MARS: Adaptive Remote Execution for Multi-threaded Mobile Devices

Asaf Cidon*
cidon@stanford.edu

Tomer M. London*
londont@stanford.edu

Sachin Katti
skatti@stanford.edu

Christos Kozyrakis
christos@ee.stanford.edu

Mendel Rosenblum
mendel@cs.stanford.edu

Stanford University

ABSTRACT

Mobile devices face a growing demand to support computationally intensive applications like 3D graphics and computer vision. However, these devices are inherently limited by processor power density and device battery life. Dynamic remote execution addresses this problem, by enabling mobile devices to opportunistically offload computations to a remote server. We envision remote execution as a new type of cloud-based heterogeneous computing resource, or a "Cloud-on-Chip", which would be managed as a system resource as if it were a local CPU, with a highly variable wireless interconnect. To realize this vision, we introduce MARS, the first adaptive, online and lightweight RPC-based remote execution scheduler supporting multi-threaded and multi-core systems. MARS uses a novel efficient offloading decision algorithm that takes into account the inherent trade-offs between communication and computation delays and power consumption. Due to its lightweight design, MARS runs on the device itself, instantly adapts its decisions to changing wireless resources, and supports any number of threads and cores. We evaluated MARS using a trace-based simulator driven by real world measurements on augmented reality, face recognition and video game applications. MARS achieves an average speedup of 57% and 33% higher energy savings over the best static client-server partitions.

1. INTRODUCTION

There is a growing demand for high performance mobile devices, capable of supporting computationally intensive applications that require high-definition 3D graphics and computer vision capabilities. Even though mobile processors are becoming more capable due to System-on-Chip architectures and hardware accelerators, local processing capabilities are still inherently limited by processor power density and device battery life. Meanwhile, high bandwidth wireless networks have become ubiquitous and are being used to connect mobile devices to the cloud. Potentially, by utilizing such connectivity to offload computation to the cloud, we could greatly amplify mobile application performance at minimal cost.

However, it is difficult to develop client-server software for mobile devices. Traditionally, client-server programming requires the programmer to decide which part of the application is executed on the client and server. In order to partition the application adequately, the programmer needs to make assumptions on the available network resources. However, wireless network latency and bandwidth are notoriously variable over time, which makes static program partitioning ineffective, leading to underperforming and energy-inefficient applications.

Dynamic remote execution is a technique that addresses the problem of static client-server partitioning. Dynamic remote execution allows devices to opportunistically offload computations to a remote server, according to the network and CPU resources available on the device at any given time, in order to improve performance and lower the energy consumption of mobile applications. Since wireless network and CPU availability are highly variable, remote execution systems need to adapt their decisions instantaneously. However to adapt quickly and accurately, the remote execution scheduler, which is in charge of making the offloading decisions, needs to be lightweight to reside on the mobile device. A lightweight scheduler design also enables the remote execution system to scale to multiple threads and cores. Therefore, in order to support multi-threaded execution in a rapidly changing wireless environment, dynamic remote execution systems should be *adaptive, online and lightweight*.

Previous remote execution systems took the approach of either offloading individual RPCs or migrating VMs to the cloud. Since wireless network bandwidth and latency change rapidly over time, and because VM migration requires a high level of synchronization between client and server, we pursue RPC offloading, which is a lower overhead mechanism that allows the system to immediately respond to changing resources.

Several existing remote execution systems focused on RPC offloading of single threads. However, these systems do not meet all three requirements for supporting multi-threaded execution in rapidly changing wireless conditions. In order to make optimal scheduling decisions, the systems utilize integer linear programming [6, 13, 18] or other mathematical solvers [8, 19]. Since these schemes are computationally intensive, the scheduler usually runs on a remote server rather than on the client device. Running the scheduler remotely

*Equal contribution, order chosen alphabetically

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MobiHeld '11, October 23, 2011, Cascais, Portugal.
Copyright © 2011 ACM 978-1-4503-0980-6/11/10 ... $10.00.

