Dynamic Parallelization of JavaScript Applications Using an Ultra-lightweight Speculation Mechanism

Mojtaba Mehrara Po-Chun Hsu Mehrzad Samadi Scott Mahlke
University of Michigan, Ann Arbor
{mehrara,pchsu,mehrzads,mahlke}@umich.edu

Abstract

As the web becomes the platform of choice for execution of more complex applications, a growing portion of computation is handed off by developers to the client side to reduce network traffic and improve application responsiveness. Therefore, the client-side component, often written in JavaScript, is becoming larger and more compute-intensive, increasing the demand for high performance JavaScript execution. This has led to many recent efforts to improve the performance of JavaScript engines in the web browsers. Furthermore, considering the wide-spread deployment of multicores in today's computing systems, exploiting parallelism in these applications is a promising approach to meet their performance requirement. However, JavaScript has traditionally been treated as a sequential language with no support for multithreading, limiting its potential to make use of the extra computing power in multicore systems. In this work, to exploit hardware concurrency while retaining traditional sequential programming model, we develop ParaScript, an automatic runtime parallelization system for JavaScript applications on the client's browser. First, we propose an optimistic runtime scheme for identifying parallelizable regions, generating the parallel code on-the-fly, and speculatively executing it. Second, we introduce an ultra-lightweight software speculation mechanism to manage parallel execution. This speculation engine consists of a selective checkpointing scheme and a novel runtime dependence detection mechanism based on reference counting and range-based array conflict detection. Our system is able to achieve an average of 2.18x speedup over the Firefox browser using 8 threads on commodity multi-core systems, while performing all required analyses and conflict detection dynamically at runtime.

1 Introduction

JavaScript was developed at Netscape in the early 1990's under the name Mocha to provide enhanced user interfaces and dynamic content on websites. It was released as a part of Netscape Navigator 2.0 in 1996, and since then it has become standard with more than 95% of web surfers using browsers with enabled JavaScript capabilities. Dynamically downloaded JavaScript programs combine a rich and responsive client-side experience with centralized access to shared data and services provided by data centers. The uses of JavaScript range from simple scripts utilized for creating menus on a web page to sophisticated applications that consist of many thousands of lines of code executing in the user's browser. Some of the most visible applications, such as Gmail and Facebook, enjoy widespread use by millions of users. Other applications, such as image editing applications and games, are also becoming more commonplace due to the ease of software distribution.

As the complexity of these applications grows, the need for higher performance will be essential. However, poor performance is the biggest problem with JavaScript. Figure 1 presents the performance of an image edge detection algorithm implemented in a variety of programming languages on an Intel Core i7 processor. Not surprisingly, the C implementation is the fastest. Following are the C++ implementation with 2.8x slowdown and the Java version with 6.4x slowdown. The default JavaScript implementation is 50x slower than the C implementation. The performance gap occurs because JavaScript is a dynamically typed language and is traditionally executed using bytecode interpretation. Interpretation makes JavaScript execution much slower in comparison to the native code generated for statically typed languages such as C or C++. Using the trace-based optimizer in Firefox, TraceMonkey [15], that identifies hot execution traces in the code and compiles them into native code, the JavaScript execution is brought down in line with the Java implementation which is still 6.9x slower.

Bridging this performance gap requires an understanding of the characteristics of JavaScript programs themselves. However, there is disagreement in the community about the forms of JavaScript applications that will dominate and thus the best strategy for optimizing performance. JSMeter [25] characterizes the behavior of JavaScript applications from commercial websites and argues that long-running loops and functions with many repeated bytecode instructions are un-

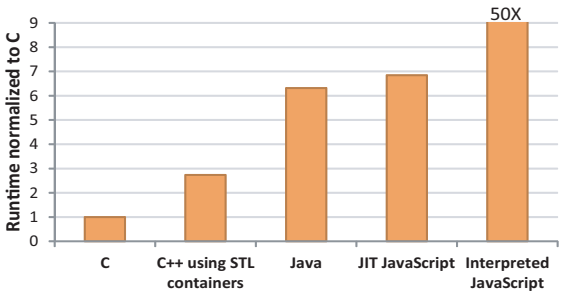


Figure 1: Performance of JavaScript, compared to C, C++ and Java.

common. Rather, they are mostly event-driven with many thousands of events being handled based on user preferences. We term this class of applications as *interaction-intensive*.

While this characterization reflects the current dominance of applications such as Gmail and Facebook, it may not reflect the future. More recently, Richards *et al.* [27] performed similar analyses on a fairly large number of commercial websites and concluded that in many websites, execution time is, in fact, dominated by hot loops, but less so than Java and C/C++. Furthermore, an emerging class of online games and client-side image editing applications are becoming more and more popular. There are already successful examples of image editing applications written in ActionScript for Adobe Flash available at `http://www.photoshop.com`. There are also many efforts in developing online games and gaming engines in JavaScript [1, 6]. These applications are dominated by frequently executed loops and functions, and are termed *compute-intensive*.

The main obstacle preventing wider adoption of JavaScript for compute-intensive applications is historical performance deficiencies. These applications must be distributed as native binaries because consumers would not accept excessively poor performance. A circular dependence has developed where poor performance discourages developers from using JavaScript for compute-intensive applications, but there is little need to improve JavaScript performance because it is not used for heavy computation. This circular dependence is being broken through the development of new dynamic compilers for JavaScript. Mozilla Firefox's TraceMonkey [15] and Google Chrome's V8 [2] are two examples of such efforts. While these engines address a large portion of inefficiencies in JavaScript execution, they do not provide solutions to scale performance.

With multicore architectures becoming the standard for desktop and server markets, and on the near-term road map for smart phones, it is apparent that parallelism must be exploited in JavaScript applications to sustain their ever increasing performance requirements. However, these applications are generally single-threaded because the language and run-time system provide little concurrency support. The primary problem is that the document object model (DOM), used by JavaScript to interact with web pages and browser state, does not have a model for shared access. So, for example, since JavaScript has no internal locking mechanism, two threads may invoke multiple event handlers that change the same DOM object and create either race conditions or correctness bugs. Considering this limitation, developing parallel JavaScript applications manually, would be cumbersome and error-prone.

To exploit hardware concurrency, while retaining the traditional sequential programming model, this paper focuses on dynamic parallelization of compute-intensive JavaScript applications. Our parallelization technique, called *ParaScript*, performs automatic speculative parallelization of loops and generates multiple threads to concurrently execute iterations in these applications. A novel and low-cost software speculation system detects misspeculations occurred during parallel execution and rolls back the browser state to a previously correct checkpoint. Our techniques are built on top of the TraceMonkey engine [15], thus we retain the benefits of trace-based optimizations.

