POWER: A Cloud-Based Mobile Augmentation Approach for Web- and Cross-Platform Applications

Andreas Reiter, Thomas Zefferer
Institute for Applied Information Processing and Communications (IAIK)
Graz University of Technology
Inffeldgasse 16a, 8010 Graz, Austria
{andreas.reiter, thomas.zefferer}@iaik.tugraz.at

***Abstract*—Despite their continuously growing popularity, mobile end-user devices still suffer from limited computing resources. This complicates the use of complex mobile applications that require resource-intensive computations. Recently, several frameworks have been developed that enable mobile applications to follow the cloud-based mobile augmentation (CMA) approach. This approach defines a strategy to dynamically outsource resource-intensive tasks to external resources. None of the existing frameworks focuses on cross-platform applicability and interoperability issues. We propose an innovative approach, which provides a high level of flexibility in order to meet requirements of different mobile use cases and application scenarios. The POWER framework is applicable on all major mobile platforms and supports various types of external resources. Furthermore, we prove the feasibility of the proposed framework on current mobile end-user devices by means of a concrete implementation. Evaluation results yield significant performance boosts for resource-intensive tasks on mobile end-user devices. With the POWER framework we pave the way for complex mobile applications.**

I. Introduction

Powered by the availability of powerful mobile end-user devices and by the continuous improvement of wireless communication networks, mobile computing has emerged as new predominating computing paradigm. As mobile devices such as smartphones or tablet computers gradually replace classical end-user devices, more and more services and applications are nowadays tailored to a mobile use. Although mobile devices are continuously improved, they are still more limited in terms of storage capacity and processing power compared to desktop PCs or laptops. This complicates the use of resource-intensive applications on mobile devices and prevents a full adoption of the mobile-computing paradigm.

At the same time, the general availability of storage capacities and computing resources increases steadily. On the one hand, cloud services offer virtually unlimited resources usually based on consumption-based pricing models. On the other hand, the number of smart devices increases steadily as well yielding a situation, in which users are permanently surrounded by proximity devices with at least rudimentary storage and processing capabilities.

Limited capabilities of mobile end-user devices and the omnipresent availability of resources, recently yielded a new promising computing concept called cloud-based mobile augmentation (CMA). The basic idea behind the concept of CMA is the dynamic outsourcing, i.e. offloading, of resource-intensive tasks from mobile end-user devices to currently available external resources. Following the CMA approach, mobile applications determine the current availability of external resources at runtime and dynamically offload resource-intensive tasks to these resources in order to improve their overall performance.

As the concept of CMA is rather new, current implementations of this concept do not tackle the usage of different types of resources, and are focusing on single types of mobile devices and operating systems. To target this problem, we propose POWER, a new and innovative CMA framework in this paper. In contrast to existing solutions, the proposed framework provides interoperability and cross-platform applicability by achieving compatibility to all major mobile platforms and operating systems. We show by means of a concrete implementation that the proposed CMA framework is feasible in practice. Finally, we evaluate and demonstrate its capabilities by means of a real-world application. This way, we show that the proposed CMA framework has the potential to leverage the concept of CMA and to pave the way for a migration of resource-intensive applications to mobile end-user devices.

II. Background and Related Work

CMA is a computing concept which enables resource-constrained devices to offload resource-intensive tasks to other devices or cloud-computing resources. A related concept called surrogate computing has already been introduced in 2001 by Satyanarayanan [1]. At that time, main focus had been put on surrounding devices, as the utilization of cloud-computing concepts was not state of the art. The surrogate-computing concept enables a device to make use of other available computing resources within reach. Applications can offload tasks to these resources to speed up the execution. Beside the speed-up, this approach saves energy and therefore extends battery lifetime, which especially is of high interest for mobile devices.

CMA extends the surrogate-computing approach by not only utilizing proximity devices with free resources, but by consuming e.g. available cloud resources as well. The connection to the different types of resources can be established using Wi-Fi, 3G, 4G, or any other kind of communication technology. Reiter and Zefferer [2] propose a classification scheme for external resources, based on their distance to the mobile device and on their degree of mobility. Concretely, they differentiate the following three types of resources: *Distant Immobile Clouds* as resources with high availability but also

a larger distance to the user and therefore a higher latency compared to the others, *Proximate Immobile Computing Entities* as resources that are located near to the user and therefore have a low latency, and *Mobile Computing Entities* as other mobile device resources which can be utilized as computing resources.

