JSCloud: Toward Remote Execution of JavaScript Code on Handheld Devices[*]

Winson Y. S. Li	Shangru Wu	W. K. Chan[†]	T. H. Tse
Department of Computer Science	Department of Computer Science	Department of Computer Science	Department of Computer Science
City University of Hong Kong	City University of Hong Kong	City University of Hong Kong	The University of Hong Kong
Tat Chee Avenue, Hong Kong	Tat Chee Avenue, Hong Kong	Tat Chee Avenue, Hong Kong	Pokfulam, Hong Kong
winsonli@gmail.com	shangruwu2@student.cityu.edu.hk	wkchan@cityu.edu.hk	thtse@cs.hku.hk

Abstract—**We present a generic framework, JSCloud, for the remote execution of JavaScript programs. Our work dynamically estimates whether a code partition should be executed remotely. The empirical result shows that JSCloud can be useful if the JavaScript engine on a handheld device is inefficient.**

Keywords—Remote execution, JavaScript, mobile computing

I. INTRODUCTION

A modern web application often has a browser-based component, which contains the JavaScript code of the application. At the same time, people increasingly use handheld devices to access such web applications. Owing to the limited hardware capability and screen sizes, some of these web applications can only be used after customization (such as the case of Facebook for Mobile webpage [5]). However, such customization comes at a cost. Many mobile versions of various applications offer only a subset of the features provided by the standard counterpart. For instance, the lab feature of Gmail [2] is not available in a mobile edition.

The execution of a piece of JavaScript code on such a device can also be slow, which further restricts the types of features that can be adapted on a handheld platform with little processing capability and power supply. Our experiment to be presented in this paper further shows that executing a sort program on Apple iPod may not be completed successfully due to resource constraints.

Code migration for mobile platforms is not a new topic [7]. Traditionally, researchers look for code optimization or feature simplification. They mostly identify, say, a subset of the Java classes to be migrated to other platforms for execution [6][9][11]. Recently, researchers explore the capability of clone code running in a virtual machine of a computing cloud infrastructure [1].

This paper presents JSCloud. It dynamically estimates the relative amount of time needed to execute a piece of JavaScript code locally and remotely and makes a decision on whether to fulfill the execution request remotely to enhance the performance of an application on a handheld device via a simple but effective linear interpolation approach, which sets it apart from the work [1] that inspired JSCloud. It also reports preliminary experimental results on the use of JSCloud on various handheld device platforms.

The rest of this paper is organized as follows. Section II illustrates a motivating study. We elaborate on JSCloud in Section III, followed by an experiment in Section IV. We discuss related work in Section V. Finally, we conclude the paper in Section VI.

II. MOTIVATING EXAMPLE

We use the function `merge_sort()` implemented in JavaScript as shown in Figure 1 to motivate our work. In the code listing, a column of integers modeled as array `col` of length `col.length` will be sorted. The computation time of the function to sort `col` on various desktop machines (denoted by devices D1−D3) and handheld devices (denoted by devices M1−M2) were measured. The results are shown in Table 1.

The computation time to sort 1000 numbers on devices M1 and M2 exceeded 100 ms. At this level of delay, the user would experience a noticeable delay in system response [10]. When sorting 100,000 numbers, the JavaScript program failed to compute on devices D2, M1, and M2. On desktop machines, the browser issued a warning and prompted the user to indicate whether to terminate the script. On mobile devices, the browser became non-responsive and required a forced quit.

III. JSCLOUD

In this section, we present our proposed system JSCloud.

A. Overview

JSCloud consists of two phases: the code analysis and instrumentation phase followed by the partition execution phase. In the first phase, JSCloud aims at (a) identifying a set of code partitions (such as a set of JavaScript functions) in a given JavaScript document that JSCloud may choose to

