CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2014; **00**:1–20
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Combining Thread-Level Speculation and Just-In-Time Compilation in Google's V8 JavaScript Engine

Jan Kasper Martinsen*, Håkan Grahn†‡ and Anders Isberg§

Blekinge Institute of Technology, Karlskrona Sweden and Sony Mobile, Lund, Sweden

SUMMARY

Thread-Level Speculation can be used to take advantage of multicore architectures for JavaScript in web applications. We extend previous studies with these main contributions; we implement Thread-Level Speculation in the state-of-the art Just-in-time enabled JavaScript engine V8 and make the measurements in the Chromium web browser both from Google instead of using an interpreted JavaScript engine. We evaluate the Thread-Level Speculation and Just-in-time compilation combination on 15 very popular web applications, 20 HTML5 demos from the JS1K competition, and 4 Google Maps use cases. The performance is evaluated on 2, 4, and 8 cores. The results clearly show that it is possible to successfully combine Thread-Level Speculation and Just-in-time compilation. This makes it possible to take advantage of multicore architectures for web applications while hiding the details of parallel programming from the programmer. Further, our results show an average speedup for the Thread-Level Speculation and Just-in-time compilation combination by a factor of almost 3 on 4 cores and over 4 on 8 cores, without changing any of the JavaScript source code.
Copyright © 2014 John Wiley & Sons, Ltd.

Received . . .

1. INTRODUCTION

JavaScript is a sequential, dynamically typed, object based scripting language with run-time evaluation typically used for clientside interactivity in web applications. Several optimization techniques have been suggested to speedup the execution time [11, 37, 26]. 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 [18, 29, 32]. A result of this difference is that optimization techniques such as Just-in-time compilation (JIT) often increase the JavaScript execution time in web applications [19].

Fortuna et al. [10] have shown that there is a significant potential for parallelism in many web applications with a speedup of up to 45× the sequential execution time. To take advantage of this observation, and to hide the complexity of parallel programming from the JavaScript programmer, one approach is to dynamically extract parallelism from a sequential program using Thread-Level Speculation (TLS) [33].

*E-mail: jan.kasper.martinsen@bth.se
†Correspondence to: Professor Håkan Grahn, Department of Computer Science and Engineering, Blekinge Institute of Technology, Karlskrona, Sweden
‡E-mail: hakan.grahn@bth.se
§E-mail: anders.isberg@sonymobile.com

Copyright © 2014 John Wiley & Sons, Ltd.

A lightweight speculation mechanism for JavaScript that focuses on loop-like constructs is introduced [23], and gives speedups by a factor of 2.8. Another approach increases the responsiveness of web applications [25].

Our goal is to reduce the JavaScript execution time by dynamically extracting parallelism. In [21], a TLS implementation with the Squirrelfish JavaScript engine (without JIT enabled) gave significant speedups to the execution time compared to the sequential execution time on a dual quadcore computer.

We present the first method-level Thread-Level Speculation implementation (TLS) in a JavaScript engine which supports Just-in-time compilation (JIT) and where we measure the execution time on a range of web applications, the Google maps application and HTML5 demos. We have not found any other studies with the TLS+JIT combination. The implementation is done in Google's V8 JavaScript engine that only supports JIT. We perform the experiments in the Chromium web browser and execute use cases to mimic normal usage in the web applications. The motivation for using this JavaScript engine is that the results in [19] show that even though the workload in web applications often is not suitable for JIT, Figure 1 shows that the V8 JavaScript engine is often faster than the Squirrelfish engine with JIT enbabled.

We measure the execution time of our TLS+JIT combination on 45 use cases from 15 popular web applications. Further, in order to evaluate our implementation on a wider range of applications, we also evaluate our implementation on 20 HTML5 demos from the JS1K competition (`http://js1k.com/`), and 4 use cases from Google Maps web application. The experiments are evaluated on 2, 4, and 8 cores on an dual quad core computer (8 cores) where we can disable the number of cores to 2 and 4.

The results of this study show that we need more than 2 cores to always improve the execution time with the TLS+JIT combination. The average speedup is 2.9 when running on 4 cores and *on average 4.7 when running on 8 cores without any changes to the sequential JavaScript code* on a set of very popular web applications. We find an average speedup to be between 2.11 and 2.98 on 4 and 8 cores on a set of HTML5 demos, and an average speedup of 3.54 and 4.17 on 4 and 8 cores for the Google maps web application.

Web applications use an event driven execution model, such that JavaScript is executed in such a way that TLS can be used with JIT, and still reduce the execution time. In addition the V8 JavaScript engine has features that are advantageous for Thread-Level Speculation and therefore it is suitable to be combined with TLS.

This paper is organized as follows; in Section 2 we introduce JavaScript and web applications, as well as principles and previous work on Thread-Level Speculation. In Section 3, we present our implementation of Thread-Level Speculation in the V8 JavaScript engine. In Section 4 we present the experimental methodology as well as the 45 web application use cases, the 20 HTML5 use cases, and the 4 use cases for Google Maps. In Section 5 we present the experimental results of running the use cases on 2, 4 and 8 cores, and discuss the results of these measurements. Finally, in Section 6 we conclude our findings.

2. BACKGROUND AND RELATED WORK

2.1. JavaScript and web applications

JavaScript [15] is a dynamically typed, object-based scripting language often used for interactivity in web applications with run-time evaluation. JavaScript application execution is done in a JavaScript engine and it has a syntax similar to C and Java, while it offers functionalities often found in functional programming languages, such as closures and *eval* functions [31].

The performance of popular JavaScript engines such as Google's V8 engine [11], WebKit's Squirrelfish [37], and Mozilla's SpiderMonkey and TraceMonkey [26] has increased during the last years, reaching a higher single-thread performance for a set of benchmarks. It has been shown that the results from these benchmarks is misleading [18, 29, 32] for real life web applications, and

Copyright © 2014 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

that optimizing towards the characteristics of the benchmarks may increase the execution time in web applications [19, 21].

Web applications are commonly web pages where interactive functionalities are executed in a JavaScript engine. Web applications' functionalities are typically defined as events. These events are defined as JavaScript functions that are executed for instance when the user clicks a mouse button, a web page loads 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 scripted functionality is executed in a JavaScript engine, but the program flow is defined in the web application.

Previous studies show that web applications use JavaScript specific programming language features extensively [18, 29, 32]. For instance, various parts of the program are defined at run-time (through the use of *eval* functions), and the types and the extensions of objects are extensively re-defined during run-time (for instance through anonymous functions).

A key concept in web applications is the Document Object Model (DOM). DOM is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. The document can be further processed and the results of that processing can be incorporated back into the presented page. The programmer of the web application can modify, create, or delete elements and content in the web applications through the DOM tree with JavaScript.

2.2. *Thread-level speculation principles*

TLS aims to dynamically extract parallelism from a sequential program. This has been done both in hardware, e.g., [6], and in software, e.g., [28, 33].

One popular approach is to allocate each loop iteration to a thread. Another method is method-level speculations, where the underlying system tries to execute function calls as threads. Then, we can (ideally) execute as many iterations or function calls in parallel as we have processors. However data dependencies may limit the number of iterations and function calls that can be executed in parallel. Further, the memory requirements and overhead for detecting data dependencies can be considerable.

Between two consecutive loop iterations or between access to global variables in two function calls, we can have three types of data dependencies: *Read-After-Write* (RAW), *Write-After-Read* (WAR), and *Write-After-Write* (WAW). In addition, when we speculate on function calls, we must be able to speculate on their return value upon speculation. 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 the *precision* of at what granularity of true-positive / (true-positives + false-positives) 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.

Another important factor in 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-positive) dependence violation is present. As a result, unnecessary rollbacks need to be done, which decreases the performance and increases the execution time. 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.

Copyright © 2014 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

2.3. Software-based Thread-Level Speculation

There exist a number of different software-based TLS proposals. In this paper, we review and divide them into three subgroups; general software Thread-Level Speculation, Thread-Level Speculation in virtual machines, and Thread-Level Speculation in JavaScript engines. We present the general software based system in Table I where they majority of the solutions support either C, FORTRAN or Java.

2.4. General software-based Thread-Level Speculation

Table I. Examples of previous work on software Thread-Level Speculation.

