A Limit Study of JavaScript Parallelism

Emily Fortuna Owen Anderson Luis Ceze Susan Eggers

Computer Science and Engineering, University of Washington
{fortuna, owen, luisceze, eggers}@cs.washington.edu
http://sampa.cs.washington.edu

***Abstract*—JavaScript is ubiquitous on the web. At the same time, the language's dynamic behavior makes optimizations challenging, leading to poor performance. In this paper we conduct a limit study on the potential parallelism of JavaScript applications, including popular web pages and standard JavaScript benchmarks. We examine dependency types and looping behavior to better understand the potential for JavaScript parallelization. Our results show that the potential speedup is very encouraging—averaging 8.9x and as high as 45.5x. Parallelizing functions themselves, rather than just loop bodies proves to be more fruitful in increasing JavaScript execution speed. The results also indicate in our JavaScript engine, most of the dependencies manifest via virtual registers rather than hash table lookups.**

I. Introduction

In an increasingly online world, JavaScript is around every corner. It's on your search engine and your new word processing software [1]. It's on your smart phone browsing the web and possibly running the applications on your phone [2]. One company has estimated that 99.6% of sites online today use JavaScript [3]. Dubbed the "Assembly Language of the Internet," JavaScript is used not only for writing applications, but also as the target language for web applications written in other languages, such as Java [4], [5], [6].

At the same time, the very features that have made this dynamically-typed language so flexible and popular make it challenging for compiler writers to efficiently optimize its execution. A number of websites offer tips and services to both manually and automatically optimize JavaScript code for download time and execution performance [7], [8], [9]. Further, many web-based companies such as Google, go as far as optimizing versions of their websites specifically for mobile devices.

These JavaScript optimizations are of particular importance for online mobile devices, where power is rapidly becoming the limiting factor in the performance equation. In order to reduce power consumption, mobile devices are following in the footsteps of their desktop counterparts by replacing a single, higher-power processor with multiple lower-power cores [10]. At the same time, advances in screen technology are reducing power consumption in the display [11], leaving actual computation as a larger and larger proportion of total power usage for mobile devices.

There have been a number of efforts to increase the speed of JavaScript and browsers in general [12], [13], [14]. More recently, researchers have sought to characterize the general behavior of JavaScript from the language perspective in order to write better interpreters and Just-In-Time (JIT) compilers [15], [16], [17]. Several of these studies compared the behavior between JavaScript on the web to standard JavaScript benchmarks such as SunSpider and V8. Researchers unanimously concluded that the behavior of JavaScript on the web differs significantly from that of standard JavaScript benchmarks.

This paper addresses the slow computation issue by exploring the potential of parallelizing JavaScript applications. In addition to the obvious program speedups, parallelization can lead to improvements in power usage, battery life in mobile devices, and web page responsiveness. Moreover, improved performance will allow future web developers to consider JavaScript as a platform for more compute-intensive applications.

Contributions and Summary of Findings

We present the first limit study to our knowledge of parallelism within serial JavaScript applications. We analyze a variety of applications, including several from Alexa's top 500 websites [18], compute-intensive JavaScript programs, such as a fluid dynamics simulator, and the V8 benchmarks [19]. In contrast to prior JavaScript behavior studies [15], [16], [17], we take a lower-level, data-dependence-driven approach, with an eye toward potential parallelism.

As a limit study of parallelism, we focused on two fundamental limitations of parallelism: data dependences and most control dependences (via task formation; see Section III-A). We impose JavaScript-specific restrictions, such as the event-based nature of execution. We believe this model captures all the first order effects on parallelism.

Our results show that JavaScript exhibits great potential for parallelization—speedups over sequential execution are 8.9x on average and as high as 45.5x. Unlike high-performance computing applications and other successfully parallelized programs, however, JavaScript applications currently do not have significant levels of loop-level parallelism. Instead, better parallelization opportunities arise between and within functions. As a result, current JavaScript applications will require different parallelization strategies.

The vast majority of data dependencies manifest themselves in the form of virtual registers, rather than hash-table lookups, which is promising for employing static analyses to detect parallelism. Finally, our results demonstrate that parallel behavior in JavaScript benchmarks differs from that of real websites, although not to the extent found for other JavaScript behaviors measured in [15], [16].

The remainder of this paper explains how we reached these conclusions. Section II provides some background information on the JavaScript language. Section III describes the model we used for calculating JavaScript parallelism, special considerations in measuring parallelism in JavaScript, and how we performed our measurements. Next, Section IV details our findings with respect to JavaScript behavior and potential speedup. Section V describes related work and its relationship to our study, and Section VI concludes.

II. Background

JavaScript is a dynamically typed, prototype-based, object-oriented language. It can be both interpreted and JIT'ed by JavaScript engines embedded in browsers, such as SpiderMonkey in Firefox [20], SquirrelFish Extreme[1] in Safari [21], and V8 in Chrome [22]. The current standard for JavaScript does not directly support concurrency.

Conceptually, JavaScript objects and their properties are implemented as hash tables. Getting, setting, or adding a property to an object requires the interpreter or JIT to either look up or modify that particular object's hash table.

The majority of production interpreters[2] convert JavaScript syntax into an intermediate representation of JavaScript opcodes and use one of two main methods to manipulate the opcodes' operands and intermediate values: stacks or registers [23]. Stack-based interpreters contain a data stack (separate from the call stack) for passing operands between opcodes as they are executed. Register-based interpreters contain an array of "register slots" (that we call "virtual registers," which are higher-level than actual machine registers) that are allocated as needed to pass around operands and intermediate data. Whether an interpreter is stack-based or register-based does not affect the way that data dependencies manifest; these two methods are simply used for storing the operands for each opcode. Interpreters such as SpiderMonkey are stack-based, whereas SquirrelFish Extreme is register-based. Some research suggests that register-based interpreters can slightly outperform stack-based interpreters [24].

Memory in a JavaScript program consists of heap-allocated objects created by the program, as well as a pre-existent Document Object Model produced by the browser that represents the web page and the current execution environment. All of these are distinct from local variables and temporaries. In this study we examine a register-based interpreter, which therefore exposes dependencies via virtual registers for local variables and temporaries, and via hash table lookups when accessing an object's hash table.[3]

[1]Also known by its marketing name, Nitro.

