The Effect of Thread-Level Speculation on a Set of Well-known Web Applications

Jan Kasper Martinsen and Håkan Grahn
School of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden
{Jan.Kasper.Martinsen,Hakan.Grahn}@bth.se

Anders Isberg
Sony Ericsson Mobile Communications AB
SE-221 88 Lund, Sweden
Anders.Isberg@sonyericsson.com

ABSTRACT

Previous studies have shown that there are large differences between the workload of established JavaScript benchmarks and popular Web Applications. It has also been shown that popular optimization techniques, such as just-in-time compilation, many times degrade the performance of Web Applications. Further, since JavaScript is a sequential language it cannot take advantage of multicore processors.

In this paper, we propose to use Thread-Level Speculation (TLS) as an alternative optimization technique for Web Applications written in JavaScript. Our TLS approach is based on speculation at the function level. We have implemented TLS in WebKit, a state-of-the-art web browser and JavaScript engine. Our results show speedups between 2 and 8 on eight cores for seven popular Web Applications, without any JavaScript source code changes at all. The results also show few roll-backs and the additional memory requirements for our speculation is up to 17.8 MB for the studied Web Applications.

1. INTRODUCTION

During the last years have many applications moved to or evolved on the World Wide Web. Such applications are often referred to as web applications. Web applications can be defined in different ways, e.g., as an application that is accessed over the network from a web browser, as a complete application that is solely executed in a web browser, and of course various combinations thereof. Social networking web applications, such as Facebook [17], Twitter [12], and Blogger [3], have turned out to be popular, being in the top-25 web sites on the Alexa list [1] of most popular web sites. All these three applications use the interpreted language JavaScript [10] extensively for their implementation. In fact, almost all of the top-100 sites on the Alexa list use JavaScript to some extent.

JavaScript is a dynamically typed, object-based scripting language with run-time evaluation, where execution is done in a JavaScript engine [9, 25, 16], i.e., an interpreter/virtual machine that parses and executes the JavaScript program. Due to a higher demand for performance, several optimization techniques have been suggested along with sets of benchmarks. However, these benchmarks have been reported as unrepresentative [14, 21, 22], and current optimization techniques could degrade the performance of popular Web Applications [15].

JavaScript is a sequential language and cannot take advantage of multicore processors. This is unfortunate, since Fortuna et al. [8] showed that there exist significant potential parallelism in many JavaScript applications, up to 45x was reported. However, they have not implemented support for parallel execution in any JavaScript engine. Many browsers support 'Web Workers' [24] that allow parallel execution of tasks in Web Applications based on a message passing paradigm, but it is still the programmer who is responsible for finding and expressing the parallelism.

To hide some of the details of the under-laying parallel hardware, an approach is to dynamically extract parallelism from a sequential program using Thread-Level Speculation (TLS) techniques [23]. The performance potential of TLS has been shown for applications with static loops, statically typed languages, and in Java bytecode environments. In an earlier work [13], we proposed to use TLS in a JavaScript context.

In this paper, we extend the previous work with mainly two contributions: (i) an implementation of TLS in the state-of-the-art SquirrelFish JavaScript engine [25] found in WebKit, and (ii) a performance evaluation of TLS for JavaScript on seven popular Web Applications, e.g., BlogSpot, Facebook, and YouTube, rather than on established benchmarks. The execution and behaviour of a Web Application is dependent not only of the JavaScript code, but also of the interaction with the web browser and the DOM tree. However, we deliberately focus only on the JavaScript part in this study.

Our results show that there is a potential to execute a large number of functions as threads, that the JavaScript in Web Applications might be well suited for such an optimization technique, and that we are able to improve the performance by between 2 and 8 times on a dual quad-core machine as compared to a of sequential execution. However there are also a number of challenges; The memory requirements increases by up to 17.8 MB in our measurements and the features of dynamical languages add additional challenges.

2. THREAD-LEVEL SPECULATION FOR WEB APPLICATIONS

2.1 Thread-Level Speculation Principles

TLS aims to dynamically extract parallelism from a sequential program. This can be done both in hardware and software. One popular approach is to allocate each loop iteration to a thread. Then, we can (ideally) execute as many iterations in parallel as we have processors. However, data dependencies may limit the number of iterations that can be executed in parallel. Further, the memory requirements and run-time overhead for detecting data dependencies can be considerable.

