Dynamic Software Deployment from Clouds to Mobile Devices

Ioana Giurgiu, Oriana Riva, Gustavo Alonso

Dynamic software deployment from clouds to mobile devices

Ioana Giurgiu[1], Oriana Riva[2⋆], and Gustavo Alonso[1]

[1] Systems Group, Dept. of Computer Science, ETH Zurich
[2] Microsoft Research, Redmond

Abstract. With the functionality of mobile applications ever increasing, designers are often confronted with either the resource limitations of the devices or of the network. As pointed out by recent work, application partitioning between mobile devices and clouds, can be used to solve some of these issues, improving performance and/or battery life. In this paper, we argue that the static decisions made in existing work cannot leverage the full potential of application partitioning. Thus, to allow for variations in the execution environment, we have developed a system that dynamically adapts the application partition decisions. The system works by continuously profiling an applications performance and dynamically updating its distributed deployment to accommodate changes in the network bandwidth, devices CPU utilization, and data loads. Using several real applications, we show that our approach provides performance gains as high as 75% over traditional approaches and achieves lower power consumption by a factor close to 45%.

Keywords: Mobile cloud computing, dynamic distribution, modularity

1 Introduction

Today's mobile users demand increasingly ubiquitous applications and ever richer functionality on their devices. They want to create panoramas from photo collections, manage their finances, and even run augmented reality or data analytics applications while interacting spontaneously and expecting fast response times. These demands and expectations create a complex design problem. On the one hand, running the applications entirely on the mobile is limited by the computational resources of the devices. On the other hand, running the applications remotely is limited by the network bandwidth and often raises usability issues due to varying latency. Thus, recent research efforts have proposed to offload parts of an application from the mobile device to the cloud [6,8,9,11,26], thereby demonstrating important gains in battery life and performance. Code offloading raises two important questions: *what* and *when* to migrate for remote execution. While most techniques exclusively focus on what to offload, by making offline partitioning decisions, we advocate that understanding when it becomes beneficial to offload code is just as important. Changes in the network bandwidth or latency,

⋆ Work done while being at ETH Zurich

sudden increases of the CPU load on the mobile device, and variations in the user's inputs during interactions can dramatically impact the performance and responsiveness of most applications, an aspect often ignored in existing work.

Consider an example from furniture houses where computer-based applications can help customers visualize the possible arrangement of furniture items in their homes. Static approaches would store the furniture catalog and perform the image rendering remotely, independent of any changes in environmental factors. However, one can easily imagine situations in which varying network conditions result in significantly slower application responsiveness (e.g.,, due to a drop in available bandwidth). In such scenarios, an adaptive system would recognize that the network is the bottleneck and not the device's CPU, and would promptly limit data transfers and move more computation to the mobile device. A similar decision can be made based on the amount of data involved, something that depends on what the user wants to upload in every interaction. There will always be situations where static partitioning has chosen the wrong configuration.

In this paper, we address the challenges of (1) what parts of an application to offload and (2) when, by considering the changing conditions one is likely to encounter when operating with mobile devices. Our system explores an adaptive deployment model where the cloud moves part of the application to the mobile device to improve user experience and minimize data transfers. To ensure high flexibility in what application parts to offload, we assume applications are modularized. Writing modular applications is already a well-established practice with increasing software support [10,19,23] and various projects recognize the benefits of decoupling an application's functionalities into pluggable modules [4, 12, 27]. Thus, given a modular application, we deliver an automatic pipeline of operations that optimally partitions it on-the-fly between the cloud and the mobile device according to the device's CPU load, network conditions, or user inputs. Full automation is key to improve user experience and to ensure the user does not have to be involved in what are complex architectural decisions. Thus, the dynamic aspects of our system guarantee that on-the-fly acquisition of an application does not result in unacceptable delays. Additionally, we introduce a novel mechanism to allow devices to autonomously and dynamically adjust an application configuration based on the user's inputs.

Our system runs on Android [3] and Amazon EC2 instances [1]. It was evaluated with three applications: a service for ticket purchase, an indoor localization application and a text-to-speech synthesizer. In all cases, for small, medium and large EC2 instances, we observe significant gains (i.e., reduced interaction time by up to 75% and lower power consumption by up to 45%), while considering all data and code migration costs. The system dynamically adapts to changes in the data load or the execution environment, by promptly finding and switching to the optimal configuration. An additional benefit of our approach is that applications that could not otherwise be run on the mobile device (except maybe for very small data loads), execute successfully for all data inputs, while minimizing the overall interaction time. An example is FreeTTS [13], a text-to-speech synthesizer application which we used in the evaluation. If running entirely on the mobile device, FreeTTS works up to a maximum input of only 5 KB of

text, showing after that an exponential increase in the execution time. We show that with our technique this restriction does no longer hold and the application performance is significantly improved.

The rest of the paper is organized as follows. In the next section, we discuss related work. Section 3 describes the system's goals and design principles, while Section 4 gives insights on its implementation. In Section 5, we describe our applications and present results in Section 6. Finally, we conclude in Section 7.

2 Related work

An increasing amount of work is being done in the context of application partitioning and offloading to remote servers or the cloud. However, most systems tackle the static problem, that of making partitioning decisions before an application interaction is initiated and without readjusting the offloading scheme at runtime. More recently, the dynamic aspect has gained more attention and several approaches have emerged [7–9, 22] although they are all based on very different premises and present different limitations.

MAUI [9] and CloneCloud [7, 8] aim at improving the performance and battery life on the mobile device by offloading application state to either remote servers or cloud clones. Both require the application to be pre-installed at the device, which creates problems with the number of platforms to be supported and as software evolves. Our system provides on-demand installation, which removes the need of having the software pre-installed and makes it significantly easier to evolve the application. Furthermore, MAUI's offloading unit (i.e., method) is finer-grained compared to ours (i.e., OSGI modules [19]). Thus it becomes unfeasible for applications with more than tens of methods, since their algorithm requires exponential time to traverse the entire search space. Finally, although MAUI can react to CPU or network changes, it cannot adapt to varying user inputs. Our approach uses a caching algorithm to solve this problem.

CloneCloud shows the effectiveness of static analysis of Java code to manage dynamic offloading. However, their evaluation shows significant gains only for large inputs, i.e.,, 100 photos, as only then the achieved speedup on the clones becomes significant. A serious drawback is that the gains observed do not consider the bandwidth cost. CloneCloud assumes that the device and remote server have fully synchronized file systems and removes the cost of such synchronization from the measurements. As soon as dynamic data is involved the observed cost in battery and performance is likely to be dominated by the data transfer. In our scenarios, we consider the cost of data transfers an integral part of the problem. Thus, we account for the data migration overhead and observe significant performance improvements by doing so even with modest amounts of data involved. Odessa [22] has also recognized the need to dynamically adjust offloading decisions, and proposes a technique to structure the parallelism across mobile devices and remote servers for streaming applications. More recently, [17] has proposed a fault-tolerant approach to save energy on mobile devices by server offloading without partitioning. The application is present at both ends and only state is migrated to switch from local to remote executions. State migration, however, has the same problems as data migration as the overhead typically comes

from the user data and cannot be ignored. Therefore, most of these dynamic approaches offload the application state only, while ignoring the hurdles of both code and data migration. We argue these problems are essential and need to be addressed, thus our system provides support to offload code and data on-the-fly.

