Calling the cloud: Enabling mobile phones as interfaces to cloud applications

Ioana Giurgiu, Oriana Riva, Dejan Juric, Ivan Krivulev, and Gustavo Alonso

Systems Group, Department of Computer Science, ETH Zurich
8092 Zurich, Switzerland
{igiurgiu,oriva,alonso}@inf.ethz.ch

Abstract. Mobile phones are set to become the universal interface to online services and cloud computing applications. However, using them for this purpose today is limited to two configurations: applications either run on the phone or run on the server and are remotely accessed by the phone. These two options do not allow for a customized and flexible service interaction, limiting the possibilities for performance optimization as well. In this paper we present a middleware platform that can automatically distribute different layers of an application between the phone and the server, and optimize a variety of objective functions (latency, data transferred, cost, etc.). Our approach builds on existing technology for distributed module management and does not require new infrastructures. In the paper we discuss how to model applications as a consumption graph, and how to process it with a number of novel algorithms to find the optimal distribution of the application modules. The application is then dynamically deployed on the phone in an efficient and transparent manner. We have tested and validated our approach with extensive experiments and with two different applications. The results indicate that the techniques we propose can significantly optimize the performance of cloud applications when used from mobile phones.

Key Words: Mobile phones, cloud applications, OSGi, performance

1 Introduction

Mobile phones are set to become a main entry point and interface to the growing number of cloud computing services and online infrastructures. They are also increasingly perceived as the most convenient access point for a variety of situations: from payments to ticket purchase, from carrying boarding passes to hotel check-in, from browsing a shop catalog to activating a coffee machine.

Today, the implementation of such scenarios is limited by the lack of flexibility in deploying mobile phone applications. They either run entirely on the server, typically incurring large data transfer costs, high latencies, and less than optimal user interfaces; or they run entirely on the phone, thereby imposing many limitations on what can be achieved due to the constraints of mobile phone hardware, as well as placing an undue burden on the end users who need to install, update,

and manage such applications. In this paper we explore how to deploy such applications in a more optimal way by dynamically and automatically determining which application modules should be deployed on the phone and which left on the server to achieve a particular performance target (low latency, minimization of data transfer, fast response time, etc.). Having such a possibility creates a wealth of opportunities to improve performance and the user experience from mobile phones, turning them into an open, universal interface to the cloud.

To optimally partition an application between a mobile phone and a server, we approach the problem in two steps. First, we abstract an application's behaviour as a data flow graph of several inter-connected software modules. Modules encapsulate small functional units supplied by the application developer. Each module provides a set of services, and modules are connected through the corresponding service dependencies. Through an offline application profiling, modules and service dependencies are characterized in terms of their resource consumption (data exchange, memory cost, code size), thus providing the knowledge base for the optimization process. Given this graph, in the second step, a partitioning algorithm finds the optimal cut that maximizes (or minimizes) a given objective function. The objective function expresses a user's goal such as to minimize the interaction latency or the data traffic. Moreover, the optimization also takes into account a mobile phone's resource constraints such as memory and network resources available.

We propose two types of partitioning algorithms: `ALL` and `K`-step. We look at the problem both as a static and dynamic optimization. In the first case, the best partitioning is computed offline by considering different types of mobile phones and network conditions. In the second case, the partitioning is computed on-the-fly, when a phone connects to the server and specifies its resources and requirements. `ALL` fits the first scenario, while `K`-step the second one.

Our approach does not require new infrastructures as it uses existing software for module management such as R-OSGi [1] and a deployment tool like AlfredO [2], that can support the actual distributed deployment of an application between a phone and a server.

This paper makes the following contributions. First, we model the partitioning problem and the algorithms that can solve it. Second, we show the effectiveness of this approach with two prototype applications. Third, we present a comprehensive evaluation involving realistic application scenarios of mobile phones. Our measurements show that the system can quickly identify the optimal partition given various phone constraints, and provide an improvement of tens of seconds compared to the case in which the phone hosts the entire application or leaves all the service logic on the server.

The rest of the paper is organized as follows. The next section gives an overview of the AlfredO platform we use. Section 2.2 describes how application profiling is used to produce an application's consumption graph, while Section 3 presents the partitioning algorithms. Section 4 evaluates our approach and Section 5 describes an application using it. We then conclude and discuss limitations and open problems of your approach as well as related work.

2 Flexible Module Deployment

This section starts by providing background on AlfredO and then describes how application profiling is used to generate a consumption graph.

2.1 AlfredO overview

We use AlfredO [2] to carry out the physical distribution of an application's modules between a mobile phone and a server. AlfredO is based on OSGi [3], which has been traditionally used to decompose and loosely couple Java applications into software modules. In the OSGi terminology, software modules are called *bundles*, and bundles typically communicate through *services*, which are ordinary Java classes with a service interface.

Given an OSGi-based application with a presentation tier, a logic tier, and a data tier, where each tier consists of several OSGi bundles, AlfredO allows developers to decompose and distribute the presentation and logic tiers between the client and server side, while always keeping the data tier on the server.

In a typical example of interaction, the minimal requirement for interacting with a certain application is to acquire the presentation tier. Once AlfredO has built the presentation tier, the logic tier's services can be invoked. This happens by either redirecting invocations to remote services provided by the server side or by acquiring and running some parts of the logic tier locally.

Figure 1 shows an example of client-server interaction. Both devices run OSGi with the R-OSGi [1] bundle installed, which enables remote service execution across OSGi platforms. The AlfredO system consists of three bundles: *AlfredOClient* and *Renderer* on the client, and *AlfredOCore* on the server.

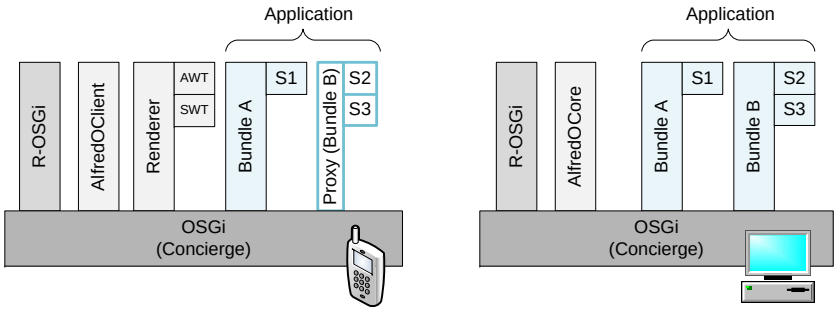


Fig. 1: AlfredO architecture.

