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

THE HYBRID COMPILER TARGETING JAVASCRIPT LANGUAGE

Abstract

The article is devoted to the prototyping of a new JavaScript virtual machine. The work is based on the Tizen platform, which uses HTML5 and JavaScript for application development. The performance and memory consumption of JavaScript programs on existing machines is worse compared to C ++ or C # applications. In our work, we tried to reduce this gap between JavaScript and other languages.

Keywords

JavaScript; virtual machine; just-in-time compilation; optimization.

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

ГИБРИДНЫЙ КОМПИЛЯТОР ДЛЯ ДИНАМИЧЕСКОГО ЯЗЫКА ПРОГРАММИРОВАНИЯ JAVASCRIPT

Аннотация

Статья посвящена созданию прототипа новой виртуальной машины языка JavaScript. Работа основана на платформе Tizen, которая использует HTML5 и JavaScript для разработки приложений. Производительность и потребление памяти JavaScript-программ на существующих машинах хуже по сравнению с C++ или C# приложениями. В нашей работе мы попытались уменьшить разрыв по производительности между JavaScript и другими языками.

Ключевые слова

JavaScript; виртуальная машина; динамическая компиляция; оптимизация.

Introduction

In this paper we cover our work dedicated to creating a prototype of novel JavaScript execution engine. The work is inspired by the Tizen platform, which announces HTML5 and JavaScript as the main approach for developing applications. The performance and memory consumption of JavaScript-based programs on existing engines is worse comparing to native or Java applications. In our work we tried to reduce this gap between JavaScript and other languages.

Common architecture of JavaScript engine is a virtual machine, which consists of several execution layers. First one is fast interpreter or fast code generator, second and next are more heavyweight and better optimizing compilers. Modern JavaScript engines, like Google's v8 [1], Apple's JavaScriptCore [2] and Mozilla's SpiderMonkey [3] have this structure. The drawback of this approach is that every run starts from parsing of JS source code and engine requires some time to compile the code and accelerate execution.

Our main idea is to apply a novel hybrid approach of virtual machines design: to mix JIT and AOT compilation. Basic way of execution is the same as in v8 and JavaScriptCore, but only for the first run. During first run we cache additional data, which improves performance of the application. This data includes profile information, including seen types, and generated native code. Moreover, we perform full compilation of the application between executions, which means that not only "hot" regions, but all JavaScript code will be highly optimized. After revealing new optimization opportunities application becomes more and more optimized. At some point, for example if the whole application was compiled, it is possible to disable incremental updates of cached information and just execute the native binary.

We implemented a prototype of a hybrid JavaScript engine. In comparison to v8 engine on LongSpider [4] benchmark (extended version of SunSpider [5] benchmark) it shows 40% performance and 25% memory improvement, which proves that hybrid approach for running JavaScript applications is efficient.

Base design and implementation

We took the JerryScript [6] project as the base. JerryScript is an ultra-lightweight JavaScript engine for internet of things. Its main feature is low memory consumption. As our solution is targeting mobile platform, therefore low memory consumption is an important feature we tried to inherit from JerryScript solution.

JerryScript is an interpreter. It parses source code into bytecode and then executes bytecode instructions step by step. JerryScript also has a feature of saving bytecode to snapshot and starting execution from it. For us this feature is a quite early prototype of hybrid compilation.

JerryScript is targeting small devices, therefore it is not designed for high performance. So at first we implemented its core parts to reach our performance goals.

Value representation

For dynamically-typed languages there is an issue with specifying a structure, which can represent any type of the language.

A union of all possible types, tagged by object type field, can be utilized. The disadvantage of this approach is that small values (e.g., integer) are stored as packaged (boxed) values, and their access is performed via an auxiliary pointer. JerryScript has this extra indirection in values access.

High performance implementations of JavaScript language utilize tagged pointers. This means that some additional information is stored inside a pointer. This is possible due to the fact that the minimum memory size is 4 bytes and the two least significant bits of a pointer are equal to zero and can be used for storing auxiliary data. For example, all of the real pointers can be marked by setting the least significant bit, and pack integers into values with the zeroed least significant bit (31-bit integers). This approach is utilized in V8.

