Are web applications ready for parallelism?

Cosmin Radoi
University of Illinois
cos@illinois.edu

Stephan Herhut
Intel Corporation
stephan@herhut.eu

Jaswanth Sreeram
Intel Corporation
jaswanth.sreeram@intel.com

Danny Dig
Oregon State University
digd@eecs.oregonstate.edu

Abstract

In recent years, web applications have become pervasive. Their backbone is JavaScript, the only programming language supported by all major web browsers. Most browsers run on desktop or mobile devices with parallel hardware. However, JavaScript is by design sequential, and current web applications make little use of hardware parallelism. Are web applications ready to exploit parallel hardware?

To answer this question we take a two-step approach. First, we survey 174 web developers regarding the potential and challenges of using parallelism. Then, we study the performance and computation shape of a set of web applications that are representative for the emerging web. To this end, we developed an automated profiling and dependence analysis tool. Using the tool, we identify performance bottlenecks and examine memory access patterns to determine possible data parallelism.

Our findings indicate that emerging web applications do have latent data parallelism, and JavaScript developers' programming style are not a significant impediment to exploiting this parallelism.

1. Introduction

Parallel hardware has become a reality of modern computing and its use is no longer confined to high performance applications and super computing. Even mobile phones now regularly feature multi-core CPUs and programmable GPUs. SIMD (Single Instruction, Multiple Data) extensions add further to the mix of exploitable hardware parallelism. Creating the best possible experience on any device therefore requires tapping into parallel hardware's potential to increase performance, save energy, or even both.

Most traditional platforms and languages have developed tools and language extensions to help developers adapt their code to run on modern parallel hardware. Yet, HTML5, an emerging web-based application ecosystem that promises portability across devices and form factors, and its implementation language JavaScript, seem still to be stuck in the sequential past. While browser vendors have invested heavily into the sequential performance of their JavaScript engines and added some support for concurrency [11], support for parallelism is still in its infancy. Parallel JavaScript [27] and WebCL [12] are two proposals to extend JavaScript to support parallel programming but neither is widely used. While this can be attributed to their prototypical implementation, the question remains: *Are web applications ready for parallelism?*

Earlier work by Fortuna et al. [20] has found that typical web applications have potential for achieving significant speedup from concurrent execution. This is encouraging but most of the potential they found stems from independent tasks rather than loops. Therefore it would be hard to exploit it using massively data-parallel hardware like GPUs or SIMD. Even more, Richards et al. [31] have studied the runtime behavior of typical JavaScript applications and found wide spread use of dynamic language features, which hinder execution on restricted hardware like GPUs and SIMD units. Both findings suggest that, while there is some potential for task parallelism, the web is not a fertile ground for data parallel programming.

While this conclusion might be true for the web of the past, our hypothesis is that it does not apply to the *emerging* web of applications. With the shift of the web to an era of application centric usages like, for example, image editing, augmented reality applications and sophisticated gaming, the characteristics of executed code change, too. As these usages are more compute intense, they also are more likely to gain from data-parallel compute capabilities. Even more, due to the increased focus on application logic over just rendering content, we also expect other high-level code properties, like use of dynamic language features, to change. Lastly, a new generation of programmers might also bring different programming styles to the table, e.g., due to influences from more declarative programming patterns during their education.

Of course, measuring such a trend in its early phases is difficult. Most production-quality web sites are still built in a legacy style and new applications are only beginning to emerge. Analyzing currently-popular web sites would bias our results towards what works well on most platforms now, not the workloads that are missing precisely because they would require more performance. Thus, in contrast to earlier studies, we had no adequate top-100 list or similar to draw from. Instead, we chose to measure the change where it starts: with the shift in developers' opinion.

We have asked 174 web developers about their coding practice and about properties of the code they write. Furthermore, we have asked them to predict what the emerging, compute intense applications of the future web will be. As a general trend, we found that applications formerly at home on the desktop are predicted to transition to the web. With the flattening of per-core performance improvement, desktop applications have become increasingly parallel in the last few years. We expect that their web counterparts will also need to be parallel in order to be competitive.

To measure the first effects of that transition we, in a second step, do a case study of 12 workloads. We selected the workloads from the categories mentioned by the developers and analyzed them for latent data parallelism. In particular, we were interested in the presence of parallelizable loops and their approximate percentage of execution time. We also looked at further code properties, like use of dynamic language features and more declarative abstractions

like `map` and `reduce`. Our findings differ from earlier work and we found a surprisingly large quantity of compute intense loops of which many were latently parallel.

This sanity check of developers' opinions against real world code has furthermore revealed an interesting trend: while developers prefer abstract declarative code they still often opt for imperative solutions in practice.

In short, this paper contributes a study on latent parallelism in emerging web applications using:

- a survey of developer opinions on their coding practice and trends in future web applications. We found that JavaScript developers generally embrace the functional nature of the language, while generally avoiding some advanced features like polymorphic variables. They generally hold the view that more desktop applications will migrate towards the web in the future.
- a tool for finding and analyzing latent data parallelism in JavaScript loops.
- a case study of latent data parallelism in 12 emerging web applications. We found that many of them do have latent data parallelism.
- a discussion of the results of our survey and study and their implications for various audiences: library and tool developers and researchers, web browser engine developers, and JavaScript developers.

The remainder of the paper is structured as follows. Section 1.1 gives a brief overview on the particulars of HTML5 and current proposals for concurrency and parallelism. Next, section 2 discusses the design and methodology of our survey, and presents key results. We back up those results by findings from a set of case studies. Section 3 describes the methodology for the case studies, including the performance analysis tool. Section 4 presents the case studies findings. We put the results in context and discuss their implications in Section 5. Finally, Section 6 describes related work.

1.1 HTML5 and parallelism

HTML5 is the latest evolution in web technologies. It extends HTML and related standards by a range of new features, including 3D rendering, offline storage, device access and direct network communication, to name only few. With these additions, an HTML5 capable web browser turns into a rich, cross-platform and cross-device application platform [16]. At the heart of HTML5 applications lies JavaScript, the only programming language supported by all modern browsers. Aside from the name similarity, JavaScript has little in common with Java. Instead, it is a dynamically typed language that combines aspects of functional, imperative and object oriented languages. In contrast to Java, it does not support classes but relies on prototype-based inheritance, similar to the Self [33] language. Originally designed for light scripting tasks, JavaScript uses a mostly sequential, event based programming model.

More recently, different solutions have been proposed to add concurrency and parallelism to JavaScript. Web workers [11] bring memory-safe threads to the HTML5 eco-system. They implement the actor model [14], i.e., threads follow a share-nothing policy and communicate only via messages. Designed for long running background tasks, web workers also are typically rather heavy weight. In the context of parallel computing, two APIs have been proposed: WebCL and Parallel JavaScript.