† Contact author

execute a partition remotely in a later partition execution phase, and (b) extending the given JavaScript document with the migration logics. We refer to the JSCloud component for this phase as JSCloud Packager and that for the partition execution phase as JSCloud Migrator. JSCloud also has a third component, which is a JavaScript engine installed in a server/cloud to host the migrated code received from JSCloud Packager and execute the selected partitions based on the instructions from JSCloud Migrator. Figures 2 and 3 show the usages of JSCloud Packager and Migrator.

```
function merge_sort(arr){
    function split_array(arr){
        if (arr.length <= 1)
            return arr;
        var middle = parseInt(arr.length / 2);
        var left = arr.slice(0, middle);
        var right = arr.slice(middle, arr.length);
        return merge(split_array(left),
            split_array(right));
    }

    function merge(left, right){
        var result = [];
        while (left.length > 0 || right.length > 0){
            if (left.length > 0 && right.length > 0){
                if (left[0] <= right[0]){
                    result.push(left.shift());
                } else {
                    result.push(right.shift());
                }
            } else if (left.length > 0){
                result.push(left.shift());
            } else if (right.length > 0){
                result.push(right.shift());
            }
        }
        return result;
    }

    return split_array(arr);
}

// timestamp t1 in milliseconds
merge_sort(col);
// timestamp t2 in milliseconds
```

Figure 1. Example code in JavaScript.

In a JavaScript-based web application, the JavaScript document is delivered as a separate file from a web server to a web browser. JSCloud Packager serves as an intermediary of the web server. It annotates the JavaScript functions and inserts our migration logic into the original JavaScript document. The modified JavaScript document will then be executed on a web browser. If JSCloud Migrator invokes a remote execution, the input values necessary for the remote execution will be passed to a remote JavaScript engine **Error! Reference source not found.** for action.

B. JSCloud Packager

A *partition* is the code between the entry and exit of a JavaScript function. A JavaScript statement may invoke some environmental operations such as opening a dialog box and prompting the user to select a file from the local file system. Such a statement could be difficult to execute

remotely. As such, JSCloud Packager statically determines whether a partition contains any system calls or references to non-local variables (e.g., global variables) or system objects (e.g., the *window* object). It marks a partition as *legitimate* if the partition contains no such statement. For instance, if we apply this scheme to the example in Figure 1, we obtain three partitions: merge_sort, split_array, and merge. It annotates the start and end of a legitimate partition by "^ ENTRY POINT" and "$ EXIT POINT", respectively.

TABLE 1. EXECUTION TIME FOR merge_sort()
FOR DIFFERENT LIST SIZES ON DIFFERENT PLATFORMS

	Time Taken (ms) to Sort Array col				
col.length	10	100	1,000	10,000	100,000
Laptop computer MacBook Pro 2.2GHz C2D Chrome v.14 (Device D1)	0	1	2	21	773
Laptop computer MacBook Pro 2.2Ghz C2D Safari v.5 (Device D2)	0	1	28	1,531	–
Laptop computer Lenovo X200 2.26GHz C2D IE 9 (Device D3)	0	0	5	135	6,561
Mobile phone BlackBerry 9780 Default browser (Device M1)	2	11	261	18,827	–
Portable multimedia player iPod Touch 1st Generation Safari (Device M2)	7	22	420	26,167	–

JSCloud Packager supports both an automatic mode and a manual mode. In the automatic mode, JSCloud Packager treats every function in a JavaScript document as a partition so that it can identify the legitimate ones automatically. In the manual mode, only the JavaScript functions that are annotated by the web application developers are defined as partitions. Functions are annotated by adding a comment line "//JSCLOUD MIGRATE" before the function declaration.

After marking partitions as legitimate, JSCloud Packager adds a preamble to the annotated JavaScript document. The preamble contains the implementation of JSCloud Migrator and the JSON serialization library [3] that serializes and de-serializes the JavaScript objects to be transmitted between the JavaScript engine in a web browser and that in the cloud. Moreover, at the position annotated with "^ ENTRY POINT", it inserts the logic to invoke JSCloud Migrator; and similarly, it inserts the logic to record the time spent taken to execute the instance of the partition at the position annotated with "$ EXIT POINT".

C. JSCloud Migrator

1) Migration cost estimation

JSCloud Migrator first estimates the cost of migration and then decides whether to service an invocation request of a legitimate partition locally or remotely. Such a decision is made based on the estimated execution time of the partition. The workflow of JSCloud Migrator is shown in Figure 3.

