Offworker: An Offloading Framework for Parallel Web Applications

An-Chi Liu and Yi-Ping You[✉]

Department of Computer Science, National Yang Ming Chiao Tung University, Hsinchu, Taiwan
acliu@cs.nycu.edu.tw, ypyou@nycu.edu.tw

Abstract. More and more applications are shifting from traditional desktop applications to web applications due to the prevalence of mobile devices and recent advances in wireless communication technologies. The Web Workers API has been proposed to allow for offloading computation-intensive tasks from applications' main browser thread, which is responsible for managing user interfaces and interacting with users, to other worker threads (or web workers) and thereby improving user experience. Prior studies have further offloaded computation-intensive tasks to remote servers by dispatching web workers to the servers and demonstrated their effectiveness in improving the performance of web applications. However, the approaches proposed by these prior studies expose potential vulnerabilities of servers due to their design and implementation and do not consider multiple web workers executing in a concurrent or parallel manner. In this paper, we propose an offloading framework (called *Offworker*) that transparently enables concurrent web workers to be offloaded to edge or cloud servers and provides a more secure execution environment for web workers. We also design a benchmark suite (called *Rodinia-JS*), which is a JavaScript version of the Rodinia parallel benchmark suite, to evaluate the proposed framework. Experiments demonstrated that Offworker effectively improved the performance of parallel applications (with up to 4.8x of speedup) when web workers were offloaded from a mobile device to a server. Offworker introduced only a geometric mean overhead of 12.1% against the native execution for computation-intensive applications. We believe Offworker offers a promising and secure solution for computation offloading of parallel web applications.

Keywords: Offloading · JavaScript · Parallelism · Web workers

1 Introduction

More and more desktop applications (e.g., Google Earth[1], Stellarium[2], and Autodesk[3]) are moving to the mobile market in the form of web applications.

[1] https://earth.google.com/web/.
[2] https://stellarium-web.org/.
[3] https://www.autodesk.com/solutions/cloud-based-online-cad-software.

Web applications for machine learning, gaming, and extended reality are getting more attractive since mobile devices as an input/output interface for end users become ubiquitous. However, these applications usually require high computation power and involve mass interactive activities, and sometimes demand a good internet connection for fast data download. These requirements can be problems for mobile devices, even for premium mobile devices, since heavy computation can drain their battery quickly, not to mention for mid-range or low-cost devices, which are usually not equipped with powerful processing units.

The Web Worker API[4] has been introduced to enable parallel JavaScript programming on the web. Many prior studies have shown that offloading computation-intensive tasks in web applications from mobile devices to edge or cloud servers can greatly enhance device performance without requiring the devices to have advanced compute capabilities or high connection bandwidth [3,6,8]. This was done by offloading the computation-intensive tasks to worker threads (or web workers for short), which are spawned to run concurrently with the main thread in JavaScript, and dispatching the web workers to remote servers. Nevertheless, these studies only addressed the benefits of offloading serial web workers, and almost none of them have examined the viability of offloading concurrent or parallel web workers. This is probably attributed to the fact that the Web Workers API has been primarily used for offloading tasks from the main browser thread so as to prevent the main browser thread from being blocked by the tasks, thereby providing a better user experience. However, with the advance of multicore mobile devices, we believe parallel web applications are gaining more attention and popularity, and offloading parallel web workers to edge or cloud servers is a demanding task. To our best knowledge, Puffin Web Browser[5] (or Puffin for short), a commercial web browser developed by CloudMosa, is the only work that enables parallel web workers to run remotely. Essentially, Puffin is basically a "thin-client" browser that renders webpages, including JavaScript code that operates using web workers, in the cloud. However, this type of offloading scheme is inherently incapable of supporting numerous interactions with user interface components because each user action request must be transferred to the cloud for processing and then back to the client, resulting in long response times.

In this paper, we propose a framework (called *Offworker*[6]) that enables offloading parallel web workers to edge or cloud servers so as to enhance the execution of web applications on mobile devices, particularly on mid-range or low-cost devices. The Offworker framework comprises two main components: (1) a front-end library (FL), which exposes Web Workers APIs and forwards requests to remote servers, and (2) a back-end manager (BM) for launching web workers on the remote servers. Offworker is designed to improve web worker applications in four aspects: (1) faster execution, (2) improved user experience, (3) secure execution environment, and (4) transparent programmability. As an offloading framework, Offworker aims to improve the execution performance of web worker appli-

[4] https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API.
[5] https://www.puffin.com/web-browser.
[6] https://github.com/nycu-sslab/offworker.

cations by offloading heavy computation, which may involve multiple web workers running in parallel, or communication to edge or cloud servers. Offworker also delivers a better user experience than other offloading framework (e.g., Puffin) for applications that frequently interact with users since it offloads only web workers, which are typically computationally intensive, rather than the whole rendering process of webpages. Moreover, with Offworker, a separate V8 isolate, which is an isolated JavaScript environment using the V8 JavaScript engine[7], is created on the remote site for each offloaded web worker, and each offloaded web worker run within the V8 isolate; therefore, Offworker provides a more secure environment for web workers. Furthermore, Offworker is partly designed and implemented as a library that conforms to the Web Workers APIs, so ordinary web worker applications can directly take advantage of Offworker without modifying any code, thereby providing a transparent offloading mechanism.

