ms
PopulateResourceRequest	0.009 ms	0.009 ms	0.009 ms	0.009 ms	0.009 ms
WebbundleURLLoader::LoadAsynchronously	0.058 ms	0.058 ms	0.011 ms	0.011 ms	0.058 ms
WebbundleURLLoader::OnResponse	0.014 ms	0.014 ms	0.010 ms	0.010 ms	0.014 ms
WebbundleURLLoader::OnData	0.011 ms	0.011 ms	0.007 ms	0.007 ms	0.011 ms
WebbundleURLLoader::OnFinish	0.022 ms	0.022 ms	0.009 ms	0.009 ms	0.022 ms
ResourceLoadObserverForFrame::DidReceiveResponse	0.004 ms	0.004 ms	0.004 ms	0.004 ms	0.004 ms
Resource::appendData	0.004 ms	0.004 ms	0.004 ms	0.004 ms	0.004 ms
ResourceFetcher::HandleLoaderFinish	0.013 ms	0.013 ms	0.013 ms	0.013 ms	0.013 ms
Totals	**0.711 ms**	**0.709 ms**	**0.191 ms**	**0.190 ms**	**0.042 ms**

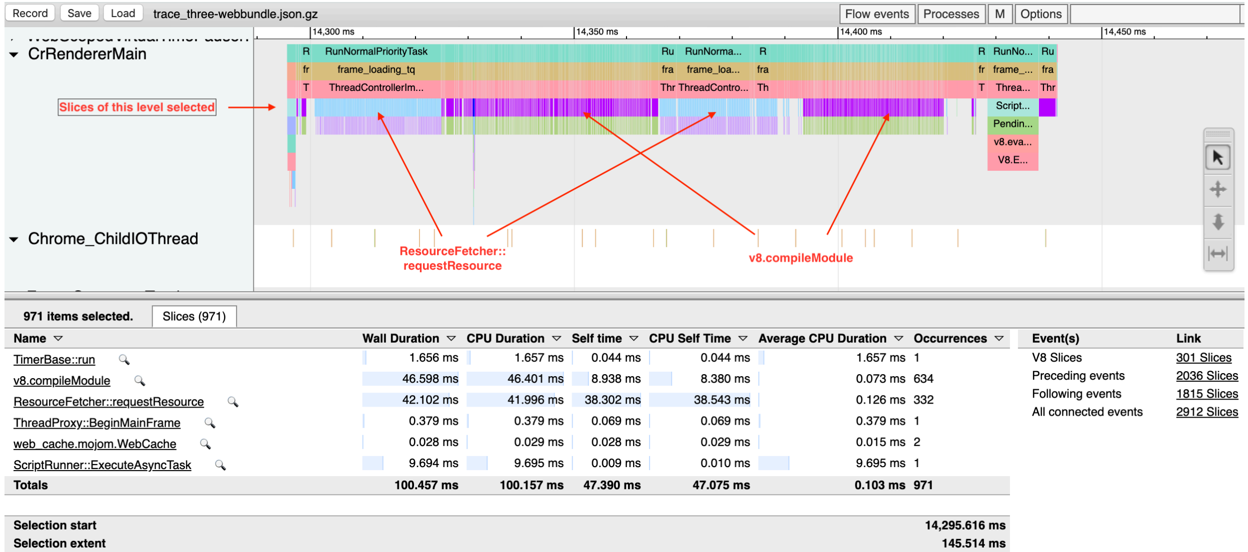
Mar 27: Time breakdown for Subresource WBN

Test case:

- Three.js (333 modules)
- “Empty” app (import three.js and do nothing)
- Subresource WBN
    - Start importing modules after the loading of WBN is completed

Result

[Trace file]



Time breakdown (from import start to execution end)

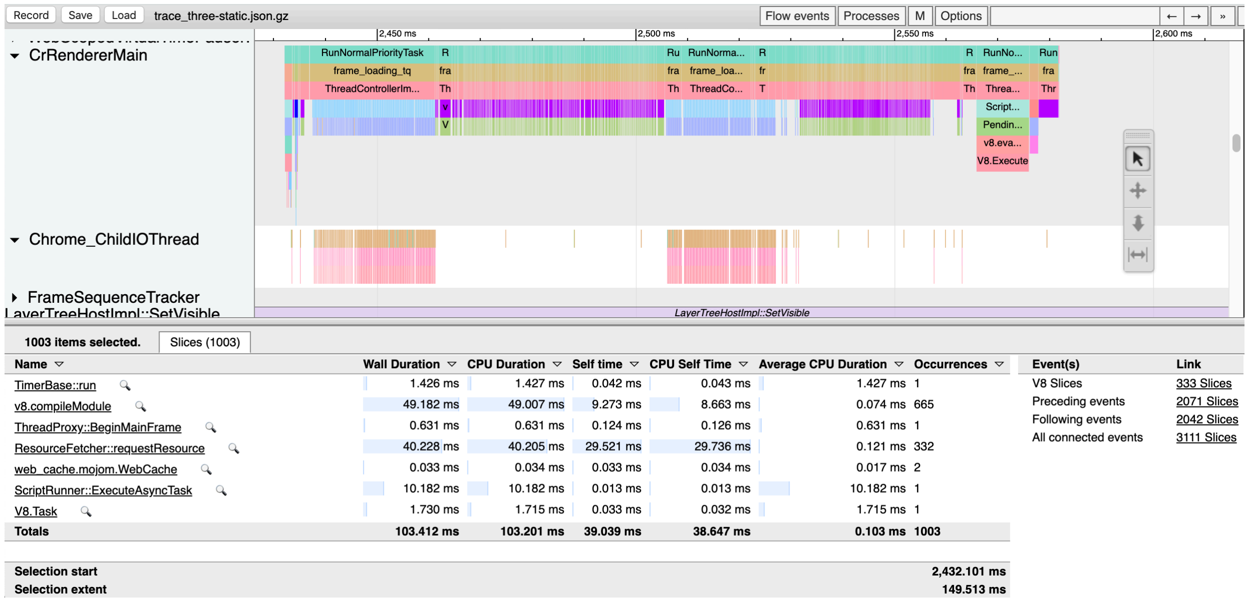
Resource request (ResourceFetcher::requestResource)	42ms
Compile (v8.compileModule)	47ms
Execution (ScriptRunner::ExecuteAsyncTask)	10ms
Other tasks + Overhead	46ms
Total	145ms

Experiment: Using ResourceFetcher::ResourceForStaticData() code path

In the above trace, almost one-third of the time is spent in ResourceFetcher::requestResource(). ResourceFetcher has a dedicated code path for “static” resources (e.g. data: URLs or resources from MHTML archive). Would using this code path in subresource WBN reduce the time spent in Resource Fetcher?

[Changes to the Subresource WBN PoC patch]()

Result ([trace]())



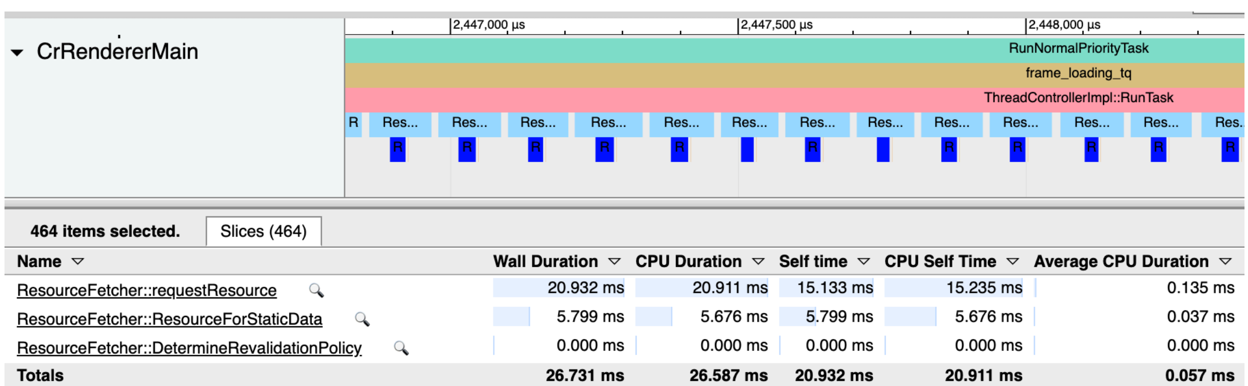
Name	Wall Duration	CPU Duration	Self time	CPU Self Time	Average CPU Duration	Occurrences
TimerBase::run	1.426 ms	1.427 ms	0.042 ms	0.043 ms	1.427 ms	1
v8.compileModule	49.182 ms	49.007 ms	9.273 ms	8.663 ms	0.074 ms	665
ThreadProxy::BeginMainFrame	0.631 ms	0.631 ms	0.124 ms	0.126 ms	0.631 ms	1
ResourceFetcher::requestResource	40.228 ms	40.205 ms	29.521 ms	29.736 ms	0.121 ms	332
web_cache.mojom.WebCache	0.033 ms	0.034 ms	0.033 ms	0.034 ms	0.017 ms	2
ScriptRunner::ExecuteAsyncTask	10.182 ms	10.182 ms	0.013 ms	0.013 ms	10.182 ms	1
V8.Task	1.730 ms	1.715 ms	0.033 ms	0.032 ms	1.715 ms	1
Totals	103.412 ms	103.201 ms	39.039 ms	38.647 ms	0.103 ms	1003

Event(s)	Link
V8 Slices	333 Slices
Preceding events	2071 Slices
Following events	2042 Slices
All connected events	3111 Slices

Selection start	2,432.101 ms
Selection extent	149.513 ms

Unfortunately this didn't improve performance, and the trace looks almost the same as the previous one, except the IO thread looks busy because of the FrameHostMsg_DidLoadResourceFromMemoryCache IPC. (I tried commenting out this IPC but it did not improve the main thread performance.)

Actually, ResourceForStaticData() only takes a quarter of requestResource(). The bottlenecks exist before and after that.



