# Dynamic Web Worker Pool Management for Highly Parallel JavaScript Web Applications

J. Verdú*,[1] and J. J. Costa[1] and A. Pajuelo[1]

[1]*Department of Computer Architecture, BarcelonaTECH (UPC), Barcelona, Spain*

SUMMARY

JavaScript web applications are improving performance mainly thanks to the inclusion of new standards by HTML5. Among others, Web Workers API allows multithreaded JavaScript web apps to exploit parallel processors. However, developers have difficulties to determine the minimum number of Web Workers that provide the highest performance. But even if developers found out this optimal number, it is a static value configured at the beginning of the execution. Since users tend to execute other applications in background, the estimated number of Web Workers could be non–optimal, since it may overload or underutilize the system. In this paper, we propose a solution for highly parallel web apps to dynamically adapt the number of running Web Workers to the actual available resources, avoiding the hassle to estimate a static optimal number of threads. The solution consists in the inclusion of a Web Worker pool and a simple management algorithm in the web app. Even though there are co–running applications, the results show our approach dynamically enables a number of Web Workers close to the optimal. Our proposal, which is independent of the web browser, overcomes the lack of knowledge of the underlying processor architecture as well as dynamic resources availability changes. Copyright © 2010 John Wiley & Sons, Ltd.

Received …

KEY WORDS: HTML5, Web Workers, JavaScript, web applications, parallelism, multithreaded

## 1. INTRODUCTION

Web applications follow the new HTML5 standard developed by the World Wide Web Consortium [1] to address the requirements of current and future platforms, web contents, and cloud services. HTML5 provides new HTML elements, libraries extensions, and APIs, paying special attention to JavaScript support, to take further advantage of the underlying platform through hardware accelerated browsers, as well as avoiding the need to install third–party plugins. As a result, current web apps are reducing the performance gap with respect to native applications.

The execution model of a web app depends on the language used. We focus on JavaScript, that is an interpreted language with event–driven single–thread execution model. Many web apps show potential speedups of up to 45.5x compared to sequential execution, due to parallelism exploitation [2]. Parallelism can be extracted from a sequential JavaScript program using Thread–Level–Speculation, TLS, either in hardware [3, 4, 5] or software [6, 7, 8].

But, HTML5 brings a new mechanism and API, called Web Workers [9, 10], that allows JavaScript codes to concurrently run in background threads, from now on workers or worker threads, in conjunction with the main thread, aka parent thread. Workers are communicated with the parent thread by message passing. Unlike other approaches that exploit parallel execution, the developers

*Correspondence to: jverdu@ac.upc.edu

are responsible to extract the parallelism, create threads, and develop a code that properly exploits the parallelism.

JavaScript web apps, based on Web Workers, use to have the main thread focused on the user interface (UI) and I/O tasks, since workers cannot do it due to access constraints [9]. Thus developers offload compute intensive tasks to background threads to preserve responsiveness and enhance user experience. If compute intensive workload would not be offloaded from the main thread, the UI could eventually be blocked during periods of time. On the one hand, developers can create parallel web apps with a single background thread. This approach is currently followed by most of the HTML5 videogames, where physics and/or artificial intelligence engines are executed in a single worker. On the other hand, highly parallel applications, such as image/video processing, are able to spawn multiple threads to exploit massively parallel CPUs. In this case, the compute intensive workload is distributed among multiple Web Workers that can execute at the same time.

Unfortunately, when using multiple workers, there is no mechanism provided by web browsers that dynamically manage the number of running workers according to some key performance metrics. Thus, developers need to find out how many available hardware resources has the computer to estimate how many workers should be spawned to get the highest performance. Hardware information can be obtained using recent releases of some web browsers, like Google Chrome, Safari, and Opera, that include a new attribute or executing a core–estimator benchmark on web browsers, such as Internet Explorer and Firefox, that do not support it yet. Both of them can provide the number of logical cores available in the system, but cannot determine how resources are shared. That is, whether or not cores are hyperthreaded.

Unlike native multithreaded applications, the behavior and performance of JavaScript web apps are totally dependent on internals of every particular web browser's virtual machine. Thus, there is no direct relation between workers and running threads in the browser and the utilization of logical cores in the processor.

Moreover, most of the web apps run on personal computers with unexpected system workload variations, since it is very likely users execute other applications while the web app is running. Thus, even though developers are able to estimate how many logical cores the computer has, they have no knowledge of how resource availability changes during the execution lifetime. Since the number of Web Workers to run is usually fixed when the web app starts, several problems may arise. On the one hand, if the web app spawns as many workers as logical cores, when another application starts running, applications can significantly collide on shared hardware resources, especially if it is a multithreaded process, and thus overload the system. On the other hand, if the developer shortens the number of Web Workers just in case there are co–running applications, when they suddenly stop, resource availability is higher. Then, increasing the number of worker threads could provide higher performance.

This paper presents the first proposal that dynamically manages a pool of Web Workers for highly parallel JavaScript web applications. Our approach consists in modifying the web apps that execute multiple workers by embedding a simple algorithm that manages a Web Worker pool. The goal is twofold: 1) to avoid the hassle to estimate a fixed optimal number of threads, and 2) to dynamically adapt the number of running threads to the really available resources. The algorithm analyzes performance monitoring. If it is higher than a given threshold the manager activates an additional thread from the pool of idle Web Workers. Due to this, we also demonstrate that idle worker threads have a negligible impact on web app performance. The results show that this browser independent proposal overcomes the lack of knowledge of the underlying processor architecture, such as how many logical cores and how hardware resources are shared, as well as dynamic changes on resources availability due to co–running applications.

The rest of this paper is organized as follows. Section 2 introduces different execution models of Web Workers. Section 3 in great details presents the dynamic worker pool manager. Section 4 presents the evaluation framework used in our paper. Section 5 analyzes the results under different scenarios. Finally, Section 6 describes the related work, and Section 7 presents the conclusions.



DOI: 10.1002/cpe

## 2. BACKGROUND

Parallel JavaScript web apps comprise the main thread, responsible of the UI, since workers cannot do it due to access constraints [9], and background threads for Web Workers aimed at compute intensive tasks to preserve responsiveness and enhance user experience.

Besides, Web Workers API distinguishes two types of workers: a dedicated worker, aka standard worker, that is only accessible by the script that spawned it; and a shared worker that is accessible from any script running in the same domain. Other emerging types of workers are still experimental [9].