The minimum memory block equals to 8 bytes in JavaScriptCore and SpiderMonkey engines. Pointer, integer, double numbers as well as some additional types can be packed in such a block. This technique is called nan-boxing [7]. In High-Performance JavaScript engine we chose this representation.

Fast property access

Another important thing is to provide fast access to object’s properties and methods. In general case to access a property in object a dynamic lookup inside a table of properties is required. To optimize the lookup procedure we dynamically create *hidden classes* behind the scenes the same way as V8 does [8]. Each object has associated runtime hidden class that describes the structure of the object and the displacement of specific properties therein. This allows to make access to the object’s properties as fast as in the languages in which the object structure is predetermined and could not be changed during program execution.

We take advantage of using another commonly used technique for optimizing dynamically typed languages called inline caching [9]. High performance JavaScript Engine maintains a cache of the type of objects, which were seen during bytecode interpretation. By assuming that future operations would happen with the same types, instruction handlers are replaced with more specific, which process only those concrete types.

Dynamic machine code generation

It is impossible to achieve performance comparable to industrial engines by doing only interpretation of JavaScript code. All mature solutions generate native code on the fly. We utilize LLVM [10] as a backend and implement optimizing compiler using its JIT infrastructure.

During runtime execution hot regions are detected and compiled for those types which were seen during previous execution. Consequently, we need to generate type checks in order to verify that types seen during interpretation and types during native code execution match. Several checks are generated for each instruction, which leads to a pretty big overhead.

There is a number of circumstances, which force a fall back from native code to interpreter, for example, mismatch of types. This situation is called a bailout. The tricky thing is restoration after bailouts. In this case several compiled functions could call each other and the inner-most could cause a bailout, which would require interpreter’s frame reconstruction.

Hybrid compilation

The key concept of this project was hybrid compilation: combination of just in time and ahead of time compilation. To do this we cache additional information about some program internals:

- Source code
- Bytecode
- Type information (inline caches)
- Compiled code

In the first run we execute in a traditional way and save the specified information into the snapshot file. When the application is executed again, inline caches and native code are loaded from cache, making the execution very fast immediately after the start, removing the necessity of accumulation of the type information and detection of hot loops.

Another beautiful thing is that snapshot contains all required information to perform complete compilation of the application. This is done offline (between application executions), if offline compilation succeeds, and no bailouts occur during next run of the application, optimizing compiler will be never called, which significantly reduces memory consumption.

The overall design of the engine is show on Fig. 1.