Author	Speedup	#cores	Benchmark	Summary	Language
Chen and Olukotun show [7, 8]	3.7–7.8	16	SPECjvm98 [34] (with others)	method-level	C and JAVA
Bruening et al. [5]	over 5	8	SPEC95FP and SPEC92	applicable on while loops	C
Rundberg and Stenström [33]	10	16	Perfect Club Benchmarks [2]	minimizing the overhead	C
Kazi and Lilja [16]	5–7	8	Perfect Club Benchmarks [2]	exploit coarse-grained parallelism	C
Bhowmik and Franklin [3]	1.64–5.77	6	SPEC CPU95, SPEC CPU2000, and Olden	exploits loop and non-loop parallism	C
Cintra and Llanos [9]	16	16	SPEC CPU2000 [35] and Perfect Club [2]	using a sliding window with loops	C
Renau et al. [30]	3.7–7.8	16	SPEC CPU2000 Benchmarks [35]	method-level	C
Picket and Verbrugge [27, 28]	2	4	SPECjvm98 [34]	method-level	JAVA
Kejariwal et al. [17]	2.5		SPEC CPU2000 Benchmarks [35]	Theoretical study	C and FORTRAN
Hertzberg and Olukotun [13]	2.04	4	SPEC CPU2000 Benchmarks [35]	method-level	C
Hertzberg and Olukotun [13]	3–4, 2–3, and 1.5–2.5	4	SPEC CPU2006 Benchmarks [36]	method-level	JAVA

2.5. TLS in JavaScript and web applications

Mehrara and Mahlke [24] 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, run-time checks (guards) are inserted to check whether control flow etc. is still valid for the trace or not. They execute the run-time checks (guards) in parallel with the main execute flow (trace), and only have one single main execution flow.

In [23], Mehrara et al. introduce a lightweight speculation mechanism that focuses on loop-like constructs in JavaScript. If a loop contains a sufficient workload, it is marked for speculation. They were able to make speculation 2.8 times faster for a set of well-known JavaScript benchmarks. Both studies of Mehrara and Mahlke are made on the official JavaScript benchmarks.

Copyright © 2014 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

However, large loop structures are rare in real web applications [19]. Instead, there are often a large number of function calls caused by events from the web application.

Unlike [23], our approach is to execute the main execution flow in parallel [21]. We implemented TLS in the Squirrelfish JavaScript engine which is part of WebKit [37], and evaluated it on 15 web applications. Our results show that there is a significant potential for parallel execution using TLS in web applications. In addition, our results show that there is a significant difference between JavaScript benchmarks and the JavaScript executed in web applications. A serious consequence of this is that Just-in-time compilation, which improves the execution time for benchmarks, often prolongs the execution time for web applications which we show in Figure 1. Our measurements were made both with Squirrelfish and V8 with the same concluding result, that Just-in-time compilation for 10 out of 15 cases (for Squirrelfish) and 8 out of 15 cases for V8 improves the execution time for JavaScript in web applications. V8 still executes with a higher speedup than Squirrelfish with JIT enabled. In the same paper we also show that nested Thread-Level Speculation is necessary in order to improve the execution time over the sequential execution.

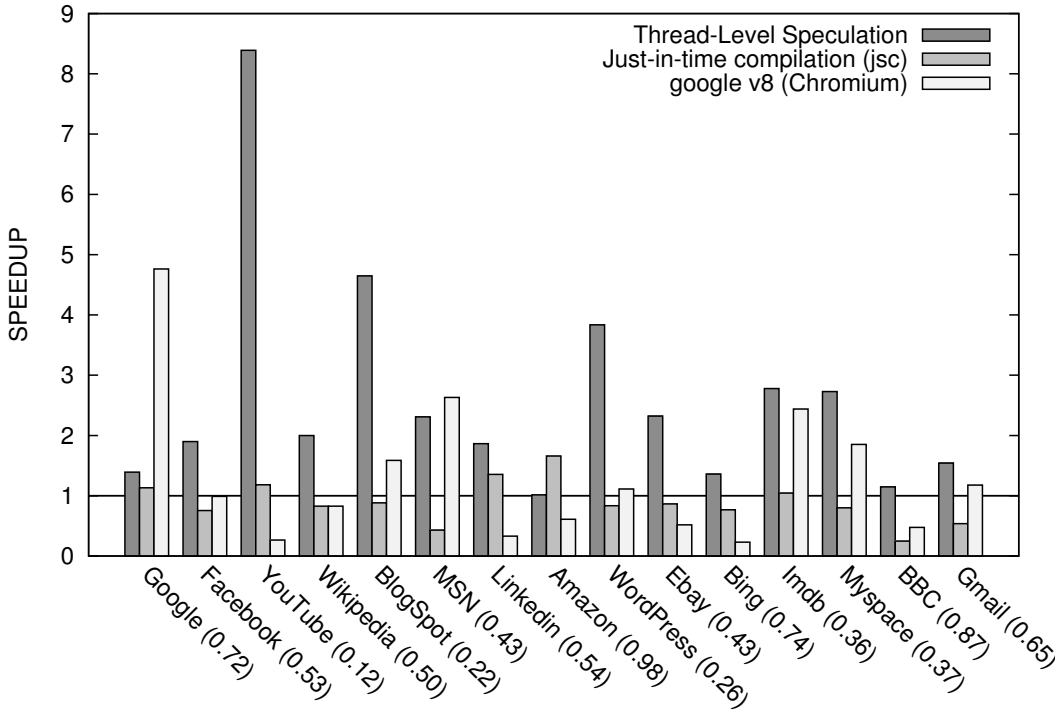

Figure 1. Speedup of Thread-Level Speculation and just-in-time compilation for a number of popular web applications. The black horizontal line is the sequential execution time of Squirrelfish without Thread-Level Speculation.

In [22] we suggested three heuristics which allowed us to re-speculate on previously failed speculations. The conclusion was that while we found that our combined heuristics gave us some gain by allowing respeculation, and reduced the relationship between speculations and rollbacks, the overall problem with TLS in JavaScript for web applications is not the number of rollbacks, as rollbacks are very rare. Rather, as we saw in [21], with a memory usage up to over 1500 MB, the main problem is the amount of memory used to store the states in case of a rollback. However, this also implies that we save many states, which most likely never will be used (i.e., for *Imdb* we make over 5000 speculations with 150 rollbacks).

To reduce the memory usage, we limit parameters such as the number of threads, the amount of memory, and the depth in nested speculation [20]. Our results show that it is possible to reduce the

Copyright © 2014 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

memory overhead and improve the execution time by adjusting the limits to these parameters. For instance, a speculation depth of 2 to 4 is enough to reach most of the speedup gained when no limit is set to the speculation depth. It also shows that we need to use nested speculation for Thread-Level Speculation for JavaScript in web applications.

In summary, there is a significant amount of research done on software-based Thread-Level Speculation. However, within managed programming languages and JIT compilation, together with TLS, there is a gap in the research.

3. IMPLEMENTATION OF TLS IN V8

3.1. JIT compilation in V8

JavaScript in web applications are to a large extent event-driven. These events are triggered e.g. when the user does a mouse gesture, when the user clicks a button, or between certain time intervals in the web application, etc. These events are often defined as anonymous JavaScript functions.

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 instead of being interpreted.

When V8 encounters a function, it checks if the function has previously been compiled. If it has not been compiled, V8 decides what kind of function it is; whether it comes from an *eval* call or from a normal function (or a JavaScript segment). It also decides whether the function is to be compiled in a normal way, or if it can be compiled in an optimized way, such that it will execute faster when executed. Once the function is compiled, it is placed in a cache. If we encounter the same function again, we do not need to re-compile it, but can execute the already compiled code as it is found in the cache.

In Figure 2 we see that for the 45 use cases, we re-use already compiled code in the cache for 18 of 45 use cases, and typically re-uses less than 10% of the compiled code in these cases. This can be understood from the behavior of the web applications [18]; there are many JavaScript functions which are short lived, many functions that are executed through calls to *eval* and many functions that are re-defined as the web application executes. Therefore re-use of existing JavaScript functions are rare. By looking at Figure 2, we observe that most of the web applications that are able to utilize already compiled functions to some degree are the web applications *Gmail*, *Google* (the search engine), *YouTube* and *BlogSpot* which are from the vendor that made the Google V8 JavaScript engine. (i.e., among the web applications, we see this clearly for all of the Google based web applications).

In Figure 3 we see that the cost of compiling a JavaScript function versus the cost of executing it is often eaten up by the cost of compiling the code, especially since re-use of previously compiled code is rare.