By considering various factors, a CMA framework decides which type of available resource to use and if a pending computation should be offloaded at all. This decision process considers the following factors:

- *Network latency:* The latency of network connections that access external resources has major impacts on the performance of an offloaded task. Evaluation results from Cuervo et al. [3] show that the energy consumption of an offloading operation almost doubles, if the round-trip time increases from 10ms to 25ms. Together with the bandwidth, the network latency also determines how fast the execution can be migrated from a client device to the offloading server.
- *Bandwidth:* Depending on the size of the offloaded part and the size of the data associated to this part, the bandwidth to the external resource has a major impact on the migration speed of the execution to the external resource.
- *Computational power:* An offloading framework can run under different profiles, e.g. to maximize battery lifetime or to achieve the best possible performance. The selected profile influences a CMA framework's offloading decisions.
- *Estimated run-time:* The estimation or prediction of the run-time can be based on sophisticated analysis tools or on data collected in previous runs.

All of these factors are fed into the CMA framework's decision engine. Based on these inputs, the decision engine determines the optimal way of execution.

There are already solutions in place, targeting different aspects. Reiter and Zefferer [2] provide an overview of the current CMA-framework landscape, and different approaches followed by available CMA solutions. The most relevant solutions are briefly sketched in the following.

The CMA framework *MAUI* [3] aims at an energy-optimizing and hence battery-saving execution. It is tailored to managed code environments and eases integration into existing applications. *CloneCloud* [4] uses externally executed virtual clones of the end-user device to offload tasks. This is advantageous, as the system can operate on unchanged applications, but requires a modified operating system to catch the relevant offloading spots. The CMA framework *ThinkAir* [5] combines the concepts of MAUI and CloneCloud and adds more flexibility and scalability. Instead of requiring the end-user device to run a custom operating system, ThinkAir induces an additional compilation step where code for remote execution is generated. The CMA solution *COSMOS* [6] provides Offloading-as-a-Service and does not dedicate a single VM to a specific user. Instead, the so-called COSMOS-Master keeps track of all available resources and distributes work among available VMs.

This brief overview shows that existing CMA frameworks and solutions follow different approaches to offload resource-intensive tasks to external resources. However, all of these solutions come with certain drawbacks, especially when it comes to interoperability among different operating systems or cross-platform applications. All of the existing solutions are tailored to specific platforms or programming languages which are only available on single platforms.

III. Requirements

To better understand the needs of CMA frameworks to also be applicable in cross-platform scenarious and at the same time protecting user's privacy, we identified the following requirements. Our analysis is based on the requirements as proposed by Reiter and Zefferer [2]

- *R1 - Reliability:* Reliability refers to the availability of the system.
- *R2 - Integrity:* Data offloaded to external resources for processing must not be modified by unauthorized parties.
- *R3 - Privacy and confidentiality:* Sensitive data offloaded to external resources must be kept confidential.
- *R4 - Non-repudiation:* A user cannot deny that he or she has used a certain external resource. This enables e.g. a reliable billing of resource usage.
- *R5 - High isolation level:* The framework needs to have a high isolation level among different users, even if they are utilizing the same resources.
- *R6 - Avoidance of misuse of provided resources:* Resource providers must have means to prevent a misuse of provided resources for fraudulent activities.

In order to appropriately consider aspects related to interoperability and cross-platform applicability, we extend the set of requirements by means of the following requirements.

- *R7 - End-user interoperability:* CMA frameworks must be applicable on all major mobile platforms.
- *R8 - Resource interoperability:* Frameworks must not statically offload tasks to fixed external resources, they have to dynamically decide at run-time which resource is currently best suited to adhere to the used profile.

Under consideration of these requirements, a CMA framework is proposed. The architecture of the proposed framework is derived in the following section.

