ANALYSIS AND OPTIMIZATION OF JAVASCRIPT ENGINES

Gem Dot†, Alejandro Martínez‡, Antonio González†

† Technical University of Catalonia
{gdot,antonio}@ac.upc.edu

‡ ARM
almavi1980@gmail.com

1. INTRODUCTION

JavaScript is an interpreted computer programming language embedded into web pages that allows the creation of sophisticated solutions in the client-side web. It consists of a core programming language together with a host environment, namely, the Document Object Model (DOM) provided by the web browser.

JavaScript has to be interpreted and executed by a dynamic translator inside the web browser, unlike desktop applications, which execute optimized code for a specific operating system. This dynamic translator is commonly called the JavaScript engine. Google uses V8 in Google Chrome and Node.js, Apple uses Nitro in Safari, Mozilla uses Rhino in Firefox and Microsoft uses JScript engine in Internet Explorer. Important efforts to improve the performance of these engines have been done in the last years. In this work, we focus on V8 JavaScript engine, which is an open source and widely used JavaScript engine developed by Google that ships with Chrome web browser.

V8 is a dynamic compiler and therefore, program execution and code generation have to be efficiently synchronized in order not to affect responsiveness. It focuses on optimizing hot functions (i.e. those that execute more often). In this regard, V8 integrates two compilers, one that runs fast (i.e. has light overhead) and produces generic code (Full-codegen); and one that does not run as fast but generates more optimized code (Crankshaft).

There is a significant number of executed instructions that are not part of the JavaScript application itself. These instructions can be considered as overhead produced by the V8 engine, which is a consequence of the characteristics of the JavaScript language along with the dynamic compilation nature of V8. This work deals with V8 targeting x86-64 processors. We have quantified two kinds of overhead in the JavaScript execution in steady state. The first one is the V8 runtime overhead needed to support native execution (i.e. garbage collector, compilation, etc.). The second one is the additional checks and guards introduced by the dynamic nature of JavaScript. These two types of overhead represent 35% and 25% respectively of the total execution time in average

There is a great opportunity to improve the performance of JavaScript platforms by reducing these overheads. Therefore, we propose using the insights gained from this study to improve certain parts of V8, concretely the overhead due to the dynamic nature of JavaScript. The optimizations that we propose are based on hw/sw codesigned techniques that incorporate new x86-64 instructions tailored for dynamic languages.

2. PROPOSAL

We propose three optimizations that are supported by the introduction of new hardware and the necessary software changes. In this work we assume the x86-64 instruction set as target for V8 and, so we introduce several new x86-64 instructions to support these optimizations.

The proposed HW/SW codesigned based optimizations have been simulated at micro-architectural level through a cycle-level simulator. We have named each optimization after the kind of operation that it tries to accelerate.

2.1 Deoptimization Bailout and *Check Stack* Optimization

When optimized code is checked to potentially deoptimize it, two instructions are used. The first one is an instruction that changes a flag. This instruction is usually a *test* or a *cmp* instruction, but not always. The second one is a conditional branch instruction that jumps to a deoptimization bailout depending on the flag set by the previous instruction. We have observed that optimized code is rarely deoptimized.

The idea behind deoptimization bailout optimization is changing these two instructions into a new one (Figure 1). This new instruction performs an ALU operation, checks a flag specified in the instruction and raises an exception in case that the check succeeds. The key point is that in the vast majority of cases (almost 100%) the code is not deoptimized. Therefore, the benefits of this new instruction are that when the code is not deoptimized, less dynamic instructions are executed and branch prediction is not needed.

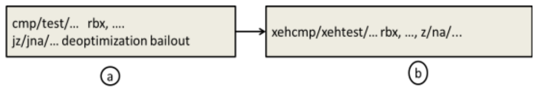
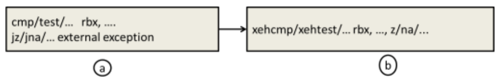

cmp/test/... rbx, ...
jz/jna/... deoptimization bailout
xehcmp/xehtest/... rbx, ..., z/na/...
a
b

Figure 1. Deoptimization pattern improvement.

If the code has to be deoptimized, a hardware exception is thrown. This exception is intercepted by a handler in the V8 runtime that executes a special routine, which finds the action to do according to the current program counter. This action is a jump to an address that targets a specific deoptimization bailout. The overhead is negligible compared with the deoptimization routine itself.

We can use the same exception mechanism to optimize the *Check Stack* pattern. It uses exactly the same approach to find the action to do. However, in this case, the action is a call that interrupts the program because another external exception has taken place. Figure 2 represents this optimization at instruction level.