[2]The notable exception is V8, which compiles JavaScript directly to assembly code.

[3]In this study, register slots that directly reference objects for hash table lookups we classify as hash table lookups, distinct from local variables and temporaries stored in the register slots that we classify as being stored in "virtual registers."

A Day in the Life of a JavaScript Program

For the most part, JavaScript is used on the web in a highly interactive, event-based manner. A JavaScript event is any action that is detectable by the JavaScript program [25], whether initiated by a user or automatically. For example, when a user loads a web page, an `onLoad` event occurs. This can trigger the execution of a programmer-defined function or a series of functions, such as setting or checking browser cookies. After these functions have completed, the browser waits for the next event before executing more JavaScript code.

Next, in our example a user might click on an HTML `p` element, causing an `onClick` event to fire. If the programmer intends additional text to appear after clicking on the `p` element, the JavaScript function must modify the Document Object Model (DOM), which contains all of the HTML elements in the page. The JavaScript function looks up the particular object representing the `p` element in the DOM and modifies the object's text via its hash table, causing the browser to display the modified page without requiring an entire page reload. Any local variables needed to complete this text insertion are stored in virtual registers.

III. Methodology

Because of JavaScript's event-based execution model described above, we must make special considerations to quantify the amount of potential speedup in JavaScript web applications. Below we explain some terminology for talking about JavaScript parallelism, how we set up our model (Section III-A), and how we made our measurements from the dynamic execution traces (Section III-B).

A. Event-driven Execution Model Considerations

For our study, we parallelize each event individually, as described in Section II by dividing them into *tasks* that can be potentially executed concurrently. We define a *task* as any consecutive stream of executed opcodes in an event, delimited either by a function call, a function return, an outermost[4] loop entrance, an outermost loop back-edge, or an outermost loop exit. A task consists of at least one JavaScript opcode and has no maximum size. Tasks are the finest granularity at which we consider parallelism; we do not parallelize instructions within tasks. Since the tasks are essentially control independent, we take into account control dependences indirectly.

Figure 1 illustrates a high-level representation of the tasks created from the execution of a simple JavaScript function. When executing function `f`, JavaScript code is converted into a dynamic trace of opcodes. Our offline analysis divides these opcodes into eight tasks, appearing in boxes on the right of Figure 1.

We define the *critical path* as the longest sequence of memory and/or virtual register dependencies between tasks that occurs *within a given event*. The critical path is the limiting factor that determines the overall execution time for an event; all other data-dependent sequences can run in parallel

[4]Not nested inside any surrounding loop.

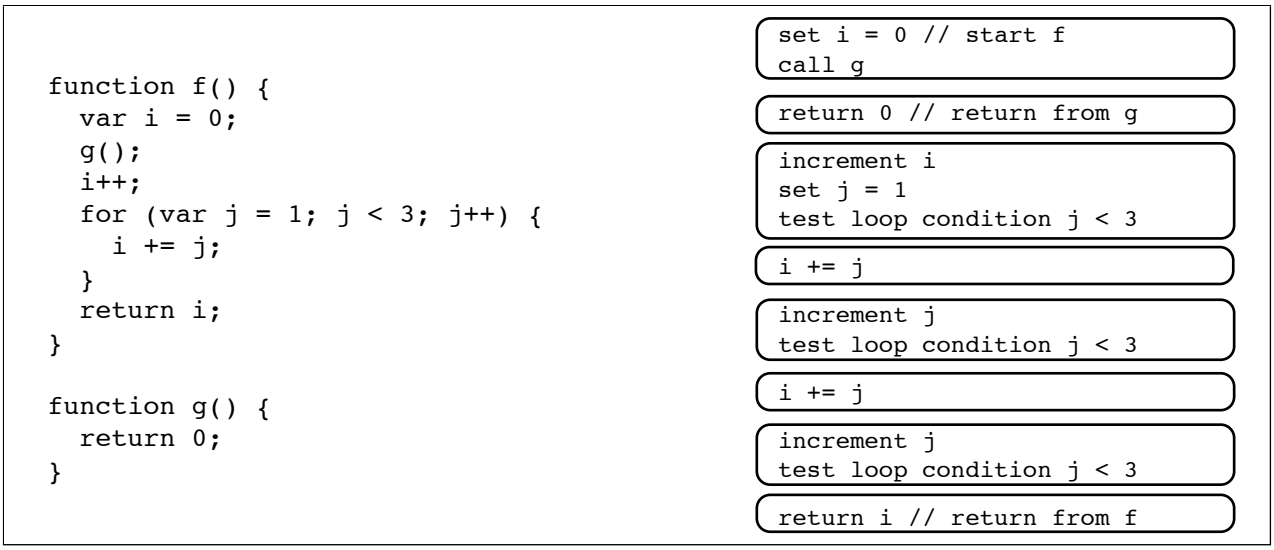

Fig. 1. An example JavaScript function `f` appears on the left. The right shows a high-level, pseudo-intermediate representation of the execution of `f`, to illustrate how tasks are delineated. Each box represents one task. The `i += j` tasks are the loop body tasks.

with the critical path. Since a typical JavaScript opcode in SquirrelFish executes in approximately 30 machine cycles, tasks rather than individual opcodes are a more realistic granularity for parallelization, given the overheads associated with parallelization. We use the procedure detailed in Algorithm 1 to calculate critical paths for the events observed in the dynamic execution traces.

Potential speedup from parallelization is measured by calculating the length of the critical path for each event that was observed in a dynamic execution trace, summing them together, and dividing into the total execution time, that is,

$$Speedup = \frac{T}{\sum_{i=0}^{n-1} c_i} \quad (1)$$

where n is the number of events, c_i is the length of the critical path for event i in cycles, and T is the total number of cycles for the entire execution. This optimistically assumes there is no limit in the number of processors available to execute independent JavaScript tasks.

We conservatively force events to execute in the order that they were observed in the original execution trace. If JavaScript were able to run in parallel, one could imagine a few JavaScript events occurring and running simultaneously, such as a timer firing at the same time as an `onClick` event generated by the user clicking on a link. However, from the interpreter's perspective, it is difficult to distinguish between true user events that must be executed sequentially and automatic events that could be parallelized, making it very challenging to know whether such events could logically be executed simultaneously. Therefore, we do not allow perturbations in *event* execution order.