2 Related Work

There have been many prior studies focusing on offloading computation-intensive tasks in web applications from mobile devices to servers. The Web Workers API has been considered the most common interface for web application developers to implement a complex computation task without interfering the user interface. Therefore, there have been several studies (with the same objective as this study) that aimed to offload web workers to servers in order to improve the execution performance of web applications [4–8]. Most of these studies proposed a roughly same architecture as proposed in this study, which includes a front-end library, which accepts computation requests on the client, and a back-end manager, which enables web workers to execute on the server.

Hwang and Ham proposed a framework, called WWF [5], that allows web workers (with some modifications to the original application) to be offloaded to servers and whose BM was implemented based on the Node.js library[8], but how web workers operates on the server was not clearly stated. Zbierski and Makosiej proposed a similar framework like WWF but without requiring modifications to the application. They also proposed an offloading decision model according to CPU and memory usages and network conditions [8]. However, the implementation details of the proposed framework and decision model were not elaborated in details. Gong et al. proposed a framework, called WWOF [3], which is also similar to WWF, but web workers are executed on servers using the VM module of Node.js. Jeong et al. introduced a different offloading scenario that allows a running web worker to migrate from a mobile device to a server—though with a larger offloading overhead [6]. They implemented a snapshot mechanism that enables web worker migration, using the subprocess module of Node.js to execute web workers on the server.

Although the aforementioned studies have demonstrated their success in offloading web workers to servers, the libraries they use for executing web work-

[7] https://v8.dev/.
[8] https://nodejs.org/.

ers on servers, such as the VM and the subprocess modules of Node.js, may expose vulnerability to the servers. In contrast, Offworker adopts the isolated-vm library[9], which guarantees an offloaded web worker to execute within a sandbox, thereby providing a more secure offloading environment. More details about the potential security issues and how Offworker addresses these issues are discussed in Sect. 3.2. In addition, these prior studies considered only applications without concurrent web workers, whereas in this study we address the issues in offloading concurrent web workers and evaluate the proposed approach with a set of parallel web applications.

3 Design and Implementation of Offworker

The Offworker framework comprises two main components: (1) the FL and (2) the BM. The FL, which is included in web applications, exposes web worker-related APIs to the applications and passes web worker requests—such as the creation of web workers and communication between web workers—to the BM. The BM, which is designed as a service daemon running on an edge or cloud server, is responsible for fulfilling web worker requests so that web workers can run on the server and communicate with one another properly. Figure 1 illustrates the workflows of the native execution and offloading execution of a web worker application, respectively, and also the conceptual architecture of the Offworker framework. We briefly introduce the two workflows and then focus on how a web work task is offloaded to the server using Offworker. It is worth mentioning that parallelism for web applications is commonly implemented using the proxy pattern[10], in which parallel worker threads are created by a proxy thread, which is created by the main thread and interacts with the worker threads. This proxy pattern avoids the main browser thread (which is typically the main application thread) from constantly synchronizing with other worker threads (i.e., web workers) and allows the main browser thread to focus on rendering and handling user interactions. In this study, we presume that parallelism is expressed in parallel web applications using the proxy pattern.

For both the native and offloading execution of the web applications in Fig. 1, the workflow starts with the initial HTTP requests from a client who intends to launch a web application, and then the web server responds to the client with the web application, which contains HTML pages with CSS formatting and JavaScript codes. Using the Web Workers API, computation-intensive tasks (sometimes referred to as *kernels*) of the web application can be offloaded onto a separate web worker or several web workers that are launched by a proxy web worker. During the native execution of a kernel, all web workers are created and run natively on the client side (i.e., the browser), and the web workers may request resources from the web server during their execution. In contrast, when the FL is included in the application, Offworker is activated, and the kernel, including the proxy web worker and its associated parallel web workers, can be

[9] https://github.com/laverdet/isolated-vm.
[10] https://emscripten.org/docs/porting/pthreads.html#proxying.

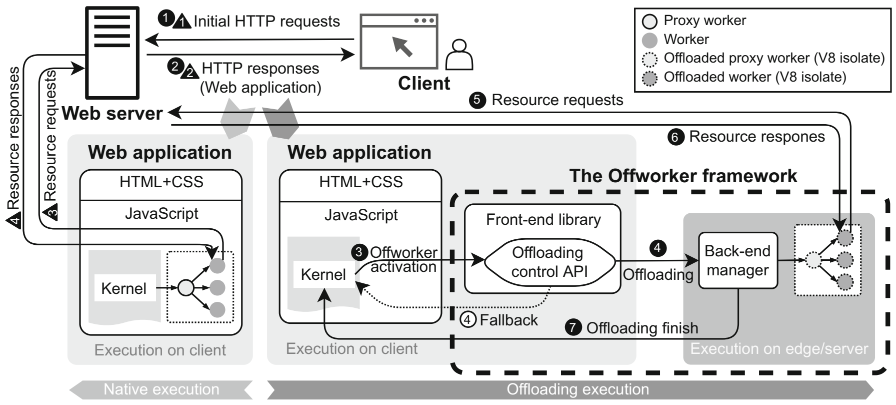

Fig. 1. The workflows of the native execution and offloading execution (using Offworker) of a web worker application, respectively.

