Computation Offloading from Mobile Devices: Can Edge Devices Perform Better Than the Cloud?

Arani Bhattcharya
SUNY Korea
arani@sunykorea.ac.kr

Pradipta De
SUNY Korea
pradipta.de@sunykorea.ac.kr

ABSTRACT

Mobile devices like smartphones can augment their low-power processors by *offloading* portions of mobile applications to cloud servers. However, offloading to cloud data centers has a high network latency. To mitigate the problem of network latency, recently offloading to computing resources lying within the user's premises, such as network routers, tablets or laptop has been proposed. In this paper, we determine the devices whose processors have sufficient power to act as servers for computation offloading. We perform trace-driven simulation of SPECjvm2008 benchmarks to study the performance using different hardware. Our simulation shows that offloading to current state-of-the-art processors of user devices can improve performance of mobile applications. We find that offloading to user's own laptop reduces finish time of benchmark applications by 10%, compared to offloading to a commercial cloud server.

Keywords

Mobile systems; computation offloading; cloud computing; edge computing

1. INTRODUCTION

Computation offloading is a method of utilizing a server system to augment the battery life and processing power of mobile devices like smartphones. In computation offloading, a mobile application is split into two components – one each for execution on mobile device and server. Prototypes of such offloading frameworks have shown improved battery life and faster application finish time of smartphone applications [1, 2].

An interesting question in computation offloading is to decide which machines to use as server. Most offloading framework prototypes developed so far use an in-house desktop or server machine. Currently, offloading frameworks have two distinct choices of machines that can be used as server. The first choice involves offloading to commercially available cloud servers and is known as mobile cloud computing. Prototype implementations of offloading utilize this technique. The second choice, known as mobile edge computing, involves offloading to other user edge devices such as tablets, laptops or network routers.

ARMS-CC'16, July 29 2016, Chicago, IL, USA

ISBN 978-1-4503-4227-8/16/07…$15.00

DOI: http://dx.doi.org/10.1145/2962564.2962569

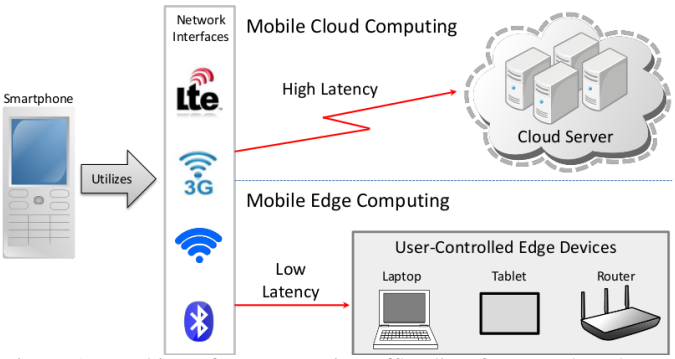


Figure 1: Working of a computation offloading framework. The offloading framework may use either low-powered user edge devices, such as routers, laptops and tablets, or commercially available cloud servers. These two cases are known as mobile edge and mobile cloud computing respectively.

Cloud servers and user edge devices have two major differences:

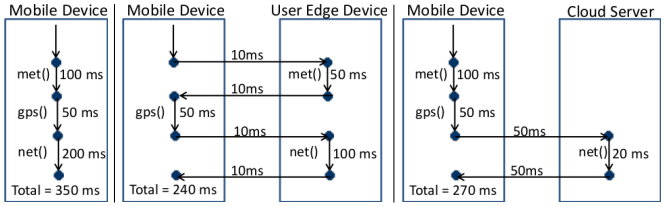
- Processors of cloud servers have faster processors. For example, running the benchmark CoreMark [3] shows that a Google Cloud Platform [4] processor is around 6 times faster than a mobile device.
- Latency of cloud servers is higher than edge devices. By performing a series of ping probes from our mobile device over Wi-Fi, we found that Google Cloud Platform has an average latency of 87 ms, compared to just 14 ms for a device within the same network.

