Performance Scalability Analysis of JavaScript Applications with Web Workers

Javier Verdú* and Alex Pajuelo†
Department of Computer Architecture, BarcelonaTECH (UPC)
Barcelona, Spain
Email: *jverdu@ac.upc.edu, †mpajuelo@ac.upc.edu

***Abstract*—Web applications are getting closer to the performance of native applications taking advantage of new standard–based technologies. The recent HTML5 standard includes, among others, the Web Workers API that allows executing JavaScript applications on multiple threads, or workers. However, the internals of the browser's JavaScript virtual machine does not expose direct relation between workers and running threads in the browser and the utilization of logical cores in the processor. As a result, developers do not know how performance actually scales on different environments and therefore what is the optimal number of workers on parallel JavaScript codes. This paper presents the first performance scalability analysis of parallel web apps with multiple workers. We focus on two case studies representative of different worker execution models. Our analyses show performance scaling on different parallel processor microarchitectures and on three major web browsers in the market. Besides, we study the impact of co–running applications on the web app performance. The results provide insights for future approaches to automatically find out the optimal number of workers that provide the best tradeoff between performance and resource usage to preserve system responsiveness and user experience, especially on environments with unexpected changes on system workload.**

***Index Terms*—HTML5, Web Workers, JavaScript, web apps, parallelism, multithreading.**

I. Introduction

Web applications follow the new HTML5 standard developed by the World Wide Web Consortium [4] to address the requirements of current and future platforms, web contents, and cloud services. HTML5 provides new HTML elements, libraries extensions, and APIs to take further advantage of the underlying hardware, as well as reducing the need to install third–party plugins. Hence, current web apps show similar performance to native applications.

Some programming languages exploit parallelism by the use of specific APIs for multithreading (*e.g.* CUDA, OpenMP) with a performance scaling closely related to the underlying hardware resources. Other parallel programming languages, that require virtual machines (*e.g.* Erlang, Java), increase deviations of performance scalability, since it is not only related to hardware resources, but also to the internals of the virtual machine [1, 2]. This paper focuses on JavaScript, an interpreted language, largely employed to develop web apps executed in web browsers. HTML5 pays special attention to the support of JavaScript and brings, among others, a new mechanism and the API called Web Workers [6]. Even though JavaScript follows a single–thread execution model, Web Workers API allows multiple JavaScript codes to concurrently run in background threads, from now on workers, communicated by message passing with the main thread. As JavaScript web apps run on top of a web browser's virtual machine, it increases the unpredictability of performance scaling of languages that run on virtual machines.

By the use of Web Workers API, the developers are responsible of extracting the parallelism and properly express it in the web apps, unlike Thread–Level–Speculation techniques of JavaScript engines that automatically extract parallelism from sequential codes [5]. Programmers currently use the processor resources availability as heuristic to find out how many workers should be spawned to get the highest performance. Although HTML5 provides support to take advantage of hardware acceleration and better use of resources, JavaScript is not able to retrieve the underlying hardware specifications, such as the number of logical cores, aka hardware threads, comprised in the CPU. Major browser vendors address this constraint in different manner [12]. Google Chrome, Safari, and Opera implement a new attribute on the browser's navigator object, called *navigator.hardwareConcurrency* to obtain the number of hardware threads, regardless of the system workload. Other web browsers, such as Internet Explorer and FireFox, do not support it yet and users have to develop a benchmark to estimate the number of logical cores available. But, this estimation is sensitive to both system workload variations, since other co–running applications can deviate the performance of the benchmark, and optimizations of JavaScript engines, since a particular benchmark can be highly optimized by some browsers, but badly interpreted by others. Thus, both approaches provide biased information to developers to determine what is the optimal number of web workers for any particular web app.

This paper presents the first performance scalability analysis of JavaScript web apps that comprise multiple workers. We introduce a classification of web apps according to the worker execution models and focus specifically on two representative case studies. We compare performance scalability between parallel microarchitectures, single–threaded and multi–threaded multi–cores, as well as between Chrome and other two major web browsers. Besides, this work also analyzes the impact of co–running applications on the performance of highly parallel JavaScript web apps. The results offer insights towards future approaches to find out the optimal number of workers to exploit parallelism.