Upon explicit discovery (e.g., using a service discovery protocol such as SLP [4] provided by R-OSGi) or by direct connection to a known address (e.g., the remote server periodically broadcasts invitations), the connection is established and the client requests a selected application. Using one of the available

partitioning algorithms (described in the next section), AlfredOCore computes the optimal deployment for such an application, and then returns to AlfredOClient the application's descriptor (also explained later) and the list of services to be fetched. The application's descriptor is used by the Renderer to generate the corresponding AWT or SWT user interface, while AlfredOClient fetches the specified services via R-OSGi.

In the general case, a server provides bundles offering services. If a client wants to use a server's service, the server provides the client with the service interface of such a service. Then the R-OSGi framework residing on the client side generates from the service interface a *local proxy*. The local proxy delegates service calls of the client to the remote server and each proxy is registered with the local R-OSGi service registry as an implementation of the particular service. If it happens that the service interface references types provided by the original service module and these are located on the server side, the corresponding classes are also transmitted and injected into the proxy module.

Alternatively, rather than invoking a remotely executing service, a client can decide to fetch it. In the example in the figure, if the client wants to acquire S1, the corresponding bundle A is transferred to the client side and plugged into the OSGi platform. When the client receives the service interface of S1, it is also provided with a list of the associated service's dependencies. Let us assume that S1 depends on S2, and S2 depends on S3, which are both offered by another bundle B. The client can either acquire only S1 and create a local proxy for S2 and S3, or it can also acquire S2 and S3.

2.2 Application profiling

The first step in optimizing an application's deployment is to characterize the behaviour of such an application through a resource consumption graph. We assume applications to be built using the OSGi module system, but the same method could be extended to work with applications modularized in other ways.

We instrument every bundle composing the application to measure the consumed memory, the data traffic generated both in input and output, and its code size. We then execute the instrumented application on one or multiple phone platforms and collect on a debug channel the amount of consumed resources. Each bundle's cost represents how much a phone has to pay if it wants to acquire and run that particular bundle locally.

In our optimization problem, we focus mainly on user interface type of functionality, since these are the modules that are more likely to be suitable for moving and running on a resource-constrained mobile phone. In addition, the large heterogeneity of the mobile platforms encountered today and the lack of one reference CPU architecture for mobile phones makes it hard to obtain stable estimations for CPU consumption that could be correctly applied to interactions with non-profiled phone platforms. We therefore simplify the profiling process by omitting a bundle's CPU cost.

An application developer classifies bundles as *movable* and *non-movable* based on their computation needs. Non-movable bundles are computing-intensive com-

ponents that are bound to always execute on the server side. This simplification has so far proven sufficient for the interactive applications we have considered and that AlfredO primarily targets, since the most critical factor in the overall performance is usually the amount of transferred data.

The profiling output is then used to generate the application descriptor. A snippet of the descriptor used in the example of Figure 1 is the following:

```
<tier>
  <requires>
    <service name="S3" data="350"/>
  </requires>
  <provides>
    <service name="S2"/>
  </provides>
  <memory>155</memory>
  <code>30</code>
  <type>movable</type>
</tier>
```

This descriptor specifies that service S2 requires S3 for its execution (i.e., S2 depends on S3) and the total amount of data that needs to acquire from S1 and return to S3 is 350 bytes. Other requirements include the memory cost of S2 when executed on the phone, and the size of the bundle to which it is associated.

2.3 Consumption graph

The output of the profiling process is used to represent the application as a directed acyclic graph $G = \{B, E\}$, where every vertex in B is a bundle B_i and every edge e_{ij} in E is a service dependency between B_i and B_j. Each bundle B_i is characterized by five parameters:

- *type*: movable or non-movable bundle,
- $memory_i$: the memory consumption of B_i on a mobile device platform,
- $code\_size_i$: the size of the compiled code of B_i,
- in_{ji}: the amount of data that B_i takes in input from B_j,
- out_{ij}: the amount of data that B_i sends in output to B_j.

Figure 2 shows an example of a graph consisting of 6 bundles. We call this an application's *consumption graph*. Notice that although our implementation currently considers these five parameters, the model is generic enough to be easily extended with more variables.

In this work we make the simplifying assumption that every bundle exposes only one service, i.e., *bundle:service* mapping is 1:1. This implies that a bundle can be interconnected to multiple bundles, but always through the same service interface. As ongoing work, we are relaxing this assumption by differentiating among the type and number of service dependencies.

The graphs we consider for optimization are not extremely large because, first, we focus on the presentation layer, and, second, modularity is not at the class or object level, but at the functional level. Therefore, we expect applications to have in most cases a few tens of modules.

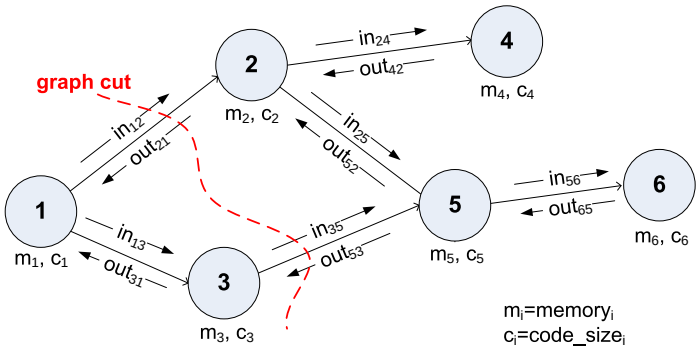


Fig. 2: Example of application's consumption graph.

3 Partitioning Algorithms

In this section, we describe the AlfredO's algorithms used to optimize an application's distribution between a phone and a server. The server is assumed to have infinite resources, while a client is characterized by several resource constraints. We start by describing how the optimization problem is defined and which assumptions are made, and then present the partitioning algorithms.

