Offloading Raspberry Pi Applications

Karan Pugla
Stony Brook University
karan.pugla@stonybrook.edu

Arpita Sheth
Stony Brook University
arsheth@cs.stonybrook.edu

Supriya Deshpande
Stony Brook University
sudeshpande@cs.stonybrook.edu

Arnav Gupta
Stony Brook University
arnav.gupta@stonybrook.edu

ABSTRACT

A low powered device like Raspberry Pi provides great cost and energy savings, however it cannot perform computationally intensive tasks. A good way to combine cost and energy savings while not compromising on performance is to do offloading wherein the Pi performs tasks locally as much as it can support, post which it offloads to remote machines. A load balancer on the Pi would decide when to offload, how much to offload and the duration of the offloading. In this paper, we are proposing a dynamic offloading algorithm to dynamically offload CPU intensive as well as Memory intensive workloads. We also provide a sensitivity analysis to evaluate the impact of request rates and number of requests on overall Response Time.

Keywords

Raspberry Pi, Dynamic Offloading, Response Time, Virtual Machines, Load Balancer

1. INTRODUCTION

The Raspberry Pi is a small, low cost device that can be plugged into a computer or a TV. Even though Raspberry pi is slower than a present day desktop or laptop, it is capable of doing everything that a desktop computer can do like browsing the web, playing videos, playing games, word-processing etc.

The Raspberry Pi is an open hardware, except for the primary chip called the Broadcom System on a chip (SoC). At the software level, it has a Linux version of operating systems designed specifically for Raspberry Pi. The typical cost of a Raspberry Pi is $35, which is much cheaper than the existing Virtual Machines/desktops/laptops/smartphones. It is powered by a 5V micro USB, using 700-1000mA, much less than even current smartphones. Thus, it is a complete Linux computer, providing all the expected benefits, at a low cost and a low power-consumption level.

A caveat of using Raspberry Pi is that it cannot handle computationally intensive tasks considering it has a 512 MB RAM. Offloading solutions have been proposed for such intensive applications where the applications are offloaded to cloud. Most work concentrating on this area has been devoted to writing applications that can be offloaded selectively. Our work focuses on dynamic offloading rather than development-time selective offloading[7]. While mobile offloading is being studied extensively, very little work has been done with Raspberry Pi to develop offloading policies that can dynamically offload any sort of CPU intensive or Memory intensive applications. A naive approach while using such low end servers such as Raspberry Pi could be to always offload the applications to the cloud and not let Pi perform any computations. While this approach is seemingly reliable, it is not cost and energy efficient. To explore the tradeoff between Pi's computational capacity and cost benefits and VM's cost and energy savings, this paper presents a dynamic offloading technique which tries to find a common ground between Pi's cheap computation and VM's reliable service to enable shortest Response times and highest performance gain as well as energy savings. The dynamic offloading policy determines the offloading decision to be taken at each instance by inspecting the CPU and Memory usage statistics of Pi and both the Backend VMs. We have written micro benchmarks to collect these statistics at each second on all three machines. We evaluate the policy using various workloads which are CPU intensive and Memory intensive and also present a sensitivity analysis showcasing the impact of varying Request rate per second on overall performance gain and reducing response time.

2. PROBLEM DESCRIPTION

Energy savings with cost effective solutions has become an important area of study today. Raspberry Pi is one such device that offers computational capacities similar to a low-end server at a very cheap price. It is good for low level computations, however, it cannot be scaled to highly resource intensive applications. This necessitates offloading of the workload. However, most work done is concentrated on selective offloading at development time in mobile applications.[3] In this paper, we have aim to create a cost-efficient solution for dynamically offloading applications. Dynamically offloading the load to remote machines when Pi reaches a certain level of CPU or memory utilization provides us with performance guarantees as well as cost and energy savings.The main purpose of this paper to address the offloading concerns like when to offload, how to offload and the duration of offloading. We use several metrics like response time, response rate, throughput, CPU and memory utilisations and, energy and cost savings to come up with an efficient solution.

3. PRIOR WORK

Over the last decade or so, energy considerations have increased substantially and extensive research is being done on improving CPU efficiency while having energy savings.

