DESIGN AND IMPLEMENTATION OF THREAD-LEVEL SPECULATION IN JAVASCRIPT ENGINES

Jan Kasper Martinsen

Blekinge Institute of Technology
Doctoral Dissertation Series No. 2014:08

Department of Computer Science and Engineering

Design and Implementation of Thread-Level Speculation in JavaScript Engines

Jan Kasper Martinsen

Blekinge Institute of Technology Doctoral Dissertation Series
No 2014:08

Design and Implementation of Thread-Level Speculation in JavaScript Engines

Jan Kasper Martinsen

Doctoral Dissertation in
Computer Systems Engineering

Department of Computer Science and Engineering
Blekinge Institute of Technology
SWEDEN

2014 Jan Kasper Martinsen
Department of Computer Science and Engineering
Publisher: Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden
Printed by Lenanders Grafiska, Kalmar, 2014
ISBN: 978-91-7295-281-2
ISSN 1653-2090
urn:nbn:se:bth-00589

Abstract

Two important trends in computer systems are that applications are moved to the Internet as web applications, and that computer systems are getting an increasing number of cores to increase the performance. It has been shown that JavaScript in web applications has a large potential for parallel execution despite the fact that JavaScript is a sequential language. In this thesis, we show that JavaScript execution in web applications and in benchmarks are fundamentally different and that an effect of this is that Just-in-time compilation does often not improve the execution time, but rather increases the execution time for JavaScript in web applications. Since there is a significant potential for parallel computation in JavaScript for web applications, we show that Thread-Level Speculation can be used to take advantage of this in a manner completely transparent to the programmer. The Thread-Level Speculation technique is very suitable for improving the performance of JavaScript execution in web applications; however we observe that the memory overhead can be substantial. Therefore, we propose several techniques for adaptive speculation as well as for memory reduction. In the last part of this thesis we show that Just-in-time compilation and Thread-Level Speculation are complementary techniques. The execution characteristics of JavaScript in web applications are very suitable for combining Just-in-time compilation and Thread-Level Speculation. Finally, we show that Thread-Level Speculation and Just-in-time compilation can be combined to reduce power usage on embedded devices.

Acknowledgements

I would like to thank my supervisors, Professor Håkan Grahn and Professor Lars Lundberg for invaluable guidance and help in my PhD studies. Without them this thesis would never have seen the light of day.

I would also like to thank all of our contacts from Sony for giving me this opportunity to work in a real world industrial project, access to relevant hardware and for providing invaluable advices in term of continual project meetings. I would also like to thanks them for being so involved in this research (too many to mention), and for that they co-authored papers which were central to this thesis.

In addition I would like to thank all my colleagues, especially those in the DIDD research group and all those who are part of the EASE project.

Finally I would like to thank Heidi Kramer, for all the proof-reading, patience and support throughout this thesis.

This work was partly funded by the Industrial Excellence Center EASE - Embedded Applications Software Engineering, (`http://ease.cs.lth.se`).

List of Papers

The following papers are included in this thesis

Paper I: 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

Paper II: J. K. Martinsen, H. Grahn, and A. Isberg. Evaluating Four Aspects of JavaScript Execution Behavior in Benchmarks and Web Applications. Research Report 2011:03, Blekinge Institute of Technology, July 2011 (A shorter version is published in Paper XII)

Paper III: J. K. Martinsen and H. Grahn. Thread-Level Speculation as an Optimization Technique in Web Applications for Embedded Mobile Devices – Initial Results. In *Proc. of the 6th IEEE Int'l Symposium on Industrial Embedded Systems (SIES'11)*, pages 83–86, June 2011

Paper IV: 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

Paper V: J. Martinsen, H. Grahn, and A. Isberg. Heuristics for Thread-Level Speculation in Web Applications. *IEEE Computer Architecture Letters*, pages 1–1, 2013 (Published online Nov 2013)

Paper VI: J. Martinsen, H. Grahn, and A. Isberg. The Effects of Parameter Tuning in Software Thread-Level Speculation in JavaScript Engines. 2014 (Submitted for journal publication. A shorter version is published in Paper XV)

Paper VII: J. Martinsen, H. Grahn, H. Sundström, and A. Isberg. Reducing Memory Usage in Software-Based Thread-Level Speculation for JavaScript in Virtual Machine Execution of Web Applications. 2014 (Submitted for conference publication)

Paper VIII: J. Martinsen, H. Grahn, and A. Isberg. Combining Thread-Level Speculation and Just-In-Time Compilation in Google's V8 JavaScript Engine. 2014 (Submitted for journal publication. A shorter version is published in Paper XVI and Paper XVIII)

Paper IX: J. Martinsen, H. Grahn, S. Drincic, and A. Isberg. Performance and Power Usage of Thread-Level Speculation in the V8 JavaScript Engine. 2014 (Submitted for journal publication)

The following papers are related but not included in the thesis.

Paper X: J. K. Martinsen and H. Grahn. Thread-Level Speculation for Web Applications. In *Second Swedish Workshop on Multi-Core Computing (MCC-09)*, pages 80–88, November 2009

Paper XI: J. K. Martinsen and H. Grahn. A Comparative Evaluation of the Execution Behavior of JavaScript Benchmarks and Real-World Web Applications(poster presentation). In *Poster Proc. of the 28th International Symposium on Computer Performance, Modeling, Measurements and Evaluation (Performance-2010)*, pages 27–28, 2010

Paper XII: 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

Paper XIII: J. K. Martinsen and H. Grahn. An Alternative Optimization Technique for JavaScript Engines. In *Third Swedish Workshop on Multi-Core Computing (MCC-10)*, pages 155–160, November 2010

Paper XIV: J. K. Martinsen, H. Grahn, and A. Isberg. The Effect of Thread-Level Speculation on a Set of Well–known Web Applications. In *Fourth Swedish Workshop on Multi-Core Computing (MCC-11)*, November 2011

Paper XV: 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

Paper XVI: J. K. Martinsen, H. Grahn, and A. Isberg. Preliminary Results of Combining Thread-Level Speculation and Just-in-Time Compilation in Google's V8. In *Sixth Swedish Workshop on Multi-Core Computing (MCC-13)*, pages 37–40, November 2013

Paper XVII: J. Martinsen. *Evaluating JavaScript Execution Behavior and Improving the Performance of Web Applications with Thread-Level Speculation.* Licentiate thesis, Blekinge Institute of Technology (BTH), Karlskrona, Sweden, Nov. 2011

Paper XVIII: J. Martinsen, H. Grahn, and A. Isberg. An Argument for Thread-Level Speculation and Just-in-Time Compilation in the Google's V8 JavaScript Engine. In *Conf. Computing Frontiers*, 2014 (short paper)

Contents

Chapter 1

Introduction

Two important trends in computer systems are that applications are moved to the Internet as web applications, and that computer systems are getting an increasing number of cores to increase the performance.

In this thesis we propose Thread-Level Speculation as a technique to increase the performance of JavaScript in web applications by taking advantage of multiple cores without the programmer's awareness. This means that the difficulty of taking advantage of parallel hardware is completely transparent for the web application programmer.

1.1 Parallel computation and multiple cores

Increasing the number of cores is a method to decrease the execution time of computer systems [73]. This applies to both large computer systems (such as servers) and embedded computer systems (such as smartphones).

The challenge with this trend is that in order to take advantage of multiple cores, the software (such as the operating system and applications) must be made into parallel programs [33]. Parallel programming is considered to be one of the most difficult programming fields within computer science [92]. However, the challenges are not solely dependent on the experience and the skills of the

programmer, but more the inherent difficulties with parallel programming. For instance:

(i) How do we decompose a problem into sub problems that we can run in parallel?

(ii) How should the program be designed in order to remain scalable when we increase/decrease the number of cores (i.e., if 4 cores lead to a speedup of 4, how do we ensure that 8 cores lead to a speedup of 8?)

(iii) How do we efficiently debug parallel programs? Scheduling of the parallel program is made by the operating system, so this means that different parts of the program can run in different order each time the program is executed.

(iv) The programmer might also run into problems such as deadlocks, livelocks and starvations. A deadlock is when two processors wait for each other to release a resource, and we therefore get a circular dependency and the program appears to be locked. A livelock is similar to a deadlock, only that even though we need to wait for a resource, the resources changes constantly. A starvation is similar to a deadlock; a process waits forever for a resource, without getting one.

For instance, in Figure 1.1 we show an example of a dual-quadcore multiple core computer where these cores read and write to the same memory. This makes programming easier, but introduces new problems which make programming more difficult, as we must for instance make sure that none of the cores write to the same part of the memory, at the same point in time. Also as we saw in the list of difficulties, as the number of cores increases it can be difficult to scale to performance.

However, the performance potential of multicore processors has been demonstrated in several applications. Examples of applications are visualization, videos, 3D graphics, computer games, web servers and database systems.

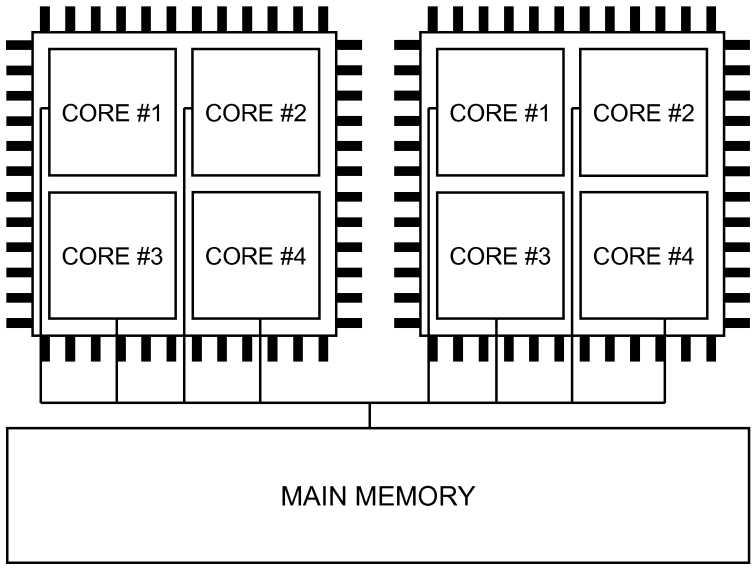

Figure 1.1: An example schematic of a dual-quadcore architecture with two CPUs which contain 4 cores each, however all cores have access to the same memory. It is usually up to the programmer to take advantage of the cores.

1.2 JavaScript and web applications

Web applications are to a large extent powered by JavaScript. JavaScript is an object based scripting language where runtime evaluation is done in a JavaScript engine. JavaScript's popularity has naturally grown with the success of the web. Initially it was a programming language for adding interactivity to the web application, thereby reducing the need for server side communication. It was for instance used to validate forms in a web application. JavaScript in web applications has since then become increasingly complex as AJAX (Asynchronous JavaScript and XML) programming has transformed static web pages into responsive applications. In addition JavaScript significantly eases the distribution of software, as web applications are dependent on a modern web browser, and therefore are virtually platform independent. However, from a parallel programming point of view, JavaScript is a sequential programming language.

Web applications are applications where parts are executed in a web browser and delivered through the web. Web applications have the advantage that they require no additional installation, will run on any machine that can run a modern web browser, and provide information from the cloud. Smartphones, such as the Sony Xperia Z1 phone, increase the number of Internet users, further increasing the importance and the reach of web applications. Web applications often combine client and server side functionality. However in this thesis we focus on the client side functionality.

Much of the client side functionality is written in the sequential language JavaScript, which in turn is executed in a JavaScript engine. As we see in Figure 1.2 there are other client side workloads that take time, other than executing JavaScript, for instance parsing the cascading style sheets (CSS) and rendering the web page on the web browser. However the importance of JavaScript is apparent from the execution time of each JavaScript function call and the number of function calls.

JavaScript has features that are similar to features in functional programming languages like Haskell (such as anonymous and eval functions) while it has a syntax similar to C and Java. To evaluate the JavaScript performance a set of benchmarks has been developed. Benchmarks often solve a test problem, and the problems these benchmarks solve are similar to problems in benchmarks used in other fields (for instance the wellknown SPEC 2000 benchmarks suite, and the V8 JavaScript benchmarks from Google both contain a raytracer). During the last years there has been an unofficial race to have the fastest web browser,

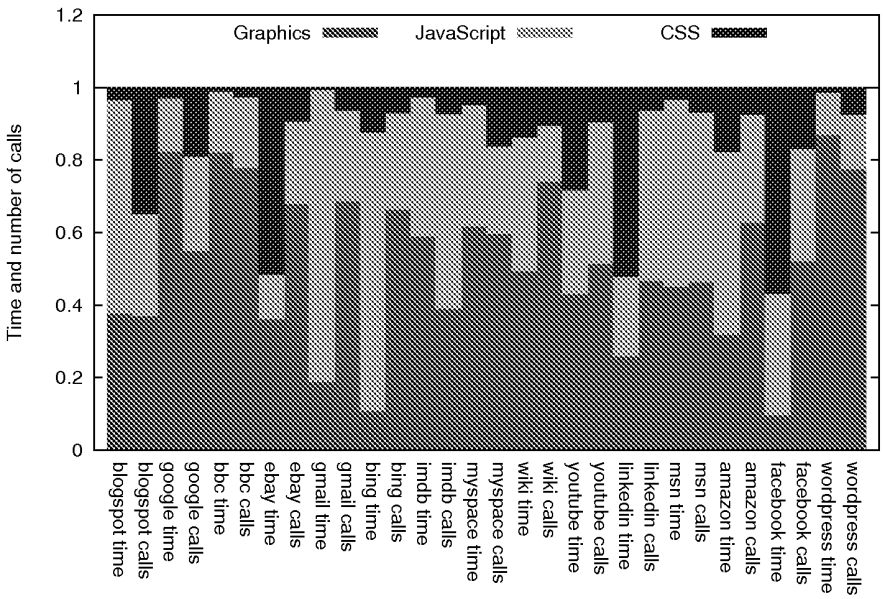

Figure 1.2: The time it takes and the number of calls it takes to render, execute JavaScript code and parse CSS definitions for 15 well known web applications.

and therefore there has been an increased focus on improving the performance of JavaScript engines.

Ratanaworabhan et al. [83] show that the established JavaScript benchmarks often misrepresent the actual execution behavior of real life web applications. Examples of factors that might lead to misleading conclusions are that the benchmarks have a large number of loops, non-string objects in web applications are extremely short lived and that the web application calls a significant number of events. Richards et al. [89] show that the dynamic features of JavaScript are used extensively in web applications. The benchmarks do not use these features and they emulate the behavior of static typed programs. It is in turn pointed out that this might lead to a misleading understanding of the execution behavior of JavaScript. Zorn et al. [84] evaluate the workload characterization of benchmarks and web applications and show that there are significant differences between them, such as different kinds of function types and different types of executed bytecode instructions.

Fortuna et al. [25] show that there is, theoretically speaking, a significant potential for parallel execution of client side computation in web applications, with up to 45 times speed up for the Google Wave web application. However, this is strictly a theoretical study, with no specific implementation where they implement their ideas in a real life JavaScript engine. While Fortuna et al. show that there is a potential for huge speedups with parallel execution in web applications for JavaScript, they base their argument on a theoretical model, rather than an implementation. They also do not have a strong methodological approach, and rather observe the executed JavaScript code in order to decide on a parallel potential.

JavaScript and web applications have support for multicore programming with Web workers [104] which uses a message passing memory model for communication between the processes. However, it is still the programmers' responsibility to find and take advantage of the parallelism in web applications for Web workers and Web workers' intention is to increase the response of the web application, rather than to speed up the execution time.

To our knowledge, there is no-one who has developed a Thread-Level Speculation system for JavaScript engines which dynamically extracts parallism, and consistently improves the execution time.

1.3 An introduction to Thread-Level Speculation

One attempt to make parallel programming easier is Thread-Level Speculation. The idea is to dynamically extract parallelism from sequential programs. This is done by speculatively executing segments of the program in parallel. To ensure the program correctness, we need to check for data conflicts between parallel segments. If a conflict occurs we need to rollback the program to a previous point of execution where there are no conflicts. Then, we re-execute from this point sequentially. Obviously we want to avoid re-executing parts of the program, to speed up the execution time. In Figure 1.3, to the leftmost we see the program that is a candidate for speculation. Right to the source code we see the program executing sequentially (i.e., it calls function f, which again calls function g). In the rightmost figure, we execute the program speculatively, when we encounter the function call g. We try to execute this in parallel, and later join this execution back to f.

Thread-Level Speculation has been done both in hardware, e.g., [14, 85, 98], and software, e.g., [13, 41, 78, 81, 91]. Two main approaches exist: loop-level

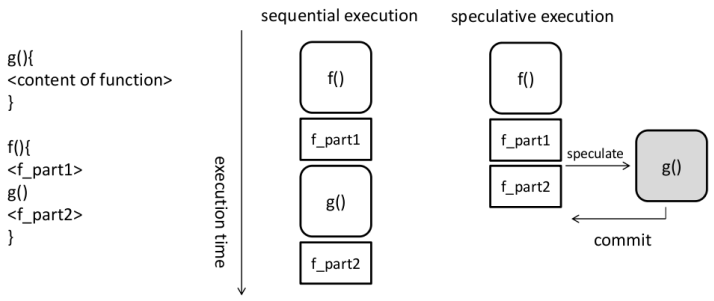

Figure 1.3: A schematic of the workings of Thread-Level Speculation.

parallelism and method-level speculation. In loop-level parallelism, 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. In method-level speculation, we execute function calls as threads. In this approach, we must correctly predict the return values when we speculate as well as the writes and reads that cause the speculative program to violate the sequential semantics. The last two are typically detected when the values associated with two function calls are committed back to their parent thread. For both approaches the memory requirements and run-time overhead for checkpointing and detecting data dependencies can be considerable.

Between two concurrent 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 runtime by 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 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 checkpoint mechanism and 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, this bookkeeping results in both memory overhead as well as run-time overhead. In order for a TLS system to be efficient, the number of rollbacks must 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 in the memory, or in a special speculation buffer. Updating data in-place usually results in a higher performance if the number of rollbacks is low, but a lower performance when the number of rollbacks is high since the cost of doing rollbacks is high.

1.3.1 Software-Based Thread-Level Speculation

There exists a number of different software-based Thread-Level Speculation (TLS) proposals, and we summarize some of the most important ones in Table 1.1. One striking observation is that all of these studies are with applications written in C, FORTRAN, or Java and that these studies are focusing most of their effort around the SPEC and Perfect Club benchmark series. We also see that none of them are made for a dynamically executing environment, such as JavaScript.

Table 1.1: Examples of previous work on software Thread-Level Speculation

Author	Speedup	#cores	Benchmark	Summary	Language
Chen and Olukotun show [15, 16]	3.7–7.8	16	SPECjvm98 [95] (with others)	method-level	C and JAVA
Bruening et al. [13]	over 5	8	SPEC95FP and SPEC92	applicable on while loops	C
Rundberg and Stenström [91]	10	16	Perfect Club Benchmarks [8]	minimizing the overhead	C
Kazi and Lilja [41]	5–7	8	Perfect Club Benchmarks [8]	exploit coarse-grained parallelism	C
Bhowmik and Franklin [9]	1.64–5.77	6	SPEC CPU95, SPEC CPU2000, and Olden	exploits loop and non-loop parallism	C
Cintra and Llanos [17]	16	16	SPEC CPU2000 [96] and Perfect Club [8]	using a sliding window with loops	C
Renau et al. [86]	3.7–7.8	16	SPEC CPU2000 Benchmarks [96]	method-level	C
Picket and Verbrugge [80, 81]	2	4	SPECjvm98 [95]	method-level	JAVA
Kejariwal et al. [42]	2.5		SPEC CPU2000 Benchmarks [96]	Theoretical study	C and FORTRAN
Hertzberg and Olukotun [34]	2.04	4	SPEC CPU2000 Benchmarks [96]	method-level	C
Hertzberg and Olukotun [34]	3–4, 2–3, and 1.5–2.5	4	SPEC CPU2006 Benchmarks [97]	method-level	JAVA

1.3.2 TLS for managed programming languages

Mehrara and Mahlke [67] show how to utilize multicore systems in JavaScript engines. They target trace-based JIT-compiled JavaScript code, where the most common execution flow is compiled into an execution trace. Then, runtime checks (guards) are inserted to check whether control flow etc. is still valid for the trace or not. They execute the runtime checks (guards) in parallel with the main execution flow (trace), and only have one single main execution flow.

In [66] they introduce a lightweight speculation mechanism that focuses on loop-like constructs in JavaScript, and if the loop contains a sufficient workload, it is marked for speculation. As this code used the trace features of Spidermonkey, a selective form of speculation is employed. They found that they were able to make speculations 2.8 times faster for well known JavaScript benchmarks.

Mickens et al. [68] suggest an event-based speculation mechanism which is deployed as a JavaScript library called Crom. Crom clones regions of the JavaScript code that are executed speculatively, and in case of a misspeculation the original state can be restored. However, their main goal is to enhance the responsiveness of web applications.

From the previous work, to our knowledge, no-one has used Thread-Level Speculation to take advantage of the parallel potential of JavaScript in web applications (which Fortuna et al. proposes).

1.4 Research questions

The main objective of this thesis is how to take advantage of multicore processors for JavaScript in web applications with Thread-Level Speculation.

Research question 1 and research question 2 focus on how to evaluate the execution behavior of JavaScript in web applications and benchmarks.

Research question 1: What method should we use to compare the execution behavior of the benchmarks with the web applications?

Research question 2: What are the differences in execution behavior between the benchmarks and the web applications?

JavaScript is a sequential programming language and to take advantage of multicore systems we investigate Thread-Level Speculation as an option in research question 3. In research question 4 we evaluate the effects of respeculating on functions that have previously caused a misspeculation and suggest three heuristics to adaptively respeculate.

Research question 3: How do we implement Thread-Level Speculation in a interpreting JavaScript engine?

Research question 4: What is the performance improvement potential with Thread-Level Speculations for JavaScript in web applications, and which are the main limiting factors?

In research question 5, we look at the memory used during Thread-Level Speculation, in case a speculation results in a rollback. Finally a very popular technique to reduce the execution time in interpretive JavaScript engines is Just-in-time compilation. In research question 6, we examine whether this technique is advantageously combined with Thread-Level Speculation, and in research question 7 we evaluate the effects on power, by using Thread-Level Speculation on an embedded device.

Research question 5: Can we reduce the memory usage and improve the performance of Thread-Level Speculation by adaptively adjusting important parameters?

Research question 6: Is a Just-in-time compilation based JavaScript engine applicable for Thread-Level Speculation to improve the performance of web applications?

Research question 7: What are the effects on power consumption when using Thread-Level Speculation on embedded devices.

Research question 1 and research question 2 are discussed in paper I and paper II, where we perform the evaluations of the execution behavior of popular web applications and benchmarks. We perform the evaluations by using the Firebug JavaScript profiler and a modified version of WebKit.

Research question 3 is discussed in paper III and paper IV, where we implement Thread-Level Speculation in two JavaScript engines and evaluate these implementations using popular web applications and the V8 benchmark suite.

Research question 4 is further discussed in paper V where we have developed a heuristic for when to speculate or not.

Research question 5 is discussed in paper VI, where we adjust the amount of memory, the number of threads and the depth of speculation during Thread-Level Speculation. In paper VII we use the measurements of paper VI to develop a heuristic which adaptively tunes parameters to reduce the memory usage of Thread-Level Speculation in JavaScript engines.

Research question 6 is discussed in paper VIII, where we show that Thread-Level Speculation and Just-in time compilation are two complementary techniques, and that we can take advantage of Thread-Level Speculation with a JavaScript engine with Just-in-time compilation for web applications.

Research question 7 is discussed in paper XI where we show that we can use Thread-Level Speculation and Just-in time compilation to both reduce the execution time, and the power usage on embedded devices.

1.5 Research methodology

1.5.1 Measuring the Execution Behavior of JavaScript benchmarks and web applications

Research question 1 and research question 2 address how to evaluate the execution behavior of the benchmarks and the web applications. We have created a set of use cases which illustrate the behavior of popular web applications. We have used two methods to evaluate this; the JavaScript profiler firebug [23] and a modified version of Webkit and Squirrelfish.

Firebug evaluates the JavaScript code, such as function names, function types and the amount of time spent executing each function. By modifying Webkit and Squirrelfish, we are able to evaluate additional information such as the executed bytecode instructions, what kind of bytecode instructions that are executed, and the execution time from outside of the JavaScript engine.

A key challenge when evaluating the execution time, function names and types, and the bytecode instructions of web applications, is to make the results reproducible. Web applications do not have a clear start and end state. For instance, if we evaluate the JavaScript execution in *Twitter*, its execution behavior might be altered by external events, such as server side functionalities, which are

beyond our control. If we evaluate the execution behavior of *Twitter* twice, the web application might receive different external events, which again will affect the evaluation of the execution behavior. To increase the reproducibility, we have reduced the amount of user interaction, and scripted predefined behavior with Autoit [11].

JavaScript is a dynamically typed language, with functional features such as anonymous functions and eval calls and executed JavaScript might be different for two identical use cases. To cope with this, we have executed each use case 10 times, and selected the median result out of the 10 for comparison. To evaluate the JavaScript functionalities, we have selected five different applications which use JavaScript:

(i) A set of four established benchmark suites for JavaScript, Google V8, SunSpider, Dromaeo and JSBenchmark [29, 107, 71, 39].

(ii) The top 100 most visited web applications from the Alexa list [77].

(iii) A set of use cases on a selection of social networks [109].

(iv) A selection of 15 popular web applications, selected based on their popularity.

(v) The top 109 web applications in HTML5 from the JS1K competition [38].

We have performed the measurements on a Linux based computer with a modified version of Webkit and 16 GB of internal memory. The use cases, which are scripted, are executed by opening a terminal into the Linux computer from a Windows computer, which executes to Autoit scripts. This way, we ensure that the behavior of the script is not distrupting the execution on the Linux computer.

1.5.2 Implementing and evaluating Thread-Level Speculation

Research question 3 and research question 4 address the implementation and performance of Thread-level Speculation in JavaScript. For research question 3 and research question 4 we have made two implementations of Thread-Level Speculation; One in the Rhino JavaScript engine [20], and the second one in the Squirrelfish engine (which is part of Webkit) [108]. The Rhino engine is a stand alone JavaScript engine implemented in Java, and is not part of any official browser implementation. In addition, it lacks some of the functionalities

needed to fully support certain workloads in web applications. The Squirrelfish JavaScript engine and Webkit are available on a number of platforms (for instance the Sony Xperia Z1 phone), and the engine supports web applications

To implement Thread-Level Speculation, we use method-level speculation; therefore we have done the following: whenever we encounter a function call, we have created a new thread for this function call. We also store the state before we execute the function as a thread. If this thread is in conflict with any of the other threads, we use the stored information to rollback. Once its associated execution completes, the results of the execution are merged to the main thread. The main difference between the two implementations is that the Squirrelfish version allows function calls that are encountered in speculated functions to create new threads, i.e., support nested speculation while the previous version we made in the Rhino engine, does not support this.

We have evaluated the effects of Thread-Level Speculation by evaluating the execution time in the Rhino and in the Squirrelfish JavaScript engines. In Rhino we have evaluated the benchmarks in the Google V8 benchmarks suite and in Squirrelfish, we have selected 15 web applications that are within the top 100 Alexa list, and that represent different uses of web applications. In addition we have evaluated the success of a return value prediction, where rollbacks occur during execution, the memory overhead of storing data in case of a rollback and various metrics related to speculations. All the evaluations are made on a dual quad-core computer with a Linux operating system.

In research question 6 and research question 7 we have implemented Thread-Level Speculation for the Just-in-time based JavaScript engine V8, so we can measure the effects of running a large number of web applications, and have measured the effects in terms of improved execution time and power consumption.

Finally, we have measured the power usage of Thread-Level Speculation on a Sony Xperia X1 phone. We have done this by compiling our test version of the chromium web browser which can be executed on a Sony Xperia Z1 phone. We can then measure the effects of our use cases' power usage by using a battery simulator. We can measure the improved execution time on this device in terms of execution time compared with the sequential execution time. We measure the power usage, and compare the sequential power usage with the power usage of Thread-Level Speculation.

1.6 Contributions

We address research question 1 and research question 2 in paper I and paper II respectively, and in paper III, paper IV and paper V we address research question 3 and research question 4. Paper VII, paper VIII and paper IX address the research questions 5 and 6.

1.6.1 Contributions in Paper I

The main focus in paper I is to develop repetable methods for evaluating the execution behavior of a set of web applications with a set of JavaScript benchmarks. There are multiple problems with comparing these two. For instance, JavaScript benchmarks execute solely within the JavaScript engine, while web applications have parts of the program flow (e.g., events) in the web browser. One of the key problems with the comparison is that unlike benchmarks, web applications do not have a start and an end state. To make web applications and benchmarks more comparable, we have created a set of use cases for the web applications (of what we consider as common behavior). We evaluate how certain language features are used, and the importance of these execution time wise in web applications and we evaluate how often the code changes for identical workloads. To extract results for this paper, we use the profiler Firebug and a modified version of Webkit.

The main contributions of this paper are the following; We propose a methodology to measure, characterize, and evaluate the JavaScript execution behavior of interactive social networking web applications such as *Facebook*, *Twitter*, and *MySpace*. We do this by defining a set of use cases that represent typical user operations for the selected web applications. These use cases are then deterministically executed using a scripted and controlled environment. Second, our evaluations confirm the conclusions in several other studies, e.g., [84, 89], that there are significant differences in the execution behavior between real-world web applications and established benchmarks and third, we identify one unpublished significant difference between web applications and the established benchmarks, i.e., the use of anonymous functions.

1.6.2 Contributions in Paper II

Paper II extends paper I, previous research has shown that the established benchmarks suites are not representative for web applications [84, 89]. In paper II

we perform a bytecode instruction analysis to compare the types of bytecode instructions that have been executed for the benchmarks and the web applications. We evaluate the importance of the eval and anonymous functions for web applications. We compare the effects of Just-in-time compilation versus when Just-in-time compilation is not enabled for benchmarks and web applications. We extend our evaluations from paper I with a set of use cases, the first 100 web applications in the Alexa list, and the first 109 entries of the JS1K JavaScript competition from 2010, where each entry used HTML5.

The main contributions are; Just-in-time compilation is beneficial for many of the benchmarks, but *increases* the execution time for more than half of the web applications. We see that arithmetic/logical bytecode instructions are significantly more common in benchmarks, while prototype related instructions and branches are more common in web applications. The `eval` function is much more commonly used in web applications than in benchmarks. Approximately half of the benchmarks use anonymous functions, while approximately 75% of the web applications use anonymous functions.

1.6.3 Contributions in Paper III

Paper III binds together paper I and paper II. Since the workload of benchmarks is fundamentally different from the workload of web applications, Just-in-time compilation might not work as good as for the benchmarks. We evaluate the types of bytecode instructions for the SunSpider benchmarks and evaluate the improved execution time for the V8 benchmarks.

Our main contributions are; from the executed bytecode instructions, we found that there will not be a large number of arithmetic instructions and it will be few jump instructions. This means that focusing on executing large loops faster might be fruitless for the web applications. We repeat the evaluations of the V8 benchmarks and see from the performance evaluations that we are able to decrease the execution time for two of the benchmarks, however for the other benchmarks the execution time increases.

1.6.4 Contributions in Paper IV

Knowing about the differences between benchmarks and web applications, this paper addresses improvements by using Thread-Level Speculation for the JavaScript workload found in web applications. We evaluate the location of rollbacks dur-

ing execution, the amount of memory requirements right before each rollback, the number of speculations, the maximum number of active threads, the number of rollbacks, the depth when searching for irrelevant information for deletion at each rollback, the depth of the speculations, the relative factor of rollbacks and speculations and the improved execution time.

The main contribution are; the first implementation of Thread-Level Speculation in a state-of-the-art JavaScript engine, i.e., Squirrelfish [108] which is part of Webkit. Performance evaluation of Thread-Level Speculation for 15 popular web applications, e.g., *Facebook*, *Gmail*, *YouTube*, and *Wikipedia*. We continue with the evaluation of Just-in-time compilation, and show that Just-in-time compilation degrades the execution time of JavaScript in web applications, not only when we enable Just-in-time compilation for Squirrelfish, but also for Google's V8 JavaScript engine. Our results show significant speedups with Thread-Level Speculation for most of the studied web applications, with up to 8.4 times speedup in the best case, on eight cores, and include a detailed analysis of the speculation and rollback behavior as well as the memory overhead.

1.6.5 Contributions in Paper V

When we encounter a JavaScript function suitable for speculation, we try to speculate on this function. If this function fails (i.e., it is responsible for a misspeculation) we mark this function as not suitable for future speculations, which means that we do not speculate on this function in the future.

In this paper, we challenge this idea, by allowing ourselves to respeculate on functions that we already speculated on and that caused a misspeculation. We try two well known heuristics (albeit not in Thread-Level Speculation, but rather in another context) for JavaScript in web applications; the 2 bit heuristic and the exponential heuristic. Out of these, we develop a new heuristic, where we combine the two. We see from the results that we are able to speed up the execution time of Thread-Level Speculation, and with the combined heuristic, we are able to lower the relationship between speculations and rollbacks. This means, that for each speculation with this heuristic, rollbacks become rarer.

Even though one of the contributions proves that we can improve the execution time with an alternative heuristic, it also shows that the main problem in Thread-Level Speculation for JavaScript in web applications is not the number of rollbacks, but the memory overhead of the speculations.

1.6.6 Contributions in Paper VI

Paper IV and paper V clearly show that the main problem with Thread-Level Speculation in JavaScript for web applications isn't the cost and the number of rollbacks, but the memory usage, i.e., we often make 5000 speculations with a misspeculation rate of less then 3%. However, we do not know when the rollbacks are going to occur therefore we need to save in case of a rollback.

In this paper we challenge this, and adjust the available memory, the number of threads and the speculation depth. Our main contribution is that we can adjust these parameters to significantly reduce the memory overhead and speed up the execution time. More specifically, we show that for JavaScript in web applications, it is sufficient with 32–128 MB memory, 16 threads, and a speculation depth of 4–16. We also see that as we speculate deeper the number of rollbacks slightly decreases, and we show that this is due to the execution characteristics of JavaScript in web applications. We also show that we need to use nested speculation in order to find a sufficient number of speculations for Thread-Level Speculation in JavaScript for web applications.

1.6.7 Contributions in Paper VII

In Paper VI we saw the effects of limiting the amount of memory, the number of threads and the speculation depth. However, the fact that we needed to manually tune these parameters, made it unpractical for real life use cases.

In this paper we develop an adaptive heuristic to significantly reduce the memory overhead (90%) and improve the execution time by being more careful when we store the checkpoint. We also show that we cannot tune after a fixed value in JavaScript because of the execution characteristics of JavaScript, i.e., there is going to be a significant number of function calls and the number of function calls are going to vary during execution.

1.6.8 Contributions in Paper VIII

Just-in-time compilation is a technique to speed up the execution time, by compiling interpretive code into native code, and thereby (hopefully) decrease the overhead of the execution.

The main contribution in this paper is that we show that Thread-Level Speculation and Just-in-time compilation are two complementary techniques, and in this paper we show that due to the execution characteristics of JavaScript in web applications, the two techniques can be successfully combined. We also show that there are features in the V8 JavaScript engine that could be suitable for Thread-Level Speculation.

We also show that Thread-Level Speculation can be used on a larger number of web application, HTML5 as well as on a number of Google map use cases to improve the performance.

1.6.9 Contributions in Paper IX

In this paper, we repeat the measurements from the previous experiments, only that this time we make the measurements on a quadcore smartphone, and in addition to measuring the execution time, we also measure the power usage while running the same use cases.

The contributions in this paper, are the following; we show that Thread-Level Speculation can be used to improve the performance of embedded multicore devices. The maximum power usage by using Thread-Level Speculation on these devices can be expensive compared to sequential execution. However, the integral of the power usage is often smaller when we use Thread-Level Speculation than when we execute it sequentially. This indicates that we can use Thread-Level Speculation to both increase the performance of the execution time, as well as reducing the power usage on embedded devices.

1.7 Validity of results

We discuss the threats to the validity of studies, more specifically how generalizable our results are; How clear is it that the effects of our studies come from our modifications?

1.7.1 Generalizability

Generalizability refers to the possibility of generalizing the study results in a setting outside of the study [90]. In paper I and paper II, much of our critique

against current evaluations is that benchmarks are not representative for web applications. This can be seen by the size of the problems of the benchmarks, the large differences in executed bytecode instructions, the effects of Just-in-time compilation, and the importance of JavaScript language functionalities.

We found that there are differences between benchmarks and web applications. For instance, many web browsers have security mechanisms which prevent large loops. This conforms to the benchmarks being small instances of problem sizes, web applications are virtually without loop related bytecode instructions, a large number of functions and the missing effect of for instance Just-in-time compilation. This suggests that we can in general say that there will be no large loops in the JavaScript code. Since the loop construct is an important factor in many of the benchmarks, we can generalize and say that the benchmarks are unrepresentative.

Even though we know that the benchmarks are unrepresentative, we do not know which parts of the web applications that are representative. The first page without any user input might happen all the time, but it tells us very little about the common workload. We have tried to create use cases that illustrate the behavior of users in web applications; however the use cases are based on a small number of users.

In paper III, paper IV and paper V we see the effects of Thread-Level Speculation for benchmarks and web applications. We see that the effects are limited for the benchmarks and promising for the web applications. From previous sections there is a risk that benchmarks are unrepresentative for the JavaScript workload of web applications; We can however generalize the following; Events in web applications are asynchronous and independent, the number of anonymous function calls are bound to be quite large, since this is a key component to events and the eval function is a key component in web applications.

One important effect of paper I and paper II, is that since the benchmarks are unrepresentative for JavaScript execution in web applications then paper V shows that the popular optimization techniques, Just-in-time compilation slows down the execution time compared to not using Just-in time compilation. We have shown that this is not only the case for the Squirrelfish JavaScript engine, but also for Google's V8 JavaScript engine as well.

Paper VI shows that the main problem with Thread-Level Speculation in JavaScript isn't the number of rollbacks or the effects of rollbacks, but rather the memory usage. Paper VII shows that we can get away with much less memory, less threads and less speculation depth, however this paper also shows that we need to use nested speculation for Thread-Level Speculation in order improve the

execution time in Thread-Level Speculation for JavaScript in web applications. Paper VII shows that we can create an adaptive heuristic to automatically tune the speculation depth, in order to significantly reduce the memory overhead and improve the execution time. Finally paper VIII and paper XI show that, even though the technique Just-in-time compilation in its current form often increases the execution time of JavaScript in web applications, it is a complementary technique to Thread-Level Speculation, and due to the workload characteristics of web applications, Thread-Level Speculation and Just-in-time compilation can be combined to reduce the execution time and the memory usage.

