Augmenting JavaScript JIT with Ahead-of-Time Compilation

Roman Zhuykov, Vahagn Vardanyan, Dmitry Melnik, Ruben Buchatskiy, Eugeniy Sharygin
Institute for System Programming of Russian Academy of Sciences
Alexander Solzhenitsyn st., 25
Moscow, Russia
Email: {zhroma, vaag, dm, ruben, eush}@ispras.ru

***Abstract*—Modern JavaScript engines use just-in-time (JIT) compilation to produce a binary code. JIT compilers are limited in a complexity of optimizations they can perform at a runtime without delaying an execution. On the contrary, ahead-of-time (AOT) compilers don't have such limitations, but they are not well suited for compiling dynamic languages such as JavaScript. In the paper we discuss methods for augmenting multi-tiered JavaScript JIT with a capability for AOT compilation, so to reduce program startup time and to move complex optimizations to AOT phase. We have implemented saving of JavaScript programs as a binary package containing bytecode and native code in open-source WebKit library. Our implementation allows shipping of JavaScript programs not only as a source code, but also as application binary packages with a precompiled code. In addition, our approach does not require any language feature restrictions. This has resulted in performance gain for popular JavaScript benchmarks such as SunSpider and Kraken on ARM platform, however, at a cost of increased package size.**

Keywords*—*JavaScript, JIT, Ahead-of-Time Compilation, JavaScriptCore, WebKit

I. Introduction

Dynamic properties of JavaScript language, such as presence of dynamic types and prototypes that can change during the execution make it almost impossible to compile the code effectively with static ahead-of-time compilers without restricting the language features. So most of the modern JavaScript execution engines use just-in-time (JIT) compilation techniques. However, JIT compilers are limited in performing complex optimizations, and take some time to compile a program before it can execute. To implement a tradeoff between quick startup and doing sophisticated optimizations, JavaScript engines usually use multiple tiers: lower tier JITs generate less efficient code, but can start almost immediately (e.g. even with interpretation), while higher tier JITs aim at generating very effective code for hot places, but at the cost of long compilation time. So even highly optimized JavaScript execution engines require some time to ”warm-up” before reaching their peak performance.

JavaScript is cross-platform, but unlike Java or .Net environments it does not have standard bytecode or other forms for binary distribution. Currently, the standard way for distributing JavaScript programs is the source code, often compacted with tools like Google Closure Compiler [1].

On the other hand, as now HTML5 and JavaScript are not only used for Web scripting, but also gain popularity as an application development platform for mobile and media devices (e.g. Tizen [2] and Firefox OS [3]), the performance and response time become even more important.

In the paper we discuss the developed framework for ahead-of-time compilation (AOTC) of JavaScript programs built upon open source engine JavaScriptCore [4] (JSC), which is the part of WebKit library. We have developed a binary format for saving JavaScript programs, which stores them in a form of bytecode, and optionally can contain a native code. We show the performance results for our implementation, as well as binary package size growth compared to plain and compacted JavaScript code. In addition, we discuss the problems that we had to solve to implement AOTC in JSC.

II. Related Work

There are number of works dedicated to ahead-of-time compilation of dynamic languages, which use two major approaches. The first one is to restrict a language to its subset, which can be compiled statically. Examples of such projects are static RPython to C compiler [5], ahead-of-time JavaScript compiler EchoJS [6], and Mozilla's asm.js [7]. Another approach is to save JIT-generated code and reuse it on the next execution, if possible. This approach is used for statically-typed languages like asm.js or Java [8], but also it was tried [9] with JavaScript.

Jeon and Choi [9] describe a method for reusing JIT-compiled code in JavaScriptCore (JSC) engine. The authors are saving and later reusing binary code generated by Baseline JIT [4]. The reported decrease in compile time when reusing the code is 44%. However, the problem of code relocation is not discussed in detail. Our current approach is based not only on saving generated machine code but also on saving Baseline JIT compiler intermediate representation (bytecode) without restricting the language features. In current version of JSC engine, it is also important to reuse the compiler code (or intermediate representation) at different JIT levels.