One of the most important factors of user satisfaction is lower application finish time [5]. An offloading framework partitions the mobile application in a way that reduces its finish time. While execution on a cloud is much faster than on a mobile device, migrating data over the network between the mobile device and the server consumes additional time. Thus, a partitioning algorithm has to balance the trade-off between more execution of tasks on the server and ensuring less time spent on migrating data. The speed of the server and the network latency, therefore, has a major impact on the way the offloading framework partitions the application.

Fig. 2 shows how the partition generated for a simple program changes due to the type of server used. The application consists of three methods – met(), gps() and net() respectively. The method gps() depends on the GPS, and thus must be executed on the mobile device. When the application is executed entirely on the mobile device, execution of the application takes 350 ms. When the offloading framework can offload to an edge device, both the methods met() and net() are offloaded. This takes a total of 240 ms.

	Latency	Processor Power
Edge Device	10 ms	2 times
Cloud Server	50 ms	10 times

(a) Parameters used for the example. Latency refers to the latency of server from mobile device. Processor power refers to speed of processor compared to mobile device.



(b) (c) (d)

Figure 2: An example of how server device affects application partitioning. Parameters used for this example are shown in Fig. 2a. Fig. 2b shows execution of the application entirely on mobile device. Fig. 2c and Fig. 2d show execution by offloading to an edge device and cloud server respectively.

However, when it has access to a cloud server, the higher latency ensures that executing met() on the mobile device is faster. Thus, only net() is offloaded. Using a cloud server, therefore, gives an application finish time of 270 ms. In this way, the application partition changes depending on the type of server used.

A key question here is if offloading to user edge devices can provide faster application finish time compared to commercial cloud servers. Edge devices have weaker processors than cloud servers, but also have lower latency. Whether the lower latency of edge devices can compensate for the weaker processors compared to cloud servers needs to be investigated. Such an investigation needs to study the performance of offloading using different cloud servers under realistic workloads.

In this work, we compare the performance of offloading using cloud server and to user edge devices. We study the impact of using different servers on application finish time and application partition. We first develop a system model that can be used to study both cases of offloading. We collect traces of SPECjvm2008 [6] programs using aspect-oriented programming. Aspect-oriented programming allows us to modify the bytecode of an existing application at run-time to log details of methods executed. We then perform trace-driven simulation to determine the application finish time using cloud server and edge devices of SPECjvm2008 benchmark programs. We find from our trace-driven simulation that edge devices for general-purpose computing, such as laptops, can perform better than cloud servers. Smaller edge devices, like tablets and routers, can also reduce application finish time, but gives slower performance than cloud servers. Our work, therefore, shows that offloading to edge devices is an attractive option for smartphone users.

The rest of this paper is organized as follows. Section 2 describes related work. We develop our formal model of offloading in Section 3. Section 4 describes our techniques to generate traces of the workload and measure the parameters required for simulation. We discuss our simulation results in Section 5, and some of the limitations of our approach in Section 6. We conclude in Section 7.

2. RELATED WORK

In this section, we first start with discussion of related offloading frameworks. We then explain studies that target managing latency of cloud servers. Finally, we discuss related works on offloading to edge devices.

The first computation offloading frameworks from mobile devices, MAUI [1], CloneCloud [2] and Odessa [7], used a single desktop or server machine as remote server. These systems usually used a software-based middleware to vary the network latency to simulate the latency of cloud servers. Other offloading frameworks, such as ThinkAir used a custom-made server with many different virtual machine (VM) configurations [8]. Barbera et al. [9] performed a trace-based study of energy gains using a commercial cloud service Amazon EC2. Another study used PlanetLab servers to study the effect of latency on interactive smartphone applications such as games [10]. These studies first identified high latency of cloud servers as a major problem in computation offloading.

A second category of studies on offloading suggested installation of computing resources with ready access to energy in the vicinity of mobile devices [11]. Such computing resources are called cloudlets. Since cloudlets are closer to mobile devices, they have much lower latency. However, utilizing cloudlets require installation of additional computing infrastructure. This has slowed their adoption.