Additionally, while interpreting some JavaScript functions, some built-in functions in the language make calls to the native runtime, and therefore we cannot observe these instructions executed from the interpreter's perspective. Examples of this function type include the JavaScript `substring`, `parseInt`, `eval`, and `alert` functions. We examine the potential speedup in two cases: the first conservatively assum-

Algorithm 1 Critical Path Calculation for Events

```
{build dependency graph}
for each opcode do
  for each argument in opcode do
    if argument is a write then
      record this opcode address as the most recent writer
    else if argument is a read then
      find last writer to the read location
      if read and last writer are in different tasks, but in
      the same event then
        mark dependency
        update longest path through this task, given the
        new dependency's longest path
      end if
    end if
  end for
  update longest path through this task with this processed
  opcode's cycle count
end for

{find critical path for each event}
for each task in tasks do
  event ← this task's event
  if the longest path through this task > the event's current
  critical path then
    event's critical path ← longest path through task
  end if
end for
```

ing that all calls to the runtime depend on the previous calls; and the second assuming no dependencies ever occur across the runtime. We take these elements into consideration in order to make our parallelism measurements as realistic as possible.

B. Experimental Setup

In order to measure the potential speedup of JavaScript on the web, we instrumented the April 1, 2010 version of the interpreter in SquirrelFish Extreme (WebKit's JavaScript engine, used in Safari and on the iPhone). We then used Safari with our instrumented interpreter to record dynamic execution traces of the JavaScript opcodes executed while interacting with the websites in our benchmark suite.

A growing number of recent studies have concluded that typical JavaScript benchmark suites, such as SunSpider and V8, are not representative of JavaScript behavior on the web [15], [26]. Therefore, we created our own benchmark suite, partly comprised of full interactions with a subset of the top 100 websites from the Alexa Top 500 Websites [18]. We supplemented our benchmark suite with a few other web applications that push the limits of JavaScript engines today, and the V8 version 2 benchmarks for reference. With the exception of the V8 benchmarks that simply run without user input, the length of logged interactions ranged from five to twenty minutes. Table I explains the details of our benchmark collection.

Traces were generated on a 2.4 GHz Intel Core 2 Duo running Mac OS 10.6.3 with 4 GB of RAM. Additionally, we used `RDTSC` instructions to access the Time Stamp Counter to count the number of cycles each JavaScript opcode took to execute. However, the cycle counts that we measured had several confounding factors, resulting in a variance of sometimes several orders of magnitude: the JavaScript interpreter was run in a browser, influencing caching behavior, along with network latency and congestion effects out of our control [27]. Therefore, out of the approximately 400M total dynamic JavaScript opcodes observed, for each type of opcode we selected the minimum cycle count observed and placed these values in a lookup table we used to calculate the critical paths for events. Additionally, if making repeated XML HTTP requests is the primary action of a JavaScript program, the perceived speedup may be less than our measurements because network latency may come to dominate the time that a user is waiting.

Benchmark	Classification	Details
ALEXA TOP 100 WEBSITES		
`bing`	search engine	two textual searches on bing.com and browsed images
`cnn`	news	read several stories on cnn.com
`facebook`	micro data	posted on wall, wrote on other's walls, expanded news feed
`flickr`	photo sharing	visited contact's pages, posted comments
`gmail`	productivity	archived emails, wrote and sent email
`google`	search engine	two textual searches and looked at images on google.com
`googleDocs`	productivity	edited and saved a spreadsheet, shared with others
`googleMaps`	visualization	found a driving route, switched to public transit route, zoomed in to map
`googleReader`	micro data	read several items, mark as unread, share item
`googleWave`	compute intensive	create new wave, add people, type to them, reply to wave
`nyTimes`	news	read several stories on nyTimes.com
`twitter`	micro data	posted tweet, clicked around home page for @replies and lists
`yahoo`	search	perform several searches and look at images
`youtube`	video	watch several videos, expand comments
COMPUTE INTENSIVE SUPPLEMENTAL BENCHMARKS		
`ballPool`	data visualization	chromeexperiments.com/detail/ball-pool/
`fluidSim`	compute intensive	fluid dynamics simulator in JavaScript at nerget.com/fluidSim/
`pacman`	compute intensive	NES emulator written in JavaScript playing Pacman at benfirshman.com/projects/jsnes/
V8 BENCHMARKS VERSION 2		
`v8-crypto`	V8	Encryption and decryption
`v8-deltablue`	V8	One-way constraint solver
`v8-earley-boyer`	V8	Classic Scheme benchmarks, translated to JavaScript by Scheme2Js compiler
`v8-raytrace`	V8	Ray tracer
`v8-richards`	V8	OS kernel simulation

TABLE I
PROVIDES THE DETAILS AND A GENERAL CLASSIFICATION FOR THE BENCHMARKS USED IN THIS STUDY.

IV. RESULTS

In this section, we first evaluate the overall degree of potential speedup. Then we explore other facets of the parallelization problem, including which types of tasks are most useful for maximizing speedup and what types of dependences manifest in typical JavaScript programs.

A. Total Parallelism

The graph in Figure 2 shows the potential speedup, conservatively assuming dependencies across calls to the native runtime, with both function call boundaries and outermost loop bodies as task delimiters. As previously mentioned, runtime calls are a set of built-in functions implemented natively rather than in JavaScript. To reiterate, we conservatively force events to execute in the same order that they were observed in the original execution trace, and we use Equation (1) to calculate the potential speedup. The potential speedups range from 2.19x and 2.31x for the `v8-crypto` and `google` benchmarks respectively, and up to 45.46x on `googleWave`,

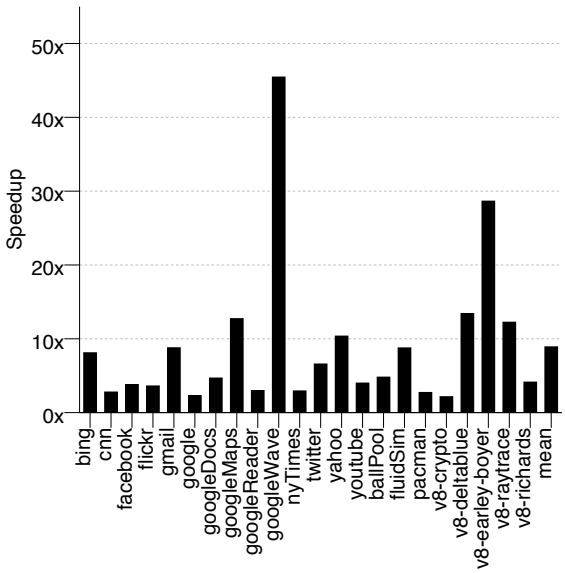

Fig. 2. Potential speedup with runtime dependencies, and function and loop task delimiters.

