2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

COCA: Computation Offload to Clouds using AOP

Hsing-Yu Chen
Intel-NTU Connected Context Computing Center
National Taiwan University
Taipei, Taiwan
Email: henry74918@gmail.com

Yue-Hsun Lin
CyLab
Carnegie Mellon University
Pittsburgh, PA, USA
Email: tenma.lin@gmail.com

Chen-Mou Cheng
Department of Electrical Engineering
National Taiwan University
Taipei, Taiwan
Email: ccheng@cc.ee.ntu.edu.tw

***Abstract*—**

In this paper, we describe COCA—Computation Offload to Clouds using AOP (aspect-oriented programming). COCA is a programming framework that allows smartphones application developers to offload part of the computation to servers in the cloud easily. COCA works at the source level. By harnessing the power of AOP, COCA inserts appropriate offloading code into the source code of the target application based on the result of static and dynamic profiling. As a proof of concept, we integrate COCA into the Android development environment and fully automate the new build process, making application programming and software maintenance easier. With COCA, mobile applications can now automatically offload part of the computation to the cloud, achieving better performance and longer battery life. Smartphones such as iPhone and Android phones can now easily leverage the immense computing power of the cloud to achieve tasks that were considered difficult before, such as having a more complicated artificial-intelligence engine.

***Keywords*-computation offload; aspect oriented programming; Android operating system;**

I. Introduction

Smartphones have become a necessity for our daily life because they empower people with lavish applications and utility. New smartphones like Apple iPhones and Google Android phones [1] allow previously unimaginable applications to be designed and developed. These smartphones are all capable of broadband communication via 3G or Wi-Fi. They also integrate many basic and advanced sensors such as cameras, GPS, inertia and motion sensors, just to name a few. As a result, we now perform a wide variety of daily activities on these smartphones, e.g., voice calls, teleconferencing, emails, online gaming, social networking, etc.

Despite Moore's law and the constant improvement in battery technology, today's smartphones still suffer from battery-life problems. For example, a Nexus One phone would completely exhaust its battery energy in six hours if the user keeps it busy on Wi-Fi [2]. Even if the phone is idle, i.e., fully awake but not running any active applications, its battery will not last for more than 15 hours. Most users will probably admit that such a battery life is at most tolerable but far from ideal.

A promising solution to this problem is *computation offload*. If an application on a smartphone requires some intensive computation, it might make sense to shift the burden to some remote servers. However, setting up and managing these servers is not trivial for average, non-expert users. Furthermore, the diversity in telecom operators' network architectures can make the problem even more complicated. In such a scenario, cloud computing naturally provides a clean solution. Cloud computing allows new services to roll out with a minimal fixed investment and a usage-proportional operational cost. Take Amazon EC2 as an example: EC2 provides on-demand computational power and scalable storage to its subscribers, often start-ups who want to provide new services. They can then focus on building high-quality services that are easy to use and configure for average users.

In this paper, we propose and build a new programming framework called COCA, which offloads computation from smartphones to the cloud using AOP. Currently, COCA works for Java applications running on Android, but in principle, it can be extended to any other programming systems where there is AOP support. COCA works at the source level, while previous works in the literature work at the binary level, mostly based on techniques that alter or rewrite part of the application binary code. The binary-level approaches have a very attractive feature: the offload can be made transparent to the application programmers and the users. Therefore, it can work with existing, unmodified applications without needing their source code. The offload framework simply does all the work under the rug.

Nevertheless, the benefits of such a binary-level approach may become less and less important in the cloud-computing era. To begin, cloud computing has changed and will continue to change many facets of computing and the software industry. For example, software distribution and deployment have been made much less expensive than before, and as a result, updates and new versions of software come out at an unprecedentedly increasing rate. This offsets most of the benefits of binary-level offload approaches, now that shipping new software is almost free, and instead of modifying the existing copy of the software on their phones where the users can now simply download updates from

978-0-7695-4691-9/12 $26.00 © 2012 IEEE
DOI 10.1109/CCGrid.2012.98

the cloud any time. Furthermore, since binary modification requires changes in the program loader, it introduces new security vulnerabilities. Open platforms like Android already have a lot of security issues, and people are working on various solutions ranging from more traditional approaches to the more radical ones such as those based on formal verification. Altering binary code and hacking program loaders probably will introduce a lot of headaches and hence are less compatible with these new security measures.