offloaded to the server depending on the developer's decision. The proxy web worker is created either natively, in which case the kernel is executed with the same workflow as the native execution, or on the server, in which case the request for creating the proxy web worker is sent to the BM. In the latter case, the BM creates the proxy web worker and its associated parallel web workers as separate V8 isolates on the server, and these workers can directly communicate with the web server. Lastly, the BM sends back the results to the client when all web workers finished.

3.1 The Front-End Library

The FL is an implementation of the Web Worker API and allows a web worker to be created and run remotely. The FL overrides the `Worker` class, in which the constructor, communication-related methods or properties (e.g., `postMessage()` and `onmessage`), and other class members are implemented in a way that they work with the BM. More specifically, once developers have included the front-end Offworker library and created a proxy web worker object with the `Worker` class, the constructor of `Worker` uses the WebSocket API[11] to create connections between the FL and the BM so as to pass the request for worker creation and the worker creation argument (the script that the worker will execute) to the BM and allow for communication between the main browser thread and the proxy web worker. How the communication is processed is discussed in detail in Sect. 3.2.

3.2 The Back-End Manager

The BM is a server-side daemon process that receives and handles requests from the FL and is responsible for creating web workers on the server and managing communication among the main browser thread and the web workers being

[11] https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API.

offloaded to the server. In essence, the BM enables the functionality to run web workers on the server and can be implemented by adopting a server-side JavaScript library that allows web workers to run on the server. The simplest and most intuitive way to run web workers on the server is by using the *worker threads* module in the Node.js library, which is almost the counterpart of the Web Workers API in Node.js. However, such an implementation of the BM exposes vulnerability of the server since worker threads (i.e., web workers created on the server using the worker threads module) are able to invoke system calls to access privileged system information and resources by using the Node.js APIs. Moreover, the BM will be unable to fully manage worker threads—for example, to control resources used by worker threads in order to prevent resource starvation attacks—, unless modifying the Node.js library. These effects also occur when using other modules, such as the *child process* or *VM* modules, in the Node.js library as the basis for implementing the BM.

In view of this, we propose to create web workers on the server by using the *isolated-vm* library, which allows code to run within an isolated environment that conforms to a V8 isolate. In other words, a V8 isolate is created for each offloaded web worker and used to run the corresponding task in a web worker. Furthermore, since a V8 isolate is an isolated instance of the V8 JavaScript engine, which runs only core JavaScript (i.e., ECMAScript[12]) code but not client- or server-side JavaScript code, an offloaded worker is guaranteed to execute within a sandbox with configurable resource limitations and unable to access any resources on the server or even call web APIs, thereby solving the aforementioned potential security issues.

Nevertheless, the fact that each offloaded worker runs as a V8 isolate also raises another two issues: creating V8 isolates within an existing V8 isolate is not feasible, and communication between V8 isolates is not possible without a proper runtime system. Therefore, the BM must serve as a proxy for isolate creation and further implement a communication mechanism that manages possible communication among different isolates. The BM involves three main components: (1) an isolate creator, which creates web workers as V8 isolates on the server, (2) a message handler, which enables the message passing mechanism between the client and the server and between V8 isolates, and (3) a shared memory manager, which allows memory to be shared between V8 isolates. The BM also implements some web APIs for V8 isolates to facilitate functionality of web applications. We elaborate in detail how the three components work by demonstrating how web workers are offloaded and created onto the server and how communication between the client and server and communication between web workers function in the following paragraphs.

Creation of Web Workers on the Server. As stated at the beginning of Sect. 3, the hypothesis underlying this study is that web applications express parallelism using the proxy pattern. Hence, each kernel is always activated by creating a proxy web worker, which further creates parallel web workers. Figure 2

[12] https://tc39.es/ecma262.

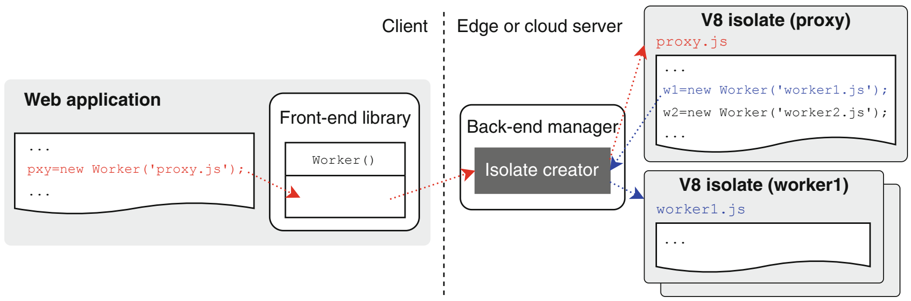

Fig. 2. The workflows of creating web workers (V8 isolates) on the server using Offworker. (Color figure online)