We define the time for migration as the time required to transmit messages between the web browser and the Java-
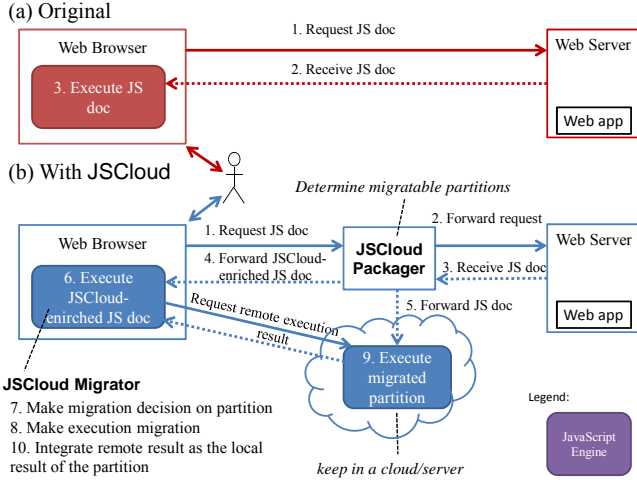
Figure 2. Usage scenario for JSCloud.
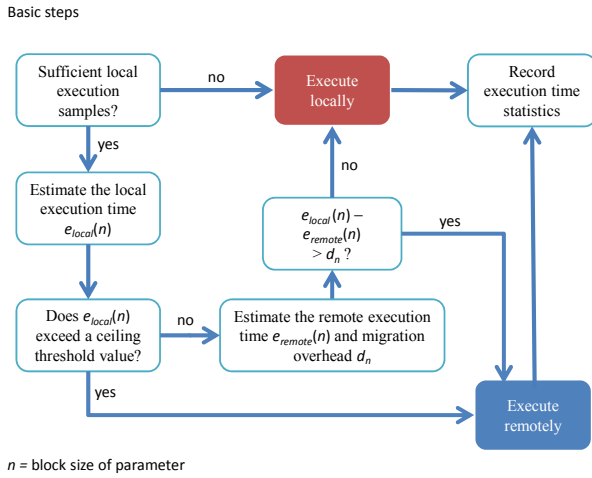

n = block size of parameter

Figure 3. Basic workflow of JSCloud Migrator.

Script engine in the cloud. Ideally, the execution of a partition can be chosen to be conducted remotely if the *time for migration* plus the *time for remote execution* is shorter than the *time for local execution*.

JSCloud Migrator assumes that the more data needed to be processed by a function, the more time needed for the function to compute the result and more time needed to transfer the data over the network. Based on this heuristics, JSCloud Migrator makes a migration decision as follows:

Suppose the input parameter for the partition P that we want to execute has a block size of n. (Note that the block size of an input parameter, say, 64 words, is determined by the specific operating system running on the handheld device and transfer data via its network protocol.) For brevity, let $P(n)$ denote the corresponding execution trace.

Let d_n be the time for migration of parameters and result for $P(n)$, $e_{remote}(n)$ be the time for the remote execution of $P(n)$, and $e_{local}(n)$ be the time for the local execution of $P(n)$. In general, a program on the handheld device runs slower than the same program on the cloud platform. Hence, we may anticipate that $e_{local}(n) > e_{remote}(n)$. Hence, in essence, if

$d_n < e_{local}(n) - e_{remote}(n)$, JSCloud Migrator will decide to execute the partition remotely.

However, the values of $e_{local}(n)$, $e_{remote}(n)$, and d_n cannot be known in advance. They should be estimated, and the estimation procedure for $e_{local}(n)$ and that for $e_{remote}(n)$ and d_n will be elaborated in the next two subsections, respectively.

2) Estimation of local execution time

JSCloud Migrator first checks whether enough samples to estimate the local execution time are available. Specifically, if there are *no more* than two samples, JSCloud Migrator simply fulfills the partition execution request by executing $P(n)$ locally. For instance, when the web application has just been loaded, no partition can have been executed either locally or remotely; as such, we do not have any samples of local execution times. In this case, JSCloud Migrator executes $P(n)$ locally and measures the time taken.

Suppose there are already *more than two* samples that JSCloud Migrator has executed locally. For every such sample with a block size of x, JSCloud Migrator has collected its local execution time, denoted by $e_{local}(x)$. Now, JSCloud Migrator aims to estimate the local execution time $e_{local}(n)$ for $P(n)$. It first finds the two most recent samples with block sizes x and x' closest to n such that $x \le n \le x'$, and computes the value of $e_{local}(n)$ such that the ratio $e_{local}(x) : e_{local}(n) : e_{local}(x')$ is the same as the ratio $x : n : x'$. In other words, it performs a linear interpolation. If an interpolation is not feasible, JSCloud Migrator finds the two most recent samples with the largest two (and smallest two, respectively) sampled block sizes if n is greater than (and smaller than, respectively) the block size of any such sample, and extrapolates the line to find the value of $e_{local}(n)$ such that the ratio $e_{local}(x) : e_{local}(x') : e_{local}(n)$ is the same as the ratio $x : x' : n$.