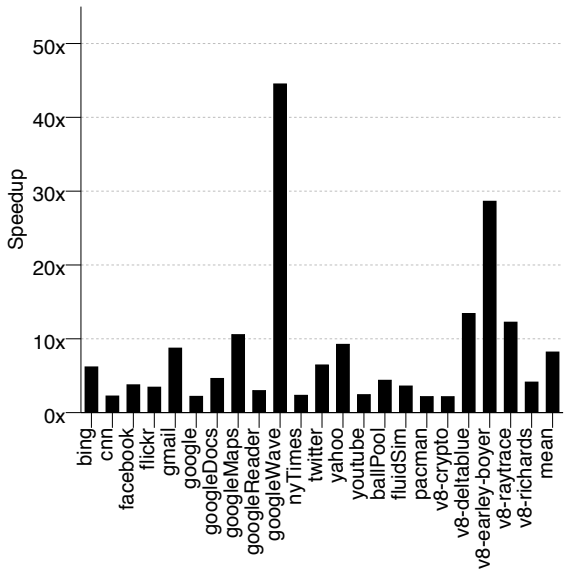

Fig. 3. The potential speedup for only function delimited tasks, with runtime dependencies.

with an average over all the benchmarks of 8.91x. If we exclude the minimum and maximum outliers (`googleWave` and `v8-crypto`, respectively), the adjusted (arithmetic) mean is 7.42x.

B. Task Granularity—Functions and Loops

Figures 3 and 4 illustrate the speedup measurements for function-delimited and outermost loop delimited tasks. We divided up the execution trace into tasks using the method in Figure 1 and discussed in Section III-A. When we break up tasks at loop boundaries, we do not include opcodes related to modifying the loop induction variable in the loop task. However, opcodes that use the loop induction variable to index into an array lookup in the loop body do cause a dependency between the loop task and its parent function task. If JavaScript had explicit support for parallel loop iterations, it is possible that some of these currently unparallelizable loop bodies, such as the array index example, could be rewritten in a way that would enable parallelization.

The graphs indicate that loops alone, in today's JavaScript web applications do not yet seem to be good candidates for automatic loop parallelization. The two leaders in speedup with only loop delimited tasks, `fluidSim`, and `ballPool` were the two most scientific-computing-like benchmarks in the benchmark suite. This does suggest that there may be benefits for loop parallelization in the future, should JavaScript performance improve to the point that developers are willing to write more complex and compute-intensive JavaScript programs.

The CDF in Figure 5 illustrates the iteration counts of each loop sequence in each benchmark, excluding the V8 benchmarks, which do not exhibit the same looping behavior as the "typical" web-site benchmarks. As an example, we count the code in Figure 1 as one distinct loop sequence executing for two iterations. If function f were called again, we would count the loop as a separate loop sequence. We find that 52% of loops in the suite iterated only one or two times. However, we can see a very long tail in the graph; a few loops were very hot—one loop in the `pacman` benchmark executed 1.4M times.

The graphs in Figures 6 and 7 show the frequency with which loops execute for different numbers of iterations in the `fluidSim` and `googleWave` benchmarks. We singled out these two benchmarks because `fluidSim` had the largest speedup with only loop delimited tasks, and `googleWave` had the largest overall speedup. The `googleWave` shows behavior typical of most of the benchmarks: a large number of short executing loops. `fluidSim` is unusual because the dramatic slope increase in the graph is bimodal; in this benchmark most loops execute only a few iterations, but then a large number of loop sequences also loop 4k times.

The two outliers for loop-only delimited tasks, `fluidSim` and `ballPool`, did not prove to have an appreciable difference in the *fraction* of loops executed relative to the entire program size compared to other benchmarks. In fact, we found that the fluid dynamics simulator actually had a *smaller* number of loops executed relative to its program size than a number of the less loop-parallelizable benchmarks. Instead, the average loop body (task) sizes were among the very largest observed in all the benchmarks. This suggests that `fluidSim` and `ballPool` have more long, independent loops, rather than a larger proportion of loop iterations. Although other benchmarks had smaller loop bodies on average, such loop bodies were not often independent from each other.

Task Size. Additionally, we examined the average task size to ensure that our level of granularity is realistic for making

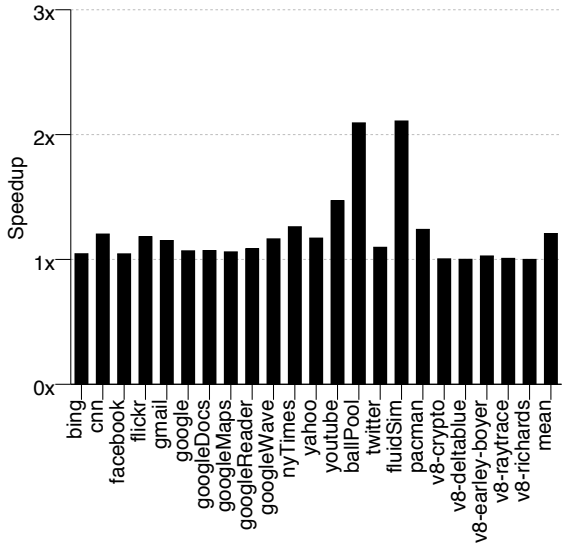

Fig. 4. The potential speedup for only loop delimited tasks, with runtime dependencies.

