ABSTRACT

This thesis addresses two issues: (i) The execution behavior of JavaScript in established benchmarks and in real-world Web Applications and (ii) whether Thread-Level Speculation is a suitable technique for taking advantage of multicore systems in Web Applications written in JavaScript.

The first key result is that JavaScript execution behavior by the benchmarks and the JavaScript execution behavior by the Web Applications differ in several important aspects. For instance Web Applications often use function types such as anonymous and eval functions. Our results also show that just-in-time compilation often increases the execution time of Web Applications, despite that just-in-time compilation decreases the execution time for most of the benchmarks.

The second key result is that our implementation of Thread-Level Speculation shows that it can be used to take advantage of multicore systems for Web Applications. We have measured the effect on the execution time for a set of Web Applications, and found that we are able to reduce JavaScript execution time more than 8 times compared to the sequential version on a dual quad core computer. For our use-cases we found that we used between 1.1 and 33.0 MB to store information associated with speculation.
Evaluating JavaScript Execution Behavior and Improving the Performance of Web Applications with Thread-Level Speculation

Jan Kasper Martinsen
Evaluating JavaScript Execution Behavior and Improving the Performance of Web Applications with Thread-Level Speculation

Jan Kasper Martinsen

Licentiate Dissertation in Computer Systems Engineering

School of Computing
Blekinge Institute of Technology
SWEDEN
Abstract

This thesis addresses two issues: (i) The execution behavior of JavaScript in established benchmarks and in real-world Web Applications and (ii) whether Thread-Level Speculation is a suitable technique for taking advantage of multicore systems in Web Applications written in JavaScript.

The first key result is that JavaScript execution behavior by the benchmarks and the JavaScript execution behavior by the Web Applications differ in several important aspects. For instance Web Applications often use function types such as anonymous and eval functions. Our results also show that just-in-time compilation often increases the execution time of Web Applications, despite that just-in-time compilation decreases the execution time for most of the benchmarks.

The second key result is that our implementation of Thread-Level Speculation shows that it can be used to take advantage of multicore systems for Web Applications. We have measured the effect on the execution time for a set of Web Applications, and found that we are able to reduce JavaScript execution time more than 8 times compared to the sequential version on a dual quad core computer. For our use-cases we found that we used between 1.1 and 33.0 MB to store information associated with speculation.
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 their help this thesis would never have seen the light of day.

I would also like to thank Anders Isberg and Troed Sängberg from Sony Ericsson for giving me this opportunity to work in a real-world industrial project and providing me with invaluable help, feedback and suggestions.

In addition I would like to thank all my colleagues, especially those in the CCS research group at BTH and all those who are part of the EASE project. The close collaboration between industry and different research fields of academic research has proven to be of great value.

Finally I would like to thank Heidi Kramer, for countless hours of 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


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


# Contents

1 Introduction ................................................. 1

1.1 JavaScript and Web Applications ......................... 1
1.2 An introduction to Thread-Level Speculation .............. 2
1.3 Related work ............................................... 3
  1.3.1 JavaScript behavior .................................... 3
  1.3.2 Software-Based Thread-Level Speculation .............. 4
1.4 Research questions ......................................... 6
1.5 Research methodology ..................................... 7
  1.5.1 Measuring the Execution Behavior of JavaScript benchmarks and Web Applications .............. 7
  1.5.2 Implementing and evaluating Thread-Level Speculation ........ 8
1.6 Contributions ............................................ 9
  1.6.1 Contributions in Paper I .............................. 9
  1.6.2 Contributions in Paper II ............................ 10
  1.6.3 Contributions in Paper III ............................ 10
  1.6.4 Contributions in Paper IV ........................... 11
  1.6.5 Contributions in Paper V ............................ 11
1.7 Validity of results ....................................... 12

vii
1.7.1 Generalizability .............................................. 12
1.7.2 Internal validity .............................................. 13
1.8 Conclusion .................................................... 13
1.9 Future work .................................................... 14

2 Paper I .......................................................... 15
2.1 Introduction ................................................... 15
2.2 Background .................................................... 17
  2.2.1 JavaScript and web applications ....................... 17
  2.2.2 Previous work ............................................. 17
2.3 Benchmarks and web applications .......................... 18
  2.3.1 JavaScript benchmarks ................................. 18
  2.3.2 Social networking web applications .................... 19
2.4 A methodology for evaluating JavaScript execution behavior .. 20
  2.4.1 Representative behavior ................................ 20
  2.4.2 Reproducible behavior .................................. 22
  2.4.3 Experimental environment ............................. 24
2.5 Measurement results ........................................ 25
  2.5.1 Distribution of function calls and execution time .... 25
  2.5.2 Anonymous function behavior .......................... 26
2.6 Discussion and future work ................................ 28
2.7 Concluding remarks ........................................ 30

3 Paper II ......................................................... 33
3.1 Introduction .................................................. 33
3.2 Background and related work ................................ 35
  3.2.1 JavaScript ............................................... 35
  3.2.2 Related work ............................................. 36
3.3 Experimental methodology .................................. 37
3.4 Application classes ........................................ 38
  3.4.1 JavaScript benchmarks .................................. 38
  3.4.2 Web applications - Alexa top 100 ....................... 39
  3.4.3 Web applications - Social network use cases .......... 40
  3.4.4 HTML5 and the canvas element ......................... 42
3.5 Experimental results ....................................... 44
  3.5.1 Comparison of the effect of just-in-time compilation . 44
  3.5.2 Comparison of bytecode instruction usage ............... 47
  3.5.3 Usage of the eval function ........................... 50
  3.5.4 Anonymous function calls ............................. 52
3.6 Conclusions .................................................. 53

4 Paper III .................................................... 55
  4.1 Introduction ............................................... 55
  4.2 Background ................................................ 57
    4.2.1 Thread-Level Speculation ............................ 57
    4.2.2 JavaScript ........................................... 60
  4.3 Method ..................................................... 60
  4.4 Experiments ............................................... 61
  4.5 Results .................................................... 63
  4.6 Concluding remarks ....................................... 64

5 Paper IV ..................................................... 69
  5.1 Introduction ............................................... 69
  5.2 Background ................................................ 70
    5.2.1 Thread-Level Speculation ............................ 71
    5.2.2 JavaScript ........................................... 73
    5.2.3 Unrepresentative benchmarks .......................... 73
  5.3 Current work .............................................. 75
5.4 Future work ......................................................... 76
5.5 Conclusion .......................................................... 77

6 Paper V ................................................................. 79

6.1 Introduction ....................................................... 79
6.2 Background .......................................................... 81
  6.2.1 JavaScript .................................................... 81
  6.2.2 Web Applications ........................................... 82
  6.2.3 Thread-Level Speculation Principles .................... 82
  6.2.4 Software-Based Thread-Level Speculation ............... 83
6.3 Thread-Level Speculation Implementation for JavaScript ... 85
6.4 Experimental Methodology .................................... 90
6.5 Experimental Results .......................................... 92
  6.5.1 Execution Time Improvements ......................... 92
  6.5.2 Speculations and Rollbacks ......................... 95
  6.5.3 Memory Usage ........................................... 98
6.6 Conclusions ...................................................... 101
Introduction

Two important trends in computer systems are that applications are moved to the Internet as Web Applications, and that 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 Web Applications.

1.1 JavaScript and Web Applications

Many web pages have been given application functionality, which informally has turned them into Web Applications. Popular examples are Google mail, Facebook, Twitter and Youtube. Web Applications have become popular due to their ease of distribution, simplicity and platform independence. Web Applications often combine client and server side functionality. In this thesis we focus on the client side functionality.

The client side functionality is commonly written in the sequential language JavaScript, which in turn is executed in a JavaScript engine. JavaScript supports features which are similar to features in functional programming languages, such as anonymous and eval functions, while JavaScript has a syntax similar to C and Java. To evaluate JavaScript performance a set of established benchmarks suites have evolved. These are often ported to JavaScript from benchmarks in other problem domains. There have been studies that evaluated the workload char-
acterization of benchmarks and Web Applications where studies \[65, 64\] suggest that they have a number of major differences such as different kinds of function types and different executed bytecode instructions. JavaScript and Web Applications have some support for multithreaded systems in Web workers \[79\]. Web workers use a message passing memory model. However, it is the programmers responsibility to find and use the parallelism in Web Applications for Web workers.

1.2 An introduction to Thread-Level Speculation

The performance potential of multithreaded has been demonstrated for several applications. Examples of applications are visualization, video, 3D, web servers and databases. To take advantage of multithreaded processors, an operating system that supports multithreaded is needed. For a programmer to take advantage of a multithreaded system, one would need to write parallel and scalable programs.

In general, parallel programming is currently error prone, time consuming and complicated. It might be far from obvious how your program should be partitioned into parallel tasks. The programmer might also run into well known problems such as deadlocks and starvation. 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. Starvation is similar to a deadlock, a process waits forever for a resource, without getting one.

There have been multiple attempts to make parallel programming easier. One attempt is Thread-Level Speculation. The idea is to dynamically extract parallelism from sequential programs. We do this by speculatively executing segments of the program in parallel. To ensure 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.

This can be done in many ways: in hardware, e.g., \[13, 66, 74\], and software, e.g., \[12, 36, 58, 62, 70\]. In most cases, the main target of the techniques is for-loops and the main idea is to allocate each loop iteration to a thread. Then, ideally, we can execute as many iterations in parallel as we have processors.

There are, however, some limitations. Data dependencies between loop iterations may limit the number of iterations that can be executed in parallel. Further, the memory requirements and run-time overhead for managing the necessary information 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 therefore be able to detect these dependencies during run-time using dynamic information about read and write addresses from each loop iteration. A key design parameter here is the precision in the detection mechanism, i.e., at what granularity can a TLS system detect data dependency violations? High dependence detection precision usually requires high memory overhead in a TLS implementation.

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 roll-back mechanism. In order to be able to do roll-backs, 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 roll-backs must be low.