II. Related Work

Some studies characterize JavaScript programs using single–threaded benchmarks [10, 7]. Other authors propose fine grain parallelization of JavaScript codes. Fortuna et al. [3] analyze the potential speedup limit of parallelizing tasks and events. Martinsen et al. [5] implement and analyze Thread–Level–Speculation for browsers' JavaScript engines to take advantage of parallel processors. Finally, Watanabe et al. [11] describes a technique to parallelize interactive animation JavaScript using Web Workers, but the authors do not study its scalability.

None of these works are focused on either analyze the performance scalability of workers based web apps or related differences among major web browsers.

J. Verdú and A. Pajuelo are with the Universitat Politècnica de Catalunya, Spain. email:{jverdu,mpajuelo}@ac.upc.edu

DOI 10.1109/LCA.2015.2494585

III. Classification of Worker Execution Models

Parallel JavaScript web apps comprise the main thread, responsible of the UI, since workers cannot do it due to access constraints [6], and background threads for Web Workers aimed at computing intensive tasks to preserve responsiveness and enhance user experience. Although communication channels can be created among Web Workers, in this paper we focus on the default message passing between workers and the parent thread. Besides, workers are classified into two categories: dedicated workers, aka standard workers, only accessible by the script that spawned it; and shared workers, accessible from any script running in the same domain. Other emerging types of workers are still experimental [6].

We introduce a classification of parallel web apps based on worker execution models, regardless of worker origins. In fact, this work is focused on the web app code behavior and how parallelism is exploited:

- **Single worker:** all computing intensive tasks are done by a single worker. Videogames, for example, offload CPU intensive tasks, like AI and physics, to sustain responsiveness and frame rate. Web apps that comprise multiple computing intensive tasks suitable to run in several threads are conveyed to one of the other categories.
- **Multiple asynchronous workers:** large/continuous workload is distributed among available workers to be processed in parallel, like spell checking. These applications have no synchronization points among workers. As soon as a given worker notifies to the parent thread that the task is done, a new workload is delivered to the worker thread for processing.
- **Multiple synchronous workers:** inherent parallel codes, such as image/video processing, use to have a synchronization point among workers, like the presentation of a new frame. Every new workload, a frame, can be split into multiple jobs, slices, to be processed by different workers. However, workers cannot directly start processing new frames until all workers have finished their work. Thus, these applications can present periods of time with idle workers, even having pending frames.

IV. Framework

We use a personal computer with an Intel® Core™ i7-3960X processor at 3.3GHz with 6 hyperthreaded cores, for a total of 12 logical cores, with 16GB DDRAM-III and a Nvidia® GTX560 videocard, running Microsoft® Windows™ Server 2008 R2. We use Windows since it is the operating system most widely used by end–users that run web applications in desktop computers [9]. All non–critical services and applications have been disabled to prevent as much as possible any deviation in the measurements.

We use Process Explorer v15.21 [8] to select, by the use of Set Affinity, the available logical cores used in the experiments to mimic different parallel processor architectures.

Web apps run in updated releases of the three major web browsers [9]: Google Chrome v42.0.2311.90m (the default browser in our experiments), Mozilla Firefox v37.0.2, and Microsoft Internet Explorer v11.0.9600.16476, IE from now on.

