SNARF: A Social Networking-inspired Accelerator Remoting Framework

Heungsik Eom*, Pierre St Juste*, Renato Figueiredo*,
Omesh Tickoo**, Ramesh Illikkal**, Ravishankar Iyer**

*Advanced Computing and Information Systems Laboratory
University of Florida
Gainesville, Florida, USA
{hseom, pstjuste, renato}@acis.ufl.edu

**Intel Corporation
2111 N.E. 25th Avenue
Hillsboro, Oregon, USA
{omesh.tickoo, ramesh.g.illikkal, ravishankar.iyer}@intel.com

ABSTRACT

The diminishing size and battery requirements of mobile devices restrict the scope of computations possible on such devices and motivate approaches that support the selective offloading of computations to remote resources. With a variety of resources available to potentially host offloaded computations – such as cloud-provisioned resources, and devices within a user's personal or social network – a key challenge lies in architecting a framework that enables applications to seamlessly discover available services, effectively and securely communicate with them, and be presented with API interfaces that hide the complexities associated with managing the interactions with a remote device from applications and present the abstraction of a local device. In this paper, we outline a framework that addresses these challenges by layering APIs and an offload infrastructure upon a virtual networking substrate that supports TCP/IP networking and widely-used resource discovery protocols. An intelligent runtime scheduling layer monitors the execution environment and provides opportunistic remote offloads based on the performance requirements, offload benefits and expendable power. We demonstrate the feasibility of the approach through experiments that evaluate end-to-end application execution times and energy consumption in offloaded mobile devices, as well as the ability to support universal plug-and-play (UPnP) resource discovery in both local- and wide-area environments.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Client/Server*

General Terms

Design, Experimentation, Performance

Keywords

offload, mobile device, energy consumption, virtual network

1. INTRODUCTION

The diminishing size and battery requirements of mobile devices restrict the scope of computations possible on such devices. Commonly used strategies to combat these limitations can be grouped under two categories:

- *Device Specialization*: Mobile devices target a specific market segment leading to well-defined use cases and limited application deployment. The specialized devices use highly efficient fixed-function hardware to speed up the common case of computation at relatively low power consumption. GPS and medical notification devices fall under this category.

- *Cloud offload*: For more general-purpose devices such as Tablets and Smartphones, applications perform limited data processing at the mobile device itself, and offload more resource-intensive computations to servers in a cloud backend.

While specialized devices address specific niche use cases, they come with inflexibility in use case variations and high cost of manufacture/upgrade. As such, it is highly attractive to design more general-purpose mobile devices with an ability to address multiple usage segments. Since it is not practical to include fixed-function specialized hardware for all the possible use cases with a general purpose mobile platform, offloading compute-intensive, domain-specific tasks to other more capable platforms over the network is an attractive solution. In an ideal case, a mobile application should be able to use computation resources (both local to the platform as well as the ones located remotely over a network) in a seamless fashion. In this paper we present design considerations for such an "aggregated" platform, and an architecture that allows dynamic platform building using both local and remote resources. An application querying the platform is able to enumerate the devices irrespective of their physical location, and the runtime presents resource discovery and job scheduling services. For our implementation, the runtime is supported by the SocialVPN [3], social virtual networking service that implements popular resource identification and discovery protocols on top of the ubiquitous TCP/IP based network stack for efficient and seamless communication over a wide range of network topologies.

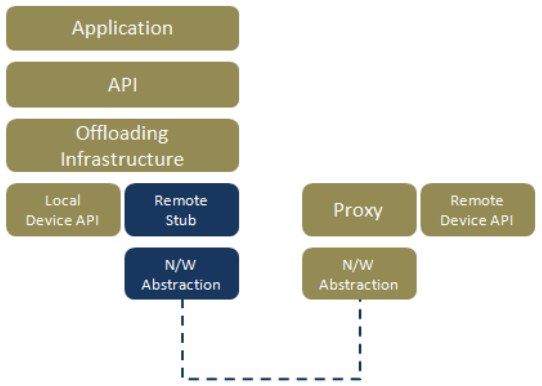

Figure 1. Block diagram of offload framework

The proposed architecture is shown in Figure 1 and has the following components:

- *Application*: An application program executes on a mobile device. It offloads various tasks to available hardware and software resources for optimal computation. In this paper, we present the requirements for the application design to achieve efficient offload in the presence of networked resources.

