УДК 004.453

Ayrapetyan R.B.[1], Gavrin E.A.[2,1], Shitov A.N.[2,1]

[1] Samsung Research Center, Moscow, Russia
[2] Lomonosov Moscow State University, Moscow, Russia

A NOVEL APPROACH FOR ENHANCING PERFORMANCE OF JAVASCRIPT ENGINE FOR WEB APPLICATIONS

Abstract

JavaScript is the most widespread language for Web programming. And, literally, it is vital for Web 2.0. With development of Web 2.0, JavaScript engines experience increasingly large performance-related challenges. The ability to boost JavaScript performance becomes the crucial point for complete replacement of desktop applications in some cases. We propose the novel approach for dramatic performance improvement of Web applications by introducing snapshot of compiled code, profile and type feedback for fast startup and ahead-of-time optimizations.

Keywords

JavaScript, Just-in-Time Compilation, Compiler, Virtual Machine, JavaScript Engine, Hybrid Compiler.

Айрапетян Р.Б.[1], Гаврин Е.А.[2,1], Шитов А.Н.[2,1]

[1] Исследовательский Центр Самсунг, г. Москва, Россия
[2] Московский государственный университет имени М.В. Ломоносова, г. Москва, Россия

НОВЫЙ ПОДХОД ДЛЯ ПОВЫШЕНИЯ ПРОИЗВОДИТЕЛЬНОСТИ ВИРТУАЛЬНОЙ МАШИНЫ ЯЗЫКА JAVASCRIPT ВЕБ-ПРИЛОЖЕНИЙ

Аннотация

JavaScript является наиболее распространенным языком для веб-программирования. Виртуальные машины для JavaScript испытывают серьезные проблемы, связанные с производительностью при исполнении вычислительно сложные приложений. Возможность повышения производительности JavaScript становится критической и мы предлагаем новый подход для существенного улучшения производительности веб-приложений.

Ключевые слова

JavaScript; компилятор; динамическая компиляция; виртуальная машина; движок JavaScript; гибридный компилятор.

Introduction

Web applications usually are simple client-server software applications with client running in a web browser, and the server side, which is remote, taking over most of heavy computations and management of data storage. Nowadays, usual web applications are e-mail clients, instant messengers, news feeds, maps, or simple games. The main characteristic of the existing applications is the computational simplicity of client part. As we suppose, this is a direct consequence of JavaScript engines slowness. By historical reasons, JavaScript is the only natively supported language on the Web [1]. However, it seems that performance of JavaScript is still not enough for efficient execution of complex software algorithms, which is absolute requirement for modern applications – starting from 3D games and to complex analytics tools.

Previously, the performance challenge was attempted to be overcome by several technologies starting from ActiveX to Native Client, asm.js and WebAssembly [1]. While some of these technologies are either not secure enough or don't provide sufficient performance, other require using another programming languages, which are not as widespread as JavaScript [1]. See table 1 for comparison of their particular features.

We propose to look at the challenge from another view point. JavaScript is dynamically-typed language, which is designed to maximize development flexibility [2]. There are several existing fine-tuned approaches for profiling and analysis of dynamically-typed languages – type feedback, type inference [3] and inline caching [4]. However, these approaches don't practically allow to analyze the executed program, globally. Under this conditions, the uncertainty derived from dynamic nature of JavaScript, becomes a significant obstacle for compiler optimizations, and consequently the core of the performance challenge.

We propose to store and update dynamic profiling data of Web Applications as a whole during their execution, for ahead-of-time analysis followed by optimized compilation. The stored data allows to fully analyze an application's code, type information, etc., and so to figure out its semantics with minimal uncertainty. And ahead-of-time compilation allows to perform comprehensive semantics analysis without introducing delays into execution of applications. These features, finally, lead to the best opportunities for performance optimizations.

There are Java, Python and JavaScript runtimes, which partially implement similar ideas. Android Runtime uses profile-guided AOT compilation [9], Pyston implements application code cache with plans to implement AOT recompilation with higher optimization levels [10], and V8 caches information for fast recreation of compiled code [11]. However, our approach is completely novel, as it involves comprehensive global semantics analysis, leading to minimization of dynamic uncertainty.

High-Performance JavaScript engine

As a highly effective illustration to the proposed idea, we developed the prototype of High-Performance JavaScript Engine (see Figure 1 for overall design).

