JavaScript Parallelizing Compiler for Exploiting Parallelism from Data-Parallel HTML5 Applications

YEOUL NA and SEON WOOK KIM, Korea University
YOUNGSUN HAN, Kyungil University

With the advent of the HTML5 standard, JavaScript is increasingly processing computationally intensive, data-parallel workloads. Thus, the enhancement of JavaScript performance has been emphasized because the performance gap between JavaScript and native applications is still substantial. Despite this urgency, conventional JavaScript compilers do not exploit much of parallelism even from data-parallel JavaScript applications, despite contemporary mobile devices being equipped with expensive parallel hardware platforms, such as multicore processors and GPGPUs.

In this article, we propose an automatically parallelizing JavaScript compiler that targets emerging, data-parallel HTML5 applications by leveraging the mature affine loop analysis of conventional static compilers. We identify that the most critical issues when parallelizing JavaScript with a conventional static analysis are ensuring correct parallelization, minimizing compilation overhead, and conducting low-cost recovery when there is a speculation failure during parallel execution. We propose a mechanism for safely handling the failure at a low cost, based on compiler techniques and the property of idempotence. Our experiment shows that the proposed JavaScript parallelizing compiler detects most affine parallel loops. Also, we achieved a maximum speedup of 3.22 times on a quad-core system, while incurring negligible compilation and recovery overheads with various sets of data-parallel HTML5 applications.

CCS Concepts: • **Theory of computation** → *Parallel computing models*; • **Software and its engineering** → *Just-in-time compilers*; *Scripting languages;*

Additional Key Words and Phrases: JavaScript, JIT, javascriptcore, loop parallelization, javascript engines, javascript compilers, HTML5

ACM Reference Format:
Yeoul Na, Seon Wook Kim, and Youngsun Han. 2015. JavaScript parallelizing compiler for exploiting parallelism from data-parallel HTML5 applications. ACM Trans. Archit. Code Optim. 12, 4, Article 64 (December 2015), 25 pages.
DOI: http://dx.doi.org/10.1145/2846098

1. INTRODUCTION

JavaScript [Flanagan 2006] is a dynamic script language. Since being developed in Netscape with the name Mocha, JavaScript has been mainly used for dynamic web page behaviors, particularly interactions with specific events created by users or system environments [Richards et al. 2010a; Ratanaworabhan et al. 2010].

This work is supported by the Industrial Strategic Technology Development Program (10052653, Development of processing in memory architecture and parallel processing for data bounding applications) funded by the Ministry of Trade, Industry & Energy (MI, Korea).
Authors' addresses: Y. Na and S. W. Kim, 513 International Center for Converging Technology, Korea University, 02841, Korea; emails: {rapidsna, seon}@korea.ac.kr; Y. Han (corresponding author), 202 The 2nd Engineering Building, Kyungil University, 38428, Korea; email: youngsun@kiu.ac.kr.

DOI: http://dx.doi.org/10.1145/2846098

Thus, JavaScript has been mostly utilized as a lightweight scripting language, and so exploiting its parallelism has been considered of less importance than boosting its single-thread performance. However, this tendency has been changing since the HTML5 [Hickson and Hyatt 2011] standard was announced. HTML5, the latest version of the HTML standard, transforms the web browsers into general-purpose application platforms by introducing its new features as JavaScript APIs [Anttonen et al. 2011]. In conjunction with WebGL [Marrin 2011] as well as HTML5, JavaScript has been quickly adopted to new computing areas such as 2D and 3D graphics, geolocation, and interactive media content, which have been areas of traditional native platforms. These new capabilities enable large-scale web applications, including gaming engines, physics simulations, and in-browser video processing, which potentially have plentiful parallelism [Herhut et al. 2013]. In other words, JavaScript is increasingly undertaking computationally intensive, data-parallel workloads that manipulate a large dataset. However, many applications utilizing those new capabilities are still infeasible with JavaScript because of performance issues caused by its thorough single-threaded environment. Although single-threaded performance of JavaScript has been notably improved after the "browser wars," JavaScript performance still lags with computationally intensive, data-parallel workloads, which have great potential to benefit from parallel execution. Therefore, providing parallel execution of JavaScript applications has now become an urgent issue.

Modern mobile computing environments incorporate plentiful parallel hardware platforms such as multicore processors and GPGPUs [Qualcomm Inc. 2015]. Thus, many approaches have been developed to exploit the parallelism made available by existing native and dynamic language applications, significantly improving their performance on parallel hardware platforms [Campanoni et al. 2014; Beletska et al. 2011; Bikshandi et al. 2006; Campanoni et al. 2012]. However, it is hard for JavaScript to take advantage of currently abundant parallel hardware resources because of its dynamically typed language feature, which allows for many irregularities and complicates the determination of the underlying parallelism of JavaScript applications by compilers.

There have been a few research projects with the goal of exploiting parallelism in JavaScript applications. Mehrara et al. [2011] proposed a dynamically parallelizing JavaScript engine based on thread-level speculation (TLS). Their JavaScript engine successfully exploits loop-level parallelism and improves JavaScript performance on a multicore system. However, the potential disadvantage of this approach is that runtime dependency checks, checkpoints, and recovery overheads necessary to support speculative execution would be quite substantial for emerging HTML5 applications, which tend to manipulate huge amounts of data. *Notably, there is great potential to minimize such speculation overheads by nonspeculatively parallelizing loops that can be resolved by compiler analysis* [Ooi et al. 2001]. Therefore, we can adopt the dependence analyses of conventional static compilers to parallelize JavaScript applications. With this static-compiler-based approach, we can eliminate runtime dependency checks and minimize unnecessary speculation failures. Also, the costly checkpoint-rollback overhead is substituted by our lightweight recovery mechanism leveraging the property of idempotence [Kim et al. 2006]. Even though the previous JavaScript parallelization scheme adopts a speculation-based approach to avoid compilation overhead at runtime [Mehrara et al. 2011], our idea is that compilation overhead would become negligible if an application is long-running or recurring enough that its execution time compensates for the extra compilation overhead, which is a general principle of JIT compilation techniques.

In this article, we propose a new JavaScript parallelizing compiler that is able to automatically detect DOALL parallel loops, generate multithreaded JIT code, and execute that code on multicore CPUs by leveraging LLVM-based static loop analysis

and parallel code generation. *To the best of our knowledge, this is the first approach that parallelizes the execution of JavaScript functions by leveraging static compiler analyses, and generates the parallelized low-level intermediate representation (IR) of JavaScript compilation.* Our environment is based on JavaScriptCore's fourth-tier LLVM JIT (FTL JIT) [Pizlo 2014] that generates LLVM IR code and utilizes loop analyses and parallel code generation from an LLVM-based static parallelizer [Kotha et al. 2013].

Our parallelization targets are realistic and compute-intensive data-parallel HTML5 applications that potentially benefit from parallel execution. Contemporary benchmark suites such as V8 [Richards et al. 2010b], SunSpider [Stachowiak 2007], and Octane [Cazzulani 2012], however, are not representative of real web applications [Ratanaworabhan et al. 2010], and the web pages in the Alexa list [Alexa 2015] do not contain intensive computations. So instead, we evaluated realistic sets of emerging, data-parallel HTML5 applications provided by River Trail [Herhut et al. 2013] and taken from Pixastic [Seidelin 2014], a widely used image processing library [Porcides et al. 2011; Mehrara et al. 2011; Bartsch et al. 2014; Cho et al. 2014]. Our proposed JavaScript parallelizer automatically exploits parallelism in the measured applications and libraries, obtaining a maximum speedup of 3.22 times on a quad-core platform.

The main contributions in this article are as follows:

—We specify the issues that arise when parallelizing JavaScript with conventional static compiler techniques and propose feasible solutions.
—We propose an idea to freely and safely manage a speculation violation in the middle of parallel loop execution by leveraging compiler techniques and the property of idempotence.
—Our proposed system automatically detects parallel loops at the same level of native applications, and the experimental data shows performance improvements for real HTML5 applications.

The rest of this article is organized as follows. Section 2 discusses challenges when parallelizing JavaScript with the static compilation techniques. Section 3 proposes the JavaScript parallelizing compiler that resolves the aforementioned issues. Section 4 describes our safe and lightweight speculation recovery mechanism. Section 5 presents a case study implementing our proposal on JavaScriptCore. Section 6 presents the experimental results, and Section 7 describes related work. Finally, we conclude in Section 8.

2. CHALLENGES IN PARALLELIZING JAVASCRIPT

In this section, we discuss the challenges when parallelizing JavaScript using static compiler analyses.

Safe Parallelization and Execution. JavaScript is a sequential programming language. Thus, a JavaScript engine manages its execution context in a serial manner so the careless parallel execution of JavaScript may violate program semantics. This difficulty has contributed to the lack of support for concurrency in contemporary JavaScript engines [Google Inc. 2012; Mozilla Developer Network 2015; Apple Inc. 2015]. However, the underlying motivation of this research is that thread-level parallelism of any application is achievable if there are no data dependence violations from concurrent execution of multiple threads [Kennedy and Allen 2002], and there is no exception for JavaScript. Our proposed JavaScript parallelizing compiler is carefully designed to avoid the data race condition so as to preserve the sequential consistency of multithreaded JavaScript programs (Section 3.1).

Dependence Analysis. In this article, our parallelization targets are loops that do not carry a dependence across iterations, that is, DOALL loops. In order to prove

whether a sequential loop is the automatically parallelizable DOALL loop, the data dependence relations in the loop should be resolved. As most parallel loops in data-parallel applications manipulate subscripted arrays, the parallelizer should be able to guarantee that all the array element accesses of different iterations are independent of each other [Wolf and Lam 1991; Banerjee 1988]. Our article applies this modern static parallelization technique to a dynamic JavaScript compiler.

However, the inherent nature of JavaScript, a dynamically typed and interpreted language, makes it much more complicated for the compiler to analyze dependencies. First, analyzing data dependencies from IR code generated during a JavaScript JIT compilation process is problematic because JavaScript objects are dynamically managed by the JavaScript engine, and the way the engine manages objects induces pointers to pointers, which hamper memory disambiguation at compile time. To access an object property, the engine loads the address of the object and then reads the address of the property from the loaded object address. This generates pointer-to-pointer accesses, an obstacle that must be overcome to detect even basic parallel loops. Our proposed compiler overcomes this issue based on the fact that each object manipulates its unique property data (Section 3.1).