Between two consecutive loop iterations we can have three types of data dependencies: *Read-After-Write* (RAW), *Write-After-Read* (WAR), and *Write-After-Write* (WAW). A TLS implementation must be able to detect these dependencies during run-time using information about read and write addresses from each loop iteration. A key design parameter is the *precision* of what granularity the TLS system can detect data dependency violations.

When a data dependency violation is detected, the execution must be aborted and rolled back to safe point in the execution. Thus, all TLS systems need a roll-back mechanism. The book-keeping related to this functionality results in both memory overhead as well as run-time overhead. In order for TLS systems to be efficient, the number of roll-backs should be low.

A key design parameter for a TLS system is the data structures used to track and detect data dependence violations. The more precise tracking of data dependencies, the more memory overhead is required. 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.

2.2 Software-Based Thread-Level Speculation

There exists a number of different software-based TLS proposals, and we review some of the most important ones here. It should be noted that all these studies have worked with applications written in C, Fortran, or Java.

Bruening et al. [4] proposed a TLS system that targets loops where the memory references are stride-predictable. Further, it is one of the first techniques that is applicable to while-loops where the loop exit condition is unknown until the last iteration. The results show speed-ups of up to almost five on 8 processors.

Rundberg and Stenström [23] proposed a TLS implementation that resembles the behaviour of a hardware-based TLS system. The main advantage with their approach is that it precisely tracks data dependencies, thereby minimizing the number of unnecessary roll-backs cased by false-positive violations. The downside is high memory overhead. They show a speedup of up to ten times on 16 processors for three applications.

Kazi and Lilja developed the course-grained thread pipelining model [11] exploiting coarse-grained parallelism. On an 8-processor machine they achieved speed-ups of between 5 and 7. Bhowmik and Franklin [2] developed a compiler framework for extracting parallel threads from a sequential program for execution on a TLS system yielding speed-ups between 1.64 and 5.77 on 6 processors.

Cintra and Llanos [7] present a software-based TLS system that speculatively executes loop iterations in parallel within a sliding window. By using optimized data structures, scheduling mechanisms, and synchronization policies they manage to reach in average 71% of the performance of hand-parallelized code for six applications.

Chen and Olukotun present two studies on [5, 6] how method-level parallelism can be exploited using speculative techniques. 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 [19, 20] developed SableSpMT, a framework for method-level speculation and return value prediction in Java programs. Their solution is implemented in a Java Virtual Machine and they obtain at most a two-fold speed-up on a 4-way multi-core processor.

Oancea et al. [18] present a novel software-based TLS proposal that supports in-place updates. Further, their proposal has a low memory overhead with a constant instruction overhead, at the price of slightly lower precision in the dependence violation detection mechanism. However, the scalability of their approach is superior due to the fact that they avoid serial commits of speculative values, which in many other proposals limit the scalability. The results show that their TLS approach reaches in average 77% of the speed-up of hand-parallelized, non-speculative versions of the programs.

3. THREAD-LEVEL SPECULATION IMPLEMENTATION FOR JAVASCRIPT

We have used the Squirrelfish JavaScript interpreter which is part of WebKit [25], a state of the art web browser environment, and modified it for TLS. We have made some modifications so it would be easier to execute as a thread. More specifically, we use a switch statement instead of a goto statement (where the goto labels are predefined memory locations), and disabled just-in-time compilation. In addition, we have modified the interpreter function so it can be executed from a thread, and the input parameters to the interpreter were modified so they sent as a part of a structure. A general view of how the speculation is done is shown in Figure 1. If the interpreter makes a call to a JavaScript function, a new thread is spawned and placed in the thread pool. Before the new thread is spawned, the state of the threads in the thread pool, a set of writes and reads, and the values of the JavaScript program are saved for possible rollbacks.

Initially the entire executed JavaScript program, which in our case is extracted from an execution in the web browser, is sent to a thread that executes the interpreter. The first thread is not speculated and will never be re-executed. Therefore, we do not need to store the data that is part of this thread's execution. We start the thread with an initial value

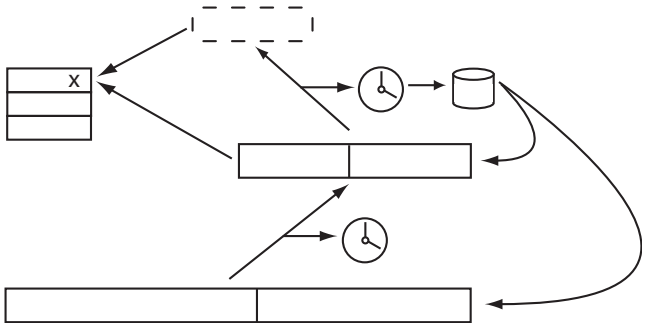

Figure 1: An example of TLS. First a new thread is spawned, the state is then saved, before it spawns another thread which in turn will have a conflict with the thread it was spawned from. Upon a conflict the state is restored.

realtime set to 0. For each executed bytecode instruction, the value of *realtime* is increased by 1.