Another group of studies aim to reduce the latency of existing cloud data centers. For example, QJUMP suggests using separate queues for latency-sensitive applications that utilize the cloud server [12]. Silo provides guarantees of network latency by utilizing network calculus [13]. Finally, a recent proposal suggests inferring the latency requirements of an application by studying its request patterns [14],

Finally, a few recent studies have suggested utilization of edge devices in the context of Internet of Things (IoT). This is known as fog computing [15]. For example, mobile fog suggested utilizing of distributed network devices such as routers closer to the mobile devices [16]. Garcia Lopez et al. [17] proposed a more general type of offloading, where application is offloaded to different user devices. This is known as mobile edge computing. Our study builds on these works by studying the feasibility of utilizing edge devices using trace-driven simulation.

3. TASK PARTITIONING MODEL

In this section, we first explain the way computation offloading works. We then utilize this technique to develop its formal mathematical model.

3.1 Preliminaries

A computation offloading system consists of several processors p_k both in the mobile device and server. We represent the set of processors as $\mathbb{P} = \{p_1, p_2, ..., p_m\}$. A subset of these processors $\mathbb{M} \subset \mathbb{P}$ belong to the mobile device.

A mobile application is represented using a Directed Acyclic Graph (DAG) $G = (\mathbb{V}, \mathbb{E})$. A vertex v_j in the vertex set $\mathbb{V}$ represents a method or a task in the application. On a processor p_k a task v_j takes t_j^k time to execute. The value of t_j^k depends on the task v_j and the power of the processor p_k. An application begins and ends on the mobile device. Moreover, the execution of a set of tasks $\mathbb{U}$ may be tied to some hardware such as camera or GPS present only on the mobile device. Dependency from a task v_i to v_j is represented as an edge (v_i, v_j). The set of dependencies is the edge set $\mathbb{E}$ of the DAG. For each dependency (v_i, v_j), execution of task v_j can begin only when v_i finishes. Moreover, if v_i and v_j execute on different machines, then program states must be migrated. Let r_{ij}^{hk} denote the time to migrate data from processor p_h to p_k for (v_i, v_j). The value of r_{ij}^{hk} depend on the location of the processors p_h and p_k as well as on the amount of data that needs

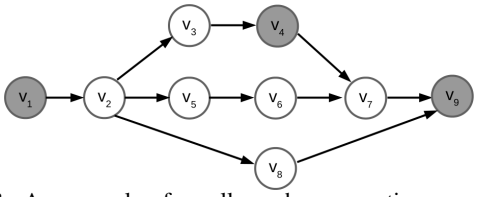


Figure 3: An example of a call graph representing a mobile application. Methods shaded gray must be executed on the mobile device.

$\mathbb{V}$	Task set of application
$\mathbb{E}$	Dependency set of application
$\mathbb{U}$	Set of tasks that must be executed on mobile device
$\mathbb{M}$	Set of processors in mobile device
$\mathbb{P}$	Set of processors in mobile device and server system
v_j	A method in the call graph
p_k	A processor in the system (mobile device or servers)
t_j^k	Execution time of a task v_j on processor p_k
m	Number of processors in the system
n	Number of tasks in the mobile application
(v_i, v_j)	Data dependency from task v_i to v_j
r_{ij}^{hk}	Migration time of dependency (v_i, v_j) from p_h to p_k
x_j^k	Decision variable denoting if v_j is executed on processor p_k
T_j	Finish time of the task v_j
S_j	Start time of the task v_j
R_{ij}	Migration time of the edge (v_i, v_j)
σ_{ij}	Decision variable denoting if v_i is executed before v_j

Table 1: Symbols and variables introduced in Section 3

to be migrated for (v_i, v_j). We assume that execution time t_j^k and migration time r_{ij}^{hk} are known a priori by profiling the application. We note that prior profiling to get execution and migration time is common in offloading systems.