III. WebKit's JavaScriptCore Multi-Tier JIT Architecture

A multi-tiered JIT structure of JavaScriptCore is shown in Fig. 1. JSC first parses source code into abstract syn-

978-1-4673-7562-7/15/$31.00 ©2015 IEEE

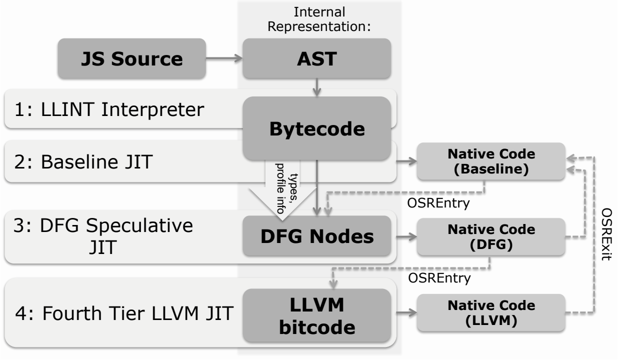


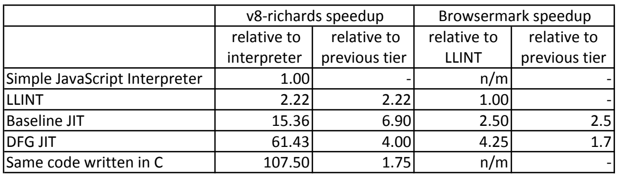
Fig. 1. JavaScriptCore Multi-Tier JIT Architecture

tax tree (AST). After that it builds internal representation called *bytecode*. Bytecode instructions are non-typed and this internal representation semantically is mostly equivalent to JavaScript. Bytecode instruction stream is stored in an array, and instructions have variable length. In modern versions of JavaScriptCore instead of classic interpreter by default LLINT (low level interpreter) is used. It is implemented in a special cross-platform assembly language called *offlineasm*. While building JavaScriptCore, offlineasm can be compiled into native code or can be converted into C source.

LLInt is intended to have zero start-up cost (not counting time required to build bytecode). At the same time it follows the same calling, stack, and register conventions used by JSC's just-in-time compilers.LLInt includes optimizations such as inline caching [10] to ensure fast property access. It also collects lightweight profiling information about types and last values of the objects.

Baseline JIT optimization starts only for hot paths. The first level of JIT-optimization kicks in for functions that gain at least 100 execution points. For each invocation the function gains 15 points, and each loop iteration adds one point. These numbers are approximate; the actual heuristic depends on function bytecode size and current memory pressure.

Baseline JIT emits appropriate native code for each bytecode instruction. This native code implements all possible cases for each operation. For example, addition for numbers would execute mathematical addition, but for strings it means concatenation. Generated native code contains many different branches to consider all possible cases. When native code is ready, it certainly would be used for new function invocations. Moreover, the LLInt will on-stack-replace (OSR) to JIT even if it is stuck in a loop; as well as all callers of the function are relinked to point to the compiled code as opposed to the LLInt prologue. OSR means that after some loop iteration LLInt will jump right to an appropriate place in JIT-generated native code instead of interpreting the next instruction. Baseline JIT also acts as a fallback for functions that are compiled by next-tier optimizing JITs: if the optimized code encounters a case it cannot handle (for example, when type of value of the variable does not correspond to profile information), it bails to Baseline JIT. Such transition is called *on-stack-replacement* exit (OSR exit).



	v8-richards speedup		Browsermark speedup	
	relative to interpreter	relative to previous tier	relative to LLINT	relative to previous tier
Simple JavaScript Interpreter	1.00	-	n/m	-
LLINT	2.22	2.22	1.00	-
Baseline JIT	15.36	6.90	2.50	2.5
DFG JIT	61.43	4.00	4.25	1.7
Same code written in C	107.50	1.75	n/m	-

Fig. 2. Speed of JavaScriptCore execution tiers for v8-richards and BrowserMark benchmarks

The next optimization level called DFG JIT (Data Flow Graph JIT, also referred as Speculative JIT), which performs speculative optimizations using collected profile information. The information collected includes variables type information, recent values loaded into arguments, loaded from the heap, or loaded from a call return. DFG JIT optimization starts only for those functions, which gain 1000 execution points, again, these numbers are approximate and are subject to additional heuristics. Speculative JIT performs aggressive type speculation based on profiling information collected by the lower tiers. All optimizations are performed on SSA internal representation called data flow graph (DFG), and instructions are nodes of the graph. DFG is build from function bytecode using profile information, and after all optimization passes native code is created for each DFG node. As described earlier, DFG uses deoptimization (OSR exit) to handle cases where speculation fails. Altogether, the Baseline JIT and the DFG JIT share a two-way OSR relationship: Baseline JIT may OSR into the DFG when a function gets hot, and the DFG may OSR to the Baseline JIT in the case of deoptimizations. Repeated OSR exits from the DFG serve as an additional profiling hint: the DFG OSR exit machinery records the reason of the exit (including potentially the values that failed speculation) as well as the frequency with which it occurred; if an exit is taken often enough, then reoptimization kicks in: callers are relinked to the Baseline JIT for the affected function, more profiling is gathered, and then DFG may be later reinvoked. Reoptimization heuristics uses exponential back-off to prevent situation when some pathological code causes permanent reoptimization and spent a lot of time in OSR transitions.