In Figure 5 we show what kind of functions that are compiled. From the figure we see that lazy compiled functions are most common (i.e., over 80% of the functions for most of the use cases are lazy), which shows the dynamicness of JavaScript in web applications (since these can be executed from events). Measurements have shown that they are often very small (in terms of number of lines of JavaScript source), but not always. We notice *Blogspot (1)*, where the number of normal compilations 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 to open an account, and where there is very little interaction, and few events calls, and therefore very little JavaScript in the form of lazy functions.

3.2. TLS implementation

In V8, various parts of the code are compiled into *stubs*. If the JavaScript code compiled is a function call, then this function is a candidate for speculation. Since we know that there are many function calls in web applications, we perform method level speculation. We base the sequential execution

Copyright © 2014 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

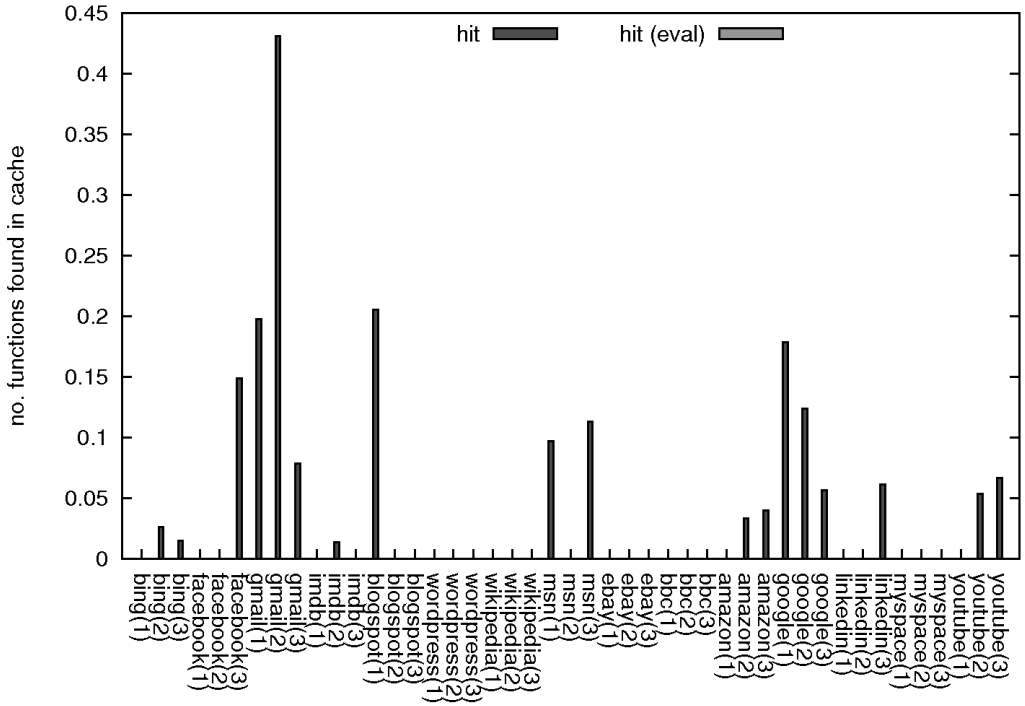

Figure 2. The relative number of times a function is found in the cache of already compiled functions.

order on function calls, since one key problem with TLS is to make sure that the parallel execution follows the sequential semantics.

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*).

During execution V8 might perform a JavaScript 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 instructions). We denote the value of the position of this function call as *function_order*, which emulates the *sequential time* in our program (Fig. 4). We check if this function has been previously 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. If the value is 0, then it has not been speculated or has been successfully speculated, and we call this position a fork point. This means that the function is not going to be found in the cache, so this function will be compiled. On rollbacks, we might encounter the same function call one more time, this time it will already be in the cache. However, it will then not be executed speculatively (i.e., as a thread).

If a function call is a candidate for speculation, we do the following after the function has been compiled; we set the position of the function *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 used global variables, and the content of *previous*.

We then allocate a thread from the threadpool, and use this for the function call in V8, with an unique id. We copy the value of *realtime* from its parent and modify the instruction pointer of the parent thread such that it continues the execution at the return point of the function so the parent thread skips the function call.

Copyright © 2014 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

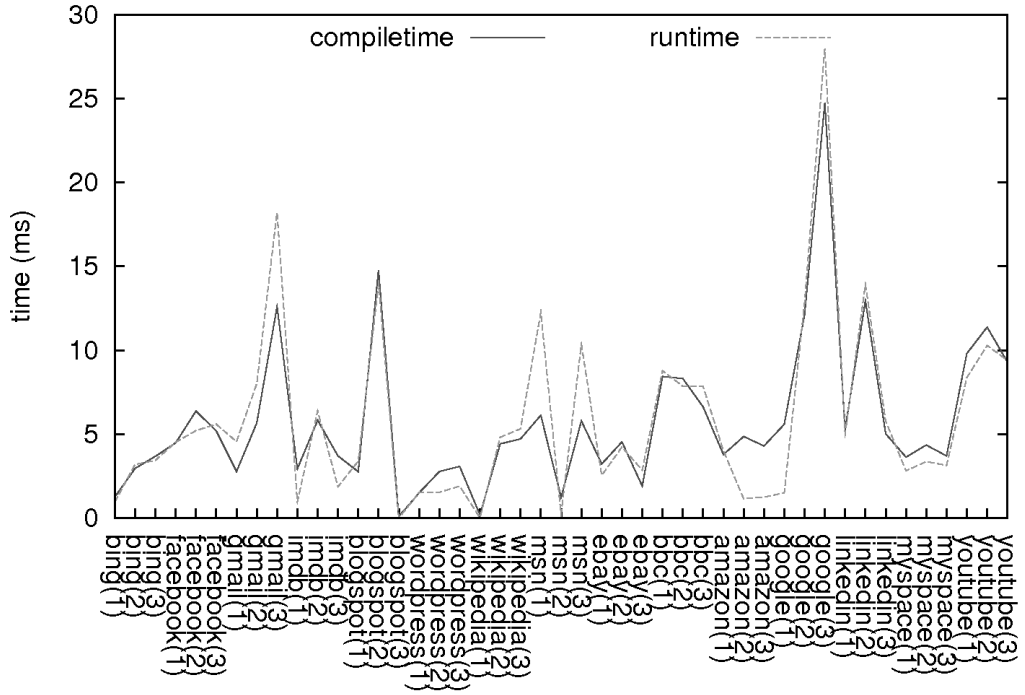

Figure 3. The time spent on compiling the various functions versus the time spent executing the functions.

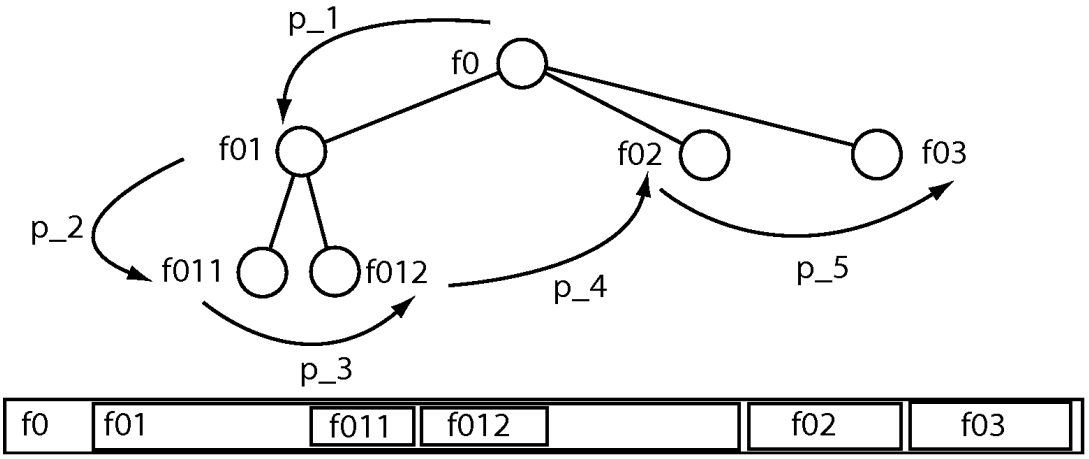

Figure 4. We use the order that the functions are called in, to determine the order in which the program would have been sequentially executed. This works in JavaScript in a web applications setting, as there are multiple function calls. For instance, in a simplified example the JavaScript function f0, 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*.