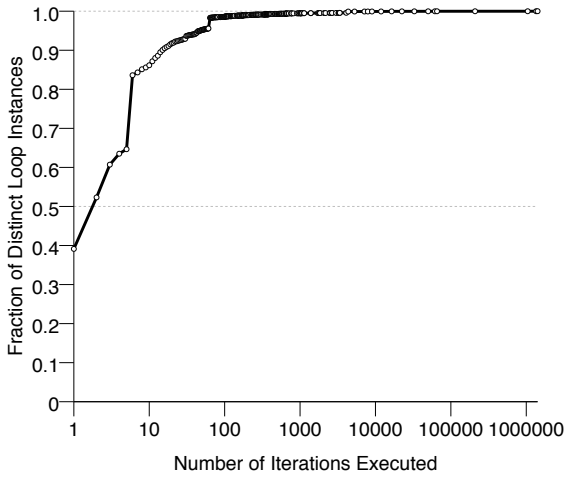

Fig. 5. CDF showing the number of iterations that each distinct loop instance took, incorporating data from all the benchmarks except for the V8 benchmarks. About half of the suite's loop sequences iterate only once or twice.

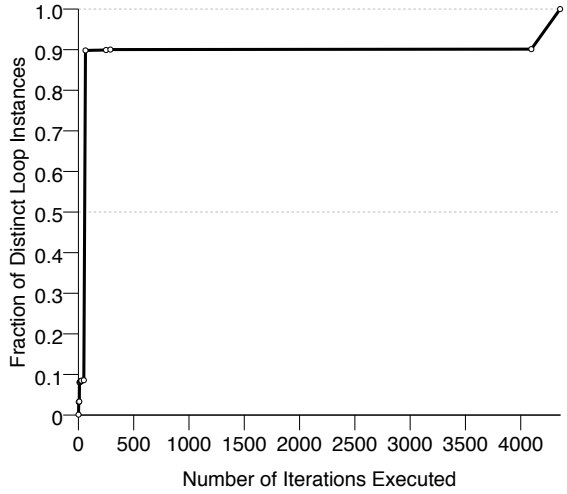

Fig. 6. CDF showing the frequency of loop instances executing for a given number of iterations in the `fluidSim` benchmark. This benchmark has a second large increase in slope in the right side of the graph, indicating a large number of loop instances with a high number of iterations (4k iterations).

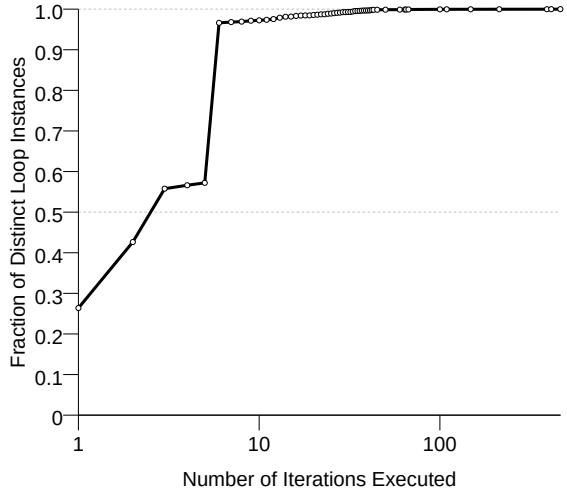

Fig. 7. CDF showing the frequency of loop instances executing for a given number of iterations throughout the `googleWave` benchmark. In this benchmark, the hottest loop instance executes 464 times. The shape of this CDF is very representative of most JavaScript benchmarks.

parallelization worthwhile. As shown in Figure 8, the average task size in most benchmarks is around 17 SquirrelFish opcode instructions, or 650 cycles, with a few outliers with much larger task sizes, such as `v8-crypto` (137 opcodes) and `fluidSim` (73 opcodes). Because task sizes were this short, we see little reason to examine parallelism at an even finer granularity. However, 17 opcodes, or 650 cycles is still quite a fine granularity — we chose this granularity due to the limit study nature of this work, and moreover, tasks can be combined where appropriate during an implementation to better amortize parallelization overheads.

C. Dependences

Below, we further explore the types of dependencies, average dependency lengths, and our assumptions about runtime dependencies in JavaScript. Figure 9 indicates that the majority of dependencies arise from local variables in virtual registers. At least 84% of the dependencies of the web page-based benchmarks (as opposed to the V8 benchmarks) were register-based, which provides promise for static analysis and potentially some automatic and even non-speculative parallelization. Many of the V8 benchmarks, in contrast, had a much higher incidence of memory dependencies.

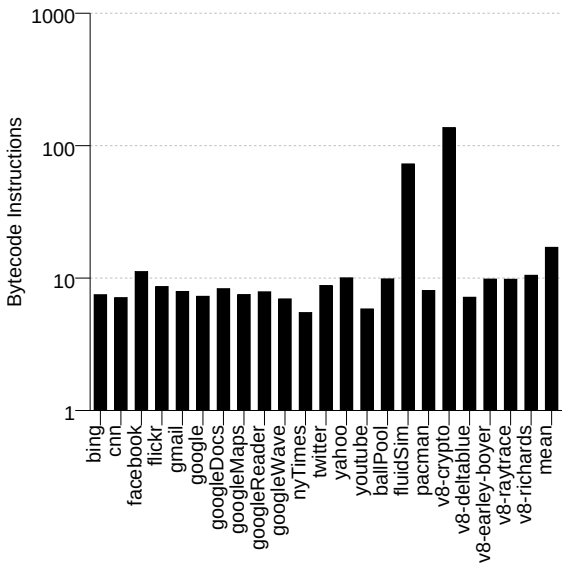

Fig. 8. The average task size of benchmarks by SquirrelFish opcode count, on a log scale.

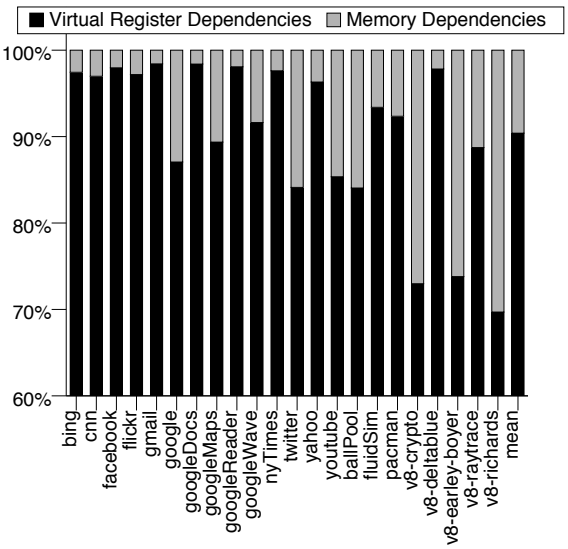

Fig. 9. Percentage of all dependencies that are virtual register to register dependencies *between* tasks versus hash-table lookups between tasks.

