Approaches to Optimizing V8 JavaScript Engine

Dmitry Botcharnikov <dmitry.b@samsung.com>
LLC Samsung R&D Institute Rus, 12, ul. Dvintsev, housing 1, office #1500, Moscow, 127018, Russian Federation

Abstract. JavaScript is one of the most popular programming languages in the world. Started as a simple scripting language for web browsers it now becomes language of choice for millions of engineers in the web, mobile and server-side development. However its interpretational nature doesn't always provide adequate performance. To speed up execution of JavaScript programs there were developed several optimization techniques in recent years. One example of modern high-performing JavaScript engine is a V8 engine used in Google Chrome browser and node.js web server among others. This is an open source project which implemented some advanced optimization methods including Just-in-Time compilation, Polymorphic Inline Caches, optimized recompilation of hot code regions, On Stack Replacement &c. In previous year we were involved in project of optimizing performance of V8 JavaScript engine on major benchmark suites including Octane, SunSpider and Kraken. The project was quite time limited, however we achieved about 10% total performance improvement compared to open source version. We have decided to focus on following approaches to achieve the project's goal: optimized build of V8 itself, because total running time is shared between compilation and execution; tuning of V8 runtime options which default values may not be always optimal; implementation of additional scalar optimizations. All of these approaches have made contribution to final result.

Ключевые слова: JavaScript; optimizations; V8; common subexpression eimination

DOI: 10.15514/ISPRAS-2015-27(6)-2

For citation: Botcharnikov Dmitry. Approaches to Optimizing V8 JavaScript Engine. *Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015,* pp. 21-32 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-2

1. Introduction

JavaScript is one of the most popular programming languages in the world [1]. Started as a simple scripting language for web browsers it now becomes language of choice for millions of engineers in the web, mobile and server-side development. However its interpretational nature doesn't always provide adequate performance.

To speed up execution of JavaScript programs there were developed several optimization techniques in recent years. One example of modern high-performing JavaScript engine is a V8 engine [2] used in Google Chrome browser and node.js web server among others. This is an open source project which implemented some advanced optimization methods including Just-in-Time compilation [3], Polymorphic Inline Caches [4], optimized recompilation of hot code regions, On Stack Replacement [5] &c.

In previous year we were involved in project of optimizing performance of V8 JavaScript engine on major benchmark suites including Octane [6], SunSpider [7] and Kraken [8]. The project was quite time limited, however we achieved about 10% total performance improvement compared to open source version.

The rest of paper is organized as follow: in Section 2 there is an architectural overview of V8, in Section 3 we enumerate and reason our approaches with more detailed discussion in Sections 4, 5, and 6. We conclude in Section 7.

2. V8 engine architecture

In contrast to other JavaScript engines V8 implements compilation to native code from the beginning. It consists of two JIT compilers: the first (called Full code generator) performs fast non-optimized compilation for every encountered JavaScript function, while the second one (called Crankshaft) compiles and optimizes only those functions (and loops) which already ran some amount of time and are likely to run further.

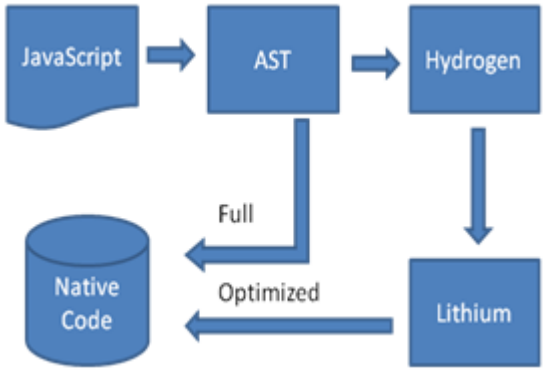


Fig. 1 V8 Engine Architecture

The overall work of V8 engine is as follows (Fig.1):

- Every new script is preliminary scanned to separate each individual function.
- The function that should run is compiled into Abstract Syntax Tree (AST) form.
- AST is compiled into native machine code instrumented with counters for

function calls and loop back edges.

