Copyright © 2013 IEEE.
Citation for the published paper:

Title:
Using speculation to enhance JavaScript performance in web applications

Author:
Jan Kasper Martinsen, Håkan Grahn, Anders Isberg

Journal:
IEEE Internet Computing

Year:
2013
Vol:
17
Issue:
2
Pagination:
10-19
URL/DOI to the paper:
10.1109/MIC.2012.146

This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of BTH's products or services Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by sending a blank email message to pubs-permissions@ieee.org.
By choosing to view this document, you agree to all provisions of the copyright laws protecting it.

Using Speculation to Enhance JavaScript Performance in Web Applications

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

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

***Abstract*—JavaScript is a programming language for client side interactivity in web applications. However, it is a sequential programming language, and cannot take advantage of multicore processors. We propose to use Thread-Level Speculation for JavaScript function calls to take advantage of multicore processors. We have implemented Thread-Level Speculation in the Squirrelfish JavaScript engine, which is part of the WebKit browser environment. Our results show that we are able to speed up the execution time by a factor of up to 8.4 times compared to the sequential version for 15 popular web applications on an 8-core computer, without making any modifications to the JavaScript source code.**

I. Introduction

JavaScript [1] is a dynamically typed, object-based scripting language where the JavaScript code is compiled to bytecode instructions at runtime which in turn are interpreted in a JavaScript engine [2], [3], [4]. One of the most common uses for JavaScript is to add interactivity to the client side part of web applications. Modern web applications are complex networks of code running on multiple servers and in the client-side web browser. Our intention with this work is to speed up the client-side execution.

Several JavaScript optimization techniques and benchmarks have been suggested, but these benchmarks have been reported as unrepresentative for JavaScript execution in web applications [5], [6], [7]. One result of this, is that the popular optimization technique just-in-time compilation (JIT) where the JavaScript code first is compiled then executed as native code decreases the execution time for JavaScript benchmarks, while it often increases the JavaScript execution time in popular web applications [8].

JavaScript is a sequential programming language and cannot take advantage of multicore processors. Fortuna et al. [9] showed that there exists significant potential parallelism in many web applications with a speedup of up to 45 times compared to the sequential execution. However, they have not implemented support for parallel execution in any JavaScript engine. Web Workers [10] allows parallel execution of tasks in web applications, but it is the programmer's responsibility to extract and express the parallelism.

To hide the details of the underlying hardware, one approach is to dynamically extract parallelism from a sequential program using Thread-Level Speculation (TLS), see sidebars "Thread-Level Speculation Principles" and "Software-Based Thread-Level Speculation". The performance potential of TLS has been shown for applications with static loops, statically typed languages, in Java bytecode environments, and recently there have been some initial attempts for JavaScript. For example, Martinsen and Grahn [11] proposed to use TLS in the Rhino JavaScript engine.

In this paper, we present an implementation of TLS in the Squirrelfish [3] JavaScript engine used in the WebKit browser environment. Our approach relies on method-level speculation, including return value prediction. We evaluate the implementation using 15 popular web applications. The execution and behaviour of a web application is dependent not only on the JavaScript engine itself, but also on the interaction between JavaScript and the web browser such as manipulation of the Document Object Model (DOM) tree. However, in this paper we deliberately focus on the JavaScript aspect.

In this work we observe and confirm that web application execution has may opportunities for speculative execution:

1) The event driven nature leads to many independent function calls that are candidates for successful Thread-Level Speculation.
2) The support and pervasive use of anonymous functions, dynamically generated un-named functions, in JavaScript programming are also good candidates for TLS.
3) Like other scripting based languages, JavaScript supports dynamic execution through primitives such as 'eval', where a string is evaluated at runtime as code. Use of this technique in web applications results in further useful TLS opportunities.

The large number of functions calls, due to events or the use of anonymous functions and 'eval' usage, can lead to a large number of TLS opportunities. However, it also means that the potential memory overheads for managing and tracking them can be significant. This is similar to why JIT based approaches can suffer excessive memory and runtime penalties for web applications execution. The

large use of event-generated dynamic functions induce many invocations of the JIT compiler and result in many JITed code fragments that must be stored, managed and eventually evicted when unfortunately they are not re-executed enough to justify the overheads.