WebCL [12] adds an interface for the C-like OpenCL [8] language to the HTML5 platform. By building on OpenCL, WebCL allows developers to make use of both CPUs and GPUs, albeit at the cost of a radically different programming model. OpenCL is closely related to C with its imperative programming model, reliance on pointers and explicit memory management. This gives the programmer full control of the underlying hardware but limits WebCL to C-like data structures and prevents a deeper integration with JavaScript's object model and heap management.

Parallel JavaScript (aka River Trail) [26, 27] stays closer to the existing JavaScript language. It adds a high-level data-parallel API built around operations like `map` and `reduce`. This removes much control over the hardware from the programmer but gives more control to the runtime and enables a deeper integration, including the use of JavaScript's object model and heap management.

Both WebCL and Parallel JavaScript enable execution of common JavaScript code on CPU and GPU. Due to hardware limitations, only a restricted subset of JavaScript is supported on GPUs.

2. JavaScript in practice : a survey

Previous work in characterizing usage of JavaScript [31] in practice has focussed on analyzing the most popular websites or analyzing benchmark suites. Our goal in this work is to understand both how JavaScript developers use the language and what they perceive as important trends. Their practice and opinions indicate whether parallelism is needed in JavaScript, and, if it is, which is the best way to achieve it.

We formulated a questionnaire consisting of 20 questions. The questions broadly fall into four categories: trends in web applications, programming style, preferred tools and frameworks, and perceived performance bottlenecks. There are both multiple choice and open-ended questions Several of the multiple-choice questions were followed by an open-ended question asking the developer to explain his choice.

We publicized this questionnaire using social media. We requested a few influential developers in the JavaScript community to tweet a link to this questionnaire. We also posted a link to this questionnaire to the JavaScript section of *reddit,* a popular social news website.

We received a total of 174 distinct responses to the questionnaire. To ensure that we obtain a representative sample of JavaScript developers, we intentionally did not target the developers of any particular company, but rather publicized the survey broadly. Also, from our demographics questions we learned that our respondents use a wide variety of libraries, IDEs, and compile-to-JavaScript languages, thus we believe our population is representative. We summarize our findings below, but the detailed question and answer reports are available publicly at `http://cos.github.io/js-ceres`.

2.1 Future trends in web applications

A principal goal of our study was to understand what JavaScript developers think the most popular web applications of the future will be. Previous works on characterizing real-world JavaScript applications have drawn on two sources: benchmark suites such as Sun Spider, Kraken and the V8 suite, and scripts served by the most popular websites [1]. Richards et al. [31] conclude that the popular benchmark suites are poor representations of real-world JavaScript programs along several metrics. We argue that the programs taken from the most popular websites are also ill-suited to our goal of understanding the future of web applications.

Firstly, the programs from the popular websites are required to support a diverse set of browsers and hardware which restricts both the functionality that they implement as well as the user experience they deliver. Many JavaScript and HTML5 features are, as of this work, still under consideration of the standards bodies. This means that browsers are not required to support them and moreover different browsers may support different subsets of the proposals. Mainstream web applications that must work on a variety of browsers therefore may not utilize these features.

Secondly, being usable across diverse platforms also means that websites have to be conservative about the client hardware. For

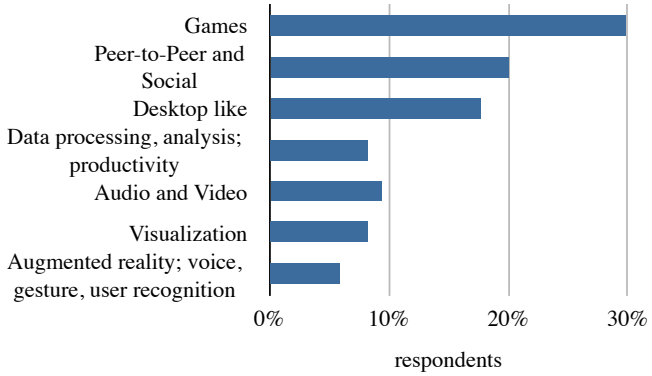

Figure 1. Future web application categories, as identified by respondents

this reason, features or user experiences that require a significant computational horsepower are uncommon. For these reasons we hypothesize that popular websites are typically not early adopters of emerging language and API features.

Our survey asked the developers: "In your opinion, what new kinds of applications will trend on the web over the next 5 years?". We hand-coded their answers using *qualitative thematic coding* [18]. We developed a set of codes that we validated by achieving an inter-rater agreement of over 80% for 20% of the data. Two coders, the second and the third authors, developed the categories which were not known a-priori. For measuring the agreement we used the Jaccard coefficient.

Figure 1 shows the resulting application categories. Many of the respondents mentioned web-based commercial-quality 3D games such as those available on modern desktop class machines or consoles. Client hardware found on even small form factor devices such as phones and tablets is rapidly becoming more powerful. In addition, APIs such as Canvas [5], Pointer lock [9] and touch enabling APIs [6] are being standardized and many recent versions of major browsers already support them. In particular, the Canvas element allows for fine-grained control over drawing and is a key enabler for cross-platform graphics in the browser without any third-party plugins. The performance of drawing operations on Canvas objects has also received considerable attention and has improved dramatically over browser generations. The WebGL API [13] allows executing shaders on client GPUs - a feature that has traditionally only been available to native games. Finally, the cross-platform portability and *access-anywhere* model of web applications means that these games can reach a wider audience. This leads us to expect HTML5 game engines to rapidly evolve from simple 2D views, primitive physics and gameplay to 3D or isometric 3D views, realistic physics [3] and game AI.

Games have traditionally been important drivers of evolution in consumer hardware. Modern native game engines make extensive use of parallel hardware to deliver quality gameplay experiences. For example, they use the increasingly sophisticated GPUs for realistic rendering and physics computations, they use multiple cores and vector instructions extensively for task level and SIMD parallelism. However, these platform capabilities are not available to web-based games engines today as browser engines do not expose parallel hardware to JavaScript programs (with the exception of shader programs written in WebGL). We argue that this restriction implies that web-based game applications will deliver lower quality user experiences unless there are programming models that appropriately expose the full spectrum of hardware parallelism to web applications in a fashion that preserves safety and programmability.

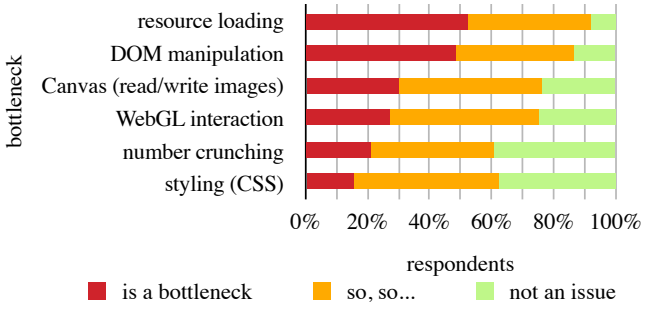

Figure 2. Performance bottlenecks importance as scaled by respondents

20% of the respondents have mentioned peer-to-peer and social applications, supporting that the current trend towards a more social web will continue.