Static approaches have been proposed in the context of "cyber foraging"' [5, 25]. Spectra [5, 11] and Chroma [6] partition applications into local and remoteable tasks, pre-installed on surrogates. Task partition is based on manually specified execution plans. Other systems use virtual machine techniques to increase flexibility. Slingshot [28] and Goyal and Carter's prototype [15] allow users to install their own functionality on surrogates. These systems are different from ours in that they rely on the developer to manage the partitioning process and require application pre-installation. In the context of program partitioning, Coign [16] provides static partitioning of COM components, while Wishbone [18] and Abacus [2] focus on partitioning either stream or data-intensive applications. However, none of these systems readjust their partitioning decision at runtime. Other work tackled the migration of Java applications [20] remotely. In addition to their static approach, their offloading unit (i.e., Java classes) is unsuitable for large applications. Other systems have treated applications as three-tier structures [29–31] to simplify partitioning. Although they put little burden on the programmer, there is no support for dynamic migration of components.

3 System overview

Our system's goal is to make cloud applications not originally designed for mobile platforms capable of running on mobile devices in a resource-efficient manner, while maintaining high performance under dynamic conditions. In order to provide an improved user experience for a wide range of applications (with long- and short-term interactions), our approach addresses several requirements.

On-the-fly application installation and updates. In practice, it is not possible to assume that a device has all necessary applications pre-installed. Moreover, for cloud providers it is important to reduce the data transfer to its clients at installation time and provide support for versioned updates. Our system eliminates full code pre-installation and enables application updates at runtime.

Dynamic and optimal application partitioning. The decision on how to distribute an application between the cloud and the mobile device is not obvious, but highly application- and platform-specific. Moreover, mobile devices can experience changes in connectivity due to mobility and network instability, as well as variations in the application load (both in CPU and data transfer) due to multiple concurrently-running applications. Our system considers an application's structure, resource requirements and device constraints to identify its best mobile-cloud partition and adjust it online. In addition, it reconfigures the current application deployment without interrupting ongoing interactions. In AlfredO, an *optimal* partitioning is the application distribution that results in the lowest interaction time. It is equivalent to the graph cutting problem and can be solved with linear programming, as described later in Section 3.3.

Adaptation to varying data inputs. The number and size of data inputs (e.g., size of images to process, length of text to synthesize, etc.) can impact an ap-

plication's execution time, and thus its optimal partitioning between client and cloud. As user inputs cannot be easily predicted, it is hard to know a priori which partitioning configuration suits best a particular user interaction. Rather than deferring the partitioning decision to the cloud side, our system allows clients to autonomously decide which configuration to adopt once the user inputs have been submitted to the application.

The system builds on top of our previous work [14,24], where we tackled the problem of static code offloading, based on offline profiling of applications.

3.1 Architecture

To benefit from our model, applications must be built in a modular fashion, where ideally *modules* contain highly-cohesive functionalities and communicate through low-coupled *dependencies*. The steps taken to distribute a modular application between the cloud and the mobile device are shown in Fig. 1. First, on the cloud, the *application profiler* instruments the application to extract a compact description of its modular structure, as well as CPU and communication statistics (step 1a). On the client, the *mobile device's profiler* collects measurements of the CPU load, network status and available storage space (step 1b). Both profilers submit this information to the *graph generator* component, which uses these measurements to generate a compact specification of the application and environment, in the form of a resource consumption graph (step 2). Based on this description, the optimizer identifies the best distribution of modules and configures the deployment accordingly (step 3a). For different simulations of the user inputs, the optimizer computes asynchronously the most suitable partitions and caches them on the mobile device (step 3b).

At bootstrap, our system offloads the minimum functionality required to start the application on the mobile device. Once the first code migration phase has completed and the acquired components are active, the user can start using the application. At runtime, due to different user inputs, fluctuations in the network connectivity, or changes in load on the mobile device (e.g., users switch from WiFi to 3G, move to low bandwidth areas or increase the device's CPU load by starting more applications) the profilers and optimizer are constantly running such that the partitioning configuration can be changed on the fly.

Fig. 1 also shows that our system runs on top of the R-OSGi [23] and OSGi [19] platforms, that provide module management and remote communication capabilities across application modules.

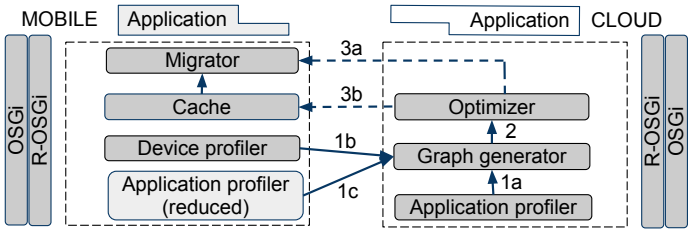


Fig. 1. Architecture and pipeline of operators

3.2 Code pre-installation and updates

Our approach removes the need to pre-install an application on the mobile device before interaction and provides users with on-the-fly installations and updates. This flexibility comes from the modular nature of our design principles (i.e., supported by the OSGI management system our system relies on), and proves to be not only efficient, but also convenient for both cloud providers and clients.

On-demand migration, especially for large applications, is significantly more efficient than full pre-installation. For example, FreeTTS [13], used in our evaluation, has a small code base of 4 MB. If application updates are constantly made available, with a code pre-installation approach the application has to be fully downloaded each time, as versioning is not supported. With AlfredO, only the code necessary to enable the interaction is fetched on the mobile device, merely 260 KB for this application. Once one starts using the application, our system acquires on-the-fly the code necessary to provide users with an optimal interaction. Moreover, by supporting versioning, when code updates are available, AlfredO fetches only the newly modified modules. Additionally, modularity allows us to naturally foster the deployment of applications that contain critical or security restricted pieces of code (i.e., banking). With our system, only those components that have no privacy issues can be installed on the client. This means that software or service providers can still benefit from increased security, while improving user experience on the device.

3.3 Optimal and dynamic application partitioning

Our system partitions applications between the cloud and mobile device while optimizing interaction time and bandwidth utilization. We describe how the optimal partition of an application is identified and how is adapted at runtime.