Also, the dynamically typed nature of JavaScript hinders the array dependence analysis essential for loop parallelization. Like other object types, which operations manipulate array-typed data is determined at runtime. Thus, the JavaScript engine should speculate which memory operations are array accesses and forward the predicted array information to an optimizing compiler. Current JavaScript engines infer data types based on profiled data when speculating which operations access arrays. However, compilers in JavaScript engines do not yet exploit such array information for optimization. Our proposed system hands over the array information to an optimizing compiler so that it can more easily perform an affine array analysis (Section 3.1).

Parallelization Overhead. Although parallelization techniques boost program performance, they entail considerable parallelization overhead. The overhead can be parallelization analysis overhead or speculation recovery overhead depending on parallelization techniques. Since JavaScript is basically an interpreted language, its overall execution time includes the compilation time, so compilation overhead should be avoided. Given this situation, the compilation overhead can be minimized by using a TLS approach. However, TLS approaches embody runtime dependency checks and misspeculation overhead [Mehrara et al. 2009; Raman et al. 2010], which becomes more critical in the case of emerging HTML5 applications that contain intensive computation with large datasets. The reason for this is that TLS is not scalable to a large number of threads because it needs data synchronization and communication for runtime dependency checks at every chunk of iterations. Decreasing the precision of dependency checks may mitigate this issue, but there will be additional recovery overhead due to the increased false positives. Kim et al. [2012] propose a scalable TLS-based solution for a cluster environment; however, their data shows that just a slight increment of misspeculation rate, that is, 0.04%, exacerbates the scalability of TLS performance. Unfortunately, JavaScript execution already includes inherent misspeculations since JavaScript compilers have adopted a speculative JIT technique, which means the existing scalable TLS solution is not suitable for the JavaScript execution environment. In addition, experimental data in Mehrara et al. [2009] shows that without the assistance of static pointer analysis to minimize runtime dependency checks, the performance of TLS alone can be much lower than that of sequential execution due to TLS overhead. Therefore, we adopt the parallel loop analysis of traditional static compilers and accept the increased compilation overhead. This compiler-based approach makes the runtime dependency checks unnecessary. Also, we can eliminate the speculation rollback

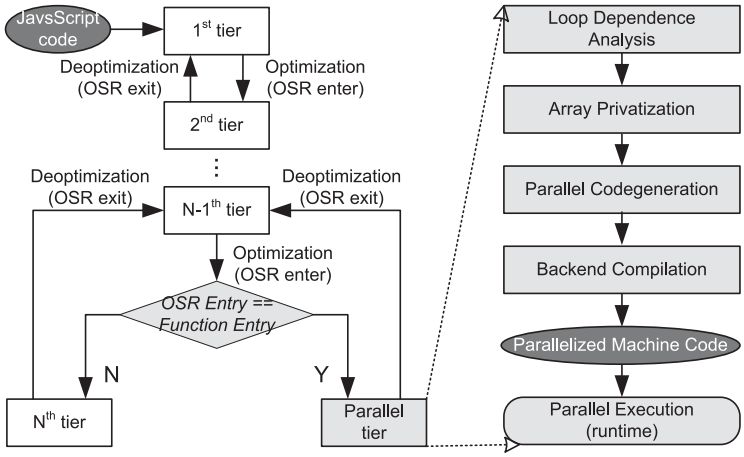


Fig. 1. Overview of our compilation framework.

caused by runtime dependency violations [Ooi et al. 2001]. As mentioned earlier, the compilation overhead can be a problem, but nowadays, JavaScript engines, including our proposed compiler, adopt a multitier strategy and a concurrent JIT compilation technique to hide the compilation overhead (Section 3).

While adopting a compiler-based approach, we still need to support speculation rollbacks during parallel execution due to other existing speculative optimizations of JavaScript compilers. However, a checkpoint-rollback mechanism [Koo and Toueg 1986], which is typically used with speculation recovery, is impractical for emerging HTML5 applications that manipulate large quantities of data. Instead, our proposed compiler leverages the property of idempotence to minimize the recovery overhead (Section 3.3).

3. JAVASCRIPT PARALLELIZING COMPILER

Our proposed system adopts multitier and concurrent JIT compilation techniques to reduce compilation overhead, as shown in Figure 1. To trade off startup delay and optimization quality, JavaScript engines utilize the multitier strategy. For JavaScript code that is executed only a few times, the engines just interpret or compile without optimization to save compilation time and minimize latency, but for frequently executed code, they perform more optimizations to improve code quality. When a hot loop is detected based on an execution counter, the JavaScript engine triggers the next tier during the execution to perform more optimizations, or when an assumption made during optimization is violated, for example, when a data type changes, the JavaScript engine invokes the lower tier to deoptimize the code. Switching between different tiers promptly without losing the temporal data generated by the former tier requires a data relocation mechanism. On-stack replacement (OSR) [Fink and Qian 2003] is the solution for safely transferring data between different tiers in most JavaScript engines. OSR replaces a compiled code and its stack frame by an equivalent code and frame, which may have variables in different locations. The entry of the optimization target code is called OSR entry, and the deoptimization to the lower tier is called OSR exit. Also, the JavaScript engines minimize the compilation overhead even further by concurrently invoking compilation of the next tier during the execution of the current one, which is called a concurrent JIT compilation.

In our proposed system in the figure, there are N tiers of JIT compilation. At first, JavaScript code is managed by the first tier until the execution time of the code exceeds

its threshold. If a code section is identified as hot through runtime profiling, the next tier is triggered to optimize the code more aggressively. On the other hand, if there is a violation of speculative optimization, the code is deoptimized to the lower tier. In this manner, the *N*th tier is finally invoked for long-running code sections. The number of tiers and their optimization thresholds vary according to the JavaScript engine and are tunable.

When the *N*th tier is requested, the decision block determines whether to enable a parallelization pass to minimize unnecessary parallelization analyses and optimizations. Above all, the algorithm checks whether the OSR entry point is also the entry of a function, because if an OSR entry point is in the middle of a function, the dependencies are difficult to analyze properly (Section 3.1). Depending on the optimization strategy, the parallelization decision block can also discard code sections that are liable to cause deoptimization. JavaScript engines infer data types based on profile data before compiling, so data types may be undecided at compile time if profiled data is insufficient. In this situation, the assumptions made at compilation are prone to be violated by causing code sections to be deoptimized to the previous tier. Our parallelization decision algorithm decides not to parallelize code sections that include the undecided data type in profiled data and censors them.

Assuming the parallelization is enabled, the parallel tier compiler conducts parallel loop analysis and parallel code generation. First of all, the parallelizing compiler analyzes loop-carried dependency to detect DOALL loops. The special considerations necessary for JavaScript dependency analysis are discussed in Section 3.1. After that, the compiler tries array privatization to detect further parallelizable loops and select the final parallelization target loops as described in Section 3.2. Finally, the compiler parallelizes those loops that have been identified as parallelizable. The issues and details of parallel code generation are covered in Section 3.3.

3.1. Dependence Analysis for Parallelization

JavaScript Object. Our proposed system helps the compiler perform precise dependence analysis by providing information from the JavaScript engine. First, we assist the compiler's pointer analysis [Hind 2001] by exposing information on JavaScript objects being internally managed by the JavaScript engine. The way the engine handles objects leads to pointer-to-pointer accesses, which compilers cannot clearly analyze. In order to access an object property, the JavaScript engine loads the address of the object and then reads the property address based on the loaded object address. Figure 2(a) shows an example of IR generation in JavaScriptCore for accessing an array-typed JavaScript object, which forms pointer-to-pointer accesses. At first, the address of the object is loaded from a constant value address (line 1) that is given to each JavaScript variable. Then, the payload address is calculated by adding the payload offset, that is, eight, to the address of the object (line 2). After that, from the payload address, the actual base address of the property is loaded (line 4). Therefore, the compiler cannot disambiguate the address of the load operation at line 4, making it hard to determine whether there is a dependency or not. In this situation, the compiler conservatively behaves as if the load operation at line 4 is dependent on all other store operations, so many loops accessing the object property cannot be parallelized. The use of type-based alias analysis (TBAA) [Diwan et al. 1998] is effective for the dependence analysis of JavaScript objects since it disambiguates distinctive alias classes of differently typed objects. The JavaScriptCore's FTL JIT compiler adopts TBAA for dependence analysis. However, if two different objects are in the same alias class, the compiler cannot identify whether the objects are dependent or not. In order to solve this problem, we leverage the fact that different objects manipulate their own unique property data so their addresses are exclusive. Our proposed system leverages the TBAA and makes

```
1  %obj = load i64* inttoptr (i64 4334912064 to i64*)
2  %payload = add i64 %obj, 8
3  %payloadptr = inttoptr i64 %payload to i64*
4  %base = load i64* %payloadptr
5  %index = load i32* %0
6  %indexz = zext i32 %index to i64
7  %index_s3 = shl i64 %indexz, 3
8  %elemaddr = add i64 %base, %index_s3
9  %elemaddrptr = inttoptr i64 %elemaddr to i64*
10 store i64 %indexz, i64* %elemaddrptr
```

(a) Original IR

```
1  %obj = load i64* inttoptr (i64 4334912064 to i64*)
2  %payload = add i64 %obj, 8
3  %payloadptr = inttoptr i64 %payload to i64*
4  %base = load i64* %payloadptr
5  %index = load i32* %0
6  %indexz = zext i32 %index to i64
7  %baseptr = inttoptr i64 %base to [10000 x i64]*
8  %scevgep = getelementptr [10000 x i64]* %baseptr, i64 %indexz
9  store i64 %indexz, i64* %scevgep
```

(b) Enhanced IR

Fig. 2. Example of IR code generation in JavaScriptCore.

property access information available to the compiler by augmenting metadata to the IR code. With this information, the compiler can identify which memory operations are property accesses, and it knows that they are independent of other stores if they belong to different alias classes. However, it is also possible that different JavaScript variables reference the same object, which is hard to resolve at compile time. Thereby, the compiler parallelizes loops based on the assumption that different variables would reference different objects. If this assumption is violated, the transformation for parallel execution should be invalidated. To solve this issue, our compiler identifies variables that are touched within each parallel loop and inserts a speculation check before entering the parallel loop. The inserted speculation check compares all objects' address pairs that include at least one write operation in a parallel loop to monitor whether different variables accessed within a parallel loop have references to the same objects at runtime. Consequently, if any object conflict is detected during the execution, the JavaScript engine bails out the parallelized code. This speculation mechanism is similar to the inspector and executor schemes [Rauchwerger and Padua 1994, 1999], in that an inspector tests the memory references that cannot be resolved at compile time before entering the parallel execution. Considering the scalability of this approach, we do not support the case that the object's property is also an object, for example, an array of object. In this case, our compiler conservatively alerts a data dependence due to unresolved memory disambiguation as its default behavior.