While speculative loop parallelization has an extensive body of prior work [11, 13, 21, 24, 29, 30, 35], the key new challenge here is performing parallelization efficiently at runtime, while ensuring correct execution through runtime speculation without any extra hardware support. Recent efforts for speculative parallelization of C/C++ applications on commodity hardware ([21, 24, 30]) make extensive use of static analysis along with static code instrumentation with speculation constructs. Straight-forward application of these static parallelization techniques is ineffective for a number of reasons including the inability to perform whole program analysis, and expensive memory dependence analysis and profiling at runtime. In addition, without effective use of static analysis, considerable overhead of per-access logging and commit time operations involved in software speculation systems such as software transactional memories [8, 19], makes immediate use of these systems impractical in a dynamic setting. Previous proposals for dynamic parallelization [13] are also not suitable for our target due to high dependency on expensive extra hardware support. In this work, we show that practical speedups can be achieved on commodity multi-core hardware with effective runtime analysis and efficient speculation management.

In particular, the main contributions of this work are:

- An optimistic, yet effective technique for automatically identifying parallelizable loops dynamically at runtime and a code generation scheme to create the parallelized version of the JavaScript application.

- An ultra-lightweight software speculation system consisting of selective checkpointing of the system state and a novel runtime dependence detection mechanism based on conflict detection using reference counting and range-based checks for arrays.

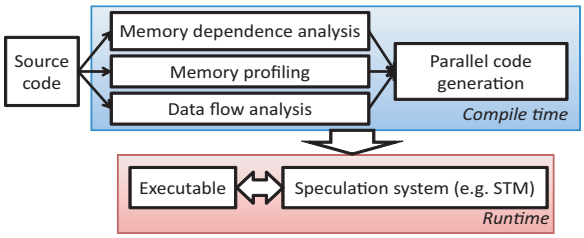


Figure 2: Static parallelization flow.

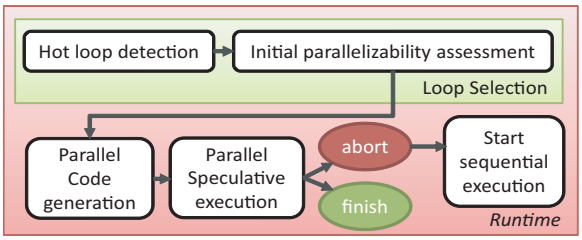


Figure 3: ParaScript dynamic parallelization flow.

2 Dynamic Parallelization Challenges

There is a long history of static techniques for automatic and speculative parallelization of scientific and general purpose applications. A typical static parallelization framework is depicted in Figure 2. This framework uses memory dependence analysis (pointer analysis), memory access profiling and data flow analysis to find candidate loops for parallelization, after which generates the parallel code. This code is later run speculatively on the target system using a software or hardware memory speculation mechanism.

Static frameworks are usually optimized to achieve the best runtime performance by employing complex and often time-consuming compile-time analysis. However, in a highly dynamic setting like that of JavaScript execution, where the code is run via interpretation or dynamic just-in-time (JIT) compilation on the client's browser, applying these analyses at compilation time would be too expensive and will offset all potential parallelization benefits. It might even lead to slowdown in some cases. For instance, [16] recently proposed a static JavaScript pointer analysis for security purposes. They show that their analysis takes about 3 seconds to complete on a set of JavaScript programs with an average of 200 lines of code. While being an effective and fairly fast analysis for offline purposes, if the analysis were to be used at runtime, it could become quite prohibitive even for programs of that size, let alone larger applications that consist of hundreds of lines of code [27]. In addition to compiler analysis, the overhead of runtime memory profiling would be unacceptable – the work in [20] reported 100x to 500x increase in the execution time as a result of performing static memory profiling on C/C++ applications. Therefore, with the exception of simple data flow analysis and code generation (both of which are already done at runtime in JavaScript engines), a dynamic parallelization framework can not afford to utilize any of the static compile time steps in Figure 2.

Furthermore, by targeting commodity systems, an efficient and low-cost software speculation mechanism is required. Traditional software speculation mechanisms (e.g. STMs) focus on flexibility and scalability to support a wide variety of speculative execution scenarios and thereby increase the runtime by 2-5x [12]. Large overheads of memory operation instrumentation along with expensive checking mechanisms are two main sources of overhead. Recent proposals [21, 24, 30] introduce customized speculation systems for parallelization of C/C++ applications. However, as was reported in [21], without the aid of static memory analysis and profiling, their overhead could go up to 4x compared to sequential as well.

With these challenges in mind, we approach dynamic parallelization of JavaScript applications by proposing the following techniques:

Light-weight dynamic analysis: Limited information is carefully collected early on at runtime, based on which initial parallelization decisions are made. A low-cost and effective initial assessment is designed to provide optimistic, yet surprisingly accurate decisions to the speculation engine. Loops that require complicated transformations, high-cost runtime considerations or have high misspeculation risk are not chosen to be parallelized.

Low-cost customized speculation mechanism: A lean and efficient software speculation mechanism is proposed, which is highly customized for providing loop-level speculation support in JavaScript programs. A low-cost conflict detection mechanism is introduced based on tracking scalar array access ranges and reporting conflict if two threads access overlapping ranges. Parallelism-inhibiting heap references are also detected at runtime to guarantee correct execution. A reference counting scheme is introduced to detect potential conflicts in arrays of objects. Finally, an efficient checkpointing mechanism is also designed that is specialized for the JavaScript runtime setting. Two checkpointing optimizations, selective variable checkpointing and array clone elimination, are introduced to further reduce speculation overhead.

3 Dynamic Analysis and Code Generation

In this work, we propose a dynamic mechanism, called ParaScript, for exploiting parallelism in JavaScript applications. Figure 3 shows the ParaScript execution flow at runtime. It first selects candidate hot loops for parallelization. Subsequently, in the dependence assessment step, using a mixture of data flow analysis and runtime tests, it detects immediately identifiable cross-iteration memory dependences during the execution of the first several loop iterations. In case of no immediate memory dependence, ParaScript dynamically generates a speculatively parallel version of the

loop and inserts necessary checkpointing and conflict detection constructs. The detailed process of instrumentation with these constructs as well as their implementation is presented in Section 4.

The runtime system continues by executing the parallel loop speculatively. In case a cross-iteration dependence is found, the parallel loop is aborted, the system reverts back to the previous correct checkpoint and runs the loop sequentially. The failed loop is also blacklisted to avoid parallelization in further loop invocations in the same run. Furthermore, a cookie is stored in the browser's cache to blacklist the loop in future application executions as well.

To keep the framework and speculation rollback overhead low, only loops whose number of iterations is known at runtime (DOALL-counted, e.g. `for` loops) are considered for dynamic speculative parallelization. Implementing a general framework similar to [35] which handles DOALL-uncounted loops (e.g. `while` loops) needs a more elaborate framework and more frequent checkpointing, imposing negative impact on our goal of having a low-cost near-certain speculation and infrequent checkpointing mechanism.