Mobile Offloading has been primarily studied for offloading computation-intensive and resource-intensive mobile appli-

cations from smartphones to cloud.[8] To enable such selective offloading, partitioning approaches have been developed to be adapted during the development phase of a mobile application. [2]
The authors have introduced xCloud as a middleware system to provide seamless, multilevel task mobility support allowing fine-grained task migration among cloud nodes and mobile nodes.[6]
A case study was performed by the authors for offloading applications from a mobile device to remote machines to show energy savings.The idea was to offload energy consuming parts of applications rather than computationally intensive tasks.[1] [5]
To increase the battery life and to reduce the CPU load on the mobile devices, the authors have put forth a framework to offload applications from multiple devices on to cloud. They have achieved lower execution time and higher energy savings through offloading. However, their policy is to offload as many tasks as possible to the cloud and perform less tasks locally. [4]

4. EXPERIMENTAL SETUP

4.1 Hardware Characterization

The experimental setup consists of a Raspberry Pi, 2 Backend Virtual Machines, Httperf and HAProxy. Raspberry Pi is a credit-card sized computer with an SOC that includes an ARM compatible CPU and an on chip graphics processing unit GPU. It has 512 MB RAM. Given its incredibly small size, it can still efficiently work as a low-cost server. Its draws almost one tenth the amount of power required by a traditional computer. These features certainly make Raspberry Pi a viable option to work as a low-cost server to handle light internal or web traffic.
We use Raspberry Pi as our main device for serving applications and it also hosts the offloading algorithm. We are using 2 OpenStack low end virtual machines with 1 VCPU each of 2.5 Hz and 1 Gb RAM to operate as Backends for offloading Pi's applications when necessary. Httperf is a load generator and Performance benchmarking tool. It allows generation of HTTP workloads for Web Server Performance monitoring and construction of Micro and macro-level benchmarks. HAProxy is an open-source load balancing and proxying solution for TCP and HTTP-based applications.

4.2 Workload Heterogeneity

The applications running on Pi can have diverse workload characteristics. They can be memory intensive or CPU intensive. Workload characteristics can have a huge impact on the dynamic offloading policy adapted by Pi for offloading applications to backend servers. To explore the impact of various workload characteristics on the offloading algorithm we have developed a CPU Intensive Workload and Memory Intensive Workload. CPU Intensive Workload is generated by Httperf by invoking applications that perform computational heavy tasks. Memory Intensive Workload is generated by invoking applications that perform numerous memory accesses and allocations. We use these workloads to analyse the impact of different Request rates and total requests on the total Response Time.

4.3 Micro-benchmarks

Analysing the impacts of CPU and Memory intensive workloads on Response Time and as a basis for dynamic offloading policy requires the CPU and memory usage to be monitored on Pi and the 2 backend servers for making dynamic offloading decisions. It required developing a set of micro benchmarks that monitor the CPU and Memory usage on all three machines every second. These benchmarks run in the background running linux commands to capture system activity (CPU and Memory usage) and returns the statistics for last n seconds where n is probed by the offloading algorithm.

5. DYNAMIC OFFLOADING POLICY - OUR SOLUTION

5.1 Dynamic Load Balancer (HAProxy)

In our approach, we use dynamic load balancing technique to offload tasks from Raspberry Pi.The main idea is to distribute tasks to the back-end virtual machines as and when the workload on Pi becomes extremely high for it to handle (which in turn reduces the performance).
We have developed a process called resource monitor which monitors the resource utilization (CPU and memory) of Pi. Resource monitor process is integrated with HAProxy such that when load increases, it can configure HAProxy to dynamically enable or disable load balancing between Pi and the two banckend VMs.

5.2 Offloading Algorithm

In this work, we are demonstrating how a small web application hosted on Pi can be offloaded. To support Pi during occasional high loads, we have deployed to VMs in Openstack cloud with same application as that on Pi. To offload the traffic we have also used a load balancer that can be instructed to start/stop offloading. We have used HAProxy as a load balancer which is hosted on Pi.
The two python modules *monitor.py* and *collector.py* collaboratively take the decision of offloading. *Collector* module starts linux *sar* command to report CPU and Memory every second. Then collector records this usage history of last 60 seconds and makes it available to *monitor* over network.
monitor process queries collector every second and takes decision to offload using algorithm shown in Figure 1.