illustrates how a kernel is offloaded to the server by showing the workflows of how V8 isolates, which act as web workers on the server, are created. There are two types of workflows: (1) the client's web application creates a proxy web worker, as shown in red text and red dotted arrows in the figure, and (2) the proxy web worker (or any other worker) creates another web worker, as shown in blue text and blue dotted arrows. In the former case, when the client's web application calls "`new Worker()`" to create a proxy web worker, the FL passes the request of the proxy worker creation to the BM, and then the isolate creator in the BM creates a V8 isolate to execute the proxy worker. In the latter case, when a web worker calls "new `Worker()`" to create another web worker, the isolate creator, which is registered as an event handler for the "`new Worker()`" event in V8 isolates, creates another V8 isolate to execute the newly created worker.

Communication Management. Apart from creating V8 isolates (i.e., web workers on the server), the other primary task of the BM is to enable communication, which involves a V8 isolate on one end. For parallelism using the proxy pattern, there are only two scenarios of communication to consider: (1) the communication between the client's web application and a proxy V8 isolate and (2) the communication between V8 isolates. Since the communication can be done by using the message passing (via `postMessage()` and `onmessage`) or shared-memory (via `SharedArrayBuffer`) mechanisms, the BM must guarantee that both the two communication mechanisms work correctly when web workers are offloaded to the server. We discuss how these two communication mechanisms are managed by the BM, respectively.

Figure 3 shows how communication using message passing is performed in Offworker. The BM implements a message handler, which is essentially an event handler, for all communication requests made in both of the aforementioned scenarios. In the first scenario, it is illustrated in red text and red dotted arrows how the client's web application sends a message to a proxy V8 isolate—the FL passes the sending request to the BM, which then directs the message to the proxy V8 isolate—, while in blue text and blue dotted arrows how it goes in reverse. The client–server communication (between the FL and the BM) is

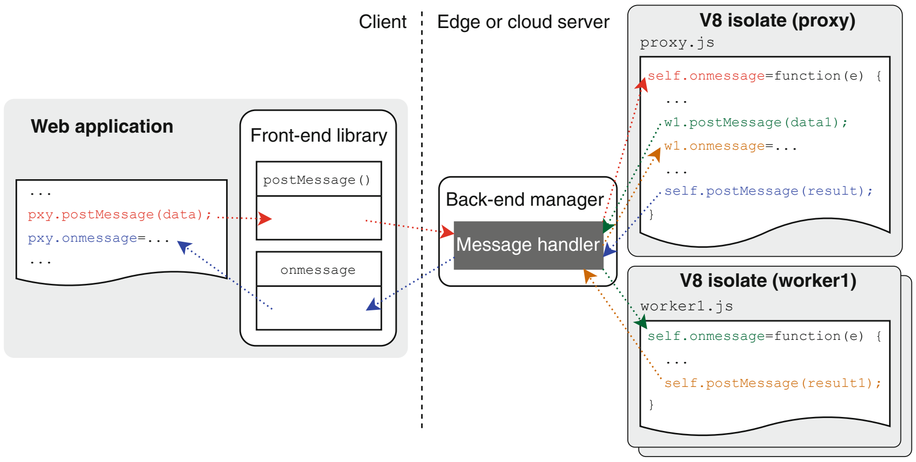

Fig. 3. Communication between the client and server and communication among web workers using message passing in Offworker. (Color figure online)

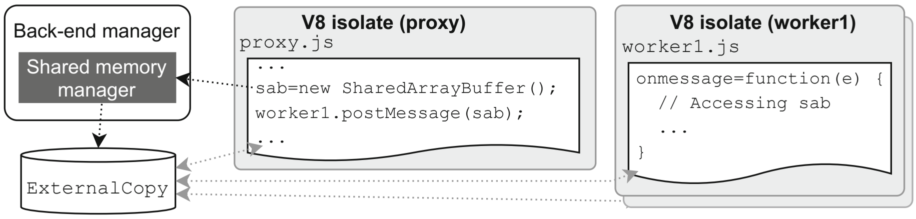

Fig. 4. Communication among web workers using shared memory in Offworker.

enabled by the WebSocket connection established when creating a proxy V8 isolate (as discussed in Sect. 3.1), and `MessageChannel` in JavaScript is used to communicate between the BM and the proxy V8 isolate. Communication in the second scenario, which can be seen in green/orange text and green/orange dotted arrows, is similar to communication in the first scenario, but all of the communication happens on the server.

Figure 4 depicts how communication using shared memory is processed in Offworker. It is worth mentioning that for parallelism using the proxy pattern, shared-memory communication typically occurs only between web workers (either proxy or regular), so only the second scenario is considered in this case. The BM deploys a shared-memory manager, which is registered as an event handler for the "`new SharedArrayBuffer()`" event in V8 isolates. When a `new SharedArrayBuffer()` request is made, the shared-memory manager allocates an `ExternalCopy` object, which is implemented by the isolated-vm library and treated as the standard shared memory that is accessible to any V8 isolate, so as to allow shared-memory communication among V8 isolates.