However, regardless of whether workers can be accessed from one or several scripts, we can classify parallel web apps based on worker execution models, namely:

- **Single worker:** all compute intensive tasks are done by a single background thread. Videogames, for example, need to offload CPU intensive tasks, like AI and physics, from the UI thread to sustain responsiveness and frame rate [11]. Web apps that comprise multiple compute intensive tasks suitable to run in several cooperative threads are candidates to be split into multiple workers and thus to be reclassified into 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 completes the current task, it sends a notification message to the parent thread and a new workload is delivered to the Web Worker for processing. That is, there are no idle workers if there is pending work.
- **Multiple synchronous workers:** inherent highly parallel codes, such as image/video processing, use to have a synchronization point among workers, like presentation of a new frame. Every new workload, a frame, can be split into multiple jobs, slices, to be processed by different Web 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.

## 3. DYNAMIC WEB WORKER POOL MANAGER

Thread pool management has been extensively studied in a variety of areas, such as multithreaded server applications. Algorithms try to dynamically decide what is the optimal number of active threads to sustain required high performance. However, proposals that deal with native threads try to minimize the number of idle threads to reduce management overhead [12, 13].

Our proposal is based on the fact that Web Workers run in a browser's virtual machine. Therefore, depending on how idle threads are managed by the virtual machine, the performance impact of idle workers can be negligible, as we analyze in Section 5.2. This Section presents the implementation of our approach. First, we introduce the proposal. Then, we present and discuss configuration parameters and, finally, we describe the algorithm.

### *3.1. The Proposal*

The dynamic Web Worker pool manager consists of an algorithm that makes a decision to activate threads from the pool of workers according to the comparison of current average performance with the performance mean at the time of the previous taken decision. To prevent the overhead of creating additional workers, the developer has to create a pool of idle workers at the beginning of the execution. Thus, the manager will determine to activate or not an additional thread of the pool of workers to start processing in conjunction with other active Web Workers.

Most of the highly parallel JavaScript web apps already have a pool of workers. Nevertheless, the pool has to be modified to create a pool of Web Workers in idle status. That is, the parent thread creates the Web Workers as usual, using the constructor $Worker()$, but does not send any message to the new threads to notify them to start processing.




DOI: 10.1002/cpe

As soon as the execution starts, the approach constantly tracks performance of the web app. Thus, a FIFO queue has to be created in order to collect measurements to calculate temporary average performance of the web app. In case the application already includes routines to measure performance, the code has to be modified to put the performance into the queue.

When the queue is full it means there are enough measurements to calculate a representative performance mean. Then, the algorithm analyzes performance variations and decides whether to increase the number of active Web Workers or not. That is, the parent thread enables a worker to start processing by sending a message, with the method $postMessage()$.

### 3.2. Configurable Parameters

Table I presents the configurable parameters that will be used to configure the algorithm. We assign them default values based on experiments done during preliminary phases of this study. However, it is out of the scope of this paper to find the optimal parameter setup, since it depends on the goals and requirements of every particular web app.

Table I. Configurable Parameters

| Parameter | Value | Description |
|---|---|---|
| $TQLength\_Limit$ | 4 | Maximum length of the queue with performance measurements |
| $\alpha$ | 1.10 | Speedup factor threshold |
| $\beta$ | 0.85 | Slowdown factor threshold |

$TQLength\_Limit$ delimits how many performance measurements have to be collected to calculate a representative average performance. The higher values the less sensitive to unexpected significant throughput variations. On the contrary, short queue length leads to high sensitivity to sudden throughput variations. According to our experiments running the benchmarks used in this paper, a queue length of 4 items is enough to have a representative average performance.

The speedup, $\alpha$, and slowdown, $\beta$, factor thresholds define the performance variation rates to take a given decision. On the one hand, $\alpha$ denotes a particular speedup threshold that leads to consider increase the number of active workers. On the other hand, $\beta$ determines the slowdown threshold that leads to identify lower resource availability, mainly due to co–running applications. For both of them, larger values mean the variation has to be more significant. That is, a conservative scaling setup, since the algorithm only takes a decision when there are important performance variations. Whereas smaller values lead to eager scaling configuration, because the algorithm takes decisions even though there are small performance variations. The default values of $\alpha$ and $\beta$ shown in Table I are assigned from the analysis of preliminar experiments of this study.

### 3.3. The Algorithm

Throughout this Section we describe the pseudocode along with a brief explanation of the variables and their purpose. Figure 1 shows our proposed algorithm and Table II presents a brief description of the variables. The algorithm has to be included in the source code of the web app, specifically in a given function executed every period of time decided by the developer, for example every second.

Table II. Variables of the Algorithm

| Variable | Description |
|---|---|
| $PerformanceQueue$ | Queue that holds the $QLength\_Limit$ recent performance measures |
| $CurAvgPerformance$ | Current average performance |
| $PrevAvgPerformance$ | Average performance in the last taken decision |
| $ActiveWorkers$ | Number of active threads |
| $FakeDecrements$ | Number of fake decrements of active Web Workers |

$PerformanceQueue$ is a queue that holds the most recent performance measurements using the current configuration of Web Workers. That is, the current number of active threads. The length of



DOI: 10.1002/cpe

the queue is defined by the $TQLength\_Limit$ parameter that specifies how many measurements are required to calculate representative average performance. While the queue is not full, the algorithm is not executed, but the $PerformanceQueue$ is updated, using FIFO policy, to include the latest performance measurement. Once the queue is full, $CurAvgPerformance$ is updated with the performance mean, calculated from the measurements collected in the queue, and the algorithm is executed.

Firstly, the algorithm compares $CurAvgPerformance$ with $PrevAvgPerformance$, the average performance calculated at the time of the last taken decision. If the current mean is higher than a given speedup threshold, the $\alpha$ parameter, the algorithm activates one of the idle workers by increasing the $ActiveWorkers$ counter. The parent thread uses this variable to identify active workers in order to distribute new workload.

On the contrary, if $CurAvgPerformance$ shows a slowdown of at least $\beta$ compared with $PrevAvgPerformance$, the manager simulates to decrement the number of active workers. That is, instead of disabling an active worker, to be again an idle thread, the algorithm counts the number of fake decrements with the $FakeDecrements$ counter. The purpose of this variable is to prevent the activation of additional workers as long as $FakeDecrements$ is greater than 0, as shown in lines 2– 6 in Figure 1. In this case, the algorithm reduces the number of $FakeDecrements$ instead of increasing $ActiveWorkers$ counter.