The fourth optimization level is called FTL JIT (Fourth Tier LLVM JIT), and is used only for functions which gain more than 10000 execution points. It performs wider set of optimizations and uses LLVM bitcode as intermediate representation. Instead of generating machine code directly from the DFG, its representation is lowered to LLVM bitcode and then LLVM optimization pipeline and backend are invoked to generate machine code.

Fig. 2 shows performance comparison of different JSC tiers. Note that we didn't run Browsermark benchmarks with classic interpreter, as well as didn't convert them to equivalent C versions. The data for `v8-richards` was measured in [11]. Overall, the peak JavaScriptCore performance for computational-intensive programs with C-like semantics can

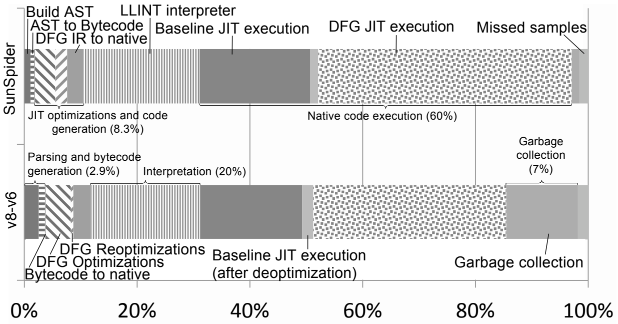


Fig. 3. JavaScriptCore execution time breakdown (the percentages shown are average among two benchmarks)

be roughly estimated as a half of that for C code compiled with a traditional optimizing compiler like GCC.

IV. General approach for AOT compilation in JavaScriptCore

In order to understand possible improvements from ahead of time compilation, we started with collecting information about how much time each JavaScriptCore execution stage takes. We have improved sampling-based profiling in JavaScriptCore and ran with it SunSpider and v8-v6 JavaScript benchmarks. JSC tiers execution time breakdown is shown in Fig. 3. Ahead of time compilation can improve the performance in several ways. First, we can save time on some operations, like building AST, creating bytecode and generating native code. But loaded bytecode and native code need to be linked in new runtime environment, and such linking may be also time consuming. Second way to improve performance is to speed up the code by shifting its execution to the next tier. For example, if we provide DFG JIT with pre-collected profile info, the code which currently executes with LLInt and Baseline JIT can potentially start its execution right on DFG. The possible speedup can be estimated from data in Fig. 2. Third, offline optimizations can be much more complex than JIT can afford, so the AOT-compiled code can be optimized better.

So the general idea of adding ahead-of-time compilation in JavaScriptCore is that by pre-saving optimized IR, native code, profile information or other data, normally built only during program execution, we may start execution at higher-level JIT, or start right with pre-optimized code.

A framework for optimization may involve profile-based optimizations, or be fully static. In addition, the optimizations may be implemented similar to caching at client-side, or performed at the server side. The latter option assumes development of a binary package format for application distribution. One of the benefits of AOT-compiled code that ships as a binary package, is that it provides basic source code protection, making reverse engineering and unauthorized copying of the code much less straightforward.

In JSC, any AOTC implementation in any case along with its data should be saving JavaScript source or bytecode, because it's the only level of execution that supports all JavaScript features, so in case of deoptimization at higher level JITs it should be able to continue execution with bytecode.

In our work, we have built AOTC framework, which involves server-side component to compile JavaScript applications into binary package, and a client-side component that can load those packages. Currently, the package may consist of bytecode, and optionally contain Baseline JIT's native code.

