An alternative optimization technique for JavaScript engines

Jan Kasper Martinsen and Håkan Grahn
School of Computing
Blekinge Institute of Technology
Karlskrona, Sweden
{jkm,hgr}@bth.se

ABSTRACT

Thread Level Speculation at function level has been suggested as a method to automatically (or semi-automatically) extract parallelism from sequential programs. While there have been multiple implementations in both hardware and software, little work has been done in the context of dynamic programming languages such as JavaScript.

In this paper we evaluate the effects of a simple Thread Level Speculation approach, implemented on top of the Rhino1_7R2 JavaScript engine. The evalauation is done using the well-known JavaScript benchmark suite V8. More specifically, we have measured the effects of our null return value prediction approach for function calls, conflicts with variables in a global scope, and the effects on the execution time.

The results show that our strategy to speculate on return values is successful, that conflicts with global variables occur, and for several applications are the execution time improved, while the performance decrease for some applications due to speculation overhead.

1. INTRODUCTION

Current and future processor generations are based on multicore architectures, and it has been suggested that performance increase will mainly come from an increasing number of processor cores [19]. However, in order to achieve an efficient utilization of an increasing number of processor cores, the software needs to be parallel as well as scalable [1, 15, 28]. Meanwhile many applications are moved to the World Wide Web, as so called Web Applications, and new popular programming languages, e.g., JavaScript [11], have emerged. JavaScript is a dynamically typed, object-based scripting language with run-time evaluation, where execution is done in a JavaScript engine. It has also been stated from the language's designers that JavaScript was created with web designers in mind, giving the designers a mean to quickly add interactivity to web pages without too much complexity. Distribution of programs in the form of source-text files have also been advantageous with respect to platform independence. However, currently no JavaScript engine fully supports parallel execution of threads.

Developing parallel applications is difficult, time consuming and error-prone and therefore we would like to ease the burden of the programmer. To hide some of the details, an approach is to dynamically extract parallelism from a sequential program using Thread-Level Speculation (TLS) techniques [4, 13, 18, 21, 25, 6]. For example, software TLS approaches often extract parallelism from loops or functions. A number of consecutive loop iterations or function calls are speculatively executed in parallel, where a data dependency check mechanism detects dependency violations which forces us to restart the program from a certain point in the execution flow (a rollback). The performance potential of TLS has been shown for applications with static loops, statically typed languages, and in byte code environments.

In this paper we describe an approach to apply TLS to the Rhino JavaScript engine [17] and evaluate the performance of the V8 benchmark [10]. In our study, JavaScript function calls are under certain conditions executed as a threads, and dependency violations are detected and solved at runtime. Initial results indicate some promises for TLS in JavaScript engines if the number of function calls is large enough. This paper makes the following contributions:

- Demonstrates that speculating on return values based on historical data in a dynamically typed language such as JavaScript is fruitful.
- Global variables are likely to cause conflicts in function calls executed as a thread.
- Demonstrates some performance increases by speculation on null-returning function calls.

The rest of the paper is organized as follows. Section 2 provides some background on thread-level speculation and JavaScript. Then, we present our method in Section 3. Our experimental setup is presented in Section 4, while the experimental results are presented in Section 5. The paper ends with the conclusions in Section 6.

2. BACKGROUND

In Section 2.1 we present the general principles of thread-level speculation and some previous implementation propos-

als. Then, we discuss the JavaScript language, that is our target in this study, in Section 2.2.

2.1 Thread-Level Speculation

2.1.1 Thread-Level Speculation Principles

Thread-level speculation (TLS) aims at dynamically extracting parallelism from a sequential program. This can be done in many ways: in hardware, e.g., [5, 23, 27], and software, e.g., [4, 13, 18, 21, 25]. In most cases, the main target of the techniques is for-loops and the main idea is to allocate each loop iteration to a thread. Then, ideally, we can execute as many iterations in parallel as we have processors.

There are, however, some limitations. Data dependencies between loop iterations may limit the number of iterations that can be executed in parallel. Further, the memory requirements and run-time overhead for managing the necessary information for detecting data dependencies can be considerable.

Between two consecutive loop iterations we can have three types of data dependencies: *Read-After-Write* (RAW), *Write-After-Read* (WAR), and *Write-After-Write* (WAW). Therefore must a TLS implementation be able to detect these dependecies during run-time using dynamic information about read and write addresses from each loop iteration. A key design parameter here is the *precision* in the detection mechnism, i.e., at what granularity can a TLS system detect data dependency violations. High dependence detection precision usually require high memory overhead in a TLS implementation.

