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

***Abstract*—Thread-Level Speculation has been used to take advantage of multicore processors in virtual execution environments for the sequential JavaScript scripting language. While the results are promising the memory overhead has so far been high. In this paper, we make the following contributions: (i) We propose to reduce memory usage by limiting the checkpoint depth, (ii) we present an in-depth study of the effects of limiting the checkpoint depth in Thread-Level Speculation, and (iii) we propose an adaptive heuristic to dynamically adjust the number of checkpoints. We evaluate our techniques using 15 web applications on an 8-core computer. The results show that we reduce the memory overhead for Thread-Level Speculation by over 90% as compared to storing all checkpoints. At the same time, the performance is often better than when we store all checkpoints and at worst 4% slower.**

I. Introduction

JavaScript is a dynamically typed, object-based scripting language with run-time evaluation used extensively in web applications, where the execution is done in a JavaScript engine such as Mozilla Spidermonkey [1]. Google [2] has suggested Just-in-time compilation (JIT) to decrease the execution time in JavaScript. However, Google's JavaScript engine V8's decrease in execution time has been measured on a set of benchmarks, which Ratanaworabhan et al. [3] show are unrepresentative for real-world web applications. Martinsen et al. [4] show the dramatic effect of this as JIT speeds up the execution of benchmarks, but often *slows down* the execution time in popular web applications.

JavaScript is a sequential scripting language and cannot take advantage of multicore processors to reduce the execution time. Fortuna et al. [5] show that there exists a significant potential for parallelism in many web applications with an estimated speedup of up to 45× compared to a sequential execution.

To hide the details of the underlying parallel hardware, we can dynamically extract parallelism from a sequential program using Thread-Level Speculation (TLS). Mehrara et al. show the performance potential of TLS in the SpiderMonkey JavaScript engine on a series of well-known benchmarks [6] and Martinsen et al. [7] show this on a number of popular web applications. However, while prior results show that TLS can significantly speed up the execution time in web applications, it uses over 1500 MB of memory.

We propose to reduce the memory overhead in TLS by being selective on when we store the checkpoints before we speculate. Martinsen et al. [7] show that less than 1.8% of all the speculations result in a rollback. However, if we choose to store the checkpoint far away from the rollback, the number of bytecode instructions that need to be re-executed increases. Martinsen et al. [8] show that nested speculation is necessary to decrease the execution time; therefore we investigate the effects on memory usage and execution time of not storing the checkpoints at all speculation depths. Further, we propose an adaptive heuristic that dynamically adjusts when we store the checkpoints depending on the speculation depth and the number of rollbacks.

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

Our main contributions are:

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

This paper is organized as follows; in Section II, we introduce JavaScript, web applications and TLS. In Section III, we present the TLS implementation used in our study. In Section IV we present our approaches to reducing the memory usage in TLS. In Section V, we present the experimental methodology, while in Section VI and Section VII we evaluate the effects of a fixed number of checkpoints and the adaptive heuristic. Finally, in Section VIII we conclude our findings.

II. Background and related work

A. JavaScript and web applications

JavaScript is a dynamically typed, object-based scripting language with run-time evaluation used in web applications. JavaScripts performance has reached a high single-thread performance on a set of benchmarks. Ratanaworabhan et al. [3] show that the results from these benchmarks are misleading for the execution behavior of web applications and Martinsen et al. [7] show that optimizing towards the characteristics of the benchmarks slows down the web applications.

Web applications manipulate parts that are not accessible from a JavaScript engine. The scripted functionality is executed in a JavaScript engine, but the program flow is defined in the web application. Richards et al. [9] show that web applications use JavaScript specific features extensively such that various parts of the program are defined at run-time.

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

B. Thread-Level Speculation principles

Picket and Verbrugge's [10] TLS approach is to allocate each function call in software as a thread. Then, they can (ideally) execute as many function calls in parallel as they have processors. However data dependencies and return values limits the number of function calls that can be executed in parallel. Martinsen et al. [7] show that the memory requirements and Rudberg et al. [11] show that the run-time overhead for detecting data dependencies can be considerable.

Between two functions we can have three types of data dependencies: *Read-After-Write* (RAW), *Write-After-Read* (WAR), and *Write-After-Write* (WAW). A TLS implementation must be able to detect these dependencies during run-time using information about read and write addresses from each function call. A design parameter for TLS is the *precision* of at what granularity of true-positive / (true-positives + false-positives) data dependency violations are detected.