There are no standard JavaScript benchmarks comprising workers. Several well known web apps and web browser portals though provide parallel JavaScript demos. Most of them are implemented with dedicated workers. Actually our analysis is independent of whether workers are dedicated or shared, but it is focused on the execution model of web apps with multiple workers, see Section III. We have run several web apps that have no human interaction requirements. This paper delves into the analysis of two case studies and, due to space limitations, we use one representative benchmark for each. Even though particular performance numbers of other web apps may differ, the trends shown in this paper are the same:

- **Multiple asynchronous workers:** Hash Bruteforcer [13], HashApp, is a web app with dedicated workers that computes MD5 hashes. We use the default configuration and data sets. From a given 128-bit MD5 encoded input, the application uses a brute force attack to decode the string. Thus, the workers perform continuous CPU intensive workload.
- **Multiple synchronous workers:** The raytracer web app [12], RayApp, performs highly CPU intensive mathematical calculations to simulate components of a scene, like ambient lights and shadows, to render every frame. We use the default configuration, but enlarging the default canvas size up to 300x300 pixels. The scene is split into a number of slices that depends on the canvas size. Thus, in our experiments every frame rendering consists of 15 slices distributed along a configurable pool size of dedicated workers. We have done experiments with other canvas sizes, showing different performance numbers, but with similar trends.

Both benchmarks have been slightly modified to do experiments from 1 to 20 Web Workers. We fix this limit since it is the maximal number of workers that Firefox supports. We prevent measurement deviations due to previous experiments, mainly due to internal optimizations and garbage collector issues, by closing and restarting the web browser after every experiment. Finally, we take average results from ten runs of every experiment running 30 seconds each, time enough to reach steady performance.

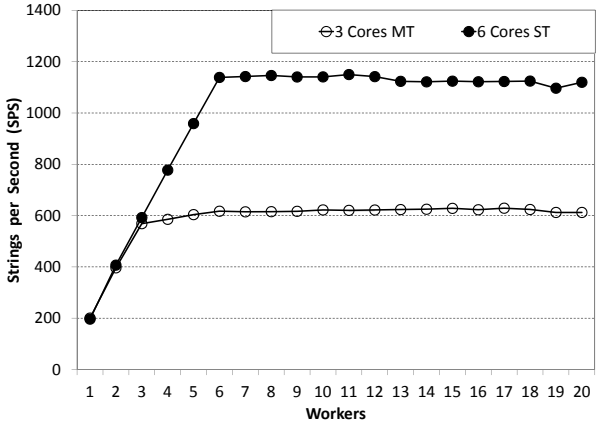
V. Experimental Results

A. Microarchitectural Impact

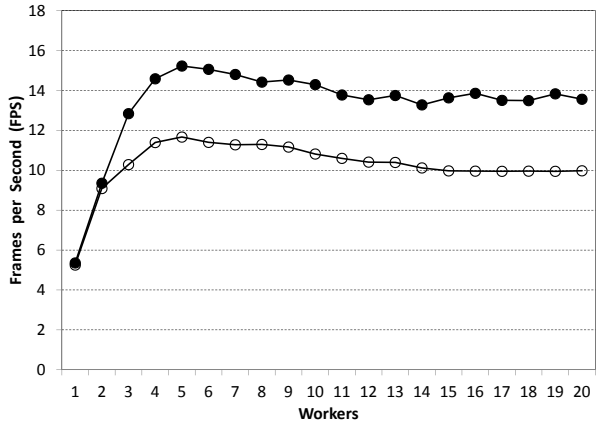
Figures 1(a) and 1(b) depict the performance of HashApp and RayApp, respectively, running in Chrome on two different microarchitectures. That is, multi–threaded (*n Cores MT*) and single–threaded (*n Cores ST*) multi–core processor, being *n* the number of cores available in the CPU. Y–axes denote performance measurements in terms of Strings per Second, SPS, for HashApp and Frames per Second, FPS, for RayApp. X–axes indicate the number of Web Workers.

Figure 1(a) plots higher performance running HashApp on multiple single–threaded cores (solid circle line) than using a multi–threaded multi–core architecture (open circle line), due to lower contention on shared hardware resources. The execution on a processor with single–threaded cores shows roughly double performance compared to multi–threaded multi–core architecture. In both cases, performance linearly scales up until there is a worker running on every core. Using more workers shows marginal or non performance improvement. Contention in hardware resources is sustained, since there are no synchronization barriers among threads and therefore Web Workers constantly demand shared hardware resources. Nevertheless, running more workers than logical cores in both microarchitectures does not degrade performance.