Almost 20% of the respondents only mentioned desktop-like applications. While this is not a category per se, we have included this response to highlight this general trend. The other common responses to this question are related to audio and video processing, data visualization, data analysis and rich productivity suites, voice and gesture recognition, and augmented reality.

Overall, the answers indicate that a majority of our respondents expect future applications to be more computationally intensive, real-time, and interactive.

2.2 Performance bottlenecks in current web applications

With the increasing richness and functionality embedded into web applications, especially real-time interactive applications, understanding typical performance bottlenecks are important considerations for developers as well as engine implementors. For example, the rapid evolution of many aspects of JIT engines in major browsers is being driven by understanding bottlenecks in commonly used programs or benchmark suites.

Our survey asked the respondents to categorize each of several components as "not an issue", "so, so...", or "is a bottleneck". The aggregated responses are shown in Figure 2.

Confirming the common complaint in the JavaScript community, 53% and 48% of respondents mentioned that resource loading and DOM manipulation (e.g., inserting or deleting elements), respectively, are a bottleneck. Large resources typically are images, videos and scripts that are either loaded before or during execution of a JavaScript program. 29% of respondents identified Canvas operations as a bottleneck.

21% of respondents consider that number crunching/math computation is a bottleneck. While the percent may seem low compared to the opinion on other operations, we see it as significant in the context of current popular web sites, which usually do not execute any computationally-intensive algorithms. Another 40% of respondents do not dismiss number crunching/math computation as an issue.

The performance bottleneck classification question was followed by an open-ended question asking for any bottleneck we might have missed. There were 17 responses to this question. Five of them highlighted various aspects of layout and styling, and two mentioned the fallbacks for old browsers. The others mentioned diverse aspects like lack of tail recursion, garbage collection, runtime optimization, low level audio APIs, compression, and local storage.

2.3 Programming style

The programming style preferred by developers offers some insights into what parallel programming model they may consider to be more "natural" to use. For example, a key pattern in (pure) functional style programming is functions that operate on immutable data-structures

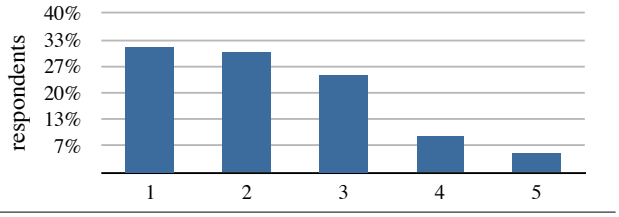

Figure 3. Programming style preference scale from Functional (1) to Imperative (5)

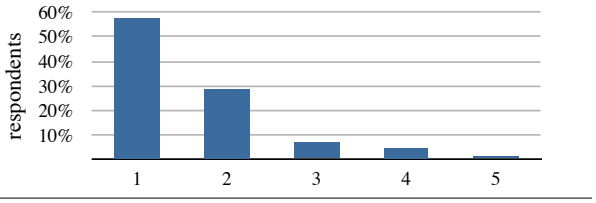

Figure 4. Preference scale for variables: from Monomorphic (1) to Polymorphic (5)

and are effectively stateless. This pattern is important in the context of parallel execution as immutable shared state simplifies value synchronization and has been used effectively in several parallel programming models [10, 22, 26]. On the other hand, programmers who like writing imperative style code may prefer task or thread-level parallel primitives with explicit synchronization on mutable state such as in languages such as C/C++ or Java.

Our survey asked developers which selection of language features they used frequently and which ones they did not. The results are summarized below.

Functional vs Imperative style While JavaScript uses the block structured syntax found in imperative languages like C/C++, Java and Python, it also supports many features commonly found in functional programming languages. For example JavaScript supports first class functions and closures.

The question is asked in order to qualitatively understand the style preferred by the respondents. We asked programmer to rate their preference on a scale of 1-5 with 1 being a strongly functional style and 5 being a strongly imperative style. The results are summarized in Figure 3. 31% of respondents replied they preferred to write code in a strongly functional style and 5% said they preferred a more imperative style. 52% of respondents also answered the "Why" follow-up question. Of these, a majority of the respondents who answered "1" (i.e., they strongly preferred a functional style) at the preference question mentioned that they found functional code to be more concise, readable, or understandable.

A few of the respondents who answered that they preferred a more imperative style pointed to performance issues as one of the reasons for their choice. An important consideration is the lack of tail call optimization in JavaScript which makes expressing iteration as recursion inefficient. Indeed ECMAScript 6, the next version of the JavaScript language standard includes support for *tail call optimization* to accommodate programming styles that are qualitatively more functional in nature.

Finally, a few of the respondents leaning towards more functional code, and a majority of those leaning towards imperative code, mentioned their background in a particular programming style as the reason for their choice.

High-level `Array` operators vs `for`-loops JavaScript Arrays have builtin operators such as `map`, `forEach`, and `every`. For example, the `map` method takes as argument a callback function, invokes it for each element in the Array in order and constructs a new Array out of the results of the callback. In addition to the pure JavaScript method, frameworks such as *Prototype* also include methods such as `map` on their own data types.

This question attempts to understand whether, why and when developers prefer to use these operators instead of iterating over the elements of the Array using a loop. Developers' preference can help determine the best way to make parallelism available. If developers prefer explicit loops, parallelism could be exposed through a loop annotation (akin to the `parallel` OpenMP pragma). If developers prefer operators, parallelism can be exposed though a special collection API, like the Parallel JavaScript proposal [27].

Of the respondents who answered this question, 74% said they preferred using the builtin operators. The principle reason given in the open-ended answer was that with the high-level operators, programmer intent was easier to convey and understand leading to better readability. Several respondents also mentioned the composability benefits of using the operators instead of explicit loops. Another common justification was that the callback functions supplied to the operators provided a scope for variables that is missing from explicit loops.

Several respondents who said they favored explicit loops cited the performance gap as an important reason for their choice. A few others mentioned that they preferred initially using the high-level operators, profiling their program to see if any bottlenecks were due to use of these constructs and replacing them with loops.

2.4 Parallelism-inhibiting language use

Use of global variables Global variables are common in JavaScript programs, despite being considered bad programming practice. They also make parallelization mode difficult and error-prone as they can generate race conditions. We asked developers the open-ended question "What would be a scenario where using global variables helps?" and got 105 responses. This question attempts to understand if and how our respondents use them. 33 of the respondents mentioned emulating a form of namespace or module system. Another common usage pattern mentioned was to communicate values between different scripts on the same page during execution and between the server and client on page load. Several respondents answered that they use global singleton for important data structures that are accessed in several parts of the program.