Paper Contributions
1. Designed MARS, an adaptive, online and lightweight remote execution scheduler for offloading RPCs 2. By making RPC-level decisions, MARS can be executed on the mobile device itself, which allows it to instantly adapt to variable network and CPU 3. MARS supports multiple threads and cores 4. Trace-based simulations show MARS provides speedup of 57% and 33% energy savings over best static client-server partitions

Table 1: Paper Contributions

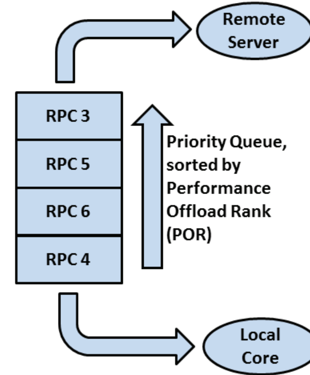

Figure 1: MARS Priority Queue

degrades the level of adaptability to the network. Supporting multi-threaded remote execution of multiple applications would only increase the complexity of such schedulers.

In this work, we introduce MARS (Multi-threaded Adaptive Remote execution Scheduler), an adaptive, online and lightweight remote execution scheduler that supports multi-threaded and multi-core systems. Unlike previous RPC-based schedulers, which partition the entire call graph of an application each time a scheduling decision is made, MARS makes lightweight scheduling decisions in the granularity of a single RPC: it decides whether to run each individual RPC locally or on a remote server in order to optimize overall performance and energy consumption. MARS uses a lightweight yet efficient offloading decision algorithm that takes into account the inherent trade-offs between communication and computation delays and power consumption. This allows MARS to run on the device itself, regardless of the number of concurrent threads. Running on the mobile device allows MARS to immediately adapt its decisions to changing wireless bandwidth and latency as well as CPU availability. MARS easily supports multi-core devices with multiple threads, without interfering with the local OS scheduler. However, this technique comes with a price; since MARS makes scheduling decisions on an RPC level, it does not look ahead in the call graph to take into account whether there are any subsequent RPCs waiting for the remote execution RPC to complete. Nonetheless, our initial evaluation of MARS using a trace-based simulator, based on performance, power and wireless network measurements, indicates that MARS provides an average speedup of 57% and 33% energy savings over the best static client-server partitions of face recognition, augmented reality and augmented reality video game applications. The paper's contributions are listed in Table 1.

Mobile devices increasingly rely on System-on-Chip architectures with heterogeneous cores. We envision remote execution as a new type of cloud-based heterogeneous computing resource, or a "Cloud-on-Chip", which would be managed as a system resource as if it were a local CPU, with a highly variable wireless interconnect. This paper is our first step in fulfilling this vision. Our goal in this work is to evaluate whether an online lightweight scheduler like MARS can significantly improve performance and energy consumption on multi-threaded systems. We believe our trace-based simulator results answer this question affirmatively. Therefore, our next step is to implement MARS as a working remote execution system on a mobile device.

The rest of this paper is organized as follows: Section 2 presents the design of MARS, while Section 4 contains our simulation methodology. Section 4 provides the results of our trace-based simulator. In section 5 we survey related work and Section 6 presents our conclusions.

2. MARS DESIGN AND ALGORITHM

MARS is an adaptive remote execution scheduler designed to optimize execution time and energy consumption of RPCs that are candidates for remote execution. MARS uses a priority queue that contains all the candidate RPCs, as depicted by Figure 1. The priority queue is sorted according to a performance metric, called POR, which estimates the speedup gained from offloading the RPC. Whenever a remote computing resource is available, MARS pops the RPC with the highest POR from the queue and offloads it. When a local core is available, MARS pops an RPC with the lowest POR and directs it to local execution. In order to take energy considerations into account, MARS uses an energy metric, called EOR. EOR estimates the energy that would be saved on the device by offloading the RPC over executing it locally. If an RPC's EOR is below a certain threshold, it will not be remotely executed, in order to prevent the device from incurring a high energy cost. Every time a new RPC needs to be scheduled, MARS adapts to changing network and CPU resources, by updating its POR and EOR metrics and resorting the queue. Since MARS has to schedule multiple RPCs in parallel under rapidly changing resources, it is designed as a lightweight heuristics-based scheduler similar to modern OS schedulers.

