Accelerate JavaScript Applications by Cross-Compiling to WebAssembly

Micha Reiser
micha.reiser@hsr.ch
HSR Hochschule für Technik Rapperswil
Department of Computer Science
Rapperswil, Switzerland

Luc Bläser
luc.blaeser@hsr.ch
HSR Hochschule für Technik Rapperswil
Department of Computer Science
Rapperswil, Switzerland

Abstract

Although the performance of today's JavaScript engines is sufficient for most web applications, faster and more predictable runtimes could be desired for performance-critical web code. Therefore, we present Speedy.js, a cross-compiler that translates JavaScript/TypeScript to WebAssembly, a new standard for native execution supported by all major browsers. Speedy.js only imposes minimal restrictions on the JavaScript code, namely that the performance-critical functions are wrapped in TypeScript and only engage a performance-optimal subset of the JavaScript language. With this approach, we manage to make compute-intense web code up to four times faster, while reducing runtime fluctuations to the half.

CCS Concepts • **Software and its engineering → Compilers**; *Scripting languages*;

Keywords WebAssembly, JavaScript, TypeScript, Cross-Compilation, Optimization, LLVM

ACM Reference Format:
Micha Reiser and Luc Bläser. 2017. Accelerate JavaScript Applications by Cross-Compiling to WebAssembly. In *Proceedings of ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL'17).* ACM, New York, NY, USA, 8 pages.
https://doi.org/10.1145/3141871.3141873

1 Introduction

The web has become a ubiquitously available application platform and increasingly popular target for a wide variety of applications. Traditionally, these applications are written in JavaScript, as it is the only natively supported high-level language of the web. Despite the tremendous and ongoing improvements of JavaScript engines [14], the performance of JavaScript is still insufficient for compute-intense applications like simulations, graphic manipulation, cryptography, or games. The reason for the performance discrepancy are not inadequately designed JavaScript engines, but mainly inherent to the design of the JavaScript language itself that favors ease of use over performance. These considerations in conjunction with the ubiquity of the web platform raise interest in a compilation target for high-level languages on the web.

The first specification that allows JavaScript as a compilation target for low-level programming languages like C++ is asm.js [11]. It is a restrictive subset of JavaScript that supports ahead-of-time-optimizing compilation and achieves near-native performance [33]. Since asm.js compiles to JavaScript, it inherits the disadvantageous properties of the non-predictable engine-specific performance, in addition to a significant overhead for parsing and compiling the source code before execution [6, 32].

The WebAssembly standard advances the idea of asm.js by defining a machine-close, yet platform-independent intermediate language in binary code format for the web [10, 28]. WebAssembly is supported in the latest stable or the upcoming versions of all major browsers [25, 26]. Furthermore, various compilers of existing languages, like C++ and Rust [4, 9, 22, 30], as well as for new languages [1, 29] specially designed for WebAssembly, support it as a compilation target. This ecosystem gives the programmer the desired freedom to choose the programming languages with the most suitable characteristics for a specific application, e.g. one that favors performance over ease of use for compute-intense applications.

Introducing one of the languages mentioned above to a project based on JavaScript adds undesired complexity, especially if there are only a few performance-critical functions. In fact, it forces a project to swap out performance-critical code into a separate project, written in a language that supports WebAssembly as a compilation target. Moreover, it requires a manually programmed bridge between the conventional JavaScript and outsourced WebAssembly code. Currently there exists no tool that automatically accelerates web applications through WebAssembly compilation.

VMIL'17, October 24, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5519-3/17/10…$15.00
https://doi.org/10.1145/3141871.3141873

Therefore, we present Speedy.js, a cross-compiler that translates performance-critical JavaScript/TypeScript code to WebAssembly and generates the necessary glue code to integrate the WebAssembly code inside the conventional JavaScript application. This approach eventually enables significant accelerations of compute-intense web code, while reducing runtime fluctuations that otherwise common in JavaScript browser engines.

