Preliminary Results of Combining Thread-Level Speculation and Just-in-Time Compilation in Google's V8

Jan Kasper Martinsen and Håkan Grahn
School of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden
{Jan.Kasper.Martinsen,Hakan.Grahn}@bth.se

Anders Isberg
Sony Mobile Communications AB
SE-221 88 Lund, Sweden
Anders.Isberg@sonymobile.com

ABSTRACT

We present the first implementation of Thread-Level Speculation in combination with Just-in-time compilation. The implementation is done in Google's V8, a well-known JavaScript engine, and evaluated on 15 popular web application executing on 2, 4, and 8 core computers. Our results show an average speedup of 1.55 on 4 cores, without any JavaScripts code modifications. Further, we have found that the Just-in-time compilation time is significant, and that most functions are lazely compiled (approximately 80%). Further, V8 contains features that are advantageous in Thread-Level Speculation.

1. INTRODUCTION

JavaScript is a dynamically typed, object based scripting language with run-time evaluation typically used to add clientside interactivity in web applications. Several optimization techniques have been suggested to speedup the execution time [4, 13, 9]. However, the optimization techniques have been measured on a set of benchmarks, which have been reported as unrepresentative for JavaScript execution in real-world web applications [6, 10, 11]. A result of this difference is that optimization techniques such as just-in-time compilation (JIT) more often than not increase the JavaScript execution time in web applications [7].

It has been shown [3] that there is a significant potential for parallelism in many web applications with a speedup of up to 45 times compared to the sequential execution time. To take advantage of this observation, and to hide the complexity of parallel programming and the underlaying hardware from the JavaScript programmer, one approach is Thread-Level Speculation (TLS) [12]. The performance potential of TLS has been shown for applications with static loops, statically typed languages, and in Java bytecode environments.

In [8], we implemented TLS in the Squirrelfish JavaScript engine (without JIT enabled), and measured the effect of TLS on 15 popular web applications. We employed an aggressive speculation scheme, and we were able to speed up the execution time up to 8 times compared to the sequential execution time.

In this paper, we present the first implementation of Just-in-time compilation and Thread-Level Speculation in JavaScript engines. Further, we have not found any other studies for other languages either with the TLS+JIT combination. The implementation is done in Google's V8 JavaScript engine that only supports JIT. In addition, we switch from the GTK Webkit to the Chromium web browser for executing the use-cases. The motivation for selecting this JavaScript engine is that the results in [7] shows that even though the workload in web applications often is not suitable for JIT, The V8 JavaScript engine is often faster than the JIT enabled Squirrelfish engine. We have also exteded the use-cases from 15 by adding more interaction to them, so we have a total of 45 use-cases on popular web applications.

Our initial results of this premlinary study show that we need more than 2 cores to be able to speedup the execution time of JavaScript execution in web applications with the TLS+JIT combination. The average speedup 1.55 when running on 4 cores, without any changes to the sequential JavaScript code. Further, implementation-wise, the web applications are designed in such a way that JIT can succesfully be combined with Thread-Level Speculation, and that the V8 JavaScript engine has features that are of an advantage for Thread-Level Speculation.

2. EXPERIMENTAL METHODOLOGY

Our Thread-Level Speculation is implemented in the V8 [4] JavaScript engine which is part of Chromium [5] from Google. From [8] and in Figure 1 we see that web applications have a larger number of JavaScript function calls. Therefore we speculate on the function level where all data conflicts are correctly detected and rollbacks are done when conflicts arise, and we support nested speculation.

In Table 1 we present the selected 15 web applications from the Alexa list [1] of most visited web applications. We have defined and recorded a set of use-cases for the selected web applications, where the use-cases are intended to exhibit example usage of the web application, and then executed them in Chromium. For each web application we have three use-cases. We can illustrate this with an example for the web application *Google*. In the first case, we load the page. In the second case, we type in the name of one of the authors of this paper. As we type in the name, it shows a number

Table 1: Web applications that are used in the experiments.

Application	Description
Google	Search engine
Facebook	Social network
YouTube	Online video service
Wikipedia	Online community driven encyclopedia
Blogspot	Blogging social network
MSN	Community service from Microsoft
LinkedIn	Professional social network
Amazon	Online book store
Wordpress	Framework behind blogs
Ebay	Online auction and shopping site
Bing	Search engine from Microsoft
Imdb	Online movie database
Myspace	Social network
BBC	News paper for BBC
Gmail	Online web client from Google

Table 2: The computers used for the experiments.

Name	#cores	OS	Mem size
Computer 1	2	Ubuntu 10.04	512MB
Computer 2	4	Ubuntu 12.04	4GB
Computer 3	8	Ubuntu 10.04	16GB

of suggestions to similar words. This is a feature that is JavaScript endured. In the third use-case we combine the two previous cases, i.e., we first load the page, then we type in the name of one of the authors, and finally we click the search button. We extended all the original use-cases with two more each, as we did for *Google*.