3.1 Optimization problem

The partitioning problem seeks to find a cut in the consumption graph such that some modules of the application execute on the client side and the remaining ones on the server side. The optimal cut maximizes or minimizes an *objective function* O and satisfies a phone's resource constraints. The objective function expresses the general goal of a partition. This may be, for instance, minimize the end-to-end interaction time between a phone and a server, minimize the amount of exchanged data, or complete the execution in less than a predefined time.

A phone's constraints include $memory_{MAX}$, the maximum memory available for all potentially acquired bundles, and $code\_size_{MAX}$, the maximum amount of bytes of compiled code a phone can afford to transfer from the server.

Let us consider an application consisting of n bundles of type movable, denoted as $B = \{B_1, ..., B_n\}$. A configuration C_c is defined as a tuple of partitions from the initial set of bundles, $< B_{client}, B_{server} >$, where $B_{client} = \{B_a | a \in [1, ..., k]\}$ and $B_{server} = \{B_b | b \in [1, ..., s]\}$ with $B_{server} \bigcap B_{client} = \phi$ and $B_{server} \bigcup B_{client} = B$.

An example of objective function that we will use to evaluate our approach minimizes the interaction latency between a phone and a server, while taking into account the overhead of acquiring and installing the necessary bundles. This can be modelled in the following way:

$$\texttt{min}O_{C_c} = \texttt{min}(\sum_{i=1}^{t<k}\sum_{j=1}^{w}\frac{(in_{ij} + out_{ji}) * f_{ij}}{\alpha} + \sum_{i=1}^{k}\frac{code\_size_i}{\beta} + \sum_{i=1}^{w<s} proxy\_cost_i)$$

The first part in the function models the cost due to the application's data exchange when k bundles run on the mobile phone and t bundles out of these have dependency relationships with w bundles residing on the server side. As we consider only movable bundles with a very low computation cost, this mainly consists of communication cost. The parameter α approximates the capacity of the communication link between the client and server achievable in real settings and also takes into account the overhead imposed by the device platform to set up the communication. Depending on the type of interaction a user may invoke a certain module multiple times. This is modelled through the f_{ij} parameter which specifies how many times the communication between i and j occurs.

The second part models the cost to fetch, install, and start the k bundles on the mobile phone. The parameter β takes into account the capacity of the communication link as well as the installation overhead. The third part represents the cost for building the local proxies necessary to interact with the w remote bundles. Notice that the f parameters appears only in the first member as having one or multiple interactions solely affects the amount of data sent back and forth between the mobile device and the server, while the cost of shipping the code and building local proxies remains the same.

Given the objective function and the consumption graph we want to find the optimal partition. Although many tools exist for graph partitioning, they do not prove to be suitable for our problem. Tools like METIS [5] are designed specifically for partitioning large scientific codes for parallel simulation. Moreover, they apply heuristic solutions in order to create a fixed number of balanced graph partitions, thus fixing predefined seeds and not allowing for flexibility. Other tools like Zoltan [6] represent an application as a graph, where data objects are vertices and pairwise data dependencies are edges. The graph partitioning problem is then to partition the vertices into equal-weighted parts, while minimizing the weight of edges with endpoints in different parts. This approach does not allow for unlimited and unspecified capacity for the server partition, and expects a single weight on each edge and each vertex. This constraint limits the applicability of the method, since it cannot support heterogeneity of different platforms.

Another option is to consider traditional task scheduling algorithms. However, the main drawback of task scheduling is that it does not fit a non-deterministic data flow model, since it assumes that all tasks are executed exactly once. Therefore, it does not fit scenarios where a user may interact with an application several times and spontaneously.

We therefore propose an alternative approach with two novel algorithms.

3.2 Pre-processing

Before running the actual algorithms, we pre-process the consumption graph to reduce the search space, but without eliminating optimal solutions. For large graphs, this step is essential to reduce the graph size and therefore the number of possible configurations, thus improving the algorithm's performance. The idea is to identify bundles that yield a very high cost and therefore cannot be moved

to the client or bundles that exchange a lot of data, and therefore should always execute on the same device.

Given a consumption graph $G = \{B, E\}$, if the cost of an edge $e_{ij} \in E$ is such that $in_{ij} + out_{ji} > data_{MAX}$, then Bi and Bj are merged into one bundle B_i: all input and output edges are updated accordingly, and the cost of the new bundle B_i is given by the sum of the relative costs of the old B_i and B_j.

3.3 ALL algorithm

After the pre-processing step, two classes of algorithms can be applied to find the optimal cut. The reason for having two different algorithms is that the optimization problem can be looked as a static problem, where the optimal partitioning for several types of mobile devices is pre-computed offline or as a dynamic problem where the partition must be calculated on-the-fly, once a mobile connects and communicates its resources. In this work, we consider both options and we propose `ALL` for offline optimization and `K`-step for online optimization.

The `ALL` algorithm always guarantees to find the optimal cut. It operates in three steps. First, it generates all "valid" configurations. Given B, we define $C = \{C_c | c \in [1, ..., m]\}$ the set of all valid configurations, where m is the total number of configurations obtained by traversing the consumption graph in an adapted topological order that combines both breadth-first and depth-first algorithms. A valid configuration is such that if bundle B_p and B_q belong to B_{client}, if B_p and B_q are not connected through a direct edge e_{pq}, then all bundles on the possible paths between B_p and B_q also belong to B_{client}.

Second, for all valid C_c configurations with k being the number of bundles to be fetched, installed, and run on the phone, it chooses the ones that satisfy the phone's constraints:

1. $\sum_{i=1}^{k} memory_i \leq memory_{MAX}$;
2. $\sum_{i=1}^{k} code\_size_i \leq code\_size_{MAX}$;

Third, the algorithm evaluates the objective function for each valid configuration and chooses the one providing its maximum (or minimum) value.

3.4 K-step algorithm

While the `ALL` algorithm inspects all possible configurations and identifies the "global" optimal cut, the `K`-step algorithm evaluates a reduced set of configurations and finds a "local" optimum. The `K`-step algorithm is therefore by design faster than the `ALL` algorithm, but can be less accurate.