In contrast, COCA developers do not need to modify the application binary code *nor* the original source code. Instead, COCA allows the application programmer to choose which target objects or functions to offload. Alternatively, this choice can be made automatically based on the result of static and dynamic profiling without human intervention. By using AOP, COCA automatically generates and recompiles the corresponding source code into the remote modules to be offloaded to the server, in addition to the local modules to be installed and run on the smartphone. After build succeeds, the user registers the remote modules to his or her cloud account, and then he or she can immediately leverage cloud computing to enhance the software performance.

II. Background

A. Aspect-Oriented Programming

Aspect-oriented programming (AOP) [3] suggests that development and design of programs focus on "aspects." According to, e.g., the Oxford Dictionary, an "aspect" is a particular part or feature of something being considered. In AOP, an aspect of a program is a particular functionality, e.g., accounting, environment checking, logging, security, etc., that horizontally cross-cuts a plural number of threads of logical flow in a program.

For example, say we have a complicated software program that may sometimes need to check the user's access right through some authentication and authorization process depending on the current security policy. Ideally, this piece of security code is logically independent of the rest of the program and hence should be considered a cross-cutting part from the program's perspective. Furthermore, to increase modularity, we would like to separate this piece of code to an independent module. Without AOP, it would require the programmer to go through every line of code because such aspects are scattered and tangled in the source code. This is an example where AOP can be useful. The programmers can write these aspects in stand-alone modules and specify in a declarative way where these aspects cross-cut.

AOP is now a widely recognized software development technology, and the further information about AOP can be found in many sources and standard textbooks [4], [5].

B. AspectJ

AspectJ is an extension developed by PARC for the Java language [6]. Currently, AspectJ offers stand-alone and dependent plug-in for Eclipse IDE tool. AspectJ allows programmers to define "aspects," which provides pointcuts and advices for specific functions. Once a function is defined as a pointcut by AspectJ, the programmer can perform the corresponding advices with this pointcut in three cases, *before*, *after*, and *around* the target function call. This is the main AOP used in COCA.

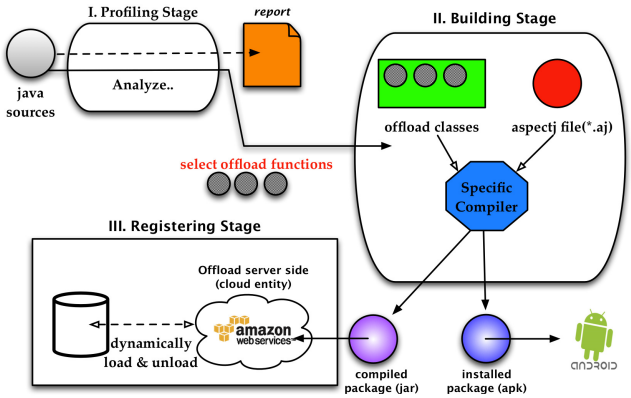


Figure 1. The System Diagram of COCA.

C. AspectJ for Android

There is no AOP support in the official Android build process. Therefore, we have modified the Android build process to support AspectJ. The major change is to alter the compilation phase of Android Java compiler. In the build process, a pre-compiled Android source file is ready to be translated to bytecode (`.class`). By swapping the Java compiler with AspectJ compiler, the modified compilation phase can now compile aspects (`.aj`) without destroying pre-processed resources, such as icons, layouts, and strings. The compiled applications will still run perfect on normal Android phones without any modification.

D. Dynamic Loading for Java Classes

The Java language suggests dynamic loading mechanism for classes in the current JVM implementation. Compiled bytecode can be loaded and run on a JVM dynamically in runtime. We utilize this technology on the cloud side to allow a running process to load the desired functions dynamically.

III. The Design of COCA

COCA consists of three stages, namely, profile, build, and register. A high-level overview of the entire system operation is shown in Figure 1.

A. Profile Stage

The first stage is profiling. COCA first marks all *pure functions* [1] in each Java class in the source code provided by the user. For each function, COCA evaluates the processing time of the function and the required memory footprint (heap

memory size) if it were to be offloaded to the cloud. The result of profiling is summarized in a report presented to the user, who can then decide which functions to offload based on the report. The question now boils down to how we can know which functions would perform better if we executed them in the cloud. To help make a smart decision, COCA allows evaluation of the candidate functions in an emulated environment, given parameters such as the actual execution time of the candidate functions on the smartphone, as well as the bandwidth available for offloading. Alternatively, with the information in the report, we can easily automate the selection process by integrating COCA with existing program partitioning schemes [7], [8], [9], [10].