A key design parameter for a TLS system is the data structures used to track and detect data dependence violations. In general, the more precise tracking of data dependencies, the more memory overhead is required. Unfortunately, one effect of imprecise dependence detection is the risk of false-positive violations. A false-positive violation is when a dependence violation is detected when no actual dependence violation is present. As a result, unnecessary roll-backs need to be done, which decrease 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 memory, or in a special speculation buffer. Updating data in-place usually results in higher performance if the number of roll-backs is low, but lower performance when the number of roll-backs is high since the cost of doing roll-backs is high.

1.3 Related work

1.3.1 JavaScript behavior

JavaScript is a dynamic programming language often used in association with the Web Applications. During the last years there has been an unofficial race to have the fastest web browser, and therefore there has been an increased focus on the performance of JavaScript.
Ratanaworabhan et al [64] have found 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 Web Applications have large number of loops (and large loops), non-string objects in Web Applications are extremely short lived and that the Web Application only deals with a small number of events.

Richards et al [68] have found that the dynamic features of JavaScript are used quite extensively in Web Application. The benchmarks do not use many of these features and they behave like static typed programs. It is in turn pointed out that this might lead to a misleading understanding of the execution behavior of JavaScript.

1.3.2 Software-Based Thread-Level Speculation

There exists a number of different software-based TLS proposals, and we review some of the most important ones here. One striking observation is that all of these studies have worked with applications written in C, Fortran, or Java. We have not found any study that addresses the applicability and performance potential of TLS in a dynamically typed scripting language, such as JavaScript.

Bruening et al. [12] 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. The results show speed-ups of up to almost five on 8 processors, but also show slow-downs for some rare cases.

Rundberg and Stenström [70] 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 roll-backs caused by false-positive violations. However, the downside of their approach is high memory overhead. They show a speed-up of up to ten times on 16 processors for three applications written in C from the Perfect Club Benchmarks [8].

Kazi and Lilja developed the coarse-grained thread pipelining model [36] for 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 model to also support Java programs [35].

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 address 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 when using both speculative and non-speculative threads.

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, e.g., the SPEC CPU2000 [73] and Perfect Club [8] Benchmark suites.

In two studies Chen and Olukotun present [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 [61, 62] developed a TLS framework, SableSpMT, 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 mainly at the byte code level. They obtain at most a two-fold speed-up on a 4-way multicore processor.

Oancea et al. [58] 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. Oancea et al. evaluate their approach using seven applications from three benchmark suites (SciMark2, BYTEmark, and JOlden). The results show that their TLS approach reaches in average 77% of the speed-up of hand-parallelized, non-speculative versions of the programs.
Kejariwal et al. [37] evaluated the performance potential of TLS using the SPEC CPU2000 Benchmarks [73]. SPEC CPU2000 consists of 26 applications written in C and Fortran. They found that TLS has a mean speed-up potential of approximately 40% over the applications in addition to the true Thread-Level parallelism exploited.

A succeeding study by Prabhu and Olukotun [63] analyzed what types of Thread-Level parallelism that can be exploited in the SPEC CPU2000 Benchmarks. 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.

In a study by Mehrara et. al [49] an attempt is made to utilize a multicore architecture in a Firefox browser. Their system is able to achieve an average of 2.18 speedup over the Firefox browser using 8 threads on a multicore system, while performing the required analyses and conflict detection dynamically at runtime.

1.4 Research questions

The main focus of this thesis is how to utilize multicore systems in Web Applications. Research question 1 and research question 2 evaluate the execution behavior of Web Applications and benchmarks.

Research question 1: What method should we use to compare the execution behaviour of the benchmarks with the Web Applications?

Research question 2: How do the execution behaviour of the Web Applications and the benchmarks differ?

JavaScript is a sequential language and to utilize multicore systems we investigate Thread-Level Speculation as a way to take advantage of multicore systems.

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

Research question 4: What is the performance improvement potential with Thread-Level Speculations for Web Applications?
Research question 1 and research question 2 are discussed in paper I and paper II, where we perform execution behavior evaluation of popular Web Applications and benchmarks. We perform evaluations by using the Firebug JavaScript profiler and a modified version of WebKit.

Research question 3 and research question 4 are discussed in paper III, paper IV and paper V, where we have implemented Thread-Level Speculation in two JavaScript engines. We evaluate these implementations using popular Web Applications and the V8 benchmark suite.

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 used two methods to evaluate this, the JavaScript profiler firebug [22] and a modified version of Webkit.

Firebug evaluates the JavaScript code, such as function names, function types and the amount of time spent executing each function. By modifying Webkit, 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 JavaScript.

A key challenge when evaluating the execution time, function names and types, and bytecode instructions of Web Applications, is to make the results reproducible and repeatable. 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 repeatability, 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 function and eval calls and executed JavaScript might be different for
two identical use cases. To cope with this, we have evaluated each use case 10 times, and selected the best result out of the 10 for comparison.

To evaluate JavaScript functionality, we have selected five different applications which use JavaScript; (i) A set of four established benchmark suites for JavaScript, Google V8, SunSpider, Dromaeo and JS Benchmark [28, 82, 52, 34], (ii) The top 100 most visited Web Applications from the Alexa list [57]. (iii) A set of use-cases on a selection of social networks [84]. (iv) A selection of 15 popular Web Applications, selected based on their popularity and (v) the top 109 Web Applications in HTML5 from the JS1K competition [33].

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 for JavaScript. We have made two implementations of Thread-Level Speculation: The first one in the Rhino engine [19], and the second one in the Squirrelish engine (which is part of Webkit) [83]. 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 Squirrelish JavaScript engine and Webkit are used on a number of platforms (for instance various Android phones and Apple iPhones), and the engine supports Web Applications.

To implement Thread-Level Speculation, we have done the following: whenever we encounter a function call, we have created a new thread for this function call. We also store associated information before we executed a thread. If this thread is in conflict with any of the other threads, we use the stored information when we rollback. Once its associated execution completes, the results of the execution are merged into the main thread. The main difference between the two implementations, is that the Squirrelish version allows function calls that are encountered in speculated functions to create new threads.

We have evaluated the effects of Thread-Level Speculation by evaluating the execution time of the associated JavaScript in Rhino and Squirrelish. In Rhino we have evaluated the benchmarks from the Google V8 benchmarks suite and for Squirrelish, 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.

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.

1.6.1 Contributions in Paper I

The main focus in paper I is to evaluate 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., an event) in the web browser. One of the key problems of comparison is that unlike benchmarks, Web Applications do not have a start and end state. To make the 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., [65, 68], 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 [65, 68]. In paper II we perform a bytecode instruction analysis to compare the type 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 effect 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 of the JS1K JavaScript competition that used HTML5.

The main contributions are: 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 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

In paper I and paper II we discussed and evaluated the representativeness of benchmarks, as compared to Web Applications. The results in paper II indicated that just-in-time compilation might degrade the performance of Web Applications. Paper I showed that loop-like structures are rare in Web Applications. In this paper we look at Thread-Level Speculation as a way to take advantage of multicore systems and improve the execution time. We implement speculation on function calls in the Rhino JavaScript engine. We evaluate the number of conflicts between function calls, the amount of functions that have a null return value, and the likeliness that we are able to guess the return value based on previous speculations. Our results show an improved execution time for some of the V8 benchmarks on a dual quad-core computer, compared to a sequential version.

Our main contributions are: We demonstrate that speculating on return values based on historical data in a dynamically typed language such as JavaScript is fruitful. Global variables are likely to cause conflicts in function calls executed
as a thread and it demonstrates some performance increases by speculation on null return function calls.

### 1.6.4 Contributions in Paper IV

Paper IV binds together paper I and paper II with paper III. 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 benchmark 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 execution time for two of the benchmarks, however for the other benchmarks the execution time will increase.

### 1.6.5 Contributions in Paper V

Unlike our previous implementation of Thread-Level Speculations, we have used the Squirrelfish JavaScript engine which is part of Webkit which is the state-of-the-art open sources browser environment used in several platforms. In this implementation we allow speculated function calls to create new threads. We make evaluations on 15 Web Applications, where we have selected Web Applications based on their functionality and popularity.

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 during 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 [83] which is part of Webkit. Performance evaluation of Thread-Level Speculation for 15 popular
Web Applications, e.g., Facebook, Gmail, YouTube, and Wikipedia. Our results show significant speedups for most of the studied Web Applications, up to 8.4 times for the best case, on eight cores, and includes a detailed analysis of the speculation and rollback behavior as well as the memory overhead.

1.7 Validity of results

We discuss the threats to the validity of studies, more specific 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 [69]. 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 effect of just-in-time compilation, and the importance of JavaScript language functionality.

We found that there are differences between benchmarks and Web Applications. For instance, many web browsers have security mechanisms which prevents large loops. This conforms with small instances of problem sizes for benchmarks, 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.

1.7.2 Internal validity

Internal validity refers to how well a study can establish the relationship between cause and effect [69]. We know from our experiments that there is 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 Web Applications, issues which were described in paper I and paper II.

1.8 Conclusion

To answer research question 1 we minimize 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 workload of Web Applications are not well represented by the benchmarks. This has several important effects, for instance that current techniques for improving execution speed, might fail for Web Applications. Another interesting result of this difference is that special features in JavaScript like anonymous functions and eval functions are commonly used in Web Applications.

To answer research question 3 and research question 4 we have implemented Thread-Level Speculation on two JavaScript engines, Rhino and Squirrelfish. We were able to execute the JavaScript in our Web Application over 8 times faster with our implementation compared to a sequential version for 15 Web Applications. While the number of rollbacks is low, the number of speculations is high. 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 1.1MB to 33.0MB for the Web Applications, to ensure that we were able to restore the application to a point in time when the execution was correct.
1.9 Future work

These studies indicate that Thread-Level Speculation is a promising technique to improve execution speed for JavaScript on multicore systems. There are four paths to further research; (i) Can we use just-in-time compilation and Thread-Level Speculation together? (ii) Which engine is better for Thread-Level Speculation, register based or stack based? (iii) Is there an adaptive heuristic that makes Thread-Level Speculation speculate correctly on promising functions and at the same time keep the number of rollbacks low? (iv) And can we reduce the memory requirements and overhead in general?