The main components of the developed engine, which is based on JerryScript [8], are execution manager, semantics analyzer, type information manager, persistent cache of code and execution profile, JS parser, and optimizing compiler (see Fig. 2). In overall, execution process is managed by execution engine, which handles an executed application's source code, loads the available information about the application from persistent cache, and starts with most effective way of application execution, which is available according to the cached data. Possible ways are generic interpretation, optimized interpretation backed by inline caches, fast-compiled region invocation, and finally execution of fully optimized AOT-compiled region representing the whole program.

Table 1. Comparison of the existing technologies attempting to solve performance challenge for Web Applications

Technology	Security	Performance	Language
Asm.js	Safe	Slow	JavaScript
WebAssembly (wasm)	Safe	Fast	C/C++
Native Client (PNaCl)	Safe	Fast	C/C++
ActiveX	Unsafe	Fast	C/C++/VB

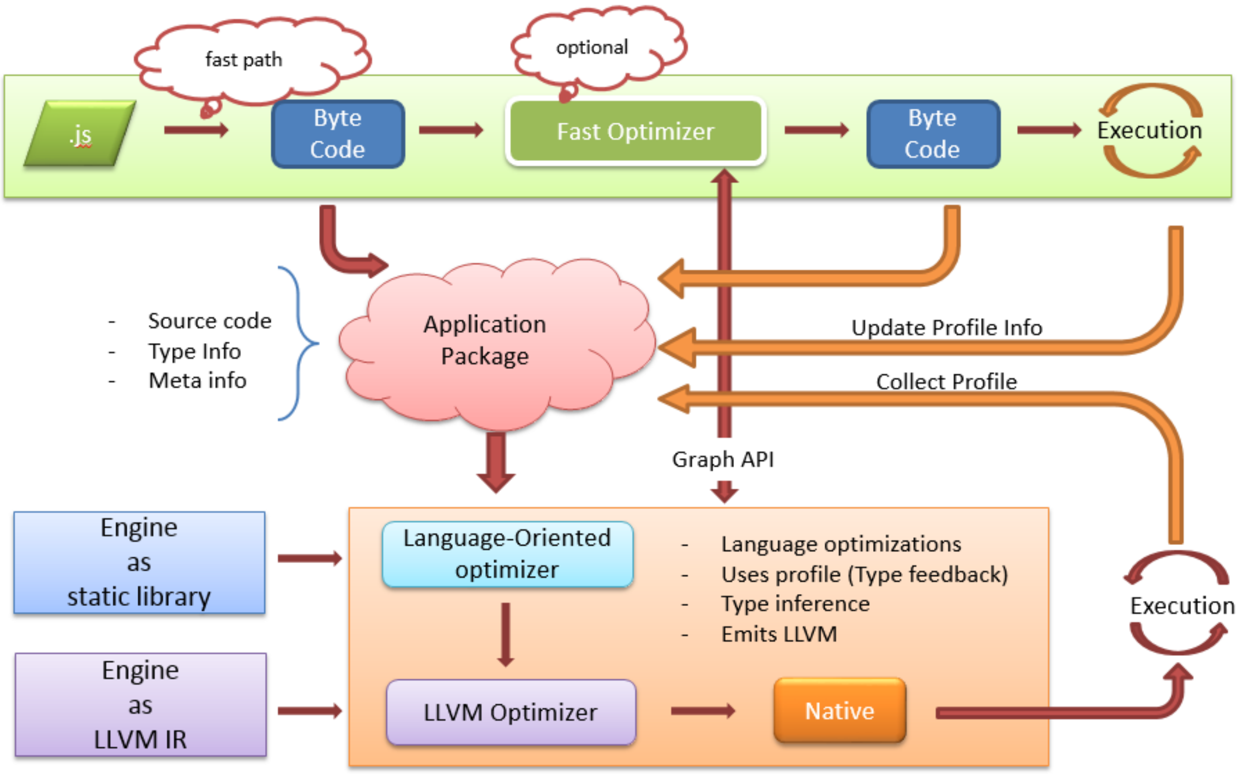

Figure 1. Overall design of High-Performance JavaScript Engine

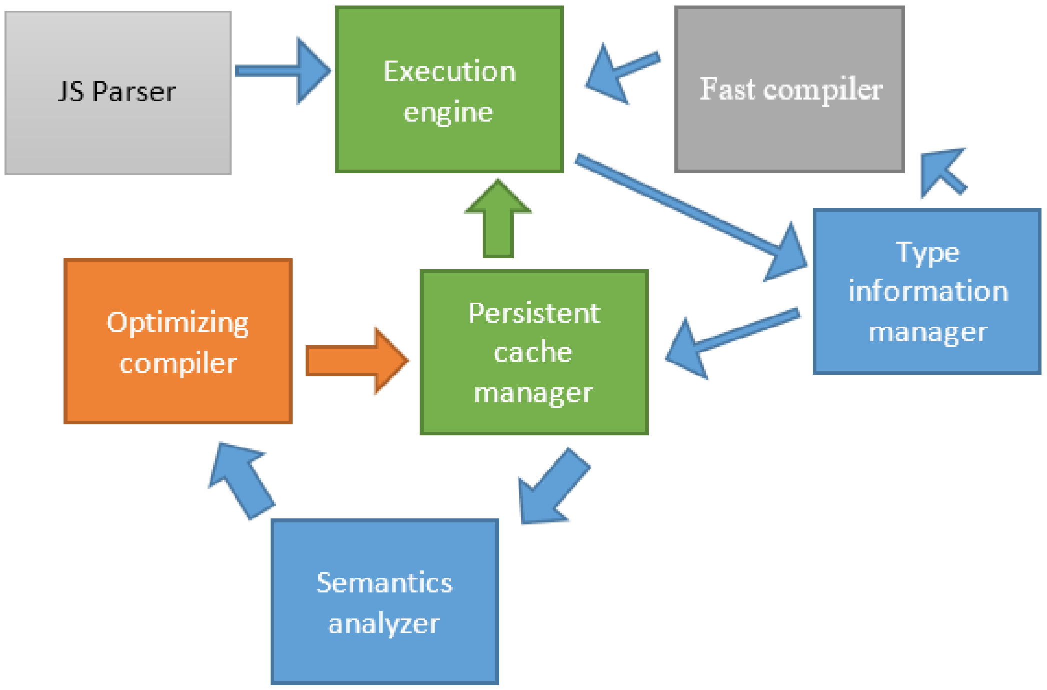

Figure 2. Architecture of High-Performance JavaScript Engine

JavaScript Parser

Initial analysis of input JavaScript program is performed by parser, which operates on modular level, i.e. considers one JS source file per invocation. Parsing of the file is performed in two passes – the first detects structure of the source and figures out what JS features are used there – "with" blocks, closures, direct-mode "eval", and the second generates actual byte-code. The information about used JS features is used as runtime hints for using fast-paths afterwards.

The produced byte-code is high-level representation of parsed JS file, constructed without any assumptions about dynamic behavior of the program. Each instruction is opcode with 3 optional arguments, and each of the argument either extends the opcode, or designates an identifier, virtual VM register, or constant value.

Type Information