In contrast RayApp presents similar performance scalability on both microarchitectures as seen in Figure 1(b). Performance increments are limited by the inherent barriers, at every render frame, of the synchronous workers execution model, as explained in Section III. That is, since Web Workers are idle during periods of time, there is lower contention on shared hardware resources and therefore the performance difference among microarchitectures is lower. The single–threaded multi–core CPU presents up to 2.85x of speedup, using 5 Web Workers, whereas the multi–threaded architecture shows up to 2.22x of speedup, running the same number of workers. Unlike

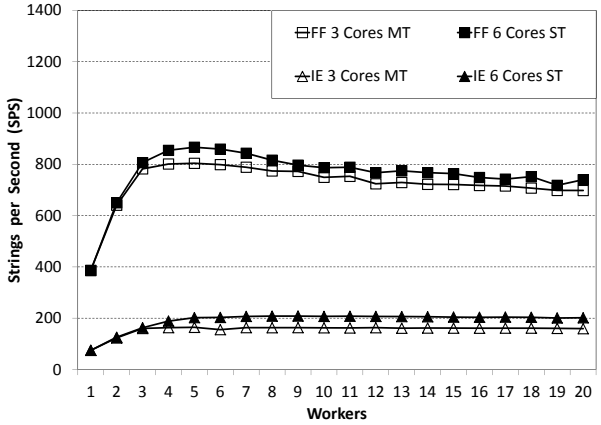


(a) HashApp

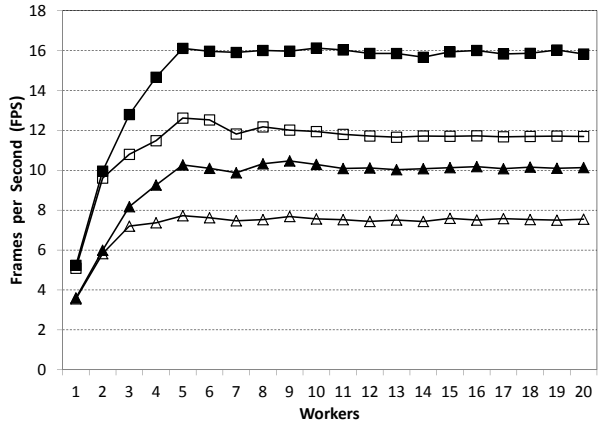


(b) RayApp

Fig. 1: Performance scalability on architectures with 6 logical cores



(a) HashApp



(b) RayApp

Fig. 2: Performance scalability on different web browsers

the asynchronous workers, running more Web Workers than logical cores slightly degrades performance.

This analysis suggests to JavaScript developers that they have to consider not only the number of logical cores, but also specifications of the processor architecture to estimate the optimal number of workers taking into account potential contention on shared hardware resources.

In addition, we use Chrome's memory profiling tools to analyze memory management, that is garbage collector behavior. HashApp has sustained memory consumption with reduced memory areas, few KBs, regularly cleaned during the execution, whereas RayApp presents larger memory sizes, tens of MBs, recurrently released. Nevertheless, due to space limitations, it is out of scope of this paper, but addressed in our future work, to analyze the garbage collector impact isolated from the impact of contention on shared hardware resources.

B. Web Browser Impact

Every web browser includes its own implementations and optimizations of the JavaScript virtual machine. This Section delves into the repercussion of using other web browsers, Firefox (FF) and Internet Explorer (IE), on the performance scalability. Figures 2 also present measurements using the same 6 logical core microarchitectures than Figure 1. Square lines stand for Firefox performance, whereas triangle lines denote IE performance. Besides, open and solid shape lines refer to multi–threaded and single–threaded multi–core architectures, respectively.