Initially, in HAProxy load balancer only Pi is enabled and backend VMs are disabled so that all the requests will be served by Pi. Also, in load balancer initial weights assigned to Pi and to Vms is 20, 80, 80 to account for imbalance between CPU/Memory of Pi and backend VMs, later being thrice as powerful as Pi. Backend VMs are enabled in steps. When load increases, first VM1 is enabled. When load stays high even after enabling VM1, then VM2 is enabled. When resources utilization stays high on Pi even when load is shared among both the VMs, monitor starts dropping weight of Pi in steps of 5. When weight of Pi reaches zero, monitor instructs HAProxy to set Pi to *DRAIN* mode so that all subsequent new requests starts getting offload completely to VMs. At anytime during offloading, if CPU and Memory usage of Pi comes down below critical value and stays there for 3 seconds, weight of Pi in HAProxy is reset to default by the monitor and backend VMs are set to *DRAIN* mode to stop offloading.

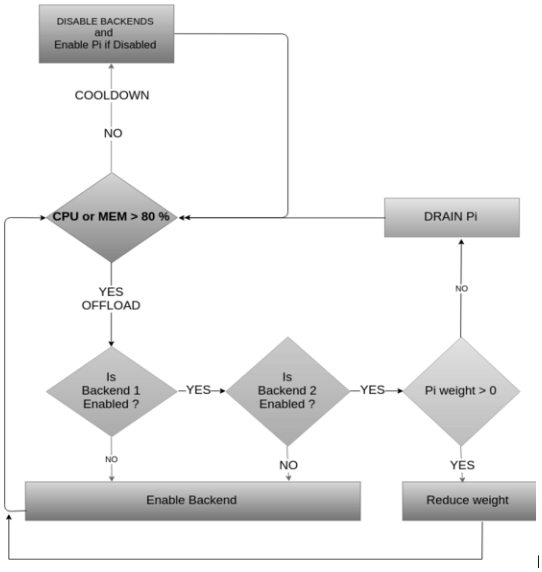

Figure 1: Dynamic Offloading Algorithm

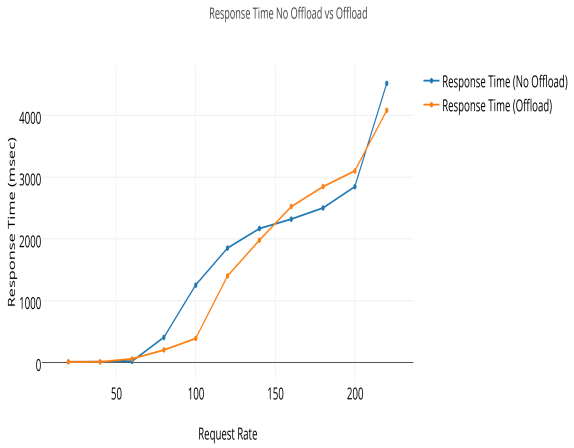

Figure 2: Response Time- No offloading vs. Offloading

6. EVALUATION AND RESULTS

Dynamic Offloading of applications on Raspberry Pi to Virtual Machines is a relatively new field of study. Not much concrete evaluation methodology has been developed on this front. Intuitively, offloading can provide much better results than not offloading for low end servers such as Raspberry Pi. Figure 1. shows the effect of offloading vs. not offloading for a CPU intensive workload. As it can be seen, offloading contributes a significant performance gain while always keeping the CPU Utilization below 100% and response times to be minimal.To evaluate our dynamic offloading policy, we perform Sensitivity Analysis where we comment on the sensitivity of overall response time to slight changes in Request rates per second. We also evaluate the performance of the offloading policy when total number of requests is quire varied. Without loss of generality, we also demonstrate the impact of varying different offloading parameters such as load threshold and types of workloads on the Response time metric.

6.1 Sensitivity Analysis

To understand the behaviour of the dynamic offloading algorithm for various scenarios, we provide the following sensitivity study. Offloading is sensitive to the request rate per second. Fig.2 shows the comparison between offloading and not offloading mechanisms at various request rates. Offloading almost always performs better except for a few times where the CPU faces offloading overheads. Following is a discussion of the sensitivity of performance to different request rates for CPU Intensive and Memory intensive workload.

6.1.1 CPU Intensive Workload

As mentioned earlier, Response times for a CPU Intensive workload are critically dependent on Request rates per second. To analyse the same, we have written some scripts that generate different request rates and capture response time for each workload. The request rates used for this analysis are shown in Figure 3.