3.2 Mathematical Model

The offloading framework needs to decide the processor p_k on which each task v_j executes. To denote this, let x_j^k be a binary decision variable such that:

$$x_j^k = \begin{cases} 1 & \text{if task } v_j \text{ is executed on processor } p_k \\ 0 & \text{if task } v_j \text{ is not executed on processor } p_k \end{cases}$$

Let the start time and execution duration of a task v_j be S_j and l_j respectively. Also, let the completion time of v_j be T_j. Our aim is to reduce the finish time of mobile application. This is equal to the finish time of the last task v_n of the application. Thus, our objective is to minimize the finish time T_n of the last task v_n:

$$\text{Min } T_n \tag{1}$$

The finish time T_j of a task v_j is equal to the sum of its start time S_j and its execution time t_j^k on the processor p_k. Mathematically,

$$\forall v_j \in \mathbb{V}, T_j = S_j + \sum_{j=1}^{m} x_j^k t_j^k \tag{2}$$

A task v_j can start executing only if its predecessor tasks v_i become available. A predecessor task v_i becomes available, when v_i finishes execution and its data is migrated to the processor where v_j is executed. Thus, starting time S_j of v_j is not less than the sum of finish time T_i of v_i and migration time R_{ij}.

$$\forall (v_i, v_j) \in \mathbb{E}, S_j \geq T_i + R_{ij} \tag{3}$$

The migration time R_{ij} of an edge (v_i, v_j) is the time needed to fetch output of v_i to execute v_j. If an edge (v_i, v_j) is migrated from processor p_h to p_k, this has a cost of r_{ij}^{hk}. Here we assume that if p_h and p_k are same processors, then $r_{ij}^{hk} = 0$. We represent the migration cost mathematically as:

$$\forall (v_i, v_j) \in \mathbb{E}, R_{ij} = \sum_{(v_i,v_j)\in\mathbb{E}} \sum_{h=1}^{m} \sum_{k=1}^{m} x_{ij}^h x_{ij}^k r_{ij}^{hk} \tag{4}$$

The constraints that we have defined so far do not limit the amount of parallelism. However, the number of processors available is limited. Thus, the offloading framework also needs to decide the sequence of execution of tasks. To denote this, let σ_{ij} be a binary variable for all pair of tasks t_i and t_j such that:

$$\sigma_{ij} = \begin{cases} 1 & \text{if } v_i \text{ finishes execution before } v_j \text{ begins execution} \\ 0 & \text{otherwise} \end{cases}$$

Thus, we can now rewrite Equation 3 as:

$$\forall v_i, v_j \in \mathbb{V}, S_j \geq \sigma_{ij}(T_i + R_{ij}) \tag{5}$$

For all edges (v_i, v_j) in the graph, the task v_j has to be executed only after v_i has completed. This precedence constraint is represented using the variable σ_{ij}.

$$\forall (v_i, v_j) \in \mathbb{E},\ \sigma_{ij} = 1 \tag{6}$$

Moreover, for any pair of tasks v_i or v_j, either v_i must be executed before v_j or vice-versa. Mathematically, we represent this as:

$$\forall v_i, v_j \in \mathbb{V},\ \sigma_{ij} + \sigma_{ji} \leq 1 \tag{7}$$

If the tasks v_i and v_j are scheduled by the offloading framework concurrently, i.e. $\sigma_{ij} = \sigma_{ji} = 0$, then they must execute on different processors. Thus, in this case, only one of the values among x_i^k and x_j^k can be equal to 1. Mathematically, we represent this constraint as:

$$\forall v_i, v_j \in \mathbb{V}, \forall k = 1...m, x_i^k + x_j^k \leq 1 + \sigma_{ij} + \sigma_{ji} \tag{8}$$

Finally, the tasks $v_j \in \mathbb{U}$ can only be executed on the mobile device. In other words, they can be executed on any one of the processors $p_k \in \mathbb{M}$. Mathematically, we represent this as:

$$\forall v_j \in \mathbb{U},\ \sum_{p_k \in \mathbb{M}} x_j^k = 1 \tag{9}$$

All other tasks $v_j \in \mathbb{V} - \mathbb{U}$, must be executed on any one processor from any device, i.e.

$$\forall v_j \in \mathbb{V} - \mathbb{U},\ \sum_{p_k \in \mathbb{P}} x_j^k = 1 \tag{10}$$

Equations 1 to 10 provide a formulation of an offloading system with multiple processors on different devices. An optimization solver on solving this simulation system gives the values of x_i^k to denote the processors on which each task is executed, and σ_{ij} to denote the execution sequence of each task. We now utilize this formal model to develop our simulation system.