2.1 RPC Model

We denote a single unit of remote execution as a remote execution RPC. We assume that remote execution RPCs are segments of a program that were defined by the programmer to serve as candidates for remote execution, and are functionally idempotent and independent RPCs. Like traditional asynchronous thread-safe RPCs, remote execution RPCs do not use any shared memory and synchronization mechanisms, and have no inter-thread dependencies. Each remote execution RPC is launched by our system as a separate thread. Previous systems [6] have implemented similar programmer-defined RPC based remote execution program-

ming models. In addition, our system relies on the developer to define remote execution RPCs that require a significant and finite amount of time to execute locally (hundreds of milliseconds or more).

2.2 Metrics

The MARS algorithm optimizes the execution time of remote execution RPCs. MARS sorts RPCs in a priority queue according to their performance offload rank (POR), which is the RPC's estimated remote execution time over local execution time:

$$POR = \frac{LocalExecutionTime}{RemoteExecutionTime + NetDelay}$$

$$NetDelay = \frac{InputSize}{UploadBW} + \frac{OutputSize}{DownloadBW} + RTT$$

If the RPC's $POR > 1$, it is more beneficial to offload it in terms of performance. RPCs with very high POR usually involve small data transfers, but incur high computation cost. The priority queue RPCs are ordered according to their most updated value of POR, where the RPC with the highest POR is at the top of the queue, as depicted in Figure 1. Note that POR takes into account several dynamic factors: the network bandwidth and latency and the local execution time on the mobile device. It does not take into account whether there are any subsequent functions waiting for the RPC to complete. Future extensions of MARS could modify POR so that it takes into account dependent functions by 'looking ahead' in the function call-graph.

In certain scenarios (e.g. low-bandwidth 3G), using the network can incur a high energy cost. Therefore, MARS should remotely execute RPCs without compromising the device's energy consumption. Before deciding whether to execute an RPC remotely or locally, MARS checks whether the RPC passes an energy-based threshold. This condition is calculated using the energy offload rank (EOR), which is the estimated mobile energy consumption of executing an RPC remotely over locally:

$$EOR = \frac{LocalExecTime \cdot DevicePower}{NetEnergy}$$

$$NetEnergy = \left(\frac{InputSize)}{UploadBW} + RTT\right) \cdot UploadPower + \left(\frac{OutputSize}{DownloadBW} + RTT\right) \cdot DownloadPower$$

In order to adapt to changing network conditions, MARS periodically profiles the network and CPU. Before a scheduling decision, MARS checks the new status of the network and CPU, updates the POR and EOR of all the RPCs, and resorts the queue according to the new POR values. When a new remote execution RPC enters the queue, its position in the queue is determined according to its POR.

2.3 Scheduler Algorithm

The MARS algorithm has two operational principles. First, given a set of remote execution RPCs, it attempts to keep both the network and the CPU fully utilized at all times, as long as using a resource will significantly improve the energy efficiency of the mobile device. Second, remote execution RPCs with high POR are executed remotely, and RPCs with low POR are executed locally.

Every time a resource in the system (CPU or network) becomes available, MARS calls the scheduler algorithm shown in Algorithm 1. $HeadOfQueue$ and $BottomOfQueue$ refer to the RPC at the head and bottom of the queue, respectively. $saveRPCState()$ saves the RPC's data, so that if the network becomes disconnected, MARS can resume the execution of the RPC locally. In our simulations, we assume $saveRPCState()$ is not on the critical path. $stallRPC()$ is called when the RPC does not pass the EOR thresholds. It then waits in the queue until the next resource is available.

Algorithm 1 MARS Algorithm

```
1: if RemoteServer.isAvailable() AND
2:          CPU.isBusy() AND
3:          EOR(HeadOfQueue) ≥ 1/G then
4:     saveRPCState(HeadOfQueue)
5:     offload(HeadOfQueue)
6: else if CPU.isAvailable() AND
7:          RemoteServer.isBusy() AND
8:          EOR(BottomOfQueue) < G then
9:     runLocally(BottomOfQueue)
10: else if CPU.isAvailable() AND
11:          RemoteServer.isAvailable() then
12:     if POR(HeadOfQueue) > 1 AND
13:          EOR(HeadOfQueue) ≥ 1/G then
14:         saveRPCState(HeadOfQueue)
15:         offload(HeadOfQueue)
16:     else
17:         runLocally(HeadOfQueue)
18:     end if
19: else
20:     stallRPC()
21: end if
```

Notice the threshold checks of G in lines 2, 8 and 11 in the scheduler algorithm. $G \geq 1$ is a configurable parameter, called $Greediness$. The higher G is set, the more 'greedy' the scheduler algorithm becomes, meaning it is more likely to utilize its network and CPU in parallel. When $G \rightarrow \infty$ the scheduler algorithm always tries to utilize all of its available CPU and network resources to optimize performance, regardless of the energy cost. By setting different values of $Greediness$, the algorithm can balance between optimizing performance and energy consumption. In Section 4 we will address the selection of G in our simulations.