Moreover, we support loop-carried dependency checks in array element accesses by providing the array information to the compiler. Figure 2(a) shows an example of array element accesses. By adding up the base address and the element's index multiplied by the size of each element, the address of an array element can be obtained (lines 7 and 8), with which we can access the array element itself (line 10). However, it is hard for the compiler to analyze array accesses because the base address does not contain any array information. We can enhance IR code generation to reveal more specific array information such as array size and whether the base address is actually pointing to an array so that the information is known at compile time. In the LLVM-based compiler, the *getelementptr* instruction [Lattner and Adve 2004] can help to conduct

affine analysis by providing information on affine subscripts. The *getelementptr* takes a base address, indices, and data type of an array element as arguments. With this modification, our parallelizing compiler is able to perform affine analysis of JavaScript's array objects and detect affine parallel loops. Figure 2(b) shows the enhanced IR code generation made possible with this modification for the same array access example as in Figure 2(a).

Admittedly, if we analyze dependence at the language level, augmenting array information to IR and identifying the object property access patterns are simply not necessary. However, we still need to analyze at the IR level because only at the language level does the compiler become ignorant of the internal behavior of the JavaScript engine, for example, object allocations or store barriers. Thus, without consideration of the engine behavior, parallelization may induce the race condition during the parallel execution.

Engine Function Call. The possible question is whether parallelizing the execution of JavaScript, an inherently sequential language, can become safe. For any complicated environment, parallel execution is safe under the condition that different threads or processes do not concurrently access the same data when at least one of those accesses is a write operation [Bernstein 1966]. As our compiler exploits loop-level parallelism, it is able to safely detect parallel loops by analyzing loop-carried dependencies, except for code including function calls. Since all call sites of JavaScript functions are invisible in the JavaScript IR code, our analysis does not analyze dependencies across function boundaries. However, we can still parallelize loops with function calls that can be inlined. Our parallelizing compiler leverages the function inlining in the JavaScript engine. However, the problem is analyzing unknown function calls, which cannot be inlined and whose name and contents are invisible to the compiler.

The unknown function call in optimized JIT code can occur when the JIT code invokes a function embedded in a JavaScript engine to handle an unusual behavior, such as when accessing an array element that is out of its array boundary. In this article, we call such unknown functions embedded in the JavaScript engine *engine functions*. Since engine functions can manipulate the JavaScript engine's call frame and heap space, their multithreaded execution may generate a data race condition, and so it is unsafe to execute engine functions concurrently. However, instead of excluding whole loops that contain engine calls from being parallelization candidates, we instantiate a *mutex* in the engine functions, allowing only one of them to execute at a time. In this environment, multiple threads that run a single program share the same VM instance, which owns the mutex lock for engine functions. In this manner, the race condition resulting from concurrent execution of two or more engine functions is avoided.

One may ask whether it is safe to just consider the engine function's atomicity not to preserve execution orders among engine functions. Determining the execution orders among engine functions can be reduced to a scheduling issue, and parallelization is also regarded as a special kind of instruction scheduling. If two operations that invoke engine functions have a loop-carried dependency in the context of JavaScript, the order between those operations will be preserved because the compiler decides not to parallelize the loop that contains loop-carried dependency in this case.

We also need to consider the dependencies between an engine function call and a memory reference exposed to JIT code. In JavaScript engines, the engine functions can only manipulate (1) VM-managed structures such as VM heap space and (2) the JavaScript objects that are passed as arguments of the engine functions, while the engine functions are unable to modify any other program visible semantics. Therefore, our compiler detects a dependence between an engine function call and a memory reference that accesses VM-managed data or an argument of the engine function call.

On the other hand, it is safe to skip dependence analysis between the other memory references in the JIT code and the engine function call.

In order to distinguish the engine function calls, we provide more function information to the compiler. For instance, we can either use metadata or extend LLVM IR with augmenting a new function attribute to inform which call operations will invoke engine functions. With this extension, the compiler is able to recognize engine function calls and conduct the proper dependence analysis.

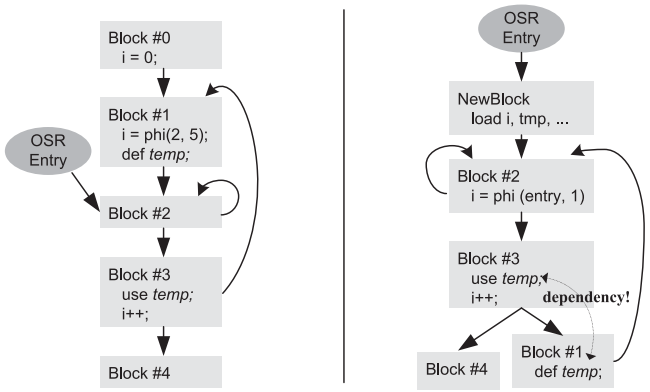
3.2. Parallel Loop Selection

Our parallel loop selection mechanism takes account of parallelization safety and profitability. Distributing loop iterations to parallel threads is safe when there is no cross-iteration dependency; parallel execution becomes beneficial when a parallel region is large enough to outweigh synchronization overhead.

As an effort to filter out loops that have no parallelization benefit, only functions with hot loops become our parallelizable target, and this can be naturally achieved by the existing hot loop detection mechanism in JavaScript engines. To ensure safeness of parallel execution, we conduct dependence analysis for loops and identify loops that do not have any loop-carried dependence as parallelizable. Some of the loop-carried dependencies can be eliminated by optimizations like array privatization or by resolving reduction patterns. Among the detected parallelizable loop nests, we select the outermost loops as our final parallelization targets. The reason for this is that parallelizing outermost loops reduces the number of synchronizations and maximizes the amount of parallel work per thread, encompassing all their inner nest loops. Also, our parallel analysis is enabled only when a target code section is a whole-function body as explained in the following paragraph.

OSR Entry. In a multitier JIT compilation system, a successive tier JIT is triggered for more optimized execution when a hot loop or a hot method is detected. On the other hand, when there is a speculation violation, the JavaScript engine invokes the lower tier to deoptimize the code. On-stack replacement (OSR) is a technique for switching between such differently optimized code for the same function. The entry of the optimization target code is called OSR entry, and the deoptimization to the lower tier is called OSR exit. The OSR entry can be either a loop header of a hot loop or a function's entry point that includes the hot section. In nested loop patterns, the innermost loop header is visited most frequently during the execution, so it is likely to be the OSR entry point. In our parallelization decision, a parallelization pass is enabled only when the OSR entry coincides with a function's start point because compilers have difficulty accurately analyzing parallel loops when the optimization target starts in the middle of a function. The reasons for this are explained next.

Figure 3(a) shows an example of a nested loop with `Block#2` as the innermost loop. With `Block#2`, the innermost loop header, as an OSR entry, the control flow of the optimization target trace becomes as in Figure 3(b). If an OSR entry occurs during the execution of a function, temporal variables should be transferred via memory, which makes it hard to determine the ranges of induction variables, and therefore, correctly disambiguating array references subscripted by multiple induction variables becomes difficult. In Figure 3(b), the compiler is ignorant of the start value of an induction variable `i` because it starts not from a constant value but from an unknown one loaded from memory. Since both the compilation with the loop header as an OSR entry and its execution happen during executing the loop, the compiler is unable to know at which loop iteration it will enter (OSR) to the optimized code. Therefore, in the generated code from the OSR entry compilation, the start iteration of a loop should be delivered from a memory location whose content is hard to be resolved at compile time.



(a) CFG in program order (b) CFG starting from OSR Entry

Fig. 3. Problems when OSR entry is not a function's entry.

Also, it becomes harder for a compiler to analyze affine expressions if the analysis scope starts from the innermost loop. Since the execution order of an instruction sequence becomes out of sequence with the inner loop OSR entry, as in Figure 3(b), unexpected cross-iteration dependencies are generated for temporal variables, which initially do not contain any dependency. For example, in Figure 3(a), a temporal variable, `temp`, is defined in `Block#1` and used in `Block#3`, so it can be privatized for each iteration and there is no cross-iteration dependency. However, after compilation with `Block#2` as OSR entry, `temp` cannot be privatized anymore. Since `Block#2`, `Block#3`, and `Block#1` form a loop in that order, the use of `temp` precedes its `def` that is consumed by the next iteration. Thus, an unexpected cross-iteration dependency for `temp` is generated in the example of Figure 3(b).

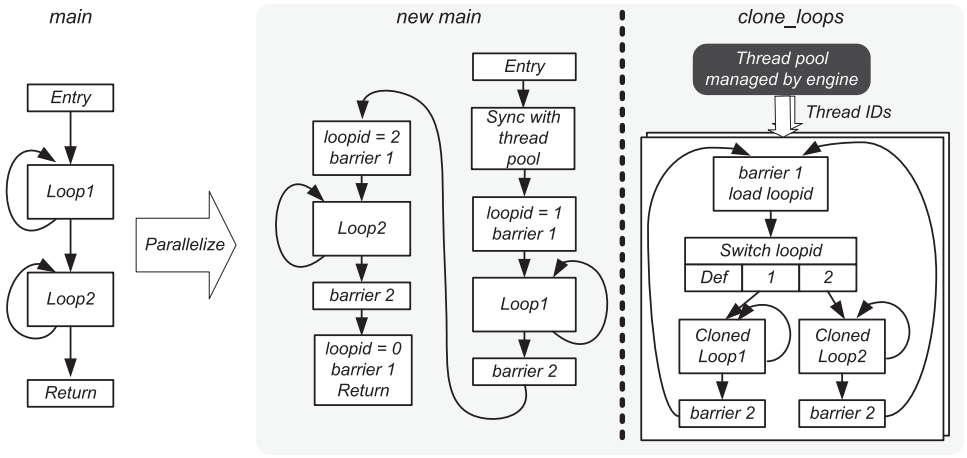
Moreover, the compilation with a hot loop header aims to fast transfer the current hot loop execution to a more optimized one during the execution. Thus, if the compilation with the hot loop header becomes heavier, entering to the optimized code would be delayed, which may negatively impact performance. In addition, even if a loop can be parallelized by analyzing from its loop header, the performance benefit would be negligible. The reason is that once JIT execution enters a function's entry, other OSR entry points will not be triggered unless the JIT execution encounters an OSR exit. It means that the parallelized code starting from a loop header will not be much executed.

