Performance and Power Usage of Thread-Level Speculation in the V8 JavaScript Engine

Jan Kasper Martinsen, *Student member, IEEE,* Håkan Grahn, *Member, IEEE*, Anders Isberg and Samir Drincic

Abstract—Thread-Level Speculation can exploit parallelism in JavaScript for web applications with a significant speedup. So far all the measurements have been made on an x86 based workstation with a dual quadcore processor. In this paper, we measure the effects of Thread-Level Speculation on a Sony Xperia Z1 smartphone with a quadcore processor. We measure both the execution time and the power usage on a large number of web applications, the Google maps application and on HTML5 applications. We show that we are able to significantly reduce the execution time, and perhaps equally important on devices with a limitied amount of battery, we show that we reduce the power usage.

Index Terms—Multicore processors, Parallel Computing, Automatic Parallelization, JavaScript

1 Introduction

JavaScript is a dynamically typed, object-based scripting language with runtime evaluation, where the execution is done in a JavaScript engine [1], [2], [3]. Fortuna et al. [4] show that there is a large potential for parallism for JavaScript in web applications with speedups up to 45 times compared to the sequential execution time. However, JavaScript is a sequential programming language.

One transparent manner to take advantage of parallism with parallel hardware is Software Thread-Level Speculation (TLS) [5] which have been demonstrated both for JavaScript benchmarks [6] and web applications [7].

In [7] we presented a method-level TLS implementation in Squirrelfish/WebKit with an on/off speculation principle in the Squirrelfish JavaScript engine without Just-in-time compilation, where a single misspeculation turns off the speculation for that function. We later extended this technique with a heuristic which adapts how aggressively we speculate [8].

In this paper we develop Thread-Level Speculation with Google's JavaScript engine V8 with Just-in-time compilation and measure its performance and power usage on a Sony Xperia Z1 mobile phone with a quadcore CPU. We evaluate the performance on 3 use cases each for 15 web applications, then on 4 different use cases for Google maps, and finally on 20 different web applications for the JS1k (http://www.js1kb.org) competion.

- *J.K. Martinsen and H. Grahn are with the School of Computing, Blekinge Institute of Technology, SE-371 79, Karlskrona, Sweden, {Jan.Kasper.Martinsen,Hakan.Grahn}@bth.se*
- *A. Isberg and S. Drincic are both with Sony Mobile Communications AB, SE-221 88 Lund, Sweden, {Anders.Isberg,Samir Drincic}@sonymobile.com*

This work was partly funded by the Industrial Excellence Center EASE - Embedded Applications Software Engineering, (http://ease.cs.lth.se), and the BESQ+ research project funded by the Knowledge Foundation (grant number 20100311) in Sweden.

2 Implementation of TLS in Google's V8 JavaScript engine

Web applications execute a large number of events, which in turn are defined as JavaScript functions.

When V8 encounters a function, it checks if the function has previously been encountered. If it has not been encountered, V8 decides what kind of function this is; if it comes from an *eval* call, from a normal function (or JavaScript segment) or a lazily compiled function. In addition, it tries to decide whether the function is to be compiled in a normal way, or if it can be compiled in an optimized way. Once the function call is compiled, it is placed in a cache, in such a way, that if we were to encounter the same function once more we execute the already compiled function.

If the JavaScript code that is going to be compiled is a function call, then this function is a candidate for speculation. Since there are many function calls in web applications, we perform method level speculation, and base the sequential execution order on the order of the function calls.

Initially, we initialize a counter *realtime* to 0. For each executed JavaScript function, this value of *realtime* is increased by 1 We give the entry point of the V8 JavaScript engine an unique *id* (*p_realtime*) (initially this will be *p_0*).