IV. Architecture

In this section, the architecture of the proposed CMA framework is introduced. For this purpose, a general technology-independent architecture is defined first, which can be applied to arbitrary technologies. To further develop this general architecture towards a concrete implementation, a technology-specific architecture is derived afterwards. Derivation of the technology-specific architecture is based on an elaborate selection of currently available technologies. The architecture introduced in this section is finally evaluated against the requirements defined in Section III.

A. General Architecture

We propose a general architecture as shown in Figure 1. This architecture comprises a central repository that stores complete application packages. Each application package contains all necessary files and dependencies required to run a specific application. The central repository is shared among all potential external resources. It is up to the resource provider to only support a certain subset of applications. To enable the referencing of application packages at run-time, a couple of identifiers are used. At compile time, each application part considered for offloading is assigned with an identifier that is unique in the context of the application package. In addition, each package is assigned with a globally unique identifier as well.

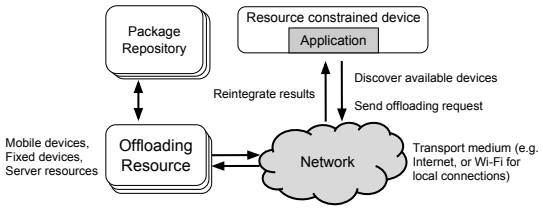

Fig. 1: High-level architecture

During run-time, a resource-constrained device maintains a list of available offloading resources. If the CMA framework decides that a particular part of an application needs to be offloaded, a connection to a suitable offloading resource is established and the offloading process is initiated. Once an offloading resource receives a request, it downloads the complete package from the package repository using the package ID, identifies the part to execute, executes it, and sends back the results to the resource-constrained device, where the results are finally reintegrated into the application. We are aiming at a lightweight approach operating on the source level of the application. This architecture basically enables two approaches. Either the application developer provides implementations in different programming languages for various platforms, or a single implementation utilizing cross-platform frameworks or common programming languages.

B. Technological Considerations

The proposed general architecture is completely technology-independent. To develop this architecture towards a concrete solution, an appropriate technology must be selected to further enhance the architecture's various building blocks. When selecting an appropriate technology, the heterogeneous landscape of current mobile platforms and programming environments must be taken into account. Concretely, to minimize the development effort a single programming language with support for all platforms is desirable. Currently the only language that is supported by all platforms and that can equally be executed on all mobile operating systems is JavaScript. We are totally aware that JavaScript is not a panacea and introduces other problems, but it forms a common basis. JavaScript is the common anchor and technological basis for a new class of programming languages. The compilers of these programming languages produce highly optimized JavaScript code to be executed in the browser. Representative examples are Coffeescript [7] or TypeScript [8]. The Dart [9] programming language, which is heavily supported by Google, even goes one step further. It does not only define a new syntax, which can be compiled to JavaScript, but provides a complete Dart ecosystem. The Dart ecosystem is based on a Dart VM executing native Dart code. In addition Dart applications can be compiled to JavaScript files to be run in the browser or in standalone JavaScript environments. Dart thereby hides the complexity and still remaining incompatibilities between the different JavaScript engines from the developer. The Dart ecosystem further offers access to public and private package repositories, which are fully integrated into the build process.

The provided overview shows that there are currently various programming languages available that base on JavaScript and hence provide cross-platform applicability. From the surveyed languages, Dart appears to be best evolved and most promising approach to meet requirements of CMA-based applications. Furthermore it eases the development of cross-platform applications by hiding peculiarities. At first sight Dart seems to be a very tailored and narrowed choice. Currently the landscape of available native Dart applications is very limited, but Dart features a high grade of compatibility with JavaScript in just a few lines of code. From a JavaScript enabled browser's perspective, there is no difference if an application uses HTML5 and JavaScript technology, or if it uses HTML5 and Dart technology. Furthermore, there is currently active development in progress to create a framework called Sky[1] enabling direct mobile app development in the Dart programming language. Currently the project primarily targets the Android platform but may be extended to also support other platforms.