The Operating System (OS) performs load balancing to distribute the CPU usage among the running applications, paying special attention when they collide in shared hardware resources, the logical cores. According to our experiments, if the number of active workers actually decrements, the OS reduces the CPU usage rate assigned to the web app, since the OS identifies lower CPU demanding of this application. As a result, the web app performance is much lower. However, with the use of fake decrements we prevent that the OS reduces CPU assignment to the web application, as well as we also prevent to overload the system with too many threads.

When the manager makes a decision, it also performs two actions, namely: update the $PrevAvgPerformance$, since it will be used for speedup and slowdown comparison in subsequent executions of the algorithm; and flush the $PerformanceQueue$, since the queue has to collect performance measurements of the new configuration of Web Workers. Otherwise, new performance mean would be deviated by the measurements obtained with an obsolete configuration, the previous number of active threads.

**Algorithm:** Dynamic Web Worker Pool Manager

**Require:** $CurAvgPerformance \leftarrow Avg(PerformanceQueue)$

1: **if** $CurAvgPerformance > (PrevAvgPerformance * \alpha)$ **then**
2: **if** $FakeDecrements == 0$ **then**
3: **ActiveWorkers++**
4: **else**
5: $FakeDecrements - -$
6: **end if**
7: $PrevAvgPerformance \leftarrow CurAvgPerformance$
8: $Flush(PerformanceQueue)$
9: **else if** $CurAvgPerformance < (PrevAvgPerformance * \beta)$ **then**
10: $FakeDecrements + +$
11: $PrevAvgPerformance \leftarrow CurAvgPerformance$
12: $Flush(PerformanceQueue)$
13: **end if**

Figure 1. Pseudocode of the Dynamic Management Algorithm of a Web Worker Pool



DOI: 10.1002/cpe

## 4. EVALUATION FRAMEWORK

### 4.1. Platform

We use a personal computer with an Intel® Core™ i7-3960X processor at 3,3GHz consisted of 6 hyperthreaded cores, for a total of 12 logical cores, with 16GB DDRAM-III and a Nvidia® GTX560 videocard. The machine has two partitions with Microsoft® Windows™ Server 2008 R2 and Ubuntu 14.04 LTS. 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 [14], by the use of Set Affinity, in Windows and the command *taskset* in Ubuntu to configure the available logical cores from the point of view of the web browsers executed to mimic a particular CPU. The configured CPU consists of 8 threads distributed along 4 hyperthreaded cores. On the one hand, the selected logical cores only execute the web browsers, whereas the non–selected logical cores execute critical services and applications. Thus, we reduce even more the deviation in the measurements. On the other hand, 8 logical cores, in the form of 4 hyperthreaded cores, can mimic the behavior of a multithreaded processor that is representative of the currently latest generation of Intel® Core™ i7 family processors.

We employ updated releases of the three most used web browsers [15]: Google Chrome v42.0.2311.90m, Mozilla Firefox v37.0.2, and Microsoft Internet Explorer v11.0.9600.16476, from now on Chrome, Firefox, and IE, respectively.

### 4.2. Benchmarks

There are no standard JavaScript benchmarks comprising Web Workers. Several well known web apps and web browsers portals provide parallel JavaScript demos, like Demo Studio Mozilla Developer Network. Most of the demos are implemented with dedicated workers. In fact, 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 2). We have run several web apps that have no human interaction requirements. This paper delves into the analysis of two case studies. 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 [16], HashApp from now on, is a web app 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 get the plain text. Thus, the workers perform continuous CPU intensive workload with no synchronization barriers among threads. The main thread sends messages to every particular Web Worker to notify the next piece of work, while waits for receiving messages from them. Additionally, it is also responsible to measure performance and execute our proposal every second.
- **Multiple synchronous workers:** The raytracer web app [17], RayApp from now on, performs highly CPU intensive mathematical calculations to simulate components of the 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 in order to generate representative number of workload to be executed on a CPU with 8 logical cores. 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 size of workers pool. We have done experiments with others canvas sizes, showing different performance numbers, but with similar trends. Unlike HashApp, the main thread has more work to do, since it is also in charged of capturing and sketching the image on the canvas. That is, it draws the data of a given image onto the bitmap to show a new frame, because Web Workers cannot use functions for the canvas [9].

We properly modify the source code of both benchmarks to include our proposal described in Section 3. Since the original source code has static pool of Web Workers, configured by parameter, we change it to be dynamically managed by our proposed algorithm. For this end, we include



DOI: 10.1002/cpe

additional data structures and configuration parameters. The algorithm has been implemented to be executed every second. In particular, we modify HashApp to implement the algorithm in the method that updates the performance statistics, *App.prototype._updateStats*, whereas we develop the algorithm in the *_camera.onFinished* method, called after rendering a new frame in RayApp.

Even though the original benchmarks already have functions to measure performance, we modify the source code to use the most accurate mechanism for JavaScript time measurement. The WebAPI *Performance.now()* returns a DOM High Resolution TimeStamp accurate to microseconds [18], and it is available in current releases of major web browsers.

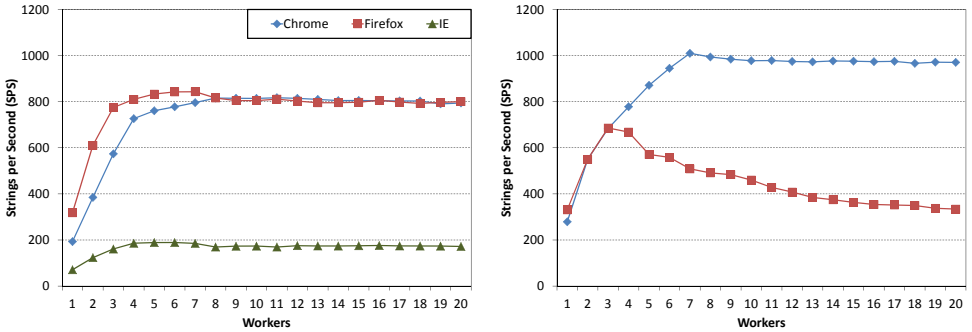
The benchmarks have been configured to spawn from 1 to 20 Web Workers. We fix this limit since it is the maximum number of workers that Firefox supports. Nevertheless, experiments with more workers running on others browsers show no performance improvements.