4. METHODOLOGY

In this section, we first describe our method of collecting traces of applications. We then explain our technique of measuring different parameters required for simulation.

4.1 Generation of Call Graph

We use aspect-oriented programming to generate an annotated call graph. Aspect-oriented programming (AOP) is a technique of adding additional code to an existing program, without directly

modifying its source code. The additional code is called aspect. AOP can even work in cases where source code is not available by modifying intermediate code of the application.

We utilize AspectJ, which is a common framework for aspect-oriented programming in Java [18]. AspectJ can add additional code at run-time to modify the behavior of an existing program. We treat the entry and exit points of each method as possible migration points. For our purpose, we log details of each method at their possible migration points.

To form a call graph from a program that is useful for our purpose, we collect the following data for each method:

- Method name, including its formal parameters and return types
- Thread identifier
- Execution time
- Amount of data that needs to be migrated

We obtain the method name by accessing the stack trace of the current thread. Similarly, Java provides a method within Thread class to access the thread identifier of the current thread. To obtain the execution time, we utilize Java's ThreadMXBean[1] interface. Finally, to obtain the amount of data, we serialize the objects of each argument and return types, write it to a memory buffer and then calculate its length. We then use a java agent at run-time to obtain these data from the benchmark programs. We use the time command on the benchmark programs to find out the overhead of utilizing aspect-oriented programming. Our measurements showed an overhead of less than 10% on the execution time of benchmark programs.

4.2 Estimation of Simulation Parameters

A realistic estimate of performance using both cloud and edge devices requires measurement of their processor performance and network latency. To compare the performance of processors with different instruction set architectures, we use CoreMark benchmarks [3]. CoreMark is a set of common benchmark programs, containing matrix multiplication, linked-list manipulation and Cyclic Redundancy Check. One of its major advantages is that it is widely available for execution on different platforms, including desktops, servers and mobile devices. It is a widely used technique of comparing processor performance across platforms. The values obtained using these benchmarks is taken as a representative of the overall processor performance. Table 2 shows a list of platforms on which we perform our experiments.

We measure the network latency of user devices and cloud servers by using ping probes. We use the ping utility to send 100 ping probes and then take their average latency values. This is a standard technique widely used in measuring the latency values. Our experiments showed an average latency of 14 ms for user-controlled edge devices and 87 ms for our cloud server.

5. TRACE-DRIVEN SIMULATION

We selected a set of SPECJVM08 benchmarks that are relevant to mobile devices. The benchmarks we selected include common workload such as encryption, data compression, Fast Fourier Transform and audio decoder. We generate the traces of these benchmark programs using the technique described in Section 3. We then implemented the mathematical model described in Section 3 as an Integer Linear Programming (ILP) problem in Matlab.

[1]http://docs.oracle.com/javase/7/docs/api

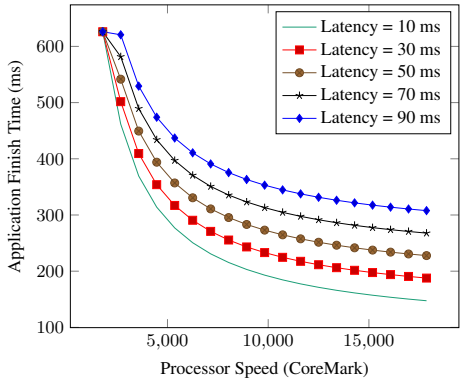


Figure 5: Impact of processor speed and latency on application finish time. Processor speed is measured using the CoreMark benchmark value.

Fig. 4 shows the application finish time of benchmark applications using both edge devices and cloud server. We note that all server systems (edge and cloud) improve application finish time. Moreover, the best performance for each benchmark is obtained using a laptop, followed by cloud server, tablet and router respectively. A cloud server reduces the average application finish time by 46% over local execution, compared to 52% for laptop, 43% for tablet and 41% for router.

