Title:
Heuristics for Thread-Level Speculation in Web Applications

Author:
Jan Kasper Martinsen, Håkan Grahn, Anders Isberg

Journal:
IEEE Computer Architecture Letters

Year:
2013
Vol:
3
Issue:
2
Pagination:
77-80
URL/DOI to the paper:
10.1109/L-CA.2013. 26

Heuristics for Thread-Level Speculation in Web Applications

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

Abstract—JavaScript is a sequential programming language, and Thread-Level Speculation has been proposed to dynamically extract parallelism in order to take advantage of parallel hardware. In previous work, we have showed significant speed-ups with a simple on/off speculation heuristic. In this paper, we propose and evaluate three heuristics for dynamically adapt the speculation: a 2-bit heuristic, an exponential heuristic, and a combination of these two. Our results show that the combined heuristic is able to both increase the number of successful speculations and decrease the execution time for 15 popular web applications.

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

1 Introduction

JavaScript is a dynamically typed, object-based scripting language with run-time evaluation, where execution is done in a JavaScript engine [1], [2], [3]. Several optimization techniques have been suggested to decrease the execution time of JavaScript, e.g., just-in-time compilation (JIT) [1], [2], [3]. Thread-Level Speculation (TLS) [4] has been proposed as an approach to take advantage of parallel hardware, both for JavaScript benchmarks [5] and web applications [6], [7].

Previous work have shown that there are significant differences in the execution behavior of JavaScript benchmarks and real-world web applications, e.g. [8], [9], [10]. One of the consequences is that JIT often increases the execution times in web applications, while TLS decreases the execution times [8], [7]. A study by Fortuna et al. [11] shows potential JavaScript speedups of up to 45 times with parallelism in web applications.

Mehrara and Mahlke [12] address how to utilize multicore systems in JavaScript engines. They target trace-based JIT-compiled JavaScript code, where the most common execution flow is compiled into an execution trace. Then, runtime checks (guards) are inserted to check whether control flow is still valid for the trace or not. They execute the runtime checks (guards) in parallel with the main execution flow (trace), and only have a single main execution flow.

In [5], a lightweight speculation mechanism is proposed that focuses on loop-like constructs in JavaScript. A loop is marked for speculation if it contains a sufficient workload. As this code used the trace feature of Spidermonkey, a selective form of speculation is employed. They found that they were able to make execution 2.8 times faster for well known JavaScript benchmarks. Unfortunately, large loop structures are rare in real web applications [8].

Mickens et al. [6] indicate an event-based speculation mechanism which is deployed as a JavaScript library called Crom. Crom clones certain regions of the JavaScript code, which then are executed speculatively. Unlike our approach, their main goal is to enhance the responsiveness, while our main goal is to reduce the JavaScript execution time by dynamically extracting parallelism.

In [7] we presented a method-level TLS implementation in Squirrelfish/WebKit with an on/off speculation principle, where a single misspeculation turns off speculation for that function.

In this paper, we propose three heuristics for dynamically adapt the speculation: a 2-bit heuristic, an exponential heuristic, and a combination of these two. We evaluate the heuristics on 15 popular web applications. Our results show that the combined heuristic is able to both increase the number of successful speculations and decrease the execution time.

2 TLS Implementation for JavaScript

We have implemented method-level TLS in the Squirrelfish JavaScript engine [2]. The speculation is done on JavaScript functions, including return value prediction, all data conflicts are detected and rollbacks are done when conflicts arise, and nested speculation is supported. The implementation and the performance of our TLS system are described in [7]. Below is a brief overview of our baseline TLS implementation.

The JavaScript execution in Squirrelfish is divided into two stages; first the JavaScript code is compiled

- *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 is with Sony Mobile Communications AB, SE-221 88 Lund, Sweden, Anders.Isberg@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.

into bytecode instructions, and then the bytecode instructions are executed. Initially, we initialize a counter $realtime$ to 0. For each executed bytecode instruction, the value of $realtime$ is increased by 1. We give the interpreter a unique id ($p\_realtime$) (initially this will be p_0).

During execution we might encounter the bytecode instruction that indicates the start of a function call. 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 TLS program (Fig. 1). We check if this function previously has been speculated 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 is 1, then the function has been speculated unsuccessfully. If the value is 0, then it has not been speculated or has been successfully speculated, and we call this position a fork point.

