Thread-Level Speculation as an Optimization Technique in Web Applications - Initial Results

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*

***Abstract*—Web Applications have become increasingly popular as they allow developers to use an uniform platform for user interactions. The dynamic programming language JavaScript used in most Web Applications has performance penalties, that have been addressed by traditional optimization techniques. We have found that while the performance gain of such techniques are positive for a set of established benchmarks, it often fails to improve the performance of real-life Web Applications.**

We suggest Thread-Level Speculation (TLS) at the JavaScript function level to automatically extract parallelism to gain performance. There have been multiple TLS proposals in both hardware and software, but little work has been done within JavaScript. Currently we are implementing our TLS ideas in a state-of-the-art JavaScript engine targeted for embedded mobile devices.

***Keywords*-JavaScript; Multithreading; Parallel Computing; Speculative execution; Runtime environment**

I. Introduction

Current and future processor generations are based on multicore architectures, and future performance increase will mainly come from an increasing number of processor cores. In order to achieve an efficient utilization of an increasing number of processor cores, software needs to be parallel as well as scalable [1], [2], [3].

Due to the simplicity of distribution along with increased platform independence, many applications are moved to the World Wide Web as so called Web Applications. Many of these Web Applications use JavaScript extensively. JavaScript is a dynamically typed, object-based scripting language with run-time evaluation, where execution is done in a JavaScript engine. To preserve platform independence and simplicity, there are currently no support for threading. With the increased popularity of Web Applications and a higher demand for performance, several optimization techniques have been suggested along with a set of benchmarks. Several studies have shown that these benchmarks are unrepresentative [4], [5], [6], and that current optimization techniques often degrades the performance of Web Applications [7]. Therefore alternative optimization techniques and multicore architectures should play a crucial part.

Developing parallel applications are 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 [8]. The performance potential of TLS has been shown for applications with static loops, statically typed languages, and in Java bytecode environments.

Previously we have evaluated the performance of TLS together with the Rhino JavaScript engine and evaluated it's performance with the V8 benchmark [9]. We are extending this study to the SquirrelFish JavaScript engine found in WebKit, and also perform experiments on Web Applications rather than the benchmarks. We have found that we are able to decrease the execution time with thread-level speculation, that function calls are well suited for this technique, and that there is more to gain by using alternative optimization techniques.

II. Background

A. JavaScript

JavaScript is a dynamically typed, object-based scripting language with run-time evaluation often used in association with Web Applications. JavaScript application execution is done in a JavaScript engine, i.e., an interpreter/virtual machine that parses and executes the JavaScript program.

The performance of JavaScript engines have increased significantly during the last years, reaching a very high single-thread performance. However, today no official JavaScript engine supports parallel execution of threads from a single JavaScript program. Further, we have not found any study that addresses the applicability and performance potential of TLS in a dynamically typed scripting language, such as JavaScript.

B. Thread-Level Speculation

TLS aims at dynamically extracting parallelism from a sequential program, and can be implemented both in hardware and in 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 dynamic 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. Unfortunately, one effect of imprecise dependence detection is the risk of a violation that is detected when no actual dependence violation is present.

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. Updating data in-place usually results in higher performance if the number of roll-backs is low, but lower performance when the number of roll-backs is high since the cost of doing roll-backs is high.

C. Software-Based Thread-Level Speculation

There exists a number of different software-based TLS proposals, and we review some of the most important ones.

Bruening et al. [10] proposed a software-based TLS system that targets loops where the memory references are stride-predictable. It was one of the first techniques applicable to while-loops where the loop exit condition is unknown. The results show speed-ups of up to almost five on 8 processors.

Rundberg and Stenström [8] proposed a TLS implementation that resembles the behaviour of a hardware-based TLS system. It supports precise data dependency tracking, but has a high memory overhead. They show a speedup of up to ten times on 16 processors.

Kazi and Lilja developed the course-grained thread pipelining model [11] for exploiting coarse-grained parallelism. They suggest to pipeline the concurrent execution of loop iterations speculatively, using run-time dependence checking. On an 8-processor machine they achieved speed-ups of between 5 and 7.

Bhowmik and Franklin [12] developed a compiler framework for extracting parallel threads from a sequential program for execution on a TLS system. The approach yields speed-ups between 1.64 and 5.77 on 6 processors.

Cintra and Llanos [13] present a software-based TLS system that speculatively executes loop iterations within a sliding window. They managed to reach in average 71% of the performance of hand-parallelized code.

Chen and Olukotun present two studies [14], [15] on how method-level parallelism can be exploited using speculative techniques. 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 [16], [17] developed SableSpMT, a framework for method-level speculation and return value prediction. Their solution is implemented in a Java Virtual Machine (SableVM), and works at the bytecode level. They obtained 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. 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. The results show that their TLS approach reaches in average 77% of the speed-up of hand-parallelized versions.

A study by Prabhu and Olukotun [19] analyzed what types of thread-level parallelism that can be exploited in the SPEC CPU2000 Benchmarks [20]. They also identified a number of obstacles that hinder or limit the usefulness of TLS parallelization.

III. Previous Results

A. Unrepresentative Benchmarks

Established JavaScript benchmarks are often ported from existing benchmark suites. However we measured the JavaScript workload for a large number of popular Web Applications, with in-depth measurements for so-called social networks [4], and we found that certain JavaScript features play an important role in real-life Web Applications [4], [7].

We found that features, which are optimized heavily for in the benchmarks, such as large loops and a large amount of arithmetic instructions (Figure 1), were absent in Web Applications. We found that one reason is that there is no interrupt mechanism in JavaScript, so large loops make the Web Application unresponsive. Large loop-type structures are instead confined into anonymous functions calls made from events.

These observations suggest that attention should be given to the features in JavaScript that come along with being a dynamic programming language. We have found evidence that new multimedia functionalities will be even more dependent on JavaScript and these features [7].

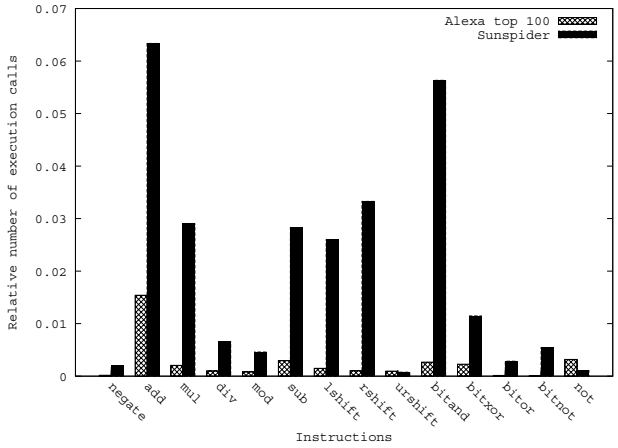

Figure 1. Number of arithmetic instructions in the bytecode produced by Squirrelfish for Sunspider benchmark and for a set of Web Applications

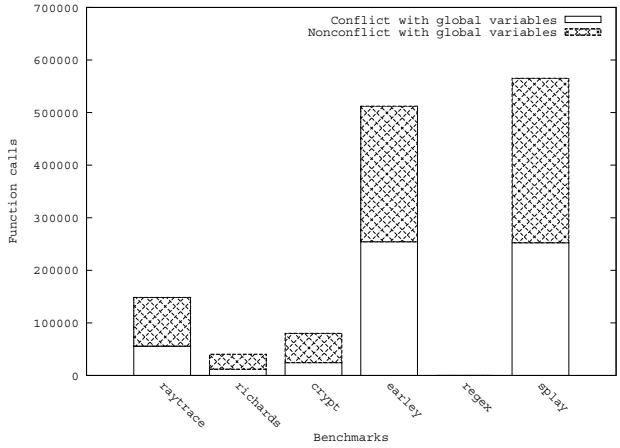

Figure 3. Number of functions with data dependence violations with global variables.

B. Early TLS Results

In [21] and [9] we have done an early implementation of the TLS technique in the Rhino [22] JavaScript engine. In Figure 2 we show some results from our early implementation on a dualcore laptop running Windows Vista and a quadcore workstation running Ubuntu 8.04 Linux, and it shows a modest speedup. The limited speedup is mainly a result of a relative large amount of conflicts between functions and global variables, as shown in Figure 3.

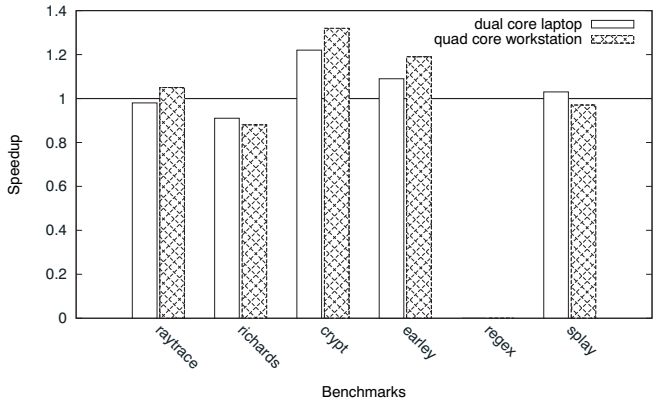

Figure 2. Relative execution time with TLS enabled, normalized to the execution time without TLS enabled. Regex benchmark did not execute correctly on the official Rhino interpreter that was used for our tests.

IV. Ongoing Work

We have acknowledged the difference between Web Applications and the established JavaScript benchmarks. These studies suggested that focusing merely on the benchmarks could lead to optimization strategies that are not effective for Web Applications.

We are currently working on incorporating TLS techniques into WebKit's register-based JavaScript interpreter SquirrelFish. Being a register-based interpreter suggests some more book-keeping challenges when it comes to managing the state before and during speculation. The registers serve as temporary placements of variables. However, the registers also seem to decrease the complexity of duplicating values before speculation. In earlier experiments, with a stack-based interpreter, we were forced to duplicate a large portion of the stack before speculation.

We have found that anonymous and eval function calls are less prone to have conflicts with global variables, and therefore would be better candidates for speculations. In addition, in the initial discussion of TLS, we suggested that for loops, the ideal would be to add one iteration per thread. Due to the lack of an interrupt mechanism, we are forced to use events to simulate large loops, and that anonymous functions were associated with events. Our previous results show that anonymous functions are quite common in Web Applications, and their relative importance increases the longer time the Web Applications are executing. Further, accesses to global variables are rare in such functions. Therefore, we believe that anonymous functions are good candidates for speculation.

V. Future Work

We believe that Thread-Level Speculation is a promising lead for optimization of Web Applications. With the increasing amount of multimedia in Web Application, JavaScript's workload might increase significantly in the near future. Similar applications in a desktop environment have large loops, and as we suggested that loops, events and anonymous functions will play a key role for Web Applications. Further, we have shown that just-in-time compilation techniques have limited or often negative effect on the execution time of Web

Applications [7].

In the near future we will extend our studies with two contributions; We will add additional logic to the speculation, making speculation adaptive, and we will make a study to determine when register-based interpreters are more suited for Thread-Level Speculation than stack-based interpreters.

VI. Conclusion

Thread-level speculation in JavaScript engines is a promising technique to increase the performance of Web Applications. The increase of multimedia workloads in Web Applications could prove this technique even more promising. We will give this workload, along with a different platform more attention in our future work.

Acknowledgment

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

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," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[2] R. McDougall, "Extreme software scaling," *Queue*, vol. 3, no. 7, pp. 36–46, 2005.

[3] H. Sutter and J. Larus, "Software and the concurrency revolution," *Queue*, vol. 3, no. 7, pp. 54–62, 2005.

[4] J. K. Martinsen and H. Grahn, "A methodology for evaluating JavaScript execution behavior in interactive web applications," in *The 9th ACS/IEEE Int'l Conference on Computer Systems and Applications*, December 2011, (to appear).

[5] 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. 27–38.

[6] 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.

[7] J. K. Martinsen, H. Grahn, and A. Isberg, "A comparative evaluation of JavaScript execution behavior," in *Proc. of the 11th Int'l Conference on Web Engineering (ICWE 2011)*, June 2011, (to appear).

[8] 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.

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

[10] 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.

[11] I. H. Kazi and D. J. Lilja, "Coarse-grained thread pipelining: A speculative parallel execution model for shared-memory multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 9, pp. 952–966, 2001.

[12] A. Bhowmik and M. Franklin, "A general compiler framework for speculative multithreading," in *SPAA '02: Proc. of the 14th ACM Symp. on Parallel Algorithms and Architectures*, 2002, pp. 99–108.

[13] M. Cintra and D. R. Llanos, "Toward efficient and robust software speculative parallelization on multiprocessors," in *Proc. of the 9th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2003, pp. 13–24.

[14] M. K. Chen and K. Olukotun, "Exploiting method-level parallelism in single-threaded java programs," in *PACT '98: Proc. of the 1998 Int'l Conference on Parallel Architectures and Compilation Techniques*, 1998, pp. 176–184.

[15] ——, "The Jrpm system for dynamically parallelizing Java programs," in *ISCA '03: Proc. of the 30th Annual Int'l Symp. on Computer Architecture*. New York, NY, USA: ACM, 2003, pp. 434–446.

[16] 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*, 2005, pp. 59–66.

[17] ——, "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.

[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*, August 2009, pp. 223–232.

[19] M. K. Prabhu and K. Olukotun, "Exposing speculative thread parallelism in SPEC2000," in *PPoPP '05: Proc. of the 10th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2005, pp. 142–152.

[20] Standard Performance Evaluation Corporation, "SPEC CPU2000 v1.3," Warrenton, VA, USA, 2000, http://www.spec.org/cpu2000/.

[21] J. K. Martinsen and H. Grahn, "Thread-level speculation for web applications," in *Second Swedish Workshop on Multi-Core Computing (MCC-09)*, November 2009, pp. 80–88.

[22] Mozilla, "Rhino JavaScript interpreter," 2010, http://www.mozilla.org/rhino/.