The benchmarks provide a toggle button to start and finish execution. We take average results from ten runs of every experiment running 30 seconds each, time enough to reach steady performance.

## 5. EXPERIMENTAL RESULTS

This Section analyzes the results obtained following few steps. Firstly, we present the performance scalability for assessed web browsers in Windows and Linux. The results will determine what is the optimal number of Web Workers in every case. Then, we study the impact of idle Web Workers on the performance of running workers. Finally, Sections 5.3 and 5.4 evaluate our proposal when the benchmarks run alone and with co–running applications, respectively.

### *5.1. Performance Scalability*




(a) Windows (b) Linux

Figure 2. Performance Scalability of HashApp running on a CPU with 8 logical cores

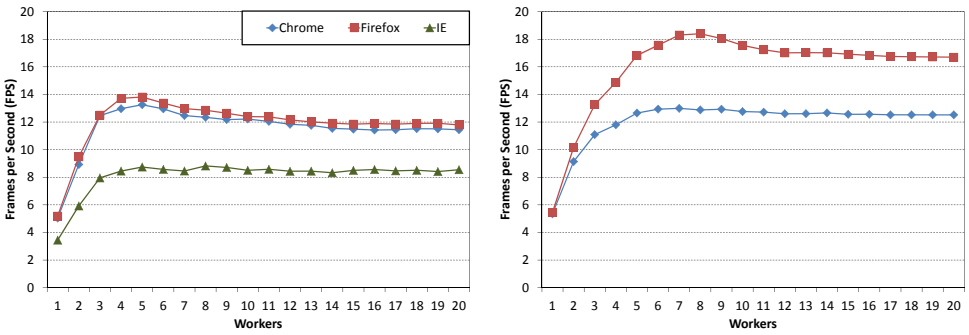
Due to the lack of reference measurements, we present the performance scalability of HashApp and RayApp in Figure 2 and Figure 3, respectively, for Chrome, Firefox, and Internet Explorer. The Y–axes indicate performance metrics, in terms of completed work per unit of time, and the horizontal axes show the number of running workers. Finally, left charts depict results in Windows, whereas the right graphs denote measurements using Ubuntu.

In spite of the diversity of absolute performance measurements, the results confirm that scalability trends along with the number of Web Workers are significantly different among browsers and OSes. This difference is highly evident for HashApp in Windows, Figure 2(a), and in Linux, Figure 2(b). Whereas in Ubuntu, Firefox presents important performance degradation using large number of workers, there are no negative effects in Windows for any web browser.

Besides, Chrome and Firefox, in Windows, show an almost linear performance improvement up to 3-4 workers, but then it reaches a stable point. The main reason behind this behavior is that running more threads increase collision on shared hardware resources. That is, running more threads do not provide significant performance improvements.



DOI: 10.1002/cpe




(a) Windows (b) Linux

Figure 3. Performance Scalability of RayApp running on a CPU with 8 logical cores

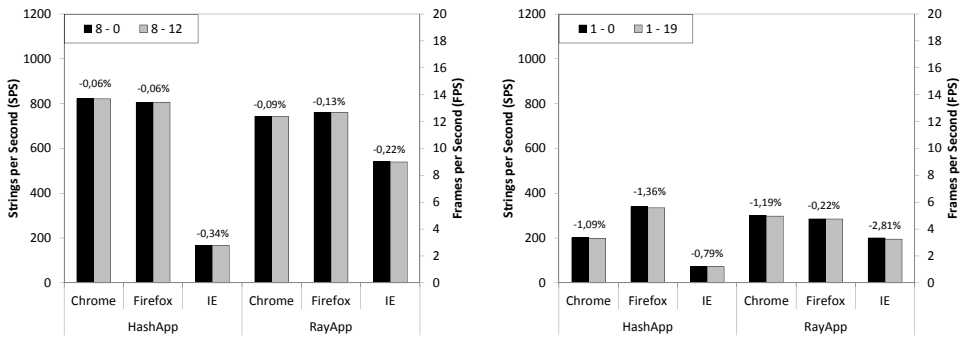
The benchmark with synchronous workers, Figure 3, shows slight performance degradation using large number of threads. In fact, if the number of threads is larger than optimal, there are two drawbacks. Firstly, every worker execute less number of work, slices of a frame. In a scenario where there are more threads than pieces of work, only a subset of the threads have work to be done. Thus, no further performance scaling is obtained by increasing the number of threads beyond the optimal. Secondly, the contention on shared hardware resources increases during the processing, since there are more threads that collide. Therefore, there is a point where performance is even downgraded.

Finally, the results also show that IE presents reduced performance scalability in both benchmarks. In fact, it is lower than linear in the best case.

These results lead to three main insights: 1) browsers show different performance scalability trends; 2) the behavior of synchronous versus asynchronous workers directly impact on performance scalability; and 3) even if the CPU has a large number of logical cores, most of the runs reach the highest performance using less number of workers than available logical cores, that can also be different for every web browser. As a result, it is not straightforward to fix a single generic number of Web Workers that involves the optimal tradeoff between number of workers and performance.

### *5.2. Idle Workers Impact*

Thread pool management, especially on network servers, is sensitive to the overhead of managing large idle thread pools [13, 12]. As our proposal requires a pool of Web Workers that are dynamically enabled, we need to previously analyze the impact of idle workers on active threads. In fact, when a Web Worker is created, it is idle. Since JavaScript is an event–driven programming model, the thread waits until it receives an event. As soon as this event happens, by the reception of a message from the parent thread, the Web Worker is activated and starts processing.