(i) We found that just-in-time compilation could degrade the execution time of Web Applications, however Web Applications are rapidly changing towards a more multimedia oriented workload, which is more suitable for just-in-time compilation. It would be interesting to see whether we could use Thread-Level Speculation to speculate on functions where just-in-time compilation is used.

(ii) There are mainly two types of engines used for JavaScript; Stack based and register based. The advantages and disadvantages of both have been discussed previously, however it seems to be unknown which one is best suited for Thread-Level Speculation.

(iii) We tried to execute the content of a function as a thread, unless the function was misspeculated earlier. Could previous speculation be used to make adaptive heuristics to reduce the memory requirements of speculations and at the same time keep the number of rollbacks fairly low? What factors should such a heuristic consider and what situation should such a heuristic be based on?

(iv) We are saving information, in case of a rollback. Could we reduce the amount of memory used, by using a different data structure?
Chapter 2

Paper I

A Methodology for Evaluating JavaScript Execution Behavior in Interactive Web Applications

Jan Kasper Martinsen and Håkan Grahn


2.1 Introduction

JavaScript was introduced primarily as an interpreted prototype-based scripting language for web pages, which allowed programmers to add interactivity to web pages [23, 32]. With JavaScript these web pages where given application-like behavior. As a result, a larger number of more or less sophisticated parts of typical desktop-like applications became accessible as web pages. One typical example is Google’s mail client Gmail [71]. These web pages are informally known as web applications [85].

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, 57]. Many of the entries among top 25 web sites are social networks, e.g., Facebook [34], Twitter [38], and MySpace [60]. Several studies have confirmed the popularity of social networking web applications [20, 21, 50].

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 [25, 26, 67].

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 [28, 52, 82]. Some of the critique [65, 68] 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., [65, 68], 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 6.2 presents some background and previous work, and then Section 2.3 presents the benchmarks
and web applications that we use. In Section 4.3, 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 [23, 32] 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 pop-
ularity of certain web applications, e.g., Gmail [72]. 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 run-
time evaluation. The execution of a JavaScript program is done in a JavaScript
engine [29, 53, 83], 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) compila-
tion techniques, fast property access, and efficient garbage collection [26, 29].

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 ap-
lications. This process is in its early phases, but there are several examples
of interesting work [54, 5]. Two concurrent studies [65, 68] explicitly compare
the JavaScript execution behavior of web applications as compared to existing
JavaScript benchmark suites.
The study by Ratanaworabhan et al. [65] 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. [68] 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 [28], Dromaeo [52], and Sunspider [82]. 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 [80]. 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.

<table>
<thead>
<tr>
<th>Benchmark suite</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dromaeo</td>
<td>3d-cube, core-eval, object-array, object-regexp, object-string, string-base64</td>
</tr>
<tr>
<td>V8</td>
<td>crypto, deltatable, earley-boyer, raytrace, richards</td>
</tr>
<tr>
<td>SunSpider</td>
<td>3d-morph, 3d-raytrace, access-binary-trees, access-fannkuch, access-nbody, access-nslieve, bitops-3bit, bitops-dotnas, bitops-dotnslieve, bitops-bitwise-and, bitops-nslieve-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</td>
</tr>
</tbody>
</table>

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 [84], where Facebook [54] is the most popular one [4, 21]. There are even examples of countries where half of the population use Facebook to some extent during the week [20]. 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 [38], and MySpace [60]. 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., [65, 68], 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 3.1, Figure 3.2 and Figure 3.3 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 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 2.1: Use cases to characterize the JavaScript workload of Facebook.
Figure 2.2: Use cases to characterize the JavaScript workload of Twitter.

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 [27] 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.

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 [18]. 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 3.3 (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 [75] 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.
Figure 2.4: Total number of functions for each of the 10 repetitions of the same case.

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.

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

2.5.2 Anonymous function behavior

A previous study of Facebook reveals that a large number of anonymous function calls are made [41]. 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 3.1, Figure 3.2 and Figure 3.3.

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

2.6 Discussion and future work

As pointed out in [65, 68] 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.

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

sively. These kind of functions are spawned from a different field, and are usually not available in other programming languages. That does, however, not mean 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,
Figure 2.10: Anonymous function calls for Facebook.

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 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.
Figure 2.11: Anonymous function calls for Twitter

Figure 2.12: Anonymous function calls for MySpace
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 [81]. 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 [55], Twitter [38], 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 lan-
guage JavaScript [32] extensively for their implementation, and as a mechanism
to improve both the user interface and the interactivity.

JavaScript [32] was introduced in 1995 as a way to introduce dynamic func-
tionality on web pages, that were executed on the client side. JavaScript has
reached widespread use through its ease of deployment and the popularity of cer-
tain web applications [72]. We have found that nearly all of the first 100 entries
in the Alexa top sites list use JavaScript.

JavaScript [32] is a dynamically typed, object-based scripting language with
run-time evaluation. The execution of a JavaScript program is done in a JavaScript
engine [29, 83, 53], 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 out-
perform each other.

In order to evaluate the performance of JavaScript engines, several benchmark
suites have been proposed, e.g., Dromaeo [52], V8 [28], SunSpider [82], and
JSBenchmark [34]. However, two previous studies indicate that the execution
behavior of existing benchmarks differs in several important aspects [65, 68].

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 [55], Twitter [38], and Blogger [10] (sometimes referred to as BlogSpot),
and finally, (iv) 109 demo applications for the emerging HTML5 standard [30].
Our measurements are performed with WebKit [83], one of the most commonly
used browser environments in mobile terminals.

We extend previous studies [65, 68] 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 ap-
lications 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 per-
formance 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 [78]. There are several reasons for this, e.g., accessibility and mobility. These applications are commonly known as web applications [81]. 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 [23, 32], 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 [56], where a scalable web server is written in JavaScript.

JavaScript [23, 32] 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 [72]. 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 [29, 53, 83], 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 tech-
niques, fast property access, and efficient garbage collections [26, 29].

The execution of JavaScript code is often invoked in web application through
events. Events are JavaScript functionalities that are executed at certain oc-
casions, 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 ex-
cution, 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 2.01.

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., [55, 5]. Two concurrent studies [65, 68] explicitly compare the JavaScript
execution behavior of web applications as compared to existing JavaScript bench-
mark suites.

The study by Ratanaworabhan et al. [65] is one of the first studies that com-
pares JavaScript benchmarks with real-world web applications. They instrumented
the Internet Explorer 8 JavaScript runtime in order to get their measure-
ments. 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.

chrome-682414

36
The study by Richards et al. [68] 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 [42]. 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):

\[ \frac{T_{exe}(JIT)}{T_{exe}(NOJIT)} \geq 1 \]
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 [52], V8 [28], Sunspider [82], and JSBenchmark [34]. 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 [80].

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.

<table>
<thead>
<tr>
<th>Benchmark suite</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dromaeo [52]</td>
<td>3d-cube, core-eval, object-array, object-regexp, object-string, string-base64</td>
</tr>
<tr>
<td>V8 [28]</td>
<td>crypto, deltablue, earley-boyer, raytrace, richards</td>
</tr>
<tr>
<td>JSBenchmark [34]</td>
<td>Quicksort, Factorials, Conway, Ribosome, MD5, Primes, Genetic Salesman, Arrays, Dates, Exceptions</td>
</tr>
</tbody>
</table>

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 alphabetically).

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

3.4.3 Web applications - Social network use cases

There exists many so-called social networking web applications [84], where Facebook [55] is the most popular one [4, 21]. There are even examples of countries where half of the population use Facebook to some extent during the week [20]. 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 [55], Twitter [38], 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 [38] 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.](image-url)

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.](image)

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-
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 [31] is the next standard version of the HyperText Markup Language. The Canvas in element HTML5 [30] 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 [51].

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, 77, 33]. 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.

| 1 | Legend Of The Bouncing Beholder         | @mariansjnh    |
| 2 | Tiny chess                             | Oscar Toledo G. |
| 3 | Tetris with sound                      | @sjoerd_visscher |
| 4 | WOLF1K and the rainbow characters      | @p01           |
| 5 | Binary clock (tweetable)               | @alexym        |
| 6 | Mother fucking lasers                  | @evilhackerdude |
| 7 | Graphical layout engine                | Lars Ronnback  |
| 8 | Crazy multiplayer 2-sided Pong         | @feiss         |
| 9 | Morse code generator                   | @chrismoak     |
| 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 craigslist.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.
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
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.
Figure 3.8: Branch, jump, and arithmetic/logical related bytecode instructions for the Alexa top 100 web sites and the SunSpider benchmarks.

The only arithmetic/logical operation that is more used in web applications is the `not` instruction, which could be used in, e.g., comparisons.

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.](image)

### 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 \texttt{eval} calls are executed, we have used the FireBug [22] JavaScript profiler to extract this information. We have measured the number of \texttt{eval} calls relative to the total number of function calls, i.e., \textit{No. of \texttt{eval} calls / Total no. of function calls}.

Figure 3.10 presents the relative number of \texttt{eval} calls. Our results show that \texttt{eval} functions are rarely being used in the benchmarks, only 4 out of 35 benchmarks use the \texttt{eval} function. However, these four use \texttt{eval} quite extensively. The dromaeo-core-\texttt{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 \texttt{eval} calls. From their name, e.g., \texttt{eval-test} in the Dromaeo benchmark, and by inspection of the JavaScript code and the amount of \texttt{eval} calls, we suspect that these benchmarks were designed specifically to test the \texttt{eval} function.

![Graph showing relative number of \texttt{eval} calls](image)

\textbf{Figure 3.10:} Number of \texttt{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 \texttt{eval} function is used more frequently in the Alexa top 100 web sites. 44 out of 100 web sites use the \texttt{eval} function. In average, the relative number of \texttt{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.