In our case study (Sections 3 and 4), we have encountered few instances of problematic use of global variables.

Polymorphism JavaScript is dynamically typed and both functions and variables can be polymorphic. A polymorphic variable can change its type during execution, e.g., we can assign a string to a variable that has so far pointed to an integer. While the flexibility can be useful in certain cases, it can also hamper compiler optimizations that depend on the variable's type.

Richards et al. [31] analyzed a large corpus of real-world JavaScript programs taken from the 100 most popular websites on the internet according to Alexa. They found that 81% of the call sites in these programs were monomorphic. Further, over 90% of functions were non-variadic i.e., their *arity* was fixed.

Our survey asked the respondents to rate their JavaScript programs on a scale of 1-5 with 1 presenting programs with *purely monomorphic variables* and 5 being programs that make extensive use of *variable polymorphism*. A summary of the responses is shown in Figure 4. About 58% of the respondents (98 out of 168) said the programs they write are purely monomorphic for variables. In contrast just 1% (2 out of 168) answered that their code made extensive use of variable polymorphism.

These results are similar to the findings of Richards et al. [31] and indicate that a majority of JavaScript code is written in a *de facto* statically-typed fashion which means that modern JIT engines may be able to infer these types effectively and produce performant

code. This is especially important for execution on parallel hardware platforms such as GPUs where type dynamism is difficult to support efficiently. Even in a multi-core setting, supporting type dynamism requires thread-safe runtime structures and algorithms for handling querying and updating types at runtime. Therefore the extent of data and function polymorphism in JavaScript programs significantly influences the space of programming models that can be implemented in browser engines.

3. Case study methodology

The case study brings more insight into the programming style and issues prevalent in the computationally-intensive parts of emerging web applications. The survey gave us a general idea of web developers' preferences, and of emerging trends. We now drill down, confirm some of the survey's findings, and take a step forward to answer the following research questions:

Q 1: How much latent data parallelism is available?

Q 2: What are the issues that may impede parallelization?

We first selected 12 web applications (shown in Table 1) by searching for the most mature implementations of the various trends identified by the survey respondents. The application set is heterogenous, it covers all the identified trend categories except the meta-category "desktop-like" and "Peer-to-Peer and Social". Each application is either a direct exponent of a trend (e.g., D3.js for Visualization) or a component for applications in a trend (e.g., Tear-able Cloth is a demo of cloth simulation, an important feature for realistic 3D games).

For each web application, we determine whether it is computationally intensive. If it is, we study the expensive computations using a combination of automated analysis and manual inspection. Thus, for each application:

1. We measure the processor time spent by the application (using the Gecko profiler [7]), and the time spent specifically in loops (using very lightweight instrumentation of JavaScript code).
2. We profile the application again using slightly heavier instrumentation that gives us statistical information about the runtime and trip count, i.e., number of iterations, of each loop in the program. Using this information, we identify the computationally-intensive loops.
3. We inspect each computationally intensive loop to determine whether it can be run in parallel and what could hamper or prevent parallelization. To ease the process, we run the web application again, this time instrumented to give detailed information about memory access patterns.
4. We interpret and summarize the results.

In order to identify the computationally-intensive loops and understand their behavior, we have developed JS-CERES, a profiling and runtime dependence analysis tool. It is implemented as a proxy server sitting between the browser and the web server. The proxy instruments JavaScript code on its way from the web server to the browser. On finishing the analysis, the browser sends the results back to the proxy, which then uploads them to github.com in a human-readable format. Our tool has on overlap in purpose to Jalangi [32], a general-purpose framework for writing dynamic analyses for JavaScript. Jalangi was not publicized at the time we developed JS-CERES. Furthermore, as we will see further, our specialized tool has a staged profiling and dependence analysis approach aiming to minimize the performance impact on the measured execution.

Fig. 5 illustrates the JS-CERES analysis process. It involves the following steps:

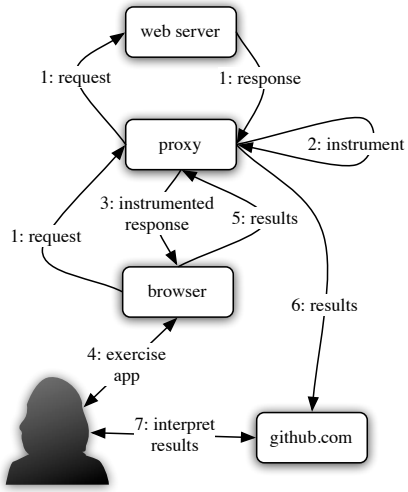

Figure 5. JS-CERES instrumentation and reporting process

1. The browser requests a document from the web server, passing through the proxy. The web server generates the requested document and sends it back, and the proxy intercepts it.
2. If the document is either HTML or JavaScript, the proxy transforms any encountered JavaScript code, adding instrumentation for profiling and dependence analysis.
3. The proxy sends the instrumented document back to the browser, fulfilling the request.
4. The user interacts with the web application to exercise any computationally-intensive code. As the application runs, the instrumentation gathers and summarizes the results.
5. The user asks the web application to send back the results by clicking a special button overlaid on top of the interface by the instrumentation engine. In response, the browser sends the results of the analysis to the web server. The request is intercepted by the proxy.
6. The proxy analyzes the results and transforms them to a human readable format. It then pairs the results to the original documents, and saves them by committing to a local git repository. Finally, the proxy pushes the results to github.com.
7. We analyze the results. github.com is used as it provides both version tracking and a convenient way to link result reports to source code.

JS-CERES has three instrumentation modes: lightweight profiling, loop profiling, and dependence. Each mode is meant to aid one of the aforementioned steps taken when analyzing each web application. The three modes are separated in order to minimize the bias in the results due to the instrumentation overhead.