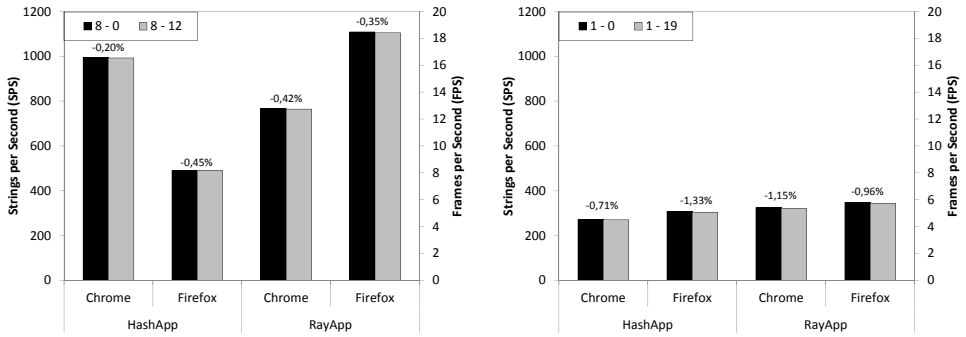



(a) 8 Logical Cores (b) 1 Logical Core

Figure 4. Impact of idle workers on performance of processing workers in Windows



DOI: 10.1002/cpe




(a) 8 Logical Cores (b) 1 Logical Core

Figure 5. Impact of idle workers on performance of processing workers in Linux

Figures 4 and Figure 5 depict the performance for both benchmarks and web browsers running in Windows and Linux, respectively. Left Y–axis, SPS metric, is followed by HashApp, while RayApp uses the right Y–axis, FPS metric. Both web apps are configured to comprise "N-M" worker threads, where N stands for the number of active workers, while M denotes the number of idle workers. Both number of Web Workers are fixed at the beginning of the execution. Black bars represent the executions with 8 and 1 active threads, but with no idle workers left and right Figures, respectively. Grey bars denote measurements using the configuration with the same number of active workers, but with 12 and 19 idle workers, respectively. Both setups have a maximum of 20 workers, because it is the upper limit that Firefox has. Chrome and Internet Explorer have no well known upper limits on the number of worker threads handling.

Figure 4(a) and Figure 5(a) show results using 8 logical cores. Thus, 8 worker threads may collide in shared hardware resources even there are no idle workers. The performance degradation is negligible, from 0.45% to 0.06% for HashApp, and from 0.42% to 0.09% for RayApp. That is, even if the JavaScript virtual machine needs to manage the idle worker threads, it has almost no impact on the performance.

In Figure 4(b) and Figure 5(b) we totally stress a single hyperthreaded core, two logical cores. In this case, the UI thread, the active Web Worker, and 19 idle worker threads are bound to the same core. That is, it is more likely that the management of a larger number of idle workers can disturb to either the active worker or the UI thread. Even though this is a worse scenario, the impact of managing 19 idle workers ranges from 1.36% to 0.71% and from 2.81% to 0.22% performance degradation running HashApp and RayApp, respectively.

These results demonstrate that developers can create a large pool of Web Workers when the web app is launched, with negligible impact on the performance, and then dynamically decide how many of them are suitable to do efficient work to get the optimal performance.

### *5.3. Automatic Web Worker Scaling*

Figure 6 presents the performance scalability of the benchmarks properly modified to include our proposal. The Y–axes denote performance, whereas the horizontal axes represent the timeline in seconds. Every shape in the lines is a performance measurement put into the $PerformanceQueue$. Dashed circles indicate the pool manager decides an additional Web Worker has to be enabled, since there is an average performance higher than the configured speedup threshold, $\alpha = 1.10$. Numbers close to the circles mean the new number of running workers. Finally, the steady performance as well as the final number of running workers of these charts have to be compared to the performance scalability graphs in Figure 2 and Figure 3.

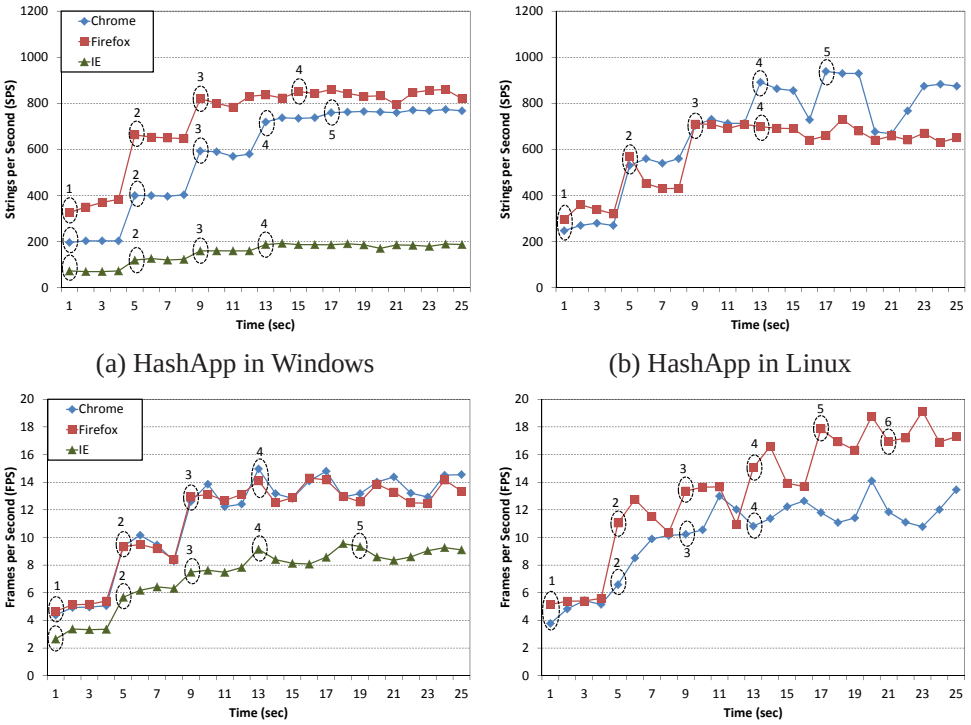
It is important to remark that both benchmarks run a long execution. Therefore, there can be differences among performance measurements taken at different instants. On the one hand, the benchmark processes diverse pieces of work at every instant with slight different characteristics. On the other hand, the more threads running the more likelihood to present performance variations since



*Prepared using* cpeauth.cls
DOI: 10.1002/cpe

collision on shared hardware resources may differ. Besides, the internals of the JavaScript virtual machine, like runtime optimizations and garbage collector, can eventually alter the performance at different moments.

The algorithm is able to find the optimal number of workers for both benchmarks when running on IE. The manager is also able to find a close to optimal configuration executing RayApp in Firefox for both OSes. However, for the asynchronous benchmark the algorithm scales the number of running workers up to almost optimal configuration. Thus, performance difference with respect to the actual optimal setup is under 4% in the worse case. The reason why the algorithm has not scaled the number of threads is that the performance improvements are lower than $\alpha$ when the configuration reaches a stable performance scaling point. That is, increasing the number of workers slightly scale or just sustain the performance.