1.7.2 Internal validity

Internal validity refers to how well a study can establish the relationship between cause and effect [90]. We know from our experiments that there are a large number of function calls. In paper III, paper IV and paper V we evaluate the effects on a set of benchmarks as well as on a set of web applications. In order to avoid issues with web applications, we were forced to store the executed bytecode for later re-execution. This did not fully illustrate the dynamics of the web applications, issues which were described in paper I and paper II.

Using Thread-Level Speculation for web applications results in a low number of rollbacks, because functions are short lived, and there are a lot of them. This is shown in Paper IV and in Paper V. The main problem of this is however, that we do not know when a speculation is going to cause problems, so we need to save all the checkpoints. Paper IV and Paper V show that the main problem is the memory overhead, and that we need to use nested speculation. Paper VII, shows how we can adaptively tune the checkpoint saves with the checkpoint depth. Paper VIII, shows that due to the findings in paper I and paper III, Just-in-time compilation and Thread-Level Speculation can be successfully combined to improve the execution time and reduce the memory usage of JavaScript in web applications.

1.7.3 Constructive validity

Constructive validity refers to how well the results match against other scientific results or how representative they are for real life results. In paper II and in paper IV we first show that the official benchmarks are unrepresentative and in paper V we show the consequence of this which is in line with [12, 84, 89]. We

also select a set of applications used by a large number of people (i.e., from the Alexia list)

However, real life web applications have many function calls, and we show in paper V that these can be taken advantage of with Thread-Level Speculation. To perform the experiments, we use a strict methodology, and make measurements on very popular web applications.

1.7.4 Conclusion validity

Conclusion validity refers to how certain we are that the cause and the results are correct [18]. One obvious threat in this study is that there initially in paper V is a small number of use cases, and that we tested it on one system. However, we can address these threats the following way; we selected web applications that are very popular, therefore we expect that these web applications have certain features which other less popular web applications use as well. Secondly, in paper VIII and paper IX we increase the number of use cases over the same benchmarks, and add two new classes of web application to our study, and we perform the experiments with a different JavaScript engine. Still the results show that Thread-Level Speculation improves the execution time with more use cases of the selected web application, and also on a larger set of web applications. From this we can conclude that these results are valid for a larger number of web applications.

1.8 Conclusion

To answer research question 1 we minimize the user interaction and make the execution of use cases automated to make them more reproducible and repeatable. For research question 2 we have collected data that show that the workloads of web applications are not well represented by the benchmarks. This has several important effects, for instance that current techniques for improving the execution speed, might fail for web applications. Another interesting result of these differences is that special features in JavaScript like anonymous functions and eval functions are commonly used in web applications.

To answer research question 3 and 4 we have implemented Thread-Level Speculation on two JavaScript engines, Rhino and Squirrelfish. We were able to execute the JavaScript in our web applications over 8 times faster with our implementation compared to the sequential version for 15 web applications. While

the number of speculations is high, the number of rollbacks is low. The results indicate that Thread-Level Speculation is a promising technique for web applications. For the V8 benchmarks, we were not able to improve the execution time for most cases. In addition the memory requirements where between 1 MB to 33 MB for the web applications, to ensure that we were able to restore the application to a point in time when the execution was correct.

To further answer research question 4 we have developed a heuristic to respeculate functions that have previously been speculated. The results show that while this improves the execution time, the main problem is the memory overhead.

To answer research question 5, we tuned three important parameters for Thread-Level Speculation, and we show that we do not need to save every checkpoint state and that nested speculation is necessary for Thread-Level Speculation. We also saw that we could tune the depth of nested speculation with an adaptive heuristic to significantly reduce memory usage and improve the execution time.

To answer research question 6, we implemented Thread-Level Speculation in a Just-in-time enabled JavaScript engine. Our results show that Thread-Level Speculation and Just-in-time compilation are complementary techniques, and these can be combined. We also show that due to the characteristics of JavaScript in web applications it is very suitable for Thread-Level Speculation.

To answer research question 7, we further evaluate the power usage by running the previous measurements, and show that we are able to reduce the power usage with Thread-Level Speculation.

The results of this thesis clearly show that Thread-Level Speculation is a viable technique to improve the performance and reduce the power usage of JavaScript in web applications.

1.9 Future work

We have shown that Thread-Level Speculation increases the JavaScript performance in web applications on a wide range of devices with multiple cores (from large server systems to embedded devices). However, there are more things to investingate.

How well does Thread-Level Speculation work with other managed languages? Web applications are for instance suitable for Thread-Level Speculation, however is Thread-Level Speculation suitable for other languages highly relevant to web

development, such as Dart? The Dart language can be compiled to JavaScript code. How well does the compiled JavaScript code work with Thread-Level Speculation? There are other managed languages where Thread-Level Speculation could be evaluated, such as Mono (the open source variant of C#).

We also saw that there has previously been attempts to use Thread-Level Speculation with Java (e.g., in Jrmp and SableVM, which do not support nested speculation). Could we further investigate the effect of Thread-Level Speculation on other features in these langauges, and extend their implementations to support Just-in-time compilation and nested speculation? Due to the importance of JIT in Java, would an extension of their implementations further improve the performance?

In the last paper, we show that one can reduce the power usage in web application, and at the same time improve the execution time. Could we instead of speculating to improve the performance, speculate to reduce the power usage? Can we for instance identify parts of web applications that are power hungry?

Chapter 2

Paper I

A Methodology for Evaluating JavaScript Execution Behavior in Interactive Web Applications

Jan Kasper Martinsen and Håkan Grahn

Proceedings of the Ninth ACS/IEEE International Conference on Computer Systems And Applications (AICCSA 2011), pages 241-248, December 2011, Sharm El-Sheikh, Egypt

2.1 Introduction

JavaScript was introduced primary as an interpreted prototype based scripting language for web pages, which allowed programmers to add interactivity to web pages [24, 37]. With JavaScript these web pages where given application like behavior. As a results, a larger number of more or less sophisticated ports of typical desktop-like applications became accessible as web pages. One typical example is Google's mail client gmail [93]. These web pages are informally known as web applications [110].

One important advantage of web applications is the ease of application distribution. Installing a conventional application usually requires that you are careful

so it gets installed onto a correct operating system and on a machine with certain specifications. In contrast, web applications can essentially be accessed and executed directly from any (reasonably modern) web browser.

Social networking [6] has become a popular type of web applications. Facebook seems to be the most popular one and is number two on the Alexa list of the most popular web sites [4, 77]. Many of the entries among top 25 web sites are social networks, e.g., Facebook [74], Twitter [43], and MySpace [79]. Several studies have confirmed the popularity of social networking web applications [21, 22, 69].

Due to the popularity and ease of distribution of web applications, JavaScript has become a very popular programming language. There have also been several approaches to improve the performance of the JavaScript interpreter [26, 27, 87].

To measure and evaluate the performance of JavaScript interpreters (and thereby measuring and quantifying the results of the interpreter optimization), a set of benchmark suites have been proposed [29, 71, 107]. Some of the critique [84, 89] against these suites, is that several of the benchmarks have been ported from benchmarks in, e.g., operating systems and numerical research. While these might help us to improve certain aspects of the interpreter, there is a significant risk that the execution behavior of these benchmarks might not fully reflect the JavaScript behavior of actual web applications, such as social networking web applications.

In this paper, we make three main contributions:

- First, we propose a methodology to measure, characterize, and evaluate the JavaScript execution behavior of interactive social networking web applications such as Facebook, Twitter, and MySpace. We do this by defining a set of use cases that represent typical user operations for the selected web applications. These use cases are then deterministically executed using a scripted and controlled environment.
- Second, our measurements confirm the conclusions from several other studies, e.g., [84, 89], that there are significant differences in the execution behavior between real-world web applications and established benchmarks.
- Third, we identify one unpublished significant difference between web applications and the established benchmarks, i.e., the use of anonymous functions.

The rest of the paper is organized as follows. Section 2.2 presents some background and previous work, and then Section 2.3 presents the benchmarks

and web applications that we use. In Section 2.4, we present our methodology to do workload characterization of interactive web applications. Section 2.5 presents our measurement results. Section 2.6 discusses some directions for future work, and finally, we conclude our findings in Section 2.7.

2.2 Background

2.2.1 JavaScript and web applications

JavaScript [24, 37] was introduced by Netscape in 1995 as a way to allow web developers to add dynamic functionality to web pages that were executed on the client side. The purposes of the functionality were typically to validate input forms and other user interface related tasks. JavaScript has gained momentum over the years, particularly due to its ease of deployment and the increasing popularity of certain web applications, e.g., Gmail [94]. We have found that almost all of the first 100 sites in the Alexa-top sites list [4] include some JavaScript functionality.

JavaScript is a dynamically typed, object-based scripting language with runtime evaluation. The execution of a JavaScript program is done in a JavaScript engine [30, 72, 108], i.e., an interpreter/virtual machine that parses and executes the JavaScript program. Due to the popularity of the language, there have been multiple approaches to increase the performance of the JavaScript engines, through well-known optimization techniques such as just-in-time (JIT) compilation techniques, fast property access, and efficient garbage collection [27, 30].

2.2.2 Previous work

With the increasing popularity of web applications, it has been suggested that the web browser could serve as a general platform for applications in the future. This would imply that JavaScript needs increased performance. Further, it also mean that one would need to look deeper into the workload of actual web applications. This process is in its early phases, but there are several examples of interesting work [74, 5]. Two concurrent studies [84, 89] explicitly compare the JavaScript execution behavior of web applications as compared to existing JavaScript benchmark suites.

The study by Ratanaworabhan et al. [84] is one of the first studies that compares JavaScript benchmarks with real-world web applications. They instrumented the Internet Explorer 8 JavaScript runtime in order to get their measurements. Their measurements were focused on two areas of the JavaScript execution behavior, i.e., (i) functions and code, and (ii) events and handlers. Based on the results, they conclude that existing JavaScript benchmarks are not representative of many real-world web applications and that conclusions from benchmark measurements can be misleading. Examples of important differences include different code sizes, web applications are often event-driven, no clear hotspot function in the web applications, and that many functions are short-lived in web applications. They also studied memory allocation and object lifetimes in their study.

The study by Richards et al. [89] also compares the execution behavior of JavaScript benchmarks with real-world web applications. In their study, they focus on the dynamic behavior and how different dynamic features are used. Examples of dynamic features evaluated are prototype hierarchy, program size, object properties, and hot loop (hotspots). They conclude that the behavior of existing JavaScript benchmarks differ on several of these issues from the behavior of real web applications.

2.3 Benchmarks and web applications

2.3.1 JavaScript benchmarks

There exist three established JavaScript benchmark suites: V8 [29], Dromaeo [71], and Sunspider [107]. The applications in these benchmark suites generally fall into two different categories: (i) testing of a specific functionality, e.g., string manipulation or bit operations, and (ii) ports of already existing benchmarks that are used extensively for other programming environments [2].

For example, the benchmarks Raytrace, Richards, Deltablue, and Earley-Boyer are included in the V8 benchmark suite. Raytrace is a well-known computational intensive graphical algorithm for rendering scenes [105]. Richards simulates an operating system task dispatcher, Deltablue is a constraint solver, and Earley-Boyer is a type theorem prover benchmark. In contrast, the Dromaeo benchmarks test specific JavaScript language features.

Typical for the established benchmarks is that they often are problem oriented, meaning that the purpose of the benchmark is to accept a problem input, solve this certain problem, and then end the computation. This eases measure-

Table 2.1: A summary of the benchmark suites used in this paper.

Benchmark suite	Applications
Dromaeo	3d-cube, core-eval, object-array, object-regexp, object-string, string-base64
V8	crypto, deltablue, earley-boyer, raytrace, richards
SunSpider	3d-morph, 3d-raytrace access-binary-trees, access-fannkuch, access-nbody, access-nsieve bitops-3bit-bits-in-byte, bitops-bits-in-byte, bitops-bitwise-and, bitops-nsieve-bits controlflow-recursive crypto-aes, crypto-md5, crypto-sha1 date-format-tofte, date-format-xparb math-cordic, math-partial-sums, math-spectral-norm regexp-dna string-fasta, string-tagcloud, string-unpack-code, string-validate-input

ments, gives the developer full control over the benchmarks, and increases the reproducibility.

2.3.2 Social networking web applications

There exists many so-called social networking web applications [109], where Facebook [74] is the most popular one [4, 22]. There are even examples of countries where half of the population use Facebook to some extent during the week [21]. The purpose and usage of social web applications might have many facets. However, the key element for a social networking application to be successful is to have a certain critical mass of users.

The users of a social networking web application can locate and keep track of friends or people that share the same interests. This set of friends represents each user's private network, and to maintain and expand a user's network, a set of functionalities is defined. For example, users can create petitions to vote for a certain cause, while other users can play video games where the final 'score' is compared with other friends in their own networks.

In this paper we study the social networking web applications Facebook, Twitter [43], and MySpace [79]. In a sense, Facebook seems to be a general purpose social networking web application, with a wide range of different functionality. Further, Facebook also has the largest number of users.

Twitter is for writing small messages, so called "tweets", which are restricted to 160 characters (giving a clear association to SMS). The users of Twitter are able to follow other people's tweets, and for instance add comments in form of tweets to their posts.

MySpace seems to be especially coined at musicians, that wish to share or obtain music. Through MySpace the users can upload music, which they in turn distribute to other MySpace users. Users are also able to write comments and search for other users with similar music taste.

2.4 A methodology for evaluating JavaScript execution behavior

While the benchmarks have a clear purpose, with a clearly defined start and end state, interactive social networking web applications behave more like operating system applications, where the user can perform a selected number of tasks. As long as the web application is viewed by the user, it remains active and performs a set of underlying tasks.

When measuring and evaluating application or system behavior, as well as when defining benchmarks, two of the most important things are: (i) the application/benchmark should be *representative* and (ii) the measurements should be *reproducible*. How representative existing JavaScripts benchmark suites are for real-world web applications have been addressed in, e.g., [84, 89], and in this paper we identify some additional differences. However, the issue of reproducibility of web application behavior measurements have not been addressed in previous studies.

2.4.1 Representative behavior

In order to mimic a representative use and behavior of social network web applications, we have defined a set of use cases. Each use case has a clear start and end state. These use cases are intended to give a realistic idea of the actual work-

load in web applications and also provide repeatability of the measurements. The use cases that we have designed represent common user behavior in Facebook, Twitter, and Myspace, rather than exhausting JavaScript execution.

Figure 2.1, Figure 2.2 and Figure 2.3 show the different use cases that we have defined for Facebook, Twitter, and MySpace, respectively. All use cases start with the user login. Then, the user has multiple options.

For Facebook, the users login to the system, then the user searches for an old friend, which the user in turn finds. When the user finds this old friend, the user marks him as a "friend", an operation where the user needs to ask for confirmation from the friend to make sure that he actually is the same person. This operation is a typical example of an use case, which in turn is composed of several sub use cases: 0 `-login/home`, 0.3 `-find friend`, 0.3.1 `-add friend`, and 0.3.1.0 `-send request`, as shown in Figure 2.1.

All use cases start with the login case, and we recognize an individual operation, such as 0.3.1 `-add friend` as a sub use case, though it must complete previous use cases. Further, we do allow use cases that goes back and forth between use cases. For example in Figure 2.2, if we want to both choose the option 0.1.0 `-follow` and 0.1.1 `-mention`, then we would need to visit the following sub use cases: 0 `-login/home`, 0.1 `-find person`, 0.1.0 `-follow`, 0.1 `-find person`, and 0.1.1 `-mention`.

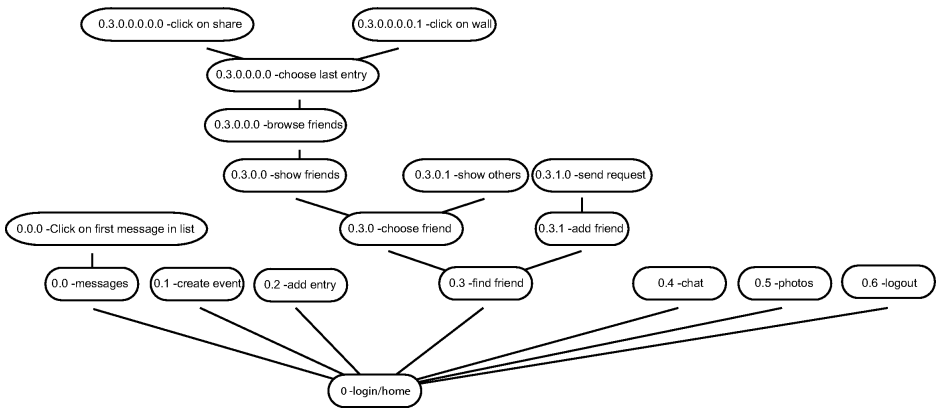

Figure 2.1: Use cases to characterize the JavaScript workload of Facebook.

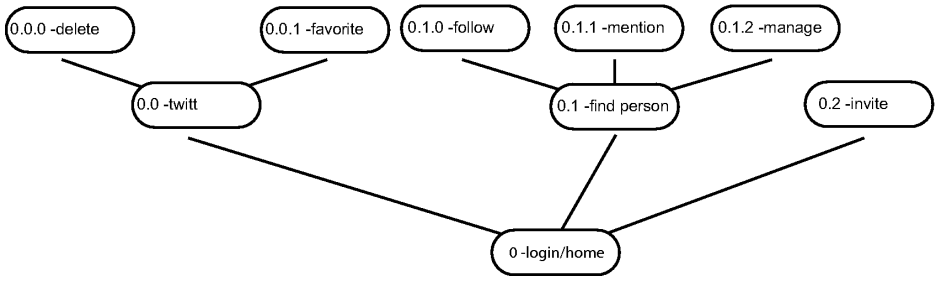

Figure 2.2: Use cases to characterize the JavaScript workload of Twitter.

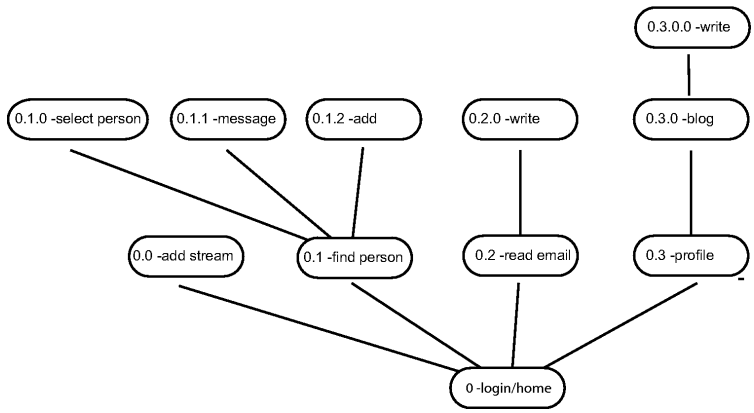

Figure 2.3: Use cases to characterize the JavaScript workload of MySpace.

2.4.2 Reproducible behavior

To enhance reproducibility, we use the AutoIt scripting environment [11] to automatically execute the various use cases in a controlled fashion. As a result, we can make sure that we spend the same amount of time on the same or similar operations, such as to type in a password or click on certain buttons. This is suitable for the selected use cases. However for certain operations, several social networks employ various web crawling countermeasures, e.g., through CAPTCHA [28] or restricts the number of login attempts.

We discovered by successively executing the same use case (the 0 `-login/home` Facebook use case) 10 times, that there is no guarantee that the executed JavaScript code would be identical in all the cases, even though the usage would be identical. Since JavaScript has a function such as `eval`, we can easily create script that dynamically generates JavaScript code. We have found that a certain fraction of the function names is unique for repetitions of identical cases, which suggests that changes occur between reloads or as a result of session specific code through AJAX calls. We also found that the number of function calls, and the number of functions that are called vary for identical cases as shown in Figure 2.4.

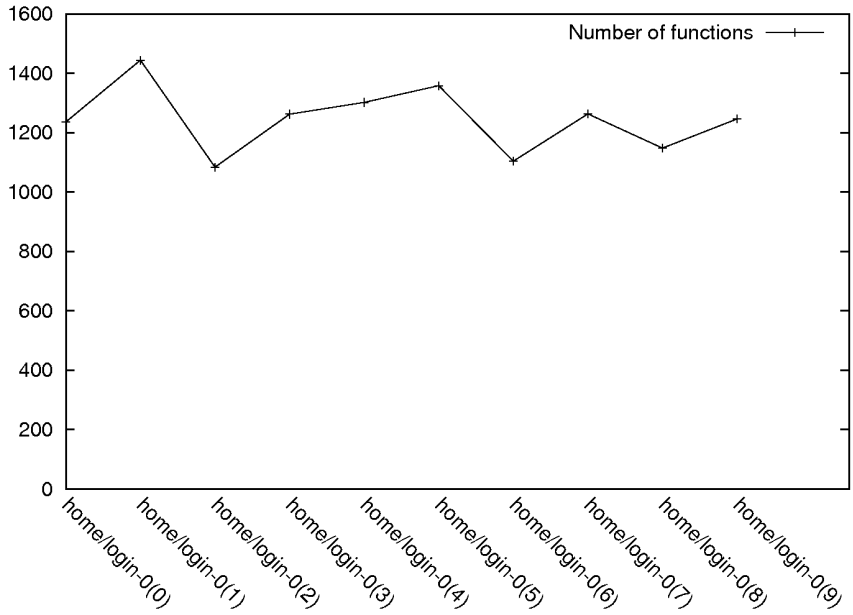

Figure 2.4: Total number of functions for each of the 10 repetitions of the same case.

A large fraction of these function calls is anonymous function calls (we will come back this issue in Section 2.5.2). Thus, we could argue that many of them were dependent on the input data, which could potentially change through AJAX calls [19]. However, at the same time, not all of them are anonymous calls for all of the 10 successive calls with functions that had unique function names.

To remedy this problem and simplify later analysis, we have done the following. For example, the two cases in Figure 2.3 (0 `-login/home`, 0.1 `-find`

`person`, 0.1.1 `-message`) and (0 `-login/home`, 0.1 `-find person`, 0.1.1 `-add`) both share the same actions (0 `-login/home`, 0.1 `-find person`), which we from now on denote as a sub case. However, as we saw above, the JavaScript execution for this sub use case might be radically different for the two cases. To simplify later analysis we have created two countermeasures, to make sure that the executed JavaScript code will be less different between the two cases.

By using the WebKit [99] tool, we have extracted and created a local copy of certain sub use cases. In more detail, when these parts are extracted, we first login to, e.g., Facebook, where there seems to be some so-called session variables that needs to be set. Then we open up the local copy, and profile this until we reach the point where we can select two different paths (e.g., `0.1.0 -add` or `0.1.1 -message`). We have found that this approach seems to work fairly well for some of the cases, but not for all of them. If this first approach does not work, we have used the following mechanism. We have instrumented and repeatedly executed the common use cases 10 times (e.g., `0 -login/home`, `0.1 -find person`) and then used the average of the common JavaScript execution profile for the measurements.

2.4.3 Experimental environment

To do the actual profiling we have used the Firebug v1.5.4 profiling tool running on a freshly installed Windows XP. Firebug runs on a custom compiled version of Firefox v3.6, which is able to automatically record executed JavaScript code as well as some simple instrumentation. Firebug reports a number of issues, and for JavaScript code it reports, e.g., the name of the JavaScript functions called, the amount of time each function is executed, the percentage of the total execution time the function uses, and the amount of time the function uses for execution. To extract use cases we have used a custom Ubuntu installation with WebKit.

2.5 Measurement results

2.5.1 Distribution of function calls and execution time

In order to understand the relative impact on the execution time of each function call, we have collected execution statistics of how many times each function is called and how much it contributes to the total execution time.

We have normalized the execution time for all the function call entries. To understand how these relates to functions that accounts for most of the applications execution time, we created a histogram for the execution time for both the benchmarks and the first 100 sites on the Alexa top sites list (Figure 2.5 and Figure 2.6). This histogram is divided into 10 categories, where each category accounts for the number of function calls that contributes to either $0-9\%$, $10-19\%$, $20-29\%$, $30-39\%$, $40-49\%$, $50-59\%$, $60-69\%$, $70-79\%$, $80-89\%$, or $90-99\%$ of the execution time.

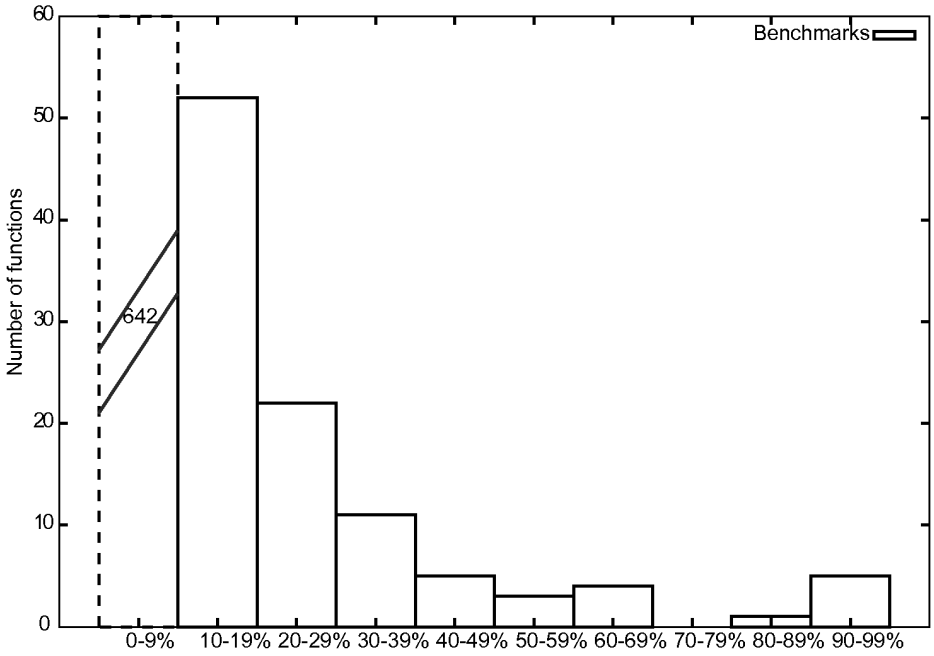

Figure 2.5: Histogram over the number of functions that contributes to a certain percentage of the total execution time for the benchmarks.

We see in Figure 2.5 and Figure 2.6 that both the benchmarks and the Web Applications have a large number of functions in the $0-9\%$ category, which indicate that there is a very large number of small functions executed. In Figure 2.5 we see that for the benchmarks, the workload is divided into most of the columns in the histogram. Especially, we find that there are a number of functions that account for more than 80% of the execution time, i.e., a clear hot spot function exists. In contrast, we see in Figure 2.6 that the execution time

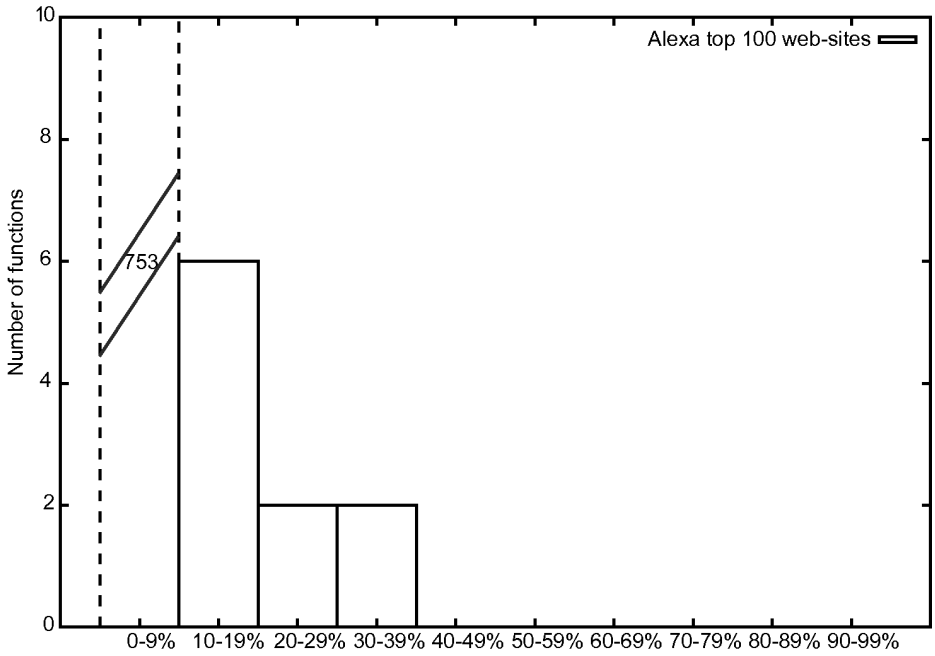

Figure 2.6: Histogram over the number of functions that contributes to a certain percentage of the total execution time for the Web Applications.

of the Web Applications only uses the first four categories. This means that no function dominates the execution time in the web applications, i.e., no hot spot exists in the code. In web applications, the workload seems to be more evently distributed, and *no* JavaScript function contributes to more than at most 39% of the total execution time.

In order to analyze the relative execution time for social network web applications, we show the relative fraction of execution time per function and the relative number of function calls per function for Facebook, Twitter, and MySpace in Figure 2.7, Figure 2.8, and Figure 2.9, respectively.

Our results show a high variance between the number of times a function is called and its contribution to the execution time for Facebook, Twitter, and MySpace. This indicates that there is a high variance in the execution times of individual functions. For example, we found that for the Facebook use case, only

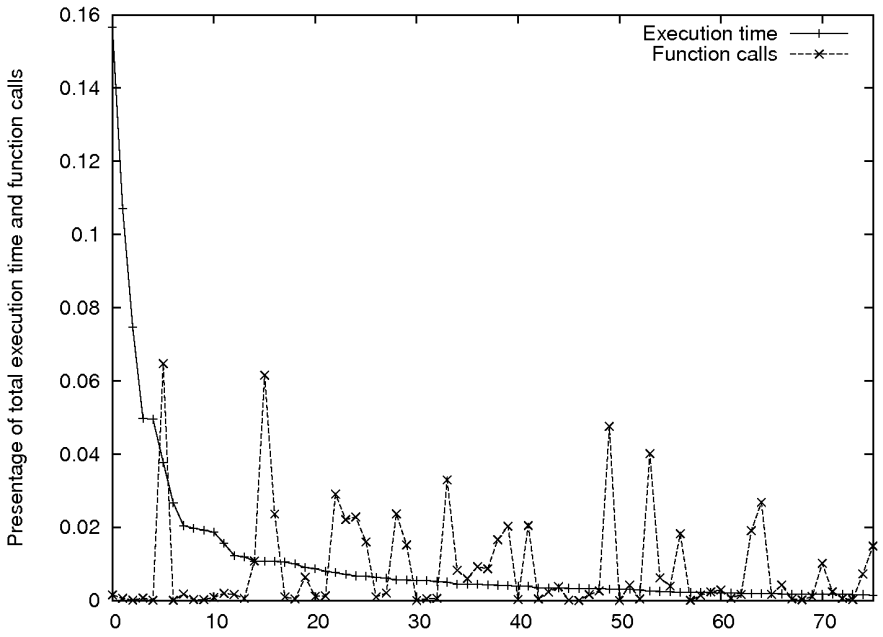

Figure 2.7: Relative number of function calls and the relative amount of execution time spent in each function for the Facebook use cases.

14 out of 75 function calls have the same relative number of function calls as the relative fraction of the execution time.

2.5.2 Anonymous function behavior

A previous study of Facebook reveals that a large number of anonymous function calls are made [54]. However, the same study reveals that these functions do not account for a large fraction of the total execution time. In Figure 2.10, Figure 2.11, and Figure 2.12, respectively, we have measured (i) the number of unique anonymous functions relative to the total number of unique functions, (ii) the total number of anonymous function calls relative to the total number of function calls, and (iii) the total execution time spent in anonymous functions relative to the total execution time for the use cases defined in Figure 2.1, Figure 2.2 and Figure 2.3.

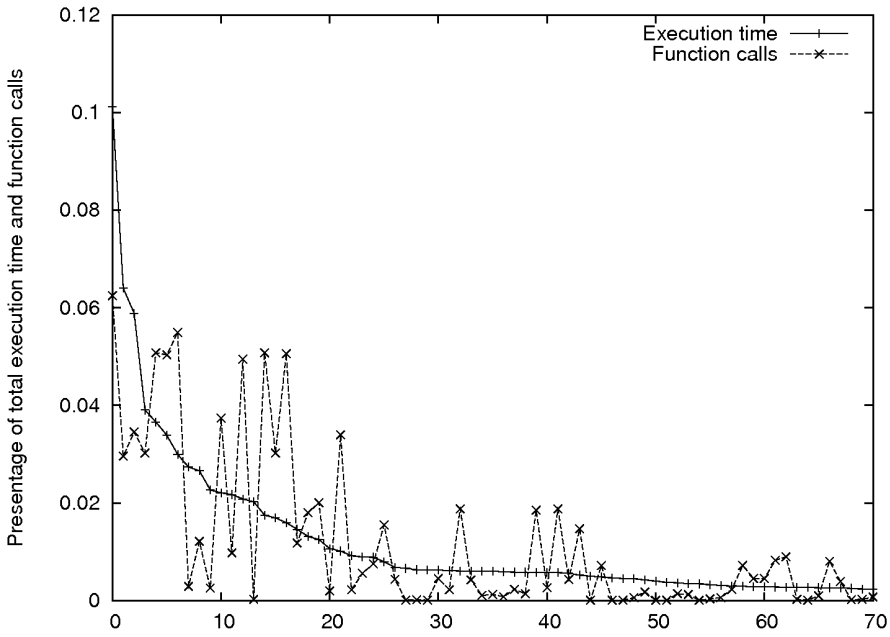

Figure 2.8: Relative number of function calls and the relative amount of execution time spent on each function for the Twitter use cases.

We see in Figure 2.10 that the number of unique anonymous function calls as well as the number of calls increase slightly as we complete the use case (25% and 31%). However, the execution time increases with a factor 5 between the login sub use case and the final use case. At the final sub use case, the anonymous function workload amounts to over half of the total workload.

However, from Figure 2.11 and Figure 2.12 we see that both Twitter and MySpace use fewer anonymous function calls than Facebook does. They have only a small number of unique anonymous functions, a small number of anonymous function calls and those functions that are called does only account for a minor part of the execution time.

In comparison, our results show that anonymous functions are used to a very little degree in the benchmarks. For instance, both the V8 benchmarks Raytracer and Earley-Boyer had both over 40000 function calls, but only 3 of them were

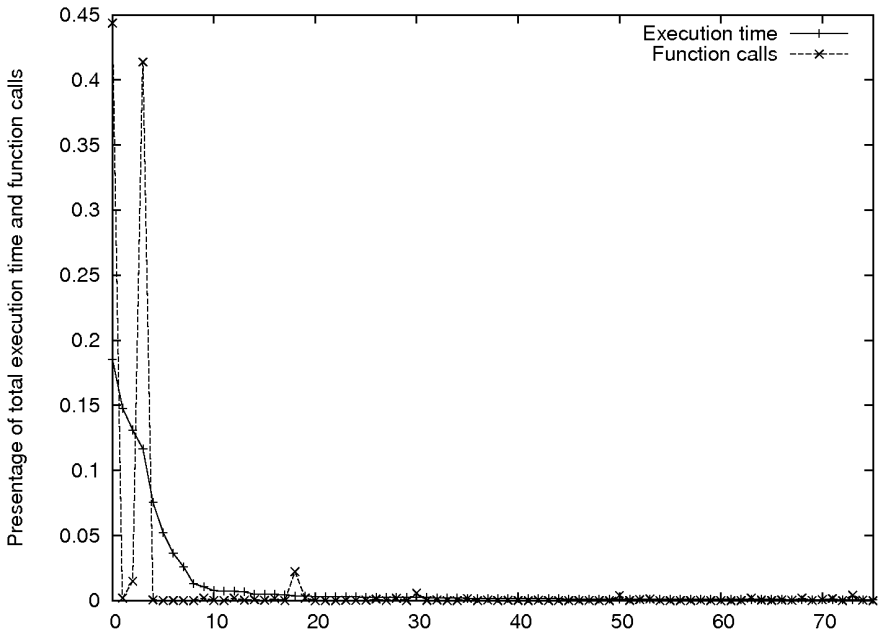

Figure 2.9: Relative number of function calls and the relative amount of execution time spent in each function for the MySpace use cases.

anonymous functions. In comparison, in Figure 2.10, for the use case where we search for friends, over 40% of the function calls were anonymous.

2.6 Discussion and future work

As pointed out in [84, 89] one could argue that the workload of JavaScript in a web application setting is not well represented by the established benchmarks. These studies and our own results suggest that the behavior of web applications is significantly different than for traditional programs, e.g., by being more event-driven and by utilizing dynamic updates of code at runtime. The lack of iterative constructs also could suggest that traditional JIT like optimization could be less effective for web applications than for the established benchmarks. This is something that will be examined in the future.

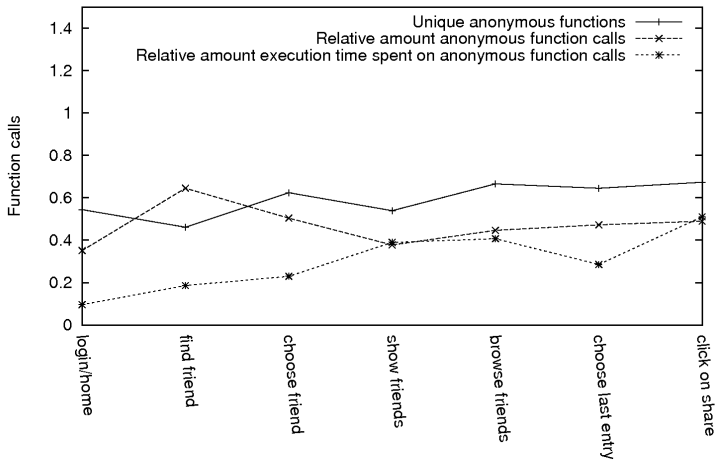

Figure 2.10: Anonymous function calls for Facebook.

However, we still need to be careful with our conclusions, and we will outline a couple of reasons why such a care is needed. Web applications is a fairly new concept that came together with the popularization of a set techniques known as Web2.0 and the possibility for dynamic updates. Our tests reveal the importance of such technologies in, e.g., Facebook. However, at the same time, there is a trend where richer multimedia possibilities are starting to be offered to the web through technologies such as for instance WebGL. For some of the workloads found in multimedia applications utilizing for instance 3D graphics, some of the current benchmarks would be more relevant.