Type feedback, essentially, represents statistics of arguments types for each byte-code instruction of an application. Type here means basic JavaScript type (number, string, object, etc.) and extended information. The extended information consists of details, which are specific for each of the basic types:

- numbers can be integer or floating point, signed or unsigned
- strings can be ASCII-only or UTF-16, one of specification-defined or arbitrary
- objects can have different field layout

Details of object layouts are packed into application-wide identifiers (which can also be referred as "hidden classes"), which are then encoded into type feedback.

Execution Engine

The execution engine is the central node of the developed JavaScript engine's architecture. The component manages execution of an application, by navigating through byte-code. For each part of application, it lookups available compiled regions, and chooses way of execution – either invocation of the regions, or of interpreter.

Upon load of an application, execution engine requests information about compiled regions, which are available for the application, from persistent cache. All the available regions are registered in code manager, to be loaded on-demand.

Execution of a part, for which no compiled region is available up to a moment, is performed by interpreter. Interpreter processes byte-code instructions one-by-one and optimizes its execution by inline caching [4]. Inline caches replace the generic implementation of an instruction's semantics by the most specific, considering the collected type feedback. If dynamic behavior of the program changes, and so the specialized implementation is invoked with a type set, which can't be handled by the specific implementation, then it is either switched to another specific (so called "monomorphic") case of implementation for the new type set, or to a less specific ("polymorphic") case, which can handle the both type sets. The monomorphic cases are faster to execute, and could require more switching in case of frequent changes of dynamic behavior, in comparison to polymorphic cases. So, the choice should be performed according to profile information.