(a) HashApp in Windows

(b) HashApp in Linux

(c) RayApp in Windows

(d) RayApp in Linux

Figure 6. Performance scalability using the Dynamic Web Worker Pool Management

Chrome has similar results for both benchmarks. That is, the algorithm is not able to identify the need to increase the number of running workers when the web app executes 5 and 4 workers for HashApp and RayApp, respectively. Therefore, if $\alpha$ would had a lower speedup factor, the algorithm would detect the required performance speedup that leads to enable an additional Web Worker. The difference of average performance between our configuration and the optimal setup on Chrome is less than 7.36% and 2.23% for HashApp and RayApp, respectively.

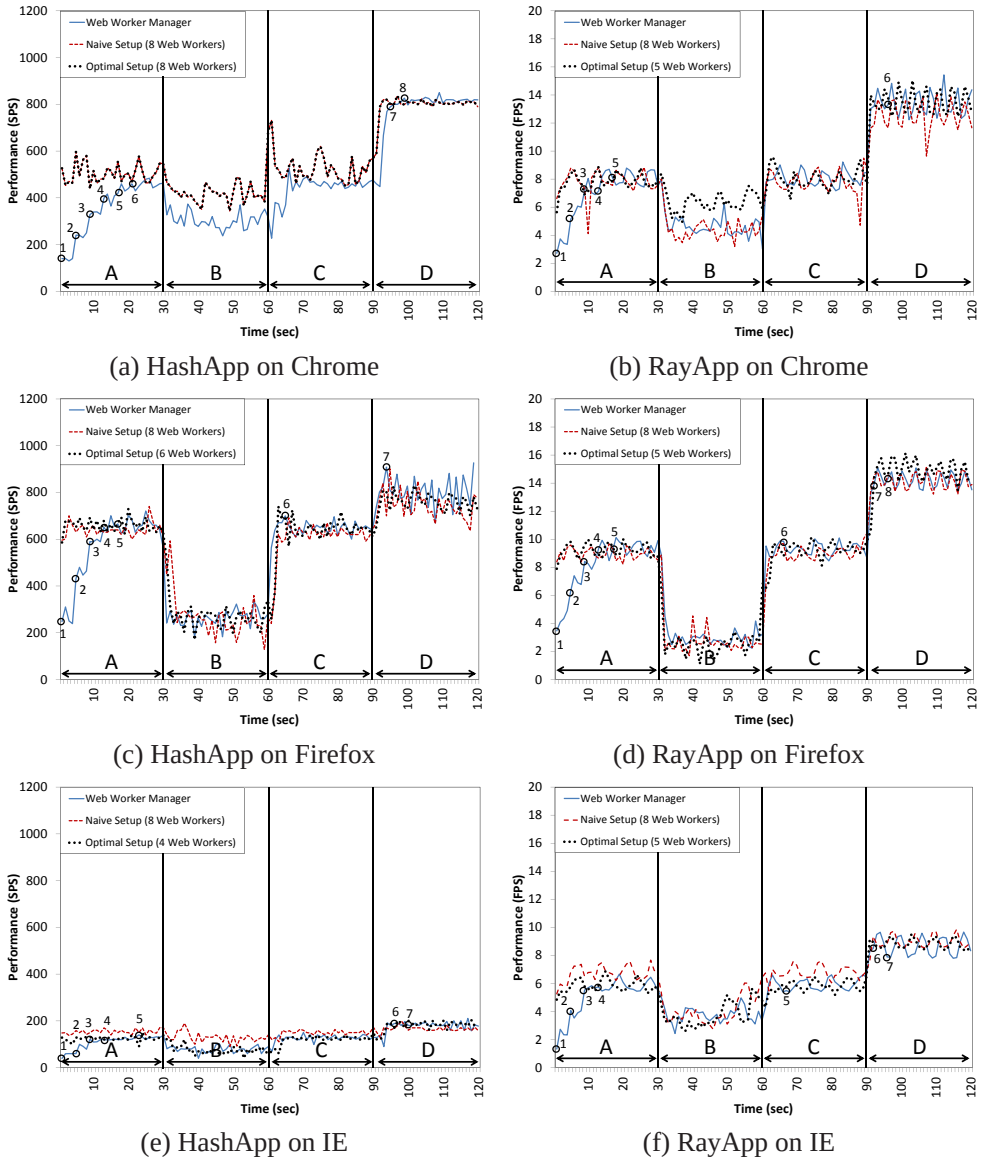
### *5.4. Execution with Co–Running Applications*

This Section presents the results of running the web app in conjunction with co–running applications. Figure 7 and Figure 8 present the results of running the benchmarks in Windows and Linux, respectively. Vertical axes indicate performance, whereas the X–axes denote the timeline in seconds.

The co–running applications are two different web browsers executing the original HashApp, without our proposal, running 4 Web Workers. We select these co–running applications as case



DOI: 10.1002/cpe




(a) HashApp on Chrome

(b) RayApp on Chrome

(c) HashApp on Firefox

(d) RayApp on Firefox

(e) HashApp on IE

(f) RayApp on IE

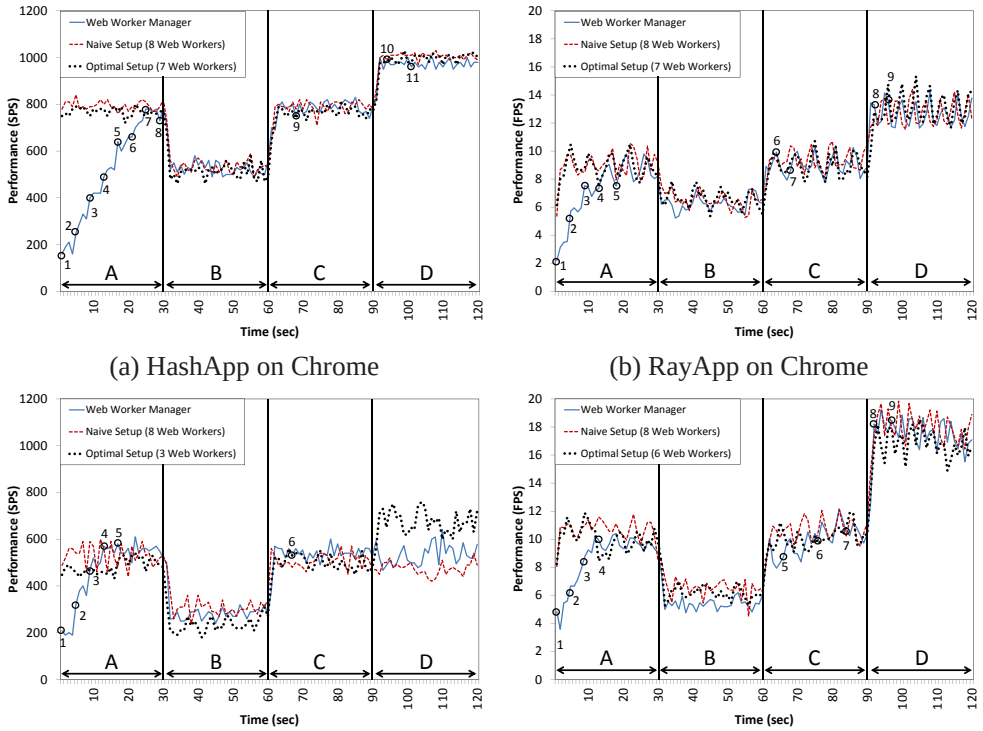
Figure 7. Web app performance with co–running applications in Windows on Chrome (Figures (a) and (b)), Firefox (Figures (c) and (d)), and Internet Explorer (Figures (e) and (f))