3.1 Lightweight profiling

In this mode, the tool only measures two scalar values: the total time from the start of the application, and the total runtime spent in all the loops in the program. JS-CERES adds before and after each loop code that increments and, respectively, decrements a counter that represents the number of open loops in the program. When encountering a loop and the counter is 0, a separate variable remembers a timestamp. When exiting a loop brings the counter to 0, the difference between the current timestamp and the last remembered timestamp is added to a global variable that holds the total time spent in loops. The timestamps are taken using the new

Table 1. Case study - web applications

Name/URL	Category/Description
HAAR.js / `github.com/foo123/HAAR.js`	User recognition / face recognition (Viola-Jones)
Tear-able Cloth / `lonely-pixel.com/lab/cloth`	Games / cloth physics simulation (Verlet integration)
CamanJS / `camanjs.com`	Audio and Video / image manipulation library
fluidSim / `nerget.com/fluidSim`	Games / fluid dynamics simulation (Navier-Stokes)
Harmony / `mrdoob.com/projects/harmony`	Audio and Video / Drawing application
Ace / `ace.c9.io`	Productivity / code editor used by the Cloud9 IDE
MyScript / `webdemo.visionobjects.com`	User recognition / handwriting recognition application
Raytracing / `gist.github.com/jwagner/422755`	Games / real-time raytracing demo
Normal Mapping / `http://29a.ch/experiments`	Games / normal mapping
sigma.js / `sigmajs.org`	Visualization / GEXF rendering
processing.js / `processingjs.org`	Visualization / interactive spiral visual effect
D3.js / `d3js.org`	Visualization / interactive azimuthal projection map

JavaScript high resolution timer [4]. We observed that this setup has no discernible impact on the runtime of the apps we analyzed.

We couple this instrumentation mode with using the Gecko profiler [7] within Firefox to also measure the amount of time the processor is active. We were surprised to find that the active time reported by Gecko is often lower than the time spent in loops measured by JS-CERES. We believe there are two main reasons for this anomaly. First, the Gecko profiler is only sampling the computation. As the sampling occurs at function level (for performance reasons), a long running computation within a single function may be seen as inactive time. Second, if there is any blocking code within the loop or the OS or Firefox decides to suspend the thread, JS-CERES continues to count the time as part of the loop. We tried to minimize this effect by not having any other expensive processes active on the machine when profiling.

We run the experiments on a quad-core Intel Core i7 at 2.6 GHz (3720QM) with 16 GB of RAM. Running the same experiments on a less performant platform (e.g., mobile) would likely further increase the effect of performance bottlenecks.

3.2 Loop profiling

In this mode, JS-CERES instruments the program to compute, for each syntactic loop: the number of times it is encountered, the total, average, and variance of its running time, and the total, average, and variance of its trip count. To compute this information, JS-CERES adds the following instrumentation:

- each syntactic loop is represented by an object in a global map
- before each loop, a trip counter is set to 0 and a timestamp is recorded
- before each iteration, the trip counter is incremented
- after each loop, the trip count and the loop's running time are added to the running totals, and variance is updated using Welford's online algorithm [36].

This instrumentation mode has only minimal discernible impact on the running time of the applications.

3.3 Dependence analysis

In this mode, JS-CERES instruments the program to gather information about the memory access patterns within specific loops. As this type of instrumentation has a very high overhead, JS-CERES allows the programmer to focus on a specific loop. The tool then reports all problematic memory accesses that happen, at runtime, within the focused loop. The following memory access types are reported, as they can break a dependency when parallelizing the loop:

a) writes to variables that are declared outside the context of the current loop iteration. As JavaScript variables have function scope, it includes all variables that are syntactically inside the loop. As expected, it excludes variables from functions called from within the loop. The writes generate *output dependencies* (write-after-write) [15] between different iterations.

b) writes to fields of objects that are initialized outside the context of the current loop iteration. The accesses generate *output dependencies* between iterations, and may be involved in *anti-dependencies* (write-after-read).

c) reads of fields which have been initialized in the loop, but in a different iteration. The accesses generate *flow dependencies* (read-after-write).

```
1  function step() {
2    computeForces();
3
4    var com = new Particle();
5
6    for(var i=0; i<bodies.length; i++) {
7      var p = bodies[i];
8
9      // update velocity
10     p.vX += p.fX / p.m * dT;
11     p.vY += p.fY / p.m * dT;
12
13     // update position
14     p.x += p.vX * dT;
15     p.y += p.vY * dT;
16
17     // update center of mass
18     com.m = com.m + p.m;
19     com.x = (com.x * ...
20     com.y = (com.y * ...
21   }
22   return com;
23 }
24 while(true) {
25   var com = step();
26   display(bodies, com);
27 }
```

Figure 6. Example - N-body simulation step

We exemplify the problematic access types with a snippet of an N-body simulation that also computes a live version of the center of mass of the entire system (Fig. 6). At each step, the force applied to each object is updated depending on the forces of the other objects. Then, the velocity and position of each point are updated depending on the resulting force applied to it. For brevity, we only focus on the second part of the computation.

JS-CERES displays all the aforementioned types of warnings for the `for` loop at line 6. Each warning report contains a list of triples representing the loop nest. Each triple characterizes a loop in the nest, with the outermost loop being first in the list. Each triple is composed of a loop identifier, followed by two boolean flags. The first flag indicates whether, for the variable in question, there is a dependency between different runtime *instances* of the loop.

The second flag indicates a dependency between different runtime *iterations* of the loop.

warning for the write to variable `p` on line 7 The variable's declaration is syntactically enclosed in loop's block, but JavaScript only has function scoping. So, the variable behaves as if it has been declared at the beginning of the `step` function. In this case it means all the `for` loop iterations will share the same `p` variable. JS-CERES also reports which loops share the variable, and how.

In this case, JS-CERES characterizes the access with the following "→"-separated list of triples:

`while(line 24) ok ok` → `for[line 6] ok dependence`

The `while` and `for` represent specific syntactic loops. "`ok`" is interpreted as "each instance/iteration of this loop has its own private version of this variable". "`dependence`" is interpreted as "all instances/iterations of this loop share the same variable". If all instances share the same variable, all iterations will also share the same variable. Thus, "`dependence ok`" is not a valid characterization.

For the `while` loop in our example, each instance and, furthermore, each iteration has a private version of `p`. Thus, the access is characterized as "`ok ok`".

From the `for` loop's perspective, each instance has its own private version of `p` (the first "`ok`"), but all iterations share the same version ("`dependence`") due to function scoping. Thus, the same access is characterized as "`ok dependence`", denoting the output inter-iteration dependence on `p`.

warnings for the writes to properties `vX`, `vY`, `x`, `y` of `p`, and `x`, `y`, `m` of `com` The accesses are considered problematic as `p` and `com` are shared between the loop's iterations. Like the write to `p`, these accesses are also characterized as:

`while(line 24) ok ok` → `for(line 6) ok dependence`

If the body of the loop would be extracted into a separate function, or the loop would be expressed as an `forEach` operation, the accesses to the properties, i.e. fields, of `p` would be characterized as:

`while(line 24) ok ok` → `for(line 6) ok ok`

They would not be considered problematic, and would not be reported. The warning on `com` would stand.

warnings for the reads of properties `x`, `y`, `m` of `com` These reads are also characterized as:

`while(line 24) ok ok` → `for(line 6) ok dependence`

but the interpretation is slightly different. The "`dependence`" indicates that the read value has been written in a *different* iteration of the loop. As the loop is sequential, it means that there is a flow, i.e. true, dependence between the loop iterations.