- *API*: The API layer is local to the mobile device and allows the application to utilize the platform resources in a seamless and portable manner. The design complexity of the API depends on the application design and is detailed in Section 2.

- *Offload Infrastructure*: This layer is responsible for managing the aggregated platform resources. The functions include keeping track of dynamic resources, optimal offload management accounting for granularity, and providing the backend services to the API.

- *Local Device API*: Interface to the device drivers for local resources at the mobile device.

- *Local Network Abstraction*: This is comprised of the infrastructure necessary to abstract a remote device locally. The layer is comprised of two sub-layers: A device stub for virtualizing a remote device or service and a network abstraction sub-layer to manage network diversity among technologies such as Wi-Fi and cellular.

- *Remote Network Abstraction*: The remote networking stack implementing the device proxy and networking peer for the mobile device.

- *Remote Device API*: Interface for job/function execution on the remote device.

The following sections present design consideration for the components listed above staring with the relevant work in this area. We then present an implementation of the architecture that we are building and a proof of concept prototype. The rest of the paper is structured as follows. In Section 2, we describe the architecture for our framework by presenting major sub-blocks. The prototype of our work and the result of experiments are presented in Section 3 and 4, respectively. We overview previous related works on computation offloading framework in Section 5, and lastly, Section 6 concludes the paper.

2. ARCHITECTURE

The main goal is to design an offloading framework for mobile devices by layering offload APIs and an offload infrastructure upon a virtual networking substrate that supports TCP/IP networking and widely-used resource discovery protocols. In this section, we present major sub-blocks of the framework such as application, offload API and infrastructure, and network layer prototype.

2.1 Application

Mobile applications involve the use of system resources to a very large degree. Most of the context-based applications use sensors from a mobile platform. These sensors include GPS, accelerometer, camera, and microphone. Apart from the local sensors, the applications use a variety of media processing engines for data processing and formatting for human consumption. Furthermore, applications may require access and processing of information hosted externally to the device. While the sensor data acquisition operations need to be local to a mobile platform, most of the other compute intensive jobs can be offloaded over a network connection under application-specific latency constraints. An application programmer has the best knowledge of the routines that can and cannot be offloaded because of local context requirements. Providing this information to the offload infrastructure can help in fine-tuning the scheduling engine. This comes at the cost of application complexity. The application offloading assistance can thus be classified into four categories:

- *Minimum assistance*: The programmer provides two versions of the offloadable routine: one for local execution, and a remote optimized version. An API is provided to invoke a routine at runtime from the application. The offload manager picks the suitable version at the time of offload.

- *Maximum assistance*: This mode keeps the programmer free of any responsibility to annotate the application program for ease of programming. The offload framework can use the activity monitoring, runtime compiler and other tools to recognize opportunities and offload in runtime. This mode features a complex implementation of the offload manager.

- *Medium assistance*: The programmer only annotates the code with flags or compiler pragmas denoting the parts of the code that can be optimally offloaded. The offload framework uses these flags to schedule on local and remote resources.

- *API assisted*: This mode falls under the category of medium assistance where an application programmer uses the API services to discover remote resource capable of offloading.

2.2 API

Application Programming Interfaces (APIs) are designed to enable program portability and decouple the application development from hardware specifications.