II. JavaScript and web applications

JavaScript [1] is a dynamically typed, object-based scripting language with run-time evaluation often used to add interactivity in web applications. JavaScript execution is first compiled to bytecode instructions, which then are executed in a JavaScript engine. JavaScript has a syntax similar to C and Java, while it offers functionalities found in dynamic programming languages, such as anonymous and evaluate functions.

JavaScript engines such as Google's V8 engine [2], WebKit's Squirrelfish [3], and Mozilla's SpiderMonkey and TraceMonkey [4] have high single-threaded performance for a set of benchmarks. However, such performance results can be misleading [5], [6], [7] for web applications, and optimizing towards the characteristics of these benchmarks may increase the execution time for real-life web applications [8].

Web applications are commonly web pages with interactive functionality executed in a JavaScript engine. Web application functionality is typically defined as events. These events are defined as JavaScript functions that are executed when certain things occur in the web application, e.g., on mouse clicks, when a web page is loaded for the first time, or tasks that are executed between time intervals. In contrast to JavaScript alone, web applications might manipulate parts of the web application that are not directly accessible from a JavaScript engine alone. The functionality is simply executed in a JavaScript engine, but the program flow is part of the web application.

Previous studies have shown that the execution behavior of JavaScript in web applications differs substantially from the execution behavior of most JavaScript benchmarks. Web applications use dynamic programming language features extensively [5], [6], [7], where various parts of the program are defined at runtime (through `eval` functions), and types and extensions of objects are redefined during runtime (for instance through anonymous functions). This has, in terms of execution time, two major consequences: First, just-in-time compilation often fails to decrease the execution time. This is caused by the lack of large loop like structures in the web applications and a significant overhead in compiling small JavaScript code sequences to native code. Second, interactive structures are often defined in the web application as events. These events are executed as functions. Therefore, there will be a large number of function calls, which is a suitable workload for TLS.

III. Thread-Level Speculation Implementation for JavaScript

A. Speculation mechanism

From a high-level view the JavaScript execution in Squirrelfish can be divided into two stages, first the JavaScript code is compiled into bytecode instructions, then the bytecode instructions are executed in the JavaScript engine. We extract two things: The compiled bytecode instructions which are to be executed, and the execution trace of a sequential execution of the bytecode instructions. We use the sequential execution trace to validate the correctness of the speculative execution in our experiments off-line. The bytecode instruction execution trace contains the order in which the bytecode instructions were executed and the values associated with the computation of the bytecode instructions. We have made modifications to the Squirrelfish interpreter so an instance of it is executed as a thread.

The compiled bytecode instructions from the web application is sent to our modified Squirrelfish interpreter. 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 (a function is called after 220 bytecode instructions from p_0). We denote the value of the position of this function call as $function\_order$, which emulates the $sequential\ time$ in a TLS program (Figure 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 $function\_order$ of the respective function that tells us whether this function has been speculated on before. If the value is 1, then the function has previously been speculated unsuccessfully. If the value is 0, then it has not been speculated or has been successfully speculated, and we denote the position of the function call as a fork point.

If the position of the function call is a fork point, we do the following; We set the position of the function call's $previous[function\_order] = 1$ to make sure that we never set this as a fork point in the future, in case of a rollback. We copy the state of the JavaScript engine right before the fork point, which will be used in case of a rollback. This state contains the following: The list of previously modified global values, the list of states from each thread, the content of the registers, and the content of $previous$.

Then we create a new (or reuse an old) thread which contains a new Squirrelfish engine. We create a unique id for this thread. In addition, we copy the value of $realtime$ from its parent. We modify the state of the parent such that the current instruction is changed from the position of the