Dependency distance refers to the number of cycles (or instructions) between a write to a virtual register or hash table and its subsequent read. Figure 10 shows the average hash table and virtual register dependency distances by JavaScript opcode count. We see that in general, virtual register dependency lengths are shorter than the lengths for memory dependences (average of 50k opcodes for virtual register dependencies versus 160k for hash table lookups). The difference is unsurprising because virtual registers house local and temporary variables, whereas storing data into objects

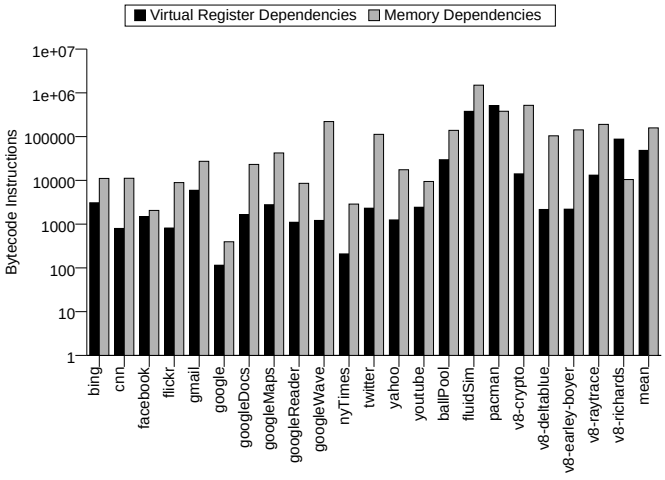

Fig. 10. Average memory and register dependency distances *between* tasks by SquirrelFish opcode count, on a log scale.

that may be referenced from function to function manifest as memory dependences. The graph indicates that the average dependency distance for both virtual registers and hash table lookups is fairly high. Large dependency distances may provide more flexibility for an automatic parallelization, since there is slack to exploit. Additionally, because the memory dependence distance is large, we can more easily amortize the cost of dependence checks.

We find that our more conservative assumption of adding "fake" dependencies[5] into and out of runtime calls did not strongly affect our parallelism measures. Figure 11 shows that although the potential speedups increased slightly from 2.31 to 2.40 for `google` and from 45.46 to 45.89 for `googleWave` without the assumption of runtime dependencies, these numbers are not significantly larger from their more conservative counterparts. This is also encouraging because tracking dependencies via the internal state of the runtime system may be expensive; we can use conservative assumptions without hurting parallelism much.

D. Case Study: googleWave

Finally, we inspected the maximum speedup outlier, `googleWave`, in more detail to better understand what makes the program so parallelizable. We first graphed a Cumulative Distribution Function (CDF) illustrating the frequency of speedup levels of for all *events*, weighted by serial event length, in Figure 12. We see that 75% of all events weighted by execution time have a speedup of 45x or higher, and 50% of all events have a potential speedup of 90x or higher. If we create a similar CDF without weighting events by execution time, we find that 46% of all events have a speedup of 1x and therefore cannot be parallelized. This data indicates that `googleWave` has a large number of very short events that

[5]Since we could not track dependencies called in native code.

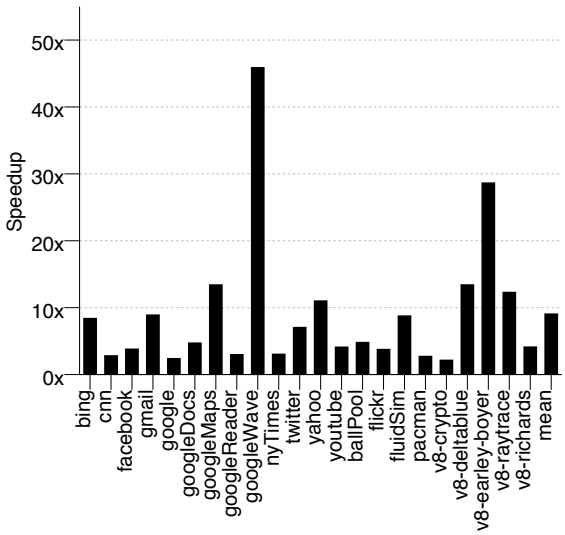

Fig. 11. Potential speedup *without* runtime dependencies, but with function and loop task delimiters

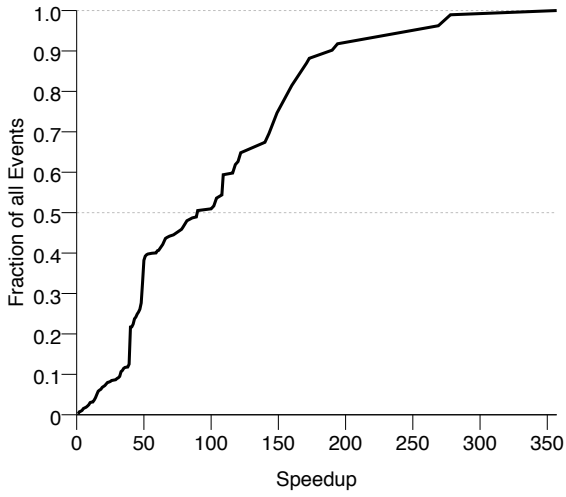

Fig. 12. CDF illustrating the fraction of events that that have a given amount of speedup in the `googleWave` benchmark, weighted by the length of the events. 50% of all events, weighted by event length, have a speedup between 90x and 357x.

will not benefit from parallelization, but a significant number of long running events that are very parallelizable.

We then created a CDF of the size of tasks over the entire program. In Figure 13 we find that 52% of the tasks contain 6 opcodes (180 cycles) or more, and it has a very long tail; the largest task contains 10k opcodes. Today, most web developers are limited by performance reasons from producing applications as JavaScript intensive as `googleWave`.

