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

Jan Kasper Martinsen, Blekinge Institute of Technology
Håkan Grahn, Blekinge Institute of Technology
Anders Isberg, Sony Mobile Communications AB

JavaScript is a sequential programming language that has a large potential for parallel execution in web applications. Thread-Level Speculation can take advantage of this, but it has a large memory overhead. In this paper, we evaluate the effects of adjusting various parameters for Thread-Level Speculation. Our results indicate that 32–128 MB memory, 16 threads, and a speculation depth of 4–16 levels are enough to reach most of the performance increase, and that nested speculation is necessary in order to achieve a high Thread-Level Speculation performance in web applications.

Categories and Subject Descriptors: C.1.4 [**PROCESSOR ARCHITECTURES**]: Parallel Architectures; C.1.m [**PROCESSOR ARCHITECTURES**]: Miscellaneous

1. INTRODUCTION

JavaScript is a dynamically typed, object-based scripting language with run-time evaluation, where execution is done in a JavaScript engine [Google 2012; WebKit 2012; Mozilla 2012]. One use of JavaScript is to add interactivity to web applications. Several optimization techniques have been suggested to decrease the execution time, e.g., just-in-time compilation (JIT) [Google 2012]. However, the decrease in execution time has been measured on a set of benchmarks, which are unrepresentative for JavaScript execution in web applications [Martinsen and Grahn 2011; Ratanaworabhan et al. 2010; Richards et al. 2010]. One result of this is that JIT decreases the execution time for benchmarks, while it often increases the JavaScript execution time in popular web applications [Martinsen et al. 2011; 2013].

JavaScript is a sequential programming language and cannot take advantage of multicore processors to decrease the execution time. It is possible to take advantage of multicore in web applications through Web Workers [W3C 2011]. However, Web Workers is intended for improving the responsiveness of web applications, rather than decreasing the execution time. Fortuna et al. [Fortuna et al. 2010] have shown that there exists a significant parallelism potential in many web applications with an estimated speed up of up to 45 times.

To hide the details of the parallel hardware, one approach is to dynamically extract parallelism from a sequential program using Thread-Level Speculation (TLS) [Rundberg and Stenström 2001]. The performance potential of TLS has been shown for applications with static loops, statically typed languages, and in Java bytecode environments, and lately for the JavaScript SpiderMonkey engine on a series of well-known benchmarks [Mehrara et al. 2011] and with speculative execution [Mickens et al. 2010].

In [Martinsen and Grahn 2010], we use TLS in the Rhino JavaScript engine and evaluated it on the V8 JavaScript benchmarks. In [Mehrara and Mahlke 2011] the parallelization is extracted with a light weight speculation mechanism.

We extended our work in [Martinsen et al. 2013] and implemented TLS in the Squirrelfish JavaScript engine which is part of WebKit [WebKit 2012]and evaluated it on a number of web applications. However, even if we were able to decrease the execution time significantly; our approach had a high memory overhead.

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

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

Our main contributions are:

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

This paper is organized as follows: In Section 2, we introduce JavaScript, web applications, and Thread-Level Speculation. In Section 3 we present our implementation of TLS and a comparisons with other JavaScript engines and in Section 4, we present the experimental methodology, and the studied web applications. Our experimental results are presented in Section 5, in Section 6 we discuss our findings and in Section 7 we conclude our findings.

2. BACKGROUND

2.1. JavaScript

JavaScript [JavaScript 2010] is a dynamically typed, object-based scripting language with run-time evaluation used to add interactivity in web applications. The execution is done in a JavaScript engine. JavaScript has a syntax similar to C and Java, while it offers features such as closures and anonymous functions often found in functional programming languages such as Haskell.

The performance of popular JavaScript engines such as Google's V8 engine [Google 2012], WebKit's Squirrelfish [WebKit 2012], and Mozilla's SpiderMonkey and TraceMonkey [Mozilla 2012] has increased, reaching a higher single-thread performance for a set of benchmarks. However the results from these benchmarks are misleading [Martinsen and Grahn 2011; Ratanaworabhan et al. 2010; Richards et al. 2010] and optimizing towards the characteristics of the benchmarks increases the execution time for real-life web applications [Martinsen et al. 2011].

2.2. Web Applications

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

Studies [Martinsen and Grahn 2011; Ratanaworabhan et al. 2010; Richards et al. 2010] have show that web applications use dynamic programming language features extensively. Parts of the program are defined at runtime, and types and extensions of objects are re-defined during runtime.

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., [Chaudhry et al. 2009; Renau et al. 2006; Steffan et al. 2005], and software, e.g., [Bruening et al. 2000; Kazi and Lilja 2001; Oancea et al. 2009; Pickett and Verbrugge 2005b; Rundberg and Stenström 2001]. Two main approaches exist: loop-level parallelism and method-level speculation. In this paper we use method-level speculation.

In method-level speculation, we execute function calls as threads and we must correctly predict the return values when we speculate as well as detect the writes and reads that cause the speculative program to violate the sequential semantics. The last two are typically detected when the values associated with two function calls are committed back to their parent thread. Between two consecutive threads we can have three types of data dependencies: *Read-After-Write* (RAW), *Write-After-Read* (WAR), and *Write-After-Write* (WAW). A TLS implementation must be able to detect these dependencies during runtime using information about read and write addresses. A key design parameter for a TLS system is the *precision* of at what granularity it can detect data dependency violations.

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

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

2.4. Software-Based Thread-Level Speculation

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

Bruening et al. [Bruening et al. 2000] proposed a software-based TLS system that targets loops where the memory references are stride-predictable. This is one of the first techniques that is applicable to while-loops where the loop exit condition is unknown until the last iteration. They evaluate their technique on both dense and sparse matrix applications, as well as on linked-list traversals. The results show speed-ups of up to almost five on 8 processors.

Rundberg and Stenström [Rundberg and Stenström 2001] proposed a TLS implementation that resembles the behavior of a hardware-based TLS system. The main advantage with their approach is that it tracks data dependencies precisely, thereby minimizing the number of unnecessary rollbacks caused by false-positive violations. The downside is the memory overhead. They show a speed up of up to 10 times on 16 processors for three applications from the Perfect Club Benchmarks [Berry et al. 1989].

Kazi and Lilja developed the course-grained thread pipelining model [Kazi and Lilja 2001] 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 [Berry et al. 1989]). On an 8-processor machine they achieved speed-ups of between 5 and 7. They later extended their approach to also support Java programs [Kazi and Lilja 2000].

Bhowmik and Franklin [Bhowmik and Franklin 2002] 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 [Cintra and Llanos 2003] present a software-based TLS system that speculatively executes loop iterations in parallel within a sliding window. As a result, given a window size of *W* at most *W* loop iterations/threads can execute in parallel at the same time. By using optimized data structures, scheduling mechanisms, and synchronization policies they manage to reach in average 71% of the performance of hand-parallelized code for six applications taken from various benchmark suites [Standard Performance Evaluation Corporation 2000; Berry et al. 1989].

Chen and Olukotun shows [Chen and Olukotun 1998; 2003] 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 [Pickett and Verbrugge 2005a; 2005b] developed Sable-SpMT.Their solution is implemented in a Java Virtual Machine, called SableVM, and thus works at the bytecode level. They obtain at most a two-fold speed-up on a 4-way multicore processor.

Oancea et al. [Oancea et al. 2009] present a TLS proposal that supports in-place updates. They have a low memory overhead with a constant instruction overhead, at the price of lower precision in the dependence violation detection mechanism. However, the scalability of their approach is superior due to the fact that they avoid serial commits of speculative values. The results show that their TLS approach reaches in average 77% of the speed-up of hand-parallelized, non-speculative versions of the programs.

A study by Prabhu and Olukotun [Prabhu and Olukotun 2005] analyzed what types of thread-level parallelism that can be exploited in the SPEC CPU2000 Benchmarks [Standard Performance Evaluation Corporation 2000]. They identified a number of useful transformations, e.g., speculative pipelining, loop chunking/slicing, and complex value prediction.

A study by Hertzberg and Olukotun [Hertzberg and Olukotun 2011] has a runtime system that decreases the execution time, and where idle cores are used to analyze potentially forthcoming speculations. It reportedly decreases the execution time of SPEC CPU2000 Benchmarks by 49%.

A study by Tian et al. [Tian et al. 2008] presents a novel Copy or Discard (CorD) execution model to efficiently support software speculation on multicore processors using profiled C code transformation with LLVM [Lattner and Adve 2004] to support parallel execution. The state of speculative parallel threads is maintained separately from the non-speculative computation state. The computation results from parallel threads are committed if the speculation succeeds; otherwise, they are discarded. They achieve speed ups ranging from 3.7 to 7.8 on a server with two Intel Xeon quad-core processors.

Renau et al. [Renau et al. 2005] presents three mechanisms; Splitting Timestamp Intervals, Immediate Successor List, and Dynamic Task Merging for out-of-order spawning in TLS. These techniques are implemented into their custom compiler, and on a quad core computer they are able to have an average speed up of 1.30 for the SPECint 2000 applications.

Mehrara and Mahlke [Mehrara and Mahlke 2011] show how to utilize multicore systems in JavaScript engines. However, their study has a different approach as well as a different target than we have. It targets trace-based JIT-compiled JavaScript code, where the most common execution flow is compiled into an execution trace. Then, runtime checks (guards) are inserted to check whether control flow etc. is still valid for the trace. They execute the runtime checks in parallel with the main execution flow (trace), and only have one single main execution flow. Our approach is to execute the main execution flow in parallel.

