Mid-Tier Compiler Investigation

Attention: Externally visible, non-confidential

Author: rmcilroy@chromium.org, mythria@chromium.org

Status: Draft | Final

Created: 2019-08-19 / **Last updated:** 2019-08-23

Background

As part of the V8 Mobile London hackathon, we experimented with the idea of adding a middle tier compiler between Ignition and TurboFan. This document outlines the findings from this analysis and follow-up prototyping. The intention is to investigate the potential benefits of a middle-tier JS compiler and evaluate a number of different options design decisions, particularly focusing on the performance of web content on low-mid end mobile devices.

TLDR;

There is an opportunity to improve JavaScript performance for certain web pages and benchmarks by introducing a new middle tier compiler between Ignition and TurboFan. The current optimization tick threshold means that on typical web page interactions (e.g., v8.browsing_mobile stories) only between 10-30% of bytecode execution is eligible for optimization by TurboFan (in practice less is optimized due to IC updates and deoptimizations). A reasonable mid-tier tick-threshold would open an additional 35-55% of bytecode execution to be eligible for executed in a faster mid-tier.

Four different prototype mid-tier implementations were developed to evaluate the potential performance benefits of different mid-tier compiler designs:

- **SparkPlug:** a single-pass non-optimizing baseline compiler
- **SparkPlugOpt:** a single-pass baseline compiler with speculative optimization of certain monomorphic operations
- **TurboFan-Lite:** Turbofan with inlining disabled
- **TurboProp:** an optimizing compiler that uses a cut-down variant of the TurboFan pipeline with an attempt to approximate the impact of a different backend for TurboFan

Evaluation of these prototypes for real world webpage interactions suggests that the most promising approach would be a TurboProp-like pipeline. While a baseline compiler like SparkPlug provides some potential benefits, with relatively small total CPU time impact, the savings on JavaScript execution are limited to likely less than 5%. TurboProp, on the other hand, provides most of the JavaScript execution improvements of TurboFan-Lite (between 5-20% on a number of pages), while alleviating the high total cpu regressions that would result from running the full TurboFan pipeline with a more aggressive tiering-up tick thresholds for a mid-tier compiler.

As a result of this investigation, we propose a follow-up plan to further develop this TurboProp prototype and evaluate its impact.

Analysis

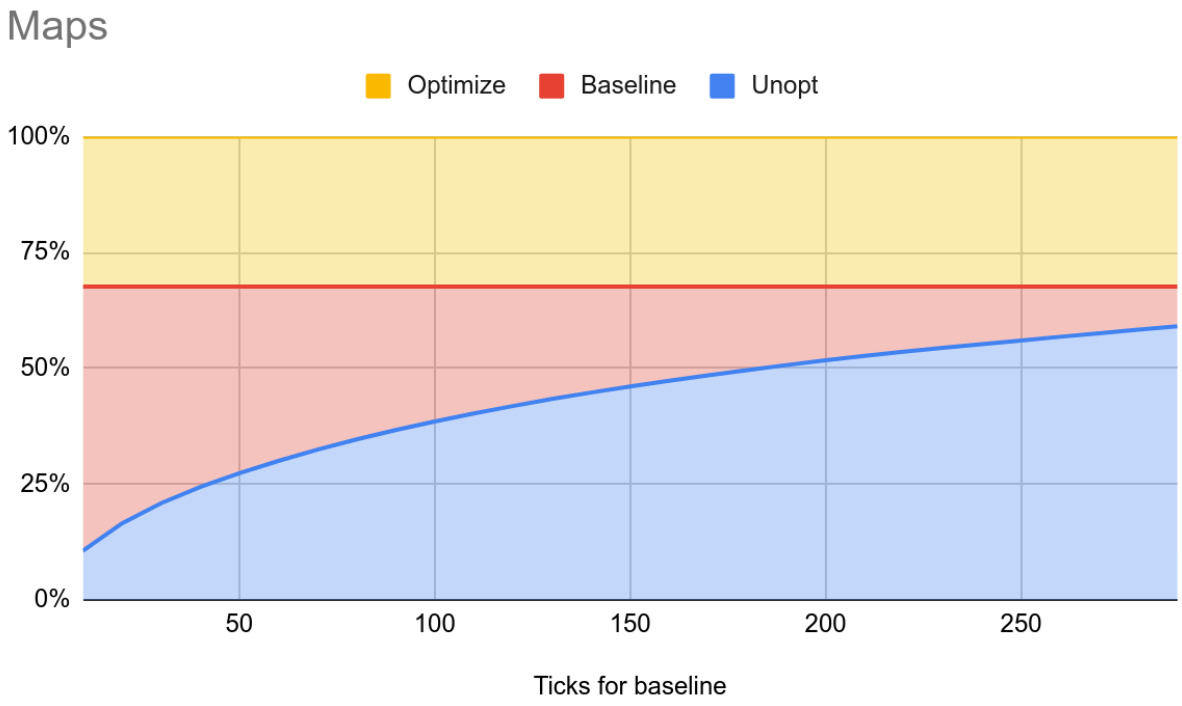
In order to estimate the effectiveness of a mid-tier compiler, we first need to determine whether there is a substantial amount of JavaScript execution which is hot enough to be tiered up to this middle tier, but not hot enough to be optimized by TurboFan.

In regular V8 we trigger a profiler tick after 144KB of bytecode has been executed by a function (e.g after a function of 1024 bytes is called 144 times, or a 1024 byte loop iterates 144 times). We will typically optimize a function after 2 profiler ticks, although we wait more ticks if a function is larger than 1200 bytes. We also reset the profiler tick count any time an IC records new type

feedback in order to avoid optimizing code that has unstable type feedback (which would likely deopt).

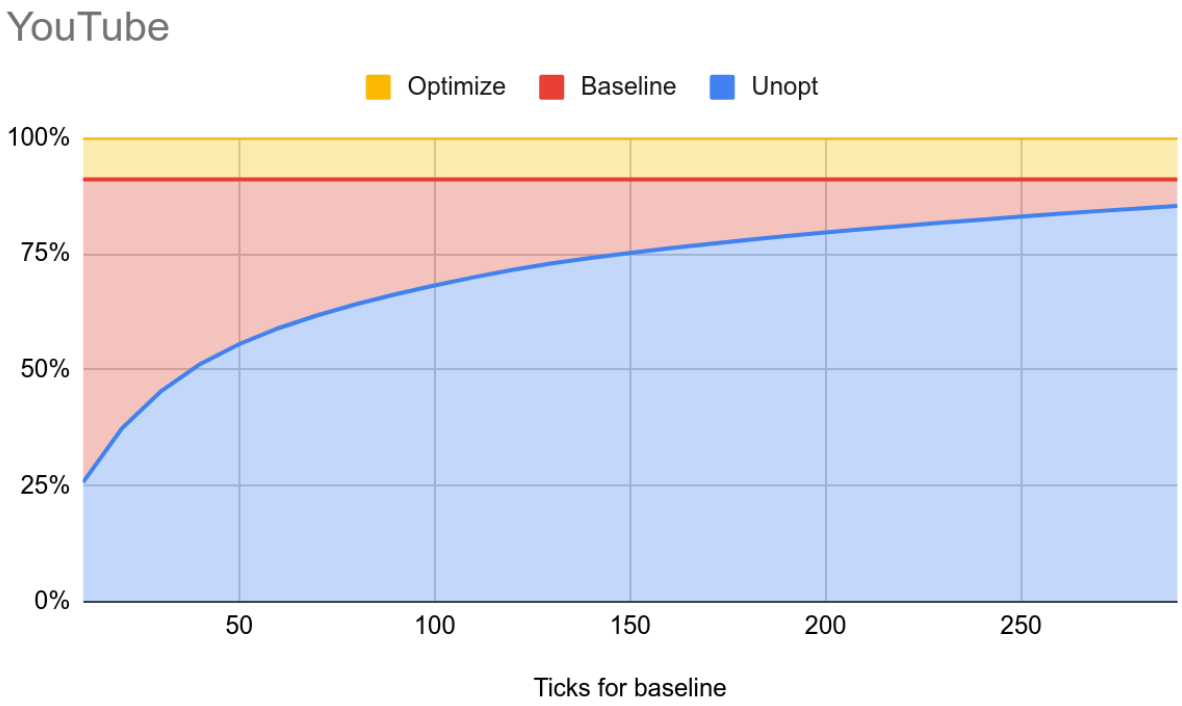
For the purposes of this analysis (and the rest of the investigation in this document) we reduced the bytecode execution budget necessary to cause a profiler tick to 1KB of bytecode execution (i.e., 144 times less than regular V8), and increased the profiling ticks necessary to cause optimization accordingly. We also removed the logic to reset ticks on IC updates in order to make tick counts a stable measure of the amount of JS code being executed (irrespective of optimization level).

In order to determine how much time might be spent in a middle tier compiler, we profiled the number of ticks spent executing each function on three pages of the v8.browsing_mobile stories. The graphs below show the proportion of time spent executing in each tier as a percentage of total ticks. This analysis is based entirely on the tick thresholds for tiering up a function to the next optimization level, namely 288 ticks to tier up to TurboFan and the value in the x-axis as the threshold to tier-up to the mid-tier compiler. This is an optimistic view, both due to the fact that in V8 we reset tick counts on IC updates, but also because it doesn't take deoptimization (i.e., tiering-down) into account.

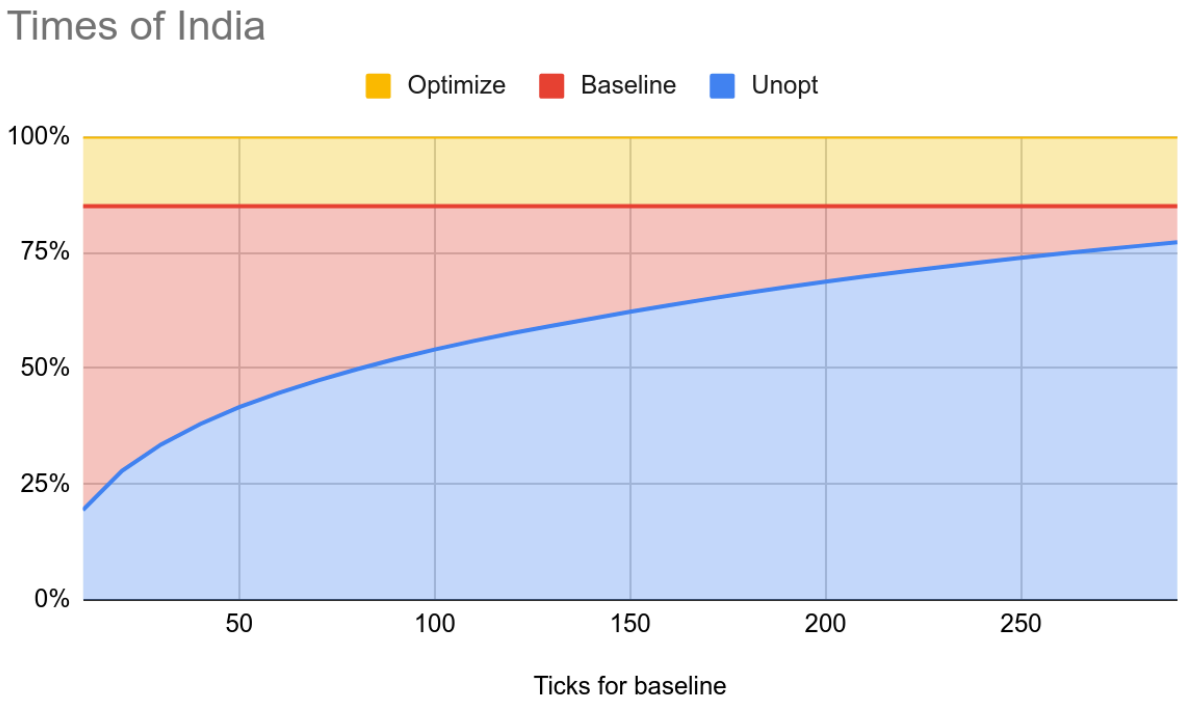


On Google Maps, a good proportion of code is already potentially executable by TurboFan (around 32% of ticks). However there is still scope for executing more code in a middle tier, with around 40-50% of execution being possible in a mid-tier compiler if we would tier-up at between

20-50 profiler ticks.



YouTube spends much less time in TurboFan (around 9% of ticks), with a middle tier, potentially having between 35-55% of execution if tiering-up at between 20-50 profiler ticks.



Times of India also spends less time in TurboFan (around 15% of ticks, although we know from experience that in the real-world it is even less due to type feedback updates), with a middle tier, potentially having between 45-55% of execution if tiering-up at between 20-50 profiler ticks.

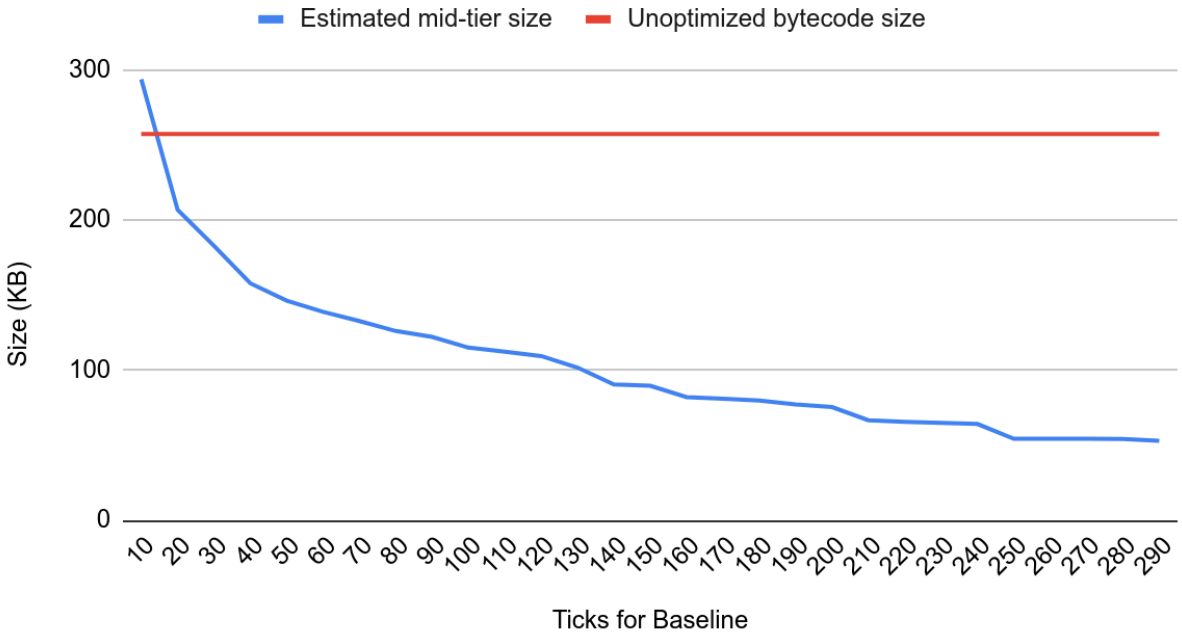
Overall it seems like there is an opportunity for a mid-tier compiler to execute a reasonable proportion of JavaScript code for web content, assuming we tier-up at around 50 profiler ticks or lower.

As well as determining whether a reasonable proportion of execution would be spent in a mid-tier, we also want to ensure that the code generated by a mid-tier compiler doesn't take up too much memory.

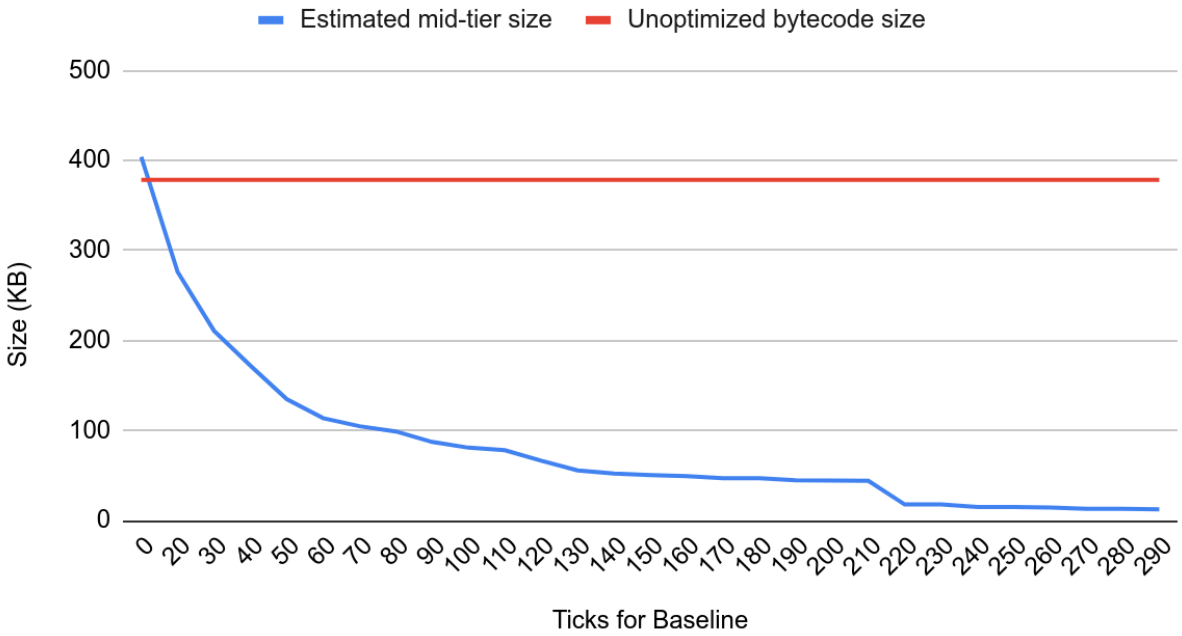
Looking at the size of the functions which get tiered-up, we see that between 10-15% of the executed bytecode will be compiled by a mid-tier compiler if we tier-up at between 30-50 ticks.

