Thread-Level Speculation for Web Applications

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

Abstract

Thread Level Speculation (TLS) has been suggested as a mean to automatically (or semi-automatically) extract parallelism from sequential programs. While there have been multiple attempts both in hardware and software to implement real time TLS, to the best of our knowledge all attempts have so far been on a byte code level or with statically typed languages.

In this study, we examine the potential of TLS for Web Applications, using the popular scripting language JavaScript(JS). We have chosen to execute the programs by traversing their parse trees, taking advantage of information from the programming language that are normally lost when compiled to, e.g., byte code.

We have performed a test where we automatically have divided the execution of the parsing tree among $1, 2, 4,$ *and* 8 *cores for four benchmark programs. We have found that this approach has a small number of rollbacks (i.e. error correction when speculation fails) and significantly increases the performance of our benchmarks.*

1. Introduction

Today we are in the middle of a paradigm shift in the computer industry. Previous processor generations were based on uni-processor technology. The performance increase came from a steady increase in clock frequency and architectural inventions [12]. Current and future processor generations are based on multicore architectures, where the performance increase are expected to mainly come from an increasing number of cores on a chip [22]. 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 [2, 19, 30].

Another important trend is that more and more applications are moved to the World Wide Web [31]. There are several reasons for that, e.g., accessibility and mobility. In order to develop Web Applications, new programming languages have emerged. One such language is JavaScript [15]. JavaScript is a dynamically typed, object-based scripting language with run-time evaluation, and application execution is done in a JavaScript engine. 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.

Developing parallel applications is both time consuming and error-prone, and therefore we would like to ease the burden of the programmers. This can be done by using static analysis of sequential programs to automatically identify parallel sections in the code, usually in static loops [1, 6]. Another approach is to dynamically extract parallelism from a sequential program using speculative methods, known as Thread-Level Speculation (TLS) or Speculative Multithreading (SpMT). TLS can be implemented in both hardware, e.g., [8, 26, 29], and software, e.g., [7, 17, 21, 24, 27].

Software TLS approaches usually extract parallelism from loops in sequential applications. A number of consecutive loop iterations are speculatively executed in parallel, a data dependency check mechanism is used to detect dependecy violations between reads and writes in parallel iterations, and finally, a roll-back mechanism is needed when a dependency violation is detected in order to recover to an earlier safe point. The performance potential of TLS has been shown for applications with static loops, e.g., [18], but there is no study that has evaluated the performance potential of TLS for dynamically typed, object-based, scripting languages such as JavaScript.

In this paper we describe an approach to apply TLS to dynamically typed JavaScript applications, and evaluate the performance potential of TLS for four different JavaScript applications. In our study, JavaScripts are executed by an interpreter, for-loops are distributed speculatively by splitting the parse-tree on different interpreter cores, and data dependency violations are detected and solved during run-

time. Our initial results show that TLS seems to be a viable approach also for dynamically typed scripting languages. For the studied applications we achieved both a balanced load between the processors as well as a low number of roll-backs.

This paper makes two main contributions:

- This is the first study that addresses how thread-level speculation can be applied and used for a dynamically typed scripting language. In our case we have used JavaScript, but the approach is applicable also to other scripting languages.
- This is the first study that presents results on performance and execution behavior of scripting languages when TLS methods are used.

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 will present the general principles of thread-level speculation and some previous implementation proposals. We will also 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., [8, 26, 29], and software, e.g., [7, 17, 21, 24, 27]. 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 in-place 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 will review some of the most important ones here.

Bruening et al. [7] 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 slowdowns for some rare cases.

Rundberg and Stenström [27] 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 [4].

Kazi and Lilja developed the course-grained thread pipelining model [17] 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 [4]). On an 8-processor machine they achieved speed-ups of between 5 and 7. They later extended their to also support Java programs [16].

Bhowmik and Franklin [5] 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[11] 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 [28] and Perfect Club [4] Benchmark suites.

Chen and Olukotun present in two studies [9, 10] 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 [23, 24] 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. [21] 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. [18] evaluated the performance potential of TLS using the SPEC CPU2000 Benchmarks [28]. 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 [25] 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. No study have been found that addresses the applicability and performance potential of TLS in a dynamically-typed scripting language, such as JavaScript.