Last but not least, we want to make sure the overhead introduced by COCA is reasonable. The majority of the overhead comes from the times when advices are inserted into the appropriate positions of the target function in AspectJ's code weaving. We will show via experiments that this overhead is quite insignificant in Section IV.

B. Build Stage

In this stage, the build scripts will divide the original Java source code into two parts, namely, those to and not to be offloaded. Once the user selects the target function to offload, COCA screens those classes dependent on the selected functions. COCA then translates the original Java source code into AspectJ code based on the result of selection. On the other hand, the filtered Java classes will be copied and compiled to JVM bytecode. In the end, the build scripts output a `jar` file for the cloud server, as well as an `apk` installation file for the Android smartphone.

C. Register Stage

The register stage is the final stage. First we assume the user already has an account on an existing cloud service provider such as Amazon EC2. The user can then use the account to run COCA server as a daemon in the cloud, waiting for requests from the mobile smartphones. As shown in Figure 1, the server daemon maintains a built-in database of functions that the smartphones may offload. After the build stage, COCA uploads the compiled bytecode in `jar` files to the cloud. The daemon then authenticates and loads the classes from the `jar` files via the dynamic loading technology for Java classes.

After these three stages, COCA is set up and ready for offloading. When the user launches the corresponding programs, COCA will make computation offload requests to the server daemon in the cloud. Upon receiving these requests, the daemon will retrieve the related classes from the database and then load the target classes into the processes. Next, it performs the computation by calling appropriate functions in the loaded classes. Finally, the result is sent back to the smartphones, and the smartphones can continue to accomplish the tasks.

[1]In a pure function, expect for input and output arguments, all objects and variables inside the function body are only used within the function itself. More information is given in Section V-B.

IV. Experimental Evaluation

A. Overhead of AspectJ on Android

We have performed two experiments to determine the overhead of AspectJ on Android on an HTC Tattoo smartphone, which has a Qualcomm MSM7225 CPU running at 528 MHz. The first is a simple approach, in which we compare the latency of function calls with and without AspectJ. The test program merely iterates through an empty for loop for ten million times. The experiment result shows that the overhead of the before or after advice is about 195 nanoseconds per function call. For the around advice, the overhead is about 290 nanoseconds per call.

The second experiment we have performed is on one of Google's open-source Android sample applications [11], namely, the **Amazed** application, a simple but addictive accelerometer-based marble-guidance game. After inserting a piece of code that calculates the frame rate, we find that it remains constant (around 26 frames per second on our test smartphone), either we call it from an AspectJ snippet or directly in Java. Therefore, we conclude that the overhead brought by AspectJ during the function calls is negligible, and using AspectJ in Android will not decrease the system performance in any significant way.

B. A Real-world Android Chess Game

To demonstrate the benefits of COCA, we have performed an experiment with an open-source chess game "Honzovy achy" [12]. COCA marks partial pure functions inside the AI module based on the profiling result. Then COCA separates objects that run locally on the smartphone and those to be offloaded. Host objects are compiled for installation on the smartphone, while objects to be offloaded are registered and uploaded to the cloud server. After running multiple rounds of the chess game, we have observed several enhancements, which we will describe in detail in the subsequent sections along with the measured overhead.

C. AI Capability Enhancement

The first enhancement that we have observed is that the artificial-intelligence (AI) level goes up under COCA. As we know, AI requires a lot of computational power. In Honzovy achy, the amount of time devoted to AI computation is limited by a configurable bound (default 5 seconds) when the program is trying to figure our the next moves in the chess game. On our experiment platform, the HTC Tattoo (Qualcomm MSM7225, 528 MHz CPU) can only reach the depth of 3 steps. Similarly, the deepest level for Google Nexus One (Qualcomm 8250 "Snapdragon," 1 GHz CPU) is 4. This result is within our expectation, as going one level deeper in such AI computation often requires exponentially