During interpretation, execution engine collects type feedback and profile for each of executed parts, and provides it to type information manager. After type feedback is collected, execution engine invokes JIT, to compile “hot” loops and frequently-invoked functions. The dynamic JIT compiler should be fast enough, so it performs only local optimizations, in comparison to fully-optimizing ahead-of-time compiler. After JIT compilation is finished, the produced compiled regions are registered in code manager, and become available to the execution engine.

All the produced compiled regions, type feedback and profiler information are mirrored to persistent cache, for usage during further executions, and for AOT compilation.

As a fallback to adapt to changes in an application’s dynamic behavior, each compiled region has information, which is necessary for rollbacks to more generic, and so less optimized, implementations of corresponding application parts. After a rollback, profile information is updated according to the newly discovered behavior, to be used in future optimizations – both JIT and AOT.

Ahead-of-Time Compiler

The developed AOT compiler is the essential part of the overall optimization pipeline in our approach. In general, the compiler loads all the existing information about a particular application from engine’s persistent cache, and produces highly optimized representation of the application based on the loaded information.

Among other compiler optimizations, the compiler performs type specialization, type check elimination, automatic fixed-sized array detection [v8], global dead-code elimination, elision of concatenation, etc.

One of optimizations, which deserves special attention, is partially dynamic by its nature – the dynamic trace-based analysis. Core of the optimization on compiler side is search of compiled region parts that consist of constant expressions, pure or independent computations - the features are figured out through global analysis. The corresponding region parts are additionally compiled in a way, which allows to partially postpone their execution by recording traces of with their input values and identifiers. After the recording is finished, the trace is analyzed to figure out which of computations are really necessary, and which are, for example, duplicate, by means of checking data dependencies figured out during compilation. Afterwards, the recording is replayed without execution of unnecessary parts. This way of processing dynamic traces is actually a dynamic dead-code elimination. As possible future development of the idea, we consider automatic non-speculative parallelization of JavaScript code based on dependencies information from compiler.

Performance and memory consumption

For estimation of the developed engine’s performance we have chosen the LongSpider benchmark [6]. This benchmark is derived from the more widely known SunSpider [5], with the main difference – number of loop iterations, which are greater in LongSpider.

Currently, our engine executes LongSpider benchmark by 40% faster (geometric mean) than V8 [7] on the second execution, taking advantage of ahead-of-time semantics analysis, which is performed using profile from the first execution. And memory consumption is 25% lower (geometric mean) on the benchmark. See table 2 and 3 for detailed measurements.

Table 2. Comparison of engine’s performance on Raspberry Pi 2 board

Benchmark	Google V8 (seconds)	High-Performance JavaScript Engine (seconds)
3d-cube.js	10.98	2.89
3d-morph.js	20.34	22.36
3d-raytrace.js	16.59	0.62
access-binary-trees.js	17.5	22.1
access-fannkuch.js	6.25	13.21
access-nbody.js	16.03	31.26
access-nsieve.js	6.1	7.92
bitops-3bit-bits-in-byte.js	0.66	0.6
bitops-bits-in-byte.js	2.11	1.47
bitops-nsieve-bits.js	16.52	19.22
controlflow-recursive.js	10.99	15.1
crypto-aes.js	17.08	0.37
crypto-md5.js	27.21	21.14
crypto-sha1.js	33.27	30.01
math-cordic.js	23.53	2.63
math-partial-sums.js	24.03	14.77
math-spectral-norm.js	13.61	39.37
string-fasta.js	23.84	11.73

Table 3. Comparison of engine’s memory consumption on ARM32

Benchmark	Google V8 (kilobytes)	High-Performance JavaScript Engine (kilobytes)
3d-cube.js	11736	56068
3d-morph.js	11060	8680
3d-raytrace.js	26020	19088
access-binary-trees.js	30424	116864
access-fannkuch.js	6564	4160

access-nbody.js	7308	4604
access-nsieve.js	86316	83864
bitops-3bit-bits-in-byte.js	6132	3972
bitops-bits-in-byte.js	6128	3940
bitops-nsieve-bits.js	18904	17172
controlflow-recursive.js	6304	4388
crypto-aes.js	15936	8936
crypto-md5.js	40732	7860
crypto-sha1.js	81824	62988
math-cordic.js	8480	4192
math-partial-sums.js	9416	4216
math-spectral-norm.js	7904	4160
string-fasta.js	8916	4532

ECMA-262 Compliance

The developed engine is 100% compliant with ECMA-262, according to test262 – the Official ECMAScript Conformance Test Suite.