Speedy.js only imposes a few restrictions on the programmers: It only supports a subset of JavaScript for the implementation of performance-relevant functions in order to avoid typical performance-problematic language features. It also favors performance by relaxing memory safety within the sandboxed WebAssembly browser execution, such as by omitting array boundary checks or memory default-initialization. Moreover, Speedy.js requires that the functions for Web Assembly cross-compilations are written in TypeScript with type annotations used by our cross compilation, together with a special directive that marks the function as performance-critical. Considering that TypeScript is widely used in practice as a typed dialect of JavaScript, this should be no relevant practical restriction for web programmers either.

In summary, this paper makes the following contributions:

- Description of a new compiler tool for accelerating performance-critical web code through translation to WebAssembly.
- Report on the experimental evaluation of the performance gains achieved by this new tool.

The remainder of this paper is structured as follows: Section 2 describes the programming model of Speedy.js. Section 3 explains the cross-compilation. Section 4 describes the runtime system. Section 5 evaluates the performance gain of Speedy.js. Section 6 compares Speedy.js with related work and is followed by section 7 concluding this paper.

2 Programming Model

Speedy.js allows to accelerate certain functions in JavaScript/TypeScript code that are compute-intense. For this purpose, the corresponding functions have to be written in TypeScript by using type annotations and attributing an additional directive. As TypeScript is a super-set of JavaScript and is also transpiled to JavaScript, these Speedy.js-enabled functions can easily be invoked by TypeScript and ordinary plain JavaScript code.

Listing 1 shows the example of an `isPrime` function (line 1) implemented in Speedy.js that determines whether the passed `int` argument is a prime number or not. The `isPrime` function is called from the regular function `main` (line 18) that logs the result to the console. The `"use speedyjs"` directive as the first statement of the `isPrime` function (line 2) identifies it as a Speedy.js function. In this example, only the `isPrime` function is translated to WebAssembly, while `main` is mapped to JavaScript code that contains the necessary glue code for invoking the WebAssembly-based `isPrime` function. Speedy.js functions are transparently callable from regular TypeScript (line 18) or equivalent plain JavaScript code. Since the JavaScript API for loading a WebAssembly module is asynchronous, these functions must be declared `async`.

```
1  async function isPrime(value: int) {
2    "use speedyjs";
3
4    if (value <= 2) {
5      return false;
6    }
7
8    const sqrt = Math.sqrt(value) as int;
9    for (let i = 2; i <= sqrt; ++i) {
10     if (value % i === 0) {
11       return false;
12     }
13   }
14   return true;
15 }
16
17 async function main() {
18   const prime = await isPrime(10000);
19   console.log("isPrime 10000: " + prime);
20 }
```

Listing 1. Prime number check implementation using Speedy.js

As for implementation of Speedy.js functions, only a subset of the JavaScript language is supported. This chosen subset only covers language features that can be mapped to efficient native code, such as variables, most expressions, assignments, control statements, arrays, classes, as well as function declarations and calls (even recursive). Features that are deliberately excluded for Speedy.js functions are prototype inheritance, object literals, closures, reflection etc.[1] The usage of unsupported language features results in a compile-time error. The motivation behind this restriction is that programmers can rely on an efficient execution of Speedy.js code without incurring performance pitfalls as it is often the case in full JavaScript.

Besides the language restriction, Speedy.js requires an increased programming discipline for the performance-specific functions, since memory safety is relaxed to achieve the highest possible performance: Array indexes must always be valid within the boundaries, as array boundary checks are omitted. Uninitialized memory must not be read, as corresponding checks are also removed. Moreover, type casts are

[1]The complete list of supported language features is available on the project's wiki [16].

unchecked and should therefore never fail. Of course, the omission of these checks impacts memory safety. However, as the Speedy.js code is executed in isolation by the JavaScript engine, a faulty application cannot access the code of another JavaScript application or system program and thus cannot corrupt other programs (apart from terminating or blocking the entire JavaScript engine in the worst case). We regard this relaxation reasonable, as the performance-critical parts are usually designed with particular care and performance has highest priority: Marking functions as Speedy.js thus implies an explicit opt-out of this memory safety guards.

