Optimizing Web Application Start with JavaScript Loading on Demand

Denys Dubovytskyi[1], Viktoriia Zhebka[1,*], Serhii Ishcheriakov[1]

[1] State University of Information and Communication Technologies, Solomianska str. 7, Kyiv, 03110, Ukraine

Abstract

Along with the rise of the web applications popularity, their increasing utility and functionality inevitably led to the ever increasing size of the JavaScript code that is required to run them. This leads to increased time to load, parse, and execute website, which increases wait times and decreases user satisfaction. One approach to tackle this issue is to reduce amount of the loaded and executed code to the absolute minimum that is required to draw first screen of an application. During user interactions with the application, as a response to user actions additional code can be loaded and executed to properly handle user events. Asynchronous nature of the events, makes their handlers perfect targets of such optimizations, since the code does not assume time of their execution, we can add additional step of the event handling that loads the handler itself. In this study, we introduce an approach that processes application code and splits out event handlers to separate chunks, replacing original handlers with our own handlers, which load original handler and execute them. We also explore techniques to accomplish that and problems that were solved along the way. Abstract syntax trees allow us to parse source code into intermediate representation that can be further analyzed by our algorithm to find and replace respective pieces of code. Full control of generated chunks opens for us possibilities to additionally insert code that can improve user experience. By utilizing code processing, this approach does not require application developers to change their code in any way, all performance benefits can be gained by simply adding another step in the build step of existing applications. Our experimental results show up to 40% initial loading size decrease of the applications that have heavy logic in their event handlers.

Keywords

JavaScript engine, web framework, lazy-loading, programming-language, preprocessor

1. Introduction

Modern web applications are made to perform in different roles and accomplish different tasks. Complex applications that previously were a possibility only in a native desktop environments, now become available to users by the press of a button and a redirect to the application. This is made possible by all of the new technologies that were gradually added to the JavaScript language itself and to the browsers.

HTML5 brought new possibilities to the HTML elements, and EcmaScript 6[1] brought JavaScript a new, better developer experience. WebAssembly[2] allows for the high performance code to be executed in the browser and WebGL introduces highly performant graphics library making it possible for the web applications to have complex 3D applications and simulations. All of this development of technologies were supplemented by the generations upon generations of web frameworks[3], each aspiring to make developer experience even better and creating infrastructure

Workshop "Software engineering and semantic technologies" SEST, co-located with 15th International Scientific and Practical Programming Conference UkrPROG'2025, May 13-14, 2025, Kyiv, Ukraine

* Corresponding author.

† These authors contributed equally.

dubovitskydev@gmail.com (D. Dubovytskyi); viktoria_zhebka@ukr.net (V. Zhebka); ismismif@gmail.com (S. Ishcheriakov)

0009-0004-2830-9197 (D. Dubovytskyi); 0000-0003-4051-1190 (V. Zhebka)); 0009-0007-5961-8218 (S. Ishcheriakov)

that makes it possible for the web developers to tackle complex business logic and to tie it with the user interface.

This explosion of the API's and libraries made web technologies one of the most chosen technologies not only for the web-sites, but also for mobile and desktop applications. This new era brings new challenges for the web, one of them is the speed of loading and execution.

Unlike desktop and mobile applications, which can be loaded into device beforehand, and only then executed, from web applications it is expected that they are powerful and complex, and also instantly available in a click of a button. This leads to the complex balance searching exercise, where speed of loading must coexist with large code bases of big applications. This problems are made worse by making HTML pages without any page specific markup content and relying on JavaScript to build DOM tree for the user to see. This leads to the problem, where applications are rising in size, and code for the application must be loaded and executed before user begins to see anything on the page. In fact, study[4] done by Amazon, found that every 100 milliseconds of latency has cost them 1 percent of sales.

Not only long loading times decrease user experience, modern search engines include page start performance, load and execution times in their rating calculations, in order to determine position of the page in the results, which makes initial startup load and execution performance ever more important.