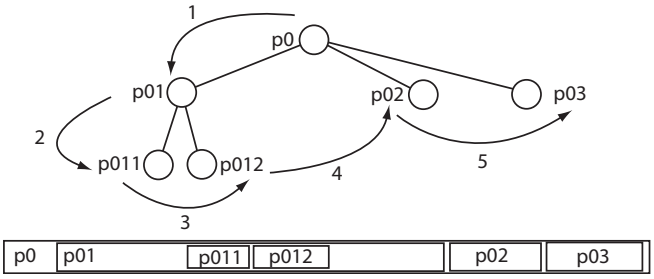

Figure 1. The JavaScript program P0, performs 3 function calls, P01, P02 and P03. P01 performs two function calls, P011 and P012. Thus, we have created a speculation tree from the function calls. If we traverse this tree from left to right, we get an order in which the functions are called, equal to the order that the functions would be sequentially called. More specific: P0 at time 1, P01 at time 2, P011 at time 3, P012 at time 4, P02 at time 5, and P03 at time 6. We denote how each function is ordered as $function\_order$.

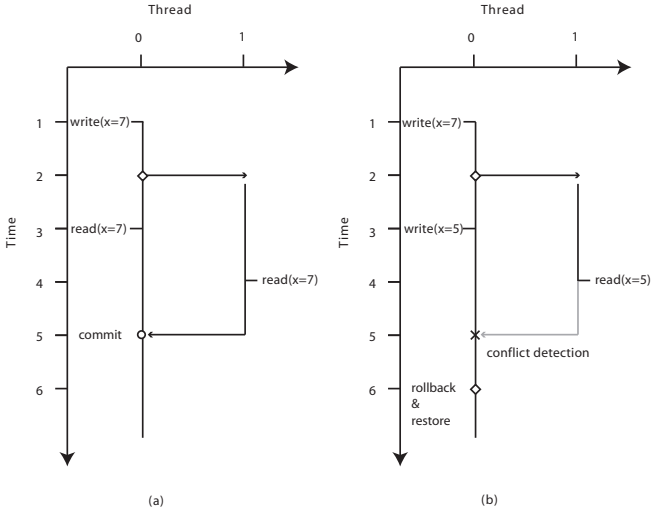

Figure 2. We have illustrated an example of a successful speculation (a) and an usuccessful one that force us to do a rollback to preserve sequential semantics (b). In (a) at time 1 in thread 0 we write the value 7 to the global value x. At time 2 in thread 0 we encounter a function which we speculate on and it becomes thread 1. At time 3, thread 0 reads 7 from the global variable x. At time 4 thread 1 reads the same variable. At time 5 the function which is thread 1, returns and the global variables are committed back to thread 0. In (b) we write 7 to x in thread 0 at time 1. At time 2 thread 0 makes a function calls that becomes thread 1. At time 3 thread 0 writes a value 5 to global variable x. At time 4 thread 1 reads 5 from the variable x. In the sequential case, the function called at time 2, the value of variable x would have been 7 at time 2, and thread 0 would write 5 to x first after the function has returned. This means that thread 1 is squashed, and when we commit the values at time 5 we can no longer ensure sequencial semantics (which is detected outside of the mechanisms described in section III-B). Therefore at time 6, we need to restore the JavaScript parent to the point before we forked the function and do not speculate on this function call.

"function call" bytecode instruction to the position of the associated "end of 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 interpreters running as concurrent threads, and this process is repeated for each function encountered which could be a suitable candidate for speculation, thereby allowing nested speculation. If there is a conflict between two global variables, an incorrect return value prediction or writing to the DOM tree we perform a rollback to the point where the speculation started. In Figure 2, we outline the process of speculation and a subsequent rollback to restore the execution to a safe state, i.e., commit or where the speculation started.

B. Data dependence violation detection

In order to achieve a correct speculative execution, we check for write and read conflicts between global variables, object property id names, and unsuccessful return value predictions of function calls. Each global variable has a unique identification, uid, which is either the index of the global variable or the name of the id in the object property.

When we encounter a read or write bytecode instruction, we check the global list $variable\_modification$. This is a list that contains previous reads and writes for all $uids$ sorted per uid. If the uid is not in the list, we lock $variable\_modification$, insert uid in $variable\_modification$, create a sublist for reads and writes to that uid, and insert the type of bytecode instruction, $realtime$, and $function\_order$ as the first element of the sublist. If there were no conflicts between the current executed bytecode and previous reads and writes of the uid, we insert an element to the head of the sublist for this uid with the type of bytecode, $realtime$, and $function\_order$.