There are, however, corner cases. If the estimated $e_{local}(n)$ is smaller than 0, JSCloud Migrator will consider the situation as having insufficient samples for estimation, and will fulfill the partition execution request locally. When the slope of the line connecting $(x, e_{local}(x))$ and $(x', e_{local}(x'))$ is negative, the estimation may not be accurate, and hence JSCloud Migrator will also execute $P(n)$ locally.

3) Estimation of remote execution parameters

After estimating $e_{local}(n)$ with a decision that $P(n)$ *may* be executed remotely, JSCloud Migrator then checks whether $e_{local}(n) >$ a threshold value (say, 100 ms as we did in the experiment, which can be adjusted arbitrarily). If it is the case, JSCloud Migrator executes $P(n)$ remotely. This is because the time needed for local execution can be large, which may deplete the battery easily. To prolong the usable hours, it would be a wise decision to remotely execute the relatively heavy computation tasks to the cloud.

On the other hand, if $e_{local}(n)$ does not exceed the threshold value above, it means that the decision to execute $P(n)$ remotely may not be obvious. As such, applying the sample ratio formulas that estimate $e_{local}(n)$ by linear interpolation or extrapolation, JSCloud Migrator further estimates $e_{remote}(n)$. Note that the handling of corner cases when estimating $e_{remote}(n)$ is similar to that of corner cases when

estimating $e_{local}(n)$: JSCloud Migrator considers that there is insufficient evidence to deem that remote execution can bring in additional benefit to complete the execution of $P(n)$ earlier, and hence it executes $P(n)$ locally.

The estimation of the overhead d_n is more complicated. We also note that, once $P(n)$ is executed remotely, we can obtain the actual migration time overhead d_n, which can be used for future estimation of d_n.

The migration time is determined by the time for serializing the parameter and result into JSON format [3], transmitting the data in JSON format over a network between the local web browser and a remote JavaScript engine, and deserialize the data back to the corresponding JavaScript objects. As the network connection may vary over time, JSCloud Migrator uses z most recent overhead timing samples (or as much as available if there are less than z samples) for the estimation of d_n.

Suppose s samples are available, where $s \leq z$, with time overheads of $d^{(i)}$ and block sizes of x_i for $i = 1, 2, ..., s$. We first calculate the weighted average time overhead per block size $r = \sum_{i=1}^{s} w_i(d^{(i)}/x_i)$. Each weight w_i for $d^{(i)}/x_i$ is computed by $timestamp_i - timetsamp_1$, where $timestamp_i$ is the time when the sample for $d^{(i)}$ is obtained. In other words, a more recent sample has a higher weight. As we have explained, the tasks involved with migration depend on the data size. On average, the value of d_n is monotonically increasing with the block size n of the input parameter of the partition, which can be computed accurately. As such, JSCloud Migrator multiplies r by n to obtain d_n.

If $d_n < e_{local}(n) - e_{remote}(n)$ or if $e_{local}(n)$ is greater than a threshold value, JSCloud Migrator executes $P(n)$ remotely. Otherwise, the partition is executed locally.

To overcome the limitation of insufficient samples initially, JSCloud Migrator allows two trials of remote executions of the partition (of any block size n) if $e_{local}(n) >$ a threshold value.

In the above procedure, every partition has its own set of parameters for estimation. As such, the cost estimation may be more accurate, but at the same time, it requires more rounds of local executions of the partitions as a whole, which consumes more energy than when using a global set of parameters.

We also note that before performing the actual migration, JSCloud Migrator also checks whether the parameters of the partition contain values of primitive data types only. If it is not the case, JSCloud Migrator will run $P(n)$ locally.

4) Limitations of the cost estimation approach

There are other limitations of our approach. We discuss some selected ones in this subsection.

First, the approach requires at least two samples that are obtained dynamically. As future work, we may replace them by a randomized approach.

In our approach, it requires linear interpolation or extrapolation of data points to compute $e_{local}(n)$ and $e_{remote}(n)$. The use of the other methods such as pattern classification or Bayesian classifiers may be more useful. However, they incur more computation cost, which may weaken the benefit of remote execution.