The Speedy.js functions eventually compile to WebAssembly, a machine-close intermediate language that runs in all major browsers as well as Node.js, and has near-native performance [10].[2] WebAssembly is also less affected by browser-specific JIT optimizations [6, 32].

A further limitation of our current programming model is that the arguments and result values of Speedy functions are copied and the functions cannot effect side-effects on memory shared with the remaining JavaScript code. This is due to the current WebAssembly standard that is based on separated (and thus isolated) memory spaces. WebAssembly only supports passing of primitive values. Speedy.js emulates the support for passing arrays and objects by transitive serialization and passing the snapshots across the call boundaries. However, reference equality is maintained for objects passed by the same call or result. Moreover, the accumulated size of all allocated objects per Speedy.js entry function invocation is limited because garbage objects are not yet reclaimed in our runtime system until the entry function exits.[3]

3 Cross Compiler

The compiler driver of Speedy.js orchestrates the compilation process. It compiles the Speedy.js functions from a TypeScript input file to WebAssembly. Furthermore, the compiler generates stub functions allowing to call Speedy.js functions transparently from TypeScript. The compiler uses a fork [17] of the official TypeScript version that has support for the base type `int`. The fork allows the compiler to generate more efficient code for integer computations.

Figure 1 shows the compilation pipeline for a single TypeScript input file. At first, the TypeScript compiler is used to parse the source code and type check the program. It is a prerequisite for the following steps that the input program contains neither syntax nor typing errors. If this is not the case, the compilation halts immediately. The directly succeeding steps work with the parsed TypeScript AST and can be separated into two pipelines:

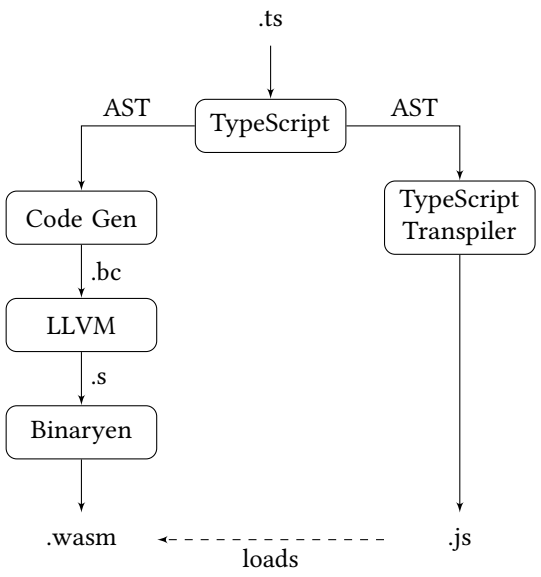


Figure 1. Speedy.js compilation pipeline

- The left pipeline compiles the Speedy.js functions to WebAssembly by transforming the TypeScript AST to a control flow graph, optimizing it using LLVM [13], and transforming it to a stack machine — as such WebAssembly is defined. Finally, Binaryen [3] is used to perform WebAssembly specific optimizations and creating the WebAssembly module.
- The right pipeline replaces the TypeScript implementations of Speedy.js functions in the transpiled output with stubs containing the glue code allowing a transparent integration into JavaScript/TypeScript applications.

The following paragraphs explain the separate compilation steps.

Code Gen The code generator is the frontend of the cross compiler. It translates the input program to the LLVM [12] assembly language (IR) by traversing the AST and generating the corresponding instructions, basic blocks, and control flow edges for each node. The IR is a Static Single-Assignment (SSA) [2] based representation of low-level operations. The code generator first creates stack based allocations for local variables and uses the LLVM *mem2reg* pass to transform the IR in pruned SSA form — as advised in the LLVM documentation [13]. The IR code is saved in a temporary LLVM bitcode file (.bc) for further processing.

[2]The latest versions of Chrome, Edge, Firefox, and Node.js support WebAssembly by default or as an experimental feature. The experimental version of Safari supports WebAssembly.

[3]The 32-bit WebAssembly standard defines 2 GBs as the maximum memory size that we also allow to exploit.

LLVM The compiler driver uses the LLVM toolchain [12] for linking the program against the Speedy.js runtime library[4] and performing compile-time optimizations. The offline optimization permits to perform memory- and time-intense optimization which a JIT compiler cannot because of its limited resources and stricter runtime requirements.