We extract the *realtime* value and the id of the thread that makes this call, e.g., *p_0220* (*p_0* calls a function after 220 bytecode instructions). We denote the value of the position of this function call as *function_order*, which emulates the *sequential time* in our program (Fig. 1). We check if this function previously has been speculated on by looking up the value of *previous[function_order]*. *previous* is a vector where each entry is organized by the *function_order*.

If the entry of *previous* is 1, then the JavaScript function has been speculated unsuccessfully (i.e., this function call has led to a rollback). If the value is 0, then it has not been speculated, and we call this position a speculation point. That means that the function is not going to be found in the cache.

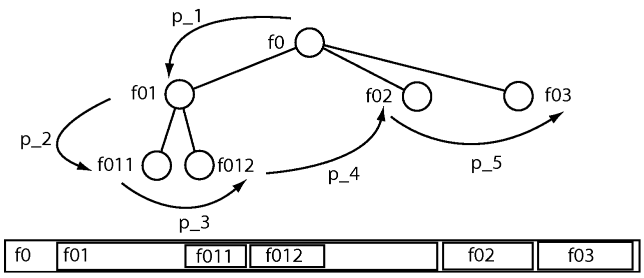

Fig. 1. We use the order that the functions are called in, to determine the order in which the program would have been sequentially executed. This works in JavaScript in a web applications setting, as there are multiple function calls. For instance, in a simplified example the JavaScript function f0, performs 3 function calls, f01, f02 and f03. f01 performs two function calls, f011 and f012. Thus, we have created a speculation tree from the function calls. If we traverse this tree from left to right, we get an order in which the functions are called, equal to the order in which the functions would be sequentially called, in order to uphold the sequential semantics during execution with TLS. More specific: f0 at time p_1, f01 at time p_2, f011 at time p_3, f012 at time p_4, f02 at time p_5, and f03 at time p_6. We denote how each function is ordered as *function_order*

We do the following after the function has been compiled; we set the position of the function calls *previous[function_order] = 1*. We save the checkpoint state which contains the list of previously modified global values, the list of states from each thread, the content of the used global variables, and the content of *previous*.

We then create a new thread (or, take a thread from the thread pool) for the function call in V8, with a unique id. We copy the value of *realtime* from its parent and modify the instruction pointer of the parent thread such that it continues execution at the return point of the function so the parent thread skips the function call.

Now we have two concurrent threads, and this process is repeated each time a suitable speculation candidate is encountered, thereby supporting nested speculation. If there is a conflict between two global variables or an incorrect return value prediction we perform a rollback to the checkpoint point where the speculation started.

In V8, the actual JavaScript global variables are not compiled into the code that are to be executed, instead these variables are accessed from function calls in the compiled code, such as *StoreIC_Initialize* and *LoadIC_Initialize*. This means, that the global JavaScript stack, and the native code are accessed separately. This further means, that we only need to save part of the JavaScript stack, when we are about to speculate. We therefore save the stack in case of a rollback. Restoration of the state, if we encounter a rollback is not very costly, as we do not need to re-compile the speculated function call. In fact the function call that is to be re-executed on a rollback is already ready for execution inside the cache. However, rollbacks are relatively rare, as we see in [7].

Like the implementation in [7] we support nested speculation, and therefore need to store states of the nested speculation, in case of a rollback. Like previously seen, JavaScript function calls rarely return any value in a web application, therefore we use a return value prediction scheme, like the *"last predicted value"* found in [9].

The largest difference between Squirrelfish where JIT is disabled and V8 is that in Squirrelfish the JavaScript code is compiled into bytecode instructions which then are interpreted, while in V8 the JavaScript code is compiled into native code which is later executed directly on the hardware.

The reuse of compiled functions opens up possibilities when we speculate and need to re-execute functions in case of a rollback. Since we do not add features to the native code, and since all JavaScript global objects are accessed through external functions, we do not need to re-compile the code upon rollbacks in TLS. The result is that the cost of doing rollbacks and re-executing functions decreases in terms of execution time, when we are using JIT as compared to an interpreted JavaScript engine such as Squirrelfish. This suggests that we could significantly improve the execution time for TLS, as we reduce the costs of rollbacks. For the implementation of TLS speculation in V8, we follow what is done in [7] where we speculate aggressively on function calls.