In [Mehrara et al. 2011] they introduce a lightweight speculation mechanism that focuses on loop-like constructs in JavaScript, and if the loop contains a sufficient workload, it is marked for speculation. As this code used the trace features of Spidermonkey, a selective form of speculation is employed. They found that they were able to make speculation 2.8 times faster for well known JavaScript benchmarks. Unfortunately, large loop structures are rare in real web applications as shown in [Martinsen et al. 2011].

Mickens et al. [Mickens et al. 2010] suggest an event-based speculation mechanism which is deployed as a JavaScript library called Crom. However, unlike our approach, their main goal is to enhance the responsiveness, while our main goal is to reduce the JavaScript execution time by dynamically extracting parallelism.

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

3. THREAD-LEVEL SPECULATION IMPLEMENTATION FOR JAVASCRIPT

In this section, we describe our TLS implementation [Martinsen et al. 2013].

3.1. Speculation mechanism

Execution in Squirrelfish is divided into two stages, first the JavaScript code is compiled into bytecode instructions, then the bytecode instructions are executed. We extract two things: The compiled bytecode instructions which are to be executed, and the execution trace of a sequential execution of the bytecode instructions. We later use the sequential execution trace to validate the correctness of the speculative execution off-line.

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

During execution we might encounter the bytecode instruction that indicates the start of a function call. We extract the *realtime* value and the id of the threaded interpreter that makes this call, e.g., *p_0220* (a function is called after 220 bytecode instructions from *p_0*). We denote the value of the position of this function call as *function_order*, which emulates the sequential time in our TLS program (Fig. 1). This is possible in JavaScript in web applications since we know that there is going to be a very large number of function calls. We check if this function previously has been speculated by looking up the value of *previous[function_order]*. *previous* is a vector where each element is indexed by the *function_order*. If the entry is 1, then the function has been speculated unsuccessfully. If the value is 0, then it has not been speculated or has been successfully speculated, and we call this position a fork point.

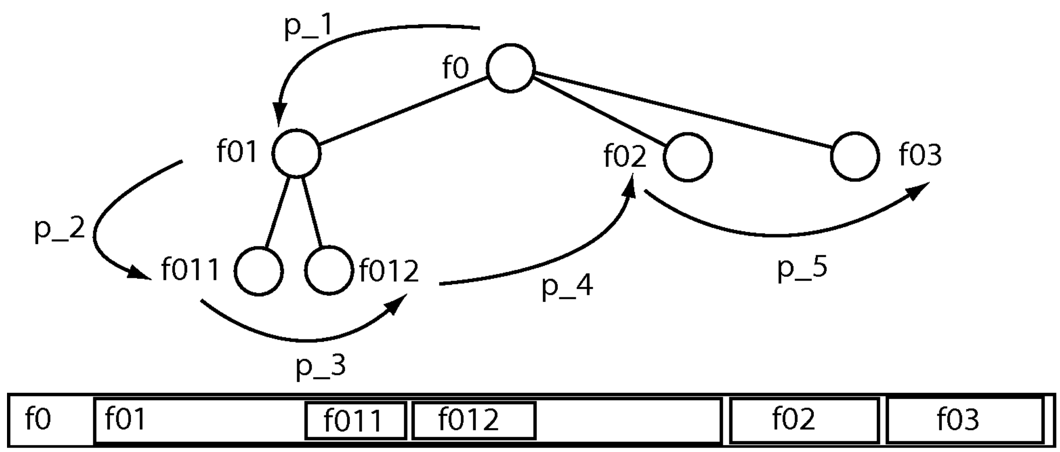

Fig. 1: We use the order that the functions are called in, to determine the order in which the program would have been sequentially executed. This works in JavaScript in a web applications setting, as there are multiple function calls. For instance, in a simplified example the JavaScript function f0, performs 3 function calls, f01, f02 and f03. f01 performs two function calls, f011 and f012. Thus, we have created a speculation tree from the function calls. If we traverse this tree from left to right, we get an order in which the functions are called, equal to the order in which the functions would be sequentially called, in order to uphold the sequential semantics during execution with TLS. More specific: f0 at time p_1, f01 at time p_2, f011 at time p_3, f012 at time p_4, f02 at time p_5, and f03 at time p_6. We denote how each function is ordered as *function_order*

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

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

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

Our return value prediction predicts the return values in *a last predicted value* manner [Hu et al. 2003] from a function with the same name (if a name is present). This is a simple heuristic for return value prediction, but as we mentioned earlier, function calls in JavaScript are often anonymous, use eval calls extensively, or these calls are events started from the web applications. These functions, do not return any value. Therefore does a heuristic such as the last returned value works fairly well for JavaScript execution in web applications.

In Fig. 2, we outline the process of speculation and a subsequent rollback to restore the execution to a safe state, i.e., commit or where the speculation started.

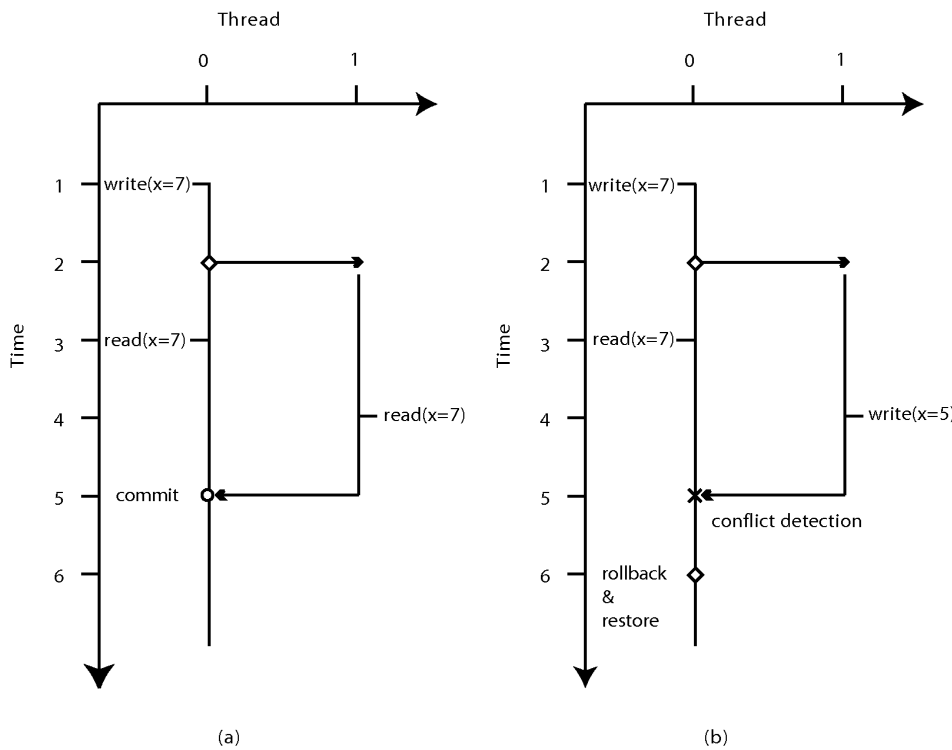

Fig. 2: In (a), at time 1 in thread 0 we write the value 7 to the global value x. At time 2 in *thread 0* we encounter a function call which we speculate on and this function call becomes *thread 1*. At time 3, *thread 0* reads 7 from the global variable x. At time 4 *thread 1* reads the same variable.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

3.2. Data dependence violation detection

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

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

Each time we encounter a read or write access to an *uid*, we evaluate the following cases in *variable_modification*.

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

(ii) The current operation is a read, and there is a previous write to the same *uid*. Therefore, we check the *realtime* and the *function_order* for the current read and the previous write. If a read occurred such that:

$$\textit{current function\_order} > \textit{previous function\_order}$$

and

$$\textit{current realtime} < \textit{previous realtime}$$

then the execution order of the program is no longer correct and we must do a rollback. Likewise, the same applies if the current operation is a write, and there is a previous read to the same *uid*. In this case, we check the *realtime* and the *function_order* from the current write and the previous read. So, if:

$$\textit{current function\_order} > \textit{previous function\_order}$$

and

$$\textit{current realtime} < \textit{previous realtime}$$

then the execution order of the program is no longer correct and we must do a rollback.

(iii) The current operation is a write, and there is a previous write to the same *uid*. We need to do a rollback if the current write happens before the previous write in *realtime* and they have the other order in *function_order*, or if the order of the write happens after the previous write in *realtime* but before the previous write in *function_order*.

3.3. Rollback

Cases (ii) and (iii) force us to do a rollback for program correctness globally, further we also do rollbacks if we write to the DOM tree. After a rollback, the program is re-executed from a point before the function was speculated. If the function where the rollback occurs is nested, we stop the JavaScript interpretion of its child threads,

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

Table I: The web applications used in the experiments.

and place the associated threads back in a thread pool for later reuse. At this point information for relevant threads are extracted, e.g., *previous* at this point, the number of associated threads at this point, the values of the associated registers in the register based JavaScript engine, the values of the global variables and object property ids are restored for the associated threads, the value of *previous* (with the index of this failed speculation set to 1), and the variable conflicts in *variable_modification*.