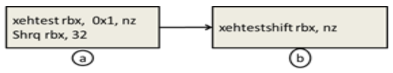
```
cmp/test/... rbx, ....
jz/jna/... external exception
```

```
xehcmp/xehtest/... rbx, ..., z/na/...
```

Figure 2. ***Check Stack*** **improvement.**

2.2 Tagged-to-Integer optimization

After applying the optimization presented in the previous section, Figure 3a presents the new code for *Tagged To Integer* conversion. We propose to join the *xehtest* and shift instructions in only a new single one, which is named *xehtestshift* (Figure 3b). This new instruction shifts the value of the register. If the value is not a SMI, an exception to deoptimization is produced.



```
xehtest rbx, 0x1, nz
Shrq rbx, 32
```

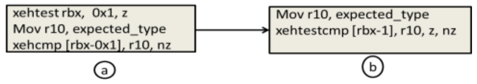
```
xehtestshift rbx, nz
```

Figure 3. Tagged-to-Integer pattern improvement.

2.3 Check Non-SMI and Check Map Optimization

We have realized that there are certain sequences of checks that are very frequent. The most repeated pattern is a *Check Non-SMI* followed by a *Check Map*. Therefore, with the optimization presented in this section, the sequence of instructions regarding this pattern is optimized.

After applying the optimization presented in section 4.1, Figure 4a presents the new code for the *Check Non-SMI* and *Check Map* pattern. We propose to join the *xehtest* and *xehcmp* instructions into a new one (Figure 4b), which is named *xehtestcmp*. This instruction checks whether *this* pointer does not contain a SMI and whether *this* points to an object whose type is the expected one. Otherwise, an exception that jumps to deoptimization code is produced. In this way, the number of dynamic instructions for this pattern are reduced to only two (Figure 4b).



```
xehtest rbx, 0x1, z
Mov r10, expected_type
xehcmp [rbx-0x1], r10, nz
```

```
Mov r10, expected_type
xehtestcmp [rbx-1], r10, z, nz
```

Figure 4. Check non-SMI and check map pattern improvement.

3 RESULTS

Below we present the performance of our optimizations in number of cycles, using three benchmark suites to analyze the V8 dynamic compiler: the Octane suite, Kraken suite and the SunSpider suite. We use a timing simulator with a micro-architectural configuration closely matching a Nehalem core.

All suites show an important improvement for the Deoptimization Bailout optimization. Overall, the proposed optimizations reduce the number of cycles by 5.2%, 6.2% and 7%, for Octane, Kraken and SunSpider suites respectively.

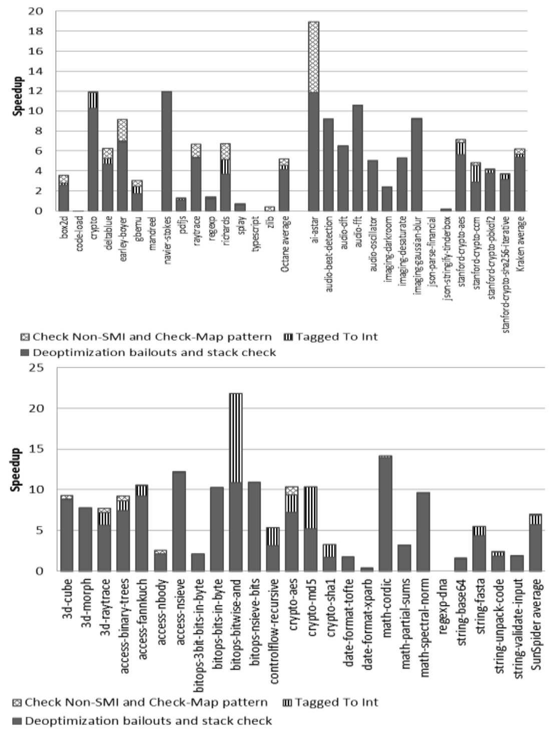


Figure 5: Improvement in number of cycles.

4 CONCLUSIONS

JavaScript is a widely-used dynamic language used by Web browsers for web applications, and its popularity is expected to increase in the future. Therefore, the optimization of JavaScript engines will have a great impact in future computing systems.

We have quantified the execution overhead and classified it into two main categories. The first one is the code that V8 runtime executes for different housekeeping activities (i.e. garbage collector, compilation, etc.). The second one is the additional checks and guards introduced by the dynamic nature of JavaScript. These overheads are important, and they represent around 60% (25% additional checks and guards, and 35% compilation, garbage collector and helpers) of the total execution time in steady state.

Finally, guided by these results, we have developed three novel HW/SW based optimizations, which reduce the most important sources of this overhead. They are based on a hybrid HW/SW approach that requires the introduction of some new machine instructions, some additional hardware support and some changes in the code generated by the dynamic compiler. We have shown that these optimizations result in a 6% average speedup for representative benchmarks.

5 ACKNOWLEDGMENTS

This work has been partially supported by the Spanish Ministry of Economy and Competitiveness under grants TIN2010-18368 and TIN2013-44375-R and the Spanish Ministry of Education, Culture and Sport under grant FPU12/05670.