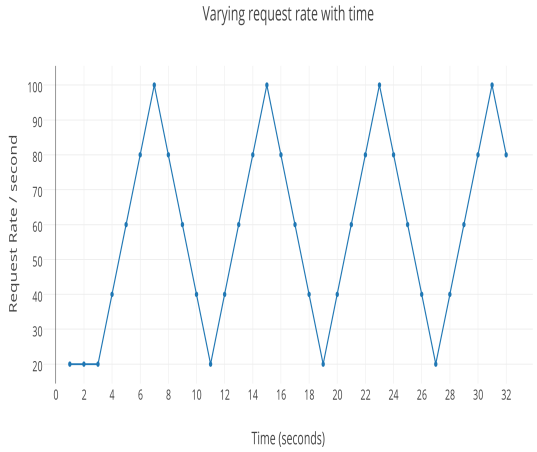

Figure 3: Varying rquest rate with time

Figure 4. shows CPU utilization with different request rates. It is obvious that Response time increases drastically with no offloading as request rate increases, however, offloading tries to maintain the CPU utilization below 80% for all request rates.

During low request rate, CPU utilization with offloading is similar to utilization without offloading as CPU stays below critical value. As request rate increases CPU utilization also increases. When CPU hits a value greater than critical value of 80%, offloading comes into action. This can be seen in plot of No-offloading where CPU stays 100% for longer time than plot of Offloading where CPU utilization gets pulled down due to dynamic offloading.

6.1.2 Memory Intensive Workload

While Response time is a critical metric for CPU intensive workload, the amount of memory pre-commit is an important metric for evaluating the performance on Memory Intensive Workloads. Memory precommit is the amount of

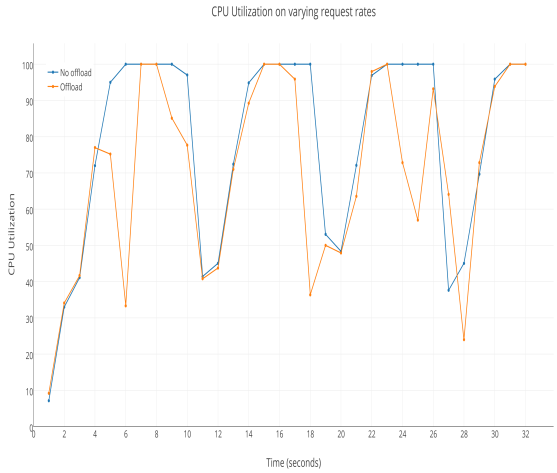

Figure 4: CPU utilization at varying request rates

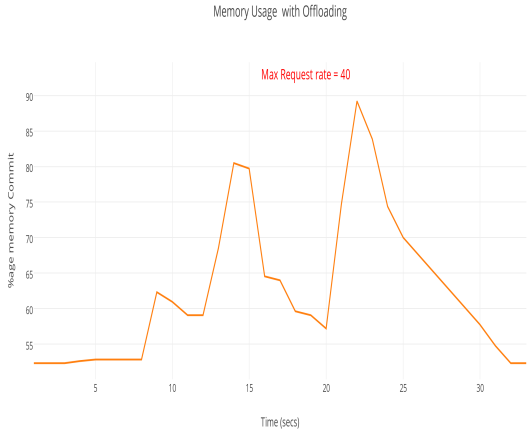

Figure 6: Memory Usage with Offloading

memory that is committed to the applications but not actually allotted. When memory commit exceeds 100%, it increases paging and swapping to free up memory to serve the applications that it has already been committed to. This also adversely impacts the response time. Keeping the memory pre-commit percentage below 100% helps in ensuring that response time will not suffer due to swapping etc.Also, with memory intensive applications, Pi cannot support a higher request rate. Figure 5. shows the comparison be-