4 Evaluations and Discussion

We implemented the proposed Offworker framework in JavaScript, based on isolated-vm v4.2 and Node.js v14.16 (with V8 JavaScript engine v8.4), and evaluated the framework with web applications running on Google Chrome v91 (with V8 JavaScript engine v9.1). And Node.js v14.16, and evaluated the framework with web applications running on Google Chrome v91.

We evaluated the Offworker framework in different scenarios in order to examine its effectiveness. Unless specified, web applications ran on a client mobile device (Sony Xperia 10), which was located in Hsinchu, Taiwan and has a total of eight CPU cores with four Cortex-A53 cores operating at up to 2.2 GHz, four Cortex-A53 cores operating at up to 1.8 GHz, and 3 GB of RAM. The BM of Offworker was deployed on three different servers: an edge server (ES) located in Hsinchu, a near cloud server (NCS) in Hong Kong, and a far cloud server (FCS) in the United States. The ES was equipped with an Intel quad-core i3-10100 processor operating at 3.6 GHz and 32 GB of RAM; the NCS was hosted in an Amazon EC2 t3.2xlarge instance, which was equipped with eight Intel Xeon Platinum 8259CL vCPUs operating at 2.5 GHz and 32 GB of RAM; the FCS, which was provided by CloudMosa, was equipped with eight Intel Xeon E3-1241 v3 vCPUs and 32 GB of RAM. The ES also acted as a web server in all experiments conducted in this work.

To our best knowledge, there is no JavaScript benchmark suite for parallel web applications. We believe that this is attributed to parallelism being a relatively new feature in JavaScript and many computation-intensive applications have not yet moved to the web. In order to evaluate the effectiveness of Offworker, especially in terms of its capability of running parallel web applications, we manually ported the Rodinia benchmark suite (version 1) [2], a popular benchmark suite for heterogeneous computing, from OpenMP programs into JavaScript programs, where the computation-intensive parts of the programs (i.e., kernels) were expressed by using the Web Workers API with the proxy pattern, and workers synchronize via a barrier at the end of a kernel. All Rodinia applications have been successfully ported, except *Leukocyte Tracking*, *Stream Cluster*, and *Similarity Score* due to their large code size (over 3,000 lines of code). We call this new benchmark suite *Rodinia-JS*.

The Rodinia benchmark suite includes default datasets, but their size is too large for client-side web applications, which fetch external data from the web server rather than from the local file system, since data fetching is likely to become a major task of the applications. We scaled down the datasets to meet the following criteria: (1) the total running time of an application on the mobile device takes less than seven seconds while the time spent in data fetching is less than two seconds, which is more reasonable for web applications as more than half of visits are abandoned if a mobile site takes over three seconds to load[13], and (2) the time spent in computation is greater than the overhead time incurred by creating workers, which makes sense for parallelization to be beneficial.

[13] https://developer.chrome.com/blog/search-ads-speed/.

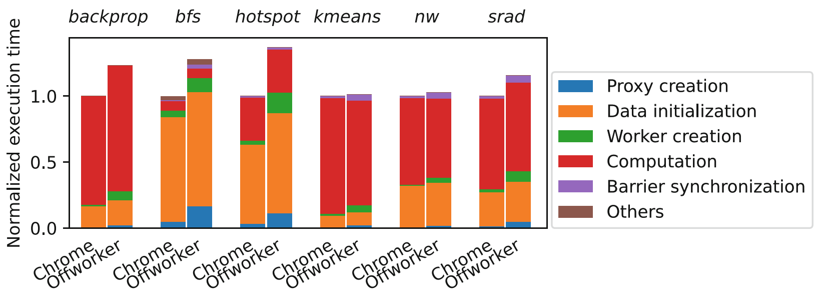

Fig. 5. Normalized execution times for the Rodinia-JS benchmark suite on the ES when running natively on Chrome and using Offworker.

We also used the Hopscotch micro-benchmark suite [1] to further evaluate the performance difference in manipulating shared memory between Offworker and Chrome. 14 (out of 16) types of memory access patterns were ported to JavaScript. Two types of patterns (*r_rand_pchase* and *w_seq_memset*) were excluded due to no pointers and "memset" in JavaScript.

4.1 Overhead Analysis

Figure 5 displays the breakdown of normalized execution times for the Rodinia-JS applications without kernel parallelization (i.e., only one web worker for computing a kernel task) when running the applications natively on the headless Chrome browser and using Offworker, respectively. Both the client and server ran on the same machine (ES) so that we could better identify the performance overhead due to the proposed Offworker framework. Each execution time is divided into six parts in our measurements: (1) proxy creation, which fetches the script of the proxy web worker and initializes the proxy worker, (2) data initialization, which fetches input data and constructs data structures, (3) worker creation, which fetches the script of the worker and initializes the worker, (4) computation, which is the main task of a kernel, (5) barrier synchronization, which is necessary at the end of each kernel due to the fork-join model of OpenMP being adopted, and (6) others, which do not belong to any of the aforementioned five parts. Overall, Offworker introduced a (geometric) mean overhead of 12.1% for computation-intensive applications (*backprop*, *kmeans*, *nw*, and *srad*) and 18.5% for all applications in Rodinia-JS. *bfs* and *hotspot* had a larger percentage overhead since they had a short application time on ES (only around 160 and 260 ms, respectively)—although they took around one and two seconds to execute on the mobile device, respectively—, and they are I/O-intensive applications.