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

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

TLS implementations can differ depending on whether they update data speculatively 'in-place', i.e., moving the old value to a buffer and writing the new value directly in memory, or in a special speculation buffer. Updating data inplace usually result in higher performance if the number of roll-backs is low, but lower performane when the number of roll-backs is high since the cost of doing roll-backs is high.

2.1.2 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.

Bruening et al. [4] proposed a software-based TLS systems that targets loops where the memory references are stride-predictable. Further, it is one of the first techniques that is applicable to while-loops where the loop exit condition is unknown until the last iteration. They evaluate their technique on both dense and sparse matrix applications, as well as on linked-list traversals. The results show speed-ups of up to almost five on 8 processors, but also show slow-downs for some rare cases.

Rundberg and Stenström [25] proposed a TLS implementation that resembles the behavior of a hardware-based TLS system. The main advantage with their approach is that it precisely tracks data dependencies, thereby minimizing the number of unnecessary roll-backs cased by false-positive violations. However, the downside of their approach is high memory overhead. They show a speedup of up to ten times on 16 processors for three applications written in C from the Perfect Club Benchmarks [2].

Kazi and Lilja developed the course-grained thread pipelining model [13] for exploiting coarse-grained parallelism. They suggest to pipeline the concurrect execution of loop iterations speculatively, using run-time dependence checking. In their evalution they used four C and Fortran applications (two were from the Perfect Club Benchmarks [2]). On an 8-processor machine they achieved speed-ups of between 5 and 7. They later extended their to also support Java programs [12].

Bhowmik and Franklin [3] developed a compiler framework for extracting parallel threads from a sequential program for execution on a TLS system. They support both speculative and non-speculative threads, and out-of-order thread spawning. Further, their work address both loop as well as as non-loop parallelism. Their results from 12 applications taken from three benchmark suites (SPEC CPU95, SPEC CPU2000, and Olden) show speed-ups between 1.64 and 5.77 on 6 processors when using both speculative and non-speculative threads.

Cintra and Llanos[9] present a software-based TLS system that speculatively execute loop iterations in parallel within a sliding window. As a result, given a window size of W at most W loop iterations/threads can execute in parallel at the same time. By using optimized data structures, scheduling mechanisms, and synchronization policies they manage to reach in average 71% of the performance of hand-parallelized code for six applications taken from, e.g., the SPEC CPU2000 [26] and Perfect Club [2] Benchmark suites.

Chen and Olukotun present in two studies [7, 8] how method-level parallelism can be exploited using speculative techniques. The idea is to speculatively execute method calls in parallel with code after the method call. Their techniques are implemented in the Java runtime parallelizing machine (Jrpm). On four processors, their results show speed-ups of 3-4, 2-3, and 1.5-2.5 for floating point applications, multimedia applications, and integer applications, respectively.

Picket and Verbrugge [20, 21] developed a TLS framework, SableSpMT, for method-level speculation and return value prediction in Java programs. Their solution is implemented

in a Java Virtual Machine, called SableVM, and thus works mainly at the byte code level. They obtain at most a two-fold speed-up on a 4-way multi-core processor.

Oancea et al. [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 slighty lower precision in the dependence violation detection mechanism. However, the scalability of their approach is superior due to the fact that they avoid serial commits of speculative values, which in many other proposals limit the scalability. Oancea et al. evaluate their approach using seven applications from three benchmark suites (SciMark2, BYTEmark, and JOlden). The results show that their TLS approach reaches in average 77% of the speed-up of hand-parallelized, non-speculative versions of the programs.

Kejariwal et al. [14] evaluated the performance potential of TLS using the SPEC CPU2000 Benchmarks [26]. SPEC CPU2000 consists of 26 applications written in C and Fortran. They found that TLS has a mean speed-up potential of approximately 40% over the applications in addition to the true thread-level parallelism exploited.

A succeeding study by Prabhu and Olukotun [22] analyzed what types of thread-level parallelism that can be exploited in the SPEC CPU2000 Benchmarks. By going through each of the application, they identified a number of useful transformations, e.g., speculative pipelining, loop chunking/slicing, and complex value prediction. They also identified a number of obstacles that hinder or limit the usefulness of TLS parallelization.

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

2.2 JavaScript

JavaScript [11] is a dynamically typed, object-based scripting language with run-time evaluation. JavaScript application execution is done in a JavaScript engine, i.e., an interpretator/virtual machine that parses and executes the JavaScript program. Examples of JavaScript engines are Google's V8 engine [10], WebKit's Squirrelfish [29], and Mozilla's SpiderMonkey and TraceMonkey [16].