Name	Wall Duration	CPU Duration	Self time	CPU Self Time	Average CPU Duration
ResourceFetcher::requestResource	20.932 ms	20.911 ms	15.133 ms	15.235 ms	0.135 ms
ResourceFetcher::ResourceForStaticData	5.799 ms	5.676 ms	5.799 ms	5.676 ms	0.037 ms
ResourceFetcher::DetermineRevalidationPolicy	0.000 ms	0.000 ms	0.000 ms	0.000 ms	0.000 ms
Totals	26.731 ms	26.587 ms	20.932 ms	20.911 ms	0.057 ms

Mar 12: Subresource WBN on a slow network

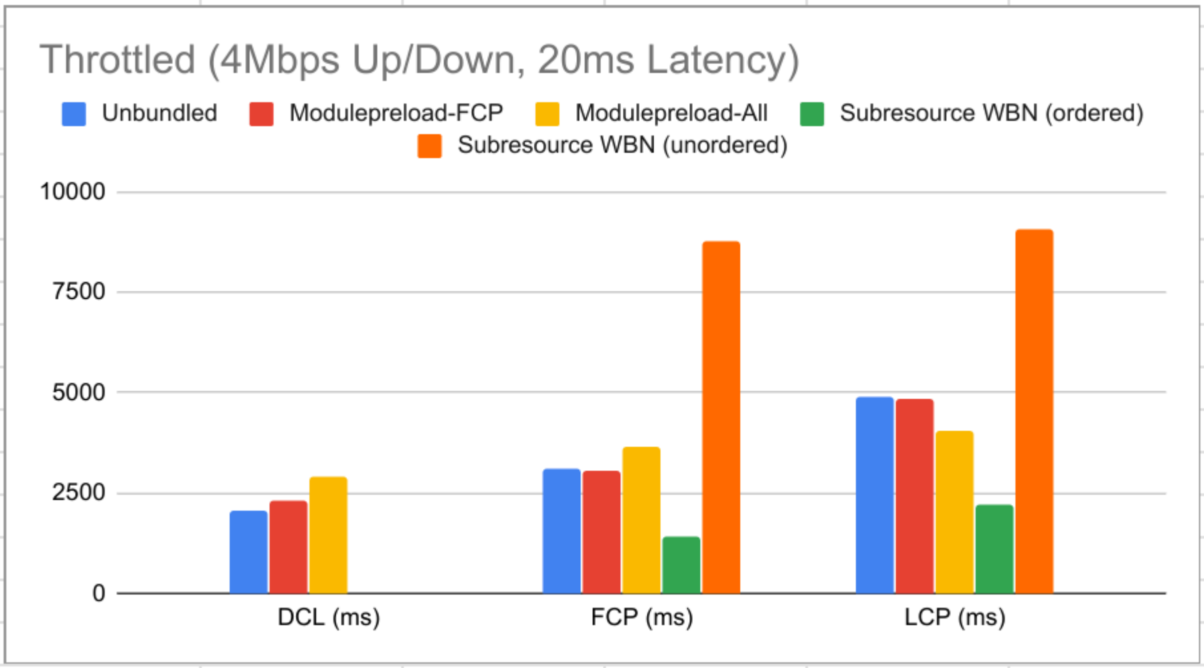
Two more cases with network throttling (4Mbps Up/Down, 20ms Latency):

- Subresource WBN created without resource ordering consideration
- Subresource WBN with optimized resource ordering (resources requested earlier comes first)

Result

spreadsheet

	Throttled (4Mbps Up/Down, 20ms Latency)				
	Unbundled	Modulepreload-FCP	Modulepreload-All	Subresource WBN (ordered)	Subresource WBN (unordered)
DCL (ms)	2069	2297	2904	N/A	N/A
FCP (ms)	3109	3046	3673	1402	8787
LCP (ms)	4892	4836	4073	2218	9073



Unordered subresource WBN is very slow because it has to wait until almost entire WBN bytes are received.

Mar 9: DevTools frontend and modulepreload

Two additional test cases to the Mar 5 benchmark:

- modulepreload_fcp: Preload 270 modules that appear to be needed for FCP
- modulepreload_all: Preload all (384) modules

Both served by a local HTTP2 server.

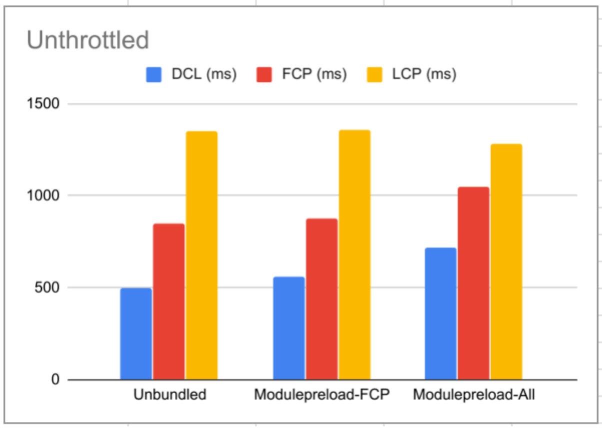
With two network conditions:

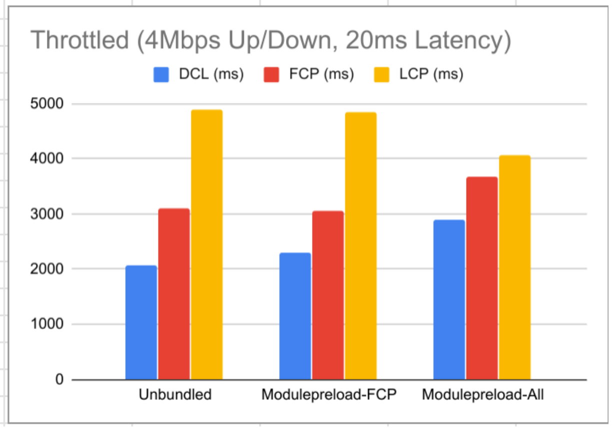
- Unthrottled
- Throttled (4Mbps Up/Down, 20ms Latency)

Result

Spreadsheet

	Unthrottled					Throttled (4Mbps Up/Down, 20ms Latency)			
	Unbundled	Modulepreload-FCP	Modulepreload-All			Unbundled	Modulepreload-FCP	Modulepreload-All	<- links to th
DCL (ms)	494	560	719		DCL (ms)	2068.9	2296.8	2904.4	
FCP (ms)	849	872	1049		FCP (ms)	3109.3	3046.4	3672.9	
LCP (ms)	1349	1357	1282		LCP (ms)	4891.8	4835.9	4072.9	





- Preloading all modules is negative for FCP, positive for LCP.
- Partially preloading scripts (only for FCP) didn't have significant impact for this benchmark.

Mar 5: Measurements against DevTools frontend

Environment

- Chromium @744551 + Subresource WBN PoC patch
- Z840 workstation / Linux
- Cache disabled via DevTools
- Localhost server

Targets

DevTools frontend, with various bundling technologies. Measured DCL/FCP/LCP of initial page load (the Network tab is selected) using (Chrome's embedded DevTools') performance profiler.

Test cases:

- Unbundled
  - From http1 server (using a node.js server)
  - From http2 server (using simplehttp2server)

- Network WBN
  - `--enable-features=WebBundlesFromNetwork`
  - Reference data, current Network WBN implementation is not optimal / it doesn't support progressive / streamed loading.
- Subresource WBN
  - All subresources are loaded from a WBN, via <link rel=webbundle>.
- WebPack
  - With code splitting (default)
  - Without code splitting (generates one big JS, including scripts that are not used for initial load)
  - Note: JSON and CSS are still loaded as separate resources, but in real-world WebPack projects these are likely to be bundled as well (using WebPack loaders).
- WebPack + Subresource WBN
  - Bundled WebPack (w/ code splitting) output as a WBN.
  - Note: WBN for this case is not optimal (unbundled scripts are included too)

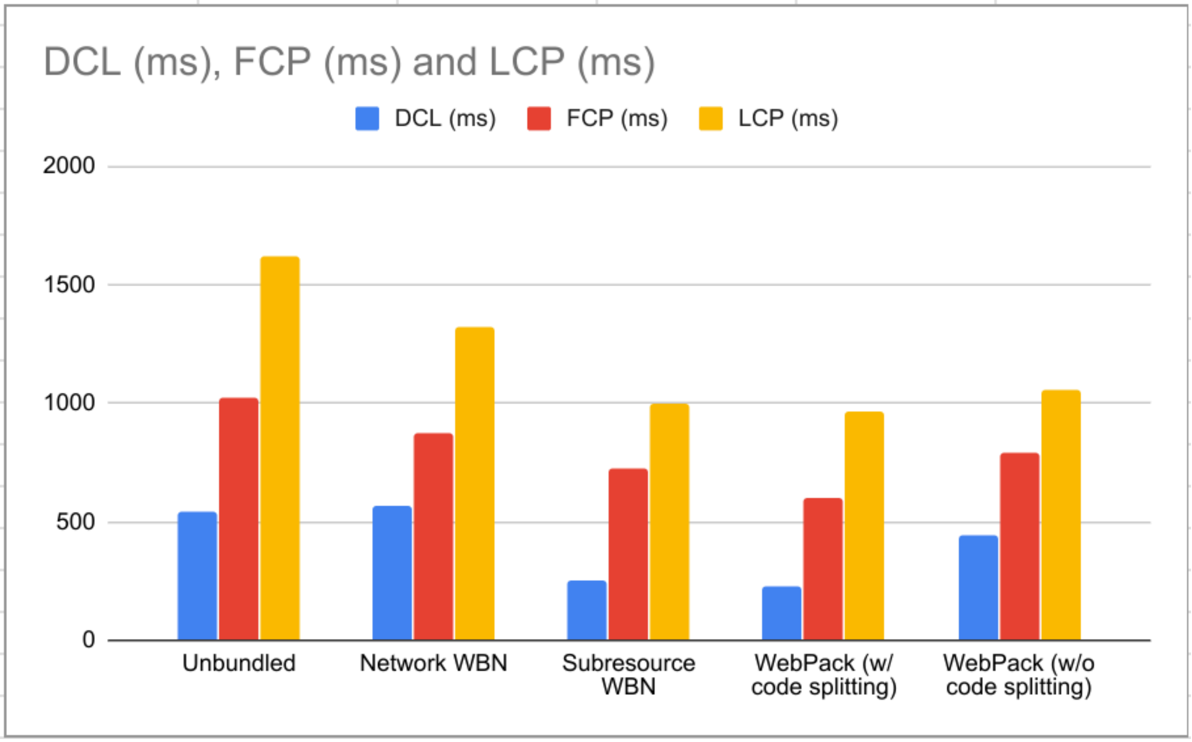
Target Data

	Unbundled	WBN	WebPack w/ code splitting	WebPack w/o code splitting
JS count	385	385	42	2
non-JS resource count	173	173	173	173
JS size	4.4 MB	4.4 MB	2.2 MB	6.3 MB
non-JS resource size	0.4 MB	0.4 MB	0.4 MB	0.4 MB

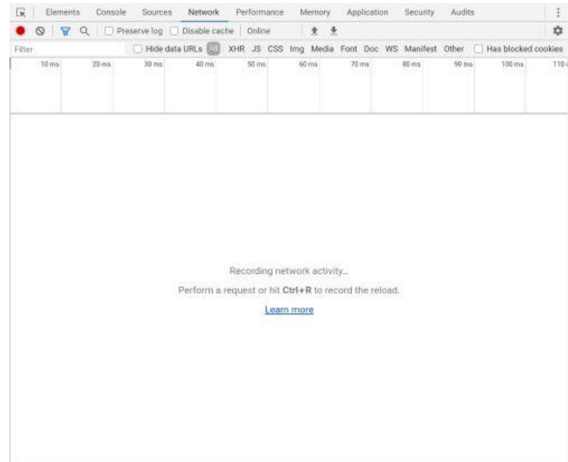
Result

See this spreadsheet for full results, and links to the timeline viewer.