Web-based applications are no longer limited to the web. In fact, web-based applications are emerging even on classical and mobile end-user devices. For instance, Firefox OS [10] natively supports HTML5 with JavaScript applications. Furthermore, Windows Phone also natively support apps based on HTML5 and JavaScript. For all other platforms, cross-platform solutions like Adobe PhoneGap [11] or Appcelerator Titanium [12] exist. They wrap web-based applications into native packages and display content by using web-view technology. Of course, there are also disadvantages of using web-based technologies on mobile devices. Native applications get the look and feel of the respective platform out of the box, whereas web-based solutions require additional effort or libraries to mimic a platform-specific look and feel. Furthermore, applications utilizing web technologies still do not achieve the performance of native applications, even though the performance of web-based applications is continuously improved[2].

The most significant advantage of web-based technologies is the fact that the hurdle of maintaining multiple code bases for different platforms is removed, which reduces maintenance effort. In the context of CMA, we see this as a great chance to build a system that is available on all major platforms, is not locked to a single system, and does not need major adaptations once new operating-system versions are released. To fully employ the potentials of web-based technologies, we have chosen Dart as our fundamental programming language

[1] https://github.com/domokit/sky_sdk

[2] http://arstechnica.com/information-technology/2013/05/native-level-performance-on-the-web-a-brief-examination-of-asm-js/

for our proposed CMA framework. In the following subsection we derive a technology-specific architecture that is tailored to the Dart ecosystem.

C. Technology-Specific Architecture

The resulting refined technology-specific architecture as shown in Figure 2 takes into account the decision to rely on Dart and applies refinements on the general architecture. According to the architecture, it is assumed that the communication with proximity resources takes place using low-latency Wi-Fi connections. High-latency 3G/4G connections are used, if no suitable proximity resources are available. Wi-Fi connections can also be used to connect to resources that are not in the proximity of the user. In this case, potentially untrusted resources are used as gateways.

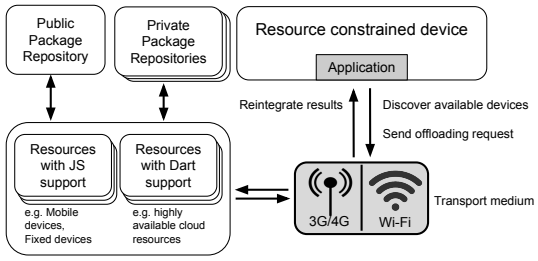

Fig. 2: Technology specific architecture

For the package repositories we are re-using entities as already provided by the Dart ecosystem. Dart provides a package repository system, with the option to use a centralized repository or a private repository. Developers can place application packages there to be referenced by offloading resources. Deploying new application versions to this repositories is completed in a matter of seconds with an already pre-existing tool chain. On the offloading resource level the refined architecture envisages the following two different types of resources:

- *Resources with native Dart support:* These resources reach a high performance, but can be only applied on target architectures providing full Dart support. This type of resources is hence mainly available in cloud-based environemnts.
- *Resources with JavaScript support:* These resources cannot provide the high performance of native Dart resources and are targeted at other end-user devices in the surrounding of the user. As JavaScript is supported by virtually all devices featuring a web browser, all end-user devices like notebooks, tablets, computers, or smartphones represent potential target offloading resources.

At the architectural level we are recommending to organize the offloading resources in a distributed manner, but we are intentionally not locking the resource discovery and communication to certain techniques as current web and cross-platform technologies are evolving fast and are changing frequently.

Even though the refined architecture has been tailored to the Dart programming language, similar architectures can easily be derived for other technologies as well. The general architecture proposed in Section IV-A is an ideal starting point for these attempts. Nevertheless, this paper focuses on the Dart-specific architecture. An evaluation of this architecture by means of the requirements defined in Section III is provided in the following subsection.

D. Evaluation Against Requirements

Eight requirements have been defined as relevant for CMA frameworks. In the following, the proposed technology-specific architecture is evaluated against these requirements. This way, the suitability of the architecture is assessed.

1) Reliability: Employed offloading resources are organized in a distributed manner with no single point of failure. Nevertheless, the package repository being an integral component of the proposed architecture needs to be replicated to different locations to achieve redundancy. In a real world example, the package repository can for instance be located and replicated in a cloud environment. This way, the requirement for reliability can be met by adopting an appropriate deployment strategy. The distributed organization of the offloading resources can be implemented using overlay networks organized in a distributed hashtable approach, so that no central servers are required.