- Also on method call sites V8 inserts special dispatch structure called Polymorphic Inline Cache (PIC). This cache is initialized with call to generic dispatch routine. After each invocation PIC is populated with direct call to type specific receiver up to some predefined limit. In such way PICs collect runtime type information of objects.
- The result code then runs.
- When instrumentation counters reach some predefined threshold, “hot” function or loop is selected for optimized recompilation.
- For this purpose V8 one more time recompiles selected function in AST form. But in this case it also performs optimizations.
- It compiles AST into Static Single Assignment (SSA) form (called Hydrogen) and propagates type information collected by PICs along SSA edges.
- Then it performs several optimizations on this SSA form using type information.
- After that it generates low level representation (called Lithium), does Register Allocation and generates optimized native code which then runs.

Note that V8 optimizing compiler performs much less transformational passes than common ahead-of-time compilers (e.g. gcc, clang/llvm). The reasons behind this we further discuss in Section 6.

3. Approaches to speed up V8 engine

To investigate possible areas of V8 optimization we have performed V8 engine profiling on ARM platform with three different profiling tools: Perf [9], ARM Streamline [10] and Gprof [11]. Each of those has advantages and disadvantages over others but results are very close: V8 JavaScript engine has no ‘hot’ functions in itself that need to be optimized. Different methods show different functions in order of share to total execution time. This is clear evidence that individual function’s contribution is very small compared to precision of measurement. Thus optimization of individual functions can’t achieve much increase in performance.

In following table object identified as perf-2549.map is a code generated by V8 engine.

Overhead	Shared Object	Symbol
65.11%	perf-2549.map	0x5aba4000
0.76%	d8	v8::internal::Scanner::ScanIdentifierOrKeyword()
0.75%	d8	v8::internal::IncrementalMarking::Step (int,v8::internal::IncrementalMarking::CompletionA

		ction)
0.66%	[kernel.kallsyms]	_raw_spin_unlock_irqrestore
0.65%	libc-2.17.so	memchr
0.52%	d8	v8::internal::Heap::DoScavenge(v8::internal::ObjectVisitor*, unsigned char*)
0.51%	libc-2.17.so	0x0004f3ac
0.50%	d8	int v8::internal::FlexibleBodyVisitor<v8::internal::NewSpaceScavenger, v8::internal::JSObject::BodyDescriptor, int>::VisitSpecialized<20> (v8::internal::Map*, v8::internal::HeapObject*)
0.44%	d8	void v8::internal::ScavengingVisitor<(v8::internal::MarksHandling)1, (v8::internal::LoggingAndProfiling)0>::EvacuateObject <(v8::internal::ScavengingVisitor<(v8::internal::MarksHandling)1, (v8::internal::LoggingAndProfiling)0>::ObjectContents)1, 4>(v8::internal::Map*, v8::internal::HeapObject**, v8::internal::HeapObject*, int)
0.43%	d8	v8::internal::ScavengeWeakObjectRetainer::RetainAs(v8::internal::Object*)
0.43%	d8	v8::internal::Scanner::Scan()
0.37%	[kernel.kallsyms]	__memzero

Fig. 2 Several top entries from detailed profile of Octane benchmark by V8 on Linux.

We have decided to focus on following approaches to achieve the project's goal:

- Optimized build of V8 itself, because total running time is shared between compilation and execution.
- Tuning of V8 runtime options which default values may not be always optimal.
- Implementation of additional scalar optimizations.

All of these approaches have made contribution to final result.

4. Optimized build

We have decided to investigate Link Time Optimization [12] and platform options tuning [13]. The latter gave us small outcome (~0.5%) while former have decreased performance.

We have made investigation on Arndale ARM (Samsung Exynos 5250 CPU) development board running Linux with Linaro gcc 4.7 toolchain for the first investigation and the same board running Android 4.4 with Android NDK 9 Linaro toolchain for the second one.

We have specified the following platform options:

- -O3 for highest optimization level
- -mcpu=cortex-a15 for target CPU.