	Unbundled	Network WBN	Subresource WBN	WebPack (w/ code splitting)	WebPack (w/o code splitting)
DCL (ms)	546	564	249	225	444
FCP (ms)	1023	872	729	599	792
LCP (ms)	1623	1322	996	966	1059



DCL fires different timings by the test cases. In the subresource WBN case, it fires without waiting for any JS resources.
FCP is when the tab bar is painted.
LCP is when the full content of the network tab is rendered.

	
FCP	LCP

Feb 13: Initial measurements for Subresource Web Bundle

Report:
https://docs.google.com/document/d/12jCr8trGxGyBw_YIqWM-DpXetz4NzfY5gfZrNxtCqDk/edit

Summary

- Subresource WebBundle (unpackaged in renderer) is faster than Navigation-to-WebBundle (unpackaged in browser), but still slower than Webpack-generated JS bundle.
- Trace suggests that ResourceFetcher is consuming considerable time in Subresource WebBundle loading.

Measurement

Environment:

- Benchmark: three.js (333 modules) in samples-module-loading-comparison
  - Just import modules, no wireframe drawing (because it adds noise)
- Chromium @739799 + Horo's PoC patch https://chromium-review.googlesource.com/c/chromium/src/+/2032692/10
- Z840 workstation / Linux
- Cache disabled via DevTools
- Very fast network (local http1.1 server)

Configurations:

- Unbundled

    - Loads every module script from the network.
- Navigation to WebBundle from file
    - Standalone WebBundle containing html and all modules, loaded from a file (`chrome://flags/#web-bundles`).
- Navigation to WebBundle from network
    - Standalone WebBundle containing html and all modules, loaded from the network (`--enable-features=WebBundlesFromNetwork`).
- Subresource WebBundle
    - WebBundle containing only the module scripts, loaded via <link rel=webbundle>.
- Webpack bundle
    - Single classic script generated by Webpack (no tree-shaking).

	Unbundled	Navigation to WBN from file	Navigation to WBN from network	Subresource WBN	Webpack bundle
Time to onload	671 ms	389 ms	370 ms	217 ms	128 ms
First module's fetchStart	31 ms	86 ms	56 ms	40 ms	48 ms
Last module's responseEnd	635 ms	N/A	N/A	203 ms	58 ms

Caveats:
- I haven't tried very hard to de-noize the results.
- No responseEnd timing data for WebBundle navigation cases, as ResourceTiming is broken

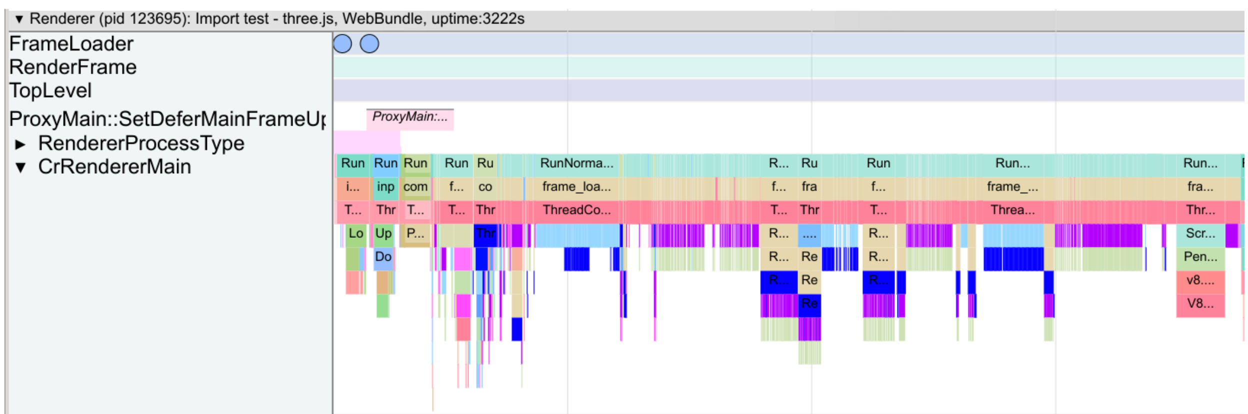
Observations:
- Even with a very fast network, unbundled is slower than navigation to WBN.
- Subresource WBN is faster than navigation to standalone WBN, because of less IPC.
- Webpack bundle is even faster.

Performance Analysis

Here's a DevTools performance profile of a "Subresource WBN" run:

See the flame graph at bottom. Yellow slices are script events (Compile / Evaluate Module). Blue slices are loading events (Send Request, Receive Response, Receive Data, Finish Loading).

Here's a trace of (another) "Subresource WBN" run:

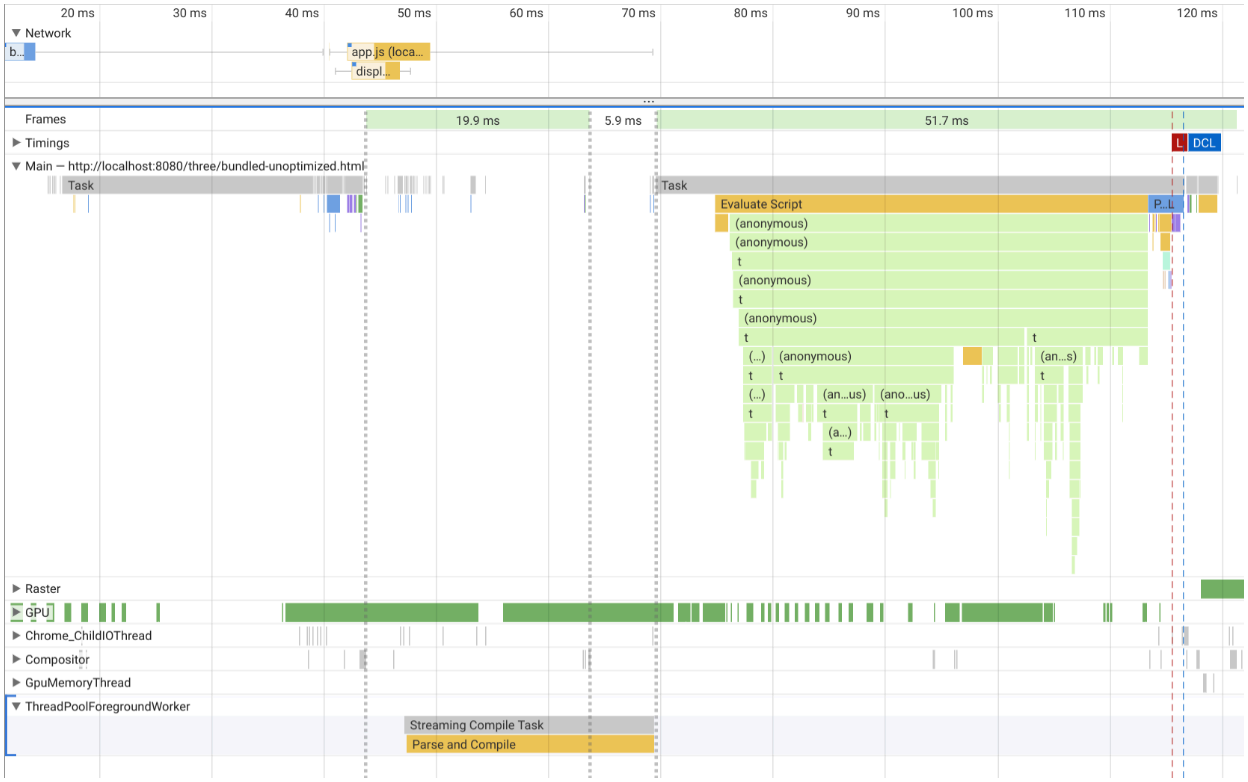


5086 items selected. Slices (5086)

Name	Wall Duration	CPU Duration	Self time	CPU Self Time	Average CPU
V8.ScriptCompiler	38.296 ms	38.695 ms	38.296 ms	38.695 ms	
ResourceFetcher::requestResource	32.913 ms	32.901 ms	29.894 ms	30.185 ms	
ThreadControllerImpl::RunTask	138.375 ms	137.727 ms	23.076 ms	23.222 ms	
Resource::appendData	32.000 ms	31.466 ms	14.207 ms	13.828 ms	
V8.Execute	12.885 ms	12.891 ms	12.345 ms	12.360 ms	
v8.compileModule	46.650 ms	46.534 ms	8.566 ms	8.052 ms	
RunNormalPriorityTask	143.055 ms	142.706 ms	5.813 ms	5.702 ms	
frame_loading_tq	134.157 ms	133.930 ms	4.168 ms	4.185 ms	
default_tq	3.036 ms	3.025 ms	1.481 ms	1.478 ms	
HTMLDocumentParser::processTokenizedChunkFromBackgroundParser	6.533 ms	6.358 ms	1.421 ms	1.252 ms	

ResourceFetcher::requestResource takes 30 ms while V8.ScriptCompiler takes 38 ms.

For comparison, here's a DevTools profile of a "Webpack bundle" run:



The network request finishes very quickly (~10ms), but parse/compile (in a worker thread) and evaluate takes a long time. On the other hand, in the “Subresource WBN” case, once all module scripts are fetched and compiled, evaluation takes only ~15ms.

Next Steps

- Identify potential performance optimizations. What's consuming time in ResourceFetcher? Could we bypass it?
- Measurement for a slow network case. Does resource ordering in bundle make difference?
- Measure against other targets (e.g. DevTools frontend) too.