2) Integrity: From an resource operator's point of view, digital signatures can be applied to the packages to assure that they have not been changed. From the user's point-of-view the integrity can only be assured when the user has a certain level of trust in the operator. Although, again highlighting the distributed manner of the offloading resources, by applying end-to-end security techniques, untrusted resources can be used as low-latency gateways to trusted resources. Still, we consider the requirement for integrity as partially met only at this point. As future work, means must be provided that enable the secure offloading and full utilization of untrusted resources.

3) Privacy and confidentiality: In general, it is up to the offloading engine to decide where to execute particular parts of an application. Furthermore, it is up to the developer to properly specify the parts of an application that operate on sensitive data. Utilizing data-security policy languages, the flow of sensitive data through the application can be tracked. Based on this flow graph, the offloading engine can decide where to offload a particular part of an application. Hence, there exist means to design and develop a mobile application such that privacy and confidentiality is assured in CMA scenarios.

4) Non-repudiation: By applying digital signatures to issued offloading requests, resource-requesting users can be reliably identified by resource providers. This requires that users can offer signatures which can be mapped to unique users by the offloading resources. Another and possibly more practical way is to use decentralized authentication systems like OpenID or the OpenID Connect identity layer. This way, the requirement for non-repudiation can be met.

5) High isolation level: Browsers already feature sophisticated isolation and sandboxing mechanisms. Running the Dart-generated JavaScript code in a separate HTML IFRAME provides a high isolation level. For the native Dart environment, there are still further improvements necessary, as Dart does not integrate a sandboxing mechanism. Therefore, this requirement is regarded as partially fulfilled only.

6) Avoidance of misuse of provided resources: The misuse of provided resources can be avoided either by locking out

fraudulent users, or by analyzing the provided source code to detect bogus applications. The threat of resource misuse can be countered by establishing a pay-per-use model. This can prevent misuse, as all consumed computational power is billed. Thus, the requirement to avoid misuse of provided resources can be met by adopting suitable strategies.

7) End-user interoperability: As the technology-specific architecture assumes reliance on Dart and JavaScript, it is available on all major platforms. Therefore, the requirement for end-user interoperability is met.

8) Resource interoperability: Regardless of the used platform, each user can use all provided external resources, also combining different platforms. The proposed architecture considers the possibility of different target resources ranging from mobile proximity devices to powerful servers in the cloud. Hence, the requirement for resource interoperability is met.

In summary the obtained evaluation results show that most relevant requirements of CMA frameworks are already met on architectural level by the solution proposed in this paper.

V. Implementation

In this section, a concrete implementation of the proposed architecture is presented and discussed. The implemented framework makes use of Dart's browser-integration feature, but also targets standalone Dart virtual machines. In Section V-A we are elaborating on our basic building blocks. Afterwards in Section V-B we are evaluating the current state of the framework by applying it to a real-world application.

A. Basic Building Blocks

Most functionality of the implemented CMA framework is covered by the three basic building blocks: Annotator, Offloading Transformer, and Offloading Engines. Implementation details of these building blocks are discussed in the following subsections.

1) Annotator: The Annotator defines a marker attribute, which represents a simple way of annotating relevant parts of an application considered for offloading. The use of this marker attribute is illustrated in Listing 1.

Listing 1: Applying the marker attribute to a method

```
@OffloadMe(SENSITIVITYLEVEL) Future<String> sample(){
  <your calculation>
}
```

Applying these attributes throughout an application provides the offloading engine with an initial developer-defined offloading decision. However, it is still up to the offloading engine to decide at run-time, if offloading is performed or if the developer decision is ignored. The attribute further allows to define the sensitivity of the marked method and processed data. Based on the specified sensitivity, the offloading engine is able to derive the required trust level of the offloading resource.

2) Offloading Transformer: The Offloading Transformer preprocesses the source code of applications at compile time. It analyzes all relevant source files and extracts annotated parts that are marked for offloading. The Offloading Transformer modifies these source-code parts such that they can be processed by the designated offloading engine at run-time. Listing 2 shows the transformed source code from Listing 1.