Loops that contain document object model (DOM) interactions, HTTP request functions and several JavaScript constructs such as the `eval` keyword, `Function` object constructor, and `setTimeout` and `setInterval` keywords are not parallelized. Parallelizing loops with DOM interactions requires locking mechanisms on DOM elements inside the browser and also a heavy-weight checkpointing mechanism to take snapshots of DOM state at various times. The `eval` keyword, `Function` constructor, `setTimeout` and `setInterval` take strings as arguments and execute them like normal JavaScript code at runtime. The problem with using these constructs is that they introduce new code at runtime and the compiler has no access to the string contents and thereby, no analysis can be done on it before execution. In the following subsections, ParaScript stages are explained in more detail.

3.1 Dynamic Parallel Loop Selection

Loop selection process essentially consists of two steps: detecting hot loops and performing initial analyses to determine parallelization potential. During runtime, after a loop execution is detected, it is marked as hot if it passes the initial selection heuristic. This runtime heuristic takes into account iteration count and number of instructions per iteration. In nested loops, if the outer loop's number of iterations is higher than a threshold, it is given the priority for being marked as hot by the system. Therefore, even if an inner loop is detected as hot, the system holds off on passing it to the next stage until a decision about the outer loop is made. However, if the outer loop turns out not to be hot or fails the dependence assessment, the inner loop is marked as hot. This mechanism is based on the intuitive assumption that parallelizing outer loops is always more beneficial and avoids premature parallelization of inner loops.

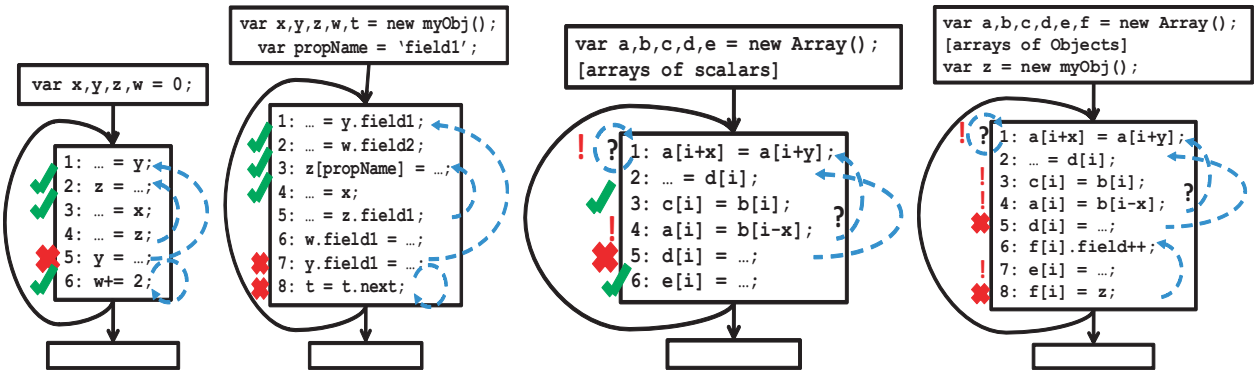
After a hot loop is detected, the system proceeds with the initial dynamic dependence assessment. There have been recent proposals for static pointer analysis in JavaScript [16]. There are also array analysis techniques such as the Omega Test [23] which could be employed for analyzing JavaScript code. However, due to the tight performance and responsiveness constraints on JavaScript execution, the overhead of using these traditional static analysis techniques at runtime in ParaScript is quite prohibitive. In order to compensate for lack of static information and prior memory profiles, several tests are performed during the dependence assessment stage at runtime, in addition to the simple data flow analysis. Analysis is categorized based on four variable types: scalars, objects, scalar arrays and object arrays.

Scalars: Figure 4(a) shows an example of a loop with various forms of scalar dependences. ParaScript avoids parallelizing loops that contain any cross-iteration scalar dependences other than reduction and temporary variables. Using basic data flow analysis (with use-def chains), all instances of cross-iteration scalar dependences can be found [33].

One example of such dependence is variable `y` in Figure 4(a) which is both a live-in and is written inside the loop. Reduction variables such as accumulator (`w` in Figure 4(a)) and min/max variables are also instances of such dependences and can be found by simple pattern matching on the code. Furthermore, global temporary variables that are used inside each iteration and do not carry values past the iteration scopes are also identified (variable `z` in the example). A scalar which is used in the loop, but is not live-in belongs to the mentioned temporary variable category. These variables can be safely privatized, allowing the loop iterations to be executed in parallel. Scalar dependences caused by temporary locals are ignored, because they will automatically become private to each parallel thread.

Object references: Since JavaScript has no pointer arithmetic, objects can also be dealt with using data flow analysis. Figure 4(b) is an example of a loop with object dependences. Initially, statically resolvable object field accesses are identified and resolved. In JavaScript, object properties and methods can be accessed both using the "." operation and also by passing the field name as string to the object name as in line 3 of the code in Figure 4(b). In this example, `propName` is a constant string and can be resolved using a local constant propagation analysis [33] and converted to an access using the "." operation (`z.field1 = ...;`).

After this initial step, data flow analysis is used to find cross-iteration dependences caused by object references. Similar to scalars, a live-in reference which has been written inside the loop is an example of such dependences (`y.field1` in 4(b)). Likewise, global temporary objects



(a) Scalar dependences (b) Object dependences (c) Scalar array dependences (d) Object array dependences

Figure 4: Examples of dependence instances in various variable categories. Solid and dashed arrows depict control flow and data flow edges respectively. Question marks on dashed arrows show possible data flow dependences. Check-marks show operations that are safe for parallelization, while cross-marked operations are the ones that hinder parallelisms. Both of these operation categories are identified by the simple JIT-time data flow analysis with DU chains. Exclamation marks are instances of possible cross iteration dependences that could not be figured out at JIT-time. Final safety decisions on these are deferred to further runtime analysis and speculation (Section 4).

are also handled by privatization (`z.field1`). Furthermore, this analysis successfully finds instances of parallelism-inhibiting linked list or tree operations in the loop (e.g. variable `t`) and avoids loop selection.

Scalar arrays: Many cross-iteration dependences in scalar arrays can also be identified using data flow analysis. Array `d` in Figure 4(c) is an example of these cases. However, the data flow analysis can not determine the self-dependence on operation 1 and also the dependence between operations 4 and 1. In order to make parallelization decisions in these cases, we introduce an initial dependence assessment along with a set of range-based runtime checks. During the initial assessment, a copy of the loop code is made and all array accesses are instrumented in this copy to record accessed element indexes during each iteration in a dependence log. After a few iterations, these logs are checked to detect any immediate cross iteration array dependences. If no dependences are found, the system optimistically assumes that the loop is DOALL and selects it for parallelization. However, since this decision is based on only a few iterations, it might be wrong. ParaScript uses a range-based checking mechanism to detect possible conflicts at runtime. More details about this mechanism is discussed in Section 4.2.