2018

Jul 25: webbundle PoC loading performance

- Benchmark: three.js (333 modules) [threejs.wbn]
- Chromium ToT (r577101)
- [Webbundle PoC Patch]
  - Loads whole .wbn in-memory, and then serve from there
- Linux on Z840 workstation

	unbundled, first load	unbundled, second load (disk cache)	webbundle	webpack bundle
Time to onload	977 ms	505 ms	513 ms	381 ms
First module's fetchStart	36 ms	24 ms	89 ms	31 ms
Last module's responseEnd	690 ms	352 ms	361 ms	98 ms

- Loading performance of current patch is close to the load from disk cache
- First fetchStart is delayed in webbundle due to the bundle loading overhead

Apr 23: Performance across browsers, as of Apr 2018

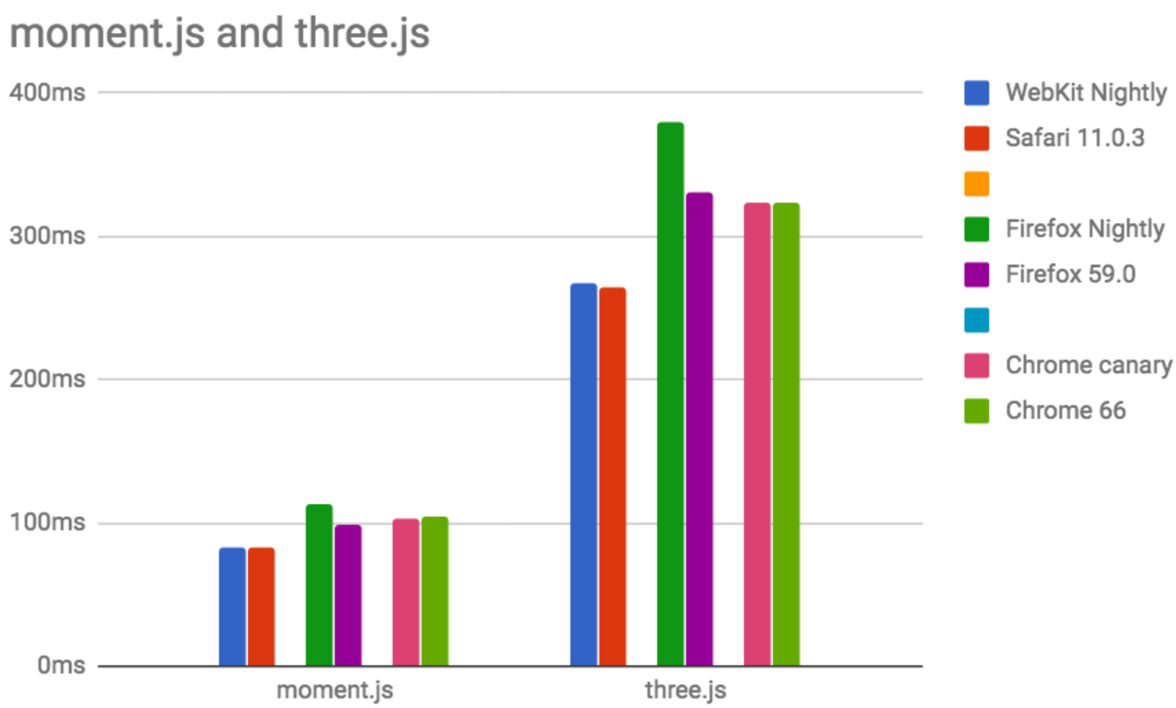
Environment:
- macOS High Sierra 10.13.3 on Mac Pro (Late 2013)
- Samples-module-loading-comparison test server on localhost

Browsers:
- WebKit r230903 (APRIL 22, 2018)
- Safari Version 11.0.3 (13604.5.6)
- Firefox Nightly 61.0a1 (2018-04-22)
- Firefox 59.0.2 (64-bit)
- Chrome Version 68.0.3403.0 (Official Build) canary (64-bit)
- Version 66.0.3359.117 (Official Build) (64-bit)

Raw results

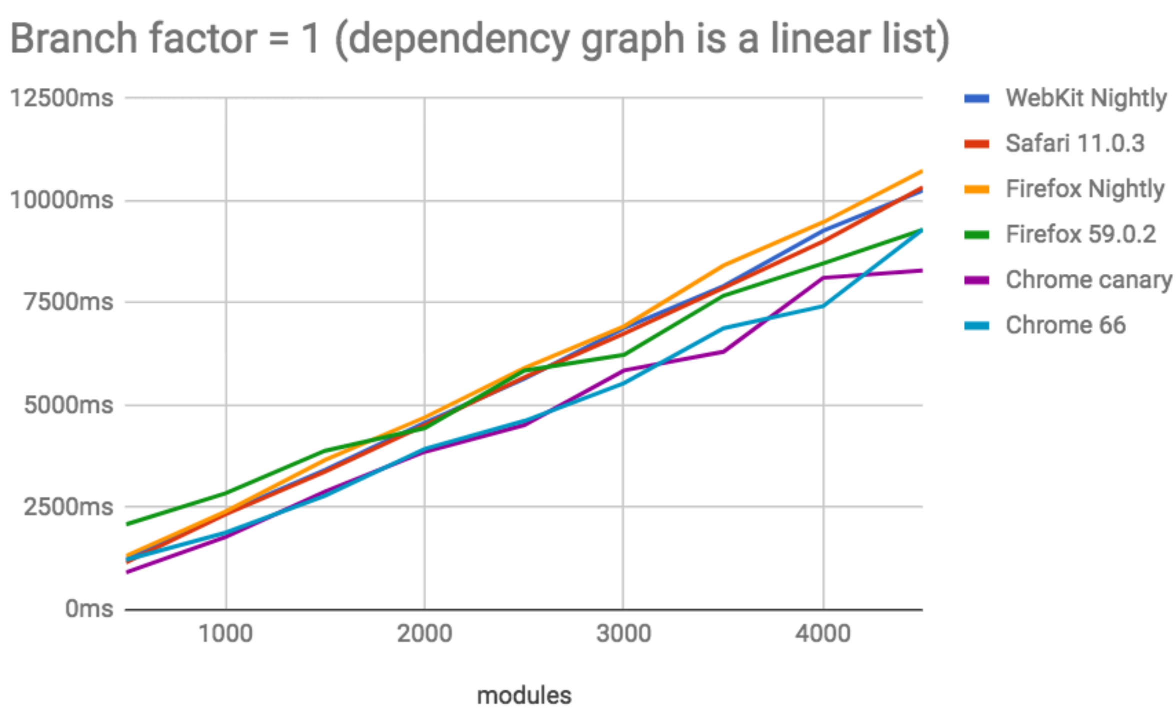
Moment.js / three.js

WebKit is 20% faster than Chrome and Firefox.



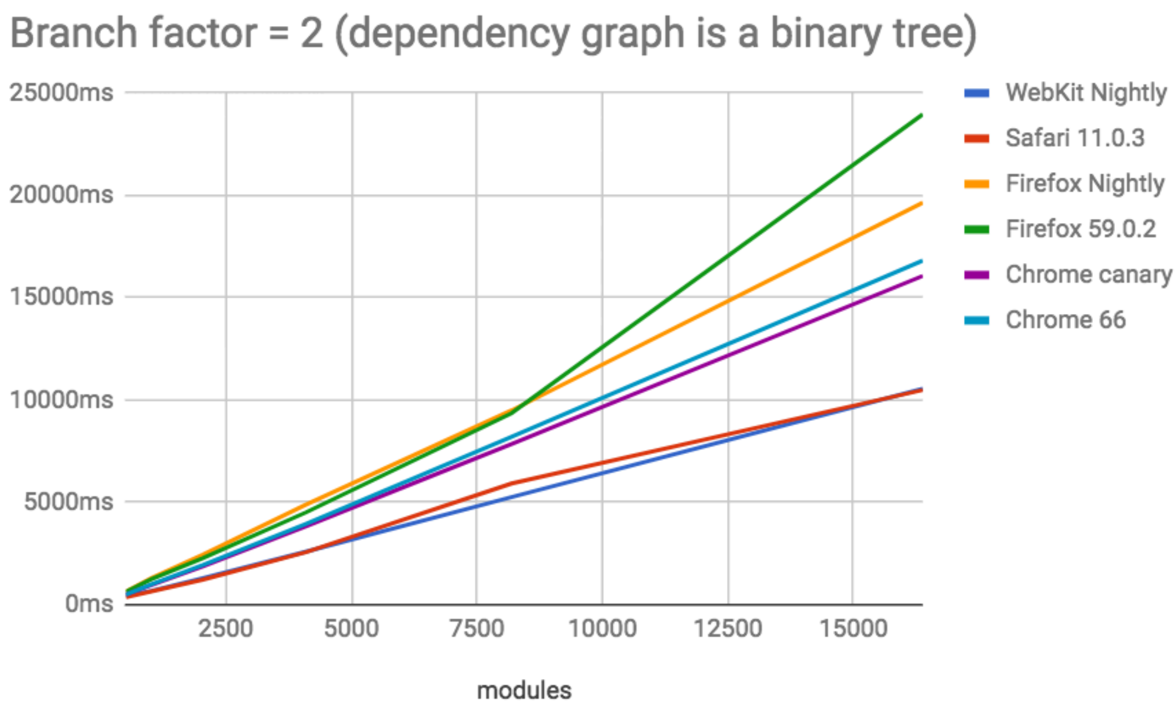
Synthesized (linear module graph)

Chrome is slightly faster than WebKit and Firefox, but **Chrome crashes on 5000+ modules**.



Synthesized (binary module graph)

WebKit was on par with Chrome in Sep 2017, but now WebKit is 1.5x faster than Chrome!



2017

Nov 28: Renderer CPU time accounting

Based on the profile result.