Instead of generating all configurations and then choosing the best ones, this algorithm computes the best configuration at every step and on-the-fly. At the beginning, `K`-step adds to the current configuration the entry node of the graph and computes the current value for the objective function. Then, at each step, it adds `K` new nodes to the current configuration, only if these new nodes provide a configuration with an objective value larger (the goal is maximize O) or smaller

(the goal is minimize O) than the current one and if the phone's constraints are still respected. Depending on K, the algorithm can add one single node (K=1) or a subgraph of size K (K>1) computed by combining depth-first and breadth-first.

More specifically, at each step the algorithm maintains a queue containing all nodes in the graph (not yet acquired) within a distance of K hops from all nodes present in the current configuration. The algorithm generates all possible configurations with the nodes in the queue and the nodes already added to the current configuration. It then evaluates the objective function for each new possible configuration. If any of the new configurations provides an objective value better than the current one, then a new local optimum has been found. However, the K nodes enabling such a configuration are added only if their resource demands respect the phone's constraints. If the constraints are violated, the algorithm will evaluate them for the second best new configuration and so forth until a better configuration respecting the phone's constraints is found. If none of the new configurations provide a better objective value, while satisfying the phone constraints, the algorithm stops and returns the current configuration. Otherwise, if a configuration is found, the new K nodes are added to the current configuration and removed from the queue. The queue will be updated and the evaluation continues. The algorithm ends when the queue is empty or when the objective value cannot be improved any further.

4 Evaluation

To evaluate our approach we have explored two directions. First, we have built from scratch a prototype application and used it to test our algorithms under various resource and network constraints. This application is specifically designed to allow several configurations and stress the operation of the algorithms. Second, we have taken an existing application for home interior design and modified it to support our approach. In this section we focus on the experiments with the first application, while the second use case is presented in the next section.

In all results presented in the following the client runs on a Nokia N810 Internet tablet and the server on a regular laptop computer (Intel Core 2 Duo T7800 at 2.60 GHz). N810 handhelds, released in November 2007, run Linux 2.6.21, have a 400 MHz OMAP 2420 processor, 128 MB of RAM, and 2 GB of flash memory built in. N810 devices were connected to the laptop either through IEEE 802.11b in ad hoc mode or through Bluetooth.

4.1 Application bundles and service dependencies

The prototype application we built implements some of the image composition functions of the interior design application described in the next session. This is a very interactive application exhibiting a good mixture of light and heavy processing components. Using it a user can upload an image of his/her house and a photo of a furniture item, position the furniture item on top of the house

plan, set several properties such as object focus, rotation, color, and dimension, and then invoke specialized image processing libraries for image composition.

This application was built using the OSGi module system. The entire application consists of bundles with varying requirements in terms of processing and communication resources. The service dependencies between bundles generate the graph configuration shown in Figure 3, where we can identify two flows of bundles that process the small image (the furniture item in this case) and the large image (the house image) separately, and then merge through bundle 15 and 19. The heavy computation bundles, namely 14, 15 and 19, are marked as non-movable by the developer (in dark gray in the figure).

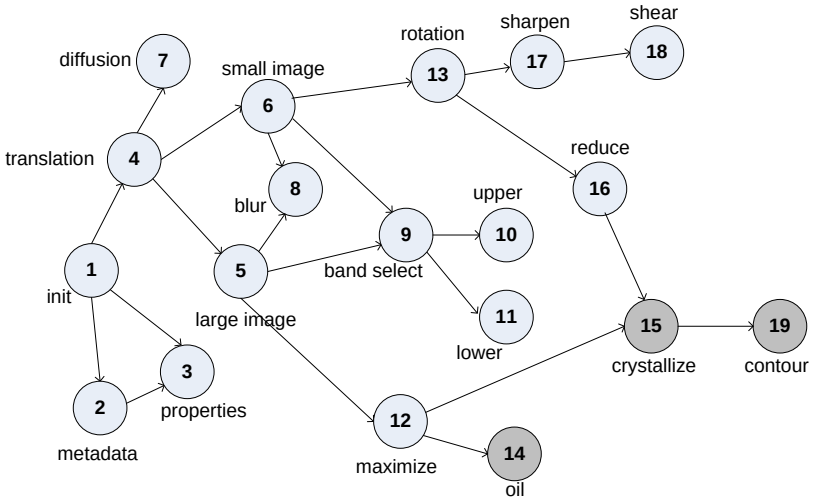


Fig. 3: Application graph.

In the following experiments, we consider the objective function described in Section 3.3. The goal is to minimize the end-to-end interaction time observed by the client, including the time necessary to acquire and install the necessary code at the beginning of the interaction.

4.2 Startup process

We start by analyzing the startup time of some selected configurations. We measure the time necessary to fetch, install, and start the necessary bundles to be run locally as well as to generate the R-OSGi proxies necessary for invoking services of remote bundles. The results are shown in Table 1.

The fetching time obviously increases with the number and size of the bundles acquired. The installation time is typically of 1–1.5 seconds per bundle. The proxy generation time depends on how many service dependencies exist

Table 1: Startup time (average and [standard deviation]) with Bluetooth.

Configuration	Fetch & Install		Proxy generation		Total time
	size (bytes)	time (ms)	num	time (ms)	(ms)
1	13940	4776 [180]	3	1044 [339]	5820
1–4	38907	9512 [100]	3	852 [66]	10364
1–5,12,14	98226	15006 [214]	5	1367 [196]	16373
1–4,6,13,16–18	82212	18650 [299]	4	1511 [176]	20161
1–6,9,12,13,16–18	109616	22666 [376]	8	2165 [221]	24831
1–6,8–13,16–18	135454	30413 [2180]	4	660 [93]	31073

with remote bundles. For example, in the first case, `init` has 3 dependencies: `translation`, `properties`, and `metadata`. Although the fetching and installation overhead can be even 30 seconds, as in the last case when 15 bundles are acquired, our algorithms opt for these kinds of configuration only when the performance gain is high enough. For smaller configurations, such as the first few cases, the overhead is comparable to the startup time of other common applications on mobile phones (e.g., text editor, web browser, etc.).

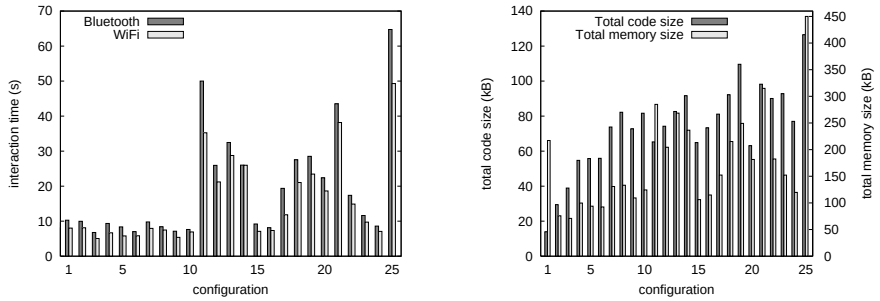
Once the interaction with an application ends, all the modules that have been fetched on the mobile device are erased such to free the phone's memory. This guarantees to consume resources only during the interaction.

4.3 Interaction time

To assess the effectiveness of our algorithms in identifying the configuration that minimizes the given objective function, we first run some grounding experiments where we quantify the cost and performance of each valid configuration of the application's bundles. The performance is in this case the overall interaction time as observed by the user. The cost is the extra price in terms of fetched bundle code and allocated memory a client has to pay to run some bundles locally.

Figure 4(a) shows the interaction time both with WiFi and Bluetooth, and Figure 4(b) the resource consumption in terms of size of shipped code and memory consumed on the tablet. The pair of images submitted to the application is $< 100kB, 30kB >$. As the number of configurations for the given application's graph is more than 100, we report results for 25 random configurations.

These experiments allow us to draw two important conclusions. First, there is no clear correlation between the interaction latency observed by the end-user and the number and size of bundles acquired by the client. There are cases in which acquiring more code does increase the interaction latency of more than 10 seconds (for example, passing from configuration 20 to 21) and others in which the opposite occurs (for example, passing from configuration 11 to 10). Second, there is a large variation in performance with even 60 seconds of difference from



(a) Interaction time (b) Resource consumption

Fig. 4: Overall interaction time (a) and resource consumption (b).

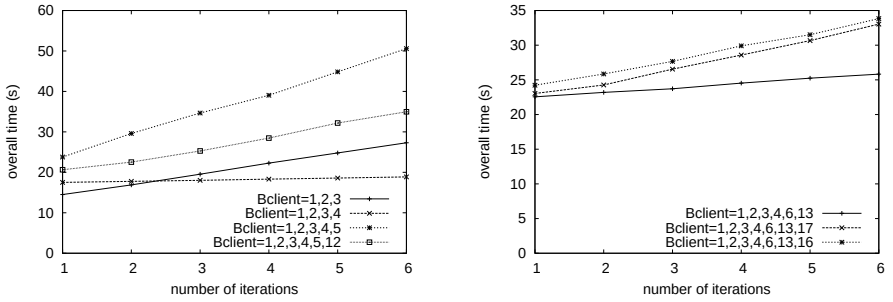
one configuration to the other. This indicates that there is potential to use the proposed algorithms to select the best configuration.

4.4 Multiple service invocations

The space of improvement is much larger than that shown by the previous results. Indeed, in running those experiments the test was configured for a "minimal number of iterations" (i.e., one invocation of every service). However, in reality this rarely occurs. For example, in setting the position, dimension, or rotation of a furniture item a user may need multiple iterations and will rarely get the properties set in a satisfying manner at the first attempt. Moreover, a user will typically place more than one furniture item in the same room thus invoking the same operations multiple items.

In these tests we investigate the impact of the number of iterations on the overall time. To this purpose we select 7 example configurations. In Figure 5 we plot the results obtained with the same images of before. The overall time includes the overhead for acquiring and installing the remote bundles and building the local proxies, and the actual interaction time measured using WiFi. The overhead installation time is 8.5 seconds for the $B_{client} = \{1, 2, 3\}$, 12 seconds for $B_{client} = \{1, 2, 3, 4\}$, and 16–18 seconds for all other configurations.

As the number of iterations increases different configurations may provide better or worse performance. In the graph in Figure 5(a), we compare the performance of the configuration $B_{client} = \{1, 2, 3\}$ with $B_{client} = \{1, 2, 3, 4\}$ when the number of invocation of bundle 4 increases, and of $B_{client} = \{1, 2, 3, 4, 5\}$ with $B_{client} = \{1, 2, 3, 4, 5, 12\}$ when the number of invocations of bundle 12 increases. The question in both comparisons is when it is convenient to acquire an additional bundle such as 4 or 12 respectively. In the first pair of configurations we see that acquiring bundle 4 becomes convenient only when the number of interactions with bundle 4 is above 2. Otherwise, the overhead of acquiring bundle 4 is higher than the benefit provided. With the second pair of configura-



(a) Increasing invocations of 4 and 12 (b) Increasing invocations of 16 and 17

Fig. 5: Overall time with multiple service invocations with WiFi.

tions, acquiring bundle 12 is always more convenient and with 6 iterations the performance gain is more than 14 seconds.

In Figure 5(b), we see the opposite behaviour. While with one iteration the performance of all configuration is similar, with an increasing number of configurations the acquisition of bundle 16 or 17 becomes less and less convenient as the number of invocations of bundle 16 and 17 respectively increases.

The number of iterations of certain operations is therefore a key factor in deciding on the best configuration. This parameter can be estimated by averaging over the behaviour of a few user interactions and it is strongly application-dependent. For example, a user interacting with a vending machine will more likely invoke the operations only once, unless in case of errors. Instead, in an application as the one considered that includes a visualization of the properties set, it is more likely to expect multiple iterations of the same function. These results clearly show that even light operations, such as bundle 4 that simply positions an image on top of another, can provide a high performance gain if performed locally (with 6 iterations, more than 12 seconds).