Therefore, in order to avoid a possible slowdown and minimize redundant compilation work, we enable parallelization only when the OSR entry is the starting point of a function, as shown in the decision block of Figure 1. Thereby, the optimization target becomes a whole-function body. When parallelizing applications automatically, this rule is generally adopted.

3.3. Parallel Code Generation

Basically, our parallel code generation adopts the SPMD (single-program, multiple-data) model. The number of parallel threads is determined by an environment variable and the iterations of a parallel loop are evenly distributed to the determined number of threads. The lower and upper bounds of iterations given for each parallel thread are calculated by its own thread ID.

Figure 4 shows an example of parallel code generation for a function `main` that includes two parallel loops. After parallelization, two new functions are generated. One is `new main`, augmented with synchronization code, and the other is `clone_loops`, which contains the copy of all parallel loops from the `main` function. A main thread



(a) Sequential function (b) Function with parallellized loops

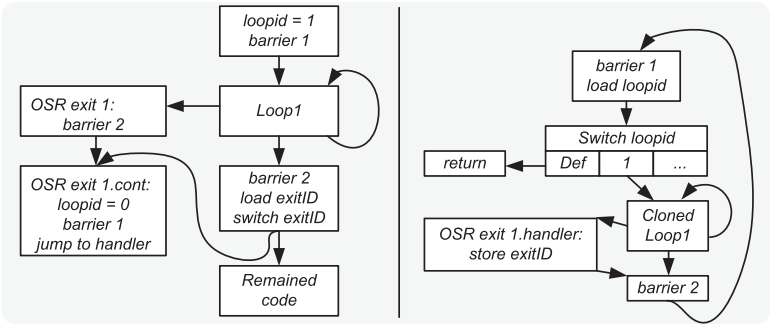
Fig. 4. Example of parallel code generation.

executes `new main`, and a number of threads in a thread pool execute `clone_loops` with their own thread IDs simultaneously. To this end, the main thread transfers a function pointer of `clone_loops` to a thread pool that is waiting for synchronization with the main thread. Then, each thread in the pool invokes the `clone_loops` function with its own thread ID as an argument after being synchronized with the main thread. Putting all parallel loops in a single function as in this work, instead of separating parallel loops to different functions, reduces unnecessary thread management overhead.

Synchronization in the parallel execution of `new main` and multiple instances of `clone_loops` is achieved by `barriers` and a global variable, `loopid`. `Barrier1` in the figure regulates the order of store and load accesses to and from `loopid`. Due to `barrier1`, assigning `loopid` in `main` always comes before loading the value of `loopid` in `clone_loops`. Since `loopid` helps child threads to select which loop is going to be entered next, the main and child threads can execute the same loop at the same time by interfacing through `loopid`. Also, `barrier2` is inserted in every loop exit in order to synchronize each parallel loop. In the figure, the function `new main` visits `barrier2` after the execution of `loop2`. Before the function `new main` returns, it resets `loopid` with `zero` and waits at `barrier1` again in order to make sure the threads executing `clone_loops` can identify `loopid` as `zero` and terminate in company with `new main`. As `loopid` becomes zero, the following `switch` statement in `clone_loops` will move the control to the `default` case and `clone_loops` will safely terminate. For simplicity, this example omits our OSR exit support during parallel execution, so the following section addresses the issue in detail.

4. SAFE AND LIGHTWEIGHT RECOVERY

It is challenging to recover a program's sequential state from an OSR exit during parallel execution. When there is an OSR exit from parallelized code, the previous tier should sequentially execute the code from the failure point after multiple threads have manipulated their program's state. In this situation, choosing a restarting point and ensuring correct execution results are challenging because the live-in variables of operations would have been changed nondeterministically by the parallel threads. Therefore, many parallel execution frameworks employ a checkpoint-rollback mechanism, which restarts from the checkpointed state. However, this mechanism leads to



(a) Main thread (b) Child thread

Fig. 5. OSR exit handling in main and child threads.

huge overhead, especially for computationally intensive HTML5 applications managing large datasets. In this section, we propose safe OSR exit handling and a lightweight recovery mechanism leveraging the property of idempotence.

Handling OSR Exits. During the execution of JavaScript JIT code, there can be one or more OSR exits due to speculation failures in parallel loops, so our compiler should safely handle the OSR exits. Our code generation policy allows the actual OSR exit operations on speculation failure only in the main thread. Child threads (i.e., threads for `clone_loops`) just forward the OSR exit information to the main thread when they have encountered a speculation failure. Figure 5 shows an example of OSR exit handling for a parallel loop in main and child threads. If an OSR exit occurs during the execution of a cloned loop in a child thread, it is not actually deoptimized at that point. Instead, the child thread stores an OSR exit ID and then waits at `barrier2` so that the main thread can be informed about the OSR exit information from the child thread by reading the exit ID after the corresponding parallel loop is finished. When the main thread itself is exiting or it has been notified of an OSR exit request from a child thread through an exit ID, it assigns `loopid` as zero and jumps to the corresponding OSR exit handler without advancing through the remaining code. Then the main thread and child threads terminate after being synchronized at `barrier1`.

There are two possible consequences in our OSR handling mechanism: (1) the OSR exit triggering point may become different from that of sequential execution, and (2) there may be a slowdown due to delaying an OSR exit until all the loop executions of other threads are finished. First, the possibly different OSR exit point will not be a problem because in our idempotence recovery mechanism, the exited parallel loops will re-execute from their beginning, no matter where the OSR exit point has been triggered. Second, the performance impact of the delayed OSR exit will be insignificant. In our SPMD model, where all parallel threads execute a single program, all the threads are likely to trigger OSR exits at the same program point. Thereby, the delay of waiting for all other threads to finish their loop iterations will be negligible.

Idempotence Recovery. In this article, we leverage the property of idempotence to tackle recovery overhead. An idempotent region is defined as a region that is safely re-executable in spite of disruption within it [de Kruijf et al. 2012]. There have been many studies leveraging the property of idempotence in speculative execution [Kim et al. 2006; de Kruijf et al. 2012; Menon et al. 2012], and Kim et al. [2006] showed that most identified parallel loops are idempotent. Inspired by these studies, *among our identified parallel loops, only idempotent parallel loops are actually executed in*

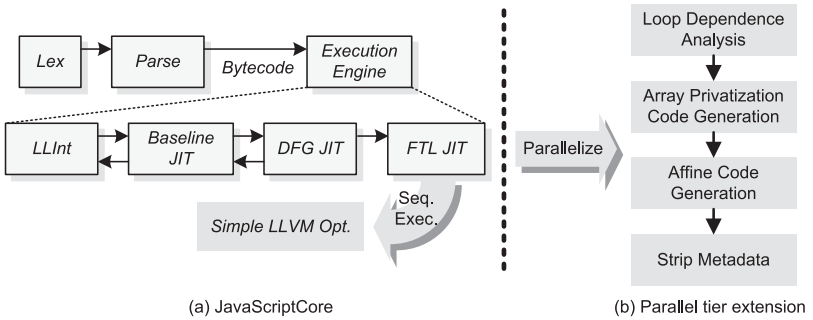


Fig. 6. Parallel tier extension on JavaScriptCore.

parallel. Thereby, our parallel loop executions can be safely recovered by re-executing from each of their first iterations when OSR exits have been triggered.

The idempotence of global variables is also guaranteed due to JavaScript's sequential nature of not allowing any other scripts or an HTML page while processing a JavaScript function. Although a new multithreading feature in HTML5, Web Workers [Hickson 2014], allows concurrent execution of scripts and browsers, the Web Workers are designated as sharing nothing and thus cannot access globals without posting or receiving messages.

To support the idempotence recovery, our compiler checks the idempotence of identified parallel loops based on the information generated by dependence analyses. Our compiler identifies a parallel loop as idempotent if all live-ins of a code section do not change [de Kruijf et al. 2012]. This inspection has been only applied to nonprimitive JavaScript objects including arrays that contain large amounts of data and that are exposed to the JavaScript engine. Primitive objects can be recovered by the engine's existing OSR mechanism at low cost.

To support OSR, put simply, JavaScript engines map the values or locations of live-in variables corresponding to each exit point. Additional consideration for primitive objects is the handling of induction variables. As our idempotence mechanism restarts from the first iteration of the loop exited, the induction variable should retrieve the initial value. Our compiler identifies the induction variable of each parallel loop and manipulates the engine's OSR mapping of the induction variable so it is reset with its initial data. However, this mechanism may interfere with profiling at OSR exit points. When an OSR exit occurs, a conventional JavaScript engine profiles information such as the resulting value and the cause of the OSR exit. Such profiling data allows a JavaScript engine to generate machine code that triggers fewer OSR exits and to be more optimized. However, in our mechanism, the restart and exit points are different, so the engine would miss the profiling information at the exiting point, and this may increase the number of OSR exits encountered compared to the sequential execution. To avoid this problem, we carefully connect the profiling information and the exact exiting point. Our parallelizing compiler induces an identical number of OSR exits to the original sequential execution.