Total: 8.76s (incl. Samples from non-main threads)
RunLoop::Run 7.22s
blink::scheduler::TaskQueueManager::ProcessTaskFromWorkQueue 5.60s

Request

- ModuleTreeLinker::FetchDescendants 1.56s
    - RenderFrameImpl::UpdatePeakMemoryStats 0.36s
    - ThrottlingURLLoader::StartNow 0.51s

Response

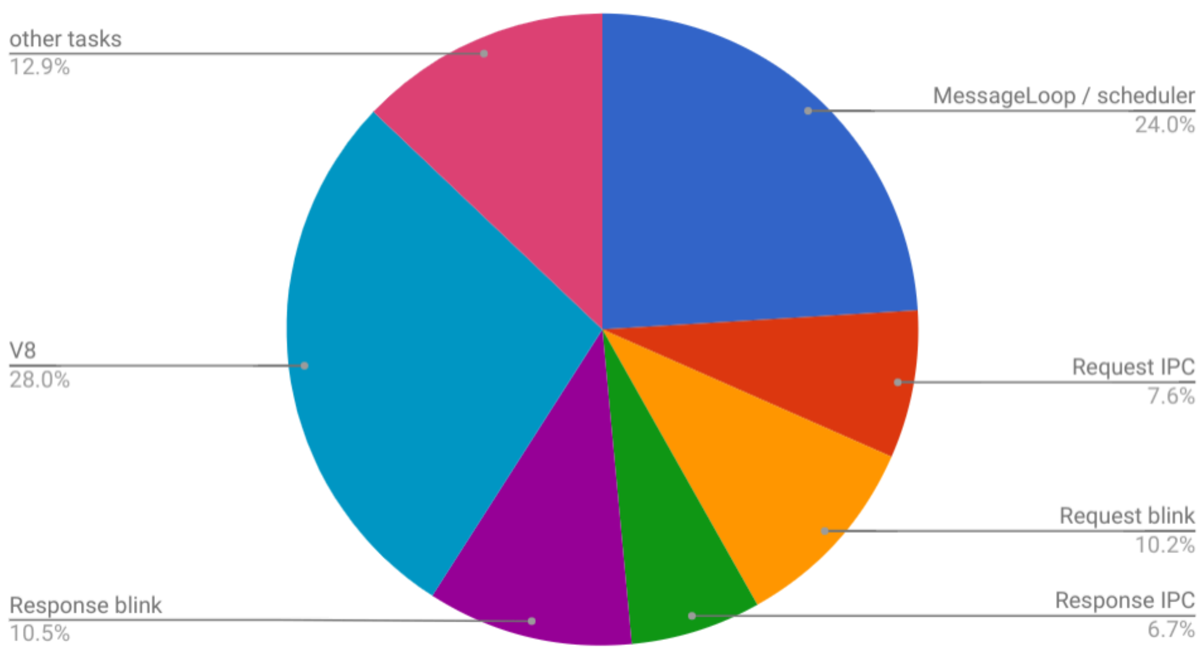
- IPC

- ○ mojom::URLLoaderClientStubDispatch::Accept 0.11s self
- ○ ThrottlingURLLoader::OnComplete 0.14s self
- ○ URLResponseBodyConsumer::OnReadable 0.20s self
- ResourceDispatcher::OnRequestComplete 2.11s
    - V8ScriptRunner::CompileModule 1.56s
- WebURLLoaderImpl::Context::OnReceivedResponse 0.28s
    - LocalFrameClientImpl::DidRunContentWithCertificateErrors 0.12s

Execution

- blink::ScriptLoader::Execute 0.33s

[Calculation]



- If we could get rid of “Request IPC” and “Response IPC” part by batching and reduce MessageLoop/scheduler part by half, it would cut at most (7.6%+6.7%+12.0%)=26.3% of the time.

Nov 21: Profiling with Google CPU Profiler

https://www.chromium.org/developers/profiling-chromium-and-webkit

Renderer process

Procedure:

1. Build with `enable_profiling = true` gn args
2. Run Chrome with renderer profiling enabled:

    - `out/Profile/chrome --no-sandbox --profiling-at-start=renderer https://localhost:44333`
3. Click “Moment.js unbundled”, and wait for result
4. Click “Three.js unbundled”, and wait for result
5. Close the Chrome window
6. Analyze the results:
    - `pprof out/Profile/chrome chrome-profile-renderer-NNN`

Renderer call graph (unfiltered)

This unfiltered view is hard to understand. By using show= option to only show nodes of blink / content layers, we can get a nice hierarchical view of tasks:

Renderer call graph (content / blink layers only)

To take closer look at each task, `focus=` option can be used to restrict samples to those going through a specific node. Here’s call graphs for specific tasks:

- ResourceDispatcher::OnRequestComplete (24.09%)
    - ScriptCompiler::CompileModule (17.12%)
- ModuleTreeLinker::FetchDescendants (16.55%)
    - ResourceFetcher::PrepareRequest (1.71%)
    - FrameFetchContext::CreateURLLoader (4.34%)
        - RenderFrameImpl::UpdatePeakMemoryStats is dominant (4.11%)
    - WebURLLoaderImpl::LoadAsynchronously (7.65%)
- ResourceDispatcher::OnReceivedResponse (3.42%)
    - RenderFrameImpl::DidRunContentWithCertificateErrors (1.37%)
        - Because the server uses a self-signed certificate

Observations:
- RenderFrameImpl::UpdatePeakMemoryStats is heavy (4.11%). Currently it’s called once per resource request, for ResourceLoadScheduler experimental groups.
- Modulator::ResolveModuleSpecifier (3.31%) is called 3 times for each module specifier, from ModuleScript::Create, ModuleTreeLinker::FetchDescendants, and ScriptModuleResolverImpl::Resolve. We should cache the resolved URL.

Browser process

- Browser-process call graph (unfiltered)
- Browser-process call graph (content:: and net:: only)

Oct 27: Response body inlining for mojo

Prototyped a mojo-loading version of IPC inlining, and measured the performance.

- Linux on Z620 workstation
- Chromium ToT @{#512063}
- Moment.js / Three.js unbundled test (median of 25 runs)
- Inlines response body up to 2KiB
  - 95 of 104 modules in moment.js and 259 of 333 modules in three.js are subject to inlining

Results:

	w/o inlining	w/ inlining
moment.js	111ms	97ms
three.js	355ms	324ms

- Reduced 12.6% for moment.js and 8.7% for three.js.

Oct 19: IPC inlining

Last year, tzik@ ran an experiment that inlines content of resource into IPC message (to avoid SharedMemory allocation), and reduces the number of IPCs per resource.
I revived that code (by reverting this) and benchmarked unbundle module loading.

- Linux on Z620 workstation
- Chromium ToT @{#509969}
- chrome://flags/#enable-mojo-loading disabled, since this optimization was built on top of legacy IPC
- Moment.js / Three.js unbundled test (median of 25 runs)

Results:

	w/o inlining	w/ inlining	mojo-loading enabled
moment.js	93ms	84ms	107ms
three.js	310ms	291ms	356ms

- Inlining made moment.js 9.7% faster and three.js 6.1% faster.
- For this test, Mojo-loading is slower than legacy IPC loading.

Tracing for single moment.js load:

- trace-momentjs-nomojo.json.gz
- trace_momentjs-nomojo-inlining.json.gz

Browser IO thread takes less time when inlined. ResourceLoader::PrepareToReadMore consumes 179us w/o inlining, 139us w/ inlining in average.

Oct 16: Sampling profiling

Profiled on Mac with Instruments.app (included with XCode).

- Chromium ToT @{#508970}
- Mac Pro 2013
- Procedure:
  a. Run server.go in samples-module-loading-comparison
  b. Open http://localhost:44333
  c. Launch “Time Profiler” in Instruments.app
  d. Attach to the {renderer, browser} process
  e. Start profiling
  f. Click “Moment.js unbundled”
  g. Click “Three.js unbundled”
  h. Stop profiling

Module-loading-profile.zip (you need Instruments.app to open trace files)

Renderer

Prettified call graph

Top 5 heavy tasks (colored in the spreadsheet):

- content::URLResponseBodyConsumer::OnReadable (2.51s)
  - Reads response body of a module and compile (first path)
  - URLResponseBodyConsumer destructor takes 321ms
- blink::ModuleTreeLinker::NotifyModuleLoadFinished (2.32s)
  - Fetches descendants of a module
- content::ThrottlingURLLoader::OnComplete (667ms)
  - Reads response body of a module and compile (second path)
  - mojo::internal::BindingStateBase::Close takes 238ms
- content::ThrottlingURLLoader::OnReceiveResponse (481ms)
  - Called when response headers are available
- blink::ScriptRunner::ExecuteTask (376ms)
  - Evaluates module after the all submodules fetched / compiled

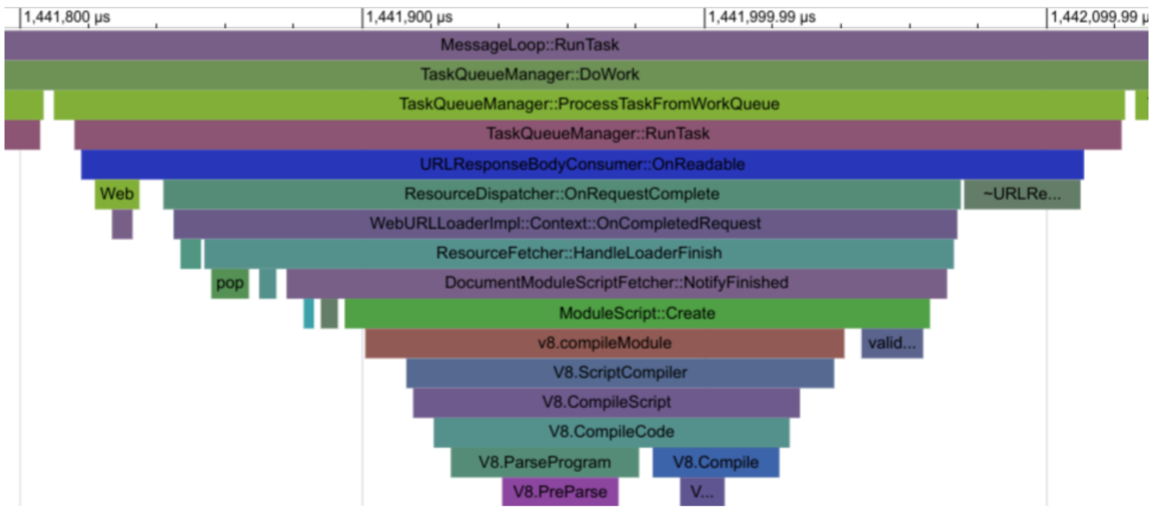
Browser IO thread

Prettified call graph

- It takes long time to post tasks? content::BrowserThread::PostTask() and base::PostTaskAndReplyWithResult show up in several places.
- Invert Call Tree shows that PostPendingTask takes 8% of the time and malloc takes 7%

Oct 13: Tracing inspection: response handling

Here's a trace during single module response handling, with Chromium ToT @{#508590} with additional tracing patch:



It seems there's no easy win, but a few observations:

- In average, it takes 300us to process single module response.
- V8 takes 160us.
- URLResponseBodyConsumer destructor takes 23us.
- Populating ResourceTiming takes 19us.
- ModuleScript::Create takes 13us to validate module specifiers, which is done again in ModuleTreeLinker.