Each time we encounter a read or write access to a uid, we evaluate the following cases in $variable\_modification$:

(i) The current operation is a read, and there is a previous read to the same uid. In this case, the order in which the uid is read does not matter.

(ii) The current operation is a read, and there is a previous write to the same uid. In this case, we must check the $realtime$ and the $function\_order$ for the current read and the previous write. If a read occured such that $current\ function\_order > previous\ function\_order$ and $current\ realtime < previous\ realtime$ then the execution order of the program is no longer correct and we must do a rollback.

(iii) The current operation is a write, and there is a previous read to the same uid. In this case, we check the $realtime$ and the $function\_order$ from the current write and the previous read. If $current\ function\_order > previous\ function\_order$ and $current\ realtime < previous\ realtime$, then the execution order of the program is no longer correct and we must do a rollback.

(iv) The current operation is a write, and there is a pre-

vious write to the same uid. We need to do a rollback if the current write happens before the previous write in $realtime$ and they have the other order in $function\_order$, or if the order of the write happens after the previous write in $realtime$ but before the previous write in $function\_order$.

C. Rollback

Cases (ii), (iii), and (iv) force us to do a rollback to ensure program correctness globally. We also do rollbacks if we write to the DOM tree. After a rollback, the program is re-executed from a point before the function was speculated. At this point information for relevant threads are extracted, e.g., $previous$ at this point, the number of associated threads at this point, the values of the associated registers, the values of the global variables and id are restored for the associated threads, the value of $previous$ (with the index of this failed speculation set to 1), and the variable conflicts in $variable\_modification$.

Even though we have a set of threads that are supposed to be active, there might be threads after the rollback that are not associated with the current state of the TLS system. Therefore, we need to recursively go through the threads and their child threads that are now part of the active state. The resulting list contains the threads which are necessary in the current state of execution. The remainder of the threads and their associated interpreter are stopped and set to an idle state for later reuse.

D. Commit

When a speculative thread reaches its end of execution, its modifications of global variables and object property ids need to be committed back to its parent thread. The commit cannot be completed before child threads from this thread have returned and have committed their values back to their parent thread. These threads are denoted as child threads, and their updates to global variables and object property ids are to be committed to the current thread. If the associated JavaScript function has a return value, which we fail to predict correctly, or if executing the function causes violations to the sequential semantics, we have to rollback.

IV. Experimental Methodology

We have selected 15 web applications from the Alexa list [12] of most visited web applications. We selected popular web applications to cover different types of web applications, while making sure that these were being used by a reasonably large group of users. The selected applications along with short descriptions are found in Table I.

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 and then executed them in WebKit. To enhance reproducibility, we use the AutoIt scripting environment [13] to automatically execute the various use-cases in a controlled fashion. The methodology for the experiments is described in [7].

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 I
List of web applications used in this study, listed from the most popular (Google) to least popular (Gmail) but all within the top 100 web applications [12].

All experiments are conducted on a system running Ubuntu 10.04 and equipped with dual quad-core processors (i.e., in total 8 cores) and 16 GB main memory. We have measured the execution time of the JavaScript execution performed in the JavaScript engine. We executed each use case ten times and take the median of the results for comparison.

To validate the correctness of our TLS implementation we have done the following: We have compared the executed bytecode instructions with the committed bytecode instructions in our TLS implementation and compared the return values and the written values against the sequential execution trace.

V. Experimental Results

A. Speedup with TLS and JIT

We compare the JavaScript execution times of a set of web applications with TLS, the Google V8 JIT engine, or the JIT enabled Squirrelfish engine to the sequential execution time on the unmodified interpreted Squirrelfish engine, as in Equation 1.

$$\frac{T_{exe}(sequential\ execution\ time)}{T_{exe}(with\ TLS\ or\ with\ JIT\ or\ with\ Google\ V8)} \quad (1)$$

In Figure 3 we see that:

- *TLS always decreases the execution time*
- *The Squirrelfish JIT based engine increases the execution time* for 10 out of 15 cases (similar to the results in [14])
- *The Google V8 JIT engine increases the execution time* for 8 out of 15 cases

YouTube executes up to 8.4 times faster than the sequential execution on a dual quad core computer when TLS is enabled. We have verified that the superlinear speedup is due to cache effects. This makes sense by looking at the large number of threads and low number of rollbacks (Table II). We see that a large number of speculations and a low number of rollbacks are typical for all 15 web applications. However, the improved execution time varies between the web applications.