Our study as well as other similar studies [84] are based on web applications. However, JavaScript has turned out to be a popular embedded language for multiple applications, e.g., an embedded language in the FireFox web browser, so its usage is in no way restricted to only web applications. Further, some of the workloads in web applications are spawned from functionalities that is not strictly part of the JavaScript specification, but rather part of the functionalities of the web browser.

Either way, JavaScript has some rather unique programming constructs, and as our test shows, a functionality such as anonymous functions are used extensively. These kind of functions are spawned from a different field, and are usually not available in other programming languages. That does, however, not mean

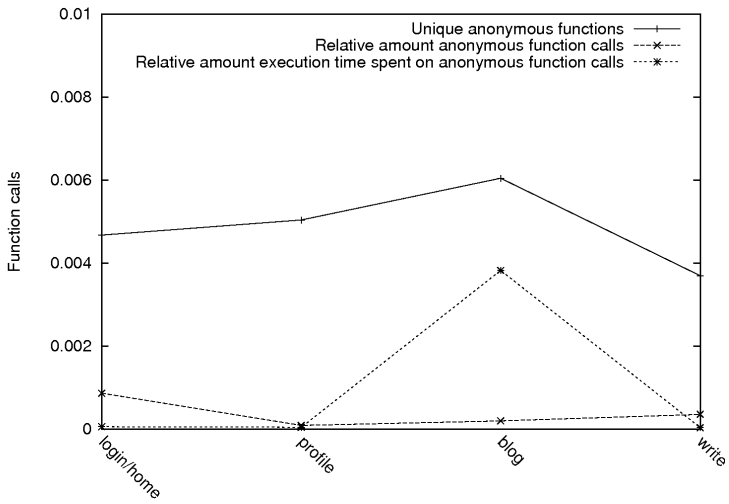

Figure 2.11: Anonymous function calls for Twitter

that they are not powerful. The established benchmarks address this issue only to a minor extent, and future benchmarks ought to take the usage of anonymous functions into account. We also suggest that there should be put some effort into simulating event-driven programs.

2.7 Concluding remarks

In this paper we have described a methodology to characterize the workload behavior of interactive web applications that are written in JavaScript. As part of the methodology, we have defined a number of use cases for three popular social networking applications, i.e., Facebook, Twitter, and MySpace. Further, we use an automatic scripting environment in order to enhance the repeatability of the measurements.

Our characterization of the workload behavior of social networking web applications shows some interesting differences as compared to the workload behavior of established JavaScript benchmarks. First, we have found that the correlation between the relative number of function calls and the relative amount of execution time spent in each function is significantly lower for web applications than for

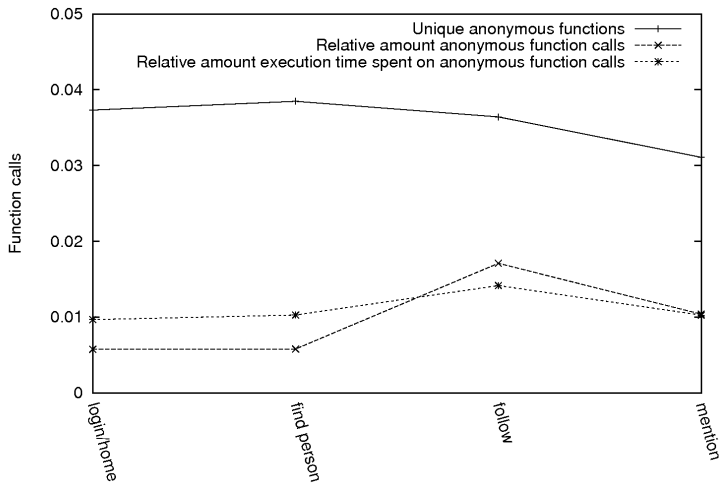

Figure 2.12: Anonymous function calls for MySpace

the benchmarks. Second, the studied web applications have a significantly larger amount of anonymous functions and function calls than the established benchmarks. Finally, the established benchmarks often contain loop constructs that account for a significant portion of the total execution time, while such hot spots have not been observed in the web applications. In contrast, web applications seem to be based on event-driven programming techniques.

Chapter 3

Paper II

Evaluating Four Aspects of JavaScript Execution Behavior in Benchmarks and Web Applications

Jan Kasper Martinsen, Håkan Grahn and Anders Isberg

Research Report, Number : 2011:03, ISSN : 1103-1581, Blekinge Institute of Technology, July 2011, Blekinge, Sweden

3.1 Introduction

The World Wide Web has become an important platform for many applications and application domains, e.g., social networking and electronic commerce. These type of applications are often referred to as web applications [106]. Web applications can be defined in different ways, e.g., as an application that is accessed over the network from a web browser, as a complete application that is solely executed in a web browser, and of course various combinations thereof. Social networking web applications, such as Facebook [75], Twitter [43], and Blogger [10], have turned out to be popular, being in the top-25 web sites on the Alexa list [4]

*A shorter version is published in Proc. of the 11th Int'l Conf. on Web Engineering (ICWE 2011), Lecture Notes in Computer Science No. 6757, pp. 399–402, June 2011.

of most popular web sites. All these three applications use the interpreted language JavaScript [37] extensively for their implementation, and as a mechanism to improve both the user interface and the interactivity.

JavaScript [37] was introduced in 1995 as a way to introduce dynamic functionality on web pages, that were executed on the client side. JavaScript has reached widespread use through its ease of deployment and the popularity of certain web applications [94]. We have found that nearly all of the first 100 entries in the Alexa top sites list use JavaScript.

JavaScript [37] is a dynamically typed, object-based scripting language with run-time evaluation. The execution of a JavaScript program is done in a JavaScript engine [30, 108, 72], i.e., an interpreter/virtual machine that parses and executes the JavaScript program. The popularity of JavaScript increases the importance of its run-time performance, and different browser vendors constantly try to outperform each other.

In order to evaluate the performance of JavaScript engines, several benchmark suites have been proposed, e.g., Dromaeo [71], V8 [29], SunSpider [107], and JSBenchmark [39]. However, two previous studies indicate that the execution behavior of existing benchmarks differs in several important aspects [84, 89].

In this study, we compare the execution behavior of four different application classes, i.e., (i) four established JavaScript benchmark suites, (ii) the start pages for the first 100 sites on the Alexa top list [4], (iii) 22 different use cases for Facebook [75], Twitter [43], and Blogger [10] (sometimes referred to as BlogSpot), and finally, (iv) 109 demo applications for the emerging HTML5 standard [32]. Our measurements are performed with WebKit [108], one of the most commonly used browser environments in mobile terminals.

We extend previous studies [84, 89] with several important contributions:

- First, we extend the execution behavior analysis with two new application classes, i.e., reproducible use cases of social network applications and HTML5 applications.
- Second, we identify the importance of anonymous functions. We have found that anonymous functions are used more frequently in real-world web applications than in the existing JavaScript benchmark suites.
- Third, our results clearly show that just-in-time compilation often *decreases* the performance of real-world web applications, while it increases the performance for most of the benchmark applications.

- Fourth, a more thorough and detailed analysis of the use of the `eval` function.
- Fifth, we provide a detailed bytecode instruction mix measurement, evaluation, and analysis.

The rest of the paper is organized as follows; In Section 3.2 we introduce JavaScript and JavaScript engines along with the most important related work. Section 3.3 presents our experimental methodology, while Section 3.4 presents the different application classes that we evaluate. Our experimental results are presented in Section 3.5. Finally, we conclude our findings in Section 3.6.

3.2 Background and related work

3.2.1 JavaScript

An important trend in application development is that more and more applications are moved to the World Wide Web [103]. There are several reasons for this, e.g., accessibility and mobility. These applications are commonly known as web applications [106]. Popular examples of such applications are: Webmails, online retail sales, online auctions, wikis, and many other applications. In order to develop web applications, new programming languages and techniques have emerged. One such language is JavaScript [24, 37], which has been used especially in client-side applications, i.e., in web browsers, but are also applicable in the server-side applications. An example of server-side JavaScript is node.js [76], where a scalable web server is written in JavaScript.

JavaScript [24, 37] was introduced by Netscape in 1995 as a way to allow web developers to add dynamic functionality to web pages that were executed on the client side. The purposes of the functionality were typically to validate input forms and other user interface related tasks. JavaScript has since then gained momentum, through its ease of deployment and the increasing popularity of certain web applications [94]. We have found that nearly all of the first 100 entries in the Alexa top sites list use some sort of JavaScript functionality.

JavaScript is a dynamically typed, prototype, object-based scripting language with run-time evaluation. The execution of a JavaScript program is done in a JavaScript engine [30, 72, 108], i.e., an interpreter/virtual machine that parses and executes the JavaScript program. Due to the popularity of the language, there have been multiple approaches to increase the performance of the JavaScript

engines, through well-known optimization techniques such as JIT related techniques, fast property access, and efficient garbage collections [27, 30].

The execution of JavaScript code is often invoked in web application through events. Events are JavaScript functionalities that are executed at certain occasions, e.g., when a web application has completed loading all of its elements, when a user clicks on a button, or events that executes JavaScript at certain regular time intervals. The last type of event is often used for so-called AJAX technologies [3]. Such AJAX requests often transmit JavaScript code that later will be executed on the client side, and can be used to automatically update the web applications.

Another interesting property of JavaScript within web applications, is that there is no mechanism like hardware interrupts. This means that the web browser usually "locks" itself while waiting for the JavaScript code to complete its execution, e.g., a large loop-like structure, which may degrade the user experience. Partial solutions exist, e.g., in Chrome where each tab is an own process, and a similar solution exists in WebKit 2.0[1].

3.2.2 Related work

With the increasing popularity of web applications, their execution behavior as well as the performance of JavaScript engines have attended an increased focus, e.g., [75, 5]. Two concurrent studies [84, 89] explicitly compare the JavaScript execution behavior of web applications as compared to existing JavaScript benchmark suites.

The study by Ratanaworabhan et al. [84] is one of the first studies that compares JavaScript benchmarks with real-world web applications. They instrumented the Internet Explorer 8 JavaScript runtime in order to get their measurements. Their measurements are focused on two areas of the JavaScript execution behavior, i.e., (i) functions and code, and (ii) events and handlers. They conclude that existing JavaScript benchmarks are not representative of many real-world web applications and that conclusions from benchmark measurements might be misleading. Important differences include; different code sizes, web applications are often event-driven, no clear hotspot function in the web applications, and that many functions are short-lived in web applications. They also studied memory allocation and object lifetimes in their study.

[1] http://www.techradar.com/news/software/webkit-2-0-announced-taking-leaf-from-chrome-682414

The study by Richards et al. [89] also compares the execution behavior of JavaScript benchmarks with real-world web applications. In their study, they focus on the dynamic behavior and how different dynamic features are used. Examples of dynamic features evaluated are prototype hierarchy, the use of `eval`, program size, object properties, and hot loop. They conclude that the behavior of existing benchmarks differs on several of these issues from the behavior of real web applications.

3.3 Experimental methodology

The experimental methodology is thoroughly described in [56]. We have selected a set of 4 application classes consisting of the first page of the 100 most popular web sites, 109 HTML5 demos from the JS1K competition, 22 use cases from three popular social networks (Facebook, Twitter, and Blogger), and a set of 4 benchmarks for measurements. We have measured and evaluated two aspects: the execution time with and without just-in-time compilation, and the bytecode instruction mix for different application classes. The measurements are made on modified versions of the GTK branch of WebKit (r69918) and Mozilla Firefox with the FireBug profiler.

Web applications are highly dynamic and the JavaScript code might change from time to time. We improve the reproducibility by modifying the test environment to download and re-execute the associated JavaScript locally (if possible). For each test an initial phase is performed 10 times to reduce the chances of execution of external JavaScript code.

Another challenge is the comparison between the social networking web applications and the benchmarks, since the web applications have no clear start and end state. To address this, we defined a set of use cases based on the behavior of friends and colleagues, and from this we created instrumented executions with the Autoit tool.

We modified our test environment in order to enable or disable just-in-time compilation. During the measurements, we executed each test case and application with just-in-time compilation disabled and enabled 10 times each, and selected the best one for comparison. We used the following relative execution time metric to compare the difference between just-in-time-compilation (JIT) and no-just-in-time-compilation (NOJIT):

$$T_{exe}(JIT)/T_{exe}(NOJIT) \geq 1$$

3.4 Application classes

An important issue to address when executing JavaScript applications is to obtain reproducible results, especially since the JavaScript code may change between reloads of the same url address. We have addressed this by downloading the JavaScript code locally, and run the code locally. Further, in most cases we execute the code several times, up to ten times in the just-in-time compilation comparison in Section 3.5.1, and then take the best execution time for each case.

3.4.1 JavaScript benchmarks

There exist a number of established JavaScript benchmark suites, and in this study we use the four most known: Dromaeo [71], V8 [29], Sunspider [107], and JSBenchmark [39]. The applications in these benchmark suites generally fall into two different categories: (i) testing of a specific functionality, e.g., string manipulation or bit operations, and (ii) ports of already existing benchmarks that are used extensively for other programming environments [2].

For instance, among the V8 benchmarks are the benchmarks Raytrace, Richards, Deltablue, and Earley-Boyer. Raytrace is a well-known computational extensive graphical algorithm that is suitable for rendering scenes with reflection. The overall idea is that for each pixel in the resulting image, we cast a ray through a scene and the ray returns the color of that pixel based on which scene objects each ray intersects [105].

Richards simulates an operating system task dispatcher, Deltablue is a constraint solver, and Earley-Boyer is a classic scheme type theorem prover benchmark. However, the Dromaeo benchmarks do test specific features of the JavaScript language and is in this sense more focused on specific JavaScript features.

Typical for the established benchmarks is that they often are problem oriented, meaning that the purpose of the benchmark is to accept a problem input, solve this certain problem, and then end the computation. This eases the measurement and gives the developer full control over the benchmarks, and increases the repeatability.

Table 3.1: A summary of the benchmark suites used in this paper.

Benchmark suite	Applications
Dromaeo [71]	3d-cube, core-eval, object-array, object-regexp, object-string, string-base64
V8 [29]	crypto, deltablue, earley-boyer, raytrace, richards
SunSpider [107]	3d-morph, 3d-raytrace access-binary-trees, access-fannkuch, access-nbody, access-nsieve bitops-3bit-bits-in-byte, bitops-bits-in-byte, bitops-bitwise-and, bitops-nsieve-bits controlflow-recursive crypto-aes, crypto-md5, crypto-sha1 date-format-tofte, date-format-xparb math-cordic, math-partial-sums, math-spectral-norm regexp-dna string-fasta, string-tagcloud, string-unpack-code, string-validate-input
JSBenchmark [39]	Quicksort, Factorials, Conway, Ribosome, MD5, Primes, Genetic Salesman, Arrays, Dates, Exceptions

3.4.2 Web applications - Alexa top 100

The critical issue in this type of study is which web applications that can be considered as representative. Due to the distributed nature of the Internet, knowing which web applications are popular is difficult. Alexa [4] offers software that can be installed in the users' web browser. This software records which web applications are visited and reports this back to a global database. From this database, a list over the most visited web pages can be extracted. In Table 3.2 we present the 100 most visited sites from the Alexa list. In our comparative evaluation, we have used the start page for each of these 100 most visited sites as representatives for popular web applications.

In addition to evaluating the JavaScript performance and execution behavior of the first page on the Alexa top-list, we have created use cases where we measure the JavaScript performance of a set of social networking web applications. These use cases are described in the next section.

Table 3.2: A summary of the 100 most visited sites in the Alexa top-sites list [4] used in this paper (listed alfabetically).

163.com	1e100.net	4shared.com	about.com	adobe.com
amazon.com	ameblo.jp	aol.com	apple.com	ask.com
baidu.com	bbc.co.uk	bing.com	blogger.com	bp.blogspot.com
cnet.com	cnn.com	conduit.com	craigslist.org	dailymotion.com
deviantart.com	digg.com	doubleclick.com	ebay.com	ebay.de
espn.go.com	facebook.com	fc2.com	files.wordpress.com	flickr.com
globo.com	go.com	google.ca	google.cn	google.co.id
google.co.in	google.co.jp	google.co.uk	google.com	google.com.au
google.com.br	google.com.mx	google.com.tr	google.de	google.es
google.fr	google.it	google.pl	google.ru	hi5.com
hotfile.com	imageshack.us	imdb.com	kaixin001.com	linkedin.com
live.co	livedoor.com	livejasmin.com	livejournal.com	mail.ru
mediafire.com	megaupload.com	megavideo.com	microsoft.com	mixi.jp
mozilla.com	msn.com	myspace.com	nytimes.com	odnoklassniki.ru
orkut.co.in	orkut.com	orkut.com.br	photobucket.com	pornhub.com
qq.com	rakuten.co.jp	rapidshare.com	redtube.com	renren.com
sina.com.cn	sohu.com	soso.com	taobao.com	tianya.cn
tube8.com	tudou.com	twitter.com	uol.com.br	vkontakte.ru
wikipedia.org	wordpress.com	xhamster.com	xvideos.com	yahoo.co.jp
yahoo.com	yandex.ru	youku.com	youporn.com	youtube.com

3.4.3 Web applications - Social network use cases

There exists many so-called social networking web applications [109], where Facebook [75] is the most popular one [4, 22]. There are even examples of countries where half of the population use Facebook to some extent during the week [21]. The users of a social networking web application can locate and keep track of friends or people that share the same interests. This set of friends represents each user's private network, and to maintain and expand a user's network, a set of functionalities is defined.

In this paper we study the social networking web applications Facebook [75], Twitter [43], and Blogger [10]. In a sense, Facebook is a general purpose social networking web application, with a wide range of different functionalities. Further, Facebook also seems to have the largest number of users.

Twitter [43] is for writing small messages, so called "tweets", which are restricted to 160 characters (giving a clear association to SMS). The users of Twitter are able to follow other people's tweets, and for instance add comments in form of twitts to their posts.

Blogger is a blogging web applications, that allows user to share their opinion wide range of people through writing. The writing (a so-called blog post) might read, and the person that reads this, can often add an comments to the blog post.

While the benchmarks have a clear purpose, with a clearly defined start and end state, social networking web applications behave more like operating system applications, where the user can perform a selected number of tasks. However, as long as the web application is viewed by the user, it often remains active, and (e.g., Facebook) performs a set of underlying tasks.

To make a characterization and comparison easier, we have defined a set of use cases, with clear start and end states. These use cases are intended to simulate common operations and to provide repeatability of the measurements. The use cases represent common user behavior in Facebook, Twitter, and Blogger. They are based on personal experience, since we have not been able to find any detailed studies of common case usage for social networks. The use cases are designed to mimic user behavior rather than exhausting JavaScript execution.

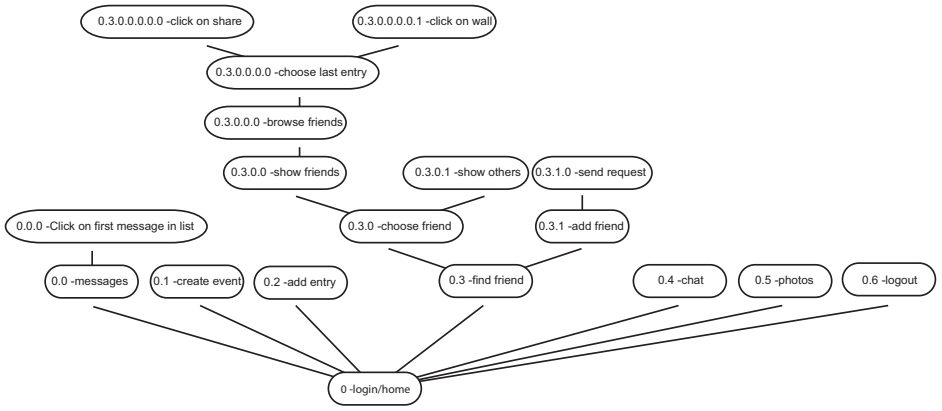

Figure 3.1: Use cases to characterize the JavaScript workload of Facebook.

Figure 3.1, 3.2, and 3.3 give an overview of the different use cases that we have defined for Facebook, Twitter, and Blogger, respectively. Common for all

use cases are that they start with the user login. From here the user has multiple options.

For Facebook, the user first logs in on the system. Then, the user searches for an old friend. When the user finds this old friend, the user marks him as a "friend", an operation where the user needs to ask for confirmation from the friend to make sure that he actually is the same person. This operation is a typical example of an use case, which in turn is composed of several sub use cases: 0 `-login/home`, 0.3 `-find friend`, 0.3.1 `-add friend`, and 0.3.1.0 `-send request`, as shown in Figure 3.1.

All use cases start with the login case, and we recognize an individual operation, such as 0.3.1 `-add friend` as a sub use case, though it must complete previous use cases. Further, we do allow use cases that goes back and forth between use cases. For example in Figure 3.2, if we want to both choose the option 0.1.0 `-follow` and 0.1.1 `-mention`, then we would need to visit the following sub use cases: 0 `-login/home`, 0.1 `-find person`, 0.1.0 `-follow`, 0.1 `-find person`, and 0.1.1 `-mention`.

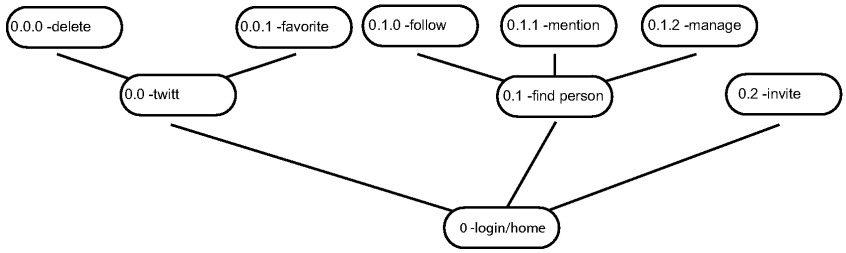

Figure 3.2: Use cases to characterize the JavaScript workload of Twitter.

To enhance repeatability, we use the AutoIt scripting environment [11] to automatically execute the various use cases in a controlled fashion. As a result, we can make sure that we spend the same amount of time on the same or similar operations, such as to type in a password or click on certain buttons. This is suitable for the selected use cases.

3.4.4 HTML5 and the canvas element

There have been several attempts to add more extensive interactive multimedia to web applications. These attempts could be roughly divided into two groups: plug-

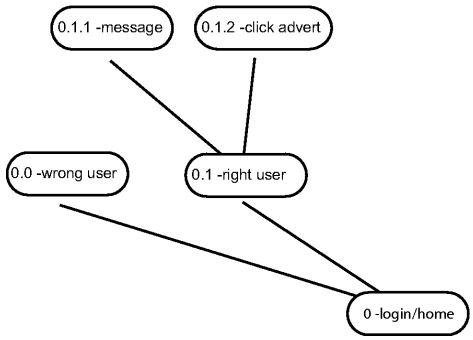

Figure 3.3: Use cases to characterize the JavaScript workload of Blogger.

in technologies and scriptable extension to web browsers. Plug-ins are programs that run on top of the web browser. The Plug-ins can execute some special type of programs, and well known examples are Adobe Flash, Java Applets, Adobe Shockwave, Alambik, Internet C++, and Silverlight. These require that the user downloads and installs a plug-in program before they can execute associated programs. Scriptable extensions introduce features in the web browser that can be manipulated through, e.g., JavaScript.

HTML5 [35] is the next standard version of the HyperText Markup Language. The Canvas in element HTML5 [32] has been agreed on by a large majority of the web browser vendors, such as Mozilla FireFox, Google Chrome, Safari, Opera and Internet Explorer 9. The Canvas element opened up for adding rich interactive multimedia to web application. The canvas element allows the user to add dynamic scriptable rendering of geometric shapes and bitmap images in a low level procedural manner to web applications. A similar technology, albeit at a higher level, is scalable vector graphics [70].

This element opens up for more interactive web applications. As an initiative for programmers to explore and develop the canvas element further, a series of competitions have been arranged [1, 101, 38]. The JS1k competition got 460 entries. The premise for this competition was that the entries should be less than 1024 bytes in total (with an extra bonus if they would fit inside a tweet). Further, it was forbidden to use external elements such as images. The entries vary in functionality and features, which can be illustrated by the top 10 entries, shown in Table 3.3, where half of them are something else than a game.

Table 3.3: The top-10 contributions in the JS1K competition.

	Name	Developer
1	Legend Of The Bouncing Beholder	@marijnjh
2	Tiny chess	Oscar Toledo G.
3	Tetris with sound	@sjoerd_visscher
4	WOLF1K and the rainbow characters	@p01
5	Binary clock (tweetable)	@alexeym
6	Mother fucking lasers	@evilhackerdude
7	Graphical layout engine	Lars Ronnback
8	Crazy multiplayer 2-sided Pong	@feiss
9	Morse code generator	@chrissmoak
10	Pulsing 3d wires	@unconed

3.5 Experimental results

3.5.1 Comparison of the effect of just-in-time compilation

We have compared the execution time where just-in-time compilation (JIT) has been enabled, against the execution time where the JIT compiler has been disabled (NOJIT). When JIT has been disabled the JavaScript is interpreted as bytecode. All modifications are made to the JavaScriptCore engine, and we have used the GTK branch of the WebKit source distribution (r69918). We have divided the execution time of the JIT version with the execution time of the interpretation mode, i.e., $T_{exe}(JIT)/T_{exe}(NOJIT)$. That means, if

$$T_{exe}(JIT)/T_{exe}(NOJIT) \geq 1$$

then the JavaScript program runs slower when just-in-time compilation is enabled. We have measured the execution time that each method call uses in the JavaScriptCore in WebKit.

In Figure 3.4 we have plotted the values of $T_{exe}(JIT)$ / $T_{exe}(NOJIT)$ for a number of use cases for the top 3 social network applications, i.e., Facebook, Twitter, and Blogger, for a set of use cases. The use cases presented in Figure 3.4 are extensions of each other, as discussed in Section 3.4. For instance, case0 is extended into case1, and case1 is then extended into case2. Our results show that the execution time *increases* in 9 out of 12 cases when JIT is enabled. This

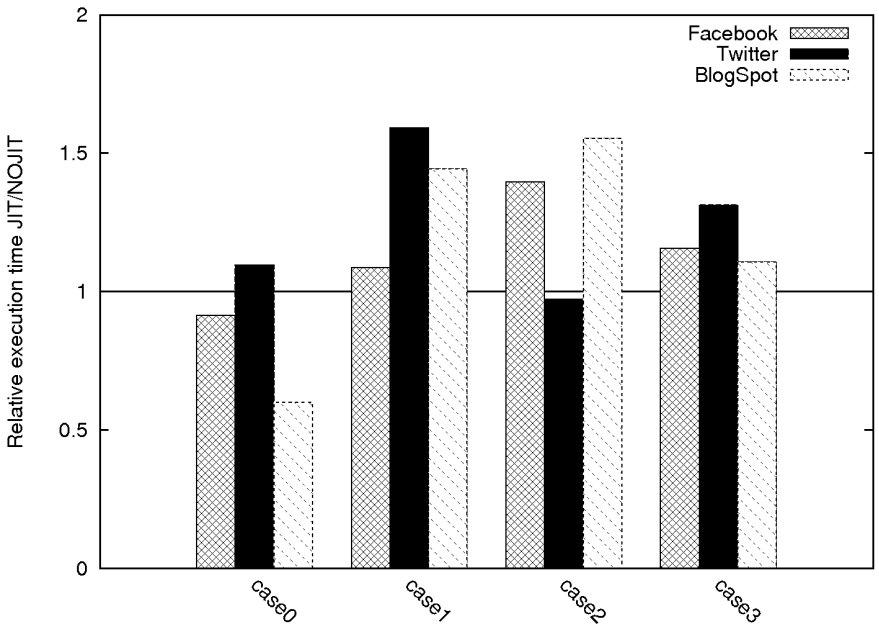

Figure 3.4: Relative execution time $T_{exe}(JIT)$ / $T_{exe}(NOJIT)$ for 4 use cases from three different social network applications.

especially pronounced for the more complicated use cases. The reason is the non-repetitive behavior of the social network application use cases.

In Figure 3.5 we present the relative execution time $T_{exe}(JIT)$ / $T_{exe}(NOJIT)$ for the Alexa top 100 web sites and the first 109 JS1K demos. We have measured the workload of them without any user interaction. The results in Figure 3.5 show that for 58 out of the 100 web applications, JIT *increases* the execution time. However, for those applications that benefit from JIT, their execution times are improved significantly. For instance, the execution time for `craiglist.com` was improved by a factor of 5000. For `yahoo.co.jp` JIT *increased* the execution time by a factor of 3.99.

Further, in Figure 3.5 we see that JIT *increased* the execution time for 59 out of the 109 JS1K demos. When JIT fails, it increases the execution time by a factor of up to 75. When JIT is successful, it decreases the execution time by up to a factor of 263.

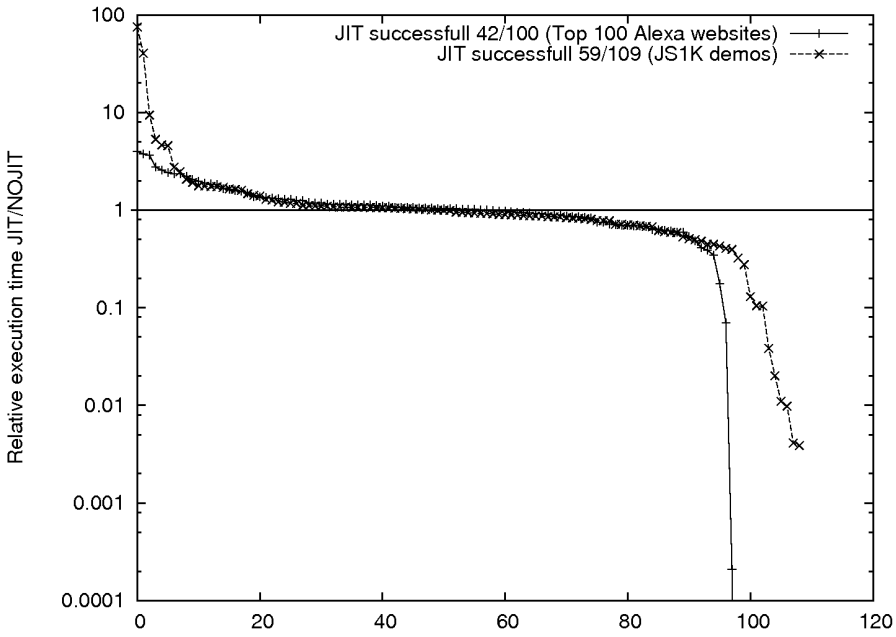

Figure 3.5: Relative execution time $T_{exe}(JIT)$ / $T_{exe}(NOJIT)$ for the first 109 JS1K demos and the top 100 Alexa web sites.

Finally, we have evaluated the effect of JIT on the four benchmark suites, i.e., Dromaeo, V8, Sunspider, and JSBenchmark, as shown in Figures 3.6 and 3.7. In Figure 3.6, we show the results for 4 out of 5 of the V8 benchmarks[2], 6 of the Dromaeo benchmarks, and 10 of the JSBenchmarks. For V8, JIT is successful in 3 out of 4 cases and the best improvement is a factor of 1.9, while in the worst case the execution time is increased by a factor of 1.14. For Dromaeo JIT improves the execution time for 3 out of 6 cases. The largest improvement is by a factor of 1.54, while largest increase in execution time is by a factor of 1.32. For the JSBenchmarks, JIT decreases the execution time for 7 out of 10 cases. The largest decrease in execution time is by a factor of 1.6. The largest increase in the execution time is by a factor of 1.07.

Finally, Figure 3.7 shows the results for the SunSpider benchmark. All the applications in the SunSpider benchmark suite run equally fast or faster

[2]Earley-boyer, did not execute correctly with the selected version of WebKit

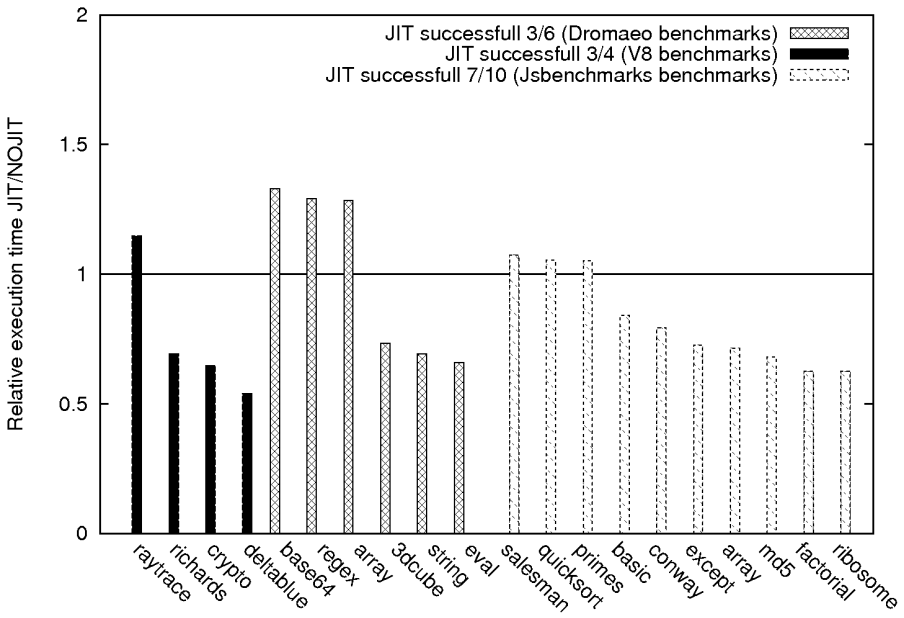

Figure 3.6: Relative execution time $T_{exe}(JIT)$ / $T_{exe}(NOJIT)$ for the V8, Dromaeo, and JSBenchmark benchmarks.

when JIT is enabled. The largest improvement is by a factor of 16.4. for the `string-validate-input` application, and the smallest improvement is 1.0, i.e., none, for the `date-format-tofte` application.

In summary, JIT decreases the execution time for most of the benchmarks. In contrast, JIT *increases* the execution time for more than half of the studied web applications. In the worst case, the execution time was prolonged by a factor of 75 (`id81` in the JS1K demos).

3.5.2 Comparison of bytecode instruction usage

We have measured the bytecode instruction mix, i.e., the number of executed bytecode instructions for each bytecode instruction, for the selected benchmarks and for the first 100 entries in the Alexa top list. Then, a comparison between the

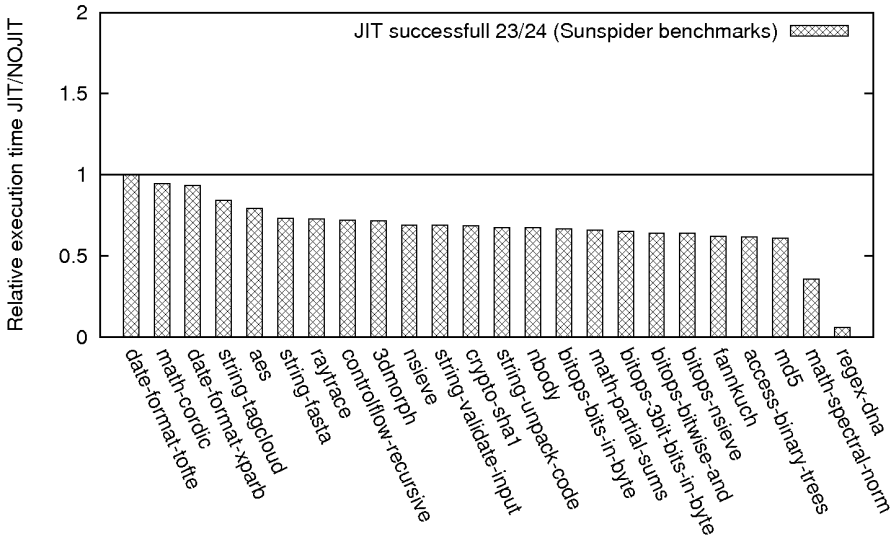

Figure 3.7: Relative execution time $T_{exe}(JIT)$ / $T_{exe}(NOJIT)$ for the Sunspider benchmarks.

web applications and the SunSpider benchmarks is done, since these two differ the most.

The SunSpider benchmarks use a smaller subset of bytecode instructions than the Alexa web sites do. The Alexa web sites use 118 out of 139 bytecode instructions, while the SunSpider benchmarks only use 82 out of the 139 instructions. We have grouped the instructions based on instructions that have similar behaviors. The instruction groups are: prototype and object manipulation, branches and jumps, and arithmetic/logical.

In Figure 3.8 we see that arithmetic/logical instructions are more intensively used in the SunSpider benchmarks than in the web applications covered by Alexa top 100. We also observe that the SunSpider benchmarks often use bit operations (such as left and right shift) which are rarely used in the web sites. This observation suggests that even though these operations are important in low level programming languages, it seems like these are rarely used in web applications. The only arithmetic/logical operation that is more used in web applications is the `not` instruction, which could be used in, e.g., comparisons.

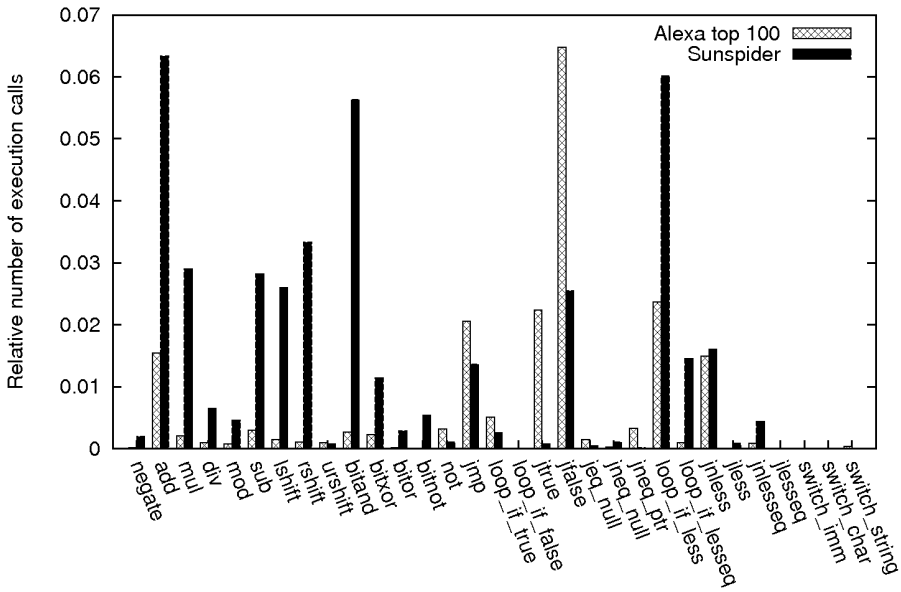

Figure 3.8: Branch, jump, and arithmetic/logical related bytecode instructions for the Alexa top 100 web sites and the SunSpider benchmarks.