When a data dependency violation is detected, the execution must be rolled back to a safe point in the execution. Thus, all TLS systems need a rollback mechanism. In order to be able to do rollbacks, we need to store both speculative updates of data as well as the original data values. The bookkeeping results in both a memory overhead as well as a run-time overhead. In order for TLS to be fast, the number of rollbacks should be low.

The more precise tracking of data dependencies, the larger memory overhead is required. One effect of imprecise dependence detection is the risk of a false-positive violation, i.e., when a dependence violation is detected when no actual (true-positive) dependence violation is present. As a result, unnecessary rollbacks are done, which increase the execution time. TLS implementations differ depending on whether they update data speculatively 'in-place', i.e., moving the old value to a buffer and writing the new value directly, or in a speculation buffer.

C. Thread-Level Speculation in JavaScript and for web applications

Mehrara and Mahlke [12] target trace-based JIT-compiled JavaScript code, where the most common execution flow is compiled into an execution trace. Run-time checks (guards) are inserted to check whether control flow etc. is still valid for the trace or not. They execute the run-time checks (guards) in parallel with the main execute flow (trace), and have one single main execution flow.

Mehrara et al. [6] introduce a lightweight speculation mechanism that focuses on loop-like constructs in JavaScript. As this code uses the trace feature of Spidermonkey, a selective form of speculation is employed.

Mickens et al. [13] suggest an event-based speculation mechanism which is deployed as a JavaScript library (Crom) which clones certain regions of the JavaScript code that are executed speculatively.

Martinsen et al's [7] approach is to execute the main execution flow in parallel which they evaluate on popular web applications and show that there is a significant potential for TLS in web applications. Figure 1 shows that Just-in-time compilation (JIT) increases the execution time of web applications as compared to interpretive execution, and shows that neither the Squirrelfish or V8 JavaScript engine improves execution time for JavaScript in web applications with JIT. They also show that nested TLS is necessary in order to improve the execution time with TLS.

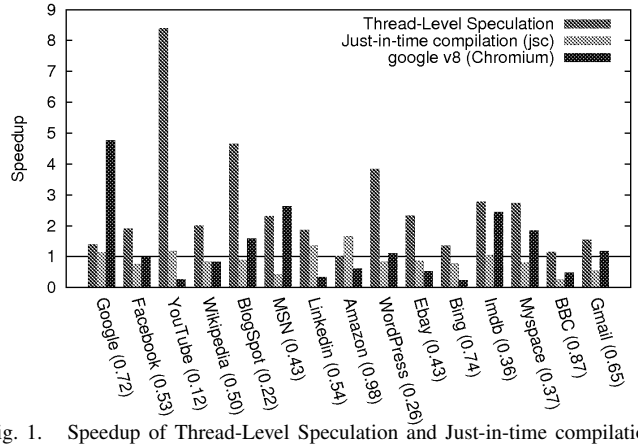

Fig. 1. Speedup of Thread-Level Speculation and Just-in-time compilation for a number of popular web applications. The black horizontal line is the sequential execution time of Squirrelfish without Thread-Level Speculation (the figure is used with permission from the authors of [7]).

Martinsen et al. [14] suggest three heuristics for re-speculation on previous mis-speculations. Their conclusion is that the overall problem with TLS in JavaScript for web applications is not the number of rollbacks but the memory usage.

Martinsen et al. [8] show that by limiting the number of threads, the amount of memory, and the speculation depth we both save memory and improve the execution time. In many cases a speculation depth of 2 to 4 is sufficient to improve the performance because of the nature of JavaScript execution in web applications. JavaScript execution in web applications as events are restricted to be executed for no more than half a minute. This indicates the lack of large loop structures, which again reduces the effect of JIT.

In summary, Martinsen et al's [7] over 1500 MB memory usage in TLS is a concern. We have not found any studies that look at reducing the memory overhead of TLS in web applications by being restrictive on when we store checkpoints.