Instrumentation For computing the three types of warnings, JS-CERES instruments the original program to maintain, at each point during execution, a characterization with respect to the open, i.e., currently iterating, loops. The characterization is maintained as a stack. When encountering a loop at runtime, the instrumentation pushes to the stack a triple containing:

- the a loop unique identifier (represents the syntactic loop)
- the current value of a counter of how many times the entire loop has been seen so far (maintained in a global map from loop identifier to counters)
- 0, representing the current iteration of the loop

Every time a loop is encountered, the counter in the aforementioned global map is incremented before pushing the triple to the stack. Also, recursive function calls may make the stack grow indefinitely. JS-CERES detects this, raises a warning, and discards the analysis results for the affected loop nest.

At the beginning of each loop iteration, the current iteration part of the triple is incremented in place.

Furthermore, each object creation site in the program (by any means, `new`, `function`, `Object.create`) is instrumented to wrap the created object in a Proxy [2, 34]. The proxy is used to save a characterization-stamp of the object instantiation moment, i.e. the current value of the stack when the object is instantiated.

Finally, each variable or property read or write in the program is instrumented to check for problematic accesses. On writing a property, a *diff* is computed between the current stack and the characterization-stamp of the property's object. If they are the same, the access is not considered problematic. If they are different, all values that are the same are reduced to "`ok`", and all differing values are reduced to "`dependence`", thus obtaining the aforementioned list of triples. Also, a snapshot of the above stack is remembered for the particular written property name. On encountering a property read, a *diff* is computed between the current stack and the snapshot of the field. If they are different, we have discovered a flow dependence, and a warning is raised.

Table 2. Case study - running time.

	Running time (s)		
Name	Total	Active	In Loops
HAAR.js	8	2	0.44
Tear-able Cloth	14	7	9
CamanJS	40	23	17
fluidSim	22	17	12
Harmony	41	0.36	0.28
Ace	30	0.4	0.4
MyScript	12	0.33	0.15
Realtime Raytracing	62	19	26
Normal Mapping	25	6	4
sigma.js	32	9	8
processing.js	21	12	2
D3.js	18	5	4

4. Case study results

We now discuss the results in the context of the original research questions.

4.1 How much latent data parallelism is available?

First, we approximate an upper bound for latent data parallelism by using the runtime spent in loops as a proxy. The results of the experiment (described in Sec. 3.1) are summarized in Table 2. The second column shows the total time each application is active, i.e. the time before starting the application and the time results are gathered. The third column shows the amount of time the CPU was active, as reported by the Gecko profiler. The last column shows the total amount of time spent in loops, are measured by JS-CERES's instrumentation. The fact the total amount of active time is sometimes lower than the time spent in loops is an artifact of our methodology (see Sec. 3.1). Still, we believe the overall conclusion stands: at least half of the applications can be considered computationally intensive (i.e. the CPU is active for a large portion of their running time) and, for most of these, a large part of the computation occurs in loops. Not all loops are parallelizable, but the fact that looping is a significant part of the computation puts a high upper bound to the amount of latent data parallelism.

Next, we identify the most computationally-intensive loop nests (a group of loops that nested within a single top-level loop) in each of the applications and check whether their computation is inherently data-parallel. This doesn't necessarily mean the parallelism can

be exploited in the near future as there are still technological challenges with current web browser technology (e.g. the DOM is not concurrent). It does, however, improve the previous (Table 2) approximation of latent-parallelism upper bound.

For each application, we inspect the top loop nests that, together, make up at least two thirds of the application's time spent in loops. Altogether, we inspect 22 loop nests across the 12 subject web applications. Table 3 shows a summary of our findings. Each row represents an inspected loop nest. The runtime part of the table shows the percentage of the total looping time spent in the particular loop nest, the number of times the loop nest has been encountered at runtime (instances), and the average and standard deviation for the trip count of the outer loop of the nest (across all instances of the said loop nest). In a few cases the parallelizable loop is not the outer loop of a nest. In these cases we consider the loop nest formed without some of the outer layers, and report the results for this inner loop nest instead.

About three fourths of the inspected loop nests have some intrinsic parallelism, i.e. do not have dependencies that we think could not be broken. Also, in most cases, the trip count and granularity is high enough for some form of data-parallelism to be potentially useful. Still, exploiting this parallelism may not be easy. In many cases it would require a combination of code changes and browsers with efficient parallel data structures and concurrent DOM and Canvas implementations.

4.2 What are the issues that may impede parallelization?

We found that JavaScript poses the traditional issues to parallelization, while also raising new ones that stem from its evolving, dynamic, and web-centric nature. In addition to discovering available parallelism and matching the parallel computation to the hardware, a JavaScript programmer also needs to get around concurrent updates to the non-concurrent DOM, concurrent reads and writes of global memory, and polymorphic variables. Columns 5-8 in Table 3 summarize these issues and how often we encountered them in the inspected loops.

Control-flow divergence Control-flow is *diverging* when the execution takes different paths depending on a dynamically evaluated predicate. Such behavior is usually generated by branching statements and loops with data-dependent number of iterations.

Control-flow divergence can make different threads execute different instructions, so it is an issue when trying to run parallel code on SIMD architectures. Several techniques have been proposed to allow control-flow divergence while minimizing the performance impact [17, 19, 21, 25, 29, 30, 37, 38]. Still, the overhead is still much higher than that exhibited on CPUs.

Column 5 shows an assessment of the amount of control-flow divergence. We found several cases where the computation would, algorithmically, be very hard to adapt for SIMD parallelism:

- The second loop in HAAR.js does, at each iteration, a recursive search through a tree which makes the iterations uneven.
- The loops in Ace only execute roughly one iteration on average. The first loop executes a rendering method until there are no more cascading changes.
- The Raytracing algorithm contains variable depth recursion.
- For MyScript, the only client-side expensive loop executes only a few iterations, computing the length of line segments.
- Some loops (in sigma.js and processing.js) execute very few iterations.

In most other cases (labeled as "little") the iterations contain branching statements but their effect is local and they only contain a few instructions. Thus, we expect they can be transformed to versions that use instructions guarded by predicates or `select` instructions instead of branches without a major performance impact. Finally, a few nests contain recursive functions or inner loops with variable data-dependent bounds. These loop nests may will pose additional challenges when attempting SIMD parallelization.

DOM accesses Column 6 shows that half of the loop nests access the DOM. This is problematic as, although there is some research in this area [28, 35], no major browser currently supports concurrent accesses to the DOM.