Object arrays: Dynamic analysis for arrays of objects is more challenging. Similar to scalar arrays, data flow analysis can determine dependences caused by the array as a whole (assuming all elements point to the same location). For instance, array `d` in the example of Figure 4(d) causes a cross iteration dependence which can be identified by data flow analysis. Assuming that object `z` is loop-invariant, data flow analysis can also find the cross-iteration dependence caused through the assignment of `z` to elements of array `f`.

In addition to the data flow edges in Figure 4(d), any two array pairs in the loop can be dependent on each other through referencing to the same object during execution. Therefore, if there are writes to array elements inside the loop and the data flow analysis does not detect dependences, ParaScript optimistically assumes that the loop is parallelizable and employs a novel runtime mechanism to ensure correctness. During the checkpointing stage, before parallel loop execution, this runtime technique which is based on object reference counting is applied to object arrays to avoid indirect writes to the same object through individual array element writes in different parallel threads. This mechanism which is explained in more detail in Section 4.1, is effective in dynamic resolution of dependences that might be caused by arrays `a`, `c` and `e` in Figure 4(d).

As we explained in this section, ParaScript is able to make optimistic parallelization decisions using only a single data flow analysis pass, while deferring complete correctness checks to the runtime speculation engine.

3.2 Parallel Code Generation

After the loop is chosen to be parallelized, the parallel loop's bytecode is generated according to the template in Figure 5. The chunk size determines the number of iterations after which speculative loop segments are checked for any possible conflicts and its value is determined based on the number of threads, total number of iterations and iteration size. A barrier is inserted at the end of each chunk and is released when all threads are done with their chunk execution after which the conflict checking routine(Section 4.2) is invoked. In order to minimize the checkpointing overhead, a single checkpoint (Section 4.4) is taken at the beginning of the parallelized loop, and if any of the conflict checks fail, the loop is executed sequentially from the beginning.

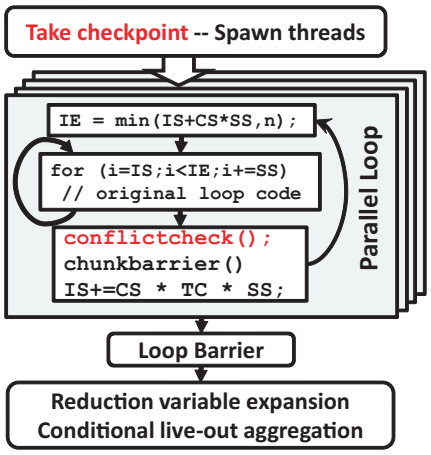


Figure 5: Dynamic code generation template (CS: chunk size, IS: iteration start, IE: iteration end, SS: step size, TC: thread count).

4 Ultra-lightweight Speculation Support

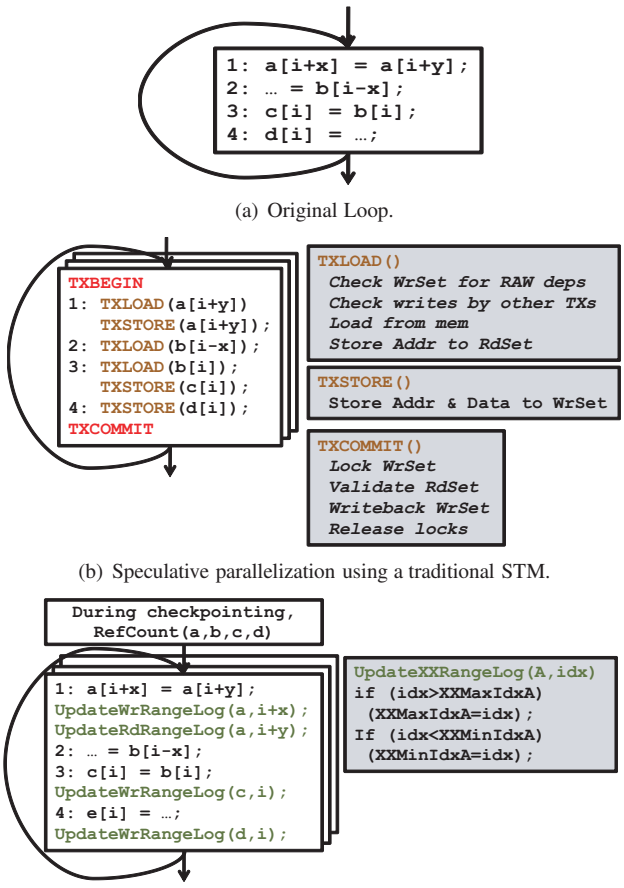
In this section, we introduce our approach for misspeculation detection, checkpointing and recovery. Furthermore, we describe two optimizations to further lower the checkpointing overhead. Figure 6(a) shows an example target loop for parallelization. Using a traditional speculation mechanism such as an STM for speculative parallelization (Figure 6(b)) requires the code generator to instrument all memory accesses with the corresponding speculative versions – `TXLOAD` and `TXSTORE` – and enclose loop chunks with `TXBEGIN` and `TXCOMMIT`. Replacing all memory accesses with transactional versions causes the runtime to spend substantial time tracking each individual memory access. Furthermore, at commit time, considerable overhead is incurred for locking target memory locations and performing consistency checks with reads. These overheads are the main reasons that STM systems usually incur up to 5x slow down over sequential versions of applications [12].

However, in ParaScript, the data flow analysis done at JIT time, obviates the need for speculation on scalar values and individual object variables. Furthermore, a reference counting based analysis is performed during checkpointing on the object arrays, while a minimal set of operations consisting mostly of array index comparisons is added to each array access to enable range-based dependence checking. These techniques are detailed in the following subsections.

4.1 Conflict Detection Using Reference Counting

As mentioned in Section 3.1, manipulating object arrays inside the loop potentially creates dependence cases that are not resolvable using simple data-flow analysis. For example, if the same object reference is assigned to two different array elements, since those elements might be written to during different iterations of the loop, the loop has potential cross iteration dependences, and can not be parallelized.

In order to identify these cases, we employ a technique



(a) Original Loop.

(b) Speculative parallelization using a traditional STM.

(c) Speculative parallelization using ParaScript.

Figure 6: Extra steps needed at runtime for speculation support for loop parallelization in a traditional STM (TL2 [14]) and in ParaScript.

similar to what reference counting garbage collectors use. At runtime, a header is added to all objects involved with loop arrays. This header, which is only used during array checkpointing, includes a reference count and entries for storing array IDs (e.g. starting address). The checkpointing mechanism already goes through all live-in array elements and stores them in the state checkpoint (Section 4.4). During this process which happens before starting parallel loop execution, if an array element refers to an object, the object's reference count and array ID entries are queried. If it had been referenced by the same array before, the system will know that there are two elements in the same array that refer to the same object, and thereby, the loop is disqualified from the parallelization process.

If the object has been referred to by another array and any elements of the two arrays are dependent through def-use chains, the loop is disqualified as well (this condition might disqualify loops that are actually parallelizable, but in order to eliminate the need for more in-depth and expensive speculation, this pessimistic assumption is made to ensure correct execution in all cases).