We use the block size of the parameter of a partition P as a proximity indicator of the computation workload for the partition P to act on the parameter as well as the amount of data transmission effort. This choice follows the work of Chun et al. [1] and is not general. However, for many applications such as the display of a photo album or computing the statistics based on a list of entries (e.g., a news feeder), such a design decision seems applicable.

5) Alternative approaches

Apart from the cost estimation approach, we also explored alternatives, namely, random outsourcing of computations, outsourcing of computations at regular intervals, and outsourcing of all computations. The evaluations are discussed in Section IV.B.

D. Other parts of JSCloud

The major component in the other parts of JSCloud includes the migration logic. The main issue is that we are working in the JavaScript environment. We simply run a JavaScript call to the remote server and waits for the result to return owing to the lack of a good synchronization mechanism in JavaScript. A drawback of this strategy is that it blocks the remaining parts of the web client from receiving other results, such as messages for AJAX callback functions.

IV. EVALUATION

In this section, we evaluate JSCloud. It covers two aspects: the overhead incurred by JSCloud Packager and the performance gain achieved by JSCloud Migrator.

The experimental subject is a webpage that embodies the sort program shown in Figure 1. All the machines were run in the same university computer laboratory. We used the campus Wi-Fi hotspot available in the laboratory and the 3G network provided in the street by a major 3G operator. The JavaScript engine in the cloud and the web applications were hosted on device D2 (see Table 1). The MacBook Pro and iPod Touch were connected to the Internet via Wi-Fi to a router that accesses the Internet from a cable modem. The BlackBerry, iPhone 4S, and (Android-based) Galaxy S2 were connected to the Internet via the 3G network. We implemented JSCloud by a total of 39,723 bytes of JavaScript. We set all the threshold values and the value of z stated in Section III to 100 ms and 100, respectively. We used the Google V8 JavaScript engine [13] to emulate a JavaScript engine in a cloud.

TABLE 2. TIME SPENT BY JSCLOUD PACKAGER ON JAVASCRIPT DOCUMENT WITH DIFFERENT NUMBERS OF PARTITIONS FOR ANALYSIS

Number of Partitions	8	12	50	110	130
Processing Time (ms)	2	4	10	53	64

TABLE 3. LOAD TIME FOR JSCLOUD-ENRICHED JAVASCRIPT DOCUMENTS

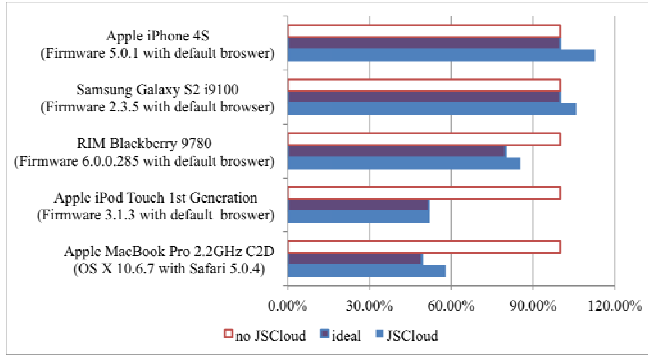| No. of Partitions	Load Time (ms)		
	Without JSCloud	With JSCloud	Overhead
1	43	73	30
10	50	138	88
100	309	805	496

Figure 4. Comparison of performance of JSCloud Migrator.

A. Overheads of *JSCloud Packager*

We have elaborated in Section III.B, the entry and exit marks of each annotated partition require the expansion of code that implements JSCloud Migrator. Hence, a program with more functions requires more preparation and analysis time. We have experimented with a JavaScript document containing different numbers of copies of the `merge_sort` function shown in Figure 1 using device D2. We find that the analysis time needed is acceptable. See Table 2 for the experimental result.

Our approach also requires code expansion at each position with the markers "^ ENTRY POINT" and "$ EXIT POINT" (see Section III.B for more details). They increase the code size to be downloaded to a web browser. This will lengthen the load time needed to display a web page. The experiment was to have the web browser, the web server, and JSCloud Packager all running on device D2 using the reverse proxy configuration. As a comparison, we also measured the time taken by D2 to load the whole Google homepage, which took 1150 ms. The result is shown in Table 3. We find that the overhead is reasonable.