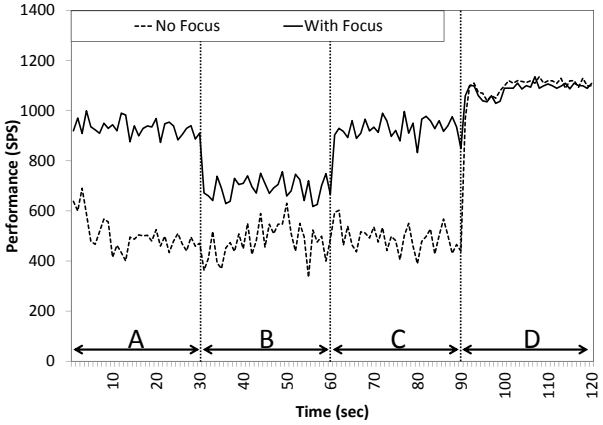
The impact of using single–threaded versus multi–threaded cores on the performance scalability of HashApp (Figure 2(a)) is much smaller in both Firefox and IE than in Chrome. Nevertheless, single–threaded multi–core CPU increases the performance of HashApp compared to multi–threaded cores up to 8% and 28% for Firefox and IE, respectively.

In spite of absolute performance differences among web browsers, Figure 2(b) indicates that multiple synchronous workers present similar performance scalability running in the three web browsers on both assessed architectures. That is, single–threaded multicores show a speedup of 3.07x (FF) and 2.91x (IE), while multi–threaded cores present a speedup of 2.47x (FF) and 2.17x (IE). All of them similar to the speedups obtained from Chrome on respective architectures.

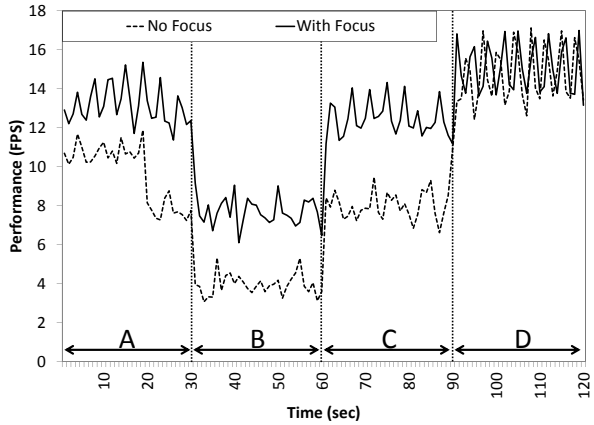
C. Co–Running Applications Impact

Scaling the number of Web Workers without having into account system workload variations due to co–running applications can either overload the machine or increase contention on shared hardware resources leading the system to a performance degradation phase.

Figures 3(a) and 3(b) depict the results of running HashApp and RayApp, respectively, in Chrome in conjunction with co–running applications using a CPU with 6 single–threaded cores. Each benchmark is setup to comprise 6 Web Workers. Vertical axes indicate performance and the X–axes denote the timeline in seconds. We present results of the benchmarks when the web browser's window



(a) HashApp



(b) RayApp

Fig. 3: Performance impact of co–running applications

has the focus (solid lines) and without the focus (dashed lines), that is a co–running application's window is selected.

The co–running applications are two different web browsers executing HashApp with 3 Web Workers each, that is half of the total available logical cores in the CPU. We use these co–running applications as representative multithreaded CPU intensive applications, especially parallel JavaScript web apps, that demand nearly 100% of the CPU when both run at the same time. As Chrome is the browser under study, the co–running applications are Firefox and IE to simulate a real system workload with multiple independent applications.

The timeline is divided into four stages labeled as *A*, *B*, *C*, and *D*. Every phase takes 30 seconds and denotes a particular status of the system workload. During the first period, *A*, there is a single co–running application that requires half of the resources. The second co–running application starts off execution at stage *B*. During this phase the CPU suffers the highest contention. The OS has to manage both co–running applications, that ask for the total number of logical cores, in addition to the benchmark, which also demands all logical cores. At the beginning of the period *C* the second co–running application finishes and thus there is theoretically similar resource contention than the step *A*. The other co–running application stops when the stage *D* starts. From then on, all hardware resources are available to the benchmark, since the web app is running alone, its performance is similar to the one shown in Figure 1.