Function	Description	new Min	new Max
`pop()`	Remove last array element	`oldMin`	`oldArray.length()`
`push()`	Add elements to end of array	`oldMin`	`oldArray.length() + 1`
`reverse()`	Reverse array elements	`0`	`oldArray.length()`
`shift()`	Remove first array element	`0`	`oldArray.length()`
`sort()`	Sorts array elements	`0`	`oldArray.length()`
`splice(i, n, element1,...,elementN)`	Removes n and/or add N elements from position i	`min(oldMin,i)`	`max(oldMax, i+N, i+n)`
`unshift(element1,...,elementN)`	Add N elements to beginning of array	`0`	`oldArray.length + N`

Table 1: JavaScript Array manipulation functions' effect on minimum and maximum access index values. `oldArray.length()` is the length of the array before applying the function. Likewise, `oldMin` and `oldMax` are old values of minimum and maximum read and write access index values.

If, according to the header, the object has no array element references so far, the reference count is incremented and the array's starting address is added to array ID list in the object header. If all object arrays in the loop pass this reference counting phase, parallel loop execution is started and range-based checks as described next are applied to the arrays.

4.2 Conflict Detection Using Range-based Checks

A mechanism is needed to detect cross-iteration conflicts for scalar arrays and for object arrays that have passed the reference counting checks. In order to avoid instrumenting every array access with speculative operations, a range-based dependence checking is employed. In this method, during the speculative loop execution, each array maintains four local variables called `rdMinIndex`, `rdMaxIndex`, `wrMinIndex`, and `wrMaxIndex`. These values keep track of the minimum and maximum read and write indexes for the array. During parallel code generation, at each array read or write access, necessary comparisons are inserted in the bytecode to update these values if needed. After several iterations of the loop, accessed array references and their minimum and maximum access index values are written to the array write and read sets (`arrayWrSet` and `arrayRdSet`) in the memory. Each thread of execution maintains its own array read and write sets. The number of iterations after which write and read sets are updated is determined based on a runtime heuristic depending on the number of arrays in the loop, number of array accesses and the difference between minimum and maximum accessed index values. As was shown in Figure 5, after a predefined number of iterations, all threads stop at a barrier and the conflict checking routine checks the read and write access ranges of each array in each thread against write access ranges of the same array in all other threads. If any conflict is found, the state is rolled back to the previous checkpoint and the loop is run sequentially.

One downside of this mechanism for array conflict checking is that strided accesses to arrays are always detected as conflicts. However, this is a minor cost compared to the large overhead reduction as a result of efficient range-based conflict detection in other cases.

4.3 Instrumenting Array Functions

As mentioned before, all array read and write accesses have to be instrumented for updating minimum and maximum access index values. In addition to normal array access using indexes, JavaScript provides a wide range of array manipulation functions which are implemented inside the JavaScript engine and read or write various points in arrays. In order to correctly capture their effect, their uses in the JavaScript application are also instrumented. Table 1 shows these functions and their effect on min/max access index values.

In JavaScript, it is possible to add custom functions to built-in object types such as `Array`, `String` or `Object` using the `prototype` keyword inside the program source code. These custom functions are able to change values or properties of their respective object. This could pose complications in determining the values of minimum and maximum access index values. Therefore, any additional properties of functions added by the `prototype` keyword are detected and instrumented to correctly update access index values.

4.4 Checkpointing and Recovery

At any point during runtime, the *global object* contains references to all application objects in the global namespace. Inside the JavaScript source code, the global object can always be accessed using the `this` keyword in the global scope and is equivalent to the `window` object in web pages. Pointers to local objects, variables and function arguments reside in the corresponding function's *call object*, and can be accessed from within the engine.

Checkpointing: When a checkpoint is needed in the global scope, only a global checkpoint is taken, whereas in case the checkpoint request is issued inside a function, both global and local checkpoints need to be taken. The stack and frame pointers are also checkpointed in the latter case. While going through variable references in the global or local namespace, they are cloned and stored in the heap. Since there will be no references to these checkpoints from within the JavaScript application, the garbage collector needs to be asked explicitly not to touch them until the next checkpoint is taken. During checkpointing, to ensure proper roll-back, ob-

jects are deep-copied. However, based on a runtime threshold, ParaScript stops the deep-copying process and avoids parallelizing the loop if checkpointing becomes too expensive. Although a function's source code can be changed at runtime in JavaScript, since ParaScript avoids parallelizing loops containing code injection constructs, the checkpointing mechanism does not need to clone the source code. All other function properties are cloned.

Recovery: In case of a rollback request, the checkpoint is used to revert back to the original state and the execution starts from the original checkpointing location. Similar to checkpointing, recovery is also done at two levels of global and local namespaces. Due to complications of handling different stack frames at runtime, cross-function speculation is not supported. This means that speculative code segments should start and end in the global scope or the same function scope (obviously, different functions could be called and returned in between). During local recovery, stack and frame pointers are over-written by the checkpointed values. This makes sure that when an exception happens in the speculative segment and the exception handler is called from within some child functions, if a rollback is triggered, the stack and frame pointers have the correct values after recovery.

4.5 Checkpointing Optimizations

Using the speculation mechanism inside the ParaScript framework provides several optimization opportunities to further reduce the checkpointing overhead.

Selective variable cloning: The original checkpointing mechanism takes checkpoints of the whole global or local state at any given time. However, in ParaScript, only a checkpoint of variables written during speculative code segment execution is needed. In the selective variable cloning optimization, these written variables are identified using the data flow information. If any variable is passed as an argument to a function, the function's data flow information is used to track down the variable accesses and determine if there are any writes to that variable. This information is passed to the checkpointing mechanism, and a selective rather than a full variable cloning is performed.

Array clone elimination: In some applications, there are large arrays holding return values from calls to functions implemented inside the browser. One example of these functions is `getImageData` from the `canvas` HTML5 element implementation inside the browser. This function returns some information about the image inside the `canvas` element. Since speculative code segments in ParaScript do not change anything inside DOM, the original image remains intact during the speculative execution. Therefore, instead of cloning the array containing the image data information, ParaScript calls the original function again with the same input at rollback time (e.g. `getImageData` is called again on the same image).

5 Evaluation

5.1 Experimental Setup

ParaScript is implemented in Mozilla Firefox's JavaScript engine, SpiderMonkey [4] distributed with Firefox 3.7a1pre. All experiments are done with the tracing optimization [15] enabled. The proposed techniques in the paper are evaluated on the SunSpider [5] benchmark suite and a set of filters from the Pixastic JavaScript Image processing Library [3].

SunSpider has 26 JavaScript programs. All these were run through the ParaScript engine and 11 were found to benefit from parallelization. Lack of parallelization opportunities in the rest of them, however, was found early on in the dependence assessment stage (Section 3) without any noticeable performance overhead (an average of 2% slow down due to the initial analysis across the 15 excluded benchmarks). From these 11 parallelizable benchmarks, three `bitops` benchmarks do not perform useful tasks and are only in the suite to benchmark bitwise operations performance. Therefore, they are also excluded from the final results (These 3 benchmarks gained an average of 2.93x speedup on 8 threads after running through ParaScript).