Copyright © 2014 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

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 we make an incorrect return value prediction we perform a rollback back to the point where the speculation started.

In V8, the global JavaScript variables are not compiled into the code that is 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 that part of the JavaScript stack, when we are about to speculate. We save the stack in case of a rollback. When we complete executing a function speculatively, we merge them into the stack and detect possible conflicts and misspeculations, which would cause a rollback to ensure the sequential semantics is not violated.

Restoration of the state on a rollback is faster than TLS in Squirrelfish. We do not need to re-compile the speculated function call (due to JavaScript behavior in web applications), and we take advantage of JIT for V8 when we rexecute the code natively, since 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 for TLS in web applications, as we have shown in [21].

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 functions decreases, in terms of execution time, when we are using JIT as compared to an interpreted JavaScript engine. For the implementation of TLS speculation in V8, we follow what is done in [21] where we speculate aggressively on function calls.

Like the implementation in [21] we support nested speculation (i.e., we make speculations from a speculated function), and therefore need to store states of the nested speculation, in case of a rollback. Also like previously seen, JavaScript function calls only rarely return any value in a web application. Therefore we use a return value prediction scheme like the *"last predicted value"* found in [14].

4. EXPERIMENTAL METHODOLOGY

In this study we have made the following experiments; we have measured the execution time and the behavior of TLS+JIT on 45 use cases for 15 well-known web applications, 4 use cases with Google maps, and 20 HTML5 web applications from the JS1K competition, the top 10 entries from the original competition in 2010 and the top 10 entries from the 2013 competition (Figure 6). We have selected these web applications since their workloads are significantly different from one another. The selected web applications and short descriptions of the use cases are presented in Table II.

We have selected the web applications from the Alexia list [1] of most popular web applications, and selected them based on the type of web application (such as, a social network like *Facebook*, a blog service such as *Wordpress* and a mail client like *Gmail*, etc). The use case behaviors are typically loading the front page, then a login to the web application (such as a social network, and an on-line auction or an on-line bookstore) and finally searching for one of the authors of this paper (if there is a search option available in the web application). We base the use cases on our own experience with these web applications, and try to perform the same actions on applications within the same realm, such that for instance creating a use case that is identical by using identical search terms for searching both the search engines *Google* and *Bing*.

For Google maps, we have 4 use cases. The first use case is the entry page of Google maps (`maps.google.com`). In the second use case we have clicked to locate where we are. In the third use case, we have made Google maps tell us the road map between Google's headquarter in Paolo Alto and the Microsoft's headquarter in Redmond, Seattle. In the fourth use case, we have asked for the road map between our university, Blekinge Institue of Technology, Karlskrona, Sweden and the island of Cagliari, Italy.

Copyright © 2014 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

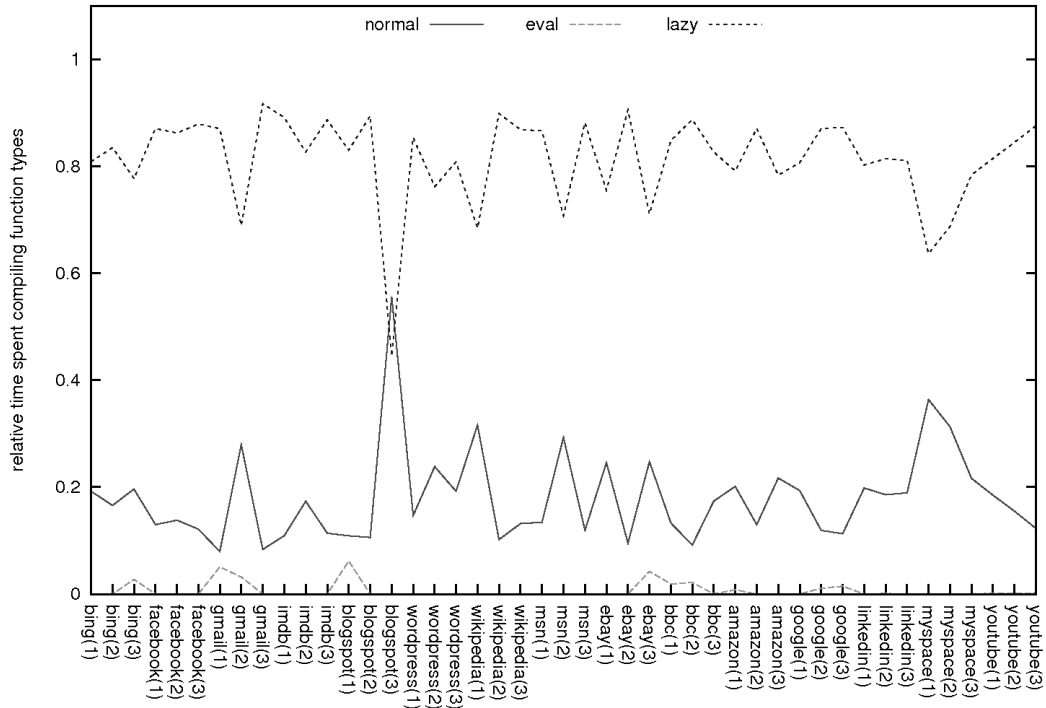

Figure 5. The relative time spent compiling the various function types. Normal functions are compiled when the web page is loaded, eval is compiled when the JavaScript code is executed using an `eval()` call, and lazy functions are compiled when they are about to be executed.

The JS1K competition (http://js1k.com/) is an annual competition, where the objective is to write a JavaScript application, where it is possible to use WebGL, HTML5 or normal DHTML. However, the restriction is that the entry must not be larger than 1 kilobyte of JavaScript code. We have evaluated the top 10 entries in the first competition in 2010 and the top 10 entries in the competition in 2013. In this way, we are able to see the development of HTML5 over 4 years.

Our Thread-Level Speculation is implemented in the V8 [11] JavaScript engine, and the use cases have be executed within Chromium [12] from Google. From [21] and in Figure 5 we see that web applications have a larger number of JavaScript function calls. Therefore we speculate at the function level where all data conflicts are correctly detected and rollbacks are done when conflicts arise, and we support nested speculation.

We perform these experiments and measurements on an eight core computer with 16GB of memory, where we adjust the number of enabled cores to 2, 4 or 8. The execution time is the JavaScript execution time of the V8 JavaScript engine, rather than the overall execution time of the whole web application. We execute each use case 10 times, and use the median of the execution time for comparison. The execution time is relative to the execution time on a Google V8 unmodified JavaScript engine where the JavaScript is executing sequentially.

To enhance reproducibility, we use a scripting environment [4] to both record and automatically execute the use cases in a controlled fashion. Due to the nature of web applications, we reduced the mouse driven interactivity, such as we do for instance use not the mouse to navigate in Google maps. We realize that this reduces the interactivity of this web application, and therefore also reduces the number of lines of executed JavaScript code; however we do this to make the use cases more reproducible. A detailed description of the methodology for performing these experiments is found in [18]. Since we reduce the interaction, such as the mouse gestures, this allows our results to be