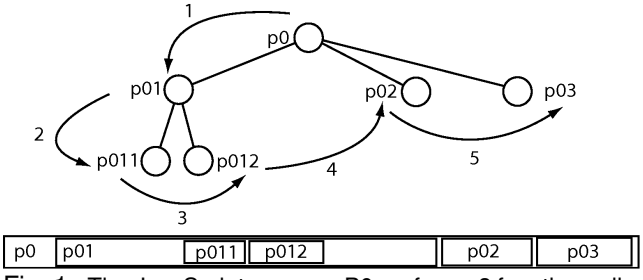

Fig. 1. The JavaScript program P0 performs 3 function calls, P01, P02, and P03, and P01 performs two function calls, P011 and P012. Thus, we have created a speculation tree. If we traverse this tree from left to right, we get the same function call order as in a sequential execution, i.e., P0, P01, P011, P012, P02, and finally P03. We denote this order as $function\_order$.

If the position of the function call is a fork point, we do the following; we set the position of the function call's $previous[function\_order] = 1$. We save the state which contains the list of previously modified global values, the list of states from each thread, the content of the register, and the content of $previous$.

We then create a new thread with a unique id. We copy the value of $realtime$ from its parent and modify the instruction pointer of the parent thread such it points to the "end of function call" bytecode instead of the position of the "function call" bytecode instruction. In other words, the parent thread skips the function call and continues to execute speculatively after 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, an incorrect return value prediction, or a write to the DOM tree we perform a rollback to the point where the speculation started.

3 HEURISTICS

In [7], we demonstrated large execution time improvements using TLS for web applications. However, when a misspeculation occurred we never re-speculated on that function again (baseline).

In Fig. 2 we present three speculation heuristics, that then are evaluated on a set of web applications (Table 1); a 2-bit heuristic, an exponential heuristic, and a combination of these two heuristics.

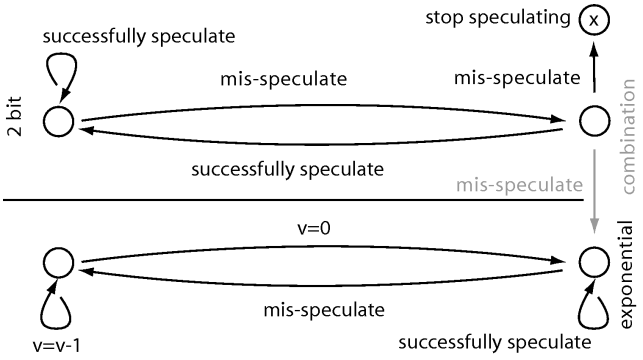

Fig. 2. The three heuristics; 2-bit, exponential, and the combination of the two. In the 2-bit heuristic, speculation on a function stops after two misspeculations in a row. In the exponential, the time between new speculation attempts doubles for each misspeculation on a function. When we combine both, we initially use the 2-bit heuristic, but after two misspeculations we switch to use the exponential heuristic.

3.1 2-bit heuristic

In the 2-bit heuristic (Fig. 2), when speculation creates rollbacks 2 times in a row on the same function, we never speculate on that function again. This could either lead to successfully speculate on a certain function, or an increased number of misspeculations.

3.2 Exponential heuristic

In the exponential heuristic, we associate the values $v = 0$ and $a = 0$, where a represents the number of misspeculations, with each potentially speculative function. If $v = 0$ we try to speculate. If speculation fails, we increase a by 1 and set $v = 2^a$. The next time we encounter this function, $v > 0$ and we do not speculate and also reduce v by 1. As we progress and the number of misspeculations increases, the value of v will increase which increases the time until we try to re-speculate on this function again. The purpose of the heuristic is to gradually decrease the probability of speculation for functions with many misspeculations.

3.3 Combined 2-bit and exponential heuristic

Instead of stop speculating after two failures in a row, like we do in the 2-bit heuristic, we continue with the exponential heuristic. This means that when we speculate enough and have enough misspeculations, it would essentially become the 2-bit heuristic.