Even though we have a set of threads that are supposed to be active, there might be threads after the rollback that is not associated with the current state of the TLS system. Therefore, we recursively go through the threads and their child threads that are now part of the active state. The resulting list contains the threads which are necessary in the current state of execution. The remaining interpreters (running as threads), which are not necessary for the current state of the execution are stopped, and returned to the thread pool for later reuse.

3.4. Commit

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

4. EXPERIMENTAL METHODOLOGY

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

4.1. Web applications

We have selected 15 web applications (Table I) from the Alexa list [Alexa 2010]. We selected different types of web applications, such as search engines (Google and Bing) and various types of social networks (*Facebook* and *Linkedin*).

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

The JavaScript executed in web applications is fundamentally different from what is executed in the JavaScript benchmarks e.g., with multiple calls to events which often are defined as anonymous functions [Martinsen and Grahn 2011]. The number of calls varies from 12 to over 10000, but the execution characteristics are the same. These events are allowed to run for a predefined time.

To enhance reproducibility and to provide a deterministic and reproducible behavior we automatically execute the use cases [Brand and Balvanz 2005]. The methodology for these experiments is described in [Martinsen and Grahn 2011]

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

4.2. JavaScript functions in web applications

We base our experiments on the Squirrelfish JavaScript engine, where just-in-time (JIT) compilation is optional. We use the interpretive mode, since Fig. 3 shows that JIT compilation increases the execution time for 11 out of 15 use cases for Squirrelfish (when JIT is enabled) and 8 out of 15 for Google's JavaScript engine V8.

JIT compilation in Squirrelfish and V8 is measured on a set of benchmarks, and it has been shown [Martinsen and Grahn 2011; Ratanaworabhan et al. 2010; Richards et al. 2010] that these are unrepresentative for the workload in web applications. Benchmarks are similar to well-known benchmarks in other fields with a large number of loops. Since there is a limit (i.e., 10 seconds in Firefox and 5 second for Internet Explorer) to how long a JavaScript call can execute, the problem sizes that are computed by the benchmarks are artificially small. If we compare the JavaScript execution in benchmarks to the JavaScript execution in web applications, we se that most JavaScript calls are events from the web application, and specific JavaScript features such as eval and anonymous functions are extensively used. This leads to little reuse of already compiled code which is an importan feature of JIT. Therefore JIT compilation speeds up the benchmarks, while it slows down web applications to a point where it is slower with JIT compilation enabled [Martinsen et al. 2013].

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

Fig. 4, shows that the number of JavaScript function calls and their size in terms of executed bytecode instructions in web applications varies (the mean max and min of the size of executed functions is 68168.75 and 2.19 bytecode instructions over an average of 15574.53 function calls). We can understand this from a nested speculation point of view. Functions at a low depth contain many of the proceeding function calls (i.e., *Wordpress* makes almost 80% of the functions call at depth 1 or 2), and as the depth increases the number of executed bytecode instructions decreases. In the figure below, we see that the most executed functions are anonymous function calls (i.e., for instance *youtube* only makes anonymous function calls). This shows that JavaScript in web applications is event driven. We also see that the functions that are not anonymous are seldom repeatedly called (i.e., for *msn* on average there are 104.64 distinct function calls (out of 39 function names) and 15609 anonymous function calls)

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

4.3. Nested function calls

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

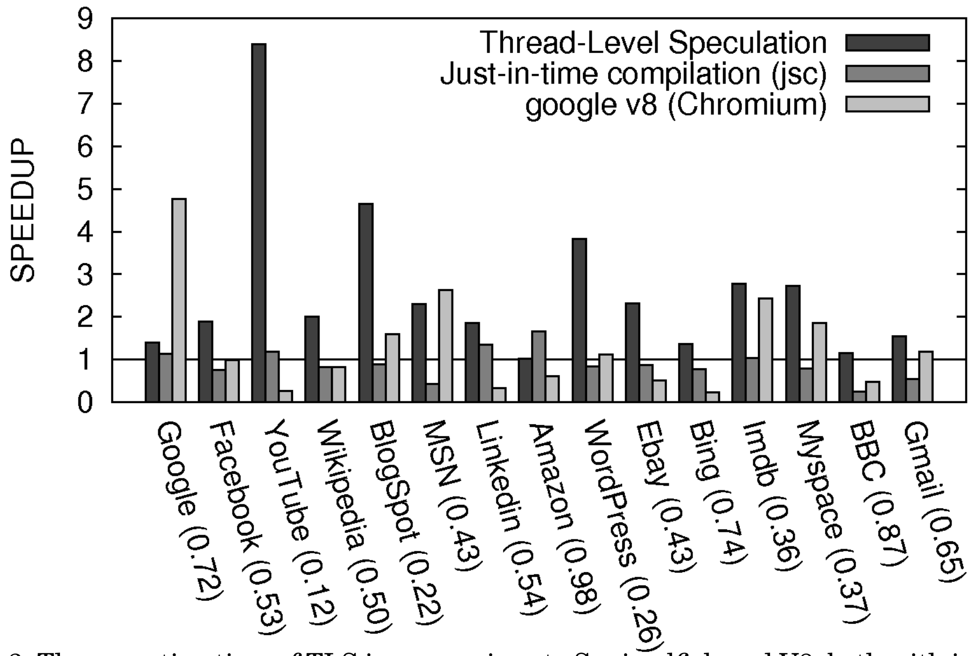

Fig. 3: The execution time of TLS in comparison to Squirrelfish and V8, both with just-in-time compilation enabled [Martinsen et al. 2013] normalized to the execution time of Squirrelfish without JIT.

Fig. 5 shows that the number of functions start to decrease after depth 3. The number of JavaScript functions calls decreases after depth 3, since calls to events in web applications are only allowed to execute for a limited time (i.e., for *youtube* nearly 90% of all the functions calls are made before depth 4). Most web browsers report that the script is unresponsive if the JavaScript executes too long. As the execution progress, so does the depth of function calls, therefore, JavaScript functions with a high depth do not account for most of the execution time in web applications.

4.4. Testing environment

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

5. EXPERIMENTAL RESULTS

In Section 5.1 we have limited the memory used for speculation, in Section 5.2 we have limited the maximum number of concurrent threads, and in Section 5.3 we have limited the speculation depth.

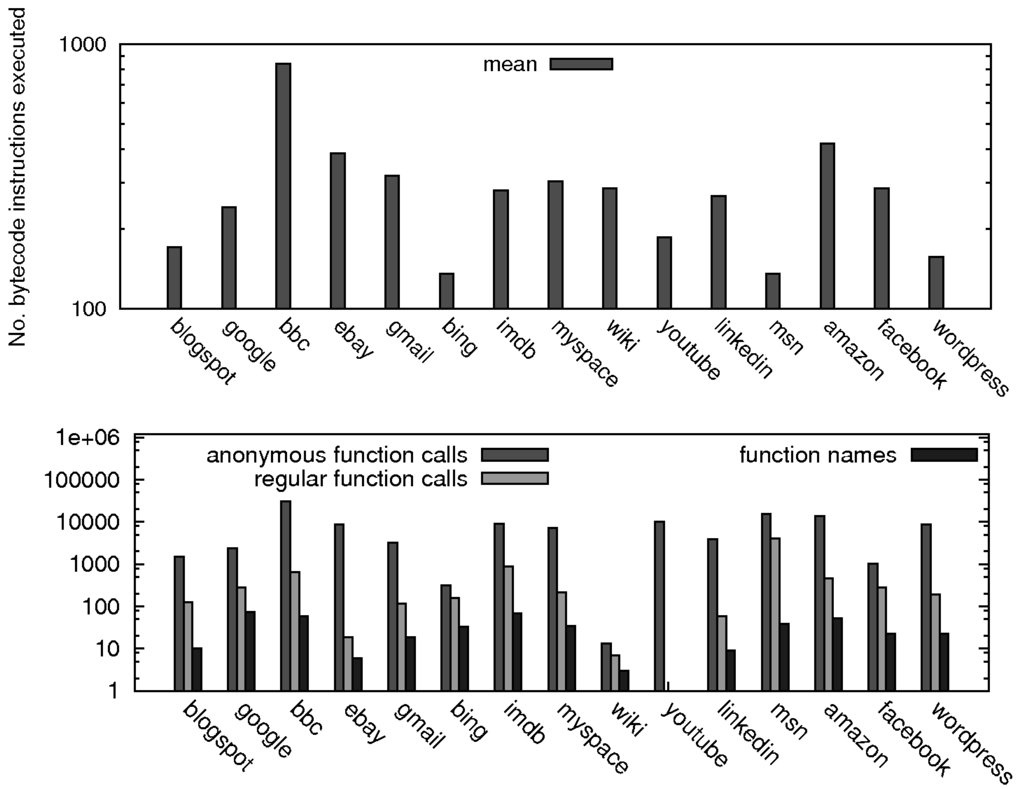

Fig. 4: The mean length of the bytecode instructions executed in functions (upper) and the number of anonymous function calls, number of regular function calls and unique function names for the regular functions (lower).

5.1. Limiting the memory usage

In Fig. 6; (i) the execution time generally decreases with increased memory usage, and (ii) most of the performance increase is achieved between 32 MB and 128 MB.