5. CASE STUDY: IMPLEMENTATION ON JAVASCRIPTCORE

As a case study, we implemented our parallelizing JavaScript compiler on top of JavaScriptCore, a WebKit's JavaScript engine [Apple Inc. 2015]. Figure 6 shows the overall architecture of our prototype system. Since the original JavaScriptCore's FTL JIT (fourth-tier LLVM) [Pizlo 2014] invokes LLVM optimization passes (Figure 6(a)), we integrated our parallelizer into the FTL JIT like other LLVM optimization passes

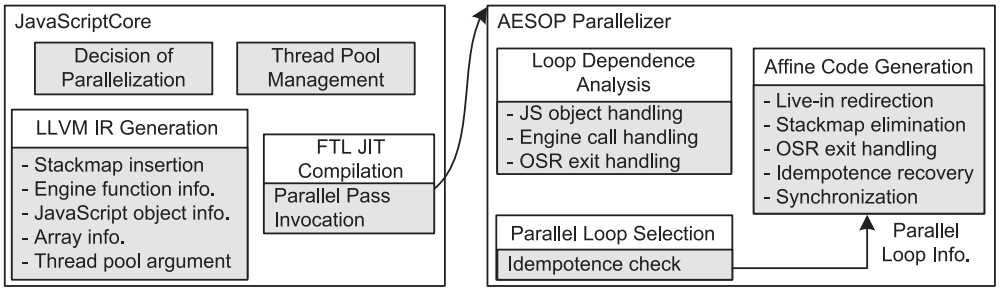


Fig. 7. Summary of our detailed implementation on JavaScriptCore and the AESOP parallelizer. Lists in gray boxes are our new implementation.

(Figure 6(b)). Our parallelization passes were implemented based on AESOP [Kotha et al. 2013], an LLVM-based static parallelizer. Figure 6(b) shows our parallel loop analysis and code generation passes adopted from the AESOP parallelizer. Also, our new implementation on top of JavaScriptCore and AESOP parallelizer is elaborated in Figure 7.

We modified the way that the parallelization passes in AESOP exchange their analysis information to make it more suitable for the web environment. The loop dependence analysis pass generates parallel loop information, which two subsequent passes (i.e., array privatization and affine code generation) use for parallel code generation. The original standalone AESOP parallelizer delivers parallel loop information in a file format, which is inappropriate for the web environment. Thus, we modified it to utilize the metadata interface supported in LLVM IR instead. In our enhancement, the names of all detected parallel loop headers are marked as LLVM IR's metadata after the loop dependence analysis has finished. There might be a suggestion of using the *getAnalysis* method to obtain an instance of the dependence analysis pass, since in LLVM this is the common way to exchange analysis information across different passes. However, the *getAnalysis* method requires the actual invocation of its argument pass, and that should be avoided considering the compilation overhead. Therefore, for our system, interfacing with metadata is a better solution.

All our proposed solutions for parallel loop analysis issues (Section 3.1) have been implemented as shown in Figure 7. The solution to the *OSR Entry* issue was well integrated into the compilation procedure in FTL JIT. There are two kinds of compilation, *replacement* and *OSR entry compilation*, in JavaScriptCore's FTL JIT. The replacement compilation translates a whole-function body into machine code, while the OSR entry compilation takes a certain OSR entry point that executes frequently as its optimization target. If the counter for detecting hot code section reaches its threshold, FTL JIT performs the replacement compilation and waits before entering the function's entry point or triggering the OSR entry compilation. Then, if the program reaches the function entry point quickly, the replacement code generated by FTL JIT is executed. Otherwise, the OSR entry compilation is triggered. In this work, we enabled parallel analysis and code generation passes only for the replacement compilation mode. Thereby, our parallelizing compiler can avoid unexpected data dependency and the problem of the ambiguous iteration ranges that may be induced if the OSR entry point is not the starting point of a function. Also, our LLVM-based solution for the *Dependence Analysis* issue can be directly applied to our JavaScript parallelizing compiler because its implementation is also based on LLVM. As proposed in Section 3.1, we utilized LLVM's *metadata* and *getelementptr* instructions for precise dependence analysis in our JavaScriptCore instance. Furthermore, JavaScriptCore's API provides *MutexLocker* to create a lock for a given scope, so we instantiated it in the *engine*

functions invokable from FTL JIT code to support the *Engine Function Call* issue. As a result, our parallelizing compiler provides the mutual exclusion of multithreaded executions of engine functions.

Also, the idempotence recovery mechanism presented in Section 4 has been realized in our JavaScriptCore instance. In order to support OSR, JavaScriptCore manages *StackMap*, a special section that records potential OSR points and their live-in variables. To support this StackMap behavior, FTL JIT utilizes the *stackmap* intrinsic function that has been supported by LLVM for JavaScriptCore. The *stackmap* intrinsic holds an OSR Exit ID and its live-in variables as its arguments. The engine in JavaScriptCore holds the exit type, the exiting point, and the exit profiling point for each OSR Exit ID. When an OSR exit occurs, FTL JIT execution terminates and triggers the JavaScriptCore's execution engine to handle the OSR exit. Then, the execution engine identifies the OSR Exit ID, collects the profiling data at that exiting point, relocates the live-in variables in StackMap, and triggers the lower-tier JIT to restart from the exiting point. For our idempotence recovery mechanism, we redirected the OSR exit point to the parallelized loop header, while the OSR profiling point remains as the point where the OSR exit had actually occurred. In this way, our implementation on JavaScriptCore can conduct idempotence recovery for speculation violations without losing any profiling information.

6. EVALUATION

6.1. Experimental Setup

In order to verify parallel loop detection coverage of our proposed compiler, we selected the PolyBench/C suite [Yuki 2014]. PolyBench/C is a polyhedral benchmark suite written in C, which contains big arrays and highly parallelizable loops. It has been established that most PolyBench/C benchmarks gain improved performance through loop parallelization [Kotha et al. 2013]. We identified whether the parallel loops detected in native PolyBench/C are also parallelizable in JavaScript using our parallelizing compiler. For this evaluation, the four benchmarks were selected from PolyBench/C and manually converted into the equivalent JavaScript code. When writing the JavaScript benchmarks, we inlined all subfunction calls and mapped all multidimensional arrays to 1D ones indexed by a linear multivariable expression. We wrote our code to use 1D arrays because our compiler does not support objects that include nonprimitive objects as their property yet. However, in terms of analysis itself, analyzing 1D arrays indexed by multiple induction variables is much more complicated than analyzing 2D arrays indexed by a single induction variable. Therefore, we maintain that this level of different condition for comparison is acceptable. Also, notably, our hand-coded JavaScript does not utilize the typed array feature, but we use the JavaScript's *Array* object to express arrays.

Since existing benchmark suites such as V8, SunSpider, and Octane show different behaviors from real web applications [Ratanaworabhan et al. 2010] and the web pages in the Alexa list [Alexa 2015] do not contain intensive computation, we selected more realistic sets of HTML5 applications to evaluate our compiler. The selected applications are the Pixastic JavaScript image processing [Seidelin 2014] library and a set of *real* emerging HTML5 applications taken from River Trail [Herhut et al. 2013]. The Pixastic image library evaluated in this article is widely used for real-world JavaScript image processing. There are many web research projects that have been developed and their work evaluated with the Pixastic image library [Porcides et al. 2011; Mehrara et al. 2011; Bartsch et al. 2014; Cho et al. 2014]. Also, HTML5 applications in Herhut et al. [2013] are representatives of emerging, data-parallel HTML5 workloads, and most of them were developed by third-party JavaScript programmers.

Table I. Comparison of Parallel Loop Detection Between PolyBench/C and Manually Converted Equivalent JavaScript Codes

	# Outer Loops	# Parallel Loops in C	# Parallel Loops in JavaScript
syrk	4	4	4 (1)[a]
gemm	4	4	4 (1)
2mm	6	6	6 (1)
fdtd-2d	6	5	5 (3)

[a]The number of parallelized array init loops.

We tested 12 out of 28 image processing filters and effects in the Pixastic Library. The unchosen benchmarks include JavaScript function calls in their function body, which are unsupported by FTL JIT in Safari Version 7.0.2 used in our system. To evaluate the performance of each image benchmark, we measured the total execution time when processing 10,000 images. Also, we selected 11 out of the real HTML5 applications introduced in River Trail and excluded applications such as Bug and TourDeBlock due to browser compatibility issues. This compatibility issue occurs because our parallelizing compiler and River Trail are based on different web browsers, Safari and Firefox, respectively. Most of the HTML5 applications from River Trail are real-time web applications for which we measured frame rate changes instead of total execution time.

All reported results are averages of 10 measurements. *It is important to note that the performance results in this article, except for measurements of real-time frame rates, include the whole elapsed time, which includes interpretation, JIT compilation, and execution of cached JIT code.* We evaluated our parallelizing compiler on a quad-core, 2.3GHz Intel i7 processor, running OS X 10.9. Due to the limitation of our hardware platform that is a quad-core environment, we could not examine speedups of more than four threads. Even though our platform is equipped with hyperthreading, we disabled that feature because the hyperthreading could not actually improve the performance when CPU utilization is already high. Thus, in order to avoid possible misleadings of scalability and potential speedups, we evaluated on a maximum of four threads and disabled the hyperthreading.

6.2. Parallel Loop Detection Coverage

Table I compares the parallel loop detection coverage between PolyBench/C by the native AESOP parallelizer and manually converted equivalent JavaScript codes by our parallelizing compiler. The second column indicates the number of outer loops in each source code. Since the parallelizer's target loop selection policy gives higher priority to the outermost loops, only the outer loops were counted as parallelizable. The third column shows the number of automatically identified parallel loops by the AESOP native parallelizer for C. Since the studied benchmarks contain simple and highly parallelizable loops, we can see that most of the target loops are detected as parallelizable except for a loop in `fdtd-2d`. Also, we checked that all the parallel loops detected in the benchmarks are idempotent after loop fusion optimization [Grosser et al. 2012], which means that the loops can be safely parallelized without considering OSR exits. The one loop that could not be parallelized in `fdtd-2d` inherently contains loop-carried dependencies in its semantics, so it is plausible for the C parallelizer not to detect parallelism within it. The last column shows the number of parallel loops detected by our parallelizing JavaScript engine and, in parentheses, the number of *array init loops* generated by the JavaScript engine. After a new array is created with a size, each array element should be initialized with a *NaN* property before being given its initial value. In JavaScriptCore, the *array init loops* visit each array element and initialize it with a *NaN*. Our parallelizing JavaScript compiler also identifies the *array*

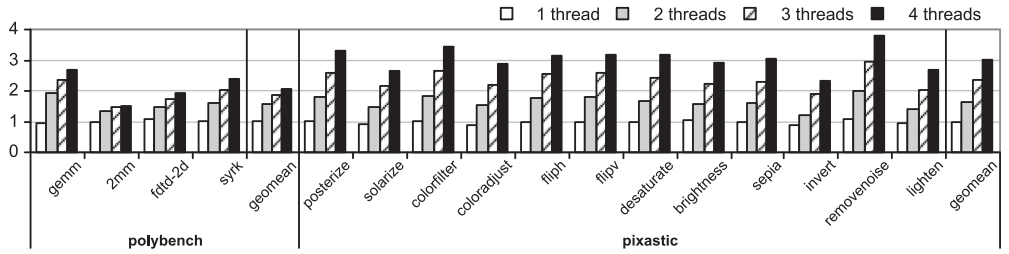


Fig. 8. Speedup in the different number of threads with Polybench and Pixastic.