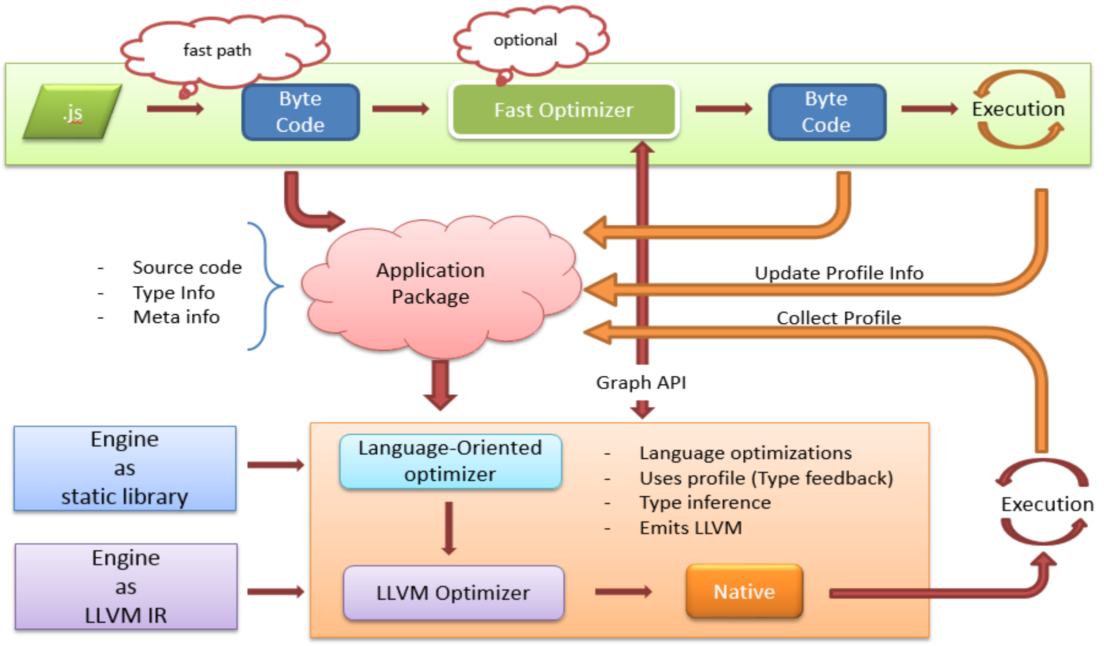

Figure 1: Overall design of High-Performance JavaScript Engine

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
Total (seconds and geometric mean)	286.64	256.77

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
Total (kilobytes and geometric mean)	390104	

This feature allowed us to perform full compilation of longspider benchmarks starting from the second run and beat v8 on longspider (see tables 2-3).

In next section we cover optimization techniques that we utilized in the project.

Optimizations

One of the big challenges is to achieve good quality of generated native code. To achieve this goal the optimizing compiler has to support several things:

- Representing temporary values as unboxed values;
- Inlining;
- Optimizations, such as global value numbering (GVN) and loop-invariant code motion (LICM).

Unboxing reduces memory pressure and permits usage of effective native machine instructions. Inlining improves performance and permits more optimizations to occur.

JIT compiler optimizations

In our project we reused optimizations, integrated into LLVM infrastructure. We have several scenarios for compilation. One is just-in-time compilation. For this case we enable only part of LLVM optimizations to make compilation faster. Another case is offline compilation. For it we turn on the maximum number of optimizations to achieve best code quality.

LLVM is a mature project which covers almost all possible optimizations for static languages, but it is not enough for JavaScript. Bluntly compiled code has JavaScript-specific semantics and contains a lot of type checks and accesses to boxed objects, which break many of LLVM optimizations. LLVM compiler has nothing to do with it.

This fact forced us to implement additional analysis layer to resolve JavaScript-specific things and generate as low-level code as possible, so that LLVM optimizations would work fine on it.

In next sections we discuss optimizations, which are implemented on the analysis layer.

Type-check elimination

As it was mentioned in the previous sections, type checks are required to provide correct execution of compiled code.

Each byte code instruction has input operands. In general case type of each operand of the instruction is checked, which gives too much overhead. Many checks could be removed, for example, if several instructions have the same input variable, and there are no writes to the variable between those instructions, one check is enough.

Optimizer's task is too remove as many redundant checks as possible.

We implemented analysis which find checks that could be avoided. Simple version works per a single basic block: it tracks all reads and writes to a variable and, when there are several consecutive reads, all the checks, except first read, are removed. More complex, per-function type-check elimination tracks reads/writes from incoming basic blocks and analyses cycles.

Fixed arrays

JavaScript arrays are pretty complicated as their size can change during execution, their elements could contain any type, they can contain holes and any object could be used as an array. This forces to have complicated runtime support for performing operations on arrays from native code.

We implemented support for arrays compilation, even if they are not used in static way: arrays are not obliged to have fixed size and uniform contents.

Profile-based dead code elimination

JavaScript applications tend to contain pretty big amount of dead code. Close look at longspider [4] benchmark revealed that its tests also could be significantly optimized by applying dead code elimination techniques.

Basic technique of dead code elimination that we used is to analyse byte code fragment and determine that output operands are never used, which means that instructions, which produce those operands, are obsolete. There are also several tricky cases that we covered.

string-fasta benchmark performs a big amount of string operations but it is clear that most of them are redundant. For instance, there is the following code:

```
ret += seq.substring(seqi, seqi+lenOut).length;
```

This fragment calculates length of a substring and the substring itself is not used. We win in this case by not doing any real operations on a string and calculating only the length.