Our trace-driven simulation, therefore, shows that user-edge devices can perform better than a cloud server if they have sufficiently powerful processors. The slower processors of user-edge devices is compensated by the lower latency to access these devices. Even a smaller edge-device like router improves the performance of the benchmark applications significantly.

To further study the effect of processor performance and network latency, we vary the latency and processor power of the server in our simulation. We then study the average application time across the ten benchmark programs.

Fig. 5 shows the impact of processor speed and network latency on application finish time. We note that at a latency of around $90ms$, improving the processor speed does not have much impact on the finish time. At lower latencies, the impact of more powerful processors is much greater. In each case, we observe that the gains of increasing the processor speed diminish after reaching a value of around 8000 Coremarks.

These observations show that the most significant factor limiting performance of mobile cloud systems is high network latency. Since cloud systems are accessed over the Internet, the cloud server provider has only a limited role in reducing latency. On the other hand, user-controlled edge devices, despite having slower processors, have much lower latency. Moreover, due to improvement in processor technology, the processor speeds of such devices is continuously improving. Thus, offloading to user-controlled edge devices is likely to become more attractive for smartphone users in the near future.

6. DISCUSSION

Our trace-driven simulation makes two assumptions. We do not discuss the consider the execution of other processes on edge devices. Execution of other processes lead to time-sharing of processors and increase the response time of smartphone requests. However, our simulation results show that even slower edge devices sig-

[2]The coremark value of this device is an estimate based on the values of similar processors.

Device	Model	Processor	CoreMark value per core
Smartphone	Samsung Galaxy S3 [19]	Quad-core 1.4 GHz Cortex-A9	1786
Tablet	Samsung Google Nexus 10 [20]	Dual-core 1.7 GHz Cortex-A15	3850
Router	ASUS onHub [21]	Qualcomm IPQ8064	3500 [2]
Laptop	Sony VAIO Notebook[22]	AMD Dual-Core Processor E-350 (1.6 GHz)	4960
Cloud	Google Cloud Platform[4]	n1-standard-8	10906

Table 2: A list of Coremark values per core in different hardware devices. Coremark values per core are taken as representative of the processing speed of a single core.

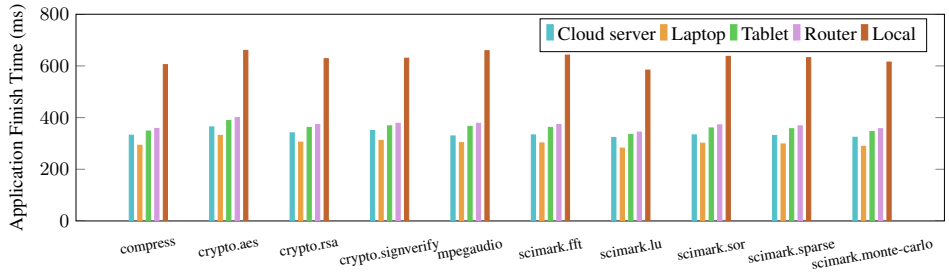


Figure 4: A comparison of finish time of different benchmark applications by offloading to cloud server, laptop, tablet, router and without utilizing offloading.

nificantly reduce application finish time. Secondly, we use CoreMark benchmark as a measure of the processor performance. Although CoreMark is a widely used processor benchmark, a proper study of processor speed requires running a variety of workloads. Our trace-driven simulation experiments assume that the processor performance remains approximately similar for different applications.

7. CONCLUSION

In this paper, we compare the performance of offloading from smartphone to a cloud server and user-controlled edge devices such as laptops, tablets and routers. We first formulate a mathematical model to represent the offloading problem. We then utilize aspect-oriented programming to obtain traces of benchmark Java programs. We perform trace-driven simulation to determine whether offloading to edge devices can reduce application execution time. Our simulation shows that offloading to larger edge devices such as laptops can provide better performance than a cloud server. Smaller edge devices such as tablets or routers provides slower performance than a cloud server, but can also significantly speed up application execution. Thus, offloading to such devices is a promising technique of augmenting the processor resources of smartphones.