B. Comparative Effects of *JSCloud Migrator*

In this section, we evaluate the performance of JSCloud Migrator after JSCloud Packager has annotated the `merge_sort` function of the subject. To account for the differences in execution performance of different devices,
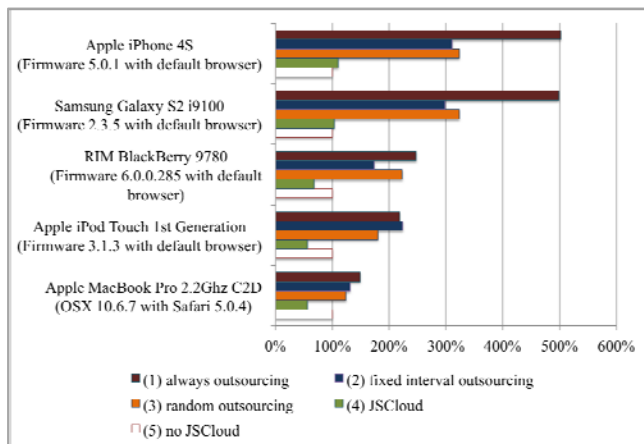

Figure 5. Comparison of performance of alternative outsourcing approaches.

the lengths of the arrays were adjusted individually for each device so that the sorting takes around 77,000 ms without JSCloud. In other words, the value of 100% represents 77,000 ms of execution time for all devices.

The result is shown in Figure 4. There are three bars for each device. The darker bar represents the ideal execution scenario that each migration decision is perfect (that is, the decision always results in the shorter execution time), the lighter bar shows the actual result of JSCloud Migrator, and the unfilled bar is the result without using JSCloud.

In the case of MacBook Pro and iPod Touch, JSCloud Migrator brought the execution time down to 57.95% and 52.15% of the original, respectively. The execution was almost twice as fast. The performance gained by JSCloud on these two devices was high. We believe that it is due to two reasons: (1) the internet connection is fast and stable, and (2) the JavaScript engine on the two devices are slow.

For the BlackBerry 9780, JSCloud Migrator brought the execution down to 85.35% of the original. We find that the 3G network in the experimental period is of high latency and instability. It made the estimation inaccurate.

On the iPhone 4S and Galaxy S2, JSCloud Migrator increased the execution time to 112.85% and 105.85%, respectively. The JavaScript engines on these two devices were already efficient enough and any migration would suffer from the high latency of the 3G network. The ideal execution time of 100% suggests that the best performance is achieved when no partition invocation is served remotely (when JSCloud Migrator is not used).

We also evaluated the performance of alternative outsourcing approaches. They include computations that were outsourced (a) randomly, (b) at regular intervals (outsourcing every other computation) and (c) all the time. They are referred to as "random outsourcing", "fixed interval outsourcing", and "always outsourcing", respectively.

The results in Figure 5 show that using any of the three alternative approaches yielded poorer results than using JSCloud. The unnecessary migration of computations incurred additional migration delays, which in turn degraded the performance. As we adjusted the array lengths for each device such that computation takes around 77,000 ms to complete without JSCloud, and given that iPhone 4S and Galaxy S2 had higher performance than BlackBerry 9780, the arrays on iPhone 4S and Galaxy S2 were longer. This finding explains the greater increase in execution time on iPhone 4S and Galaxy S2 than that on BlackBerry 9780. This comparison with alternative outsourcing approaches demonstrates the effectiveness of our cost estimation approach.

The overall result shows that JSCloud is useful on slow devices. For high-performance handheld devices, other methods to save energy may need to be explored.

V. RELATED WORK

CloneCloud [1] is the most recent related work. It requires manual annotations to API methods to define the partitioning policies for each type of virtual machine. For instance, different Android operating systems by various vendors may have been customized. CloneCloud requires a

separate manual annotation for every member of such family of operating systems. JSCloud is implemented as a JavaScript framework and complies with the standardized JavaScript. Therefore, as long as the JavaScript engine is standard-complaint, our approach is applicable to handle any program running on it. The result presented in Section IV.B shows that this assumption applies to different browsers on different handheld devices. We are not aware of a similar adaptation to JavaScript programs in the literature.

Another difference between JSCloud and CloneCloud lies in our algorithm that estimates various migration parameters and makes a migration decision. JSCloud uses a simplified method-local approach to trend estimation in order to save the computation overhead in making a migration decision, whereas CloneCloud uses a more complicated approach that involves dynamic profiling and global optimization among the statistics of different methods. Our result in Section IV.B has shown that on some devices, executing all methods locally is optimal, and hence spending more effort to compute the optimal solution may defy the purpose of enhancing efficiency by remote execution.