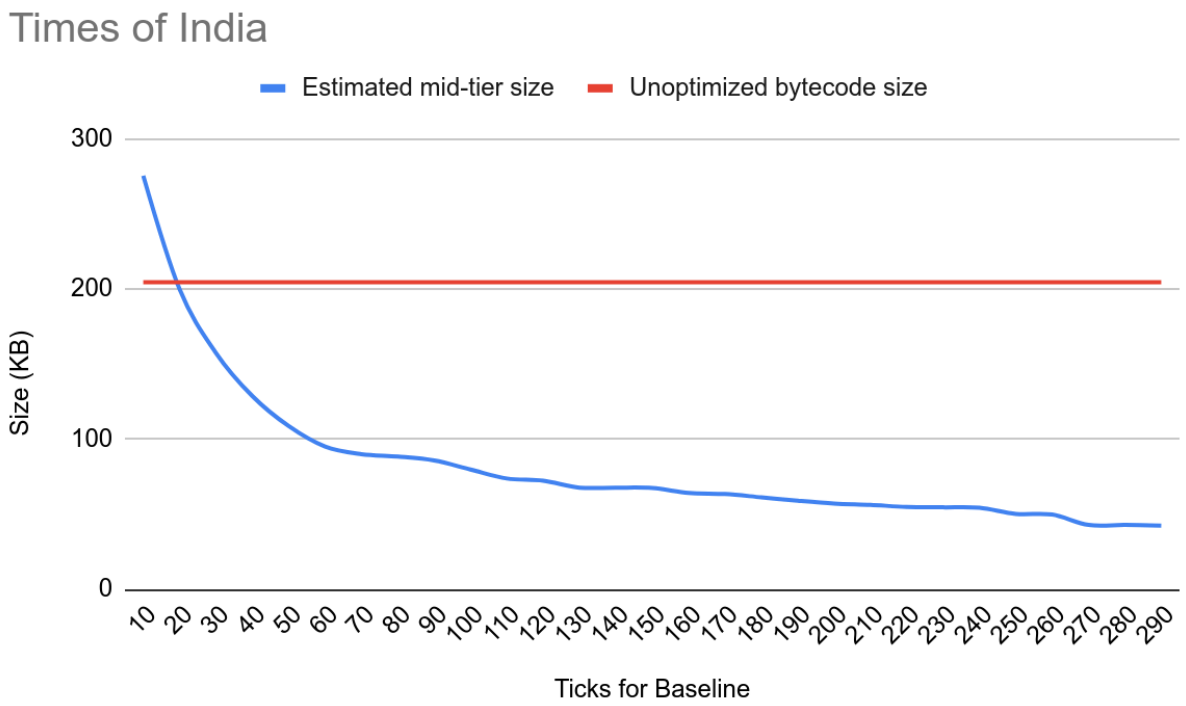
We estimated the size of mid-tier machine code would be around 4x larger than the corresponding bytecode. With this assumption, the code generated by a mid-tier compiler would likely be between 30-60% the size of the bytecode generated by Ignition on the heap if we tier-up between 30-50 ticks. The graphs below show this estimated size across the range of tier-up thresholds.

Maps



YouTube





Prototype Implementations

To evaluate the performance opportunities of various mid-tier compiler options, we implemented a variety of prototype mid-tier compilers and evaluated their performance on various benchmarks on a low-end Nokia 1 device.

All prototype implementations were based on V8 version 6.8.158, Chrome revision r555955.

The following prototypes were implemented:

SparkPlug

SparkPlug is intended to model a fast single-pass compiler (although given its use of CSA, it is actually multi-pass). It iterates through the bytecode, generating code for each bytecode directly (with potential deferred code for slow-paths).

Simple bytecodes (e.g., Ldar, Star, LdaContextSlot, etc.) are directly compiled into inline code, while more complex bytecodes (Add, LdaNamedProperty, Call, etc.) are compiled into calls to builtins which performs exactly the same operations as the equivalent bytecode handler, but avoid the bytecode decoding and bytecode dispatch logic. These builtins collect exactly the same type feedback as would be collected by Ignition (unlike the equivalent builtins used by TurboFan for generic lowering), therefore optimized code generated by TurboFan is equivalent whether it was executed by Ignition or SparkPlug beforehand.

SparkPlug supports tier-up to optimized code. SparkPlug code updates the interrupt budget on Jump and Return bytecodes, and will call into the runtime profiler if the interrupt budget is exhausted, which can trigger optimization of the current function by Turbofan. SparkPlug also emits prologue code that checks the OptimizedCode field of the feedback vector on function entry, and tail-calls a builtin that will tail-call the optimized code (if it exists) or the runtime (if the field has an optimization marker set).

More details of the implementation of SparkPlug are available in the previous analysis, with the code being available here.

SparkPlugOpt

SparkPlugOpt is intended to model a fast single-pass compiler that does speculative optimizations based on type feedback. It is a variant of SparkPlug which performs two optimizations based on the type-feedback in the FeedbackVector at the time of baselining:

- Inlining of Smi binary and unary ops
- Inlining of named property loads in the case of Monomorphic ICs

If the type feedback assumptions made by SparkPlugOpt are violated, the code will jump to a deferred code which calls a BailoutFromBaselineCode, to return execution to Ignition. Since SparkPlug's stackframe is laid out the same as an interpreter stack frame, all BailoutFromBaselineCode needs to do is drop any additional spill slots inserted, then re-enter Ignition using Generate_InterpreterEnterBytecode in the same way as exception catching or deoptimization re-enters Ignition.

Unlike the previous analysis, in this investigation SparkPlugOpt was modified to *de-baseline* the function when it bails out, after which Ignition will continue to collect type feedback and the function can re-tier-up to SparkPlugOpt when it reaches the profiler tick threshold again.

More details of the implementation of SparkPlugOpt are available in the previous analysis, with the code being available here.

TurboFan-Lite

The inspiration for TurboFan-Lite came from investigations by the compiler team during the V8 Mobile London Hackathon that turning off TurboFan function inlining caused a significant (~45%) reduction in the time TurboFan spent optimizing, and didn't seem to have much impact on JavaScript execution performance for the web page benchmarks.

TurboFan-Lite in this analysis is simply Turbofan with inlining disabled (--no-turbo-inlining). As such, it is not a mid-tier (we don't later tier-up to full TurboFan), but a variant of TurboFan to see

the performance impact of a fully multi-pass optimizing compiler when used for earlier optimization (e.g., as a mid-tier).

In this analysis we also disabled concurrent optimization because we are specifically interested in the impact of optimization on JavaScript execution time, and by optimizing on the main thread we remove the variable of optimization time impacting JavaScript execution time due to the function continuing to be interpreted until optimization completes. This gives an optimistic picture of how much JavaScript execution time might be reduced, since we in-effect pause execution until the function has been fully optimized. In the analysis we estimate the proportion of optimization time which would likely be spent on the main thread and on a background worker thread.

TurboProp

TurboProp is intended to model a mid-tier compiler that uses TurboFan's machinery, but has a much reduced optimization pipeline and a significantly cheaper back-end. The inspiration came from looking at the time spent in each phase of the TurboFan-Lite prototype and determining that many of the expensive phases were optional, and the majority of the remaining optimization time was spent in the backend after effect-control-linearization (namely the schedule, instruction selection and register allocation phases).

The TurboProp prototype disables the following optimization passes entirely:

- LoopVariableInduction
- LoopPeelingPhase
- LoadEliminationPhase
- EscapeAnalysisPhase
- EarlyOptimizationPhase
- StoreStoreEliminationPhase
- ControlFlowOptimizationPhase
- MemoryOptimizationPhase (the memory operations are lowered during effect-control-linearization instead)
- LateOptimizationPhase
- JumpThreadingPhase

In addition, the InliningPhase has the following reducers disabled (JSNativeContextSpecialization, JSIntrinsicLowering and DeadCodeElimination are retained):

- JSInliningHeuristic
- CheckpointElimination
- CommonOperatorReducer
- JSCallReducer
- JSContextSpecialization

And finally, the time spent in the following phases is attributed to a different RuntimeCallStats bucket (and so not counted as part of the optimization time) on the assumption that they could be eliminated or made significantly cheaper by developing a new backend for a TurboProp-like compiler (they contribute more than 50% of the remaining optimization time spent in TurboProp):

- ComputeScheduledGraph
- InstructionSelectionPhase
- RegisterAllocationPhase

The ComputeScheduledGraph is actually the second schedule of the TurboFan graph (the first is done just before EffectControlLinearlization and is thrown away). There has been a previous attempt to avoid this second schedule, and much of the infrastructure necessary to avoid it (e.g., rewriting EffectControlLinearization to use the GraphAssembler) has already landed in V8. It would be easier to finish this work for a TurboProp-like pipeline since the later passes after EffectControlLinearization are almost all removed and so don't need to be ported to GraphAssembler.

As with TurboFan-Lite, TurboProp is not a mid-tier (we don't later tier-up to full TurboFan), but it does give a sense of the difference in performance that might be seen for a more lightweight optimizing compiler.

The code changes made to the pipeline for TurboProp can be seen here.

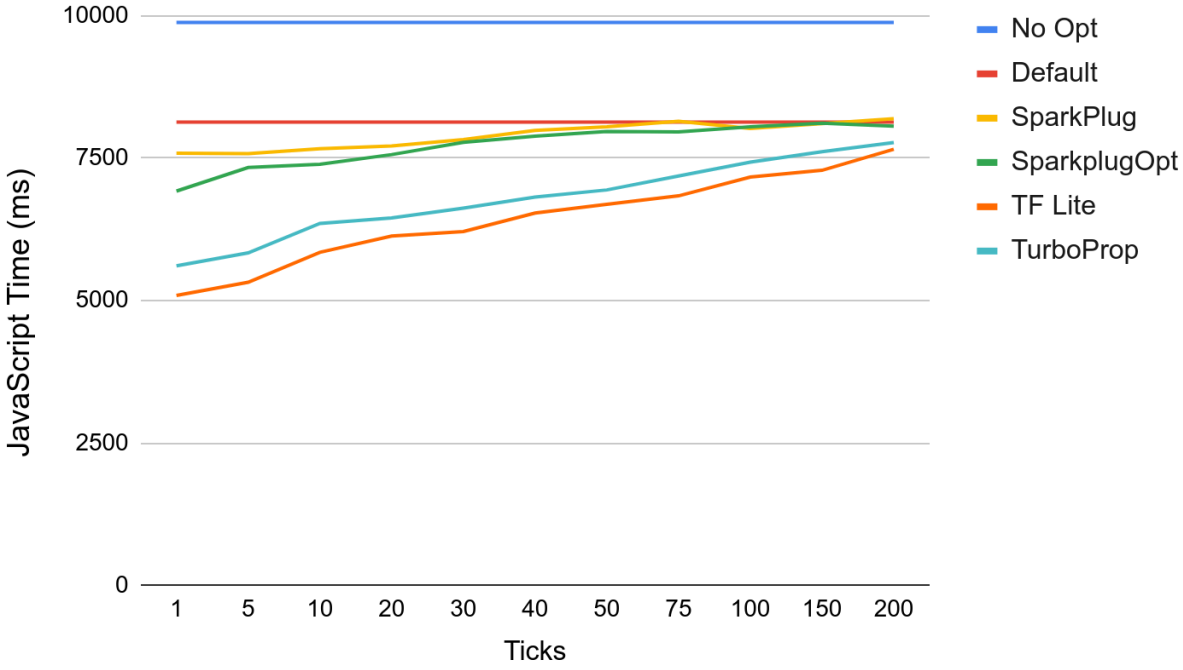
Results

All results were measured on a low-end Nokia 1 device. Each benchmark configuration was run 10 times and the average taken. Full traces from all the results are available at: https://rmcilroy.users.x20web.corp.google.com/www/mid-tier-investigation/

v8.browsing_mobile

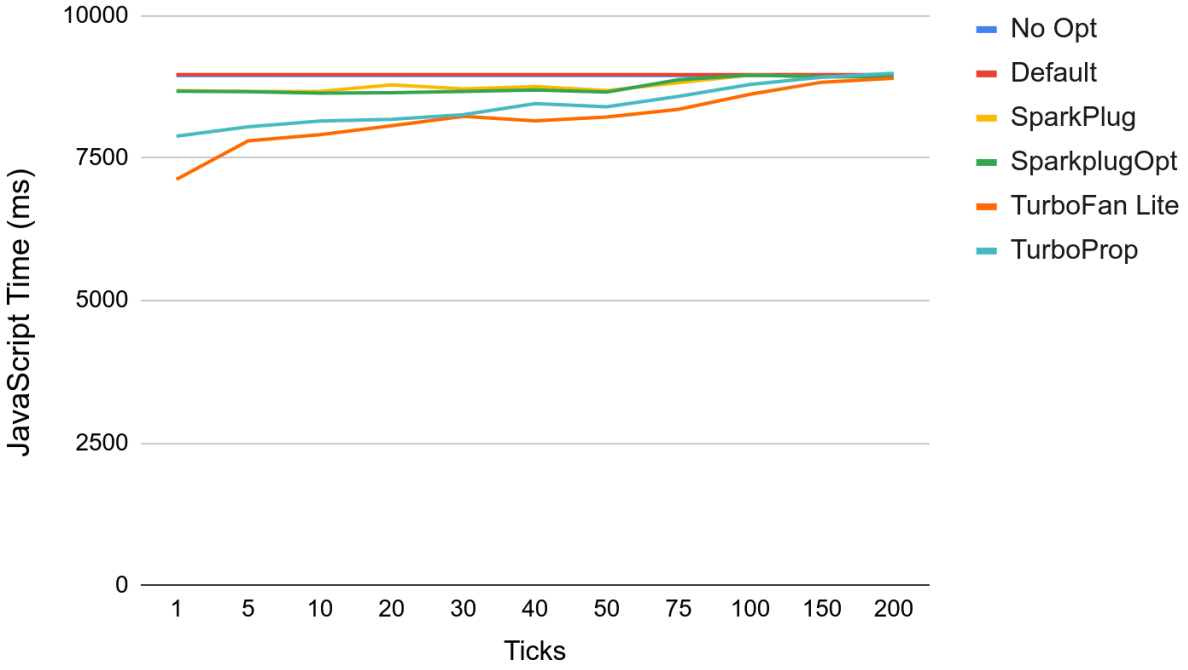
We concentrated on 5 browsing stories from v8.browsing_mobile which had quite different execution characteristics: Google Maps, YouTube, Amazon, Washington Post and Twitter. This section will show more detailed graphs for Maps and Amazon and summaries for the other pages, but all graphs are available in the Appendix.

We ran the benchmarks with different tick-thresholds before tiering-up to the given prototype mid-tier compiler to evaluate the potential JavaScript execution opportunities. Looking at the time spent in the JavaScript RCS bucket for Maps we see that a mid-tier could provide substantial savings (lower is better):



In general the Maps page performs well with optimization. It already gets a 17.7% reduction in JavaScript execution time with TurboFan (No Opt compared to Default). At the most aggressive tick-thresholds, TurboFan-Lite can provide an additional 37.4% reduction, with TurboProp not far off at 31% reduction (both compared to Default). It is clear that most of this benefit is coming from speculative optimization when we compare SparkPlug (6.7% reduction) and SparkPlugOpt (14.9% reduction).