III. TLS IMPLEMENTATION FOR JAVASCRIPT

Martinsen et al. [7] implemented TLS in the Squirrelfish JavaScript engine. The speculation is done on JavaScript func-

tions in a nested manner, including return value prediction, and all data conflicts are detected at run-time. When a conflict is detected, a rollback is performed. We extend Martinsen et al.'s implementation with a mechanisms to limiting the number of checkpoints, and use it for the measurement and the analysis.

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

When we execute a *op_call* bytecode instruction we extract the *realtime* value and the id of the speculated function that makes this function call, e.g., *p_0220* (a function is called after 220 bytecode instructions from *p_0*). In Figure 2 the value of the position of this function call emulates the sequential execution order for TLS. This is possible in web applications since there is going to be a large number of JavaScript function calls. We check if this function previously has been speculated by looking up the value of *previous[function_order]*. *previous* is a vector where each element is indexed by the *function_order*. If the value is 1, then the function has been speculated unsuccessfully. If not, this function call is a candidate for speculation.

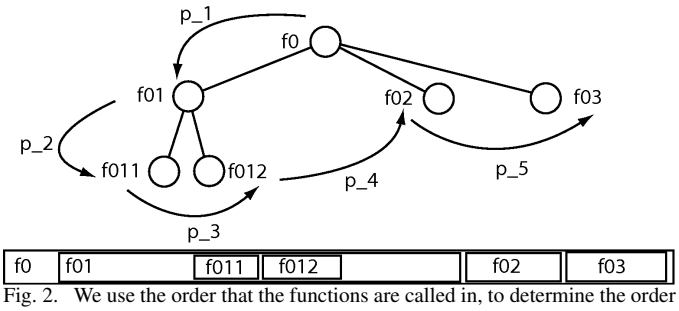

Fig. 2. We use the order that the functions are called in, to determine the order in which the program would have been sequentially executed. This works in JavaScript in a web applications setting, as there are multiple function calls. For instance, in a simplified example the JavaScript function f0, performs 3 function calls, f01, f02 and f03. f01 performs two function calls, f011 and f012. Thus, we have created a speculation tree from the function calls. If we traverse this tree from left to right, we get an order in which the functions are called, equal to the order in which the functions would be sequentially called, in order to uphold the sequential semantics during execution with TLS. More specific: f0 at time *p_1*, f01 at time *p_2*, f011 at time *p_3*, f012 at time *p_4*, f02 at time *p_5*, and f03 at time *p_6*. We denote how each function is ordered as *function_order*

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

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

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

IV. Reducing the memory usage for TLS

A. Motivation for limiting the checkpoint depth

In nested TLS, we store a checkpoint before we speculatively execute a function. The checkpoint is used if we mis-speculate and need to perform a rollback, e.g., due to a data conflict. However, when the speculation is correct, then the stored checkpoint is removed when we commit back to it's parent thread. Martinsen et al. [7] show that 1.8% of the speculations result in a rollback, therefore most checkpoints are never used. The number in the parenthesis in Figure 6 shows that the memory usage can be 1527 MB for TLS.

Martinsen at al. [8] show that nested speculation is necessary to reduce the execution time, and that the speculation depth for 85% of all functions is between 2 and 4. Therefore, most rollbacks occur between depth 2 and 4, and as a result, it could be more meaningful to store the checkpoints below these depths as we expect a rollback at such a depth.

B. Fixed checkpoint depth limit

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

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

Before a speculation, a predefined *checkpoint depth limit* is compared to the function's checkpoint depth. If the checkpoint depth is equal or below the checkpoint depth limit, we store the checkpoint. If the value of the checkpoint depth is higher than the checkpoint depth limit, we do not store the checkpoint, instead, on rollbacks we go to the previous stored checkpoint. This reduces the memory as we store a lower number of checkpoints, but this also means that rollbacks require a larger number of bytecode instructions to be re-executed. An example is shown in Figure 3.