Accesses to shared memory Code within loops may access shared (i.e., not local to the loop) memory locations. These accesses may generate dependencies between loop iterations (see Sec.3.3). These dependencies need to be broken in some way in order to correctly parallelize the loop. Column 7 shows our assessment of how hard it would be for a programmer to break those dependencies for each loop nest. We made this assessment by manually inspecting access patterns within each loop nests with the help of our dependence analysis tool. The dependence analysis tool was particularly helpful in identifying flow dependencies but it also failed to scale to some of the case studies.

Most loop nests make complex accesses to variables from global memory, and all loops at least read global memory. The good news is that in more than two thirds of the loop nests the write accesses have a well-defined pattern that allows parallelism.

Polymorphic variables In the survey, we have asked programmers about their use of polymorphic variables (see Sec. 2.4). We now confirm the results in the context of the computationally-intensive loops in the case study by manually inspecting the code for polymorphic variable accesses. We consider a variable polymorphic if the property accesses or method invocations made through dereferencing this variable assume objects of different types. E.g., we consider a variable polymorphic if at one point in the program it is a number, while at another point it is invoked as a function. We do not consider a variable polymorphic if it changes between `defined`, `undefined`, and `null`. Our manual inspection did not reveal any polymorphic variables within the computationally-intensive loops.

Finally, column 8 of Table 3 shows our estimate of how easy it would be to parallelize the loop nests, by considering both how easy it would be to break dependencies and current browser limitations (i.e. non-concurrent DOM and Canvas). Considering Amdahl's law, the upper bound for speedup is greater than $3\times$ for 5 of the 12 applications when only counting easy to parallelize loops. On the other end of the spectrum we think it would be hard or very hard to obtain any significant speedup for 5 of the 12 applications.

5. Implications

Our study has several practical implications. We organize them based on the community for which they are relevant.

5.1 Library developers and researchers

Using the survey data from Section 2.3, the preference for iterating through functional-style operators is perhaps intuitively unsurprising given that JavaScript as a whole is a high-level language with higher order functions, and extensive usage of closures in practice. What is more surprising is that programmers often prefer these high-level operators even at the cost of some performance. This means that any proposed parallel programming model should present a sufficiently high-level interface that abstracts away concurrency and synchronization issues, hardware features and scheduling. Thus, libraries can take a functional approach to exposing data parallelism (like RiverTrail did) instead of an annotative one (e.g., OpenMP pragma directives).

Table 3. Case study - detailed inspection of loop nests

name	%	runtime instructions	trips/instruction	control flow divergence	DOM access	breaking dependencies	parallelization difficulty
HAAR.js	38	10	31±23	little	no	easy	easy
	36	50k	15±15	yes	no	easy	medium
Tear. Cloth	80	1077	1581	little	no	medium	medium
CamanJS	72	536	90k	little	no	easy	easy
	15	16	90k±300	little	no	easy	easy
	7	12	360k	little	no	easy	easy
fluidSim	90	40k	168±147	none	no	easy	easy
Harmony	33	207	50	none	yes	easy	very hard
	32	498	50	none	yes	easy	very hard
	15	123	5±3	none	yes	easy	very hard
Ace	42	125	1±0.1	yes	yes	very hard	very hard
	22	123	1±0.2	yes	yes	very hard	very hard
MyScript	70	511	4±2	yes	yes	very hard	very hard
Raytracing	98	772	120	yes	no	very easy	easy
Norm. Map.	99	64	65k	little	no	very easy	easy
sigma.js	68	2070	191±27	little	yes	very hard	very hard
	22	638	196±21	yes	yes	very hard	very hard
processing.js	25	54.6k	4±37	no	no	easy	medium
	22	54.6k	4±37	no	no	easy	medium
	16	54.5k	2	yes	yes	medium	very hard
	13	54.6k	4±37	no	no	easy	medium
D3.js	99	51	156±57	yes	yes	hard	hard

Looking at the case study data from Table 3, the non-concurrent DOM is a bottleneck to exploiting data parallelism, but not the biggest issue. Most of the loop nests that were not parallelizable due to DOM operations were not that compute-intensive to begin with.

The complexity of global memory accesses seen in the case studies implies that library developers will need to provide easy ways of making arbitrary variables available to the parallel kernels. Ideally, the memory management would be part of the JIT compilation.

5.2 Builders of web browser engines

Runtime polymorphic variables typically have some performance cost since the JavaScript JIT engine must resolve the type of such variables by leaving the fast JIT-ed code path and entering the browser runtime. Modern browser engines implement sophisticated type inferencing [24] and speculation to reduce these overheads. On the other hand our survey indicates that many developers write programs that are predominantly monomorphic with respect to variables (see Fig. 4). Our case studies of compute-intensive JavaScript programs also support this observation. This suggests that compute-intensive programs would benefit from aggressive type speculation and other mechanisms that provide a fast path for code that is completely monomorphic and can be statically analyzed. This is especially important in the context of parallelism since running polymorphic code in parallel usually requires browser runtimes be made thread safe.

Our survey indicates that many developers prefer using high-level operators such as `map` or `foreach` instead of explicit loops. Firstly this suggests that browser engines need to have efficient implementations of these operators. Secondly, many of these higher level constructs such as `map`, `reduce` etc., are particularly suited for specifying parallelism as they capture the underlying parallelism-enabling structure of the computation. This approach is taken by Parallel Javascript [27].

5.3 Tool developers and researchers

Our experience analyzing the case study applications shows that a standard profiler is insufficient for identifying parallelism opportunities as it does not provide any information about the loops. A profiler with integrated loop information retrieval can help - along the lines of our prototype or, more advanced, the modeling tools in mature IDEs like Microsoft Visual Studio (for C#) or Intel Parallel Studio (for C++).

If parallelized, most of the loop nests we analyzed would have races. Most of these races would be fixable but the user would need to be aware of them and devise a strategy around them. As speculative parallelization gains ground for JavaScript, it means that it does not only need to abort when it fails to run a loop in parallel, but also have ways to report to the developer the reason for aborting. Furthermore, once the detailed reason for aborting is identified, the developer would need to transform the code significantly to solve the issue, part of which may be automated.

Our case studies show that all loops that are compute-intensive are written in a imperative style. Refactoring tools [23] that can transform imperative iteration into functional style could make these loops amenable to parallelism via libraries with parallel operators such as RiverTrail[27].

5.4 JavaScript developers

As the workloads for emerging web application trends indicate parallelism would be useful, and considering the WebCL [12] and Parallel JavaScript [27] proposals, JavaScript developers should expect to have access to parallel constructs in the next few years.

In this context, a clean design and implementation (e.g., avoiding global variables) not only helps with maintainability, but reduces number of hard-to-find parallelism-inhibiting dependencies.

Also, it may be that developers should trust their instincts: if they like functional code more (as the data in Fig. 3 indicates), they may be better of writing it, despite fears of loss of performance. JavaScript engines tend to adapt quickly to the usage scenarios that are frequent in practice. And they may get parallelism as an added bonus in the future.