Typical web-site startup routine consists of several consecutive steps which are presented on the figure 1.

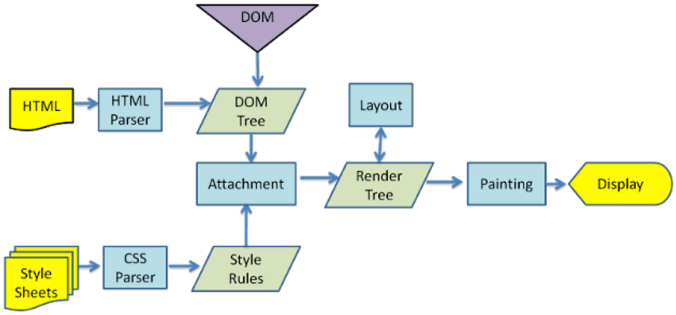

Figure 1: Web page display pipeline.

When user is navigated onto web-page, browser begins to load HTML and simultaneously during the loading of the web page it begins to parse it and build a DOM tree. These processes are executed in parallel, when enough data is loaded for the next token to be available, it gets parsed.

When during this process of loading and parsing, browser encounters a script tag, it immediately begins to load and execute said script. After script is loaded and executed, the process continues. There are several possible modes of loading embedded scripts, such as async and deferred. Async splits loads into separate thread, without interruption of the parser. When script is loaded it will be executed immediately without any delay. Deferred, compared to async, will be downloaded asynchronously in parallel, but execution order of deferred scripts is the same as they are declared in the file. Those two present one of the most basic optimizations possible for the page start time.

JavaScript execution is a complex process, which consists of parsing, execution and compiling, all done by JavaScript engine. Various browsers employ different engines, but many of them include execution specific optimizations such as optimizing compilers and tiered execution. Various approaches have been proposed to improve performance of this step, such as AOTC[5] which targets general performance of the browser engine, or snapshots[6] which also targets execution speed by utilizing results of the previous executions.

While those approaches tackle page display performance on the parsing and execution steps, our approach aims at improving performance of both, loading and executing stages of the web page display process.

The way it is done, is by analyzing source code and stripping code that handles user events into separate modules, which will be loaded and executed when event occurs. Event handlers are great targets for this specific type of optimization, since event handlers do not contain any logic that is required for the page render, and by their nature are asynchronous. Code does not assume when any of the events will fire, which allows us to insert additional step of downloading handler code before it's execution.

Another great benefit of the approach being proposed is that it does not require any changes to the source code by the application developers, meaning any code written in JavaScript which has event handlers written in a way that static analyzer can understand, can gain performance from the optimizations proposed. It is done by utilizing syntax trees as an intermediate source code representation, which allows us to traverse source code looking for the places where event handlers defined, split callback code to the separate chunk file, and replace original definition by our stub that loads original handler code from the chunk and executes it when user event is fired.

With applications that contain heavy code in their handlers, we were able to achieve up to 40 percent initial JavaScript bundle size decrease. This ability to improve application size by reducing amount of the code required to run application can work together with other optimizations done at the loading or execution level, for even better application loading performance. The remainder of the paper is organized as follows. Section 2 presents the background information abound bundle sizes and parsing of the source code. Section 3 describes proposed approach, and solutions to the problems we have encountered. Section 4 describes results of technology evaluation. Section 5 explores related work. Finally, Section 6 concludes the paper with next steps as a future work.

2. Background

2.1. JavaScript bundle sizes

Growing complexity of the web has led to the application bundle size increases. According to the report done in 2024[7], median web page size for mobile devices has grown from the 233 kilobytes to 2311, which is almost ten time increase. Highest contribution to this size is made by the images used on pages with JavaScript taking second place as the biggest contributor to the web page weight. Amount of the kilobytes loaded has profound effects on the user experience and site performance in the search engines. According to the study[8] done in June 2024 around 60% of all website traffic comes from mobile devices. While mobile devices performance greatly differs from model to model, on the average they have less performant chips than desktop and laptop counterparts. Another characteristic that is specific mostly to mobile devices, is that due to their nature they often access web-sites from the cellular network compared to the laptops and desktops which are used inside buildings and offices with stable connections.