To enhance reproducibility, we use a scripting environment [2] to automatically execute the use-cases in a controlled fashion. A detailed description of the methodology for performing these experiments is found in [6].

We conduct these experiments on 3 different systems (Table 2) where the execution time is the JavaScript execution time in V8, rather than the overall execution time of the whole web application. We execute each use case 10 times, and then take the median of the execution time.

3. COMBINING JIT COMPILATION AND TLS

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.

JavaScript execution in web applications often have a large number of function calls. The measurements in Figure 1 indicate that there are many function calls, and that the function calls are continuously being compiled. Therefore, we can continue to speculate as we have done previously, namely on function calls. In Figure 1 we show how the

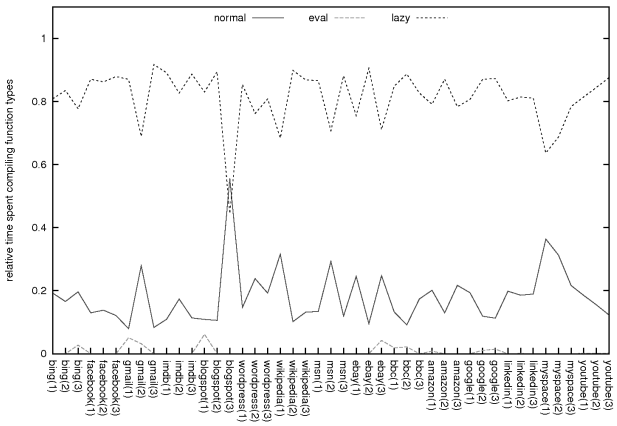

Figure 1: The relative time spent compiling the various function types. Normal functions are compiled when the web page is loaded, eval is compiled when the JavaScript code is executed using an `eval()` call, and lazy functions are compiled when they shall be executed.

functions are compiled. It is obvious that lazy compiled functions are most common. Measurements have shown that these are often very small (in terms of number of lines of JavaScript source), but not always. We notice *blogspot (1)*, where the amount of normal compilation is higher than the lazy compilations. This is due to that the use-case takes us to a page, where we simply only register our name, and where there is very little interaction (and thereby very little JavaScript, in the form of lazy functions).

The next observation is that writes and reads to global objects in V8 are not compiled into the native source. Instead, these are written or read by invoking certain functions. This means that we do not need to insert and compile specific instrumentation code into the native code to detect hazards such as WAR, RAW and WAR conflicts. We deal with these cases in the function calls that manipulates the global JavaScript objects. We therefore perform the conflict detection and rollback similarly to how we did in [8], only that instead of interpreting the code we run it as native code. Still each time we read or write to a JavaScript value, we determine if its order is sequentially correct.

Finally, when a function is compiled into native code it is stored in a cache, in case we shall re-execute the same function again. So the next time we are to compile a piece of code, we first lookup if the JavaScript code has already been compiled. In Figure 2, we see that in the use-cases this is a fairly rare occurrence (except for use-cases from Google).

The reuse of compiled functions opens up a possibility 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 re-execution. The result is that the cost of doing rollbacks and re-executing function decreases, in terms of execution time, when we are using JIT as compared to an

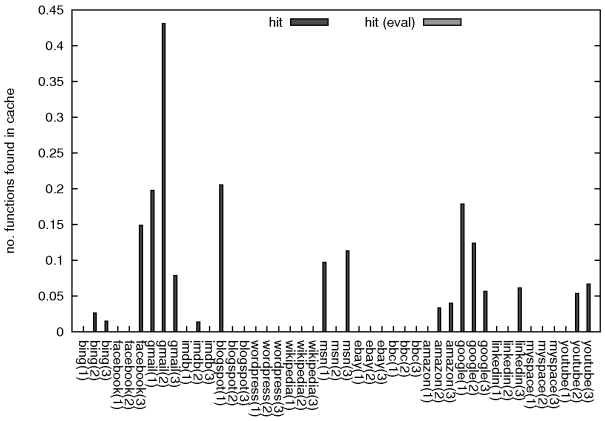

Figure 2: The normalized number of times a function is found in the cache of already compiled functions.