B. General execution behavior

In Table II we have collected a set of numbers related to the execution behavior of a set of web applications when we use TLS.

The results show that we are able to speculate on a very large number of JavaScript functions for all the web applications except Wikipedia, and in most cases we speculate well over a thousand times. This indicates that there is a high potential parallelism in web applications, which is in line with [9]. We also observe that the *number of rollbacks* during execution is small as compared to the number of speculations. In general, we speculate a large number of times for each rollback. For 13 out of 15 web applications we speculate between 16 and 92 times for each rollback.

We have measured the largest number of functions that are executed concurrently during the execution, i.e., the *maximum number of threads*. We observe that in general we are able to execute a large number of threads concurrently, e.g., 7 of the web applications execute more than 50 threads concurrently. We have also measured the deepest nesting speculation during execution, i.e., the *maximum speculation depth*. Our results show no obvious relationship between the maximum speculation depth and the maximum number of threads.

The *average depth* shows the depth of the recursive search when we complete the execution of a speculated function (either after a rollback or when we commit the result of the execution of a function). In order to find the information to evict, we need to do a recursive search in the speculation tree, and we have measured the average depth for these searches. We have not observed any relationship between the maximum speculation depth and the average depth we need to search to remove information.

Finally, we have measured the *average memory usage* for storing information during speculation by measuring it at each rollback. In general, the memory requirements for storing data associated with the speculation are relatively small, but we have observed high peaks of memory usage (up to 300 MB).

We know from previous research [5], [6], [7] that JavaScript execution in web applications is to a large extent event driven, and that these events are visible to the JavaScript engine as function calls. This behavior is visible as a large number of function speculations.

Our results indicate that even though there is a risk for write and read conflicts, these rarely happen for the use cases. We can see this by a large number of speculations, and a small number of rollbacks, i.e., the relative number of speculations over rollbacks is high. Another indication of the lack of write and read conflicts is the large number of threads that we are able to execute concurrently. Due to the large number of speculations and low number of rollbacks, we are able to execute a relatively large number of functions concurrently (*maximum number of threads*).

From the maximum speculation depth we see that a large speculation depth does not necessarily mean a large number of threads. There is also no relationship between a large search depth and the amount of data we clear out after a rollback or a commit. We also see that the average memory usage varies between the cases, and there is no clear relationship between average memory usage and a large speculation depth.

VI. Concluding remarks

JavaScript is a sequential language and cannot take advantage of multicore processors. Our approach is to dynamically identify and extract parallelism using Thread-Level Speculation (TLS).

In this paper, we have presented and evaluated an implementation of TLS in the Squirrelfish JavaScript engine [3]. We speculate at function level and support nested speculation, i.e., a function that is executing speculatively can create new speculatively executed functions. Our evaluation is based on 15 popular web applications from the Alexa top list [12].

Our TLS implementation improves the performance for all studied web applications. In contrast, we found that just-in-time compilation decreases the performance for 10 out of 15 web applications. Our results show that for the selected web applications, TLS significantly reduces the execution time. Speedups of up to 8.4 times were achieved compared to a sequential execution. *The performance improvements are achieved without modifying any of the JavaScript source code* (thus hiding the details of taking advantage of multicore processors from the JavaScript programmer). Our results show a large number of speculations with few rollbacks.

We have also evaluated how nested speculation works. The maximum speculation depth ranges from 4 (Wikipedia) to 99 (Wordpress), while the average depth when we search for data made irrelevant after a rollback or a commit goes up to 9.16 (Facebook). Our results indicate that we can find a large number of function calls, which will be a suitable workload for TLS. Further, nested speculation is important in order to find a sufficient number of suitable functions for speculation, and thus achieving a high degree of dynamic parallelism. Since speculation requires the state of the JavaScript engine to be stored at the speculation point, the memory overhead sometimes can be large.