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

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

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

In these studies, we look at conflicts between global variables and ids. Ids are special for JavaScript as they can be created at any point of time, and can be defined with a global scope. During execution, we might encounter four different opcodes which manipulate global variables or ids:
op_put_global_var, which writes a value to a specific global variable, *op_get_global_var*, which reads a value from a specific global variable, *op_put_by_id*, which writes to a specific globally accessible id, and *op_get_by_id*, which reads from a specific globally accessible id.

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

There are four kind of cases that we test against, partly shown in Figure 2:

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

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

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

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

In addition we could end up in a situation where several of the threads perform a write or read operation at the same

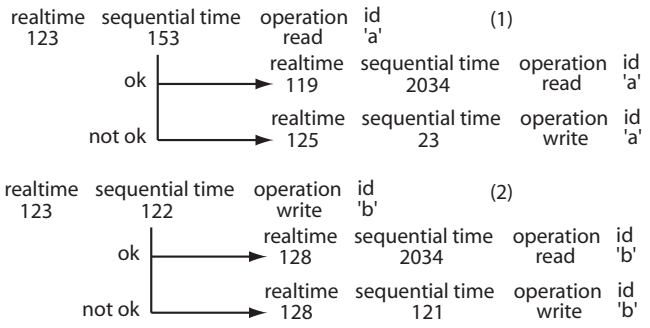

Figure 2: Values of *sequential time* and *realtime* at different phases of the speculation.

realtime. To handle this, we have done this check after *realtime* is increased by 1, and perform the test above iterative for all the operations. Likewise, if the list of unique identities is empty, we insert the value.

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

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

Even though we have a set of threads that are supposed to be active, it is likely that there might have been created threads after this point of time, and that these not associated with the current state of the TLS system. Therefore, we need to recursively go through the threads and their parent threads that are now part of the active state. The resulting list contains the threads which are necessary in the current state of execution. The remainder of the threads and their associated interpreter are stopped and set to an idle status for later reuse.

When a thread reaches its end of execution (encounter its associated *op_ret*), its modification of global variables and id need to be committed back to its parent thread, as shown in Figure 3. However, this can first be done after threads that have been created from the completed thread's fork points have in turn completed their execution. These threads, are denoted as child threads and their manipulations to global variables and ids are to be committed to the current thread. This is also the case for the main or the initial thread, after the program completes execution after all the threads have completed execution they are committed to the main thread.

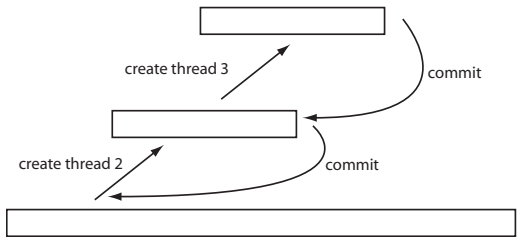

Figure 3: Two speculative threads are executed and committed when no conflict occurs.

In our implementation we are rather pessimistic. When a thread completes execution, and commits its values, we do not remove the associated read and writes from global variables and ides that are no longer relevant. Therefore there might be conflicts, which are not between active threads, but rather are conflicts that were before the threads were committed.

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

4. EXPERIMENTAL METHODOLOGY

Our Thread-Level Speculation is implemented in the Squirrelfish [25] JavaScript engine which is part of WebKit, a state-of-the-art browser environment. We have selected seven popular Web Applications from the Alexa list [1] of most used web sites. Then, we have defined and recorded a set of use-cases from *Wikipedia, YouTube, Imdb, Wordpress, Facebook, BlogSpot*, and *Linkedin* using minimal user interaction. The methodology is further described in [14].

To escape some of the runtime issues, we do the following; We first execute the Web Application in a modified version of WebKit. This modified version saves the executed JavaScript bytecodes, together with the associated values for each execution. This information is later re-executed in the use cases.