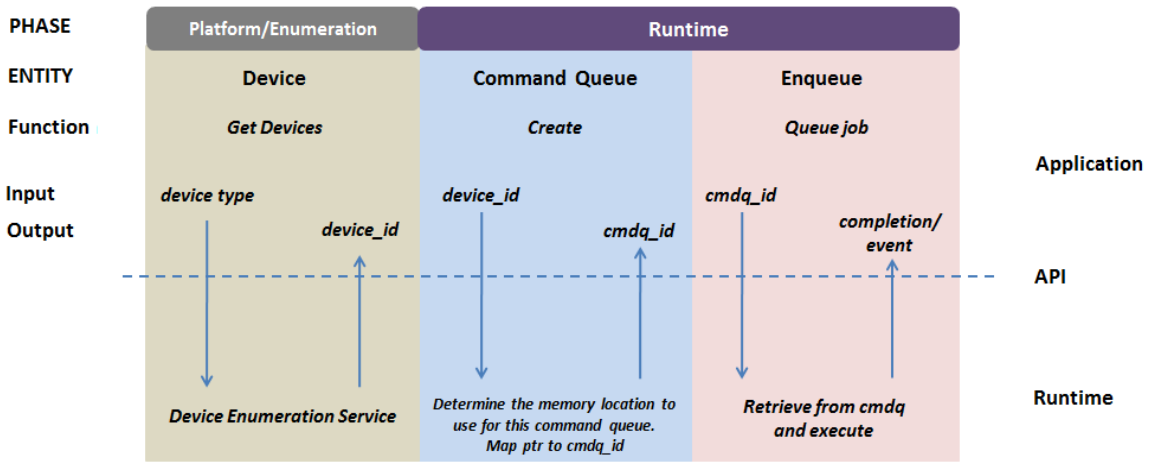

Figure 2. API invocation for platform and runtime services

A typical API layer that enables aggregation of hardware resources (both local and remote) needs to provide a set of fundamental service hooks for the application. The exact nature of the API calls depends on the programming model chosen from the set of options discussed in previous subsection. From the architecture perspective, the application API should support the following services:

2.2.1: Platform services comprise of APIs for discovering and enumerating the platform hardware. Following platform APIs are needed:

- *Device/Function Discovery*: This API allows an application to query and discover devices of different types. The device types queried can include the functionality of the device (graphics, decoder, crypto etc.) or the location of the device (remote/local). Querying for remote devices can be supported depending on the offloading framework. For our implementation described in Section 3, the use of local and remote devices is managed by the runtime and therefore the device location does not need to be exposed to the application programmer. Runtime provides a device handle for each device that matches the query for the application to use.

- *Device/Function Selection*: API needs to allow for an application to choose a device using available device handles.

2.2.2: Runtime services comprise of APIs for enabling the actual job offload on the local and remote devices. These APIs include:

- *Job customization*: To enable application portability, this API provides the translation of user requirements into actual hardware configuration for a job offload. The runtime functions supporting this include compiler-based tools for generating hardware-specific binaries, software libraries for hardware set-up, and network encapsulation for transporting the job over the network for remote execution.

- *Buffer Management*: This API is needed to allow the application to marshal data for offload and provide for returned data storage. This includes methods for allocating new buffers and copying data between the user application and these buffers. Keeping the offload buffers separate from the memory actively used by the application enables decoupled offload and efficient program implementations with reduced synchronization requirements.

- *Offload Queue Management*: This API is needed to provide methods for creating offload queues targeted at enumerated devices by the application. A created queue is identified by a handle and each offload job gets queued in these queues. Further, the application needs APIs to specify the properties of these queues, including synchronization requirements and the nature of the queue (e.g. FIFO vs. out-of-order execution). Once the queues are created, an application should be able to query the status of a queue using a handle provided by the runtime through the API.

- *Job Offload*: Using the information from the APIs above, the application can now “package” a job targeted for a specific device or a device type (with runtime deciding the exact device to be used based on scheduling metrics). The job offload API allows for queuing jobs in the job queues targeting the hardware devices along with any input/output buffers. Synchronization details between different jobs and completion notifications to the application can also be specified at this time.