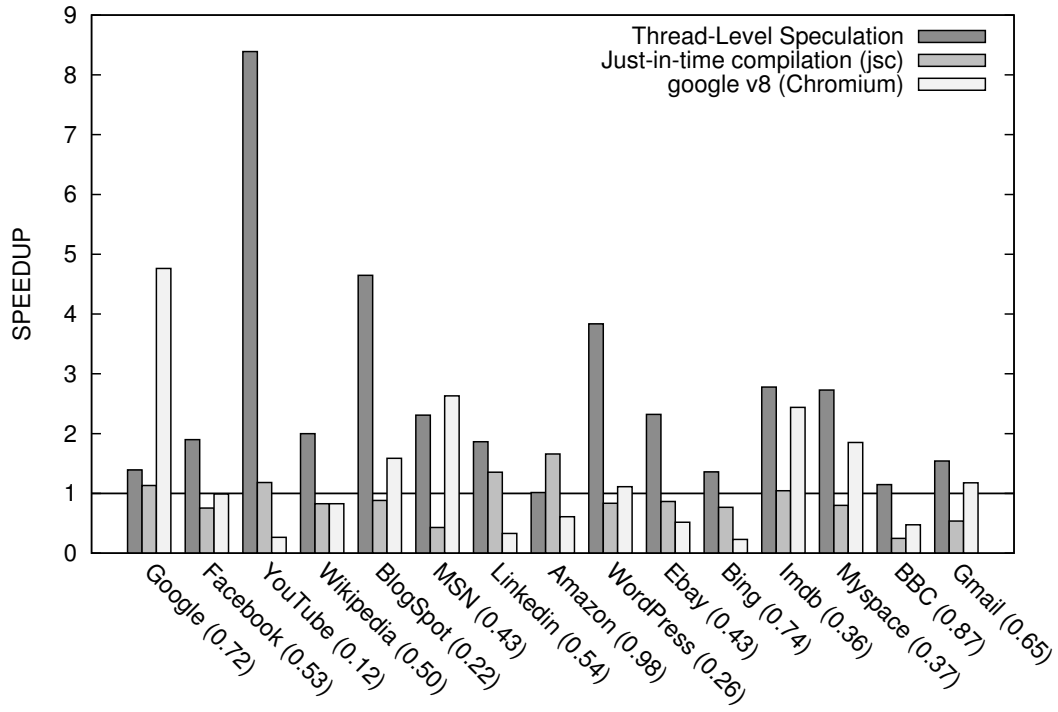

Figure 3. JavaScript execution speedup over the interpreted sequential JavaScript engine for the 15 web applications when TLS is enabled, when just-in-time compilation is enabled and for the Google v8 JIT engine in the chromium web browser. We have written the execution time of TLS relative to the sequential execution time in parenthesis after the name of the web applications.

ACKNOWLEDGMENT

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

REFERENCES

[1] JavaScript, "http://en.wikipedia.org/wiki/JavaScript," 2010.

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

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

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

[5] 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.

[6] 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.

[7] 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*, December 2011, pp. 241–248.

[8] 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)*, June 2011, pp. 399–402.

[9] 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.

[10] W3C, "Web Workers — W3C Working Draft 01 September 2011," Sep. 2011, http://www.w3.org/TR/workers/.

[11] J. K. Martinsen and H. Grahn, "An alternative optimization technique for JavaScript engines," in *Third Swedish Workshop on Multi-Core Computing (MCC-10)*, November 2010, pp. 155–160.

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

[13] 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*. New York, NY, USA: ACM, 2005, pp. 12–15.

Application	Number of speculations	Number of rollbacks	Speculations / Rollbacks	Maximum number of threads	Max speculation depth	Average depth	Memory usage (MB)
Amazon	10768	267	40.31	83	23	8.0	14.1
BBC	6392	154	41.51	117	14	5.12	33.0
Bing	303	18	16.83	30	7	2.22	1.4
Blogspot	778	15	51.87	16	14	2.16	1.6
Ebay	7140	101	70.69	63	15	5.33	27.0
Facebook	968	51	18.98	27	22	9.16	7.1
Gmail	1193	19	62.79	34	10	2.68	1.95
Google	1282	36	35.61	40	10	3.9	5.5
Imdb	5300	156	33.97	54	24	6.85	17.8
LinkedIn	1815	51	35.59	36	11	2.27	7.1
MSN	12012	133	90.32	191	24	5.85	20.1
Myspace	3679	93	39.56	39	14	5.54	17.4
Wordpress	5852	63	92.89	63	99	4.55	9.7
Wikipedia	12	0	$undefined$	8	4	0	1.1
YouTube	7349	25	293.96	407	13	5.44	17.1