Another case is from 3d-raytrace benchmark. It has main cycle with a call to a pair of functions inside of it:

```
testOutput = arrayToCanvasCommands(raytraceScene());
```

On every iteration the same scene is rendered. This functions itself intensively allocate objects and contain global object accesses. So they are not pure. But in fact on each iteration they access only objects which were created on the same iteration. We implemented the optimization which tracks generations of objects and determines that created and accessed objects always belong to the same iterations, which proves that in fact these functions are pure and their result value can be cached.

Trace-based compilation

This optimization is a tricky case of eliminating redundant computations. This was primarily targeted for cases, like in the access-binary-trees benchmark, which performs the following:

```
for ( var n = 4; n <= 16; n += 1 ) {
  var minDepth = 4;
  var maxDepth = Math.max(minDepth + 2, n);
  var stretchDepth = maxDepth + 1;

  var check =
bottomUpTree(0,stretchDepth).itemCheck();
  var longLivedTree = bottomUpTree(0,maxDepth);
  for (var depth=minDepth; depth<=maxDepth;
depth+=2){
    var iterations = 1 << (maxDepth - depth +
minDepth);
    check = 0;
    for (var i=1; i<=iterations; i++){
      check +=
bottomUpTree(i,depth).itemCheck();
      check += bottomUpTree(-
i,depth).itemCheck();
    }
    }
    ret += longLivedTree.itemCheck();
  }
```

This particular loop contains a big amount of calls to bottomUpTree(..).itemCheck(). Output value of this calls modify *check* variable. But in the middle of the loop there is a store of zero to the *check* variable. This store discards all previous modifications of this variable, therefore all previous calls to itemCheck() function are redundant. Straight-forward dead code elimination techniques do not work in this case.

Our approach is to break compiled region parts to blocks of independent computations and record a trace of the blocks' execution without performing actual computations with except of those, which are necessary to track control flow. The trace is optimized to remove blocks, which compute unused values, and is replayed to perform only those of actual computations, which are actually necessary.

The final trace contains only few calls to bottomUpTree(..).itemCheck() and the loop executes significantly faster.

Conclusion

In our work we implemented prototype of a hybrid JavaScript engine, which combines ahead of time and just in time compilation approaches. As far as we know, for the current moment this is the first existing hybrid implementation of a JavaScript engine. Our prototype is targeting Tizen platform, which claims Web technologies in general and JavaScript in particular, as the main way of application development. This allows us to assume that source JavaScript code is always located on a device and we can perform its optimization offline. We cache type information, bytecode and native code which provide us all required knowledge to perform fully-optimized offline compilation. By utilizing this technique, we achieve 40% performance improvement over Google's v8 engine.

Our work is an early prototype and we plan to extend optimizations that we have developed to more general cases.

References

1. Chrome V8. https://developers.google.com/v8.
2. JavaScriptCore. https://developer.apple.com/reference/javascriptcore.
3. SpiderMonkey. https://developer.mozilla.org/ru/docs/SpiderMonkey.
4. WebKit / LongSpider, 2016. https://github.com/WebKit/webkit/tree/master/PerformanceTests/LongSpider.
5. WebKit. SunSpider JavaScript Benchmark, 2017. https://webkit.org/perf/sunspider/sunspider.html
6. 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.
7. "Value representation in JavaScript implementations", https://wingolog.org/archives/2011/05/18/value-representation-in-javascript-implementations.
8. Design elements of V8 https://developers.google.com/v8/design.
9. Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming, Urs Hölzle, 163p.
10. LLVM compiler infrastructure. http://llvm.org/.
11. Hölzle, U., Chambers, C., AND Ungar, D. 1991. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In Proceedings of the ECOOP '91 Conference. Lecture Notes in Computer Science, vol. 512. Springer-Verlag, Berlin.

Поступила: 10.05.2017

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