![Chart showing eval calls relative to total number of function calls for the first 100 entries in the Alexa list.](image)

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 seem 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](image_url)  

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 [65, 68], 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.
Paper III

An Alternative Optimization Technique for JavaScript Engines

Jan Kasper Martinsen and Håkan Grahn


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 [59]. However, 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, 48, 76]. Meanwhile many applications are moved to the World Wide Web, as so called Web Applications, and new popular programming languages, e.g., JavaScript [32], have emerged. JavaScript is a dynamically typed, object-based scripting language with run-time evaluation, where execution is done in a JavaScript engine. It has also been stated from the language’s designers that JavaScript was created with web designers in mind, giving the designers a mean to quickly add interactivity to web pages without
too much complexity. Distribution of programs in the form of source-text files have also been advantageous with respect to platform independence. However, currently no JavaScript engine fully supports parallel execution of threads.

Developing parallel applications is 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 [12, 36, 58, 62, 70, 14]. For example, software TLS approaches often extract parallelism from loops or functions. A number of consecutive loop iterations or function calls are speculatively executed in parallel, where a data dependency check mechanism detects dependency violations which forces us to restart the program from a certain point in the execution flow (a rollback). The performance potential of TLS has been shown for applications with static loops, statically typed languages, and in byte code environments.

In this paper we describe an approach to apply TLS to the Rhino JavaScript engine [19] and evaluate the performance of the V8 benchmark [29]. In our study, JavaScript function calls are under certain conditions executed as a threads, and dependency violations are detected and solved at runtime. Initial results indicate some promises for TLS in JavaScript engines if the number of function calls is large enough. This paper makes the following contributions:

- Demonstrates that speculating on return values based on historical data in a dynamically typed language such as JavaScript is fruitful.

- Global variables are likely to cause conflicts in function calls executed as a thread.

- Demonstrates some performance increases by speculation on null-returning function calls.

The rest of the paper is organized as follows. Section 6.2 provides some background on thread-level speculation and JavaScript. Then, we present our method in Section 4.3. Our experimental setup is presented in Section 4.4, while the experimental results are presented in Section 6.5. The paper ends with the conclusions in Section 6.6.
4.2 Background

In Section 5.2.1 we present the general principles of thread-level speculation and some previous implementation proposals. Then, we discuss the JavaScript language, that is our target in this study, in Section 6.2.1.

4.2.1 Thread-Level Speculation

Thread-Level Speculation Principles

Thread-level speculation (TLS) aims at dynamically extracting parallelism from a sequential program. This can be done in many ways: in hardware, e.g., [13, 66, 74], and software, e.g., [12, 36, 58, 62, 70]. In most cases, the main target of the techniques is for-loops and the main idea is to allocate each loop iteration to a thread. Then, ideally, we can execute as many iterations in parallel as we have processors.

There are, however, some limitations. Data dependencies between loop iterations may limit the number of iterations that can be executed in parallel. Further, the memory requirements and run-time overhead for managing the necessary information 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). Therefore must a TLS implementation be able to detect these dependencies during run-time using dynamic information about read and write addresses from each loop iteration. A key design parameter here is the precision in the detection mechanism, i.e., at what granularity can a TLS system detect data dependency violations. High dependence detection precision usually require high memory overhead in a TLS implementation.

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. In order to be able to do roll-backs, we need to store both speculative updates of data as well as the original data values. As result, this book-keeping results in both memory overhead as well as run-time overhead. In order for TLS system to be efficient, the number of roll-backs shall be low.

A key design parameter for a TLS system is the data structures used to track and detect data dependence violations. In general, the more precise tracking of
data dependencies, the more memory overhead is required. Unfortunately, one effect of imprecise dependence detection is the risk of false-positive violations. A false-positive violation is when a dependence violation is detected when no actual dependence violation is present. As a result, unnecessary roll-backs 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 memory, 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 performance when the number of roll-backs is high since the cost of doing roll-backs is high.

Software-Based Thread-Level Speculation

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

Bruening et al. [12] proposed a software-based TLS systems 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. The results show speed-ups of up to almost five on 8 processors, but also show slow-downs for some rare cases.

Rundberg and Stenström [70] proposed a TLS implementation that resembles the behavior of a hardware-based TLS system. The main advantage with their approach is that it precisely tracks data dependencies, thereby minimizing the number of unnecessary roll-backs cased by false-positive violations. However, the downside of their approach is high memory overhead. They show a speedup of up to ten times on 16 processors for three applications written in C from the Perfect Club Benchmarks [8].

Kazi and Lilja developed the coarse-grained thread pipelining model [36] for 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 to also support Java programs [35].
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 address 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 when using both speculative and non-speculative threads.

Cintra and Llanos[17] present a software-based TLS system that speculatively execute 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, e.g., the SPEC CPU2000 [73] and Perfect Club [8] Benchmark suites.

Chen and Olukotun present in two studies [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 [61, 62] developed a TLS framework, SableSpMT, 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 mainly at the byte code level. They obtain at most a two-fold speed-up on a 4-way multi-core processor.

Oancea et al. [58] 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. Oancea et al. evaluate their approach using seven applications from three benchmark suites (SciMark2, BYTEmark, and JOlden). The results show that their TLS approach reaches in average 77% of the speed-up of hand-parallelized, non-speculative versions of the programs.

Kejariwal et al. [37] evaluated the performance potential of TLS using the SPEC CPU2000 Benchmarks [73]. SPEC CPU2000 consists of 26 applications written in C and Fortran. They found that TLS has a mean speed-up potential
of approximately 40% over the applications in addition to the true thread-level parallelism exploited.

A succeeding study by Prabhu and Olukotun [63] analyzed what types of thread-level parallelism that can be exploited in the SPEC CPU2000 Benchmarks. By going through each of the application, 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.

One striking observation from all studies presented above is that they all have worked with applications written in C, Fortran, or Java. The Java studies have usually been done at the bytecode level. We have found no study that addresses the applicability and performance potential of TLS in a dynamically typed scripting language, such as JavaScript.

### 4.2.2 JavaScript

JavaScript [32] is a dynamically typed, object-based scripting language with runtime evaluation. JavaScript application execution is done in a JavaScript engine, i.e., an interpreter/virtual machine that parses and executes the JavaScript program. Examples of JavaScript engines are Google’s V8 engine [29], WebKit’s Squirrelfish [83], and Mozilla’s SpiderMonkey and TraceMonkey [53].

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

### 4.3 Method

To evaluate the effects of TLS in JavaScript we have modified an existing Java based JavaScript engine Rhino1.7R2 (Rhino) [19]. Rhino takes the JavaScript source code and compiles it into an internal byte code representation which in turn is executed.

When we encounter an interpreted function call instruction (rather than a native one) we consider this function call as a candidate that could be executed
as a thread. We have made the following assumption: If a function calls returns a null value it is more likely to be independent from the rest of the program. To do this, we record the return value of the function call (along with the ID of the function). If it returns null, we will execute this function as a thread the next time it is called. If a data dependence violation (conflict) occurs for this function call, it is marked and will never be called as a thread again.

If a function call violates any other parts in the program (or similar executing functions), we perform a rollback, i.e., restore the engine to the point before we executed the selected function call, and re-execute the function sequentially. The process, both to store the state and to restore a state of the JavaScript engine is expensive. In our experiments we utilize the continuations functionality found in later versions of Rhino. However, this also adds some limitations in a sense that we are, e.g., not allowed to restart eval functions.

During the execution, we monitor writes and reads, both in the active threads that execute function calls, as well as the main program that spawned the threads. We do not yet allow threaded function calls to spawn new threaded function calls. Once the threaded function call returns, changes are committed, writes and reads are compared, and common objects are merged (the threaded function keeps track of the id of the original object). We keep track of a counter together with read and writes which allow us to keep track of conflicts between threaded function calls as well as the main thread. If a conflict is detected (e.g., both a speculative thread and the main program have modified an object), a rollback is performed. If we have multiple concurrent function calls, these are ended, and we return to the first executed function call.

To evaluate the simple TLS approach implemented in the modified Rhino JavaScript engine, we use the V8 JavaScript benchmark suite. The different applications in the V8 suite is listed in Table 5.2.3, along with short description. The deltablue benchmark does not execute correctly in Rhino, so we choose to omit it from our experiments.

4.4 Experiments

In our experiments, we have measured and evaluated the following aspects:

- The number of functions that return a null value.
- The effect of two strategies for return value prediction.
- The number of data dependence violations (conflicts) with global variables.
Table 4.1: V8 Benchmark programs.

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Richards</td>
<td>OS kernel simulation benchmark</td>
</tr>
<tr>
<td>DeltaBlue</td>
<td>One-way constraint solver</td>
</tr>
<tr>
<td>Crypto</td>
<td>Encryption and decryption benchmark</td>
</tr>
<tr>
<td>RayTrace</td>
<td>Ray tracer benchmark</td>
</tr>
<tr>
<td>EarleyBoyer</td>
<td>Classic Scheme benchmarks</td>
</tr>
<tr>
<td>RegExp</td>
<td>Regular expression benchmark generated</td>
</tr>
<tr>
<td>Splay</td>
<td>Data manipulation benchmark</td>
</tr>
</tbody>
</table>

- The relative performance improvement by enabling TLS.

If the interpreted function call does not return a value (or returns a null value), it indicates less dependencies with other parts of the executed program. We have counted the number of functions that return a null value and the number of functions that return a non-null value, respectively.

The first time we encounter a function call we do not know its return value. We have made two approaches to predict if the function returns a null or not. First we count the number of return null instructions in the byte code associated with the given function call. If the number of return null instructions is larger or equal to 1 and the number of null return instructions are higher than the other return instructions, we predict that the function will return a null value.

The second approach is an extension of the first approach. Each time a function returns, we store: the name of the function and the function’s return value. When we encounter a function call instruction, we first search the associated byte code for return null instructions, and then a previous return value. If this function has been executed previously, we override the decision made from the findings in the associated byte code, and then choose null return or not based on historical data.