- *Status Queries*: To enable an application to determine the status of an offloaded task, the API needs to provide for status queries. These queries can be used by the application to either poll for a job completion or determine if there were any errors executing the offloaded job. The exact error codes implemented depend on the offload methodology chosen. Figure 2 shows the API invocation at different offload stages.

2.3 Network layer – Communication and Resource discovery

Related works such as MAUI, Cuckoo, and Mobile MapReduce [4-6] have focused on code partitioning to deal with the power management of mobile devices, but assume static resource discovery. Our system enables an automatic and seamless service discovery in order to give end-users flexibility during service selection. Our approach is based on *network virtualization*, presenting the IP layer to clients and services, coupled with a *self-organizing overlay topology* that abstracts away the set up and management of network connections from the upper layers. With this approach, a wealth of applications and middleware written for TCP/IP can be readily reused and devices can communicate using a unified interface (e.g. the Berkeley socket API) whether on a LAN, personal-area network, or WAN. For network virtualization, we propose the use of SocialVPN [3] which is tunneling techniques through Virtual Private Network (VPN). SocialVPN provides an easy-to-use, scalable, yet secure virtual IP network.

With support from virtualization, the network layer adopts universal plug-and-play (UPnP) which allows devices to seamlessly discover each other using a set of IP protocols such as Internet Group Management Protocol (IGMP) and Simple Service Discovery Protocol (SSDP). A challenge with multicast resource discovery is that the vast majority of ISPs prevent multicast traffic from traversing their networks. We sidestep this issue with virtualization – each UPnP discovery message is tunneled and encapsulated in a unicast IP packet and overlay-routed, possibly over WAN.

3. PROTOTYPE

3.1 SocialVPN

While many VPN tunneling techniques would be applicable (e.g. OpenVPN [1], Hamachi [2]), the approach chosen in our prototype is SocialVPN, due to its ability to autonomously create and manage VPN links to social peers, and its support for tunneling IP multicast and discovery via UPnP. SocialVPN is a decentralized, self-configuring VPN based on structured peer-to-peer overlays and a public-key based security model where certificates are exchanged and used to set up VPN links automatically using online social network APIs. Furthermore, it supports users behind network address translators (NATs) to have bidirectional virtual IP connectivity to other SocialVPN users through decentralized NAT "hole punching". SocialVPN maintains a private IP address space, and assigns it to SocialVPN users to which are intended to connect dynamically and locally such that it is able to handle the constraints of limited IPv4 address space.

3.2 Computation offload

While various options are possible to implement application-level data transfer between mobile devices and remote hosts, we use a REST-like RPC system. This REST-like RPC system encapsulates the method name and parameters to be processed on the remote host into an HTTP header and sends the binary data as HTTP body incurring a negligible overhead for marshalling data to be transferred compared with other RPC systems such as XMLRPC. Then, the remote host executes the method which is included in HTTP header it receives.

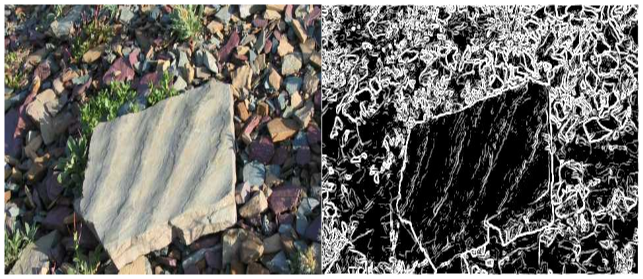

Figure 3. Image used for the offload experiments and image after *Sobelfilter*

We implemented offload client/server primitives written in Python and targeted a simple image processing workload: *Sobelfilter*. *Sobelfilter* is used for image edge detection and implemented in OpenCL for CUDA GPU architecture and C++ for a regular CPU host [9]. Figure 3 shows the image used for the offload experiments before and after applying *sobelfilter*.