Oct 2: Tracing comparison: unbundled vs bundled

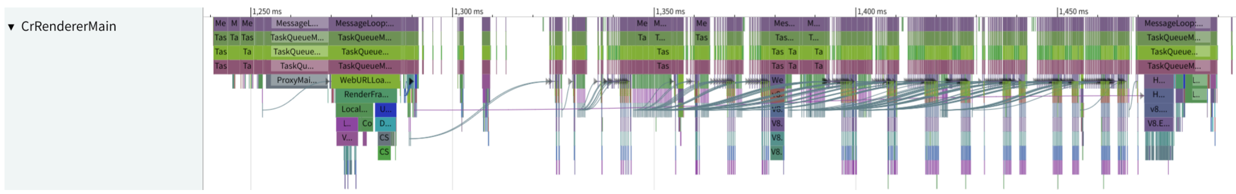
See report doc here

- Test: moment.js in samples-module-loading-comparison
- Chromium ToT @{#505529} with modulepreload patch
- Linux on Z620 workstation
- Four cases:
    - Unbundled: unbundled moment.js (104 modules)

- Unbundled+modulepreload: Injected <link rel=modulepreload> for all modules
- Bundled-unoptimized: minified, unoptimized single classic script. Bundled using webpack with tree-shaking disabled (66kb)
- Bundled-optimized: optimized single classic script (50kb)

Unbundled

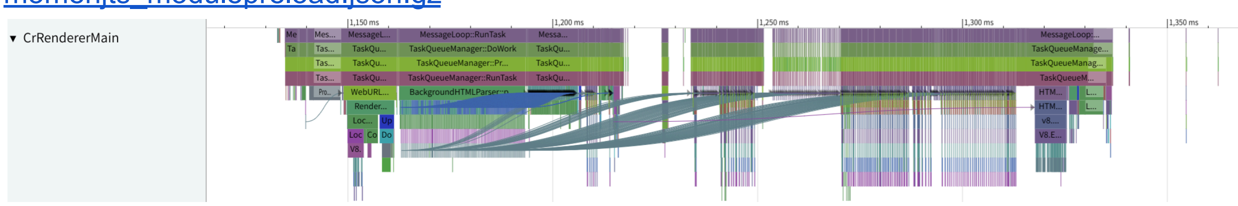
[momentjs_unbundled.json.gz]



Timestamp	⊿Time	Event
1287ms	0ms	HTML ParseStart
1288ms	1ms	Root module (app.js) requested
1325ms	38ms	app.js arrived
1471ms	184ms	All modules fetched and ExecuteScript started
1487ms	200ms	Finished

Unbundled + modulepreload

[momenjts_modulepreload.json.gz]

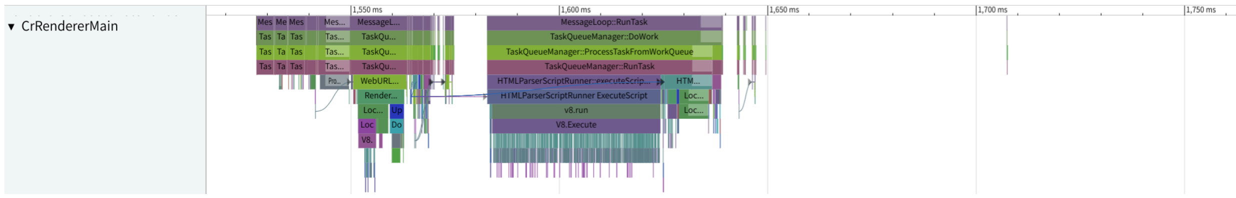


Timestamp	⊿Time	Event
1162ms	0ms	HTML ParseStart
1162~1192ms	0~30ms	HTMLPreloadScanner kicks fetches for all the 104 modules
1193~1205ms	31ms~43ms	<link> tags attached to the DOM, creating another preload requests

1317ms	155ms	All modules fetched and ExecuteScript started
1335ms	173ms	Finished

Bundled-unoptimized

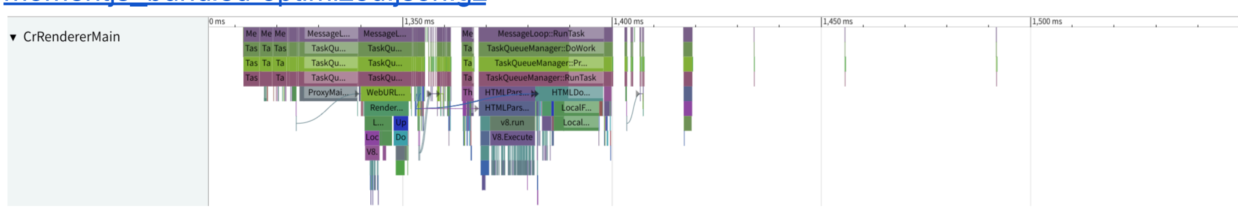
momentjs_bundled-unoptimized.json.gz



Timestamp	⊿Time	Event
1564ms	0ms	HTML ParseStart
1565ms	1ms	app.js requested
1572~1581ms	8~17ms	app.js gets parsed in ScriptStreamer thread (not shown in the screenshot)
1582ms	18ms	ExecuteScript started
1636ms	72ms	Finished

Bundled-optimized

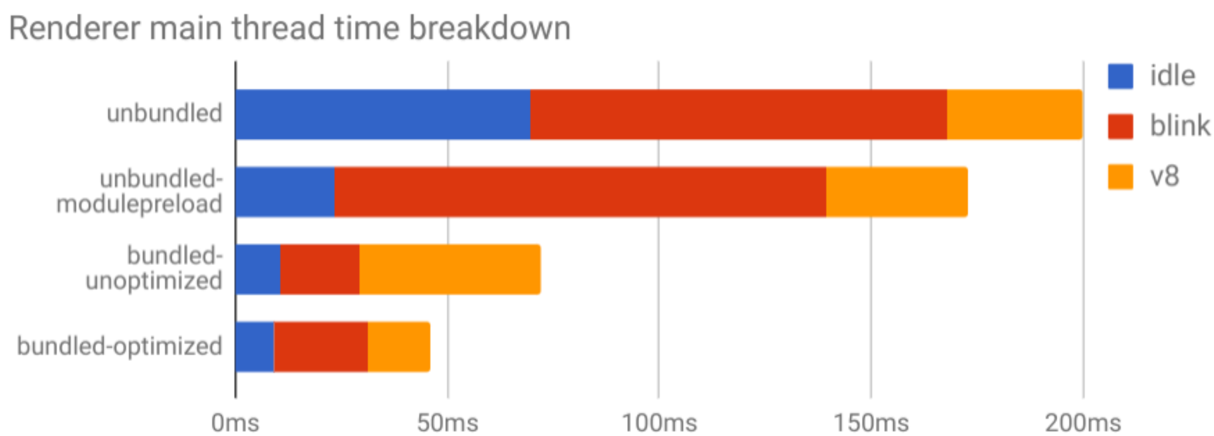
momentjs_bundled-optimized.json.gz



Timestamp	⊿Time	Event
1352ms	0ms	HTML ParseStart
1353ms	1ms	app.js requested
1358~1377ms	6~15ms	app.js gets parsed in ScriptStreamer thread (not shown in the

		screenshot)
1368ms	16ms	v8.compile on main thread
1370ms	18ms	ExecuteScript started
1398ms	46ms	Finished

Time breakdown



(spreadsheet)

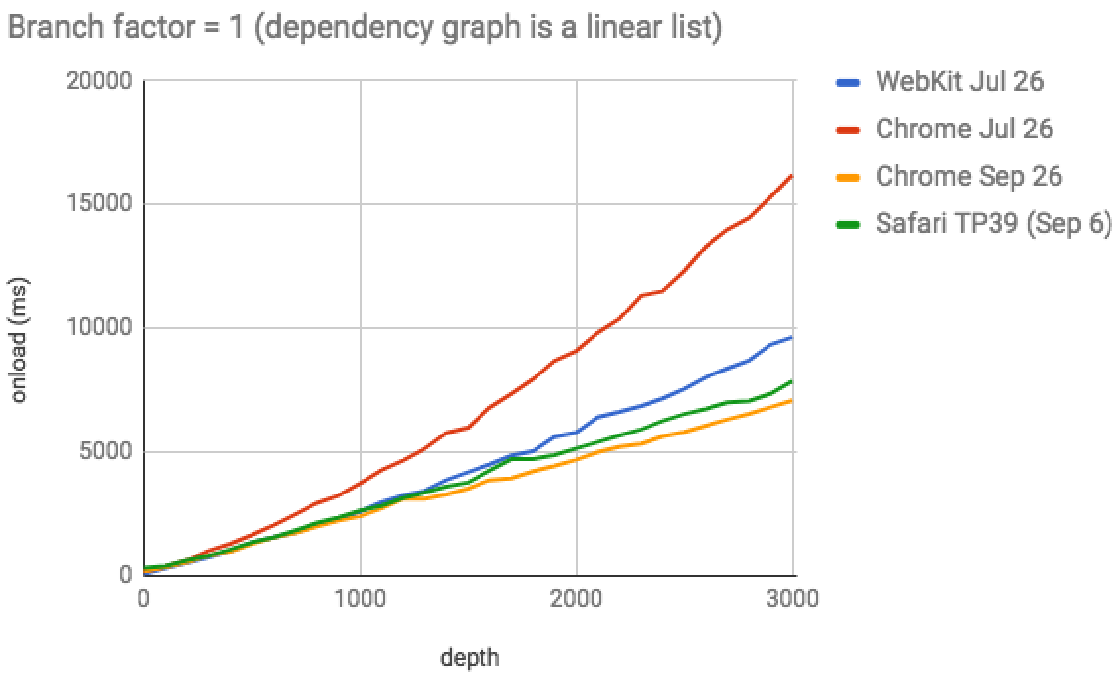
- In unbundled case, renderer main thread is 70ms (35%) idle waiting for network resources
- Modulepreload reduces idle time but increases blink overhead
- In bundled-unoptimized case, v8 time has increased compared with unbundled cases. Bundling overhead?
    - 32ms when unbundled -> 42ms + 9ms in ScriptStreamer thread
- In bundled-optimized, v8 time is 16ms in main thread + 9ms in ScriptStreamer thread