One of the main overhead sources lied in the creation of the proxy web worker (geometric mean of 4.5%) and regular web workers initialization (6%), which involves additional HTTP connections to fetch the script of web workers, creating workers, and initializing workers. Offworker added an additional overhead of around 20 ms for each worker creation (excluding worker initialization). Another main overhead source was from the barrier synchronization (2.2%), which was implemented using the message passing mechanism, because each message passing operation between web workers was around 0.7 ms slower when using Offworker than running natively on Chrome. This slight overhead was attributed to an implementation difference, where a worker's script is executed on the Node.js platform and non-core JavaScript API calls (including message passing) are implemented in JavaScript in Offworker, whereas all the script is executed by Chrome (which is implemented in C++) when running natively. This implementation difference also leaded to some overheads in data initialization (5.2%). For example, *bfs* and *hotspot* had a larger overhead in data initialization (7.5% and 15.8%, respectively) since they both invoked the `split` function, which splits a string into substrings, where a `split` function call was around 70 ms slower on Node.js than on Chrome. Each HTTP connection was also around 30 ms slower when using Offworker than on Chrome. The computation parts also showed slight variations between the two platforms due to the implementation difference. Offworker performed slightly better than Chrome with respect to the computation part for *nw*, but worse than Chrome for *backprop* and *kmeans*. We discuss these variations in details in Sect. 4.2.

4.2 Effectiveness in Running Parallel Applications

As mentioned in Sect. 1, a significant novelty of this study is to examine the viability of offloading parallel web workers to servers; therefore, we evaluated the effectiveness of Offworker in terms of running parallel web workers. We do not discuss how Offworker could scale different Rodinia-JS applications since the scalability of an application is highly dependent on its design. Instead, we focus on the execution time differences of applications between running on Chrome and Offworker, and therefore both the client and server were on the ES.

Figure 6 shows the execution time ratios of Offworker over Chrome for the Rodinia-JS applications when different numbers of web workers were used. As observed in Fig. 5, when using Offworker, all applications (with only one web worker being created for each kernel) had longer execution time due to the extra offloading manipulation. The inferiority of Offworker persisted for applications with more parallelism and went slightly up as the number of web workers increased for most applications. The ratios for *backprop* stayed roughly the same when increasing the number of web workers, whereas the ratios for *kmeans* grew more significantly as the number of web workers increased. We further investigated the contributing components of the execution time differences between Offworker and Chrome for each application in order to identify why Offworker performed differently.

Figure 7 illustrates the execution time differences in terms of their contributing components between Offworker and Chrome for the Rodinia-JS benchmark suite when different numbers of web workers were used. A positive difference represents that Chrome was better than Offworker. We observed that the time differences for the proxy creation, data initialization, and other parts were less

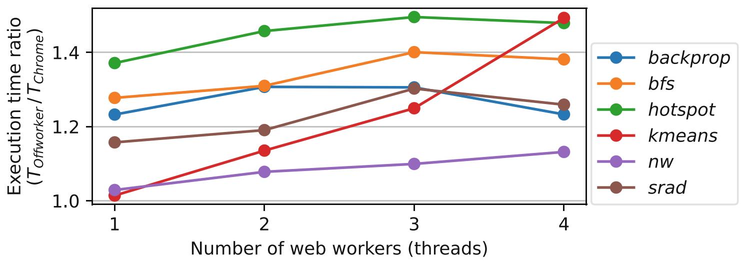

Fig. 6. Execution time ratios of Offworker over Chrome for the Rodinia-JS benchmark suite on the ES when different numbers of web workers were used.

than 50 ms and consistent for different numbers of web workers since these components were irrelevant to parallelization. The time differences for the barrier synchronization part were also almost identical when different numbers of web workers were used since there are no more than 300 barriers required for each Rodinia-JS application, and the overhead added by Offworker for each barrier synchronization was insignificant. The time differences for the worker creation part grew slightly with the number of web workers due to more worker initialization costs when using Offworker. This effect was more obvious for *backprop* and *kmeans* because they required more initialization work than others. However, different applications showed different trends in time differences for the computation part. This abnormal phenomenon was believed to result from the differences in handling arithmetic and shared-memory access operations between Offworker (which is based on isolated-vm and Node.js) and Chrome.