The Pixastic library has 28 filters and effects, out of which the 11 most compute-intensive filters were selected. The rest of the filters perform simple tasks such as inverting colors, so their loops did not even pass the initial loop selection heuristic based on the loop size. Out of the 11 selected filters, 3 benchmarks (namely `blur`, `mosaic` and `pointilize`) were detected by ParaScript to be not parallelizable, due to cross iteration array dependence between all iterations in `blur`, and DOM accesses inside loops in `mosaic` and `pointilize`. All benchmarks were run 10 times, and the average execution time is reported. The target platform is an 8-processor system with 2 Intel Xeon Quad-core processors, running Ubuntu 9.10.

5.2 Parallelism Potential and Speculation Cost

Figure 7 shows the fraction of sequential execution time selected by ParaScript for parallelization in our subset of the SunSpider and Pixastic suites. These ratios show the upper bound on parallelization benefits. In the image processing benchmarks, a high fraction of sequential execution outside parallelized loops is spent in the `getImageData` function implementation inside the browser. This function takes the image inside an HTML `canvas` element as the input and returns a 2D array containing the RGB and alpha value information for all pixels in the image. The function's implementation inside the Firefox browser mainly consists of a DOALL loop that walks over all pixels in the image. We exploited this parallelization opportunity and hand-parallelized this DOALL loop inside the browser. The effect of this parallelization is discussed in the next subsection.

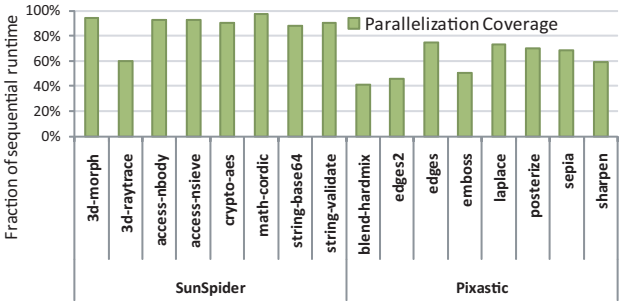


Figure 7: Parallelization coverage as ratio of parallelizable code runtime to total sequential program execution.

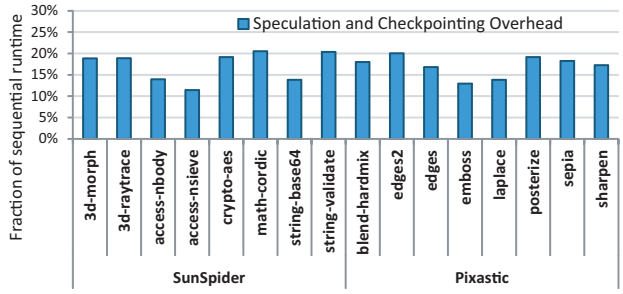


Figure 8: Speculation and checkpointing overhead as a fraction of total sequential program execution.

Checkpointing and speculation overheads are presented in Figure 8. This study has been done on single-threaded versions of the applications. The experiments are set up such that the sequential program is passed through the ParaScript system and parallelization and speculation constructs required for proper execution of the parallelized version are inserted into the sequential benchmark. The benchmark is then executed with one thread. The overhead is measured after applying both optimizations in Section 4.5. Array clone elimination proved to be quite effective in Pixastic benchmarks, due to cloning elimination in the return array of the `getImageData` function. The only overhead missing in these bars is the checking overhead at the end of each parallel chunk. Due to the efficient storage of accessed array ranges, this checking overhead turned out to be negligible in the experiments. On average, checkpointing and speculation overhead is around 17% of the sequential execution runtime of the whole application. The main reasons for this low overhead are accurate dependence prediction, light-weight array access instrumentation, efficient conflict checking mechanism due to the use of array access bound checks instead of individual element checks, and the proposed optimizations on checkpointing. Furthermore, due to the high prediction success rate of the parallelization assessment step, ParaScript is able to take checkpoints only once at the beginning of the loop, which in turn lowers the overhead.

5.3 Results

Figure 9 shows performance improvements as a result of speculative parallelization using the ParaScript framework, across subsets of SunSpider suite and the Pixastic image processing applications. The Y-axis shows the speedup versus sequential. These performance numbers include all overheads from the JIT compilation time analysis, runtime analysis and runtime speculation. The black line on top of each bar presents the effect of replacing our light-weight speculation mechanism with hardware speculation support such as a hardware transactional memory system. Speculation costs of such a hardware system is assumed to be zero.

Overall, across the subset of SunSpider benchmarks, the experiments show and average of 1.51x, 2.14x and 2.55x improvement over sequential for 2, 4 and 8 threads of execution (Figure 9(a)) [1]. In case of using hardware speculation support, the speedups would have increased to an average of 1.66x, 2.27x and 2.72x over sequential for 2, 4 and 8 threads.

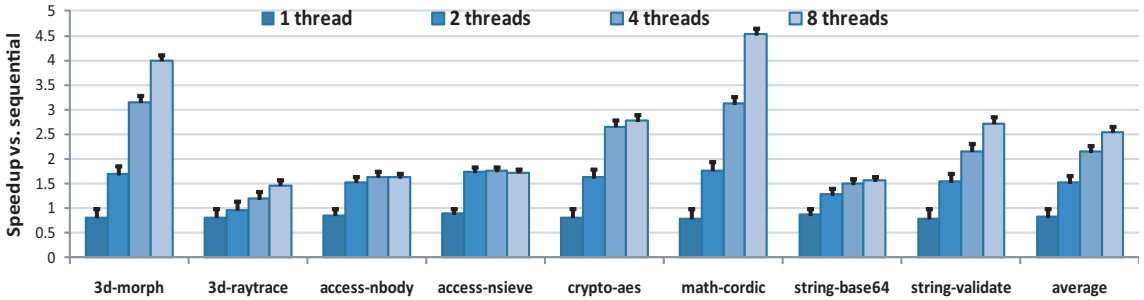
Figure 9(b) presents the results for Pixastic applications. The lower part of each bar shows the performance improvement as a result of speculatively parallelizing the application itself. The top part of each bar is the extra performance improvement gained by parallelizing the `getImageData` function inside the browser with the same number of threads as the parallelized application. Finally, the black line on top of each bar shows the performance improvement, assuming hardware speculation support. On average, speedups of 1.29x, 1.43x, and 1.55x are gained after parallelization for 2, 4, and 8 threads of execution. Additional speedups of 14%, 23%, and 27% are gained after parallelizing the Firefox implementation of the `getImageData` function which increases the performance improvement to an average of 1.43x, 1.66x, and 1.82x respectively. Using hardware speculation support, speedups go upto 1.58x, 1.79x and 1.92x for 2, 4 and 8 threads.