C. An adaptive heuristic

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

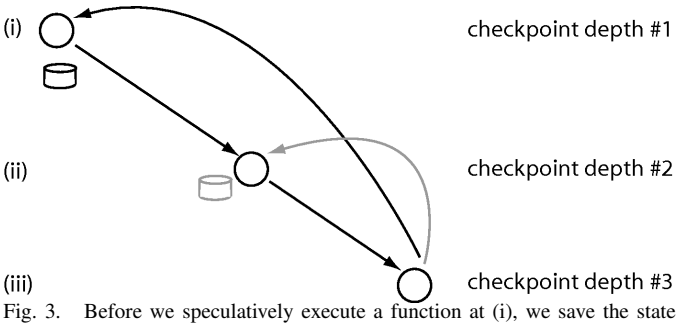

Fig. 3. Before we speculatively execute a function at (i), we save the state so we can rollback to this point. At (ii), i.e., the speculative function made as a speculative function call at (i), we speculatively execute another function call (iii). In normal TLS, we also save the state in (ii) in case of a rollback. In our proposal, we do not store the state at checkpoint in (ii) if the checkpoint depth is set to 1. If a rollback occurs in (iii), we would normally rollback to (ii). However, in our proposal we would rollback to (i). As a result, we do not need to store the checkpointed state in (ii), with the cost of doing a rollback back to (i) instead of to (ii).

depth limit based on the speculation depth and rollback behavior of a web application.

The heuristic in Listing 1 speeds up the execution time up to 8× and reduces the memory usage by over 90%, by being selective at which checkpoint depth limit we store the checkpoints. Martinsen et al. [7] show that as the speculation depth increases we have more rollbacks. If a rollback occurs, we want to reduce the number of bytecode instructions that needs to be re-executed. Martinsen et al. [7] show that rollbacks are rare, but that they often occur between speculative functions with the same depth and occur closely after each other. Therefore, when a rollback occurs, we want to increase the limit to ensure that the number of re-executing bytecode instructions is reduced for preceding rollbacks.

Listing 1. Since we are using nested speculation, each thread has a *depth*. First we go through all the threads executing and place their depth in a list *l*. In the next stage, we sort the list *l* ascending. Initially we set a variable *m* to 0.5. The value *m* is increased to *m = m + 1.0 / pow(2, no_rollback + 1)* if there is a rollback. Therefore, after the first rollback *m* would be 0.75, after the second rollback *m* would be 0.825, etc. We pick the element *a* from *l[m×length of l]*, if the depth of the function we are about to speculate on is lower than *a*, we save the state. If not, we make sure that, in case of a rollback, we rollback to the last checkpoint were the state was saved. If the length of *l* is lower than 3 we set *a* to 2.

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

If a speculative function makes a function call, we create another thread. This threads' parent will be in the list of executing functions. One of the functions in this list could have a suitable checkpoint to rollback to and the motivations for doing this, is that the threads executing when you speculate on a function call, is probably one of the depths you will rollback to in case of a rollback. We can choose one such thread by the median of the currently executing threads' depths. Therefore the median could be a suitable automatic choice for a limitation of the checkpoint depth. However a fixed median value (like 0.75 or 0.25× the length of the list with depths), even though it made the memory usage lower, increased the execution time. This can be understood from the characteristics of JavaScript execution shown by Martisen at al. [4]; JavaScript functions are small in terms of number of executed bytecode instructions and quickly returns. Therefore the depth of the speculated functions is going to vary.

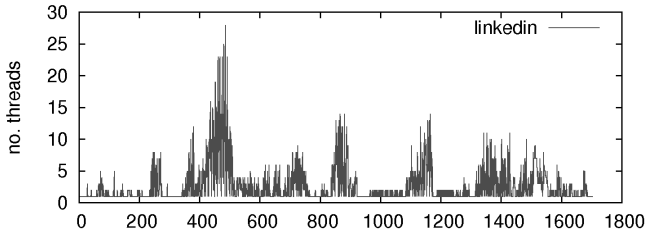

Fig. 4. The maximum number of threads for various points during the execution for *linkedin*.

In Figure 4 the number of threads executing varies greatly. This is a result of nested speculations; each speculated function executes a small number of bytecode instructions, but they are idle while waiting for their child threads to commit back. This argues for an adaptive approach to find a suitable limit to the checkpoint depth. In addition, a fixed checkpoint depth limit does not take rollbacks into account. Since a rollback is often followed by new rollbacks we use a moving median *m*, as seen in Listing 1. JavaScript in web applications is restricted to a single call to the JavaScript engine, so we do not increase *m* when we speculate successfully.

V. Experimental Methodology

We have modified the TLS implementation from Martinsen et al. [7] so we control whether we store a checkpoint or not. The execution behavior of a web application is dependent not only on the JavaScript isolated, but also on the interaction between JavaScript and the web browser such as manipulation of the DOM tree, but we deliberately focus on the JavaScript execution time.