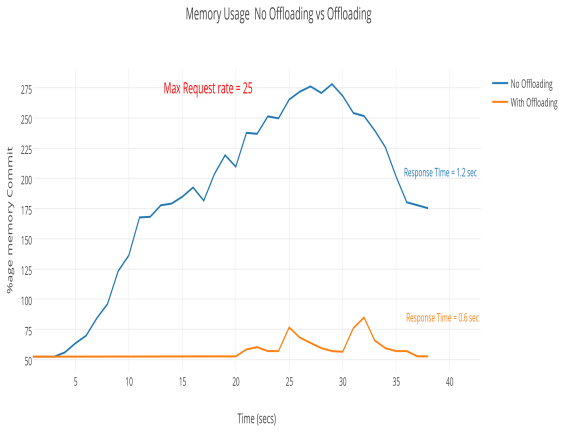

Figure 5: Memory Usage - Offloading vs. No Offloading

tween the request rates supported after offloading and without offloading. As it can be seen, Pi supports a maximum request rate of 25 with a memory pre-commit of 275%. Offloading, on the other hand always maintains the precommit below 100% and even at request rate equal to 25. Also, the response time for Pi without offloading is almost double as compared to the response time achieved via offloading. As demonstrated in Figure. 6, offloading allows the Pi to support request rates as high as 40 where the maximum supported without offloading was 25. Even with request rate equal to the max request rate, the memory pre-commit of Pi always stays below 100%. Figure 7. shows the distribu-

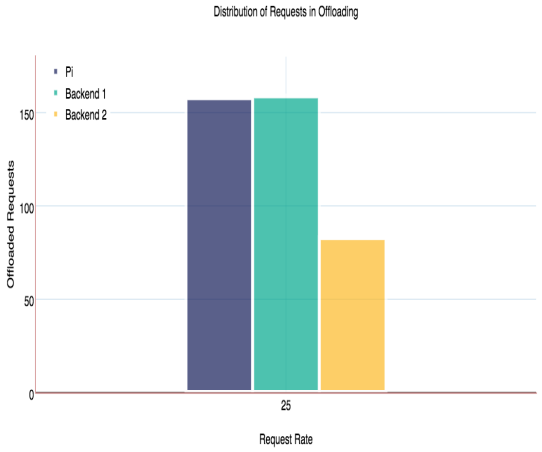

Figure 7: Distribution of requests in Offloading

tion of requests generated by the dynamic offloading algorithm. As it can be seen, the distribution in case of memory intensive workloads is not as rigorous as compared to the distribution in CPU intensive workload. This is due to the fact that the memory ratios of Pi and Virtual Machines is 1:2., i.e Pi has 512 Mb of RAM whereas Virtual Machines have 1 Gb of RAMs each. However, the ratio is very wide in case of computational capacity.

6.2 Varying Number of Requests

In the previous section, we presented the impacts of changing request rate on the response time obtained by implying dynamic offloading. In this section, we are presenting a study of impact on Response Time by varying the number of requests and keeping the request rate almost constant. This gives an overall sense of the extent of scaling that can be handled by using the dynamic offloading algorithm. We keep the request rate closer to 9.5 requests per second and vary the total number of requests from 20 to 520 gradually. In a normal scenario, without offloading, the reply rate

should intuitively decrease and the response time should increase with increasing number of requests with a stable request rate per sec. Since these requests are computationally heavy, we expect Pi's CPU to be fully utilized with a few requests and the reply rate would decrease resulting in the increase in Response time as it gets overwhelmed by incoming requests. This analysis is demonstrated in Figure.1