Listing 2: Transformed source code

```
Future<String> sample(){
 var m=() { <your calculation> };
 String s="""() { <your calculation source> }""";
 return engine.execute(m, s, [], "sample",
  "example/example.dart", "samplePackage-0.0.1",
  "samplePackage");
}
```

At run-time, the offloading engine has the choice to execute the preprocessed method directly (using the passed method variable *m*) or remotely based on the provided method source code, method identifier, file identifier, package identifier, and package name. All these parameters minimize the required amount of data being transferred to the offloading resource.

Generally speaking, the Offloading Transformer modifies and redirects the execution flow of a particular method. In an unmodified execution flow a method is executed directly. In the modified execution flow a method call is redirected to the offloading engine, where the decision happens if offloading should be performed. If offloading should be performed the execution is migrated to an available offloading resource. Once the execution result is available from the external resource or the local execution, it is returned and re-integrated.

3) Offloading Engines: The framework includes two Offloading Engines. The Dart-VM Engine relies on Dart's command-line environment and a JavaScript based engine for browser environments. The implementation of both engines is similar and involves the following steps. Both types are separated into a client and a server part. The client part is responsible for providing the server part with all the required information to run a particular method of the application, and to re-integrate the results. The server part is responsible for loading the correct application and the relevant state of the application as received from the client, before executing. Once the execution completed, the result is returned to the client. To speed up the loading process the server can cache particular applications or may even pre-load frequently used packages.

The process includes additional steps for the browser engine. The heart of the browser engine is a Dart-to-JavaScript compiler inside the browser. Once an offloading request is received, the server part of the browser engine basically performs the same steps as described before. Additionally, the Dart-to-JavaScript compiler is invoked to compile the modified package and all its required dependencies to a single JavaScript file. The JavaScript file is then executed in a dynamically added HTML IFRAME and obtained results are posted back to the caller.

For the communication with the offloading resources we are using WebSocket and WebRTC technology. WebSockets provide an asynchronous communication channel between clients and servers based on HTTP. WebRTC is a technology enabling the direct communication between browsers, without requiring intermediary servers. WebRTC provides more flexibility in terms of finding a path between two nodes, even if they are behind NATs, but requires more set-up time compared to WebSockets. Using these technologies we are not constrained by the same-origin-policy, which would prevent the browser communication to any other domain than the origin domain. The current implementation makes direct use of WebRTC, for the future we plan to use an overlay network, to gain flexibility and at the same time cover the resource discovery process.

B. Evaluation with Real-World Application

To evaluate the capabilities of the proposed CMA framework, it has been applied to a real-world application. Obtained results, which show the strengths of the underlying concept and reveal rooms for improvement, are discussed in this subsection. We have tested our implementation on a test-system comprising a Motorola G2 smartphone with Android 5 and a desktop computer equipped with an Intel i5 processor at 3.4GHz and operated with Ubuntu 14.04. We focused our testing system on the Android platform for basically two reasons: (a) Android is today's most used mobile platform and (b) POWER makes use of bleeding-edge web technologies, Android with its Chrome browser definitely has the best support for these technologies among all mobile browsers[3]. They are connected through a low-latency WiFi connection. Higher latencies have direct impact on the performance values. The offloading engine should decide at which latency it is still beneficiary to offload. We have conducted two tests to investigate general programming constructs and to show its applicability to real-world applications. For the evaluation we assume that the applications are already available on the offloading resources as this overhead may anyhow only appear once, for all benchmarks the offloading engine uses a cached version.

1) General Programming Constructs: Our first test row is based on the ported version of a Java and C++ benchmark[4]. We ported the Fibonacci and the Nestedloop benchmark. Furthermore we added a SHA1 benchmark, performing n-SHA1 calculations in a row. All benchmarks take a single input parameter n which specifies the length of the Fibonacci row, the iterations in the Nestedloop or the number of SHA1 iterations. The goal of this evaluation is to show the lowest number of n where offloading gives benefits in terms of performance. The results are shown in Table I. They show that even for state-of-the-art smartphones, the offloading approach is very suited to gain application performance. The data transmitted to the offloading resource is nearly equal over all tests, the received data depends on the resulting values.