4. EXPERIMENT

In this section, we verify the feasibility of UPnP on top of the SocialVPN from the perspective of service discovery and the benefits of mobile offload in terms of processing time and energy consumption. We conducted a series of experiments over local and wide area networks. In order to measure the service discovery time, we used WireShark [11] to measure the latency of UPnP request and response messages. Also, we measured the energy consumption of the mobile device through PowerTutor [8] which is an application for the variants of Android devices that displays the power consumed by major system components such as CPU, network interface, display, and GPS receiver.

4.1 Service discovery time

We conducted a WAN experiment in which 100 UPnP servers bound to the SocialVPN overlay are deployed in virtual machines at FutureGrid [7] resources at U. Chicago, UC San Diego, and U. Texas, while an UPnP client located at U. Florida requests service discovery. In this experiment, we observed that the UPnP client is able to discover all of the UPnP servers over three sites and the service discovery times for 100 UPnP servers range from 27ms to 57ms.

Table 1. Service discovery time for location of UPnP servers on FutureGrid resources (client at U. Florida)

	U. Texas		U. Chicago		UC San Diego	
Service discovery time (ms)	min	max	min	max	min	max
	27.0	29.4	45.4	47.6	54.6	57.0

Table 1 shows service discovery times for each location of UPnP servers, demonstrating the ability of the SocialVPN to support unmodified UPnP applications and middleware across wide-area resources.

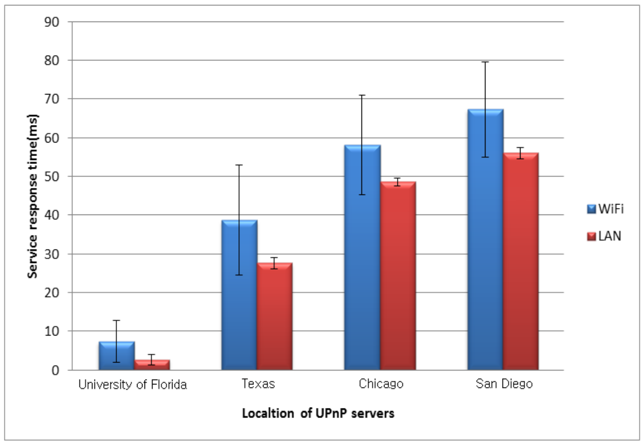

Figure 4. Service discovery time for Wi-Fi vs. wired LAN (client is at U. Florida)

An additional experiment has been conducted to observe the variation in service discovery times when the client is connected by Wi-Fi or wired LAN. The service discovery request is repeated 100 times, and the distribution of times is summarized in Figure 4. The service discovery time is proportional to the underlying physical network latency to services, and the Wi-Fi setup has longer service discovery time than wired LAN.

4.2 Offloading: performance and energy consumption

In this experiment, we consider the potential benefits of offloading either to a cloud instance or to a personal computer connected to a user's SocialVPN (e.g. their own, or a friend's desktop). For the client, we used an Android tabletPC, Samsung GalaxyTab 10.1, running Android honeycomb. For the SocialVPN-connected PC, we utilized a desktop with 3.0 GHz Intel Core2 Duo CPU and 4 GB of memory, and for the cloud offload instance, we used an Amazon Elastic Compute Cloud (EC2) resource with two NVIDIA GPUs equipped with 3 GB on-board memory per GPU. Firstly, we executed *sobelfilter* on the tabletPC locally and measured the processing time and energy consumption using PowerTutor. Secondly, the Android tabletPC offloads the image processing task to the PC and cloud resources. In the experiments, we varied the size of JPEG images: 26KB (480×270), 674KB (1920×1080), and 1,743KB (1936×2592) to assess the impact of having different workload sizes.