Copyright © 2014 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

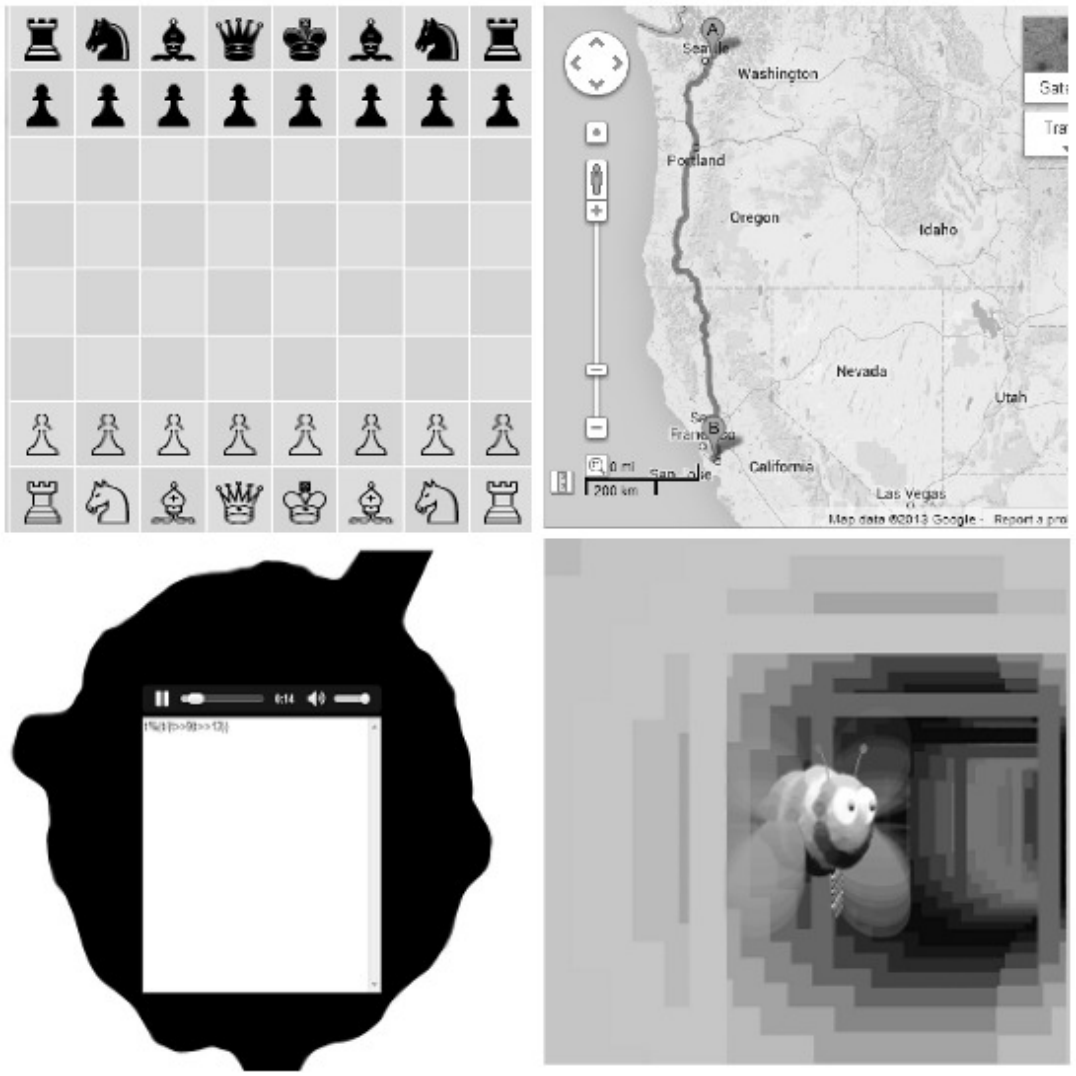

Figure 6. The roadmap generated in Google Maps between the Google Headquarter in Paolo Alto to the Microsoft headquarter in Redmond, Seattle (top right) and screenshot of 3 random entries (within the top 10) from the JS1K competition in 2010 and 2013.

more applicable to other platforms as well (such as laptops of different screensizes, smartphones, game consoles etc).

5. EXPERIMENTAL RESULTS

5.1. The effects of TLS on 15 web applications

One important observation from running web applications on V8 with TLS enabled is that there is going to be a large number of function calls, where each function call is small in terms of how many lines of JavaScript code that is executed. It does not really mean much to the execution time whether they are compiled into native code and executed or not, because the speedup of the native execution is limited to a small number of JavaScript lines. It also means that the dependencies between a

Copyright © 2014 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

Table II. The web applications, and a description of the use cases, that are used in the experiments.

Application	Description	use case #1	use case #2	use case #3
Google	Search engine	Load the frontpage of www.google.com	Type in the name of one of the authors of this paper (allow it to automatically load), and click search	Type in the name of one of the authors, and click image search
Facebook	Social network	Load the frontpage of www.facebook.com	login with an existing user account	Search for one of the authors of this paper
YouTube	Online video service	Load the frontpage of www.youtube.com	Search for one of the authors of this paper	View a video from the previous use case
Wikipedia	Online community driven encyclopedia	Load the frontpage of www.wikipedia.com	Search for one of the authors of this paper	Start to create an article of the previous search term
Blogspot	Blogging social network	Load the front page of www.blogspot.com (be registered on as a Google user)	Click ”New Blog” (But end it by clicking ”Cancel”)	Click ”View Blog”
MSN	Community service from Microsoft	Load the front page of www.msn.com	Click ”News”	Click on ”Pictures”
LinkedIn	Professional social network	Load the front page of www.linkedin.com	Login with an existing user-account	Search for one of the authors of this paper
Amazon	Online book store	Load the front page of www.amazon.com	Login with an existing user-account by clicking ”Sign-in”	Search for one of the authors of this paper
Wordpress	Framework behind blogs	Load the front page of www.wordpress.com	Click on ”Get Started”	Click on ”Discover WordPress”
Ebay	Online auction and shopping site	Load the front page of www.ebay.com	Search for one of the authors of this paper	Click on ”register”
Bing	Search engine from Microsoft	Load the front page of www.bing.com	Type in the name of one of the authors of this paper (allow it to load automatically), and click search	Type in the name of one of the authors, and click ”Images”
Imdb (Internet movie database)	Online movie database	Load the front page of www.imdb.com	Search for the name of one of the authors of this paper (allow it to load automatically), and click search	Click on ”See all birthdays”
Myspace	Social network	Load the front page of www.myspace.com	Click on the search symbol, and type in the name of one of the authors of this paper	Click on ”Discover”
BBC	News paper for BBC	Load the front page of www.bbc.co.uk	Search for one of the authors of this paper	Click on ”News”
Gmail	Online web client from Google	Load the front page of mail.google.com	Login with an existing user-account	Search for mails written by one of the authors, by typing in their mail address.

certain executing part of the code, is limited, so rollbacks are rare. These features make JIT suitable for TLS in web applications.

In Figure 7 we see the effects of running our TLS enabled version of the JavaScript engine V8 on 2, 4 and 8 cores for 45 use cases on 15 different web applications.

Copyright © 2014 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

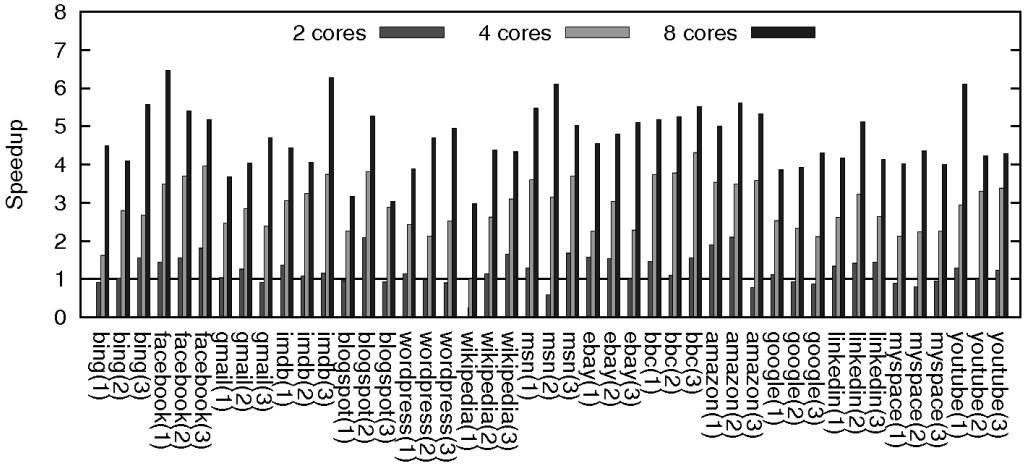

Figure 7. The speedup for 2, 4 and 8 cores on 45 use cases in 15 web applications.

We see that the average speedups for 2, 4 and 8 cores are 1.22, 2.9 and 4.7 respectively. If we look at the maximum and minimum speedup time, we see that it is 2.0 and 0.02 for 2 cores, 4.3 and 1.0 for 4 cores and 6.5 and 2.9 for 8 cores.

These results indicate that you need more than 2 cores to take advantage of TLS in combination with JIT in web applications. Based on the average and the maximum and minimum measurements, the set up that gives the best return in terms of number of cores and speedup is the 4 cores, as the distance between the maximum speedup and the average speedup is shorter for 4 cores than for 8 cores.

We can further see this as when we double the number of cores, the speedup does not double. It becomes on average 39% faster going from 2 to 4 cores, and 25% faster going from 4 to 8. This indicates 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.