For the branch and jump bytecode instruction group, we observe in Figure 3.8 that jumps related to objects are common in Alexa, while jumps that are associated with conditional statements, such as loops are much more used in the benchmarks. A large number of `jmp` instructions also illustrates the importance of function calls in web applications.

We notice that Alexa top 100 web applications use the object model of JavaScript, and therefore use the object special features more than the benchmarks. In Figure 3.9 we see that instructions such as `get_by_id`, `get_by_id_self`, and `get_by_id_proto` are used more in the web applications than in the benchmarks. Features such as classless prototyped programming are rarely found in traditional programming languages which the benchmarks are ported from. A closer inspections of the source code of the benchmarks confirms this. It seems like many of the benchmarks are embedded into typical object-based constructions, which assist in measuring execution time and other benchmarks related tasks. However, these object-based constructions are rarely a part of the compute intensive parts of the benchmark.

The observation above is further supported in Figure 3.9, by looking at instructions such as `get_val` and `put_val`, which the SunSpider benchmarks use more extensively than the web applications. This suggests that the benchmarks do not take advantage of JavaScript classless prototype features, and instead try to simulate the data structures found in the original benchmarks.

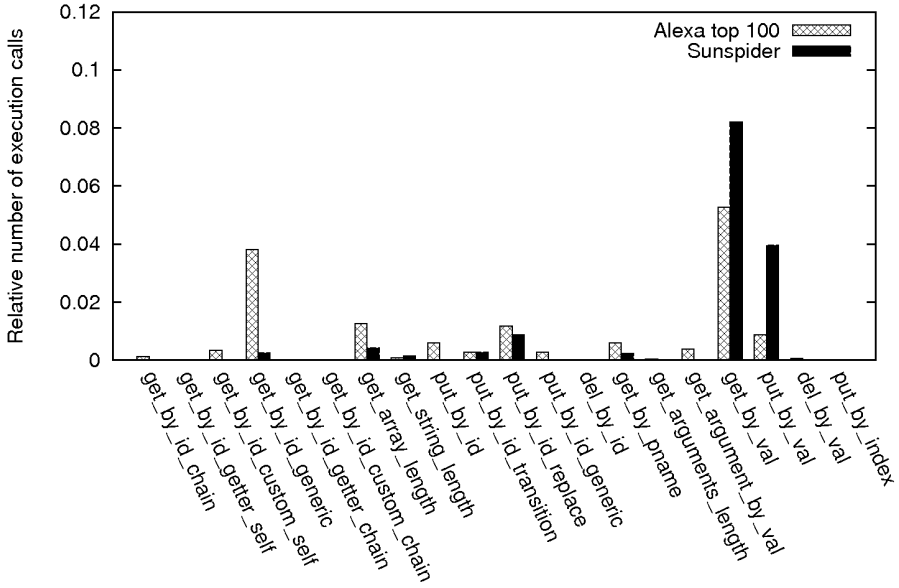

Figure 3.9: Prototype and object related instructions for the Alexa top 100 web sites and the SunSpider benchmarks.

3.5.3 Usage of the `eval` function

One JavaScript feature is the evaluate function, `eval`, that evaluates and executes a given string of JavaScript source code at runtime. To extract information on how frequently `eval` calls are executed, we have used the FireBug [23] JavaScript profiler to extract this information. We have measured the number of `eval` calls relative to the total number of function calls, i.e., *No. of* `eval` *calls / Total no. of function calls.*

Figure 3.10 presents the relative number of `eval` calls. Our results show that `eval` functions are rarely being used in the benchmarks, only 4 out of 35 benchmarks use the `eval` function. However, these four use `eval` quite extensively. The `dromaeo-core-eval` benchmark has 0.27, `sunspider-date-format-tofte` has 0.54, `sunspider-date-format-xparb` has 0.28, and `sunspider-string-tagcloud` has 0.15 relative number of `eval` calls. From their name, e.g., `eval-test` in the Dromaeo benchmark, and by inspection of the JavaScript code and the amount of `eval` calls, we suspect that these benchmarks were designed specifically to test the `eval` function.

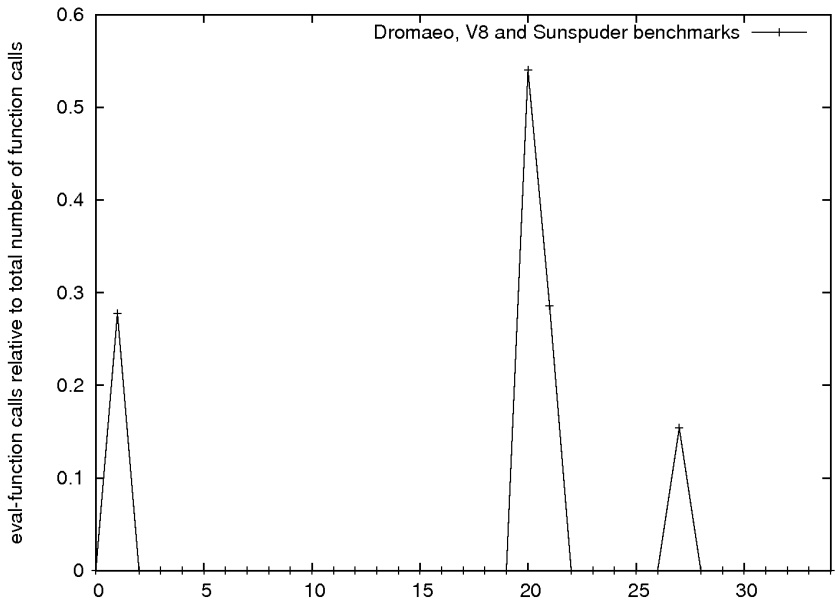

Figure 3.10: Number of `eval` calls relative to the total number of function calls in the Dromaeo, V8, and SunSpider benchmarks.

We observe in Figure 3.11 that the `eval` function is used more frequently in the Alexa top 100 web sites. 44 out of 100 web sites use the `eval` function. In average, the relative number of `eval` calls is 0.11. However, there are web sites with a large relative number of `eval` calls, e.g., in `sina.com.cn` 55% of all function calls are `eval` calls.

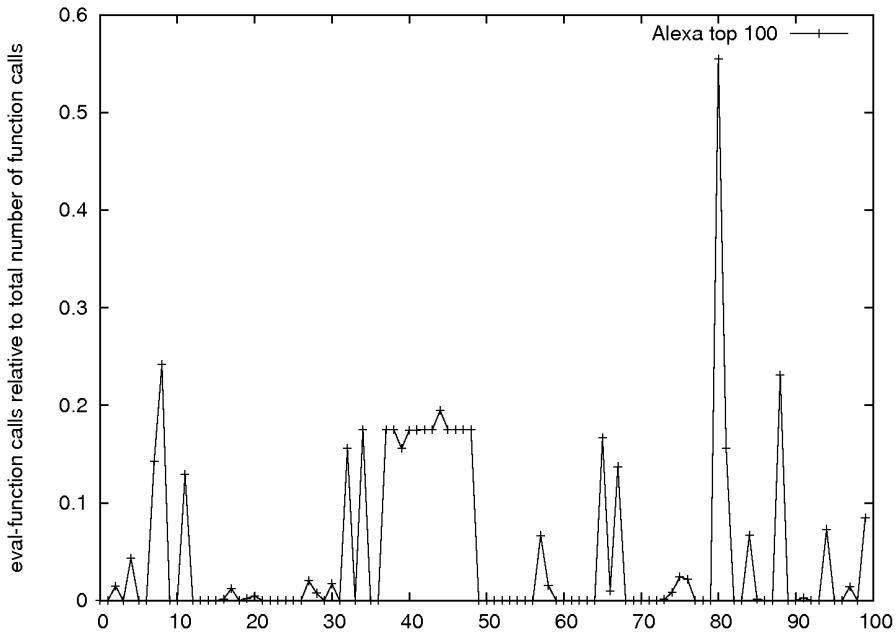

Figure 3.11: Number of `eval` calls relative to the number of total function calls for the first 100 entries in the Alexa list.

3.5.4 Anonymous function calls

An anonymous function call is a call to a function that does not have a name. In many programming languages this is not possible, but it is possible to create such functions in JavaScript. Since this programming construct is allowed in JavaScript, we would like to find out how common it is in JavaScript benchmarks and web applications The relative number of anonymous function calls in the benchmarks and the Alexa top 100 sites are shown in Figure 3.12.

We found that 3 of the anonymous function calls in the benchmarks were instrumentations of the benchmark to measure execution time. If we removed these 3 function calls we found that 17 out of the 35 benchmark used anonymous function calls to some degree. For the entries in the top 100 Alexa web sites, we found that 74 out of 100 sites used anonymous function calls. Some benchmarks use anonymous function calls extensively. However, these seems to be specifically

tailored for anonymous function calls, much like certain benchmarks were tailored to test `eval` in Section 3.5.3.

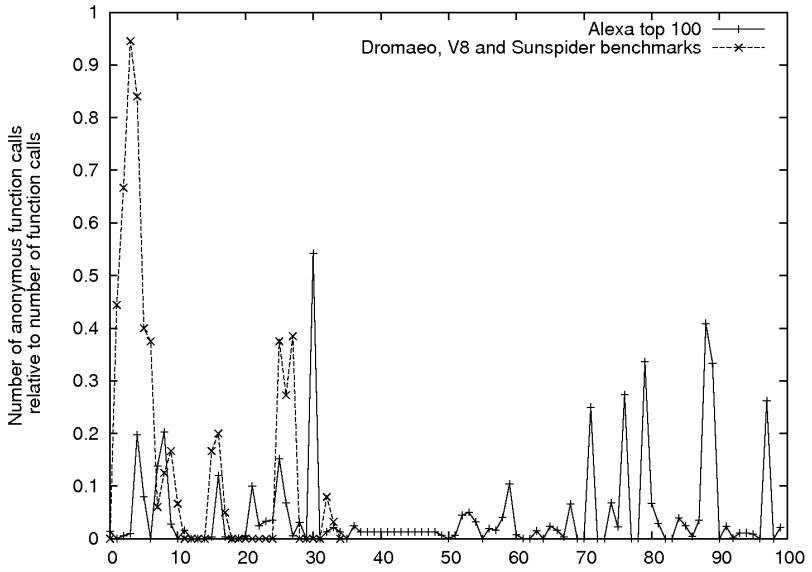

Figure 3.12: Relative number of anonymous function calls in the Alexa top 100 web sites and the benchmarks.

3.6 Conclusions

In this study, we have evaluated and compared the execution behavior of JavaScript for four different application classes, i.e., four JavaScript benchmark suites, popular web sites, use cases from social networking applications, and the emerging HTML5 standard. The measurements have been performed in the WebKit browser and JavaScript execution environment.

Our results show that benchmarks and real-world web applications differ in several significant ways:

- Just-in-time compilation is beneficial for most of the benchmarks, but actually *increases* the execution time for more than half of the web applications.

- Arithmetic/logical bytecode instructions are significantly more common in benchmarks, while prototype related instructions and branches are more common in real-world web applications.
- The `eval` function is much more commonly used in web applications than in benchmark applications.
- Approximately half of the benchmarks use anonymous functions, while approximately 75% of the web applications use anonymous functions.

Based on the findings above, in combination with findings in previous studies [84, 89], we conclude that the existing benchmark suites do not reflect the execution behavior of real-world web applications. For example, special JavaScript features such as dynamic types, `eval` functions, anonymous functions, and event-based programming, are omitted from the computational parts of the benchmarks, while these features are used extensively in web applications. A more serious implication is that optimization techniques employed in JavaScript engines today might be geared towards workloads that only exist in benchmarks.

Chapter 4

Paper III

Thread Level Speculation as an Optimization Technique in Web Applications for Embedded Mobile Devices - Initial Results

Jan Kasper Martinsen, Håkan Grahn and Anders Isberg

Proceedings of the Sixth IEEE International Symposium on Industrial Embedded Systems (SIES'11), pages 83–86, June 2011, Västerås, Sweden

4.1 Introduction

Current and future processor generations are based on multicore architectures, and it has been suggested that performance increase will mainly come from an increasing number of processor cores. In order to achieve an efficient utilization of an increasing number of processor cores, the software needs to be parallel as well as scalable [7, 65, 100]. Due to the simplicity of distribution, along with increased platform independence, many applications are moved to the World Wide Web, as so called Web Applications, and new programming languages, e.g., JavaScript, have emerged. JavaScript is a dynamically typed, object-based scripting language with run-time evaluation, where execution is done in a JavaScript engine.

To preserve platform independence and simplicity, there are currently no support for threading. With the increased popularity of Web Applications and a higher demand for performance, several classical hardware optimisation techniques have been suggested along with a set of benchmarks to measure their effect. We have argued that these benchmarks are unrepresentative, and that current optimisation techniques often degrades the performance for Web Applications [56]. Therefore we have called for different optimisation techniques, and suggest that multicore architectures should play a crucial part.

Developing parallel applications are difficult, time consuming and error-prone and therefore we would like to ease the burden of the programmer. To hide some of the details, an approach is to dynamically extract parallelism from a sequential program using Thread-Level Speculation (TLS) techniques [91].The performance potential of TLS has been shown for applications with static loops, statically typed languages, and in byte code environments.

Previously we have evaluated the performance of TLS together with the Rhino JavaScript engine and evaluated it's performance with the V8 benchmark [55]. We are extending this study to the SquirrelFish JavaScript engine found in WebKit, and also perform experiments on Web Applications rather than the benchmarks that we found to be unrepresentative.

The rest of the paper is organized as follows; Section 4.2 provides some background on TLS, JavaScript and Unrepresentative benchmarks. In Section 4.3, we discuss our current research, in Section 4.4 we make an outline for future work and finally in Section 4.5 a conclusion.

4.2 Background

In Section 4.2.1 we present the general principles of thread-level speculation and some previous implementation proposals. In Section 4.2.2 we discuss the JavaScript language and finally in Section 4.2.3 a brief summary of the unrepresentativness of current JavaScript benchmarks.

4.2.1 Thread-Level Speculation

Thread-Level Speculation Principles

Thread-level speculation (TLS) aims to dynamically extract parallelism from a sequential program. One popular approach is to allocate each loop iteration to a thread. Then, we can(ideally) execute as many iterations in parallel as we have processors.

However data dependencies may limit the number of iterations that can be executed in parallel. Further, the requirements and overhead for detecting data dependencies can be considerable.

Between two consecutive loop iterations 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 dependecies during run-time using dynamic information about read and write addresses from each loop iteration. A key design parameter is the *precision* of what granularity the TLS system can detect data dependency violations.

When a data dependency violation is detected, the execution must be aborted and rolled back to safe point in the execution. Thus, all TLS systems need a roll-back mechanism. 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 roll-backs 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 violation that is detected when no actual dependence violation is present.

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. Updating data in-place usually result in higher performance if the number of roll-backs is low, but lower performane when the number of roll-backs is high since the cost of doing roll-backs is high.

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

Bruening et al. [13] 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.The results show speed-ups of up to almost five on 8 processors.

Rundberg and Stenström [91] proposed a TLS implementation that resembles the behavior of a hardware-based TLS system. They show a speedup of up to ten times on 16 processors for three applications.

Kazi and Lilja developed the course-grained thread pipelining model [41] for exploiting coarse-grained parallelism. They suggest to pipeline the concurrect execution of loop iterations speculatively, using run-time dependence checking. On an 8-processor machine they achieved speed-ups of between 5 and 7. Bhowmik and Franklin [9] 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 yielding speed-ups between 1.64 and 5.77 on 6 processors.

Cintra and Llanos[17] present a software-based TLS system that speculatively execute loop iterations in parallel within a sliding window. By using optimized data structures, scheduling mechanisms, and synchronization policies they manage to reach in average 71% of the performance of hand-parallelized code for six applications

Chen and Olukotun present two studies on [15, 16] how method-level parallelism can be exploited using speculative techniques. Their techniques are implemented in the Java runtime parallelizing machine (Jrpm). On four processors, their results show speed-ups of $3-4$, $2-3$, and $1.5-2.5$ for floating point applications, multimedia applications, and integer applications, respectively.

Picket and Verbrugge [80, 81] 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 byte code level. They obtain at most a two-fold speed-up on a 4-way multi-core processor.

Oancea et al. [78] 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 slighty 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. The results show that their TLS approach reaches in average 77% of the speed-up of hand-parallelized, non-speculative versions of the programs.

A study by Prabhu and Olukotun [82] analyzed what types of thread-level parallelism that can be exploited in the SPEC CPU2000 Benchmarks[96]. By going through each of the applications, they identified a number of useful transformations, e.g., speculative pipelining, loop chunking/slicing, and complex value prediction. They also identified a number of obstacles that hinder or limit the usefulness of TLS parallelization.

All studies presented above have worked with applications written in C, Fortran, or Java. The Java studies have been done at the bytecode level.

In [53] and [55] we have used the TLS technique for a dynamic language such as JavaScript. As seen in Figure 4.1 there is a modest speedup for a few instances

4.2.2 JavaScript

JavaScript [37] is a dynamically typed, object-based scripting language with runtime evaluation often used in assosiation with Web Applications. JavaScript application execution is done in a JavaScript engine, i.e., an interpretator/virtual machine that parses and executes the JavaScript program.

The performance of these script engines have increased significantly during the last years, reaching very high single-thread performance. However, today no official JavaScript engine supports parallel execution of threads. Although this could change in the future, it is still the programmer who is responsible for finding and expressing the parallelism.

4.2.3 Unrepresentative benchmarks

We found that JavaScript benchmarks were ported from existing benchmark suites, and that thanks to the flexibility of the JavaScript porgramming language, one could port without utilizing certain JavaScript features (for instance such as

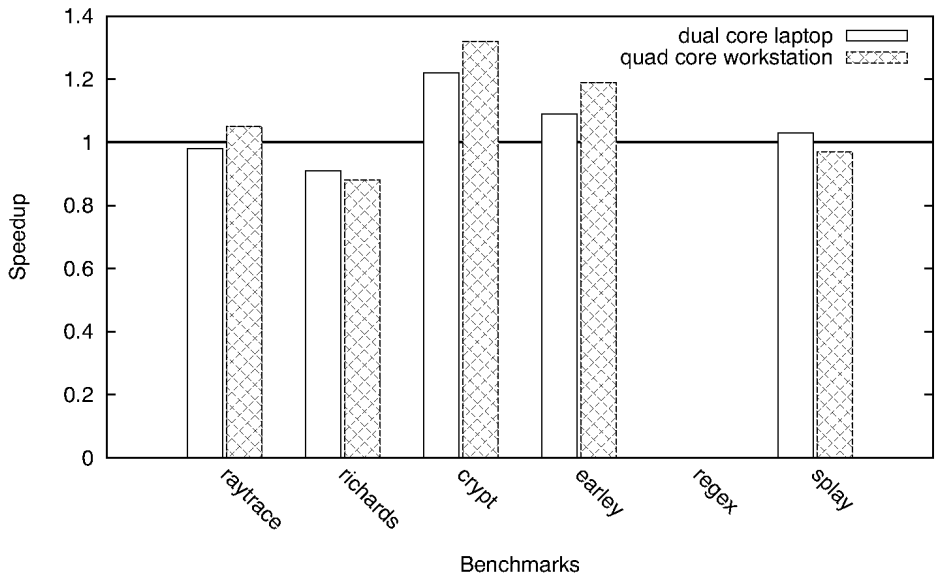

Figure 4.1: Relative execution time with TLS enabled, normalized to the execution time withou TLS enabled.

anonymous and eval function call) [54, 56]. However we measured the JavaScript workload for a large number of popular Web Applications, with in-depth measurements for so-called social networks, and we found that these JavaScript features play an important role in real-life Web Applications.

We also found that features, which are optimized for quite heavily in the benchmarks, such as large loops, a large amount of artimetic instructions (Figure 4.2) were to a large extent absent in Web Applications.

We found the reason to be, that there is no interrupt mechanism in JavaScript, so that large loops make the Web Application unresponsive. Large loop like structures are instead confined into anonymous functions calls made by events.

This observations suggests that attention should be given to the features in JavaScript that come along with being a dynamic programming language. We have also found evidence that new multimedial functionalities will be even more dependent on JavaScript and these features.

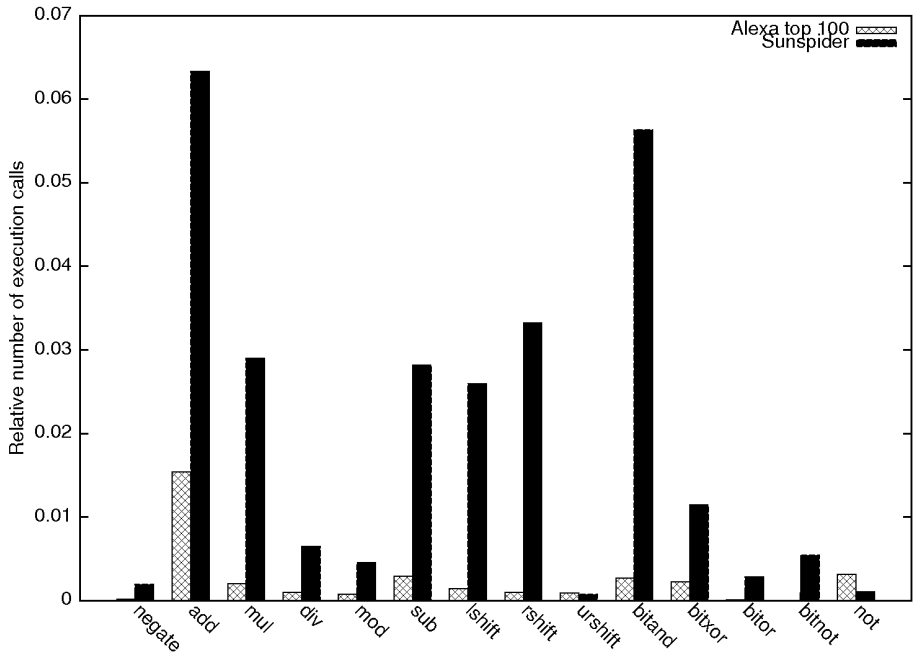

Figure 4.2: Number of aritmetic instructions in the bytecode produced by Squirrelfish for Sunspider benchmark and for a set of Web Applications

4.3 Current work

We have acknowledged the difference between real world web applications and the established benchmarks. These studies suggested that only focusing on the JavaScript interpreter in an environment without the web browser could lead to false results. We are currently working to incorporate these techniques into WebKit's JavaScript interpreter SquirrelFish. SquirrelFish is an register based interpreter.

Being a register based interpreter suggests some more book-making challenges when it comes to recording changes before speculation. The register could be serving as a temporary placement of a variable.

However, the register also seems to decrease the complexity of duplicating values before speculation (on earlier experiments, with stack based interpreter we were forced to duplicate a large amount of the stack before speculation).

We have also found that for anonymous and evaluate function calls the number of conflicts is quite low, if we compare to normal JavaScript functions calls. It seems that these functions are less prone to have conflicts with global variables, and therefore could possibly be better candidate for speculations. In addition, in the initial discussion of TLS, we suggested that for loops, the ideal were that one would be able to add one iteration per thread. As we mentioned in the previous section, due to the lack of an interrupt mechanism we were forced to use events to simulate large loops, and that anonymous functions typically were associated with events. From this, we suggest that anonymous function, as an function that for certain Web Applications is called extensively. Global variables access might be quite rare from such a function, and therefore it will be a good candidate for speculation.

4.4 Future work

We believe that TLS is a promising route for optimisation of Web Applications. With the increasing amount of dynamic multimedia in Web Application, JavaScript's workload might increase significantly in the near future. Similar applications in a desktop environment have large loops, and as we suggested that loops, events and anonymous functions will play a key role for Web Applications.

We have shown that traditional hardware centric optimisation techniques (such as just in time compilation) have limited effect on real-life Web Applications. This would be true for both ARM and Intel architectures. However, we have not found any studies that decides which one of is the better when such an optimisation is successful.

Another interesting proposition is the following; Web Applications usually reside on the Internet, we could have some sort of mechanism to record successful and not so successful speculations attempts. This information could later be used for future speculations.

4.5 Conclusion

TLS speculation in Web Application is a promising technique to increase performance. One of the more interesting parts is the increase of multimedia workloads, which could prove this technique even more promising. We will give this workload, along with a different platform much attention in our future work.

Chapter 5

Paper IV

Enhancing JavaScript Performance in Web Applications Using Speculation

Jan Kasper Martinsen, Håkan Grahn, and Anders Isberg

IEEE Internet Computing - special issue on "Virtualization", 17(2): 10–19, March/April 2013

5.1 Introduction

JavaScript [37] 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 [30, 108, 72]. 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 [84, 89, 56]. 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 [58].

JavaScript is a sequential programming language and cannot take advantage of multicore processors. Fortuna et al. [25] 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 [104] 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 [55] proposed to use TLS in the Rhino JavaScript engine.

In this paper, we present an implementation of TLS in the Squirrelfish [108] 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.

5.2 JavaScript and web applications

JavaScript [37] is a dynamically typed, object-based scripting language with runtime 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 [30], WebKit's Squirrelfish [108], and Mozilla's SpiderMonkey and TraceMonkey [72] have high single-threaded performance for a set of benchmarks. However, such performance results can be misleading [84, 89, 56] for web applications, and optimizing towards the characteristics of these benchmarks may increase the execution time for real-life web applications [58].

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 [84, 89, 56], 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.

5.3 Thread-Level Speculation Implementation for JavaScript

5.3.1 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 5.1). 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.

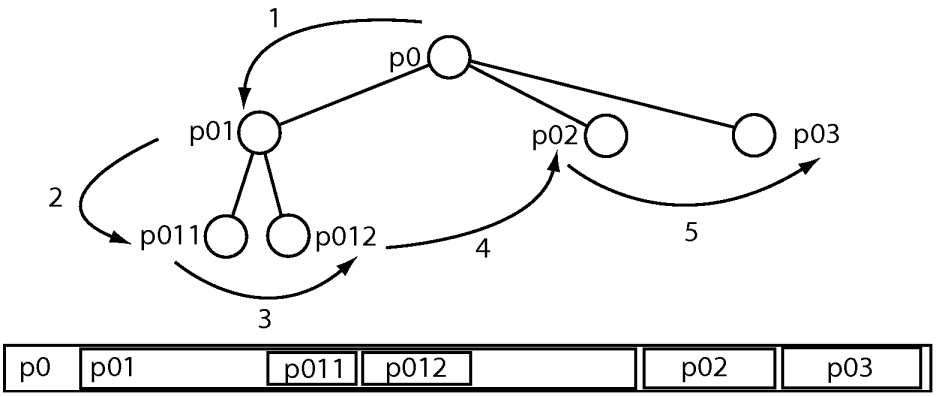

Figure 5.1: The JavaScript program P0, performs 3 function calls, P01, P02 and P03. P01 performs two function calls, P011 and P012. Thus, we have created a speculation tree from the function calls. If we traverse this tree from left to right, we get an order in which the functions are called, equal to 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 $function\_order$.

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 "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 5.2, 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.

5.3.2 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 previous 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 $function\_order$.

5.3.3 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.

5.3.4 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.

5.4 Experimental Methodology

We have selected 15 web applications from the Alexa list [4] 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 5.1.

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 [11] to automatically execute the various use-cases in a controlled fashion. The methodology for the experiments is described in [56].

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.

5.5 Experimental Results

5.5.1 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

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 5.1: 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 [4].

sequential execution time on the unmodified interpreted Squirrelfish engine, as in Equation 5.1.

$$\frac{T_{exe}(sequential\ execution\ time)}{T_{exe}(with\ TLS\ or\ with\ JIT\ or\ with\ Google\ V8)} \tag{5.1}$$

In Figure 5.3 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 [59])
- *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 5.2). We see that a large number of

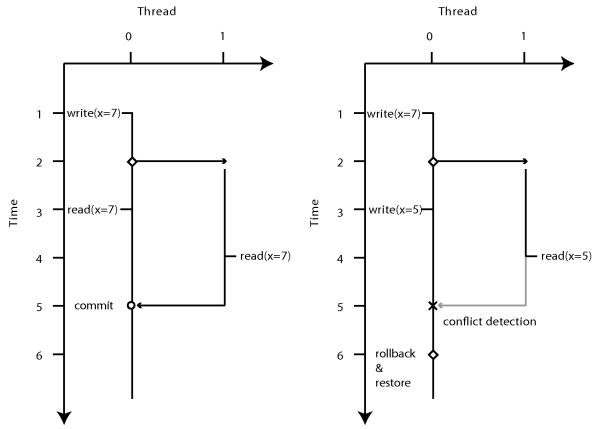

Figure 5.2: We have illustrated an example of a successful speculation (a) and an usuccessful one that force us to do a rollback to preserve sequential semantics (b). In (a) at time 1 in thread 0 we write the value 7 to the global value x. At time 2 in thread 0 we encounter a function which we speculate 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 5.3.2). 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.

speculations and a low number of rollbacks are typical for all 15 web applications. However, the improved execution time varies between the web applications.

5.5.2 General execution behavior

In Table 5.2 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

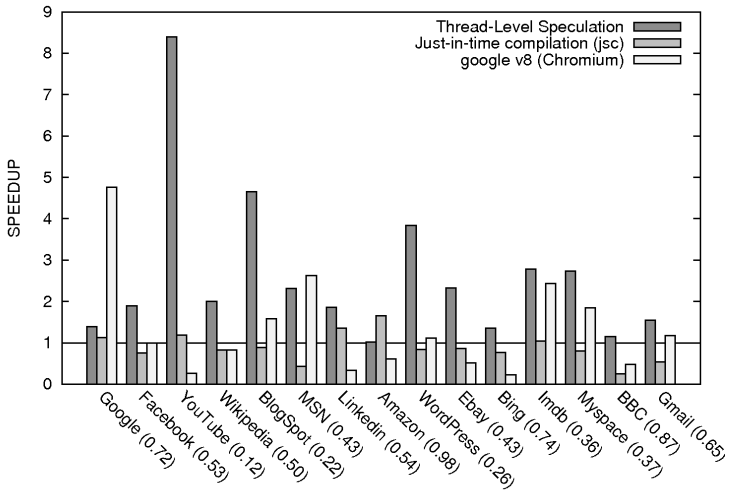

Figure 5.3: JavaScript execution speedup over the interpreted sequential JavaScript engine for the 15 web applications when TLS is enabled, when just-in-time compilation is enabled and for the Google v8 JIT engine in the chromium web browser. We have written the execution time of TLS relative to the sequential execution time in parenthesis after the name of the web applications.

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 [25]. 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 [84, 89, 56] 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.

5.6 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 [108]. We speculate at function level and support nested speculation, i.e., a function that is executing speculatively can create

App.	No. spec.	No. rollbacks	Spec. / rollbacks	Max. no. threads	Max. spec. depth	Avg. depth	Mem. 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 5.2: 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).

new speculatively executed functions. Our evaluation is based on 15 popular web applications from the Alexa top list [4].

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.

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

Sidebar: Thread-Level Speculation Principles

TLS aims to dynamically extract parallelism from a sequential program. This can be done both in hardware, e.g., [14, 85, 98], 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.

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. [13] 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 [91] 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 [41] exploiting coarse-grained parallelism. They suggest to pipeline the concurrent execution of loop iterations speculatively, using run-time dependence checking.

Bhowmik and Franklin [9] 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 [16] 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 [80] 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. [78] 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 [68], 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 [67] as well as a lightweight speculation mechanism for dynamic parallelization of JavaScript applications [66]. 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).

Chapter 6

Paper V

Heuristics for Thread-Level Speculation in Web Applications

Jan Kasper Martinsen, Håkan Grahn, and Anders Isberg

IEEE Computer Architecture Letters, (published on-line 20 November 2013)

6.1 Introduction

JavaScript is a dynamically typed, object-based scripting language with run-time evaluation, where execution is done in a JavaScript engine [30, 108, 72]. Several optimization techniques have been suggested to decrease the execution time of JavaScript, e.g., just-in-time compilation (JIT) [30, 108, 72]. Thread-Level Speculation (TLS) [91] has been proposed as an approach to take advantage of parallel hardware, both for JavaScript benchmarks [66] and web applications [68, 63].

Previous work have shown that there are significant differences in the execution behavior of JavaScript benchmarks and real-world web applications, e.g. [56, 84, 89]. One of the consequences is that JIT often increases the execution times in web applications, while TLS decreases the execution times [56, 63]. A study by Fortuna et al. [25] shows potential JavaScript speedups of up to 45 times with parallelism in web applications.

Mehrara and Mahlke [67] address how to utilize multicore systems in JavaScript engines. They target trace-based JIT-compiled JavaScript code, where the most common execution flow is compiled into an execution trace. Then, runtime checks (guards) are inserted to check whether control flow is still valid for the trace or not. They execute the runtime checks (guards) in parallel with the main execution flow (trace), and only have a single main execution flow.

In [66], a lightweight speculation mechanism is proposed that focuses on loop-like constructs in JavaScript. A loop is marked for speculation if it contains a sufficient workload. As this code used the trace feature of Spidermonkey, a selective form of speculation is employed. They found that they were able to make execution 2.8 times faster for well known JavaScript benchmarks. Unfortunately, large loop structures are rare in real web applications [56].

Mickens et al. [68] indicate an event-based speculation mechanism which is deployed as a JavaScript library called Crom. Crom clones certain regions of the JavaScript code, which then are executed speculatively. Unlike our approach, their main goal is to enhance the responsiveness, while our main goal is to reduce the JavaScript execution time by dynamically extracting parallelism.

In [63] we presented a method-level TLS implementation in Squirrelfish/WebKit with an on/off speculation principle, where a single misspeculation turns off speculation for that function.

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

6.2 TLS Implementation for JavaScript

We have implemented method-level TLS in the Squirrelfish JavaScript engine [108]. The speculation is done on JavaScript functions, including return value prediction, all data conflicts are detected and rollbacks are done when conflicts arise, and nested speculation is supported. The implementation and the performance of our TLS system are described in [63]. Below is a brief overview of our baseline TLS implementation.

The JavaScript execution in Squirrelfish is divided into two stages; first the JavaScript code is compiled into bytecode instructions, and then the bytecode

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

During execution we might encounter the bytecode instruction that indicates the start of a function call. We extract the *realtime* value and the id of the thread that makes this call, e.g., *p_0220* (*p_0* calls a function after 220 bytecode instructions). We denote the value of the position of this function call as *function_order*, which emulates the *sequential time* in our TLS program (Fig. 6.1). We check if this function previously has been speculated by looking up the value of *previous*[*function_order*]. *previous* is a vector where each entry is organized by the *function_order*.

If the entry is 1, then the function has been speculated unsuccessfully. If the value is 0, then it has not been speculated or has been successfully speculated, and we call this position a fork point.

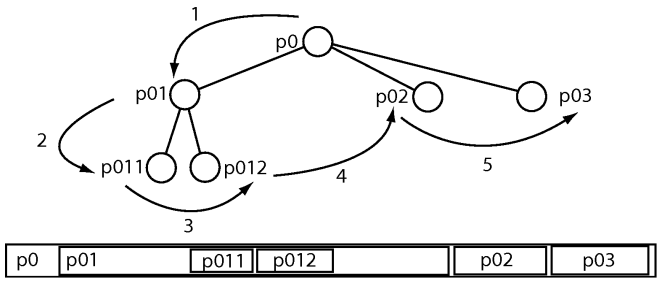

Figure 6.1: The JavaScript program P0 performs 3 function calls, P01, P02, and P03, and P01 performs two function calls, P011 and P012. Thus, we have created a speculation tree. If we traverse this tree from left to right, we get the same function call order as in a sequential execution, i.e., P0, P01, P011, P012, P02, and finally P03. We denote this order as *function_order*.

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

We then create a new thread with a unique id. We copy the value of *realtime* from its parent and modify the instruction pointer of the parent thread such it points to the "end of function call" bytecode instead of the position of the "function call" bytecode instruction. In other words, the parent thread skips the function call and continues to execute speculatively after the function call.

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

6.3 Heuristics

In [63], we demonstrated large execution time improvements using TLS for web applications. However, when a misspeculation occurred we never re-speculated on that function again (baseline).

In Fig. 6.2 we present three speculation heuristics, that then are evaluated on a set of web applications (Table 6.1); a 2-bit heuristic, an exponential heuristic, and a combination of these two heuristics.

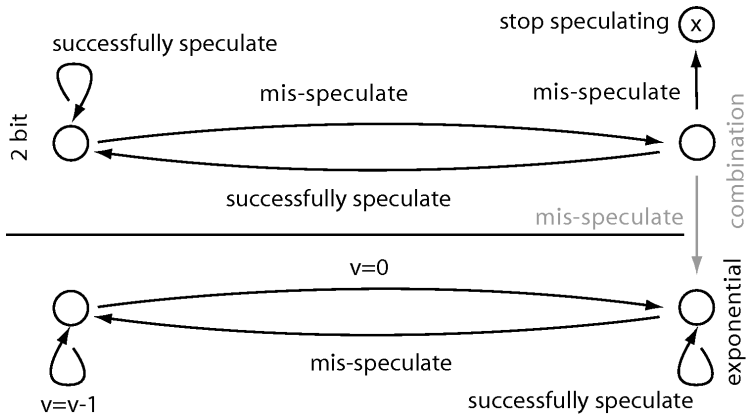

Figure 6.2: The three heuristics; 2-bit, exponential, and the combination of the two. In the 2-bit heuristic, speculation on a function stops after two misspeculations in a row. In the exponential, the time between new speculation attempts doubles for each misspeculation on a function. When we combine both, we initially use the 2-bit heuristic, but after two misspeculations we switch to use the exponential heuristic.

6.3.1 2-bit heuristic

In the 2-bit heuristic (Fig. 6.2), when speculation creates rollbacks 2 times in a row on the same function, we never speculate on that function again. This could either lead to successfully speculate on a certain function, or an increased number of misspeculations.