The remote execution system should not interfere with the OS scheduler. Therefore, we designed MARS as a user-level process that is scheduled by the OS scheduler itself. MARS manages the remote execution RPCs by launching them one at a time as separate threads. MARS enforces its decisions by locking and unlocking mutexes that the RPC threads lock when they call the remote execution library function. Once they are unlocked by MARS, the RPCs run alongside other processes scheduled by the device OS as normal user-level RPCs. We assume in our simulations that the context switches and synchronization overhead caused by MARS do not incur a high performance cost. We plan to measure this overhead in our implementation.

Our scheduler can be easily applied to multi-core processors. Each time one of the cores becomes available, the scheduler pops a remote execution RPC from the bottom of the queue (after checking whether it passes the EOR threshold). We do not take into account multi-core features such as shared caches and thread migration among cores.

2.3.1 System Profiling Assumptions

Previous single-threaded remote execution systems [15, 6, 12] have demonstrated the use of history-based profilers to

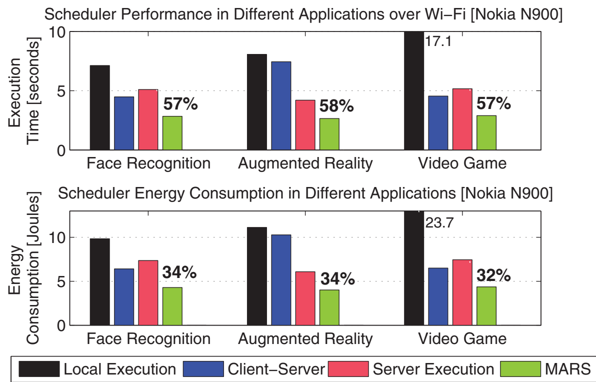

Figure 2: MARS performance on 3 different applications, averaged over 40 outdoors Wi-Fi traces

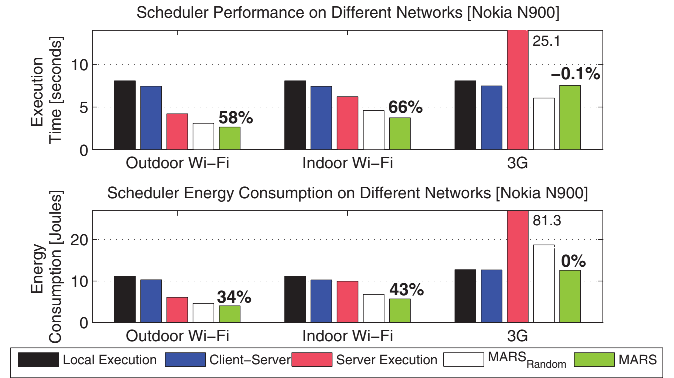

Figure 3: MARS performance on augmented reality application under Wi-Fi and 3G traces

make predictions of RPC execution time and power consumption. These mechanisms have been shown to be fairly accurate; their error rates average between 5-20% for graphics and augmented reality applications. CPU resources can be estimated using standard system profilers, while network bandwidth can be estimated by measuring the arrival time differences of a packet pair. Network latencies are estimated using end-to-end network pings. Mobile power consumption can be profiled using I2C Tools or CPU power counters (or by more accurate power estimation methods [2]). By combining history-based and system resource profilers, MARS estimates the execution times and energy consumption of each remote execution RPC. We assume that even if it uses optimal profilers, MARS would always have to deal with uncertainty regarding the network latency and bandwidth. MARS' adaptive, simple and lightweight design is particularly valuable in situations where device and network resources are highly unpredictble.