It should also be noted that `googleWave` is the only benchmark we investigated written using the Google Web Toolkit (GWT). As mentioned in the introduction, this tool compiles Java into optimized JavaScript. It is possible that the

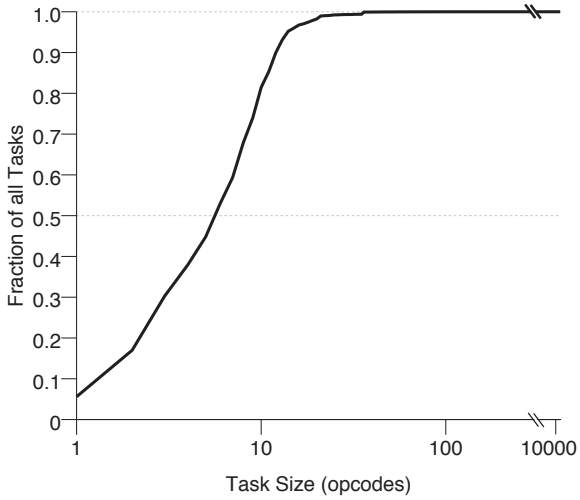

Fig. 13. CDF of task size (in opcodes) in Google Wave. We find over forty percent of all tasks observed in Google Wave are 10 opcodes or more. The largest task in this benchmark contains 10096 opcodes.

coding style of Java lends itself to more short function calls (such as accessor methods in Java) than in JavaScript (which might instead use a hashtable lookup), or GWT automatically generates more parallelizable JavaScript.

Although `googleWave` is something of an outlier amongst other current uses of JavaScript, we believe that in the future we will see a greater number of applications with similar characteristics and similar parallelism opportunities to `googleWave`. There has been a recent trend toward browser-based applications (such as Google Docs, Microsoft's new web-based Office applications, and Chrome OS). To provide a desktop-like experience on the web, such applications would benefit from multi-threading and more performance that could be gained from parallelization.

V. Related Work

JavaScript and its behavior on the web has only recently become a more widely explored topic. Several researchers have explored the behavior of JavaScript from a higher-level language perspective.

Lebresne et al. examined JavaScript objects, how frequently their properties and data are modified, and the length of their prototype chains in the SunSpider benchmarks and a small set of real web pages [17]. This higher-level analysis focused on objects and their modification in order to ultimately provide a type system for JavaScript. Their analysis indicates that the average lengths of object prototype chains across the benchmarks (and also maximum chain lengths for websites) were relatively similar and short, supporting our decision to select the minimum observed cycle value for `op_put_by_val` and `op_get_by_val`.

Ratanaworabhan et al. very extensively measured the usage of JavaScript on the web compared to the V8 and SunSpider benchmarks by instrumenting Internet Explorer 8 and char-

acterizing the behavior of functions, heap-allocated objects and data, and events and handlers [15]. They conclude that the behavior of the "standard" benchmarks is significantly different from actual web pages. Their web page traces were generally based on short web page interactions (*e.g.*, sign in, do one small task, sign out) in comparison to our study's longer running logs. Their results corroborate ours in that in web pages there are a small number of longer-running "hot" functions (that may be worthy of parallelism).

Richards et al. also performed a more extensive analysis of the dynamic behavior of JavaScript usage on the web and compared it to that of standard industry JavaScript benchmarks as well as the conventional wisdom of how JavaScript is used [26]. Whereas Ratanaworabhan et al. used short web interactions, the Richards et al. study recorded longer running traces when other users used the instrumented browser as their default browser. The study investigated, among many things, the frequency of `eval`, the frequency that properties were added to an object after initialization, the variance of the prototype hierarchy, and the frequency of variadic functions. They concluded that the behavior on the web and usage of JavaScript language constructs differed significantly from the prevailing opinion as well as that of the standard benchmarks. While our study analyzes the dynamic behavior of JavaScript on the web and benchmarks arising from low level data dependencies in JavaScript opcodes in order to characterize the potential speedup through parallelization, Richards et al. focuses the analysis primarily at the higher level syntactic layer of JavaScript, with a nod to applications of their results to type systems.

Meyerovich and Bodík find that at least 40% of Safari's time is spent in page layout tasks [16]. They also break down the length of time for several websites spent on various tasks during page loads, of which JavaScript is a relatively small component (15%-20% of page load time). Given these results, they detail several algorithms to enable fast web page layout in parallel. However, *after* page load time, the proportion of time spent executing JavaScript and making XML HTTP requests increases. Indeed, if Meyerovich and Bodík succeed in dramatically reducing the amount of time spent in page loads, then the time spent after page loads, and therefore in JavaScript would increasingly dominate the execution time.

Mickens et al. developed the Crom framework in JavaScript that allows developers to hide latency on web pages by speculatively executing and fetching content that will then be readily available should it be needed [28]. Developers mark specific JavaScript event handlers as speculable, and then the system fetches data speculatively while waiting on the user, reducing the perceived latency of the page.

Meanwhile, some work has been done on the compiler end to improve the execution of JavaScript. Gal et al. implemented TraceMonkey, to identify hot *traces* for the Just-In-Time (JIT) compiler, as opposed to the standard method-based JIT [14]. However, they found that a trace must be executed at least 270 times before they break even from the additional JIT'ing of traces that they do during execution. As we saw in the benchmarks in our study, half of all loops executed in JavaScript programs only execute at most for two iterations. Ha et al. alleviated the performance problem in [14] by building another trace-based Just-In-Time compiler that does its hot trace compilation work in a separate thread so that code execution need not halt to compile a code trace [13]. Martinsen and Grahn have investigated implementing thread level speculation for scripting languages, particularly, JavaScript [12]. Their algorithm achieves near linear speedup for their for-loop intensive test programs. However, as we found in our study, JavaScript as currently used on the web is not written in such a way that its iterations are independent.