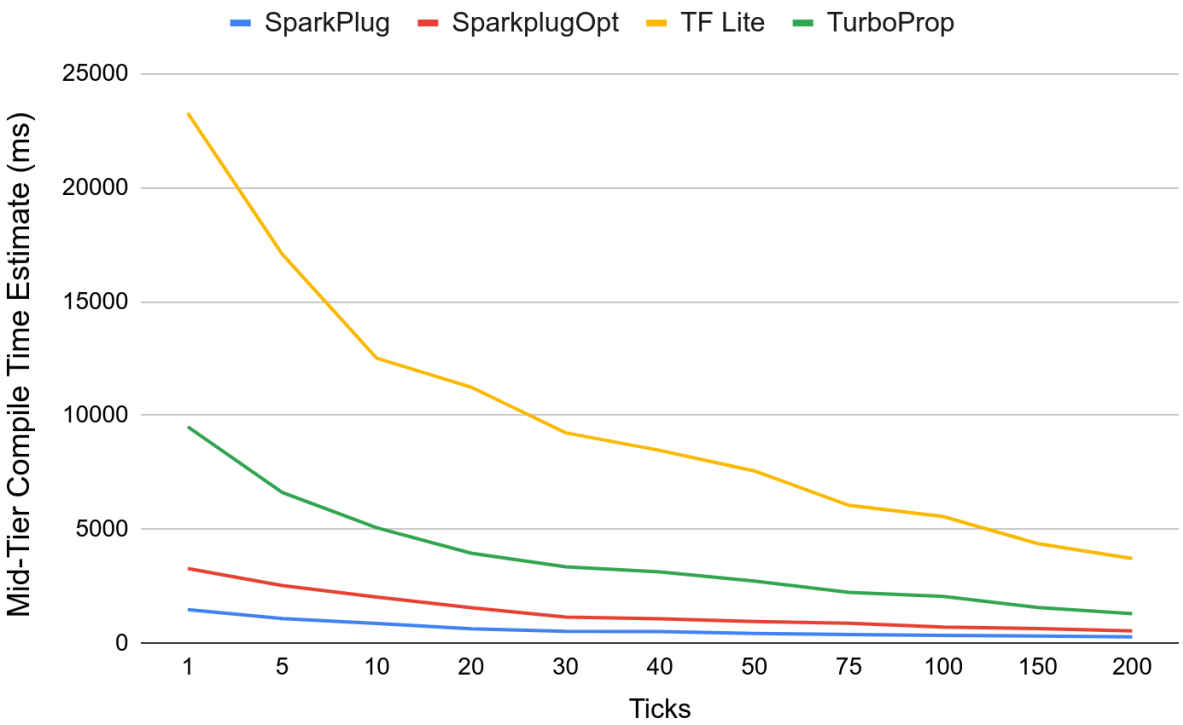
Looking at Amazon's JavaScript execution, we see a slightly different behaviour:



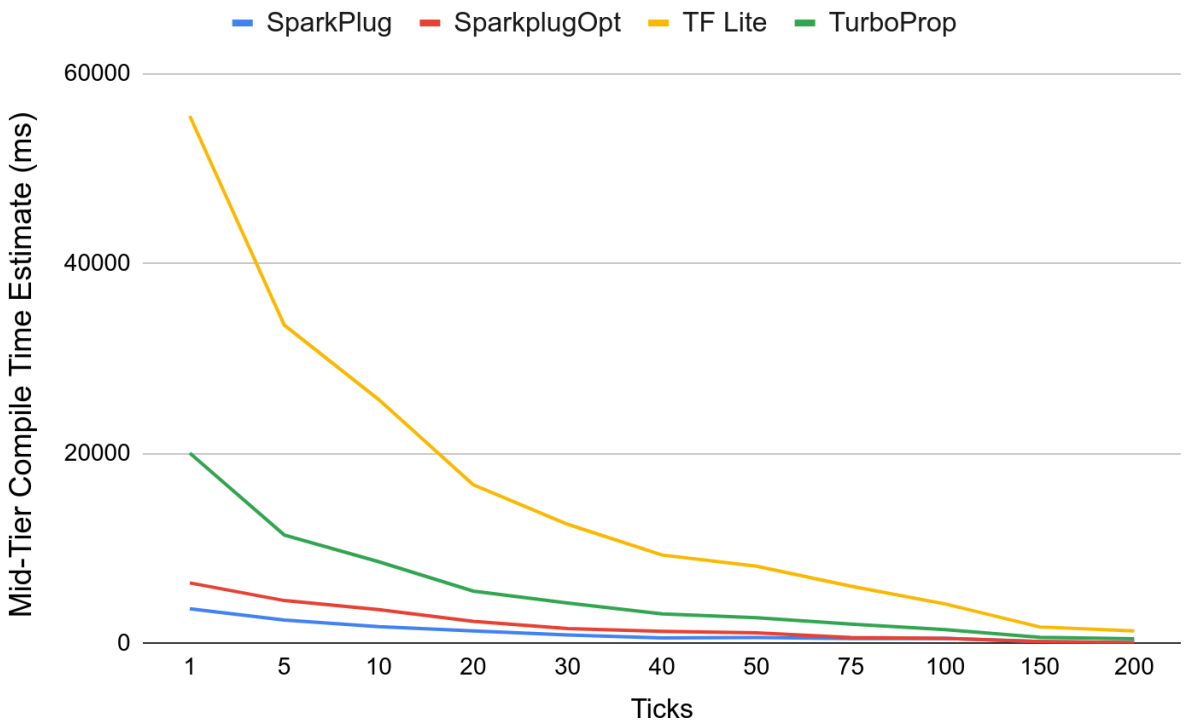
Amazon currently doesn't see any improvement from enabling TurboFan (No Opt compared to Default). However at the most aggressive tick thresholds, a mid-tier could provide some benefit, with again TurboFan-Lite and TurboProp (20% reduction and 12% reduction respectively at most aggressive tick-threshold) doing better than SparkPlug and SparkPlugOpt (3.1% and 3.2% reduction respectively). Notably, the optimizations in SparkPlugOpt don't seem to make much impact compared to SparkPlug, while those in TurboFan provide a decent improvement.

Of course, we also need to consider the time taken to compile the mid-tier code. For these graphs we looked at the Optimization RCS bucket for TurboFan-Lite and TurboProp, but for SparkPlug we needed to do some estimation since it is currently using CSA and is not a single-pass compiler. As such we scaled down the SparkPlug and SparkPlugOpt compile times to be 1/5th of what was reported - we estimated this scaling factor based on the amount of time spent by SparkPlug reading the bytecode and doing the final code-generation, thus eliminating the middle passes of CSA that wouldn't be part of a production SparkPlug-like mid-tier compiler.

For Maps we see the following time spent compiling by each mid-tier compiler:



As expected, at aggressive tick-thresholds the amount of time spent in compilation can be very high, particularly TurboFan-Lite which spends 5x more time compiling mid-tier code than executing JavaScript. In general, compared to TurboFan-Lite, TurboProp takes around 1/3rd of the compile time, SparkPlugOpt reduces this to 1/7th and SparkPlug is faster still at 1/16th of the compile time.



Looking at Amazon the overheads are even larger, with TurboFan-Lite spending more than 7x the time to compile mid-tier code compared to executing JavaScript. The relative time differences between TurboFan-Lite, TurboProp, SparkPlugOpt and SparkPlug are similar.

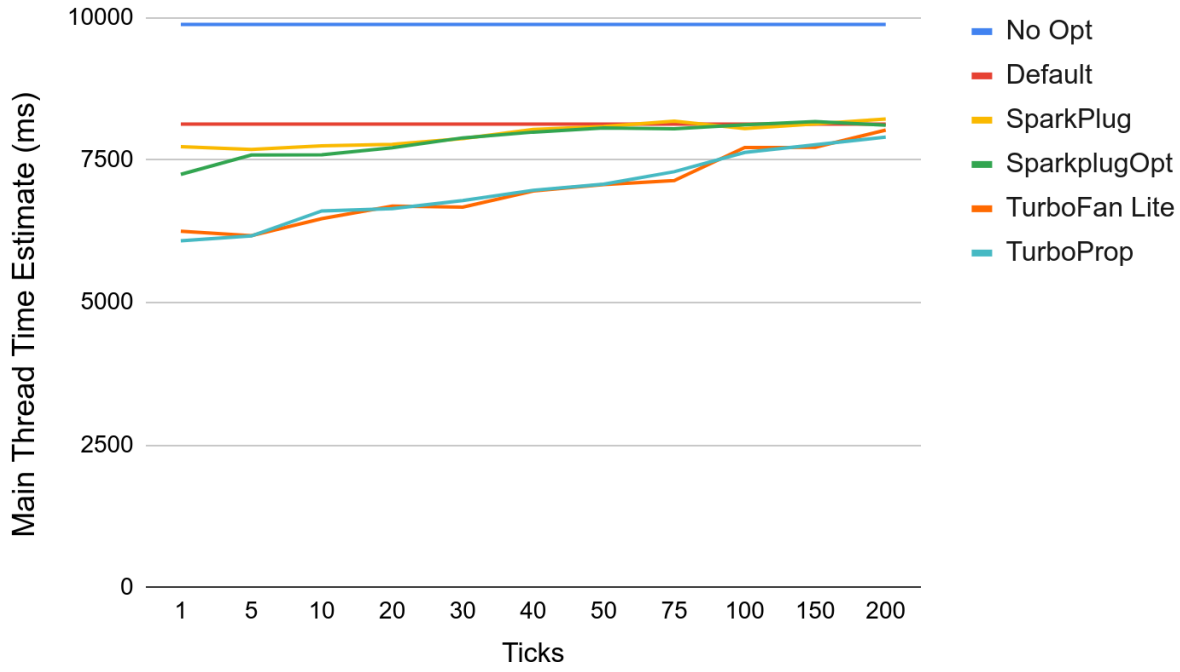
It should be noted that all but SparkPlug will deoptimize code if speculative optimisations are invalidated. For aggressive tick thresholds the depot rate seen was quite high on all of these pages (perhaps ⅛ to ¼ of the number of optimizations), so if we add back tick resetting on IC updates and tune tier-up better, this mid-tier compilation might be reduced by a similar magnitude.

Much of this compilation time will be on a background worker thread, therefore even a large compilation time could result in a better user experience by freeing the main thread. However too long will be detrimental for multiple reasons:

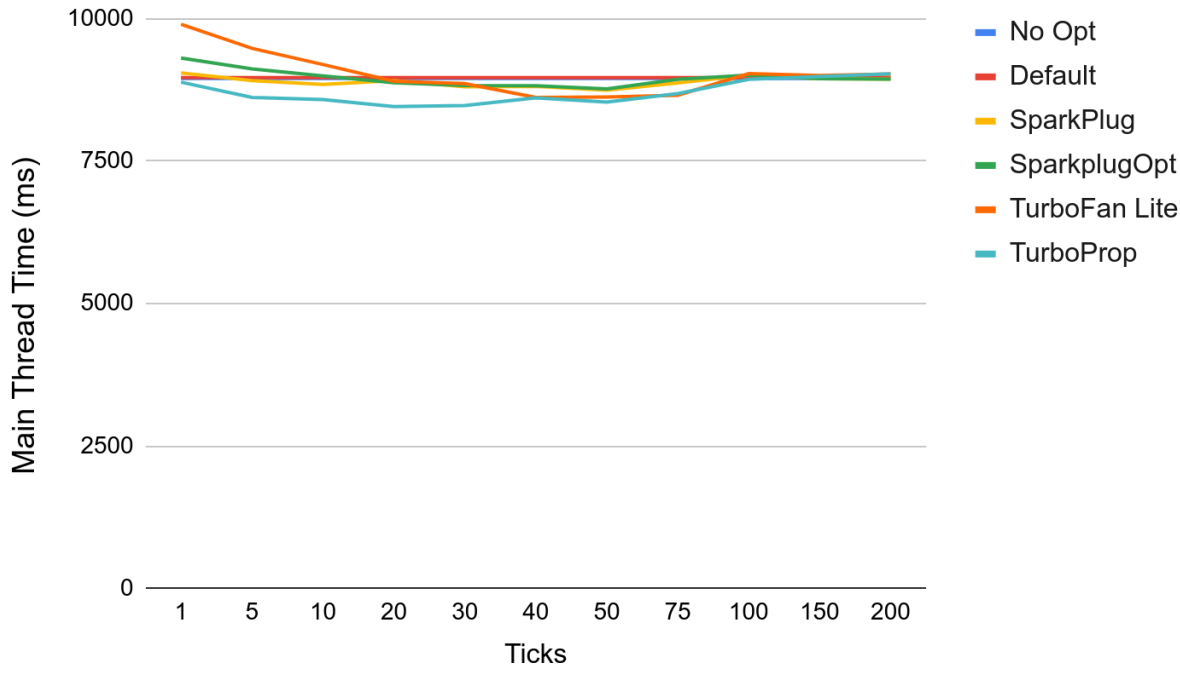
- It makes use of CPU cores which might be used for other work, e.g., parallel garbage collection or script streaming.
- It wastes power, particularly on mobile devices.
- It is likely to impact main-thread / JavaScript execution time since we will continue to execute code in Ignition until the mid-tier compiler has finished compilation.
- Some proportion of the compilation will likely always remain on the main thread (e.g., reading the bytecode arrays and feedback vectors).

As a rough approximation of time spent on the main thread, I estimated that around 5% of TurboFan-Lite and TurboProp might be spent on the main thread, and 10% of SparkPlug and

SparkPlugOpt would be spent on the main thread. With these estimates, the time spent on the main thread looks as follows for Maps:



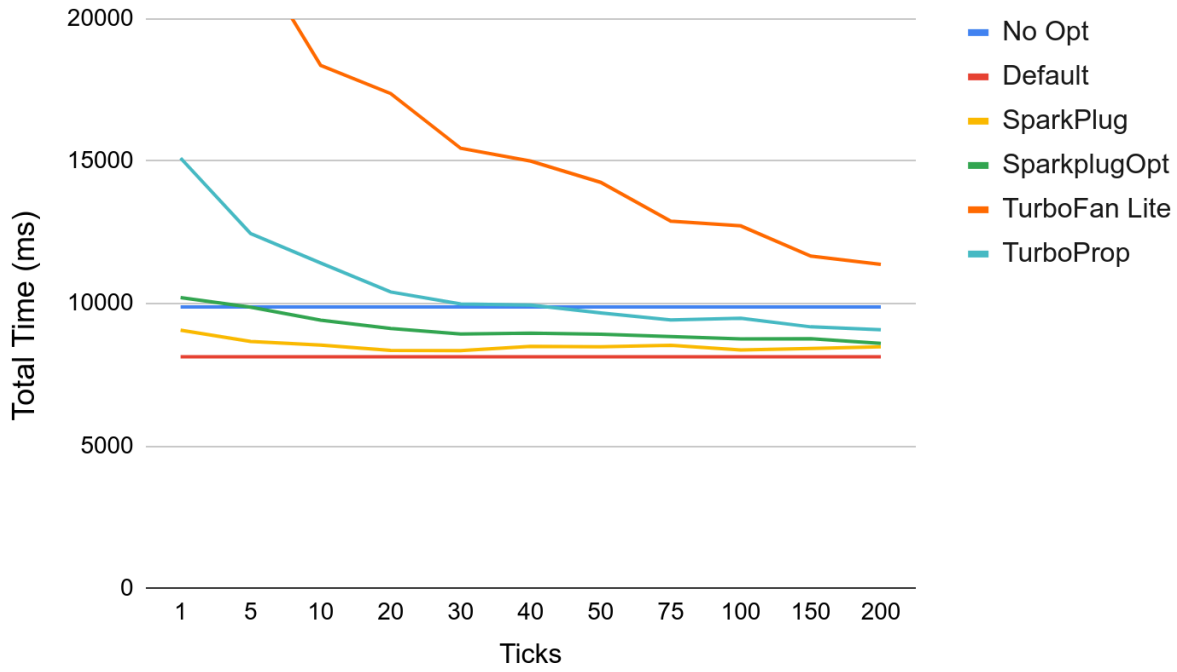
And for Amazon:



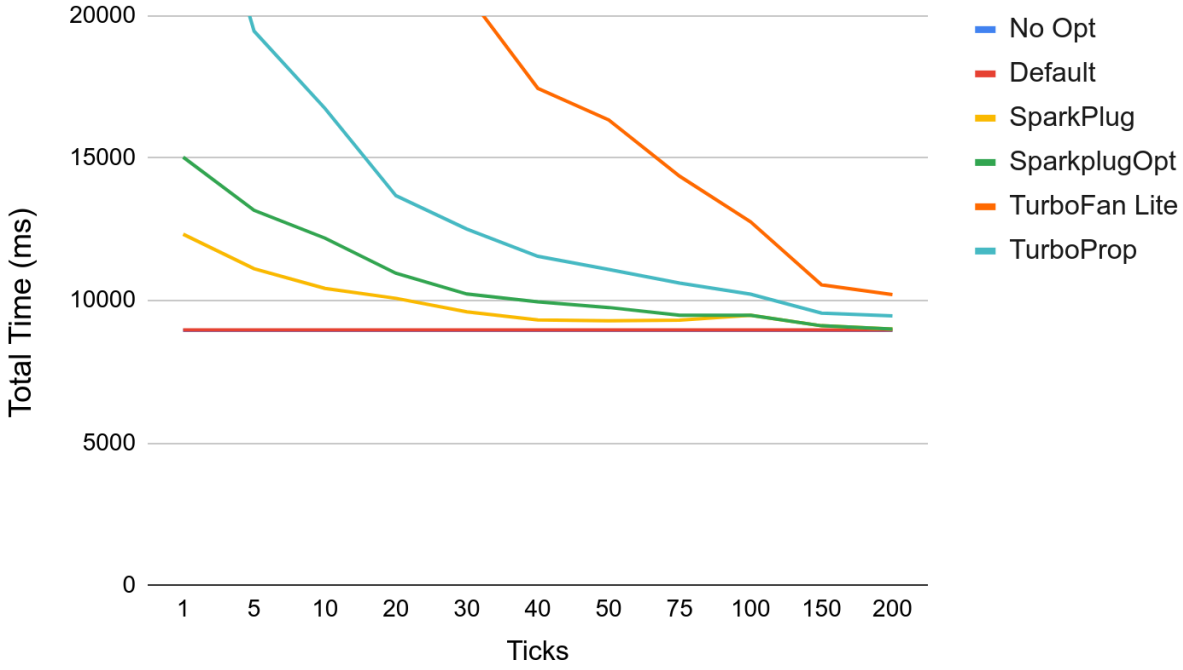
Here TurboProp is on-par with TurboFan-Lite in Maps, and generally better for Amazon. The slope of the graphs also changes showing that, even when looking just at the main thread, being too aggressive tiering up could cause a regression in main-thread time. Overall a tick-threshold of between 30-50 seems the best suited to get the most benefits (similar to what we saw in the Analysis section). For a tick-threshold of 30 ticks, we see the following reductions in main-thread time:

	SparkPlug	SparkPlugOpt	TurboFan-Lite	TurboProp
Maps	3.1%	3.0%	18.0%	16.5%
Amazon	1.8%	1.6%	1.17%	5.4%

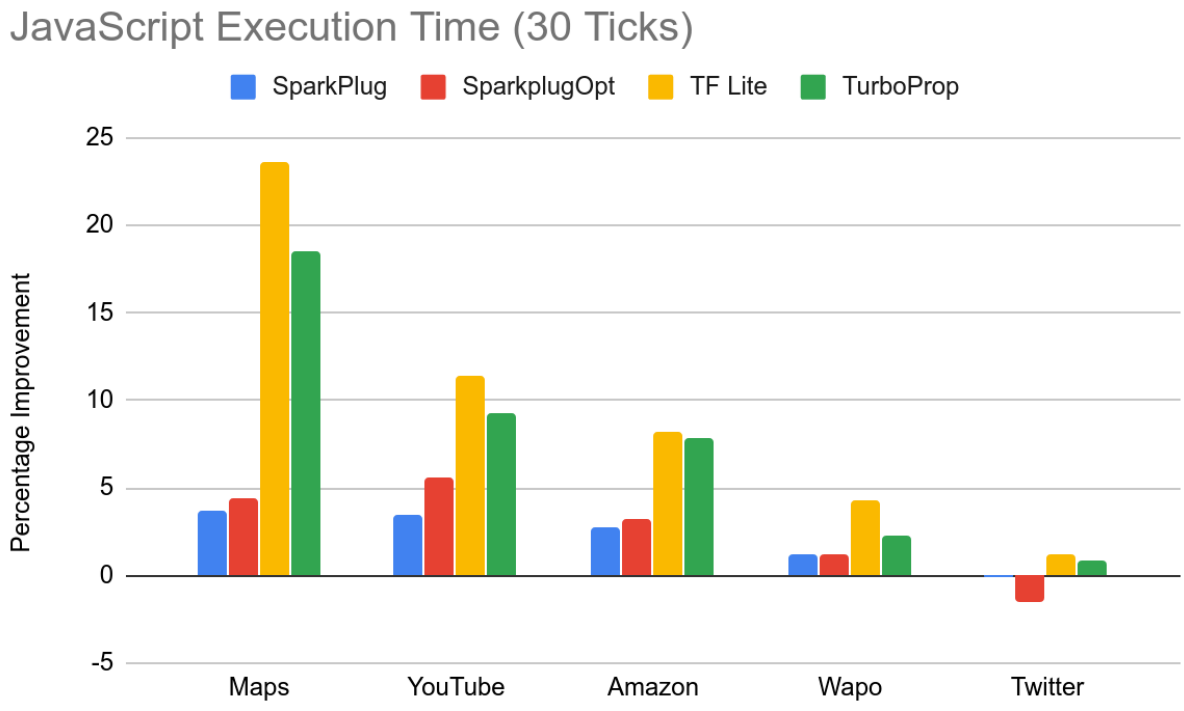
Looking at total CPU time across all threads for both JavaScript execution and mid-tier compilation we see the following, for Maps:

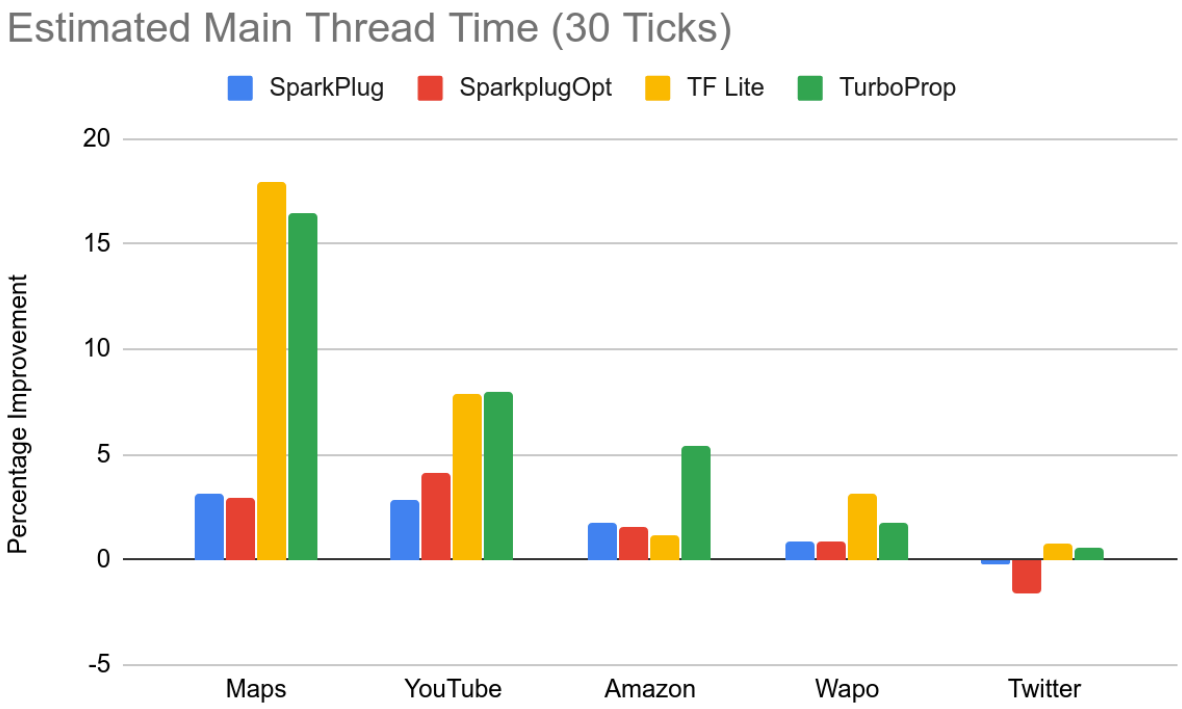


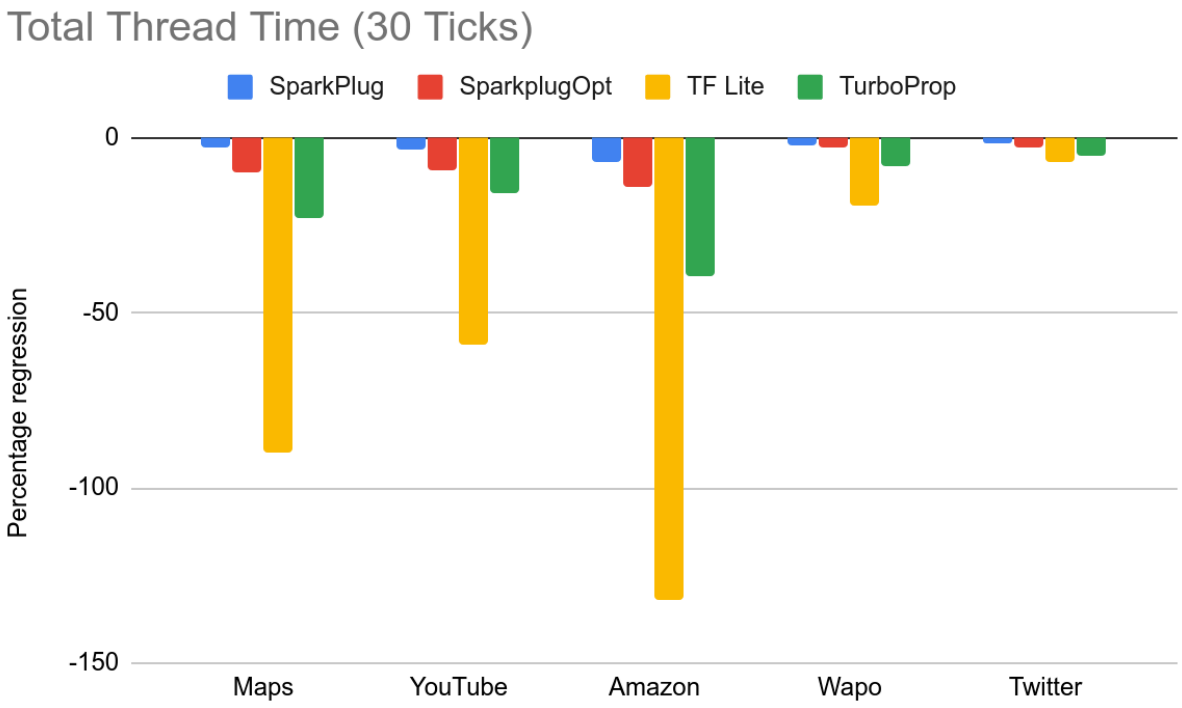
and for Amazon:



Looking across all five pages, for a tick-threshold of 30 ticks, we see the following:







There are also some pages like Washington Post and Twitter which don't see much improvement from a mid-tier compiler, however they also don't show regression in total CPU time either, and so are simply don't have hot enough functions to tier-up to the mid-tier.

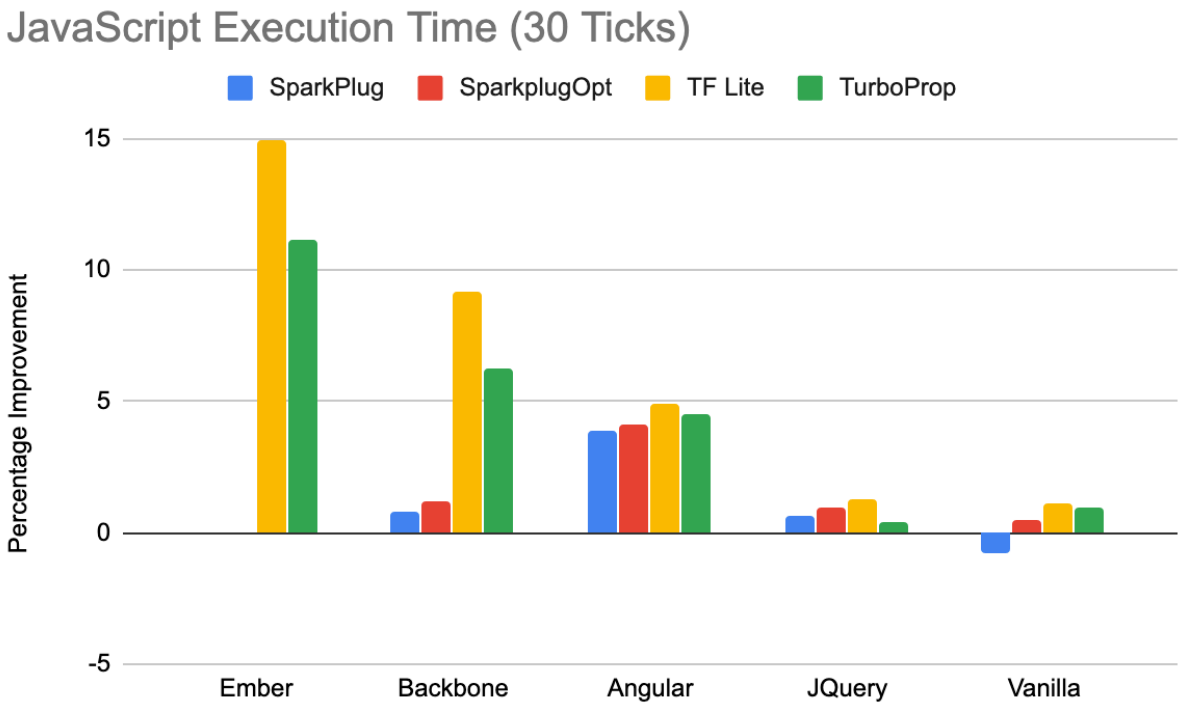
For those that do show improvement, TurboProp and TurboFan-Lite have similar percentage improvements in JavaScript execution time and main thread (between 5-15%). However, TurboProp has substantially less total CPU usage (15-40% regression) compared to TurboFan-Lite (60-130% regression).

Looking at SparkPlug and SparkPlugOpt we see that the improvements are much more limited, typically between 1.5-4% reduction in main-thread time. The regressions in total CPU time are also more limited at between 2-7% for SparkPlug and 9-14% for SparkPlugOpt.

Speedometer

We also ran Speedometer 1.0[1] with the prototype mid-tier compilers. One issue of these runs is that Ember crashed on SparkPlug and SparkPlugOpt, so there are no results included for it.

Looking at JavaScript execution time for Speedometer for a tick-threshold of 30 ticks shows good improvements for three of the five tested benchmarks (JQuery and Vanilla don't show much improvement).



Ember, Angular and Backbone see the most significant improvements, between 5-15% for TurboFan-Lite and 4-12% for TurboProp. There is a bit more of a gap between TurboProp and

[1] SparkPlug doesn't support all the ES6 features needed to run Speedometer 2.0

TurboFan-Lite than there was on the web browsing stories, which suggests that Speedometer benefits more from some of the optimization phases which were removed for TurboProp.

Looking at SparkPlug and SparkPlugOpt, the savings for Angular aren't far off those of TurboFan-Lite (~4%), however the Backbone savings are much lower (0.8-1.2%) and it's unknown what savings might be achieved for Ember.

Conclusion

This investigation shows that there is an opportunity to improve JavaScript performance for certain web pages and benchmarks by introducing a middle tier compiler.

A baseline compiler like Sparkplug provides some potential benefits, with relatively small total CPU time impact. However the savings on JavaScript execution are limited to likely less than 5%.

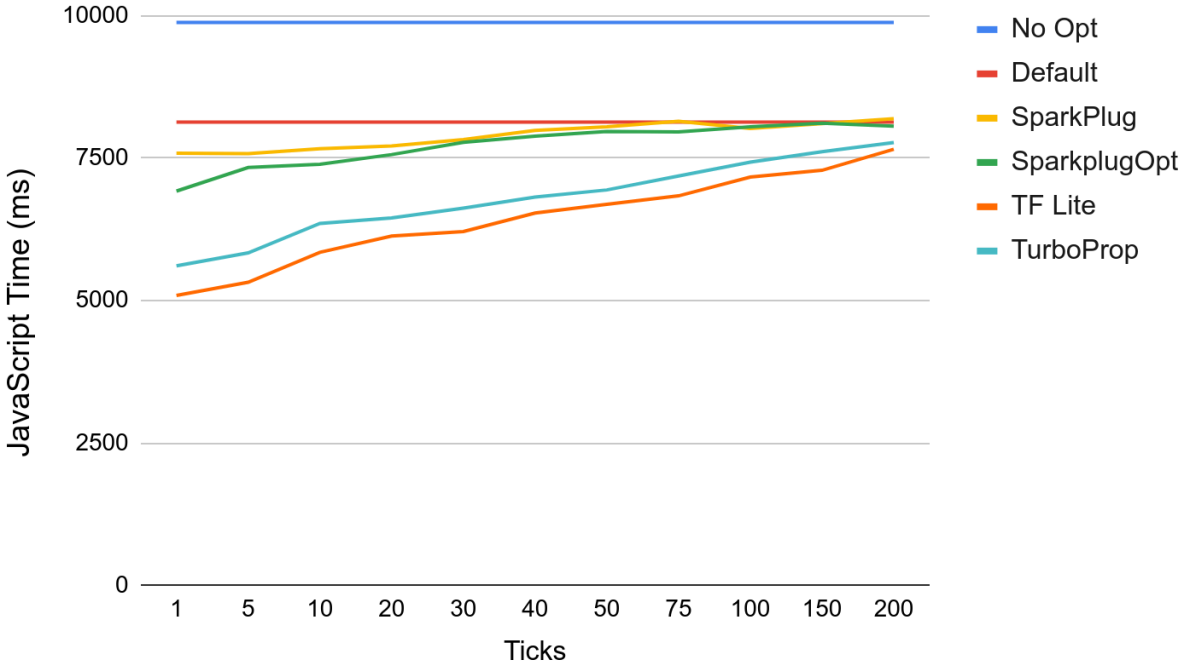
A speculatively optimizing mid-tier compiler provides more benefit on the main-thread, with TurboFan-Lite giving between 5-20% JavaScript execution improvements on some pages, however running the full TurboFan pipeline for all functions that tier-up to the mid-tier tick threshold has a prohibitive increase in total CPU time.