2.3.2 Fairness and Starvation

Similar to OS scheduling, our scheduler needs to deal with fairness and starvation. Remote execution RPCs may become 'stuck' in the middle of the queue, while functions with higher and lower POR continuously enter the queue and block them. In order to address this issue, in our implementation we will bin our functions in zones of different POR levels (e.g. a zone may be $1 < POR < 1.5$), and within each zone pop functions according to a FIFO order, and use a weighted round-robin to pop from each bins, similar to a multilevel feedback queue.

3. SIMULATION METHODOLOGY

Our experiments measure three computationally intensive applications: augmented reality (AR) pattern recognition, face recognition and an interactive augmented reality video game. The AR pattern recognition application [16] uses a 3D engine engine to render 3D objects on top of 2D markers. The AR application is divided into two types of RPCs: the first identifies a marker in the picture, and the second renders a 3D object on top of the marker. The face recognition application [7] recognizes the faces of several people at once from a given picture. It is also divided into two types of RPCs: the first detects a face in an image and the second recognizes the identity of the face. Our third application simulates an interactive augmented reality game, which involves face recognition and rendering of 3D objects. The game is divided into a random number of RPCs in parallel (between 1 and 5).

The simulator generates a different number of RPCs in parallel for each application: 10 pairs of RPCs (10 face recognition and 10 face detection) for the augmented reality application, 6 pairs for the face recognition application, as well as 5 different sets (with 1-5 RPCs each) for the interactive augmented reality game. The number of RPCs was chosen to present realistic traces.

In order to evaluate MARS, we ran each of these applications on two different mobile devices and a data center server. We used a Nokia N900 smartphone with an ARM Cortex A8 running at 600MHz, and an NVIDIA Tegra 250 board with a dual ARM Cortex A9 running at 1 GHz. We used an I2C power tool to measure energy consumption, for different CPU and network conditions, as depicted in Figure 2. For the server we used a high-end Amazon EC2 instance running the equivalent of an 8-core 1.0-1.2 GHz Opteron 2007. Our network bandwidth and latency traces consist of 40 Wi-Fi and 3G indoors and outdoors measurements, taken on campus. In order to measure bandwidth we sent 1 KB packet pairs every 10 milliseconds from the Nokia N900 to a remote server over UDP, and latency was measured by sending 32 byte pings between the device and server.

4. SIMULATION RESULTS

In our simulations we compare the performance of MARS with the performance of *static policies*. Static policies include all possible statically partitioned client-server setups. The developer can run all of the code locally (*Local Execution*), remotely (*Server Execution*), or statically partition it between the device and server ($Client - Server$).

Static policies are limited, because every thread has to maintain the same client-server partition. In contrast, due to the fact that MARS uses RPC level scheduling, it may partition each thread differently, in order to minimize execution time and energy consumption per RPC. This allows MARS to better utilize all of the system resources, and to significantly outperform the static code partitions over dynamic Wi-Fi network traces, as presented in Figure 2. Figure 3 also shows that MARS is beneficial for both indoors and outdoors Wi-Fi connections.

Table 2 shows the Wi-Fi and 3G power consumption on

Metric	Wi-Fi	3G
Idle network power	1.31 Watts	0.66 Watts
Upload network power	1.464 Watts	2.36 Watts
Download network power	1.39 Watts	2.26 Watts
Upload network power overhead	10.51%	72.03%

Table 2: Comparison between average Wi-Fi and 3G power consumption of Nokia N900 while wireless interface is on

the Nokia N900. For Wi-Fi connections, the network is a cheap resource, adding about 10% to the device's power consumption when it is in active mode. Hence, minimal energy consumption is achieved by reducing execution time, which means MARS should keep all of its resources fully utilized (CPU and server). In contrast, in 3G, using the network increases the power consumption by over 70%, so keeping all resources utilized will not always be the best strategy. Therefore, MARS should only remotely execute RPCs when they pass the threshold for energy savings.

Figure 3 demonstrates that MARS indeed does not try to fully utilize the remote server in a low bandwidth 3G scenario, and performs almost identically to the *Local Execution* policy. In order to evaluate the importance of POR and EOR, we modified our scheduler to give RPCs random values of POR and ignore the G threshold. The modified scheduler is called *MARS Random*. Figure 3 shows that *MARS* performs almost identically to *MARS Random* for Wi-Fi, but for 3G, *MARS* saves 49% more energy, which demonstrates that using a POR sorted queue and EOR thresholds is especially significant for scenarios where using a network incurs a high energy cost.