3 EXPERIMENTAL METHODOLOGY

We measure the effects of TLS on 15 well known [10] web applications, where we have created 3 use cases for typical functionality, then we measure the effects of TLS on 4 use cases for the Google map web application and finally we measure the effects of TLS on 20 entries in the JS1k competition, the top 10 from 2010 and 2013, which use HTML5 extensively.

In the *Amazon* web application, for the 1th use case we go to the front page, in the 2th use case we login with our personal account, and in the 3th use case we search for the name of one of the authors of this paper. The use cases progressively consist of the previous use cases, for instance the 3th use case, starts with the 1th, then the 2th use case before we complete it with the 3th use case.

In Google maps, in the 1th use case we load the front page, in the 2th, we locate our position and in the 3th and 4th use case we find the road map between Google's headquarter in Paolo Alto and Microsoft headquarter in Seattle respectively.

In the JS1k, the competitors submit one kilobyte of JavaScript sourcecode. We have measured the effects of TLS on the 10 top entries from 2010 and from 2013 these applications take advantage of features in HTML5.

The measurements are made on a Sony Xperia Z1 phone equipped with a 2.4Ghz quadcore CPU, and 4GB main memory. We have measured the JavaScript execution time in the JavaScript engine V8, and compared the TLS enabled version against the sequential version.

In addition we have measured the power usage with a battery simulator connected to a Sony Xperia Z1 phone, with a program which allowed us to instrument and measure the power usage.

4 Experimental Results

4.1 Improved execution time

In Fig. 2, Fig. 3 and Fig. 4 we see that TLS almost always improves the execution time for JavaScript in web applications even though the application types are very different from one another. We notice that the effects of TLS, in terms of reduction of execution time is increasing with more interaction (for instance in Fig. 2, *Facebook(3)* executes faster than *Facebook(2)* and *Facebook(2)* executed faster than *Facebook(1)*). This indicate that as we eleborate the use cases with more interaction, the effects of TLS becomes more apparent. The relatively small increase in execution time between the use cases also indicates that we could have an even better speedup with a larger number of cores.

For Google maps, we see that the effects of TLS are almost the same for all of the use cases. This indicates that the executed JavaScript is the same for all of the use cases, and that most of the functionality in Google maps (i.e., for instance calculating the distance between two points) is executed on the serverside.

For JS1K, we do not find the same execution time for TLS as for the other results, but there is obviously not the same amount of JavaScript code (since these programs are at most 1 kilobyte in size). The 3th application from 2010 and the 4th application from 2013 are slower than the rest. Both of these applications use sound intensively (A syntheziser and a version of the game Tetris^{TM}, which both use the functionality of HTML5 to play sounds). This slows down the execution time of TLS, as intensive access to the sound chip seems to make the cores halt.

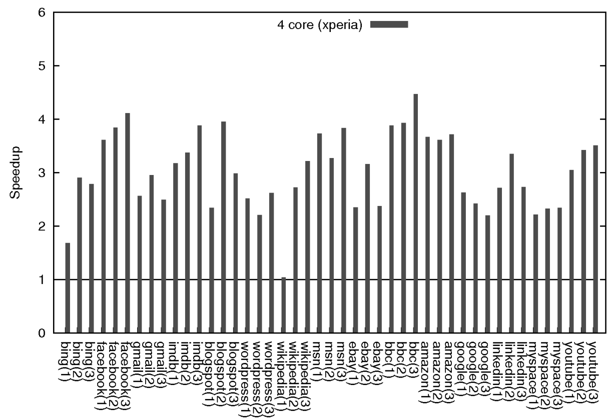

Fig. 2. The execution time for 45 use-cases from very popular web applications.