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

VI. Results of fixed checkpoint depths

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

A. Improved execution time

Increasing the checkpoint depth does not decrease the execution time. In Figure 5, the highest speedup for 11 of the 15 use cases is when we limit the checkpoint depth to

either 2, 4 or 8. Without a limit to the checkpoint depth is the fastest for 3 out of 15 cases. If we limit the checkpoint depth to 2, it is the fastest for 4 out of 15 cases, if we limit the checkpoint depth to 4 or 8, it is the fastest for 6 out of 15 (for the cases *Wikipedia* and *Blogspot* the maximum checkpoint depth is 4, therefore the behaviour is identical when we limit the checkpoint depth to either 4 or 8).

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

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

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

In Figure 5 both *Wikipedia* and *Gmail* are faster for a checkpoint depth limit = 1. *Wikipedia* has no rollbacks, and compared to the other cases, a small number of JavaScript bytecode instructions that are executed with for instance 12 speculation versus 12012 for *MSN*. Therefore, we do not see an increased execution time with rollbacks, as there are none, independent of what checkpoint depth limit we set. Further, we do not get a significant speedup, since the number of bytecode instructions and the number of functions to speculate on are much lower.

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

In Figure 5 the gain in improved execution time is marginal if we limit the checkpoint depth to 8 instead of 2 or 4. This is in line with the results of Martinsen et al. [8]. Most of the JavaScript function calls have a depth of 2 and 4. There is a limit to the amount of time JavaScript is allowed to execute for each event in the web application. Therefore, as the depth of a function increases, the number of executing bytecode instruction decreases. This explains why there isn't a large increase in the cost of doing rollbacks, when we rollback from a checkpoint depth larger than 4 and that the relative increase in execution time decreases as the depth increases.

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

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

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

In Figure 7 the number of executed bytecode instructions is, as long as there are rollbacks, always higher with TLS. We see that the number of executed bytecodes increases when the checkpoint depth limit decreases which shows that the costs to do rollbacks and to do re-execution increase.

In the lower part in Figure 7, we see that the number of rollbacks increases, as the checkpoint depth limit increases. The cost of doing a rollback decreases as the checkpoint depth increases, even though the number of rollbacks decreases. This is because the amount of bytecode instructions re-executed will be lower, even though there are more rollbacks. Therefore we reduce the memory usage, and have a higher speedup, if we are more restrictive on the checkpoint depth since the number of re-executed bytecode instructions will be smaller.

The highest speedup is at checkpoint depth limit = 4. Then the speedup is gradually lowered going to checkpoint depth limit = 8, and further lowered when no limit on the checkpoint depth is set. Still it is faster than the sequential execution time. This indicates that the overhead of TLS increases when we increase the checkpoint depth limit, and the gain in terms of potential more functions to execute from a higher checkpoint depth limit is lowered.

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

At checkpoint depth limit = 2 the cost of the rollbacks in terms of executed bytecode instructions is increasing to such