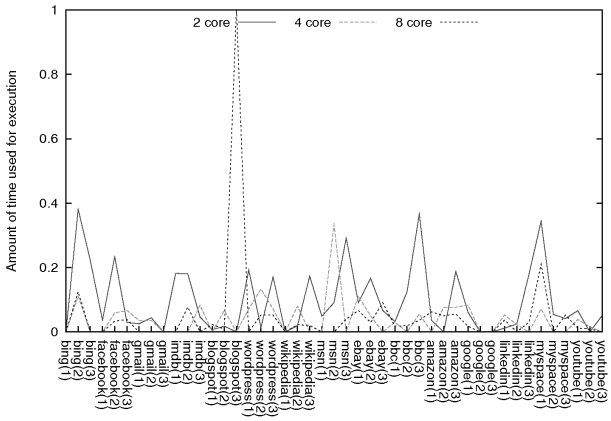

Figure 4: The thread execution time relative to the time initializing the threadpool.

interpreted JavaScript engine. For the implementation of TLS speculation in V8, we follow what we did in [8] where we speculate aggresively on function calls.

4. EXPERIMENTAL RESULTS

In Figure 3 we see the effect of running our TLS enabled version of V8 on 2, 4 and 8 core computers. We see that the average speedup for 2, 4 and 8 cores are 0.95, 1.55 and 2.17. This indicate that we need more than 2 cores to take advantage of TLS in combination with JIT in a web application. It shows that we are able to take advantage of an increasing number of cores. It also shows that when the number of cores doubles, the speedup does not. It becomes 39% faster going from 2 to 4 cores, and 25% faster going from 4 to 8. This could indicate that the speedup would decrease even further going from 8 to an even higher number of cores. It also indicates that it could be quite sufficient with 4 cores to take advantage of TLS+JIT in web applications.

One thing that could explain the behavior is the following; It executes faster with a higher number of cores, but it also takes more time to initialize the threadpool with an increased number of cores. In Figure 4 we have measured the time it takes to initialize the threadpool and the time it takes to execute the threads. We see that as the number of cores increases, the thread execution time decreases. This indicates that the time to set up the threadpool increases as the number of cores increases.

In our experiments, we have measured the execution time as the time it takes to set up the threadpool and the time it takes to execute each thread. In real-life, this is however unrealistic? It would be the equivalent of, when we visit *Facebook*, we would restart Chromium before we visited *Gmail*. This suggests that the improved speedup with Thread Level Speculation might be even higher, because setting up the threadpool is a one time expense, namly when we initialize Chromium. One use-case that differs is *blogspot (1)* because for this use-case, we only make one JavaScript call.

In Figure 5 we see the maximum and average number of threads executing concurrently. If we measure the average maximum number, the average average number of threads and the distance between them we find them to be 204.42, 123.11 and 81.31. This suggests that even though we compile the functions to be executed, the functions that are executed speculatively are relatively short in terms of number of executed instructions. There is on average difference of 40% between the maximum number of threads and the average number of threads.

Deviations are *blogspot(3)*, *wikipedia(1)* and *msn(2)*. For *blogspot(3)* and *wikipedia(1))* the number of speculations are very small, 5 and 17 respectively. We have discussed the *blogspot(3)* case where there is a limited amount of interaction. Likewise, in *wikipedia(1)*, we know from [8] that the front page of *wikipedia* has a limited amout of JavaScript interactivity. For *msn(2)*, the number of speculations are relatively close to the average number of speculations for the other use-cases, but several of the functions have are very large in terms of instructions to be executed. This makes the average number of functions executing concurrently become low compared to the maximum number of function executing concurrently. These functions have a low number of writes, which in turn makes them easy to speculate on. This could be caused by, in the *msn* use-case, where we watch news on *msn*. This use-case has several "tickers" where the functionality in terms of JavaScript is long enduring, which again creates large functions which are suitable for speculation. As we see in Figure 3 this creates one of the largest speedups with over 4 times faster than the sequential execution time of V8.

5. CONCLUSION

In this study, we have presented the first implementation of thread-level speculation in combination with just-in-time compilation. The implementation is done in the Google V8 JavaScript engine. We have evaluated the performance effects using in total 45 use-cases of popular web applications. Our results indicate that thread-level speculation can be used advantagously in combination with Just-in-time compilation, and that there are features in V8 that are useful for Thread-Level Speculation. Further, our results indicate