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

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

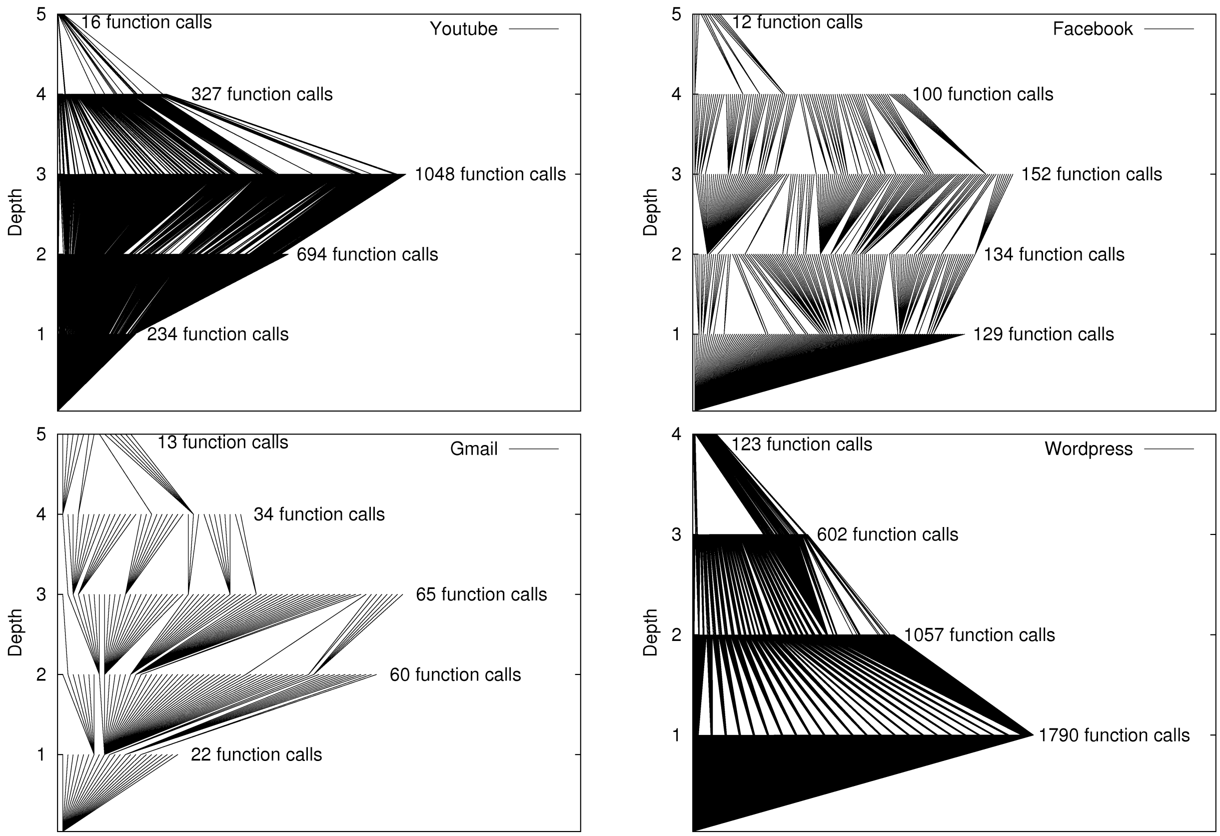

Fig. 5: The number of nested function calls up to 7 levels for *Youtube*, *Facebook*, *Gmail* and *Wordpress* . We see as the depth of the nested function calls increases, the number of function calls decreases. We also see that the largest number of function calls is often not found at depth 1, but rather at depth 2 and depth 3. Each line represents a function call, and we can see by tracing the lines that some of the function calls spawn many new function calls. (For instance such as the number of function calls between depth 2 and depth 3 in *Facebook*). The rightmost number at each depth (vertical y-axis) indicates the number of function calls for each depth. From the Figures, the number of function calls is the largest at a higher depth than 1.

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

Fig. 6 shows that *Youtube* executes 1.86× as fast as the next fastest use cases, *wikipedia*. In *Youtube* there is a large number of identical functions running as events since Fig. 4 shows that all the function calls in this use case are anonymous. These are related to updating and suggesting similar videos to the one the user is currently watching. In Fig. 8 we execute 5.06× as many threads as the average number of threads for this use case. In Fig. 9 the number of speculations is 1.69× as many as the average number of speculations, but 31% of the average number of rollbacks.

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

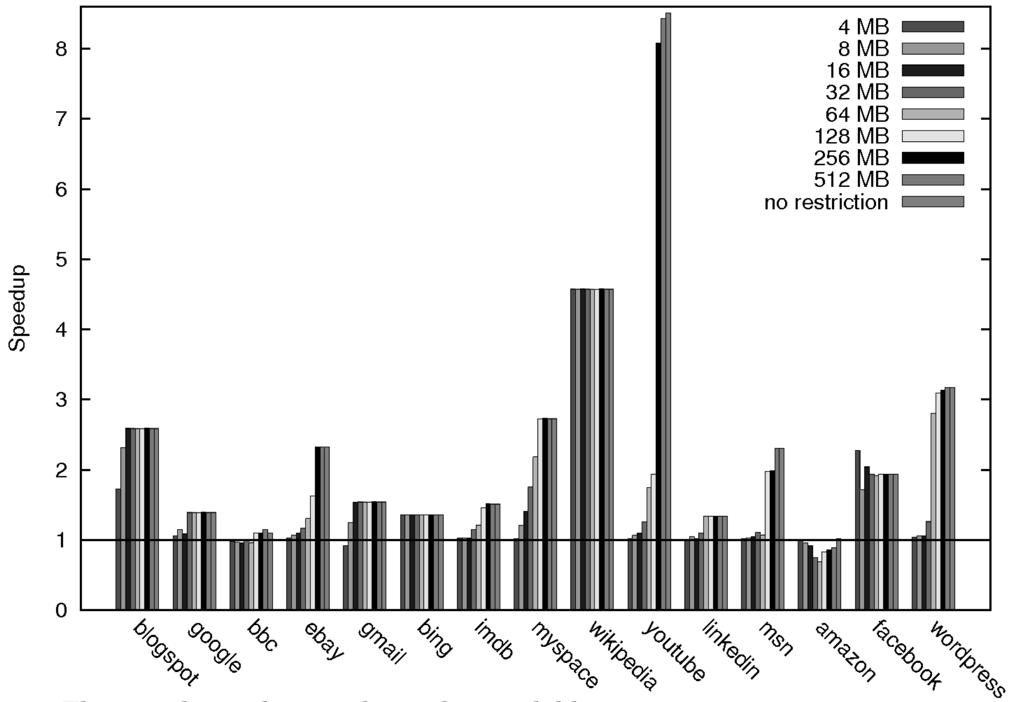

Fig. 6: The speed up when we limit the available memory to 4,8,16,32,64, 128, 256, 512MB and with no restriction on the memory usage. The horizontal line in the figure indicates the sequential execution time for comparison average speed up is 2.52 (excluding the *Youtube* use case, it is 2.09).

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

5.1.2. Overhead of saving checkpoint states and committing values. In Fig. 7 we have measured the relative execution time of TLS relative to the sequential execution time. We have measured the time it takes to commit values and the time it takes to save states when we limit the memory usage to 4, 8, 16, 32, 64, 128, 256 and 512 MB relative to the execution time. Generally, the time it takes to save checkpoint states increases, while the time it takes to commit values when a function returns decreases as the memory usage increases. Therefore we spend less time committing data as the memory increases, but spend more time saving checkpoint states in case of a rollback. The overhead for saving states varies between 24% and 1% of the total execution time, and the overhead of committing values varies between 3% and 0.01%. Thus, the overhead values for TLS is in general very small.

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

5.1.3. No. of threads. In Fig. 8, 5 of the web applications are able to execute more than 50 threads. The functions in JavaScript can execute 2.19 bytecode instructions on a function call. Since we use nested speculation each thread has to wait until the threads it created returns. Due to the large number of function calls in web applications, and

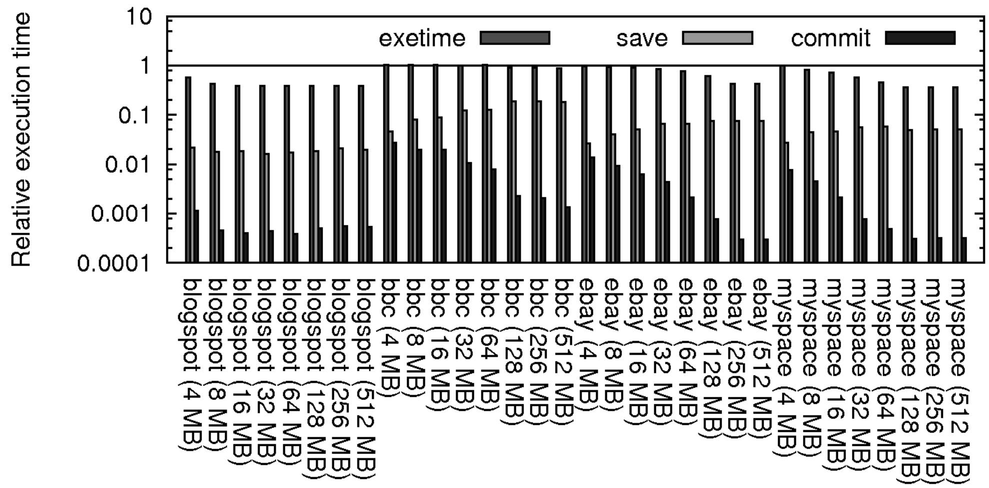

Fig. 7: The relative improvement in execution time and the overhead of saving checkpoint states and commiting values