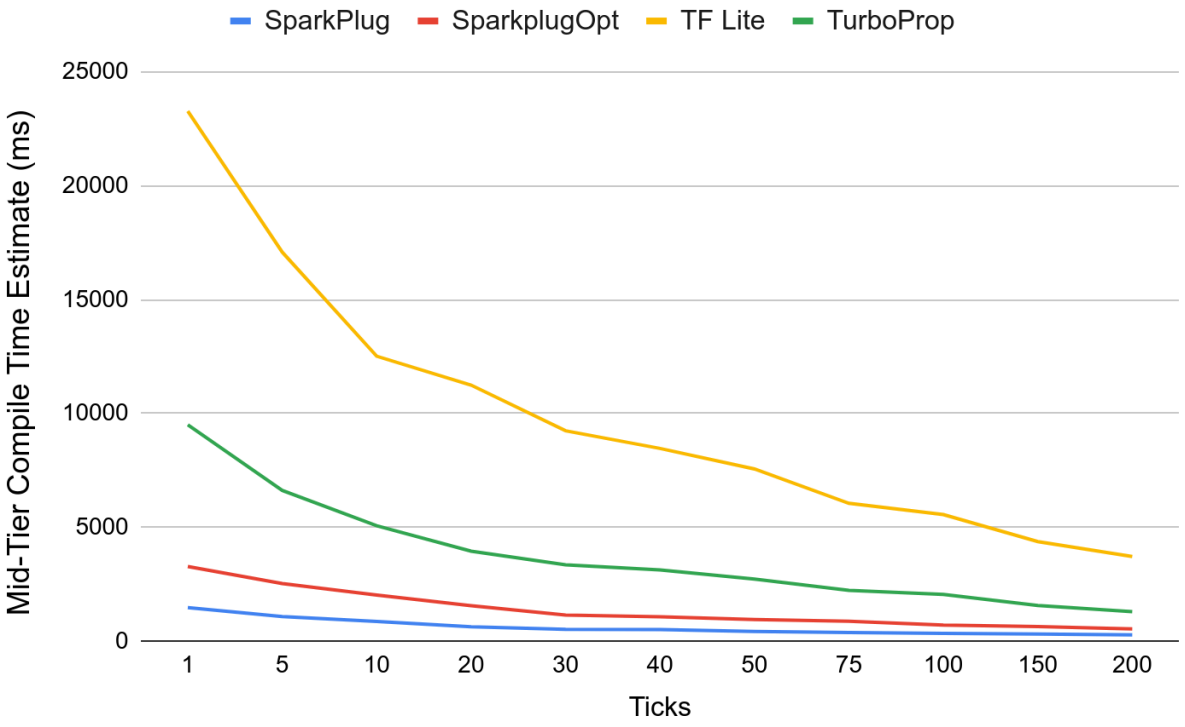
Evaluation of the TurboProp prototype shows that there is potential to achieve most of the JavaScript execution time benefits of a mid-tier TurboFan with significantly less CPU intensive pipeline. This seems like an avenue that is worth pursuing, and a plan to do so is described in this design doc.

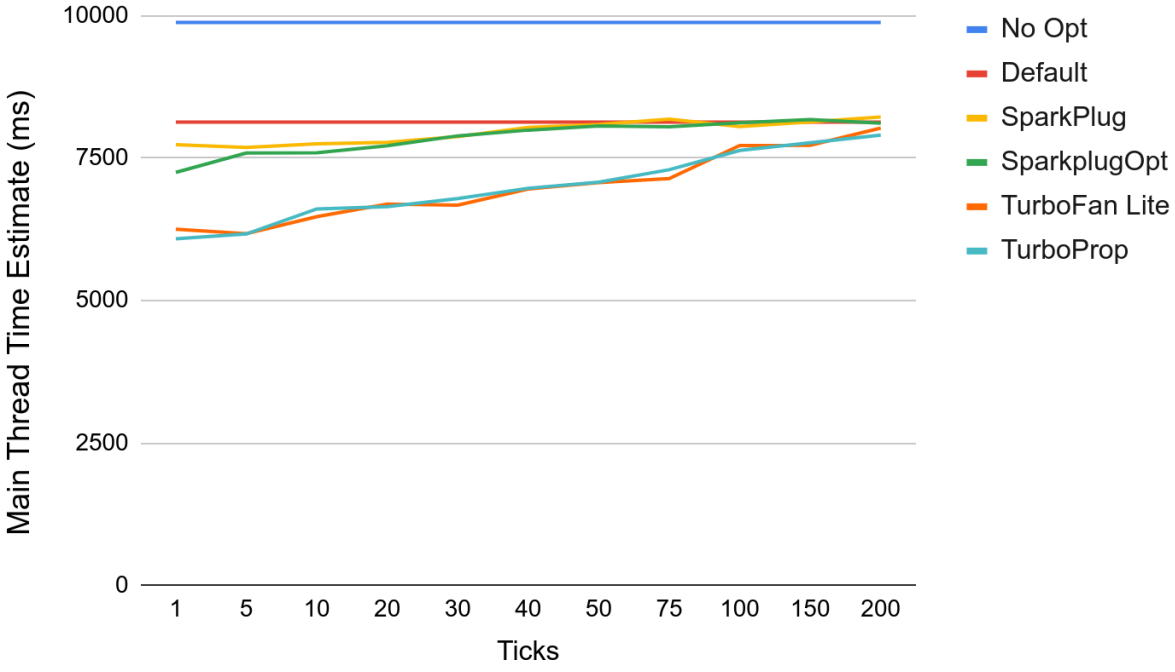
Appendix

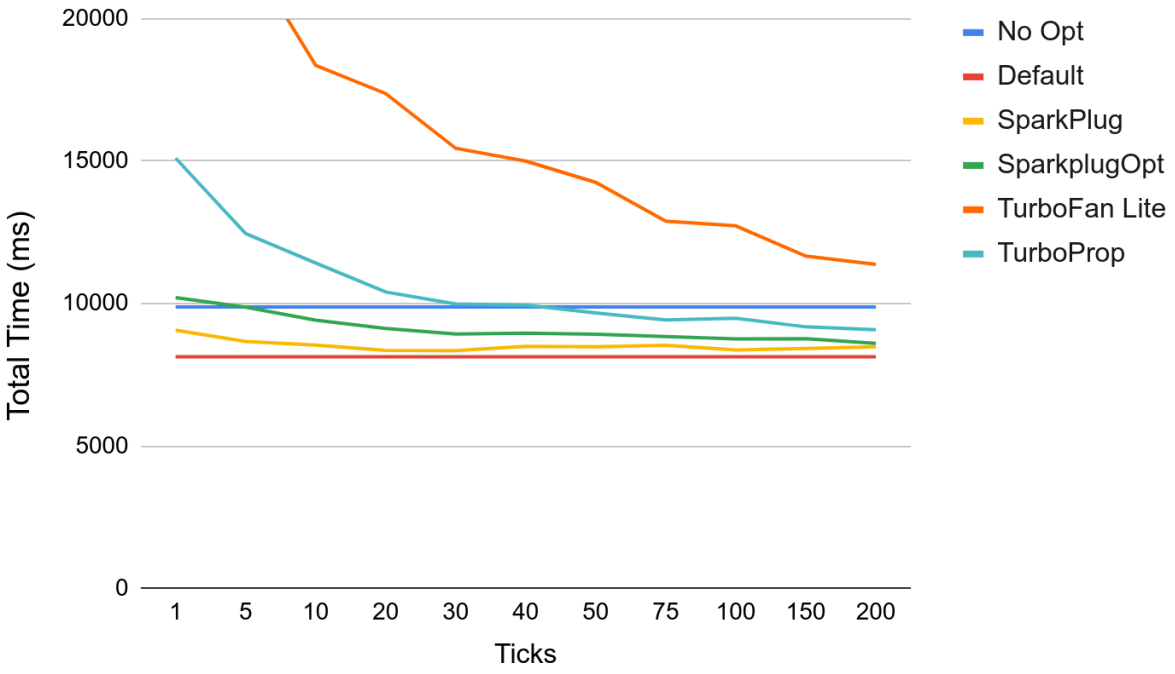
v8.browsing_mobile Graphs

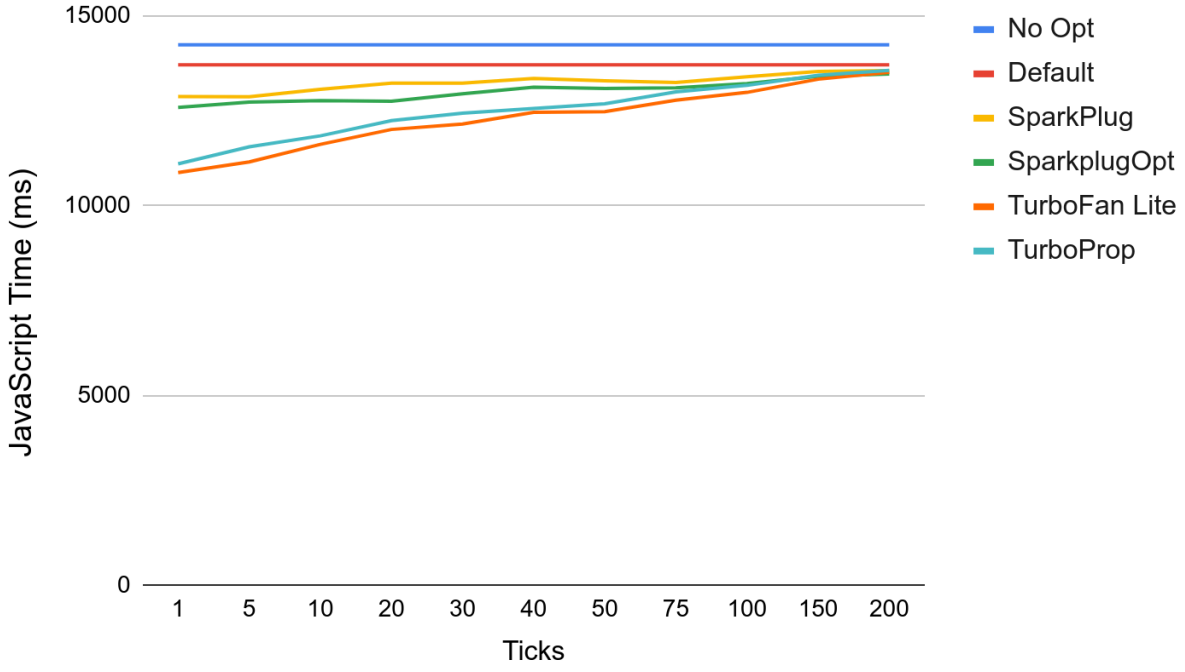
Maps

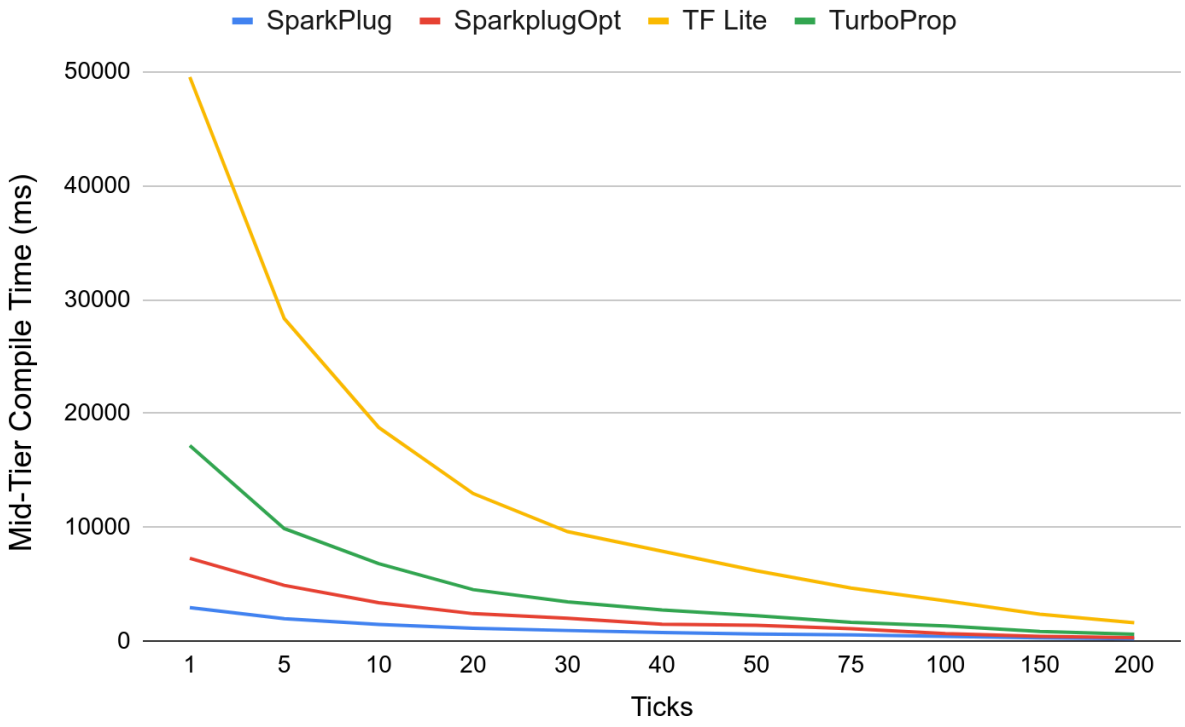

10000
7500
5000
2500
0
JavaScript Time (ms)
1
5
10
20
30
40
50
75
100
150
200
Ticks
No Opt
Default
SparkPlug
SparkplugOpt
TF Lite
TurboProp

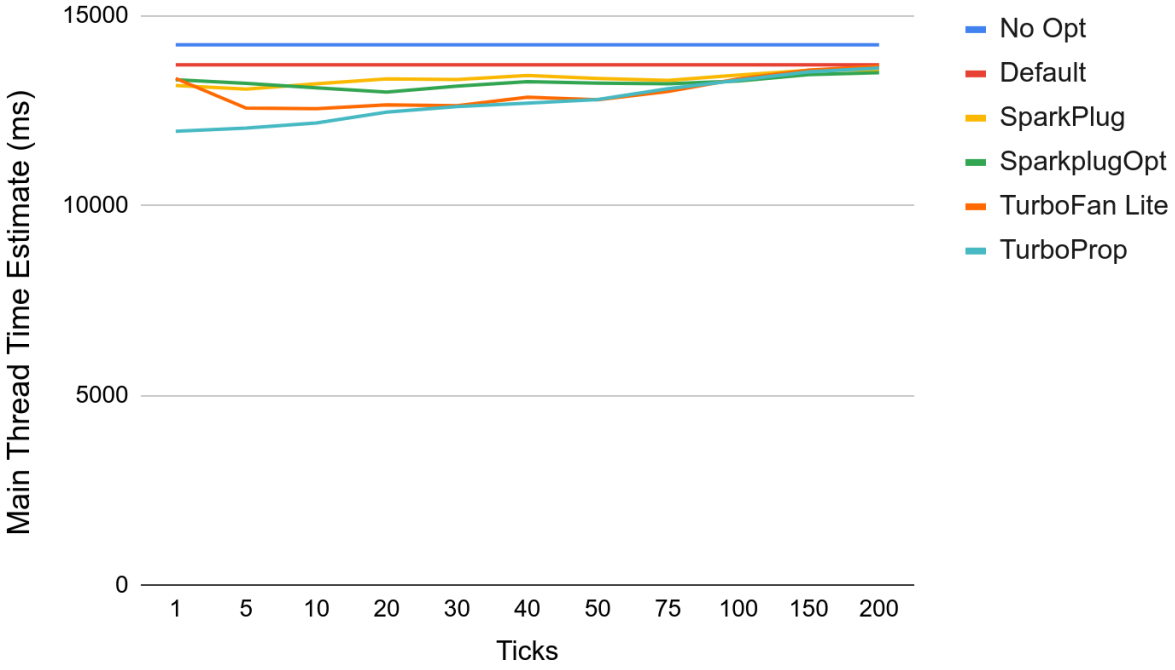

SparkPlug
SparkplugOpt
TF Lite
TurboProp
25000
20000
15000
10000
5000
0
Mid-Tier Compile Time Estimate (ms)
1
5
10
20
30
40
50
75
100
150
200
Ticks