In the result, trends indicate that web is more commonly accessed from devices having slower processors and slower internet, while average page size continues to grow. One simple approach we can take to reduce the effect of the increased application size, is to split application into chunks, only loading code that is required to perform a given task at hand, instead of loading everything. Less code, means that there is less traffic required to load website, less parsing and execution is required by the processor to display web-site. In order to split web-site, we must specify targets that we aim to separate from the main body of the source code, in this study we are proposing event handlers as a target for this kind of optimization. They are not required for the initial page presentation, and their asynchronous nature allows us to insert additional step of loading between user action and execution of the handler.

2.2. JavaScript code structure

In a typical web application, interactive pages consist of elements that are present to the user and handlers attached to them. In complex applications, event handler code can grow significantly and have its own dependencies. But since it's not required at start-up, we can improve application

performance by cutting it out from the main bundle, and then loading them on demand. To achieve this, we need to find a way to locate places in the source code that contain event handlers. We cannot just run code, and then register every executed event handler.

The reason we cannot do that is because compared to the initialization and render logic, event handlers require specific user interaction to begin execution. Since code execution ruled out as not a viable solution, we decided to turn to static code analysis. We can find places where original event handlers are defined in the source code, by first parsing them into abstract syntax tree. Abstract syntax tree is a format, shown on figure 2.

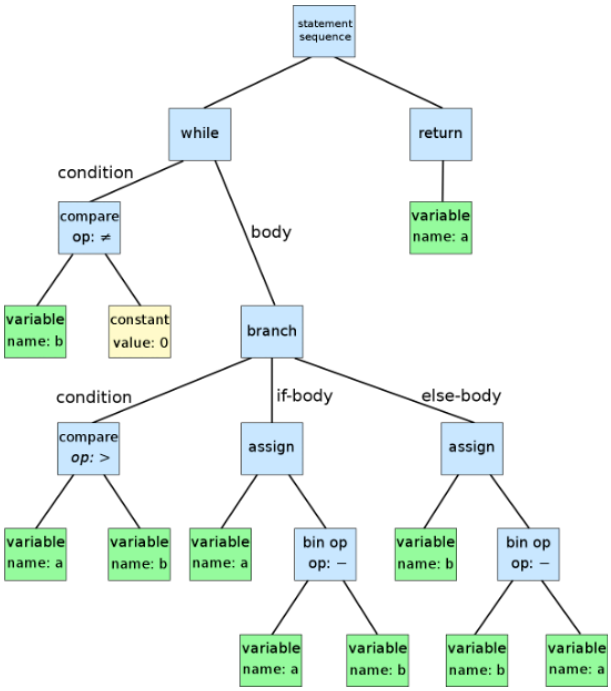

Figure 2: JavaScript abstract syntax tree.

In abstract syntax tree every aspect of the source code is presented as a leaf or branch of the tree. With this representation of the source code, algorithms used to analyze and traverse trees can be applied to the code in order to perform search and replace required places for technique to run. The structure of the syntax tree is specific for each language due to syntactic differences, and particular parser realization.

In JavaScript, by accident, standard source tree representation was derived from the work done by the Mozilla on the Spider Monkey JavaScript interpreter and just in time compiler. Currently it lives as a ESTree[9] specification that is maintained by the volunteers on the Github, and utilities that perform work related to JavaScript source trees can utilize this standard in order to be compatible with other utilities performing same work. In ESTree source code is composed by two types of nodes, one is expression and the other is statement. Expression is something that after evaluation by the interpreter is expected to return some kind of a result, while statement is part of the code that does not return any kind of the result and instead just changes state of the application. This standard serves us well in our mission to analyze JavaScript source tree, since we can utilize utilities which already available to us such as Acorn[10] and estree-util-to-js[11].