All experiments are conducted on a system running Ubuntu 10.04 and equipped with two quad-core processors and 16

GB main memory. In all measurements we have measured the execution time in the JavaScript engine, rather than the execution time of the overall Web Application.

5. EXPERIMENTAL RESULTS

We start by comparing the execution times of the different Web Applications without and with TLS. The execution times shown in Figure 4, are:
$T_{exe}(with\ TLS)$ / $T_{exe}(without\ TLS)$, i.e., a value lower than 1 means that the execution time is lower with TLS enabled.

Our results show that TLS improves the execution time of the JavaScript in the selected Web Applications between 8.39 (*Youtube*) and 1.86 (*Linkedin*) as compared to the sequential execution time. We see further that the execution time is improved for all of the cases.

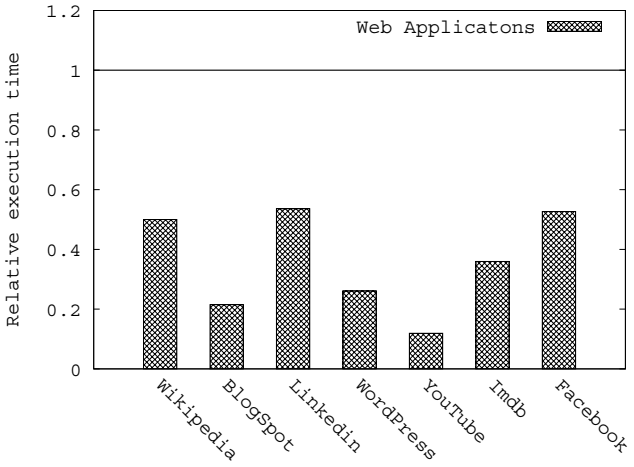

Figure 4: Improved execution time of JavaScript in the Web Application relative to running it as a sequential application

In order to better understand and analyze the execution time reduction, we have measured a number of metrics. In Table 1 we show measurements of the number of rollbacks, the average call depth when we need to do a rollback, the maximum number of concurrently runnable threads, the total number of speculations, and the memory usage. The number of rollbacks are the number of times we have to go back to the previous fork point after we had a conflict, the maximum number of concurrently runnable threads is the highest number of threads running at once, the total number of speculations is the total number of speculations we were able to make (including the rollbacks), the average depth of the search to remove data is how deep we had to search to find functions that were associated with the current state and the total of memory requirements of how much data the various cases uses.

The results show that in general, there are many functions that we are able to speculate successfully on. It is possible to speculate on between 12 and 7349 JavaScript functions in the applications. We also found that there are very few rollbacks in general, at most 156 rollbacks are performed. The average depth of the recursive search to remove data from previous speculations are between 2.27 and 9.16, which indicates that speculations are deeply nested. We also found a very large number of concurrently runnable threads, which is between 8 and 407 . Finally, the average memory usage before each rollback[1] it is between 1.1 and 17.8 MB. This amount to the part saved, in case of a rollback.

The worst execution time is 1.86 times faster than the execution time of the sequential version, while the best is 8.39 times faster (*Youtube*). For the *Youtube* case there is a much higher number of functions running as threads, potential functions for speculations, together with a low number of rollbacks. We also notice that the average depth of search to remove data associated with previous speculations is fairly low relative to the number of speculations. For the *Facebook* case it is different, there is a relative small number of speculations, a relative high number of rollbacks, and the recursive searches for data associated with previous speculations are fairly deep, which affects the execution time negatively.

We see that conflicts between functions are relatively rare. The worst ratio (*rollbacks/speculation*) between rollbacks and speculations are 0.05, while the best (ignoring the *Wikipedia* case without any rollbacks) is 0.003. There is also a potential for running a large number of threads simultaneously from 8 (*Wikipedia*) to 407 (*Youtube*).

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented an implementation of thread-level speculation in the Squirrelfish JavaScript engine. Our results shows that TLS is an appropriate method for significantly reducing the execution time of JavaScript in Web Applications. Speedups of between 1.86 and 8.39 were achieved as compared to a sequential execution, and for all of the cases the execution time is improved. For most of the cases the improved execution speed is close to two-fold or less.