5.4 Discussion

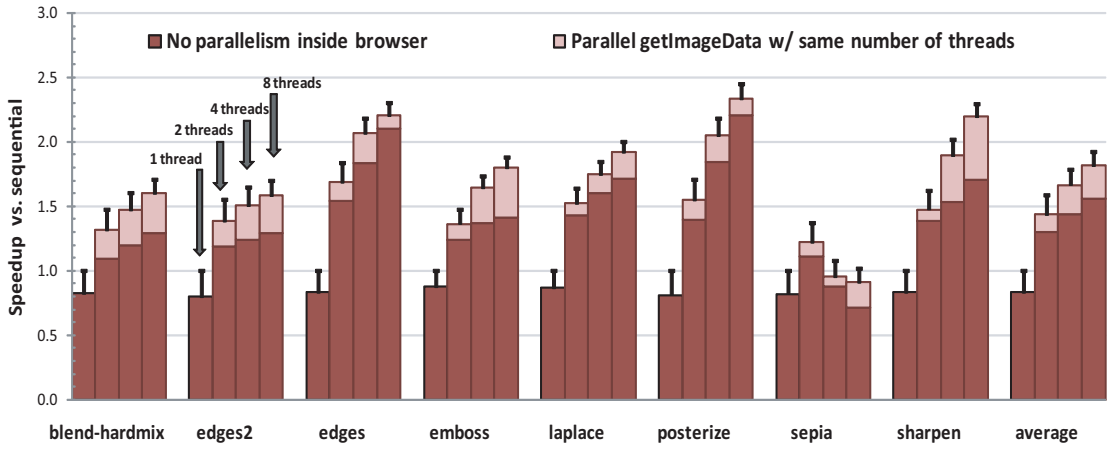
In SunSpider benchmarks (Figure 9(a)), `access-nbody`, `access-nsieve` and `string-base64` do not show much scaling past 2 threads. The reason is that the parallelized loops in these benchmarks are unbalanced and the final runtimes are always bounded by the longest running iteration, irrespective of the number of threads used for parallelization. Comparatively low overall speedup in `raytrace` (1.65x for 8 threads) is due to the low coverage (less than 60%) of parallelizable loops.

The loop selection mechanism discovered many loops with cross-iteration dependences in these programs and avoided parallelizing them, so the abort rate was zero in almost all the benchmarks. The only case of abort was from an inner loop in 3d-raytrace, which contributed to less than 5%

[1] Average speedup for all 26 SunSpider benchmarks is 1.69x for 8 threads.



(a) SunSpider speedup compared to the sequential execution.



(b) Pixastic speedup compared to the sequential execution. The top part of each graph shows the extra performance gain obtained by parallelizing the `getImageData` implementation inside Firefox.

Figure 9: ParaScript performance improvement over sequential for 2, 4 and 8 threads. Black lines on top of the graphs show the performance of a system with hardware support for speculation (e.g. HTM).

of the total execution time.

In Pixastic applications, other than the inherent parallelization limitation due to relatively low coverage of parallelizable sections (an average of 60% coverage), the large ratio of memory to computation operations in these benchmarks is a limiting factor to their parallelism potential. This causes the program to become limited by the memory system latency rather than the amount of computation.

The negative effect of this factor has the most impact on the `sepia` benchmark where it causes slowdown rather than speedup for 4 and 8 threads. Each iteration in the inner loop of this benchmark has 3 multiply operations, 7 adds, and 6 memory operations. This causes the program to be quite memory intensive and therefore be bound by the performance of the memory system. Although the computations and memory requests are parallelized in the 4 and 8 threaded versions, the program is sequentialized at the memory system bus, waiting for its memory requests to be serviced and returned. Based on this observation, the loop selection heuristic was updated to account for the memory to computation operation ratio when making parallelization decision to avoid such slowdowns.

The same effect limits the performance improvements from parallelizing the `getImageData` function inside the browser. Although, this function can be parallelized without any speculation, the innermost loop has 8 memory operations versus 8 add, 3 multiply and 3 division operations. Therefore, becoming sequential at the memory system bus has negative impacts on the parallelization gains of this function as well.

6 Related Work

There is a long history of work in the area of parallelizing sequential C/C++ applications. Hydra [17] and Stampede [29] are two of the first efforts in the area of general purpose program parallelization. The authors in [35] proposed compiler transformations to extract more loop level parallelism from sequential programs. Speculative decoupled software pipelining [31] is another approach that focuses on extracting parallelism from loops with cross iteration dependences. All these works use static analysis and profiling for parallelization and assume hardware speculation support, while in this work, we do not rely on any static analysis and we perform the speculation completely in software.

There have been previous proposals for dynamic parallelization techniques. The JPRM system [13] focuses on dynamic parallelization of Java applications, but it is dependent on complex and costly hardware support for online memory profiling that does not exist in commodity processor systems. The LRPD test [26], performs speculative dynamic parallelization of Fortran applications. This framework is dependent on static analysis and transformation of loops, and runtime checkpointing and speculative execution. In this work, no static analysis is done on the code and the code is generated on the fly. Furthermore, LRPD's array dependence testing is based on tracking individual array accesses, while ParaScript only tracks array bounds which significantly reduces tracking and checking overheads. The work in [10] employs range tests based on static symbolic analysis, whereas ParaScript performs these tests dynamically and at a lower cost at runtime. The Ninja project [9] uses array reference comparison for linear arrays, which does not cover arrays of objects or arrays of arrays. Their tests on higher order arrays involves introducing a custom array package and changing the source code.

Authors in [34] and [32] investigate dynamic parallelization of binary executables. They use binary rewriting to transform sequential to parallel code. However, [32] while exploiting slice-based parallelization, assumes hardware speculation and hardware parallel slice support which do not exist in current systems. The work in [34] only performs control speculation and does not speculate on data. Therefore, after identifying the parallel loop, they have to perform data dependence analysis to prove lack of cross iteration memory dependence, which would be quite costly. Furthermore, none of these dynamic parallelization techniques work on dynamic languages such as JavaScript.

There is a significant amount of previous efforts in the area of memory speculation. Harris *et al.* go through a detailed survey of different transactional memory techniques in [18]. In particular, Shavit *et al.* proposed the first implementation of software transactional memory in [28]. The authors in [19, 8] proposed a lock-based approach where write locks are acquired when an address is written. Also, they maintain a read set which needs to be validated before commit. Our speculation mechanism is not a full-featured software transactional memory. ParaScript introduces a customized speculation system which is tailored towards efficient speculative loop level parallelism in JavaScript. By excluding many features required by general-purpose TM models, speculation in ParaScript has become highly lean and efficient.