Similar to CloneCloud, MAUI [4] optimizes execution time or energy consumption of a mobile device by estimating and trading off the cost of local execution with the transmission of remote execution, but MAUI requires more programmer help to annotate methods. Based on CloneCloud, Zhang et al. **Error! Reference source not found.** proposed an elastic application model that aims to remove the constraints of mobile platforms through a distributed framework. This model partitions a single application into multiple components called Weblets, and dynamically configures these Weblets to execute in the cloud or mobile devices. However, the cost estimation introduced by this elastic application model can be complicated. For example, there are four attributes to consider when calculating the cost: power consumption, monetary cost, performance attributes, and security and privacy. As mentioned above, spending more effort to compute the solution may decrease the efficiency rather than improving it. Other work like Odessa [12] dynamically makes offloading decisions based on runtime profiles. Our work complements all these studies.

It should also be noted that using a simple logic to save energy (while lowering the performance requirements on the hardware platform) may be further improved by using a classification approach as what we did in EClass [8].

VI. CONCLUSION

In this paper, we have proposed JSCloud, a generic framework that supports the remote execution of JavaScript programs. We have elaborated on its key design and reported the preliminary empirical result of its application to various handheld devices.

REFERENCES

[1] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "CloneCloud: elastic execution between mobile device and cloud," in *Proceedings of the 6th Conference on Computer Systems* (*EuroSys 2011*). New York, NY: ACM, 2011, pp. 301–314.

[2] K. Coleman, *Introducing Gmail Labs*. Google Gmail, 2008. Available from http://gmailblog.blogspot.hk/2008/06/introducing-gmail-labs.html.

[3] D. Crockford, *JSON in JavaScript*. GitHub Inc., 2010. Available from https://github.com/douglascrockford/JSON-js.

[4] E. Cuervo, A. Balasubramanian, D.-K. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: making smartphones last longer with code offload," in *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services* (*MobiSys 2010*). New York, NY: ACM, 2010, pp. 49–62.

[5] *Facebook Mobile*. Facebook. Available from https://www.facebook.com/FacebookMobile. Last access January 2012.

[6] X. Gu, A. Messer, I. Greenberg, D. Milojicic, and K. Nahrstedt, "Adaptive offloading for pervasive computing," *IEEE Pervasive Computing*, vol. 3, no. 3, pp. 66–73, 2004.

[7] F. Hohl, P. Klar, and J. Baumann, "Efficient code migration for modular mobile agents," in *Proceedings for the 2nd ECOOP Workshop on Mobile Object Systems*. Berlin, Germany: Springer, 1997.

[8] E. Y. Y. Kan, W. K. Chan, and T. H. Tse, "EClass: an execution classification approach to improving the energy-efficiency of software via machine learning," *Journal of Systems and Software*, vol. 85, no. 4, pp. 960–973, 2012.

[9] A. Messer, I. Greenberg, P. Bernadat, D. Milojicic, D. Chen, T. J. Giuli, and X. Gu, "Towards a distributed platform for resource-constrained devices," in *Proceedings of the 22nd International Conference on Distributed Computing Systems* (*ICDCS 2002*). Los Alamitos, CA: IEEE Computer Society, 2002, pp. 43–51.

[10] R. B. Miller, "Response time in man-computer conversational transactions," in *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I* (*AFIPS 1968* (*Fall, Part I*)). New York, NY: ACM, 1968, pp. 267–277.

[11] S. Ou, K. Yang, and J. Zhang, "An effective offloading middleware for pervasive services on mobile devices," *Pervasive and Mobile Computing*, vol. 3, no. 4, pp. 362–385, 2007.

[12] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan, "Odessa: enabling interactive perception applications on mobile devices," in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services* (*MobiSys 2011*). New York, NY: ACM, 2011, pp. 43–56.

[13] *V8 JavaScript Engine*. Google. Available from http://code.google.com/p/v8/. Last access January 2012.

[14] X. Zhang, A. Kunjithapatham, S. Jeong, and S. Gibbs, "Towards an elastic application model for augmenting the computing capabilities of mobile devices with cloud computing," *Mobile Networks and Applications*, vol. 16, no. 3, pp. 270–284, 2011.