In a next step, the LLVM WebAssembly backend transforms the optimized LLVM IR to a WebAssembly *module* in text format. A WebAssembly module is the counterpart of an object file for a natively compiled application and is either represented in human readable text format or compact binary format. This step uses the text format, as it is the only currently supported format of the LLVM backend. Part of this step is the transformation of the used register to local variables because WebAssembly is defined as a stack machine. The LLVM backend reduces the number of local variables by using *register coloring* before the transformation. Finally, the WebAssembly module is stored in a temporary file (.s) using the text format.

Binaryen The Binaryen toolchain [3] is the compiler backend. It performs WebAssembly specific optimizations in the same way as a traditional compiler backend for the target architecture. Furthermore, it transcodes the WebAssembly module from text format to the binary format. The result of this step is the final, binary encoded WebAssembly module (.wasm) containing the Speedy.js functions.

TypeScript Transpiler The TypeScript transpiler is part of the right compilation pipeline and hooks into the TypeScript emitter generating the transpiled JavaScript output. It generates the glue code needed to call Speedy.js functions transparently from regular TypeScript code. Listing 2 shows an excerpt of the transpiled output for the `isPrime` example.[5] The transpiler includes the Speedy.js runtime environment (line 1) and parameterizes it with the path to the WebAssembly module and the compilation options affecting the runtime behavior (line 6). The runtime environment provides commonly used functions for integrating Speedy.js into a JavaScript environment, e.g. loading the WebAssembly module from disk or remote.

The transpiler further replaces the Speedy.js entry functions with stubs (the `isPrime` function on line 13). The stub loads the WebAssembly module by using the functionality of the runtime environment (line 14) and forwards the call to the implementation in the WebAssembly module as soon as the module is loaded (line 17). The stub also adds the necessary casts and serializations for values passed as call arguments, and values returned by Speedy.js entry functions to the stub implementation (line 18, casts the result to a boolean). The stubs further calls the garbage collector before returning the result to release all heap allocated memory (line 19). The Transpiler omits non-entry Speedy.js functions from the JavaScript output as these are only callable from Speedy.js functions. On the contrary, Speedy.js instance methods are not omitted from the output as class instances can be passed between Speedy.js and TypeScript. Therefore, instance methods can be invoked from Speedy.js and regular TypeScript as well. The output of the transpiler is a JavaScript file that can be executed in the browser or using Node.js.

```
1  function __moduleLoader(wasmUri, options) {
2    // ... Speedy.js runtime environment
3    return loader;
4  }
5
6  var loadWasmModule_1 = __moduleLoader(
7    "./isPrime-spdy.wasm",
8    {
9      // ... runtime parameter such as stack and
           initial heap size
10   }
11 );
12
13 function isPrime(value) {
14   var loaded = loadWasmModule_1();
15   return loaded.then(function(instance_1) {
16     var exports = instance_1.exports;
17     var result_1 = exports._isPrime(value);
18     var casted = result_1 === 1;
19     loadWasmModule_1.gc();
20     return casted;
21   });
22 }
23
24 async function main() {
25   const prime = await isPrime(10000);
26   console.log("isPrime 10000: " + prime);
27 }
```

Listing 2. Excerpt from the transpiled JavaScript output for the `isPrime` example

4 Runtime System

The Speedy.js runtime system consists of a *runtime library* implementing the built-in JavaScript objects and a *runtime environment* seamlessly embedding Speedy.js into TypeScript applications.

Runtime Library The runtime library implements the essential built-in types and global objects of the JavaScript standard. Most notably, it implements dynamically sized arrays, the `Math` object, and operations for type coercion. The

[4]The Speedy.js runtime library implements the JavaScript built-in types.
[5]The listing omits the Speedy.js runtime environment, which embeds Speedy.js into TypeScript applications, for brevity. Furthermore, some temporary variables have been introduced so that the code fits nicely into a single column.

library is realized in C++, with a focus on compactness and efficiency rather than completeness. Supporting all TypeScript features is unreasonable since some, like `eval` or Proxies, require a JIT compiler to be efficiently realized.