TABLE I: General Programming Constructs benchmark results

Benchmark	n	Data Tx (bytes)	Data Rx (bytes)
Fibonacci	23	~450	~80
Nestedloop	16	~450	~80
SHA1	41	~450	~100

2) Face-Detection Performance: In this test, a face-detection algorithm[5] has been applied to an image composed of 2.000.000 pixels. The application implementing this algorithm has been executed on the mobile device, whereas the entire face-detection functionality has been offloaded to the external resource. The application requires a single offloading call only, as the complete operation has been carried out on the target resource. The image has been transported as link, therefore only a small amount of data had to be transferred. The face detection has taken nearly 19 seconds when being performed directly on the mobile device. The overall execution time could be reduced to 1.8 seconds by employing the proposed framework's offloading functionality. The obtained results again highlight the potential of the proposed framework and its implementation, when being applied to real-world applications. In fact this is an ideal use case, as only little transfer is required to trigger a computational heavy operation.

[3]https://html5test.com/results/mobile.html

[4]http://keithlea.com/javabench/

[5]http://liuliu.me/ccv/js/nss/

VI. Conclusions and Future Work

In this paper we presented POWER, a CMA framework based on a novel architecture. In contrast to existing CMA solutions, the proposed framework provides interoperability and cross-platform applicability by relying on web technologies. Development of the proposed framework's architecture has been based on a set of relevant requirements. A conducted evaluation against these requirements has revealed that the proposed framework meets most requirements already on architectural level. The feasibility of the proposed framework has been evaluated by means of a concrete implementation. The implementation already demonstrates the proposed framework's capabilities to boost the performance of resource-intensive processes on mobile end-user devices and even for web-based applications. We will focus our future work on further improving the framework in terms of resource discovery and end-to-end security. Furthermore, adherence to our defined requirements is our top priority.

References

[1] M. Satyanarayanan, "Pervasice Computing: Vision and Challenges," *IEEE Personal Communications*, pp. 10–17, 2001.

[2] A. Reiter and T. Zefferer, "Paving the Way for Security in Cloud-Based Mobile Augmentation Systems," in *3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (IEEE Mobile Cloud 2015) (Note: to appear)*, 2015.

[3] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Stefan, R. Chandra, and B. Paramvir, "MAUI : Making Smartphones Last Longer with Code Offload," in *Proceedings of the 8th international conference on Mobile systems, applications, and services*, vol. 17, 2010, pp. 49–62.

[4] B. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: Elastic Execution Between Mobile Device and Cloud," in *Proceedings of the sixth conference on Computer systems*, 2011, pp. 301–314.

[5] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *2012 Proceedings IEEE INFOCOM*. Ieee, Mar. 2012, pp. 945–953.

[6] C. Shi, K. Habak, P. Pandurangan, M. Ammar, M. Naik, and E. Zegura, "COSMOS : Computation Offloading as a Service for Mobile Devices," in *Proceedings of the 15th ACM international symposium on Mobile ad hoc networking and computing*, 2014.

[7] T. Burnham and M. Swaine, *Coffeescript: accelerated Javascript development*. Pragmatic Bookshelf, 2011.

[8] G. Bierman, M. Abadi, and M. Torgersen, "Understanding typescript," in *ECOOP 2014 Object-Oriented Programming*, ser. Lecture Notes in Computer Science, R. Jones, Ed. Springer Berlin Heidelberg, 2014, vol. 8586, pp. 257–281.

[9] M. Belchin and P. Juberias, "Darts flightpath so far," in *Web Programming with Dart*. Apress, 2015, pp. 1–11.

[10] Mozilla, "Firefox OS https://developer.mozilla.org/en-US/Firefox_OS, accessed on February 18th 2015."

[11] Adobe, "PhoneGap Documentation Overview, Available Online at http://docs.phonegap.com/ accessed on 10th of March 2015."

[12] Appcelerator, "Titanium Documentation, Available Online at http://docs.appcelerator.com/titanium/latest/ accessed on 10th of March 2015."