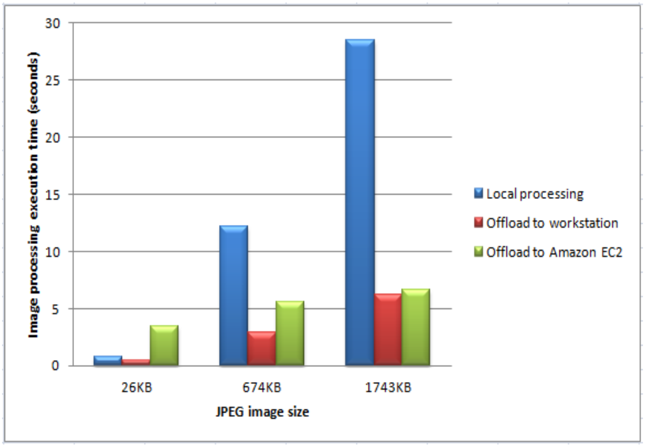

Figure 5. Image processing execution time for client processing and offloading to PC workstation and cloud

Figure 5 shows the image processing execution time for local processing and offloading to the local workstation and Amazon EC2 GPU node. In the 674KB and 1,743KB sizes, offloading to the local workstation and Amazon EC2 GPU is faster than local processing. For the smallest image size, however, offloading to Amazon EC2 GPU took longer time than local processing. The reason is that offloading needs an additional time to setup the GPU and to move the image data to GPU memory. Consequentially, the larger image size to be processed, the bigger improvement obtained in terms of the execution time. As shown in Figure 6, for the smallest size image, local processing is better than offloading for energy consumption of the mobile device because image processing time is fairly small and no image data transfer to remote host is required. However, as image size increases, client processing results in much more energy consumption, and eventually both offloading to the local workstation and Amazon EC2 consume less energy than local processing. It is worth noting that offloading to EC2 GPU cluster consumes more energy than offloading to the local workstation in all the image sizes. This is caused by the power consumption characteristics of Wi-Fi networking card and the increased latency. Wi-Fi networking card stays in high power state only when it has a certain number or more packets to be sent or received in a second. In the cloud setup, it is common that packets sent or received are spread over longer periods compared to the LAN workstation because each packet exchange is required to be *ACK*ed. Thus the Wi-Fi card stays in high power state for longer time and consumes more energy.

5. RELATED WORK

Offloading computation from mobile devices to remote hosts has been considered in the literature to handle challenges such as managing the energy of mobile devices efficiently and improving the performance of offloaded tasks qualitatively and quantitatively. MAUI and Cuckoo [4, 5] have presented the runtime frameworks for computation offloading to minimize user's response time while maximizing the life span of mobile devices. Mobile MapReduce [6] has implemented the mobile version of the MapReduce framework by employing several remote hosts to parallelize mobile applications. Also, CrowdSearch [10] adopted the combination of local processing on mobile devices and remote processing on powerful servers to increase the accuracy of image search engine.

While these works target a static remote host to offload computation, our framework is novel in the way it integrates with virtual networking to seamlessly discover remote hosts using universal plug-and-play (UPnP) on top of the SocialVPN, such that mobile devices are able to connect to remote hosts over wide area networks as well as local area network and supporting existing, well-established resource discovery protocols and a wealth of TCP/IP applications.

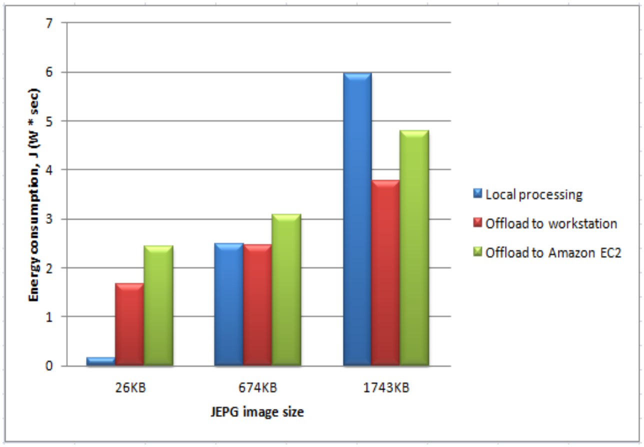

Figure 6. Energy consumption of Android tabletPC for local processing and offloading