The runtime library implements the memory free routine that releases all allocated heap objects at the end of the exit of each top-most Speedy.js function. Implementing a full garbage collector is not intended since a WebAssembly proposal exists for exposing the JavaScript engine's garbage collectors [7, 10, 21, 27]. Besides, such a collector significantly increases the size of the runtime library and needs to be environment-agnostic to determine the right time for performing a collection. For example, in a browser, it is desired to run the collection in between the rendering of two frames and while the UI-thread is idle. In a WebWorker or for Node.js, other criteria apply. Therefore, Speedy.js will use the proposed garbage collector of the host environment as soon as the standard is available.

The runtime library is compiled using Emscripten[6] [31] and is statically linked against the user program at compile time.

Runtime Environment The runtime environment integrates Speedy.js into TypeScript applications which incorporates: the loading and instantiating of the WebAssembly module, providing and managing the memory of the Speedy.js application, and serializing values passed across language boundaries. The runtime environment loads the WebAssembly module for the first called Speedy.js function and instantiates it. Like a process, each WebAssembly *instance* possesses a linear memory [10]. The runtime environment creates the linear memory during the initialization of the WebAssembly module and enlarges it whenever the Speedy.js application runs out of heap memory. The linear memory is further accessed by the runtime environment to pass objects and arrays between JavaScript and Speedy.js since this is not yet natively supported by the WebAssembly standard. The serialization logic for objects is as follows: The runtime environment allocates a new object on the heap of the WebAssembly instance if a JavaScript object is passed as an argument to a Speedy.js function. It recursively copies the values of the JavaScript object to the allocated Speedy.js object. Finally, the pointer to the heap allocated object is passed to Speedy.js. The inverse logic applies for object pointers returned by Speedy.js functions. In this case, the runtime environment creates a new JavaScript object and recursively fills it with the values of the Speedy.js object. The implementation ensures that the created object is an instance of the corresponding JavaScript class. The (de-)serialization logic applies equally to arrays.

[6]Emscripten is a C(++) to WebAssembly compiler.

Table 1. Description of the performance evaluation test cases

Case	Description
arrayReverse	Computes the reverse of an array with 10'000 elements 999 times.
fib	Computes the Fibonacci number of 40.
isPrime	Tests if the prime number $2^{31} - 1$ is prime.
mergeSort	Sorts an array of 10'000 double elements using the merge sort algorithm.
nsieve	Counts the prime numbers in the range of 2 to 39'999 using the sieve of Eratosthenes.
simjs	Generates one thousand random numbers using the SIM.js library [24].
tspDouble	Nearest neighbor based traveling salesman problem solver for the test data of Tanzania [23].
tspInt	The same case as tspDouble but uses integer instead of double coordinates.

5 Evaluation

This section evaluates the performance of Speedy.js based implementations relative to regular TypeScript. The performance is evaluated by benchmarking the test cases listed in the table 1.

The tests have first been implemented using regular TypeScript[7] and were then ported to Speedy.js by only replacing unsupported language features, but without performing any optimizations. This process showed that almost no alternations to the TypeScript implementations were necessary. The most notable modification was the refactoring of the SIM.js library [24] to use classes instead of prototypes.

Figure 2a shows the results of the benchmark when running in Firefox.[8] Each bar represents the relative performance of Speedy.js compared to the regular TypeScript implementation in percent where a value greater than 100% indicates a speedup of the Speedy.js based implementation and a value smaller than 100% a slowdown.[9] These results show that using Speedy.js improves the runtime performance and can even lead to a speedup as high as a factor of four. The performance of the *fib*, *simjs*, and *tspInt* tests improve

[7]The tests are transpiled to ECMAScript 5 to ensure that all JavaScript engines have good support for optimizing the used language features.

[8]Experiments were performed on a MacBook Pro Mid 2014 with a 3 GHz Intel Core i7 processor and 16 GB RAM running MacOS Sierra. The used browser versions are Firefox 53.0 and Chrome 59.0.3071.86. For Speedy.js implementations, the overall runtime is measured including the invocation of the Speedy.js implementation from JavaScript code, the runtime of the Speedy.js implementation and the returning of the result to the JavaScript code.