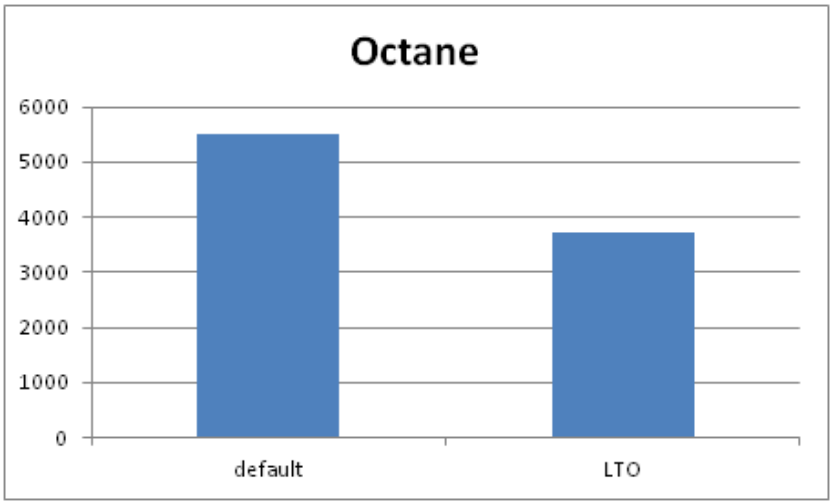


Fig. 3 Effect of LTO

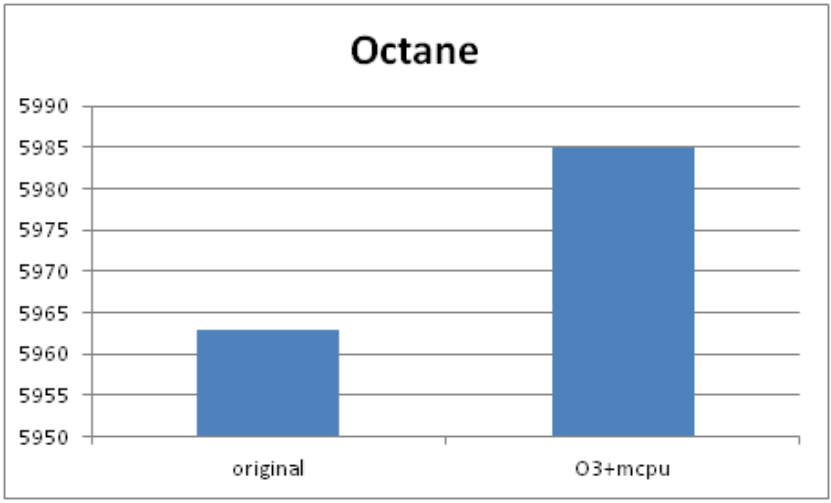


Fig. 4 Effect of platform options tuning

5. Runtime parameters tuning

V8 engine has quite large set of parameters which guides JIT compilation and execution of JavaScript programs. We have found that their default values are not adequate in all cases, e.g. we have found that disabling lazy compilation can substantially improve performance.

As noted in Section 2 V8 performs preliminary parsing of each new script source to separate each individual function. However when we specify parameter '--no-lazy', it instead compiles all functions at once in given script.

Enabling this mode has various impacts on different benchmark. We can see big degradation of CodeLoad test score by about 40% while in the same time huge increase 2.5 times of MandreelLatency test score. The overall increase about 5% was also reproduced on Galaxy Note 3 devices running Android 4.4.