We have measured each time a threaded function call accesses a global variable, when this variable is manipulated also by the main program. We have measured the amount of functions that accesses a global variable, while the main program at the same time manipulates the same variable.
We have evaluated the effects of TLS on the execution time on two different systems. One dual-core laptop with a Windows Vista operating system and a quad core workstation with an Ubuntu Linux operating system. We ran Rhino with Java 1.7.0. In the experiments, we speculate on all function calls with a null return value that are called the first time or that have previously been successfully executed as a thread.

4.5 Results

We see in Figure 4.1 that functions with null return account for a relatively small fraction (at most 16% in the Richards benchmark) of the total number of function calls in the benchmarks. However, the number of function calls do not completely reflect the workload of these functions. We have studied the benchmark code, and found that some functions that return null also encapsulate a large amount of value returning function calls.

Figure 4.2 and Figure 4.3 show the results from the extended prediction strategy, where we also predict on the function return value. In the figures we show how often the prediction is correct, for both static and dynamic guesses. We see that the extension to predicting the return value based on a function significantly improves the chances for predicting if the return value is null or not.

For most of the cases this strategy seems to almost always make a correct prediction. However, this is not the case when making a prediction based on the associated byte code. We have denoted the percentage of the correct static predictions and correct dynamic predictions after the name of the benchmark in the Figure 4.2. We also see in Figure 4.3 that the prediction of "not-null" value based on the associated byte code performs very badly. The reason seems to be that for functions with a large amount of null returning instructions, this does not necessarily imply that it is more likely that a null returning instruction is executed.

The results presented in Figure 4.4 show that conflicts between function calls executed speculatively and the main program's global variables happen for approximately half of the functions. Therefore, accesses to global variables from functions pose a threat.

While the functions with a null return value do not account for so many of the total number of function calls, we have found some performance gain for some of the benchmarks. In Figure 5.1 we present the relative execution times
Figure 4.1: Number of null value returns and total number of function calls. Note that the regex benchmark has no interpreted function calls.

for the applications running with and without TLS on two different computer systems. The execution times are normalize to the sequential execution time without TLS. We see that the execution time is improved for 4 of the cases. However, the improvement is relatively small, only a 22% reduction in the best case.

4.6 Concluding remarks

Figure 5.1 shows that the V8 benchmarks do not offer much of a performance improvement with the TLS technique. The largest gain in execution time is 22% on a quad core computer for the Richards benchmark. This can be explained by looking at Figure 4.1, where we see that the number of interpreted functions with a null value is low. However, for the Richard case, there is a larger improvement
Figure 4.2: Number of null function returns, with the number of successful predictions from functions byte code and the number of successful predictions from previous functions executions.

than the number of null return values indicates. This suggests that null value functions perhaps also encapsulates return value functions.

We also observe for the Earley benchmark, that there could be a relationship between the difficulties to speculate on the return values. For instance, we see in Figure 4.1 that there is a large amount of non-null return values, and a small number of null return values for the Earley benchmark. This suggests that is harder to speculate on return values. In addition, we see in Figure 4.4 that the number of global conflicts is especially high for the Earley benchmark. The raytracing benchmark in Figure 4.4 has a large amount of conflicts with global variables, but still there are some improvements when the benchmark is run on a dual core PC. This illustrates the penalty of creating and handling a large amount of threads, and that a large number of speculations also increases the risk for rollbacks which are demanding to administrate.

65
Figure 4.3: Number of value function returns, with the number successful of predictions from functions byte code and the number of successful predictions from previous functions executions.

Even though the execution time is not significantly improved it have been suggested that benchmarks such as V8 might not be representative for the workload of real-world web applications [68]. We know from previous studies that real world web application’s workloads consist of a large number of anonymous function calls (which return a null value). The event-driven model found in web browsers, also suggests that these functions might be even more important than the V8 benchmark suite suggests. This suggests that a TLS technique could be more powerful, with a different set of benchmark, or perhaps in a real-life context.
Figure 4.4: Number of functions with data dependence violations with global variables.

Figure 4.5: Relative execution time with TLS enabled, normalized to the execution time without TLS enabled.
Chapter 5

Paper IV

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

Jan Kasper Martinsen, Håkan Grahn and Anders Isberg


5.1 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, 48, 76]. 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 [42]. 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 [70]. 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 [40]. 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 6.2 provides some background on TLS, JavaScript and Unrepresentative benchmarks. In Section 5.3, we discuss our current research, in Section 5.4 we make an outline for future work and finally in Section 6.6 a conclusion.

### 5.2 Background

In Section 5.2.1 we present the general principles of thread-level speculation and some previous implementation proposals. In Section 6.2.1 we discuss the JavaScript language and finally in Section 5.2.3 a brief summary of the unrepresentativeness of current JavaScript benchmarks.
5.2.1 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 dependencies 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 performance when the number of roll-backs is high since the cost of doing roll-backs is high.
Software-Based Thread-Level Speculation

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

Bruening et al. [12] 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 [70] 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 coarse-grained thread pipelining model [36] for exploiting coarse-grained parallelism. They suggest to pipeline the concurrent 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 Olukorun 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 [61, 62] 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. [58] 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. 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 [63] analyzed what types of thread-level parallelism that can be exploited in the SPEC CPU2000 Benchmarks[73]. 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 [39] and [40] we have used the TLS technique for a dynamic language such as JavaScript. As seen in Figure 5.1 there is a modest speedup for a few instances

5.2.2 JavaScript

JavaScript [32] is a dynamically typed, object-based scripting language with runtime evaluation often used in association with Web Applications. JavaScript application execution is done in a JavaScript engine, i.e., an interpreter/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.

5.2.3 Unrepresentative benchmarks

We found that JavaScript benchmarks were ported from existing benchmark suites, and that thanks to the flexibility of the JavaScript programming language, one could port without utilizing certain JavaScript features (for instance such as
anonymous and eval function call) [41, 42]. 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 arithmetic instructions (Figure 5.2) 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 multimedia functionalities will be even more dependent on JavaScript and these features.
5.3 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.

5.4 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.
5.5 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 6

Paper V

Using Thread-Level Speculation to Improve the Performance of JavaScript Execution in Web Applications

Jan Kasper Martinsen, Håkan Grahn, and Anders Isberg

Submitted for publication

6.1 Introduction

During the last years have many applications moved to or evolved on the World Wide Web. Such applications are often referred to as web applications. 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 [55] and Blogger [10], have turned out to be popular, being in the top-25 web sites on the Alexa list [4] of most popular web sites. Both these applications use the interpreted language JavaScript [32] extensively for their implementation. In fact, almost all of the top-100 sites on the Alexa list use JavaScript to some extent.
JavaScript is a dynamically typed, object-based scripting language with runtime evaluation, where execution is done in a JavaScript engine [29, 83, 53], i.e., an interpreter/virtual machine that parses and executes the JavaScript program. With the increased popularity of Web Applications and due to higher performance demands, several optimization techniques have been suggested along with sets of benchmarks. However, these benchmarks have been reported as unrepresentative [42, 65, 68], and current optimization techniques, e.g., just-in-time compilation, could even degrade the performance of popular Web Applications [44].

JavaScript is a sequential language and cannot take advantage of multicore processors. This is unfortunate, since Fortuna et al. [24] showed that there exist significant potential parallelism in many JavaScript applications, a potential speedup of up to 45 times was reported. However, they have not implemented support for parallel execution in any JavaScript engine. Many browsers support ‘Web Workers’ [79] that allow parallel execution of tasks in Web Applications based on a message-passing paradigm, but the programmer is still responsible for finding and expressing the parallelism.

To hide some of the details of the underlaying parallel hardware, an approach is to dynamically extract parallelism from a sequential program using Thread-Level Speculation (TLS) techniques [70]. The performance potential of TLS has been shown for applications with static loops, statically typed languages, and in Java bytecode environments. Martinsen and Grahn [40] proposed to use TLS in a JavaScript context using the Rhino JavaScript engine and some established JavaScript benchmarks.

In this paper, we present an implementation of thread-level speculation in the Squirrelfish [83], a state-of-the-art JavaScript engine found in the WebKit browser environment, along with an evaluation of it. The execution and behaviour of a Web Application is dependent not only of the JavaScript code, but also of the interaction with the web browser and the DOM tree. However, we deliberately focus only on the JavaScript part in this study.

Our main contributions are:

- The first implementation of thread-level speculation in a state-of-the-art JavaScript engine, i.e., Squirrelfish [83].
- A performance evaluation of thread-level speculation for 15 popular web applications, e.g., Facebook, Gmail, YouTube, and Wikipedia.
- Our results show significant speedups for most of the studied Web Applications, up to 8.4 times in the best case, on eight cores.
• A detailed analysis of the speculation and rollback behavior as well as the memory overhead.

Our results show that web applications are suitable for speculative execution, since there is a potential to execute a large number of functions as threads. Further, the results show that there are, in general, few rollbacks. Finally, the memory overhead is modest, up to 33.0 MB in the worst case.

This paper is organized as follows: In Section 6.2, we present an introduction to JavaScript, Web Applications, and thread-level speculation as well as an overview of related work. Section 6.3 presents our implementation of TLS. In Section 6.4, we present our experimental methodology, including the studied Web Applications. Our experimental results are presented in Section 6.5. Finally, in Section 6.6 we conclude our findings.

6.2 Background

In Section 6.2.1 and Section 6.2.2 we discuss the JavaScript language and Web Applications, respectively. Then, in Section 6.2.3 we present the general principles of thread-level speculation and in Section 6.2.4 some previous implementation proposals.

6.2.1 JavaScript

JavaScript [32] is a dynamically typed, object-based scripting language with runtime evaluation often used in association with Web Applications. JavaScript application execution is done in a JavaScript engine, i.e., an interpreter/virtual machine that parses and executes the JavaScript program. Popular examples of JavaScript engines are Google's V8 engine [29], WebKit's Squirrelfish [83], and Mozilla's SpiderMonkey and TraceMonkey [53]. JavaScript offers some flexibility, with a syntax similar to C and Java, while it at the same time offers functionalities associated with dynamic programming languages, such as modifying types, execution of new code as strings, and extending objects and definitions at runtime.