2.2 JavaScript

An important trend in application development today is that more and more applications are moved to the World Wide Web [31]. There are several reasons for that, e.g., accessibility and mobility. Users would like to access information located anywhere from locations anywhere. In order to develop web applications, new programming languages have emerged. One such language is JavaScript [15], which has been used especially in client-side applications, i.e., in web browsers, but are also applicable in the server-side applications.

JavaScript 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. Exemples of JavaScript engines are Google's V8 engine [14], WebKit's Squirrelfish [32], and Mozilla's SpiderMonkey and TraceMonkey [20]. 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 for the JS language, we have implemented a JS interpreter, written in the Python programming language. The JS program is first transformed into a prefix parse tree t_p, which in turn is evaluted by traversion. We perform TLS the following way: Since we have a parse tree structure t_p, we can easily identify the first for-loop, which would be a subtree $t_f \subseteq t_p$. Let us assume that t_f iterates from $0 \ldots 64$. Next, we create four copies from t_f:, t_1, t_2, t_3 and t_4 and modify each of them so they iterate from $0 \ldots 16, 16 \ldots 32, 32 \ldots 48$ and $48 \ldots 64$ respectively.

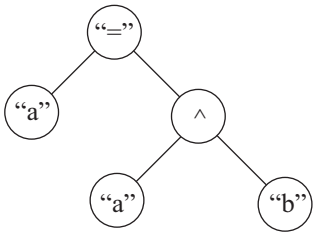

Figure 1. A parse tree for the expression $a = a \wedge b$.

There is obviously a potential for name conflicts when running this concurrently, so we rename each of the variables that are declared in the body and in the for-loop (Figure 2), thereby reducing the chance of such conflicts, which in turn would require us to do rollbacks.

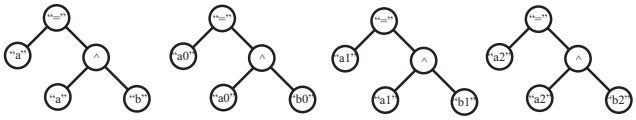

Figure 2. The four subtrees with renamed nodes.

Python is suitable for rapid prototyping, however it lacks proper support for threads which is the most common mechanism used in TLS. Therefore we simulate concurrency and we have chosen the following strategy to do so: Let us assume that we want to simulate a system of 4 processors. To be able to trace the interaction between the variables defined in the trees, we use the following technique to extract the traversion of the parse tree. For instance consider the tree in Figure 1. From the traversion of this tree we could generated the following Python program

```
mem["a"] = mem["a"] ^ mem["b"]
```

The generated program can in turn be executed, and will (in our cases) create indential results as executing the parse tree by traversion.

Next, assume that the parse tree in Figure 1 is the body of the for-loop t_f, that we will divide among 4 processors.

We traverse each of the four trees in turn, node by node. Each time we read from a variable, we first check if the current processor was the last to write to that variable. We name this operation memory access. This information is stored in a hashlist $umem$ that contains the variable name, and the number of the process that modified the variable last. If this equals the process that currently tries to read from the variable, we write the process number to $umem$, if not we perform a rollback. This is done inside the method $conflict$, that checks if the variables we are about to read from have been written to by the same process as the process that is about to read them. We try to combine the trees, such that each process is executing in turn, however this might not always be possible, since the output from the parse trees might vary in size. If this is the case, where one tree is larger than the others we do the following: Try executing the trees in turn, and when one of the other trees are completed, we continue to execute the remaining trees.

```
#check for conflict
if(conflict(["a","b"],umem,processor0):
 #if a conflict is detected, a rollback performed
 rollback()
#perform the actual calculation
mem["a"] = mem["a"] ^ mem["b"]
#assosiate the variable with the current processor
umem["a"] = processor0
if(conflict(["a0","b0"],umem,processor1):
 rollback()
mem["a0"] = mem["a0"] ^ mem["b0"]
umem["a0"] = processor1
if(conflict(["a1","b1"],umem,processor2):
 rollback()
mem["a1"] = mem["a1"] ^ mem["b1"]
umem["a1"] = processor2
if(conflict(["a2","b2"],umem,processor3):
 rollback()
mem["a2"] = mem["a2"] ^ mem["b2"]
umem["a2"] = processor3
```

This simulation disregards numerous important factors in TLS, such as problems related to synchronizations, however it gives us an indication of the number of rollbacks required, which is important for the performance when optimizing with the TLS.