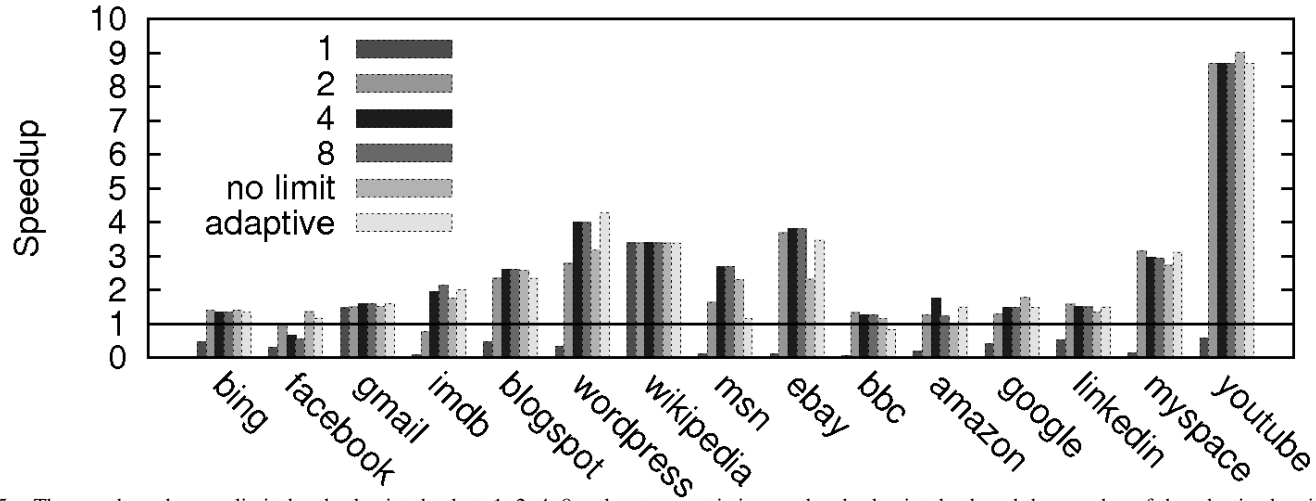

Fig. 5. The speedup when we limit the checkpoint depth to 1, 2, 4, 8 and put no restriction on the checkpoint depth and the speedup of the adaptive heuristic.

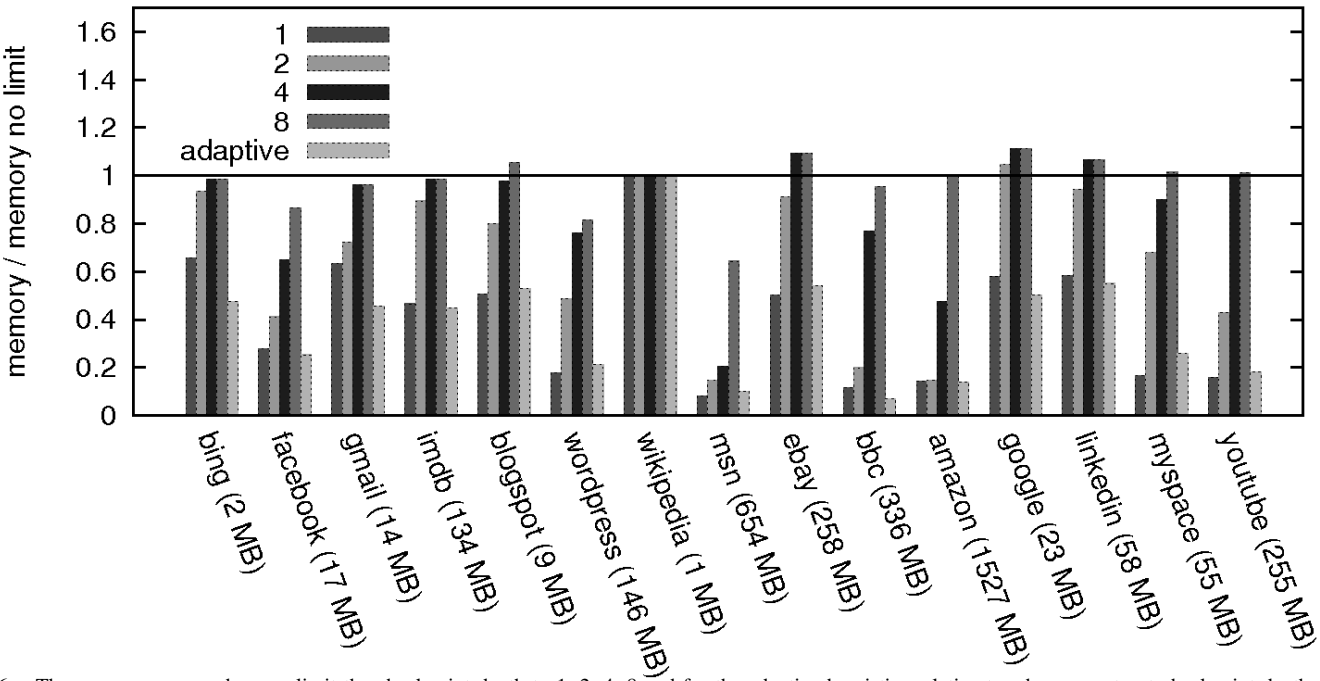

Fig. 6. The memory usage when we limit the checkpoint depth to 1, 2, 4, 8 and for the adaptive heuristics relative to when we set not checkpoint depth.

an extent that it is 2% slower than when no limit is set on the checkpoint depth. When we set a checkpoint depth limit = 1 it is slower than the sequential execution time.