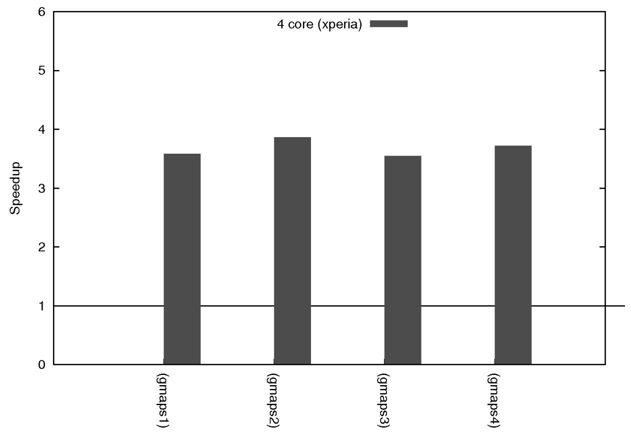

Fig. 3. The execution time for 4 google maps use-cases.

4.2 Power usage

In Fig. 5 we have measured the power usage of *Facebook* when running with TLS and when running without TLS. We see from the results that TLS version executes much faster (more than twice as fast), but uses at most twice as much power.

In Fig. 6 we have measured the power usage of Google maps when running with TLS and when running without TLS. We see from the results that the TLS version executes much faster (roughly three times as fast), but uses at most almost twice as much power.

In Fig. 7 we have measured the power usage of JS1K when running with TLS and when running without TLS. We see from the results that the TLS version

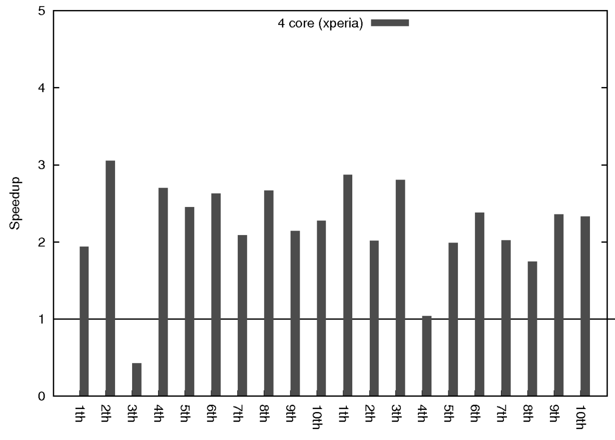

Fig. 4. The execution time for 20 HTML5 demos

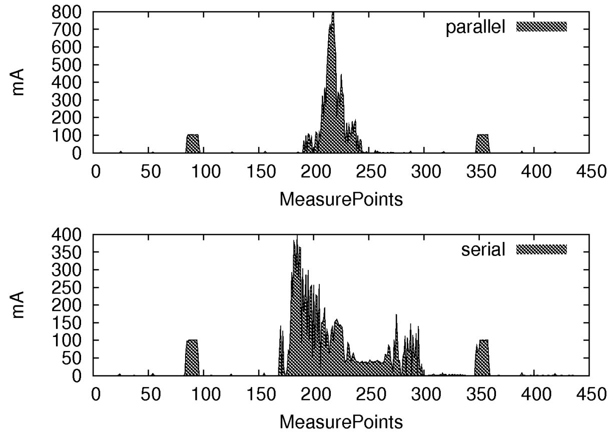

Fig. 5. The power usage for *Facebook* running on the Sony Xperia Z1 smartphone both in parallel and sequentially

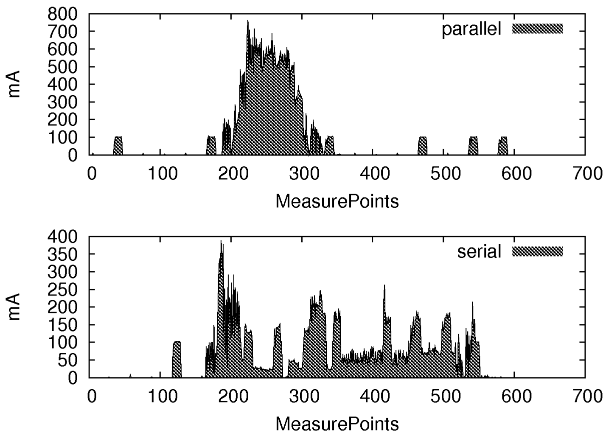

Fig. 6. The powerusage for Google maps.