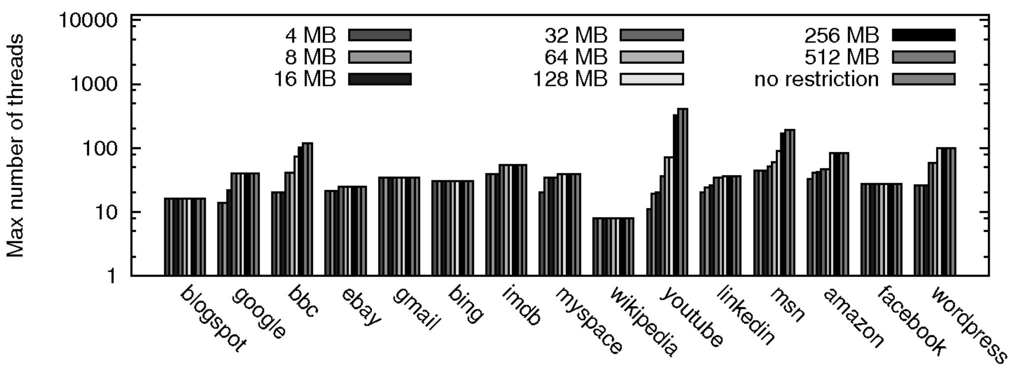

Fig. 8: The largest number of threads when we limit the available memory to 4, 8, 16, 32, 64, 128, 256, 512MB and with no restriction on the memory usage

that functions are quite short; the number of threads running at certain points in time varies greatly. For instance, for *linkedin* the average number of threads executing are 2.64, while the maximum number of threads is 36.

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

5.1.4. No. of speculations and no. of rollbacks. Fig. 9 shows a clear correlation between an increased memory and an increased number of speculations and an increased number of rollbacks. For instance between 4MB and an unrestricted amount of memory we get 16.12$\times$ as many speculations, and 26$\times$ as many rollbacks. However comparing the number of speculations and the number of rollbacks, we find that few of the speculations result in a rollback. For example, *Imdb* makes over 5000 speculations, with less than 150 rollbacks. The behavior of other applications are similar.

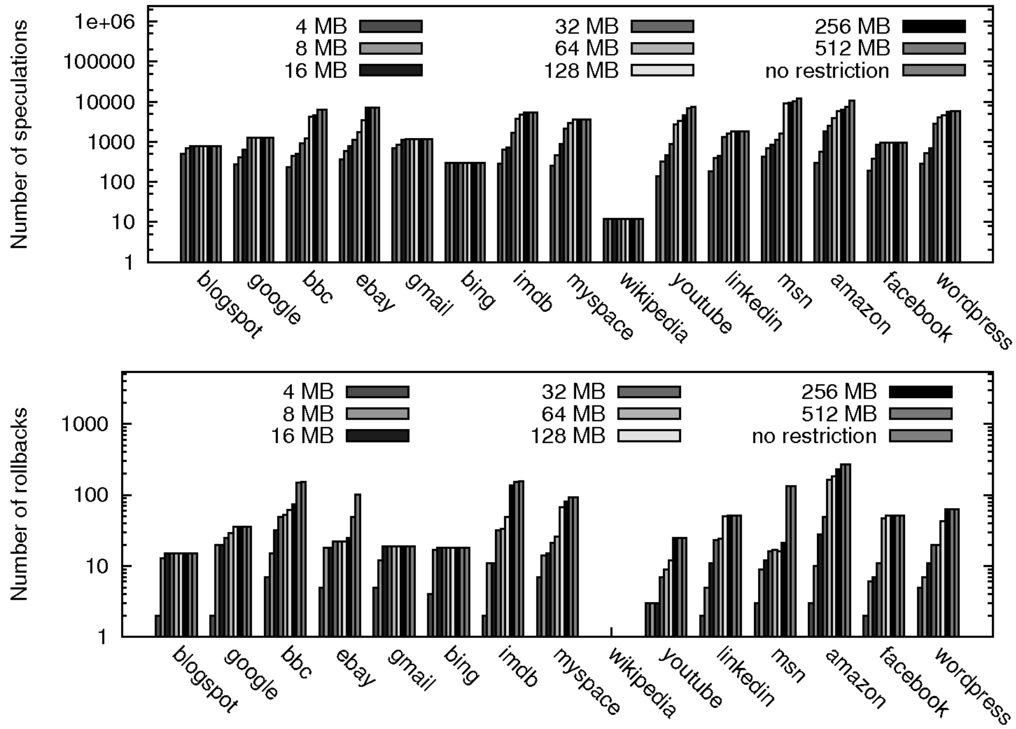

Fig. 9: The number of speculations (upper) and the number of rollbacks (lower) when we limit the memory usage to 4, 8, 16, 32, 64, 128, 256, 512MB and with no restriction on memory usage

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

5.2. Limiting the number of threads

Fig. 10 shows that the optimal number of threads in order to achieve the lowest execution time is between 8 and 32 ant that 8 web applications we have the highest speed up with 16 threads.

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

The execution time decreases for *Youtube* (i.e.,it executes 3.89× faster with 128 threads than 4 threads). There are 1.69× speculations as the other use cases, and 32% of the rollbacks. By inspecting the executed bytecode and the JavaScript code we see that 68% of them have the same JavaScript code, even though they are anonymous.

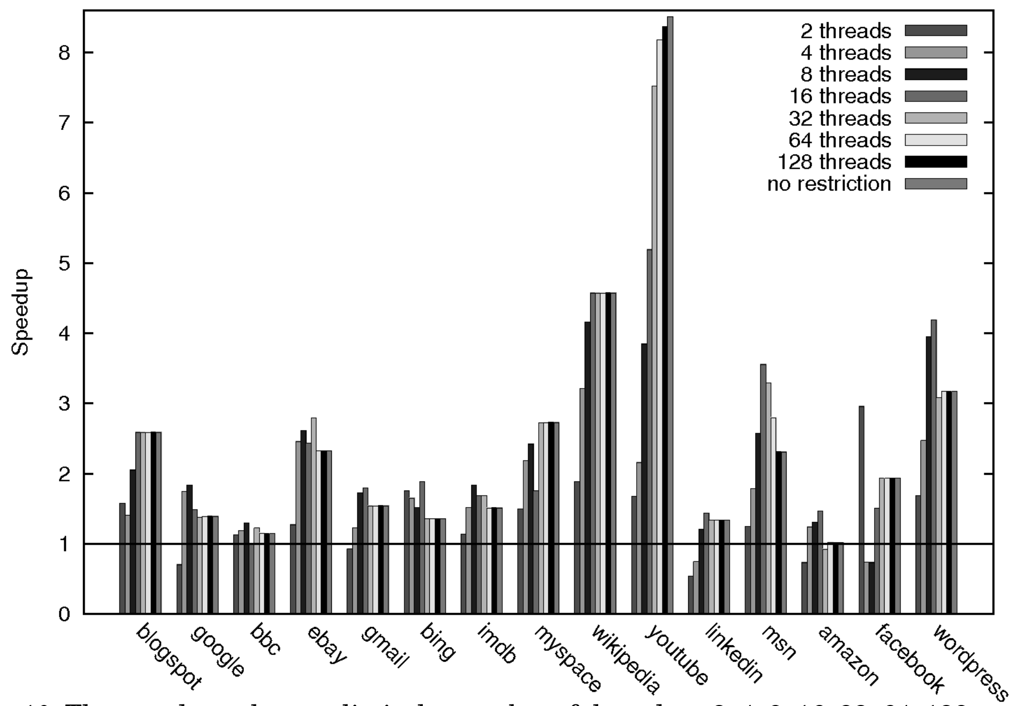

Fig. 10: The speed up when we limit the number of threads to 2, 4, 8, 16, 32, 64, 128 and with no restriction on the maximum number of threads (average speed up, excluding the *Youtube* use case is 2.09).

These are great candidates for being speculatively executed, many of them are events, and since they are anonymous function calls, they do not return anything.

If we limit the number of threads to 2 in *Facebook*, it executes 1.72× faster. We can understand this from the following; by using two threads, the overhead is significantly reduced (29% of when we do not limit the number of threads). In *Facebook*, we are unable to find an increased number of threads executing concurrently when going from 32 to 128 threads. In Fig. 11 there is a 3.2× increase between the number of executing threads going from 128 to no restriction on the number of threads. If we look at the JavaScript execution in *Facebook*, there is a large number of executing functions at each depth. We also see that in Fig. 5 the functions are distributed evenly at each depth. For a limited number of threads, there is a limit to how many functions we can use for nested speculation. Without such a limit, we are able to execute more functions. This does not speed up the execution time. The memory usage of *Facebook* in Fig. 16 suggests that the functions are small in terms of number of executed bytecode instructions, and therefore commit quickly. This enables us to speculate on a lot of functions, but the increase in speculation due to the depth of function in Fig. 5 limits the gain in execution time (even though the number of available threads is very high).

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

For the ones that are slower than sequential execution time, they use between 2 and 8 threads (*Amazon* is slower for 16 threads). We see in Fig. 12 that even though we are using 2 threads, the number of rollbacks is almost the same as for 4 threads, while the number of speculations is much higher for 4 than for 2 threads. This shows that the cost of doing a rollback, along with the lack of speculation using 2 threads makes the execution time slower than the sequential execution time. We see the contrary in *Wikipedia* which has no rollbacks, and therefore the speed up is above the sequential execution time. This suggests that the number of threads must be higher than 2 in order to take advantage of TLS to decrease the execution time, which is an argument for nested speculation.