2.3. Parsing techniques

Composition of the abstract source tree from the source code involves several steps, the first one is called lexing or scanning, and the second is called parsing. During the lexing step of the abstract tree creation, source code is being traversed character by character separating unique language specific pieces of syntax called tokens. Each token represents a separate piece of the language

grammar. At this step there are minimum effort taken to analyze source code for syntactic and grammar errors, most of the analyzing work is delegated to the parser. The lexer's output will be an array of strings called tokens.

Then, it is parser's job to take array of tokens and to create abstract syntax tree from them. During parser execution tokens are organized into nodes of the tree. Taking into account nested nature of the JavaScript source code, it is a popular choice for the parsers to employ recursion as a mechanism to convert linear array of tokens into tree-like data structure. Recursion allows to store previous execution context in the closures of the function, which in turn utilizes stack as a data-structure that stores ancestors for the current token being evaluated.

Parser has knowledge of the language syntax semantics and structure, during execution it keeps a chain of ancestors for each node. This knowledge of context that precedes current token, allows it to evaluate any token as a node and derive it's type. If parser encounters any of the tokens that are not expected in the current context, it will notify us by raising an error. Each node in the tree has its own specific set of fields for children, which are unique for the specific node type. For example, multiply operator expression node will have left and right operands stored in a separate fields, and both operands will be expressions themselves. At the end of the parsers work, result will be either error or a syntax tree that represents valid source code.

2.4. Recursive tree-walker

As we have to analyze the source tree, after parser is executed the resulting syntax tree is available to be analyzed. As a part of the source analysis, we have to traverse nodes of the tree, looking for specific tokens. The mechanism that allows us to traverse nodes easily is the same as with the parser, recursion.

Recursion allows storing node ancestors using closure variables, which end up stored in stack along with functions that got pushed during entry into nested structures such as tree, and then retrieved from when we exit any particular branch. Since recursion uses stack to store scope of nested functions, one possible downside of this approach can be stack overflow error. It can be eliminated by using programming languages that support tail-call optimization during compile step of the application.

For each particular node, we can take decision whether we want to continue to walk over its children, or not. This ability to stop walking any particular branch any further is useful, since we are going to look for event handlers at the top level, any nested event handlers are not of interest for this study.

3. JavaScript loading on demand

3.1. Overall scenario

Many web frameworks currently allows for web app loading optimizations such as server side rendering. While it is viable technology to improve website loading times, it comes with its own drawbacks, specifically a need for hydration. Another problem is that if project does not use any frameworks, or uses frameworks that do not come with such optimization, it means that developers are left on their own to implement complex optimization techniques for rendering and code splitting.

In this study, we are exploring technique that will be applicable to any web-site, regardless of the technology it was built on. As long as it uses JavaScript and our static analyzer can determine that particular piece of code corresponds to event handler, they will gain benefits from using proposed technology.

The desired result, is that by splitting out parts of the original source code, we can decrease initial website loading time, and progressively load required pieces of code as a response to the user interaction.

When user interacts with element, as a result of action additional step of loading respective code will be done, and execution of original handler will be delayed for the time of loading. Implications of user experience as a result of such delay will be explored further in this study.

This in turn should result not only in improving initial load times, but in complex applications it can lead to the general decrease of the user loaded code, since functionalities that were never requested by user would never be loaded and executed for them, decreasing traffic for client and for server. While additional step in the build toolchain of the particular project may incur additional build times, it is our understanding that larger project would already have long build times, and as large projects are the ones to gain the most from the proposed techniques, additional build time seems justifiable.