4 Experimental Setup

To evaluate our TLS for JS, we use the following programs; 8queens, fibonacci, fractal and raytrace. 8queens is a solver for 8queens puzzle, fibonacci writes the fibonacci sequence for $n = 100$, fractal draws a mandelbrot fractal,

and raytrace is a simple Whitted based raytracer that draws a scene consisting of nine spheres and a plane. Common for these programs is that they consist of one or more for-loops, and all of them can be easily paralellized manually.

Table 1. Benchmark programs used in the study.

Program	Description	Data set size
Fibonacci	Fibonacci sequence	$n = 100$
Fractal	Mandelbrot fractal	$n = 1500$
Raytrace	Whitted raytracer	9 spheres and a plane
8queen	8-queens puzzle	$n = 8$

We have performed the following experiments: We have divided the first for-loop into $2, 4$ and 8 subtrees. This happens the following way; We generate a program from the parse tree, where each processor execute in a round-robin manner. The generated program is measured, and we measure the number of rollbacks along with the number of memory accesses required during its execution. In addition we have looked at how the workload is divided between the processors, for the various execution scenarios. We have also defined a potential of speed up by the following formula $memoryaccesses/((memoryaccesses + rollbacks)/processors)$

5 Experimental Results

5.1 Fibbonacci

We start by analyzing the results from Fibbonacci. We see from Table 2, that the results seem to evenly distributed among the processors for 2 and 4 processors, however not for 8. From the description, we recall that n was equal to 100, which obviously is divisible by 2 and 4, however not by 8. (Which would mean that we would divide the for-loops ranging from 12,13,12,13,12,13,12 and 13 iterations), thereby seeing the unevenly distribution in Figure 3.

There is also a quite large number of rollbacks; 197 for 1219 memory accesses, as shown in Table 2. We also observe that the number of rollbacks increases with the number of processors. We can explain this in the following way: The number of variable name conflicts increases with the number of processors. However, the number of rollbacks does not double with the number of processors (from 2 to 4, the number of rollbacks increases by a factor of 0.185, from 4 to 8 the number of rollbacks increases with 0.344). The rollbacks for the Fibonacci case are linked to the construction of the Fibonacci program, where there are two variables

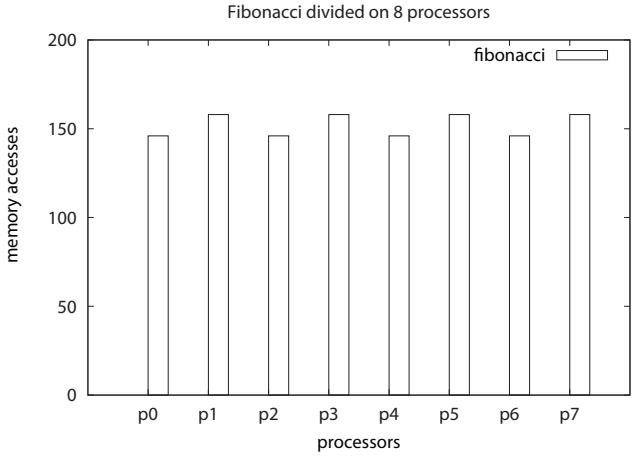

Figure 3. Fibonacci divided on 8 processors.

that are constantly manipulated, but not declared in the for-loop.

5.2 Fractal

The next application we have studied is Fractal, which calculates a Mandelbrot Fractal set. We see in Table 2 that Fractal has a low number of rollbacks, and that the number of rollbacks increases only by 2 from 4 to 8 processors. This can be explained in two ways; All the variables manipulated inside the for-loop is declared inside the for-loop, and with our renaming routine these variables becomes unique, minimizing the chance for name conflicts, which in turn would require a rollback. The workload of Fractal isn't evently distributed among the processors. This can be seen in the graphs in Figure 4 and Figure 5, where we see that processor $p3, p4$ and $p5$ and $p1$ and $p2$ accounts for most of the work. This can be explained the following way; Large field of black color indicates areas in the Fractal where the value converges to infitity. This can be seen in Figure 6 together with Figure 4, where we see that column assosiated with $p2$ contains the largest black field in the image, thereby requiring $p2$ to do most of the work.