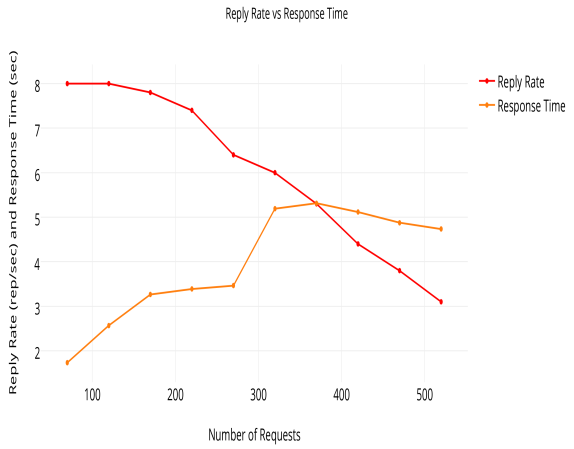

Figure 8: Reply Rate vs. Number of requests

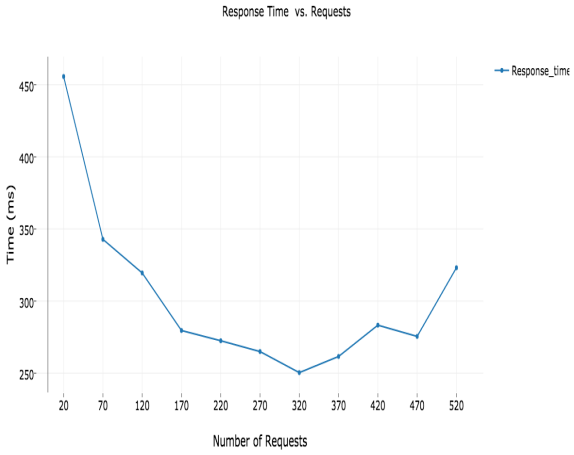

Figure 9: Number of Requests vs. Response Time

As it can be seen from Figure 2, the Response time actually decreases with offloading even when the number of requests increase. This behaviour can be attributed to the fact that, Pi does not always statically offload its requests. Initially it serves the requests on its own when the utilization is below the threshold. As it gets overwhelmed with new requests, it dynamically offloads some requests to 1 virtual machine while itself serving a few requests. Additional virtual machines are added if necessary which further lowers the response time. The spike in response time can be explained by the fact that even after addition of 1 Virtual machine, the gradual increase in number of requests might keep the utilization above threshold thus necessitating the addition of 2nd Virtual machine.

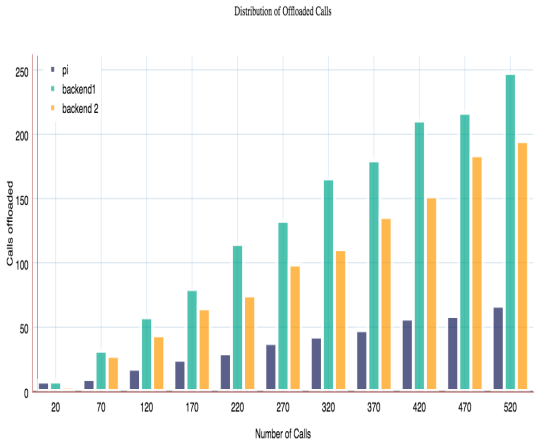

Figure 10: Distribution of Offloaded Calls

Figure 3. shows the distribution of requests generated by the dynamic offloading algorithm. The number of requests served by Pi is always very less as compared to the number of requests diverted towards the virtual machines by the algorithm. This can be reasoned by the fact that the computational capacity of the virtual machines is much higher than that of Pi. The number of requests diverted towards the second virtual machine is always less as compared to the first virtual machine. This is because the algorithm manages requests between Pi and first Virtual Machine until the CPU utilization still exceeds the threshold, after which the requests are also routed towards the second virtual machine.

7. CONCLUSION

We have presented a novel solution of using Raspberry Pi as a low end device to save power consumption and cost. We dynamically offload the tasks once the Pi's limits have reached to Backend VMs, ensuring that there is no drop in performance. Our results have shown that the average CPU and memory utilization does not go beyond critical values for longer periods of time because the load balancer dynamically offloads the tasks to the backend machines. Offloading also keeps response times under check and enables Pi to handle higher request rates for both CPU and memory intensive workloads.

8. REFERENCES

[1] Aki Saarinen, Matti Siekkinen, Y. X. J. K. N. M. K. P. H. Can offloading save energy for popular apps?

[2] Atta ur Rehman Khan, Mazliza Othman, F. X. A. N. K. Context-aware mobile cloud computing and its challenges.

[3] Byung-Gon Chun, Sunghwan Ihm, P. M. M. N. A. P. Clonecloud: Elastic execution between mobile device and cloud.

[4] D., K., and Klamma, R. Framework for computation offloading in mobile cloud computing.

[5] Mohammed A. Hassan, Kshitiz Bhattarai, Q. W., and Chen, S. Pomac: Properly offloading mobile applications to clouds.

[6] Ricky K. K. Ma, King Tin Lam, C.-L. W. Transparent runtime support for scaling mobile applications in cloud.

[7] Sokol Kosta Deustche, Andrius Aucinas, P. H. R. M. X. Z. Thinkair: Dynamic resource allocation and parallel execution in cloud for mobile code offloading.

[8] Soumya Simanta, Kiryonh Ha Grace Lewis, E. M., and Satyanarayanan, M. A reference architecture for mobile code offload in hostile environments.