3.2. Analyzing for split

In order to find places where we can separate main source file, we have to look for specific tokens. In a web page, there are three ways to register a callback. The first way to set up a callback is to specify it on HTML tag of eligible element. As a result of user interaction, browser will look for a handler in the global window object. This approach is not common due to the fact that for large applications it is not feasible to have all of their possible callbacks inside one global window object, due to the possible collisions and interference. Another problem is that in large applications most of the user interface is built using JavaScript so there is no HTML tag to add this kind of callback. We will not optimize this type of the event handlers due to the fact that they are not common, and it would require parsing of HTML which is out of scope for this particular study. Two another approaches to add event handler to element done using JavaScript. The first of them is that given a reference to the element we want to add handler, element object itself contains fields for the handlers, and we can assign our callback function as we would assign another regular property. The main drawback of this approach is that unless taken additional precautions it will overwrite previously set callbacks. Since field names are specific words defined in the element API, inside our abstract tree walker we can look for assignments where field being assigned to has specific name that corresponds to event handler, e.g. “onclick” as shown in figure 3.

```
const buttonElement : Element = document.querySelector( selectors: '#button');
buttonElement.onclick = () : any => console.log('event');
```

Figure 3: Assignment of onclick handler.

After finding this kind of assignment, we can replace original handler with our own. The second, and the most popular, is using API called addEventListener, shown on figure 4.

```
const buttonElement : Element = document.querySelector( selectors: '#button');
buttonElement.addEventListener( type: 'click', listener: () : void =>{
    console.log('event');
});
```

Figure 4: Assignment of click handler using function API.

Elements expose function with a respective name, and new event handlers can be registered for element using this function. The first argument is event type to be handled, and the second is callback function itself. This is the most common way to assign new event handlers, since it will not override any previous callbacks. We can detect this kind of event handler registration by looking for a function call, where name of the function being called is “addEventListener” and first argument is name of event we are looking for e.g. “click”. After finding this function execution, we can replace original function with our own.

In both cases, it is possible instead of passing function definition to pass just a reference to a handler function. In this cases, we cannot replace function being called since we deal with the reference, and same function can be used in multiple handlers. In scope of this study we do not handle such scenarios and instead just ignore situations where reference to a function is passed instead of anonymous function.

Another consideration that is not taken into account, is that source code may have multiple files with similar handlers. In this study we focus on one source file which we split into several, assuming that all of the source is bundled inside one file which is common scenario in the web.

3.3. Eval vs import

After we have placed the original code to the separate file, another great challenge is deciding what should be put instead. Ideally we would want to load and execute code as it never was transferred to any other file, but instead was executed as a native code. In JavaScript, there are several ways to achieve loading and execution of the code. With new standards, we now have module script types, that can dynamically and statically import other files using import keyword. This is the most common and suggested approach. We tested it and unfortunately, while this approach is viable for the dynamic loading and execution of code, each file downloaded this way will be executed as another top-level script, without access to the scoped variables of the original code. Since we want to execute loaded code as it executed in the original context, with access to all of the variables and context, this approach does not work well for us. Also the fact that original script has to be declared with the type "module" does not make this solution universal enough.

Another approach is to use other JavaScript API's, such as XMLHttpRequest, fetch and eval. Using XMLHttpRequest or fetch we can load code chunk as a regular string, and then use eval to execute it. While eval is not recommended for general use due to security considerations, since we are executing original code from the application that was split from the main file before, it is safe to assume that this code will be safe, because we control both ends of the loading: generating chunk file, and loading said file using URL that we generate. One problem that we have encountered with this approach, is that original handler code can have return statements inside of it, to finish handler execution early. When loading using fetch and executing such code using eval, during parse step of the evaluation JavaScript parser has no knowledge that this code will be executed inside function context, and raises exception that return statement is illegal here. Fortunately, since we control code generation for chunks, one way to avoid this kind of errors while preserving functionality is to wrap code in chunks in immediately executed function expression IEFE. Which will make return statement legal, and will preserve original intention of return statement to finish function execution.