Conclusion

The developed engine, while being an early prototype, represents highly effective illustration to the proposed technology, which is based on ahead-of-time analysis of full-application profile. Deep analysis provides way to figure out all the necessary application semantics from its high-level source and execution profile, which gives the best possible opportunities for optimizations of JavaScript-based Web Applications.

The developed prototype is faster than V8 JavaScript engine by 40% on LongSpider benchmark, with 25% lower achieved performance results demonstrate that the proposed technology opens a real way for efficient execution of complex fully-functional Web Applications with, under certain conditions, near-native performance.

As future directions, we consider research of dynamic trace-based optimizations. From our view point, the way of dynamic representation of application's execution line, can be developed to make possible automatic non-speculative parallelization of JavaScript applications.

References

1. A., Hass; A, Rossberg; D.L., Schuff; B.L., Titzer; D., Gohman; L., Wagner; A., Zakai; M., Holman; JF, Bastien (2017). Bringing the Web up to Speed with WebAssembly..
2. Richards, G., Lebresne, S., Burg, B., & Vitek, J. (2010, June). An analysis of the dynamic behavior of JavaScript programs. In ACM Sigplan Notices (Vol. 45, No. 6, pp. 1-12). ACM.
3. Hackett, B., & Guo, S. (2012). Fast and precise hybrid type inference for JavaScript. ACM SIGPLAN Notices, 47(6), 239-250.
4. Brunthaler, S. (2010, June). Inline caching meets quickening. In European Conference on Object-Oriented Programming (pp. 429-451). Springer Berlin Heidelberg.
5. WebKit / LongSpider, 2016. https://github.com/WebKit/webkit/tree/master/PerformanceTests/LongSpider.
6. WebKit. SunSpider JavaScript Benchmark, 2017. https://webkit.org/perf/sunspider/sunspider.html.
7. Chrome V8. https://developers.google.com/v8.
8. Gavrin, E., Lee, S. J., Ayrapetyan, R., & Shitov, A. (2015, October). Ultra-lightweight JavaScript engine for internet of things. In Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity (pp. 19-20). ACM.
9. Implementing ART Just-In-Time (JIT) Compiler, 2017. https://source.android.com/devices/tech/dalvik/jit-compiler.
10. The Pyston Blog. Caching object code, 2016. https://blog.pyston.org/2015/07/14/caching-object-code.
11. V8 JavaScript Engine. Code caching, 2015. https://v8project.blogspot.com/2015/07/code-caching.html.

Поступила: 15.05.2017

Об авторах:

Айрапетян Рубен Борисович, ведущий инженер-программист отдела компиляции, Исследовательский Центр Самсунг, cv.ru@samsung.com

Гаврин Евгений Александрович, аспирант факультета вычислительной математики и кибернетики, Московский государственный университет имени М.В. Ломоносова; руководитель отдела компиляции, Исследовательский Центр Самсунг, eugene.a.gavrin@gmail.com

Шитов Андрей Николаевич, аспирант факультета вычислительной математики и кибернетики, Московский государственный университет имени М.В. Ломоносова; ведущий инженер-программист отдела компиляции, Исследовательский Центр Самсунг, sand1k@yandex.ru

Note on the authors:

Ayrapetyan Ruben, leading engineer-programmer of Compilation Department, Samsung Research Center, cv.ru@samsung.com

Gavrin Evgeny, Postgraduate Student of Faculty of Computational Mathematics and Cybernetics, Lomonosov Moscow State University; Head of Compilation Department, Samsung Research Center, eugene.a.gavrin@gmail.com

Shitov Andrey, Postgraduate Student of Faculty of Computational Mathematics and Cybernetics, Lomonosov Moscow State University; engineer-programmer of Compilation Department, Samsung Research Center, sand1k@yandex.ru