B. Reduction in memory usage

Increasing the limit of the checkpoint depth increases the memory usage. In Figure 6 we have measured the maximum memory usage for the selected use cases when we limit the checkpoint depth to 1, 2, 4, 8 and when we have no limit on the checkpoint depth. Since the memory usage varies between 1 and 1527MB, we have divided each memory usage with the memory usage when we do not limit the checkpoint depth. We have written the memory usage for TLS in Martinsen et al. when we do no limit on the checkpoint depth in parenthesis in Figure 1.

Figure 6 shows that the memory usage is increasing as we increase the checkpoint depth limit. The reason is that we are saving more states in case of rollbacks. When we are limiting the checkpoint depth to 2, we reduce the memory usage of TLS with 65%. When we limit the checkpoint depths to 4 and 8, we reduce the memory usage with 14% and 3% respectively.

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

In *Linkedin* we have almost the same number of threads and speculations; however when we do not limit the checkpoint depth, the number of rollbacks becomes much higher, therefore we are able to reduce more memory than when we limit the

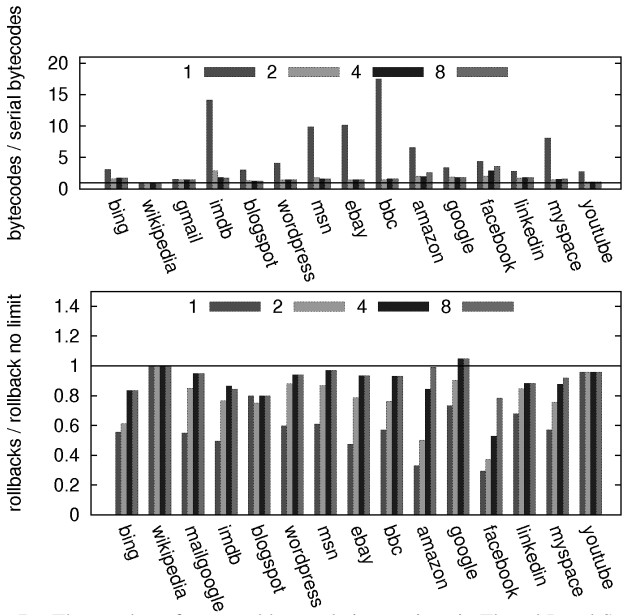

Fig. 7. The number of executed bytecode instructions in Thread-Level Speculation relative to the number of sequentially executed bytecode instructions (upper) and the number of rollbacks relative to the number of rollbacks when we do not limit the checkpoint depth (lower). A special case in the Figure is the *wikipedia* case, were there are no rollbacks, so the number of executed bytecode instructions are the same for TLS and for the sequential execution.

checkpoint depth to 8.

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

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

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

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

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

VII. Results of the adaptive heuristic

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

A. Improved execution time

Figure 5 shows that we are able to increase the speedup of JavaScript execution from 1.14 – 8.69 times faster (*MSN* and *YouTube*), and have a speedup which more than half the time is faster than when we do not set any limitation on the checkpoint depth, and always better than the sequential execution time (except for *BBC*). One example of the effect of the heuristic is the *YouTube* web application; when no limit is set on the checkpoint depth, it uses 255 MB and is over 8 times faster. With the heuristic it only uses 44 MB (a reduction of 83%) while it is only 4% slower compared to when no limit is set on the checkpoint.

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

Figure 8 shows the checkpoint depths in the only example which is slower than the sequential execution with the heuristic. The memory usage is 93% of the memory usage when no checkpoint depth is set. This is caused by the heuristic rolling back to a checkpoint depth which is close to 1, repeatedly. This significantly reduces the memory usage, but increases the cost of rollbacks as it forces us to re-execute many bytecode instructions.