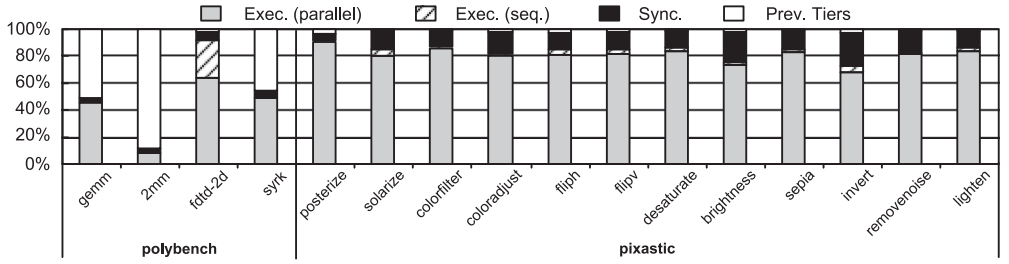


Fig. 9. Time breakdown of parallel execution with Polybench and Pixastic on four threads.

init loops as parallelizable targets. The data shows that our parallelizing JavaScript engine achieves 100% detection of parallel loops with our studied benchmarks and all *array init loops* are also parallelized. Although all the loops in the studied benchmarks are affine loops that can be almost perfectly analyzed by conventional static compilers, this outstanding level of coverage is meaningful because we obtained this data from the execution of JavaScript, which could not be automatically parallelized before.

6.3. Performance Analysis

Polybench and Pixastic Benchmarks. Figure 8 shows the performance improvement of our parallelizing JavaScript engine for one, two, three, and four threads compared to the original JavaScriptCore with FTL optimization enabled. The rightmost bar of each benchmark set shows the geometric mean. It is shown that execution on four threads always produces the best performance, with average speedups of 2.07 and 3.02 times, with the Polybench and Pixastic benchmarks, respectively. Also, the speedups scale well with the number of threads except for `2mm`. `2mm` is saturated after three threads since the execution of previous tiers dominates most of the time spent in `2mm` and thus its parallelizable region occupies only a small portion of the total execution time.

Figure 9 shows the time breakdown on four-thread execution with Polybench and Pixastic. Parallel and sequential parts in multithreaded JIT execution are depicted as the first and second items, respectively, in the stacked graph. These parallel and sequential portions indicate pure parallel and sequential execution time. The synchronization part in the graph represents the thread waiting time at barriers, which stands for thread synchronization overhead. The portion of `Prev. Tiers` indicates the whole previous execution time before the multithreaded JIT execution, so the most dominant part of `Prev. Tiers` is the sequential execution time of the previous tier JITs. Also, `Prev. Tiers` includes JIT compilation time and previous OSR recovery time as well, although the amounts are minimal.

It is shown that `2mm` execution is dominated by previous tiers. Thereby, the maximum speedup of `2mm` is limited according to Amdahl's law [Amdahl 1967]. In this experiment,

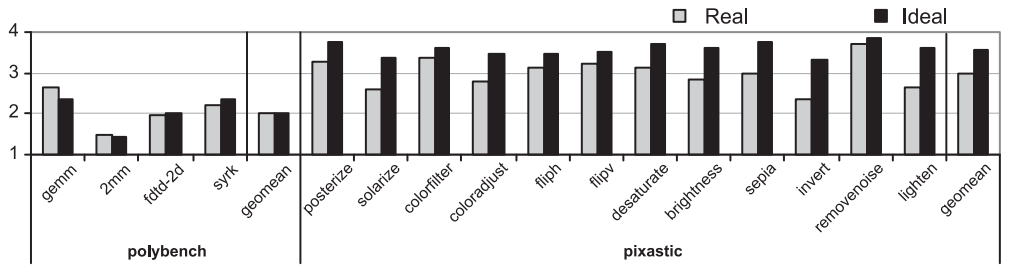


Fig. 10. Comparison of real and ideal speedups with Polybench and Pixastic for four threads.

Table II. Description of the Studied HTML5 Applications

Benchmark	Description
Video-*	Runtime filters on an 896x514 video stream
NBody	4,000-particle simulation/animation
Mandelbrot	Mandelbrot set kernel
Procedural Terrain	Procedural map rendering based on "Minecraft in JavaScript" code
Liquid-Resize	Content-aware image resizing with an 800x542 image

we invoked `2mm` 10 times, and the whole execution had a total of five OSR exits. Therefore, our parallelized execution could only affect the last half of the execution time and the parallelizable region was restricted.

Figure 10 compares the performance of real execution and the upper bounds of theoretical performance with Polybench and Pixastic. The theoretical speedup was calculated based on Amdahl's law. We assumed that the parallel loop regions that do not trigger OSR exits as parallel regions and the others as sequential regions when estimating the ideal speedups. Since our theoretical estimation does not consider any parallelization overheads such as synchronization (i.e., waiting at barriers) and changes in cache behavior, this estimation can differ from our real experimental results.

For some of the Polybench suite, our parallelizing compiler shows higher performance improvement than the theoretical case. This result is possible because capacity misses have been reduced by parallelization due to the data distribution, which can boost the performance more than was expected from parallel execution alone. All the studied Polybench benchmarks have arrays that are first initialized in loops and then accessed in the following kernel loops. Therefore, splitting loop iterations and data into different cores can avoid capacity misses and bring more local cache hits. Since `gemm` and `2mm` contain huge amounts of data that exceed the private cache size, parallelization significantly reduces cache misses and improves the performance of the parallel region more than n times, where n is the number of threads. On the other hand, `fdtd-2d` and `syrk` are less memory bound, and the speedups are slightly lower than the theoretical case due to the synchronization overhead, as shown in Figure 9.

With Pixastic, the performance improvement of our parallelizing compiler is on average 17% lower than the theoretical potential speedups. The time breakdown in Figure 9 implies that most of these performance gaps result from the synchronization overhead. In our measurement, the impact of cache contention, limited memory bandwidth, and increased rate of garbage collection due to parallel execution are negligible in total execution time.

Data-Parallel HTML5 Applications. We evaluated the proposed parallelizing JavaScript compiler with emerging, data-parallel HTML5 applications from third parties, and those were also evaluated in River Trail [Herhut et al. 2013]. Table II lists the

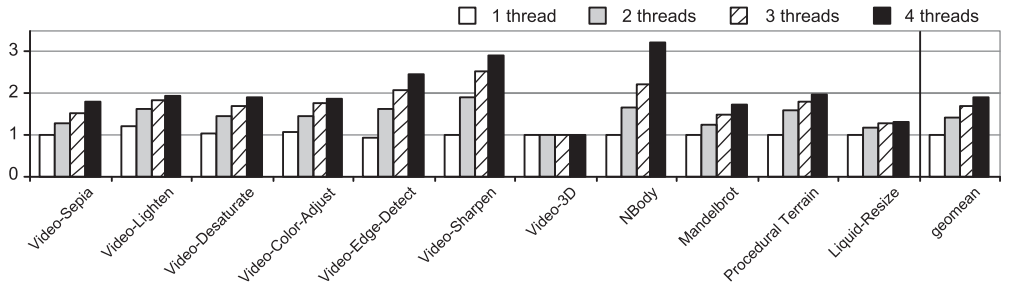


Fig. 11. Speedup in the different number of threads with various HTML5 applications from third parties [Herhut et al. 2013].

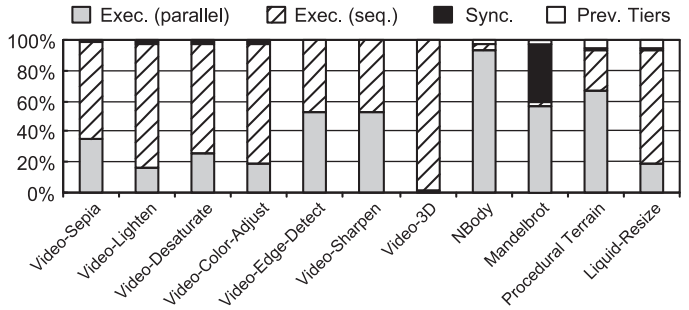


Fig. 12. Time breakdown of parallel execution with the studied HTML5 applications on four threads.

description of the studied HTML5 benchmarks, and all the detected parallel loops are idempotent after separating input and output arrays. Figure 11 shows the speedups of our parallelizing JavaScript engine for one, two, three, and four threads. We can see that execution on four threads always produces the best performance, with average speedups of 1.92 times, and the speedups scale well with the number of threads except for `Video-3D`. `Video-3D` in Figure 11 does not benefit from the parallel execution. The reason is that our parallelizing compiler failed to parallelize `Video-3D` because of loop-carried memory dependencies in array allocations.

Figure 12 shows the time breakdown of four threads' parallel execution with the River Trail benchmarks. Most of the *Video-3D* execution is sequential because the loop that dominates the entire execution of *Video-3D* could not be parallelized with our proposed compiler as mentioned earlier. In all the *Video-** benchmarks, sequential execution parts dominate overall execution time since their main threads conduct rendering and manipulating a video file as well as processing parts of data-parallel computations. *Video-Edge-Detect* and *Video-Sharpen* are computationally intensive and thus their parallel execution portions are relatively large compared to the other *Video-** benchmarks. Since *NBody* contains excessive computation, its parallel execution occupies most of the time. This high level of parallelism also explains the steep performance improvement with *NBody*, that is, 3.22 times, as shown in Figure 11. *Mandelbrot* contains abundant parallelism but suffers a load imbalance with our proposed load distribution mechanism that distributes an equally divided chunk of iterations to each thread. The reason is that with *Mandelbrot*, the larger the iteration number is, the more execution time it takes, which incurs high synchronization time waiting for other threads to complete, as shown in Figure 12.