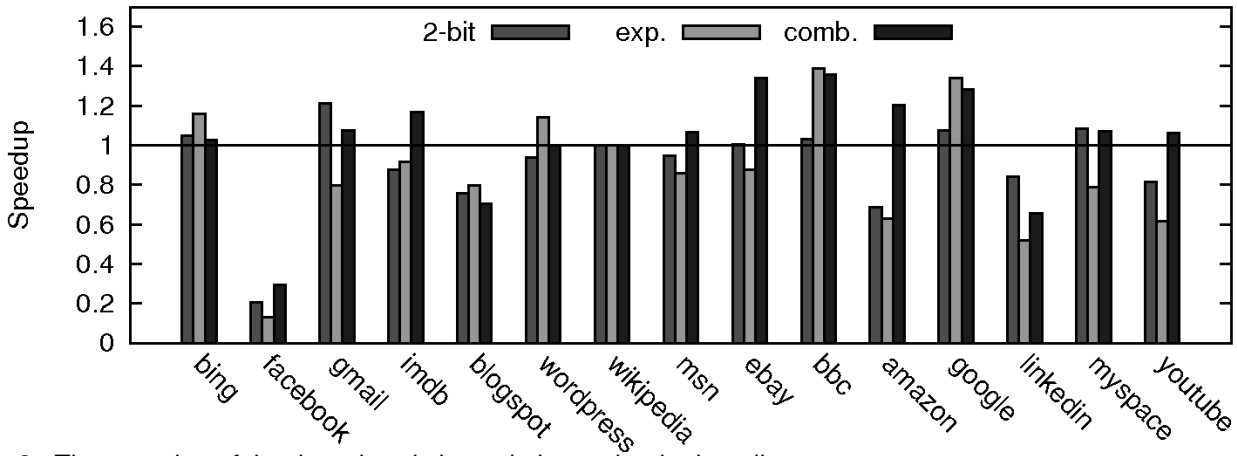

Fig. 3. The speedup of the three heuristics, relative to the the baseline

TABLE 1
The web applications 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 email client from Google

However, unlike the 2-bit heuristic, this allows us to re-speculate again after a number of subsequent speculation attempts.

4 Experimental Methodology

We have selected 15 web applications from the Alexa list of most visited web sites, as shown in Table 1. To enhance reproducibility, we use the AutoIt scripting environment to automatically execute the various use-cases in a controlled fashion. To validate the correctness of our TLS implementation we have compared the committed bytecode instructions, the return values, and all written data values in our TLS implementation with the sequential execution trace. The full methodology is described in [8].

Our experiments are executed on a system running Ubuntu 10.04 equipped with 2 quad core processors, i.e., in total 8 cores, and 16 GB main memory. We have measured the JavaScript execution time in the JavaScript engine.

5 Experimental Results

Fig. 3 shows the speedup of the heuristics normalized to the baseline's speedup. It is up to 36% faster than the baseline and the combined heuristic is faster than the baseline for 12 out of 15 cases. Comparing the speedup for the three heuristics, we observe that the combined heuristic is faster than the 2-bit and the exponential heuristic for 8 of the use cases. Fig. 6 shows the relative number of rollbacks and speculations. We see that the relation between rollbacks and speculations is much lower for the combined heuristics, than for the baseline, the 2-bit and the exponential heuristic.

For *LinkedIn*, *Amazon*, and *Facebook* the heuristics result in longer execution times than the baseline TLS. For *Facebook* we find that 43% of all function calls are regular functions calls, while *YouTube* has none. Regular JavaScript [13] function calls are in web applications more prone to misspeculations than anonymous function calls since anonymous functions often are events. With a larger number of anonymous function calls, TLS is suitable to speedup JavaScript execution in web applications. *Gmail* and *Myspace* are 5% and 1% faster with the 2-bit than with the combined heuristic. In Figure 4 the 2-bit has a higher maximum number of threads than the combined heuristic. This indicates that even though the relationship between rollbacks and speculations is lower for the combined than for the 2-bit heuristic (Figure 6), the 2-bit heuristic allows us to extract most threads without rollbacks, which again improves the execution time.

In Fig 4, 13 out of the 15 uses cases have a lower maximum number of threads than the baseline. The number of threads are lower or similar for the combined heuristic in comparison to the 2-bit and the exponential heuristics. Together with the lower number of speculations relative to rollbacks in Fig 6, this indicate that we are able to speculatively execute function calls for a longer time with the combined heuristic than for the 2-bit, the exponential heuristic and the baseline. The effect is that the probability of a rollback decreases as we re-speculate on JavaScript function calls. Further the number bytecode instructions that have to be re-executed on rollbacks de-

creases compared to the 2-bit, the exponential, and the baseline. This indicates that the nested nature of our TLS implementation allows us to gradually choose to respeculate on the right function calls, and that the number of threads that execute is different after a rollback.

There are two exceptions, i.e., *blogspot* and *linkedin*. Normally, a large number of threads are useful in order to increase the speedup. However when there is a large number of threads created from regular functions, the number of bytecode instructions that needs to be re-executed increases on a rollback. In these two cases, the number of bytecode instructions increases to the point where we were unable to speedup the execution over the baseline. *bbc* and *google* is faster with the exponential than the combination, since they have a low speculation depth, not taking advantage of nested speculation.