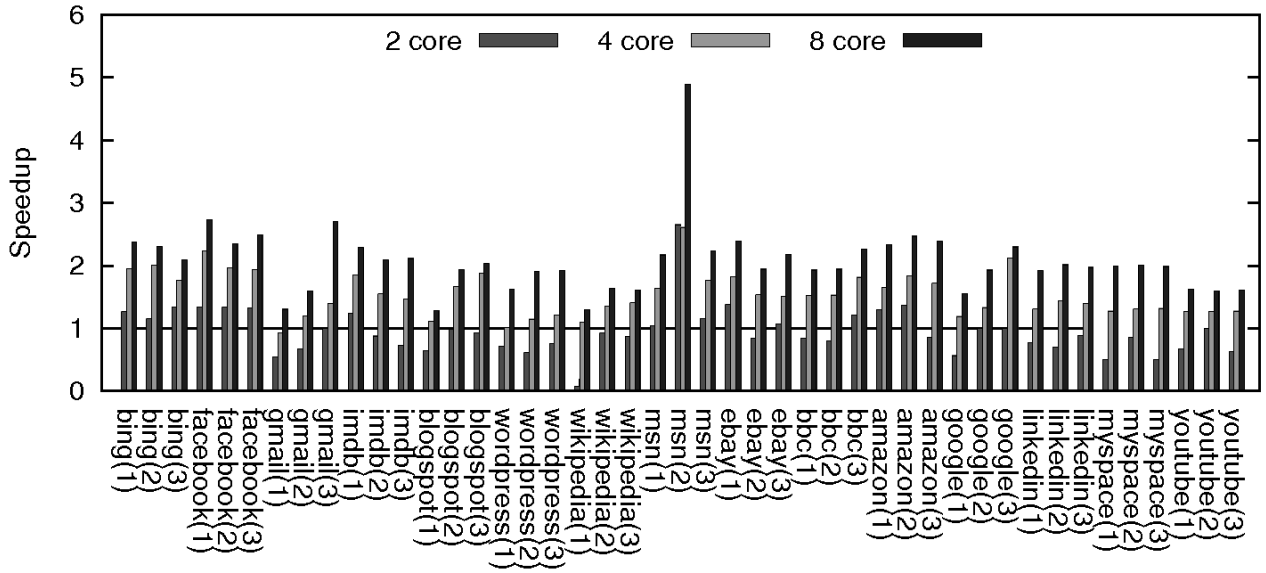

Figure 3: The relative execution time for 2, 4 and 8 cores.

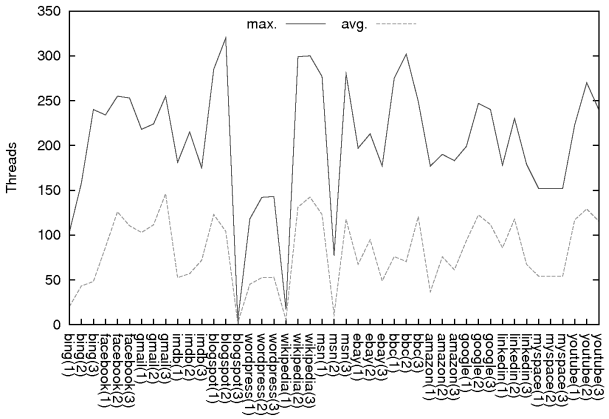

Figure 5: The maximum and the average number of threads executing concurrently.

that we need more than 2 cores to successfully be able to take advantage of TLS+JIT in web applications, and that 4 cores yield an average speedup of 1.55.

Acknowledgments

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.

6. REFERENCES

[1] Alexa. Top 500 sites on the web, 2010. `http://www.alexa.com/topsites`.

[2] J. Brand and J. Balvanz. Automation is a breeze with autoit. In *SIGUCCS '05: Proc. of the 33rd Annual ACM SIGUCCS Conf. on User services*, pages 12–15, New York, NY, USA, 2005. ACM.

[3] 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)*, pages 1–10, Dec. 2010.

[4] Google. V8 JavaScript Engine, 2012. `http://code.google.com/p/v8/`.

[5] Google. Chromium web browser, 2013. `http://www.chromium.org/`.

[6] J. K. Martinsen and H. Grahn. A methodology for evaluating JavaScript execution behavior in interactive web applications. In *Proc. of the 9th ACS/IEEE Int'l Conf. On Computer Systems And Applications*, pages 241–248, December 2011.

[7] J. K. Martinsen, H. Grahn, and A. Isberg. A comparative evaluation of JavaScript execution behavior. In *Proc. of the 11th Int'l Conf. on Web Engineering (ICWE 2011)*, pages 399–402, June 2011.

[8] J. K. Martinsen, H. Grahn, and A. Isberg. Using speculation to enhance javascript performance in web applications. *Internet Computing, IEEE*, 12(4):37–45, March 2013.

[9] Mozilla. SpiderMonkey – Mozilla Developer Network, 2012. `https://developer.mozilla.org/en/SpiderMonkey/`.

[10] P. Ratanaworabhan, B. Livshits, and B. G. Zorn. JSMeter: Comparing the behavior of JavaScript benchmarks with real web applications. In *WebApps'10: Proc. of the 2010 USENIX Conf. on Web Application Development*, pages 3–3, 2010.

[11] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *PLDI '10: Proc. of the 2010 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 1–12, 2010.

[12] P. Rundberg and P. Stenström. An all-software thread-level data dependence speculation system for multiprocessors. *Journal of Instruction-Level Parallelism*, pages 1–28, 2001.

[13] WebKit. The WebKit open source project, 2012. `http://www.webkit.org/`.