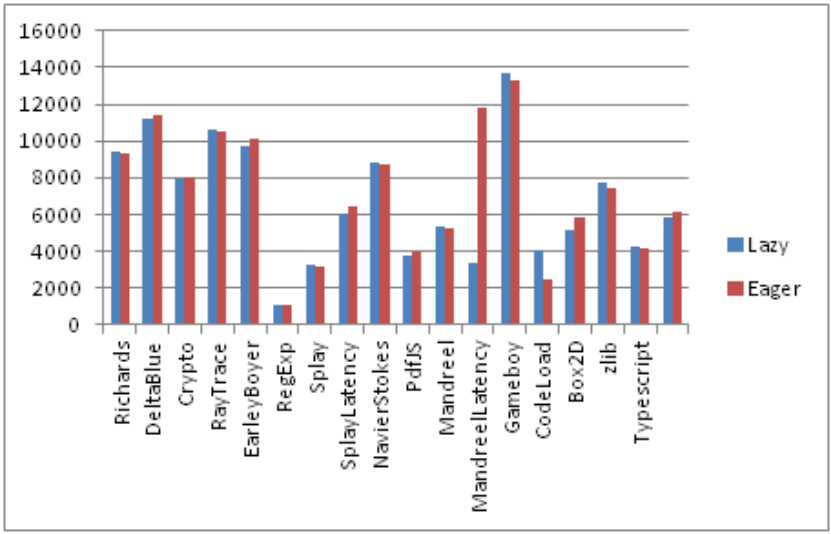


Fig. 5 Effect of eager compilation on Octane benchmarks.

6. Scalar optimizations

We have tried to implement several well-known scalar optimizations in V8 however with varying success. In contrast to ahead of time compilers for classic imperative languages such as C/C++, Pascal, Ada &c., just-in-time compiler has to share time among analysis, optimization and execution. That's why sophisticated optimizations which require thorough analysis don't necessarily lead to increasing performance in such case.

As noted in Section 2 the V8 engine performs optimized compilation of 'hot' regions similar to off-line compiles did. At this stage PICs already collected type

information so we can apply well-known scalar optimization techniques in AST and SSA representations.

The platform used in benchmark was Samsung Galaxy Note 3 with Qualcomm Snapdragon (N9005) CPU. Devices run Android 4.4.2 (KitKat). Octane benchmark suite used in tests was Version 9 download from corresponding repository. For development we use Android NDK r9c on Linux x86_64 Ububtu 12.04 TLS

6.1 Algebraic Simplification

The Algebraic Simplification uses algebraic identities like *a - 0 = a* to simplify expressions. This transformation was implemented in V8 parser when it builds AST representation for Crankshaft.

As was noted above at this point we have collected type information so we can safely optimize algebraic expression given that operands are numeric.

Despite the large amount of optimized expressions in Octane benchmark suite the final result was very small.

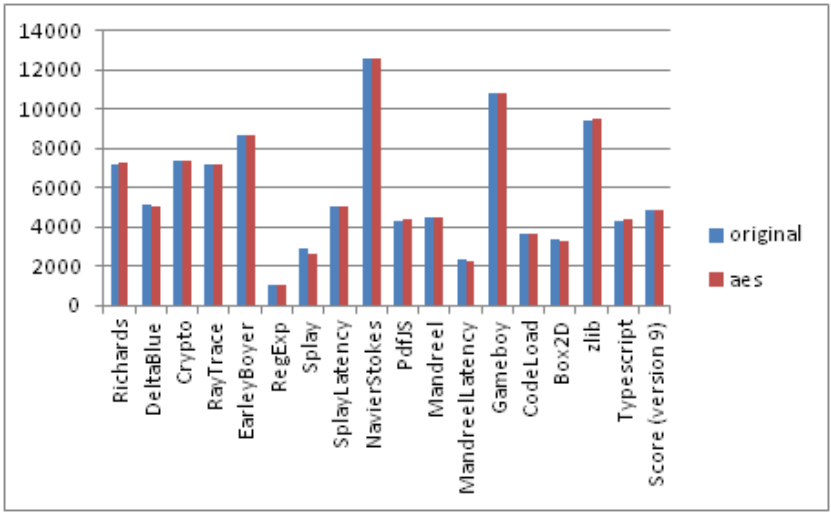


Fig. 6 Effect of Algebraic Expression Simplification

6.2 Common Subexpression Elimination

V8 engine already has implemented Global Value Numbering optimization which eliminates redundant code. However there are related but not identical optimizations such as Constant Propagation and Common Subexpression Elimination. For their differences see [14].

Because V8 already has some kind of Constant Propagation we decided to implement Global Common Subexpression Elimination in SSA form.

We have found that running this optimization before and after Global Value Numbering gives net effect about 2% performance improvement.