Choosing an effective *Greediness* value for MARS presents a trade-off between performance and energy efficiency. When network bandwidth is low, setting G too high may cause MARS to offload RPCs that incur a high energy cost, while in high bandwidths, setting G too low may cause MARS to underutilize its network and CPU. We found empirically that setting $G = 5$ provides a balanced trade-off for typical Wi-Fi and 3G connections.

Figure 4 shows the performance improvements of MARS when the mobile device has the processing capabilities of the Tegra 250 dual-core, instead of the Nokia N900's ARM Cortex A8 used in previous traces. Despite the superior computational capabilities of the Tegra 250, MARS still achieves a 29% speedup over the best static client-server policy.

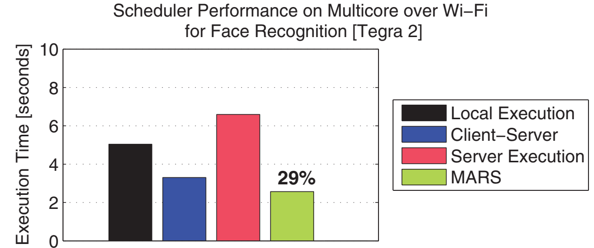

Figure 4: MARS performance on face recognition application using Tegra 250 multi-core

5. RELATED WORK

Resource-aware remote execution technologies have been implemented since the 80's, in systems such as Abacus [3], Butler [14], Coign [9] and Condor [11]. These systems apply automatic adaptive program partitioning and process migration in large computer clusters. Naturally, these systems do not address the large variance in resource variability of mobile application environments.

Previous remote execution systems for mobile devices have taken one of two approaches. The first approach, employed by systems like MAUI [6] and Chroma [4], conducts remote execution in the granularity of RPCs and their inputs and outputs. The second approach, utilized by systems such as CloneCloud [5, 10] and Cloudlets [17], is to conduct remote execution by migrating a VM from the mobile device to a remote server, or emulating the entire mobile operating system on the server, in order to preserve the software environment of executed functions. VM migration systems have to keep the mobile device environment in sync in real-time with the server, which requires offline static analysis, frequent online profiling and tight coordination and checkpointing between the client and server. In contrast, offloading individual RPC inputs and outputs requires minimal overhead, because RPCs are stateless and self-contained, and do not require any synchronization between client and server. Furthermore, since mobile network bandwidth and latency are highly variable, working in the granularity of single RPCs rather than VMs allows remote execution systems to make decisions in a shorter time-scale, and therefore increases adaptability to network dynamics. For these reasons, we chose RPC-level offloading rather than VM migration for MARS.

Several previous adaptive remote execution implementations supported offloading RPC-like functions to a remote server. MAUI [6], HYDRA [18] and Wishbone [13] are systems that partition the execution of a single application among several machines, using an integer linear programming solver. Since the solver is computationally intensive, it may introduce a high computational overhead to the mobile device. In order to avoid this overhead, the MAUI [6] solver's logic resides in the data-center rather than on the mobile device, trading energy consumption for application adaptability. In contrast to these schedulers, MARS is lightweight and resides on the device itself, which allows it to closely monitor and respond to changing network and CPU resources, as well as support multiple multi-threaded applications concurrently. Chroma [4] uses a "tactics selection engine" to optimally choose a client-server partition from a relatively small number of programmer-defined "tactics". Unlike MARS, Chroma's scheduler would incur a high computational overhead if it had to calculate all the different partitioning combinations for a large number of threads.

Other remote execution systems use different scheduling techniques for remotely executing a single thread. OLIE [8] utilizes a fuzzy control model, while Zhang et al. [19] partition applications utilizing naïve Bayesian learning techniques. The authors themselves observe that such techniques cannot be applied to multiple different types of concurrent threads, since they would incur a heavy overhead.

6. CONCLUSIONS

This paper presents an evaluation of MARS, an adaptive, online and lightweight remote execution scheduler for multi-threaded and multi-core systems. Our trace-based simulator results show that on average, MARS outperforms the best static client-server partitions by 57% and saves 33% more energy. In addition, our results demonstrate that remote execution schedulers, like MARS, have to be adaptive to changing network power costs that occur in different network types and bandwidths. Our next step is to implement a full "Cloud-on-Chip" system on a mobile device that uses MARS as its scheduler.