Amazon (2) is twice as fast on 2 cores. We see this by looking at Figure 12 where we see that this use case has the largest maximum number of threads and the largest average number of threads.. If we look at the use case in Table II, where we log into *Amazon*, which obviously does some sort of personalization in terms of client side functionality (for instance certain fields in the web application are modified) which increases the use of JavaScript. This gives us the possibility to find many events / functions to speculate on, and we know from previous results that there will be little dependencies between such functions, which in turn allows us to speculate on many function calls. We also see that this is the result of the sum of previous uses of *Amazon*, which forces *Amazon* to present the web application according to the previous uses. This is interesting, as there is less interaction in this use case compared to *Amazon (3)*, but still a lot of JavaScript interaction of setting up the page.

We see that *Wikipedia* has the slowest execution time with TLS+JIT. We can understand this the following way; The JavaScript that is executed in this use case is limited in terms of number of lines of JavaScript code. However, we do not know that when we enter the web application, so we still try to speculate aggressively. This means that we set up the entire TLS, with thread-pool etc, for a use case where it turns out that there are few JavaScript function calls to speculate on, which in turn results in that TLS hardly will be used. We can see this in Figure 11. This is an interesting feature of TLS in web applications. It needs to find a certain number of functions to speculate on, or the costs of setting up the use for TLS will outweigh the gain in execution time by speculating.

If we look at the results of the execution on 4 cores, we see that for 43 of the use cases, the performance is doubled. The two use cases where it is slower, are *Bing (1)* and *Wikipedia (1)*. Both of these are the front page of the web application, which is shown in Figure 12 to consist of a small

Copyright © 2014 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

number of executing threads. This can be understood by that there is little interaction in these use cases (Table II) and that there is a need for some interaction to take advantage of JavaScript with TLS, as interaction allows us to execute more event generated JavaScript functions.

For almost half of the use cases, the speedup is three folded with 4 cores. If we look at the use cases, this applies in general to certain JavaScript intensive web applications (such as *Amazon*, *BBC*, *MSN*, *Imdb*, *Facebook*). It also, in general applies to use cases 2 and 3, where there is more interaction for the use cases.

For the *BBC (3)* use case the execution time is four times faster on a 4 cores than the sequential execution time , this seems to be the ”news” page that is rapidly updated thanks to JavaScript with ”news tickers”. These are quite independent of one another which makes speculation successful.

When using 8 cores, we are able to at most six double the execution speed for the use cases *Facebook*, *Imdb*, *MSN* and *Youtube*. Again, this improvement is found in both use case 2 and 3 (except for *Facebook* and *Youtube*) None of the applications are able to have a speedup over 7 or 8 times the sequential execution speed with 8 cores.

These observations show that increased interactivity of the web application increases the speedup with TLS, as this will be controlled by an increased number of events, which in turn increases the number of function calls. We also see that the speedup and number of cores ratio is the highest with 4 cores, this can be understood by that there is a limit to the number of events/function calls in a web application.

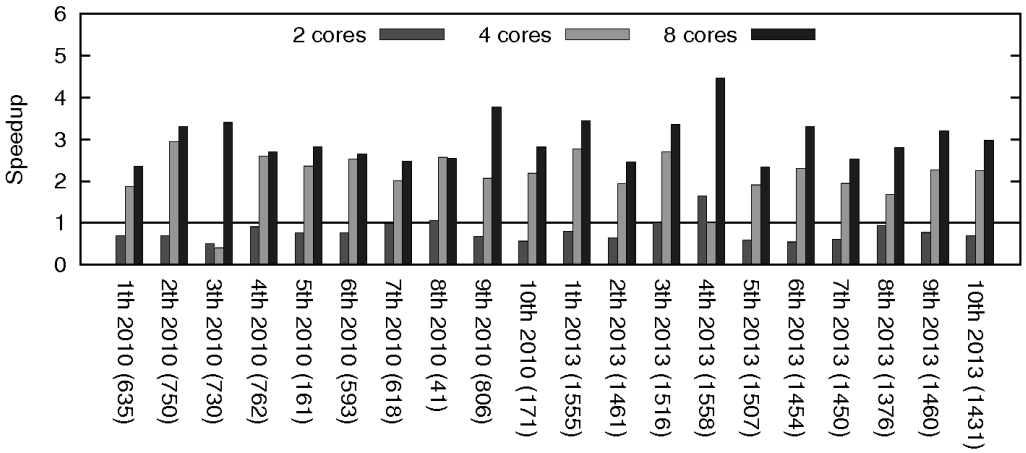

Figure 8. The speedup for 2, 4 and 8 cores on the set of top 10 entries of JS1K for 2010 and 2013.

5.2. The effects of TLS on 20 HTML5 demos

For HTML5 applications, there is a smaller number of function calls. Therefore, the speedup is lower with TLS. In addition, following the execution model of JavaScript in web applications, each function does not represent many lines of JavaScript code, therefore it will not be executing for a long period of time (which is the reason why we need to speculate on many JavaScript functions). One interesting feature is that even though it is event driven, part of the execution is more suitable for JIT compilation. These results show that both TLS and JIT can be successfully combined.

In Figure 8 we have measured the execution time on a set of HTML5 demos from the JS1K competition on 2, 4 and 8 core computers. We see that the average speedup for 2 cores is 0.79, the average speedup for 4 cores is 2.11 and the average speedup for 8 cores is 2.98. The maximum and minimum speedups for 2 cores are 1.65 and 0.50, for 4 cores they are 2.94 and 0.41 and for 8 cores the speedups are 4.46 and 2.34.

One interesting observation is that the JS1K entries consist of one large event, a loop which is executed repeatedly. In web applications this loop is commonly initialized with the events

Copyright © 2014 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

setInterval or *setTimeout*. In such a competition the function called for this event will be quite large, as this does most of the work and contains most of the JavaScript code. This indicates that this large event, which is a function call, could create a rollback quite early during the execution. This means that during the execution, we do not in general speculate on function calls made by this event, which in turn reduces the effect of running it in parallel. We can see this from the speedup, in general, we need more than 2 cores, to take advantage of this. We are usually able to get a speedup with more than 4 cores, but we see that the speedup if we compare the 4 cores to the 8 cores is limited. This indicates that we are not able to take advantage of a large number of function calls, which again could be caused by a relatively small code (i.e., few speculations) and that there is one large speculation which we are unable to speculate on efficiently.

We also notice that this large event driven loop, has dependencies, as it consists of most of the code, and that each iteration of this loop is more dependent of each other. This does in turn mean that this function call is likely to cause a rollback, which in turn reduces the effect of TLS in these web applications.

Since the JavaScript code is one kilobyte, there is a limit to how many functions we can speculate on. However, regardless of the fact that the codebase of these applications is small, we are able to have an on average 2.12 and 2.98 speedup on 4 and 8 core. This indicates that there is a potential to speculate, and that there is little dependency between the functions.

5.3. The effects of TLS on 4 Google maps use cases

The use cases for Google maps have limited user interaction, as we want them to be reproducible. Therefore the JavaScript execution is limited. This means that most of the work is done on the serverside.

In Figure 9 we have measured the speedup of 4 cases from Google maps on a 2, 4 and 8 cores. The average speedup with 2 cores is 1.13, the average speedup on 4 cores is 3.54 and the average speedups with 8 cores are 4.14. Likewise, if we look at the minimum and maximum speedup with 2 cores we find it to be 1.03 and 1.27 with 2 cores, 3.41 and 3.72 with 4 core and 4.10 and 4.17 with 8 cores.

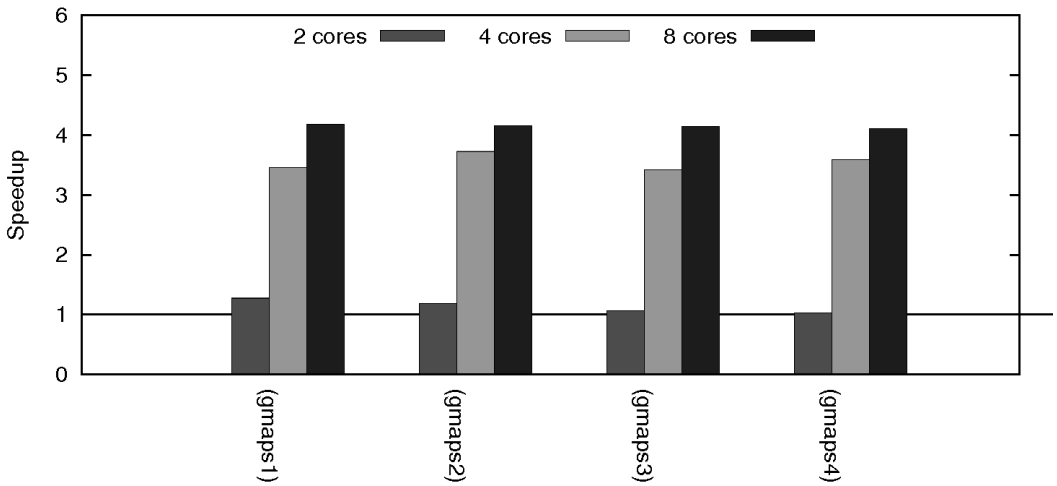

Figure 9. The speedup for 2, 4 and 8 cores on the set of the 4 use cases for Google maps.