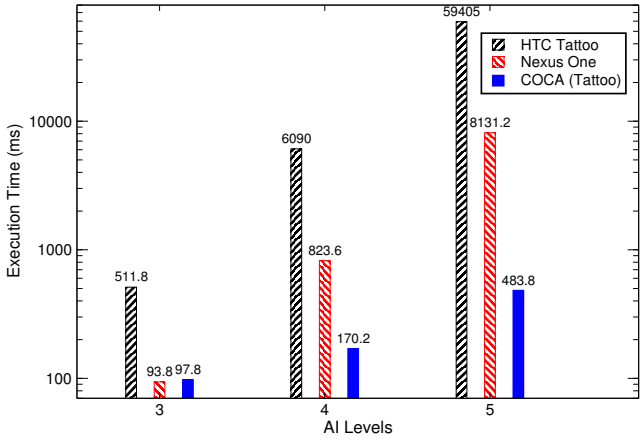


Figure 2. AI Computing Time for Different Levels and Devices

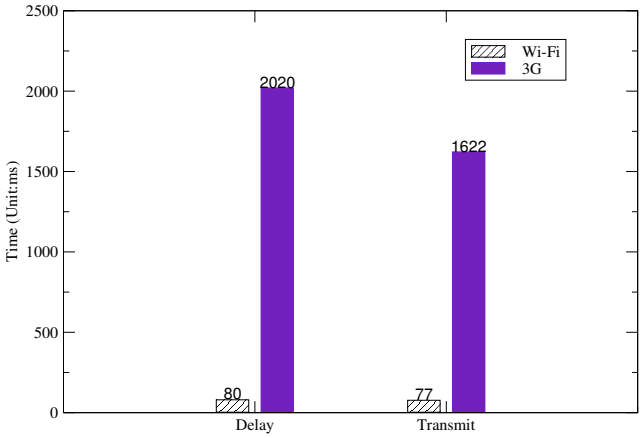


Figure 3. Transmission Cost for COCA Running on Google Nexus One

larger computational power. However, once we offload this AI computation to the cloud server, both Tattoo and Nexus One smartphones can easily search within a level of 5 steps in the same time bound. We have also asked Honzovy achy to compute at different levels of AI depths and measured the computing time each of the three schemes takes. If AI level is high, e.g., 5, Tattoo should spend around one minute to perform one move during the chess game. Even for Nexus One, it needs at least 8.13 seconds. The result is shown in Figure 2, in which we can clearly see that COCA helps enhance the AI capability of Honzovy achy significantly.

D. Communication Cost

Smartphones pay a communication price when leveraging the computation power of cloud servers via COCA, namely, they need to tell the servers *which* functions to perform on *what* data. We have performed several experiments to estimate this price for the most common communication interfaces including Wi-Fi and 3G data networks. For the 3G network we use, the typical uplink and downlink speed we get are 120 kbps and 509 kbps, respectively. Figure 3 shows the transmission time required when Google Nexus One smartphone utilizes COCA services to offload the AI computation in Honzovy achy.

Through a conventional Wi-Fi network, it takes about 77 milliseconds to transmit around 30 kilobytes of data from Nexus One to the cloud. For 3G network, the transmission time takes about 1622 milliseconds, as the speed is much lower than that of Wi-Fi network.

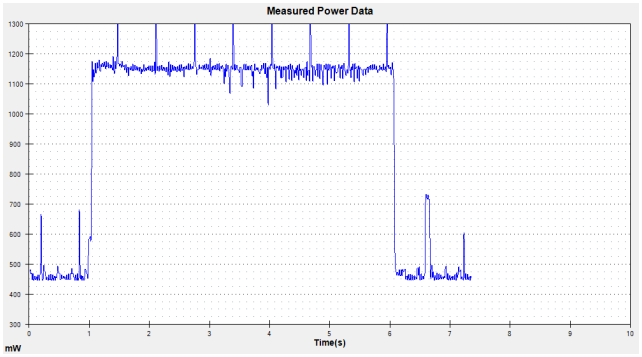
The Wi-Fi network also performs better in terms of latency. On the experiment Wi-Fi network, the latency is smaller than 100 milliseconds. The 3G network has a much worse latency, possibly due to a high level of congestion in the experiment urban area. Take Taipei city as an example, the average latency for 3G networks provided by Chunghwa Telecom is around 200 to 600 milliseconds based on our experience. To conclude, we believe that COCA should work very well on current Wi-Fi networks. For 3G networks, users should be able to get acceptable performance from COCA unless the network latency is extremely bad due to congestion.