Table II
NUMBER OF SPECULATIONS, NUMBER OF ROLLBACKS, RELATIONSHIP BETWEEN ROLLBACKS AND SPECULATIONS, MAXIMUM NUMBER OF THREADS, MAXIMUM NESTED SPECULATION DEPTH, AVERAGE DEPTH FOR RECURSIVE SEARCH WHEN DELETING VALUES ASSOCIATED WITH PREVIOUS SPECULATIONS, AND AVERAGE MEMORY USAGE BEFORE EACH ROLLBACK (IN MEGABYTES).

[14] J. K. Martinsen, H. Grahn, and A. Isberg, "Evaluating four aspects of JavaScript execution behavior in benchmarks and web applications," Blekinge Institute of Technology, Research Report 2011:03, July 2011.

SIDEBAR: THREAD-LEVEL SPECULATION PRINCIPLES

TLS aims to dynamically extract parallelism from a sequential program. This can be done both in hardware, e.g., [1], [2], [3], and software, see sidebar "Software-Based Thread-Level Speculation". Two main approaches exist: Loop level parallelism and method level speculation.

Loop level is a popular approach where each loop iteration is assigned to a thread. Then, we can (ideally) execute as many iterations in parallel as we have processors. However there are limitations, data dependencies may limit the number of iterations that can be executed in parallel. Further, the memory requirements and run-time overhead for detecting data dependencies can be considerable.

In method level we try to execute each function call as a thread. With this approach, in addition to loop level parallism we need to correctly predict the return values when we speculate and the writes and reads that cause the speculative program to violate sequential semantics. The last two are typically detected when the values associated with two function calls are commited back to its parent thread.

Between two speculative threads we can have three types of data dependencies: *Read-After-Write* (RAW), *Write-After-Read* (WAR), and *Write-After-Write* (WAW). A TLS implementation must be able to detect these dependencies during run-time using information about read and write addresses from each loop iteration. A key design parameter for a TLS system is *precision* of what granularity it can detect data dependency violations.

When a data dependency violation is detected, the execution must be aborted and rolled back to a safe point in the execution. Thus, all TLS systems need a rollback mechanism. In order to be able to do rollbacks, we need to store both speculative updates of data as well as the original data values. As a result, the book-keeping related to this

functionality results in both memory overhead as well as run-time overhead. In order for TLS systems to be efficient, the number of rollbacks should be low.

A key design parameter for a TLS system is the data structures used to track and detect data dependence violations. The more precise tracking of data dependencies, the more memory overhead is required. Unfortunately, one effect of imprecise dependence detection is the risk of a false-positive violation, i.e., when a dependence violation is detected when no actual (true) dependence violation is present. As a result, unnecessary rollbacks need to be done, which decreases the performance. TLS implementations can differ depending on whether they update data speculatively 'in-place', i.e., moving the old value to a buffer and writing the new value directly, or in a special speculation buffer.

References Thread-Level Speculation

[1] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay, "Rock: A High-Performance Sparc CMT Processor," *IEEE Micro*, vol. 29, no. 2, pp. 6–16, 2009.

[2] J. Renau, K. Strauss, L. Ceze, W. Liu, S. R. Sarangi, J. Tuck, and J. Torrellas, "Energy-efficient thread-level speculation," *IEEE Micro*, vol. 26, no. 1, pp. 80–91, 2006.

[3] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry, "The STAMPede approach to thread-level speculation," *ACM Transactions on Computer Systems*, vol. 23, no. 3, pp. 253–300, 2005.

Sidebar: Software-Based Thread-Level Speculation

There exist a number of different software-based TLS proposals, and we review some of the most important ones here.