Sep 26: Synthesized tests revisited

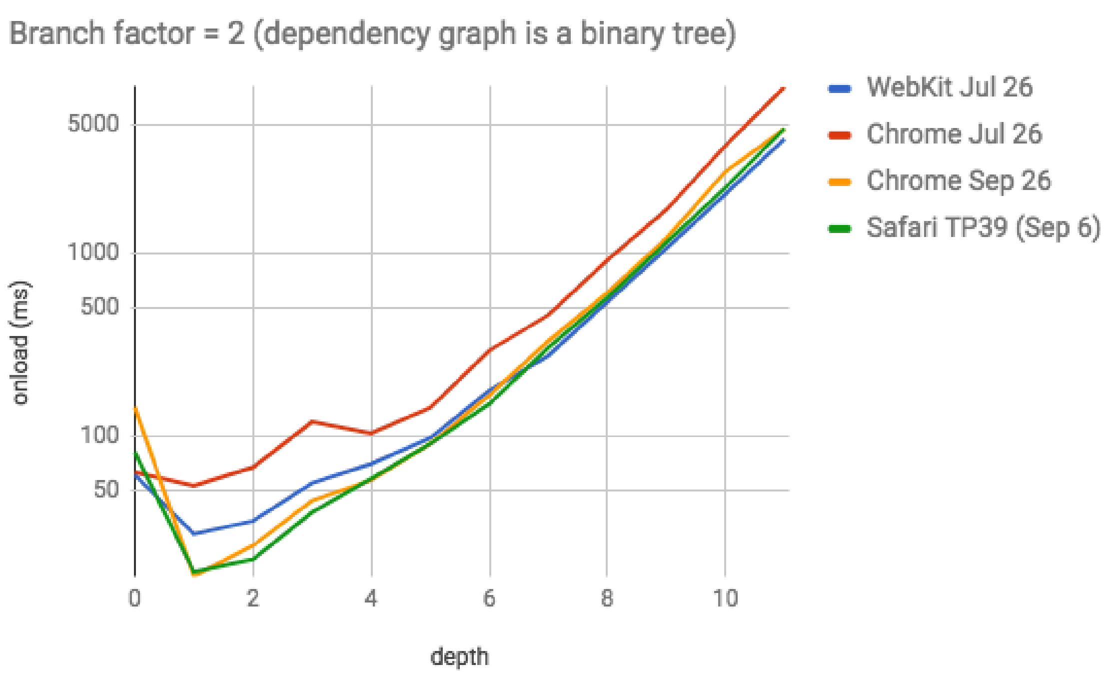
The new algorithm of module tree fetching has been landed, so I did measurement for synthesized module graphs again.

See results in spreadsheet

- branch=1
    - Chrome's performance is now close to linear, as the O(n^2) operation has been removed. Actually Chrome is faster than Safari for this (unrealistic) test.



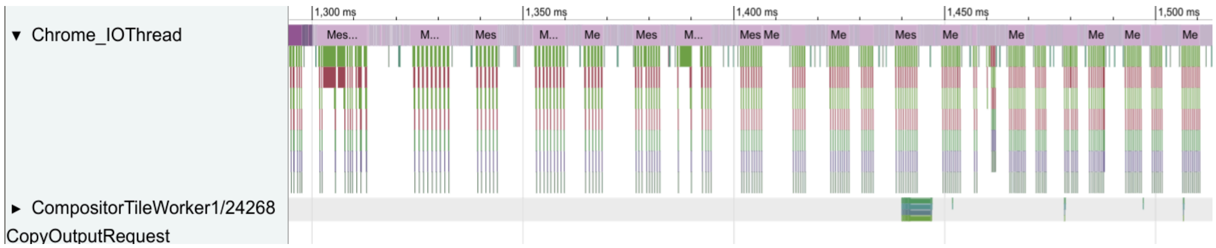
- branch=2
    - Now Chrome is on par with Safari. (Was 1.5x~2x slower than Safari in July)



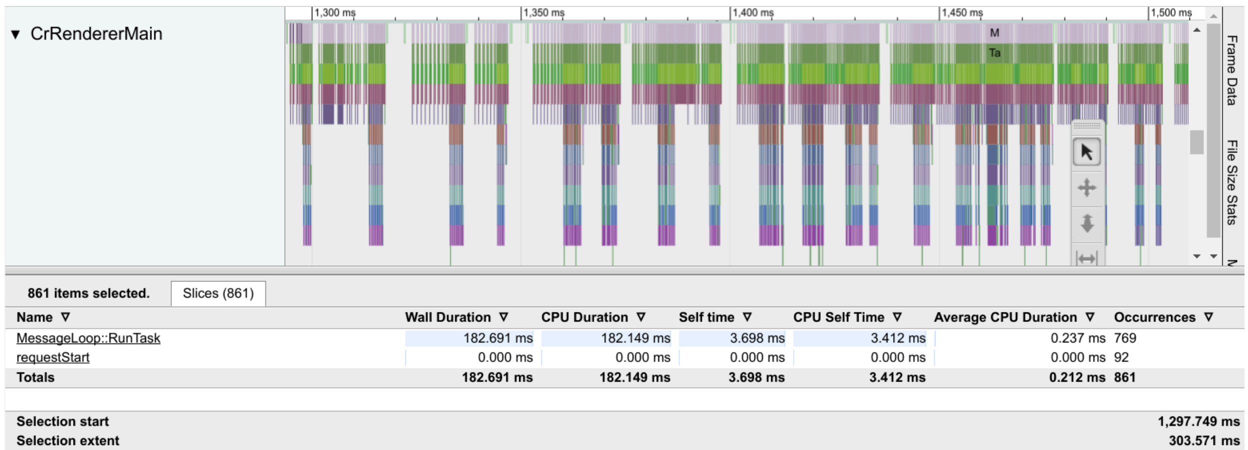
Sep 5: Where is the bottleneck?

Here is the trace while loading modules of unbundled Three.js test.

Browser IO thread:



Renderer main thread:



Name	Wall Duration	CPU Duration	Self time	CPU Self Time	Average CPU Duration	Occurrences
MessageLoop::RunTask	182.691 ms	182.149 ms	3.698 ms	3.412 ms	0.237 ms	769
requestStart	0.000 ms	0.000 ms	0.000 ms	0.000 ms	0.000 ms	92
Totals	**182.691 ms**	**182.149 ms**	**3.698 ms**	**3.412 ms**	**0.212 ms**	**861**

Selection start	**1,297.749 ms**
Selection extent	**303.571 ms**

Renderer main thread is only 60% busy while browser IO thread is 100% busy. This means we have to make IO handling faster (e.g. batching IPC etc.) in order to improve module loading speed.

Aug 16: Test server update - repetitive loading test

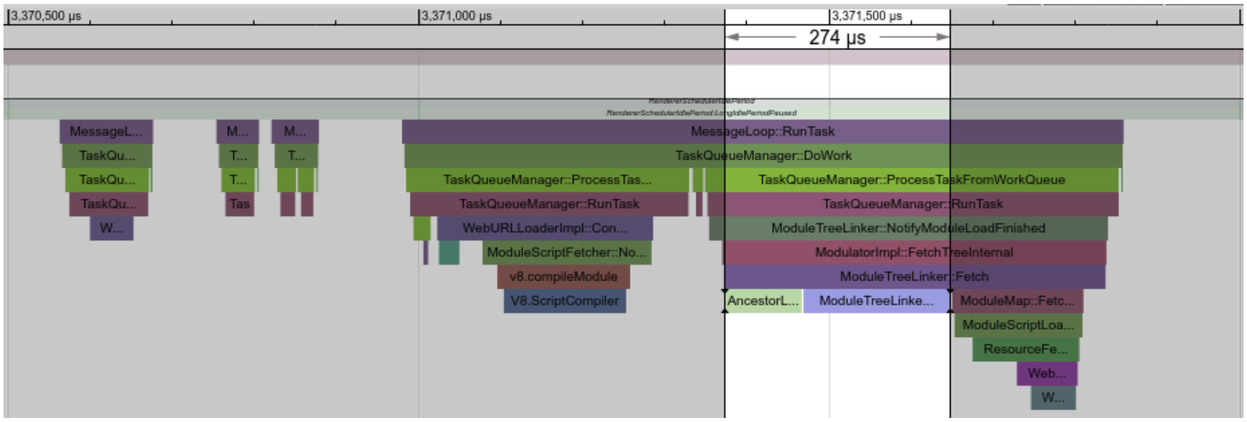
In order to reduce variability, added a functionality to load the module tree repeatedly (under different URLs) and report the median time. (commit)

Aug 9: Upstreamed test server

The Go test server (with synthesized test support) is now part of the samples-module-loading-comparison repo.

Aug 2: A finding in deep-dependency case

In ModuleTreeLinker, HashSet for the ancestor list is copied twice, in Fetch() and in the constructor. This can be slow when dependency is very deep:



This could be a reason why [Chrome gets slow as #depth increases](#).

Note: This wouldn't be a problem in the new algorithm, as the ancestor list will be going away.

Jul 28: Modules graph visualization

Download: visualizable-module-loader-bundled.js

Built on top of the ES Module Loader Polyfill.

Usage:

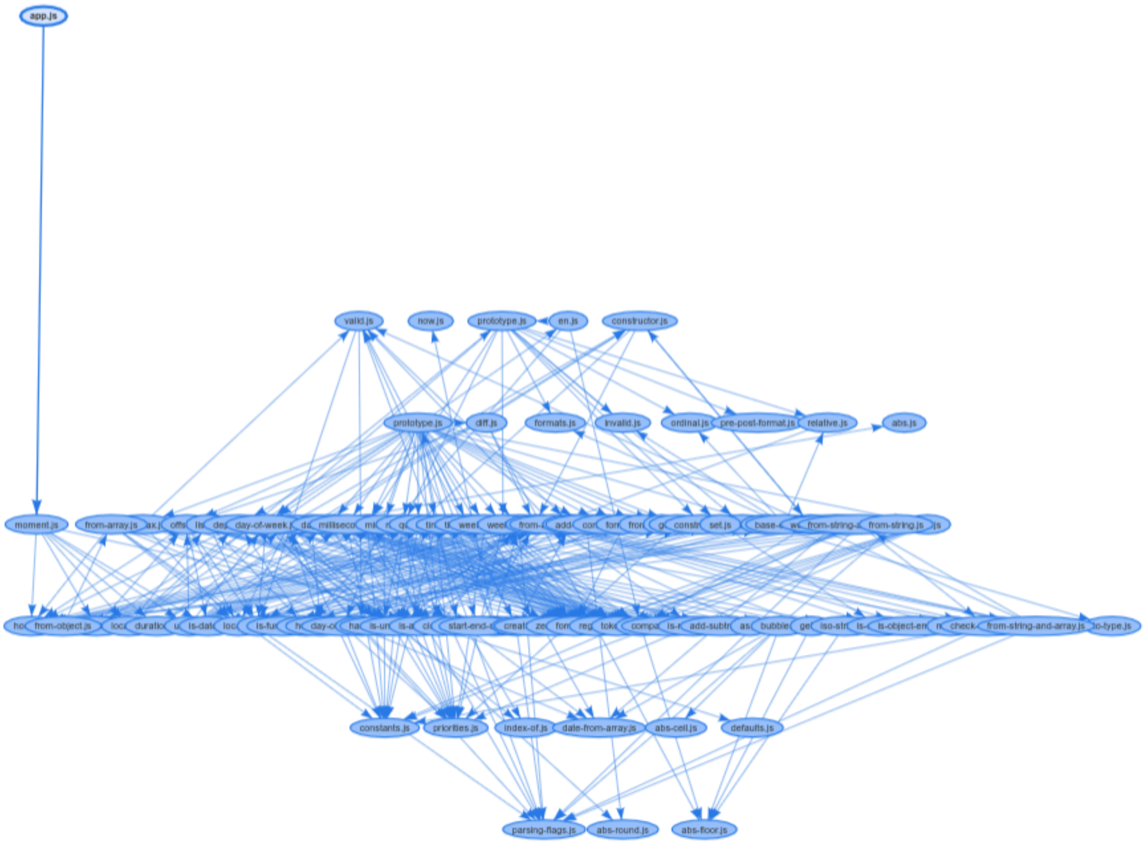
```
<script src="path/to/visualizable-module-loader-bundled.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vis/4.20.1/vis.min.js"></script>

<div id="module-graph"></div>

<script>
  // Replace <script type="module" src="module.js"> with this:
  var loader = new VisualizableModuleLoader();
  loader.import('module.js').then(() => {
    loader.visualize(document.getElementById('module-graph'));
  });
</script>
```