[9]The absolute figures of the benchmark are available at https://goo.gl/tBPm7t.

because WebAssembly has distinct operations for integer computations whereas TypeScript only has a single numeric data type. The performance of the *arrayReverse*, *mergeSort*, and *nsieve* tests improve mainly because of the omitted array boundary checks. This relaxation of the safety constraints simplifies the control flow of the generated code and, as a result thereof, facilitates function inlining which enables even further optimization. The runtimes of the *isPrime* and *tspDouble* tests are scarcely improved by using Speedy.js because the Firefox JIT compiler is sophisticated enough to determine that the numeric operations of the *isPrime* test always have integer results, and for the *tspDouble* test, that all array boundary checks can be eliminated. The runtimes of the test cases are nearly constant and, in average, the margin of errors are almost identical between the TypeScript (2.2%) and Speedy.js (1.9%) implementations.

Figure 2b shows the results of the same tests run on Chrome. The results of the *arrayReverse*, *fib*, *isPrime* and *nsieve* tests remarkably differ from those of Firefox. There are three possible causes for these discrepancies:

1. The runtimes of the TypeScript based implementation — used as baseline — vary between the two browsers because the JIT optimizations are distinct for each browser. This is the cause for the discrepancies of the *isPrime* and *nsieve* tests. The runtimes of the TypeScript implementations differ by 80%, respectively 65% across the tested browsers.
2. The runtimes of the Speedy.js based implementation differ between the tested browsers as it is the case for the *fib* test. The runtimes of the TypeScript implementation are almost identical on Chrome an Firefox but the runtimes of the Speedy.js implementation differ by 22%. This difference is caused by the fact that web browsers use the JavaScript engine as WebAssembly VM. Therefore, the Speedy.js performance depends upon how well a browser's JIT compiler supports WebAssembly [10].
3. Neither are the results for the TypeScript based implementation nor the one using Speedy.js consistent across different browsers. This applies for the *arrayReverse* test (TypeScript 81% difference, Speedy.js 62% difference).

In average, Speedy.js based implementations have a more consistent performance across browsers (17% difference) than the ones using regular TypeScript (31% difference). The performance of Speedy.js should become more consistent since it can be assumed that the browsers are improving their WebAssembly support [32].

To sum up, the results evidently show that the use of Speedy.js significantly improves the performance of TypeScript applications while only requiring slight changes to the code. The speedup depends on the actual use case and might, in some cases, result in a slowdown. In these cases, a more consistent performance across different browsers can still outweigh the slowdown observed in another browser.

6 Related Work

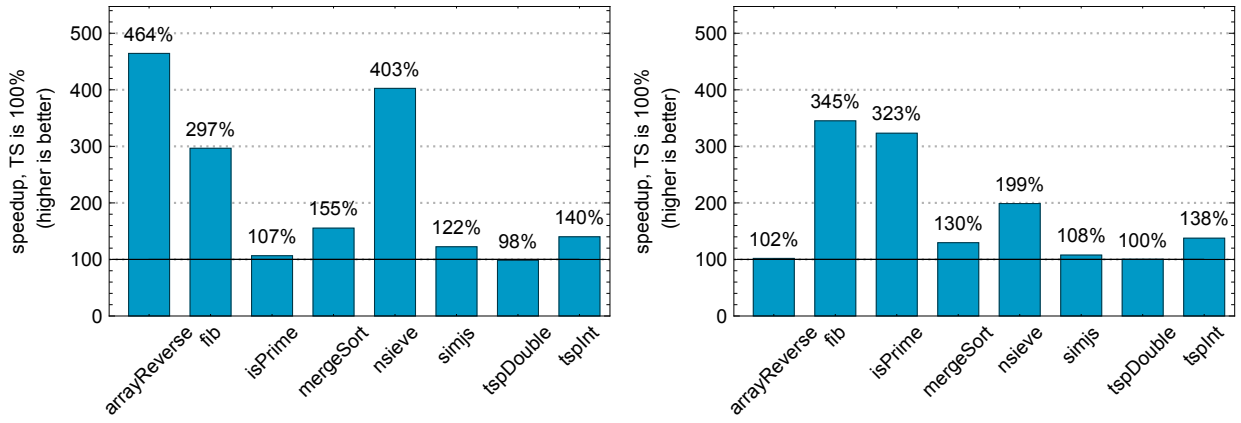
Although WebAssembly is a relatively new technology, a growing number of compilers supporting WebAssembly as compilation target exist [1, 4, 22, 29, 30]. These compilers — like Speedy.js — use WebAssembly as compile target to benefit from the ubiquitous presence of the browser platform, the near-native, and predictable performance that WebAssembly offers. The main difference lies in the use case. Speedy.js targets JavaScript projects written in TypeScript that intend to accelerate performance-critical code by compiling to WebAssembly. On the contrary, the existing compilers aim for projects where most of the application is written in the specific source language, and JavaScript is only used where it is needed to embed the application in the browser environment or to provide a JavaScript API (e.g. for a library).