The performance of these script engines have increased during the last years, reaching a higher single-thread performance for a set of benchmarks. It has been suggested that the results from these benchmarks might be misleading [42, 65, 68],
and that optimizing towards the characteristics of the benchmarks may even cause a degrading of the execution time for real-life Web Applications [44].

Many browsers support 'Web Workers' that allow parallel execution of tasks in Web Applications, using a massage passing paradigm. However, it is still the programmer who is responsible for finding and expressing the parallelism. Initial experiments report that only a small number of tasks has been used concurrently.

6.2.2 Web Applications

Web Applications is an easy way to distribute programs. Most commonly they are Web Pages, with functionality written in JavaScript. This functionality is often related to UI tasks. A lot of the Web Applications functionality is typically defined as a set of events. These events are JavaScript functions that are executed when certain things occur in the Web Application. Common examples of events are mouse clicks, task that are repeated between time intervals, or task that are performed upon loading the page. 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 show that Web Applications use dynamic programming language features extensively [42, 65, 68]. For instance, various part of the program are defined at run-time (through eval functions), and types and extensions of objects are re-defined during runtime (through anonymous functions).

6.2.3 Thread-Level Speculation Principles

TLS aims to dynamically extract parallelism from a sequential program. This can be done both in hardware, e.g., [13, 66, 74], as software, e.g., [12, 36, 58, 62, 70]. 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 memory requirements and run-time 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 dependencies during run-time using 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 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 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.

6.2.4 Software-Based Thread-Level Speculation

There exists a number of different software-based TLS proposals, and we review some of the most important ones here. It should be noted that all these studies have worked with applications written in C, Fortran, or Java. We have not found any study that addresses the applicability and performance potential of TLS in a dynamically typed scripting language, such as JavaScript.

Bruening et al. [12] 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 [70] proposed a TLS implementation that resembles the behaviour of a hardware-based TLS system. The main advantage with their approach is that it precisely tracks data dependencies, thereby minimizing the number of unnecessary rollbacks cased by false-positive violations. The downside is high memory overhead. They show a speedup of up to ten times on 16 processors for three applications written in C from the Perfect Club Benchmarks [8].

83
Kazi and Lilja developed the course-grained thread pipelining model [36] 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 [35].

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 execute 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, e.g., the SPEC CPU2000 [73] and Perfect Club [8] benchmark suites.

Chen and Olukotun present two studies [15, 16] on 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 [61, 62] 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. They obtain at most a two-fold speed-up on a 4-way multi-core processor.

Oancea et al. [58] 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. 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 [63] analyzed what types of thread-level parallelism that can be exploited in the SPEC CPU2000 Benchmarks [73]. 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.

The study by Mehrara and Mahlke [49] addresses 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 or not. They execute the runtime checks (guards) 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.

6.3 Thread-Level Speculation Implementation for JavaScript

We have implemented thread-level speculation in the Squirrelfish JavaScript interpreter which is part of WebKit [83], a state of the art web browser environment. Initially, we made some modifications so it would be easier to execute the main interpreter function as a thread. More specifically, we use a switch statement instead of a goto statement (where the goto labels are predefined memory locations), and disabled just-in-time compilation. In addition, we have modified the interpreter function so it can be executed from a thread, and the input parameters to the interpreter were modified so they sent as a part of a structure.

A general view of how the speculation is done is shown in Figure 6.1. If the interpreter makes a call to a JavaScript function, a new thread is spawned and placed in a thread pool. Before the new thread is spawned, the state of the threads in the thread pool, a set of writes and reads, and the values of the JavaScript program are saved for possible rollbacks. We support nested speculation, i.e., a speculated thread can create new speculative threads. Upon a conflict, e.g., on variable X in Figure 6.1, we need to do a rollback and restore the execution to a safe state.
Figure 6.1: An example of TLS. First a new thread is spawned, the state is then saved, before it spawns another thread which in turn will have a conflict with the thread it was spawned from. Upon a conflict the state is restored.

Initially the entire executed JavaScript program, which in our case is extracted from an execution in the web browser, is sent to a thread that executes the interpreter. The first thread is not speculated and will never be re-executed. Therefore, we do not need to store the data that is part of this thread’s execution. We start the thread with an initial value \textit{realtime} set to 0. For each executed bytecode instruction, the value of \textit{realtime} is increased by 1.

When the main thread is initialized, it is given a unique id and starts to execute. The extracted program contains the data of Squirrelfish (which means for instance the content of the Squirrelfish registers) and the opcodes of the bytecodes. We have added a counter which we denote as the \textit{sequential time}. The value of \textit{sequential time} starts from 0 and is equivalent to the number of executed bytecode instructions.

When we execute the main thread and encounter a section that is suitable for speculation, i.e., it starts with the opcode \textit{op$_{-}$enter} and ends with the opcode \textit{op$_{-}$ret}. This might be a JavaScript function defined in the JavaScript program, or it could be a function which is part of a Web Application’s event. When we encounter this type of opcodes, we do the following. We record the \textit{sequential time} for \textit{op$_{-}$enter}. We examine whether it has previously been speculated, by looking up the \textit{sequential time} counter \textit{ps} in a list of previous speculations. We denote this list as \textit{previous}. If the value at this index is equal to 0 then it has not previously been speculated, otherwise, if this value is equal to 1 then it has previously been speculated. If \textit{ps} is 1, we continue execution in the same thread, i.e., we do
not speculate, and execute this section. However, inside this (non-speculative) section we might encounter another section that is suitable for speculation. If $ps$ is 0, then this section is an candidate for speculation and we denote $ps$ as a fork point.

If $ps$ is a fork point, then we set the value of its index to 1 in previous, to be sure that this is not speculated later in case of a rollback. We copy all the associate values which will be used in case this speculation is unsuccessful. These values are the following: The list of modified global values (we describe this below), the list of associated values from each thread (we describe this below). In addition we store the id of the parent thread. We pass a copy of realtime, equal to the parent thread's realtime. We assign the program from the sequential time $op_{\text{enter}}$ to the sequential time of $op_{\text{ret}}$ for this thread. If there is no thread available, we create a new one from a list of uninitialized threads. If there is an available thread, e.g., available as a previous failed speculation, this thread is repopulated from this fork point.

In these studies, we look at conflicts between global variables and ids. Ids are special for JavaScript as they can be created at any point of time, and can be defined with a global scope. During execution, we might encounter four different opcodes which manipulate global variables or ids:

- $op_{\text{put\_global\_var}}$, which writes a value to a specific global variable,
- $op_{\text{get\_global\_var}}$, which reads a value from a specific global variable,
- $op_{\text{put\_by\_id}}$, which writes to a specific globally accessible id, and $op_{\text{get\_by\_id}}$, which reads from a specific globally accessible id.

When we encounter one of the four cases during one of the thread execution we do the following. We extract the realtime, the sequential time, an unique identification for the variable (which is either the index of the global variable or the name of the id), the type of variable (either global or id) and the type of operation (either a write or a read operation). We then check the variable conflict against a list previous, where earlier reads or writes are indexed by a unique identity of the variable.

There are four kind of cases that we test against, partly shown in Figure 6.2:

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

(ii) The current operation is a read, and there is a previous write operation with the same unique identification. In this case, we must check the realtime and
the sequential time, so that the following does not occur. We do not accept that the read happened in realtime before the write, if the read happened after the write in sequential time. Likewise, we do not accept that the read happened in realtime after the write, if the read was happening before the write in sequential time.

(iii) The current operation is write, and there is a previous read operation with the same unique identification. In this case we check that the realtime together with the sequential time, so that the following does not occur. We do not accept that the write happens in realtime before the read if read happens before the write in sequential time. Likewise, we do not accept that write happens after read in realtime, if write happens before read in sequential time.

(iv) The current operation is a write and the previous operation is a write. We do not accept that this write happens before the previous write in realtime, if this had the other order in sequential time. Likewise, we do not accept that write happens after the compared write if write happened in realtime before write in sequential time. Once we have checked against all earlier entries and the previous (and no conflict did occur) that value of this operation is added to the previous list.

Figure 6.2: Values of sequential time and realtime at different phases of the speculation.

In addition we could end up in a situation where several of the threads perform a write or read operation at the same realtime. To handle this, we have done this check after realtime is increased by 1, and perform the test above iterative
for all the operations. Likewise, if the list of unique identities is empty, we insert the value.

To get an unique identifier for id is trivial as it is simply a string with the associated name. Global variables on the other hand is an index of a list, the same global variable has a different list position in this list when the function calls are nested. To be able to track the global variable we are tracking this global variables between function calls. From this tracking we are able to find an unique identifier from a global variable that is computed based on the depth of the function call, as well as its position in the list.

Case (ii), (iii), and (iv) force us to do a rollback to ensure program correctness. The idea of a rollback is that the program is re-executed from a point before the conflict occurred. More specifically, we rollback to a point before the current speculation that led to the conflict. When we encounter such a problem, we note the current thread where the conflict is, and we note its parent thread (i.e., the thread where the spawn point is found). At this point information related to the various threads are extracted. We extract information from this point, such as 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, and so are variable conflicts in previous.

Even though we have a set of threads that are supposed to be active, it is likely that there might have been created threads after this point of time, and that these not associated with the current state of the TLS system. Therefore, we need to recursively go through the threads and their parent 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 status for later reuse.

When a thread reaches its end of execution (encounter its associated op_ret), its modification of global variables and id need to be committed back to its parent thread, as shown in Figure 6.3. However, this can first be done after threads that have been created from the completed thread’s fork points have in turn completed their execution. These threads, are denoted as child threads and their manipulations to global variables and ids are to be committed to the current thread. This is also the case for the main or the initial thread, after the program completes execution after all the threads have completed execution they are committed to the main thread.