Since the potentials of TLS in Google maps (Figure 9) are similar for the various use cases, this indicates that the use cases from a JavaScript point of view often are performing the same thing, as the computation of the route between two destinations is done on the serverside, and the JavaScript of the web application is about drawing the route on the screen. As we said in Section 4 we have no user interaction of moving the map in Google maps to make the use cases easier to reproduce. This

Copyright © 2014 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

gives less user interaction, and reduces the potential of TLS in Google maps. This makes for instance the speedup with 4 cores and 8 cores close to one another, as the reduced interaction decreases the potential of TLS.

If we compare the 15 web applications, the 20 HTML5 demos and the 4 Google maps use cases, we see that they are significantly different types of web applications. However, if we look at for instance web applications from Google (such as *Gmail*, *Google*, *Blogspot*, *Yotube* and *Google maps*) we see that their speedups are similar. This indicates that web applications have a lot of functions calls (due to events in the web application) for all the web application types, which in turn shows that TLS is advantagous for all of the web applications.

If we compare the results of TLS on the 20 HTML5 demos in Figure 8, these web applications have a lower speedup with TLS than the 15 web application in Figure 7 and the 4 use cases with Google maps in Figure 9. We can understand this the following way; the HTML5 web applications for the JS1K competition are naturally only one kilobytes in size. This means, that there is a limit to the number of functions defined in the JavaScript, which in turn reduces the number of speculations. In comparison with the 15 use cases of very popular web applications, the web application size of the JavaScript code is much larger than one kilobyte. For instance, in Figure 10 we see the number of lines of JavaScript source code that are compiled in a lazy manner for *Gmail*. In addition in the HTML5 demos, there is often one single function call which consists of much of the workload of the execution. This function call, could cause a rollback, and will not be used in the future for more speculations. This reduces the number of speculations even further.

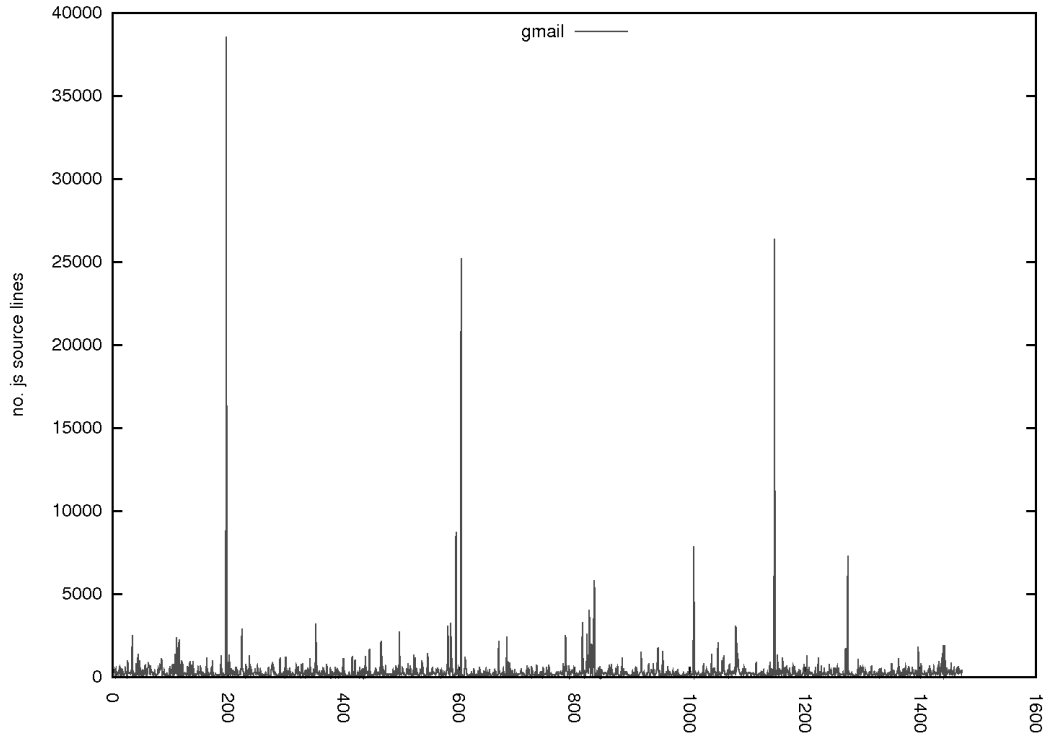

Figure 10. The number of lines of JavaScript code for the various calls to the compiler in V8 for the web application *Gmail*.

A surprising observation is that 8 cores is not much faster than 4 cores. This could be understood by the fact that there is a limit to the number of speculations. For instance, we see that there is less speculation in the HTML5 demos than in the 15 web applications and in the 4 use cases of Google maps.

Copyright © 2014 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

We could explain the behavior in the following way; 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 11 we have measured the time it takes to initialize the threadpool and the time it takes to execute the threads. We see that the time it takes to set up the threadpool increases as the number of cores increases.

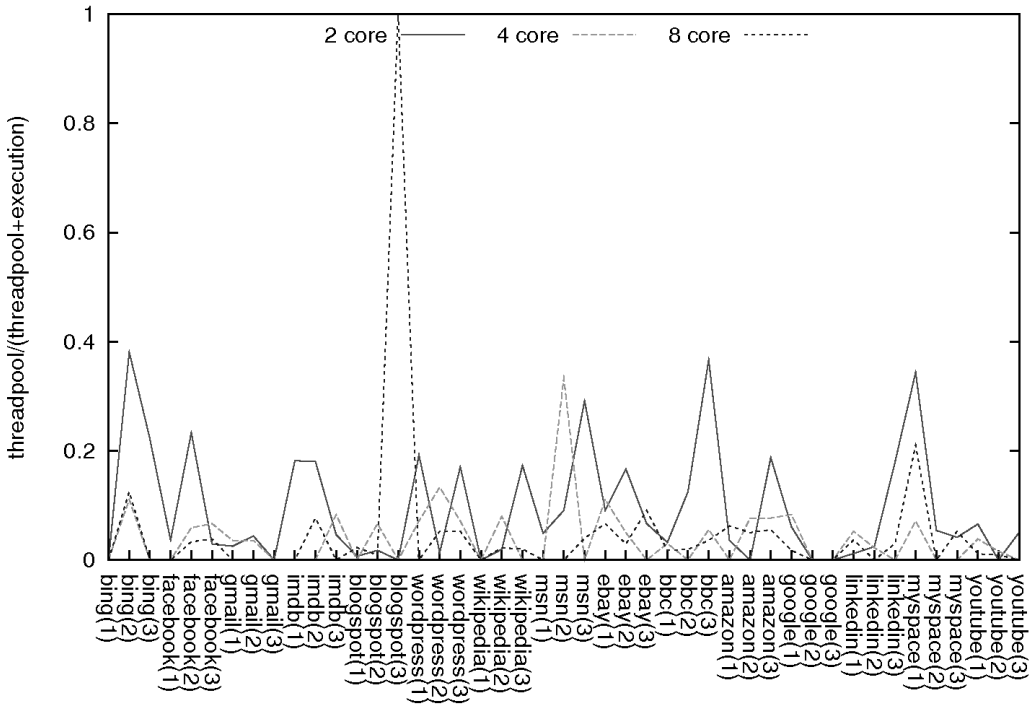

Figure 11. The thread pool initialization time as propotion of the total execution time.