Figure 8 displays execution time ratios of Offworker over Chrome for 14 memory access patterns in the Hopscotch micro-benchmark suite on a large (or small) array when using different numbers of web workers. We observed that the execution times for each micro benchmark operating on a large shared-memory array (of length 10^4–10^8) were similar between using Offworker and running natively on Chrome; however, the execution time ratios of Offworker over Chrome were up to five when operating on a small shared-memory array (of length 10^0–10^4). As indicated in Fig. 7, the computation time differences between Offworker and Chrome (CTDs for short) were larger for *backprop* since there were five memory access patterns on six small and two large shared-memory arrays, and more time was spent accessing these small arrays. The CTDs grew up greatly with the number of web workers for *kmeans* since *r_ seq_ reduce* on small arrays provided the majority of shared-memory access patterns. The shared-memory access patterns in *bfs*, *nw*, and *srad* also explained their trend of CTDs, respectively: *r_ seq_ ind* and *w_ seq_ fill* on six small arrays, *r_ tile* on two large arrays, and *r_ rand_ ind* and *w_ seq_ fill* on four small arrays and seven large arrays provided the majority of shared-memory access patterns for *bfs*, *nw*, and *srad*, respectively. Despite these negative consequences, we believe isolated-vm still provides an adequate solution for offloading JavaScript applications due to its secure nature.

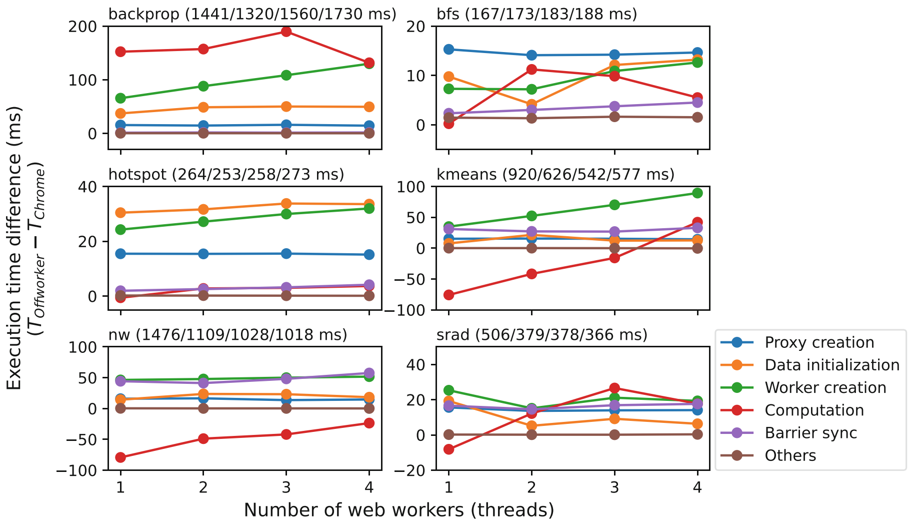

Fig. 7. Execution time differences between Offworker and Chrome for the Rodinia-JS benchmark suite on the ES when different numbers of web workers were used. The values follow a benchmark name indicate the execution times of the benchmark for 1–4 workers, respectively, when using Offworker.

4.3 Effectiveness in Different Server Capabilities

Figure 9 illustrates the execution times for the Rodinia-JS benchmark suite on the client when using five different offloading decisions: (1) mobile, in which web workers ran natively on the client, (2) ES-Offworker, in which web workers ran on the ES using Offworker, (3) NCS-Offworker, in which web workers ran on the NCS using Offworker, (4) FCS-Offworker, in which web workers ran on the FCS using Offworker, and (5) FCS-Puffin, in which almost an entire application (including web workers) ran on the FCS using Puffin, which is a commercial browser developed by CloudMosa based on Chromium v79 and renders webpages on the cloud.

ES-Offworker had the best performance (2.8–4.8x faster than Mobile) among all offloading decisions, while NCS-Offworker came second (0.9–2.6x faster than Mobile). These results were expected because both the ES and NCS had more powerful computing capabilities than the client mobile device and were located close to the client mobile device—the round-trip time (RTT) between the client and the ES or NCS was sufficiently low to allow for offloading web workers to the ES or NCS with benefit—, and because the ES was physically closer to the client than the NCS.

Despite the fact that the FCS might not be a good candidate for offloading operations, we conducted evaluations for FCS-Offworker in order to compare Offworker with Puffin, while the FCS, which had similar hardware configurations with the NCS, was the only platform that worked for both Offworker and

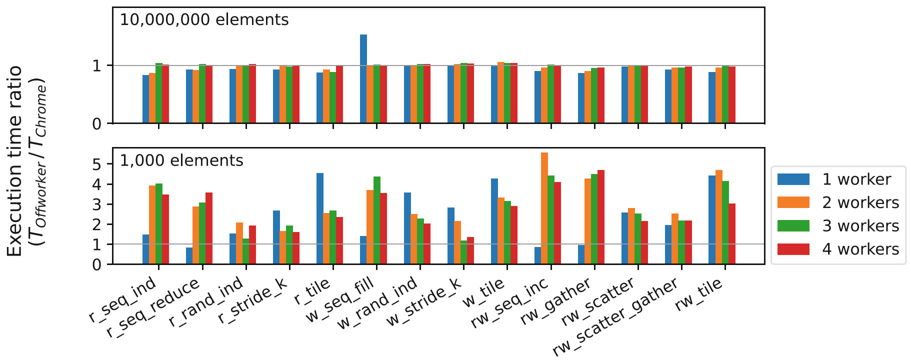

Fig. 8. Execution time ratios of Offworker over Chrome for 14 memory access patterns on a large (or small) array when different numbers of web workers were used.