In our implementation we are rather pessimistic. When a thread completes execution, and commits its values, we do not remove the associated read and writes from global variables and ids that are no longer relevant. Therefore there
Figure 6.3: Two speculative threads are executed and committed when no conflict occurs.

might be conflicts, which are not between active threads, but rather are conflicts that were before the threads were committed.

A challenge when using TLS in Web Applications, is the underlying run-time system. There might be several events linked to user interaction, timed events, or modification of a specific element that are outside of the JavaScript interpreter, e.g., accesses to the DOM tree, but are sent to the interpreter for execution. Many of these events are suitable candidates for speculation. However, they also pose a problem. Assume that we speculate on a mouse click event, that is associated with a certain JavaScript function. Assume also that this function manipulates something, such that there will be a conflict, and we would need to rollback to ensure program correctness. We are able to rollback to a safe state in the interpreter, but the event and the executed JavaScript could become inconsistent. In this study, we deliberately focus only on the JavaScript interpreter part.

6.4 Experimental Methodology

Our thread-level speculation is implemented in the Squirrelfish [83] JavaScript engine which is part WebKit, a state-of-the-art browser environment. We have selected 15 popular Web Applications from the Alexa list [4] of most used web sites. We tried to select popular Web Application to cover a wide range of different types of Web Applications, while being used by a reasonable large user group. The selected applications along with a short description is found in Table 6.1. Then,
Table 6.1: List of web applications used in this study, listed from the most popular (Google) to least popular (Gmail) [4].

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

we have defined and recorded a set of use-cases for applications and 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. As a result, we can ensure 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. The methodology is further described in [42].

All experiments are conducted on a server running Ubuntu 10.04 and equipped with two quad-core processors and 16 GB main memory. In all measurements we have measured the execution time in the JavaScript engine, rather than the execution time of the overall Web Application.
6.5 Experimental Results

6.5.1 Execution Time Improvements

We start our results by evaluating how much faster the JavaScript execution time is when thread-level speculation is enabled. Therefore, we compare the execution times of the different Web Applications with and without TLS. The relative execution times are shown in Figure 6.4; $T_{exe}(\text{with TLS}) / T_{exe}(\text{without TLS})$, i.e., a value lower than 1 means that the execution time is lower with TLS enabled. The results in Figure 6.4 show that TLS improves the execution time of the JavaScript in the Web Applications between 8.39 (YouTube) and 1.02 (Amazon) times as compared to the sequential execution time.

In order to understand the performance improvement with TLS enabled, we have measured a number of metrics and show their values in Table 6.2. We
have measured the maximum number of threads active during the execution, the number of speculations and rollbacks, the maximum and average speculation depths, and the memory usage for each of the applications (use cases). Our results show, e.g., that the maximal number of threads varies significantly, from 8 (Wikipedia) to 407 (YouTube).

The use case with the highest speedup is YouTube, which executes 8.39 times faster with TLS than the sequential version. The YouTube use case has more than twice as many maximum number of runnable threads as compared to the second one, which is MSN. YouTube has at most 407 active threads as compared to 191 for MSN. The YouTube use case has a large number of functions, together with a low number of rollbacks. The average search depth to remove data associated with previous speculations are low relative to the number of speculations (0.003). We observe in Figure 6.13 that YouTube only has two large rollbacks, in terms of memory size, which is among the last rollbacks.

The Amazon use case has the lowest speedup, it only runs 1.02 times faster with TLS than the sequential execution time. In Table 6.2 we see that the Amazon use case has the highest number of rollbacks, and a large number of speculations. Even though the relation between the number of rollbacks and speculations is low, i.e., 2.5%, we are unable to decrease the execution time using TLS, and the TLS performance is only slightly better than for the sequential version.

To compare the various use cases against each other is difficult, since they may very different characteristics. However, we choose to compare use cases that have approximately the same number of speculations since this indicates that the programs have a similar number of function calls.

The two use cases with the largest number of speculations are Amazon and MSN with 12012 and 10768 speculations, respectively. However, Amazon has 3 times as many rollbacks as MSN. Amazon also has a larger average depth in the search for information upon rollbacks while MSN has a smaller depth, 8.0 and 5.54, respectively. MSN has a larger number of maximum active threads (the second largest with 191), while Amazon only has 83. MSN has a higher average memory requirement of 20.1 MB, while Amazon has only has 14.1 MB. The higher execution overhead and lower number of threads for Amazon results in a very low speedup, only 1.02, as compared to a speedup of 2.3 for the MSN use case.

YouTube and Ebay have similar numbers of speculations, 7349 and 7140, respectively. However, Ebay has four times as many rollbacks as YouTube, but they scan through almost the same depth of relevant information upon rollbacks
(except that Ebay does that four times as often). The maximum number of parallel threads for the You Tube use case is 6 times higher than for Ebay. While the speculation depth are similar for Ebay (15) and YouTube (13), the average memory requirement is higher for Ebay (27.0 MB) than for YouTube (17.1 MB). In total, the lower number of rollbacks and the higher number of parallel threads for YouTube results in a significantly higher speedup for YouTube (8.3) than for Ebay (2.3).

Gmail, Google, and LinkedIn have 1193, 1282, and 1815 speculations, respectively. The number of rollbacks are different; 19 for Gmail, 36 for Google, and 51 for LinkedIn. Upon rollbacks, the three use cases need to search for relevant information on depths 2.68, 3.9, and 2.27. The maximum number of threads are similar with 34, 40, and 36. The average memory requirements differ between the use cases, 5.5MB, 1.95MB, and 7.1MB, respectively, while the maximum speculation depths are similar, 10 for Gmail and Google and 11 for LinkedIn. If we compare the TLS enabled version with the sequential version, the finds that the speedup for Gmail is 1.6, the speedup for Google is 1.4, and the speedup for LinkedIn is 1.9.

Facebook and Blogspot have 968 and 778 speculations, respectively. The Facebook use case has a relatively large number of rollbacks relative to the number of speculated functions, i.e., 0.052. It also has a low maximal number of threads, and compared to the other cases it has a low number of speculated functions. In addition, we need to search rather deep in the speculation nest upon rollbacks in the Facebook use case. The overall result is that the Facebook use case runs 1.9 times faster with TLS than without it.

For the Blogspot use case we have a lower number of rollbacks than for Facebook, a lower speculation depth, a lower memory usage, a lower average depth on rollbacks, and a lower maximal number of simultaneous threads. Blogspot has lower values for all metrics as compared to Facebook, except for the number of speculations. In total, this results in a TLS speedup of 4.8 for the Blogspot use case.

Wordpress and Imdb have approximately the same number of speculations, 5852 and 5300, respectively. If we compare the number of rollbacks for the Wordpress and Imdb use cases, we see that Wordpress has less than half of the rollbacks of Imdb (63 and 156), and Imdb also has a larger search depth upon rollbacks than Wordpress (6.85 and 4.55). Wordpress has almost twice as many maximum number of threads as Imdb (99 and 54). Imdb uses more memory than Wordpress, and has a slightly larger average speculation depth (17.8 and 6.85).
than Wordpres (9.7 and 4.55). If we compare the execution times, we see that Wordpres has a larger speedup than Imdb, 3.8 as compared to 2.8.

In summary, our results indicate the importance of a large number of threads running simultaneously. For example, both MSN and YouTube with a large maximum number of threads improve the execution time more than similar use cases. From the examples Amazon, MSN, YouTube, Ebay, Wordpres, and Imdb it also is important with a low number of rollbacks to ensure a low execution time. From the YouTube case, we see that, as compared to examples with a similar or lower number of rollbacks, a high maximum number of concurrent threads is important.

### 6.5.2 Speculations and Rollbacks

In Table 6.2, we observe that the number of speculation and the nested speculation depth of functions that are spawned from other functions are high for the Amazon and MSN use cases (12012, 24 and 10768, 23). When encountering a rollback, we need to recursively delete irrelevant information that we saved in case of future speculations. We traverse through all the check-pointed stored information at the rollback. Once this is done, we remove information that is not a part of the state we have rollbacked to. In Table 6.2 we see that the average depth of this traversal varies from 2.22 (Bing, excluding the Wikipedia case) to 9.16 (Facebook).

In Table 6.2 we see that the number of rollbacks varies from 0 to 267. If we consider the number of rollbacks relative to the number of speculations, i.e., rollbacks/speculation, and excluding the Wikipedia case, we see that it goes from 0.0034 (YouTube) to 0.05940 (Bing). In other words, between 0.3% up to 5.9% of all speculations result in a rollback, which is very low.

In Figures 6.5, 6.6, 6.7, 6.8 and 6.9 we have measured how the rollbacks are distributed in time during the execution. Since the execution time of the different Web Application use cases are very different, we have normalized the time points when the rollbacks occur relative to the total number of executed JavaScript bytecode instructions. We have created a list of 1000 elements, where we denote each element as a slot that is initialized to 0. If we for instance are executing a use case with 50000 bytecodes, and perform a rollback at bytecode instruction number 23000, we add one to the list at elements[1000 x 23000/50000].

As can be seen in Figures 6.5, 6.6, 6.7, 6.8 and 6.9, the rollbacks are not evenly distributed over the program execution time. If a rollback occurs, it is
likely that another rollback occurs shortly after. We can partially observe that in Figures 6.10 and 6.13, the relative memory requirements. If we have a rollback, then the memory requirements will often be lower, which indicates that rollbacks follow each other.

When we encounter a rollback, we go back to a previous correct (safe) execution state. We have previously seen that the amount of memory (Figure 6.10 and Figure 6.13) is reduced after each rollback. We do this to avoid wasting memory for speculations we will not use anyway and delete information irrelevant to the point we rollback to. When we rollback to the previous state, we recursively through all parents' functions. Once we know which one is associated with the restored state we can remove irrelevant states. In Table 6.2 we present the average number of recursive steps in order to extract parent functions that are associated with the restored state. We see that the average depth is between 2.22 and 9.16.
Figure 6.6: Distribution of rollbacks for the MSN, Ebay and Myspace use cases.