5.2.2. The ability to take advantage of the threads. Fig. 11 shows that 13 of the use cases are able to execute 32 threads concurrently when going from 16 to 32 threads. For 32 to 64, we are often able to use more than 32 threads, but only 5 use cases are able to use 64 threads. This shows that the real number of threads that we are able to execute concurrently is between 32 and 64. Since we see that for up to 32 threads most use cases are able to double the highest number of threads by adjusting the maximum number of threads, there is rarely any point increasing the maximum number of threads beyond 32. Only *Youtube* and *Wordpress* are able to take advantage of a maximum number of threads over 128. However, their speed up in execution time is negligible for this number of threads compared to 128 threads. *Youtube* is 4% faster, while *Wordpress* is only able to use a large number of threads, not to improve the execution time, because as we increase the number of threads, we are usually able to speculate deeper, however the number of bytecode instructions executed at a high speculation depth is limited. Therefore, there is a limit to how much we are able to speed up the execution time even if we are able to execute more threads.

5.2.3. No. of speculations and rollbacks. Fig. 12 shows that the number of speculations increases by 7.92× with an increasing number of threads. For 12 out of 15 web applications, the number of speculations does not increase when the maximum number of threads is higher than 16. This shows that we are unable to find a sufficient number of functions to execute concurrently.

From the number of rollbacks there is often an over 3× increase in the number of rollbacks going from 2 to 8 threads. However, there is often a decrease in the number of rollbacks as the number of threads increases from 16 up to no limitation on the number of threads. This pattern is common; first the number of rollbacks increases, then the number of rollbacks gradually decreases as the number of threads increases. In Fig. 12, there is not a clear correlation between an increased number of speculations and an increased number of rollbacks. This indicates that a larger number of threads does not necessarily mean a larger number of rollbacks, In fact, it might mean the opposite, and an increased number of threads might reduce the number of rollbacks. We get 3.33× the speculations when we limit the number of threads to 4 compared to when we limit the number of threads to 2. The significant change in the number of speculations is because of the reduction in the number of available threads. In Fig. 4 we see that there are many functions in web applications, but that they are small. Then we could end up executing many small functions with a limited number of threads, which would have a marginal effect on the execution time. This is the reason why we need a certain number of threads to improve the execution time. If we reduce the number of cores on the system (i.e., two or four) we end up using more threads to have the same execution time as using eight cores.

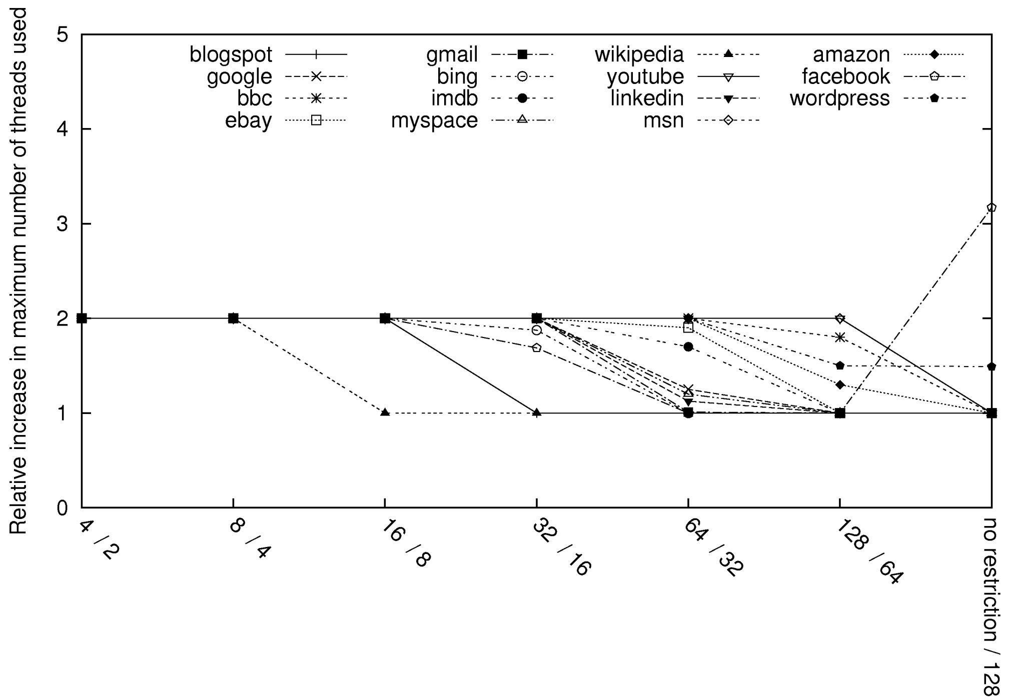

Fig. 11: The relative increase in the highest number of concurrent threads increasing the maximum number of threads from 2 to 4, 4 to 8, 8 to 16, 16 to 32, 32 to 64, 64 to 128 and 128 to no limitation.

As we restrict the number of threads, the speculation depth decreases. This makes us unables us take full advantage of nested speculation. In Fig. 12 the number of speculations increases as the number of threads increases. However, JavaScript TLS characteristics in web applications also indicate that the number of bytecode instructions decreases as the depth increases. This shows that as the depth increases, a large number of functions are able to execute simultaneously, and that the functions are often able to commit quicker. This reduces the number of dependencies between speculated functions, which in turn reduces the number of rollbacks. In addition the number of anonymous functions of JavaScript in web applications show that there are few return values.

5.2.4. Memory usage. In Fig. 13 we see that as we increase the number of threads we increase the memory usage by 8.69×. For example, the extremes are *msn* and *Amazon* that use more than 937MB and 1.5GB of memory if we do not limit the maximum number of threads. One interesting use case is *Google*, where the memory increases with 1024× when we do not restrict the number of threads. However, these threads are very small in terms of bytecode instructions, but by not restricting the number of threads, we are able to speculate multiple threads in a nested manner, which in turn increases the memory usage.

The results show that uncritically increasing the number of threads only has the lowest execution time for 3 out of 15 use cases, and has a high cost in terms of memory. The optimal number of threads to decrease the execution time seems to be between 8

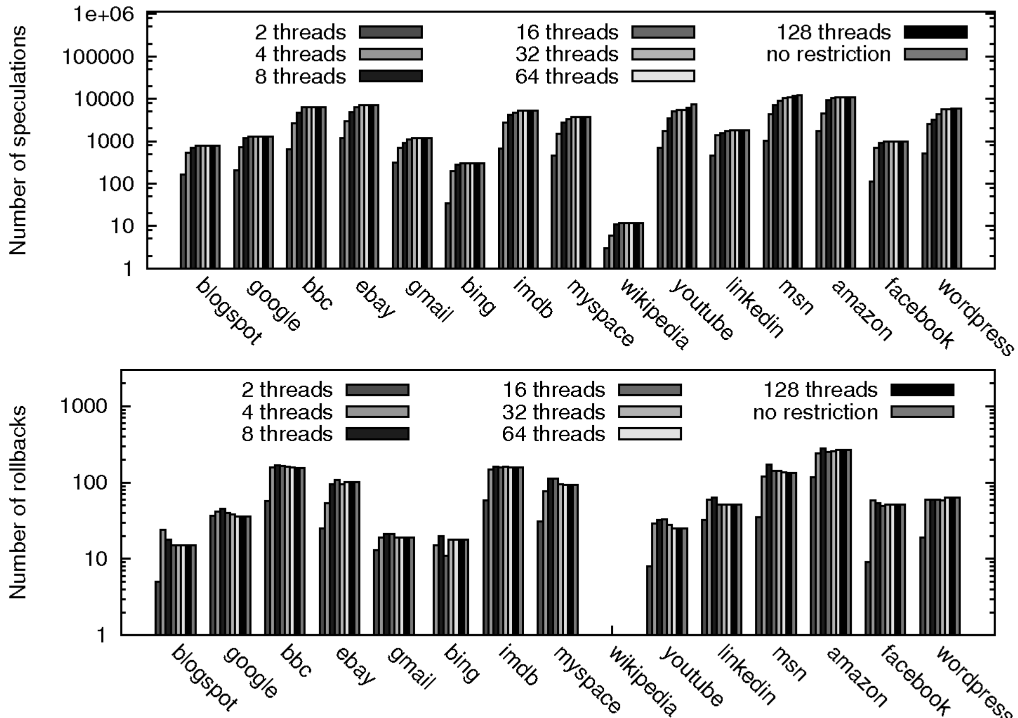

Fig. 12: The number of speculations (upper) and the number of rollbacks (lower) when we limit the maximum number of threads to 2, 4, 8, 16, 32, 64, 128, and with no restrictions on the number of threads.