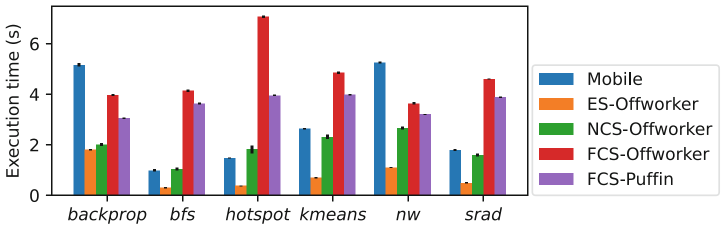

Fig. 9. Execution times for the Rodinia-JS benchmark suite on the client when using different offloading decisions, mostly in terms of server capabilities.

Puffin. Both FCS-Offworker and FCS-Puffin suffered from the problem of long RTTs between the FCS and the web server (hosted on the ES) such that they did not perform well for some applications, especially for I/O-intensive applications (*bfs* and *hotspot*), which involve fetching massive resources from the web server. Nevertheless, this negative effect may disappear or even be reversed if the FCS is close to the web server due to lower RTTs. In general, Offworker performed slightly worse than Puffin because Puffin was implemented based on Chromium, which is an open-source browser that Chrome is built on, and Chrome performed slightly better than Offworker, as discussed in Sect. 4.1. FCS-Offworker performed poorly for *hotspot* because *hotspot* included massive HTTP connections, and the implementation for handling HTTP connections in Offworker was not optimized and did not performed well.

While Puffin performed better than Offworker for the Rodinia-JS benchmark suite, it has some weaknesses when used with applications that require extensive user interaction. These weaknesses are attributed to the design of Puffin, which offloads an entire web page to the server, and consequently the delay time between firing a user action and receiving its corresponding rendering results from the server can significantly decrease the user experience. This application scenario with high user interaction will become common as more and more desktop applications are converted into web applications (e.g., online image editors). Compared with Puffin's coarse-grained offloading approach, Offworker provides a fine-grained offloading mechanism that allows users to offload tasks on demand, thereby achieving wider applicability.

5 Conclusion

We have proposed a framework, called Offworker, for transparently offloading parallel web workers to edge or cloud servers. To our best knowledge, this is the first work that supports inter-worker communication (with message passing or shared memory) for offloaded web workers. Furthermore, Offworker ensures that each offloaded web worker runs within a sandbox (V8 isolate) so as to provide a more secure execution environment for web workers and servers. We have also presented a parallel JavaScript benchmark suite, called Rodinia-JS, and evaluated Offworker with Rodinia-JS. The experimental results show that Rodinia-JS applications with Offworker enabled ran up to five times faster than they running natively on a mobile device, and Offworker had a small overhead (mean of 12.1%) for computation-intensive Rodinia-JS applications. We believe our proposed solution best serves the needs for parallel web applications. We consider to support WebAssembly threads in JavaScript applications and propose an offloading decision model in the future.

Acknowledgements. This study was partially supported by the Ministry of Science and Technology of Taiwan under Grant No. MOST 110-2221-E-A49-030-MY3. We would like to thank CloudMosa, Inc. for providing supports and hardware resources for benchmarking.

References

1. Ahmed, A., Skadron, K.: Hopscotch: a micro-benchmark suite for memory performance evaluation. In: Proceedings of the International Symposium on Memory Systems, MEMSYS 2019, pp. 167–172 (2019)
2. Che, S., et al.: Rodinia: a benchmark suite for heterogeneous computing. In: Proceedings of the 2009 International Symposium on Workload Characterization, pp. 44–54 (2009)
3. Gong, X., Liu, W., Zhang, J., Xu, H., Zhao, W., Liu, C.: WWOF: an energy efficient offloading framework for mobile webpage. In: Proceedings of the 13th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services, pp. 160–169 (2016)
4. Hwang, I., Ham, J.: Cloud offloading method for web applications. In: Proceedings of the 2nd International Conference on Mobile Cloud Computing, Services, and Engineering, pp. 246–247 (2014)
5. Hwang, I., Ham, J.: WWF: web application workload balancing framework. In: Proceedings of the 28th International Conference on Advanced Information Networking and Applications Workshops, pp. 150–153 (2014)

6. Jeong, H.J., Shin, C.H., Shin, K.Y., Lee, H.J., Moon, S.M.: Seamless offloading of web app computations from mobile device to edge clouds via HTML5 web worker migration. In: Proceedings of the ACM Symposium on Cloud Computing 2019, pp. 38–49 (2019)
7. Wang, Z., Deng, H., Hu, L., Zhu, X.: HTML5 web worker transparent offloading method for web applications. In: Proceedings of the 18th International Conference on Communication Technology, pp. 1319–1323 (2018)
8. Zbierski, M., Makosiej, P.: Bring the cloud to your mobile: transparent offloading of HTML5 web workers. In: Proceedings of the 6th International Conference on Cloud Computing Technology and Science, pp. 198–203 (2014)