6.3.2 Exponential heuristic

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

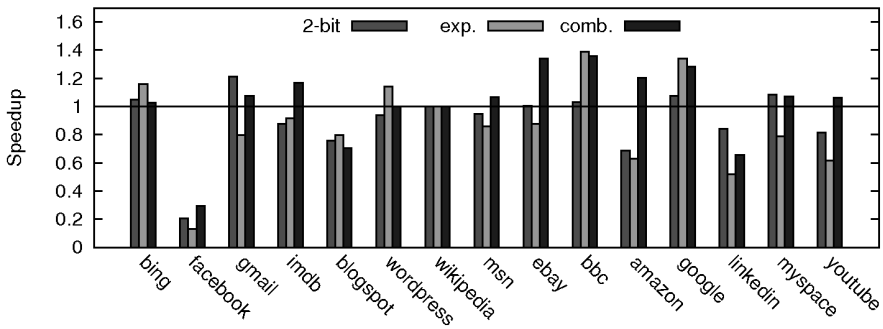

Figure 6.3: The speedup of the three heuristics, relative to the the baseline

6.3.3 Combined 2-bit and exponential heuristic

Instead of stop speculating after two failures in a row, like we do in the 2-bit heuristic, we continue with the exponential heuristic. This means that when we speculate enough and have enough misspeculations, it would essentially become the 2-bit heuristic. However, unlike the 2-bit heuristic, this allows us to re-speculate again after a number of subsequent speculation attempts.

Table 6.1: The web applications used in the experiments.

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

6.4 Experimental Methodology

We have selected 15 web applications from the Alexa list of most visited web sites, as shown in Table 6.1. To enhance reproducibility, we use the AutoIt scripting environment to automatically execute the various use-cases in a controlled fashion. To validate the correctness of our TLS implementation we have compared the committed bytecode instructions, the return values, and all written data values in our TLS implementation with the sequential execution trace. The full methodology is described in [56].

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

6.5 Experimental Results

Fig. 6.3 shows the speedup of the heuristics normalized to the baseline's speedup. It is up to 36% faster than the baseline and the combined heuristic is faster

than the baseline for 12 out of 15 cases. Comparing the speedup for the three heuristics, we observe that the combined heuristic is faster than the 2-bit and the exponential heuristic for 8 of the use cases. Fig. 6.6 shows the relative number of rollbacks and speculations. We see that the relation between rollbacks and speculations is much lower for the combined heuristics, than for the baseline, the 2-bit and the exponential heuristic.

For *LinkedIn*, *Amazon*, and *Facebook* the heuristics result in longer execution times than the baseline TLS. For *Facebook* we find that 43% of all function calls are regular functions calls, while *YouTube* has none. Regular JavaScript [61] function calls are in web applications more prone to misspeculations than anonymous function calls since anonymous functions often are events. With a larger number of anonymous function calls, TLS is suitable to speedup JavaScript execution in web applications. *Gmail* and *Myspace* are 5% and 1% faster with the 2-bit than with the combined heuristic. In Figure 6.4 the 2-bit has a higher maximum number of threads than the combined heuristic. This indicates that even though the relationship between rollbacks and speculations is lower for the combined than for the 2-bit heuristic (Figure 6.6), the 2-bit heuristic allows us to extract most threads without rollbacks, which again improves the execution time.

In Fig 6.4, 13 out of the 15 uses cases have a lower maximum number of threads than the baseline. The number of threads are lower or similar for the combined heuristic in comparison to the 2-bit and the exponential heuristics. Together with the lower number of speculations relative to rollbacks in Fig 6.6, this indicate that we are able to speculatively execute function calls for a longer time with the combined heuristic than for the 2-bit, the exponential heuristic and the baseline. The effect is that the probability of a rollback decreases as we respeculate on JavaScript function calls. Further the number bytecode instructions that have to be re-executed on rollbacks decreases compared to the 2-bit, the exponential, and the baseline. This indicates that the nested nature of our TLS implementation allows us to gradually choose to respeculate on the right function calls, and that the number of threads that execute is different after a rollback.

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

In Fig. 6.5 we show the maximum memory usage for the heuristics relative to the baseline. As in Fig. 6.4, we see that the combined heuristic allows the majority of the threads to execute for a longer period of time. Therefore we are making a lower number of speculations, and thereby reduce the memory with up to 53% (*ebay*).

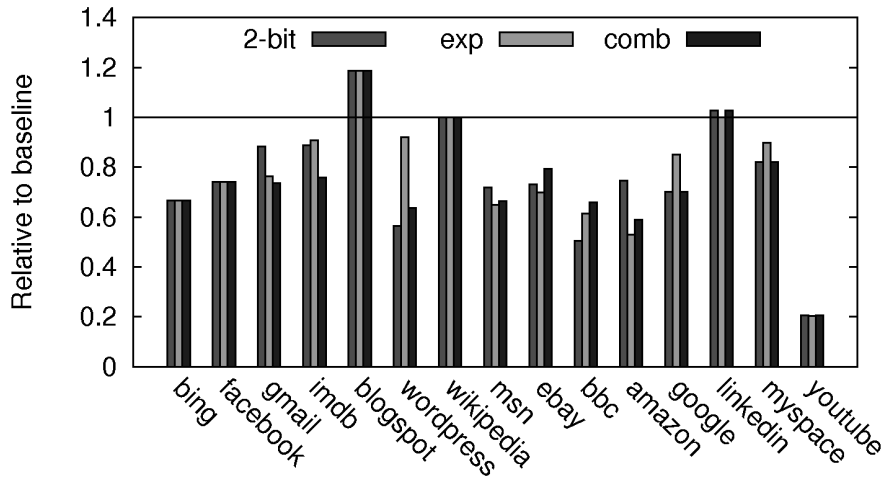

Figure 6.4: The maximum number of threads for 2-bit, exponential and the combined heuristic relative to the maximum number of threads for the baseline (i.e., when no heuristic is used).

6.6 Conclusion

We have shown in previous work [63] that Thread-level speculation consistently improves the JavaScript performance for a selection of web applications, which just-in-time compilation failed to do. In this paper, we present three heuristics to dynamically adjust the aggressiveness of the speculation. We evaluated them using use-cases for 15 popular web applications. Our results indicate that a combined heuristic, based on a 2-bit and an exponential, has the potential to both reduce the execution time and also increase the number of successful speculations.

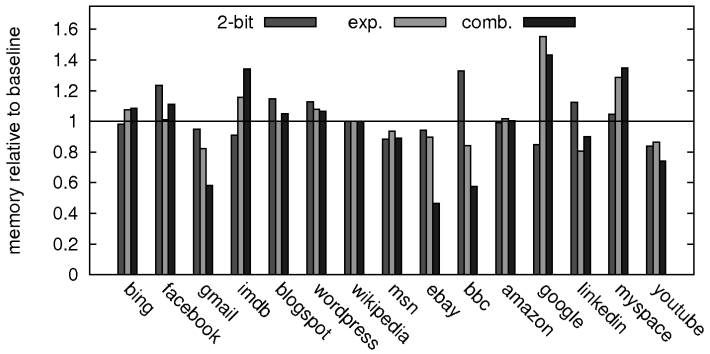

Figure 6.5: The maximum amount of memory used by the heuristics relative to the baseline.

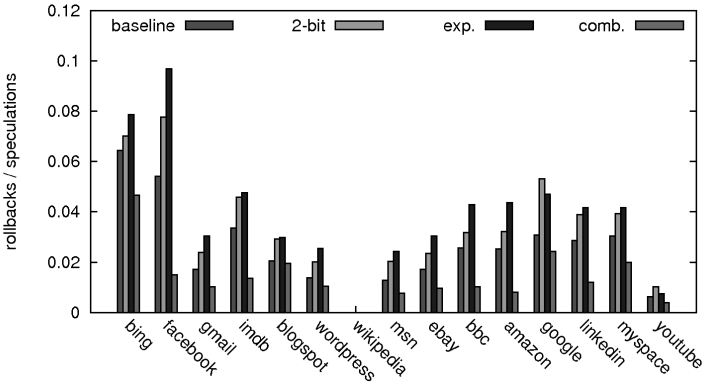

Figure 6.6: The relation between rollbacks and speculations when using the baseline, the 2-bit heuristic, the exponential heuristic, and the combination of the two heuristics.

Chapter 7

Paper VI

The Effects of Parameter Tuning in Software Thread-Level Speculation in JavaScript Engines

Jan Kasper Martinsen, Håkan Grahn, and Anders Isberg

Submitted to journal for publication

7.1 Introduction

JavaScript is a dynamically typed, object-based scripting language with run-time evaluation, where execution is done in a JavaScript engine [30, 108, 72]. One use of JavaScript is to add interactivity to web applications. Several optimization techniques have been suggested to decrease the execution time, e.g., just-in-time compilation (JIT) [30]. However, the decrease in execution time has been measured on a set of benchmarks, which are unrepresentative for JavaScript execution in web applications [56, 84, 89]. One result of this is that JIT decreases the execution time for benchmarks, while it often increases the JavaScript execution time in popular web applications [58, 63].

JavaScript is a sequential programming language and cannot take advantage of multicore processors to decrease the execution time. It is possible to take

advantage of multicore in web applications through Web Workers [104]. However, Web Workers is intended for improving the responsiveness of web applications, rather than decreasing the execution time. Fortuna et al. [25] have shown that there exists a significant parallelism potential in many web applications with an estimated speed up of up to 45 times.

To hide the details of the parallel hardware, one approach is to dynamically extract parallelism from a sequential program using Thread-Level Speculation (TLS) [91]. The performance potential of TLS has been shown for applications with static loops, statically typed languages, and in Java bytecode environments, and lately for the JavaScript SpiderMonkey engine on a series of well-known benchmarks [66] and with speculative execution [68].

In [55], we use TLS in the Rhino JavaScript engine and evaluated it on the V8 JavaScript benchmarks. In [67] the parallelization is extracted with a light weight speculation mechanism.

We extended our work in [63] and implemented TLS in the Squirrelfish JavaScript engine which is part of WebKit [108]and evaluated it on a number of web applications. However, even if we were able to decrease the execution time significantly; our approach had a high memory overhead.

In this study, we evaluate the effects of adjusting the amount of available memory, the maximum number of threads, and the speculation depth. We implement the limitations in the Squirrelfish engine and evaluate them on 15 web applications. We measure the effects of the adjustment on the execution time, memory usage, number of threads, speculation depth, number of speculations and the number of rollbacks.

Our results show that we can decrease the execution time and reduce the memory overhead by tuning these parameters.

Our main contributions are:

- The effects of limiting the execution resources for a Thread-Level Speculation scheme for a JavaScript engine.
- We find that 32–128 MB of memory, 16 threads, and a speculation depth of 4–16 is enough to reach most of the performance increase for the studied web applications.
- Nested speculation is necessary in order to achieve a high TLS performance for web applications.

This paper is organized as follows: In Section 7.2, we introduce JavaScript, web applications, and Thread-Level Speculation. In Section 7.3 we present our implementation of TLS and a comparisons with other JavaScript engines and in Section 7.4, we present the experimental methodology, and the studied web applications. Our experimental results are presented in Section 7.5, in Section 7.6 we discuss our findings and in Section 7.7 we conclude our findings.

7.2 Background

7.2.1 JavaScript

JavaScript [37] is a dynamically typed, object-based scripting language with runtime evaluation used to add interactivity in web applications. The execution is done in a JavaScript engine. JavaScript has a syntax similar to C and Java, while it offers features such as closures and anonymous functions often found in functional programming languages such as Haskell.

The performance of popular JavaScript engines such as Google's V8 engine [30], WebKit's Squirrelfish [108], and Mozilla's SpiderMonkey and TraceMonkey [72] has increased, reaching a higher single-thread performance for a set of benchmarks. However the results from these benchmarks are misleading [56, 84, 89] and optimizing towards the characteristics of the benchmarks increases the execution time for real-life web applications [58].

7.2.2 Web Applications

In web applications the client side computations are executed in a JavaScript engine and these functionalities are often defined as events. These events are defined as JavaScript functions that are executed for instance when the user clicks a mouse button, when a web page loads for the first time or certain tasks that are executed between time intervals. In contrast to JavaScript alone, web applications might manipulate parts of the web application that are not accessible from a JavaScript engine alone. The functionality is executed in a JavaScript engine, but the program flow is part of the web application. A key concept in web applications is the Document Object Model (DOM) that defines each element in the web application. The programmer can modify and create content in the web applications through the DOM tree with JavaScript.

Studies [56, 84, 89] have show that web applications use dynamic programming language features extensively. Parts of the program are defined at runtime, and types and extensions of objects are re-defined during runtime.

7.2.3 Thread-Level Speculation Principles

TLS aims to dynamically extract parallelism from a sequential program. This can be done both in hardware, e.g., [14, 85, 98], and software, e.g., [13, 41, 78, 81, 91]. Two main approaches exist: loop-level parallelism and method-level speculation. In this paper we use method-level speculation.

In method-level speculation, we execute function calls as threads and we must correctly predict the return values when we speculate as well as detect the writes and reads that cause the speculative program to violate the sequential semantics. The last two are typically detected when the values associated with two function calls are committed back to their parent thread. Between two consecutive 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 runtime using information about read and write addresses. A key design parameter for a TLS system is the *precision* of at 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 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.

7.2.4 Software-Based Thread-Level Speculation

In this section we review some of the most important software-based TLS proposals.

Bruening et al. [13] proposed a software-based TLS system that targets loops where the memory references are stride-predictable. This 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. The results show speed-ups of up to almost five on 8 processors.

Rundberg and Stenström [91] proposed a TLS implementation that resembles the behavior 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 down-side is the memory overhead. They show a speed up of up to 10 times on 16 processors for three applications from the Perfect Club Benchmarks [8].

Kazi and Lilja developed the course-grained thread pipelining model [41] exploiting coarse-grained parallelism. They suggest to pipeline the concurrent execution of loop iterations speculatively, using run-time dependence checking. In their evaluation they used four C and Fortran applications (two were from the Perfect Club Benchmarks [8]). On an 8-processor machine they achieved speed-ups of between 5 and 7. They later extended their approach to also support Java programs [40].

Bhowmik and Franklin [9] 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. Their results from 12 applications taken from three benchmark suites (SPEC CPU95, SPEC CPU2000, and Olden) show speed-ups between 1.64 and 5.77 on 6 processors.

Cintra and Llanos [17] present a software-based TLS system that speculatively executes loop iterations in parallel within a sliding window. As a result, given a window size of W at most W loop iterations/threads can execute in parallel at the same time. By using optimized data structures, scheduling mechanisms, and synchronization policies they manage to reach in average 71% of the performance of hand-parallelized code for six applications taken from various benchmark suites [96, 8].

Chen and Olukotun shows [15, 16] 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). On four processors, their results show speed-ups of 3–4, 2–3, and 1.5–2.5 for floating point applications, multimedia applications, and integer applications, respectively.

Picket and Verbrugge [80, 81] developed SableSpMT.Their solution is implemented in a Java Virtual Machine, called SableVM, and thus works at the bytecode level. They obtain at most a two-fold speed-up on a 4-way multicore processor.

Oancea et al. [78] present a TLS proposal that supports in-place updates. They have a low memory overhead with a constant instruction overhead, at the price of 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. The results show that their TLS approach reaches in average 77% of the speed-up of hand-parallelized, non-speculative versions of the programs.

A study by Prabhu and Olukotun [82] analyzed what types of thread-level parallelism that can be exploited in the SPEC CPU2000 Benchmarks [96]. They identified a number of useful transformations, e.g., speculative pipelining, loop chunking/slicing, and complex value prediction.

A study by Hertzberg and Olukotun [34] has a runtime system that decreases the execution time, and where idle cores are used to analyze potentially forthcoming speculations. It reportedly decreases the execution time of SPEC CPU2000 Benchmarks by 49%.

A study by Tian et al. [102] presents a novel Copy or Discard (CorD) execution model to efficiently support software speculation on multicore processors using profiled C code transformation with LLVM [44] to support parallel execution. The state of speculative parallel threads is maintained separately from the non-speculative computation state. The computation results from parallel threads are committed if the speculation succeeds; otherwise, they are discarded. They achieve speed ups ranging from 3.7 to 7.8 on a server with two Intel Xeon quad-core processors.

Renau et al. [86] presents three mechanisms; Splitting Timestamp Intervals, Immediate Successor List, and Dynamic Task Merging for out-of-order spawning in TLS. These techniques are implemented into their custom compiler, and on

a quad core computer they are able to have an average speed up of 1.30 for the SPECint 2000 applications.

Mehrara and Mahlke [67] show how to utilize multicore systems in JavaScript engines. However, their study has a different approach as well as a different target than we have. It targets 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. They execute the runtime checks in parallel with the main execution flow (trace), and only have one single main execution flow. Our approach is to execute the main execution flow in parallel.

In [66] they introduce a lightweight speculation mechanism that focuses on loop-like constructs in JavaScript, and if the loop contains a sufficient workload, it is marked for speculation. As this code used the trace features of Spidermonkey, a selective form of speculation is employed. They found that they were able to make speculation 2.8 times faster for well known JavaScript benchmarks. Unfortunately, large loop structures are rare in real web applications as shown in [58].

Mickens et al. [68] suggest an event-based speculation mechanism which is deployed as a JavaScript library called Crom. However, unlike our approach, their main goal is to enhance the responsiveness, while our main goal is to reduce the JavaScript execution time by dynamically extracting parallelism.

In summary, there is a significant amount of research done on software-based Thread-Level Speculation. However, we have not found any study that thoroughly evaluates the effects of adjusting the amount of memory, the number of threads, or the depth of speculation, for web applications.

7.3 Thread-Level Speculation Implementation for JavaScript

In this section, we describe our TLS implementation [63].

7.3.1 Speculation mechanism

Execution in Squirrelfish is divided into two stages, first the JavaScript code is compiled into bytecode instructions, then the bytecode instructions are exe-

cuted. 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 later use the sequential execution trace to validate the correctness of the speculative execution off-line.

Initially we initialize a counter *realtime* to 0. For each executed bytecode instruction, the value of *realtime* is increased by 1. We give the interpreter a unique *id* (*p_realtime*) (initially this will be *p_0*).

During execution we might encounter the bytecode instruction that indicates the start of a function call. We extract the *realtime* value and the id of the threaded interpreter 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 our TLS program (Fig. 7.1). This is possible in JavaScript in web applications since we know that there is going to be a very large number of function calls. We check if this function previously has been speculated by looking up the value of *previous[function_order]*. *previous* is a vector where each element is indexed by the *function_order*. If the entry is 1, then the function has been speculated unsuccessfully. If the value is 0, then it has not been speculated or has been successfully speculated, and we call this position a fork point.

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

We then create a new thread which contains an interpreter with an unique id which contains a new Squirrelfish engine. We copy the value of *realtime* from its parent and modify the state of the parent such that the current instruction is changed from the position of the "function call" bytecode instruction to the position of the associated "end of function call" bytecode instruction. In other words, the parent thread ships 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 each time a suitable candidate for speculation is encountered, 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.

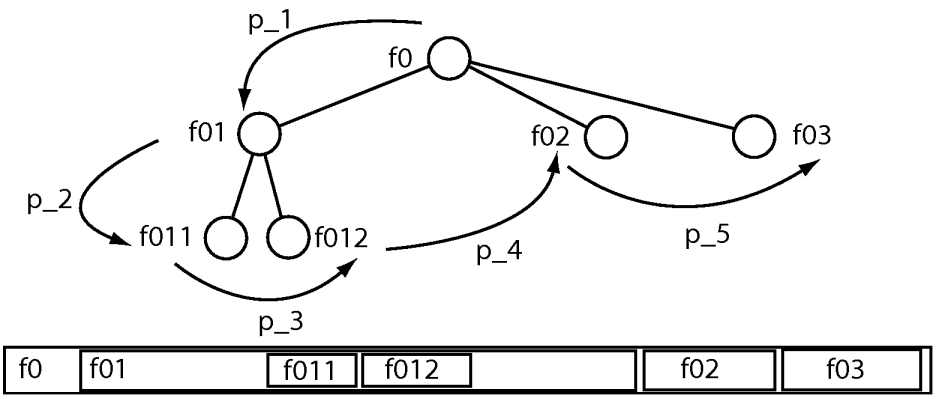

Figure 7.1: We use the order that the functions are called in, to determine the order in which the program would have been sequentially executed. This works in JavaScript in a web applications setting, as there are multiple function calls. For instance, in a simplified example the JavaScript function f0, performs 3 function calls, f01, f02 and f03. f01 performs two function calls, f011 and f012. Thus, we have created a speculation tree from the function calls. If we traverse this tree from left to right, we get an order in which the functions are called, equal to the order in which the functions would be sequentially called, in order to uphold the sequential semantics during execution with TLS. More specific: f0 at time p_1, f01 at time p_2, f011 at time p_3, f012 at time p_4, f02 at time p_5, and f03 at time p_6. We denote how each function is ordered as *function_order*.

Our return value prediction predicts the return values in *a last predicted value* manner [36] from a function with the same name (if a name is present). This is a simple heuristic for return value prediction, but as we mentioned earlier, function calls in JavaScript are often anonymous, use eval calls extensively, or these calls are events started from the web applications. These functions, do not return any value. Therefore does a heuristic such as the last returned value works fairly well for JavaScript execution in web applications.

In Fig. 7.2, 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.

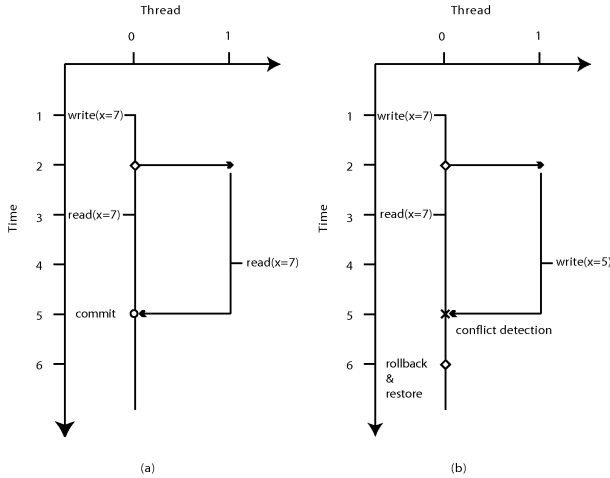

Figure 7.2: In (a), at time 1 in thread 0 we write the value 7 to the global value x. At time 2 in *thread 0* we encounter a function call which we speculate on and this function call becomes *thread 1*. At time 3, *thread 0* reads 7 from the global variable x. At time 4 *thread 1* reads the same variable. 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 call that becomes *thread 1*. At time 3 *thread 0* reads the value 7 from global variable x. At time 4 *thread 1* writes 5 from the variable x. In the sequential case, the function called at time 2, the value of variable x would have been 5 at time 2 (i.e., when the function returns), and *thread 0* would read 5 from x after the function has returned. This means that *thread 0* is squashed, and when we commit the values at time 5 we can no longer ensure sequential semantics. Therefore at time 6, we need to restore the JavaScript execution state to the point before we speculated on the function call and do not speculate on this function call again.

7.3.2 Data dependence violation detection

For 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 an 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 list 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* into *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 an *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*. Therefore, we check the *realtime* and the *function_order* for the current read and the previous write. If a read occurred 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. Likewise, the same applies if 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. So, 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.

(iii) The current operation is a write, and there is a previous 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*.

7.3.3 Rollback

Cases (ii) and (iii) force us to do a rollback for program correctness globally, further 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. If the function where the rollback occurs is nested, we stop the JavaScript interpretion of its child threads, and place the associated threads back in a thread pool for later reuse. 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 in the register based JavaScript engine, the values of the global variables and object property ids 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 is not associated with the current state of the TLS system. Therefore, we 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 remaining interpreters (running as threads), which are not necessary for the current state of the execution are stopped, and returned to the thread pool for later reuse.

7.3.4 Commit

When a speculative thread reaches the 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. 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.

Table 7.1: The web applications used in the experiments.

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

7.4 Experimental Methodology

We have extended our TLS implementation with three parameters to control the maximum memory, the maximum number of concurrent threads and the maximum depth in nested speculation. When we encounter a JavaScript function suitable for speculation, we first check these parameters. If they are below the specified limit, we speculatively execute the function. If a parameter is above the limit we executed the function sequentially.

7.4.1 Web applications

We have selected 15 web applications (Table 7.4.1) from the Alexa list [4]. We selected different types of web applications, such as search engines (Google and Bing) and various types of social networks (*Facebook* and *Linkedin*).

We have based our use cases on personal usage (such as searching in *Amazon* for one of the authors of this paper). In addition, we have tried to reduce the mouse interaction, as the screen size and navigation devices vary across different platforms. This way our results could be applicable on many types of devices.

The JavaScript executed in web applications is fundamentally different from what is executed in the JavaScript benchmarks e.g., with multiple calls to events which often are defined as anonymous functions [56]. The number of calls varies from 12 to over 10000, but the execution characteristics are the same. These events are allowed to run for a predefined time.

To enhance reproducibility and to provide a deterministic and reproducible behavior we automatically execute the use cases [11]. The methodology for these experiments is described in [56]

To validate the correctness of our TLS implementation, 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.

7.4.2 JavaScript functions in web applications

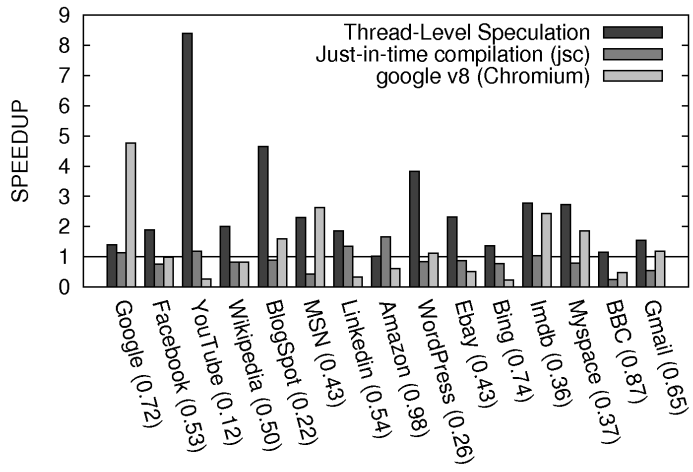

Figure 7.3: The execution time of TLS in comparison to Squirrelfish and V8, both with just-in-time compilation enabled [63] normalized to the execution time of Squirrelfish without JIT.

We base our experiments on the Squirrelfish JavaScript engine, where just-in-time (JIT) compilation is optional. We use the interpretive mode, since Fig. 7.3 shows that JIT compilation increases the execution time for 11 out of 15 use cases for Squirrelfish (when JIT is enabled) and 8 out of 15 for Google's JavaScript engine V8.

JIT compilation in Squirrelfish and V8 is measured on a set of benchmarks, and it has been shown [56, 84, 89] that these are unrepresentative for the workload in web applications. Benchmarks are similar to well-known benchmarks in other fields with a large number of loops. Since there is a limit (i.e., 10 seconds in Firefox and 5 second for Internet Explorer) to how long a JavaScript call can execute, the problem sizes that are computed by the benchmarks are artificially small. If we compare the JavaScript execution in benchmarks to the JavaScript execution in web applications, we se that most JavaScript calls are events from the web application, and specific JavaScript features such as eval and anonymous functions are extensively used. This leads to little reuse of already compiled code which is an importan feature of JIT. Therefore JIT compilation speeds up the benchmarks, while it slows down web applications to a point where it is slower with JIT compilation enabled [63].

These arguments are not against JIT, but that JIT is optimized towards the behavior of unrepresentative benchmarks. This leads to increased execution time in web applications.

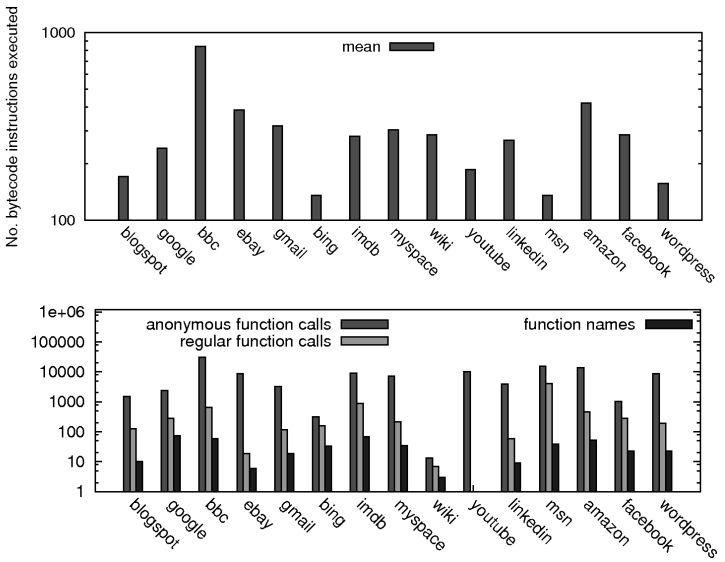

Figure 7.4: The mean length of the bytecode instructions executed in functions (upper) and the number of anonymous function calls, number of regular function calls and unique function names for the regular functions (lower).

Fig. 7.4, shows that the number of JavaScript function calls and their size in terms of executed bytecode instructions in web applications varies (the mean max

and min of the size of executed functions is 68168.75 and 2.19 bytecode instructions over an average of 15574.53 function calls). We can understand this from a nested speculation point of view. Functions at a low depth contain many of the proceeding function calls (i.e., *Wordpress* makes almost 80% of the functions call at depth 1 or 2), and as the depth increases the number of executed bytecode instructions decreases. In the figure below, we see that the most executed functions are anonymous function calls (i.e., for instance *youtube* only makes anonymous function calls). This shows that JavaScript in web applications is event driven. We also see that the functions that are not anonymous are seldom repeatedly called (i.e., for *msn* on average there are 104.64 distinct function calls (out of 39 function names) and 15609 anonymous function calls)

As an argument against JIT in these cases, each function call gets compiled, however most of the compiled code is not going to be re-executed and what is getting reused is very short. Therefore it is not going to be beneficial to execute it as native code (even if we optimize the native code).

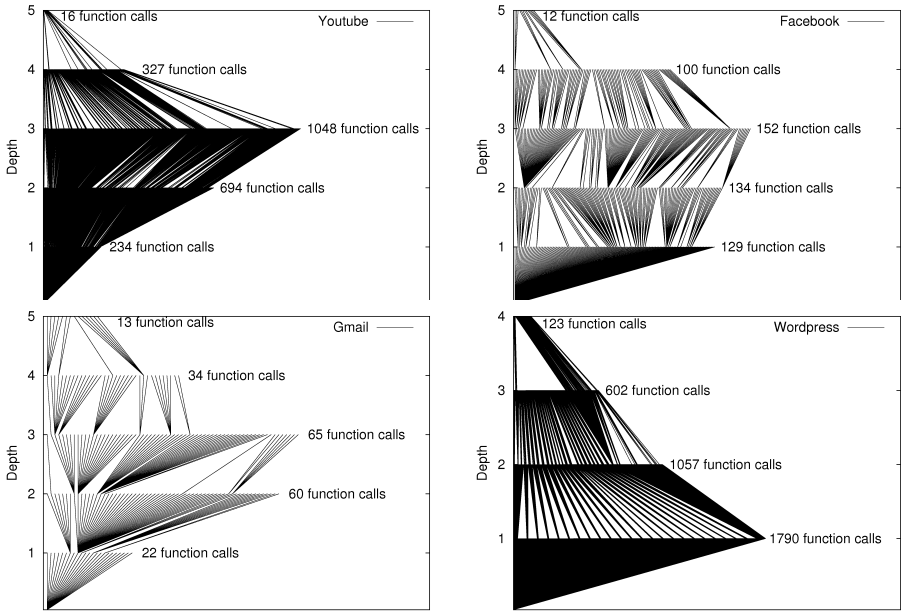

Figure 7.5: The number of nested function calls up to 7 levels for *Youtube*, *Facebook*, *Gmail* and *Wordpress* . We see as the depth of the nested function calls increases, the number of function calls decreases. We also see that the largest number of function calls is often not found at depth 1, but rather at depth 2 and depth 3. Each line represents a function call, and we can see by tracing the lines that some of the function calls spawn many new function calls. (For instance such as the number of function calls between depth 2 and depth 3 in *Facebook*). The rightmost number at each depth (vertical y-axis) indicates the number of function calls for each depth. From the Figures, the number of function calls is the largest at a higher depth than 1.

7.4.3 Nested function calls

Initially the depth of a function is 1. If this function makes a call to a function, the depth of the new function call will be 2, and if this function makes a function call, it will have depth 3, etc.

Fig. 7.5 shows that the number of functions start to decrease after depth 3. The number of JavaScript functions calls decreases after depth 3, since calls to events in web applications are only allowed to execute for a limited time (i.e., for *youtube* nearly 90% of all the functions calls are made before depth 4). Most web browsers report that the script is unresponsive if the JavaScript executes too

long. As the execution progress, so does the depth of function calls, therefore, JavaScript functions with a high depth do not account for most of the execution time in web applications.

7.4.4 Testing environment

All experiments are conducted on a system running Ubuntu 10.04 equipped with 2 quadcore, Xeon® 2Ghz processors with 4 MB cache each, i.e., a total of 8 cores, and 16 GB main memory. We have measured the execution time of the JavaScript execution performed in the JavaScript engine. There are other factors, I/O and css processing which affect the execution time of a web application. However, since one of the initial arguments is the difference between the JavaScript execution behavior of benchmarks and the JavaScript execution behavior in web applications, we focus on the JavaScript execution time. We have also disabled the number of cores to 2 and 4, to see which effects this has on the execution time.

7.5 Experimental Results

In Section 7.5.1 we have limited the memory used for speculation, in Section 7.5.2 we have limited the maximum number of concurrent threads, and in Section 7.5.3 we have limited the speculation depth.

7.5.1 Limiting the memory usage

In Fig. 7.6; (i) the execution time generally decreases with increased memory usage, and (ii) most of the performance increase is achieved between 32 MB and 128 MB.

Execution time

Up to 128 MB, we get on average a 2× speedup compared to sequential excution time. With more than 128 MB, 7 out of 15 web applications are unable to further decrease the execution time.

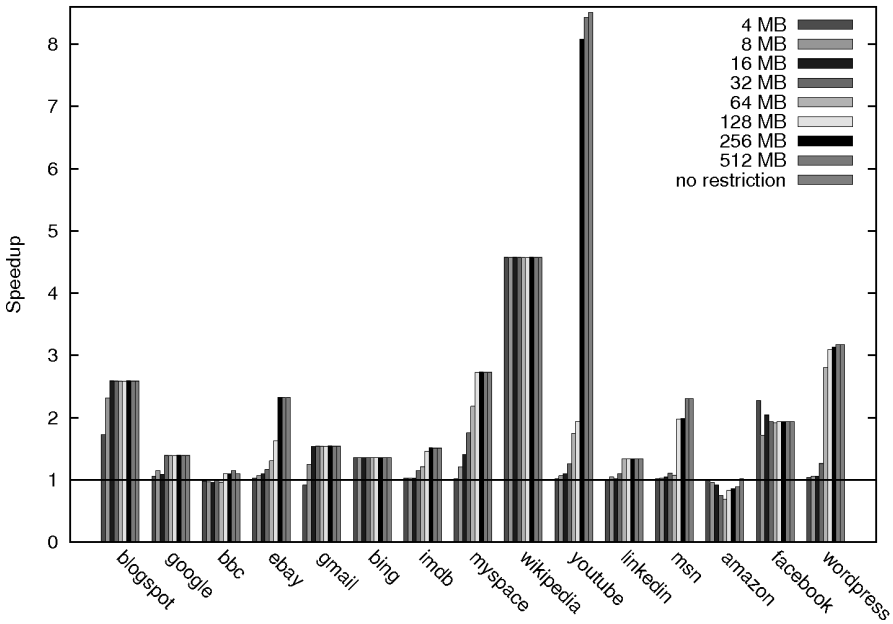

Figure 7.6: The speed up when we limit the available memory to 4,8,16,32,64, 128, 256, 512MB and with no restriction on the memory usage. The horizontal line in the figure indicates the sequential execution time for comparison average speed up is 2.52 (excluding the *Youtube* use case, it is 2.09).

Amazon is 1% faster than the sequential execution time for 4 MB, then the execution time gradually increases to 64 MB (where it executes 54% slower than the sequential execution time), before the execution time decreases gradually, up to when no limitation is set, where it is faster 2% faster than the sequential execution time. This is the only use case where TLS could increase the execution time. Comparing *BBC* to *Amazon*, *BBC* executes 10% more bytecode instructions (which are the use case where the difference in terms of executed bytecodes are the smallest), but *Amazon* makes 2× as many function calls as *BBC*, and 44% of these function calls have a depth of 2. So when we speculate, we could choose a function at a low depth, and speculate on several function calls from this function call, and use up all of the memory on that. As we increase the memory we are allowed to speculate more and deeper, and therefore we are able to find enough speculations to reduce the execution time. The reason for this behavior is that many of the JavaScript functionalities read information from web-cookies,

since JavaScript is used to customized the web application to the visiting users previous behavior.

Fig. 7.6 shows that *Youtube* executes 1.86× as fast as the next fastest use cases, *wikipedia*. In *Youtube* there is a large number of identical functions running as events since Fig. 7.4 shows that all the function calls in this use case are anonymous. These are related to updating and suggesting similar videos to the one the user is currently watching. In Fig. 7.8 we execute 5.06× as many threads as the average number of threads for this use case. In Fig. 7.9 the number of speculations is 1.69× as many as the average number of speculations, but 31% of the average number of rollbacks.

The execution times of *Bing* and *Wikipedia* does not increase with more than 4 MB. The number of functions in these use cases is 5.6% and 0.24% of the number of functions for the other use cases, which explains why we are unable to take advantage of more than 4 MB.

These measurement indicates that although TLS requires memory, it is in many cases sufficient with between 32 MB and 128 MB to double the speed up.

Overhead of saving checkpoint states and committing values

In Fig. 7.7 we have measured the relative execution time of TLS relative to the sequential execution time. We have measured the time it takes to commit values and the time it takes to save states when we limit the memory usage to 4, 8, 16, 32, 64, 128, 256 and 512 MB relative to the execution time. Generally, the time it takes to save checkpoint states increases, while the time it takes to commit values when a function returns decreases as the memory usage increases. Therefore we spend less time committing data as the memory increases, but spend more time saving checkpoint states in case of a rollback. The overhead for saving states varies between 24% and 1% of the total execution time, and the overhead of committing values varies between 3% and 0.01%. Thus, the overhead values for TLS is in general very small.

Since commiting values and saving states usually consist of a low total amount of the cost of TLS, we have found that what is really expensive is to initialize the threadpool, especially if the initializtion of new threads is spread out while executing. We also found that the cost increases with an increasing number of cores.