As future work, we would like to study the impact of offloading to user-controlled edge devices on energy consumption. Smartphones can utilize bluetooth to connect to user devices, which consumes much lower energy. We would like to explore the impact of utilizing bluetooth for offloading smartphone applications.

Acknowledgment

This research was supported by MSIP (Ministry of Science, ICT and Future Planning), Korea, under the ICT Consilience Creative Program (IITP-2015-R0346-15-1007) supervised by IITP (Institute for Information & communications Technology Promotion).

References

[1] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: Making smartphones last longer with code offload," in *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '10. New York, NY, USA: ACM, 2010, pp. 49–62.

[2] B.-G. 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*, ser. EuroSys '11. New York, NY, USA: ACM, 2011, pp. 301–314.

[3] "Eembc – the embedded microprocessor benchmark consortium," http://www.eembc.org/coremark, 2015, online; accessed 26 February 2016.

[4] "Google cloud computing, hosting platform & apis," cloud.google.com, 2016, online; accessed 26 February 2016.

[5] J. C. Mogul and R. R. Kompella, "Mobile cpu's rise to power: Quantifying the impact of generational mobile cpu design trends on performance, energy, and user satisfaction," in *High Performance Computer Architecture (HPCA), 2016 IEEE 22nd International Symposium on*, Mar. 2016.

[6] "Specjvm2008," https://www.spec.org/jvm2008/.

[7] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan, "Odessa: Enabling interactive perception applications on mobile devices," in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '11. New York, NY, USA: ACM, 2011, pp. 43–56.

[8] 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 *INFOCOM, 2012 Proceedings IEEE*, March 2012, pp. 945–953.

[9] M. Barbera, S. Kosta, A. Mei, and J. Stefa, "To offload or not to offload? the bandwidth and energy costs of mobile cloud computing," in *INFOCOM, 2013 Proceedings IEEE*, April 2013, pp. 1285–1293.

[10] H. Wang, R. Shea, X. Ma, F. Wang, and J. Liu, "On design and performance of cloud-based distributed interactive applications," in *Network Protocols (ICNP), 2014 IEEE 22nd International Conference on*, Oct 2014, pp. 37–46.

[11] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *Pervasive Computing, IEEE*, vol. 8, no. 4, pp. 14–23, Oct 2009.

[12] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. M. Watson, A. W. Moore, S. Hand, and J. Crowcroft, "Queues don't matter when you can jump them!" in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, May 2015, pp. 1–14.

[13] K. Jang, J. Sherry, H. Ballani, and T. Moncaster, "Silo: Predictable message latency in the cloud," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. New York, NY, USA: ACM, 2015, pp. 435–448.

[14] J. C. Mogul and R. R. Kompella, "Inferring the network latency requirements of cloud tenants," in *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. Kartause Ittingen, Switzerland: USENIX Association, May 2015.

[15] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, ser. MCC '12. New York, NY, USA: ACM, 2012, pp. 13–16.

[16] K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwälder, and B. Koldehofe, "Mobile fog: A programming model for large-scale applications on the internet of things," in *Proceedings of the Second ACM SIGCOMM Workshop on Mobile Cloud Computing*, ser. MCC '13. New York, NY, USA: ACM, 2013, pp. 15–20.

[17] P. Garcia Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere, "Edge-centric computing: Vision and challenges," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 5, pp. 37–42, Sep. 2015.

[18] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, "Getting started with aspectj," *Commun. ACM*, vol. 44, no. 10, pp. 59–65, Oct. 2001.

[19] "Samsung galaxy s3," http://www.samsung.com/global/galaxys3/, online; accessed 26 February 2016.

[20] "Nexus – google," https://www.google.com/nexus/, online; accessed 26 February 2016.

[21] "onhub – google," https://on.google.com/hub/, online; accessed 26 February 2016.

[22] "Sony asia-pacific," http://www.sony-asia.com.