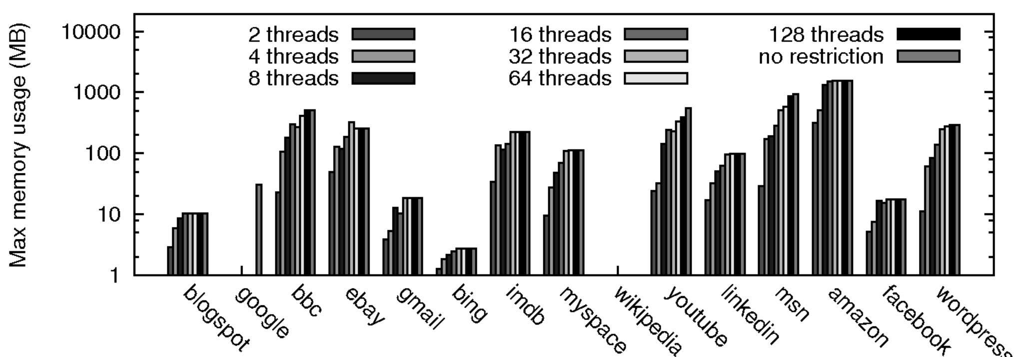

Fig. 13: The memory usage when we limit the number of threads to 2, 4, 8, 16, 32, 64, 128 and with no restriction on the maximum number of threads.

and 32. A maximum number of threads set to less than 8 indicates that we are unable to create a sufficient number of threads (e.g. *linkedin*).

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

5.3. Limiting the speculation depth

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

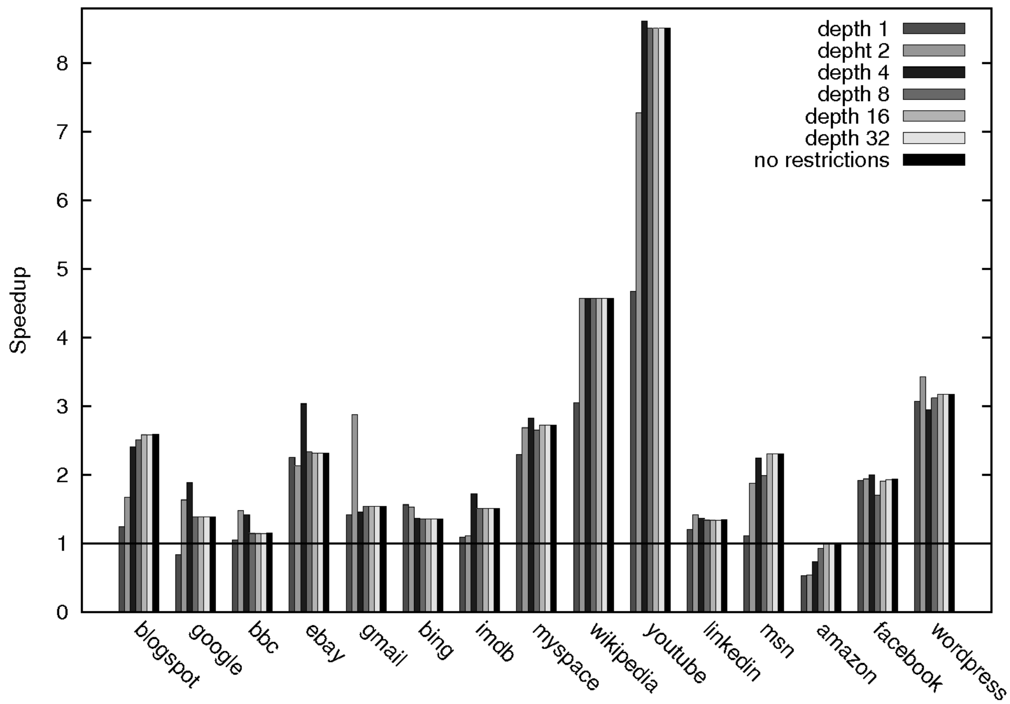

Fig. 14: The speed up when we limit the speculation depth to 2, 4, 8, 16, 32, and with no restriction on the depth (average speed up when we exclude the *Youtube* use-case is 2.34).

5.3.1. Execution time. Fig. 14 shows that nested speculation is necessary to improve the execution time

With a speculation depth of 2 for *Gmail*, it is 52% faster than when we do not limit the speculation depth. In Fig. 15 the number of speculations for *Gmail* is the highest for speculation depth 2, and the number of rollbacks is the lowest. The memory usage is lower for depth 2, which decreases the overhead of TLS. The behavior in *Gmail* is caused by much JavaScript functionality (compared to some of the other use cases) executed when the page loads. Further JavaScript execution is caused by more user interaction. Our use cases have reduced user interaction, therefore we would probably see at better effect with more user interactions. In Fig. 5 most of the functions are found at depth 2 and 3. This explains the large speed up of *Gmail* at depth 2.

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

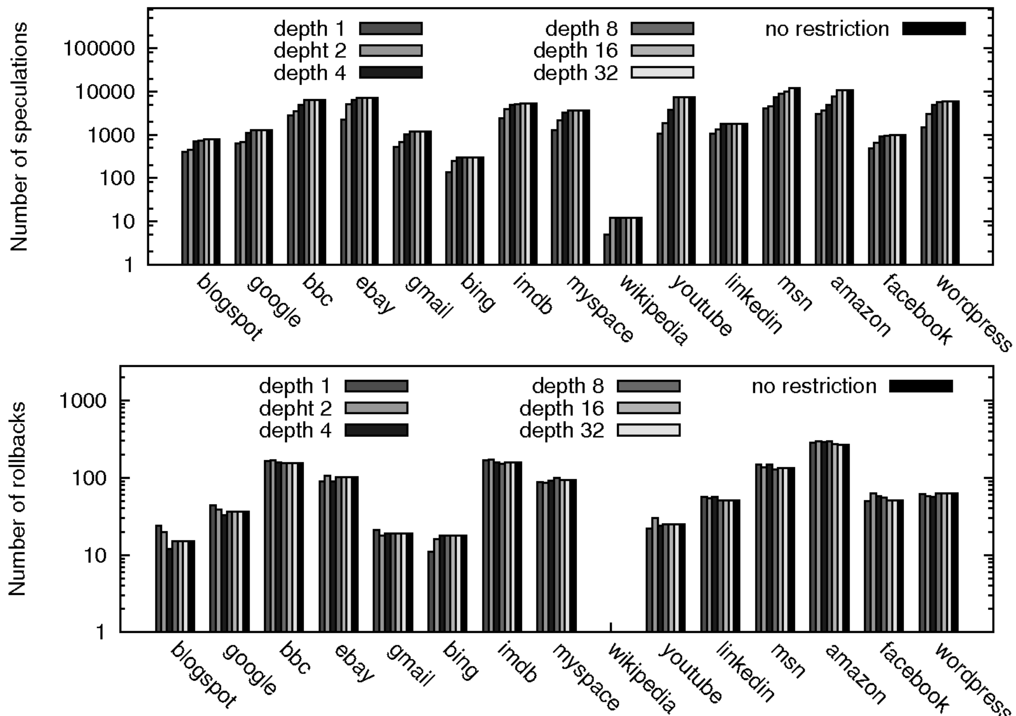

Fig. 15: The number of speculations (upper) and rollbacks (lower) when we limit the depth to 2, 4, 8, 16, 32, and with no restriction on the depth.

5.3.2. No. of speculation and no. of rollbacks. Fig. 15 shows that there is a relationship between an increased speculation depth and an increased number of speculations, although there is a limit to the number of speculations we are able to make with a speculation deeper than 8. We execute fewer and fewer bytecode instructions as the speculation depth increases ,since the number of JavaScript functions decreases as the speculation depth increases (Fig. 5). This means that the potential gain of speculation decreases, as the number of functions and the size of each function decreases, while we save more states. Therefore, the speed up rarely increases with a speculation depth higher than 4.

For a speculation depth over 8, the number of rollbacks decreases as the speculation depth increases. Since the size of the functions decreases, they commit back to the parent faster than they would if the size of the function was bigger. Given that a function speculates on a new function (i.e., nested speculation) it has fewer dependencies between itself and the function it speculates on, than there is between two functions which have the same depth (i.e., for instance function calls that are made as part of a loop). These functions, rarely return a value, or at least one that we were unable to predict correctly. This is because many of these functions read elements in the DOM tree.

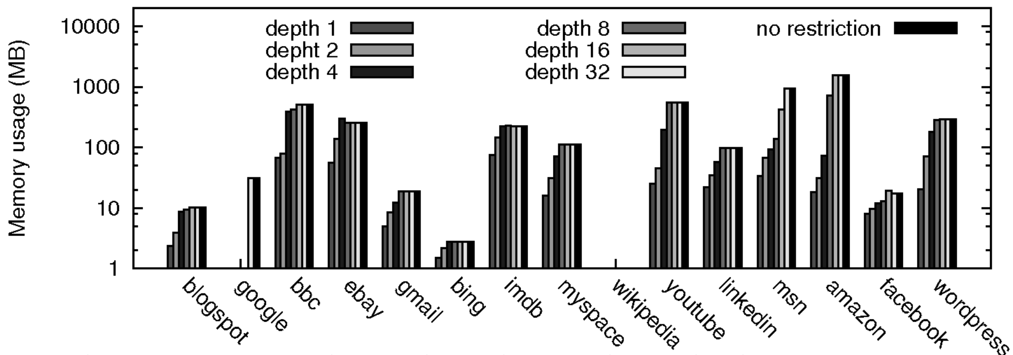

Fig. 16: The memory usage when we limit the speculation depth to 2, 4, 8, 16, 32, and with no limit on the depth.

5.3.3. Memory usage. In Fig. 16 we see that an increased speculation depth means more speculations (Fig. 5), and as a result more checkpoints states must be saved. This means that we get an increased overhead of saving the checkpoint states, relative to a lower depth. However there are a lower number of variable checks as the number of bytecode instructions decreases as the depth increases, and the functions commit earlier.