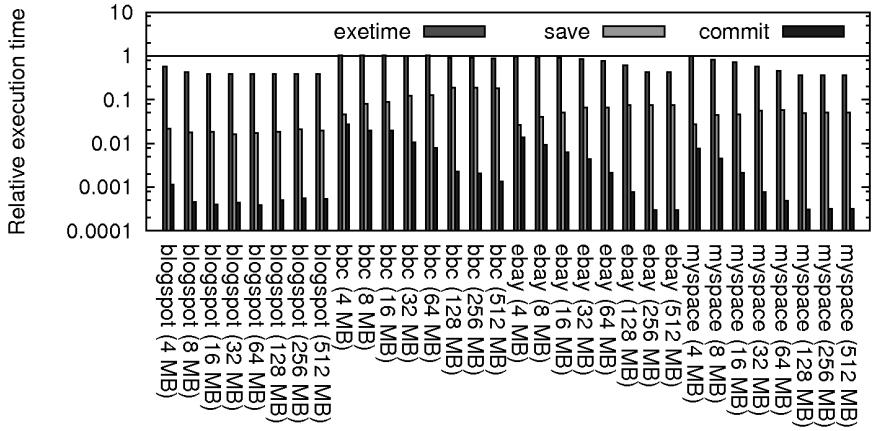

Figure 7.7: The relative improvement in execution time and the overhead of saving checkpoint states and commiting values.

No. of threads

In Fig. 7.8, 5 of the web applications are able to execute more than 50 threads. The functions in JavaScript can execute 2.19 bytecode instructions on a function call. Since we use nested speculation each thread has to wait until the threads it created returns. Due to the large number of function calls in web applications, and that functions are quite short; the number of threads running at certain points in time varies greatly. For instance, for *linkedin* the average number of threads executing are 2.64, while the maximum number of threads is 36.

If we reduce the number of cores from 8 to 4, our results indicate that we need to use 2.3× as much memory to get the same speed up, as when we have all cores enabled. This indicates that we need more memory to more memory to create a larger number threads to have the same execution time with a lower number of cores.

No. of speculations and no. of rollbacks

Fig. 7.9 shows a clear correlation between an increased memory and an increased number of speculations and an increased number of rollbacks. For instance between 4MB and an unrestricted amount of memory we get 16.12× as many spec-

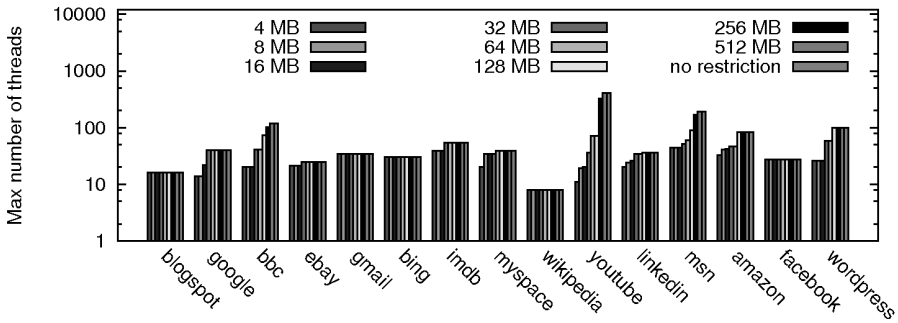

Figure 7.8: The largest number of threads when we limit the available memory to 4, 8, 16, 32, 64, 128, 256, 512MB and with no restriction on the memory usage.

ulations, and 26× as many rollbacks. However comparing the number of speculations and the number of rollbacks, we find that few of the speculations result in a rollback. For example, *Imdb* makes over 5000 speculations, with less than 150 rollbacks. The behavior of other applications are similar.

Summary

It is sufficient with between 32 MB and 128 MB since this is responsible for 97% of the performance improvements of TLS. In order to have the lowest possible execution time it is important to have between 35.6 and 48.3 threads running simultaneously, between 1267.6 and 3033.3 speculations and between 21.8 and 43.0 rollbacks. If the number of cores decreases, we need to use more memory to create more threads, and decrease the execution time.

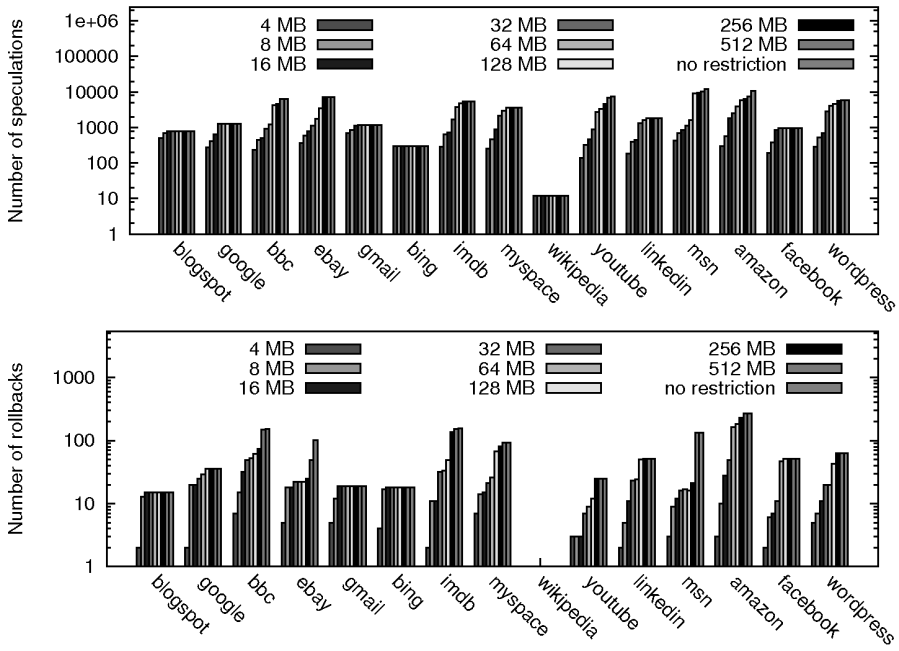

Figure 7.9: The number of speculations (upper) and the number of rollbacks (lower) when we limit the memory usage to 4, 8, 16, 32, 64, 128, 256, 512MB and with no restriction on memory usage.

7.5.2 Limiting the number of threads

Fig. 7.10 shows that the optimal number of threads in order to achieve the lowest execution time is between 8 and 32 ant that 8 web applications we have the highest speed up with 16 threads.

Execution time

We divide web application that are faster with TLS into three; (i) when the execution time increases with an increased number of threads (e.g. *Youtube*), (ii) when the execution time decreases with the number of threads, but after a certain number of threads, the execution time increases (e.g. *msn*) and finally, (iii) when there are spikes in the execution time, i.e., sudden improvements in execution

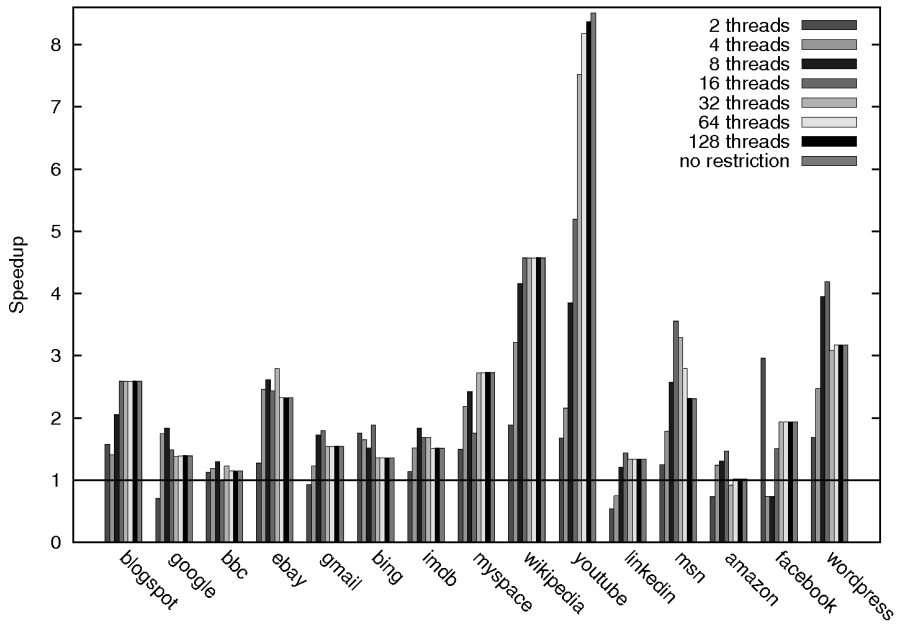

Figure 7.10: The speed up when we limit the number of threads to 2, 4, 8, 16, 32, 64, 128 and with no restriction on the maximum number of threads (average speed up, excluding the *Youtube* use case is 2.09).

time for a certain number of threads, while the previous and the proceeding ones are lower (e.g. *Facebook*).

The execution time decreases for *Youtube* (i.e.,it executes 3.89× faster with 128 threads than 4 threads). There are 1.69× speculations as the other use cases, and 32% of the rollbacks. By inspecting the executed bytecode and the JavaScript code we see that 68% of them have the same JavaScript code, even though they are anonymous. These are great candidates for being speculatively executed, many of them are events, and since they are anonymous function calls, they do not return anything.

If we limit the number of threads to 2 in *Facebook*, it executes 1.72× faster. We can understand this from the following; by using two threads, the overhead is significantly reduced (29% of when we do not limit the number of threads). In *Facebook*, we are unable to find an increased number of threads executing

concurrently when going from 32 to 128 threads. In Fig. 7.11 there is a 3.2× increase between the number of executing threads going from 128 to no restriction on the number of threads. If we look at the JavaScript execution in *Facebook*, there is a large number of executing functions at each depth. We also see that in Fig. 7.5 the functions are distributed evenly at each depth. For a limited number of threads, there is a limit to how many functions we can use for nested speculation. Without such a limit, we are able to execute more functions. This does not speed up the execution time. The memory usage of *Facebook* in Fig. 7.16 suggests that the functions are small in terms of number of executed bytecode instructions, and therefore commit quickly. This enables us to speculate on a lot of functions, but the increase in speculation due to the depth of function in Fig. 7.5 limits the gain in execution time (even though the number of available threads is very high).

For *msn* the performance increases with 2.86× from 2 threads to 16 threads. After that, the performance drops to 64% when we do not limit the number of threads to 16 threads. The drop in execution time occurs for the following reason; This use case has a large depth, which means a number of speculated functions are going to wait for the function they speculated to return, before they can return. This causes the threadpool to create and initialize more threads, which we showed in the previous section to have a significant cost. If we limit the number of threads, new threads will not be created by the threadpool at the same rate, which again reduces this overhead, since if all the threads are occupied, the function call will be executed sequentially.

For the ones that are slower than sequential execution time, they use between 2 and 8 threads (*Amazon* is slower for 16 threads). We see in Fig. 7.12 that even though we are using 2 threads, the number of rollbacks is almost the same as for 4 threads, while the number of speculations is much higher for 4 than for 2 threads. This shows that the cost of doing a rollback, along with the lack of speculation using 2 threads makes the execution time slower than the sequential execution time. We see the contrary in *Wikipedia* which has no rollbacks, and therefore the speed up is above the sequential execution time. This suggests that the number of threads must be higher than 2 in order to take advantage of TLS to decrease the execution time, which is an argument for nested speculation.

The ability to take advantage of the threads

Fig. 7.11 shows that 13 of the use cases are able to execute 32 threads concurrently when going from 16 to 32 threads. For 32 to 64, we are often able to use more than 32 threads, but only 5 use cases are able to use 64 threads. This shows that

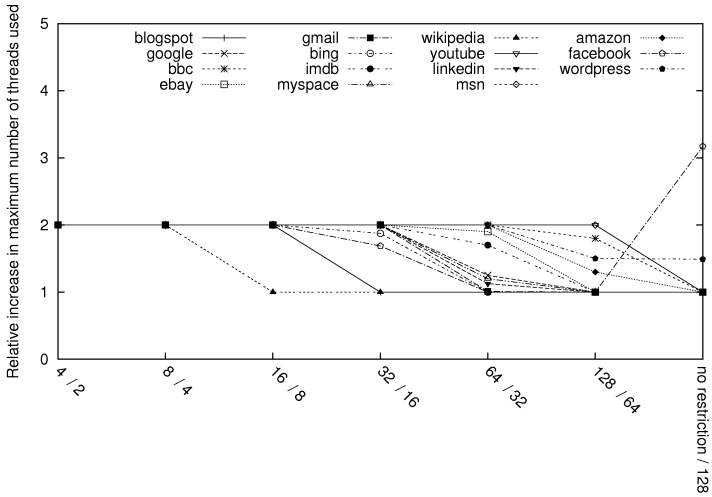

Figure 7.11: The relative increase in the highest number of concurrent threads increasing the maximum number of threads from 2 to 4, 4 to 8, 8 to 16, 16 to 32, 32 to 64, 64 to 128 and 128 to no limitation.

the real number of threads that we are able to execute concurrently is between 32 and 64. Since we see that for up to 32 threads most use cases are able to double the highest number of threads by adjusting the maximum number of threads, there is rarely any point increasing the maximum number of threads beyond 32. Only *Youtube* and *Wordpress* are able to take advantage of a maximum number of threads over 128. However, their speed up in execution time is negligible for this number of threads compared to 128 threads. *Youtube* is 4% faster, while *Wordpress* is only able to use a large number of threads, not to improve the execution time, because as we increase the number of threads, we are usually able to speculate deeper, however the number of bytecode instructions executed at a high speculation depth is limited. Therefore, there is a limit to how much we are able to speed up the execution time even if we are able to execute more threads.

No. of speculations and rollbacks

Fig. 7.12 shows that the number of speculations increases by 7.92× with an increasing number of threads. For 12 out of 15 web applications, the number of speculations does not increase when the maximum number of threads is higher

than 16. This shows that we are unable to find a sufficient number of functions to execute concurrently.

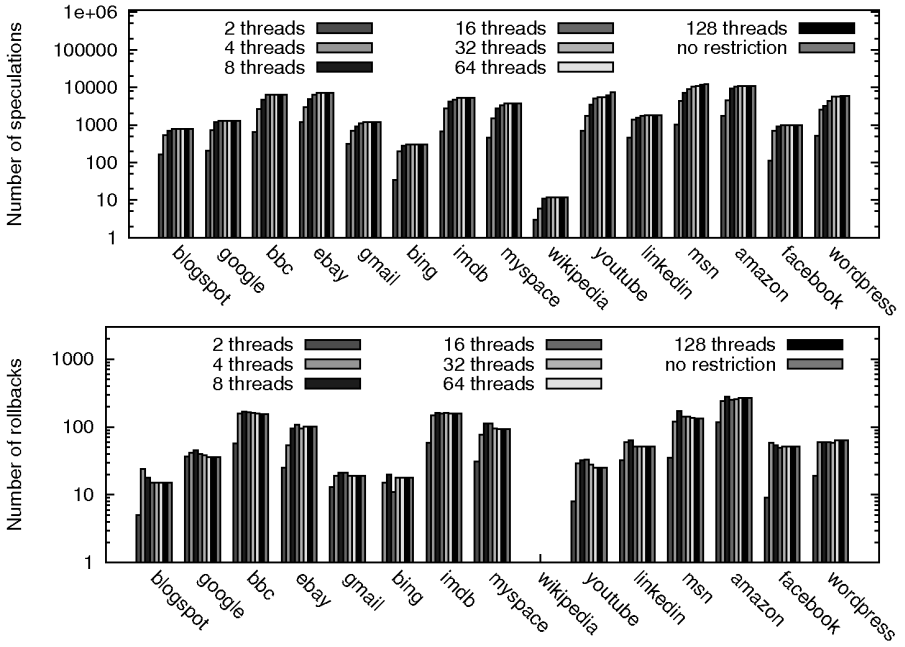

Figure 7.12: The number of speculations (upper) and the number of rollbacks (lower) when we limit the maximum number of threads to 2, 4, 8, 16, 32, 64, 128, and with no restrictions on the number of threads.

From the number of rollbacks there is often an over 3× increase in the number of rollbacks going from 2 to 8 threads. However, there is often a decrease in the number of rollbacks as the number of threads increases from 16 up to no limitation on the number of threads. This pattern is common; first the number of rollbacks increases, then the number of rollbacks gradually decreases as the number of threads increases. In Fig. 7.12, there is not a clear correlation between an increased number of speculations and an increased number of rollbacks. This indicates that a larger number of threads does not necessarily mean a larger number of rollbacks, In fact, it might mean the opposite, and an increased number of threads might reduce the number of rollbacks. We get 3.33× the speculations when we limit the number of threads to 4 compared to when we limit the number of threads to 2. The significant change in the number of speculations is because

of the reduction in the number of available threads. In Fig. 7.4 we see that there are many functions in web applications, but that they are small. Then we could end up executing many small functions with a limited number of threads, which would have a marginal effect on the execution time. This is the reason why we need a certain number of threads to improve the execution time. If we reduce the number of cores on the system (i.e., two or four) we end up using more threads to have the same execution time as using eight cores.

As we restrict the number of threads, the speculation depth decreases. This makes us unables us take full advantage of nested speculation. In Fig. 7.12 the number of speculations increases as the number of threads increases. However, JavaScript TLS characteristics in web applications also indicate that the number of bytecode instructions decreases as the depth increases. This shows that as the depth increases, a large number of functions are able to execute simultaneously, and that the functions are often able to commit quicker. This reduces the number of dependencies between speculated functions, which in turn reduces the number of rollbacks. In addition the number of anonymous functions of JavaScript in web applications show that there are few return values.

Memory usage

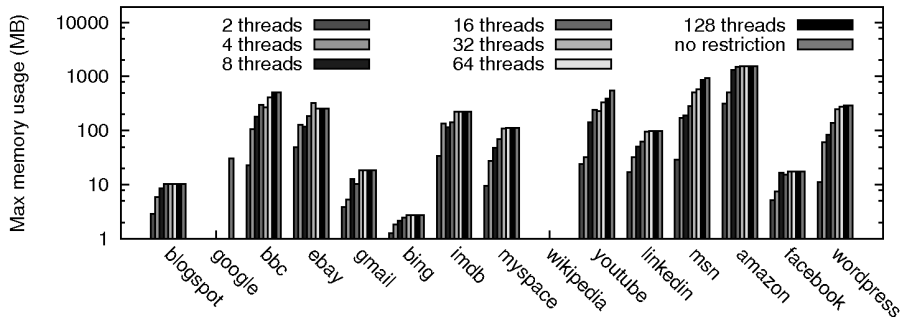

Figure 7.13: The memory usage when we limit the number of threads to 2, 4, 8, 16, 32, 64, 128 and with no restriction on the maximum number of threads.

In Fig. 7.13 we see that as we increase the number of threads we increase the memory usage by 8.69×. For example, the extremes are *msn* and *Amazon* that use more than 937MB and 1.5GB of memory if we do not limit the maximum number of threads. One interesting use case is *Google*, where the memory

increases with 1024× when we do not restrict the number of threads. However, these threads are very small in terms of bytecode instructions, but by not restricting the number of threads, we are able to speculate multiple threads in a nested manner, which in turn increases the memory usage.

The results show that uncritically increasing the number of threads only has the lowest execution time for 3 out of 15 use cases, and has a high cost in terms of memory. The optimal number of threads to decrease the execution time seems to be between 8 and 32. A maximum number of threads set to less than 8 indicates that we are unable to create a sufficient number of threads (e.g. *linkedin*).

Summary

We need no more than 32 threads to reduce the execution time. Only 2 use cases use more than 128 threads. The speed up from 64 threads and upwards is negligible (i.e., at best 4% faster than when we restrict the number of threads to 64). This shows that there is a potential for extracting a large number of threads from the JavaScript code in web applications. However, as the number of threads increases the overhead of having a larger number of threads increases the amount of memory used for speculation which again reduces the improved execution time along with a decreasing potential of speculation as the depth increases, since the functions are so short, we are often able to re-use threads. One interesting observation is, if we reduce the number of cores, we need to extract more threads to have the same speedup.

7.5.3 Limiting the speculation depth

The most important observations in this section are; (i) we need to use nested speculation in order to decrease the execution time and (ii) that a speculation depth of 16 leads to the best perfromance.

Execution time

Fig. 7.14 shows that nested speculation is necessary to improve the execution time

With a speculation depth of 2 for *Gmail*, it is 52% faster than when we do not limit the speculation depth. In Fig. 7.15 the number of speculations for *Gmail*

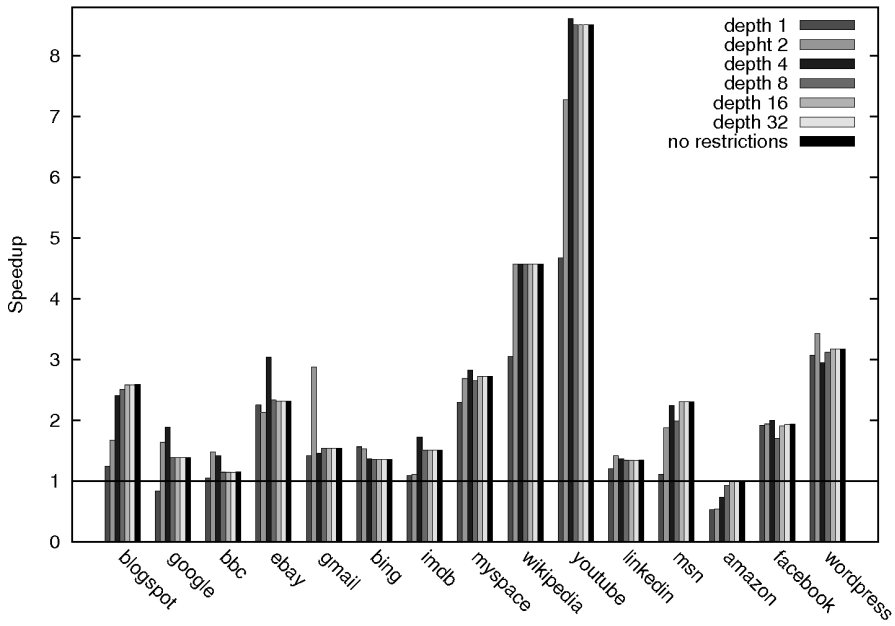

Figure 7.14: The speed up when we limit the speculation depth to 2, 4, 8, 16, 32, and with no restriction on the depth (average speed up when we exclude the *Youtube* use-case is 2.34).

is the highest for speculation depth 2, and the number of rollbacks is the lowest. The memory usage is lower for depth 2, which decreases the overhead of TLS. The behavior in *Gmail* is caused by much JavaScript functionality (compared to some of the other use cases) executed when the page loads. Further JavaScript execution is caused by more user interaction. Our use cases have reduced user interaction, therefore we would probably see at better effect with more user interactions. In Fig. 7.5 most of the functions are found at depth 2 and 3. This explains the large speed up of *Gmail* at depth 2.

13 of the 15 use cases have the largest speed up with speculation depths set to 4, 8, or 16. A speculation deeper than 16 only gives the highest speed up for *Blogspot*. This means that the cost of speculating deeper increases and the potential speed up by being able to speculate decreases.

No. of speculation and no. of rollbacks

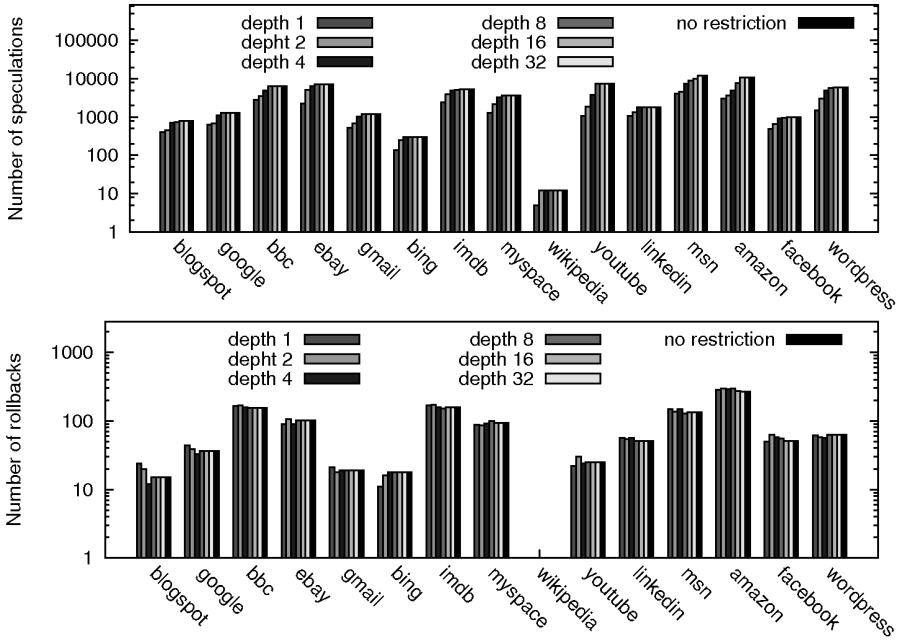

Figure 7.15: The number of speculations (upper) and rollbacks (lower) when we limit the depth to 2, 4, 8, 16, 32, and with no restriction on the depth.

Fig. 7.15 shows that there is a relationship between an increased speculation depth and an increased number of speculations, although there is a limit to the number of speculations we are able to make with a speculation deeper than 8. We execute fewer and fewer bytecode instructions as the speculation depth increases ,since the number of JavaScript functions decreases as the speculation depth increases (Fig. 7.5). This means that the potential gain of speculation decreases, as the number of functions and the size of each function decreases, while we save more states. Therefore, the speed up rarely increases with a speculation depth higher than 4.

For a speculation depth over 8, the number of rollbacks decreases as the speculation depth increases. Since the size of the functions decreases, they commit back to the parent faster than they would if the size of the function was bigger. Given that a function speculates on a new function (i.e., nested speculation) it has

fewer dependencies between itself and the function it speculates on, than there is between two functions which have the same depth (i.e., for instance function calls that are made as part of a loop). These functions, rarely return a value, or at least one that we were unable to predict correctly. This is because many of these functions read elements in the DOM tree.

Memory usage

In Fig. 7.16 we see that an increased speculation depth means more speculations (Fig. 7.5), and as a result more checkpoints states must be saved. This means that we get an increased overhead of saving the checkpoint states, relative to a lower depth. However there are a lower number of variable checks as the number of bytecode instructions decreases as the depth increases, and the functions commit earlier.

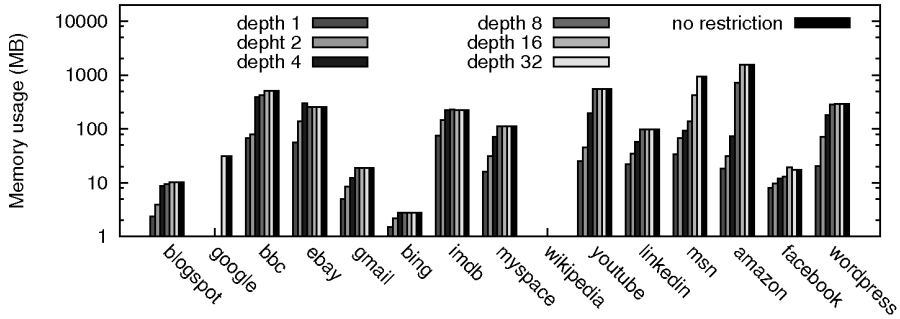

Figure 7.16: The memory usage when we limit the speculation depth to 2, 4, 8, 16, 32, and with no limit on the depth.

Summary

Nested speculation speeds up the execution, but any benefit of speculating deeper than 16 is rare. Since the size of the function decreases as we speculate deeper, then the cost of speculation outweighs the potential gain of executing the function in parallel. One interesting observation is that as the speculation depth increases, then for 12 out of 15 use cases, the number of rollbacks is reduced.

7.6 Discussion

As observed in Section 7.5.1 and Section 7.5.3 both *Amazon* use cases can be slower than the sequential execution. This is because when we limit the memory, we are often unable to speculate deep enough, which in turn could slow down the execution. When we increase the speculation depth, the execution time improves. In Section 7.5.2 we limit the number of threads, then the same use cases are often able to find the correct threads to speculate on, and initially the overhead is reduced so we get the highest speed up.

When we increase the depth, we find more functions to speculate on; therefore we save more checkpoint states. However the size when we speculate with an increased depth is decreasing compared to a lower depth, as the number of executed JavaScript bytecode instructions is decreasing. The number of variable checks for each commit is decreasing; as the depth increases (we see this in terms of reduction of rollbacks with a high depth). There is a significant increase in overhead related to commiting going from depth 1 to depth 4, but for higher depths this overhead is reduced. There is also a significantly higher cost of a rollback at a low depth, than at a high depth.

To get the bound of improved execution time of JavaScript using TLS in web applications, we compare our results against the results of [25]. Their average speed up is 8.9× faster which is clearly faster than the results in this paper, but they make their argument from a theoretical point of view. Our use cases are methodologically performed with a focus on reproducibility [56]. This causes our use cases to have less JavaScript execution, and fewer JavaScript functions to speculate on.

Our study is based on a real implementation of TLS in a state-of-art JavaScript engine. We see from the speed up figures that we could benefit from a larger number of cores to increase the speed up for some of the use cases. For the other use cases, they are limited due to the limited user interaction, and thereby reduced JavaScript execution. For *Youtube*, we claim that our TLS solution would further speed up with a larger number of cores, as the execution time decreases when we disable the number of cores to 2 or 4, on our 8 core computer, for other use cases the gain of a larger number of cores is not nearly as high. There is also a cost (in terms of saving the checkpoint state) for each speculation.

7.7 Conclusion

We must use nested speculation in order to speed up the execution time. 16 threads, 32MB–128 MB of memory, and a speculation depth between 4–16 levels often result in the highest speed up

TLS is a suitable technique for increasing the performance in web applications on devices with multicore processors. From the number of speculations and the number of threads running concurrently, there is an indication that there is a potential for a higher speed up with an increased number of cores.

Chapter 8

Paper VII

Reducing Memory in Software-Based Thread-Level Speculation for JavaScript Virtual Machine Execution of Web Applications

Jan Kasper Martinsen, Håkan Grahn, Anders Isberg and Henrik Sundstrőm

Submitted for publication, patent filed

8.1 Introduction

JavaScript is a dynamically typed, object-based scripting language with run-time evaluation used extensively in web applications, where the execution is done in a JavaScript engine such as Mozilla Spidermonkey [72]. Google [30] has suggested Just-in-time compilation (JIT) to decrease the execution time in JavaScript. However, Google's JavaScript engine V8's decrease in execution time has been measured on a set of benchmarks, which Ratanaworabhan et al. [84] show are unrepresentative for real-world web applications. Martinsen et al. [58] show the dramatic effect of this as JIT speeds up the execution of benchmarks, but often *slows down* the execution time in popular web applications.

JavaScript is a sequential scripting language and cannot take advantage of multicore processors to reduce the execution time. Fortuna et al. [25] show that there exists a significant potential for parallelism in many web applications with an estimated speedup of up to 45× compared to a sequential execution.

To hide the details of the underlying parallel hardware, we can dynamically extract parallelism from a sequential program using Thread-Level Speculation (TLS). Mehrara et al. show the performance potential of TLS in the SpiderMonkey JavaScript engine on a series of well-known benchmarks [66] and Martinsen et al. [63] show this on a number of popular web applications. However, while prior results show that TLS can significantly speed up the execution time in web applications, it uses over 1500 MB of memory.

We propose to reduce the memory overhead in TLS by being selective on when we store the checkpoints before we speculate. Martinsen et al. [63] show that less than 1.8% of all the speculations result in a rollback. However, if we choose to store the checkpoint far away from the rollback, the number of bytecode instructions that need to be re-executed increases. Martinsen et al. [61] show that nested speculation is necessary to decrease the execution time; therefore we investigate the effects on memory usage and execution time of not storing the checkpoints at all speculation depths. Further, we propose an adaptive heuristic that dynamically adjusts when we store the checkpoints depending on the speculation depth and the number of rollbacks.

We show that we can reduce the memory usage in TLS by nearly 90% by limiting at what speculation depth we store the checkpoint, and in several cases also improve the execution time. Based on these findings, we develop and evaluate an adaptive heuristic, which reduces the memory usage by over 90% and has an execution speed close to the results of Martinsen et al.

Our main contributions are:

- Reducing the memory requirements of software-based TLS in JavaScript by only storing a limited number of checkpoints.
- An in-depth study of the effects of limiting the number of checkpoints for a TLS in JavaScript.
- An adaptive heuristic which significantly reduces the memory usage for TLS and improves the execution time.

This paper is organized as follows; in Section 8.2, we introduce JavaScript, web applications and TLS. In Section 8.3, we present the TLS implementation

used in our study. In Section 8.4 we present our approaches to reducing the memory usage in TLS. In Section 8.5, we present the experimental methodology, while in Section 8.6 and Section 8.7 we evaluate the effects of a fixed number of checkpoints and the adaptive heuristic. Finally, in Section 8.8 we conclude our findings.

8.2 Background and related work

8.2.1 JavaScript and web applications

JavaScript is a dynamically typed, object-based scripting language with run-time evaluation used in web applications. JavaScripts performance has reached a high single-thread performance on a set of benchmarks. Ratanaworabhan et al. [84] show that the results from these benchmarks are misleading for the execution behavior of web applications and Martinsen et al. [63] show that optimizing towards the characteristics of the benchmarks slows down the web applications.

Web applications manipulate parts that are not accessible from a JavaScript engine. The scripted functionality is executed in a JavaScript engine, but the program flow is defined in the web application. Richards et al. [89] show that web applications use JavaScript specific features extensively such that various parts of the program are defined at run-time.

A key concept in web applications is the Document Object Model (DOM). DOM is an interface that allows programs and scripts to dynamically access and update the content of documents. The document can be further processed and the results can be incorporated back into the presented page. The programmer can modify, create, or delete elements and content in the web applications through the DOM tree with JavaScript.

8.2.2 Thread-Level Speculation principles

Picket and Verbrugge's [81] TLS approach is to allocate each function call in software as a thread. Then, they can (ideally) execute as many function calls in parallel as they have processors. However data dependencies and return values limits the number of function calls that can be executed in parallel. Martinsen et al. [63] show that the memory requirements and Rudberg et al. [91] show that the run-time overhead for detecting data dependencies can be considerable.

Between two functions 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 function call. A design parameter for TLS is the *precision* of at what granularity of true-positive / (true-positives + false-positives) data dependency violations are detected.

When a data dependency violation is detected, the execution must be 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. The book-keeping results in both a memory overhead as well as a run-time overhead. In order for TLS to be fast, the number of rollbacks should be low.

The more precise tracking of data dependencies, the larger memory overhead is required. 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 are done, which increase the execution time. TLS implementations 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 speculation buffer.

8.2.3 Thread-Level Speculation in JavaScript and for web applications

Mehrara and Mahlke [67] target trace-based JIT-compiled JavaScript code, where the most common execution flow is compiled into an execution trace. 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 have one single main execution flow.

Mehrara et al. [66] introduce a lightweight speculation mechanism that focuses on loop-like constructs in JavaScript. As this code uses the trace feature of Spidermonkey, a selective form of speculation is employed.

Mickens et al. [68] suggest an event-based speculation mechanism which is deployed as a JavaScript library (Crom) which clones certain regions of the JavaScript code that are executed speculatively.

Martinsen et al's [63] approach is to execute the main execution flow in parallel which they evaluate on popular web applications and show that there is a

significant potential for TLS in web applications. Figure 8.1 shows that Just-in-time compilation (JIT) increases the execution time of web applications as compared to interpretive execution, and shows that neither the Squirrelfish or V8 JavaScript engine improves execution time for JavaScript in web applications with JIT. They also show that nested TLS is necessary in order to improve the execution time with TLS.

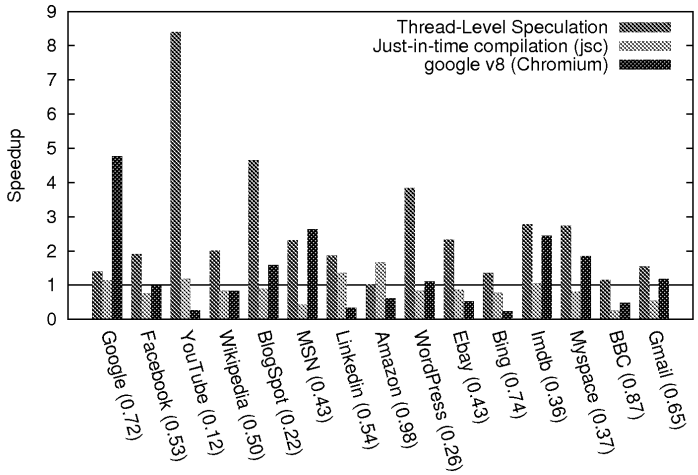

Figure 8.1: Speedup of Thread-Level Speculation and Just-in-time compilation for a number of popular web applications. The black horizontal line is the sequential execution time of Squirrelfish without Thread-Level Speculation (the figure is used with permission from the authors of [63]).

Martinsen et al. [47] suggest three heuristics for re-speculation on previous mis-speculations. Their conclusion is that the overall problem with TLS in JavaScript for web applications is not the number of rollbacks but the memory usage.

Martinsen et al. [61] show that by limiting the number of threads, the amount of memory, and the speculation depth we both save memory and improve the execution time. In many cases a speculation depth of 2 to 4 is sufficient to improve the performance because of the nature of JavaScript execution in web applications. JavaScript execution in web applications as events are restricted to be executed for no more than half a minute. This indicates the lack of large loop structures, which again reduces the effect of JIT.

In summary, Martinsen et al's [63] over 1500 MB memory usage in TLS is a concern. We have not found any studies that look at reducing the memory

overhead of TLS in web applications by being restrictive on when we store checkpoints.

8.3 TLS implementation for JavaScript

Martinsen et al. [63] implemented TLS in the Squirrelfish JavaScript engine. The speculation is done on JavaScript functions in a nested manner, including return value prediction, and all data conflicts are detected at run-time. When a conflict is detected, a rollback is performed. We extend Martinsen et al.'s implementation with a mechanisms to limiting the number of checkpoints, and use it for the measurement and the analysis.

The execution in Squirrelfish is divided into two; first the JavaScript code is compiled into bytecode instructions, then the bytecode instructions are executed. We extract the bytecode instructions which are to be executed, and the execution trace of a sequential execution of the bytecode instructions. We use this to validate the correctness of the speculative execution off-line. We initialize a counter *realtime* to 0. For each executed bytecode instruction, the value of *realtime* is increased by 1. We give the function call a unique *id* (*p_realtime*)

When we execute a *op_call* bytecode instruction we extract the *realtime* value and the id of the speculated function that makes this function call, e.g., *p_0220* (a function is called after 220 bytecode instructions from *p_0*). In Figure 8.2 the value of the position of this function call emulates the sequential execution order for TLS. This is possible in web applications since there is going to be a large number of JavaScript function calls. We check if this function previously has been speculated by looking up the value of *previous[function_order]*. *previous* is a vector where each element is indexed by the *function_order*. If the value is 1, then the function has been speculated unsuccessfully. If not, this function call is a candidate for speculation.