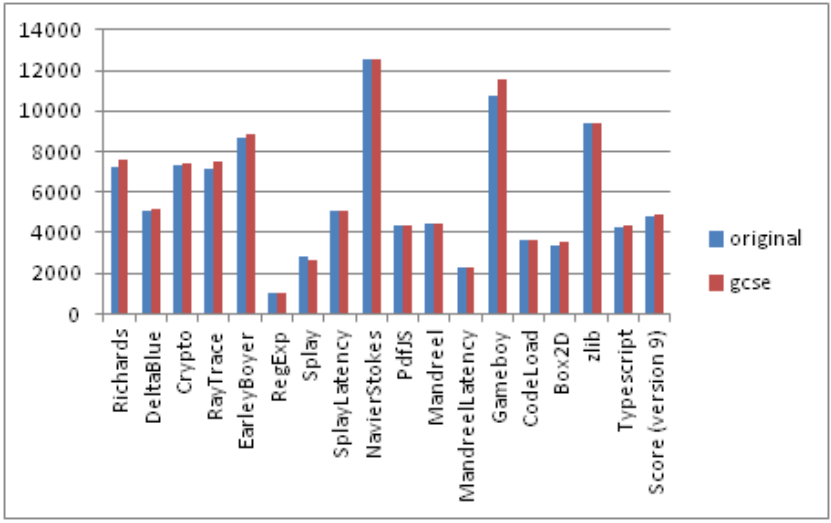


Fig. 7 Effect of Global Common Subexpression Elimination

6.3 Fast call frame for ARM

In our investigations we also have found interesting instruction sequence that speeds up call frame management on original ARMv7 CPUs.

To support EABI [15] compiler typically generate the following prologue and epilogue in each function.

Prologue:

```
func:
    stmdb    sp!, {r4-r5, fp, lr}
    add      fp, sp, #N
```

Epilogue:

```
    mov      sp, fp
    ldmia    sp!, {r4-r5, fp, lr}
    bx       lr
```

We have found however that the following sequences of instruction while provide the same functionality are executed faster on ARMv7 CPUs:

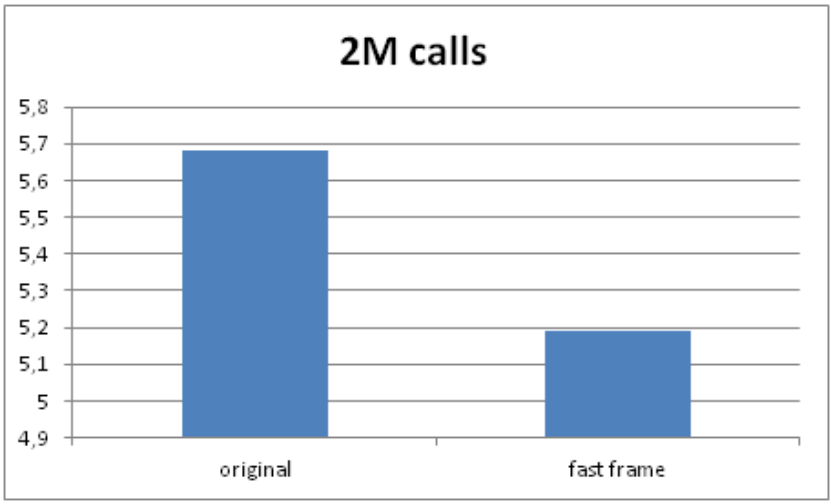
Prologue:

```
func:
    sub sp, sp, #16
    stm sp, {r4,r5,fp, lr}
    add      fp, sp, #N
```

Epilogue:

```
mov       sp, fp
ldm sp, {r4, r5, fp, lr}
add sp, sp, #16
bx        lr
```

The results on synthetic benchmark (~2 million calls, sec):



It is interesting however, that such results are not reproduced on Qualcomm Snapdragon 800 CPU.

7. Conclusion

We have found that even in the presence of type information in V8 optimizing compiler application of traditional scalar optimizations in JavaScript gives diminishing returns.

On the other hand successful application of optimized build gives us evidence that there is a space for optimizations in JavaScript engines.