3.4. Preserving function context

Another problem that we have encountered is that due to the differences in a way JavaScript handles context of execution of regular functions and arrow functions. In JavaScript, "this" keyword is handled differently taking into account how function was defined, and how it was executed.

In case of arrow functions, function body gets executed in lexical scope, which just means that scope of the execution can be determined just by locating code inside of the function. This is different when compared to the dynamic context of the regular functions. In regular function, "this" means object on which function was called. This is relevant to the use case of the event handlers, since when event handler is defined as a function, it gets called on the DOM element, having "this" set as an element on which event was fired.

Since handler might want to access details of the element it was being called on, we have to preserve this behavior. The way to correctly handle this problem is to generate functions of the same type as the original function being replaced, which means that if we are replacing arrow function handler we have to generate arrow function in its place, and when we are replacing

regular function, our replace stub should be of regular function type. In this way, behavior of "this" gets preserved while chunk is being executed. Another aspect of the generated function is that handler might want to access parameters it is called with, so our replacement function should have same parameter list as the function it is replacing. Figure 5 shows possible scopes of the JavaScript application.

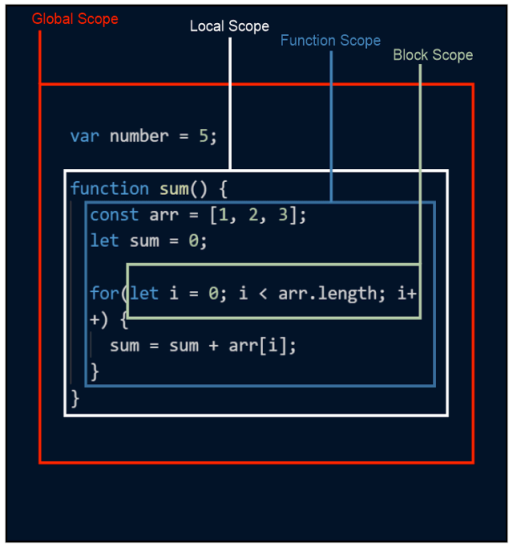

Figure 5: JavaScript scopes.

3.5. Chunk generation

As a result of the preprocessor algorithm execution, chunks will be generated for each separate handler. This is desired behavior and is a necessary tradeoff for the technique being used. That being said, large amount of chunks can become burdensome for the application to load. For example, if application already makes a lot of requests, adding additional chunk request may impede overall network performance of the application if it is being run on the HTTP/1. This problems are negligible if the application being served using HTTP/2, where multiplexing allows for a large amount of simultaneous requests from the application.

Another potential problem with chunks is that they should be cached instead of reloaded on every execution. This can be done in several ways, one of them is to configure caching on the server by specifying caching headers in the request that serves chunks, but this is something that our utility does not have control over. Another way is to store record of what chunks have already been loaded, and access their code directly. This technique will require for the utility to insert some additional runtime code to allow for storing and retrieving of the already loaded chunks. This has great potential and is important aspect, but it is out of scope for this particular study. Currently we assume that all chunks are being generated from the one source file, so we can generate their unique names by simply having a counter, and increasing it with each generated chunk. With multi-source implementations it might be required to have more sophisticated chunk name generation logic.

3.6. User experience

The main goal of this approach is to decrease initial time that is required to display page. It is done by reducing amount of work done at every step: downloading, parsing and executing. And while proposed approach saves some of the time during initial loading and execution of the application, code that was stripped out of the program on loading step still has to be downloaded and executed during the application lifetime. Since user application is expected to handle all of the possible user events, need to download additional code was not eliminated but was shifted in time until user actions demand them. Any delays avoided during startup must be incurred during user interaction

with application. This will cause user actions to have delayed execution and will decrease user experience. There are ways to mitigate that, one is making user cursor to display “progress” state, as shown on figure 6, so that it is visible that something happening and users have to wait.