StrongMode [19], or SaneScript [15, 18], is a proposal from Google that defines a more restrictive subset of ECMAScript 6 with the intent to improve performance and performance predictability by removing behaviors, which are common performance or correctness pitfalls. StrongMode is selectively enabled for specific functions or classes. Thus, StrongMode follows the same idea as Speedy.js — improving the performance of web applications — by allowing an optional opt-in. The main difference is that Speedy.js goes one step further and introduces some acceptable relaxations of safety constraints of JavaScript to bring out the best possible performance. A second difference is that StrongMode is designed to be part of the ECMAScript standard and, therefore, runs in the JavaScript engine, whereas Speedy.js compiles to WebAssembly and uses a custom runtime implementation. This difference is rooted in the implementation decision of Speedy.js to use WebAssembly to achieve a cross-platform implementation in a relatively short time. Google stopped the work on StrongMode at the beginning of 2016 [20] mainly because ECMAScript 6 — on which StrongMode is based — still performs considerably slower than ECMAScript 5 [8].

LLJS [5] is a typed dialect of JavaScript that offers a C-like type system and manual memory management. It allows writing efficient, GC pause-free applications that are compiled to JavaScript. Speedy.js, on the contrary, provides automatic memory management to be more familiar to TypeScript developers allowing easier adoption. Furthermore, LLJS compiles to JavaScript whereas Speedy.js uses WebAssembly as the compilation target.

7 Conclusion

The importance of JavaScript steadily increased since the web became a ubiquitous application platform. However, it's performance is insufficient for several compute-intense applications, even though the JavaScript engines remarkably



(a) Firefox **(b)** Chrome

Figure 2. Speedups of the Speedy.js implementations compared to regular TypeScript implementations per browser.

improved in recent years. This paper presented Speedy.js, a selected subset of JavaScript/TypeScript designed to accelerate performance-critical functions. It seamlessly integrates into regular TypeScript code and the emitted JavaScript can be used by any plain JavaScript application too. To achieve the highest possible performance, we relax certain memory safety conditions in JavaScript and require the programmer's discipline not to violate them. As our cross compiler supports most of the central JavaScript language features, it is often sufficient to only add the `"use speedyjs"` directive to enable the Speedy acceleration. Since Speedy.js compiles to WebAssembly, it runs in all major browsers as well as in Node.js.

The evaluation shows that compiling compute-intense operations with Speedy.js results in a remarkable speedup for most test cases — sometimes up to a factor of four. It's cross-browser performance exhibits less fluctuations than in ordinary JavaScript engine execution. It can be expected that the performance gets even more predictive with the improving WebAssembly implementation by the web browsers.

Speedy.js shows that the performance for compute-intense web code can be significantly increased by only imposing minimal language restrictions and conventions. Therefore, it might be worth considering to natively support a typed, performance-optimized subset of JavaScript with relaxed safety guarantees in the browser to allow to speed up specific application portions when needed.

8 Availability

The source code of the runtime and the compiler are available on GitHub [16] under the MIT license.