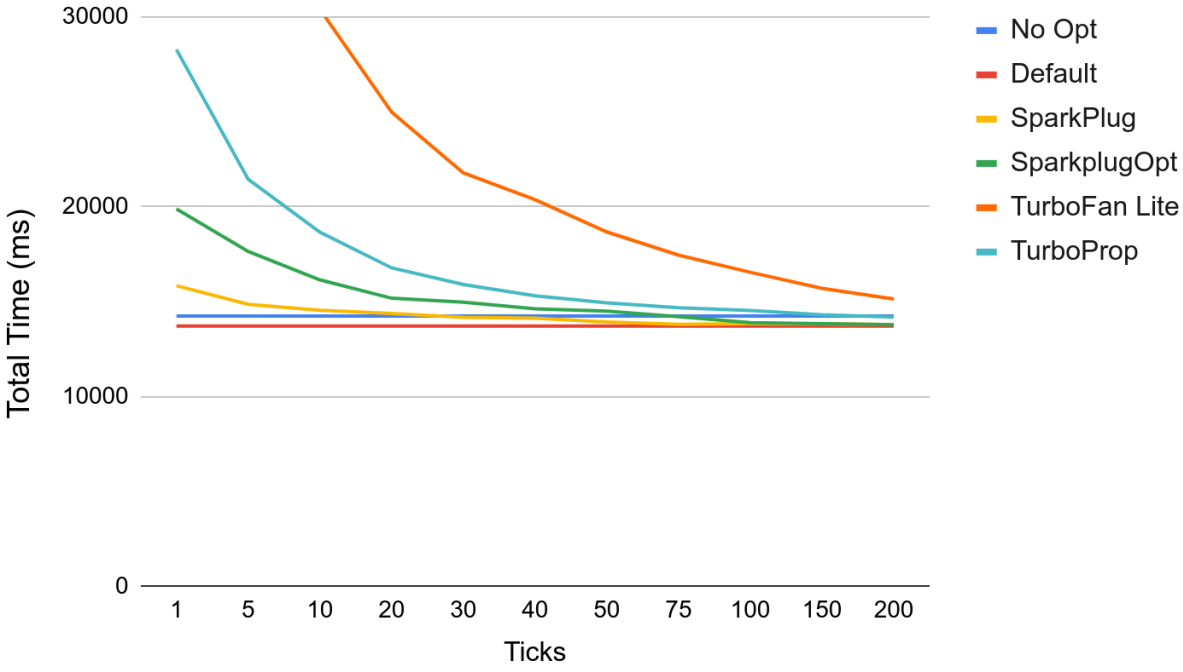

Main Thread Time Estimate (ms)
10000
7500
5000
2500
0
1
5
10
20
30
40
50
75
100
150
200
Ticks
No Opt
Default
SparkPlug
SparkplugOpt
TurboFan Lite
TurboProp


Total Time (ms)
20000
15000
10000
5000
0
1
5
10
20
30
40
50
75
100
150
200
Ticks
No Opt
Default
SparkPlug
SparkplugOpt
TurboFan Lite
TurboProp

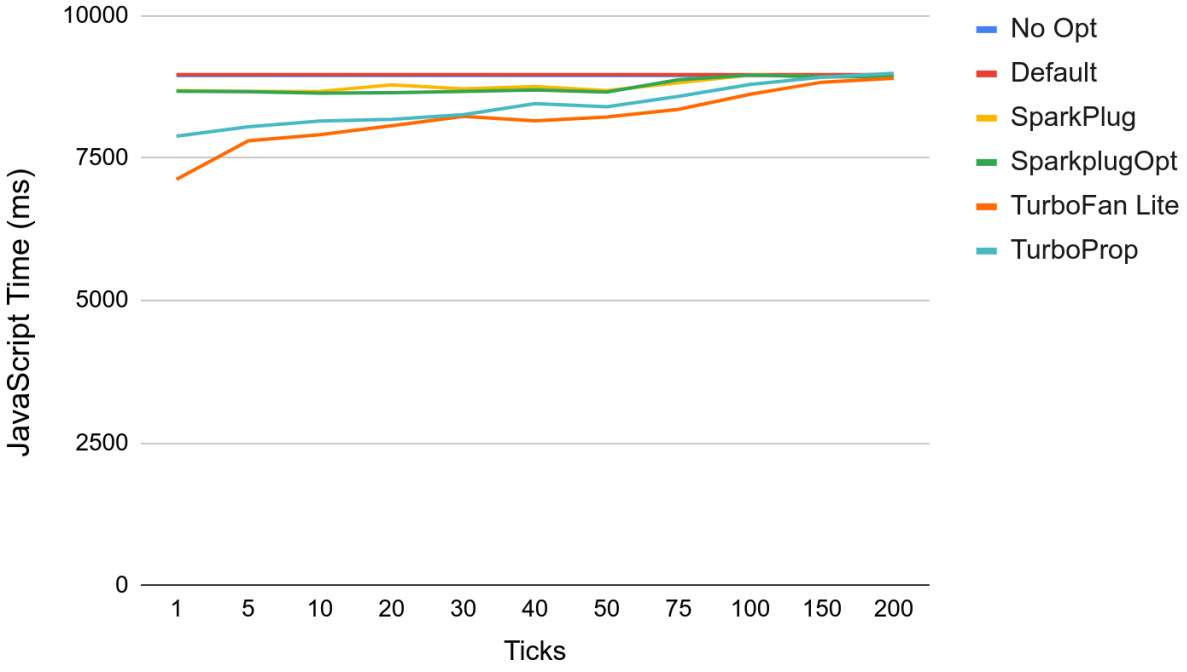

15000
10000
5000
0
JavaScript Time (ms)
1
5
10
20
30
40
50
75
100
150
200
Ticks
No Opt
Default
SparkPlug
SparkplugOpt
TurboFan Lite
TurboProp

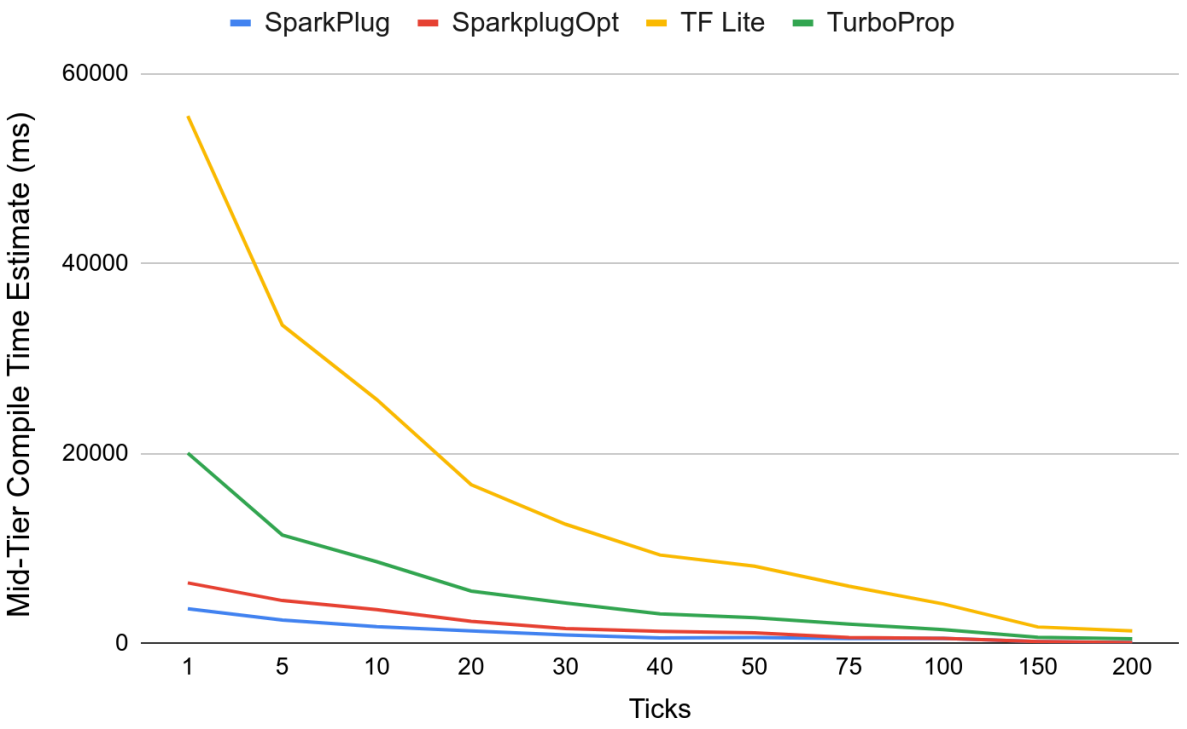

SparkPlug
SparkplugOpt
TF Lite
TurboProp
50000
40000
30000
20000
10000
0
Mid-Tier Compile Time (ms)
1
5
10
20
30
40
50
75
100
150
200
Ticks

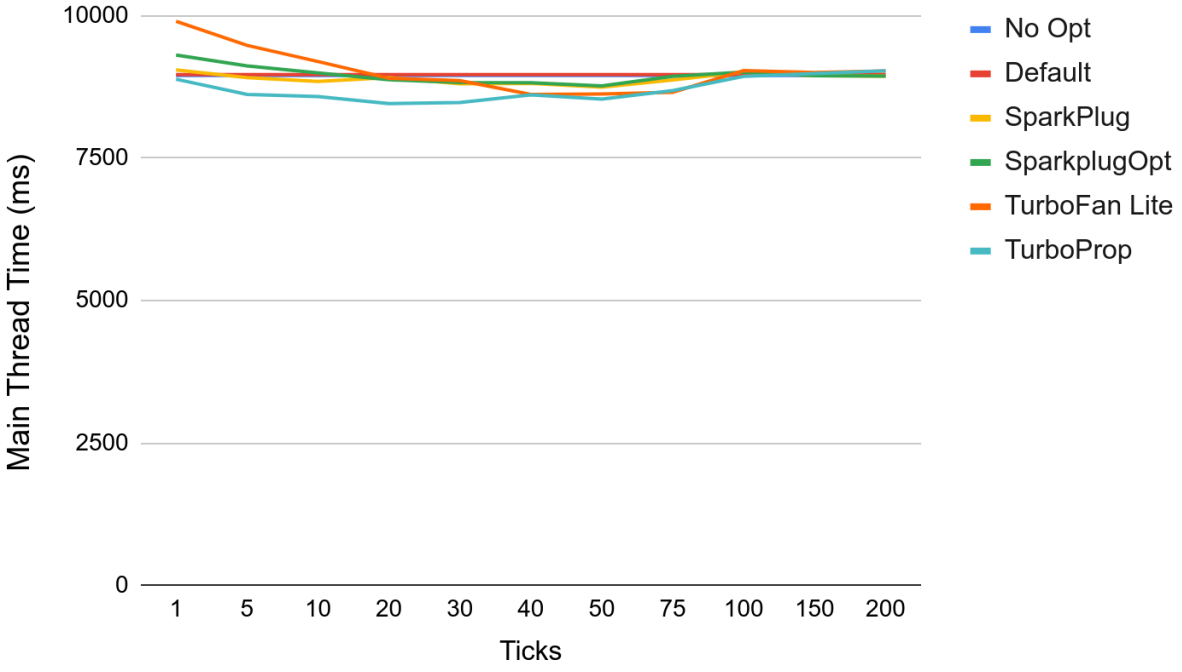

15000
10000
5000
0
Main Thread Time Estimate (ms)
1
5
10
20
30
40
50
75
100
150
200
Ticks
No Opt
Default
SparkPlug
SparkplugOpt
TurboFan Lite
TurboProp

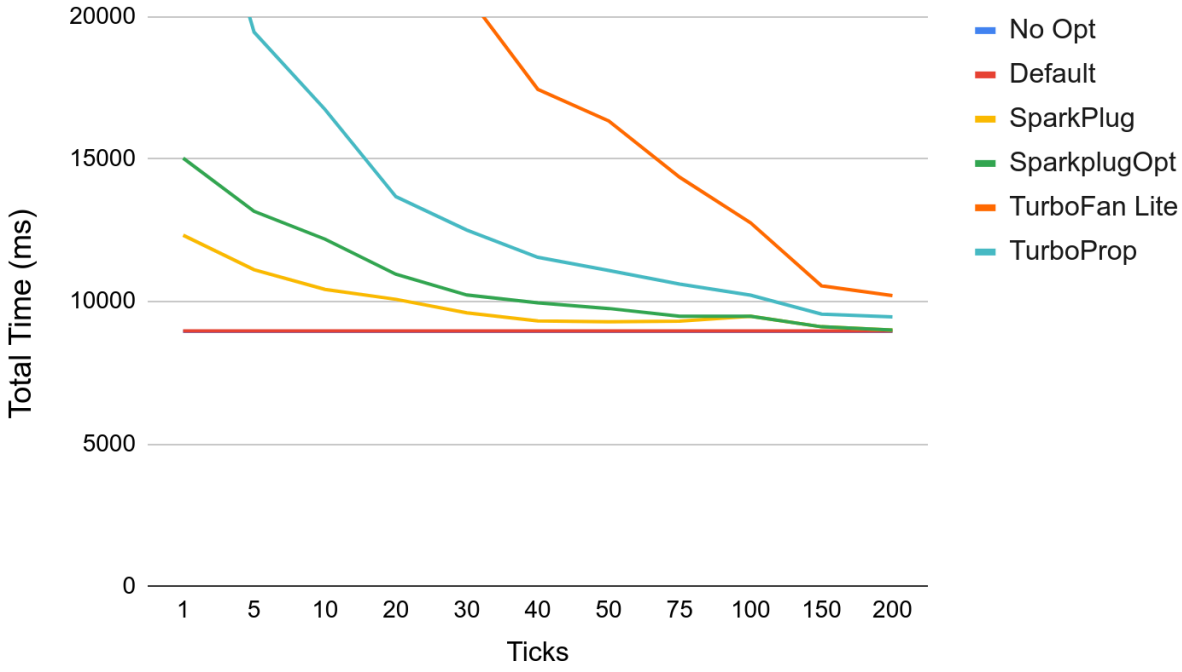

30000
20000
10000
0
Total Time (ms)
1
5
10
20
30
40
50
75
100
150
200
Ticks
No Opt
Default
SparkPlug
SparkplugOpt
TurboFan Lite
TurboProp

Amazon

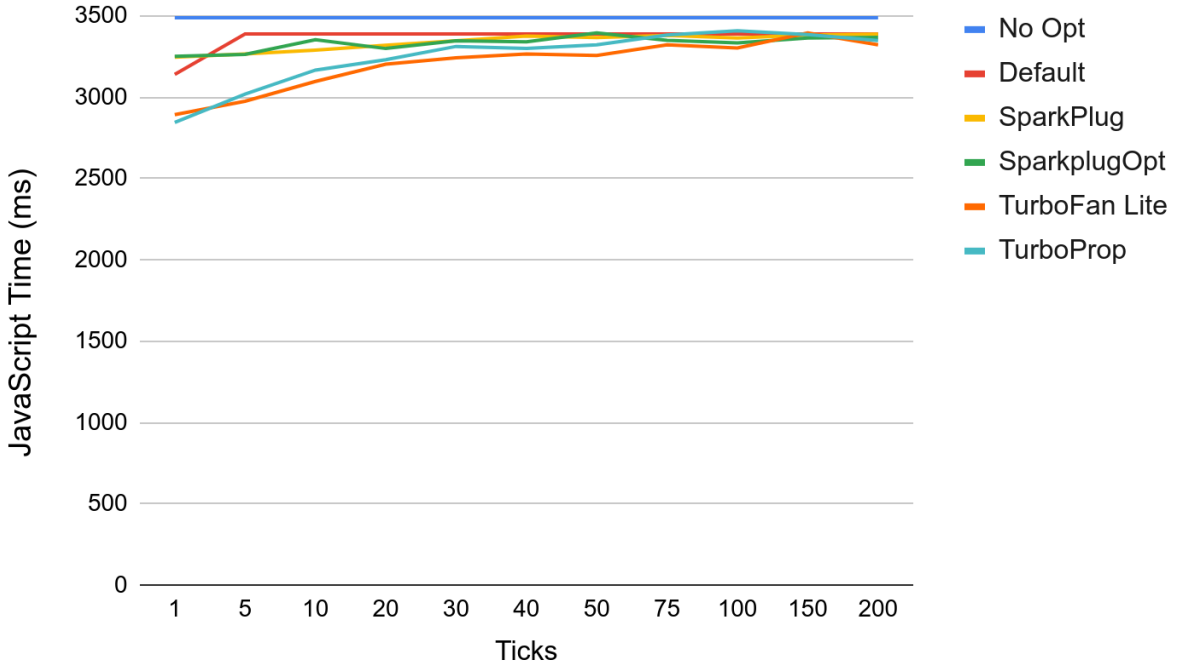

10000
7500
5000
2500
0
JavaScript Time (ms)
1
5
10
20
30
40
50
75
100
150
200
Ticks
No Opt
Default
SparkPlug
SparkplugOpt
TurboFan Lite
TurboProp