Application instrumentation and profiling. The profiler is responsible for characterizing the structure and behaviour of a given cloud application, as well as collecting measurements at every user interaction. First, it extracts the inter-module dependencies which have a direct impact on bootstrapping and executing the application, as well as the partitioning decision. Dependencies impose the order in which modules need to be started and restrict their location. The more dependencies a module has, the more expensive its remote invocations become if moved to the mobile device. Second, for each module, the profiler measures its code size, the amount of sent and received data, and its execution time.

From the network perspective, mobile devices connect to the outside world through 3G or WiFi, if available. The differences in their data rates are well known, with a theoretical maximum below 14 Mbps for 3G (HSDPA) and 54 Mbps for WiFi (802.11g). In practice, the gap is much higher and mobility makes network conditions even more unstable. Thus, the profiler monitors on the device which network interface is currently in use and what are the bandwidth and latency on the link. Section 4 provides more details on the profiling step.

Application and network specification. The profiled data is used to provide a compact description of each application module. A *module* is a logical unit encompassing one or more application functions. An example of module specification provided by the profiler is the following:

```
module 'mPayment' {
   deps: [mRSA, mBank]; type: [nIO, movable]; CPU: 168ms;  size: 17kB}
wire 'wPR': ('mPayment', 'mRSA', 50kB)
wire 'wPB': ('mPayment', 'mBank', 7kB)
network: ('WiFi', 6Mbps, 87ms)
```

A module, such as the *mPayment* used in the ticket machine application, has a number of dependencies (*deps*) on other modules (i.e., it uses functions provided by such modules). In this case, *mPayment* depends on *mRSA* and *mBank*. Its *type* property specifies whether it communicates with components outside the application (type=IO) or only with internal modules (type=nIO). In addition, as not all modules can be migrated to a mobile device (e.g., due to privacy issues, database management), they are also classified into *movable* and *non-movable*. Some of these properties can be automatically extracted by processing a module's dependencies (e.g., database connections), but, if not obvious, it is also possible for the developer to manually annotate them. *CPU* and *size* report the average execution time for such module and its code size. A *wire* specifies how much data is transferred between two inter-connected modules, while *network* specifies the type of network connection the device is currently using (WiFi or 3G), as well as its measured bandwidth and latency.

Application optimization. Based on the application and network specifications, the optimizer decides how to partition the application while minimizing the overall interaction time and respecting device-specific constraints, such as maximum storage space available for installing modules and maximum amount of data which can be transferred to the cloud.

The application specification is represented through a *consumption graph*, which captures both the application structure and the gathered statistics. It consists of a directed acyclic graph $G = \{M, D\}$, where a vertex M_i is a module and an edge d_{ij} models a dependency between M_i and M_j. Each vertex M_i has a cost expressed through two parameters: the code size c_i and the execution time t_i of the corresponding module. Each edge d_{ij} has a cost expressed by the size of data transferred between the connecting modules, $in_{ij} + out_{ji}$.

Given the consumption graph, the optimization problem consists of finding a cut in the graph such that some application modules execute on the device and the rest on the cloud. Let us consider an application with n modules, M_1, M_2, ..., M_n and a partition $P = P_{device} \bigcup P_{cloud}$, where $P_{device} = \{M_p | p \in [1, ..., k]\}$ is the set of modules to migrate on the mobile device and $P_{cloud} = \{M_s | s \in [1, ..., l]\}$ is the set of modules residing in the cloud. The objective function minimizes the overall interaction time of the application, while taking into account the overhead of acquiring and installing the necessary modules on the device, as well as generating proxies for all remote dependencies.

$$min\ O_P = min\ (\sum_{i=1}^{k} \frac{c_i}{B} + t_{is} * k + t_p * r + \sum_{i=1}^{k} t_i + \sum_{j=1}^{l} t_j + \sum_{i=1}^{t \leq k} \sum_{j=1}^{w \leq l} \frac{(in_{ij} + out_{ji})}{B})$$

$$such\ that:\ \sum_{i=1}^{k} c_i \leq C_{MAX}\ \ and\ \ \sum_{i=1}^{t \leq k} \sum_{j=1}^{w \leq l} in_{ij} \leq D_{MAX}$$

The first part in the function models the cost of migrating k modules to the mobile device over a link of bandwidth B, installing and starting them ($t_{is} * k$), as well as generating the proxies for all remote dependencies ($t_p * r$). As we explain in Section 4, in order to become active, modules need to be installed and started, and proxies must also be established to manage the client-cloud communication. Our measurements show that the overall installation and starting time of an application's partition linearly increases with the number (k) of modules fetched on the device. t_{is} is a parameter characteristic of the phone platform, which can be measured at bootstrap. For the phone platform we used, for instance, we found $t_{is} = 1700\ ms$. The proxy generation time depends on the number of remote dependencies (r) the fetched modules require. We found the startup time per proxy (t_p) to be in average 360 ms (300 ms for WiFi, 420 ms for 3G).

The second part of the function models the computation time of the modules executing on the client and on the cloud. We explain in Section 4 how the client's CPU time is estimated. Finally, the last term in the function captures the time necessary for transferring data between the distributed modules. The solution to the problem must also satisfy a group of user-defined constraints. The example above shows constraints on the maximum size of bytecode to be migrated to the mobile device and on the data transferred from the device to the cloud at each application invocation. To find the optimal partition we modify the *ALL* algorithm proposed in [14] to account also for the CPU and network analysis. The *ALL* algorithm takes as input the consumption graph and generates all possible partitioning configurations obtained by traversing the graph in an adapted topological order that combines both breadth-first and depth-first search. The algorithm first eliminates the configurations that do not satisfy the user's constraints, and then evaluates the objective function for each valid configuration such that the optimum can be found. Its complexity is $O(|M||D|log|D|)$.

Dynamically adjusting partitions. Since the execution environment changes dynamically (variations in CPU load on the mobile phone, network bandwidth, etc.) our system needs to be able to promptly switch from an application distribution to another, if necessary. In order to do this, the optimizer periodically runs and detects when the current partitioning is no longer optimal. In replacing a current distribution with a newer (optimal) one, it is important to minimize the application's interruption time, and possibly carry out most of the reconfiguration work in parallel to the ongoing execution. To reduce the overall bootstrap cost, our technique takes into account which modules have been fetched and installed on the mobile device by the previous distribution. The optimizer searches for the optimal configuration, and, if different from the current one, it transfers the missing modules to the mobile device. While the previous configuration continues to operate, the system installs the newly fetched modules. Once the initialization of the new configuration has finished, if there is an ongoing interaction, at its termination our system seamlessly switches to the new configuration which will be used from the next interaction onwards.

Adaptation to varying data inputs. Besides variations in CPU load and network, a user's data inputs can significantly affect the partitioning decision.

This is relevant for a large class of interactive applications that our system targets. While in some applications, user inputs are relatively standardized or it is possible to build an accurate approximation model (e.g., a ticket machine), for other applications it is hard to predict properties such as number, type, and size of the inputs. For instance, in an image processing application, the size and number of images during a user session are relevant factors in determining the CPU and network requirements of the application. Likewise, for a text-to-speech synthesizer, the text size to be translated can impact the application's behavior.

We exclude the possibility of running the optimizer on the client side because this would involve extra communication for collecting the profiling information from the cloud side, as well as extra CPU overhead for running the algorithm. Instead, we allow clients to *cache* some of the optimizer's solutions and autonomously decide on which partitioning configuration to use.

The optimizer first computes the optimal partitioning with the current network conditions and some default user inputs. It then generates additional solutions by simulating possible operating scenarios. Scenarios are defined by varying various *features*, describing both the operating environment and user inputs. For instance, the *network bandwidth* feature has the format *network(lower, upper)* and examples are *wifi(0.0,3.0), wifi(3.1-6.0), 3G(0.0,1.5), 3G(1.6-3.0)*. The *input* feature has the format *input([lower_num,upper_num],[lower_size,upper_size])* and qualifies number and size ranges of a specific input (e.g., images submitted to an image-processing application, text sent to a speech synthesizer). Examples are *intext([1-5],[1-500]), intext([1-5],[501-1000]), intext([6-10],[1-500])*.

By generating all the possible combinations of such features, a pool of scenario configurations is derived. The optimizer computes the optimal partitioning for each configuration and returns to the client a report consisting of tuples <*configuration_type,solution*>. At each interaction with the application, the client consults the cached report and based on the inputs received and the operating conditions, it autonomously decides on which configuration to adopt.

A potential risk with this approach is that by considering all possible values that the features might take, the number of scenarios to process grows exponentially. To limit this number, the server maintains a history of the minimum and maximum values previously observed for each feature and computes a maximum number of ranges for each one (typically in the order of 4 ranges). In addition, the features are manually specified by developers such that only relevant aspects are monitored. If a new input does not fit in any of the ranges, then the chosen configuration will be done corresponding to the range closest to the input.

4 Implementation

Our system is implemented for the Android platform and is based on ApacheFelix [10] (i.e., a Java implementation of the OSGi module management system), with the addition of R-OSGi [23] for remote execution across platforms. The architecture is shown in Fig. 1. The runtime on the cloud includes the application profiler, the consumption graph generator and the optimizer. The optimizer returns to the mobile device the list of modules, *bundles* in OSGi terminology, to fetch using the migrator, and a pool of selected configurations to cache. The

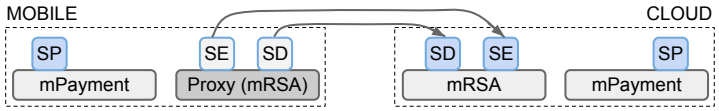


Fig. 2. Example of bundle deployment using R-OSGi

client runs the device profiler and a reduced version of the application profiler collecting only CPU statistics.

Flexible bundle deployment. Bundles are reusable pieces of software packaged in binary components, containing bytecode and metadata (i.e., versioning and dependencies). Modular designs encourage the coupling of related functions in the same bundle, exposed through a *service interface*. Any bundle that wants to use another bundle needs references to the registered services, thus being oblivious to any implementation details. OSGi allows an application to install, start, stop and uninstall bundles, as well as register services.

Since OSGi is restricted to single machines, our system requires an additional layer for remote communication, namely R-OSGi. R-OSGi's main goal is to provide dynamism and full location transparency for bundles, without changing their implementation or structure. To provide remote communication across bundles, R-OSGi generates a *proxy* on the calling bundle's side, which delegates service calls to the remote side. The proxy is registered with the local service registry as an implementation of the remote bundle service. An alternative to proxy generation is the actual fetching and installation of the remote bundle.

In Fig. 2, we consider a cloud application consisting of two bundles *mPayment* and *mRSA*, with their services *SP*, *SE* and *SD*, such that SP depends on both SE and SD. Initially only *mPayment* is fetched on the mobile device and remote proxies are generated for *mRSA*'s services. As the optimal distribution can dynamically change, our goal is to switch between partitions without interrupting an ongoing interaction, by exploiting R-OSGi's dynamic bundle management. Changing a configuration means acquiring the new bundles, installing and starting them, stopping and uninstalling the currently running ones, as well as generating the necessary remote proxies. Let us assume that the optimizer decides to fetch also *mRSA* on the mobile device. To initialize *mRSA* on the client, our system performs the following operations: (a) it migrates the code of *mRSA* to the device; (b) it installs and starts *mRSA*; (c) it generates remote proxies for all dependencies on *mRSA*; (d) it removes its proxy used by *mPayment*. When the process is completed, the new configuration is ready to be used. With the exception of operation (d), all other steps can occur in parallel with an ongoing interaction, without the need for the current configuration to stop.

Profiling with structural reflection. Application profiling uses load-time reflection at bytecode level. Every bundle has a MANIFEST file with metadata on versioning, services and dependencies on other bundles. For each service, the profiler identifies the Java classes implementing it and injects in all methods code to measure the execution time and the size of I/O parameters. The overall exe-

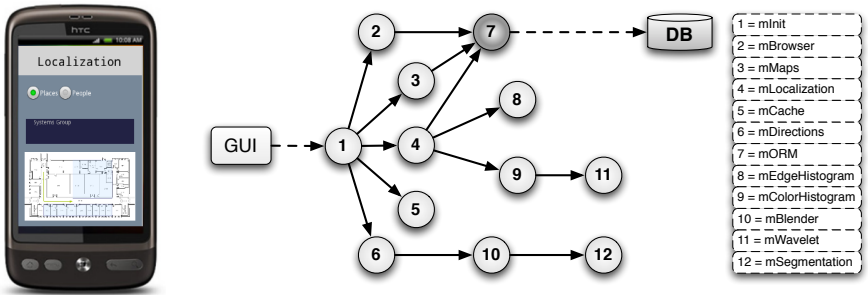
cution time per bundle is the sum of the running times of all executed methods. Measuring the data transfer between bundles allows us to identify which bundles are closely coupled and can benefit from colocation. The execution time helps to identify computational-intensive bundles which might cause performance degradation if ran on the mobile device. Finally, by inspecting the JAR package of a bundle, the profiler extracts its bytecode size, which is relevant to estimate the bundle's migration time and the storage required on the mobile device.

The first time an application is profiled, all static (i.e., bytecode size, services and dependencies) and dynamic (i.e., running time and I/O data size) parameters are measured on the cloud. At runtime, the profiler monitors only the dynamic variables. To avoid flapping in the measurements, it maintains a history of measurements and computes exponentially smoothed moving averages.

CPU, network and power profiling. Our system does not assume offline profiling for all applications and mobile platforms. Relative to CPU, this would require running all configurations for every application on the mobile device, and measuring the execution time for each invocation. In practice, we found a simple approximation to be accurate enough for our optimization problem, with the benefit of a small overhead on the client. The execution time of each bundle on the device is approximated as $t_c = t_s * K$, where t_s is its execution time on the cloud and K is a factor indicating how much slower the client's CPU is compared to the remote machine. Offline, we experiment with various mobile platforms and estimate the corresponding K parameters. In our setup, we found K=*3* to work well for all our applications. At the beginning of a user interaction, the optimizer uses the estimated K parameter, and then dynamically corrects the initial estimation based on the CPU execution time of all bundles running on the device. In addition, on the mobile side we periodically obtain the current CPU load of the device, over all active processes, from Android API functions.

Relative to network, the profiler detects whether the user is using WiFi or 3G by parsing the content of *proc/net/dev*. To estimate the bandwidth and latency of the current network, the system prunes the network by periodically sending 50 kB of data to the cloud. We found 50 kB to be a good representative size for our applications. Measurements are carried out every 30 seconds, but once an application interaction starts, *opportunistic profiling* is used instead: the bandwidth estimation is based on transfers carrying actual application data.

In order to profile the power consumed by running application bundles on the mobile device, our system uses PowerTutor [21], an online power estimation system that has been implemented for the Android platform. Since CPU and network are prime factors for application bundle distribution, we profile the CPU and WiFi/3G statistics provided by PowerTutor and define the power consumption as $Power_{total} = Power_{WiFi|3G} + Power_{CPU}$. Measuring the power consumed by an application can validate whether our latency-based model is effective in both minimizing the interaction time and reducing the device's energy consumption. Thus, we require that the power consumed with the optimal configuration found by the system is smaller than those experienced when the entire application is running either on the device or in the cloud. Our results in Section 6 validate these conditions.



(a) Localization (b) Modularization of the indoor localization

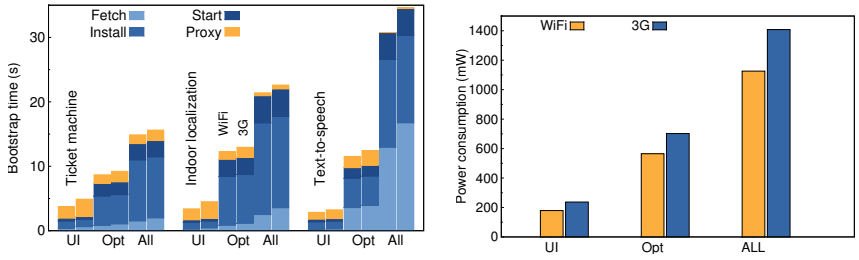
Fig. 3. IL application on Android and its modularization scheme

5 Applications

We briefly describe the three prototype applications we used to evaluate our system: indoor localization (IL), text-to-speech synthesizer (TTS) and ticketing machine (TM). IL and TTS belong to the *maps* and *media* application categories, while TM is an example of infrastructure service. The class of applications we target are computationally and network intensive, and are characterized by request-reply interactions. However, the model can be extended to other categories, such as streaming, by incorporating queuing networks to naturally emulate the behavior of application modules and capture clients arrival rate.

We developed IL and TM from scratch, and modularized an already existing TTS synthesizer [13]. A screenshot of the IL application is shown in Fig. 3(a). The IL application provides users with visualization facilities of a building map, including map tracking, browsing and directions to people and places within the building. Localization is carried out using the phone camera. As shown in Fig. 3(b), these functions were implemented using 12 bundles. *mInit* sets parameters and user preferences. *mMaps* and *mBrowser* allow the user to choose buildings, places and people for which *mORM* retrieves maps from a database. *mCache* can save the searched maps for future use. To locate themselves inside a building, users take photos of their surroundings, which are then compared for similarity against existing snapshots in the database. To determine the similarity degree, photos are decomposed in wavelets and features, such as color and edge histograms (*mWavelet*, *mColorHistogram*, *mEdgeHistogram*). The average values are then compared to the precomputed ones in the database. Finally, *mDirections* displays a map highlighting the path from the user's current position to the browsed place or person. *mBlender* and *mSegmentation* use image processing algorithms to draw the required directions. Only *mORM* is marked as *non-movable* to the mobile device, since it is strongly coupled with the database.

For the TTS and TM applications we only provide a brief description. TTS supports two operations: (a) the translation of a text extracted from a photo taken with the device's camera, and (b) the generation of speech from the translated text. The application has been implemented by adapting modules from the FreeTTS [13] synthesizer. The application was modularized in 10 bundles. Fi-



(a) Startup time (b) Power consumption (IL)

Fig. 4. Startup time and power consumption on HTC Desire for the three applications on EC2 US-East instances (WiFi and 3G in use)

nally, the TM application was the result of a joint project with the Swiss national railway (SBB). TM allows users to purchase train tickets, browse train routes, check prices and receive electronic tickets from their mobile devices. The application's functions were split into 10 bundles, out of which some contained private data of SBB and therefore were bound to remain in the cloud. For both applications the graph obtained by modularization is similar to the one in Fig. 3(b).

6 Evaluation

We evaluate how our system meets the following goals: (a) improving an application's performance in the cloud-mobile device setup, (b) dynamically switching between partitioning configurations, (c) reacting to variations in CPU load, network and user inputs, and (d) maintaining a reasonable overhead on the device. In the tests we use the three applications described and all experiments consider 15 repetitions. The client runs on a HTC Desire smartphone and the server on small, medium or large standard Amazon EC2 instances. The HTC Desire phone runs Android 2.1, has a Qualcomm QSD 8250 1 GHz processor and 576 MB of RAM. The smartphone communicates with the server using WiFi or 3G.

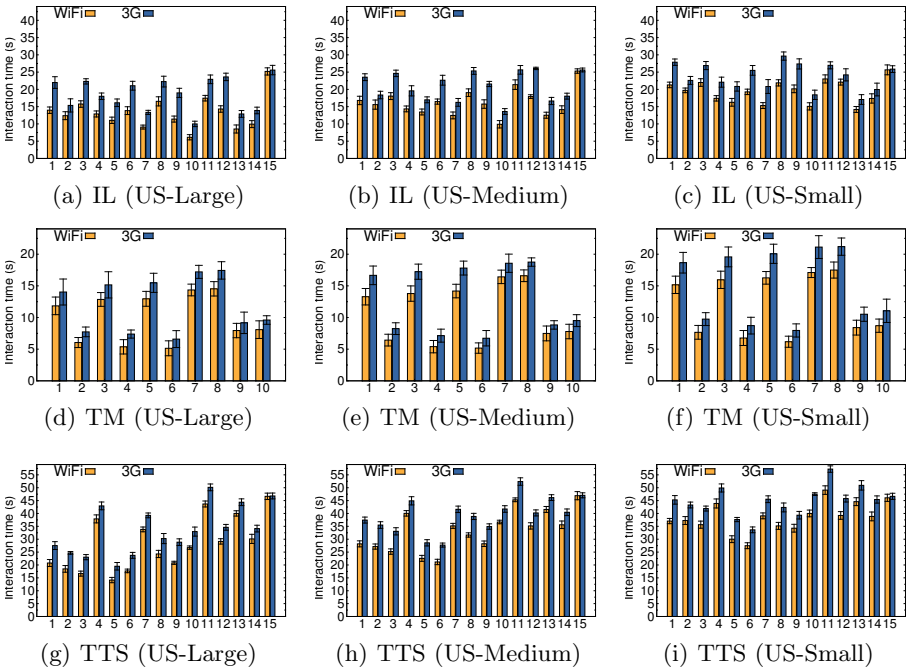
To show that AlfredO chooses the optimal configuration, in all experiments we consider all possible distributions of the applications and execute them in the setup described above. Then, by comparing the measurements obtained against the optimizer output, we can argue whether AlfredO's decision matches reality.

6.1 Initialization cost

First, we characterize the performance overhead of our system on the Android platform. On the HTC Desire, the start up consists of launching the client components shown in Fig. 1 and registering their inter-dependencies. This takes on average 12–14s. Once the system is running, the startup time of an application varies depending on the module distribution between client and cloud.

Fig. 4(a) shows the installation times for all three applications. For each application, we report 3 pairs of bars. For each pair, the first represents the WiFi case and the second one the 3G case. The first set of bars reports the installation time for the UI configuration, in which only the user interface is fetched on the mobile device. The second set (Opt) represents the installation

time for the optimal distribution, while the last (All) evaluates the case in which the entire application is installed on the client. Fig. 4(b) reports how much power is consumed for the IL application for all three configurations (UI, Opt and All). The purpose of this experiment is to show how the time and power consumption for an application deployment can be greatly reduced by acquiring only parts of it on the device. For the TTS application, the gap between the Opt and All installation times is of almost 18s, which is a considerable overhead for a mobile device. We also notice a significantly less power consumption by 600–700mW when installing the optimal distribution, compared to acquiring the whole application locally. On the other hand, when comparing the Opt and UI configurations, one may think that acquiring only the user interface represents always the quickest option. In the next set of experiments, we show that the problem is more complex, and analyze how the resulting application interaction time and power consumption varies with the chosen partitioning configuration.



(a) IL (US-Large) (b) IL (US-Medium) (c) IL (US-Small)

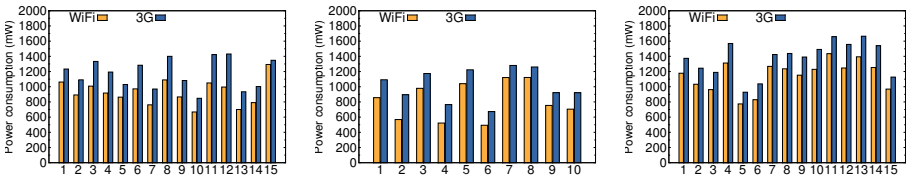
(d) TM (US-Large) (e) TM (US-Medium) (f) TM (US-Small)

(g) TTS (US-Large) (h) TTS (US-Medium) (i) TTS (US-Small)

Fig. 5. Interaction time for the three applications with varying configurations, on different EC2 instances (WiFi and 3G in use)

A more detailed analysis of Fig. 4 also shows the overall installation time breakdown. This includes the overhead for fetching, installing and starting the selected bundles, as well as generating proxies for remote dependencies. The fetching time depends on the bundles' code size, while the installation-start time is typically around 1.7s per bundle. Generating one remote proxy takes around 300ms for WiFi and 420ms for 3G. For the IL application (see Fig. 3(b)), the UI setup acquires 1 bundle and generates 5 proxies, while the Opt configuration

fetches 6 bundles and generates 3 proxies. The time required to stop-uninstall a bundle is in the range of 1.6s and a proxy removal requires around 350–400ms.



(a) IL (US-Large) (b) TM (US-Large) (c) TTS (US-Large)

Fig. 6. Power consumption for the three applications with varying configurations, on EC2 US-Large instances (WiFi and 3G in use)

6.2 Steady-state behaviour

Next, we consider the system's steady state, when all required bundles for a configuration have been fetched, installed and started. Fig. 5 reports the observed interaction time for a subset of the possible partitionings for the three applications on three EC2 instance types (small, medium and large). Configurations are ordered by increasing number of bundles acquired on the client.

Acquiring more bundles on the mobile side does not necessarily improve performance, and installing only the user interface (configuration 1 in all figures) is not always the optimal choice. In general, how an application's performance varies is not so easily correlated to how many modules are moved to the client or what their size is. These results motivate the need for a partitioning algorithm capable of picking the most suitable partition. The optimal partitioning for IL depends on the EC2 instance type, with configuration 10 for large and medium instances and 13 for small machines. Similarly, for TTS configuration 5 is best for large instances and 6 for medium and small ones. In the case of TM, the optimal configuration is 6 for all instances. Fig. 6 reports the power consumed by the same configurations for all applications, but only on large EC2 instances. Our measurements confirm that the optimal configuration in terms of interaction time is also the most power-efficient for all applications. In addition, we observe similar trends between the power consumption and interaction times for small and medium EC2 instances, where the best configurations for both IL and TTS applications change. The results show that an interaction latency-based model is enough to find those distributions that are also the most power-efficient. The reasoning behind is that network operations are more expensive in terms of mWs consumed, and therefore solving the partitioning problem with the goal of minimizing overall data transfers also achieves optimal power consumptions.

Next, we apply our solving algorithm and measure the achieved improvement on the interaction time and power consumption. For all our applications, the algorithm is able to select the best configuration. Table 1 reports the interaction time and power consumption of the optimal configuration, as well as the algorithm solving time. The performance gain on the mobile device is very promising. For TM the gain in performance is up to 61%, when we compare with the two extreme cases, All and UI. For IL and TTS, the improvements are

Table 1. Gains on performance and power consumption (on EC2-Large)

Solver		TM (0.16s)		IL (0.2s)		TTS (0.17s)	
		Time(s)	Power(mW)	Time(s)	Power(mW)	Time(s)	Power(mW)
Opt	WiFi	5.17	493	6.13	668	14.14	774
	3G	6.72	673	10.01	848	19.05	928
All	WiFi	33%	30%	75%	46%	69%	21%
	3G	29%	27%	60%	38%	58%	18%
UI	WiFi	57%	43%	56%	37%	31%	34%
	3G	52%	39%	54%	32%	29%	32%

even higher, up to 75% and 69% respectively. Finally, the comparison on power consumption presents similar trends and gains for all three applications by 20–46%, with smaller values for 3G as expected due to its increased latency. The percentages are computed as the ratio between $Diff_t$ and Opt_t, where $Diff_t$ represents the difference between the execution time obtained with ALL or UI depending on the case and the execution time obtained with Opt, while Opt_t is the optimal interaction time.

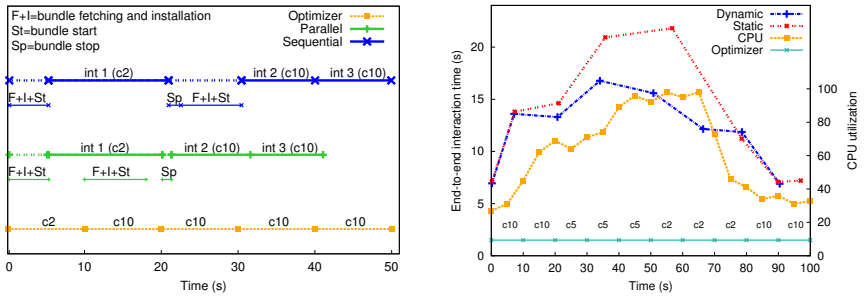
To understand how performance is improved we need to consider the applications' bundle structure. In the case of IL, shown in Fig. 3(b), the best configuration contains *mInit*, *mCache*, *mMaps*, *mBrowser*, *mDirections* and *mBlender*. For this application, it is not convenient to fetch more bundles because they are too computational intensive for the mobile platform and their execution time on the client exceeds the time required for data transfers to the cloud.

Once the optimizer has identified the optimal configuration, the mobile client fetches, installs and starts the corresponding bundles, and sets up the remote proxies. However, the more the user interacts with the application, the faster the initialization cost is amortized. We measured the number of invocations necessary to fully amortize the initialization cost. For WiFi at most two interactions are sufficient to pay off the initial overhead, while for 3G already one invocation is enough to amortize the overhead. This confirms that our approach can bring such a high performance improvement to fully hide the installation overhead, and thus makes it suitable for both long-term short-term interactions.

6.3 Dynamic optimization and redeployment

We investigate which configuration is optimal at bootstrap, when no bundle has been acquired yet on the mobile device. Ideally, in choosing the best partitioning one should consider the number of user interactions. In the case of one interaction, the configuration with minimal installation time is probably better, while for more interactions, having a lower invocation time is more important. As the number of interactions cannot be easily known a priori, our system makes the decision entirely online and periodically re-evaluates it, as shown in Fig. 7(a).

Consider a user performing three consecutive interactions with the IL application. The optimizer processes the objective function every 10s and returns



(a) Parallel bundle initialization (3G) (b) Adapting to CPU changes (WiFi)

Fig. 7. Parallel bundle initialization and adapting to CPU changes (IL)

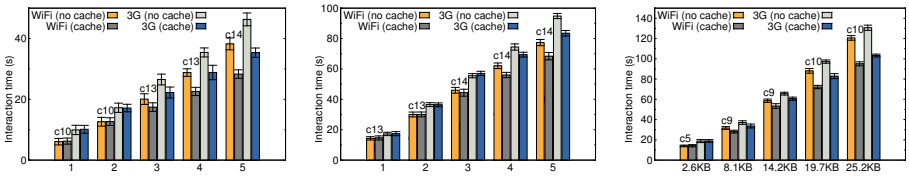
the best configuration. At time 0 (no code is acquired yet on the device), the optimizer picks configuration 2 (*c2*), which does not provide the lowest interaction time, but minimizes the overall time for initialization and interaction. As some bundles have already been fetched on the client, at time 10, the optimizer picks the configuration with the lowest interaction time, *c10*. Installing the new distribution is done in parallel to the application running. Its cost is roughly 8s, since c2 is already active, and it can be used already for the second interaction.

Fig. 7(a) shows the benefits of initializing *c10* in parallel to the application execution (*Parallel*). In this way, before *c10* becomes active, there is an interruption of only 1.2 s (*Sp*) due to the removal of the 3 remote proxies used by *c2*. This is the only operation that cannot be executed concurrently, but its overhead is negligible compared to the performance gain. In fact, when comparing against the case when the initialization happens sequentially (*Sequential*), the parallel approach allows the user to carry out 3 full interactions in the time the sequential one completes 2. Moreover, in the sequential approach, the application is not available for roughly 10s (represented with a dotted line).

6.4 Reactivity to CPU load

The ability of reconfiguring an application online allows the system to quickly react to changing network conditions or CPU load. We give an example of the latter in the next experiment. We cause an increase in the device's CPU utilization to 67% and 95%, by running for a few minutes a CPU-intensive process. The optimizer reacts to the CPU variations by choosing a "cheaper" configuration in terms of consumed CPU. Fig. 7(b) shows the performance improvements our approach (*Dynamic*) has over the *Static* one, which always runs with the configuration chosen at the first interaction.

As shown in Fig. 7(b), the first interaction with IL uses *c10*. When the CPU increase to 67% occurs, based on new profiled data the optimizer decides to switch to *c5*, which contains 4 of the 6 bundles from *c10*. This decision improves performance by roughly 5s over the static approach (visible in the 4th interaction). Between the 4th and 5th interactions, an additional increase to 95% occurs. Again, the optimizer reacts by choosing *c2*, which reduces the number of bundles running on the mobile device to 2. The performance improvement of the



(a) Localization (US-Large) (b) Localization (US-Small) (c) TTS (US-Large)

Fig. 8. Reactivity to changing user inputs for the IL and TTS applications

dynamic approach compared to the static case is even larger. Finally, we decrease the CPU utilization to 37% and the optimizer decides to switch back to *c10*. Its bootstrap is done in parallel to the 7th interaction and the performance becomes similar to the static case. Given that application interactions running with specific configurations are not interrupted once the optimizer switches to different partitionings, allows our approach to become stable. This stability comes from the fact that the optimizer is given the opportunity to periodically verify and strengthen its decision while the application is executing.

This test shows the efficiency of our dynamic approach over a static one. The same mechanism has shown to be effective in reacting to changes in network bandwidth caused by unstable wireless connectivity or switching between WiFi and 3G. Due to space restrictions we only briefly present our observations. By reducing the available bandwidth on the device, for WiFi the optimizer switches from *c10* to *c13*, which brings more bundles locally and reduces the remote data transfer. By doing so, a user is able to perform 11 interactions in the same time 8 interactions are executed with the static approach. For 3G, *c14* is chosen and more bundles than in the WiFi scenario are brought on the device. This is due to the lower available bandwidth with 3G connections. The dynamic AlfredO manages to perform 8 interactions compared to 6 with the static approach.

6.5 Adapting to changing user inputs

Next, we consider the system's ability to adapt to changing user inputs, based on a set of usage scenarios and associated configurations cached on the device. We evaluate this feature with the IL and TTS applications.

In Fig. 8, we compare the interaction time when the cache is disabled or enabled. If disabled, the device adopts the configuration used for the previous interaction. If enabled, the client chooses the best configuration depending on the inputs. For IL, we test by increasing the set of inputs from 1 to 5 images, each 300 kB in size. For TTS, we vary the size of the text from 2.6 kB to 25.2 kB.

On large EC2 instances, both applications show gains of 20-25% with cache enabled. For IL, Fig. 8(a) shows that already with 3 photos it is best to switch from *c10* to *c13*. With 5 or more photos moving to *c14* is optimal. In the TTS application (Fig. 8(c)), the system changes from *c5* to *c9* at the first increase in text size. With a text increase to 19.7kB, the best configuration becomes *c10* and improves performance by over 25s for both WiFi and 3G. For small EC2 instances, the cache decides to switch from *c13* to *c14* with 3 input photos (Fig. 8(b)), when in fact *c13* has a lower interaction time for 3G. This is due

to the generic nature of the cached solutions, which cannot cover all possible scenarios. Even so, the penalty in performance is very small (1.5s).

6.6 Resource overhead

Finally, we discuss the system's overhead on the mobile platform. The code size of all components residing on the mobile device is 178 kB, while on the cloud side is 903 kB. The memory footprint is typically less than 7 MB and is comparable to other applications or processes running simultaneously on the Android platform.

Profiling requires code injection for all bundles. The code increase depends on the number of classes and methods to be profiled, but it typically does not exceed 2–3 kB for a bundle of 20–25 kB. We observed that the performance degradation due to profiling is under 8% for all bundles. The data generated by the profiler represents the statistics collected at each user interaction. In average, the logged measurements require less than 2 kB of data.

7 Conclusions

With the ever richer functionality of mobile applications, users are confronted with either the computational limitations of their devices or the network limitations. Recent work has proposed application partitioning between mobile devices and remote servers or clouds, to improve performance and battery life. In this paper, we argue that static decisions or ignoring the effects of user data cannot leverage the full potential of code offloading when variations in network, device CPU load, or user inputs occur. Our system shows that dynamically adapting partitioning decisions is key to improve user experience. Our experiments over different networks and cloud infrastructures show that our approach significantly reduces interaction time and power consumption by (1) fetching application parts to the mobile device when appropriate, (2) dynamically adjusting the distributed configuration to changes in the network conditions, client load and user inputs, and (3) caching deployment settings for efficient execution with varying application inputs. Additionally, our system offers a greater degree of flexibility for applications on mobile devices as it supports a wider range of scenarios than just services running completely in the cloud.

References

1. Amazon EC2. `aws.amazon.com/ec2/`.
2. K. Amiri, D. Petrou, G. Ganger, and G. Gibson. Dynamic Function Placement for Data-intensive Cluster Computing. In *Proc. of USENIX*, pages 307–322, 2000.
3. Android. `code.google.com/android`.
4. Google AppInventor. `appinventor.googlelabs.com/about`.
5. R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H. Yang. The case for cyber foraging. In *Proc. of the 10th Workshop on ACM SIGOPS European Workshop: Beyond the PC*, pages 87–92. ACM, 2002.
6. R. K. Balan, M. Satyanarayanan, S. Y. Park, and T. Okoshi. Tactics-based remote execution for mobile computing. In *Proc. of MobiSys*, pages 273–286. ACM, 2003.
7. B. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: Elastic execution between mobile device and cloud. In *Proc. of EUROSYS*. ACM, 2011.

8. B. Chun and P. Maniatis. Augmented smarphone applications through clone cloud execution. In *Proc. of the 12th USENIX HotOS Workshop*, 2009.
9. E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: making smartphones last longer with code offload. In *Proc. of MobiSys*, pages 49–62. ACM, 2010.
10. Apache Felix. `felix.apache.org/site/index.html`.
11. J. Flinn, S. Park, and M. Satyanarayanan. Balancing performance, energy and quality in pervasive computing. In *Proc. of ICDCS*, pages 217–. IEEE, 2002.
12. Fragments. `android.com/guide/topics/fundamentals/fragments.html`.
13. FreeTTS. `freetts.sourceforge.net/docs/index.php`.
14. I. Giurgiu, O. Riva, D. Juric, I. Krivulev, and G. Alonso. Calling the cloud: Enabling mobile phones as interfaces to cloud applications. In *Proc. of Middleware*, pages 1–20, 2009.
15. S. Goyal and J. Carter. A lightweight secure cyber foraging infrastructure for resource-constrained devices. In *Proc. of WMCSA*, pages 186–195, 2004.
16. G. Hunt and M. Scott. The coign automatic distributed partitioning system. In *Proc. of OSDI*, pages 187–200. USENIX, 1999.
17. Y.-W. Kwon and E. Tilevich. Power-efficient and fault-tolerant distributed mobile execution. In *Proc. of ICDCS*, 2012.
18. R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden. Wishbone: Prole-based Partitioning for Sensornet Applications. In *Proc. of NSDI*, pages 395–408, April 2009.
19. OSGi Alliance. *OSGi Service Platform, Core Specification, v4.1, Draft*, 2007.
20. S. Ou, K. Yang, and J. Zhang. An effective offloading middleware for pervasive services on mobile devices. volume 3, pages 362–385. Elsevier Science, 2007.
21. Powertutor, 2009. http://ziyang.eecs.umich.edu/projects/powertutor/.
22. M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan. Odessa: enabling interactive perception applications on mobile devices. In *Proc. of MobiSys*, 2011.
23. J. S. Rellermeyer, G. Alonso, and T. Roscoe. R-OSGi: Distributed applications through software modularization. In *Proc. of Middleware*, pages 1–20, 2007.
24. J. S. Rellermeyer, O. Riva, and G. Alonso. AlfredO: An Architecture for Flexible Interaction with Electronic Devices. In *Proc. of Middleware*, pages 22–41, 2008.
25. M. Satyanarayanan. Pervasive computing: vision and challenges. *IEEE Personal Communications*, 8(4):10–17, 2001.
26. M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for VM-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, 2009.
27. Microsoft Silverlight. `www.silverlight.net`.
28. Y. Su and J. Flinn. Slingshot: deploying stateful services in wireless hotspots. In *Proc. of MobiSys*, pages 79–92. ACM, 2005.
29. F. Yang, N. Gupta, N. Gerner, X. Qi, A. Demers, J. Gehrke, and J. Shanmugasundaram. Computation offloading to save energy on handheld devices: A partition scheme. In *Proc. of CASES*, pages 238–246. ACM, 2001.
30. F. Yang, N. Gupta, N. Gerner, X. Qi, A. Demers, J. Gehrke, and J. Shanmugasundaram. A unified platform for data driven web applications with automatic client-server partitioning. In *Proc. of WWW*, pages 341–350. ACM, 2007.
31. F. Yang, J. Shanmugasundaram, M. Riedewald, and J. Gehrke. Hilda: A high-level language for data-driven web applications. In *Proc. of ICDE*, pages 32–43. IEEE Computer Society, 2006.