To remedee this problem, we have tried the following strategy: divide the two for-loops for the fractal program recursively. We start by dividing the outer for-loop in 4, then each for-loop in the body of the other for-loop in 4. For the fractal program, this creates 16 for-loop subtrees. When we measure the amount of work performed, it is much more evently distributed (Figure 8) than by just dividing the outer for-loop. We can also combine them so it is suitable for a smaller number of processors, however since we are testing it on a larger number of processors, tests show that this requires a somewhat larger number of rollbacks. To be able to compare we recursively divide for-loop in 2 (using a to-

Table 2. The number of rollbacks and read memory statements for $2, 4$ and 8 processors.

	2 processors			4 processors			8 processors		
Program	rollbacks	mem.access	speed-up	rollbacks	mem.access	speed-up	rollbacks	mem.access	speed-up
Fibonacci	197	1219	1.72	242	1223	3.33	369	1231	6.15
Fractal	7	209944	1.99	46	209944	3.99	48	209944	7.99
Raytrace	7124	593927	1.97	18921	593927	3.87	33123	593927	7.57
8queen	0	8053522	2	0	8053522	4	0	8053522	8

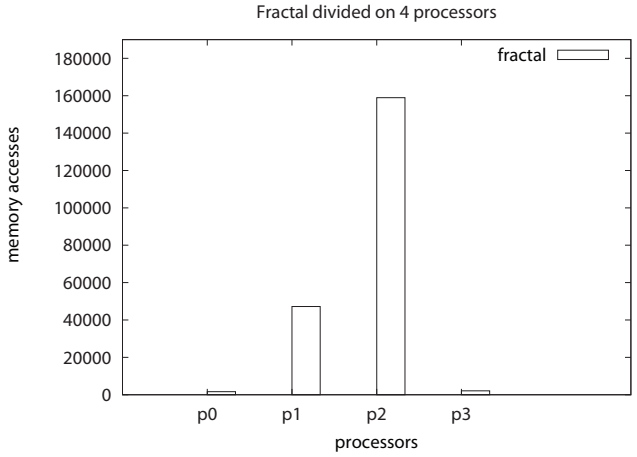

Figure 4. Fractal divided on 4 processors.

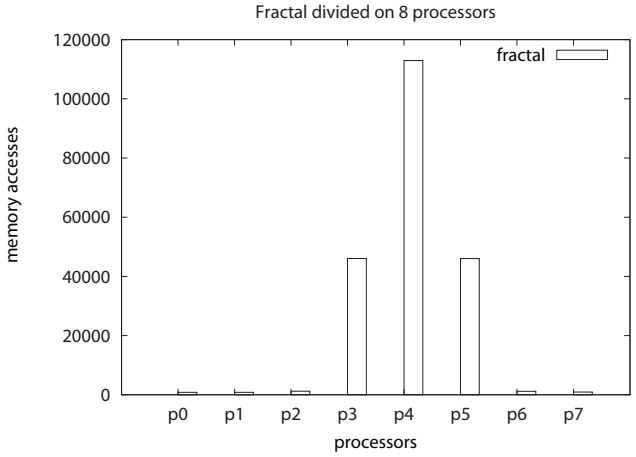

Figure 5. Fractal divided on 8 processors.

tal of 4 processors). Evaluating the program required 57 rollbacks, which is 11 more than in Table 2. However, if we view how the work is distributed among the processors, with a standard deviation, we find that the standard deviation for the processors in Figure 4 is 74127.3 against 407.4 for Figure 7.

5.3 Raytrace

The Raytrace application performs Whitted-based raytracing of scene with 9 spheres and one plane. We see in Figure 9 that the work distribution of 2 processors for the raytracer program is quite even. We suspect that this is due to that the raytraced image is almost symmetrical. The raytracer has a quite large number of rollbacks, which we suspect could be caused by the recursive calls to simulate reflection. There is also a larger number of writes to global variables inside the for-loop. This symmetry does however not apply when we increase the number of processors to 4 and 8. From Figure 10, we see that there is a larger number of reflections (indicated by the arrow) for processor $p1$ and $p2$, which again requires $p1$ and $p2$ to do a larger number of recursive calls than $p1$ and $p2$.