Also, we compared the performance of our work to that of River Trail, a parallel JavaScript engine [Herhut et al. 2013]. The River Trail engine works with its

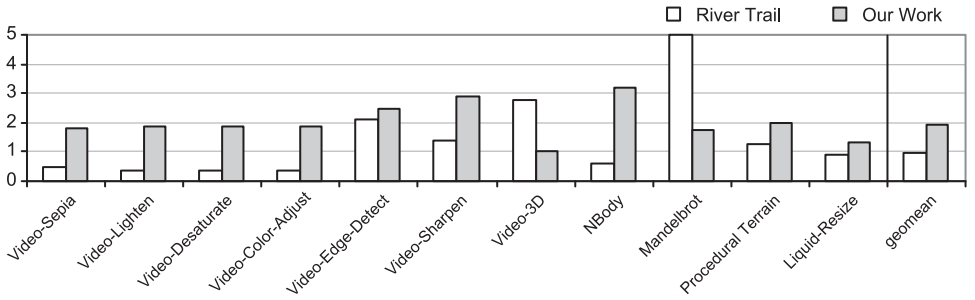


Fig. 13. Speedup of River Trail and our proposed work on JavaScriptCore with HTML5 applications.

parallelized version of JavaScript, while our work automatically parallelizes the sequential version of JavaScript. Both of them ran on our quad-core platform, and the comparison of speedups normalized to each of their own base web browsers is shown in Figure 13. As shown in the figure, our parallelizing compiler outperforms the River Trail engine except for *Mandelbrot,* with the load imbalance, and *Video-3D,* from which our proposal failed to detect parallel loops. The River Trail engine shows performance degradation due to OpenCL runtime overheads when parallelizable workloads are relatively small. However, the River Trail engine takes advantage of SIMD vectorization in the OpenCL compiler, so performance improvement can exceed the number of threads, as in the *Mandelbrot* case. On average, the River Trail engine and our proposal show 0.97 and 1.92 times speedups, and this result becomes more impressive when considering that the River Trail engine requires JavaScript applications that have been written with its parallel APIs, while our work automatically exploits parallelism from sequential JavaScript applications.

Compilation Overhead. We measured the FTL JIT compilation time of both the original sequential JavaScriptCore and our parallelizing compiler. Our compilation overheads are only a maximum of 0.215% of the total execution time. The majority of JavaScript engines including JavaScriptCore support concurrent multilevel JIT compilation so the compilation overhead can be hidden by JavaScript execution. We can infer the performance impact of the compilation overhead from the performance degradation of the single-threaded execution time of our parallelizing compiler compared to the execution time of the original JavaScriptCore in Figure 8. The single-threaded execution in the figure undertook all of the parallel loop analysis and code generation process of our parallelizing compiler but ran on only a single thread, and thus, its performance degradation indicates our compilation overhead. However, since the average performance degradation ranges from −0.099% to 0.049%, there is no meaningful degradation, so compilation overhead is minimal.

Idempotence Recovery Overhead. In order to minimize checkpoint and rollback overhead, this article proposes an idempotence recovery mechanism that restarts execution from the beginning of the exited loop instead of manipulating checkpoints. Since the re-execution of an entire loop may increase the execution time, we measured this recovery overhead, and it is shown in Figure 14.

The graph describes three benchmarks, the only benchmarks that encounter OSR exits due to misspeculations among the studied benchmarks, which means that the other benchmarks do not experience any misspeculation overheads with our proposed compiler. In our experiment, `2mm` shows the biggest recovery overhead, that is, 5% of the total execution time of a function invocation. The overheads with `gemm` and `syrk` are negligible. These low re-execution overheads are leveraged by the fact that

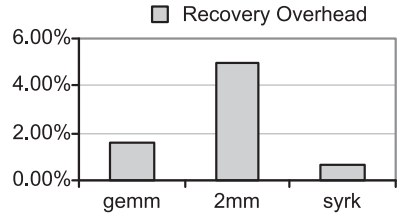


Fig. 14. Idempotence recovery overhead.

Table III. Applications Invoking Engine Functions that Acquire a Mutex Lock

Benchmark	Engine Function	# Calls
gemm	operationEnsureDouble	112
syrk	operationEnsureDouble	8
Video-Sharpen	operationEnsureDouble	n/a[1]

[1]The exact number of engine calls in Video-Sharpen could not be specified since it is a real-time application.

the main cause of the OSR exits in this experiment is an inline cache miss, which normally appears in the first iteration of data-parallel loops. Notably, this low level of recovery overhead is remarkable because, in the previous speculation-based research, checkpointing and speculation overhead is around 17% on average [Mehrara et al. 2011].

Mutex Overhead. In this work, we initiated a mutex for engine function calls in parallel loops to guarantee the mutual exclusion of each engine function call. Normally, mutex locks behave as a performance bottleneck in parallel execution, so in order to show the performance impact of the mutex, we list the occurrence of the engine function calls acquiring a mutex in our studied benchmarks, and this is shown in Table III. It is important to note that branching to slow operations that contain engine function calls is unusual in highly optimized JIT code. Data in the table supports this tendency by showing that only three applications among all studied benchmarks actually invoke engine functions, and their impacts are minimal. `gemm`, `syrk`, and `Video-Sharpen`, the only applications that actually invoke engine functions at runtime, have shown a scalable performance improvement, as presented in Figures 8 and 11.

7. RELATED WORK

There have already been many efforts to exploit parallelism in JavaScript. Herhut et al. [2013] proposed River Trail, a parallel JavaScript engine with a programming model and APIs, which makes JavaScript execution on multicore CPUs and GPUs possible. River Trail provides a new data type, *ParallelArray*, which is guaranteed to have equally typed elements and no array holes. Also, there are new elemental functions such as *scatter* that can be parallelized to process *ParallelArray* typed data. Therefore, River Trail requires the modification of existing JavaScript applications to benefit from the parallelism with *ParallelArray*.

Moreover, Herhut et al. [2013] present a realistic set of HTML5 applications, most of which were developed by third parties. The authors point out that (1) the current situation of real web applications not containing much parallelism is a chicken-and-egg problem, and (2) existing JavaScript benchmarks such as V8, SunSpider, and Octane do not actually reflect real-world JavaScript applications. While agreeing with this point, in this article, we evaluated our parallelizing compiler with the HTML5 applications

presented in River Trail. Our proposed compiler successfully exploits parallelism from the sequential version of River Trail applications. The experimental results show that our automatically parallelizing compiler outperforms River Trail that works with the manually parallelized version of applications.

In order to dynamically extract the parallelism in JavaScript, the TLS principle has been adopted [Martinsen et al. 2013, 2015; Mehrara et al. 2011]. The basic approach of TLS is to speculatively execute parts of the program in parallel , and if the parallel threads are found to have a dependency violation at runtime, the speculative parallel execution is halted. TLS has two kinds of approaches, method-level and loop-level speculations.

Martinsen et al. [2013] proposed the method-level TLS environment. In their environment, every method call instantiates a new thread and runs concurrently with its parent method. In this manner, they exploit the method-level parallelism abundant in conventional interaction-based JavaScript programs. Although their approach is prominent, we are more interested in loop-level parallelism because HTML5 has been bringing out more computationally intensive applications for which JavaScript's low-grade performance is increasingly a critical problem.

ParaScript [Mehrara et al. 2011] is a runtime JavaScript parallelization system based on a loop-level TLS mechanism. ParaScript optimistically selects target loops that can be speculatively executed in parallel. As other TLS-based approaches do, it requires runtime violation checking and speculation rollback if there is any violation of sequential semantics. Supporting the speculation rollback requires checkpointing the live-in variables of parallel loops before their execution. Our approach is distinguished from ParaScript in three respects: (1) we fully utilize static compiler analysis passes instead of runtime dependency checks; (2) we propose a lightweight speculation recovery mechanism based on the property of idempotence, instead of checkpointing; and (3) we generate parallelization code in LLVM IR, a low-level IR, while ParaScript actually parallelizes its bytecodes.

Although the TLS-based loop parallelization technique is promising, successfully exploits loop-level parallelism, and improves the performance, in this article, we introduce another research direction, one that applies traditional compiler analysis to JavaScript parallelization. It may be debatable whether adopting a compiler analysis instead of following the TLS principle is the better choice. We agree that static compiler analysis entails limitations on resolving memory dependency, which can make a parallelizer drop opportunities to detect some parallel loops. However, in the JavaScript engine, the analysis can be conducted with much richer information since the engine manipulates runtime and profiling data. With the support of this runtime information, possibilities of reducing analysis overhead and increasing parallel loop detection accuracy will open up in the near future. Also, it is undeniable that TLS approaches entail not only runtime dependency violation checks but also speculation rollback overhead, while in the compiler analysis approach, the JavaScript engine just reuses the optimized machine code from detected and parallelized loops at runtime.

8. CONCLUSION

In this work, we proposed an automatically parallelizing JavaScript compiler by adopting the parallelization mechanism of a static compiler. Taking JavaScript's dynamism into consideration, we also proposed a safe and lightweight speculation recovery mechanism for handling disruption during the parallel execution based on the idempotence theory. The experimental results showed that the proposed parallelizing JavaScript compiler has the same level of parallel loop detection coverage with a native parallelizer. We achieved a maximum speedup of 3.22 times with various sets of realistic

HTML5 applications on a multicore platform, and there were negligible compilation and recovery overheads.

To the best of our knowledge, this is the first research work that automatically and dynamically parallelizes JavaScript applications by leveraging the mature static compiler analysis. This achievement enables the support of a variety of cross-platform HTML5 applications that may have been infeasible to single-threaded execution of JavaScript. Also, combining an interpreted language like JavaScript with static analysis enables further improvement of the compiler by utilizing the engine's abundant high-level and runtime information in the future. In fact, the time complexity of dependence analysis, an essential component of static parallelizing compilers, means that previous commodity JavaScript compilers could not easily adopt static parallelization techniques. Nevertheless, omitting dependence analysis under certain conditions is expected to become possible in our follow-up research, which will use the additional high-level semantics and runtime information available for JavaScript objects in our JavaScript parallelizing compiler, speeding it up further.

REFERENCES

Alexa. 2015. Alexa Top 500 Global Sites. Retrieved November 14, 2015, from http://www.alexa.com/topsites.