study of representative workload, since they are multithreaded CPU intensive applications. That is, the execution of both of them consume nearly 100% of the total of 8 logical cores available in the CPU. We use the two web browsers that are not studied for the particular experiment. For example, if we analyze the performance on Chrome, the co–running applications are Firefox and IE, both of them running HashApp.

The timeline is divided into four slots labeled with “A”, “B”, “C”, and “D”. Every slot takes 30 seconds and identifies a particular status of the system workload. In the first slot, “A”, there is a single co–running application and thus only half of the resource availability is theoretically available. When slot “B” starts, the second co–running application initiates execution. Therefore, the OS has to manage three multithreaded CPU intensive applications that demand more hardware resources than available in the system. At the beginning of slot “C” the second co–running application finishes and there is similar resource availability than the slot “A”. Finally, the remaining co–running application



*Prepared using* cpeauth.cls
DOI: 10.1002/cpe




(a) HashApp on Chrome (b) RayApp on Chrome

(c) HashApp on Firefox (d) RayApp on Firefox

Figure 8. Web app performance with co–running applications in Linux on Chrome (Figures (a) and (b)), and Firefox (Figures (c) and (d))

stops when the slot "D" starts. Therefore, all hardware resources are available to the benchmark, since the web app is running alone in the last slot.

The Figure also shows three different configurations for each benchmark, namely: Naive Setup, spawns as many workers as logical cores comprised in the CPU; Optimal Setup, runs the number of workers that provide the highest performance from the analysis of performance scalability in Section 5.1; and Web Worker Pool Manager, includes our proposal. When the pool manager decides to increment the number of running workers, a circle denotes the decision time as well as a number indicating the new number of running threads.

On the one hand, we can see that due to lower resource availability, slots with co–running applications, specially the slot "B", present significantly lower web app performance. Therefore, in those cases, running large number of workers can overload the system and downgrade the performance of all applications. The OS manages all processes to properly schedule them to use the logical cores. Thus, the more threads that intensively demand CPU the lower CPU usage for every thread and the lower performance of every application, since the system is at performance degradation phase due to overloading. Nevertheless, we can observe that the impact of two co–running applications, slot "B", is different on every browser.

We can see the three configurations provide similar performance, but using diverse number of worker threads. This difference is accentuated on HashApp charts, Figures 7(a), (c), and (d), that remark difference among the three web browsers on performance and optimal Web Worker configuration. The Optimal Setup is different on every browser, namely: 8, 6, and 4 threads on Chrome, Firefox, and IE, respectively. Nevertheless, our proposal finds out an number of running Web Workers close to the optimal. In all experiments that there are co–running applications, except HashApp on IE, Figure 7(e), that runs an additional worker than the optimal configuration, our proposal enables the minimum number of workers, equal or even lower than the Optimal Setup, but provides similar performance than the other configurations. Our approach alleviates the system



DOI: 10.1002/cpe

by reducing the contention on shared hardware resources, since there are less CPU intensively demanding threads.

Although there are significant performance variations when co–running applications start or finish the execution, our approach is able to detect whether it is advisable to increase the current number of running threads. For example, from slot "B" to "C", there is an important performance improvement, but the manager detects that it previously simulated to reduce the number of active workers, with the $FakeDecrements$ counter. That is, the performance was lower than the configured slowdown threshold, $\beta$ = 0.85. Therefore, the manager only enables an additional Web Worker when actually improves the performance with respect to the previous highest average performance.

With our default configuration of the parameters the algorithm provides good results on the three web browsers. That is, JavaScript developers can implement optimal highly parallel web apps, browser independent, without the hassle to do complex estimations. However, the main problems come from running on IE. The reason behind this is that the performance is slow and with our default configuration, slight performance increments result in speedups higher than 1.10x. Therefore, although the resulting number of Web Workers is between the Optimal and the Naive setups, the manager could be improved using lower values of $\alpha$ and $\beta$. Nevertheless, deep analysis of tuning the configuration parameters is out of the scope of this paper.

Figure 8 shows similar results for Linux. With the default configuration of the parameters the manager properly detects performance improvement to increase the number of Web Workers. The only case that presents significantly lower performance than the optimal configuration is running HashApp on Firefox 8(c). Due to co–running applications the manager wrongly decides to increase the number of workers more than the optimal number. Thus, the performance is about 15% lower than the optimal configuration. Nevertheless, it is higher than the Naive setup.

## 6. RELATED WORK

Some authors propose fine grain parallelization of JavaScript codes. Fortuna et al. [2] analyze the potential speedup limit of parallelizing events and tasks, such as functions and loops. Martinsen et al. [19] though propose Thread–Level–Speculation for browsers' JavaScript engines to take advantage of multicore multithreaded CPUs. Other authors propose a technique to parallelize interactive JavaScript animations using Web Workers [11]. But there is no study of threads scaling.

None of these works are focused on either analyze the performance scalability of workers based web apps or related differences among major web browsers. Besides, according to our knowledge, there is no proposal to dynamically manage a pool of Web Workers to sustain efficient computing.

Even if this is the first proposal that manages a pool of Web Workers for JavaScript web apps running on personal computers, thread pool systems have been commonly adopted in other areas, like grid computing and server applications.