References

[1] 01Alchemist. 2017. TurboScript. (2017). https://goo.gl/9uHnL4

[2] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[3] Zakai Alon. 2017. Binaryen. (2017). https://goo.gl/4Pk5Kw

[4] Alexey Andreev. 2017. TeaVM. (2017). https://goo.gl/sDMWyV

[5] Michael Bebenita. 2017. LLJS. (2017). https://goo.gl/9kIua4

[6] Lin Clark. 2017. What makes WebAssembly fast? (2017). https://goo.gl/20JjFq

[7] Lin Clark. 2017. Where is WebAssembly now and what's next? (2017). https://goo.gl/M7nq91

[8] Kevin Decker. 2017. Six Speed. (2017). https://goo.gl/b1Kg7b

[9] Katelyn Gadd. 2017. ilwasm. (2017). https://goo.gl/Z4AeGT

[10] Andreas Haas, Andreas Rossberg, Derek Schuff, Ben Titzer, Dan Gohman, Luke Wagner, Alon Zakai, JF Bastien, Michael Holman, and Google Mozilla. 2017. Bringing the Web up to Speed with WebAssembly. *PLDI17* (2017).

[11] David Herman, Wagner Luke, and Alon Zakai. 2014. asm.js. (2014). https://goo.gl/sxWVss

[12] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04).* IEEE Computer Society, Washington, DC, USA, 75—-.

[13] LLVM Project. 2017. LLVM Tutorial. (2017). https://goo.gl/DCSU2v

[14] Mozilla. 2017. ARE WE FAST YET? (2017). https://arewefastyet.com

[15] Axel Rauschmayer. 2015. Google SoundScript: faster OOP for JavaScript. (2015). https://goo.gl/mZaFSJ

[16] Micha Reiser. 2017. Speedy.js. (2017). https://goo.gl/kbzKLn

[17] Micha Reiser. 2017. TypeScript with Int Support. (2017). https://goo.gl/BtPPj9

[18] Andreas Rossberg. 2015. Experimental New Directions for JavaScript. (2015). https://goo.gl/YQknPp

[19] Andreas Rossberg. 2015. Strong Mode Proposal. (2015). https://goo.gl/wOb9ry

[20] Andreas Rossberg. 2016. An update on strong mode. (2016). https://goo.gl/1l2eIz

[21] Andreas Rossberg. 2017. GC Extension. (2017). https://goo.gl/Jce6eu

[22] The Rust Core Team. 2016. Announcing Rust 1.14 - The Rust Programming Language Blog. (2016). https://goo.gl/UqCp72

[23] University of Waterloo. 2017. National Traveling Salesman Problems. (2017). https://goo.gl/D6VULu

[24] Maneesh Varshney. 2011. SIM.JS | Discrete Event Simulation in JavaScript. (2011). http://www.simjs.com/

[25] Luke Wagner. 2016. WebAssembly Browser Preview. (2016). https://goo.gl/Ar5lvG

[26] Luke Wagner. 2017. WebAssembly consensus and end of Browser Preview. (2017). https://goo.gl/cdxlGI
[27] WebAssembly Community Group. 2017. Features to add after the MVP - WebAssembly. (2017). https://goo.gl/Lo5ILA
[28] WebAssembly Community Group. 2017. WebAssembly Specification. (2017). https://webassembly.github.io/spec/
[29] Daniel Wirtz. 2017. AssemblyScript. (2017). https://goo.gl/d8oxVr
[30] Alon Zakai. 2011. Emscripten: An LLVM-to-JavaScript Compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA '11)*. ACM, New York, NY, USA, 301–312. https://doi.org/10.1145/2048147.2048224
[31] Alon Zakai. 2017. Emscripten. (2017). http://emscripten.org
[32] Alon Zakai. 2017. Why WebAssembly is Faster Than asm.js. (2017). https://goo.gl/5iTDI6
[33] Alon Zakai and Robert Nyman. 2013. Gap between asm.js and native performance gets even narrower with float32 optimizations. (2013). https://goo.gl/5PZ2jK