E. Energy Savings

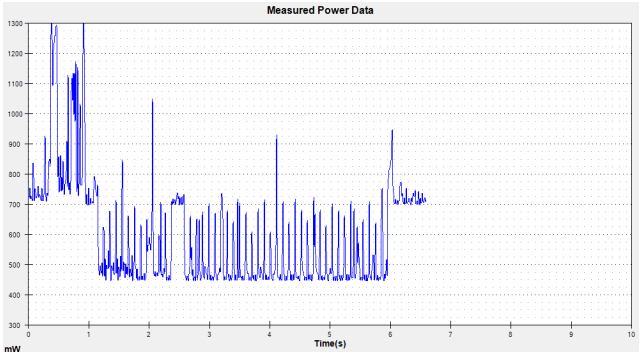
COCA can also provide energy saving on the smartphones, a very important feature for battery-powered devices in general. We use a Monsoon power monitor [13] to measure Nexus One's energy consumption. Figure 4(a) shows the power trace when the Honzovy achy AI computation is performed locally on a Nexus One smartphone. The power it consumes peaks around 1150 milliwatts, which is quite high for a smartphone, showing that AI computation is indeed a computationally intensive task. We then use COCA to offload this computation to the cloud side. As shown in Figure 4(b), the power consumption is clearly reduced after the first second. The burst in the first second is due to the fact that the smartphone needs to send some data to the cloud server through its Wi-Fi interface. In total, the smartphone consumes 443.88 μAh of energy without the help from COCA. After using COCA, the energy consumption is reduced to 247.91 μAh, which represents a 56% energy reduction on the smartphone, not to mention the resulting AI performance enhancement.

To obtain a breakdown of component-wise energy consumption inside the smartphone, we have also performed an experiment using the PowerTutor Android application [14], [15]. PowerTutor obtains component-wise energy consumption information based on modeling and estimation, so it is less accurate than Monsoon monitor. Based on our observation, PowerTutor tends to overestimate the actual energy consumption in many cases.

Although PowerTutor doe not give very accurate results as hardware power monitors do, the result is still reasonable



(a) Local Execution



(b) Cloud Execution

Figure 4. Power Traces of Local vs. Cloud Execution on Nexus One

Table I
COMPONENT-WISE ENERGY CONSUMPTION (UNIT: μAH)

Case	CPU	OLED	Network	Total
Local	180.18 (42.8%)	240.24 (57.2%)	0	420.42
COCA +Wi-Fi	2.25 (0.8%)	255.25 (87.1%)	32.96 (12.1%)	292.46
COCA +3G	2.18 (0.3%)	345.35 (44.2%)	431.43 (55.4%)	778.96

and does provide some values because it gives the detailed energy consumption information for each hardware component, such as the processor, Wi-Fi, and 3G interfaces. As shown in Table I, the processor itself consumes about 42.8% of the energy in a Honzovy achy game that runs locally. After COCA offloads the AI computation to the cloud, the processor uses a very small amount (less than 1%) of energy because it is in idle mode most of the time. On the other hand, we need to pay a price in network interface energy consumption. When Wi-Fi is used, COCA clearly gives a very good result, a 68% energy reduction compared to the local case. However, when 3G is used, the total energy consumption increases to about 1.85 times of the local case because 3G links consume a lot more energy than Wi-Fi links per bit transmitted. This shows that COCA can have a negative impact on energy consumption if communication is more "expensive" than computation, which is the case when 3G networks are used in our experiments.

V. DISCUSSIONS

A. Additional Arguments for Working at Source Level

In our design, COCA only works at the source level. It is arguable that other solutions like modified Dalvik VM is totally transparent to applications and hence might be superior and more user-friendly. The additional overhead for developers is zero, and the users only need to install the patched VM by themselves. As we have argued, patching the Dalvik VM on smartphones is not safe and convincing for most users. They may not accept unofficial patches for their smartphones, as this might bring additional security vulnerabilities.

Our motivation is to design a feasible offload mechanism for developers, not users. Developers (or programmers) could leverage COCA to reduce their burden on configuring their application for remote execution. COCA allows developers to simply set up the remote cloud servers by running a daemon to handle the offloaded computation. For users' Android smartphones, COCA guarantees the compiled application installation file works well on their devices.

One additional reason is to provide a better approach to modularize the source code. Basically, developers will prefer to offload those functions into the cloud that are complex or time-consuming. Hence, developers can simply isolate the design from mobile side and cloud side. If a new version of application has been released, developers can easily modify the program through remote objects (cloud side). The only overhead is recompiling the remote objects and registering them again. This model makes maintenance much easier for developers. We believe that COCA indeed saves developers time and shortens the software develop cycle on Android platform.

B. Pure vs. Non-pure Functions

As we have mentioned in Section III, COCA only allows developers to select pure functions that are independent of program states. The reason we select pure functions as first candidates is that offloading parameters are short and simple. Non-pure functions tend to access global variables, including primitive variables (int, double, float, char, etc.), active objects, and static object function calls. If we want to synchronize the function with the remote object, serialization is one of the feasible technologies to utilize. However, serializing a whole object only for few functions is a severe cost when the smartphone is on a network with limited bandwidth, e.g., GPRS or 3G data networks. Therefore, we select pure functions as our target in COCA. The cost is minimized because transmission simply includes the input and output of the function. Also, memory footprint

is optimized since we do not need to upload (register) the whole object class, only a trimmed version consisting of the selected pure functions.

C. Potential Applications

More and more applications are expected to perform on smartphones, but the resource requirement is never satisfied by current hardware capabilities. For example, 3D image rendering on smartphones are motivated by the cloud architecture. NVIDIA RealityServer [16] and OTOY's streaming platform [17] provides similar systems to enhance presentation's quality and performance through transmitting processed images. For Amazon EC2, there is also a solution called EnFuzion [18], which offers Image Rendering optimization to help lightweight devices. Therefore, 3D games seem a good fit for COCA. COCA can definitely improve the performance of these potential applications.

VI. Related Works

Remote execution of computationally intensive applications for resource-limited hardware is a subject under extensive study in mobile and pervasive computing [19], [20]. Among previous studies, we classify them into two categories, namely, *program partitioning* and *process and virtual machine migration*.

A. Program Partitioning

The first category for offloading to remote execution depends on programmers to partition the program and handle state synchronization among the threads of execution in different places. Spectra is a self-tuning tool that lets applications on lightweight devices utilize computational resources on large devices [21], [22]. In Spectra, the programmer needs to provide the strategy on how to partition an application through giving "fidelities." A fidelity is a metric of quality, e.g., vocabulary size in the context of speech recognition. Based on Spectra, Chroma introduced a declarative language to specify how a program utilizes the infrastructure resources.

Later, researchers began to investigate how to partition target programs automatically. Such tools separate the original program into two components, client and server, that are authorized to execute on the two sides. For example, Coign offers a coarse-grained splitting tool of DCOM applications [23]. For image processing applications, Kremer et al. proposed a compiler tool to select part of the workload for remote execution [24]. Following a similar spirit, Neubauer and Thiemann divided a C-like program into a set of nesC programs that are ready to carry out on sensors [25].

Another strategy of program partitioning is to build replicas from the mobile applications and synchronize the replicas over the network. Mobile applications first create their duplications on the remote side and then invoke them to perform particular tasks. For instance, cyber foraging uses surrogates opportunistically to improve the performance of mobile devices [26]. Similarly, both data staging [27] and Slingshot [28] use surrogate-based approaches. To enhance the performance, Slingshot creates a secondary replica of a home server at nearby surrogates.

There have been quite a few works on partitioning and offloading for binary Java programs [7], [8], [9]. Often in these methods, the application binary code is dynamically partitioned, and part of the functionalities are moved to a powerful nearby surrogate device for execution. Among them, a general approach for partitioning Java classes into groups is to transform the partitioning problem into the MINCUT problem [29]. In a typical problem of this sort, a cut that minimizes the component interactions between partitions needs to be found on some graph. Also, people have tried optimizing for a different metric, e.g., for energy saving instead of performance enhancing in MAUI [10]. In addition, MAUI also emphasized that the additional programming effort should be minimized using their tool.

B. Process and Virtual Machine Migration

A second type of approaches is to migrate the target processes to a duplicated operating system running on the remote side. There are several implementations such as Zap [30] that offer process migration on Linux through building checkpoints and restart. Similar to process migration, Clark et al. proposed live migration of virtual machines between distinct physical hosts [31]. They successfully achieved an impressive performance on resuming the migrated operating system to run in as little as 60 milliseconds. Alternatively, ISR provides an ability to suspend a virtual machine on one host and resume it on another by storing the VM images in a distributed storage system [32]. Later, Cloudlets [33] and CloneCloud [34] built on top of the work of Clark et al. to assist offloading in a mobile computing environment. Their approaches effectively reduce the efforts required by the application programmers through the use of replicas of the running operating systems on the users' mobile devices. As a result, the programmers do not need to modify their code at the source level or binary level.

C. Comparisons with COCA

Lastly, we compare COCA and existing works found in the literature from several aspects. Even though the experiments are performed in a wide variety of environments, we still try our best to compare COCA with these approaches in a fair and appropriate way. The features under comparison include the fundamental methodologies, the applicable devices, the applicable applications, energy savings, and performance improvements. The results are summarized in Table II.

Among these approaches, we find two major directions. The first is to leverage replica architectures [20], [27], [28], [22]. The earliest one by Rudenko et al. allows several

Table II
COMPETITIVE STUDY OF OFFLOAD SCHEMES

Approach	Methodology	Client/Server	Applications	Energy Savings (%)	Speed-up
Kermer et al. [24]	Compilation	Skiff sensor/ PC (P3 450 MHz)	Image processing	87.5%	13.9x
Rudenko et al. [20]	Replica	Dell Latitude XP	Gaussian elimination (matrix size: 500×500)	43%	N/A
Flinn et al. [27]	Replica (cache)	Compaq iPAQ 3850/ PC (P4 2 GHz)	Image browsing	5%–26% (cold) 10% (warm)	1.79x (cold) 2.17x (warm)
Slingshot [28]	Replica (full-VM)	Compaq iPAQ 3970/ PC (P4 3.06 GHz)	VNC Speech recognition	N/A	2.6x
Spectra [22]	Replica (RPC)	Compaq Itsy / PC (P3 700 MHz)	Speech recognition Document preparing (small files) Document preparing (large files)	91.7% 2% 68.4%	12.7x 1.73x 3x
MAUI [10]	Compilation	HTC Fuze/ PC (3 GHz 2-core, 4G RAM)	Speech recognition Speech recognition Video game Video game 30-move chess game 30-move chess game	89% (Wi-Fi) 76% (3G) 20% (Wi-Fi) -10% (3G) 18% (Wi-Fi) -27% (3G)	6.55x (Wi-Fi) 4.75x (3G) 3.1x (Wi-Fi) 2.81x (3G) 1.14x (Wi-Fi) 1.03x (3G)
COCA	Compilation	Google Nexus One PC (AMD 3 GHz, 4G RAM)	Chess AI calculation	30% (Wi-Fi) -85.2% (3G)	18.6x (Wi-Fi) 1.59x (3G)

laptops to share their resources through remotely executing processes [20]. Later, Flinn et al. [27], Slingshot, and Spectra all create replicas to benefit clients via caching, remote process execution, or remote procedure call capabilities. The second is to adopt some kind of compilation mechanisms [24], [10]. COCA also belongs to this kind. Kermer et al. builds a toolchain to separate the whole program into PC part and sensor part [24]. MAUI partitions the program into client and server parts and facilitates smartphones by passing several unsupported or heavy tasks to a remote server [10]. Similar to the above two works, COCA accelerates part of the computation on smartphones through offloading some components onto the cloud server.

We observe that in MAUI and COCA, better performance improvements and energy savings can be achieved when Wi-Fi networks are used compared with 3G networks. Furthermore, MAUI was only able to achieve positive energy savings for the speech recognition application, possibly because only this application has high enough compute-to-communication ratio, or that it can benefit from the larger database on the server side. We also note that for the same type of application, Spectra achieves the best performance on both computation speed-up and energy saving, making replica-based approach looking more attractive in applications of the same or similar nature.

Overall, we find that COCA can achieve comparable speed-ups and energy savings with existing approaches when Wi-Fi networks are used. This shows that AOP can provide a not only elegant but also efficient solution to the computation offload problem for smartphones.

VII. CONCLUDING REMARKS

COCA's objective is to offload partial computation from resource-limited devices such as smartphones to cloud servers. COCA provides a source language level mechanism to generate remote objects and local smartphone program automatically, allowing various trade-offs between communication overhead and computation enhancement. To demonstrate COCA's effectiveness, a chess game has been adopted to see how fast it could be, how much energy it could save, and how much delay it could incur. The experiment results show that COCA can indeed provide computation enhancements and energy savings by offloading part of the computation to cloud servers.

ACKNOWLEDGMENT

This research was supported by the M.O.E.A under Domestic Information, Communications Infrastructure Construction Project (No. 100-EC-17-A-05-01-0626), as well as the National Science Council under the Grant NSC 100-2911-I-002-001 and 10R70501.

REFERENCES

[1] http://www.android.com/.

[2] A. Carroll and G. Heiser, "An analysis of power consumption in a smartphone," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*. USENIX Association, 2010, pp. 21–21.

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proceedings of the 11th European Conference on Object Oriented Programming*, 1997, pp. 220–242.

[4] http://en.wikipedia.org/wiki/Aspect-oriented_programming.

[5] R. E. Filman, T. Elrad, S. Clarke, and M. Aksit, *Aspect-oriented software development*. Addison-Wesley Professional, 2004.