Then we do the following; we set the position of the function call's *previous[function_order] = 1*. We save the state which contains the list of previously modified global values, the list of states from each thread, the content of the variables in the JavaScript engine, and the content of *previous*.

We create a new thread for the function with a unique *id*. We copy the value of *realtime* from its parent and modify the state of the parent such that the current instruction is changed from the position of the *op_call* bytecode instruction to the position of the associated *op_ret* bytecode instruction.

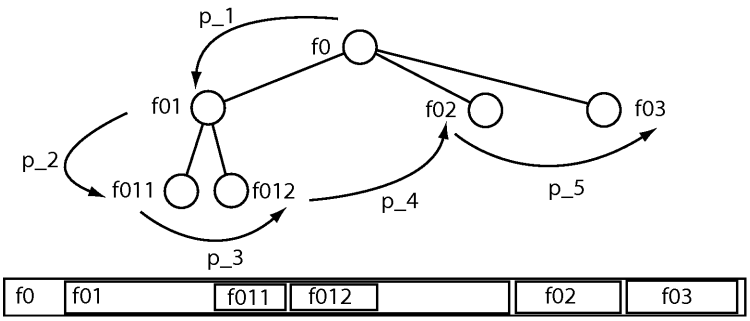

Figure 8.2: We use the order that the functions are called in, to determine the order in which the program would have been sequentially executed. This works in JavaScript in a web applications setting, as there are multiple function calls. For instance, in a simplified example the JavaScript function f0, performs 3 function calls, f01, f02 and f03. f01 performs two function calls, f011 and f012. Thus, we have created a speculation tree from the function calls. If we traverse this tree from left to right, we get an order in which the functions are called, equal to the order in which the functions would be sequentially called, in order to uphold the sequential semantics during execution with TLS. More specific: f0 at time *p_ 1*, f01 at time *p_ 2*, f011 at time *p_ 3*, f012 at time *p_ 4*, f02 at time *p_ 5*, and f03 at time *p_ 6*. We denote how each function is ordered as *function_ order*

We have two functions executing as concurrent threads, and this process is repeated each time a *op_ call* bytecode instruction is encountered, thereby allowing nested speculation. For correct speculative execution, we check for write and read conflicts between global variables, object property id names, unsuccessful return value predictions of function calls, and whether we write to the DOM tree. If a conflict occurs, we perform a rollback. When a speculative function encounters *op_ ret*, modifications of global variables and object property ids are committed back to their parent thread. However the commit cannot be completed before its speculative function calls return.

8.4 Reducing the memory usage for TLS

8.4.1 Motivation for limiting the checkpoint depth

In nested TLS, we store a checkpoint before we speculatively execute a function. The checkpoint is used if we mis-speculate and need to perform a rollback, e.g., due to a data conflict. However, when the speculation is correct, then the stored

checkpoint is removed when we commit back to it's parent thread. Martinsen et al. [63] show that 1.8% of the speculations result in a rollback, therefore most checkpoints are never used. The number in the parenthesis in Figure 8.6 shows that the memory usage can be 1527 MB for TLS.

Martinsen at al. [61] show that nested speculation is necessary to reduce the execution time, and that the speculation depth for 85% of all functions is between 2 and 4. Therefore, most rollbacks occur between depth 2 and 4, and as a result, it could be more meaningful to store the checkpoints below these depths as we expect a rollback at such a depth.

8.4.2 Fixed checkpoint depth limit

When a speculatively executed function makes a speculative function call, the depth of the speculated function is the caller's depth+1. The checkpoint of a speculative function is saved at a *checkpoint depth* equal to the depth of the function speculated. When we make the first speculation, the checkpoint is stored at checkpoint depth = 1.

Our idea is to limit the checkpoint depths were we store the checkpoints, but still allow an unlimited speculation depth. Normally, in case of a rollback we would go back to the caller function's parent checkpoint. In our approach, we suggest to only store checkpoints at a certain checkpoint depth.

Before a speculation, a predefined *checkpoint depth limit* is compared to the function's checkpoint depth. If the checkpoint depth is equal or below the checkpoint depth limit, we store the checkpoint. If the value of the checkpoint depth is higher than the checkpoint depth limit, we do not store the checkpoint, instead, on rollbacks we go to the previous stored checkpoint. This reduces the memory as we store a lower number of checkpoints, but this also means that rollbacks require a larger number of bytecode instructions to be re-executed. An example is shown in Figure 8.3.

8.4.3 An adaptive heuristic

A fixed checkpoint depth limit does not adapt to functions speculatively executing at different depths and rollbacks. We would like to store less checkpoints to reduce memory usage, but this makes rollbacks take more time. We therefore propose *an adaptive heuristic that dynamically adjusts the checkpoint depth limit* based on the speculation depth and rollback behavior of a web application.

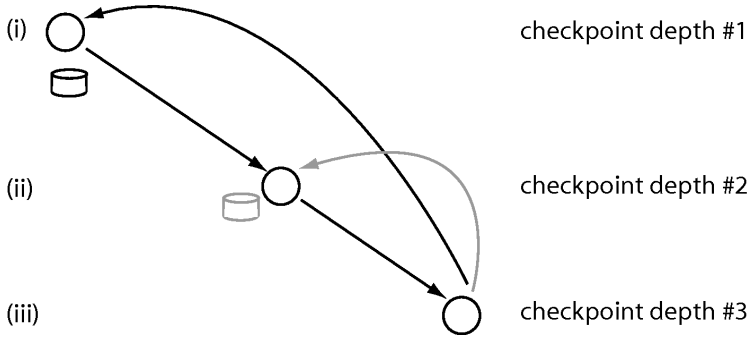

Figure 8.3: Before we speculatively execute a function at (i), we save the state so we can rollback to this point. At (ii), i.e., the speculative function made as a speculative function call at (i), we speculatively execute another function call (iii). In normal TLS, we also save the state in (ii) in case of a rollback. In our proposal, we do not store the state at checkpoint in (ii) if the checkpoint depth is set to 1. If a rollback occurs in (iii), we would normally rollback to (ii). However, in our proposal we would rollback to (i). As a result, we do not need to store the checkpointed state in (ii), with the cost of doing a rollback back to (i) instead of to (ii).

The heuristic in Listing 8.1 speeds up the execution time up to 8× and reduces the memory usage by over 90%, by being selective at which checkpoint depth limit we store the checkpoints. Martinsen et al. [63] show that as the speculation depth increases we have more rollbacks. If a rollback occurs, we want to reduce the number of bytecode instructions that needs to be re-executed. Martinsen et al. [63] show that rollbacks are rare, but that they often occur between speculative functions with the same depth and occur closely after each other. Therefore, when a rollback occurs, we want to increase the limit to ensure that the number of re-executing bytecode instructions is reduced for preceding rollbacks.

Listing 8.1: Since we are using nested speculation, each thread has a *depth*. First we go through all the threads executing and place their depth in a list l. In the next stage, we sort the list l ascending. Initially we set a variable m to 0.5. The value m is increased to $m = m + 1.0 / pow(2, no\_rollback + 1)$ if there is a rollback. Therefore, after the first rollback m would be 0.75, after the second rollback m would be 0.825, etc. We pick the element a from $l[m \times length\ of\ l]$, if the depth of the function we are about to speculate on is lower than a, we save the state. If not, we make sure that, in case of a rollback, we rollback to the last checkpoint were the state was saved. If the length of l is lower than 3 we set a to 2.

```
bool speculate(int depth){
 l = fetch_depth_of_threads();
 sort(l);
 m = m + 1.0 / pow(2, no_rollback + 1);
 if(len(l) < 3)
  return 2 > depth;
 int a = l[|m * len(l)|];
 return a > depth;
}
```

If a speculative function makes a function call, we create another thread. This threads' parent will be in the list of executing functions. One of the functions in this list could have a suitable checkpoint to rollback to and the motivations for doing this, is that the threads executing when you speculate on a function call, is probably one of the depths you will rollback to in case of a rollback. We can choose one such thread by the median of the currently executing threads' depths. Therefore the median could be a suitable automatic choice for a limitation of the checkpoint depth. However a fixed median value (like 0.75 or 0.25$\times$ the length of the list with depths), even though it made the memory usage lower, increased the execution time. This can be understood from the characteristics of JavaScript execution shown by Martisen at al. [58]; JavaScript functions are small in terms of number of executed bytecode instructions and quickly returns. Therefore the depth of the speculated functions is going to vary.

In Figure 8.4 the number of threads executing varies greatly. This is a result of nested speculations; each speculated function executes a small number of bytecode instructions, but they are idle while waiting for their child threads to commit back. This argues for an adaptive approach to find a suitable limit to the checkpoint depth. In addition, a fixed checkpoint depth limit does not take

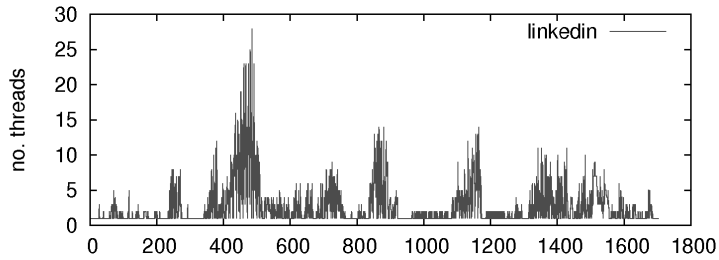

Figure 8.4: The maximum number of threads for various points during the execution for *linkedin*.

rollbacks into account. Since a rollback is often followed by new rollbacks we use a moving median m, as seen in Listing 8.1. JavaScript in web applications is restricted to a single call to the JavaScript engine, so we do not increase m when we speculate successfully.

8.5 Experimental Methodology

We have modified the TLS implementation from Martinsen et al. [63] so we control whether we store a checkpoint or not. The execution behavior of a web application is dependent not only on the JavaScript isolated, but also on the interaction between JavaScript and the web browser such as manipulation of the DOM tree, but we deliberately focus on the JavaScript execution time.

We have selected 15 web applications from the Alexa list [4] of most visited web applications. The experiments are made on a computer running Ubuntu 10.04 equipped with 2 quadcore, Xeon® 2Ghz processors with 4MB cache each, i.e., in total 8 cores (without hyper-threading), and with 16 GB main memory.

8.6 Results of fixed checkpoint depths

The main results of limiting the checkpoint depth are that we are able to reduce the memory usage, and even in certain cases have a higher speedup. We evaluate the effects of storing the checkpoint up to the checkpoint depth limits 1, 2, 4, 8, and when we set no limitation on the checkpoint depth limit, both in terms of execution time, memory usage for speculation, rollbacks, number of threads, and number of speculations.

8.6.1 Improved execution time

Increasing the checkpoint depth does not decrease the execution time. In Figure 8.5, the highest speedup for 11 of the 15 use cases is when we limit the checkpoint depth to either 2, 4 or 8. Without a limit to the checkpoint depth is the fastest for 3 out of 15 cases. If we limit the checkpoint depth to 2, it is the fastest for 4 out of 15 cases, if we limit the checkpoint depth to 4 or 8, it is the fastest for 6 out of 15 (for the cases *Wikipedia* and *Blogspot* the maximum checkpoint depth is 4, therefore the behaviour is identical when we limit the checkpoint depth to either 4 or 8).

The overhead of TLS is increasing with an increased limit on the checkpoint depth, and the potential for finding functions to speculate on decreases as the checkpoint depth increases over 4. This follows the JavaScript execution model in web applications, where we are limited by a certain amount of time for each JavaScript call.

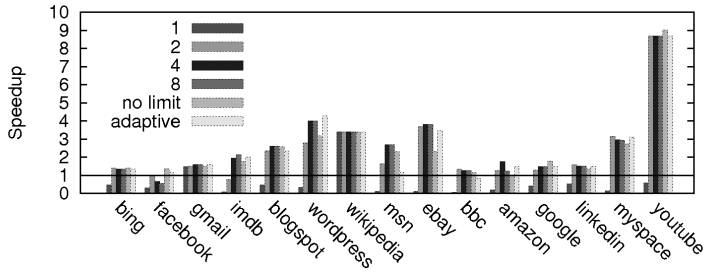

Figure 8.5: The speedup when we limit the checkpoint depth to 1, 2, 4, 8 and put no restriction on the checkpoint depth and the speedup of the adaptive heuristic.

When we limit the checkpoint depth to 2, it is on average 2.39 times faster than the sequential execution time. When we limit the checkpoint depth to 4, it is 2.64 times faster and when we limit the checkpoint depth to 8, it is 2.61 times faster. When we do not limit the checkpoint depth, we see that it is on average 2.45 times faster than the sequential execution time.

When we set the checkpoint depth limit to 2, it is on average 2% slower than when we do not limit the checkpoint depth limit, but uses only 65% of the memory. When we set the checkpoint depth limit to 4 or 8, it is 7% and 6% faster and uses 83% and 97% of the memory.

In Figure 8.5 both *Wikipedia* and *Gmail* are faster for a checkpoint depth limit = 1. *Wikipedia* has no rollbacks, and compared to the other cases, a small number of JavaScript bytecode instructions that are executed with for instance

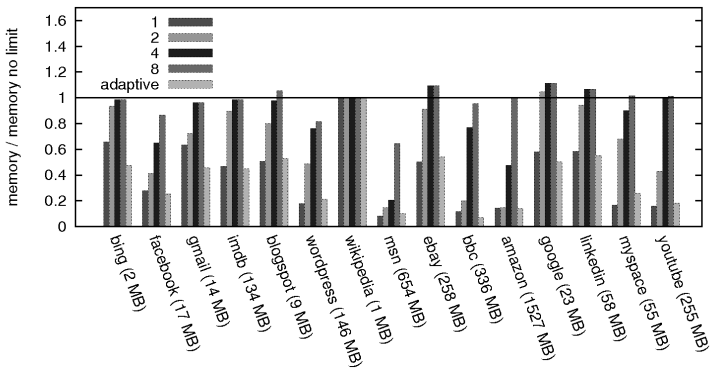

Figure 8.6: The memory usage when we limit the checkpoint depth to 1, 2, 4, 8 and for the adaptive heuristics relative to when we set not checkpoint depth.

12 speculation versus 12012 for *MSN*. Therefore, we do not see an increased execution time with rollbacks, as there are none, independent of what checkpoint depth limit we set. Further, we do not get a significant speedup, since the number of bytecode instructions and the number of functions to speculate on are much lower.

Gmail has 40 threads executing at checkpoint depth 1 and 32 threads executing at checkpoint depth 2. If we count the number of rollbacks, we see that there are 11 rollbacks when we set the checkpoint depth limit to 1, and 17 rollbacks when we set the checkpoint depth limit to 2. Still the execution time is 2% faster setting the checkpoint depth limit = 2, than when checkpoint depth limit = 1. This appears counterintuitive, since there is a larger number of threads and a lower number of rollbacks, so it should be able to exploit running more JavaScript functions in parallel than for checkpoint depth limit=2 and therefore should be faster. However, the cost of doing rollbacks is much higher for checkpoint depth limit=1, than it is for checkpoint depth limit=2. So the effect of having a larger number of threads for checkpoint depth 1 does not outweigh the cost of doing rollbacks, therefore it is slower than checkpoint depth limit=2, but because of the large number of threads and a lower number of rollbacks, it still faster than sequential execution.

In Figure 8.5 the gain in improved execution time is marginal if we limit the checkpoint depth to 8 instead of 2 or 4. This is in line with the results of Martinsen et al. [61]. Most of the JavaScript function calls have a depth of 2 and 4. There is a limit to the amount of time JavaScript is allowed to execute for each event in the web application. Therefore, as the depth of a function

increases, the number of executing bytecode instruction decreases. This explains why there isn't a large increase in the cost of doing rollbacks, when we rollback from a checkpoint depth larger than 4 and that the relative increase in execution time decreases as the depth increases.

The highest speedup is at checkpoint depth limit=4. Then the speedup is gradually reduced to checkpoint depth limit=8, and further reduced when no checkpoint depth limit is set. Still it is faster than the sequential execution time. This shows that the overhead of TLS increases when we increase the checkpoint depth limit, and the gain in terms of more functions to speculative execute decreases with an increased depth.

For *Facebook* the execution time improves for checkpoint depth limit = 2, then for checkpoint limit = 4 it is slower than the sequential execution time. If we compare the number of rollbacks with the number of executed bytecode instructions, this gradually decreases to checkpoint depth limit=8. From the observation of the execution time when we set no limit to the amount of checkpoint stores, this shows that the number of executed bytecode instructions decreases, and that we at some point are able to have a lower execution time such that it is lower than the sequential one.

At checkpoint depth limit=2 the cost of the rollbacks is increasing so that it is 2% slower than when no limit is set. When we set a checkpoint depth limit=1, it is slower than the sequential execution time.

In Figure 8.7 the number of executed bytecode instructions is, as long as there are rollbacks, always higher with TLS. We see that the number of executed bytecodes increases when the checkpoint depth limit decreases which shows that the costs to do rollbacks and to do re-execution increase.

In the lower part in Figure 8.7, we see that the number of rollbacks increases, as the checkpoint depth limit increases. The cost of doing a rollback decreases as the checkpoint depth increases, even though the number of rollbacks decreases. This is because the amount of bytecode instructions re-executed will be lower, even though there are more rollbacks. Therefore we reduce the memory usage, and have a higher speedup, if we are more restrictive on the checkpoint depth since the number of re-executed bytecode instructions will be smaller.

The highest speedup is at checkpoint depth limit = 4. Then the speedup is gradually lowered going to checkpoint depth limit = 8, and further lowered when no limit on the checkpoint depth is set. Still it is faster than the sequential execution time. This indicates that the overhead of TLS increases when we

increase the checkpoint depth limit, and the gain in terms of potential more functions to execute from a higher checkpoint depth limit is lowered.

For *Facebook* we see that the execution time improves for checkpoint depth limit = 2, then for checkpoint depth limit = 4 the execution time decreases to below the sequential execution time. If we compare the number of rollbacks with the total number of executed bytecode instructions, this number gradually decreases toward checkpoint depth limit=8. This shows that the number of executed bytecode instructions decreases, and that we are able to have a lower execution time.

At checkpoint depth limit = 2 the cost of the rollbacks in terms of executed bytecode instructions is increasing to such an extent that it is 2% slower than when no limit is set on the checkpoint depth. When we set a checkpoint depth limit = 1 it is slower than the sequential execution time.

8.6.2 Reduction in memory usage

Increasing the limit of the checkpoint depth increases the memory usage. In Figure 8.6 we have measured the maximum memory usage for the selected use cases when we limit the checkpoint depth to 1, 2, 4, 8 and when we have no limit on the checkpoint depth. Since the memory usage varies between 1 and 1527MB, we have divided each memory usage with the memory usage when we do not limit the checkpoint depth. We have written the memory usage for TLS in Martinsen et al. when we do no limit on the checkpoint depth in parenthesis in Figure 8.1.

Figure 8.6 shows that the memory usage is increasing as we increase the checkpoint depth limit. The reason is that we are saving more states in case of rollbacks. When we are limiting the checkpoint depth to 2, we reduce the memory usage of TLS with 65%. When we limit the checkpoint depths to 4 and 8, we reduce the memory usage with 14% and 3% respectively.

For *Linkedin*, *Blogspot*, *Google*, *Ebay*, *YouTube* and *myspace*, the memory usage is higher with a checkpoint depth limit = 8, than when we do not limit the checkpoint depth. There are two operations in TLS which reduce the memory usage; when we rollback on mis-speculations and when we commit a function back to its parent thread when a function completes execution.

In *Linkedin* we have almost the same number of threads and speculations; however when we do not limit the checkpoint depth, the number of rollbacks

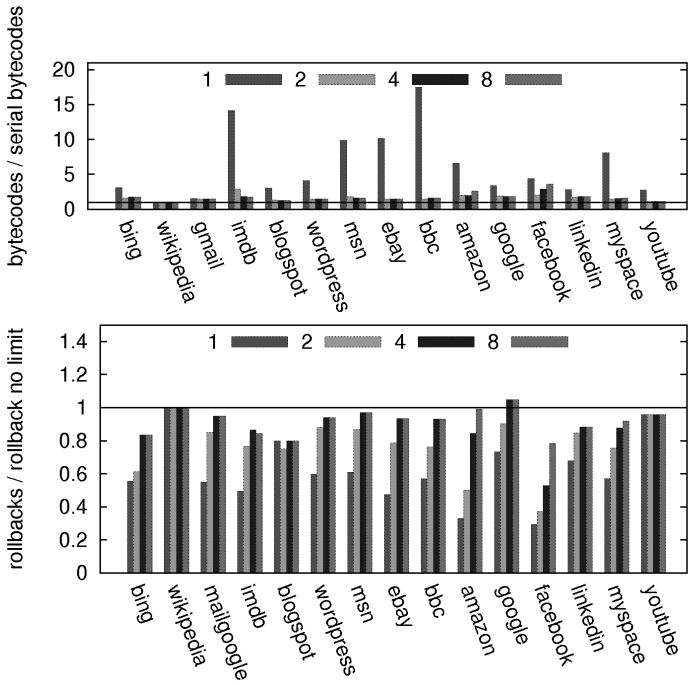

Figure 8.7: The number of executed bytecode instructions in Thread-Level Speculation relative to the number of sequentially executed bytecode instructions (upper) and the number of rollbacks relative to the number of rollbacks when we do not limit the checkpoint depth (lower). A special case in the Figure is the *wikipedia* case, were there are no rollbacks, so the number of executed bytecode instructions are the same for TLS and for the sequential execution.

becomes much higher, therefore we are able to reduce more memory than when we limit the checkpoint depth to 8.

Blogspot and *Ebay* almost have the same number of speculations, but without a limit on the checkpoint depth we get a larger number of threads and rollbacks. Then we reduce the memory both from rollbacks and when we commit values to parents' threads.

For *Myspace* and *Google*, we have a larger number of speculations than when we limit the checkpoint depth to 8, while the number of threads and rollbacks are the same, which shows that we have more rollbacks and threads relative to the number of speculations, which reduces the memory.

For *YouTube* we have the same number of threads, fewer speculations, but a huge increase in the number of rollbacks. We free more memory on rollbacks, and therefore have a lower maximum memory usage.

For these 6 cases, the memory usage is larger for a checkpoint depth of 8 than when no checkpoint depth is set. If we rollback to a different checkpoint depth, we may find other speculation possibilities, which may use more memory as we speculate differently.

The main observations from these measurements are; We are able to improve the execution time by 7% by limiting the checkpoint depth over when we do not limit the checkpoint depth. We are also able to reduce the memory usage by 65%. This shows that the effect of not limiting the checkpoint depth in terms of execution time is limited, but that not limiting the checkpoint depth requires a large amount of memory.

8.7 Results of the adaptive heuristic

The key idea of the heuristic is to select the checkpoint depth limit in relation to already executing threads. The adaptive heuristic significantly reduces the memory usage for TLS, and gives an execution time that is close to the execution time when we set no limit on the checkpoint depth. Since we are using nested speculation, there might be a large number of threads already executing when we speculate. When a rollback occurs, we try to select a checkpoint depth in the speculation tree such that the number of bytecode instructions in case of rollbacks in the future will be lowered.

8.7.1 Improved execution time

Figure 8.5 shows that we are able to increase the speedup of JavaScript execution from 1.14 – 8.69 times faster (*MSN* and *YouTube*), and have a speedup which more than half the time is faster than when we do not set any limitation on the checkpoint depth, and always better than the sequential execution time (except for *BBC*). One example of the effect of the heuristic is the *YouTube* web application; when no limit is set on the checkpoint depth, it uses 255 MB and is over 8 times faster. With the heuristic it only uses 44 MB (a reduction of 83%) while it is only 4% slower compared to when no limit is set on the checkpoint.

Overall, the heuristic makes us select a checkpoint depth such that there is a low number of bytecode instructions to re-execute if there will be a rollback. This can be seen from measuring the number of bytecode instructions that is re-executed at a rollback. The number of rollbacks is higher with the heuristic than for checkpoint depth limit = 1, but each rollback requires less bytecode instructions to be re-executed with the heuristic. We also see that if there will be a rollback, and there are a large number of threads executing, we are bound to rollback to the parents speculative functions, which are nearby, which in the future makes rollbacks require less bytecode instructions to re-execute. If the depth is lower than 3, we choose to save the state to 1, which we saw was costly for checkpoint depth 1, but at the same time, if the number of executing threads are low Martinsen et al. show that it is not very likely to have a rollback.

Figure 8.8 shows the checkpoint depths in the only example which is slower than the sequential execution with the heuristic. The memory usage is 93% of the memory usage when no checkpoint depth is set. This is caused by the heuristic rolling back to a checkpoint depth which is close to 1, repeatedly. This significantly reduces the memory usage, but increases the cost of rollbacks as it forces us to re-execute many bytecode instructions.

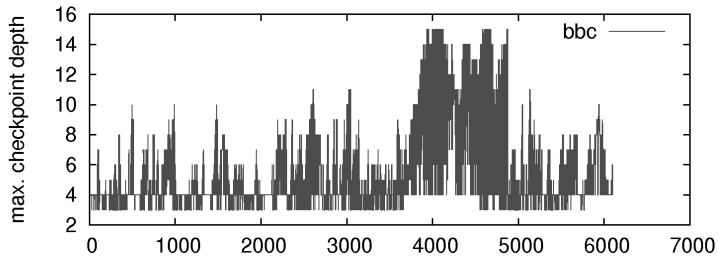

Figure 8.8: The selected checkpoint depths during execution of the *bbc* use-case

When a small number of threads are executing, we do not really need to store the checkpoint. If a rollback occurs, the amount of bytecode instructions we have to re-execute will be small. This has two consequences; first, given the limitations of allowed execution time in JavaScript in web applications, the functions that are executing are at this point quite large. The next consequence is, if there will be a rollback in these, the number of bytecode instructions for the re-execution is limited. Therefore the heuristic reduces the execution time.

8.7.2 Reduction in memory usage

Figure 8.6 shows that we reduce the memory usage between 93% – 45% (*BBC* and *Linkedin*). One exception is *Wikipedia*, but this use case does not have any rollbacks, little JavaScript execution, and a low number of JavaScript function calls, which limits the potential gain from speculations.

The heuristic is able to reduce the memory usage below when we set the checkpoint depth limit to 1. In Figure 8.6, for 9 out of the 15 use cases, the memory usage is lower with the heuristic than when we set the checkpoint depth limit to 1.

The heuristic has a higher number of rollbacks and a higher number of speculations. This explains why the memory usage is lower, both with more rollbacks with small number of bytecode instructions that needs to re-executed and more commits. Since we are using nested speculation, a speculated function could be created from a function executing speculativly. We could imagine the functions executing in a tree like structure, similar to the one in Figure 8.3. When we do a rollback, we would like to rollback to a state, which is part of the speculation tree, to reduce the number of bytecode instructions that needs to be re-executed. With a checkpoint of 1, we often re-execute bytecode instructions which are not part of the speculation tree, but with a moving median we might not.

When the number of already executing threads is below 3, there will probably be no rollback; therefore we set the checkpoint depth to 2. This means that we in many cases do not save the state when there is a low number of threads, and therefore we save memory, which leads to a memory usage similar to checkpoint depth 1. So when we stay in the speculation tree, we have a higher number of rollbacks (we saw this from the increase in rollbacks when we increased the speculation depth) and we are careful where we store the checkpoint when there are few active threads. This indicates that the number of threads already executing when we are about to speculate on a function call, is highly dynamic, since the amount of execution performed by each JavaScript function is small. This means that the value of the checkpoint depth needs to be dynamically set. When there are a small number of threads already executing, there is not really a need to save the checkpoint. If there is a large number of threads, there is likely going to be many threads with different depths, and in that case, make sure that the checkpoint is near so you rollback in the speculation tree.

The results of this heuristic, is that we are able to significantly improve the execution time, while reducing the memory usage by over 90% by adaptively selecting at what depth we are storing checkpoints.

8.8 Conclusions

We have (i) proposed to reduce the memory usage of Thread-Level Speculation in JavaScript virtual machines by only storing states up to certain limited checkpoint depths, and (ii) we proposed and evaluated an adaptive heuristic to dynamically adjust the checkpoint depth.

Our results show that we do not need to save the state each time we speculatively execute a function. As a result, we can reduce the amount of memory used for speculation. However, since nested speculation has been shown to be necessary, we need to save states on at least checkpoint depth 2 in order to improve the execution time, or it will be too expensive to do the necessary rollbacks.

Further, our results show that our proposed adaptive heuristic reduces the memory usage significantly, over 90% as compared to when no checkpoint limit is set. The execution time of our adaptive heuristic is approximately the same as when using no checkpoint limit, sometimes it is slightly (4%) slower but it has also been shown to be over 50% faster.

Chapter 9

Paper VIII

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

Jan Kasper Martinsen, Håkan Grahn, and Anders Isberg

Submitted to journal for publication

9.1 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 [30, 108, 72]. 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 [56, 84, 89]. 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 [58].

Fortuna et al. [25] 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) [91].

A lightweight speculation mechanism for JavaScript that focuses on loop-like constructs is introduced [66], and gives speedups by a factor of 2.8. Another approach increases the responsiveness of web applications [68].

Our goal is to reduce the JavaScript execution time by dynamically extracting parallelism. In [63], 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 [58] show that even though the workload in web applications often is not suitable for JIT, Figure 9.1 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 9.2 we introduce JavaScript and web applications, as well as principles and previous work on Thread-Level Speculation. In Section 9.3, we present our implementation of Thread-Level Speculation in the V8 JavaScript engine. In Section 9.4 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 9.5 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 9.6 we conclude our findings.

9.2 Background and related work

9.2.1 JavaScript and web applications

JavaScript [37] 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 [88].

The performance of popular JavaScript engines such as Google's V8 engine [30], WebKit's Squirrelfish [108], and Mozilla's SpiderMonkey and TraceMonkey [72] 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 [56, 84, 89] for real life web applications, and that optimizing towards the characteristics of the benchmarks may increase the execution time in web applications [58, 63].

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 [56, 84, 89]. 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.

9.2.2 Thread-level speculation principles

TLS aims to dynamically extract parallelism from a sequential program. This has been done both in hardware, e.g., [14], and in software, e.g., [81, 91].

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.

9.2.3 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 9.1 where they majority of the solutions support either C, FORTRAN or Java.

9.2.4 General software-based Thread-Level Speculation

9.2.5 TLS in JavaScript and web applications

Mehrara and Mahlke [67] 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.

Table 9.1: Examples of previous work on software Thread-Level Speculation.

Author	Speedup	#cores	Benchmark	Type	Language
Chen and Olukotun show [15, 16]	3.7–7.8	16	SPECjvm98 [95] (with others)	method-level	C and JAVA
Bruening et al. [13]	over 5	8	SPEC95FP and SPEC92	applicable on while loops	C
Rundberg and Stenström [91]	10	16	Perfect Club Benchmarks [8]	minimizing the overhead	C
Kazi and Lilja [41]	5–7	8	Perfect Club Benchmarks [8]	exploit coarse-grained parallelism	C
Bhowmik and Franklin [9]	1.64–5.77	6	SPEC CPU95, SPEC CPU2000, and Olden	exploits loop and non-loop parallism	C
Cintra and Llanos [17]	16	16	SPEC CPU2000 [96] and Perfect Club [8]	using a sliding window with loops	C
Renau et al. [86]	3.7–7.8	16	SPEC CPU2000 Benchmarks [96]	method-level	C
Picket and Verbrugge [80, 81]	2	4	SPECjvm98 [95]	method-level	JAVA
Kejariwal et al. [42]	2.5		SPEC CPU2000 Benchmarks [96]	Theoretical study	C and FORTRAN
Hertzberg and Olukotun [34]	2.04	4	SPEC CPU2000 Benchmarks [96]	method-level	C
Hertzberg and Olukotun [34]	3–4, 2–3, and 1.5–2.5	4	SPEC CPU2006 Benchmarks [97]	method-level	JAVA

In [66], 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.

However, large loop structures are rare in real web applications [58]. Instead, there are often a large number of function calls caused by events from the web application.

Unlike [66], our approach is to execute the main execution flow in parallel [63]. We implemented TLS in the Squirrelfish JavaScript engine which is part of We-

bKit [108], 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 9.1. 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.

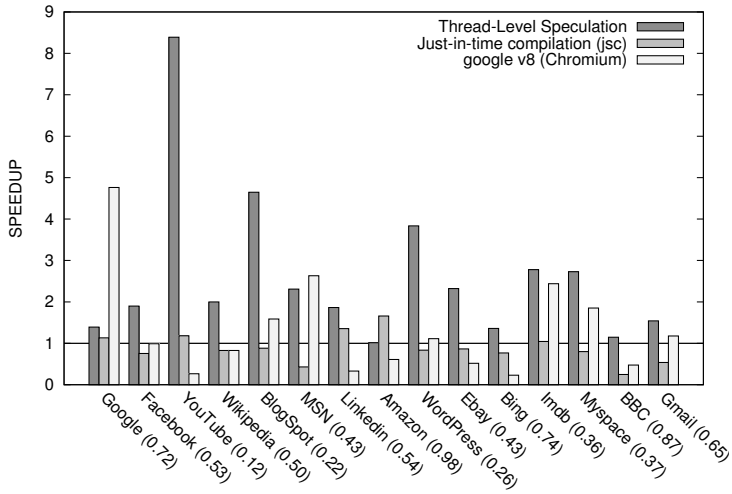

Figure 9.1: Speedup of Thread-Level Speculation and just-in-time compilation for a number of popular web applications. The black horizontal line is the sequential execution time of Squirrelfish without Thread-Level Speculation.

In [64] 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 [63], 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 [61]. Our results show that it is possible to reduce the 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.

9.3 Implementation of TLS in V8

9.3.1 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 9.2 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 [56]; 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 9.2, 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).

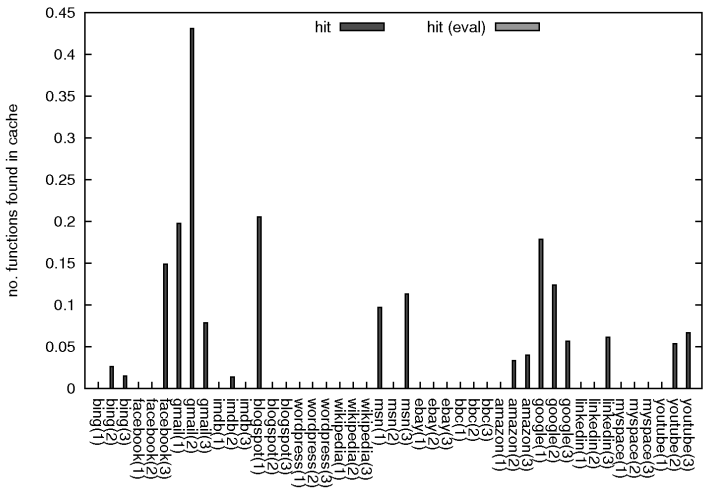

Figure 9.2: The relative number of times a function is found in the cache of already compiled functions.

In Figure 9.3 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 9.5 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

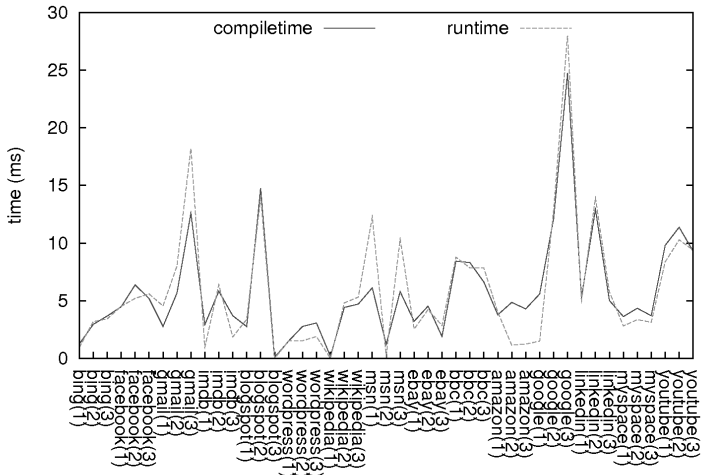

Figure 9.3: The time spent on compiling the various functions versus the time spent executing the functions.

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.

9.3.2 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 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. 9.4). 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).

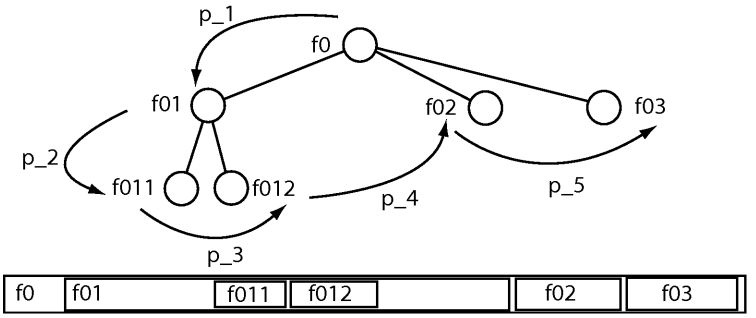

Figure 9.4: We use the order that the functions are called in, to determine the order in which the program would have been sequentially executed. This works in JavaScript in a web applications setting, as there are multiple function calls. For instance, in a simplified example the JavaScript function 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*.

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.

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 [63].

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 [63] where we speculate aggressively on function calls.

Like the implementation in [63] 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 [36].

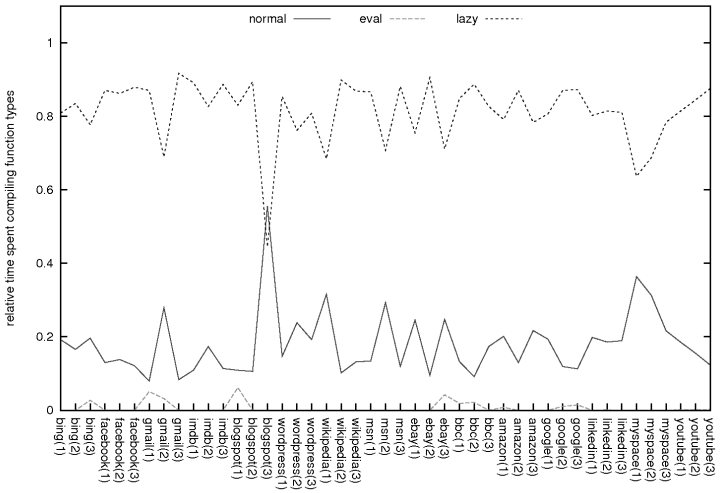

Figure 9.5: The relative time spent compiling the various function types. Normal functions are compiled when the web page is loaded, eval is compiled when the JavaScript code is executed using an `eval()` call, and lazy functions are compiled when they are about to be executed.