VI. Conclusion

Our study is the first exploration of the limits of parallelism within JavaScript applications. We used a collection of traces from popular websites and the V8 benchmarks to observe the data and user-limited dependences in JavaScript. In summary, we found:

a) JavaScript programs on the web hold promise for parallelization: Even without allowing parallelization between JavaScript events, the speedups ranged from 2.19x up to 45.46x, with an average of 8.91x. These numbers are great news for automatic parallelization efforts.

b) Programs are best parallelized within or between functions, not between loop iterations: Most JavaScript loops on the web today do not iterate frequently enough, and iterations are generally not independent enough to be parallelized. Instead, parallelization opportunities arise between tasks formed based on functions. These tasks are large enough to amortize reasonable overheads.

c) The vast majority of data dependences come from virtual registers, and the length of these dependences are shorter than hash-table lookup dependences: This finding makes static analysis conceivable and argues that disambiguation for parallelization can likely be made low cost.

These results suggest that JavaScript holds significant promise for parallelization, reducing execution time and improving responsiveness. Parallelizing JavaScript would enable us to make better use of lower-power multi-core processors on mobile devices, improving power usage and battery life. Additionally, faster JavaScript execution may enable future programmers to write more compute intensive, richer applications.

Acknowledgments

The authors would like to thank Jeff Kilpatrick, Benjamin Lerner, Brian Burg, members of the UW sampa group, and the `#squirrelfish` IRC channel for their helpful discussions.

References

[1] A. Vance, "Revamped Microsoft Office Will Be Free on the Web," *New York Times*, May 11, 2010. [Online]. Available: http://www.nytimes.com/2010/05/12/technology/12soft.html

[2] M. Allen. (2009) Overview of webOS. Palm, Inc. [Online]. Available: http://developer.palm.com/index.php?option=com_content\&view=article\&id=1761\&Itemid=42

[3] (2010, Jun) JavaScript Usage Statistics. BuiltWith. [Online]. Available: http://trends.builtwith.com/javascript

[4] (2010) Google Web Toolkit. Google. [Online]. Available: http://code.google.com/webtoolkit/

[5] F. Loitsch and M. Serrano, "Hop Client-Side Compilation," in *Symposium on Trends on Functional Languages*, 2007.

[6] (2010, June) wave-protocol. [Online]. Available: http://code.google.com/p/wave-protocol/

[7] (2009) w3compiler. Port80 Software. [Online]. Available: http://www.w3compiler.com/

[8] G. Baker and E. Arvidsson. (2010) Let's Make the Web Faster. [Online]. Available: http://code.google.com/speed/articles/optimizing-javascript.html

[9] D. Crockford. (2003, Dec 4,) The JavaScript Minifier. [Online]. Available: http://www.crockford.com/javascript/jsmin.html

[10] (2010, June) Qualcomm Ships First Dual-CPU Snapdragon Chipset. [Online]. Available: http://www.qualcomm.com/news/releases/2010/06/01/qualcomm-ships-first-dual-cpu-snapdragon-chipset

[11] (2010) Mirasol Display Technology. [Online]. Available: http://www.qualcomm.com/products_services/consumer_electronics/displays/mirasol/index.html

[12] J. Martinsen and H. Grahn, "Thread-Level Speculation for Web Applications," in *Second Swedish Workshop on Multi-Core Computing*, Nov 2009.

[13] J. Ha, M. Haghighat, S. Cong, and K. McKinley, "A Concurrent Trace-based Just-in-Time Compiler for JavaScript," in *PESPMA '09: Workshop on Parallel Execution of Sequential Programs on Multicore Architectures*, Jun 2009.

[14] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz, "Trace-based Just-in-Time Type Specialization for Dynamic Languages," in *PLDI '09: Proceedings of the Conference on Programming Language Design and Implementation*, Jun 2009.

[15] P. Ratanaworabhan, B. Livshits, D. Simmons, and B. Zorn, "JSMeter: Characterizing Real-World Behavior of JavaScript Programs," Microsoft Research, Microsoft Research Technical Report 2009-173, Dec 2009. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=115687

[16] L. Meyerovich and R. Bodik, "Fast and Parallel Webpage Layout," in *WWW '10: Proceedings of the Conference on the World Wide Web*, Apr 2010.

[17] S. Lebresne, G. Richards, J. Östlund, T. Wrigstad, and J. Vitek, "Understanding the Dynamics of JavaScript," in *STOP '09: Proceedings for the Workshop on Script to Program Evolution*, Jul 2009.

[18] (2010, May) Top Sites: The Top 500 Sites on the Web. [Online]. Available: http://www.alexa.com/topsites

[19] (2010, June) V8 Benchmark Suite - version 2. [Online]. Available: http://v8.googlecode.com/svn/data/benchmarks/v2/run.html

[20] (2010, June) What Is SpiderMonkey? [Online]. Available: http://www.mozilla.org/js/spidermonkey/

[21] M. Stachowiak. (2008, September) Introducing SquirrelFish Extreme. [Online]. Available: http://webkit.org/blog/214/introducing-squirrelfish-extreme/

[22] (2010, June) V8 JavaScript Engine. [Online]. Available: http://code.google.com/p/v8/

[23] D. Mandelin. (2008, June) SquirrelFish. [Online]. Available: http://blog.mozilla.com/dmandelin/2008/06/03/squirrelfish/

[24] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg, "Virtual Machine Showdown: Stack Versus Registers," *Transactions on Architecture and Code Optimization*, vol. 4, no. 4, pp. 21:1–21:36, January 2008.

[25] (2010, June) JavaScript Events. [Online]. Available: http://www.w3schools.com/js/js_events.asp

[26] G. Richards, G. Lebresne, B. Burg, and J. Vitek, "An Analysis of the Dynamic Behavior of JavaScript Programs," in *PLDI '10: Proceedings of the Conference on Programming Language Design and Implementation*, Jun 2010.

[27] Microsoft, "Measuring Browser Performance: Understanding Issues in Benchmarking and Performance Analysis," pp. 1–14, 2009. [Online]. Available: http://www.microsoft.com/downloads/details.aspx?displaylang=en\&FamilyID=cd8932f3-b4be-4e0e-a73b-4a373d85146d

[28] J. Mickens, J. Elson, J. Howell, and J. Lorch, "Crom: Faster Web Browsing Using Speculative Execution," in *NSDI '10: Proceedings of Symposium on Networked Systems Design and Implementation*, Apr 2010.