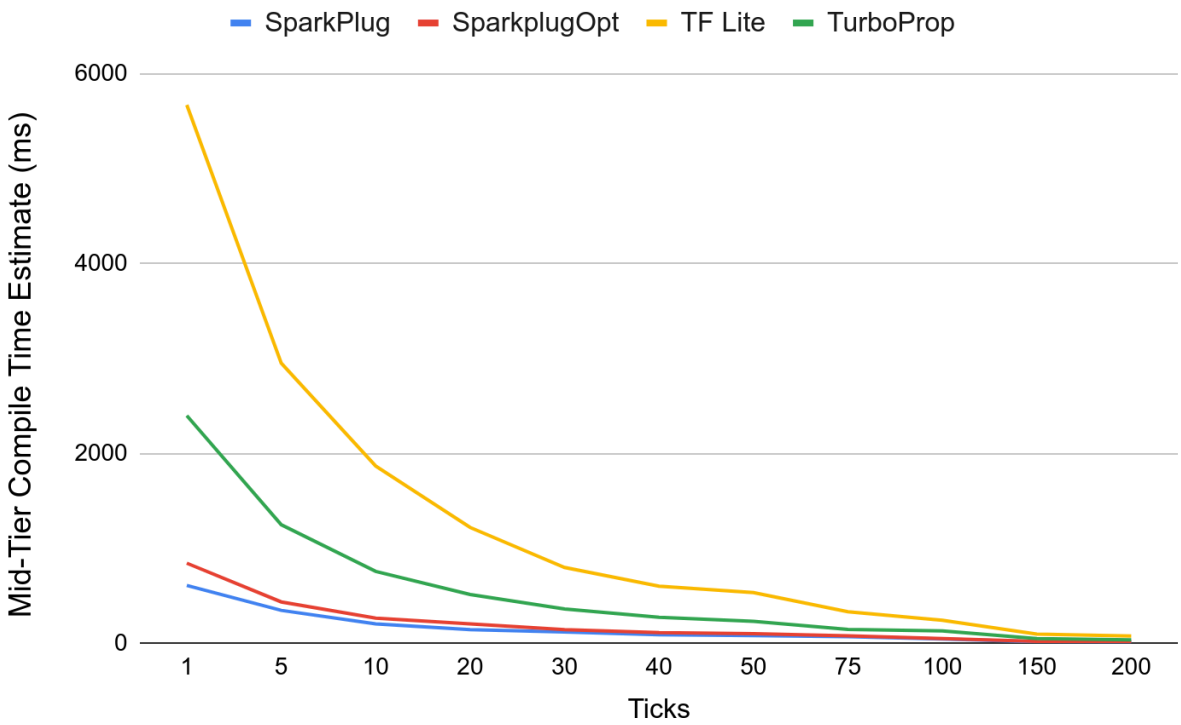

SparkPlug
SparkplugOpt
TF Lite
TurboProp
60000
40000
20000
0
Mid-Tier Compile Time Estimate (ms)
1
5
10
20
30
40
50
75
100
150
200
Ticks

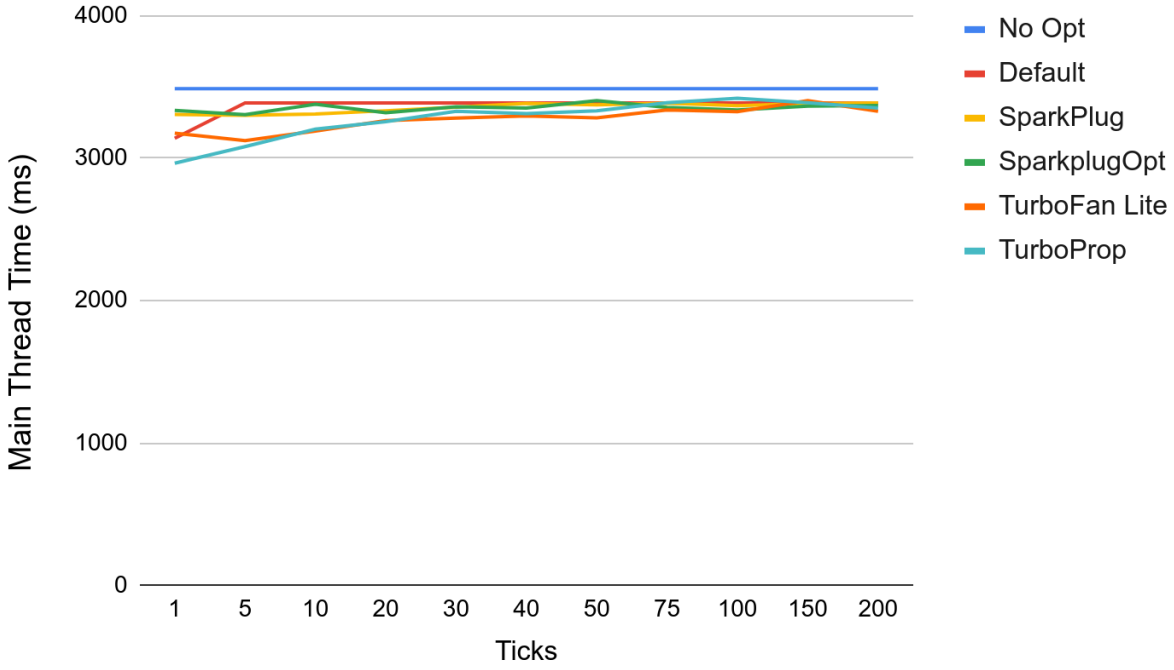


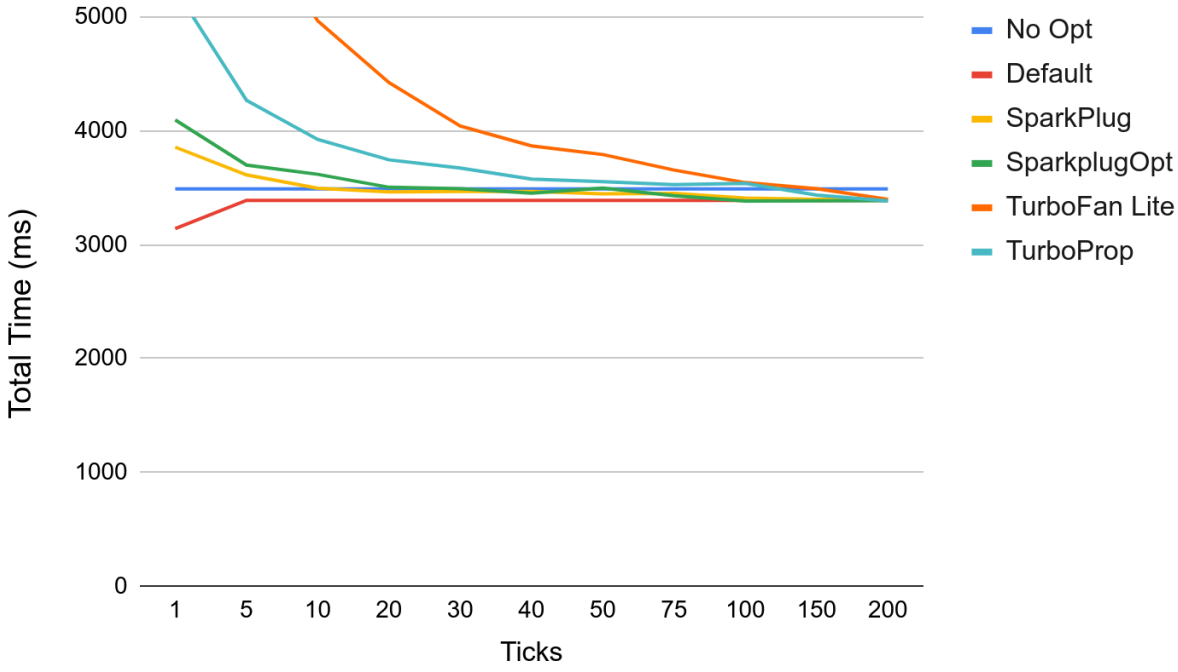


Washington Post

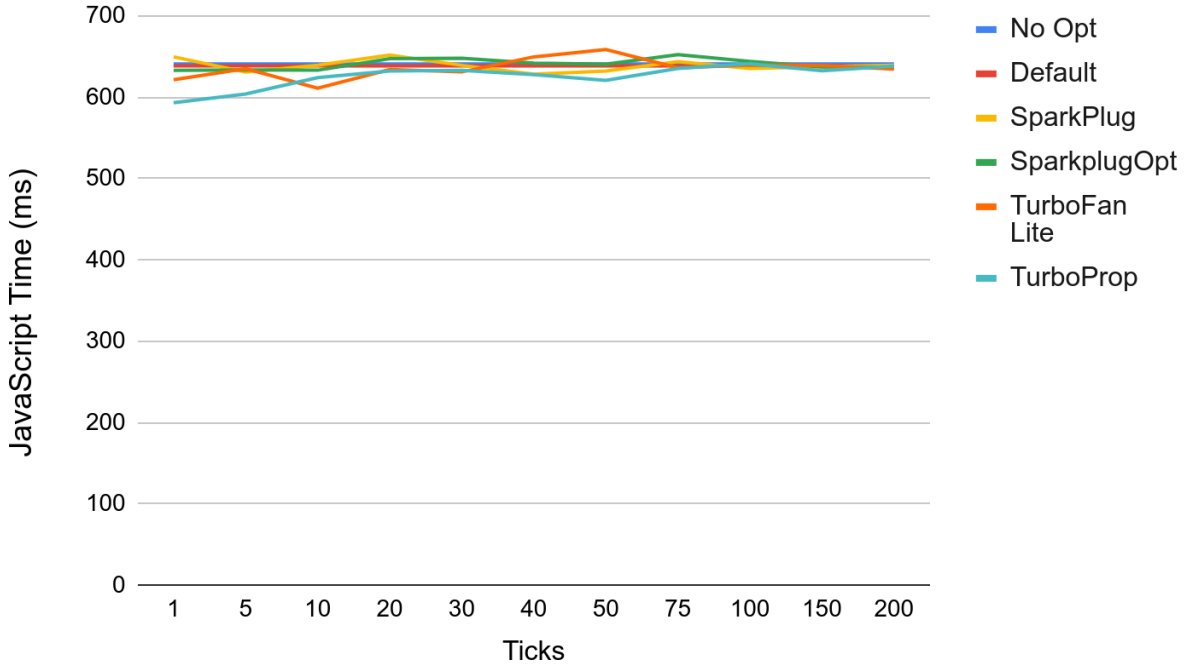

JavaScript Time (ms)
3500
3000
2500
2000
1500
1000
500
0
1
5
10
20
30
40
50
75
100
150
200
Ticks
No Opt
Default
SparkPlug
SparkplugOpt
TurboFan Lite
TurboProp

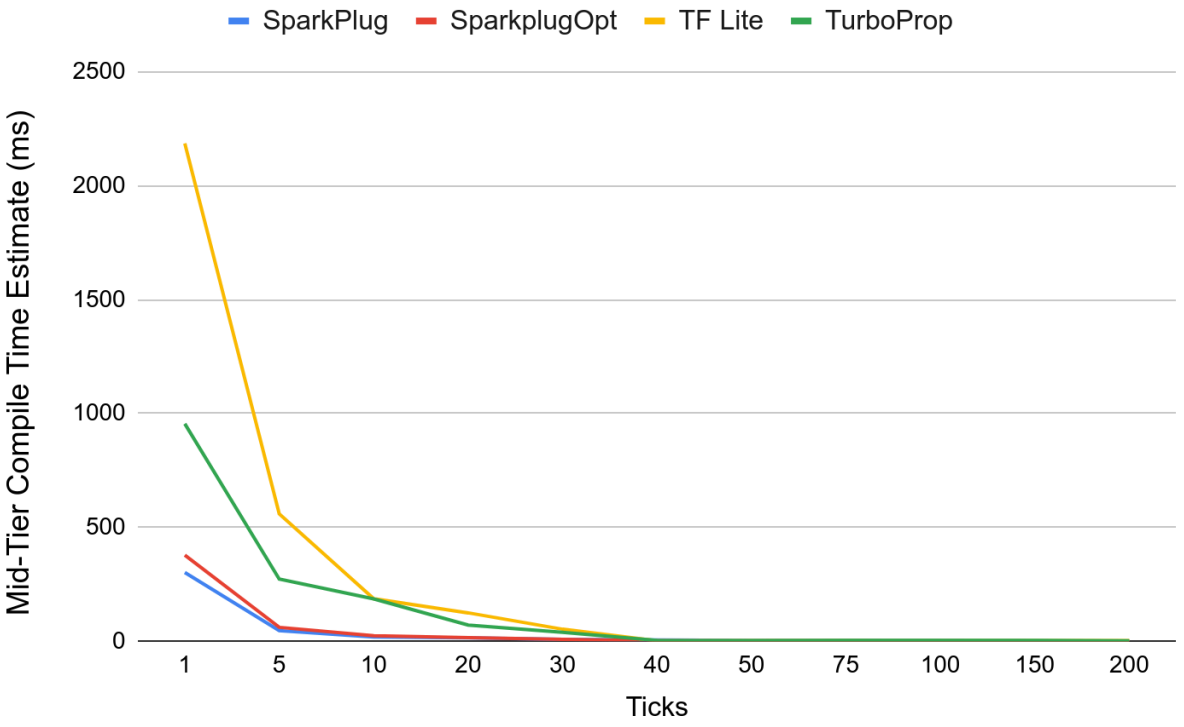

SparkPlug
SparkplugOpt
TF Lite
TurboProp
Mid-Tier Compile Time Estimate (ms)
6000
4000
2000
0
1
5
10
20
30
40
50
75
100
150
200
Ticks

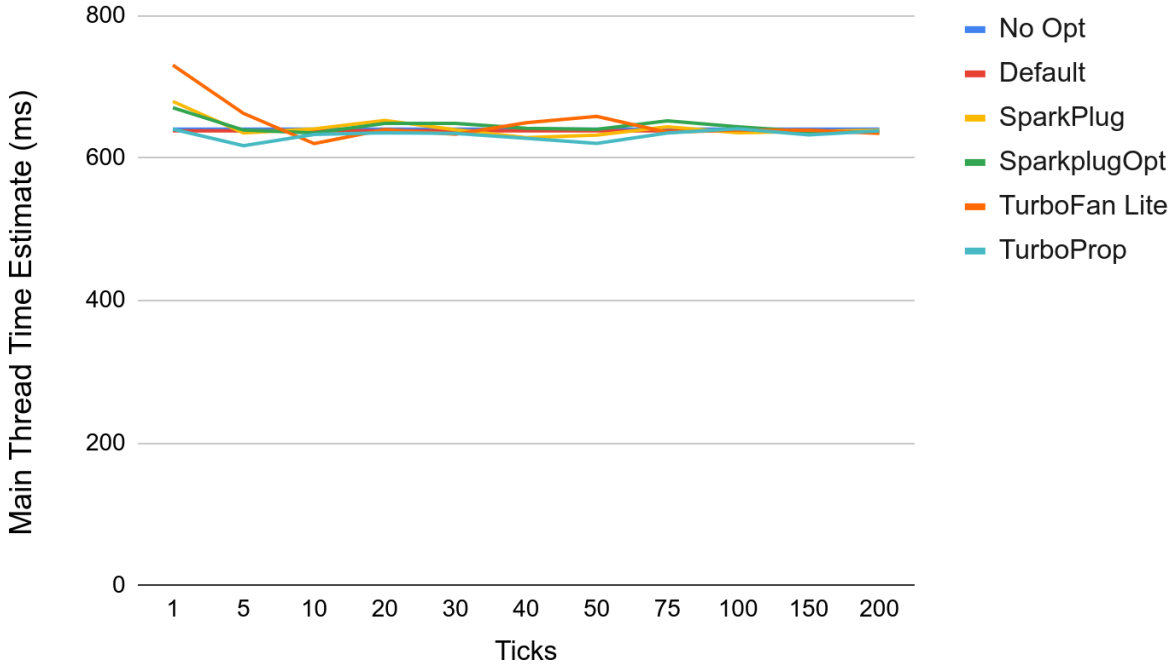

Main Thread Time (ms)
4000
3000
2000
1000
0
1
5
10
20
30
40
50
75
100
150
200
Ticks
No Opt
Default
SparkPlug
SparkplugOpt
TurboFan Lite
TurboProp

