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

Jan Kasper Martinsen and Håkan Grahn
Department of Computer Science and Engineering
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden
Email: {Jan.Kasper.Martinsen,Hakan.Grahn}@bth.se

Anders Isberg and Henrik Sundström
Sony Mobile Communications AB
SE-221 88 Lund, Sweden
{Anders.Isberg,Henrik.Sundstrom}@sonymobile.com

***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 is high. Here we propose to reduce the memory usage by limiting the checkpoint depth based on an in-depth study of the memory and execution time effects. We also propose an adaptive heuristic to dynamically adjust the checkpoints. We evaluate this using 15 web applications on an 8-core computer. The results show that the memory overhead is reduced for Thread-Level Speculation by over 90% as compared to storing all checkpoints. Further, the performance is often better than when storing all the checkpoints and at worst 4% slower.**

I. Introduction

JavaScript is a sequential scripting language with run-time evaluation and cannot take advantage of multicore processors to reduce the execution time. Fortuna et al. [1] 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 on a series of well-known benchmarks [2] and Martinsen et al. [3] show this on a number of popular web applications. Unfortunately, we found in [3] that the memory requirements are significant for web applications. Further, we show in [4] that there is a potential to reduce the memory usage and improve the execution time by limiting the speculation depth. In many cases a speculation depth of 2 to 4 is sufficient to improve the performance.

In this paper we show that we can reduce the memory usage by nearly 90% by limiting at what speculation depth we store the checkpoints, and in most 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 in [3].

II. Reducing the memory usage for TLS

A. 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.

Our idea is to limit at what depths we store the checkpoints, but still allow an unlimited speculation 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 Fig. 1.

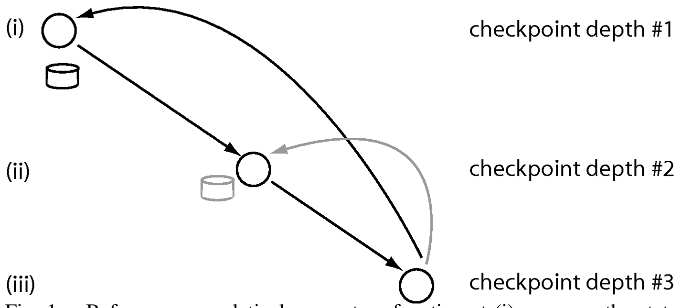


Fig. 1. Before we speculatively execute a function at (i), we save the state so we can rollback to this point. At (ii), i.e., the speculative function made as a speculative function call at (i), we speculatively execute another function call (iii). In normal TLS, we also save the state in (ii) in case 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).

B. 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 fewer checkpoints to reduce the memory usage, however this makes rollbacks more time consuming. We therefore propose *an adaptive heuristic that dynamically adjusts the checkpoint depth limit* based on the speculation depth and rollback behavior of a web application.

The heuristic in Listing 1 improves 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. We have shown 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. Rollbacks are rare, but they often occur between speculative functions with the same depth and occur closely after each other. Therefore, when a rollback occurs, we 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 thread's 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 one of the depths you will rollback to in case of a mis-speculation. We 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 [5]; 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.

III. Experimental Methodology

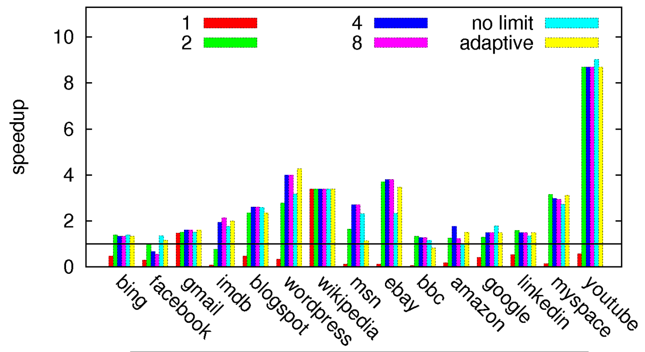
Our Thread-Level Speculation is implemented in the Squirrelfish [6] JavaScript engine, and the basic TLS implementation is described in [3]. We have modified the TLS implementation to control whether to store a checkpoint or not. The execution behavior of a web application is dependent not only on the JavaScript isolated, e.g., interaction and manipulation of the Document Object Model (DOM) tree, but in this paper we focus on the JavaScript execution time.

We have selected 15 web applications from the Alexa list of most visited web applications. The experiments are done on a computer with 2 quadcore processors, i.e., in total 8 cores, and 16 GB main memory.

IV. Results of Fixed Checkpoint Depths

A. Improved execution time

Increasing the checkpoint depth to a certain limit generally reduces the execution time. The highest speedup for 11 of the 15 use cases is when we limit the checkpoint depth to either 2, 4 or 8 (Fig. 2). However, an unlimited checkpoint depth is fastest only for 3 out of 15 cases. The overhead of TLS is increasing with an increased checkpoint depth, and the potential for finding speculative functions decreases as the checkpoint depth is 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.