Figure 6: Cursor in progress state.

Since we control generated code and all of the steps of loading and execution, it is possible to introduce additional code that will manage display of loading state. It is common on the web for actions to have some kind of the progress state, so while the user experience will be degraded, most of the time it should still be good enough. But even this drawback can be even further minimized in future iterations of the technology.

3.7. Additional possible optimizations

Most of what described above sets strong foundation for the further modifications that can be done to optimize web-site loading even better. Currently our static analyzer only removes bodies of the event handler function, and while this has great benefit for the loading times, we can go even further with this concept by analyzing all of the dependencies that event handler requires, and cutting them out too. Currently, if event handler uses some library, library still will be present in the initial loading code, but with additional source analysis, when determined that some particular dependencies are required only for the handler code, it can also be cut out into separate chunk, reducing initial load size even more.

Another great potential of this technology is in the fact, that while cutting out event handlers, we are replacing them with our own event handler. This gives us opportunity to have fine control over event execution and analysis. We can insert profiler code, which will collect information about user interaction with handlers, and optimize chunk generation based on the data from the interaction with users. For example, if profiler determines that some of the events are called more than other, we may start loading handler even before user has interacted with it. Various strategies can be implemented upon this analysis, for example start of the handler loading can happen on mouse hover, or on element appearance on screen. Ability to profile events can provide insights in which particular scenarios any given event is probably the most needed.

Current analyzer can detect only events that have specific names our parser knows about, but often developers can store names in variables and use them as event names during registration, and therefore finding places where user adds event handlers continues to be area that can be improved upon.

User experience still remains an issue, and additional runtime code can be generated in order to mitigate consequences of loading JavaScript after every user interaction. Few possible approaches are adding additional parameters to the generated code in order to provide a way for the application to display loading state, whether using mouse cursor, loading progress bar, or disabling element that was interacted upon.

4. Evaluation

Since the target of the technology being described are applications that do not rely on the frameworks for their user interface, we tested our approach on applications that have been written using vanilla JavaScript without additional frameworks. We have used calculator and to-do list applications to evaluate results of the technology.

With to do list application it was observed that main script size was decreased from 8200 bytes to 8000 bytes. This result reveals to us that in applications that do not contain heavy code in their handlers, current approach can have negligible effect on the application size.

Another application that we have evaluated is calculator application. In this application, original script size was 6200 bytes, and was reduced to 4700 bytes, nearly 25 percent of the original size.

This is due to the fact that most of the handler code were defined inside handler function, instead of being moved somewhere else where our analyzer could not detect and extract them. With additional analysis, gains as this or similar can be expected on most web application that contain complex code to handle user interactions.

5. Related works

During research for this article studies such as AOTC[5] were considered. In this article authors propose a novel way to improve performance of the multi-tiered browser engines by saving results of the optimization compiler in the special file, and during the next execution instead of going through the tiers of the interpreters browser can reuse optimized compiled code from the previous executions.

While this study addresses speed of the execution, it does not mitigate loading times. Another study that was considered is snapshots[6]. In their article authors explore technology to save heap and DOM tree of the latest execution, so all subsequent website visits can be restored to the particular saved state instead of the browser going through all the steps of loading, parsing and executing JavaScript.

As with previous study, loading times of the application are not addressed in this study. Compared to the previous methods, our approach targets loading times as well as execution times, and is not mutually exclusive, meaning all of the methods that can be used to improve browser performance can be used with the approach proposed in this study. Concurrent JavaScript parsing which was proposed in study[12] was also considered, and while it has positive effects, our approach saved time on every step of the application display pipeline, and can be combined with this approach too. Another study[13] that was considered is optimization of v8 engine, in which authors increase performance of the JavaScript executing engine itself. Additionally we have considered study of selective hot-spot compilation[14] and thread level speculation[15] to improve JavaScript startup performance. Compared to those studies, our approach saves amount of the code to be executed, and can be applied along with this study to achieve even better results.