V. Saving JavaScriptCore Bytecode

The first part of our AOTC framework is dedicated to saving JavaScript source in bytecode form and loading it before execution. In normal JavaScriptCore workflow bytecode is generated for each function only at the moment of its invocation. We developed another workflow to allow saving bytecode without running the script. We save bytecode data into a file together with some additional information, such as constant tables, switch tables, and all necessary data for exception handling and regular expressions. To save bytecode without running the script we have to emulate work of namespaces stack. JavaScriptCore bytecode was not designed as an internal representation to be written into a file, its main purpose is effective interpretation and fast generation of native code on Baseline JIT level. Unlike JavaScript source, bytecode reflects the semantics of the program only in particular context. For example, it may depend on object properties which are already created at the moment of function invocation. This difference is mainly in some operation flags which allow optimizing the property access. However, sometimes bytecode saved in another context may produce a wrong result. Futhermore, global objects contain absolute addresses in bytecode and it was necessary to relink these addresses to correct ones while loading bytecode. All these details are taken into account when developing our framework for ahead-of-time bytecode saving. First we tried to store all the bytecode data in an SQLite database, but the performance overhead for using SQLite was too big while loading bytecode for some tests. Now we just store bytecode and all additional data as a byte array inside a file. In the beginning of the file, we insert an offset map, which allows fast access to function information. When modified JavaScriptCore version reads a bytecode file, it loads an offset map and global bytecode (bytecode that corresponds to JavaScript source not inside any function). After the execution starts, and if a function saved in bytecode is invoked, its bytecode is lazily loaded from a file using an given offset from the map.

JavaScriptCore has interesting feature that for each function in the source code there could be two distinct bytecode versions: one for a regular function call, and another when a function is called as a constructor using a `new` keyword. In our implementation, we save only the normal bytecode version and transform it to a constructor form when it is necessary.

We compile bytecode from JavaScript source statically without actually running the script. Still, we provide full support for ECMA-262 standard, including `eval()`. When a call to `eval()` is encountered, its argument string follows regular execution path for JavaScript source code, which involves building AST and bytecode at a run time. The only exception is a method which explicitly needs a JavaScript source. One example of such functions is `toString()` method applied to a function. Another example is using the line property of an exception object (it should contain the line number where exception was thrown).

A. Performance and Binary Packages Size

We have tested our AOTC version of JavaScriptCore (with saving bytecode only, without saving native code) on popular JavaScript benchmarks. It has shown 3.3% speedup on SunSpider, 1% on v8-v6 and 16% on Kraken benchmarks. Great speedup for Kraken is explained by its 2MB data files which contain JavaScript arrays initialized with floating-point numbers. With AOTC the parsing of those numbers is eliminated, as they can reside within our package in binary form.

Resulting binary sizes are 1.2–4.4 times larger than original JavaScript sources (on average x1.3 size growth for SunSpider, x4.4 for v8-v6 and x1.3 for Kraken). The sources were compacted with Google Closure Compiler, and both sources and binaries were compressed by gzip. For original non-compacted sources and both uncompressed sources and binaries the growth rates were x1.2 for SunSpider, x2.3 for v8-v6 and x2 for Kraken.

VI. Saving Baseline JIT Native Code

As the next step, we have extended our original AOTC approach for saving bytecode in JavaScriptCore to save the native code produced by Baseline JIT. The main problem with loading previously saved native code is the address relocation. Namely, the absolute addresses of objects saved in native code (e.g. `GlobalObject`, identifiers, core internal functions addresses) should be replaced upon loading the code with addresses those objects have during current execution.

At the saving time, we use our original AOTC driver to visit each CodeBlock and then we run Baseline JIT in order to generate the native code.

During code generation phase, we identify objects that will need relocation, and capture their offsets and a link to original objects, and save them in relocation tables.

We save the resulting native code to same binary file as we saved bytecode, along with relocation tables and some additional information required to patch the addresses properly. We don't save original absolute addresses in the native code at to-be-patched offsets, but pad them with zeroes for better compression.

The objects that need relocation in native code include:

- Callee addresses. There are calls to JavaScriptCore engine internal functions, as well as to generated code (trampolines and stubs to call translated user functions and runtime functions);
- Addresses of global variables, constants and identifiers (for referencing object properties), `GlobalObject` address;

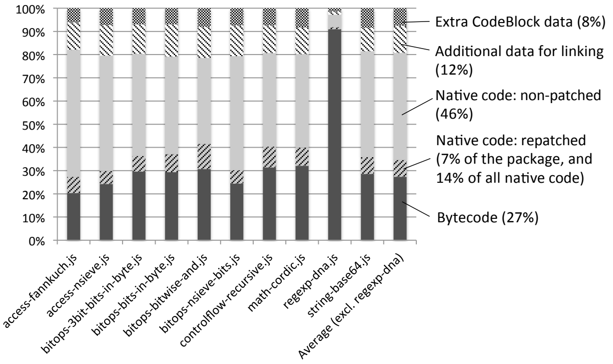


Fig. 4. Binary package (bytecode + native code) structure

- Pointers to memory allocators structures, both generic (remaining capacity, current payload, free-list head) and object-specific (e.g. for `JSArray`) are also emitted by JIT as absolute values;
- Address of execution counters for a CodeBlock (used to decide whether the code is hot enough to switch to DFG JIT), and other JavaScriptCore internal data structures.