Bruening et al. [1] proposed a software-based TLS system that targets loops where the memory references are stride-predictable. Further, it is one of the first techniques that is applicable to while-loops where the loop exit condition is unknown until the last iteration. They evaluate their technique on both dense and sparse matrix applications, as well as on linked-list traversals.

Rundberg and Stenström [2] proposed a TLS implementation that resembles the behaviour of a hardware-based TLS system. The main advantage with their approach is that it tracks data dependencies precisely, thereby minimizing the number of unnecessary rollbacks caused by false-positive violations. The downside is the high memory overhead.

Kazi and Lilja developed the course-grained thread pipelining model [3] exploiting coarse-grained parallelism. They suggest to pipeline the concurrent execution of loop iterations speculatively, using run-time dependence checking.

Bhowmik and Franklin [4] developed a compiler framework for extracting parallel threads from a sequential program for execution on a TLS system. They support both speculative and non-speculative threads, and out-of-order thread spawning. Further, their work addresses both loop as well as non-loop parallelism.

Chen and Olukotun [5] have studied how method-level parallelism can be exploited using speculative techniques. The idea is to speculatively execute method calls in parallel with code after the method call. Their techniques are implemented in the Java runtime parallelizing machine (Jrpm).

Picket and Verbrugge [6] developed SableSpMT, a framework for method-level speculation and return value prediction in Java programs. Their solution is implemented in a Java Virtual Machine, called SableVM, and thus works at the bytecode level.

Oancea et al. [7] present a novel software-based TLS proposal that supports in-place updates. Further, their proposal has a low memory overhead with a constant instruction overhead, at the price of slightly lower precision in the dependence violation detection mechanism. However, the scalability of their approach is superior due to the fact that they avoid serial commits of speculative values, which in many other proposals limit the scalability.

Mickens et al. introduce Crom [8], a JavaScript speculation engine. Crom rewrite event handles to speculative ones and execute them in a cloned browser context. Crom is implemented in a JavaScript library and runs on unmodified JavaScript engines. Speculative event handlers are mainly executed when the main execution thread is idle.

Mehrara et al. have addressed how to utilize multicore systems in JavaScript engines [9] as well as a lightweight spec-

ulation mechanism for dynamic parallelization of JavaScript applications [10]. However, their studies have a different approach as well as a different target than we have. They target mainly loop parallelization, and 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 etc. is still valid for the trace or not. They execute the runtime checks (guards) in parallel with the main execute flow (trace).

REFERENCES RELATED WORK

[1] D. Bruening, S. Devabhaktuni, and S. Amarasinghe, "Softspec: Software-based speculative parallelism," in *FDDO-3: Proceedings of the 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization*, 2000.

[2] 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.

[3] I. H. Kazi and D. J. Lilja, "Coarse-grained thread pipelining: A speculative parallel execution model for shared-memory multiprocessors," *IEEE Trans. on Parallel and Distributed Systems*, vol. 12, no. 9, pp. 952–966, 2001.

[4] A. Bhowmik and M. Franklin, "A general compiler framework for speculative multithreading," in *SPAA '02: Proc. of the 14th ACM Symp. on Parallel Algorithms and Architectures*, 2002, pp. 99–108.

[5] M. K. Chen and K. Olukotun, "The Jrpm system for dynamically parallelizing Java programs," in *ISCA '03: Proc. of the 30th Int'l Symp. on Computer Architecture*, 2003, pp. 434–446.

[6] C. J. F. Pickett and C. Verbrugge, "SableSpMT: a software framework for analysing speculative multithreading in java," in *PASTE '05: Proc. of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2005, pp. 59–66.

[7] C. E. Oancea, A. Mycroft, and T. Harris, "A lightweight in-place implementation for software thread-level speculation," in *SPAA '09: Proc. of the 21st Symp. on Parallelism in Algorithms and Architectures*, August 2009, pp. 223–232.

[8] 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.

[9] M. Mehrara and S. Mahlke, "Dynamically accelerating client-side web applications through decoupled execution," in *Proc. of the 9th Annual IEEE/ACM Int'l Symp. on Code Generation and Optimization (CGO)*, april 2011, pp. 74–84.

[10] 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.