9.4 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 9.6). 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 9.2.

We have selected the web applications from the Alexia list [4] 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

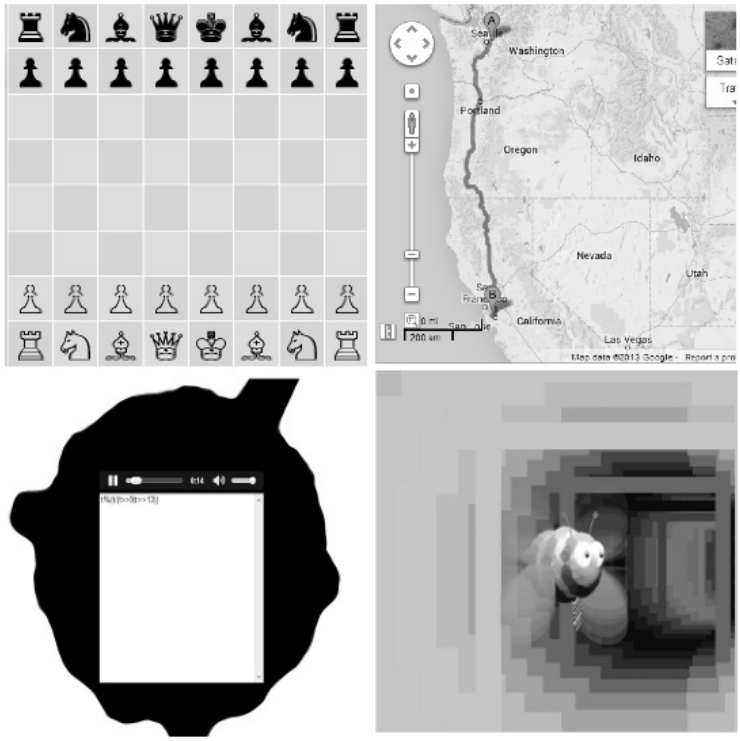

Figure 9.6: The roadmap generated in Google Maps between the Google Headquarter in Paolo Alto to the Microsoft headquarter in Redmond, Seattle (top right) and screenshot of 3 random entries (within the top 10) from the JS1K competition in 2010 and 2013.

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.

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 [30] JavaScript engine, and the use cases have be executed within Chromium [31] from Google. From [63] and in Figure 9.5 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 [11] 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 [56]. Since we reduce the interaction, such as the mouse gestures, this allows our results to be more applicable to other platforms as well (such as laptops of different screensizes, smartphones, game consoles etc).

9.5 Experimental results

9.5.1 The effects of TLS on 15 web applications

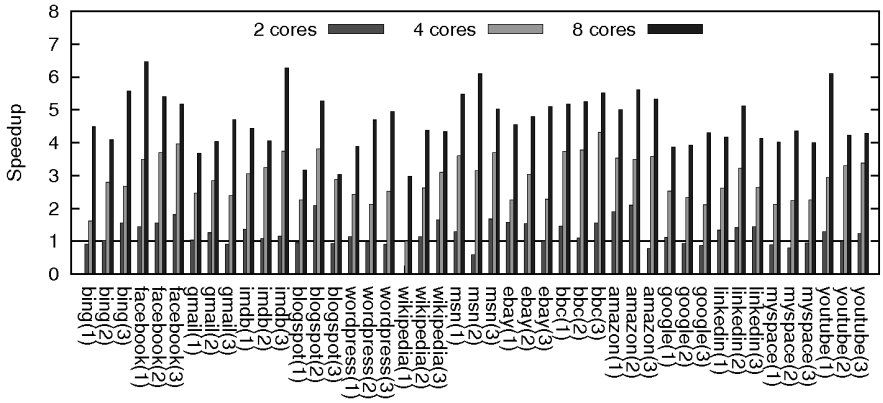

Figure 9.7: The speedup for 2, 4 and 8 cores on 45 use cases in 15 web applications.

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

In Figure 9.7 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.

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 9.12 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 9.2, 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 9.11. 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 9.12 to consist of a small number of executing threads. This can be understood by that there is little interaction in these use cases (Table 9.2) 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.

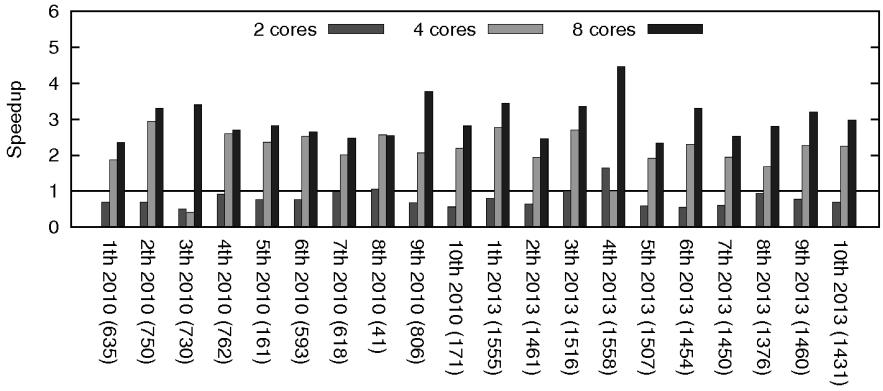

Figure 9.8: The speedup for 2, 4 and 8 cores on the set of top 10 entries of JS1K for 2010 and 2013.

9.5.2 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 9.8 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 *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.

9.5.3 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 9.9 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.

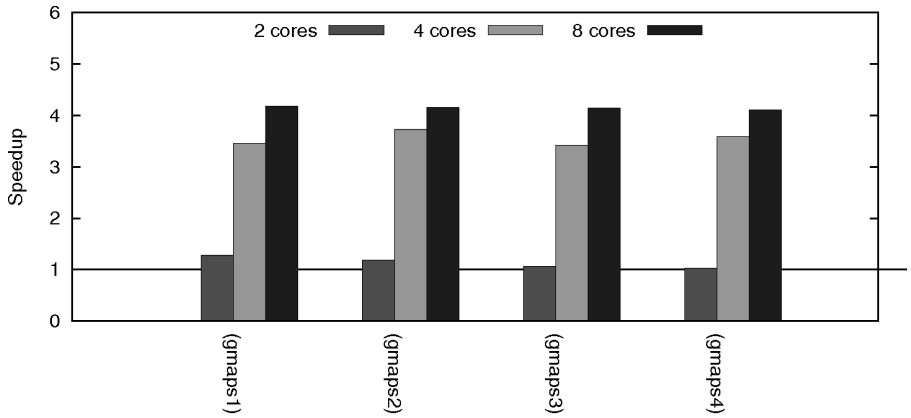

Figure 9.9: The speedup for 2, 4 and 8 cores on the set of the 4 use cases for Google maps.

Since the potentials of TLS in Google maps (Figure 9.9) 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 9.4 we have no user interaction of moving the map in Google maps to make the use cases easier to reproduce. This 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 9.8, these web applications have a lower speedup with TLS than the 15 web application in Figure 9.7 and the 4 use cases with Google maps in Figure 9.9. 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 9.10 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.

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.

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 9.11 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.

In Figure 9.12 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

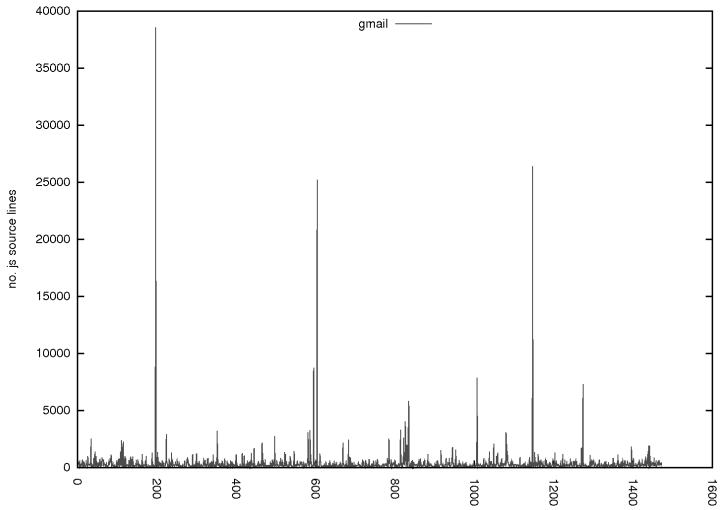

Figure 9.10: The number of lines of JavaScript code for the various calls to the compiler in V8 for the web application *Gmail*.

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 [63] 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 9.7 this creates one of the largest speedups which are over 6 times faster than the sequential execution time of V8.

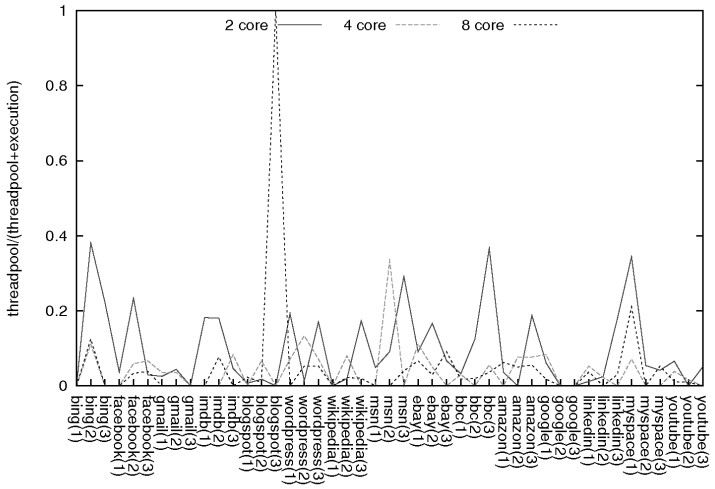

Figure 9.11: The thread pool initialization time as propotion of the total execution time.

9.6 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.

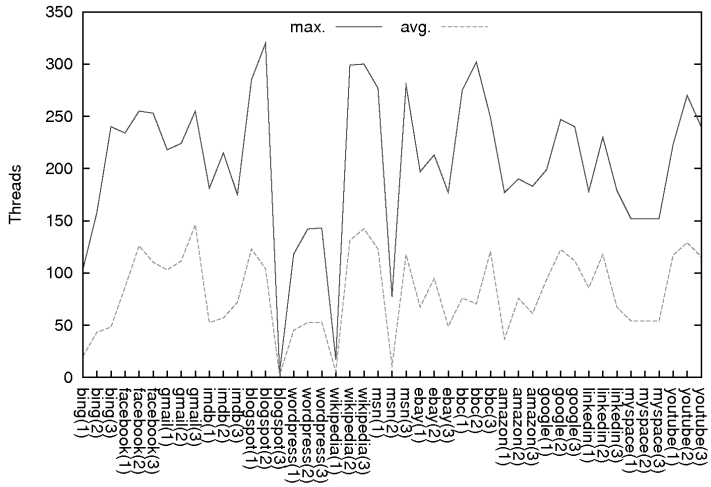

Figure 9.12: The maximum and the average number of threads during execution.

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.

Table 9.2: 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.

Chapter 10

Paper IX

Performance Enhancement and Power Usage of Thread-Level Speculation on Web Applications for Embedded Devices with the V8 JavaScript Engine

Jan Kasper Martinsen, Håkan Grahn, Samir Drincic, and Anders Isberg

Submitted to journal for publication

10.1 Introduction

JavaScript is a dynamically typed, object-based scripting language with runtime evaluation, where the execution is done in a JavaScript engine [30, 108, 72]. Fortuna et al. [25] show that there is a large potential for parallism for JavaScript in web applications with speedups up to 45 times compared to the sequential execution time. However, JavaScript is a sequential programming language.

One transparent manner to take advantage of parallism with parallel hardware is Software Thread-Level Speculation (TLS) [91] which have been demonstrated both for JavaScript benchmarks [66] and web applications [63].

In [63] we presented a method-level TLS implementation in Squirrelfish/WebKit with an on/off speculation principle in the Squirrelfish JavaScript engine without Just-in-time compilation, where a single misspeculation turns off the speculation for that function. We later extended this technique with a heuristic which adapts how aggressively we speculate [48].

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

10.2 Implementation of TLS in Google's V8 JavaScript engine

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

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

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

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

We extract the *realtime* value and the id of the thread that makes this call, e.g., *p_0220* (*p_0* calls a function after 220 bytecode instructions). We denote the value of the position of this function call as *function_order*, which emulates

the *sequential time* in our program (Fig. 10.1). We check if this function previously has been speculated on by looking up the value of *previous[function_ order]*. *previous* is a vector where each entry is organized by the *function_ order*.

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

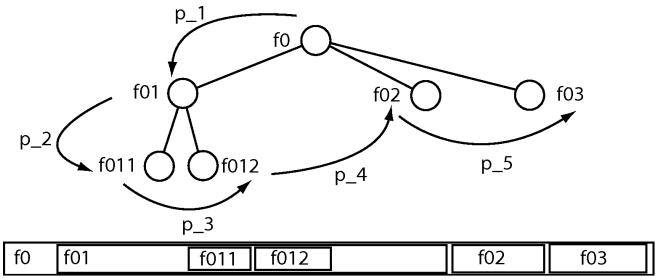

Figure 10.1: We use the order that the functions are called in, to determine the order in which the program would have been sequentially executed. This works in JavaScript in a web applications setting, as there are multiple function calls. For instance, in a simplified example the JavaScript function f0, performs 3 function calls, f01, f02 and f03. f01 performs two function calls, f011 and f012. Thus, we have created a speculation tree from the function calls. If we traverse this tree from left to right, we get an order in which the functions are called, equal to the order in which the functions would be sequentially called, in order to uphold the sequential semantics during execution with TLS. More specific: f0 at time p_1, f01 at time p_2, f011 at time p_3, f012 at time p_4, f02 at time p_5, and f03 at time p_6. We denote how each function is ordered as *function_ order*

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

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

Now we have two concurrent threads, and this process is repeated each time a suitable speculation candidate is encountered, thereby supporting nested specula-

tion. If there is a conflict between two global variables or an incorrect return value prediction we perform a rollback to the checkpoint point where the speculation started.

In V8, the actual JavaScript global variables are not compiled into the code that are to be executed, instead these variables are accessed from function calls in the compiled code, such as *StoreIC_Initialize* and *LoadIC_Initialize*. This means, that the global JavaScript stack, and the native code are accessed separately. This further means, that we only need to save part of the JavaScript stack, when we are about to speculate. We therefore save the stack in case of a rollback. Restoration of the state, if we encounter a rollback is not very costly, as we do not need to re-compile the speculated function call. In fact the function call that is to be re-executed on a rollback is already ready for execution inside the cache. However, rollbacks are relatively rare, as we see in [63].

Like the implementation in [63] we support nested speculation, and therefore need to store states of the nested speculation, in case of a rollback. Like previously seen, JavaScript function calls rarely return any value in a web application, therefore we use a return value prediction scheme, like the *"last predicted value"* found in [36].

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

The reuse of compiled functions opens up possibilities when we speculate and need to re-execute functions in case of a rollback. Since we do not add features to the native code, and since all JavaScript global objects are accessed through external functions, we do not need to re-compile the code upon rollbacks in TLS. The result is that the cost of doing rollbacks and re-executing functions decreases in terms of execution time, when we are using JIT as compared to an interpreted JavaScript engine such as Squirrelfish. This suggests that we could significantly improve the execution time for TLS, as we reduce the costs of rollbacks. For the implementation of TLS speculation in V8, we follow what is done in [63] where we speculate aggressively on function calls.

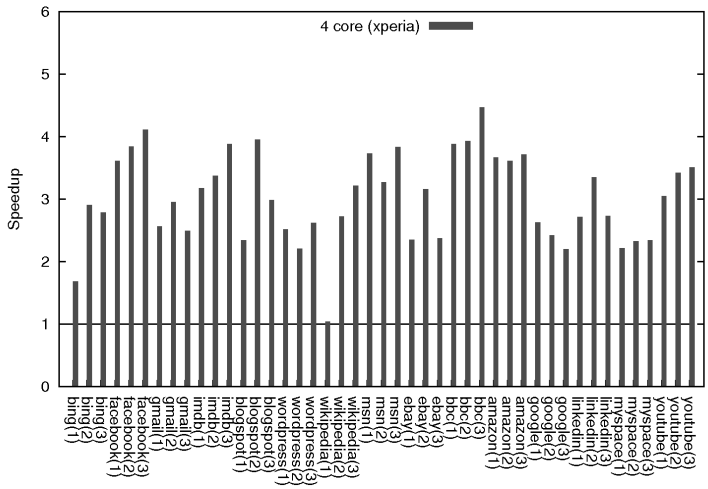

Figure 10.2: The execution time for 45 use-cases from very popular web applications.

10.3 Experimental Methodology

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

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

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

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

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

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

10.4 Experimental Results

10.4.1 Improved execution time

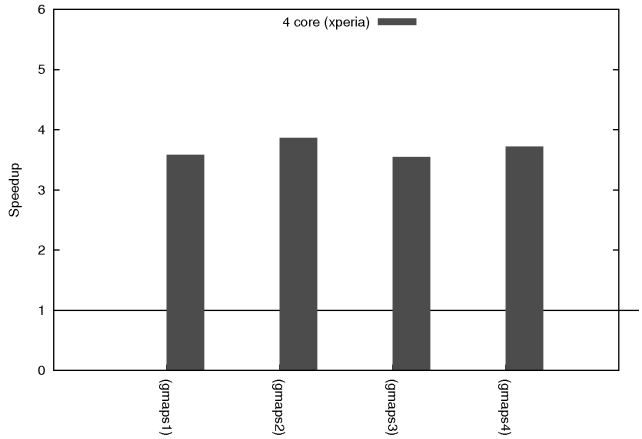

Figure 10.3: The execution time for 4 google maps use-cases.

In Fig. 10.2, Fig. 10.3 and Fig. 10.4 we see that TLS almost always improves the execution time for JavaScript in web applications even though the application types are very different from one another. We notice that the effects of TLS, in terms of reduction of execution time is increasing with more interaction (for instance in Fig. 10.2, *Facebook(3)* executes faster than *Facebook(2)* and *Facebook(2)* executed faster than *Facebook(1)*). This indicate that as we eleborate the use cases with more interaction, the effects of TLS becomes more apparent. The relatively small increase in execution time between the use cases also indicates that we could have an even better speedup with a larger number of cores.

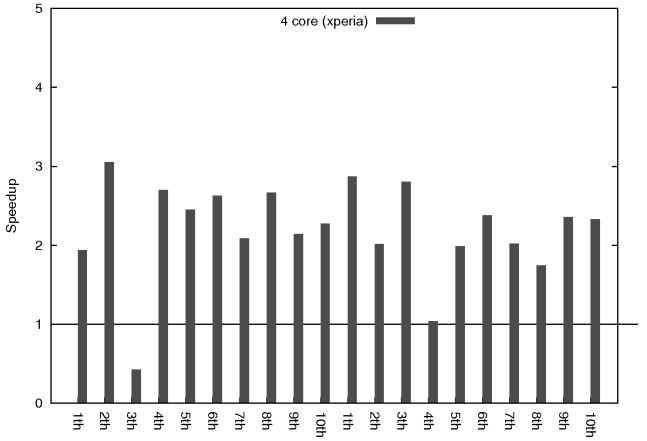

Figure 10.4: The execution time for 20 HTML5 demos

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

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

10.4.2 Power usage

In Fig. 10.5 we have measured the power usage of *Facebook* when running with TLS and when running without TLS. We see from the results that TLS version executes much faster (more than twice as fast), but uses at most twice as much power.

In Fig. 10.6 we have measured the power usage of Google maps when running with TLS and when running without TLS. We see from the results that the TLS version executes much faster (roughly three times as fast), but uses at most almost twice as much power.

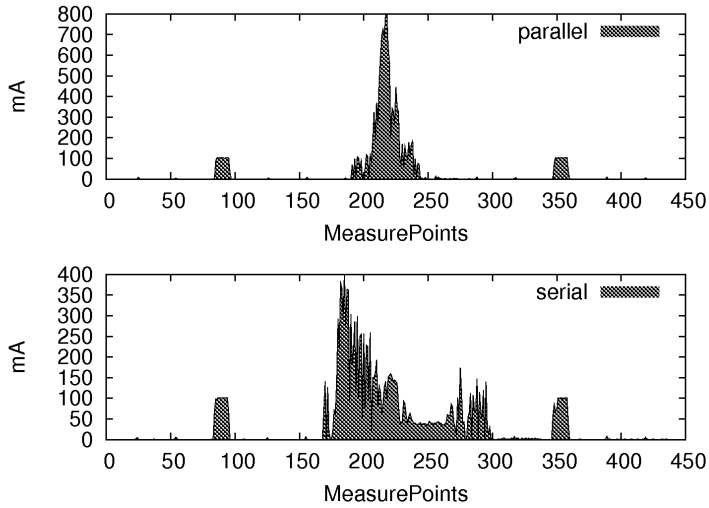

Figure 10.5: The power usage for *Facebook* running on the Sony Xperia Z1 smartphone both in parallel and sequentially

In Fig. 10.7 we have measured the power usage of JS1K when running with TLS and when running without TLS. We see from the results that the TLS version executes slightly faster, and uses the lowest power compared to the sequential execution og the use cases.

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

10.5 Conclusion

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

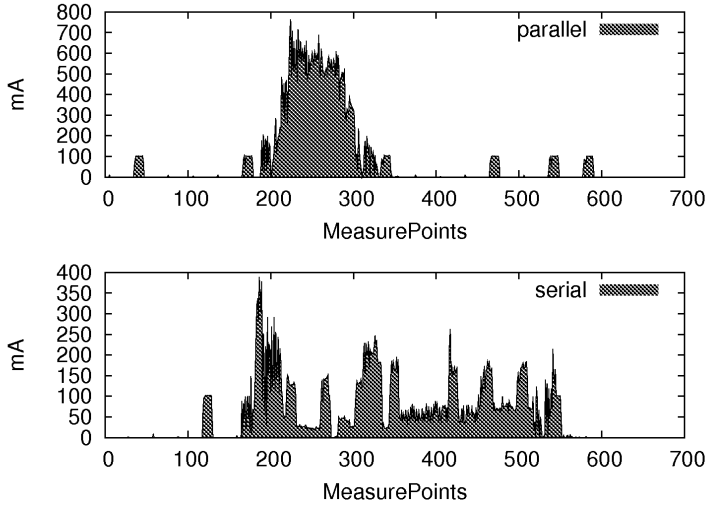

Figure 10.6: The powerusage for Google maps.

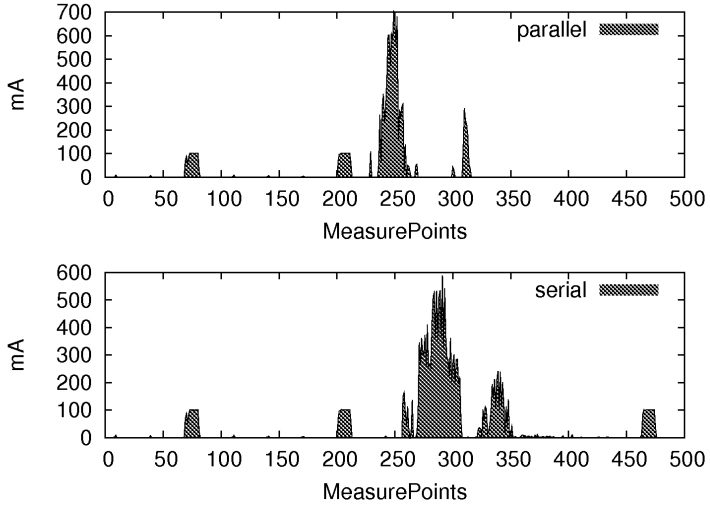

Figure 10.7: The power usage for a HTML5 use-case.

Bibliography

[1] 10KApart. Inspire the web with just 10k, 2010. `http://10k.aneventapart.com/`.

[2] O. Agesen. GC points in a threaded environment. Technical report, Sun Microsystems, Inc., Mountain View, CA, USA, 1998.

[3] T. J. Albert, K. Qian, and X. Fu. Race condition in ajax-based web application. In *ACM-SE 46: Proc. of the 46th Annual Southeast Regional Conf.*, pages 390–393, New York, NY, USA, 2008. ACM.

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

[5] A. A. Andrews, J. Offutt, C. Dyreson, C. J. Mallery, K. Jerath, and R. Alexander. Scalability issues with using FSMWeb to test web applications. *Inf. Softw. Technol.*, 52(1):52–66, 2010.

[6] R. Arjan, U. Pfeil, and P. Zaphiris. Age differences in online social networking. In *CHI '08: CHI '08 extended abstracts on Human factors in computing systems*, pages 2739–2744, New York, NY, USA, 2008. ACM.

[7] K. Asanovic et al. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[8] 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 supercomputers.

Technical Report CSRD-827, Center for Supercomputing Research and Development, Univ. of Illinois, Urbana-Champaign, May 1989.

[9] 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.

[10] Blogger: Create your free blog, 2010. `http://www.blogger.com/`.

[11] 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.

[12] J. Brown. *Using Surveys in Language Programs.* Cambridge Language Teaching Library. Cambridge University Press, 2001.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] T. Cook and D. Campbell. *Quasi-experimentation: design & analysis issues for field settings.* Rand McNally College, 1979.

[19] A. A. Dahlan and T. Nishimura. Implementation of asynchronous predictive fetch to improve the performance of Ajax-enabled web applications. In *IIWAS '08: Proc. of the 10th Int'l Conf. on Information Integration and Web-based Applications & Services*, pages 345–350, New York, NY, USA, 2008. ACM.

[20] O. Danvy, C.-C. Shan, and I. Zerny. J is for javascript: A direct-style correspondence between algol-like languages and javascript using first-class continuations. In *Proceedings of the IFIP TC 2 Working Conference on Domain-Specific Languages*, DSL '09, pages 1–19, Berlin, Heidelberg, 2009. Springer-Verlag.

[21] E. Eldon. Facebook used by the most people within Iceland, Norway, Canada, other cold places, 2009. `http://tinyurl.com/y9kegs8`.

[22] Facebook. Facebook statistics, 2010. `http://www.facebook.com/press/info.php?statistics`.

[23] FireBug. FireBug, JavaScript profiler, 2010. `http://getfirebug.com`.

[24] D. Flanagan. *JavaScript: The Definitive Guide, 5th edition.* O'Reilly Media, 2006.

[25] 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.

[26] A. Gal. TraceMonkey vs V8, 2008. `http://andreasgal.wordpress.com/2008/09/03/tracemonkey-vs-v8/`.

[27] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *PLDI '09: Proc. of the 2009 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 465–478, New York, NY, USA, 2009. ACM.

[28] P. Golle. Machine learning attacks against the Asirra CAPTCHA. In *CCS '08: Proc. of the 15th ACM Conf. on Computer and Communications Security*, pages 535–542, New York, NY, USA, 2008. ACM.

[29] Google. V8 benchmark suite - version 5, 2010. `http://v8.googlecode.com/svn/data/benchmarks/v5/run.html`.

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

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

[32] M. Grady. Functional programming using JavaScript and the HTML5 canvas element. *J. Comput. Small Coll.*, 26:97–105, December 2010.

[33] S. Halladay and M. Wiebel. A practical use for multiple threads. *C Users J.*, 10(1):73–84, Jan. 1992.

[34] 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.

[35] W3C HTML Working Group, 2010. `http://www.w3.org/html/wg/`.

[36] 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.

[37] JavaScript. `http://en.wikipedia.org/wiki/JavaScript`, 2010.

[38] JS1k. This is the website for the 1k JavaScript demo contest #js1k, 2010. `http://js1k.com/home`.

[39] JSBenchmark, 2010. `http://jsbenchmark.celtickane.com/`.

[40] I. H. Kazi and D. J. Lilja. JavaSpMT: A speculative thread pipelining parallelization model for java programs. In *IPDPS'00: Proc. of the 14th Int'l Parallel and Distributed Processing Symp.*, page 559, May 2000.

[41] 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.

[42] 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.

[43] B. Krishnamurthy, P. Gill, and M. Arlitt. A few chirps about twitter. In *WOSP '08: Proc. of the First Workshop on Online Social Networks*, pages 19–24, New York, NY, USA, 2008. ACM.

[44] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proc. of the Int'l Symp. on Code Generation and Optimization*, CGO '04, pages 75–86, 2004.

[45] J. Martinsen. *Evaluating JavaScript Execution Behavior and Improving the Performance of Web Applications with Thread-Level Speculation*. Licentiate thesis, Blekinge Institute of Technology (BTH), Karlskrona, Sweden, Nov. 2011.

[46] J. Martinsen, H. Grahn, S. Drincic, and A. Isberg. Performance and Power Usage of Thread-Level Speculation in the V8 JavaScript Engine. 2014.

[47] J. Martinsen, H. Grahn, and A. Isberg. Heuristics for Thread-Level Speculation in Web Applications. *IEEE Computer Architecture Letters*, pages 1–1, 2013.

[48] J. Martinsen, H. Grahn, and A. Isberg. Heuristics for thread-level speculation in web applications. *Computer Architecture Letters*, PP(99):1–1, 2013.

[49] J. Martinsen, H. Grahn, and A. Isberg. An Argument for Thread-Level Speculation and Just-in-Time Compilation in the Google's V8 JavaScript Engine. In *Conf. Computing Frontiers*, 2014.

[50] J. Martinsen, H. Grahn, and A. Isberg. Combining Thread-Level Speculation and Just-In-Time Compilation in Google's V8 JavaScript Engine. 2014.

[51] J. Martinsen, H. Grahn, and A. Isberg. The Effects of Parameter Tuning in Software Thread-Level Speculation in JavaScript Engines. 2014.

[52] J. Martinsen, H. Grahn, H. Sundström, and A. Isberg. Reducing Memory Usage in Software-Based Thread-Level Speculation for JavaScript in Virtual Machine Execution of Web Applications. 2014.

[53] J. K. Martinsen and H. Grahn. Thread-Level Speculation for Web Applications. In *Second Swedish Workshop on Multi-Core Computing (MCC-09)*, pages 80–88, November 2009.

[54] J. K. Martinsen and H. Grahn. A Comparative Evaluation of the Execution Behavior of JavaScript Benchmarks and Real-World Web Applications(poster presentation). In *Poster Proc. of the 28th International Symposium on Computer Performance, Modeling, Measurements and Evaluation (Performance-2010)*, pages 27–28, 2010.

[55] J. K. Martinsen and H. Grahn. An Alternative Optimization Technique for JavaScript Engines. In *Third Swedish Workshop on Multi-Core Computing (MCC-10)*, pages 155–160, November 2010.

[56] 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.

[57] J. K. Martinsen and H. Grahn. Thread-Level Speculation as an Optimization Technique in Web Applications for Embedded Mobile Devices – Initial Results. In *Proc. of the 6th IEEE Int'l Symposium on Industrial Embedded Systems (SIES'11)*, pages 83–86, June 2011.

[58] 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.

[59] J. K. Martinsen, H. Grahn, and A. Isberg. Evaluating Four Aspects of JavaScript Execution Behavior in Benchmarks and Web Applications. Research Report 2011:03, Blekinge Institute of Technology, July 2011.

[60] J. K. Martinsen, H. Grahn, and A. Isberg. The Effect of Thread-Level Speculation on a Set of Well–known Web Applications. In *Fourth Swedish Workshop on Multi-Core Computing (MCC-11)*, November 2011.

[61] 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.

[62] J. K. Martinsen, H. Grahn, and A. Isberg. Preliminary Results of Combining Thread-Level Speculation and Just-in-Time Compilation in Google's V8. In *Sixth Swedish Workshop on Multi-Core Computing (MCC-13)*, pages 37–40, November 2013.

[63] 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.

[64] 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.

[65] R. McDougall. Extreme software scaling. *Queue*, 3(7):36–46, 2005.

[66] 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.

[67] 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.

[68] 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.

[69] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *IMC '07: Proc. of the 7th ACM SIGCOMM Conf. on Internet Measurement*, pages 29–42, New York, NY, USA, 2007. ACM.

[70] F. Molina, B. Sweeney, T. Willard, and A. Winter. Building cross-browser interfaces for digital libraries with scalable vector graphics (SVG). In *JCDL '07: Proc. of the 7th ACM/IEEE-CS Joint Conf. on Digital libraries*, pages 494–494, New York, NY, USA, 2007. ACM.

[71] Mozilla. Dromaeo: JavaScript performance testing, 2010. `http://dromaeo.com/`.

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

[73] A. Munir, F. Koushanfar, A. Gordon-Ross, and S. Ranka. High-performance optimizations on tiled many-core embedded systems: A matrix multiplication case study. *J. Supercomput.*, 66(1):431–487, Oct. 2013.

[74] A. Nazir, S. Raza, and C.-N. Chuah. Unveiling Facebook: A measurement study of social network based applications. In *IMC '08: Proc. of the 8th ACM SIGCOMM Conf. on Internet Measurement*, pages 43–56, New York, NY, USA, 2008. ACM.

[75] A. Nazir, S. Raza, and C.-N. Chuah. Unveiling Facebook: A measurement study of social network based applications. In *IMC '08: Proc. of the 8th ACM SIGCOMM Conf. on Internet Measurement*, pages 43–56, 2008.

[76] Node.js. Evented I/O for V8 JavaScript, 2010. `http://nodejs.org/`.

[77] G. R. Notess. Alexa: Web archive, advisor, and statistician. *Online*, 22(3):29–30, 1998.

[78] 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*, pages 223–232, August 2009.

[79] U. Pfeil, R. Arjan, and P. Zaphiris. Age differences in online social networking - a study of user profiles and the social capital divide among teenagers and older users in MySpace. *Comput. Hum. Behav.*, 25(3):643–654, 2009.

[80] 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.

[81] 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.

[82] M. K. Prabhu and K. Olukotun. Exposing speculative thread parallelism in SPEC2000. In *Proc. of the 10th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 142–152, 2005.

[83] P. Ratanaworabhan, B. Livshits, D. Simmons, and B. Zorn. JSMeter: Characterizing real-world behavior of JavaScript programs. Technical Report MSR-TR-2009-173, Microsoft Research, December 2009. `http://research.microsoft.com/en-us/projects/jsmeter/`.

[84] 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.

[85] J. Renau, K. Strauss, L. Ceze, W. Liu, S. R. Sarangi, J. Tuck, and J. Torrellas. Energy-efficient thread-level speculation. *IEEE Micro*, 26(1):80–91, 2006.

[86] 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.

[87] J. Resig. `http://ejohn.org/blog/tracemonkey/`, 2008.

[88] 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.

[89] 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.

[90] C. Robson. *Real World Research - A Resource for Social Scientists and Practitioner-Researchers*. Blackwell Publishing, Malden, second edition, 2002.

[91] 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.

[92] C. Sadowski, T. Ball, J. Bishop, S. Burckhardt, G. Gopalakrishnan, J. Mayo, M. Musuvathi, S. Qadeer, and S. Toub. Practical parallel and concurrent programming. In *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11, pages 189–194, New York, NY, USA, 2011. ACM.

[93] C. A. Sanders. Coming down the e-mail mountain, blazing a trail to gmail. In *SIGUCCS '08: Proceedings of the 36th annual ACM SIGUCCS conference on User services conference*, pages 101–106, New York, NY, USA, 2008. ACM.

[94] E. Schonfeld. Gmail grew 43 percent last year. AOL mail and Hotmail need to start worrying, 2009. `http://tinyurl.com/39mgwxu`.

[95] Standard Performance Evaluation Corporation. SPEC jvm98, 1998. http://www.spec.org/jvm98/.

[96] Standard Performance Evaluation Corporation. SPEC CPU2000 v1.3, 2000. http://www.spec.org/cpu2000/.

[97] Standard Performance Evaluation Corporation. SPEC CPU2006 v1.0, 2006. http://www.spec.org/cpu2006/.

[98] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. The STAMPede approach to thread-level speculation. *ACM Transactions on Computer Systems*, 23(3):253–300, 2005.

[99] M. Stringer, J. A. Rode, E. F. Toye, A. F. Blackwell, and A. R. Simpson. The WebKit tangible user interface: A case study of iterative prototyping. *IEEE Pervasive Computing*, 4(4):35–41, 2005.

[100] H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[101] The 5K. An award for excellence in web design and production, 2002. http://www.the5k.org/.

[102] C. Tian, M. Feng, V. Nagarajan, and R. Gupta. Copy or discard execution model for speculative parallelization on multicores. In *Proc. of the 41st IEEE/ACM Int'l Symp. on Microarchitecture*, MICRO 41, pages 330–341, 2008.

[103] W3C. World Wide Web Consortium, 2010. http://www.w3c.org/.

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

[105] A. Watt. *3D Computer Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[106] Web applications, 2010. http://en.wikipedia.org/wiki/Web_application.

[107] WebKit. SunSpider JavaScript Benchmark, 2012. http://www2.webkit.org/perf/sunspider-0.9/sunspider.html.

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

[109] Wikipedia. List of social networking websites, 2010. http://en.wikipedia.org/wiki/List_of_social_networking_websites.

[110] J. Wright and J. Dietrich. Survey of existing languages to model interactive web applications. In *APCCM '08: Proc. of the 5th Asia-Pacific Conf. on Conceptual Modelling*, pages 113–123, Darlinghurst, Australia, Australia, 2008. Australian Computer Society, Inc.

ABSTRACT

Two important trends in computer systems are that applications are moved to the Internet as web applications, and that computer systems are getting an increasing number of cores to increase the performance. It has been shown that JavaScript in web applications has a large potential for parallel execution despite the fact that JavaScript is a sequential language. In this thesis, we show that JavaScript execution in web applications and in benchmarks are fundamentally different and that an effect of this is that Just-in-time compilation does often not improve the execution time, but rather increases the execution time for JavaScript in web applications. Since there is a significant potential for parallel computation in JavaScript for web applications, we show that Thread-Level Speculation can be used to take advantage of this in a manner completely transparent to the programmer. The Thread-Level Speculation technique is very suitable for improving the performance of JavaScript execution in web applications; however we observe that the memory overhead can be substantial. Therefore, we propose several techniques for adaptive speculation as well as for memory reduction. In the last part of this thesis we show that Just-in-time compilation and Thread-Level Speculation are complementary techniques. The execution characteristics of JavaScript in web applications are very suitable for combining Just-in-time compilation and Thread-Level Speculation. Finally, we show that Thread-Level Speculation and Just-in-time compilation can be combined to reduce power usage on embedded devices.

2014:08

1653-2090

ISBN: 978-91-7295-281-2