In Figure 12 we see the maximum and average number of threads executing concurrently during execution. If we measure the average maximum number and the average number of threads and the distance between them we find them to be: 204, 123 and 81 threads. 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 an 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 JavaScript execution. Likewise, in *Wikipedia (1)*, we know from [21] that the front page of *Wikipedia* has a limited amount 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 are very large in terms of instructions to be executed. This makes the average number of functions executing low compared to the maximum number of functions executing. These functions have a low number of writes, which in turn makes them easy to speculate on. This could be caused by the *MSN* use case since it has several ”tickers” where the functionality in terms of JavaScript is long enduring, which again creates a large number of functions which are suitable for speculation. As we see in Figure 7 this creates one of the largest speedups which are over 6 times faster than the sequential execution time of V8.

Copyright © 2014 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

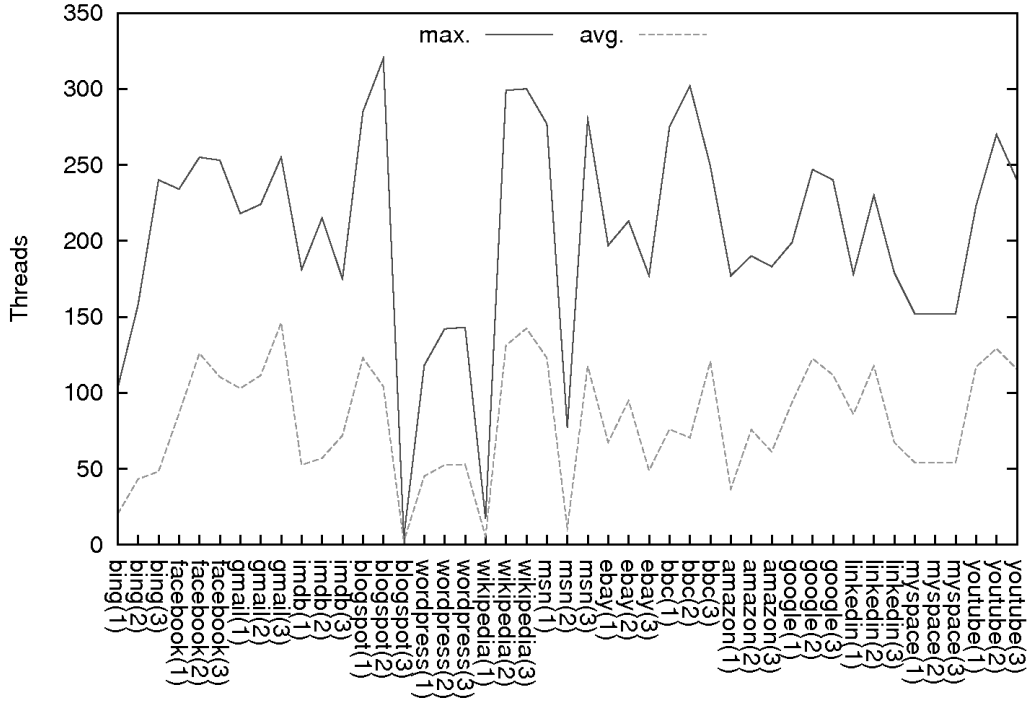

Figure 12. The maximum and the average number of threads during execution.

6. CONCLUSION

We have presented the first implementation of Thread-Level Speculation in combination with Just-in-time compilation (TLS+JIT) with a method-level approach running on a set of popular web applications, the Google maps web application and finally a set of HTML5 demos. We have evaluated it on 45 use cases in 15 web applications, 20 HTML5 demos, and 4 use cases for Google maps, and made the experiments on 2, 4, or 8 cores on a dual quadcore computer.

Our results show that TLS can be successfully combined with JIT, and that the programming model employed in the web applications makes TLS appropriate for taking advantage of multicore hardware. We also see that there are features in Google's V8 that are very useful for TLS, that we need more than 2 cores to successfully take advantage of TLS+JIT in web applications, and that 4 cores yield an average speedup of 2.9, and 8 cores an average speedup of 4.7. For 20 HTML5 demos, the speedup is on average 2.11 for 4 cores and 2.98 on 8 cores. In Google maps, the average speedup is 3.54 on 4 cores and 4.17 on 8 cores. This indicates that the largest gains in terms of the ratio between the execution time and the number of cores is 4.

In summary, we have presented the first implementation of TLS and JIT combination which run on the JavaScript web application, and our results show significant speedups without any changes of the JavaScript source code at all. We believe that TLS+JIT is a very promising approach to enhance the performance of JavaScript in web applications.

REFERENCES

1. Alexa. Top 500 sites on the web, 2010. http://www.alexa.com/topsites.
2. M. Berry, D. kai Chen, P. F. Koss, D. J. Kuck, S. lo, Y. Pang, R. R. Roloff, A. Sameh, E. Clementi, S. Chin, D. J. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. A. Orszag, F. Seidl, O. Johnson, G. Swanson, R. Goodrun, and J. Martin. The PERFECT Club Benchmarks: Effective performance evaluation of

Copyright © 2014 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

supercomputers. Technical Report CSRD-827, Center for Supercomputing Research and Development, Univ. of Illinois, Urbana-Champaign, May 1989.
3. 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*, pages 99–108, 2002.
4. 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.
5. 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.
6. 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*, 29(2):6–16, 2009.
7. M. K. Chen and K. Olukotun. Exploiting method-level parallelism in single-threaded Java programs. In *Proc. of the 1998 Int'l Conf. on Parallel Architectures and Compilation Techniques*, page 176, 1998.
8. 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*, pages 434–446, 2003.
9. M. Cintra and D. R. Llanos. Toward efficient and robust software speculative parallelization on multiprocessors. In *PPoPP '03: Proc. of the 9th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 13–24, 2003.
10. 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.
11. Google. V8 JavaScript Engine, 2012. http://code.google.com/p/v8/.
12. Google. Chromium web browser, 2013. http://www.chromium.org/.
13. B. C. Hertzberg and K. Olukotun. Runtime automatic speculative parallelization. In *Proc. of the 9th Annual IEEE/ACM Int'l Symp. on Code Generation and Optimization (CGO)*, pages 64–73, April 2011.
14. 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*, 5, 2003.
15. JavaScript. http://en.wikipedia.org/wiki/JavaScript, 2010.
16. 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*, 12(9):952–966, 2001.
17. A. Kejariwal, X. Tian, W. Li, M. Girkar, S. Kozhukhov, H. Saito, U. Banerjee, A. Nicolau, A. V. Veidenbaum, and C. D. Polychronopoulos. On the performance potential of different types of speculative thread-level parallelism. In *ICS '06: Proc. of the 20th Int'l Conf. on Supercomputing*, page 24, New York, NY, USA, 2006. ACM.
18. 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.
19. 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.
20. 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)*, pages 75–82, November 2012.
21. J. K. Martinsen, H. Grahn, and A. Isberg. Using Speculation to Enhance JavaScript Performance in Web Applications. *IEEE Internet Computing*, 17(2):10–19, 2013.
22. 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.
23. 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*, pages 87–98, 2011.
24. 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)*, pages 74–84, april 2011.
25. 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)*, pages 127–142, April 2010.
26. Mozilla. SpiderMonkey – Mozilla Developer Network, 2012. https://developer.mozilla.org/en/SpiderMonkey/.
27. 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*, pages 59–66, 2005.
28. C. J. F. Pickett and C. Verbrugge. Software thread level speculation for the Java language and virtual machine environment. In *LCPC '05: Proc. of the 18th Int'l Workshop on Languages and Compilers for Parallel Computing*, pages 304–318, October 2005. LNCS 4339.
29. 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.
30. J. Renau, J. Tuck, W. Liu, L. Ceze, K. Strauss, and J. Torrellas. Tasking with out-of-order spawn in tls chip multiprocessors: Microarchitecture and compilation. In *In ICS*, pages 179–188, 2005.
31. G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do: A large-scale study of the use of eval in javascript applications. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, pages 52–78, Berlin, Heidelberg, 2011. Springer-Verlag.
32. 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.

Copyright © 2014 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

33. 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.
34. Standard Performance Evaluation Corporation. SPEC jvm98, 1998. http://www.spec.org/jvm98/.
35. Standard Performance Evaluation Corporation. SPEC CPU2000 v1.3, 2000. http://www.spec.org/cpu2000/.
36. Standard Performance Evaluation Corporation. SPEC CPU2006 v1.0, 2006. http://www.spec.org/cpu2006/.
37. WebKit. The WebKit open source project, 2012. http://www.webkit.org/.

Copyright © 2014 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe