ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading

Sokol Kosta
Deustche Telekom Labs
Berlin, Germany
kosta@di.uniroma1.it

Andrius Aucinas
University of Cambridge
Cambridge, UK
aa535@cam.ac.uk

Pan Hui
Deutsche Telekom Labs
Berlin, Germany
pan.hui@telekom.de

Richard Mortier
University of Nottingham
Nottingham, UK
rmm@cs.nott.ac.uk

Xinwen Zhang
Huawei Research Center
Santa Clara, CA, USA
xinwen.zhang@huawei.com

ABSTRACT

Smartphones have exploded in popularity in recent years, becoming ever more sophisticated and capable. As a result, developers worldwide are building increasingly complex applications that require ever increasing amounts of computational power and energy. In this paper we propose *ThinkAir*, a framework that makes it simple for developers to migrate their smartphone applications to the cloud. ThinkAir exploits the concept of smartphone virtualization in the cloud and provides method-level computation offloading. Advancing on previous work, it focuses on the elasticity and scalability of the cloud and enhances the power of mobile cloud computing by parallelizing method execution using multiple virtual machine (VM) images. We implement ThinkAir and evaluate it with a range of benchmarks starting from simple micro-benchmarks to more complex applications. First, we show that the execution time and energy consumption decrease two orders of magnitude for a N-queens puzzle application and one order of magnitude for a face detection and a virus scan application. We then show that a parallelizable application can invoke multiple VMs to execute in the cloud in a seamless and on-demand manner such as to achieve greater reduction on execution time and energy consumption. We finally use a memory-hungry image combiner tool to demonstrate that applications can dynamically request VMs with more computational power in order to meet their computational requirements.

I. INTRODUCTION

Smartphones are becoming increasingly popular, with approximately 550,000 new Android devices being activated worldwide every day [1]. These devices have a wide range of capabilities, typically including GPS, WiFi, cameras, gigabytes of storage, and gigahertz-speed processors. As a result, developers are building ever more complex smartphone applications such as gaming, navigation, video editing, augmented reality, and speech recognition, which require considerable computational power and energy. Unfortunately, as applications become more complex, mobile users have to continually upgrade their hardware to keep pace with increasing performance requirements but still experience short battery lifetime.

Considerable research work have proposed solutions to address the issues of computational power and battery lifetime by offloading computing tasks to cloud. Prominent among those are the MAUI [1] and the CloneCloud [2] projects. MAUI provides method level code offloading based on the .NET framework. However, MAUI work does not address the scaling of execution in cloud. CloneCloud tries to extrapolate the binary pieces of a given process whose execution on the cloud would make the overall process execution faster. It determines these pieces with an offline static analysis of different running conditions of the process binary on both a target smartphone and the cloud. The output of such analysis is then used to build a database of pre-computed partitions of the binary; this is used to determine which parts should be migrated on the cloud. However, this approach only considers limited input/environmental conditions in the offline pre-processing and needs to be bootstrapped for every new application built.

In this paper, we propose *ThinkAir*, a new mobile cloud computing framework which takes the best of the two worlds. ThinkAir addresses MAUI's lack of scalability by creating virtual machines (VMs) of a complete smartphone system on the cloud, and removes the restrictions on applications/inputs/environmental conditions that CloneCloud induces by adopting an online method-level offloading. Moreover, ThinkAir (1) provides an efficient way to perform on-demand resource allocation, and (2) exploits parallelism by dynamically creating, resuming, and destroying VMs in the cloud when needed. To the best of our knowledge, ThinkAir is the first to address these two aspects in mobile clouds. Supporting on-demand resource allocation is critical as mobile users request different computational power based on their workloads and deadlines for tasks, and hence the cloud provider has to dynamically adjust and allocate its resources to satisfy customer expectations. Existing work does not provide any mechanism to support on-demand resource allocation, which is an absolute necessity given the variety of applications that can be run on smartphones, in addition to the high variance of CPU and memory resources these applications could demand. The problem of exploiting parallelism becomes important because mobile applications require increasing amounts of processing power, and parallelization reduces the execution time and energy consumption of these applications with significant margins when compared to non-parallel executions of the same.

ThinkAir achieves all the above mentioned goals by providing a novel execution offloading infrastructure and rich resource consumption profilers to make efficient and effective code migration possible; it further provides library and

[1] Google's 2011 Q2 earnings call

978-1-4673-0775-8/12/$31.00 ©2012 IEEE

compiler support to make it easy for developers to exploit the framework with minimal modification of existing code, and a VM manager and parallel processing module in cloud to manage smartphone VMs as well as automatically split and distribute tasks to multiple VMs.

We now continue by positioning ThinkAir with respect to related work (§II) before outlining the ThinkAir architecture (§III). We then describe the three main components of ThinkAir in more detail: the execution environment (§IV), the application server (§V), and the profilers (§VI). We then evaluate the performance of benchmark applications with ThinkAir (§VII), discuss design limits and future plans (§VIII), and conclude the paper (§IX).

II. RELATED WORK

The basic idea of dynamically switching between (constrained) local and (plentiful) remote resources, often referred as cyber-foraging, has shed light on many research work [3], [4], [5], [6], [7], [8], [9]. These approaches augment the capability of resource-constrained devices by offloading computing tasks to nearby computing resources, or *surrogates*. ThinkAir takes insights and inspirations from these previous systems, and shifts the focus from alleviating memory constraints and provides evaluation on hardware of the time, typically laptops, to more modern smartphones. Furthermore, it enhances computation performance by exploiting parallelism with multiple VM creation on elastic cloud resources and provides a convenient VM management framework for different QoS expectation [10].

Several approaches have been proposed to predict resource consumption of a computing task or method. Narayanan *et al*. [11] use historical application logging data to predict the fidelity of an application, which decides its resource consumption although they only consider selected aspects of device hardware and application inputs. Gurun *et al*. [12] extend the Network Weather Service (NWS) toolkit in grid computing to predict offloading but give less consideration to local device and application profiles.

MAUI [1] describes a system that enables energy-aware offloading of mobile code to infrastructure. Its main aim is to optimize energy consumption of a mobile device, by estimating and trading off the energy consumed by local processing *vs*. transmission of code and data for remote execution. Although it has been found that optimizing for energy consumption often also leads to performance improvement, the decision process in MAUI only considers relatively coarse-grained information, compared with the complex characteristics of mobile environment. MAUI is similar to ThinkAir in that it provides method-level, semi-automatic offloading of code. However, ThinkAir focuses more on scalability issues and parallel execution of offloaded tasks.

More recently, CloneCloud [2] proposes cloud-augmented execution using a cloned VM image as a powerful virtual device. Cloudlets [13], [14] proposes the use of nearby resource-rich computers, to which a smartphone connects over a wireless LAN, with the argument against the use of the cloud due to higher latency and lower bandwidth available when connecting. In essence, Cloudlets makes the use of smartphone simply as a thin-client to access local resources, rather than using the smartphone's capabilities directly and offloading only when required. Paranoid Android [15] uses QEMU to run replica Android images in the cloud to enable multiple exploit and attack detection techniques to run simultaneously with minimal impact on phone performance and battery life. Virtual Smartphone [16] uses Android x86 port to execute Android images in the cloud efficiently on VMWare ESXi virtualization platform, although it does not provide any programmer support for utilizing this facility. ThinkAir shares the same design approach as these of using smartphone VM image inside the cloud for handling computation offloading. Different from them, ThinkAir targets a commercial cloud scenario with multiple mobile users instead of computation offloading of a single user. Hence, we focus not only on the offloading efficiency and convenience for developers, but also on the elasticity and scalability of the cloud side for the dynamic demands of variant customers.

III. DESIGN GOALS AND ARCHITECTURE

The design of ThinkAir is based on some assumptions which we believe are already, or soon will become, true: (1) Mobile broadband connectivity and speeds continue to increase, enabling access to cloud resources with relatively low Round Trip Times (RTTs) and high bandwidths; (2) As mobile device capabilities increase, so do the demands placed upon them by developers, making the cloud an attractive means to provide the necessary resources; and (3) Cloud computing continues to develop, supplying resources to users at low cost and on-demand. We reflect these assumptions in ThinkAir through four key design objectives.

(*i*) *Dynamic adaptation to changing environment*. As one of the main characteristics of mobile computing environment is rapid change, ThinkAir framework should adapt quickly and efficiently as conditions change to achieve high performance as well as to avoid interfering with the correct execution of original software when connectivity is lost.

(*ii*) *Ease of use for developers*. By providing a simple interface for developers, ThinkAir eliminates the risk of misusing the framework and accidentally hurting performance instead of improving it, and allows less skilled and novice developers to use it and increase competition, which is one of the main driving forces in today's mobile application market.

(*iii*) *Performance improvement through cloud computing*. As the main focus of ThinkAir, we aim to improve both computational performance and power efficiency of mobile devices by bridging smartphones to the cloud. If this bridge becomes ubiquitous, it serves as a stepping stone towards more sophisticated software.

(*iv*) *Dynamic scaling of computational power*. To satisfy the customer's performance requirements for commercial grade service, ThinkAir explores the possibility of dynamically scaling the computational power at the server side as well

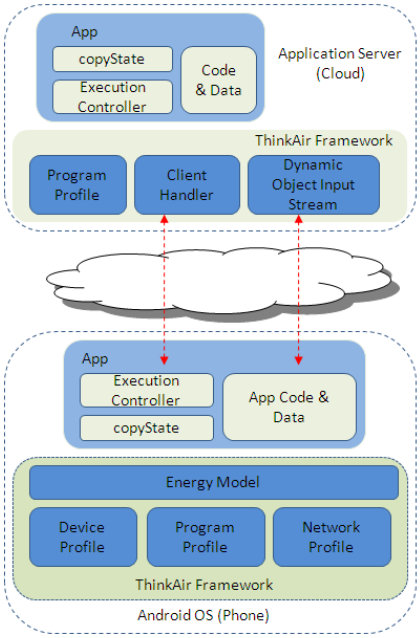

Fig. 1. Overview of the ThinkAir framework.

as parallelizing execution where possible for optimal performance.

The ThinkAir framework consists of three major components: the execution environment (§IV), the application server (§V) and the profilers (§VI). We will now give details of each component of the framework, depicted in Figure 1.

IV. Compilation and Execution

In this section we describe in detail the process by which a developer writes code to make use of ThinkAir, covering the programmer API and the compiler, followed by the execution flow.

A. Programmer API

Since the execution environment is accessed indirectly by the developer, ThinkAir provides a simple library that, coupled with the compiler support, makes the programmer's job very straightforward: any method to be considered for offloading is annotated with `@Remote`.

This simple step provides enough information to enable the ThinkAir code generator to be executed against the modified code. This takes the source file and generates necessary *remoteable* method wrappers and utility functions, making it ready for use with the framework - method invocation is done via the *ExecutionController*, which detects if a given method is a candidate for offloading and handles all the associated profiling, decision making and communication with the application server *without* the developer needing to be aware of the details.

B. Compiler

A key part of the ThinkAir framework, the compiler comes in two parts: the Remoteable Code Generator and the Customized Native Development Kit (NDK). The Remoteable Code Generator is a tool that translates the annotated code as described above. Most current mobile platforms provide support for execution of native code for the performance-critical parts of applications, but cloud execution tends to be on x86 hosts, while most smartphone devices are ARM-based, therefore the Customized NDK exists to provide native code support on the cloud.

C. Execution Controller

The Execution Controller drives the execution of remoteable methods. It decides whether to offload execution of a particular method, or to allow it to continue locally on the phone. The decision depends on data collected about the current environment as well as that learnt from past executions.

When a method is encountered for the first time, it is unknown to the Execution Controller and so the decision is based only on environmental parameters such as network quality: for example, if the connection is of type WiFi, and the quality of connectivity is good, the controller is likely to offload the method. At the same time, the profilers start collecting data. On a low quality connection, however, the method is likely to be executed locally.

If and when the method is encountered subsequently, the decision on where to execute it is based on the method's past invocations, i.e., previous execution time and energy consumed in different scenarios, as well as the current environmental parameters. Additionally, the user can also set a policy according to their needs. We currently define four such policies, combining execution time, energy conservation and cost:

- *Execution time*. Historical execution times are used in conjunction with environmental parameters to prioritise fast execution when offloading, i.e. offloading only if execution time will improve (reduce) no matter the impact on energy consumption.
- *Energy*. Past data on consumed energy is used in conjunction with environmental parameters to prioritize energy conservation when offloading, i.e., offloading only if energy consumption is expected to improve (reduce) no matter the expected impact on performance.
- *Execution time and energy*. Combining the previous two choices, the framework tries to optimize for both fast execution and energy conservation, i.e., offloading only if both the execution time and energy consumption are expected to improve.
- *Execution time, energy and cost*. Using commercial cloud services also implies cost - you pay for as much as you use, therefore offloading decision based on execution time and energy could also be adjusted according to how much a user is prepared to pay for the retained CPU time and battery power.

Clearly more sophisticated policies could be expressed; discovering policies that work well, meeting user desires and expectations is the subject of future work.

D. Execution flow

On the phone, the Execution Controller first starts the profilers to provide data for future invocations. It then decides whether this invocation of the method should be offloaded or not. If not, then the execution continues normally on the phone. If it is, Java reflection is used to do so and the calling object is sent to the application server in the cloud; the phone then waits for results, and any modified local state, to be returned. If the connection fails for any reason during remote execution, the framework falls back to local execution, discarding any data collected by the profiler. At the same time, the Execution Controller initiates asynchronous reconnection to the server. If an exception is thrown during the remote execution of the method, it is passed back in the results and re-thrown on the phone, so as not to change the original flow of control.

In the cloud, the Application Server manages clients that wish to connect to the cloud, which is illustrated in the next section.

V. Application Server

The ThinkAir Application Server manages the cloud side of offloaded code and is deliberately kept lightweight so that it can be easily replicated. It is started automatically when the remote Android OS is booted, and consists of three main parts, described below: a client handler, cloud infrastructure, and an automatic parallelization component.

A. Client Handler

The Client Handler executes the ThinkAir communication protocol, managing connections from clients, receiving and executing offloaded code, and returning results.

To manage client connections, the Client Handler registers when a new application, i.e., a new instance of the ThinkAir Execution Controller, connects. If the client application is unknown to the application server, the Client Handler retrieves the application from the client, and loads any required class definitions and native libraries. It also responds to application-level *ping* messages sent by the Execution Controller to measure connection latency.

Following the initial connection set up, the server waits to receive execution requests from the client. A request consists of necessary data: containing object, requested method, parameter types, parameters themselves, and a possible request for extra computational power. If there is no request for more computational power, the Client Handler proceeds much as the client would: the remoteable method is called using Java reflection and the result, or exception if thrown, is sent back. There are some special cases regarding exception handling in ThinkAir, however. For example, if the exception is an *OutOfMemoryError*, the Client Handler does not send it to the client directly; instead, it dynamically resumes a more powerful clone (a VM), delegates the task to it, waits for the result and sends it back to the client. Similarly, if the client explicitly asks for more computational power, the Client Handler resumes a more powerful clone and delegates the task to it. In the case that the client asks for more clones to execute its task in parallel, the Client Handler resumes needed clones, distributes the task among them, collects and sends results back to the client. Along with the return value, the Client Handler also sends profiling data for future offloading decisions made by the Execution Controller at the client side.

TABLE I
Different configurations of VMs.

Resource	basic	main	large	×2 large	×4 large	×8 large
CPUs	1	1	1	2	4	8
Memory (MB)	200	512	1024	1024	1024	1024
Heap size (MB)	32	100	100	100	100	100

B. Cloud Infrastructure

To make the cloud infrastructure easily maintainable and to keep the execution environment homogeneous, e.g., w.r.t. the Android-specific Java bytecode format, we use a virtualization environment allowing the system to be deployed where needed, whether on a private or commercial cloud. There are many suitable virtualization platforms available, e.g., Xen [17], QEMU [18], and Oracle's VirtualBox. In our evaluation we run the Android x86 port [2] on VirtualBox [3]. To reduce its memory and storage demand, we build a customized version of Android x86, leaving out unnecessary components such as the user interface and built-in standard applications.

Our system has 6 types of VMs with different configurations of CPU and memory to choose from, which are shown in Table I. The VM manager can automatically scale the computational power of the VMs and allocate more than one VMs for a task depending on user requirements. The default setting for computation is only one VM with 1 CPU, 512MB memory, and 100MB heap size, which clones the data and applications of the phone and we call it the *primary server*. The primary server is always online, waiting for the phone to connect to it. The second type of VMs can be of any configuration shown in Table I, which in general does not clone the data and applications of a specific phone and can be allocated to any user on demand - we call them the *secondary servers*. The secondary servers can be in any of these three states: *powered-off*, *paused*, or *running*. When a VM is in powered-off state, it is not allocated any resources. The VM in paused state is allocated the configured amount of memory, but does not consume any CPU cycles. In the running state the VM is allocated the configured amount of memory and also makes use of CPU.

The Client Handler, which is in charge of the connection between the client (phone) and the cloud, runs in the main server. The Client Handler is also in charge of the dynamic control of the number of running secondary servers. For example, if too many secondary VMs are running, it can decide to power-off or pause some of them that are not executing

[2] http://android-x86.org/
[3] http://www.virtualbox.org/

any task. Utilizing different states of the VMs has the benefit of controlling the allocated resources dynamically, but it also has the drawback of introducing latency by resuming, starting, and synchronizing among the VMs. From our experiments, we observe that the average time to resume one VM from the paused state is around 300ms. When the number of VMs to be resumed simultaneously is high (seven in our case), the resume time for some of the VMs can be up to 6 or 7 seconds because of the instant overhead introduced in the cloud. We are working on finding the best approach for removing this simultaneity and staying in the limit of 1s for total resume time. When a VM is in powered-off state, it takes on average 32s to start it, which is very high to use for methods that run in the order of seconds. However, there are tasks that take hours to execute on the phone (e.g., virus scanning), for which it is still reasonable to spend 32s for starting the new VMs. A user may have different QoS requirements (e.g. completion time) for different tasks at different times, therefore the VM manager needs to dynamically allocate the number of VMs to achieve the user expectations.

To make tests consistent, in our environment all the VMs run on the same physical server which is a large multicore system with ample memory to avoid effects of CPU or memory congestion.

C. Automatic Parallelization

Parallel execution can be exploited much more efficiently on the cloud than on a smartphone, either using multiprocessor support or splitting the work among multiple VMs. Our approach for parallelization considers intervals of input values. It is particularly useful in two main types of computationally expensive algorithms:

- Recursive algorithms or ones that can be solved using *Divide-and-Conquer* method. They are often based on constructing a solution when iterating over a range of values of a particular variable, allowing sub-solution computation to be parallelized (e.g. the classic example of 8-queens puzzle, which we present in VII-B).
- Algorithms using a lot of data. For example, a face recognition application requires comparison of a particular face with a large database of pre-analyzed faces, which can be done on a distributed database on the cloud much more easily than just on a phone. Again, this allows to split computations based on the intervals of data to be analyzed on each clone.

We provide results of such parallelization in the evaluation section (§VII).

VI. Profiling

Profilers are a critical part of the ThinkAir framework: the more accurate and lightweight they are, the more correct offloading decisions can be made, and the lower overhead is introduced. The profiler subsystem is highly modular so that it is straightforward to add new profilers. The current implementation of ThinkAir includes three profilers (hardware, software, and network), which collect variant data and feed into the energy estimation model.

For efficiency we use Android *intents* to keep track of important environmental parameters which do not depend on program execution. Specifically, we register listeners with the system to track battery levels, data connectivity presence, and connection types (WiFi, cellular, etc.) and subtypes (GPRS, UMTS, etc.). This ensures that the framework does not spend extra time and energy polling the state of these factors.

A. Hardware Profiler

The Hardware Profiler feeds hardware state information into the energy estimation model, which is considered when an offloading decision is made. In particular, CPU and screen have to be monitored [4]. We also monitor the WiFi and 3G interfaces. The following states are monitored by the Hardware Profiler in current ThinkAir framework.

- *CPU*. The CPU can be *idle* or have a utilization from 1–100% as well as different frequencies;
- *Screen*. The LCD screen has a brightness level between 0–255;
- *WiFi*. The power state of WiFi interface is either *low* or *high*;
- *3G*. The 3G radio can be either *idle*, or in use with a *shared* or *dedicated* channel.

B. Software Profiler

The Software Profiler tracks a large number of parameters concerning program execution. After starting executing a remoteable method, whether locally or remotely, the Software Profiler uses the standard Android Debug API to record the following information.

- Overall execution time of the method;
- Thread CPU time of the method, to discount the affect of pre-emption by another process;
- Number of instructions executed [5];
- Number of method calls;
- Thread memory allocation size;
- Garbage Collector invocation count, both for the current thread and globally.

C. Network Profiler

This is probably the most complex profiler as it takes into account many different sets of parameters, by combining both intent and instrumentation-based profiling. The former allows us to track the network state so that we can e.g., easily initiate re-estimation of some of the parameters such as RTT on network status change. The latter involves measuring the network RTT as well as the amount of data that ThinkAir sends/receives in a time interval, which are used to estimate the *perceived network bandwidth*. This includes the overheads

[4]We consider that simply turning off the screen during offloading would be too intrusive to users.

[5]This requires an adaptation of the distributed kernel due to what we believe is a bug in the OS using cascading profilers leading to inconsistent results and program crashes.

of serialization during transmission, allowing more accurate offloading decisions to be taken.

In addition, the Network Profiler tracks several other parameters for the WiFi and 3G interfaces, including the number of packets transmitted and received per second, uplink channel rate and uplink data rate for the WiFi interface, and receiving and transmitting data rate for the 3G interface. These measurements enable better estimation of the current network performance being achieved.

D. Energy Estimation Model

A key parameter for offloading policies in ThinkAir is the effect on energy consumption. This requires dynamically estimating the energy consumed by methods during execution. We take inspiration from the recent PowerTutor [19] model which accounts for power consumption of CPU, LCD screen, GPS, WiFi, 3G, and audio interfaces on HTC Dream and HTC Magic phones. PowerTutor indicates that the variation of estimated power on different types of phones is very high, and presents a detailed model for the HTC Dream phone which is used in our experiments. We modify the original PowerTutor model to accommodate the fact that certain components such as GPS and audio have to operate locally and cannot be migrated to the cloud. By measuring the power consumption of the phone under different cross products of the extreme power states, PowerTutor model further indicates that the maximum error is 6.27% if the individual components are measured independently. This suggests that the sum of independent component-specific power estimates is sufficient to estimate overall system power consumption.

Using this approach we devise a method with only minor deviations from the results obtained by PowerTutor. We implement this energy estimation model inside the ThinkAir Energy Profiler and use it to dynamically estimate the energy consumption of each running method.

VII. Evaluation

We evaluate ThinkAir using two sets of experiments. The first is adapted from the Great Computer Language Shootout [6], which was originally used to perform a simple comparison of Java *vs*. C++ performance, and therefore serves as a simple set of benchmarks comparing local *vs*. remote execution. The second set of experiments uses four applications for a more realistic evaluation: an instance of the N-queens problem, a face detection program, a virus scanning application, and an image merging application.

To evaluate, we define the *boundary input value* (BIV) as the minimum value of the input parameter for which offloading would give a benefit. We use the *execution time policy* throughout, so for example when running *Fibonacci(n)* under the execution time profile, we find a BIV of 18 when the phone connects to the cloud through WiFi, i.e., the execution of *Fibonacci(n)* is faster when offloaded for $n \geq 18$ (cf. Table II). We run the experiments under four different connectivity scenarios as follows.

[6]http://kano.net/javabench/

TABLE II
Boundary input values of benchmark applications, with WiFi and 3G connectivity, and the complexity of algorithms.

Benchmark	BIV		Complexity	Data (bytes)	
	WiFi	3G		Tx	Rx
Fibonacci	18	19	$O(2^n)$	392	307
Hash	550	600	$O(n^2 log(n))$	383	293
Methcall	2500	3100	$O(n)$	338	297
Nestedloop	7	8	$O(n^6)$	349	305

- *Phone*. Everything is executed on the phone;
- *WiFi-Local*. The phone directly connects to a WiFi router attached to the cloud server via WiFi link;
- *WiFi-Internet*. The phone connects to the cloud server using a normal WiFi access point via the Internet;
- *3G*. The phone is connected to the cloud using 3G data network.

Every result is obtained by running the program 20 times for each scenario and averaging; there is a pause of 30 seconds between two consecutive executions. The typical RTT of the 3G network that we use for the experiments is around 100ms and that for the WiFi-Local is around 5ms. In order to test the performance of ThinkAir with different quality of WiFi connection, we use both a very good dedicated residential WiFi connection (RTT 50ms) and a commercial WiFi hotspot shared by multiple users (RTT 200ms), which the device may encounter on the move, for the WiFi-Internet setting. We do not find any significant difference for these two cases, and hence we simplify them to a single case except for the full application evaluations.

A. Micro-benchmarks

Originally used for a simple Java *vs*. C++ comparison, each of these benchmarks depends only on a single input parameter, allowing for easier analysis. Our results are shown in Table II. We find that, especially for operations where little data needs to be transmitted, network latency clearly affects the boundary value, hence the difference between boundary values in the case of WiFi and 3G network connectivity. This effect is also noted by Cloudlets [13]. We also include computational complexity of the core parts of the different benchmarks to show that, with growing input values, ThinkAir becomes more efficient. Note that there are large constant factors hidden by the O notation, hence the different BIVs with the same complexity.

B. Application benchmarks

We consider four complete benchmark applications representing more complex and compute intensive applications: a solver for the classic N-Queens problem, a face detection application, a Virus scanning application, and an application combining two pictures into an unique large one.

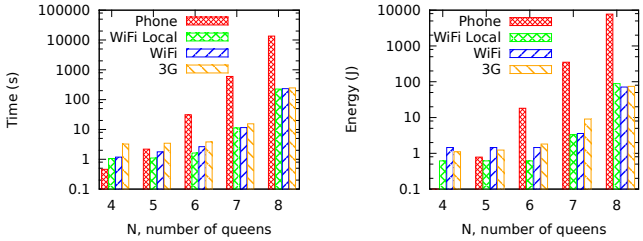

Fig. 2. Execution time and energy consumption of the N-queens puzzle, $N = \{4, 5, 6, 7, 8\}$.

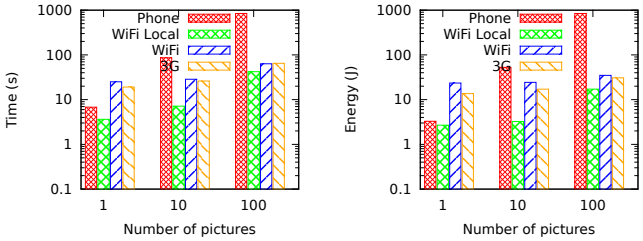

Fig. 4. Execution time and energy consumed for the face detection experiments.

1) N-Queens Puzzle: In this application, we implement the algorithm to find all solutions for the N-Queens Puzzle and return the number of solutions found. We consider $4 \leq N \leq 8$ since at $N = 8$ the problem becomes very computationally expensive as there are 4,426,165,368 (i.e., 64 choose 8) possible arrangements of eight queens on a 8×8 board, but only 92 solutions. We apply a simple heuristic approach to constrain each queen to a single column or row. Although this is still considered as a brute force approach, it reduces the number of possibilities to just $8^8 = 16,777,216$. Figure 2 shows that for $N = 8$, the execution on the phone is unrealistic as it takes hours to finish. However, we consider the problem a suitable benchmark because many real problems get solved in a brute-force fashion.

Figure 2 shows the time taken and the energy consumed. We notice that the BIV is between 5: for higher N, both the time taken and energy consumed in the cloud are less than that on the phone. In general, WiFi-Local is the most efficient offload method as N increases, probably because the higher bandwidths lead to lower total network costs. Ultimately though, computation costs come to dominate in all cases.

Figure 3 breaks down the energy consumption between components for $N = 8$. As expected, when executing locally on the phone, the energy is consumed by the CPU and the screen: the screen is set to 100% brightness and the CPU runs at the highest possible frequency. When offloading, some energy is consumed by the use of the radio, with a slightly higher amount for 3G than WiFi. The difference in CPU energy consumed between WiFi and WiFi-Local is due to the difference of the CPU speed of the local device and cloud servers.

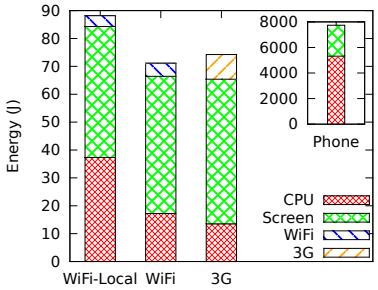

Fig. 3. Energy consumed by each component when solving 8-queens puzzle in different scenarios.

2) Face Detection: We port a third party program [7] towards a simple face detection program that counts the number of faces in a picture and computes simple metrics for each detected face (e.g., distance between eyes). This demonstrates that it is straightforward to apply the ThinkAir framework to existing code. The actual detection of faces uses the Android API `FaceDetector`, so this is an Android optimized program and should be fast even on the phone. We consider one run with just a single photo and runs with comparing the photo against multiple (10 and 100) others, which have previously been loaded into the cloud e.g., comparing against photos from a user's Flickr account. When running over multiple photos, we use the return values of the detected faces to determine if the initial single photo is duplicated within the set. In all cases, the execution time and energy consumed are much lower than that in the cloud.

Figure 4 shows the results for the face detection experiments. The single photo case actually runs faster on the phone than offloading if the connectivity is not the best: it is a native API call on the phone and quite efficient. However, as the number of photos being processed increases, and in any case when the connectivity has sufficiently high bandwidth and low latency, the cloud proves more efficiency. Figure 5 shows the breakdown of the energy consumed among components. Similar to the 8-Queens experiment results shown in Figure 3, the increased power of the cloud server makes the offloaded cases dramatically more efficient than that when everything is run locally on the phone.

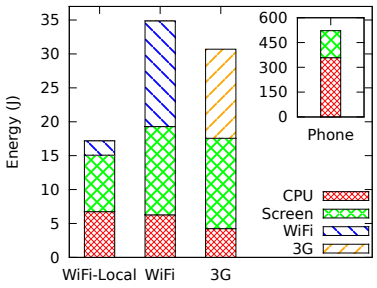

Fig. 5. Energy consumed by each component for face detection with 100 pictures in different scenarios.

3) Virus Scanning: We implement a virus detection algorithm for Android, which takes a database of 1000 virus signatures and the path to scan, and returns the number of viruses found. In our experiments, the total size of files in the directory is 10MB, and the number of files is around 3,500. Figure 6 shows that the execution on the phone takes more than one hour to finish, while less than three minutes if offloaded. As the the data sent for offloading is larger compared to previous ones, the comparison of the energy consumed by the WiFi and 3G is more fair. As a result we observe that WiFi is less energy efficient per bit transmitted than 3G, which is also supported by the face detection experiment (Figure 5). Another

[7] http://www.anddev.org/quick_and_easy_facedetector_demo-t3856.html

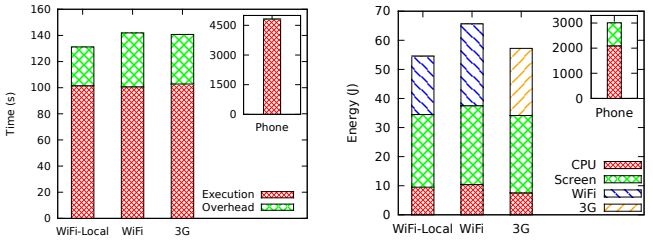

Fig. 6. Execution time and energy consumption of the virus scanning in different scenarios.

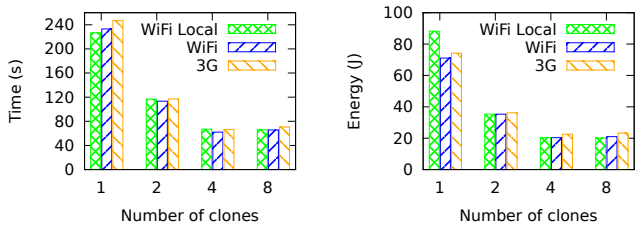

Fig. 7. Time taken and energy consumed on the phone executing 8-queens puzzle using $N = \{1, 2, 4, 8\}$ servers.

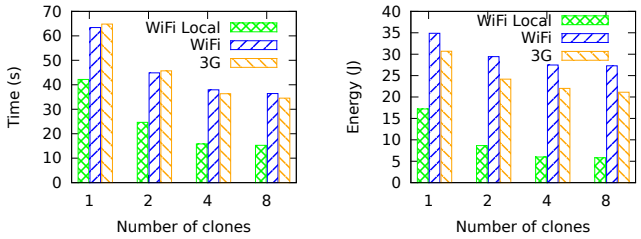

Fig. 8. Time taken and energy consumed for face detection on 100 pictures using $N = \{1, 2, 4, 8\}$ servers.

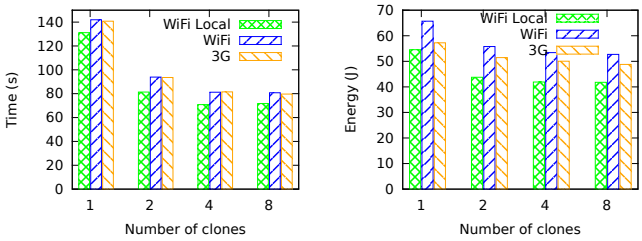

Fig. 9. Time taken and energy consumed for virus scanning using $N = \{1, 2, 4, 8\}$ servers.

interesting observation is related to the energy consumed by the CPU. In fact, from the results of all the experiments we observe that the energy consumed by the CPU is lower when offloading using 3G instead of WiFi.

4) Images Combiner: The intention of this application is to address the problem that some applications cannot be run on the phone due to lack of resources other than CPU, such as, the Java VM heap size is a big constraint for Android phones. If one application exceeds 16MB [8] of the allocated heap, it throws an `OutOfMemoryError` exception [9]. Working with bitmaps in Android can be a problem if programmers do not pay attention to memory usage. In fact, our application is a naïve implementation of combining two images into a bigger one. The application takes two images of size (w_1, h_1), (w_2, h_2) as input, allocates memory for another image of size $(\max\{w_1, w_2\}, \max\{h_1, h_2\})$, and copies the content of each original image into the final one. The problem here arises when the application tries to allocate memory for the final image, resulting in `OutOfMemoryError` and making the execution aborted. We are able to circumvent this problem by offloading the images to the cloud clone and explicitly asking for high VM heap size. First, the clone tries to execute the algorithm. When it does not have enough free VM heap size the execution fails with `OutOfMemoryError`. It then resumes a more powerful clone and delegates the job to it. In the meantime, the application running on the phone frees the memory occupied by the original images, and waits for the final results from the cloud.

C. Parallelization with Multiple VM Clones

In previous subsection we have showed that the ThinkAir framework can scale the processing power up by resuming more powerful clones and delegating task to them. Another way of achieving the scaling of the processing power of a client is to exploit parallel execution. We have previously described how we expect to split parallelizable applications to tasks by using intervals of input parameters. In this section, we discuss the performance of three representative applications, 8-Queens, Face Detection with 100 pictures, and Virus Scanner, using multiple cloud VM clones.

[8]http://developer.android.com/reference/android/app/ActivityManager.html#getMemoryClass

[9]The maximum heap size can be configured from the phone producers, so it can be different from 16MB, which is the default on the Android API

Our experiment is setup as follow. A single primary server communicates with the client and k secondary clones, where $k \in \{1, 3, 7\}$. When the client connects to the cloud, it communicates with the primary server which in turn manages the secondaries, informing them that a new client has connected. All interactions between the client and the primary are as usual, but now the primary behaves as a (transparent) proxy for the secondaries, incurring extra synchronization overheads. Usually the secondary clones are kept in pause state to minimize the resources allocated. Every time when the client asks for service requiring more than one clone, the primary server resumes the needed number of secondary clones. After the secondaries finish their jobs, they are paused again by the primary server.

The modular architecture of the ThinkAir framework allows programmers to implement many parallel algorithms with no modification to the ThinkAir code. In our experiments, as the tasks are highly parallelizable, we evenly divide them among the secondaries.

In the 8-Queens puzzle case, the problem is split by allocating different regions of the board to different clones and combining the results. For the face detection problem, the

100 photos are simply distributed among the secondaries for duplicates detection. In the same way, the files to be scanned for virus signatures are distributed among the clones and each clone runs the virus scanning algorithm on the files allocated. In all experiments, the secondary clones are resumed from the paused state, and the resume time is included in the overhead time, which in turn is included in the execution time.

Figure 7, 8, and 9 show the performance of the applications as the number of clones increases. In all 3 applications, the 4-clone case obtains the most performance benefits, since synchronization overheads start to outweigh the running costs as the regions which the board has been divided to become very small. We can also see that the increased input size makes the WiFi less efficient in terms of energy compared to 3G, which again supports our previous observations.

VIII. Discussion

ThinkAir currently employs a conservative approach for data transmissions, which is obviously suboptimal as not all instance object fields are accessed in every method and so do not generally need to be sent. We are currently working on improving the efficiency of data transfer for remote code execution, combining static code analysis with data caching. The former eliminates the need to send and receive data that is not accessed by the cloud. The latter ensures that unchanged values need not be sent, in either direction, repeatedly. This could be further combined with speculative execution to explore alternative execution paths for improved caching. Note that these optimization would need to be carefully applied however, as storing the data between calls and checking for changes has large overheads on its own.

ThinkAir assumes a trustworthy cloud server execution environment: when a method is offloaded to the cloud, the code and state data are not maliciously modified or stolen. We also currently assume that the remote server faithfully loads and executes any code received from clients although we are currently working on integrating a lightweight authentication mechanism into the application registration process. For example, a device agent can provide UI for the mobile user to register the ThinkAir service before she can use the service, generating a shared secret based on user account or device identity.

Privacy-sensitive applications may need more security requirements than authentication. For example, if a method executed in cloud needs private data from the device, e.g., location information or user profile data, its confidentiality needs to be protected during transmission. We plan to extend our compiler to support `SecureRemoteable` class to support these security properties automatically and release the burden from application developers.

IX. Conclusions

We present ThinkAir, a framework for offloading mobile computation to the cloud. Using ThinkAir requires only simple modifications to an application's source code coupled with use of our ThinkAir tool-chain. Experiments and evaluations with micro benchmarks and computation intensive applications demonstrate the benefits of ThinkAir for profiling and code offloading, as well as accommodating changing computational requirements with the ability of on-demand VM resource scaling and exploiting parallelism. We are continuing the development of several key components of ThinkAir: we have ported Android to Xen allowing it to be run on commercial cloud infrastructure, and we continue to work on improving programmer support for parallelizable applications. Furthermore, we see improving application parallelization support as a key direction to use the capabilities of distributed computing of the cloud.

References

[1] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. MAUI: making smartphones last longer with code offload. In *MobiSys*, 2010.

[2] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: Elastic execution between mobile device and cloud. In *Proc. of EuroSys*, 2011.

[3] J.P. Sousa and D. Garlan. Aura: an architectural framework for user mobility in ubiquitous computing environments. In *Proc. of Working IEEE/IFIP Conference on Software Architecture*, 2002.

[4] Rajesh Krishna Balan, Mahadev Satyanarayanan, SoYoung Park, and Tadashi Okoshi. Tactics-based remote execution for mobile computing. In *Proc. of Mobisys*, 2003.

[5] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H. Yang. The case for cyber foraging. In *Proc. of ACM SIGOPS European Workshop*, 2002.

[6] B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, and K. Walker. Agile application-aware adaptation for mobility. In *Proc. of SOSP*, 1997.

[7] O. Riva J. Porras and M. D. Kristensen. *Dynamic Resource Management and Cyber Foraging*, chapter Middleware for Network Eccentric and Mobile Applications. Springer Press, 2008.

[8] Andres Lagar-Cavilla Niraj, Niraj Tolia, Rajesh Balan, Eyal De Lara, and M. Satyanarayanan. Dimorphic computing. Technical report, 2006.

[9] Moo-Ryong Ra, Anmol Sheth, Lily Mummert, Padmanabhan Pillai, David Wetherall, and Ramesh Govindan. Odessa: enabling interactive perception applications on mobile devices. In *Proc. of ACM MobiSys*, 2011.

[10] M. Armbrust et al. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[11] D. Narayanan, J. Flinn, and M. Satyanarayanan. Using history to improve mobile application adaptation. In *Proc. of Mobisys*, 2000.

[12] S. Gurun, C. Krintz, and R. Wolski. Nwslite: A light-weight prediction utility for mobile devices. In *Proc. of Mobisys*, 2004.

[13] Mahadev Satyanarayanan, Paramvir Bahl, Ramon Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 2009.

[14] Adam Wolbach, Jan Harkes, Srinivas Chellappa, and M. Satyanarayanan. Transient customization of mobile computing infrastructure. In *MobiVirt*, 2008.

[15] Georgios Portokalidis, Philip Homburg, Kostas Anagnostakis, and Herbert Bos. Paranoid android: Versatile protection for smartphones. In *ACSAC*, 2010.

[16] Eric Y. Chen and Mistutaka Itoh. Virtual smartphone over IP. In *IEEE WoWMoM*, 2010.

[17] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM SOSP*, 2003.

[18] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proc. of Usenix ATC*, 2005.

[19] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. Mao, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proc. Int. Conf. Hardware/Software Codesign and System Synthesis*, 2010.