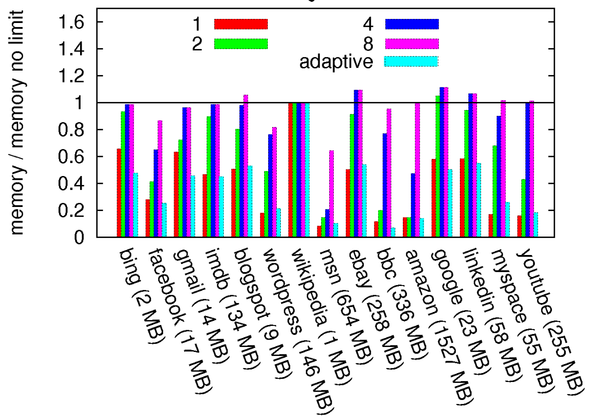


Fig. 2. The speedup when we limit the checkpoint depth to 1, 2, 4, 8 and put no restriction on the checkpoint depth and the speedup of the adaptive heuristic. The memory usage when we limit the checkpoint depth to 1, 2, 4, 8 and for the adaptive heuristics relative to when we set not checkpoint depth.

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.

Wikipedia and *Gmail* are faster for a checkpoint depth limit = 1 (Fig. 2). *Wikipedia* has no rollbacks, few speculations (only 12), and compared to the other web applications, limited JavaScript interaction. Therefore, we do not see an increased execution time with rollbacks, as there are none, independent of what checkpoint depth limit we set.

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 and 17 rollbacks when we set the checkpoint depth limit to 1 and 2, respectively. The execution time is 2% faster setting the checkpoint depth limit = 2 which is 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. However, the cost of doing rollbacks is much higher for checkpoint depth limit=1, than it is for checkpoint depth limit=2. The effect of having a larger number of threads does not outweigh the increased cost of doing rollbacks, therefore it is slower than checkpoint depth limit = 2. However, it exploits a large number of threads and is therefore faster than sequential execution.

The gain in improved execution time is marginal when limiting the checkpoint depth to 8 instead of 2 or 4 (Fig. 2). This is in line with the results in [4], where we show that most of the JavaScript function calls have a depth of 2 and 4. Further, 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 executed bytecode instructions per function decreases. As a result, there is only a small increase of the cost of doing rollbacks from a checkpoint depth larger than 4.

The highest speedup is at checkpoint depth limit = 4, then the speedup is gradually reduced. 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. The number of rollbacks relative to the number of executed bytecode instructions gradually decreases from checkpoint depth limit = 2 to limit = 8. Based on the observation of the execution time for an unlimited number of checkpoints stored, we observe that the number of executed bytecode instructions decreases and that we at some point reach a lower execution time 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 Fig. 3 the number of executed bytecode instructions is, as long as there are rollbacks, always higher with TLS. The number of executed bytecode instructions increases when the checkpoint depth limit decreases which shows that the costs of doing rollbacks and doing re-executions increase.

As the checkpoint depth limit increases the number of rollbacks increases (Fig. 3). The cost of doing a rollback decreases as the checkpoint depth increases since the number 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.

B. Reduction in memory usage

In Fig. 2 we show 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. We have normalized the memory usage to the memory usage of unlimited checkpoint depth, i.e., each memory usage is divided with the memory usage with not limit on the checkpoint depth.

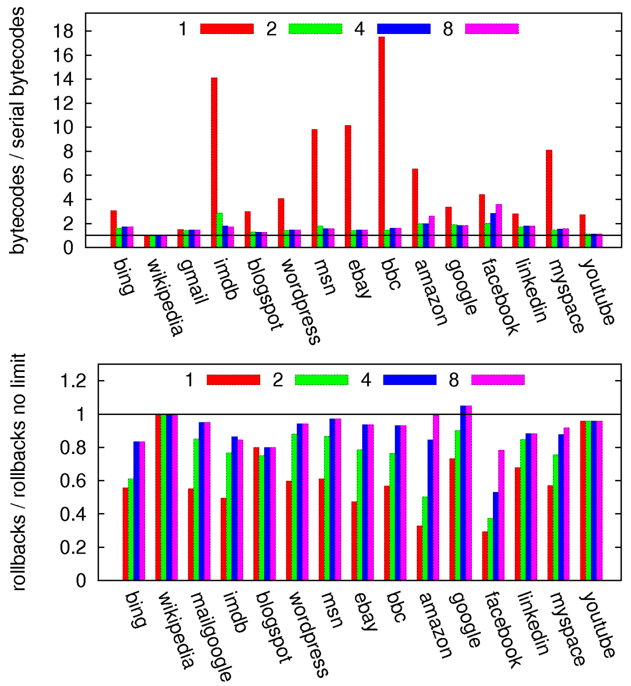


Fig. 3. The number of executed bytecode instructions in Thread-Level Speculation relative to the number of sequentially executed bytecode instructions (upper) and the number of rollbacks relative to the number of rollbacks when we do not limit the checkpoint depth (lower). A special case in the Figure 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.

In general, we find that increasing the checkpoint depth increases the memory usage. The reason for this is that we are saving more checkpoints. 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 with unlimited checkpoint depth. There are two operations which reduce the memory usage; when we rollback due to a mis-speculation or 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 can reduce the memory usage more than when we limit the checkpoint depth to 8.

Blogspot and *Ebay* have almost 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 when we limit the checkpoint depth to 8 than when we do not limit the checkpoint depth, while the number of threads and rollbacks are the same. This 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.

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 we can save large amounts of memory.

V. 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

When a small number of threads are executing, we do not really need to store the checkpoint. If a rollback occurs, the number 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

Fig. 2 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 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 Fig. 2, 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 speculatively. We could imagine the functions executing in a tree like structure, similar to the one in Fig. 1. 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, the probability of rollbacks is very low; 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.

VI. Conclusions

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.

References

[1] 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.

[2] 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.

[3] 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.

[4] ——, "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.

[5] ——, "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.

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