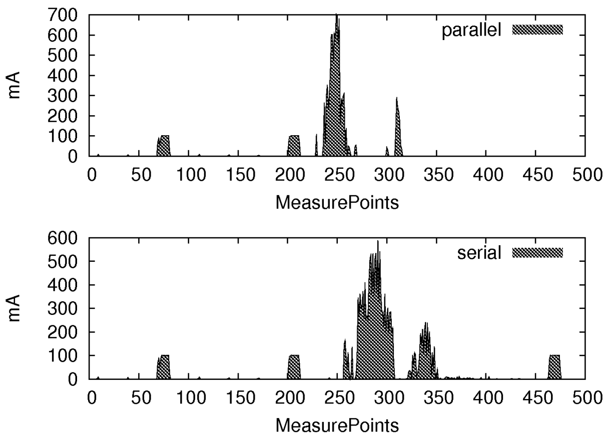

Fig. 7. The power usage for a HTML5 use-case.

executes slightly faster, and uses the lowest power compared to the sequential execution og the use cases.

While the maximum power usage is higher for TLS than the sequential version, the integral of the power usage is almost 10% lower for *Facebook*, over 40% lower for the HTML5 application and is 40% higher for TLS with *Gmaps*. We can understand this from the limited interaction for *Gmaps* use case. Therefore, we use a lot of power to set up the TLS system, while not being able to fully take advantage of it.

5 Conclusion

Thread-Level Speculation in the JavaScript engine V8 is suitable also for smartphones, as we both get a significant speedup and are able to reduce the overall power usage on a Sony Xperia Z1 phone.

References

[1] Google, "V8 JavaScript Engine," 2012, http://code.google.com/p/v8/.
[2] WebKit, "The WebKit open source project," 2012, http://www.webkit.org/.
[3] Mozilla, "SpiderMonkey – Mozilla Developer Network," 2012, https://developer.mozilla.org/en/SpiderMonkey/.
[4] E. Fortuna, O. Anderson, L. Ceze, and S. Eggers, "A limit study of javascript parallelism," in *2010 IEEE Int'l Symp. on Workload Characterization (IISWC)*, Dec. 2010, pp. 1–10.
[5] P. Rundberg and P. Stenström, "An all-software thread-level data dependence speculation system for multiprocessors," *Journal of Instruction-Level Parallelism*, pp. 1–28, 2001.
[6] M. Mehrara, P.-C. Hsu, M. Samadi, and S. Mahlke, "Dynamic parallelization of JavaScript applications using an ultra-lightweight speculation mechanism," in *Proc. of the 17th Int'l Symp. on High Performance Computer Architecture*, 2011, pp. 87–98.
[7] J. K. Martinsen, H. Grahn, and A. Isberg, "Using speculation to enhance javascript performance in web applications," *IEEE Internet Computing*, vol. 17, no. 2, pp. 10–19, 2013.
[8] J. Martinsen, H. Grahn, and A. Isberg, "Heuristics for thread-level speculation in web applications," pp. 1–1, 2013.
[9] S. Hu, R. Bhargava, and L. K. John, "The role of return value prediction in exploiting speculative method-level parallelism," *Journal of Instruction-Level Parallelism*, vol. 5, 2003.
[10] Alexa, "Top 500 sites on the web," 2010, http://www.alexa.com/topsites.