Example: moment.js, three.js

Update Oct 10: Changed to layout hierarchically by depth of nodes



Jul 27: Test automation

Updated the test archive: [module-loading-bench.tar.gz]

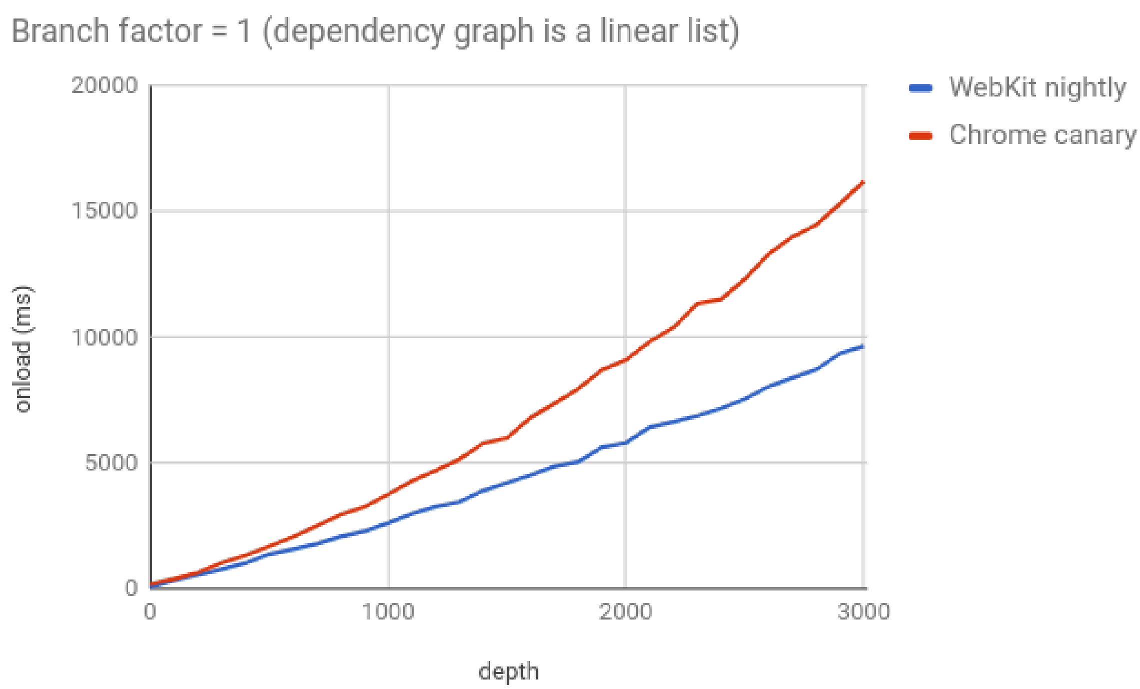
- Added a script to run the synthesized test repeatedly, using WebDriver. See runner/README.md for details.

Jul 26: Synthesized test case results

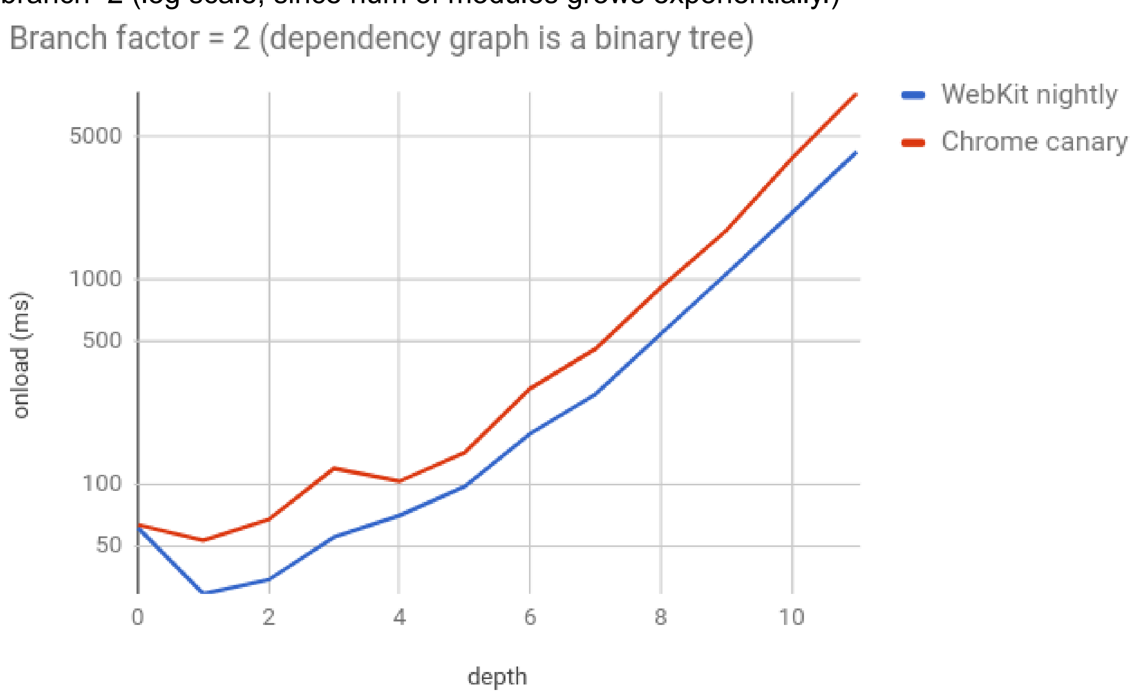
Measured the time to onload, changing the depth of dependency graph.

[See results in spreadsheet]

- branch=1
  - For depth=100, Chrome is 1.2x slower than WebKit. For depth=3000, Chrome is 1.7x slower than WebKit.



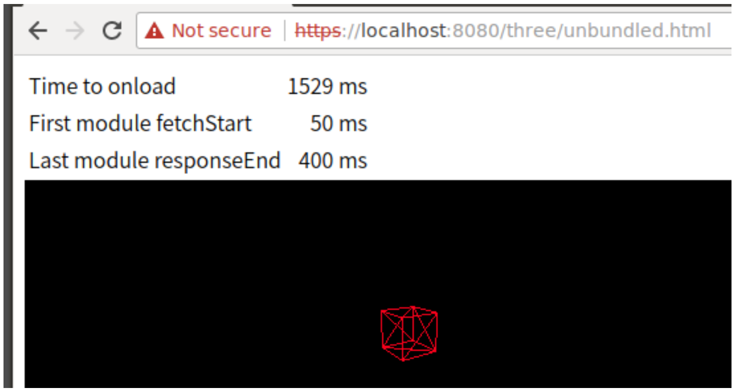
- branch=2 (log scale, since num of modules grows exponentially.)



Jul 25: Server update

Module-loading-bench.tar.gz

- Shows window.performance based timings



- Added synthesized module-tree test case
    - You can specify the shape of module dependency tree by the query parameters `depth=n` and `branch=m`. See README.md for details.

	Chrome 62.0.3166.0 canary	Safari TP35	WebKit nightly r219860
depth=10, branch=2 (2047 modules)	3674ms	2249ms	2165ms
depth=2, branch=45 (2071 modules)	2554ms	1955ms	2070ms
depth=2000, branch=1 (2001 modules)	8899ms	5986ms	5823ms

Jul 24: Unbundling Vue.js

Just to get some idea about how hard to add a new test case.

TodoMVC with unbundled Vue.js (108 modules)

Cloned vue.js repo and did the followings:

- Stripped flow types by flow-remove-types
- Rewrote import statements to browser-loadable relative paths (using ad-hoc ruby script)
- ..and modify several places manually to get it working

Takeaway: It's hard without knowledge about modern JS build system and a systematic approach, like using rollup / babel plugins Sergio used in his test.

Jul 21: Go test server

Ported the HTTP/2 server used in Sergio's test into Go language, to see if the server is adding any performance overhead.

Download link: module-loading-bench.tar.gz

Here's the result of loading the "Unbundled" tests three times for each server. (Time to onload, Chrome Canary 62.0.3164.0 on Macbook Air)

Unbundled moment.js (104 modules)

	node-spdy	Go
1st	424ms	371ms
2nd	433ms	354ms
3rd	440ms	363ms

Unbundled three.js (333 modules)

	node-spdy	Go
1st	1.18s	1.04s
2nd	1.17s	1.07s
3rd	1.16s	1.05s

The Go implementation is 10%~15% faster.

Jul 20: Try Sergio's test on latest Chromium

https://sgom.es/posts/2017-06-30-ecmascript-module-loading-can-we-unbundle-yet/

- It worked without modification
- But unbundled moment.js is very slow (3.8s on Z620)
    - -> Kouhei's patch made it 20x faster!

Safari TP35 vs Chromium after the Kouhei's patch (Macbook Air, no network shaping)

	Safari	Chromium
Moment.js bundled, optimized	23ms	61ms

Moment.js bundled, unoptimized	33ms	72ms
Moment.js unbundled	328ms	439ms
Three.js bundled, optimized	65ms	182ms
Three.js bundled, unoptimized	79ms	258ms
Three.js unbundled	915ms	1190ms