4.5 Algorithm performance

The last set of tests quantifies the performance of the two proposed algorithms, `ALL` and `K`-step. We consider several user scenarios, with varying mem_{MAX} and $code_{MAX}$ constraints and two different consumption graphs. Table 2 presents the results obtained.

In the first consumption graph ("one iteration app"), every bundle is invoked exactly once. Therefore, we expect few components to be acquired on the client side, since the fetching overhead is in most cases higher than the performance gain. As expected, `ALL` provides in all scenarios the optimal solution. As the optimal solution always correspond to an early cut in the graph, the `1`-step and `3`-step algorithms also find the best solution.

The second consumption graph ("multiple iteration app") models the situation in which a user invokes an application's functions (i.e., bundles) multiple

Table 2: ALL and K-step performance.

Scenario	Algorithm	One iteration app			Multiple iteration app		
		Conf	O	Error	Conf	O	Error
mem_{MAX}: 1MB	ALL	1	14.45	0.03	1,4	39.37	0.02
$code_{MAX}$: 50kB	1-STEP	1	14.45	0.03	1,4	39.37	0.02
	3-STEP	1	14.45	0.03	1,4	39.37	0.02
mem_{MAX}: 10MB	ALL	1,4	11.66	0.07	1,4,6,13	38.2	0.05
$code_{MAX}$: 50–100kB	1-STEP	1,4	11.66	0.07	1,4,6	50.53	0.32
	3-STEP	1,4	11.66	0.07	1,4,6,13	38.2	0.05
mem_{MAX}: 20–30MB	ALL	1,4	11.66	0.07	1,4,5,12	38.07	0.06
$code_{MAX}$: 50kB	1-STEP	1,4	11.66	0.07	1,4,5,6	47.72	0.32
	3-STEP	1,4	11.66	0.07	1,4,5,12	38.07	0.06
mem_{MAX}: 20–30MB	ALL	1,4	11.66	0.07	1,4–6,12,13,16	37.76	0.06
$code_{MAX}$: 100kB	1-STEP	1,4	11.66	0.07	1,4–6,13,16-18	49.78	0.24
	3-STEP	1,4	11.66	0.07	1,4–6,12,13,16–18	45.51	0.14

times. In this case, acquiring some bundles locally allows for a larger improvement of the performance. In most cases the optimal solution is to acquire 5 or more bundles, except in the first case where the phone's constraints do not allow for large acquisitions. The results of the three algorithms vary quite a lot, with 3-step outperforming 1-step in all cases.

Although the performance of ALL and K-step with an increasing K is typically the best, there exists a trade-off between processing time and accuracy of the solution. We explore this by measuring the processing time of ALL, 1-step, 3-step, and 5-step with an application graph consisting of 50 bundles and a varying number of bundle dependencies. Results are shown in Figure 6.

The processing time of all algorithms increases as the average number of edges from each node in the graph increases. This happens because a larger number of graph cuts become possible. The 5-step algorithm can be even 10 times faster than ALL and 1-step even 100 times faster.

We can conclude that while ALL suits an offline optimization, the K-step algorithm fits better dynamic scenarios where the decision has to be made on-the-fly. While 1-step can easily incur in a wrong local optimum, 3-step or 5-step provide a limited error.

5 Use Case

In addition to building the image composition application and assess the algorithms performance, we also took an existing application and applied AlfredO to it. This experience helped us to quantify the effort necessary to use AlfredO with

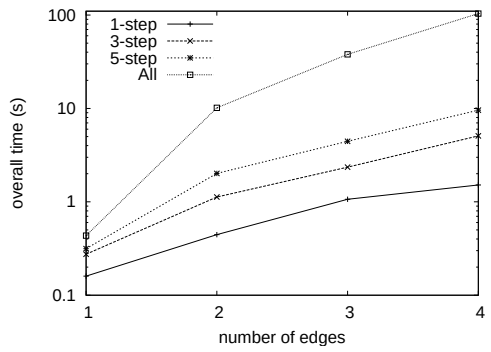


Fig. 6: Processing time for `ALL`, `1`-step, `3`-step and `5`-step.

real-world applications and to better identify the limitations of our approach. To this purpose, we used the open source Sweet Home 3D [7] application. This is a quite popular application for home interior design, which allows users to browse furniture items, place them in a 2D plan of their house, and visualize a 3D preview of it.

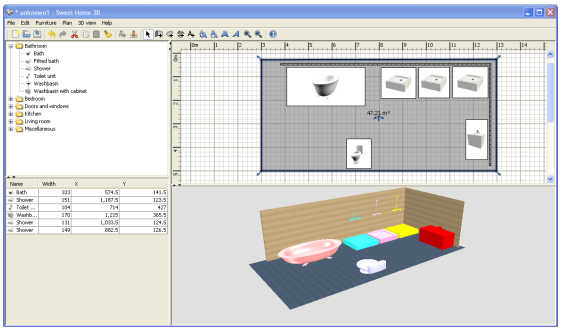
To run this application on a mobile phone several problems need to be considered. First, the application is too computational intensive to run on a phone platform. Second, its user interface uses Java Swing components, not supported by standard Java implementations available on phone platforms (i.e., Java ME CDC or CLDC). Third, considering the user interface of the application, the limited screen size of the phone would not allow for a good user experience.

To solve these problems, one possible solution would be to re-implement the entire application and customize it to the phone platform's characteristics. Instead, AlfredO solves these problems in a much faster way and provides a more extensible approach capable of integrating future extensions of the application.

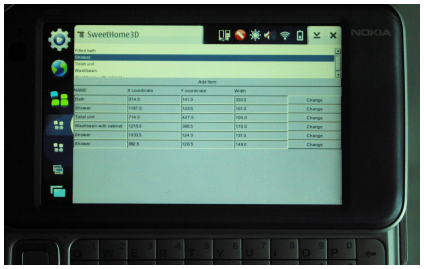
We applied AlfredO by first modularizing the application and running it on the OSGi platform. As the application was originally designed according to the Model-View-Controller design pattern, this allowed us to quickly identify its main functional components. The current modularization accounts for 13 bundles. However, we are currently working on further decomposing some of the identified bundles to provide even more flexibility. A second modification we made was providing other alternative user interfaces implemented using the Java AWT library, which is supported by existing phone platforms. We currently support three different user interfaces with an increasing level of complexity. Also, having three rather than one user interface provides more flexibility and customization.