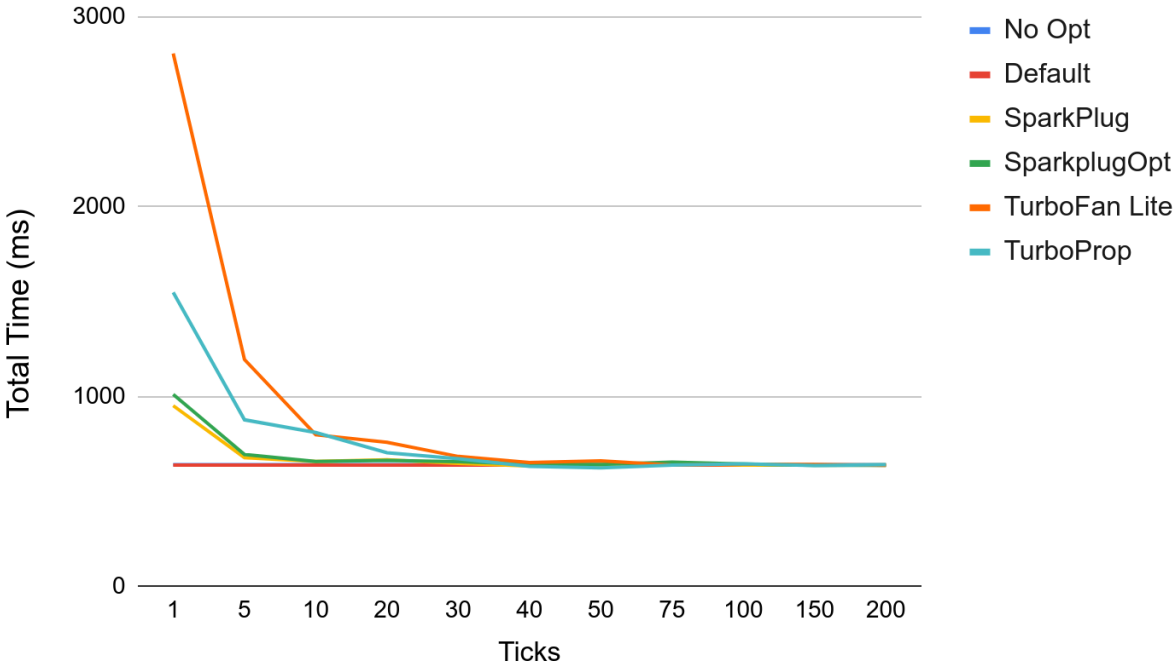

Total Time (ms)
5000
4000
3000
2000
1000
0
1
5
10
20
30
40
50
75
100
150
200
Ticks
No Opt
Default
SparkPlug
SparkplugOpt
TurboFan Lite
TurboProp

Twitter


JavaScript Time (ms)
700
600
500
400
300
200
100
0
1
5
10
20
30
40
50
75
100
150
200
Ticks
No Opt
Default
SparkPlug
SparkplugOpt
TurboFan Lite
TurboProp

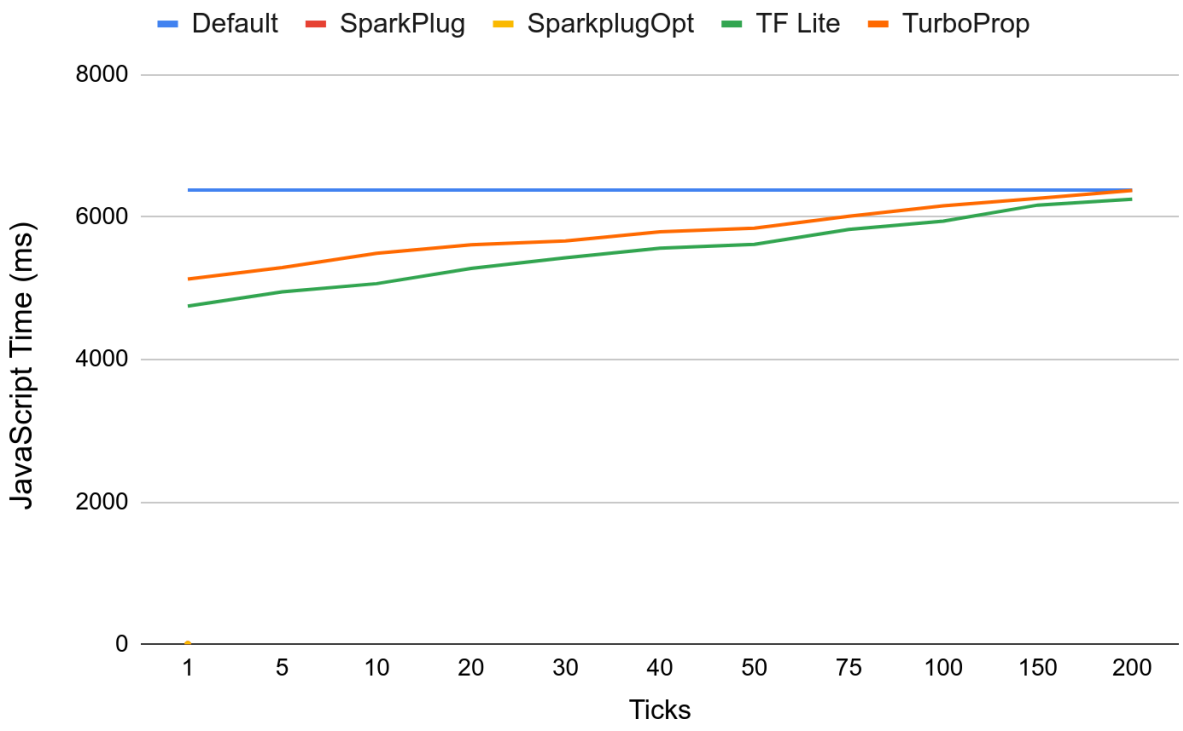

SparkPlug
SparkplugOpt
TF Lite
TurboProp
Mid-Tier Compile Time Estimate (ms)
2500
2000
1500
1000
500
0
1
5
10
20
30
40
50
75
100
150
200
Ticks


Main Thread Time Estimate (ms)
800
600
400
200
0
1
5
10
20
30
40
50
75
100
150
200
Ticks
No Opt
Default
SparkPlug
SparkplugOpt
TurboFan Lite
TurboProp

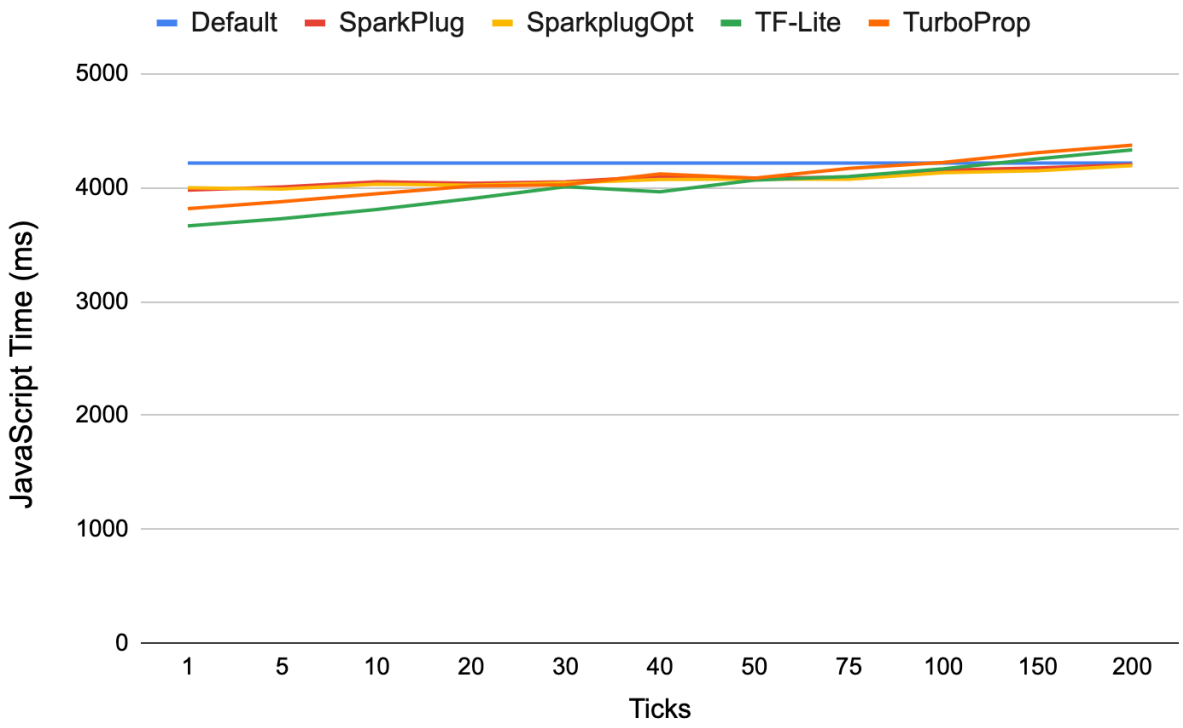

Total Time (ms)
3000
2000
1000
0
1
5
10
20
30
40
50
75
100
150
200
Ticks
No Opt
Default
SparkPlug
SparkplugOpt
TurboFan Lite
TurboProp

Speedometer Graphs

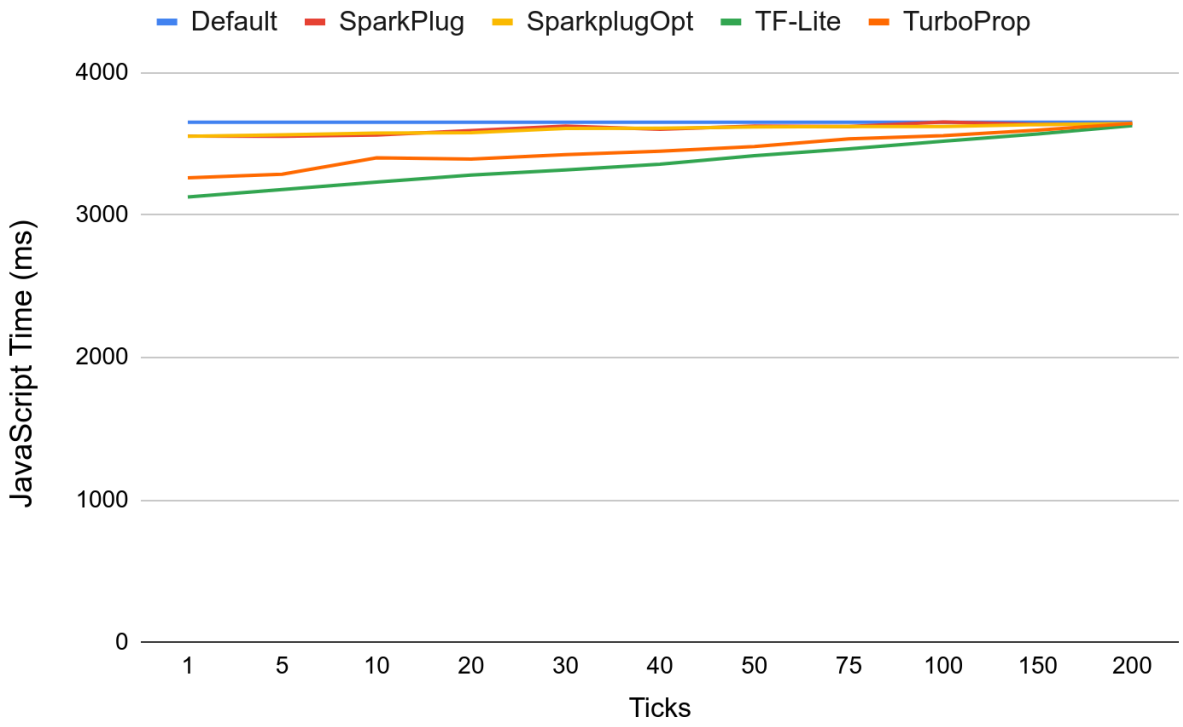
Ember



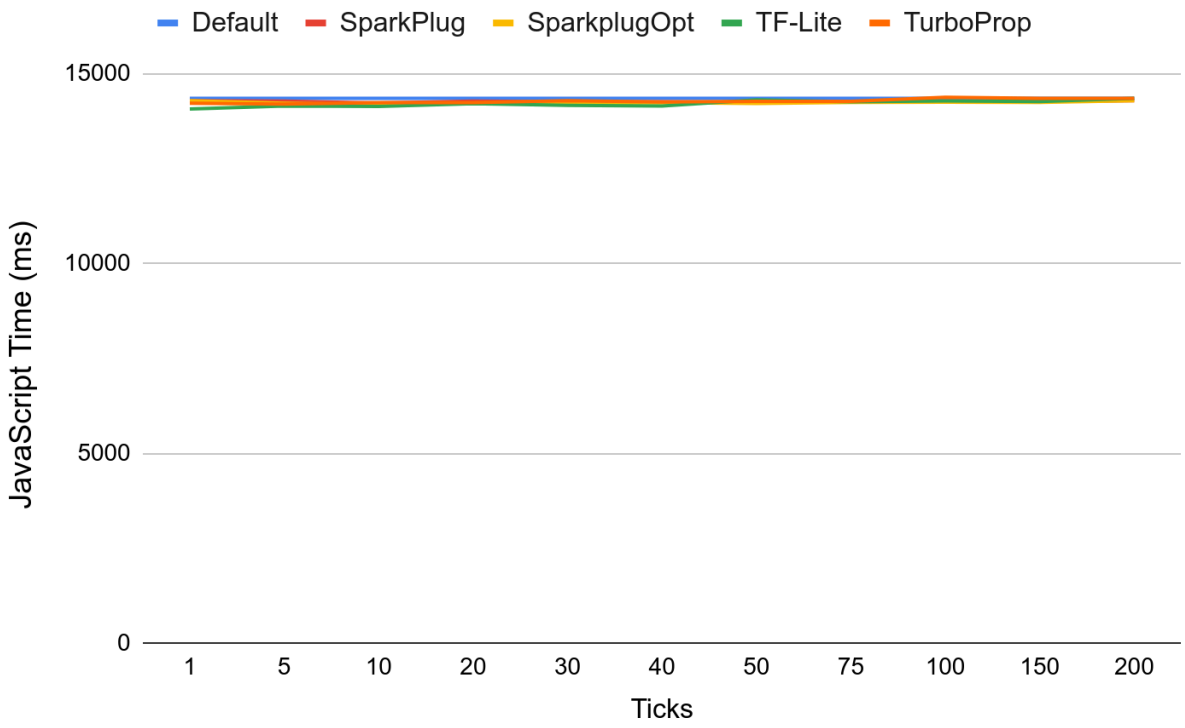
Angular



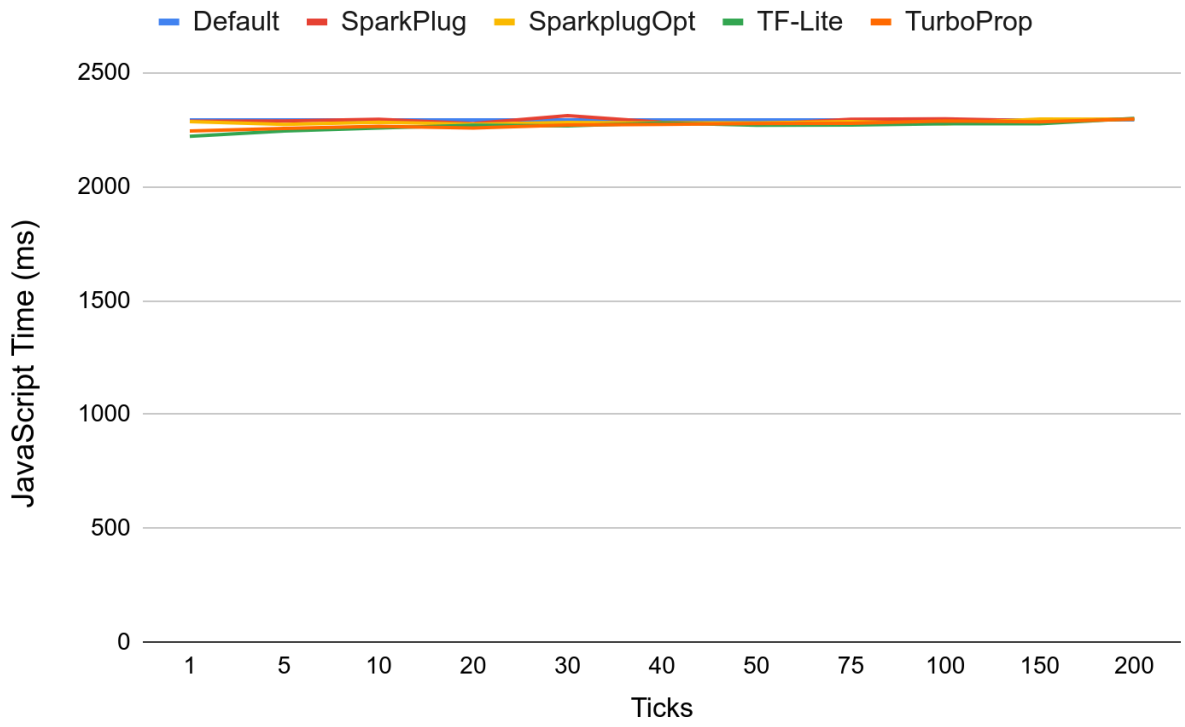
Backbone



JQuery


Default
SparkPlug
SparkplugOpt
TF-Lite
TurboProp
15000
10000
5000
0
JavaScript Time (ms)
1
5
10
20
30
40
50
75
100
150
200
Ticks

Vanilla


Default
SparkPlug
SparkplugOpt
TF-Lite
TurboProp
2500
2000
1500
1000
500
0
JavaScript Time (ms)
1
5
10
20
30
40
50
75
100
150
200
Ticks