6. Conclusion

This article proposed approach to analyze and strip away parts of the source code, with ability to load and execute it on demand in the same context as it was stripped away from. It does not need any modification by the application developers in any way and can be applied to any existing codebase without additional work. It also can be combined with other methods of increasing performance for additional gains in speed.

We made the following contributions in this article:

- Proposed approach of analyzing and stripping away code based on abstract syntax tree.
- We resolved the issues related to the parse errors of the downloaded handler code.
- We decreased bundle size by up to 25% for the applications that have heavy event handlers.
- We efficiently resolved the issues related to the dynamic context of the function execution.

Since trends of the web applications size growth do not indicate stopping, approaches that do not only increase performance of the browsers, but reduce overall amount of the code that will be

downloaded and executed can make great impact for applications that have complex code bases and loading everything up front does not seem feasible anymore.

We also identified possible areas for the improvement of said technology, which can have outsized impact on large codebases, such as further analysis of the source code and profilers in the event handlers.

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] ECMAScript® 2026 Language Specification. URL: https://tc39.es/ecma262.

[2] WebAssembly. URL: https://developer.mozilla.org/en-US/docs/WebAssembly.

[3] J. Hannah, JavaScript Frameworks by the Numbers, 2018. URL: https://javascriptreport.com/javascript-frameworks-by-the-numbers-winter-2018.

[4] Amazon Found Every 100ms of Latency Cost them 1% in Sales. URL: https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales.

[5] H. Park, S. Kim, J-G. Park, S-M. Moon, Reusing the optimized code for JavaScript ahead-of-time compilation, ACM Trans Archit Code Optim, volume 15, issue 4, 2018. doi:10.1145/3291056.

[6] J. Yeo, J. Oh, S-M. Moon, Accelerating web application loading with snapshot of event and DOM handling, in: Proceedings of the 28th International Conference on Compiler Construction, Association for Computing Machinery, New York, NY, USA, 2019, pp. 111-121. doi:10.1145/3302516.3307355.

[7] D. Smart, J. Indigo, Page Weight, The Web Almanac by HTTP Archive, 2024. URL: https://almanac.httparchive.org/en/2024/page-weight.

[8] J. Howarth, Internet Traffic from Mobile Devices, 2025. URL: https://explodingtopics.com/blog/mobile-internet-traffic.

[9] The ESTree Spec. URL: https://github.com/estree/estree.

[10] acornjs/acorn: A small, fast, JavaScript-based JavaScript parser. URL: https://github.com/acornjs/acorn.

[11] syntax-tree/estree-util-to-js: estree (and esast) utility to serialize as JavaScript. URL: https://github.com/syntax-tree/estree-util-to-js.

[12] H. Park, M. Cha, and S-M. Mook Moon, Concurrent JavaScript Parsing for Faster Loading of Web Apps, ACM Transactions on Architecture and Code Optimization, volume 13, issue 4, 2016, pp. 1–24. doi:10.1145/3004281.

[13] W. Ahn, J. Choi, T. Shull, M. J. Garzarán, J. Torrellas, Improving JavaScript performance by deconstructing the type system, ACM SIGPLAN Notices, volume 49, issue 6, 2014, pp. 496–507. doi:10.1145/2666356.2594332.

[14] S-W. Lee, S-M Moon, Selective just-in-time compilation for client-side mobile javascript engine, in: Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded, Association for Computing Machinery, New York, NY, USA, 2011, pp. 5–14. doi:10.1145/2038698.2038703.

[15] J. Martinsen, H. Grahn, Thread-level speculation as an optimization technique in Web Applications — Initial results, In: Proceedings of 6th IEEE International Symposium on Industrial Embedded Systems, 2011, pp. 83 - 86. doi:10.1109/SIES.2011.5953686.