6. CONCLUSIONS AND FUTURE WORK

The main contribution of this paper is to demonstrate the feasibility of a virtual networking-based framework for opportunistic function offloading. We demonstrated the efficacy of offloading compute-intensive functions of different granularities to remote systems over different network latencies to establish the benefits of such a scheme. Clear benefits in terms of both performance and power were achieved in certain offloading scenarios. In order to achieve these benefits for a variety of applications and connectivity scenarios, the framework needs to be capable of dynamically discovering the remote capabilities and intelligently offload to cloud opportunistically. SNARF provides a prototype implementation of such a framework which was built on top of SocialVPN. It also provides a consistent interface for local and remote function offloading absorbing the complexities of discovery, configuration and dynamic offloading away from the programmer. We demonstrated the power and performance benefits of an image processing application on an Android tabletPC using SNARF. The benefits were quite compelling at large image sizes – which clearly offsets the remoting overheads.

We continue to characterize the benefits of SNARF using different workloads related to Tablets and Smartphone use cases. Detailed characterization of the remoting overhead and investigation of potential architectural enhancements to reducing this overhead are parts of our future work. Also, we are currently working on the implementation of SocialVPN tailored to low-power operation in mobile platforms. We will characterize the cost of such a secure connection for mobile devices in terms of energy consumption, bandwidth, and latency in the future work.

7. REFERENCES

[1] Openvpn – the open source vpn. 2009. http://openvpn.net/

[2] Hamachi – instant, zero configuration vpn. May, 2009. https://secure.logmein.com/products/hamachi/

[3] P. St. Juste, D. Wolinsky, P. Oscar Boykin, M. J. Covington, and R. J. Figueiredo, SocialVPN: Enabling wide-area collaboration with integrated social and overlay networks. Computer Networks. January, 2010.

[4] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: Making Smartphones Last Longer with Code Offload. In Proceedings of the Eighth International Conference on Mobile Systems, Applications, and Services. San Francisco, CA, USA. June 15-18, 2010.

[5] R. Kemp, N. Palmer, T. Kielmann, and H. Bal. Cuckoo: a Computation Offloading Framework for Smartphones. In Proceedings of the Second International Conference on Mobile Computing, Applications, and Services. Santa Clara, CA, USA. October 25-28, 2010.

[6] M. A. Hassan and Songqing Chen. Mobile MapReduce: Minimizing Response Time of Computing Intensive Mobile Applications. In Proceedings of the Third International Conference on Mobile Computing, Applications, and Services. Los Angeles, CA, USA. October 24-27, 2011.

[7] Gregor von Laszewski, Geoffrey C. Fox, Fugang Wang, Younge, Andrew J, Archit Kulshrestha, Gregory G. Pike, Warren Smith, Jens Voeckler, Renato J. Figueiredo, Jose Fortes, Kate Keahey and Ewa Delman. Design of the FutureGrid Experiment Management Framework. GCE2010 at SC10. New Orleans, LA, USA. November 14, 2010.

[8] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P. Dick, Zhuoqing Morley Mao, and Lei Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In Proceedings of the Eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis. Scottsdale, AZ, USA. October 24-29, 2010.

[9] OpenCL SDK Code Samples http://developer.nvidia.com/opencl-sdk-code-samples#oclSobelFilter

[10] T. Yan, V. Kumar, and D. Ganesan. CrowdSearch: Exploiting Crowds for Accurate Real-time Image Search on Mobile Phones. In Proceedings of the Eighth International Conference on Mobile Systems, Applications, and Services. San Francisco, CA, USA. June 15-18, 2010.

[11] Wireshark: A Network Protocol Analyzer, http://www.wireshark.org/