[6] http://www.eclipse.org/aspectj/.

[7] X. Gu, K. Nahrstedt, A. Messer, I. Greenberg, and D. Milojicic, "Adaptive offloading inference for delivering applications in pervasive computing environments," in *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*. IEEE, 2003, pp. 107–114.

[8] A. Messer, I. Greenberg, P. Bernadat, D. Milojicic, D. Chen, T. Giuli, and X. Gu, "Towards a distributed platform for resource-constrained devices," in *Proceedings of the 22nd International Conference on Distributed Computing Systems*. IEEE Computer Society, 2002.

[9] S. Ou, K. Yang, and J. Zhang, "An effective offloading middleware for pervasive services on mobile devices," *Pervasive and Mobile Computing*, vol. 3, no. 4, pp. 362–385, 2007.

[10] 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*. ACM, 2010, pp. 49–62.

[11] http://code.google.com/p/apps-for-android/.

[12] http://honzovysachy.sourceforge.net/.

[13] http://www.msoon.com/.

[14] http://ziyang.eecs.umich.edu/projects/powertutor/.

[15] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. Dick, Z. Mao, and L. 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*. ACM, 2010, pp. 105–114.

[16] http://www.realityserver.com/products/realityserver.html.

[17] http://www.otoy.com/#/tech/.

[18] http://www.axceleon.com/.

[19] J. Flinn and M. Satyanarayanan, "Energy-aware adaptation for mobile applications," in *ACM SIGOPS Operating Systems Review*, vol. 33, no. 5. ACM, 1999, pp. 48–63.

[20] A. Rudenko, P. Reiher, G. Popek, and G. Kuenning, "Saving portable computer battery power through remote process execution," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 2, no. 1, pp. 19–26, 1998.

[21] J. Flinn, D. Narayanan, and M. Satyanarayanan, "Self-tuned remote execution for pervasive computing," in *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*. IEEE, 2001, pp. 61–66.

[22] J. Flinn, S. Park, and M. Satyanarayanan, "Balancing performance, energy, and quality in pervasive computing," in *Proceedings of the 22nd International Conference on Distributed Computing Systems*. IEEE, 2002, pp. 217–226.

[23] G. Hunt and M. Scott, "The Coign automatic distributed partitioning system," *Operating Systems Review*, vol. 33, pp. 187–200, 1998.

[24] U. Kremer, J. Hicks, and J. Rehg, "Compiler-directed remote task execution for power management," in *Workshop on Compilers and Operating Systems for Low Power*, 2000.

[25] M. Neubauer and P. Thiemann, "From sequential programs to multi-tier applications by program transformation," in *ACM SIGPLAN Notices*, vol. 40, no. 1. ACM, 2005, pp. 221–232.

[26] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H. Yang, "The case for cyber foraging," in *Proceedings of the 10th workshop on ACM SIGOPS European workshop*. ACM, 2002, pp. 87–92.

[27] J. Flinn, S. Sinnamohideen, N. Tolia, and M. Satyanaryanan, "Data staging on untrusted surrogates," in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*. USENIX Association, 2003, pp. 15–28.

[28] Y. Su and J. Flinn, "Slingshot: Deploying stateful services in wireless hotspots," in *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services*. ACM, 2005, pp. 79–92.

[29] M. Stoer and F. Wagner, "A simple min-cut algorithm," *Journal of the ACM (JACM)*, vol. 44, no. 4, pp. 585–591, 1997.

[30] S. Osman, D. Subhraveti, G. Su, and J. Nieh, "The design and implementation of Zap: A system for migrating computing environments," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 361–376, 2002.

[31] C. Clark, K. Fraser, S. Hand, J. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation—Volume 2*. USENIX Association, 2005, pp. 273–286.

[32] M. Satyanarayanan, B. Gilbert, M. Toups, N. Tolia, D. O'Hallaron, A. Surie, A. Wolbach, J. Harkes, A. Perrig, D. Farber *et al.*, "Pervasive personal computing in an internet suspend/resume system," *IEEE Internet Computing*, pp. 16–25, 2007.

[33] M. Satyanarayanan, V. Bahl, R. Caceres, and N. Davies, "The case for VM-based Cloudlets in mobile computing," *IEEE Pervasive Computing*, 2009.

[34] B. Chun and P. Maniatis, "Augmented smartphone applications through clone cloud execution," in *Proceedings of the 12th Conference on Hot Topics in Operating Systems*. USENIX Association, 2009, p. 8.