7. ACKNOWLEDGEMENTS

We would like to thank Ronny Ko, Anthony J. Romano and Jacob Leverich, for helping us set up the mobile and data-center experiment infrastructure. We thank Daniel Sanchez, Christina Delimitrou and David Lo for their valuable comments in writing the paper. We are grateful to Juha Vesanen from SVS Innovations [1] for allowing us to use their 3D rendering engine. Asaf Cidon is supported by the Leonard J. Shustek Stanford Graduate Fellowship, and Tomer M. London is supported by the Robert Bosch Stanford Graduate Fellowship.

8. REFERENCES

[1] Svs innovations. http://www.svsi.fi/blog/.

[2] Agrawala, A. K., Corner, M. D., and Wetherall, D., Eds. *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys 2011), Bethesda, MD, USA, June 28 - July 01, 2011* (2011), ACM.

[3] Amiri, K., Petrou, D., Ganger, G. R., and Gibson, G. A. Dynamic function placement for data-intensive cluster computing. In *USENIX Annual Technical Conference, General Track* (2000), pp. 307–322.

[4] Balan, R. K., Satyanarayanan, M., Park, S., and Okoshi, T. Tactics-based remote execution for mobile computing. In *MobiSys* (2003).

[5] Chun, B.-G., Ihm, S., Maniatis, P., and Naik, M. Clonecloud: Boosting mobile device applications through cloud clone execution. *CoRR abs/1009.3088* (2010).

[6] Cuervo, E., Balasubramanian, A., ki Cho, D., Wolman, A., Saroiu, S., Chandra, R., and Bahl, P. Maui: making smartphones last longer with code offload. In *MobiSys* (2010), pp. 49–62.

[7] Emami, S. Introduction to face detection and face recognition. http://www.shervinemami.co.cc/faceRecognition.html, June 2010.

[8] Gu, X., Nahrstedt, K., Messer, A., Greenberg, I., and Milojicic, D. S. Adaptive offloading inference for delivering applications in pervasive computing environments. In *PerCom* (2003), pp. 107–114.

[9] Hunt, G. C., and Scott, M. L. The coign automatic distributed partitioning system. In *OSDI* (1999), pp. 187–200.

[10] Kirsch, C. M., and Heiser, G., Eds. *European Conference on Computer Systems, Proceedings of the Sixth European conference on Computer systems, EuroSys 2011, alzburg, Austria - April 10-13, 2011* (2011), ACM.

[11] Litzkow, M. J., Livny, M., and Mutka, M. W. Condor - a hunter of idle workstations. In *ICDCS* (1988), pp. 104–111.

[12] Narayanan, D., Flinn, J., and Satyanarayanan, M. Using history to improve mobile application adaptation. In *WMCSA* (2000), pp. 31–.

[13] Newton, R., Toledo, S., Girod, L., Balakrishnan, H., and Madden, S. Wishbone: Profile-based partitioning for sensornet applications. In *NSDI* (2009), pp. 395–408.

[14] Nichols, D. A. Using idle workstations in a shared computing environment. In *SOSP* (1987), pp. 5–12.

[15] Pathak, A., Hu, Y. C., Zhang, M., Bahl, P., and Wang, Y.-M. Enabling Automatic Offloading of Resource-Intensive Smartphone Applications. Tech. rep., 2011.

[16] Prabhuswamy, B. Opencv simple augmented reality program. http://dsynflo.blogspot.com/2010/06/simplar-augmented-reality-for-opencv.html, June 2010.

[17] Satyanarayanan, M., Bahl, P., Cáceres, R., and Davies, N. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing 8*, 4 (2009), 14–23.

[18] Weinsberg, Y., Dolev, D., Anker, T., Ben-Yehuda, M., and Wyckoff, P. Tapping into the fountain of cpus: on operating system support for programmable devices. In *ASPLOS* (2008), pp. 179–188.

[19] Zhang, X., Jeong, S., Kunjithapatham, A., and Gibbs, S. Towards an elastic application model for augmenting computing capabilities of mobile platforms. In *Third International ICST Conference on Mobile Wireless Middleware, Operating Systems, and Applications* (2010).