We speculate on JavaScript function calls, and JavaScript functions in Web Applications are often bound to events. Our results indicate that these functions rarely access the same variables (being either global variables or ids). One challenge of using TLS for Web Applications is that there is a large number of functions that we are able to successfully speculate on. On beforehand we do not know which one will cause a rollback. This means that the memory requirements may be high due to a small number of rollbacks. For example, for the *Youtube* case we have a small number of rollbacks, so we continuously accumulate information to handle potential rollbacks (on average 17.8 MB), even though it seldom happens.

We outline four paths for future work on thread-level speculation in Web Applications: (i) We need to develop better, maybe adaptive, heuristics for deciding when to speculate on a particular function. (ii) Just-in-time compilation is a common technique in web browsers, but it has been shown to be unsuitable for many Web Applications [15]. A possible approach can be to combine just-in-time compilation and TLS. (iii) The memory requirements for TLS is rather

[1]The *Wikipedia* case has no rollbacks, therefore we present the amount of memory upon completion of the program in that case.

Table 1: Number of rollbacks, average depth for recursive search deleting associated values with previous speculations, maximum number of threads, number of speculations for the 7 selected applications, and average memory usage before each rollback (in megabytes).

Application	Number of rollbacks	Average depth	Maximum number of threads	Number of speculations	Memory usage (MB)
facebook	51	9.16	27	968	7.1
linkedin	51	2.27	36	1815	7.1
imdb	156	6.85	54	5300	17.8
youtube	25	5.44	407	7349	17.1
wordpress	63	4.55	99	5852	9.7
blogspot	15	2.6	16	778	1.6
wikipedia	0	0	8	12	1.1

high. We have observed that rollbacks rarely occur, so in most cases we don't really need the information saved. (iv) Finally, the interaction between the JavaScript engine and the runtime system (web browser) needs to be studied. For example, how does the JavaScript execution interact with the DOM tree, and how do we perform speculations and potential rollbacks in this context.

Acknowledgments

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

7. REFERENCES

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

[2] A. Bhowmik and M. Franklin. A general compiler framework for speculative multithreading. In *SPAA '02: Proceedings of the fourteenth annual ACM Symposium on Parallel Algorithms and Architectures*, pages 99–108, New York, NY, USA, 2002. ACM.

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

[4] D. Bruening, S. Devabhaktuni, and S. Amarasinghe. Softspec: Software-based speculative parallelism. In *FDDO-3: Proceedings of the 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization*, 2000.

[5] M. K. Chen and K. Olukotun. Exploiting method-level parallelism in single-threaded Java programs. In *Proc. of the 1998 Int'l Conf. on Parallel Architectures and Compilation Techniques*, page 176, 1998.

[6] M. K. Chen and K. Olukotun. The Jrpm system for dynamically parallelizing Java programs. In *ISCA '03: Proc. of the 30th Int'l Symp. on Computer Architecture*, pages 434–446, 2003.

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

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

[9] Google. V8 JavaScript Engine, 2010. `http://code.google.com/p/v8/`.

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

[11] I. H. Kazi and D. J. Lilja. Coarse-grained thread pipelining: A speculative parallel execution model for shared-memory multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 12(9):952–966, 2001.

[12] B. Krishnamurthy, P. Gill, and M. Arlitt. A few chirps about twitter. In *WOSP '08: Proc. of the 1st Workshop on Online Social Networks*, pages 19–24, 2008.

[13] J. K. Martinsen and H. Grahn. An alternative optimization technique for JavaScript engines. In *Third Swedish Workshop on Multi-Core Computing (MCC-10)*, pages 155–160, 2010.

[14] J. K. Martinsen and H. Grahn. A methodology for evaluating JavaScript execution behavior in interactive web applications. In *The 9th ACS/IEEE Int'l Conf. On Computer Systems And Applications*, 2011.

[15] J. K. Martinsen, H. Grahn, and A. Isberg. A comparative evaluation of JavaScript execution behavior. In *Proc. of the 11th Int'l Conf. on Web Engineering (ICWE 2011)*, pages 399–402, June 2011.

[16] Mozilla. What is SpiderMonkey?, 2010. `http://www.mozilla.org/js/spidermonkey/`.

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

[18] C. E. Oancea, A. Mycroft, and T. Harris. A lightweight in-place implementation for software thread-level speculation. In *SPAA '09: Proc. of the 21st Symp. on Parallelism in Algorithms and Architectures*, pages 223–232, August 2009.

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

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

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

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

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

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

[25] WebKit. The WebKit open source project, 2010. `http://www.webkit.org/`.