In the application's graph this translates in having different entry points to the same application. The appropriate entry point is selected by taking into account additional properties of the phone client such as screen size, color resolu-

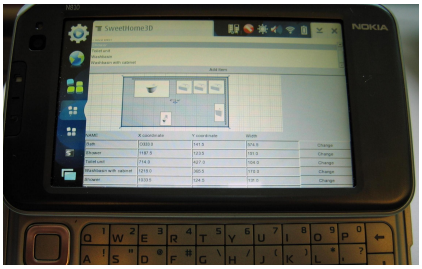
tion, etc. Once the entry point is selected our algorithms are applied to determine the best graph cut and the corresponding configuration.



(a) SweetHome 3D server



(b) Catalog list and item placement on a N810



(c) Catalog list, item placement and map preview on N810

Fig. 7: Sweet Home 3D application running on a server (a) and on Nokia N810 handhelds (b and c).

At a high-level, three classes of configurations can be supported. The simplest case is when the phone client acts only as a mouse controller of the remote application. The output returned to the mobile phone consists of a screenshot of the application display. This kind of interaction reuses the MouseController concepts we presented in [2].

A second possibility shown in Figure 7(b) is when the phone acquires the user interface necessary to select furniture items and specify their width and position (as x,y coordinates) locally. On the server side, items are placed accordingly in the 2D plan and the 3D preview is generated. A final option shown in Figure 7(c) is when the phone supports item selection and also placement in the 2D plan.

Within each of these three types of configuration, different distributions of the application components are possible depending on the algorithm's decision. The overhead introduced by our modularization approach was found to be negligible compared to the original application.

6 Limitations and open problems

The experiment of using AlfredO with an existing application helped us to define better its scope of applicability and limitations.

In many applications, the user interface and the service logic are tightly coupled in complex relationships. This means that a modularization at the level of service logic requires changes at the user interface too. As we saw with Sweet Home 3D, we modularized the application into several bundles and identified three high-level functionalities, such as catalog selection and item placement, 2D plan operations, and 3D rendering and visualization. In order to support these functions alone or all together, we needed to provide different user interfaces: one that displays a list of furniture items and a table with the item properties, one that adds to the first interface a 2D plan, and one that adds a further image panel for 3D functions. On the other hand, we saw how with a much simpler application, such as the one for image composition that we built, one user interface was enough to allow high flexibility at the service level.

Our experience has shown that AlfredO can work well with both types of application. Obviously, more complex applications require to be modularized, but the effort has proven to be reasonable. In the case of Sweet Home 3D, a Master student, with no knowledge of the application and OSGi, took less than two months to modularize it and build the three user interfaces described in Section 5. Thereby, we expect that for simpler applications, less than a month would be enough to make them run on AfredO. Furthermore, the advantage of AlfredO is that it builds on existing technology for distributed module management, based on the OSGi standard. OSGi is maintained by the OSGi Alliance with many major players of the software industry, such as IBM, Oracle, and SAP, and also device vendors, such as Nokia, Ericsson, Motorola. Moreover, OSGi has been used in several applications including Eclipse IDE [8] and we expect that in the future more and more developers will be acquainted with it.

Finally, our algorithms require profiling of the resource consumption of an application's bundles and their inter-communication. We instrumented our applications manually, however there are tools available or under study for automatic profiling of applications. Some are discussed in the related work section.

7 Related Work

There is a considerable amount of reseach on how to automatically partition and distribute applications based on resource profiling. One of the very early work in this context was the Interconnected Processor System (ICOPS) [9]. ICOPS used scenario-based profiling to collect statistics about resource requirements. Static data such as procedure inter-connections and dynamic statistics about resource usage were then combined to find the best assignment of procedures to processors. ICOPS was the first system using a minimum cut algorithm to select the best distribution. On the other hand, ICOPS considered very small programs of seven modules and only three of these could be moved between the

client and server. A more recent work in this context is Coign [10]. Coign assumes applications to be built using components conforming Microsoft's COM. It builds a graph model of an application's inter-component communication by scenario-based profiling. The application is then partitioned to minimize execution delay due to network communication. Several other works exist such as [11].

Our techniques share with these systems the idea of building a graph model of the application and applying a graph-cutting algorithm to partition it. However, we differ from them in several aspects. We do not focus on building a tool for application resource profiling, but rather on dynamically optimizing the interaction with an application given the constraints of the current execution environment. On the other hand, these or similar tools could be integrated in AlfredO to automatically characterize the resource requirements of an application's modules or even partition a non-modularized application. This would allow to extend AlfredO to non-OSGi applications.

A second important difference is the concept of "distribution" itself. The algorithms we propose do not target distributing an application on a cluster of machines or a cloud infrastructure, but rather installing all or parts of it in order to use on a mobile phone. The decision is intrinsically client-driven. In this sense, AlfredO is closer to a web browser that provides access to Internet services and requires the user to install plugins.

Finally, AlfredO is designed to work in heterogeneous and dynamic contexts. Clients exhibit large variability in terms of device platforms, local resources, type of network communication, etc. This heterogeneity needs to be captured by the optimization problem and hidden to the end user.

Related work is also in the context of other non-phone specific distributed systems. For example, in the context of sensor networks systems like Wishbone [12], Tenet [13], VanGo [14], in the context of mobile ad hoc networks with SpatialViews [15], and in the context of cluster computing with Abacus [16]. All these systems are not applicable to our problem space for different reasons. For instance, Wishbone partitions programs to run on multiple and heterogeneous devices in a sensor network. Wishbone is primarily concerned with high-rate data processing applications, aims at statically minimizing a combination of network bandwidth and CPU load, and is used at compile time. Abacus dynamically partitions applications and filesystem functionality over a cluster of resources. It primarily targets data-intensive applications and attempts to optimize the placement of mobile objects, by using a fixed objective function that combines variations in network topology, application cache access pattern, application data reduction, contention over shared data and dynamic competition for resources by concurrent applications. Our techniques target less computational- and data-intensive applications and provide support for multiple objective functions.