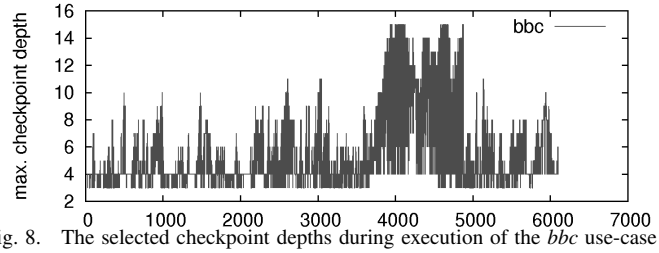

Fig. 8. The selected checkpoint depths during execution of the *bbc* use-case

When a small number of threads are executing, we do not really need to store the checkpoint. If a rollback occurs, the amount of bytecode instructions we have to re-execute will be small. This has two consequences; first, given the limitations of allowed execution time in JavaScript in web applications,

the functions that are executing are at this point quite large. The next consequence is, if there will be a rollback in these, the number of bytecode instructions for the re-execution is limited. Therefore the heuristic reduces the execution time.

B. Reduction in memory usage

Figure 6 shows that we reduce the memory usage between 93% – 45% (*BBC* and *Linkedin*). One exception is *Wikipedia*, but this use case does not have any rollbacks, little JavaScript execution, and a low number of JavaScript function calls, which limits the potential gain from speculations.

The heuristic is able to reduce the memory usage below when we set the checkpoint depth limit to 1. In Figure 6, for 9 out of the 15 use cases, the memory usage is lower with the heuristic than when we set the checkpoint depth limit to 1.

The heuristic has a higher number of rollbacks and a higher number of speculations. This explains why the memory usage is lower, both with more rollbacks with small number of bytecode instructions that needs to re-executed and more commits. Since we are using nested speculation, a speculated function could be created from a function executing speculativly. We could imagine the functions executing in a tree like structure, similar to the one in Figure 3. When we do a rollback, we would like to rollback to a state, which is part of the speculation tree, to reduce the number of bytecode instructions that needs to be re-executed. With a checkpoint of 1, we often re-execute bytecode instructions which are not part of the speculation tree, but with a moving median we might not.

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

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

VIII. Conclusions

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

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

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

References

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

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

[3] P. Ratanaworabhan, B. Livshits, and B. G. Zorn, "JSMeter: Comparing the behavior of JavaScript benchmarks with real web applications," in *WebApps'10: Proc. of the 2010 USENIX Conf. on Web Application Development*, 2010, pp. 3–3.

[4] J. K. Martinsen, H. Grahn, and A. Isberg, "A Comparative Evaluation of JavaScript Execution Behavior," in *Proc. of the 11th Int'l Conf. on Web Engineering (ICWE 2011)*, June 2011, pp. 399–402.

[5] E. Fortuna, O. Anderson, L. Ceze, and S. Eggers, "A limit study of javascript parallelism," in *2010 IEEE Int'l Symp. on Workload Characterization (IISWC)*, Dec. 2010, pp. 1–10.

[6] M. Mehrara, P.-C. Hsu, M. Samadi, and S. Mahlke, "Dynamic parallelization of JavaScript applications using an ultra-lightweight speculation mechanism," in *Proc. of the 17th Int'l Symp. on High Performance Computer Architecture*, 2011, pp. 87–98.

[7] J. K. Martinsen, H. Grahn, and A. Isberg, "Using Speculation to Enhance JavaScript Performance in Web Applications," *IEEE Internet Computing*, vol. 17, no. 2, pp. 10–19, 2013.

[8] ——, "A Limit Study of Thread-Level Speculation in JavaScript Engines – Initial Results," in *Fifth Swedish Workshop on Multi-Core Computing (MCC-12)*, November 2012, pp. 75–82.

[9] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of JavaScript programs," in *PLDI '10: Proc. of the 2010 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2010, pp. 1–12.

[10] C. J. F. Pickett and C. Verbrugge, "Software thread level speculation for the Java language and virtual machine environment," in *LCPC '05: Proc. of the 18th Int'l Workshop on Languages and Compilers for Parallel Computing*, October 2005, pp. 304–318, lNCS 4339.

[11] P. Rundberg and P. Stenström, "An all-software thread-level data dependence speculation system for multiprocessors," *Journal of Instruction-Level Parallelism*, pp. 1–28, 2001.

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

[13] J. Mickens, J. Elson, J. Howell, and J. Lorch, "Crom: Faster web browsing using speculative execution," in *Proc. of the 7th USENIX Symp. on Networked Systems Design and Implementation (NSDI 2010)*, April 2010, pp. 127–142.

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

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