Due to the lack of concurrency support in the JavaScript language, there have not been many previous efforts for exploiting multicore systems to improve the execution performance of client-side web applications. There has been a recent proposal to add a limited form of concurrency called *web workers* to the language. There are already standardization efforts being undertaken on this proposal [7] and almost all major browsers are starting to support these constructs. However, due to separate memory spaces among the workers, the only means of communication between them is through message passing, which makes developing multithreaded applications very difficult. Crom [22], is a recent effort in employing speculative execution to accelerate web browsing. Crom runs speculative versions of event handlers based on user behavior speculation in a shadow context of the browser, and if the user generates a speculated-upon event, the pre-computed result and the shadow context are committed to the main browser context. Crom exploits a different parallelization layer and is orthogonal to our work. One could use both approaches at the same time to enjoy parallelization benefits at multiple levels.

7 Conclusion

JavaScript is the dominant language in the client-side web application domain due to its flexibility, ease of prototyping, and portability. Furthermore, as larger and more applications are deployed on the web, more computation is moved to the client side to reduce network traffic and provide the user with a more responsive browsing experience. However, JavaScript engines fall short of providing required performance when it comes to large and compute-intensive applications. At the same time, multicore systems are the standard computing platform in the laptop and desktop markets and are making their way into cell phones. Therefore, in addition to the efforts underway by browser developers to improve engines' performance, parallel execution of JavaScript applications is inevitable to sustain required performance in the web application market. In this work, we proposed ParaScript, a fully dynamic parallelizing engine along with a highly customized software speculation engine to automatically exploit speculative loop level parallelism in client-side web applications. We show that our technique can efficiently identify and exploit implicit parallelism in JavaScript applications. The prototype parallelization system achieves an average of 2.55x speedup on a subset of the SunSpider benchmark suite and 1.82x speedup a set of image processing filters, using 8 threads on a commodity multi-core system.

Acknowledgements

We would like to thank Dr. Tim Harris for insightful discussions and feedback on this work. We extend our thanks to anonymous reviewers for their valuable comments. We also thank Ganesh Dasika, Shuguang Feng, Shantanu Gupta and Amir Hormati for providing feedback on this work. This research was supported by the National Science Foundation grant CNS-0964478 and the Gigascale Systems Research Center, one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program.

References

[1] gameQuery - a javascript game engine with jQuery - http://gamequery.onaluf.org/.

[2] Google V8 JavaScript Engine - http://code.google.com/p/v8.

[3] Pixastic: JavaScript Image Processing - http://www.pixastic.com.

[4] Spidermonkey engine - http://www.mozilla.org/js/spidermonkey/.

[5] Sunspider javascript benchmark - http://www2.webkit.org/perf/sunspider-0.9/sunspider.html.

[6] The Render Engine - http://www.renderengine.com/.

[7] Web hypertext application technology working group specifications for web workers - http://whatwg.org/ww.

[8] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *Proc. of the '06 Conference on Programming Language Design and Implementation*, pages 26–37, 2006.

[9] P. V. Artigas, M. Gupta, S. Midkiff, and J. Moreira. Automatic loop transformations and parallelization for Java. In *Proc. of the 2000 International Conference on Supercomputing*, pages 1–10, 2000.

[10] W. Blume and R. Eigenmann. The range test: a dependence test for symbolic, non-linear expressions. In *Proc. of the 1994 International Conference on Supercomputing*, pages 528–537, 1994.

[11] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August. Revisiting the sequential programming model for multicore. In *Proc. of the 40th Annual International Symposium on Microarchitecture*, pages 69–81, Dec. 2007.

[12] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software Transactional Memory: Why Is It Only a Research Toy? *Queue*, 6(5):46–58, 2008.

[13] M. K. Chen and K. Olukotun. The Jrpm system for dynamically parallelizing Java programs. In *Proc. of the 30th Annual International Symposium on Computer Architecture*, pages 434–446, 2003.

[14] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proc. of the 2006 International Symposium on Distributed Computing*, 2006.

[15] A. Gal et al. Trace-based just-in-time type specialization for dynamic languages. In *Proc. of the '09 Conference on Programming Language Design and Implementation*, pages 465–478, 2009.

[16] S. Guarnieri and B. Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code. In *Proceedings of the USENIX Security Symposium*, pages 151–163, Aug. 2009.

[17] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 58–69, Oct. 1998.

[18] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, 2nd Edition*. Morgan & Claypool Publishers, 2010.

[19] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. *Proc. of the '06 Conference on Programming Language Design and Implementation*, 41(6):14–25, 2006.

[20] T. Mason. LAMPVIEW: A Loop-Aware Toolset for Facilitating Parallelization. Master's thesis, Dept. of Electrical Engineeringi, Princeton University, Aug. 2009.

[21] M. Mehrara, J. Hao, P. chun Hsu, and S. Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proc. of the '09 Conference on Programming Language Design and Implementation*, pages 166–176, June 2009.

[22] J. Mickens, J. Elson, J. Howell, and J. Lorch. Crom: Faster Web Browsing Using Speculative Execution. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation*, pages 127–142, Apr. 2010.

[23] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 35(8):102–114, Aug. 1992.

[24] A. Raman, H. Kim, T. R. Mason, T. Jablin, and D. August. Speculative parallelization using software multi-threaded transactions. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 65–76, 2010.

[25] P. Ratanaworabhan, B. Livshits, D. Simmons, and B. Zorn. Jsmeter: Characterizing real-world behavior of javascript programs. Technical Report MSR-TR-2009-173, Microsoft Research, Dec. 2009.

[26] L. Rauchwerger and D. A. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):160, 1999.

[27] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An Analysis of the Dynamic Behavior of JavaScript Programs. In *Proc. of the '10 Conference on Programming Language Design and Implementation*, pages 1–12, 2010.

[28] N. Shavit and D. Touitou. Software transactional memory. *Journal of Parallel and Distributed Computing*, 10(2):99–116, Feb. 1997.

[29] J. G. Steffan and T. C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proc. of the 4th International Symposium on High-Performance Computer Architecture*, pages 2–13, 1998.

[30] C. Tian, M. Feng, and R. Gupta. Supporting speculative parallelization in the presence of dynamic data structures. In *Proc. of the '10 Conference on Programming Language Design and Implementation*, pages 62–73, 2010.

[31] N. Vachharajani, R. Rangan, E. Raman, M. Bridges, G. Ottoni, and D. August. Speculative Decoupled Software Pipelining. In *Proc. of the 16th International Conference on Parallel Architectures and Compilation Techniques*, pages 49–59, Sept. 2007.

[32] C. Wang, Y. Wu, E. Borin, S. Hu, W. Liu, D. Sager, T. fook Ngai, and J. Fang. Dynamic parallelization of single-threaded binary programs using speculative slicing. In *Proc. of the 2009 International Conference on Supercomputing*, pages 158–168, 2009.

[33] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[34] E. Yardimci and M. Franz. Dynamic parallelization and mapping of binary executables on hierarchical platforms. In *2006 Symposium on Computing Frontiers*, pages 127–138, 2006.

[35] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *Proc. of the 14th International Symposium on High-Performance Computer Architecture*, pages 290–301, Feb. 2008.