Even though we remove irrelevant information upon a rollback, the memory requirement at a rollback is in our results from 1.1MB to 33MB \(^1\). The relationship between rollbacks and speculations goes from 0.3\% to 5.9\%, i.e., between 0.3\% to 5.9\% of the speculations result in a rollback. In previous research, referred to in Section 6.2.4, it has been suggested that a program should have less than 10\% rollbacks in order to benefit from TLS. All of the Web Application use cases in this study are well below this boundary.

We define the function depth of a speculated function as follows: Let’s assume that we execute a function, we give this function a \( \text{depth}(1) = 1 \). While executing the bytecode instructions of this function, we might encounter another function. We give this function a depth relative to the parent function by \( \text{depth}(2) = \text{depth}(1) + 1 \). In Table 6.2 we present the maximum depth of the various Web

---

\(^1\)The Wikipedia use case has no rollbacks, therefore we present the amount of memory upon completion of the program in that case.
Application use cases, which vary from 4 to 24 and that the average speculation depth is 15.

We have observed that there is no clear relationship between function depth and the number of speculations. For example, The Facebook use case has a function depth of 22 and only 968 functions were speculatively executed, while for the Wordpress use case, the function depth is 21 and 5852 functions were speculatively executed.

6.5.3 Memory Usage

In Table 6.2 we show the average memory requirements at a rollback to an earlier state for each of the use cases. The average memory requirements vary from 1.1MB (Wikipedia) to 33.0MB (BBC). For the Wikipedia use case there was no
rollbacks, so we have measured the total amount of memory used for speculation when the execution is completed.

In Figure 6.10 and Figure 6.13, we present the total memory requirements before we do each rollback for the use cases. To better understand the behavior of the memory requirements upon rollbacks across the use cases, we have normalized the memory requirement at each rollback to the maximum memory requirement for each use case. That is, we have taken the memory requirement at each rollback and divided it with the largest memory requirement upon a rollback for each use case. We denote this memory requirement as the relative memory requirement.

For all use cases the relative memory requirement for the first rollback is on average 9% of the largest relative memory requirement for the first rollback. An exception is the Bing use case where the relative memory requirement upon a rollback is 41% of the largest relative memory requirement.

Figure 6.8: Distribution of rollbacks for the Google, Youtube and Gmail use cases.
The relative memory requirement is also likely to decrease after a rollback
with high relative memory requirements; After a rollback, for 68% of the cases the
next rollback will have lower relative memory or an equivalent relative memory.
We have also observed that if we compare the two last values before the largest
relative memory requirement, then for 5 out of 6 cases they are significantly lower.
This indicates that the memory requirements during execution vary significantly.
The largest relative memory requirement is in 5 out of 6 cases after we have gone
halfway through the rollbacks, and for two cases it is the last or the second last
of the rollbacks (Imdb and Bing).

From these measurements it is clear that memory requirements for rollbacks
have a non-uniform distribution during program execution. We also identified two
patterns; (i) When we have a large relative memory requirement, it is likely that
the memory requirement will become lower, and (ii) the largest relative memory
requirement is often preceded by a number of rollbacks with very low relative
memory requirements. The relative memory requirement increases stepwise up
Figure 6.10: Memory usage upon rollbacks for the Amazon, imdb and BBC use cases.

to the largest relative memory requirement only for 1 out of 6 cases on average. We also notice that two of the use cases have their largest relative memory requirements just before the end of the execution (MSN and Bing). For these two use cases we see in Figure 6.10 and Figure 6.13 that the overall relative memory requirements up to the last rollback have been much lower.

6.6 Conclusions

JavaScript is an important language for most Web Applications. Unfortunately, JavaScript is a sequential language and cannot take advantage of multicore processors. An approach is to dynamically identify and extract parallelism using thread-level speculation (TLS).
Figure 6.11: Memory usage upon rollbacks for the MSN, Ebay and Myspace use cases.

In this paper, we have presented an implementation of thread-level speculation in the Squirrelfish JavaScript engine [83] found in the WebKit browser environment. We speculate at the function level and support nested speculation, i.e., a function that is executing speculatively can create new speculatively executed functions. Our evaluation is based on 15 popular Web Applications from the Alexa top list [4], e.g., Facebook, Blogspot, LinkedIn, and Wordpress. The performance measurements are done on a dual quad-core machine running Ubuntu.

Our results clearly show that TLS significantly reduces the execution time of JavaScript in Web Applications. Speedups of up to 8.4 were achieved as compared to a sequential execution. This performance improvement is achieved without any JavaScript source code changes at all. Our results show a high number of speculations, between 12 (Wikipedia) and 12012 (MSN) functions could be executed speculatively, while there were very few rollbacks, between 0 (Wikipedia) and 267 (Amazon). The relative number of rollbacks in relation to
Figure 6.12: Memory usage upon rollbacks for the Wordpress, Facebook and linkedin use cases.

the number of speculations varies from 0% (Wikipedia) to 5.9% (Bing), i.e., in
the worst case at most 5.9% of the speculations cause a rollback.

We have also measured how the nested speculation works. The maximum
speculation depth ranges from 4 to 99, while the average speculation depth ranges
from 0 up to 9.2. These results indicate that nested speculation is important in
order to achieve a high degree of dynamic parallelism. Since speculation requires
that state information is store in order to enable rollbacks, an important question
to address is how large the memory overhead is. Our measurements show that the

average memory requirements are between 1.1 MB and 33.0 MB for the studied Web Applications.
Figure 6.13: Memory usage upon rollbacks for the Google, Youtube and Gmail use cases.
Table 6.2: 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).

<table>
<thead>
<tr>
<th>Application</th>
<th>Number of speculations</th>
<th>Number of rollbacks</th>
<th>Rollbacks / Speculations</th>
<th>Maximum number of threads</th>
<th>Max speculation depth</th>
<th>Average depth</th>
<th>Memory usage (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google</td>
<td>1282</td>
<td>36</td>
<td>0.028</td>
<td>40</td>
<td>10</td>
<td>3.9</td>
<td>5.5</td>
</tr>
<tr>
<td>Facebook</td>
<td>968</td>
<td>51</td>
<td>0.052</td>
<td>27</td>
<td>22</td>
<td>9.16</td>
<td>7.1</td>
</tr>
<tr>
<td>YouTube</td>
<td>7349</td>
<td>25</td>
<td>0.003</td>
<td>407</td>
<td>13</td>
<td>5.44</td>
<td>17.1</td>
</tr>
<tr>
<td>Wikipedia</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>4</td>
<td>0</td>
<td>1.1</td>
</tr>
<tr>
<td>Blogspot</td>
<td>778</td>
<td>15</td>
<td>0.019</td>
<td>16</td>
<td>14</td>
<td>2.16</td>
<td>1.6</td>
</tr>
<tr>
<td>MSN</td>
<td>12012</td>
<td>133</td>
<td>0.011</td>
<td>191</td>
<td>24</td>
<td>5.85</td>
<td>20.1</td>
</tr>
<tr>
<td>LinkedIn</td>
<td>1815</td>
<td>51</td>
<td>0.028</td>
<td>36</td>
<td>11</td>
<td>2.27</td>
<td>7.1</td>
</tr>
<tr>
<td>Amazon</td>
<td>10768</td>
<td>267</td>
<td>0.025</td>
<td>83</td>
<td>23</td>
<td>8.0</td>
<td>14.1</td>
</tr>
<tr>
<td>Wordpress</td>
<td>5852</td>
<td>63</td>
<td>0.011</td>
<td>63</td>
<td>99</td>
<td>4.55</td>
<td>9.7</td>
</tr>
<tr>
<td>Ebay</td>
<td>7140</td>
<td>101</td>
<td>0.014</td>
<td>63</td>
<td>15</td>
<td>5.33</td>
<td>27.0</td>
</tr>
<tr>
<td>Bing</td>
<td>303</td>
<td>18</td>
<td>0.059</td>
<td>30</td>
<td>7</td>
<td>2.22</td>
<td>1.4</td>
</tr>
<tr>
<td>Imdb</td>
<td>5300</td>
<td>156</td>
<td>0.029</td>
<td>54</td>
<td>24</td>
<td>6.85</td>
<td>17.8</td>
</tr>
<tr>
<td>Myspace</td>
<td>3679</td>
<td>93</td>
<td>0.025</td>
<td>39</td>
<td>14</td>
<td>5.54</td>
<td>17.4</td>
</tr>
<tr>
<td>BBC</td>
<td>6392</td>
<td>154</td>
<td>0.024</td>
<td>117</td>
<td>14</td>
<td>5.12</td>
<td>33.0</td>
</tr>
<tr>
<td>Gmail</td>
<td>1193</td>
<td>19</td>
<td>0.015</td>
<td>34</td>
<td>10</td>
<td>2.68</td>
<td>1.95</td>
</tr>
</tbody>
</table>

105
Figure 6.14: Memory usage upon rollbacks for the Bing, Blogspot and Wikipedia use cases.
Bibliography


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


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


ACM SIGCOMM Conf. on Internet Measurement, pages 43–56, New York, NY, USA, 2008. ACM.


112


ABSTRACT

This thesis addresses two issues: (i) The execution behavior of JavaScript in established benchmarks and in real-world Web Applications and (ii) whether Thread-Level Speculation is a suitable technique for taking advantage of multicore systems in Web Applications written in JavaScript.

The first key result is that JavaScript execution behavior by the benchmarks and the JavaScript execution behavior by the Web Applications differ in several important aspects. For instance Web Applications often use function types such as anonymous and eval functions. Our results also show that just-in-time compilation often increases the execution time of Web Applications, despite that just-in-time compilation decreases the execution time for most of the benchmarks.

The second key result is that our implementation of Thread-Level Speculation shows that it can be used to take advantage of multicore systems for Web Applications. We have measured the effect on the execution time for a set of Web Applications, and found that we are able to reduce JavaScript execution time more than 8 times compared to the sequential version on a dual quad core computer. For our use-cases we found that we used between 1.1 and 33.0 MB to store information associated with speculation.