5.4 8queens

The final application that we have studied is 8queen, which solves the problem of putting 8 queens on an 8×8 chess board such that none of them is able to capture any other queen. We see from Table 2 that dividing the outer loop in $2, 4$ and 8 creates no rollbacks for the 8queen program. We suspect that this is because there is no outer variables defined, and there is no dependency between the variables in various for-loops. (The program is constructed as a total of eight nested for-loops).

6 Conclusions

Web Applications are an emerging application domain, enabling information and services to be accessible from everywhere. New languages, e.g., JavaScript [15], have emerged to support this development. Although a number of high-performing JavaScript engines exist, e.g., V8 [14], Squirrelfish [32], and SpiderMonkey [20], none of them exploit parallelism in order to enhance the performance.

In this paper we have simulated how thread-level speculation (TLS) might be used to increase the performance of a

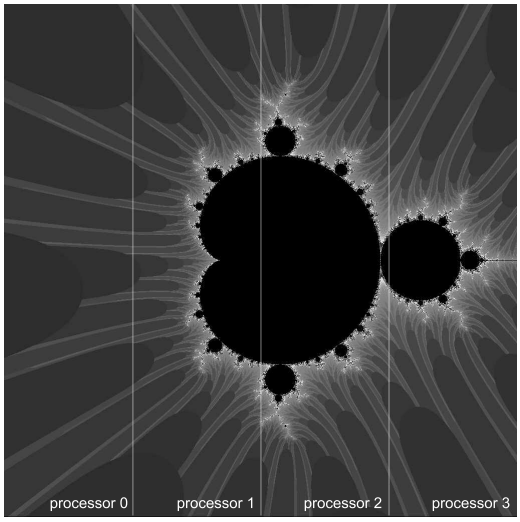

Figure 6. Graphical image for Fractal divided on 4 processors.

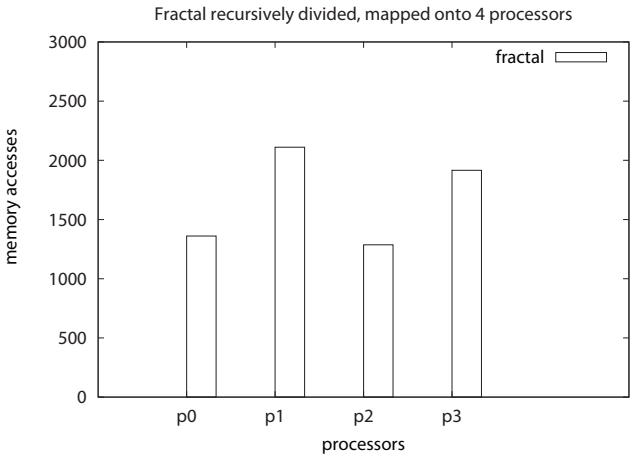

Figure 7. Fractal where we divide the for-loops recursively on 4 processors.

dynamically typed, scripting language, i.e., JavaScript. To the best of our knowledge, this is the first study of using TLS for this type of languages. Our approach is based on an interpreter that speculatively split up the parse tree into several parallel entities that can be executed in parallel.

This study gave some hints of the potential of TLS in a Web Application context. We will continue our work by adding TLS functionality into a more established javascript interpretator.

Acknowledgement

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

References

[1] V. Adve, J. Mellor-Crummey, M. Anderson, K. Kennedy, J. Wang, and D. Reed. An integrated compilation and performance analysis environment for data parallel programs. In *Proceedings of Supercomputing '95*, pages 1370–1404, November 1995.

[2] 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.

[3] I. M. Author. Some related article I wrote. *Some Fine Journal*, 99(7):1–100, January 1999.

[4] 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.

[5] 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.