Gene M. Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference (AFIPS'67 (Spring))*. ACM, New York, NY, 483–485. DOI:http://dx.doi.org/10.1145/1465482.1465560

Matti Anttonen, Arto Salminen, Tommi Mikkonen, and Antero Taivalsaari. 2011. Transforming the web into a real application platform: New technologies, emerging trends and missing pieces. In *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC'11)*. ACM, New York, NY, 800–807. DOI:http://dx.doi.org/10.1145/1982185.1982357

Apple Inc. 2015. JavaScriptCore Engine. Retrieved November 14, 2015, from https://www.webkit.org/projects/javascript/

Utpal K. Banerjee. 1988. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, MA.

Hauke Bartsch, Wesley K. Thompson, Terry L. Jernigan, and Anders M. Dale. 2014. A web-portal for interactive data exploration, visualization, and hypothesis testing. *Frontiers in Neuroinformatics* 8 (2014), 25.

Anna Beletska, Wlodzimierz Bielecki, Albert Cohen, Marek Palkowski, and Krzysztof Siedlecki. 2011. Coarse-grained loop parallelization: Iteration space slicing vs affine transformations. *Parallel Computing* 37, 8 (Aug. 2011), 479–497.

Arthur J. Bernstein. 1966. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers,* EC-15, 5 (Oct. 1966), 757–763. DOI:http://dx.doi.org/10.1109/PGEC.1966.264565

Ganesh Bikshandi, Jia Guo, Daniel Hoeflinger, Gheorghe Almasi, Basilio B. Fraguela, María J. Garzarán, David Padua, and Christoph von Praun. 2006. Programming for parallelism and locality with hierarchically tiled arrays. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06)*. New York, NY, USA, 48–57. DOI:http://dx.doi.org/10.1145/1122971.1122981

Simone Campanoni, Kevin Brownell, Svilen Kanev, Timothy M. Jones, Gu-Yeon Wei, and David Brooks. 2014. HELIX-RC: An architecture-compiler Co-design for automatic parallelization of irregular programs. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA'14)*. IEEE Press, 217–228. http://dl.acm.org/citation.cfm?id=2665671.2665705

Simone Campanoni, Timothy Jones, Glenn Holloway, Vijay Janapa Reddi, Gu-Yeon Wei, and David Brooks. 2012. HELIX: Automatic parallelization of irregular programs for chip multiprocessing. In *Proceedings of the 10th International Symposium on Code Generation and Optimization (CGO'12)*. ACM, New York, NY, 84–93. DOI:http://dx.doi.org/10.1145/2259016.2259028

Stefano Cazzulani. 2012. Octane: The javascript benchmark suite for the modern web. Retrieved December 21, 2015 from https://developers.google.com/octane/.

Myeongjin Cho, Seon Wook Kim, and Youngsun Han. 2014. Web-based image processing using JavaScript and WebCL. In *2014 IEEE International Conference on Consumer Electronics (ICCE)*. 337–338. DOI:http://dx.doi.org/10.1109/ICCE.2014.6776030

Marc A. de Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. 2012. Static analysis and compiler design for idempotent processing. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*. ACM, New York, NY, 475–486. DOI:http://dx.doi.org/10.1145/2254064.2254120

Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. 1998. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI'98)*. ACM, New York, NY, 106–117. DOI:http://dx.doi.org/10.1145/277650.277670

Stephen J. Fink and Feng Qian. 2003. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (CGO'03)*. IEEE Computer Society, 241–252.

David Flanagan. 2006. *JavaScript: The definitive guide*. O'Reilly Media.

Google Inc. 2012. Google V8 JavaScript Engine. Retrieved November 14, 2015, from https://code.google.com/p/v8/.

Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly - Performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters* 22, 04 (2012), 1250010.

Stephan Herhut, Richard L. Hudson, Tatiana Shpeisman, and Jaswanth Sreeram. 2013. River trail: A path to parallelism in JavaScript. *ACM SIGPLAN Notices* 48, 10 (Oct. 2013), 729–744. DOI:http://dx.doi.org/10.1145/2544173.2509516

Ian Hickson. 2014. Web workers. *W3C Working Draft*, May 19, 2014. Retrieved from http://dev.w3.org/html5/workers/.

Ian Hickson and David Hyatt. 2011. HTML5: A vocabulary and associated APIs for HTML and XHTML. *W3C Working Draft,* May 25, 2011.

Michael Hind. 2001. Pointer analysis: Haven't we solved this problem yet?. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*. ACM, New York, NY, 54–61. DOI:http://dx.doi.org/10.1145/379605.379665

Ken Kennedy and John R. Allen. 2002. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers, San Francisco, CA.

Hanjun Kim, Nick P. Johnson, Jae W. Lee, Scott A. Mahlke, and David I. August. 2012. Automatic speculative doall for clusters. In *Proceedings of the 10th International Symposium on Code Generation and Optimization (CGO'12)*. ACM, New York, NY, 94–103. DOI:http://dx.doi.org/10.1145/2259016.2259029

Seon Wook Kim, Chong-Liang Ooi, Rudolf Eigenmann, Babak Falsafi, and T. N. Vijaykumar. 2006. Exploiting reference idempotency to reduce speculative storage overflow. *ACM Transactions on Programming Languages and Systems* 28, 5 (Sept. 2006), 942–965. DOI:http://dx.doi.org/10.1145/1152649.1152653

Richard Koo and Sam Toueg. 1986. Checkpointing and rollback-recovery for distributed systems. In *Proceedings of 1986 ACM Fall Joint Computer Conference (ACM'86)*. IEEE Computer Society Press, Los Alamitos, CA, 1150–1158.

Aparna Kotha, Timothy Creech, and Rajeev Barua. 2013. AESOP: The autoparallelizing compiler for shared memory computers. Retrieved November 14, 2015, from http://aesop.ece.umd.edu/doc/aesop-white-paper.pdf.

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (CGO'04)*. Washington, DC, 75.

Chris Marrin. 2011. Webgl specification. *Khronos WebGL Working Group.*

Jan Kasper Martinsen, Håkan Grahn, and Anders Isberg. 2013. Using speculation to enhance JavaScript performance in web applications. *IEEE Internet Computing* 17, 2 (March 2013), 10–19. DOI:http://dx.doi.org/10.1109/MIC.2012.146

Jan Kasper Martinsen, Håkan Grahn, and Anders Isberg. 2015. The effects of parameter tuning in software thread-level speculation in JavaScript engines. *ACM Transactions on Architecture and Code Optimization (TACO)* 11, 4, Article 46 (Jan. 2015), 25 pages. DOI:http://dx.doi.org/10.1145/2686036

Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu, and Scott Mahlke. 2009. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*. ACM, New York, NY, 166–176. DOI:http://dx.doi.org/10.1145/1542476.1542495

Mojtaba Mehrara, Po-Chun Hsu, Mehrzad Samadi, and Scott Mahlke. 2011. Dynamic parallelization of JavaScript applications using an ultra-lightweight speculation mechanism. In *Proceedings of the 17th International Symposium on High Performance Computer Architecture (HPCA'11)*. Washington, DC, 87–98.

Jaikrishnan Menon, Marc De Kruijf, and Karthikeyan Sankaralingam. 2012. iGPU: Exception support and speculative execution on GPUs. *SIGARCH Computer Architecture News* 40, 3 (June 2012), 72–83. DOI:http://dx.doi.org/10.1145/2366231.2337168

Mozilla Developer Network. 2015. SpiderMonkey. Retrieved November 14, 2015, from https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/.

Chong-Liang Ooi, Seon Wook Kim, Il Park, Rudolf Eigenmann, Babak Falsafi, and T. N. Vijaykumar. 2001. Multiplex: Unifying conventional and speculative thread-level parallelism on a chip multiprocessor. In *Proceedings of the 15th International Conference on Supercomputing (ICS'01)*. ACM, New York, NY, 368–380. DOI:http://dx.doi.org/10.1145/377792.377863.

Filip Pizlo. 2014. Introducing the WebKit FTL JIT. Retrieved from https://www.webkit.org/blog/3362/introducing-the-webkit-ftl-jit/.

Gustavo M. Porcides, Luiz A. P. Neves, Luiz C. M. de Aquino, and Gilson A. Giraldi. 2011. An on-line system for medical and biological image sharing. *Computational Vision and Medical Image Processing: (VipIMAGE'11),* 23.

Qualcomm Inc. 2015. Qualcomm Snapdragon. Retrieved from https://www.qualcomm.com/products/snapdragon.

Arun Raman, Hanjun Kim, Thomas R. Mason, Thomas B. Jablin, and David I. August. 2010. Speculative parallelization using software multi-threaded transactions. In *Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, 65–76. DOI:http://dx.doi.org/10.1145/1736020.1736030

Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin G Zorn. 2010. JSMeter: Comparing the behavior of javascript benchmarks with real web applications. In *Proceedings of the 2010 USENIX Conference on Web Application Development (WebApps'10)*. Berkeley, CA, 3.

Lawrence Rauchwerger and David Padua. 1994. The privatizing DOALL test: A run-time technique for DOALL loop identification and array privatization. In *Proceedings of the 8th International Conference on Supercomputing (ICS'94)*. ACM, New York, NY, 33–43. DOI:http://dx.doi.org/10.1145/181181.181254

L. Rauchwerger and D. A. Padua. 1999. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems* 10, 2 (Feb. 1999), 160–180. DOI:http://dx.doi.org/10.1109/71.752782

Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. 2010a. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10)*. New York, NY, 1–12. DOI:http://dx.doi.org/10.1145/1806596.1806598

M. Richards, J. Maloney, M. Wolczko, T. Wu, and A. Burmister. 2010b. Google, V8 benchmark suite-version 6. (2010). Retrieved December 21, 2015 from https://v8.googlecode.com/svn/data/benchmarks/v6/.

Jacob Seidelin. 2014. Pixastic JavaScript Image Processing Library. Retrieved November 14, 2015, from https://github.com/jseidelin/pixastic/.

Maciej Stachowiak. 2007. Announcing SunSpider 0.9. (2007). Retrieved December 21, 2015 from https://webkit.org/blog/152/announcing-sunspider-09.

Michael E. Wolf and Monica S. Lam. 1991. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems* 2, 4 (Oct. 1991), 452–471. DOI:http://dx.doi.org/10.1109/71.97902

Tomofumi Yuki. 2014. Understanding PolyBench/C 3.2 kernels. In *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques (IMPACT'14)*. 1–5.

Received April 2015; revised October 2015; accepted November 2015