On the one hand, when the benchmark has the focus it suffers slight performance impact running with one co–running application, nearly 15% for HashApp and 17% for RayApp, whereas the impact is more significant when there are two co–running applications, about 37% and 49% for HashApp and RayApp, respectively. On the other hand, when a co–running application has the focus instead of the benchmark's window, performance is significantly reduced. In fact, HashApp shows constant performance reduction, about 55% on average, while RayApp shows nearly 45% and 73% slowdown when there are one and two co–running applications, respectively. Besides, the performance of executions without focus, during stage *A*, is considerably reduced after 5 and 20 seconds on HashApp and RayApp, respectively, but sustained from then, including stage *C*.

VI. Conclusion

We presented the first performance scalability analysis of JavaScript web apps consisted of multiple workers classified according to their execution model. Our results demonstrated that current approaches to estimate the number of Web Workers of highly parallel web apps do not provide enough information to developers. An optimal worker pool size depends on the worker execution model, the underlying CPU architecture, and even web browser internals. In most cases few workers show similar or even slightly higher performance than spawning larger number of workers. Besides, co–running applications may significantly impact on the web app performance.

From the results of this paper we can conclude that dynamic mechanisms, such as performance monitoring based, are suitable to determine the optimal number of Web Workers. These type of approaches can detect at runtime performance scalability variations and contention on shared hardware resources due to co–running applications or workers overloading.

Acknowledgments

This work has been supported by the Spanish Ministry of Economy and Competitiveness (MINECO) under contract TIN2012–34557.

References

[1] S. Aronis, N. Papaspyrou, K. Roukounaki, K. Sagonas, Y. Tsiouris, and I. E. Venetis. A Scalability Benchmark Suite for Erlang/OTP. In *Procs. of the Eleventh ACM SIGPLAN Workshop on Erlang Workshop*, Erlang '12, pages 33–42, New York, NY, USA, 2012.

[2] K.Y. Chen, J.M. Chang, and T.W. Hou. Multithreading in Java: Performance and Scalability on Multicore Systems. *IEEE Trans. Comput.*, 60(11):1521–1534, November 2011.

[3] E. Fortuna, O. Anderson, L. Ceze, and S. Eggers. A Limit Study of JavaScript Parallelism. In *Procs. of IISWC*, pages 1–10, Dec 2010.

[4] Hickson, I. et al. HTML5 Specification. *http://www.w3.org/TR/html5*.

[5] J. Martinsen, H. Grahn, and A. Isberg. Using Speculation to Enhance JavaScript Performance in Web Applications. *IEEE Internet Computing*, 17(2):10–19, March 2013.

[6] Mozilla Developer Network. Web Workers API. *https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API*, March 2015.

[7] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An Analysis of the Dynamic Behavior of JavaScript Programs. *SIGPLAN Not.*, 45(6):1–12, June 2010.

[8] M. Russinovich. *http://technet.microsoft.com/sysinternals/bb896653.aspx*.

[9] StatCounter Global Stats. *http://gs.statcounter.com/*.

[10] D. Tiwari and Y. Solihin. Architectural Characterization and Similarity Analysis of Sunspider and Google's V8 Javascript Benchmarks. In *Procs. of ISPASS*, pages 221–232, Washington, DC, USA, April 2012.

[11] Y. Watanabe, S. Okamoto, M. Kohana, M. Kamada, and T. Yonekura. A Parallelization of Interactive Animation Software with Web Workers. In *Procs. of NBiS*, pages 448–452, September 2013.

[12] Web Hypertext Application Technology Working Group (WHATWG) Wiki. Navigator Hardware Concurrency. *https://wiki.whatwg.org/wiki/Navigator_HW_Concurrency*, July 2014.

[13] O. Zára. Hash Bruteforcer. *Demo Studio Mozilla Developer Network, https://developer.mozilla.org/es/demos/ detail/hash-bruteforcer*, April 2013.