All references to these objects are saved as indices in corresponding object tables.

The resulting binary package consists of the following sections:

- Bytecode and other related original AOTC data;
- Native code – the code generated by Baseline JIT;
- Additional data for linking – native code offsets, function indices and other data sufficient to patch absolute addresses with valid values at the loading time;
- Extra CodeBlock data – the data generated after bytecode generation at Baseline JIT along with the native code. It is necessary to preserve this data if Baseline JIT generation step is skipped at runtime.

A. Code Size Growth

The code size of binary packages produced when saving native code along with bytecode is on average 2.5-5 times larger than those containing just bytecode (measured on SunSpider and Octane benchmarks on `x86_64`). Figure 4 shows binary package structure for 10 of SunSpider tests. Significant part of saved native code has to be repatched (10-23%, 14% on average) with new absolute addresses upon loading.

B. Performance

The performance impact of loading precompiled code turned out to be negligible: it didn't show any significant performance change on SunSpider and v8 tests. This can be explained by the following reason. Sampling data for original JavaScriptCore shows that the code generation part in Baseline JIT takes just 0.1% of total JavaScriptCore run time for both SunSpider and V8-V6 tests (see *Bytecode to native* thin bar in Fig. 3). Though this data was collected with LLINT enabled, and cold functions weren't compiled with Baseline JIT at all, the code generation part of lower-tier JIT appears to be quite straightforward even compared to parsing source to AST (1-2.5%) and bytecode generation (0.7-1%). Considering the time necessary to load native code and to link it properly using address relocation tables, we can't load the code much faster than original JSC backend can directly generate it from bytecode. In addition, large binary size (on average, 5-10 times larger than original JavaScript source) contributes to slow processing too.

The performance issues with saving native code need further investigation, but the reason described above makes it unlikely for Baseline JIT native code saving to result in a speedup. Still, it could be possible to use native code without corresponding bytecode for selected program functions that require source code protection better than that provided by bytecode.

VII. Conclusion

We have developed a framework for ahead-of-time compilation of JavaScript programs. It was implemented in JavaScriptCore (JSC) open source JavaScript engine (a part of WebKit library). The framework consists of two components: command-line compiler, which compiles source JavaScript program into compressed binary package, consisting of bytecode (JSC IR) and optionally native code (produced by JSC's Baseline JIT). The second component is the patched JSC engine with a capability for loading and executing binary packages produced by the compiler.

The ahead-of-time compilation framework fully supports ECMA-262 standard. Also it provides 1%, 3% and 16% speedups for SunSpider, v8-v6 and Kraken benchmarks respectively when executing with precompiled bytecode. However, the binary sizes are 1.2–4.4 times larger than original JavaScript source. Generated native code saving, resulted in further 2.5–5 times binary size increase, but without any additional speedup due to the large part of JIT-generated code requires relinking.

We plan to continue work by researching the possibility to save optimized intermediate representation of higher JSC tiers, namely, DFG and FTL JITs. We'll be also investigating options for saving type profile information and inline cache data.

References

[1] Google Closure Compiler open source project website. [Online]. Available: https://code.google.com/p/closure-compiler

[2] Tizen platform website. [Online]. Available: http://www.tizen.org

[3] Mozilla website. [Online]. Available: https://www.mozilla.org

[4] JavaScriptCore engine webpage on WebKit open source project site. [Online]. Available: http://trac.webkit.org/wiki/JavaScriptCore

[5] RPython to C compiler webpage. [Online]. Available: http://pypy.readthedocs.org/en/latest/translation.html

[6] Ahead-of-time JavaScript compiler EchoJS webpage. [Online]. Available: https://github.com/toshok/echo-js

[7] Asm.js website. [Online]. Available: http://asmjs.org

[8] S. Hong, J. Kim, J. W. Shin, S. Moon, H. Oh, J. Lee, and H. Choi, "Java client ahead-of-time compiler for embedded systems," *SIGPLAN Not. 42*, 2007, pp. 63-72.

[9] S. Jeon and J. Choi, "Reuse of JIT compiled code based on binary code patching in JavaScript engine," *J. Web Eng. 11*, 2012, pp. 337-349.

[10] U. Holzle, C. Chambers and D. Ungar, "Optimizing dynamically-typed object-oriented languages with polymorphic inline caches," *Proceedings of the European Conference on Object-Oriented Programming*, 1991, pp. 21-38

[11] F. Pizlo, "Optimizing JavaScript" presentation on website. [Online]. Available: https://trac.webkit.org/wiki/JavaScriptCore