Grid computing present, among others, problems of scheduling and allocating resources to jobs. In this sense, our proposal present some similarities with previous research of this area. The resource allocation is a process of task scheduling to available resources, code and data transferring, and monitoring of available resources and application performance. There is a large list of Grid resource allocation mechanisms [20]. Although our proposal also presents dynamic resource allocation based on an algorithm to optimize performance and aware of variations on available resources, we focus our work on a significantly different scenario. Grid systems deal with distributed computers with different number of resources and have to take into account requirements significantly different to ours, such as communication overhead. However, our manager only monitors performance variations in order to activate a new thread.

In the area of server applications, like Apache and Microsoft IIS, Xu et al. [12] propose dynamic thread pool management for server applications. Although their algorithm is slightly similar to our approach, the management is substantially different. They are focused on server environments and thus performance is not disturbed by unknown applications that suddenly start running. Besides, studies that directly deal with threads, instead of Web Workers, take into account the overhead of managing large pools of threads. Ling et al. [13] characterize costs related to the thread pool




DOI: 10.1002/cpe

management in order to analytically estimate the optimal thread pool size for network servers to respond larger number of simultaneous service requests. However, our results show that, due to the internals of the JavaScript virtual machines, the impact of many idle worker threads on the performance is negligible.

Unlike all these works, our goal is focused on providing a simple and generic approach that overcomes the difficulties of JavaScript developers to implement highly parallel web applications making an efficient thread scaling in most circumstances as well as independent of the browser.

## 7. CONCLUSIONS

Developers of highly parallel JavaScript web applications have many difficulties to develop applications with a generic configuration that spawns the minimum number of threads that provide the highest performance. The lack of information of the underlying processor architecture, such as how many cores are available and how resources are shared, as well as web browser internals, such as how the JavaScript virtual machine optimizes codes and scales the performance along with Web Workers, lead to very difficult estimation of the optimal number of workers. This problem becomes more accentuated taking into account that most of the web app users execute other background applications that constantly change the hardware resource availability. Thus, it is very likely that the number of Web Workers estimated could be non–optimal, since it may overload the system, degrading the performance of all applications, or underutilize the available resources.

In this paper we propose to modify highly parallel JavaScript web apps to include a pool of idle Web Workers and a simple algorithm. The goal of our approach is to dynamically manage the number of running threads to the actual available resources and to prevent the difficulties to estimate a fixed optimal number of running threads. To do this, we have also demonstrated that parallel web apps do not suffer significant performance degradation when there are many idle Web Workers. The results confirm that our browser independent proposal enables similar number of threads than the optimal setup for any of the three major web browsers in the market. Even though there are co–running applications that start or finish execution during the lifetime of the web app, the behavior of the manager is sustained.

Our evaluation concludes that with this proposal developers do not need hardly reachable details of every particular underlying platform to efficiently exploit the available resources, since the spawning degree is automatically determined by the dynamic manager.

## ACKNOWLEDGMENTS

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

## REFERENCES

1. Hickson, I. et al., “HTML5 Specification,” *http://www.w3.org/TR/html5*, Oct., 2014.
2. E. Fortuna, O. Anderson, L. Ceze, and S. Eggers, “A Limit Study of JavaScript Parallelism,” in *Procs. of IISWC*, pp. 1–10, Dec 2010.
3. J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry, “The STAMPede Approach to Thread-level Speculation,” *ACM Trans. Comput. Syst.*, vol. 23, pp. 253–300, Aug. 2005.
4. J. Renau, K. Strauss, L. Ceze, W. Liu, S. R. Sarangi, J. Tuck, and J. Torrellas, “Energy-Efficient Thread-Level Speculation,” *IEEE Micro*, vol. 26, pp. 80–91, Jan. 2006.
5. S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay, “Rock: A High-Performance Sparc CMT Processor,” *IEEE Micro*, vol. 29, no. 2, pp. 6–16, 2009.
6. P. Rundberg and P. Stenstrm, “An all-software thread-level data dependence speculation system for multiprocessors,” *Journal of Instruction-Level Parallelism*, vol. 3, p. 2002, 2001.
7. M. K. Chen and K. Olukotun, “The JRPM System for Dynamically Parallelizing Java Programs,” in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA ’03, (New York, NY, USA), pp. 434–446, ACM, 2003.



DOI: 10.1002/cpe

8. M. Mehrara and S. Mahlke, "Dynamically Accelerating Client-side Web Applications Through Decoupled Execution," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, (Washington, DC, USA), pp. 74–84, IEEE Computer Society, 2011.
9. Mozilla Developer Network, "Web Workers API," *https://developer. mozilla.org/en-US/docs/Web/API/Web_Workers_API*, Mar. 2015.
10. Hickson, I., "Web Workers. W3C Candidate Recommendation," *http://www.w3.org/TR/workers*, May, 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*, pp. 448–452, Sept. 2013.
12. D. Xu and B. Bode, "Performance Study and Dynamic Optimization Design for Thread Pool Systems," in *Procs. of CCCT*, 2004.
13. Y. Ling, T. Mullen, and X. Lin, "Analysis of Optimal Thread Pool Size," *SIGOPS Oper. Syst. Rev.*, Apr. 2000.
14. M. Russinovich *Process Explorer*, 2015.
15. StatCounter Global Stats 2015.
16. Ondřej Žára, "Hash Bruteforcer," *Demo Studio Mozilla Developer Network, https://developer.mozilla.org/es/demos/detail/hash-bruteforcer*, Apr. 2013.
17. Web Hypertext Application Technology Working Group (WHATWG) Wiki, "Navigator Hardware Concurrency," *https://wiki.whatwg.org/wiki/Navigator_HW_Concurrency*, July 2014.
18. Mozilla Developer Network *https://developer.mozilla.org/en-US/docs/Web/API/Performance/now*, 2015.
19. J. Martinsen, H. Grahn, and A. Isberg, "Using Speculation to Enhance JavaScript Performance in Web Applications," *IEEE Internet Computing*, vol. 17, pp. 10–19, Mar. 2013.
20. M. Qureshi, M. Dehnavi, N. Min-Allah, M. Qureshi, H. Hussain, I. Rentifis, N. Tziritas, T. Loukopoulos, S. Khan, C.-Z. Xu, and A. Zomaya, "Survey on Grid Resource Allocation Mechanisms," *Journal of Grid Computing*, vol. 12, no. 2, pp. 399–441, 2014.




DOI: 10.1002/cpe