The vision of pervasive computing [17], as the creation of physical environments saturated with a variety of computing and communication capabilities, is also relevant to our work. The solutions proposed in that context allow devices to interact with the surrounding environment by either statically preconfiguring the devices and the environment with the necessary software [18], or by moving

around the necessary software through techniques such as mobile agents [19, 20]. The first approach can work only in static environments or to support applications that are accessed on a very frequent basis. The second approach is more flexible, but it is not used in practice due to security issues.

We address the problem in a different manner. To use an application on a mobile phone, today a user has two options: *1)* install the application locally or *2)* if this is available in the Internet, access it through a web browser. We propose a new model where the phone is seen as an application controller. The minimal configuration sees a phone that acquires only a user interface, thus achieving high security. For more advanced and optimized interactions, some parts of the application can be installed. The acquisition of an application occurs in a more controlled manner and with clearly identified boundaries dependent on the resource constraints of the mobile device and the type of network communication.

8 Conclusions

We have presented our approach to automatically and dynamically distributing several components of an application between a mobile and a serve in order to optimize different objective functions such as interaction time, communication cost, memory consumption, etc. Compared to the current state-of-the-art in building applications on mobile phones, our approach enables an efficient deployment of several types of applications on mobile phones thus allowing these resource-constrained platforms to achieve better performance with a controlled overhead. Our optimization has focused so far only on the client side and has assumed the server's resources to be infinite. As future work, we are investigating how to extend our application's model to include also CPU consumption and include into the optimization problem also how the server side can be distributed over a cloud infrastructure with heterogeneous resources.

Acknowledgments

The work presented in this paper was supported by the Microsoft Innovation Cluster for Embedded Software (ICES) and the ETH Fellowship Program. We thank Jan Rellermeyer for his advise and help during the development of AlfredO on top of R-OSGi.

References

1. Rellermeyer, J.S., Alonso, G., Roscoe, T.: R-OSGi: Distributed applications through software modularization. In: Proceedings of the ACM/IFIP/USENIX 8th International Conference on Middleware (Middleware'07). Volume 4834 of LNCS., Springer (2007) 1–20
2. Rellermeyer, J.S., Riva, O., Alonso, G.: AlfredO: An Architecture for Flexible Interaction with Electronic Devices. In: Proceedings of the 9th International Middleware Conference (Middleware'08). Volume 5346 of LNCS., Springer (2008) 22–41

3. OSGi Alliance: OSGi Service Platform, Core Specification Release 4, Version 4.1, Draft. (2007)
4. Guttman, E., Perkins, C., Veizades, J.: Service Location Protocol, Version 2. RFC 2608, Internet Engineering Task Force (IETF) (1999) Available at http://www.ietf.org/rfc/rfc2608.txt.
5. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. **20**(1) (1998) 359–392
6. Boman, E., et al.: Zoltan: Parallel partitioning, load balancing and data-management services user's guide. Sandia National Laboratories (2007)
7. Sweet Home 3D: (2009) http://www.sweethome3d.eu/.
8. Eclipse Foundation: Eclipse. http://www.eclipse.org (2001)
9. Stabler, G.: A system for interconnected processing. PhD thesis, Providence, RI, USA (1975)
10. Hunt, G., Scott, M.: The coign automatic distributed partitioning system. In: Proceedings of the 3rd symposium on Operating systems design and implementation (OSDI'99), USENIX Association (1999) 187–200
11. Hamlin, J., Foley, J.: Configurable applications for graphics employing satellites (cages). In: Proceedings of the 2nd annual conference on Computer graphics and interactive techniques (SIGGRAPH '75), ACM (1975) 9–19
12. Newton, R., Toledo, S., Girod, L., Balakrishnan, H., Madden, S.: Wishbone: Prole-based Partitioning for Sensornet Applications. In: Proceedings of the 5th Symposium on Networked Systems Design and Implementation (NSDI'09). (2009) 395–408
13. Gnawali, O., Jang, K.Y., Paek, J., Vieira, M., Govindan, R., Greenstein, B., Joki, A., Estrin, D., Kohler, E.: The Tenet architecture for tiered sensor networks. In: Proceedings of the 4th international conference on Embedded networked sensor systems (SenSys'06), ACM (2006) 153–166
14. Greenstein, B., Mar, C., Pesterev, A., Farshchi, S., Kohler, E., Judy, J., Estrin, D.: Capturing high-frequency phenomena using a bandwidth-limited sensor network. In: Proceedings of the 4th international conference on Embedded networked sensor systems (SenSys'06), ACM (2006) 279–292
15. Ni, Y., Kremer, U., Stere, A., Iftode, L.: Programming ad-hoc networks of mobile and resource-constrained devices. In: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI'05), ACM (2005) 249–260
16. Amiri, K., Petrou, D., Ganger, G., Gibson, G.: Dynamic Function Placement for Data-intensive Cluster Computing. In: Proceedings of the 18th USENIX annual technical conference (USENIX'00). (2000) 307–322
17. Weiser, M.: The Computer for the Twenty-First Century. Scientific American **265**(3) (1991) 94–104
18. Want, R., Pering, T., Danneels, G., Kumar, M., Sundar, M., Light, J.: The personal server: Changing the way we think about ubiquitous computing. In: Proceedings of the 4th international conference on Ubiquitous Computing (UbiComp'02), Springer-Verlag (2002) 194–209
19. Fuggetta, A., Picco, G.P., Vigna, G.: Understanding code mobility. IEEE Transactions on Software Engineering **24**(5) (1998) 342–361
20. Kindberg, T., Barton, J., Morgan, J., Becker, G., Caswell, D., Debaty, P., Gopal, G., Frid, M., Krishnan, V., Morris, H., Schettino, J., Serra, B., Spasojevic, M.: People, places, things: Web presence for the real world. In: Proceedings of the 3rd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'00). (2000) 19