References

[1]. TIOBE Index for October 2015 (http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html).
[2]. Chrome V8, September 10, 2015 (https://developers.google.com/v8/?hl=en)
[3]. Just-in-time compilation, Wikipedia, October 17, 2015 (https://en.wikipedia.org/wiki/Just-in-time_compilation)
[4]. Hölzle U., Chambers C., Ungar D. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches, ECOOP ‘91 proceedings, Springer Verlag Lecture Notes in Computer Science 512, July, 1991

[5]. Wingo A., On-stack replacement in V8, June 20, 2011 (https://wingolog.org/archives/2011/06/20/on-stack-replacement-in-v8)
[6]. Octane 2.0 (https://chromium.github.io/octane)
[7]. SunSpider 1.0.2 JavaScript Benchmark (https://www.webkit.org/perf/sunspider/sunspider.html)
[8]. Kraken JavaScript Benchmark (version 1.1) (http://krakenbenchmark.mozilla.org)
[9]. perf (Linux), Wikipedia, (https://en.wikipedia.org/wiki/Perf_%28Linux%29)
[10]. Streamline Performance Analyzer, (http://ds.arm.com/ds-5/optimize)
[11]. Gprof, Wikipedia, (http://en.wikipedia.org/wiki/Gprof)
[12]. Interprocedural optimization, Wikipedia (https://en.wikipedia.org/wiki/Interprocedural_optimization)
[13]. GCC ARM options (https://gcc.gnu.org/onlinedocs/gcc-4.8.5/gcc/ARM-Options.html#ARM-Options)
[14]. Muchnik S., Advanced Compiler Design and Implementation, Morgan Kauffmann Publishers, San Francisco, USA, 1997, 856p
[15]. Application Binary Interface for the ARM Architecture v2.09 (http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0036b/index.html)

Подходы к оптимизации движка JavaScript V8

Дмитрий Бочарников <dmitry.b@samsung.com>
Московский исследовательский центр Самсунг,
Москва, ул. Двинцев, 12, корп. 1

Аннотация. JavaScript является одним из наиболее распространенных языков программирования. Однако производительность движков JavaScript не всегда удовлетворительна. Автором разработаны подходы, позволяющие повысить производительность движка V8 на 10% на основных тестовых наборах.

Ключевые слова: JavaScript, оптимизации, V8, исключение общих подвыражений

DOI: 10.15514/ISPRAS-2015-27(6)-2

Для цитирования: Бочарников Дмитрий. Подходы к оптимизации движка JavaScript V8. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 21-32. DOI: 10.15514/ISPRAS-2015-27(6)-2.

Литература

[1]. TIOBE Index for October 2015 (http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html).
[2]. Chrome V8, September 10, 2015 (https://developers.google.com/v8/?hl=en)
[3]. Just-in-time compilation, Wikipedia, October 17, 2015 (https://en.wikipedia.org/wiki/Just-in-time_compilation)

[4]. Hölzle U., Chambers C., Ungar D. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches, ECOOP '91 proceedings, Springer Verlag Lecture Notes in Computer Science 512, July, 1991
[5]. Wingo A., On-stack replacement in V8, June 20, 2011 (https://wingolog.org/archives/2011/06/20/on-stack-replacement-in-v8)
[6]. Octane 2.0 (https://chromium.github.io/octane)
[7]. SunSpider 1.0.2 JavaScript Benchmark (https://www.webkit.org/perf/sunspider/sunspider.html)
[8]. Kraken JavaScript Benchmark (version 1.1) (http://krakenbenchmark.mozilla.org)
[9]. perf (Linux), Wikipedia, (https://en.wikipedia.org/wiki/Perf_%28Linux%29)
[10]. Streamline Performance Analyzer, (http://ds.arm.com/ds-5/optimize)
[11]. Gprof, Wikipedia, (http://en.wikipedia.org/wiki/Gprof)
[12]. Interprocedural optimization, Wikipedia (https://en.wikipedia.org/wiki/Interprocedural_optimization)
[13]. GCC ARM options (https://gcc.gnu.org/onlinedocs/gcc-4.8.5/gcc/ARM-Options.html#ARM-Options)
[14]. Muchnik S., Advanced Compiler Design and Implementation, Morgan Kauffmann Publishers, San Francisco, USA, 1997, 856p
[15]. Application Binary Interface for the ARM Architecture v2.09 (http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0036b/index.html)