In Fig. 5 we show the maximum memory usage for the heuristics relative to the baseline. As in Fig. 4, we see that the combined heuristic allows the majority of the threads to execute for a longer period of time. Therefore we are making a lower number of speculations, and thereby reduce the memory with up to 53% (*ebay*).

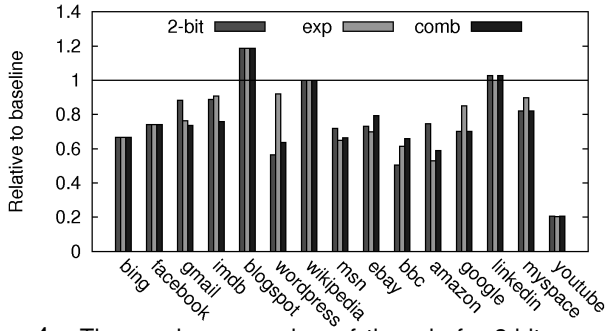

Fig. 4. The maximum number of threads for 2-bit, exponential and the combined heuristic relative to the maximum number of threads for the baseline (i.e., when no heuristic is used).

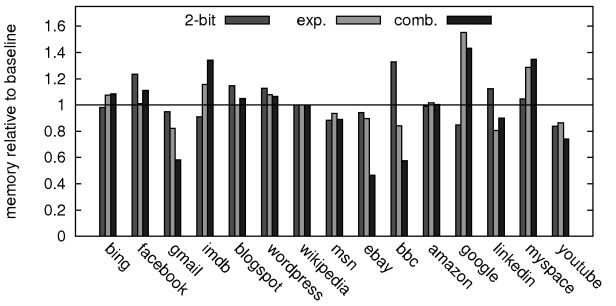

Fig. 5. The maximum amount of memory used by the heuristics relative to the baseline.

6 Conclusion

We have shown in previous work [7] that Thread-level speculation consistently improves the JavaScript performance for a selection of web applications, which

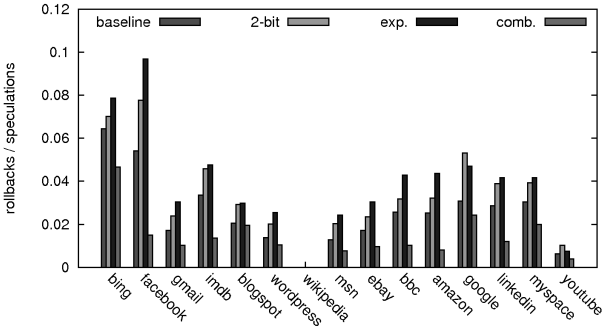

Fig. 6. The relation between rollbacks and speculations when using the baseline, the 2-bit heuristic, the exponential heuristic, and the combination of the two heuristics.

just-in-time compilation failed to do. In this paper, we present three heuristics to dynamically adjust the aggressiveness of the speculation. We evaluated them using use-cases for 15 popular web applications. Our results indicate that a combined heuristic, based on a 2-bit and an exponential, has the potential to both reduce the execution time and also increase the number of successful speculations.

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] 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.

[5] M. Mehrara, P.-C. Hsu, M. Samadi, and S. Mahlke, "Dynamic parallelization of JavaScript applications using an ultra-lightweight speculation mechanism," in *the 17th Int'l Symp. on High Performance Computer Architecture*, 2011, pp. 87–98.

[6] J. Mickens, J. Elson, J. Howell, and J. Lorch, "Crom: Faster web browsing using speculative execution," in *Proc. of the 7th USENIX Symp. on Networked Systems Design and Implementation (NSDI 2010)*, April 2010, pp. 127–142.

[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. K. Martinsen and H. Grahn, "A methodology for evaluating JavaScript execution behavior in interactive web applications," in *the 9th ACS/IEEE Int'l Conf. On Computer Systems And Applications*, Dec 2011, pp. 241–248.

[9] 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*, 2010, pp. 3–3.

[10] 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*, 2010, pp. 1–12.

[11] 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.

[12] M. Mehrara and S. Mahlke, "Dynamically accelerating client-side web applications through decoupled execution," in *the 9th Int'l Symp. on Code Generation and Optimization*, Apr. 2011, pp. 74–84.

[13] J. K. Martinsen, H. Grahn, and A. Isberg, "A limit study of thread-level speculation in javascript engines – initial results," in *Fifth Swedish Workshop on Multi-Core Computing (MCC-12)*, November 2012, pp. 75–82.