[6] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with polaris. *Computer*, 29(12):78–82, 1996.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

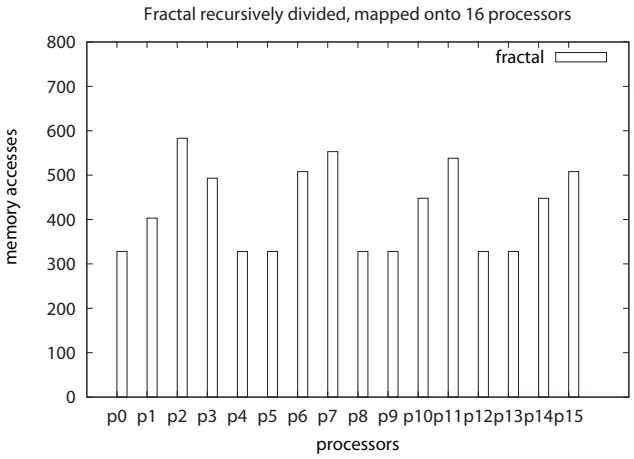

Figure 8. Fractal where we divide the for-loops recursively on 16 processors.

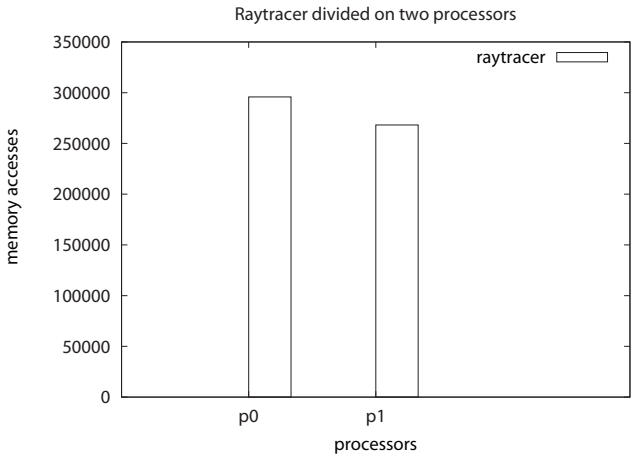

Figure 9. Raytrace divided on 2 processors.

[11] 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.

[12] J. Emer, M. D. Hill, Y. N. Patt, J. J. Yi, D. Chiou, and R. Sendag. Single-threaded vs. multithreaded: Where should we focus? *IEEE Micro*, 27(6):14–24, Nov/Dec 2007.

[13] A. N. Expert. *A Book He Wrote*. His Publisher, Erewhon, NC, 1999.

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

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

[16] I. H. Kazi and D. J. Lilja. Javaspmt: A speculative thread pipelining parallelization model for java programs. In *IPDPS'00: Proceedings of the 14th International Parallel and Distributed Processing Symposium*, page 559, Los Alamitos, CA, USA, May 2000. IEEE Computer Society.

[17] 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*, 12(9):952–966, 2001.

[18] 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: Proceedings of the 20th Annual International Conference on Supercomputing*, page 24, New York, NY, USA, 2006. ACM.

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

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

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

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

[31] W3C. World wide web consortium, 2009. http://www.w3c.org/.

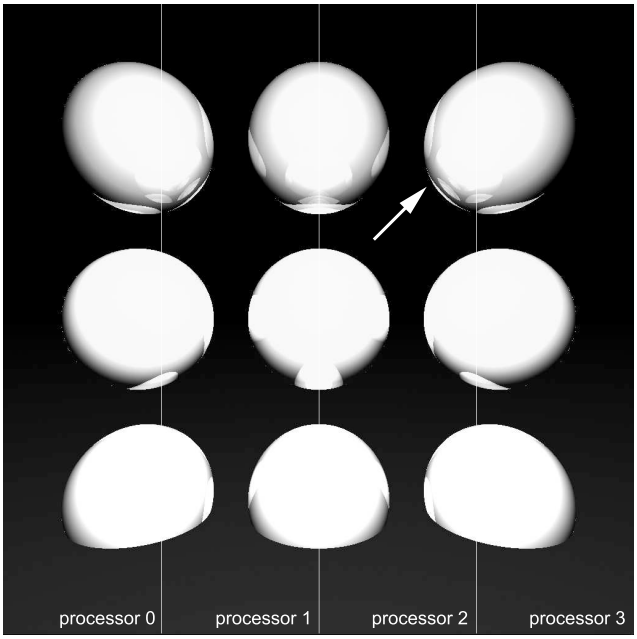

Figure 10. Raytrace divided on 4 processors.

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