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

3. METHOD

To evaluate the effects of TLS in JavaScript we have modified an existing Java based JavaScript engine Rhino1_7R2 (Rhino) [17]. Rhino takes the JavaScript source code and compiles it into an internal byte code representation which in turn is executed.

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

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

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

To evaluate the simple TLS approach implemented in the modified Rhino JavaScript engine, we use the V8 JavaScript benchmark suite. The different applications in the V8 suite is listed in Table 1, along with short description. The deltablue benchmark does not execute correctly in Rhino, so we choose to omit it from our experiments.

Table 1: V8 Benchmark programs.

Program	Description
Richards	OS kernel simulation benchmark
DeltaBlue	One-way constraint solver
Crypto	Encryption and decryption benchmark
RayTrace	Ray tracer benchmark
EarleyBoyer	Classic Scheme benchmarks
RegExp	Regular expression benchmark generated
Splay	Data manipulation benchmark

4. EXPERIMENTS

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

- The number of functions that return a null value.

- The effect of two strategies for return value prediction.
- The number of data dependence violations (conflicts) with global variables.
- The relative performance improvement by enabling TLS.

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

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

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

We have measured each time a threaded function call accesses a global variable, when this variable is manipulated also by the main program. We have measured the amount of functions that accesses a global variable, while the main program at the same time manipulates the same variable.

We have evaluated the effects of TLS on the execution time on two different systems. One dual-core laptop with a Windows Vista operating system and a quad core workstation with an Ubuntu Linux operating system. We ran Rhino with Java 1.7.0. In the experiments, we speculate on all function calls with a null return value that are called the first time or that have previously been successfully executed as a thread.

5. RESULTS

We see in Figure 1 that functions with null return account for a relatively small fraction (at most 16% in the Richards benchmark) of the total number of function calls in the benchmarks. However, the number of function calls do not completely reflect the workload of these functions. We have studied the benchmark code, and found that some functions that return null also encapsulate a large amount of value returning function calls.

Figure 2 and Figure 3 show the results from the extended prediction strategy, where we also predict on the function return value. In the figures we show how often the prediction is correct, for both static and dymanic guesses. We see that the extension to predicting the return value based on a function significantly improves the chances for predicting if the return value is null or not.

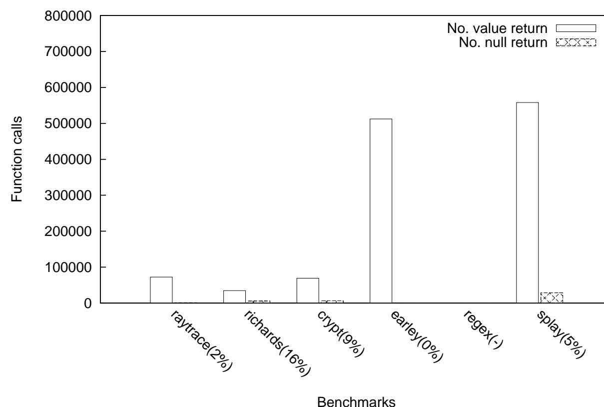

Figure 1: Number of null value returns and total number of function calls. Note that the regex benchmark has no interpreted function calls.

For most of the cases this strategy seems to almost always make a correct prediction. However, this is not the case when making a prediction based on the associated byte code. We have denoted the percentage of the correct static predictions and correct dynamic predictions after the name of the benchmark in the Figure 2. We also see in Figure 3 that the prediction of "not-null" value based on the associated byte code performs very badly. The reason seems to be that for functions with a large amount of null returning instructions, this does not necessarily imply that it is more likely that a null returning instruction is executed.

The results presented in Figure 4 show that conflicts between function calls executed speculatively and the main program's global variables happen for approximately half of the functions. Therefore, accesses to global variables from functions pose a threat.

While the functions with a null return value do not account for so many of the total number of function calls, we have found some performance gain for some of the benchmarks. In Figure 5 we present the relative execution times for the applications running with and without TLS on two different computer systems. The execution times are normalize to the sequential execution time without TLS. We see that the execution time is improved for 4 of the cases. However, the improvement is relatively small, only a 22% reduction in the best case.

6. CONCLUDING REMARKS

Figure 5 shows that the V8 benchmarks do not offer much of a performance improvement with the TLS technique. The largest gain in execution time is 22% on a quad core computer for the Richards benchmark. This can be explained by looking at Figure 1, where we see that the number of interpreted functions with a null value is low. However, for the Richard case, there is a larger improvement than the number of null return values indicates. This suggests that null value functions perhaps also encapsulates return value functions.

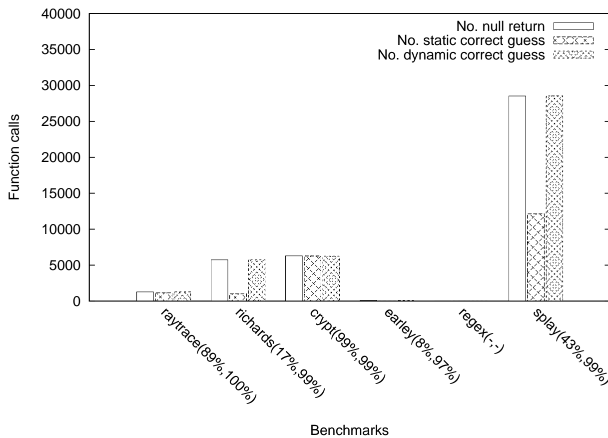

Figure 2: Number of null function returns, with the number of successful predictions from functions byte code and the number of successful predictions from previous functions executions.

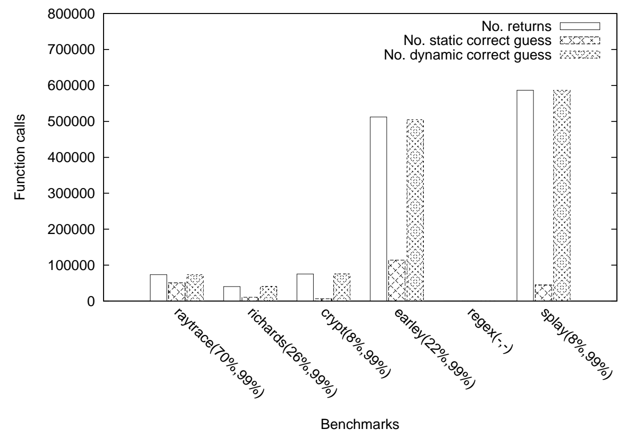

Figure 3: Number of value function returns, with the number successful of predictions from functions byte code and the number of successful predictions from previous functions executions.

We also observe for the Earley benchmark, that there could be a relationship between the difficulties to speculate on the return values. For instance, we see in Figure 1 that there is a large amount of non-null return values, and a small number of null return values for the Earley benchmark. This suggests that is harder to speculate on return values. In addition, we see in Figure 4 that the number of global conflicts is especially high for the Earley benchmark. The raytracing benchmark in Figure 4 has a large amount of conflicts with global variables, but still there are some improvemens when the benchmark is run on a dual core PC. This illustrates the penalty of creating and handling a large amount of threads, and that a large number of speculations also increases the risk for rollbacks which are demanding to administrate.

Even though the execution time is not significantly improved it have been suggested that benchmarks such as V8 might not be representative for the workload of real-world web applications [24]. We know from previous studies that real world web application's workloads consist of a large number of anonymous function calls (which return a null value). The event-driven model found in web browsers, also suggests that these functions might be even more important than the V8 benchmark suite suggests. This suggests that a TLS technique could be more powerful, with a different set of benchmark, or perhaps in a real-life context.

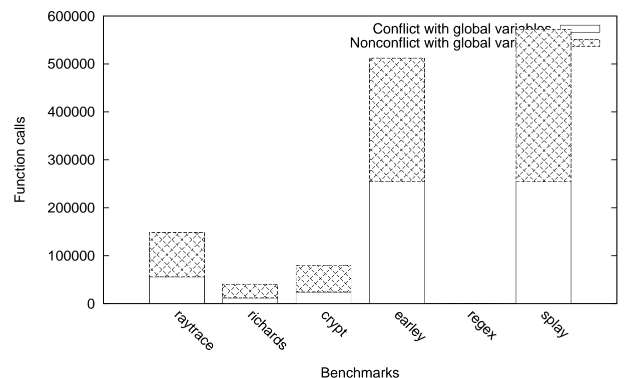

Figure 4: Number of functions with data dependence violations with global variables.

Acknowledgement

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

7. REFERENCES

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] M. Berry, D. Chen, P. Koss, D. Kuck, S. lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. S. adn K. Lue, S. Orzag, F. Seidl, O. Johnson, G. Swanson, R. Goodrun, and J. Martin. The PERFECT Club Benchmarks: Effective performance evaluation of supercomputers. Technical Report CSRD-827, Center for Supercomputing Research and Development, Univ. of Illinois, Urbana-Champaign, May 1989.

[3] 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.

[4] D. Bruening, S. Devabhaktuni, and S. Amarasinghe. Softspec: Software-based speculative parallelism. In

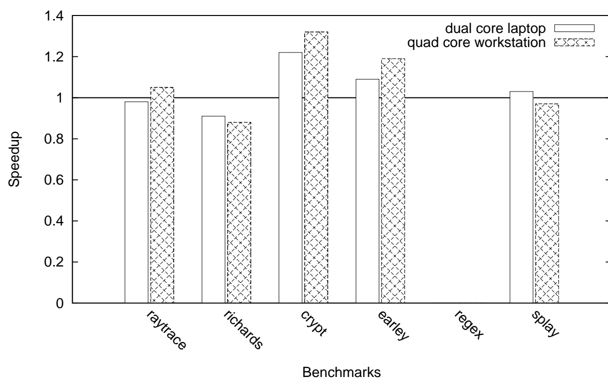

Figure 5: Relative execution time with TLS enabled, normalized to the execution time withou TLS enabled.

FDDO-3: Proceedings of the 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization, 2000.

[5] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Rock: A High-Performance Sparc CMT Processor. *IEEE Micro*, 29(2):6–16, 2009.

[6] M. K. Chen. *The jrpm system for dynamically parallelizing sequential java programs.* PhD thesis, Stanford, CA, USA, 2004. Adviser-Olukotun, Kunle.

[7] M. K. Chen and K. Olukotun. Exploiting method-level parallelism in single-threaded java programs. In *PACT '98: Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, page 176, Washington, DC, USA, 1998. IEEE Computer Society.

[8] M. K. Chen and K. Olukotun. The Jrpm system for dynamically parallelizing Java programs. In *ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 434–446, New York, NY, USA, 2003. ACM.

[9] M. Cintra and D. R. Llanos. Toward efficient and robust software speculative parallelization on multiprocessors. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 13–24, New York, NY, USA, 2003. ACM.

[10] Google. V8 javascript engine, 2009. http://code.google.com/p/v8/.

[11] JavaScript, 2009. http://en.wikipedia.org/wiki/JavaScript.

[12] I. H. Kazi and D. J. Lilja. JavaSpMT: A speculative thread pipelining parallelization model for java programs. In *IPDPS'00: Proc. of the 14th Int'l Parallel and Distributed Processing Symp.*, page 559, May 2000.

[13] 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.

[14] A. Kejariwal, X. Tian, W. Li, M. Girkar, S. Kozhukhov, H. Saito, U. Banerjee, A. Nicolau, A. V. Veidenbaum, and C. D. Polychronopoulos. On the performance potential of different types of speculative thread-level parallelism. In *ICS '06: Proc. of the 20th Int'l Conference on Supercomputing*, page 24, New York, NY, USA, 2006. ACM.

[15] R. McDougall. Extreme software scaling. *Queue*, 3(7):36–46, 2005.

[16] Mozilla. What is spidermonkey?, 2009. http://www.mozilla.org/js/spidermonkey/.

[17] Mozilla. Rhino javascript interpreter, 2010. http://www.mozilla.org/rhino/.

[18] C. E. Oancea, A. Mycroft, and T. Harris. A lightweight in-place implementation for software thread-level speculation. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 223–232, New York, NY, USA, August 2009. ACM.

[19] K. Olukotun and L. Hammond. The future of microprocessors. *Queue*, 3(7):26–29, 2005.

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

[21] C. J. F. Pickett and C. Verbrugge. Software thread level speculation for the java language and virtual machine environment. In *LCPC '05: Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing*, pages 304–318, Berlin / Heidelberg, October 2005. Springer. LNCS 4339.

[22] M. K. Prabhu and K. Olukotun. Exposing speculative thread parallelism in SPEC2000. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 142–152, New York, NY, USA, 2005. ACM.

[23] J. Renau, K. Strauss, L. Ceze, W. Liu, S. R. Sarangi, J. Tuck, and J. Torrellas. Energy-efficient thread-level speculation. *IEEE Micro*, 26(1):80–91, 2006.

[24] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of javascript programs. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 1–12, New York, NY, USA, 2010. ACM.

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

[26] Standard Performance Evaluation Corporation. SPEC CPU2000 v1.3, 2000. http://www.spec.org/cpu2000/.

[27] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. The STAMPede approach to thread-level speculation. *ACM Transactions on Computer Systems*, 23(3):253–300, 2005.

[28] H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[29] WebKit. The webkit open source project, 2009. http://www.webkit.org/.