5.5 Educators

While our survey shows that developers are not adverse to functional-style operators for iteration in principle, the case study applications contain very few loops that use functional operators. This may suggest that, while developers understand and like the concept, they are using explicit `for` loops out of habit. Early-on education about

alternate ways of iteration may help. The other possible reason for this anomaly is that developers are wary that functional-style operators are slower than explicit loops.

6. Related work

We are aware of two related studies in the context of JavaScript. Fortuna et al. [20] study a set of widely used web-sites and come to the conclusion that current web workloads offer significant potential for parallelization, with projected speedups ranging from a factor of 2.19 to 45.46 and averaging around a factor of 8.91. While this suggest that there is use for parallelism, the authors also found that the majority of speedups stem from parallel execution of independent tasks rather than independent loop iterations. This suggests that the web would benefit less from data parallelism oriented approaches like WebCL and Parallel JavaScript and that a lightweigth task-based approach might be more appropriate.

Another study on JavaScript [31], looking at its dynamic behavior, comes to the conclusion that web-sites indeed make significant use of dynamic features: Many websites use *eval* to generate code on the fly, object properties change throughout their lifetime, including properties being deleted or their types changing, and a significant number of call sites are polymorphic. Such dynamic behavior not only makes static analysis of JavaScript hard but also renders execution on more restricted parallel hardware like SIMD extensions or even GPUs challenging. Thus, it gives another reason to believe that data parallelism and the web do not pair well.

However, as mentioned earlier, both studies focus on websites that mostly use a page-centric approach and have only low compute density. While these are valid studies to understand the status quo, they are not well suited to judge behavior of new and emerging application-centric web usages, which is the focus of this work.

7. Conclusion

With the proliferation of desktop and especially mobile operating systems, the web is increasingly seen as a cross-platform solution for delivering applications. In our survey, when asked about emerging trends in web applications, JavaScript developers mostly identified kinds of applications that, not long ago, were only available as native desktop applications.

But this transitioning comes with a challenge: native desktop applications had to resort to multi and many-core parallelism for performance. Should the web follow suit? If so, how hard will it be?

To answer these questions we conducted a survey among JavaScript developers asking them about their use of JavaScript language-features that may impede parallelism. Furthermore, we did a case study looking at the computationally-intensive loop nests in 12 web applications. While JavaScript is highly dynamic, we found that developers seldom use language features that impede parallelism. An important current limitation is that browsers have non-concurrent implementations of basic data structures (e.g., the DOM). Much of the compute-intensive code we inspected is written in a style typical of non-dynamic imperative languages. This means that many of the lessons learned by the programming community while parallelizing desktop applications will translate to the web.

References

[1] Alexa. http://www.alexa.com.

[2] Ecmascript proxy. http://wiki.ecmascript.org/doku.php?id=harmony:direct_proxies.

[3] Epic citadel. http://www.unrealengine.com/html5.

[4] High Resolution Time. http://www.w3.org/TR/hr-time/.

[5] Html canvas. http://www.w3.org/TR/2dcontext2/.

[6] HTML Touch Events API. http://www.w3.org/TR/touch-events/.

[7] Mozilla Gecko Profiler. https://developer.mozilla.org/en-US/docs/Performance/Profiling_with_the_Built-in_Profiler.

[8] openCL. http://www.khronos.org/opencl/.

[9] Pointer lock. http://www.w3.org/TR/pointerlock/.

[10] Rust language. http://www.rust-lang.org.

[11] Web Workers. http://www.w3.org/TR/workers/.

[12] webCL. http://www.khronos.org/webcl/.

[13] webgl. http://www.khronos.org/webgl/.

[14] G. A. Agha. Actors: a model of concurrent computation in distributed systems. 1985.

[15] R. Allen and K. Kennedy. Automatic translation of fortran programs to vector form. *TOPLAS '87*.

[16] M. Anttonen, A. Salminen, T. Mikkonen, and A. Taivalsaari. Transforming the web into a real application platform: new technologies, emerging trends and missing pieces. In *SAC '11*.

[17] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu. An adaptive performance modeling tool for gpu architectures. In *PPoPP '10*.

[18] D. S. Cruzes and T. Dyba. Recommended steps for thematic synthesis in software engineering. In *ESEM '11*.

[19] R. Farivar, H. Kharbanda, S. Venkataraman, and R. H. Campbell. An algorithm for fast edit distance computation on gpus. In *InPar 2012*.

[20] E. Fortuna, O. Anderson, L. Ceze, and S. Eggers. A limit study of javascript parallelism. In *IISWC '10*.

[21] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In *MICRO '97*.

[22] C. S. Gordon, M. J. Parkinson, J. Parsons, A. Bromfield, and J. Duffy. Uniqueness and reference immutability for safe parallelism. In *OOPSLA '12*.

[23] A. Gyori, L. Franklin, D. Dig, and J. Lahoda. Crossing the gap from imperative to functional programming through refactoring. In *FSE '13*.

[24] B. Hackett and S.-y. Guo. Fast and precise hybrid type inference for JavaScript. In *PLDI '12*.

[25] T. D. Han and T. S. Abdelrahman. Reducing branch divergence in gpu programs. In *GPGPU '11*.

[26] S. Herhut, R. L. Hudson, T. Shpeisman, and J. Sreeram. Parallel programming for the web. In *HotPar '12*.

[27] S. Herhut, R. L. Hudson, T. Shpeisman, and J. Sreeram. River Trail: A path to parallelism in JavaScript. In *OOPSLA '13*.

[28] C. G. Jones, R. Liu, L. Meyerovich, K. Asanović, and R. Bodík. Parallelizing the web browser. In *HotPar '09*.

[29] J. Meng, D. Tarjan, and K. Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *ISCA '10*.

[30] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. Gpu computing '08.

[31] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of javascript programs. In *PLDI '10*.

[32] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *FSE '13*.

[33] D. Ungar and R. B. Smith. Self: The power of simplicity. In *OOPSLA '87*.

[34] T. Van Cutsem and M. S. Miller. Proxies: design principles for robust object-oriented intercession apis. *SIGPLAN Not.*, 45(12), Oct. 2010.

[35] C. A. Vick, B. Wang, and M. M. H. Reshadi. Concurrent parsing and processing of HTML and JAVASCRIPT. US Patent 20,120,290,924.

[36] B. Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962.

[37] E. Z. Zhang, Y. Jiang, Z. Guo, and X. Shen. Streamlining GPU applications on the fly: Thread divergence elimination through runtime thread-data remapping. In *ICS '10*.

[38] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly elimination of dynamic irregularities for GPU computing. In *ASPLOS'11*.