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

6. DISCUSSION

As observed in Section 5.1 and Section 5.3 both *Amazon* use cases can be slower than the sequential execution. This is because when we limit the memory, we are often unable to speculate deep enough, which in turn could slow down the execution. When we increase the speculation depth, the execution time improves. In Section 5.2 we limit the number of threads, then the same use cases are often able to find the correct threads to speculate on, and initially the overhead is reduced so we get the highest speed up.

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

To get the bound of improved execution time of JavaScript using TLS in web applications, we compare our results against the results of [Fortuna et al. 2010]. Their average speed up is 8.9× faster which is clearly faster than the results in this paper, but they make their argument from a theoretical point of view. Our use cases are methodologically performed with a focus on reproducibility [Martinsen and Grahn 2011]. This causes our use cases to have less JavaScript execution, and fewer JavaScript functions to speculate on.

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

7. CONCLUSION

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

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

REFERENCES

ALEXA. 2010. Top 500 sites on the web. `http://www.alexa.com/topsites`.

BERRY, M., CHEN, D., KOSS, P., KUCK, D., LO, S., PANG, Y., ROLOFF, R., SAMEH, A., CLEMENTI, E., CHIN, S., SCHNEIDER, D., FOX, G., MESSINA, P., WALKER, D., HSIUNG, C., ADN K. LUE, J. S., ORZAG, S., SEIDL, F., JOHNSON, O., SWANSON, G., GOODRUN, R., AND MARTIN, J. 1989. The PERFECT Club Benchmarks: Effective performance evaluation of supercomputers. Tech. Rep. CSRD-827, Center for Supercomputing Research and Development, Univ. of Illinois, Urbana-Champaign. May.

BHOWMIK, A. AND FRANKLIN, M. 2002. A general compiler framework for speculative multithreading. In *SPAA '02: Proc. of the 14th ACM Symp. on Parallel Algorithms and Architectures*. 99–108.

BRAND, J. AND BALVANZ, J. 2005. Automation is a breeze with autoit. In *SIGUCCS '05: Proc. of the 33rd Annual ACM SIGUCCS Conf. on User services*. ACM, New York, NY, USA, 12–15.

BRUENING, D., DEVABHAKTUNI, S., AND AMARASINGHE, S. 2000. Softspec: Software-based speculative parallelism. In *FDDO-3: Proceedings of the 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization*.

CHAUDHRY, S., CYPHER, R., EKMAN, M., KARLSSON, M., LANDIN, A., YIP, S., ZEFFER, H., AND TREMBLAY, M. 2009. Rock: A High-Performance Sparc CMT Processor. *IEEE Micro 29,* 2, 6–16.

CHEN, M. K. AND OLUKOTUN, K. 1998. Exploiting method-level parallelism in single-threaded Java programs. In *Proc. of the 1998 Int'l Conf. on Parallel Architectures and Compilation Techniques*. 176.

CHEN, M. K. AND OLUKOTUN, K. 2003. The Jrpm system for dynamically parallelizing Java programs. In *ISCA '03: Proc. of the 30th Int'l Symp. on Computer Architecture*. 434–446.

CINTRA, M. AND LLANOS, D. R. 2003. Toward efficient and robust software speculative parallelization on multiprocessors. In *PPoPP '03: Proc. of the 9th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*. 13–24.

FORTUNA, E., ANDERSON, O., CEZE, L., AND EGGERS, S. 2010. A limit study of javascript parallelism. In *2010 IEEE Int'l Symp. on Workload Characterization (IISWC)*. 1–10.

GOOGLE. 2012. V8 JavaScript Engine. `http://code.google.com/p/v8/`.

HERTZBERG, B. AND OLUKOTUN, K. 2011. Runtime automatic speculative parallelization. In *Proc. of the 9th Annual IEEE/ACM Int'l Symp. on Code Generation and Optimization (CGO)*. 64–73.

HU, S., BHARGAVA, R., AND JOHN, L. K. 2003. The role of return value prediction in exploiting speculative method-level parallelism. *Journal of Instruction-Level Parallelism 5*.

JAVASCRIPT. 2010. `http://en.wikipedia.org/wiki/JavaScript`.

KAZI, I. H. AND LILJA, D. J. 2000. JavaSpMT: A speculative thread pipelining parallelization model for java programs. In *IPDPS'00: Proceedings of the 14th International Parallel and Distributed Processing Symposium*. IEEE Computer Society, Los Alamitos, CA, USA, 559.

KAZI, I. H. AND LILJA, D. J. 2001. Coarse-grained thread pipelining: A speculative parallel execution model for shared-memory multiprocessors. *IEEE Trans. on Parallel and Distributed Systems 12,* 9, 952–966.

LATTNER, C. AND ADVE, V. 2004. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. CGO '04. IEEE Computer Society, Washington, DC, USA, 75–.

MARTINSEN, J. K. AND GRAHN, H. 2010. An alternative optimization technique for JavaScript engines. In *Third Swedish Workshop on Multi-Core Computing (MCC-10)*. 155–160.

MARTINSEN, J. K. AND GRAHN, H. 2011. A methodology for evaluating JavaScript execution behavior in interactive web applications. In *Proc. of the 9th ACS/IEEE Int'l Conf. On Computer Systems And Applications*. 241–248.

MARTINSEN, J. K., GRAHN, H., AND ISBERG, A. 2011. A comparative evaluation of JavaScript execution behavior. In *Proc. of the 11th Int'l Conf. on Web Engineering (ICWE 2011)*. 399–402.

MARTINSEN, J. K., GRAHN, H., AND ISBERG, A. 2013. Using speculation to enhance javascript performance in web applications. *Internet Computing, IEEE 12,* 4, 37–45.

MEHRARA, M., HSU, P.-C., SAMADI, M., AND MAHLKE, S. 2011. Dynamic parallelization of JavaScript applications using an ultra-lightweight speculation mechanism. In *Proc. of the 17th Int'l Symp. on High Performance Computer Architecture*. 87–98.

MEHRARA, M. AND MAHLKE, S. 2011. Dynamically accelerating client-side web applications through decoupled execution. In *Proc. of the 9th Annual IEEE/ACM Int'l Symp. on Code Generation and Optimization (CGO)*. 74–84.

MICKENS, J., ELSON, J., HOWELL, J., AND LORCH, J. 2010. Crom: Faster web browsing using speculative execution. In *Proc. of the 7th USENIX Symp. on Networked Systems Design and Implementation (NSDI 2010)*. 127–142.

MOZILLA. 2012. SpiderMonkey – Mozilla Developer Network. `https://developer.mozilla.org/en/SpiderMonkey/`.

OANCEA, C. E., MYCROFT, A., AND HARRIS, T. 2009. A lightweight in-place implementation for software thread-level speculation. In *SPAA '09: Proc. of the 21st Symp. on Parallelism in Algorithms and Architectures*. 223–232.

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

PICKETT, C. J. F. AND VERBRUGGE, C. 2005b. Software thread level speculation for the Java language and virtual machine environment. In *LCPC '05: Proc. of the 18th Int'l Workshop on Languages and Compilers for Parallel Computing*. 304–318. LNCS 4339.

PRABHU, M. K. AND OLUKOTUN, K. 2005. Exposing speculative thread parallelism in SPEC2000. In *Proc. of the 10th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*. 142–152.

RATANAWORABHAN, P., LIVSHITS, B., AND ZORN, B. G. 2010. JSMeter: Comparing the behavior of JavaScript benchmarks with real web applications. In *WebApps'10: Proc. of the 2010 USENIX Conf. on Web Application Development*. 3–3.

RENAU, J., STRAUSS, K., CEZE, L., LIU, W., SARANGI, S. R., TUCK, J., AND TORRELLAS, J. 2006. Energy-efficient thread-level speculation. *IEEE Micro 26,* 1, 80–91.

RENAU, J., TUCK, J., LIU, W., CEZE, L., STRAUSS, K., AND TORRELLAS, J. 2005. Tasking with out-of-order spawn in tls chip multiprocessors: Microarchitecture and compilation. In *In ICS*. 179–188.

RICHARDS, G., LEBRESNE, S., BURG, B., AND VITEK, J. 2010. An analysis of the dynamic behavior of JavaScript programs. In *PLDI '10: Proc. of the 2010 ACM SIGPLAN Conf. on Programming Language Design and Implementation*. 1–12.

RUNDBERG, P. AND STENSTRÖM, P. 2001. An all-software thread-level data dependence speculation system for multiprocessors. *Journal of Instruction-Level Parallelism*, 1–28.

STANDARD PERFORMANCE EVALUATION CORPORATION. 2000. SPEC CPU2000 v1.3. http://www.spec.org/cpu2000/.

STEFFAN, J. G., COLOHAN, C., ZHAI, A., AND MOWRY, T. C. 2005. The STAMPede approach to thread-level speculation. *ACM Transactions on Computer Systems 23,* 3, 253–300.

TIAN, C., FENG, M., NAGARAJAN, V., AND GUPTA, R. 2008. Copy or discard execution model for speculative parallelization on multicores. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 41. IEEE Computer Society, Washington, DC, USA, 330–341.

W3C. 2011. Web Workers — W3C Working Draft 01 September 2011. `http://www.w3.org/TR/workers/`.

WEBKIT. 2012. The WebKit open source project. `http://www.webkit.org/`.